CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

VHDL “Digital Lock” design implemtntation on Altera DE2 board

AND

Area and Time optimization of ASIC “FIFO” design using Synopsys design compiler

A graduate project submitted in partial fulfillment of

The requirements for the degree of Masters of Science

In Electrical Engineering

By

Pranav Suryakant Biscuitwala

May 2012

The graduate project of Pranav Biscuitwala is approved:

_____	_____
Dr. Ali Amini	Date
_____	_____
Dr. Somnath Chattopadhyay	Date
_____	_____
Dr. Ramin Roosta, Chair	Date

California State University, Northridge

ACKNOWLEDGEMENT

I would like to express my gratitude to the Electrical and Computer Engineering Department Faculty at the California State University, Northridge for their time, guidance and patience. I hope this project will be a good starting point from where students can learn about Synopsys Design Compiler, VHDL code implementation on DE2 and they will be able to move ahead of me.

I would like to thank Dr. Ramin Roosta for introducing me to the field of ASIC Design and to learn the basics of Front end ASIC design flow, Types of ASIC Architecture and allowing me to use Synopsys compiler for verification of my design.

I also want to thank Dr. Ali Amini and Dr. Somnath Chattopadhyay for their valued attention and encouragement.

Most importantly, I would like to thank my parents, Mr.Suryakant G Biscuitwala and Mrs. Ramilaben S Biscuitwala for giving me an opportunity to come to USA to pursue a highly qualified technical education of Master's Degree that has not only improved my career but has also helped me in defining my long term career goals.

Special thanks to Shashank, Keyur, Mayur, Tejas, Naveen for their guidance and motivation. I also want to thank all my friends and my brother back in my home country and CSUN for supporting and believing in me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

VHDL "DIGITAL LOCK" DESIGN IMPLEMENTATION ON ALTERA DE2 BOARD
AND
AREA AND TIME OPTIMIZATION OF ASIC "FIFO" DESIGN USING SYNOPSYS DESIGN COMPILER

By

Pranav Suryakant Biscuitwala

Master of Science in Electrical Engineering

Part 1: In this the main objective of the project was to produce a VHDL design which can be programmed on Altera DE2 board. This project gives an overview of how different components of DE2 board can be interfaced. This project uses the various part of DE2 board like LCD screen, LEDs , Push Buttons, 7-segment display etc.

Part 2: In this the main objective of the project was to produce ASIC design and optimize the design using synopsys tool. This optimization is done using 90nm technology. The target was to create design of asynchronous FIFO for n-bit wide (using parameter) data. Whose default width is 16 bits. Since it is asynchronous it's input clock chosen is 200 MHz and output clock is 40 MHz. In this data arrive in 32 word (n-bit wide) bursts. This interface signals to the transmitting device when it is clear to send a burst.

This design was synthesized and simulated at the gate level. This design was optimized for time and area.

Chapter 1: Layout and Components of DE2 board

VHDL "DIGITAL LOCK" DESIGN IMPLEMENTATION ON ALTERA DE2 BOARD

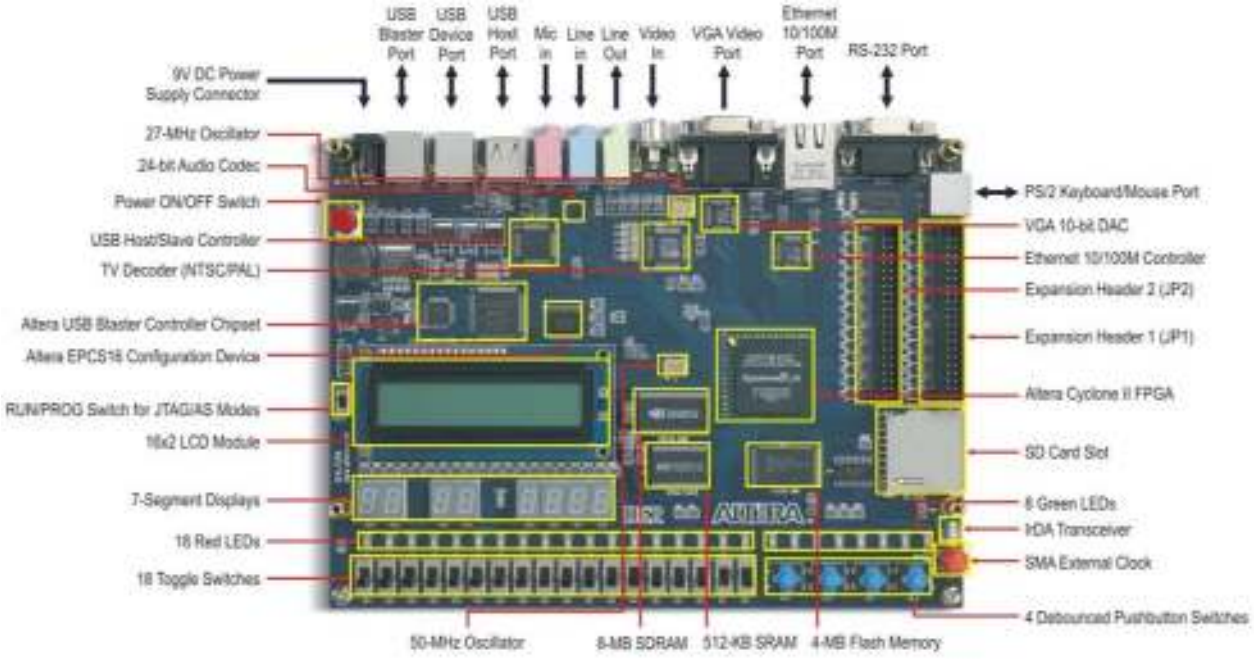


Figure 1.1 The DE2 board

The following hardware is provided on the DE2 board:Altera Cyclone® II 2C35 FPGA device

- Altera Serial Configuration device - EPCS16
- USB Blaster (on board) for programming and user API control; both JTAG and Active Serial (AS) programming modes are supported
- 512-Kbyte SRAM
- 8-Mbyte SDRAM4-Mbyte Flash memory (1 Mbyte on some boards)
- SD Card socket
- 4 pushbutton switches
- 18 toggle switches
- 18 red user LEDs
- 9 green user LEDs
- 50-MHz oscillator and 27-MHz oscillator for clock sources
- 24-bit CD-quality audio CODEC with line-in, line-out, and microphone-in jacks
- VGA DAC (10-bit high-speed triple DACs) with VGA-out connector
- TV Decoder (NTSC/PAL) and TV-in connector
- 10/100 Ethernet Controller with a connector
- USB Host/Slave Controller with USB type A and type B connectors

- RS-232 transceiver and 9-pin connector
- PS/2 mouse/keyboard connector
- IrDA transceiver
- Two 40-pin Expansion Headers with diode protection

1.2 Block Diagram of the DE2 Board

Figure 1.2 gives the block diagram of the DE2 board. To provide maximum flexibility for the user, all connections are made through the Cyclone II FPGA device. Thus, the user can configure the FPGA to implement any system design.

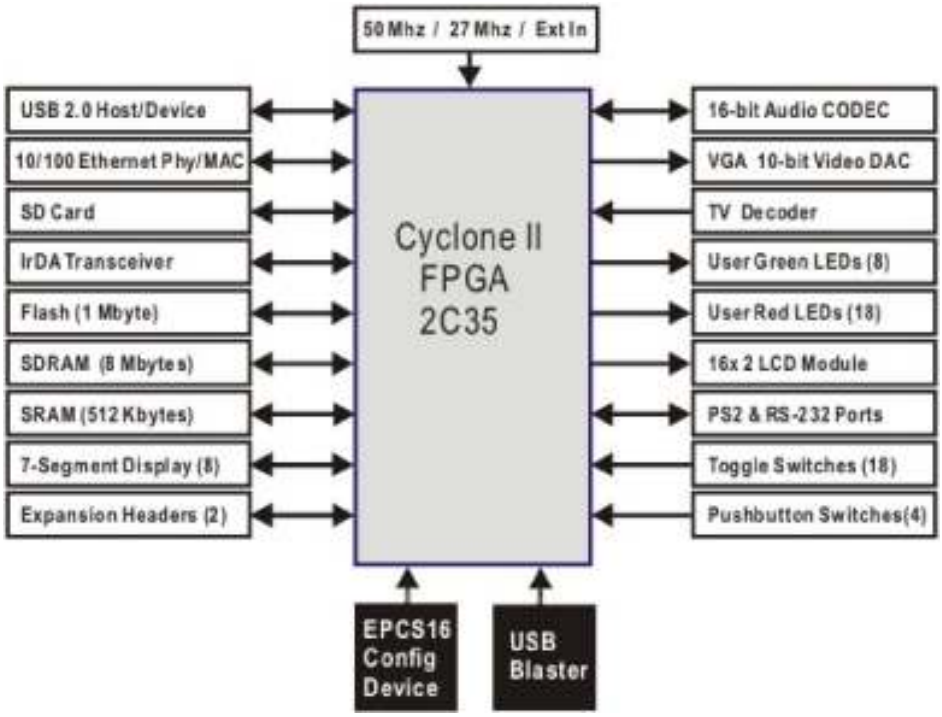


Figure 1.2. Block diagram of the DE2 board.

Following is more detailed information about the blocks in Figure 1.2:

Cyclone II 2C35 FPGA

- 33,216 LEs
- 105 M4K RAM blocks
- 483,840 total RAM bits
- 35 embedded multipliers
- 4 PLLs
- 475 user I/O pins
- FineLine BGA 672-pin package

Serial Configuration device and USB Blaster circuit

- Altera's EPCS16 Serial Configuration device
- On-board USB Blaster for programming and user API control
- JTAG and AS programming modes are supported

SRAM

- 51 2-Kbyte Static RAM memory chip
- Organized as 256K x 16 bits
- Accessible as memory for the Nios II processor and by the DE2 Control Panel

SDRAM

- 8-Mbyte Single Data Rate Synchronous Dynamic RAM memory chip
- Organized as 1M x 16 bits x 4 banks
- Accessible as memory for the Nios II processor and by the DE2 Control Panel

Flash memory

- 4-Mbyte NOR Flash memory (1 Mbyte on some boards)
- 8-bit data bus
- Accessible as memory for the Nios II processor and by the DE2 Control Panel

SD card socket

- Provides SPI mode for SD Card access
- Accessible as memory for the Nios II processor with the DE2 SD Card Driver

Pushbutton switches

- 4 pushbutton switches
- Debounced by a Schmitt trigger circuit
- Normally high; generates one active-low pulse when the switch is pressed

Toggle switches

- 18 toggle switches for user inputs
- A switch causes logic 0 when in the DOWN (closest to the edge of the DE2 board) position and logic 1 when in the UP position

Clock inputs

- 50-MHz oscillator
- 27-MHz oscillator
- SMA external clock input

Audio CODEC

- Wolfson WM873 1 24-bit sigma-delta audio CODEC
- Line-level input, line-level output, and microphone input jacks
- Sampling frequency: 8 to 96 KHz
- Applications for MP3 players and recorders, PDAs, smart phones, voice recorders, etc.

VGA output

- Uses the ADV7 123 240-MHz triple 10-bit high-speed video DAC
- With 15-pin high-density D-sub connector
- Supports up to 1600 x 1200 at 100-Hz refresh rate
- Can be used with the Cyclone II FPGA to implement a high-performance TV Encoder

NTSC/PAL TV decoder circuit

- Uses ADV7 181 B Multi-format SDTV Video Decoder
- Supports NTSC-(M,J,4.43), PAL-(B/D/G/H/I/M/N), SECAM
- Integrates three 54-MHz 9-bit ADCs
- Clocked from a single 27-MHz oscillator input
- Supports Composite Video (CVBS) RCA jack input.
- Supports digital output formats (8-bit/16-bit): ITU-R BT.656 YCrCb 4:2:2 output + HS, VS, and FIELD

- Applications: DVD recorders, LCD TV, Set-top boxes, Digital TV, Portable video devices

10/100 Ethernet controller

- Integrated MAC and PHY with a general processor interface
- Supports 100Base-T and 10Base-T applications
- Supports full-duplex operation at 10 Mb/s and 100 Mb/s, with auto-MDIX
- Fully compliant with the IEEE 802.3u Specification
- Supports IP/TCP/UDP checksum generation and checking
- Supports back-pressure mode for half-duplex mode flow control

USB Host/Slave controller

- Complies fully with Universal Serial Bus Specification Rev. 2.0
- Supports data transfer at full-speed and low-speed
- Supports both USB host and device
- Two USB ports (one type A for a host and one type B for a device)
- Provides a high-speed parallel interface to most available processors; supports Nios II with a Terasic driver
- Supports Programmed I/O (PIO) and Direct Memory Access (DMA)

Serial ports

- One RS-232 port
- One PS/2 port
- DB-9 serial connector for the RS-232 port
- PS/2 connector for connecting a PS2 mouse or keyboard to the DE2 board

IrDA transceiver

- Contains a 115.2-kb/s infrared transceiver
- 32 mA LED drive current
- Integrated EMI shield
- IEC825-1 Class 1 eye safe
- Edge detection input

Two 40-pin expansion headers

- 72 Cyclone II I/O pins, as well as 8 power and ground lines, are brought out to two 40-pin expansion connectors
- 40-pin header is designed to accept a standard 40-pin ribbon cable used for IDE hard drives
- Diode and resistor protection is provided

1.3 Power-up the DE2 Board

The DE2 board comes with a preloaded configuration bit stream to demonstrate some features of the board. This bit stream also allows users to see quickly if the board is working properly. To power-up the board perform the following steps:

1. Connect the provided USB cable from the host computer to the USB Blaster connector on the DE2 board. For communication between the host and the DE2 board, it is necessary to install the Altera USB Blaster driver software. If this driver is not already installed on the host computer, it can be installed as explained in the tutorial *Getting Started with Altera's DE2 Board.* This tutorial is available on the **DE2 System CD-ROM** and from the Altera DE2 web pages.

2. Connect the 9V adapter to the DE2 board
3. Connect a VGA monitor to the VGA port on the DE2 board
4. Connect your headset to the Line-out audio port on the DE2 board
5. Turn the RUN/PROG switch on the left edge of the DE2 board to RUN position; the PROG position is used only for the AS Mode programming
6. Turn the power on by pressing the ON/OFF switch on the DE2 board

At this point you should observe the following:

- All user LEDs are flashing
- All 7-segment displays are cycling through the numbers 0 to F
- The LCD display shows **Welcome to the Altera DE2 Board**
- The VGA monitor displays the image shown in Figure 1.3.
- Set the toggle switch SW17 to the DOWN position; you should hear a 1-kHz sound
- Set the toggle switch SW17 to the UP position and connect the output of an audio player to the Line-in connector on the DE2 board; on your headset you should hear the music played from the audio player (MP3, PC, iPod, or the like)
- You can also connect a microphone to the Microphone-in connector on the DE2 board; your voice will be mixed with the music played from the audio player

Figure 1.3. The default VGA output pattern.

Chapter:2 DE2 Control Panel

The DE2 board comes with a Control Panel facility that allows a user to access various components on the board through a USB connection from a host computer. This chapter first presents some basic functions of the Control Panel, then describes its structure in block diagram form, and finally describes its capabilities.

2.1 Control Panel Setup

To run the Control Panel application, it is first necessary to configure a corresponding circuit in the Cyclone II FPGA. This is done by downloading the configuration file *DE2_USB_API.sof* into the FPGA. The downloading procedure is described in Section 4.1.

In addition to the *DE2_USB_API.sof* file, it is necessary to execute on the host computer the program *DE2_control_panel.exe.* Both of these files are available on the **DE2 System CD-ROM** that accompanies the DE2 board, in the directory *DE2_control_panel.* Of course, these files may already have been installed to some other location on your computer system.

To activate the Control Panel, perform the following steps:

1. Connect the supplied USB cable to the USB Blaster port, connect the 9V power supply, and turn the power switch ON
2. Set the RUN/PROG switch to the RUN position
3. Start the Quartus II software
4. Select **Tools > Programmer** to reach the window in Figure 2.1. Click on **Add File** and in the pop-up window that appears select the *DE2_USB_API.sof* file. Next, click on the **Program/Configure** box which results in the image displayed in the figure. Now, click **Start** to download the configuration file into the FPGA.
5. Start the executable *DE2_control_panel.exe* on the host computer. The Control Panel user interface shown in Figure 2.2 will appear.
6. Open the USB port by clicking **Open > Open USB Port 0.** The DE2 Control Panel application will list all the USB ports that connect to DE2 boards. The DE2 Control Panel can control up to 4 DE2 boards using the USB links. The Control Panel will occupy the USB port until you close that port; you cannot use Quartus II to download a configuration file into the FPGA until you close the USB port.
7. The Control Panel is now ready for use; experiment by setting the value of some 7-segment display and observing the result on the DE2 board.

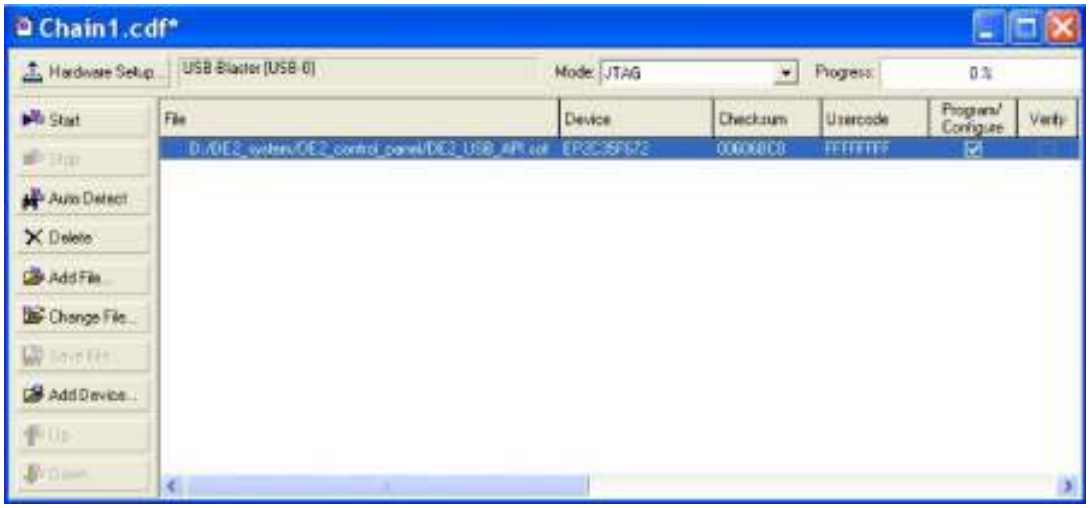


Figure 2.1. Quartus II Programmer window.

Figure 2.2. The DE2 Control Panel.

The concept of the DE2 Control Panel is illustrated in manual. The IP that performs the control functions is implemented in the FPGA device. It communicates with the Control Panel window, which is active on the host computer, via the USB Blaster link. The graphical interface is used to issue commands to the control circuitry. The provided IP handles all requests and performs data transfers between the computer and the DE2 board.

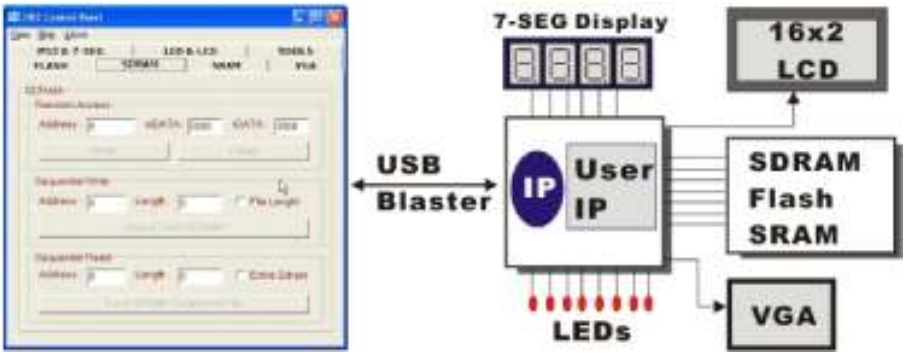


Figure 2.3. The DE2 Control Panel concept.

The DE2 Control Panel can be used to change the values displayed on 7-segment displays, light up LEDs, talk to the PS/2 keyboard, read/write the SRAM, Flash Memory and SDRAM, load an image pattern to display as VGA output, load music to the memory and play music via the audio DAC. The feature of reading/writing a byte or an entire file from/to the Flash Memory allows the user to develop multimedia applications (Flash Audio Player, Flash Picture Viewer) without worrying about how to build a Flash Memory Programmer.

2.2 Controlling the LEDs, 7-Segment Displays and LCD Display

A simple function of the Control Panel is to allow setting the values displayed on LEDs, 7-segment displays, and the LCD character display.

In the window shown in Figure 2.4, the values to be displayed by the 7-segment displays (which are named *HEX*7-0) can be entered into the corresponding boxes and displayed by pressing the **Set** button. A keyboard connected to the PS/2 port can be used to type text that will be displayed on the LCD display.

Choosing the **LED & LCD** tab leads to the window in Figure 2.4. Here, you can turn the individual LEDs on by selecting them and pressing the **Set** button. Text can be written to the LCD display by typing it in the LCD box and pressing the corresponding **Set** button.

The ability to set arbitrary values into simple display devices is not needed in typical design activities. However, it gives the user a simple mechanism for verifying that these devices are functioning correctly in case a malfunction is suspected. Thus, it can be used for troubleshooting purposes.

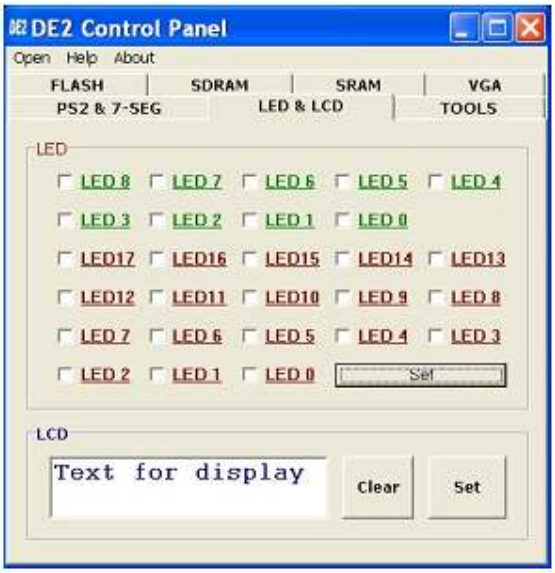


Figure 2.4. Controlling LEDs and the LCD display.

2.3 SDRAM/SRAM Controller and Programmer

The Control Panel can be used to write/read data to/from the SDRAM and SRAM chips on the DE2 board. We will describe how the SDRAM may be accessed; the same approach is used to access the SRAM. Click on the SDRAM tab to reach the window in Figure 2.5.

Figure 2.5. Accessing the SDRAM.

A 16-bit word can be written into the SDRAM by entering the address of the desired location, specifying the data to be written, and pressing the **Write** button. Contents of the location can be read by pressing the

Read button. Figure 2.5 depicts the result of writing the hexadecimal value 6CA into location 200, followed by reading the same location.

The Sequential Write function of the Control Panel is used to write the contents of a file into the SDRAM as follows:

1. Specify the starting address in the **Address** box.
2. Specify the number of bytes to be written in the **Length** box. If the entire file is to be loaded, then a checkmark may be placed in the **File Length** box instead of giving the number of bytes.
3. To initiate the writing of data, click on the **Write a File to SDRAM** button.
4. When the Control Panel responds with the standard Windows dialog box asking for the source file, specify the desired file in the usual manner.

The Control Panel also supports loading files with a *.hex* extension. Files with a *.hex* extension are ASCII text files that specify memory values using ASCII characters to represent hexadecimal values. For example, a file containing the line 0123456789ABCDEF defines four 16-bit values: 0123, 4567, 89AB, CDEF. These values will be loaded consecutively into the memory.

The Sequential Read function is used to read the contents of the SDRAM and place them into a file as follows:

1. Specify the starting address in the **Address** box.
2. Specify the number of bytes to be copied into the file in the **Length** box. If the entire contents of the SDRAM are to be copied (which involves all 8 Mbytes), then place a checkmark in the **Entire SDRAM** box.
3. Press **Load SDRAM Content to a File** button.
4. When the Control Panel responds with the standard Windows dialog box asking for the destination file, specify the desired file in the usual manner.

2.4 Flash Memory Programmer

The Control Panel can be used to write/read data to/from the Flash memory chip on the DE2 board. It can be used to:

- Erase the entire Flash memory
- Write one byte to the memory
- Read one byte from the memory
- Write a binary file to the memory
- Load the contents of the Flash memory into a file

Note the following characteristics of the Flash memory:

- The Flash memory chip is organized as 4 M (or 1 M on some boards) x 8 bits.
- You must erase the entire Flash memory before you can write into it. (Be aware that the number of times a Flash memory can be erased is limited.)
- The time required to erase the entire Flash memory is about 20 seconds. Do not close the DE2 Control Panel in the middle of the operation.

To open the Flash memory control window, shown in Figure 2.6, select the FLASH tab in the Control Panel.

Figure 2.6. Flash memory control window.

A byte of data can be written into a random location on the Flash chip as follows:

1. Click on the **Chip Erase** button. The button and the window frame title will prompt you to wait until the operation is finished, which takes about 20 seconds.

2. Enter the desired address into the **Address** box and the data byte into the **wDATA** box. Then, click on the **Write** button.

To read a byte of data from a random location, enter the address of the location and click on the **Read** button. The **rDATA** box will display the data read back from the address specified.

The Sequential Write function is used to load a file into the Flash chip as follows:

1. Specify the starting address and the length of data (in bytes) to be written into the Flash memory. You can click on the **File Length** checkbox to indicate that you want to load the entire file.
2. Click on the **Write a File to Flash** button to activate the writing process.
3. When the Control Panel responds with the standard Windows dialog box asking for the source file, specify the desired file in the usual manner.

The Sequential Read function is used to read the data stored in the Flash memory and write this data into a file as follows:

1. Specify the starting address and the length of data (in bytes) to be read from the Flash memory. You can click on the **Entire Flash** checkbox to indicate that you want to copy the entire contents of the Flash memory into a specified file.
2. Click on the **Load Flash Content to a File** button to activate the reading process.
3. When the Control Panel responds with the standard Windows dialog box asking for the destination file, specify the desired file in the usual manner.

2.5 Overall Structure of the DE2 Control Panel

The DE2 Control Panel facility communicates with a circuit that is instantiated in the Cyclone II FPGA. This circuit is specified in Verilog code, which makes it possible for a knowledgeable user to change the functionality of the Control Panel. The code is located inside the *DE2_demonstrations* directory on the **DE2 System CD-ROM.**

To run the Control Panel, the user must first set it up as explained before depicts the structure of the Control Panel. Each input/output device is controlled by a controller instantiated in the FPGA chip. The communication with the PC is done via the USB Blaster link. A Command Controller circuit interprets the commands received from the PC and performs the appropriate actions. The SDRAM, SRAM, and Flash Memory controllers have three user-selectable asynchronous ports in addition to the Host port that provides a link with the Command Controller. The connection between the VGA DAC Controller and the FPGA memory allows displaying of the default image shown on the left side of the figure, which is stored in an M4K block in the CycloneII chip.

The connection between the Audio DAC Controller and a lookup table in the FPGA is used to produce a test audio signal of 1 kHz.

To let users implement and test their IP cores (written in Verilog) without requiring them to implement complex API/Host control software and memory (SRAM/SDRAM/Flash) controllers, we provide an integrated control environment consisting of a software controller in C++, a USB command controller, and a multi-port SRAM/SDRAM/Flash controller.

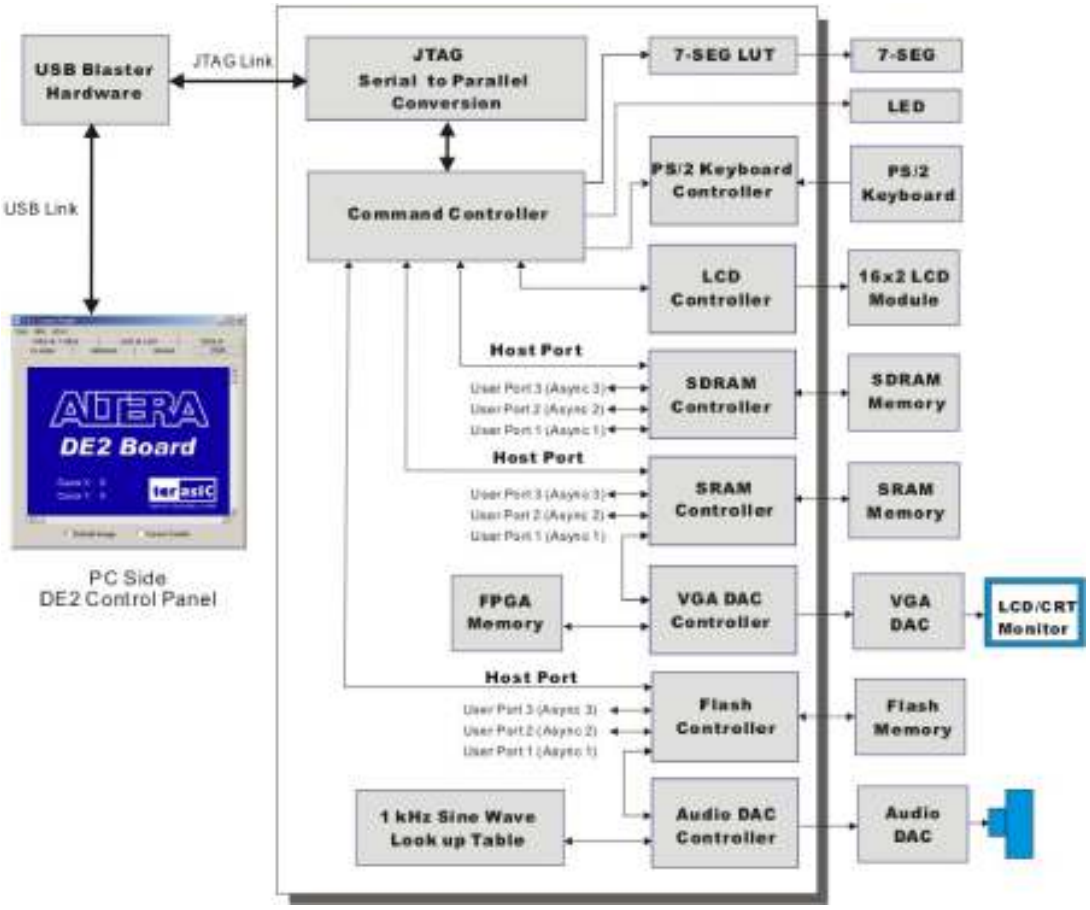


Figure 2.7. The DE2 Control Panel block diagram.

Users can connect circuits of their own design to one of the User Ports of the SRAM/SDRAM/Flash controller. Then, they can download binary data into the SRAM/SDRAM/Flash. Once the data is downloaded to the SDRAM/Flash, users can configure the memory controllers so that their circuits can read/write the SDRAM/Flash via the User Ports connected.

Chapter 3: Using the DE2 Board

This chapter gives instructions for using the DE2 board and describes each of its I/O devices.

3.1 Configuring the Cyclone II FPGA

The procedure for downloading a circuit from a host computer to the DE2 board is described in the tutorial *Quartus I Introduction.* This tutorial is found in the *DE2_tutorials* folder on the **DE2 System CD-ROM,** and it is also available on the Altera DE2 web pages. The user is encouraged to read the tutorial first, and to treat the information below as a short reference.

The DE2 board contains a serial EEPROM chip that stores configuration data for the Cyclone II FPGA. This configuration data is automatically loaded from the EEPROM chip into the FPGA each time power is applied to the board. Using the Quartus II software, it is possible to reprogram the FPGA at any time, and it is also possible to change the non-volatile data that is stored in the serial EEPROM chip. Both types of programming methods are described below.

1. *JTA G* programming: In this method of programming, named after the IEEE standards *Joint Test Action Group,* the configuration bit stream is downloaded directly into the Cyclone II FPGA. The FPGA will retain this configuration as long as power is applied to the board; the configuration is lost when the power is turned off.
2. *AS* programming: In this method, called *Active Serial* programming, the configuration bit stream is downloaded into the Altera EPCS16 serial EEPROM chip. It provides non-volatile storage of the bit stream, so that the information is retained even when the power supply to the DE2 board is turned off. When the board's power is turned on, the configuration data in the EPCS16 device is automatically loaded into the Cyclone II FPGA.

The sections below describe the steps used to perform both JTAG and AS programming. For both methods the DE2 board is connected to a host computer via a USB cable. Using this connection, the board will be identified by the host computer as an Altera *USB Blaster* device. The process for installing on the host computer the necessary software device driver that communicates with the USB Blaster is described in the tutorial *Getting Started with Altera's DE2 Board.* This tutorial is available on the **DE2 System CD-ROM** and from the Altera DE2 web pages.

Configuring the FPGA in JTAG Mode

Figure 3.1 illustrates the JTAG configuration setup. To download a configuration bit stream into the Cyclone II FPGA, perform the following steps:

- Ensure that power is applied to the DE2 board
- Connect the supplied USB cable to the USB Blaster port on the DE2 board
- Configure the JTAG programming circuit by setting the RUN/PROG switch (on the left side of the board) to the RUN position.
- The FPGA can now be programmed by using the Quartus II Programmer module to select a configuration bit stream file with the *.sof* filename extension

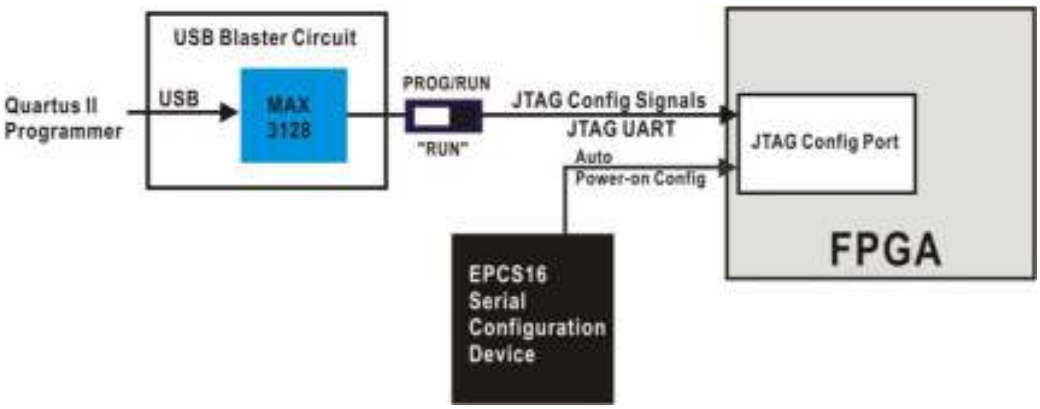


Figure 3.1. The JTAG configuration scheme.

Configuring the EPCS16 in AS Mode

Figure 3.2 illustrates the AS configuration set up. To download a configuration bit stream into the EPCS16 serial EEPROM device, perform the following steps:

- Ensure that power is applied to the DE2 board
- Connect the supplied USB cable to the USB Blaster port on the DE2 board
- Configure the JTAG programming circuit by setting the RUN/PROG switch (on the left side of the board) to the PROG position.
- The EPCS16 chip can now be programmed by using the Quartus II Programmer module to select a configuration bit stream file with the *.pof* filename extension
- Once the programming operation is finished, set the RUN/PROG switch back to the RUN position and then reset the board by turning the power switch off and back on; this action causes the new configuration data in the EPCS 16 device to be loaded into the FPGA chip.

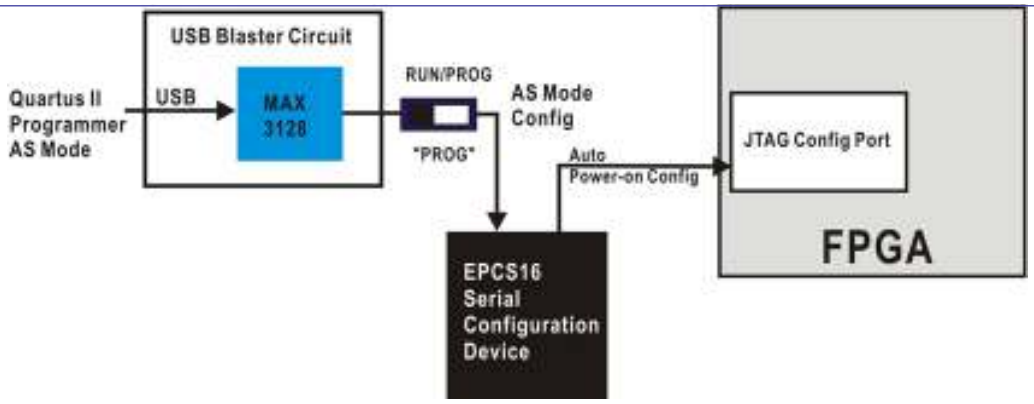


Figure 3.2. The AS configuration scheme.

In addition to its use for JTAG and AS programming, the USB Blaster port on the DE2 board can also be used to control some of the board's features remotely from a host computer.

Chapter 4 :Working of the Design and Code

In this User gets tree attempts to open the lock. The length of the password is of 8 bit, 2 bit for each push buttons. This can be modified to improve the safety .
Initially LCD screen display “WELCOME HOME ENTER YOUR PIN”

Figure 4.1 welcome display

Then as user start entering code following screen is displayed. Once the four digit code is entered, lock is either unlocked or user gets another trial.

Figure 4.2 Enter code display

If the entered code is right. The following message is displayed on LCD screen. Which is “DOOR IS UNLOCKED PLEASE COME IN”

Figure 4.3 Unlocked display

If the entered code is wrong. User gets second trial.

Figure 4.4 Second trial display

If the entered code is wrong in second trial, user gets one more trial.

Figure 4.5 Third trial display

If the entered code is wrong in the third trial. The program enters into Admin mode. In this mode Admin Password has to be entered. This password is different than the normal user password. Unless the correct Admin password is entered, program will stay in the same state. Once the correct code is entered, program will get reset and user gets three trials again to enter the code.

Figure 4.6 Admin Mode display

Once the correct code is entered, LEDs on the board will blink in ring counter pattern.

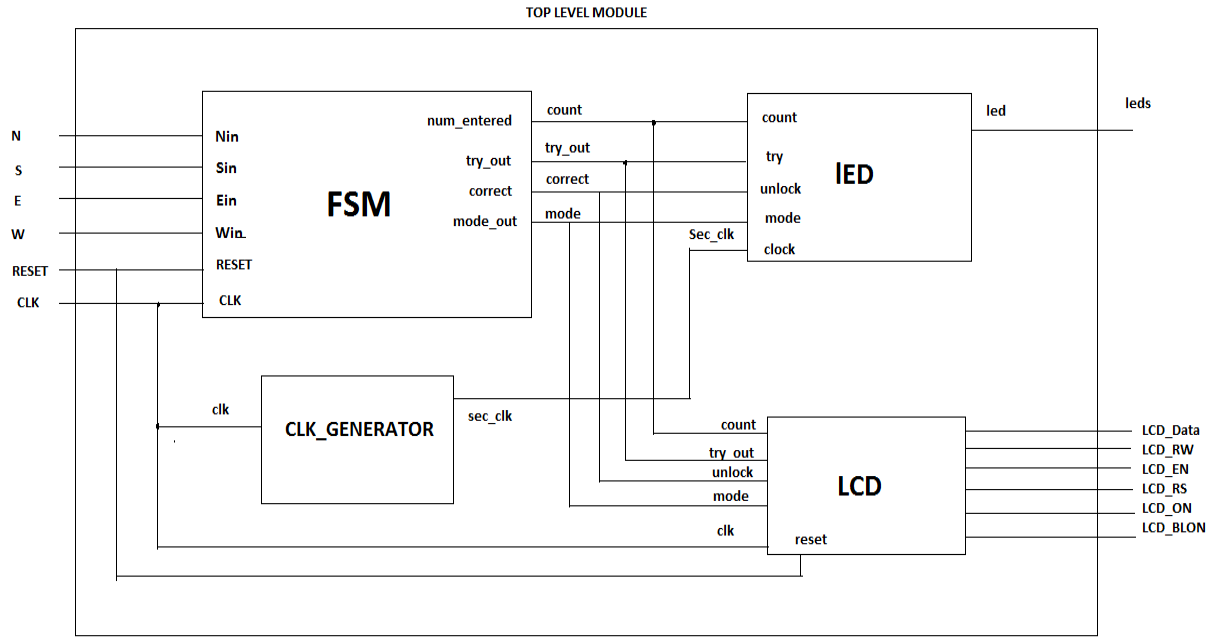


Figure 4.7 Block diagram of Digital Lock

4.1 Design Code for Top level Module

TOP LEVEL DESIGN

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

entity digilock is
        port(clk, reset, N, S, E, W: IN STD_LOGIC;
                leds: out STD_LOGIC_VECTOR(7 DOWNTO 0);
                LCD_DATA: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
                LCD_RW,    LCD_EN, LCD_RS: OUT STD_LOGIC;
                LCD_ON, LCD_BLON: OUT STD_LOGIC);
end digilock;

architecture Behavioral of digilock is
--led_display
        COMPONENT led_display
   PORT(
      count : IN  std_logic_vector(2 downto 0);
      clk : IN  std_logic;
      mode : IN  std_logic;
      unlock : IN  std_logic;
      try : IN  std_logic_vector(1 downto 0);
      led : OUT  std_logic_vector(7 downto 0)

      );
```

```
   END COMPONENT;

--FSM
   COMPONENT FSM
   PORT(
      Nin : IN  std_logic;
      Sin : IN  std_logic;
      Ein : IN  std_logic;
      Win : IN  std_logic;
      clk : IN  std_logic;
      reset : IN  std_logic;
      num_entered : OUT  std_logic_vector(2 downto 0);
      state_out : OUT  std_logic_vector(2 downto 0);
      try_out: OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
      mode_out : OUT  std_logic;
      correct : OUT  std_logic
     );
   END COMPONENT;


--LCDdigilock
        COMPONENT LCDdigilock
        PORT (
                Clk50Mhz, reset, unlock, mode: IN STD_LOGIC;  --clk, mode, count, try,
unlock
                count_key: in STD_LOGIC_VECTOR(2 DOWNTO 0);
                try: in STD_LOGIC_VECTOR(1 DOWNTO 0);
                LCD_DATA: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
                LCD_RW,      LCD_EN, LCD_RS: OUT STD_LOGIC;
                LCD_ON, LCD_BLON: OUT STD_LOGIC);
        END COMPONENT;

--clk generator
        component clockGenerator
        port(
                sysClk:         in std_logic;--50Mhz clock
                msClk :         out std_logic;
                secClk: out std_logic);
        end component;
        SIGNAL msclk, secClk: STD_LOGIC;
        SIGNAL mode, correct: STD_LOGIC;
        SIGNAL count: std_logic_vector(2 downto 0);
        SIGNAL try: STD_LOGIC_VECTOR(1 DOWNTO 0);
        SIGNAL state_out: STD_LOGIC_VECTOR(2 DOWNTO 0);

begin
        clkgen: clockGenerator port map(sysclk=>clk, msclk=>msclk, secClk=>secClk);
```

```
state_machine: fsm port map(  Nin => not(N),  Sin => not(S),   Ein => not(E), Win =>
not(W),  clk => clk,  reset => reset,  num_entered => count,  state_out => state_out,
try_out=>try,   mode_out => mode,  correct => correct   );
led: led_display port map(count=>count, clk=>secClk,mode=>mode, unlock=>correct,
try=>try, led=>leds);
lcd: LCDdigilock port map(Clk50Mhz=>clk, reset=>reset, unlock=>correct,
mode=>mode,count_key=>count,try=>try,LCD_DATA=>LCD_DATA,LCD_RW=>LCD
_RW, LCD_EN=>LCD_EN, LCD_RS=>LCD_RS,LCD_ON=>LCD_ON,
LCD_BLON=>LCD_BLON);
end Behavioral;
```

4.2 desigh code for Clock Generator

CLOCK GENERATOR

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

entity clockGenerator is
        port(
                sysClk:         in std_logic;--50Mhz clock
                msClk :         out std_logic;
                secClk: out std_logic);
end clockGenerator;

architecture Behavioral of clockGenerator is
begin
        process (sysclk)
                variable cnt: integer range 0 to 50000 := 0;
                variable mscnt:         integer range 0 to 1000 := 0;
        begin
                if(sysClk'event and sysClk='1')then
                                        cnt:=cnt+1;
                        --generates msclk
                        if (cnt=25000) then --half a millisecond?
                                msClk<='1';
                        elsif (cnt=50000) then --a full millisecond?
                                msClk <= '0';
                                cnt:= 0;
                                msCnt:= msCnt + 1;
                        end if;

                        --generates secClk
                        if (msCnt = 500)then --half a second?
                                secclk<='1';
                        elsif (mscnt = 1000) then --a full second?
                                secClk <='0';
```

```
                    mscnt:=0;
                end if;

            end if;
        end process;
end Behavioral;
```

4.3 Using the LCD Module

The LCD module has built-in fonts and can be used to display text by sending appropriate commands to the display controller, which is called HD44780. Detailed information for using the display is available in its datasheet, which can be found on the manufacturer's web site, and from the *Datasheet* folder on the **DE2 System CD-ROM.** A schematic diagram of the LCD module showing connections to the Cyclone II FPGA is given in Figure 4.8. The associated pin assignments appear in Table 4.6.

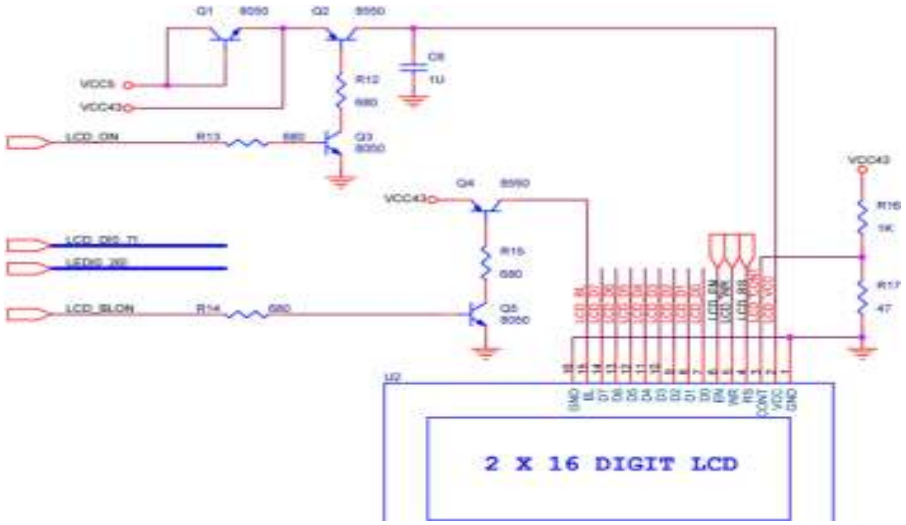


Figure 4.8 Schematic diagram of the LCD module.

Signal Name	FPGA Pin No.	Description
LCD_DATA[0]	PIN_J1	LCD Data[0]
LCD_DATA[1]	PIN_J2	LCD Data[1]
LCD_DATA[2]	PIN_H1	LCD Data[2]
LCD_DATA[3]	PIN_H2	LCD Data[3]
LCD_DATA[4]	PIN_J4	LCD Data[4]
LCD_DATA[5]	PIN_J3	LCD Data[5]

LCD_DATA[6]	PIN_H4	LCD Data[6]
LCD_DATA[7]	PIN_H3	LCD Data[7]
LCD_RW	PIN_K4	LCD Read/Write Select, 0 = Write, 1 = Read
LCD_EN	PIN_K3	LCD Enable
LCD_RS	PIN_K1	LCD Command/Data Select, 0 = Command, 1 = Data
LCD_ON	PIN_L4	LCD Power ON/OFF
LCD_BLON	PIN_K2	LCD Back Light ON/OFF

Table 4.1. Pin assignments for the LCD module.

Note: The user can specify any pattern for character-generator RAM.

Table 4.2 Pattern for character-generator RAM

Character to be displayed are written in “chararray” in the code. Each character has 8 bit value. This value is as per the table above.
For example to display character “W”, It’s upper four bits are 0101 and lower four bits are 0111. So it’s equivalent Hex value is 57.

So X”57” will display “W” on LCD screen.

This are the few predefined code by HITACHI for reset, set, Off , On etc for LCD

Reset	Data_bus _value=x”38”
Fun_set	Data_bus_value=x”30”
Display off	Data_bus_value=x”08”
Display clear	Data_bus_value=x”01”
Display On	Data_bus_value=x”0C”
Mode_set	Data_bus_value=X”06”

4.3.1 Design code for LCD controller

LCD display controller

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY LCDdigilock IS
PORT (
        Clk50Mhz, reset, unlock, mode: IN STD_LOGIC;  --clk, mode, count, try, unlock
        count_key: in STD_LOGIC_VECTOR(2 DOWNTO 0);
        try: in STD_LOGIC_VECTOR(1 DOWNTO 0);
        LCD_DATA: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        LCD_RW,      LCD_EN, LCD_RS: OUT STD_LOGIC;
        LCD_ON, LCD_BLON: OUT STD_LOGIC);
END LCDdigilock;

ARCHITECTURE FSMD OF LCDdigilock IS
        TYPE state_type IS (s1,s2,s3,s4,s10,s11,s12,s13,s20,s21,s22,s23,s24);
        SIGNAL state: state_type;

        CONSTANT max: INTEGER := 50000; --50000
        CONSTANT half: INTEGER := max/2;
        SIGNAL clockticks: INTEGER RANGE 0 TO max;
        SIGNAL clock: STD_LOGIC;

        SUBTYPE ascii IS STD_LOGIC_VECTOR(7 DOWNTO 0);
        TYPE charArray IS array(1 to 16) OF ascii;
        TYPE initArray IS array(1 to 7) OF ascii;
        -- LCD initialization sequence codes
        -- 0x38 init four times
        -- 0x06 Entry mode set: Increment One; No Shift
        -- 0x0F Display control: Display ON; Cursor ON; Blink ON
        -- 0x01 Display clear
        CONSTANT initcode: initArray := (x"38",x"38",x"38",x"38",x"06",x"0F",x"01");
        -- WELCOME HOME
        CONSTANT line1: charArray :=
```

```
(x"20",x"20",x"57",x"45",x"4c",x"43",x"4f",x"4d",x"45",x"20",x"48",x"4f",x"4d",x"45",x
"20",x"20");
        -- ENTER YOUR PIN
        CONSTANT line2: charArray :=
(x"20",x"45",x"4e",x"54",x"45",x"52",x"20",x"59",x"4f",x"55",x"52",x"20",x"50",x"49",x
"4e",x"20");
        -- _ _ _TRIAL_2/4_ _ _ _
        CONSTANT line3: charArray :=
(x"20",x"20",x"20",x"54",x"52",x"49",x"41",x"4C",x"20",x"32",x"2F",x"33",x"20",x"20",
x"20",x"20");
        -- _ _ _TRIAL_3/4_ _ _ _
        CONSTANT line4: charArray :=
(x"20",x"20",x"20",x"54",x"52",x"49",x"41",x"4C",x"20",x"33",x"2F",x"33",x"20",x"20",
x"20",x"20");
        -- _ _ _TRIAL_4/4_ _ _ _
        --CONSTANT line5: charArray :=
(x"20",x"20",x"20",x"54",x"52",x"49",x"41",x"4C",x"20",x"34",x"2F",x"34",x"20",x"20",
x"20",x"20");
        -- ENTER ADMIN CODE
   CONSTANT line6: charArray :=
(x"45",x"4e",x"54",x"45",x"52",x"20",x"41",x"44",x"4D",x"49",x"4E",x"20",x"43",x"4F",
x"44",x"45");
        -- DOOR IS UNLOCKED
        CONSTANT line7: charArray :=
(x"44",x"4F",x"4F",x"52",x"20",x"49",x"53",x"20",x"55",x"4E",x"4C",x"4F",x"43",x"4B"
,x"45",x"44");
        -- YOUR CODE *---
        CONSTANT line8: charArray :=
(x"20",x"59",x"4F",x"55",x"52",x"20",x"43",x"4F",x"44",x"45",x"20",x"20",x"2A",x"2D"
,x"2D",x"2D");
   -- YOUR CODE **--
        CONSTANT line9: charArray :=
(x"20",x"59",x"4F",x"55",x"52",x"20",x"43",x"4F",x"44",x"45",x"20",x"20",x"2A",x"2A"
,x"2D",x"2D");
        -- YOUR CODE ***-
        CONSTANT line10: charArray :=
(x"20",x"59",x"4F",x"55",x"52",x"20",x"43",x"4F",x"44",x"45",x"20",x"20",x"2A",x"2A"
,x"2A",x"2D");
        -- YOUR CODE ****
        CONSTANT line11: charArray :=
(x"20",x"59",x"4F",x"55",x"52",x"20",x"43",x"4F",x"44",x"45",x"20",x"20",x"2A",x"2A"
,x"2A",x"2A");
        -- PLEASE COME IN
        CONSTANT line12: charArray :=
(x"20",x"50",x"4C",x"45",x"41",x"53",x"45",x"20",x"43",x"4F",x"4D",x"45",x"20",x"49",
x"4E",x"20");
```

```
        SIGNAL count: INTEGER;

BEGIN

        LCD_ON <= '1';
        LCD_BLON <= '0';

        lcd_control: PROCESS(clock,unlock,try,count_key,mode,reset)   --clk, mode,
count, try, unlock
        BEGIN
                IF(Reset = '1') THEN
                        count <= 1;
                        state <= s1;
                ELSIF(clock'EVENT AND clock = '1') THEN


                CASE state IS
                        -- LCD initialization sequence
                        -- The LCD_DATA is written to the LCD at the falling edge of the E
line
                        -- therefore we need to toggle the E line for each data write
                        WHEN s1 =>
                                LCD_DATA <= initcode(count);
                                LCD_EN <= '1';          -- EN=1;
                                LCD_RS <= '0';          -- RS=0; an instruction
                                LCD_RW <= '0';          -- R/W'=0; write
                                state <= s2;
                        WHEN s2 =>
                                LCD_EN <= '0';          -- set EN=0;
                                count <= count + 1;
                                IF count + 1 <= 7 THEN
                                        state <= s1;
                                ELSE
                                        state <= s10;
                                END IF;

                        -- move cursor to first line of display
                        WHEN s10 =>
                                LCD_DATA <= x"80";          -- x80 is address of 1st position
on first line
                                LCD_EN <= '1';          -- EN=1;
                                LCD_RS <= '0';          -- RS=0; an instruction
                                LCD_RW <= '0';          -- R/W'=0; write
                                state <= s11;
                        WHEN s11 =>
                                LCD_EN <= '0';          -- EN=0; toggle EN
                                count <= 1;
```

```
                state <= s12;

            -- write 1st line text
            WHEN s12 =>
            ------------------------------------------

            if(unlock='0') then--locked
                if(mode='0') then --usr mode
                    --led(7)<='1';
                    if(try="00") then
                        --led(6 DOWNTO 4)<="000";   -- welcome
home
                        LCD_DATA <= line1(count);
                        LCD_EN <= '1';         -- EN=1;
                        LCD_RS <= '1';         -- RS=1; data
                        LCD_RW <= '0';         -- R/W'=0; write
                        state <= s13;
                    elsif(try="01") then
                        --led(6 DOWNTO 4)<="001";   -- trial 2/3
                        LCD_DATA <= line3(count);
                        LCD_EN <= '1';         -- EN=1;
                        LCD_RS <= '1';         -- RS=1; data
                        LCD_RW <= '0';         -- R/W'=0; write
                        state <= s13;

                    elsif(try="10") then
                        --led(6 DOWNTO 4)<="011";   -- trial 3/3
                        LCD_DATA <= line4(count);
                        LCD_EN <= '1';         -- EN=1;
                        LCD_RS <= '1';         -- RS=1; data
                        LCD_RW <= '0';         -- R/W'=0; write
                        state <= s13;
                    else--(try="11") then
                        --led(6 DOWNTO 4)<="111";   -- trial 4/4
                        --LCD_DATA <= line5(count);
                        --LCD_EN <= '1';       -- EN=1;
                        --LCD_RS <= '1';       -- RS=1; data
                        --LCD_RW <= '0';       -- R/W'=0; write
                        state <= s13;
                    end if;
                else --admin mode
                    --led(7 DOWNTO 4)<=X"F";        -- enter admin
code
                        LCD_DATA <= line6(count);
                        LCD_EN <= '1';         -- EN=1;
                        LCD_RS <= '1';         -- RS=1; data
                        LCD_RW <= '0';         -- R/W'=0; write
```

```
                                state <= s13;

                        end if;



                else
                        --led<=temp_count;              -- door is unlocked
                                        LCD_DATA <= line7(count);
                                        LCD_EN <= '1';          -- EN=1;
                                        LCD_RS <= '1';          -- RS=1; data
                                        LCD_RW <= '0';          -- R/W'=0; write
                                        state <= s13;
                end if;

                -------------------------------------
                -- if (unlock = '0' ) then
                --      LCD_DATA <= line3(count);
                --      LCD_EN <= '1';          -- EN=1;
                --      LCD_RS <= '1';          -- RS=1; data
                --      LCD_RW <= '0';          -- R/W'=0; write
                --      state <= s13;
                --      else
                --  LCD_DATA <= line1(count);
                --   LCD_EN <= '1';   -- EN=1;
                --      LCD_RS <= '1';          -- RS=1; data
                --      LCD_RW <= '0';          -- R/W'=0; write
                --      state <= s13;
                --  end if;
                 ----------------------------------------

                WHEN s13 =>
                        LCD_EN <= '0';          -- EN=0; toggle EN
                        count <= count + 1;
                        IF count + 1 <= 16 THEN
                                state <= s12;
                        ELSE
                                state <= s20;
                        END IF;

                -- move cursor to second line of display
                WHEN s20 =>
                        LCD_DATA <= x"BF";                  -- xBF is address of 1st
position on second line
                        LCD_EN <= '1';          -- EN=1;
                        LCD_RS <= '0';          -- RS=0; an instruction
                        LCD_RW <= '0';          -- R/W'=0; write
```

```
        state <= s21;
WHEN s21 =>
        LCD_EN <= '0';         -- EN=0; toggle EN
        count <= 1;
        state <= s22;

-- write 2nd line text
WHEN s22 =>


------------------
if(unlock='0') then--locked
        if(count_key="000") then
                --led(3 DOWNTO 0)<=X"0";       -- enter your pin

                LCD_DATA <= line2(count);
                LCD_EN <= '1';         -- EN=1;
                LCD_RS <= '1';         -- RS=1; data
                LCD_RW <= '0';         -- R/W'=0; write
                state <= s23;
        elsif(count_key="001") then
                --led(3 DOWNTO 0)<=X"1";      -- your code *

                LCD_DATA <= line8(count);
                LCD_EN <= '1';         -- EN=1;
                LCD_RS <= '1';         -- RS=1; data
                LCD_RW <= '0';         -- R/W'=0; write
                state <= s23;
        elsif(count_key="010") then
                --led(3 DOWNTO 0)<=X"3";             -- your code **

                LCD_DATA <= line9(count);
                LCD_EN <= '1';         -- EN=1;
                LCD_RS <= '1';         -- RS=1; data
                LCD_RW <= '0';         -- R/W'=0; write
                state <= s23;
        elsif(count_key="011") then
                --led(3 DOWNTO 0)<=X"7";     -- your code ***

                LCD_DATA <= line10(count);
                LCD_EN <= '1';         -- EN=1;
                LCD_RS <= '1';         -- RS=1; data
                LCD_RW <= '0';         -- R/W'=0; write
                state <= s23;
        elsif(count_key="100") then
                --led(3 DOWNTO 0)<=X"F";      -- your code ****
```

```
                    LCD_DATA <= line11(count);
                    LCD_EN <= '1';          -- EN=1;
                    LCD_RS <= '1';          -- RS=1; data
                    LCD_RW <= '0';          -- R/W'=0; write
                    state <= s23;
                end if;
            else
               -- Please Come In
                    LCD_DATA <= line12(count);
                    LCD_EN <= '1';          -- EN=1;
                    LCD_RS <= '1';          -- RS=1; data
                    LCD_RW <= '0';          -- R/W'=0; write
                    state <= s23;
               end if;

               --------------------

            -- if (unlock = '0') then
            --      LCD_DATA <= line4(count);
            --      LCD_EN <= '1';          -- EN=1;
            -- LCD_RS <= '1';     -- RS=1; data
            --      LCD_RW <= '0';          -- R/W'=0; write
            --      state <= s23;

            --      else
--  LCD_DATA <= line2(count);
            --  LCD_EN <= '1';     -- EN=1;
            --      LCD_RS <= '1';          -- RS=1; data
            --      LCD_RW <= '0';          -- R/W'=0; write
            --      state <= s23;
            --  end if;

                    ------------------------------------------------------------

            WHEN s23 =>
                    LCD_EN <= '0';          -- set EN=0;
                    count <= count + 1;
                    IF count+1 <= 16 THEN
                            state <= s22;
                    ELSE
                            state <= s1;
                    END IF;

            WHEN OTHERS =>
                    LCD_EN <= '0';
                    state <= s1;
            END CASE;
```

```
        END IF;

    END PROCESS;

    ClockDivide: PROCESS
    BEGIN
        WAIT UNTIL Clk50Mhz'EVENT and Clk50Mhz = '1';
        IF clockticks < max THEN
            clockticks <= clockticks + 1;
        ELSE
            clockticks <= 0;
        END IF;
        IF clockticks < half THEN
            clock <= '0';
        ELSE
            clock <= '1';
        END IF;
    END PROCESS;
END FSMD;
```

4.4 Using the LEDs and Switches

The DE2 board provides four pushbutton switches. Each of these switches is debounced using a Schmitt Trigger circuit.. The four outputs called *KEY0, …, KEY3* of the Schmitt Trigger device are connected directly to the Cyclone II FPGA. Each switch provides a high logic level (3.3 volts) when it is not pressed, and provides a low logic level (0 volts) when depressed. Since the pushbutton switches are debounced, they are appropriate for use as clock or reset inputs in a circuit.

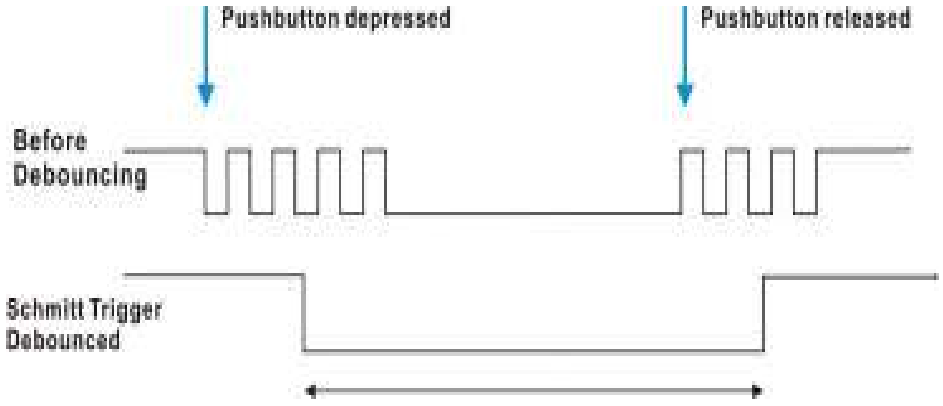


Figure 4.9. Switch debouncing.

There are also 18 toggle switches (sliders) on the DE2 board. These switches are not debounced, and are intended for use as level-sensitive data inputs to a circuit. Each

switch is connected directly to a pin on the Cyclone II FPGA. When a switch is in the DOWN position (closest to the edge of the board) it provides a low logic level (0 volts) to the FPGA, and when the switch is in the UP position it provides a high logic level (3.3 volts)

There are 27 user-controllable LEDs on the DE2 board. Eighteen red LEDs are situated above the 18 toggle switches, and eight green LEDs are found above the pushbutton switches (the 9th green LED is in the middle of the 7-segment displays). Each LED is driven directly by a pin on the Cyclone II FPGA; driving its associated pin to a high logic level turns the LED on, and driving the pin low turns it off. A schematic diagram that shows the pushbutton and toggle switches is given in Figure 4.10. A schematic diagram that shows the LED circuitry appears in Figure 4.11.

A list of the pin names on the Cyclone II FPGA that are connected to the toggle switches is given in Table 4.3. Similarly, the pins used to connect to the pushbutton switches and LEDs are displayed in Tables 4.4 and 4.5, respectively.

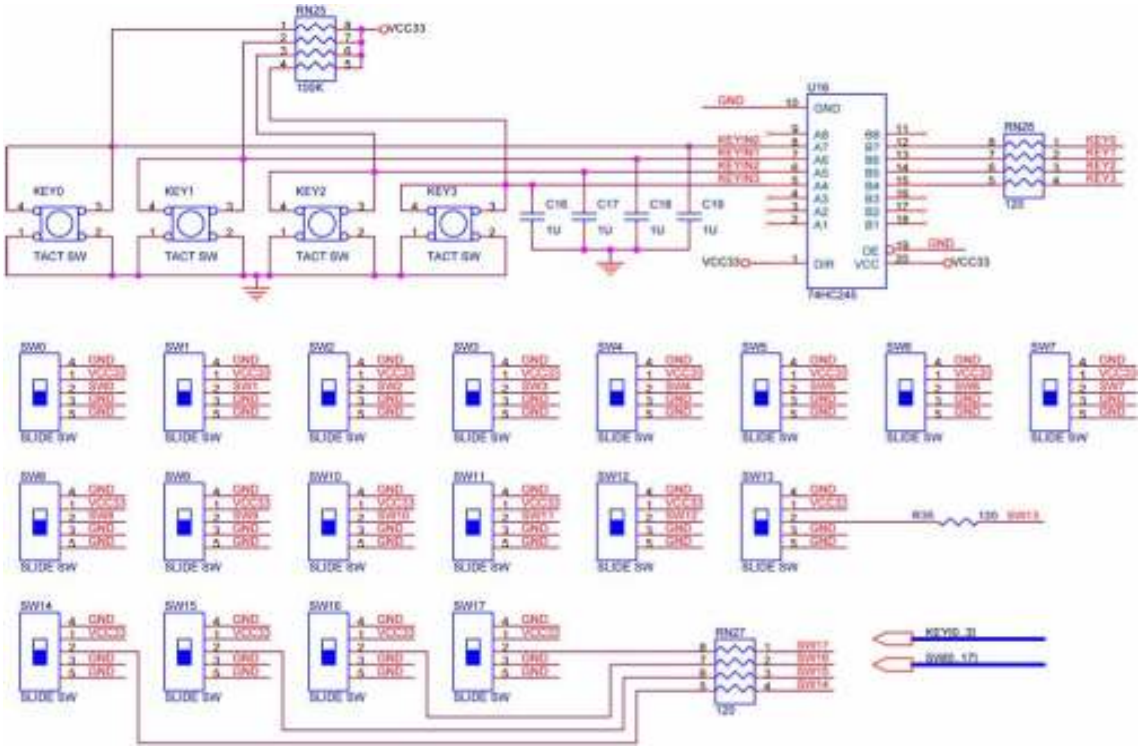


Figure 4.10 Schematic diagram of the pushbutton and toggle switches.

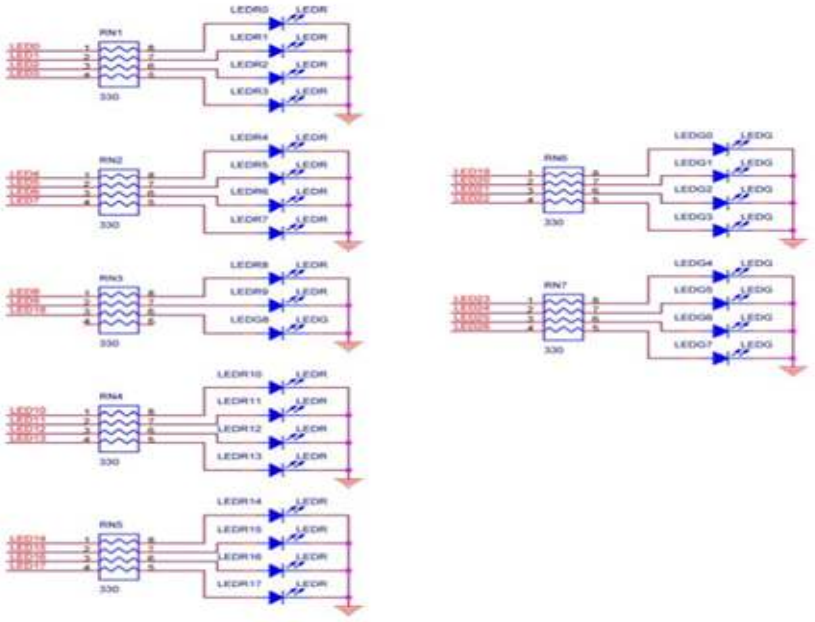


Figure 4.11. Schematic diagram of the LEDs.

Signal Name	FPGA Pin No.	Description
SW[0]	PIN_N25	Toggle Switch[0]
SW[1]	PIN_N26	Toggle Switch[1]
SW[2]	PIN_P25	Toggle Switch[2]
SW[3]	PIN_AE14	Toggle Switch[3]
SW[4]	PIN_AF14	Toggle Switch[4]
SW[5]	PIN_AD13	Toggle Switch[5]
SW[6]	PIN_AC13	Toggle Switch[6]
SW[7]	PIN_C13	Toggle Switch[7]
SW[8]	PIN_B13	Toggle Switch[8]
SW[9]	PIN_A13	Toggle Switch[9]
SW[10]	PIN_N1	Toggle Switch[10]
SW[11]	PIN_P1	Toggle Switch[11]
SW[12]	PIN_P2	Toggle Switch[12]
SW[13]	PIN_T7	Toggle Switch[13]
SW[14]	PIN_U3	Toggle Switch[14]
SW[15]	PIN_U4	Toggle Switch[15]
SW[16]	PIN_V1	Toggle Switch[16]
SW[17]	PIN_V2	Toggle Switch[17]

Table 4.3. Pin assignments for the toggle switches

Signal Name	FPGA Pin No.	Description
KEY[0]	PIN_G26	Pushbutton[0]
KEY[1]	PIN_N23	Pushbutton[1]
KEY[2]	PIN_P23	Pushbutton[2]
KEY[3]	PIN_W26	Pushbutton[3]

Table 4.4. Pin assignments for the pushbutton switches.

Signal Name	FPGA Pin No.	Description
LEDR[0]	PIN_AE23	LED Red[0]
LEDR[1]	PIN_AF23	LED Red[1]
LEDR[2]	PIN_AB21	LED Red[2]
LEDR[3]	PIN_AC22	LED Red[3]
LEDR[4]	PIN_AD22	LED Red[4]
LEDR[5]	PIN_AD23	LED Red[5]
LEDR[6]	PIN_AD21	LED Red[6]
LEDR[7]	PIN_AC21	LED Red[7]
LEDR[8]	PIN_AA14	LED Red[8]
LEDR[9]	PIN_Y13	LED Red[9]
LEDR[10]	PIN_AA13	LED Red[10]
LEDR[11]	PIN_AC14	LED Red[11]
LEDR[12]	PIN_AD15	LED Red[12]
LEDR[13]	PIN_AE15	LED Red[13]
LEDR[14]	PIN_AF13	LED Red[14]
LEDR[15]	PIN_AE13	LED Red[15]
LEDR[16]	PIN_AE12	LED Red[16]
LEDR[17]	PIN_AD12	LED Red[17]
LEDG[0]	PIN_AE22	LED Green[0]
LEDG[1]	PIN_AF22	LED Green[1]
LEDG[2]	PIN_W19	LED Green[2]
LEDG[3]	PIN_V18	LED Green[3]
LEDG[4]	PIN_U18	LED Green[4]

LEDG[5]	PIN_U17	LED Green[5]
LEDG[6]	PIN_AA20	LED Green[6]
LEDG[7]	PIN_Y18	LED Green[7]
LEDG[8]	PIN_Y12	LED Green[8]

Table 4.5. Pin assignments for the LEDs.

4.4.1 Design code of LED display

LED DISPAY

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

entity led_display is
	port( count: in STD_LOGIC_VECTOR(2 DOWNTO 0);
			clk, mode, unlock: in STD_LOGIC;
			try: in STD_LOGIC_VECTOR(1 DOWNTO 0);
			led: out STD_LOGIC_VECTOR(7 DOWNTO 0));
end led_display;

architecture Behavioral of led_display is
	signal temp_count : std_logic_vector(7 downto 0);
begin
	process(clk, mode, count, try, unlock)
	begin
		if(clk'EVENT AND clk='1') then
			if(unlock='0') then--locked
				if(mode='0') then --usr mode
					led(7)<='1';
					if(try="00") then
						led(6 DOWNTO 4)<="000";
					elsif(try="01") then
						led(6 DOWNTO 4)<="001";
					elsif(try="10") then
						led(6 DOWNTO 4)<="011";
					elsif(try="11") then
						led(6 DOWNTO 4)<="111";
					end if;
				else --admin mode
					led(7 DOWNTO 4)<=X"F";
				end if;

				if(count="000") then
```

```
                led(3 DOWNTO 0)<=X"0";
            elsif(count="001") then
                led(3 DOWNTO 0)<=X"1";
            elsif(count="010") then
                led(3 DOWNTO 0)<=X"3";
            elsif(count="011") then
                led(3 DOWNTO 0)<=X"7";
            elsif(count="100") then
                led(3 DOWNTO 0)<=X"F";
            end if;
        else --unlcocked
            case temp_count is
                when "00000001" => temp_count
<="00000010";
                when "00000010" => temp_count
<="00000100";
                when "00000100" => temp_count
<="00001000";
                when "00001000" => temp_count
<="00010000";
                when "00010000" => temp_count
<="00100000";
                when "00100000" => temp_count
<="01000000";
                when "01000000" => temp_count
<="10000000";
                when "10000000" => temp_count
<="00000001";
                when others => temp_count <="00000001";
            end case;
            led<=temp_count;
        end if;
    end if;
  end process;
end Behavioral;
```

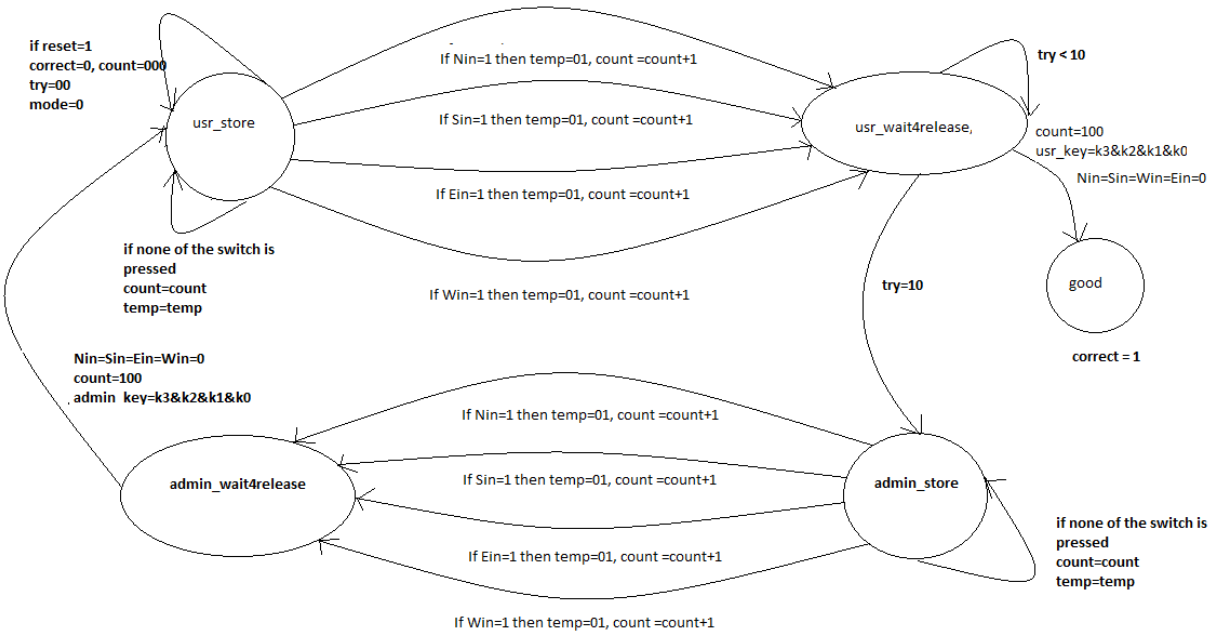


Figure 4.12 STATE MACHINE FOR DIGITAL LOCK

4.5 Design Code of FSM

FSM STATE MACHINE

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

entity FSM is
        port(Nin, Sin, Ein, Win, clk, reset: in STD_LOGIC;
                num_entered: out STD_LOGIC_VECTOR(2 DOWNTO 0);
                state_out: out STD_LOGIC_VECTOR(2 DOWNTO 0);
                try_out: out STD_LOGIC_VECTOR(1 DOWNTO 0);
                mode_out, correct: out STD_LOGIC);
end FSM;

architecture Behavioral of FSM is
        SIGNAL k0, k1, k2, k3: STD_LOGIC_VECTOR(1 DOWNTO 0);
        SIGNAL mode: STD_LOGIC;
        SIGNAL try: STD_LOGIC_VECTOR(1 DOWNTO 0);
        SIGNAL count: STD_LOGIC_VECTOR(2 DOWNTO 0);

  type STATE_TYPE is (usr_store, usr_wait4release, admin_store, admin_wait4release,
good);
```

```
        attribute ENUM_ENCODING: STRING;
        attribute ENUM_ENCODING of STATE_TYPE:type is "000 001 010 011 100";
        signal state: STATE_TYPE;

         CONSTANT usr_key: STD_LOGIC_VECTOR(7 DOWNTO 0):= "11100100";--
enter user key as desired in reverse order. 4ht3rd2nd1st
         CONSTANT admin_key: STD_LOGIC_VECTOR(7 DOWNTO 0):= "00011011";

begin
        process(state,clk, Nin, Win, Sin, Ein, reset, count, try, mode, k0,k1,k2,k3)
        variable temp: STD_LOGIC_VECTOR(1 DOWNTO 0);
        begin
                if(reset='1') then
                        state<=usr_store;
                        correct<='0';
                        count<="000";
                        try<="00";
                        mode<='0';
                else
                  if(clk'EVENT AND clk='1') then
                        case state is
                                when usr_store=>
                                                if(Nin='1') then
                                                        temp:="00";
                                                        count<=count+1;
                                                        state<=usr_wait4release;
                                                elsif(Sin='1') then
                                                        temp:="01";
                                                        count<=count+1;
                                                        state<=usr_wait4release;
                                                elsif(Ein='1') then
                                                        temp:="10";
                                                        count<=count+1;
                                                        state<=usr_wait4release;
                                                elsif(Win='1') then
                                                        temp:="11";
                                                        count<=count+1;
                                                        state<=usr_wait4release;
                                                else
                                                        count<=count;
                                                        temp:=temp;
                                                        state<=state;
                                                end if;
                                                if(count="000") then
                                                        k0<=temp;
                                                elsif(count="001") then
                                                        k1<=temp;
```

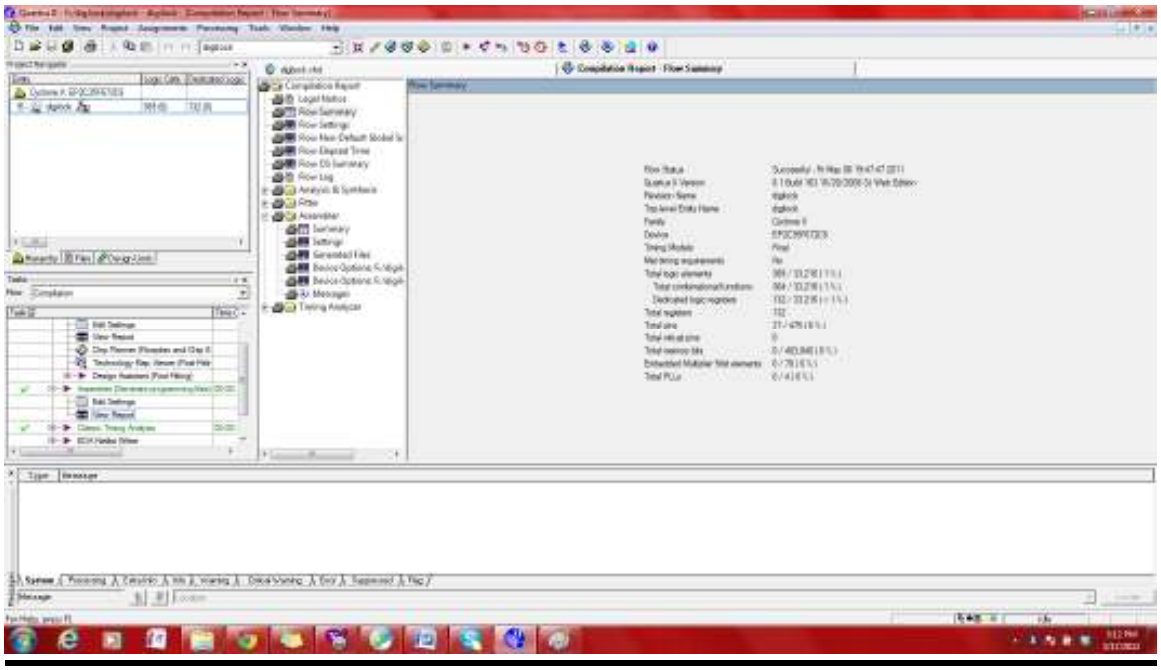
```
                                        elsif(count="010") then
                                                k2<=temp;
                                        elsif(count="011") then

                                                k3<=temp;
                                        end if;
                                when usr_wait4release=>
                                        if(Nin='0' AND Sin='0' AND Win='0' AND
Ein='0') then
                                                        if(count="100") then --all
entered
                                                                if(usr_key=(k3 & k2 &
k1 & k0)) then --correct key
                                                                        state<=good;
                                                                else --wrong try
                                                                        if(try="10") then
                                                                        mode<='1';

        state<=admin_store;
                                                                        count<="000";
                                                                else
                                                                        try<=try+1;

state<=usr_store;

        count<="000";
                                                                        end if;
                                                                end if;
                                                        else
                                                                state<=usr_store;
                                                        end if;
                                        end if;

                                when admin_store=>
                                        if(Nin='1') then
                                                temp:="00";
                                                count<=count+1;
                                                state<=admin_wait4release;
                                        elsif(Sin='1') then
                                                temp:="01";
                                                count<=count+1;
                                                state<=admin_wait4release;
                                        elsif(Ein='1') then
                                                temp:="10";
                                                count<=count+1;
                                                state<=admin_wait4release;
                                        elsif(Win='1') then
```

```
                                        temp:="11";
                                        count<=count+1;
                                        state<=admin_wait4release;
                                    else
                                        count<=count;
                                        temp:=temp;
                                        state<=state;
                                    end if;

                                    if(count="000") then
                                        k0<=temp;
                                    elsif(count="001") then
                                        k1<=temp;
                                    elsif(count="010") then
                                        k2<=temp;
                                    elsif(count="011") then
                                        k3<=temp;
                                    end if;

                            when admin_wait4release=>
                                    if(Nin='0' AND Sin='0' AND Win='0' AND
Ein='0') then
                                    if(count="100") then --all entered
                                    if(admin_key=(k3 & k2 & k1 & k0)) then --
correct key

        state<=usr_store;
                                                        count<="000";
                                                        mode<='0';
                                                        try<="00";
                                                    else --wrong try
                                                        count<="000";

        state<=admin_store;
                                                        end if;
                                                    else
                                                        state<=admin_store;
                                                    end if;
                                    end if;
                            when good=>
                                    correct<='1';
                                    state<=good;
                            when OTHERS=> state<=usr_store;
                        end case;
                    end if;
                end if;
        end process;
```

```
        mode_out<=mode;
        num_entered<=count;
        try_out<=try;
        process(state)
        begin
                case state is
                        when usr_store=> state_out<="000";
                        when usr_wait4release=> state_out<="001";
                        when admin_store =>state_out<="010";
                        when admin_wait4release=> state_out<="011";
                        when good=> state_out<="101";
                        when others=> state_out<="111";
                end case;
        end process;
end Behavioral;
```

Pin Planner:

Flow summery:



Analysis and Synthesis Summary

Analysis & Synthesis Summary

Analysis & Synthesis Status	Successful - Sat Mar 17 17:14:22 2012
Quartus II Version	8.1 Build 163 10/28/2008 SJ Web Edition
Revision Name	digilock
Top-level Entity Name	digilock
Family	Cyclone II
Total logic elements	364
Total combinational functions	364
Dedicated logic registers	132
Total registers	132
Total pins	27
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	0
Total PLLs	0

Chapter 5: FIFO structure and Design code

AREA AND TIME OPTIMIZATION OF ASIC "FIFO" DESIGN USING SYNOPSYS DESIGN COMPILER

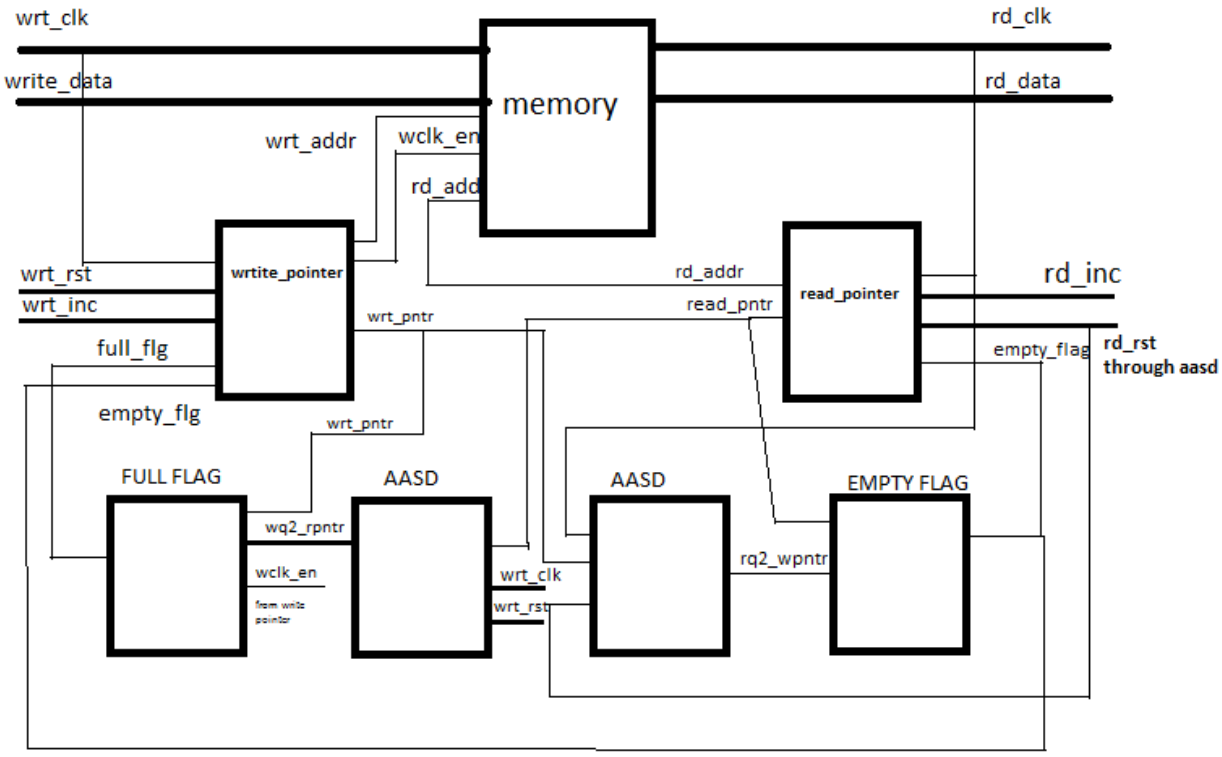


Figure 5.1 Block diagram of FIFO

5.1 design code for Top level module

Top level Module
Filename: fifo_top.v

This is the top module of FIFO design here the aysnchrnously received data from the one clock domain is transferred to the other clock domain. The data received from one clock domain is stored in FIFO and then the second domain reads the data that are stored in FIFO.

```
`timescale 1ns/1ns
```

```
module fifo_top(wrt_clk, wrt_rst, wrt_inc, full_flag, rd_clk, rd_rst, rd_inc,
                empty_flag, write_data, read_data, read_add, write_add,
                   write_pointer, read_pointer);

        parameter length = 16;
        parameter addr = 6;

        input wrt_clk, wrt_rst, wrt_inc, rd_clk, rd_rst, rd_inc;
        input [length-1:0] write_data;
        output [addr-2:0] read_add, write_add;
        output [addr-1:0] write_pointer, read_pointer;
        output full_flag, empty_flag;
        output [length-1:0] read_data;

        wire full_flag, empty_flag;
        wire [length-1:0] read_data;
        wire [addr-1:0] write_pointer, read_pointer;
        wire empty_flg, full_flg, syn_empty_flg;
        wire [addr-2:0] wrt_addr, rd_addr;
        wire [addr-1:0] wrt_pntr, rd_pntr, rq2_wpntr, wq2_rpntr;
        wire wclk_en, wrt_rst_syn, rd_rst_syn;


full_flag #(addr) fl(.write_pointer(wrt_pntr), .read_pointer(wq2_rpntr),
                     .full_flag(full_flg), .clk_en(wclk_en));

empty_flag #(addr) emt(.read_pointer(rd_pntr), .write_pointer(rq2_wpntr),
                       .empty_flag(empty_flg));

memory #(16,5) mem(write_data, wrt_addr, wclk_en, wrt_clk, rd_addr,
                           rd_clk, read_data);

//aasd #(1) empty(empty_flg, wrt_clk, wrt_rst, syn_empty_flg);
 // to synchronize the empty flag coming from read domain into the write domain

aasd #(1) read(rd_rst, rd_clk, rd_rst, rd_rst_syn);

aasd #(1) write(wrt_rst, wrt_clk, wrt_rst, wrt_rst_syn);

aasd #(6) aasd1(rd_pntr, wrt_clk, wrt_rst_syn, wq2_rpntr);

aasd #(6) aasd2(wrt_pntr, rd_clk, rd_rst_syn, rq2_wpntr);

write_pntr #(6) wpt(wrt_clk, wrt_rst_syn, wrt_inc, full_flg, empty_flg, wclk_en,
                wrt_pntr, wrt_addr);
read_pntr #(6) rpt(rd_clk, rd_rst_syn, rd_inc, empty_flg, rd_pntr, rd_addr);
```

```
assign empty_flag = empty_flg;
assign full_flag = full_flg;
assign read_add = rd_addr;
assign write_add = wrt_addr;
assign write_pointer = wrt_pntr;
assign read_pointer = rd_pntr;

endmodule
```

5.2 design code for Lower level module of AASD

<u>This design consist of different modules as shown in above block diagram.</u>

1. AASD (asynchronous assert and synchronouts deassert)
   AASD reset synchrnoizer to avoid metastability state at the end of the reset output

<u>Filename: aasd.v</u>

```
`timescale 1ns/1ns

module aasd(in,clock,reset,aasd);
parameter width = 16;
        input clock, reset;
        input [width-1:0]in;
        output [width-1:0]aasd; // output aasd is of size 8

        reg [width-1:0]aasd;
        reg [width-1:0]temp; //temporary signal used as output of one register
                                // and input to other register

        always@(posedge clock or negedge reset)
        begin
                if(!reset)
                begin
                        temp = 'b0;
                        aasd = 'b0; //when reseted output becomes 0
                end
                else
                begin
                        temp = in;
                        aasd = temp; //always on rising edge of clock if not
                                        //resetted the aasd will get the
                                         //input after one clock cycle delay
                end
        end
endmodule
```

5.3 design code for Lower level module of Full_flag

2. Full_flag

   This code will generate the full flag after comparing the read pointer from the read clock domain which will be synchronized into the write clock domain and then will be compared with the write pointer

Filename: full_flag.v

This code will generate the full flag after comparing the read pointer from
the read clock domain which will be synchronized into the write clock domain and
then will be compared with the write pointer

```
`timescale 1ns/1ns

module full_flag(write_pointer, read_pointer, full_flag, clk_en);

        parameter addr = 6;

        input [addr-1:0] write_pointer, read_pointer;
        input clk_en;
        output full_flag;

        reg full_flag;

always@(read_pointer, write_pointer)
begin
        if((write_pointer[addr-1] != read_pointer[addr-1]) &&
          (write_pointer[addr-2] != read_pointer[addr-2]) &&
          (write_pointer[addr-3:0] == read_pointer[addr-3:0]) || !clk_en)
                full_flag = 1'b1; //if the first two MSB's are opposite and
                                  //then all other bits are equal then full
                                  //flag will go high
        else
                full_flag = 1'b0;// else it will remain low
end
endmodule
```

5.4 design code for Lower level module of Empty_flag

3. Empty_flag

   This code will generate the empty flag after comparing the write pointer from the write clock domain which will be synchronized into the read clock domain and then will be compared with the read pointer

   Filename: empty_flag.v

   This code will generate the empty flag after comparing the write pointer from
   the write clock domain which will be synchronized into the read clock domain and

then will be compared with the read pointer

```
`timescale 1ns/1ns

module empty_flag(read_pointer, write_pointer, empty_flag);

        parameter size = 6;

        input [size-1:0] read_pointer, write_pointer;
        output empty_flag;

        reg empty_flag;

always@(read_pointer, write_pointer)
begin
        if(read_pointer == write_pointer)
                empty_flag = 1'b1; //if gray pointer of read and write are equal
                                   // then the empty flag will go high
        else
                empty_flag = 1'b0; //else it will remain low
end
endmodule
```

5.5 design code for Lower level module of Write_pntr

4. Write_pntr

   Generating the binary address to address the memory location of FIFO and then converting the binary address to gray address to send it to the other clock domain where the gray code will be used to compare the address. The gray code will be generated by designing the function which will covert binary code to gray code

   Filename: write_pntr.v

```
`timescale 1ns/1ns

module write_pntr(clock, reset, incrementer, write_flag, empty_flag, wclk_en,
          write_pointer, write_address);

parameter size = 6;  // declaring the size of the pointer and address

input clock, reset, incrementer, write_flag, empty_flag;
output wclk_en;
output [size-1:0] write_pointer;  // for the gray code pointer crossing the clock
domain
output [size-2:0] write_address;  // for the binary pointer to address the FIFO
memory location
```

```
reg wclk_en;
reg [size-1:0] write_pointer;
reg [size-2:0] write_address;
reg [size-1:0] bnext;

// Defining the function to convert binary to gray
function [size-1:0] binary_gray;
input [size-1:0] binary;
        begin
                binary_gray[size-1] = binary[size-1];
                binary_gray[size-2:0] = binary[size-1:1] ^
                                        binary[size-2:0];
        end
endfunction

//writing the always block to generate the pointer
always@(posedge clock or negedge reset)
begin
if(!reset)
  begin
        write_pointer = 'b0;
        write_address = 'b0;
        bnext = 'b0;
  end
else
  begin
        write_address = bnext[size-2:0];
        write_pointer = binary_gray(bnext);
        if(incrementer && !write_flag)
        begin
                bnext = bnext + 1;
                wclk_en = 1'b1;
        end
        else
        begin
                if(empty_flag)
                begin
                        bnext = bnext + 1;
                        wclk_en = 1'b1;
                end
                else
                begin
                        bnext = bnext + 0;
                        wclk_en = 1'b0;
                end
        end
  end
```

```
    end
endmodule
```

5.6 design code for Lower level module of Read_pntr

5. Read_pntr

   Generating the binary address to address the memory location of FIFO and then converting the binary address to gray address to send it to the other clock domain where the gray code will be used to compare the address. The gray code will be generated by designing the function which will covert binary code to gray code

   Filename: read_pntr.v

```
`timescale 1ns/1ns

module read_pntr(clock, reset, incrementer, flag, read_pointer, read_address);

parameter size = 6;  // declaring the size of the pointer and address

input clock, reset, incrementer, flag;
output [size-1:0] read_pointer;  // for the gray code pointer crossing the clock domain
output [size-2:0] read_address;  // for the binary pointer to address the FIFO memory location

reg [size-1:0] read_pointer;
reg [size-2:0] read_address;
reg [size-1:0] b_next;

// Defining the function to convert binary to gray
function [size-1:0] bin_gray;
input [size-1:0] bin;
        begin
                bin_gray[size-1] = bin[size-1];
                bin_gray[size-2:0] = bin[size-1:1] ^ bin[size-2:0];
        end
endfunction

//writing the always block to generate the pointer
always@(posedge clock or negedge reset)
begin
if(!reset)
  begin
        read_pointer = 'b0;
        read_address = 'b0;
        b_next = 'b0;
```

```
  end
 else
  begin
        read_address = b_next[size-2:0];
        read_pointer = bin_gray(b_next);
        if(incrementer && !flag)
              b_next = b_next + 1;
        else
              b_next = b_next + 0;
  end
 end
endmodule
```

5.7 design code for Lower level module of Memory

6. Merory

Filename: memory.v

Here is a design of a memory with parallel read and write. Read clock will enable to read data from the memory and write clock will allow us to write to the memory

```
`timescale 1ns/1ns

module memory(wdata, waddr, w_clken, wclk, raddr, rclk, rdata);

//defining the parameter for width and length of the FIFO memory
parameter width = 16;
parameter addr = 5;
parameter length = 2**addr;

//defining the input and output of the memory
input [width-1:0] wdata;   //write data
input [addr-1:0] waddr, raddr;   //write address and read address
input wclk, rclk, w_clken;   //write clock, read clock, and write
                                    // clock enable
output [width-1:0] rdata;   // read data

reg [width-1:0] memory [length-1:0];
reg [width-1:0] rdata;

//writing the always block
always@(posedge wclk)
begin
if(w_clken == 1'b1)
        memory[waddr] <= wdata;
else
        memory[waddr]  <= memory[waddr];
end
```

```
always@(posedge rclk)
begin
rdata <= memory[raddr];
end
endmodule
```

All this modules were connected as shown in block diagram. All the modules created in this lab is scalable.
Value of addr is passed to full_flag and empty_flag.

5.8 Calculation and analyzing the output waveforms

Calcualtions:

I pass the value for memory width and address, which is 16 and 5 respectively. So length is calculated , which is 2^5 =32 (00 to 1f)

Input clock : write clock
Input clock frequency 200 Mhz
Clock period is 1/200 Mhz = 5 ns

Output clock : read clock

Output clock frequency 40 Mhz

Clock period is 1/40 Mhz = 25 ns

Analyze:
In the test bench data is created using counter. Since data has to be written on the memory in synchronous with the write clock I incremented data with the period of 5ns.

1. **Write onto memory.**
   From the waveform it can be seen the data is being writen on the address starting from the address 00

Write address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E

Write data	00	00	00	01	02	03	04	05	06	07	08	09	0A	0B	0C

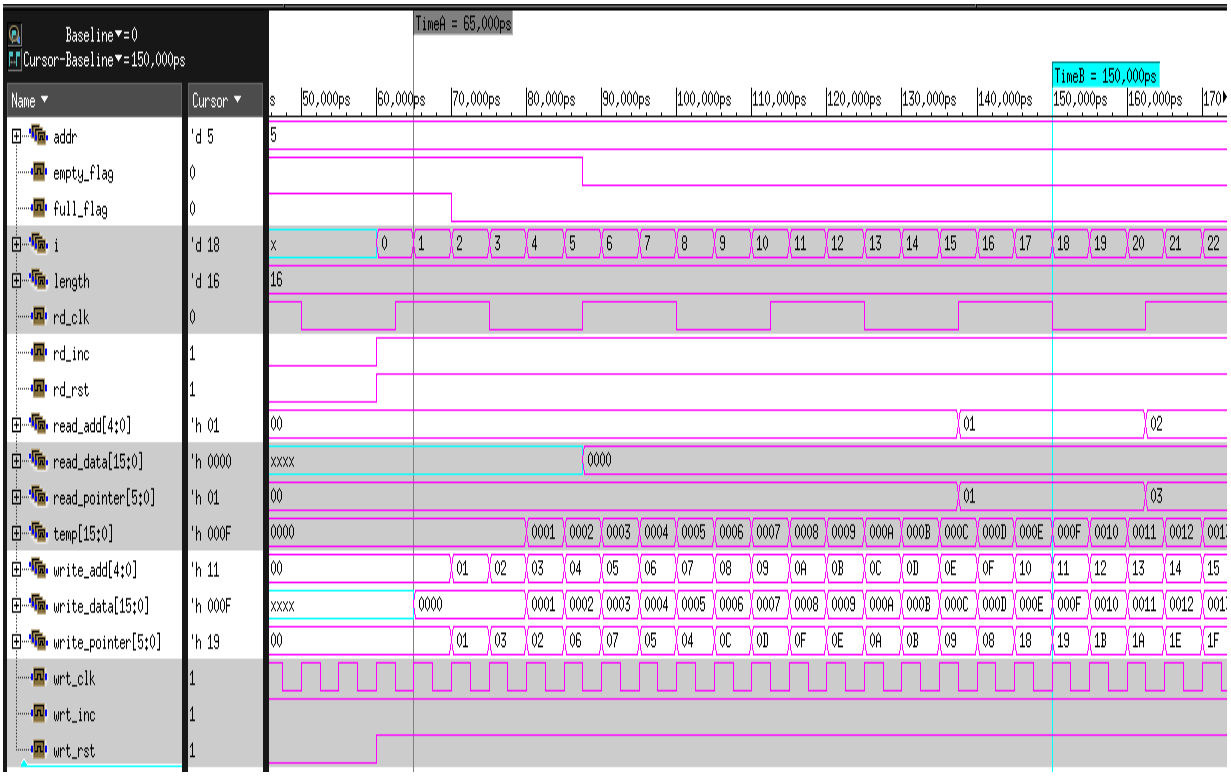


Figure 5.2 Write onto Memory

2. Read from the memory.

From the memory it can be seen that the data is being read from the memory location starting from the memoty location 00.

Read address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E
Read data	00	00	00	01	02	03	04	05	06	07	08	09	0A	0B	0C

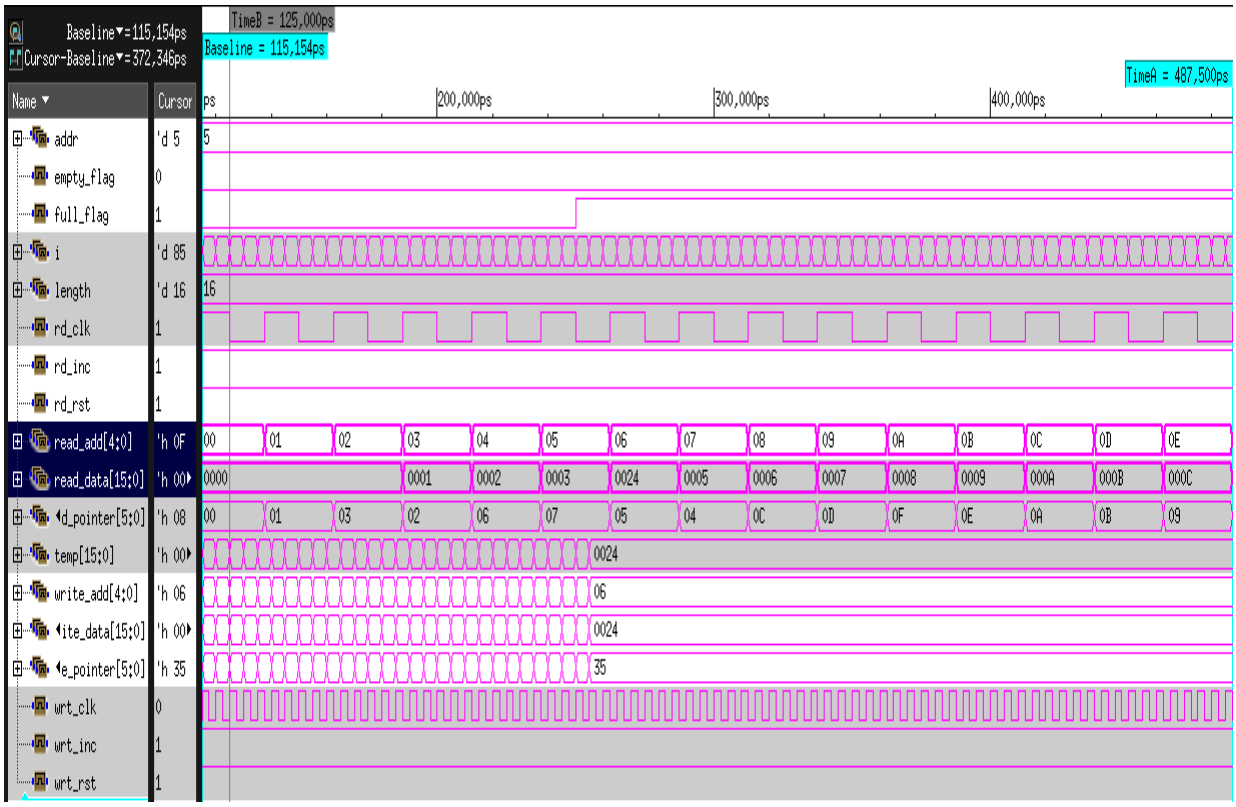


Figure 5.3 Read from the Memory

3. **Empty flag get clear when data is written onto the memory.**
   From the following waveform we can clearly see that initially value of signal “empty_flag” is 1. Once we start writing data onto memory,as soon as the posedge of “rd_clk” comes the “empty_flag” is cleared. Since the “empty_flag “ is cleared data is being read , which is 0000 at address 00 and so on.

Figure 5.4 Empty Flag

Empty flag is high as soon as the value for the read and the write pointer is equal. From the figure below we can see that , value for the read pointer and write pointer is 110101.

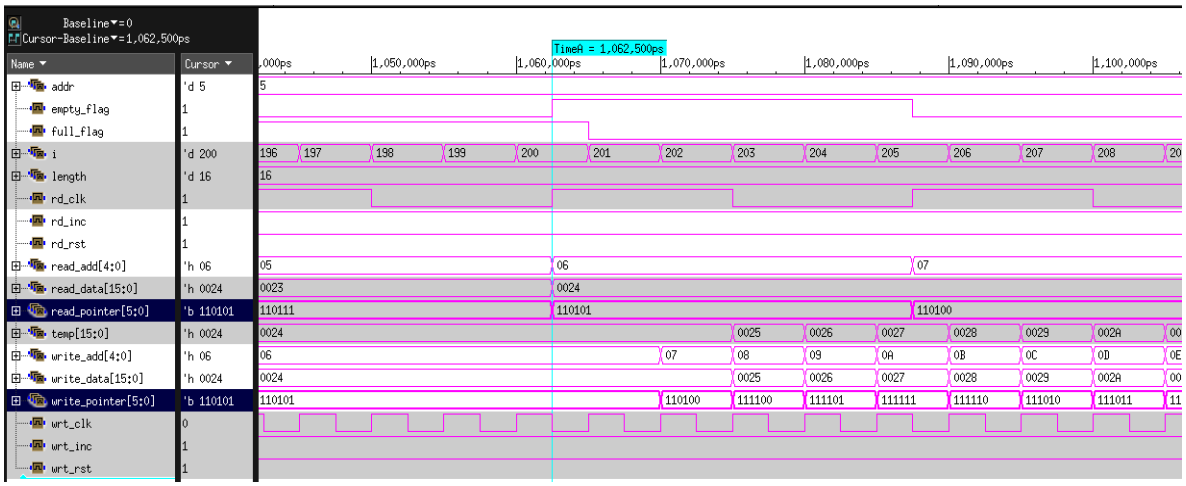


Figure 5.5 Empty Flag Clear

4. **Full flag concept**

The cocept I used for the full flag is that First two bits of the read and write pointer is not equal and if rest of the bits are equal then full_flag will be generated. At time 297 ns full_flag goes high and writing data onto memory stops with the latency of 2 clock. This does not effect the reading side .

Latest Data written in memory

Write_add	05	06	07
Write_data	002E	002F	0030

Latest data read from the memory

Read_add	05	06	07
Read_data	0008	0009	000A

By comparing this two I am sure that data being read is not the data written on 2nd time. So fifo is functioning properly even with the latency of 2.

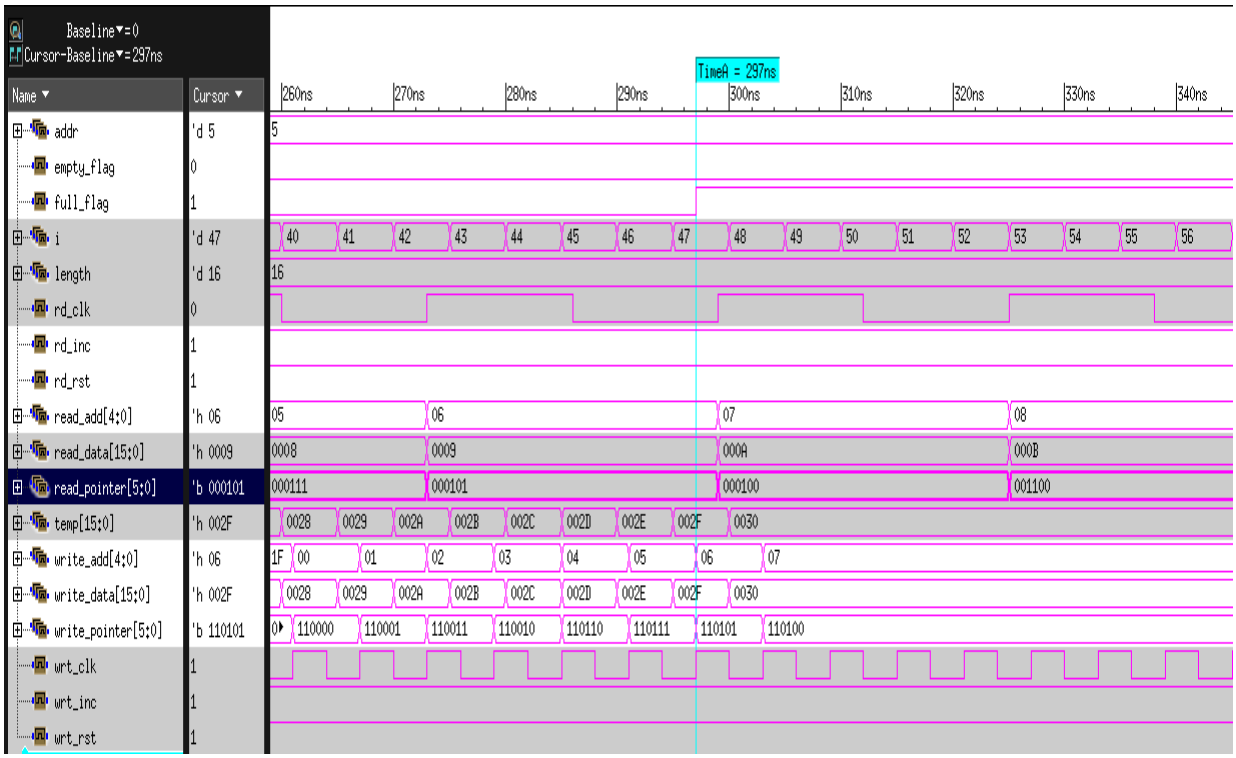


Figure 5.6 Full Flag high

Chapter 6: Preparing Design Files for Synthesis

(Reference: Naveen Kumar Project 2011 "Power Optimization in ASIC Design using Synopsys design ")

Designs (that is, design descriptions) are stored in design files. Design files must have unique names. If a design is hierarchical, each sub-design refers to another design file, which must also have a unique name. Note, however, that different design files can contain sub-designs with identical names.

6.1 Organizing the Design Data

Establishing and adhering to a method of organizing data are more important than the choosing a method. After the essential design data is placed under a consistent set of controls, meaningful data organization can be created. To simplify data exchanges and data searches, designers should adhere to this data organization system.

Hierarchical directory structure can be used to address data organization issues. Compile strategy will influence the directory structure. The following figures show directory structures based on the top-down compile strategy (Figure 6.1) and the bottom-up compile strategy (Figure 6.2).

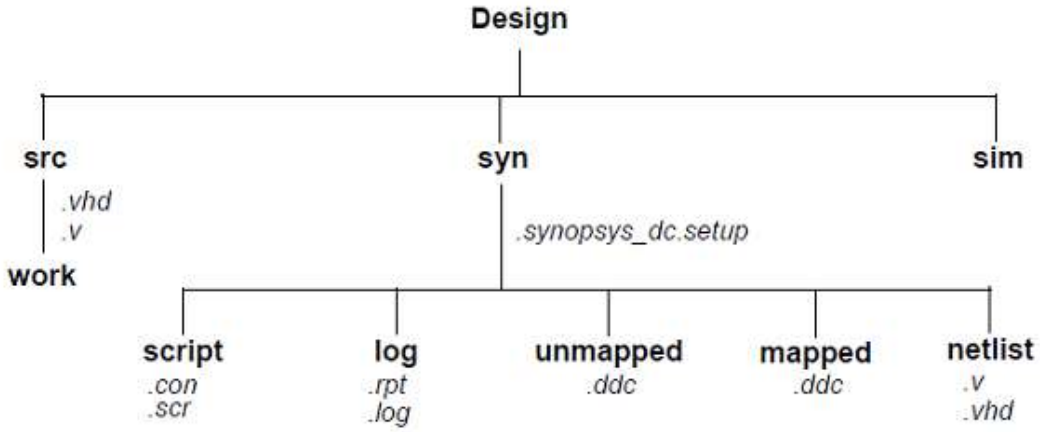


Figure 6.1 Top-Down Compile Directory Structure

6.2 Partitioning for Synthesis

Partitioning a design effectively can enhance the synthesis results, reduce compile time, and simplify the constraint and script files.Partitioning affects block size, and although Design Compiler has no inherent block size limit, proper care should be taken to control block size.

If the blocks are made too small, artificial boundaries can be created that restrict effective optimization. If very large blocks are created, compile runtimes can be lengthy. The following strategies to partition your design and improve optimization and runtimes:

- Partition for design reuse.
- Keep related combinational logic together.
- Register the block outputs.
- Partition by design goal.
- Partition by compile technique.
- Keep sharable resources together.
- Keep user-defined resources with the logic they drive.
- Isolate special functions, such as pads, clocks, boundary scans, and asynchronous logic.

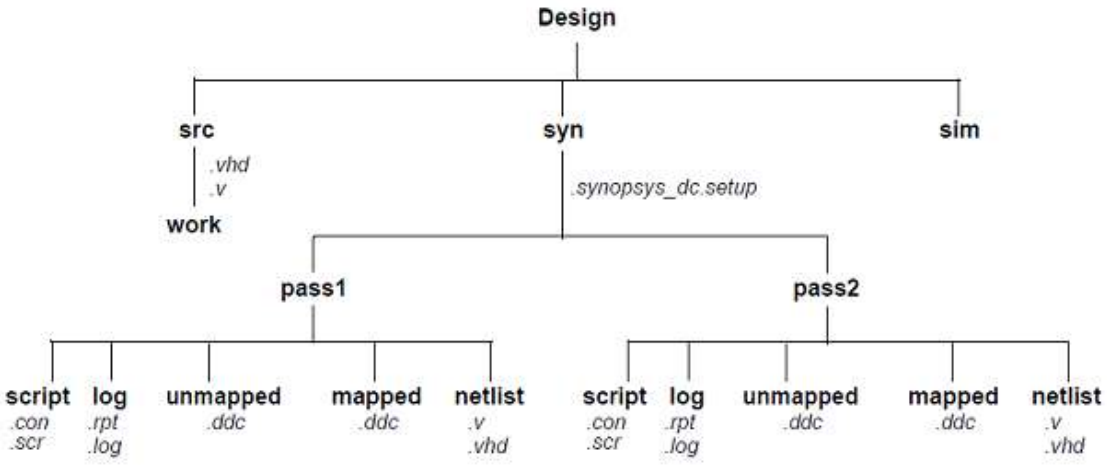


Figure 6.2 Bottom-Up Compile Directory Structure

Chapter 7: Basic Commands

(reference: Naveen Kumar Project 2011 "Power Optimization in ASIC Design using Synopsys design ")

This chapter lists the basic dc_shell commands for synthesis and provides a brief description for each command. The commands are grouped in the following sections: x Commands for Defining Design Rules

- Commands for Defining Design Environments
- Commands for Setting Design Constraints
- Commands for Analyzing and Resolving Design Problems

7.1 Commands for Defining Design Rules

The commands that define the design environment are:

- *set_max_capacitance*

  Sets a maximum capacitance for the nets attached to the specified ports or to all the nets in a design.
- *set_max _fanout*

  Sets the expected fan-out load value for output ports.
- *set_max_transition*

  Sets a maximum transition time for the nets attached to the specified ports or to all the nets in a design.
- x*set_min_capacitance*

  Sets a minimum capacitance for the nets attached to the specified ports or to all the nets in a design.

7.2 Commands for Defining Design Environments

The commands that define the design environment are:

- *set_drive*

  Sets the drive value of input or in-out ports. The set_drive command is superseded by the set_driving_cell command.
- *set_driving_cell*

  Sets attributes on input or in-out ports, specifying that a library cell or library pin drives the ports. This command associates a library pin with an input port so that delay calculators can accurately

model the drive capability of an external driver.

- *set_fanout_load*

  Defines the external fan-out load values on output ports.

- *set_load*

  Defines the external load values on input and output ports and nets.

- *set_operating_conditions*

  Defines the operating conditions for the current design.

- *set_wire_load_model*

  Sets the wire load model for the current design or for the specified ports. With this command, one can specify the wire load model to use for the external net connected to the output port.

7.3 Commands for Setting Design Constraints

The basic commands that set design constraints are:

- *create_clock*

  Creates a clock object and defines its waveform in the current design. x

  set_clock_latency, set_clock_uncertainty, set_clock_transition

  Sets clock attributes on clock objects or flip-flop clock pins.

- *set_input_delay*

  Sets input delay on pins or input ports relative to a clock signal.

- *set_max_area*

  Specifies the maximum area for the current design.

- *set_output_delay*

  Sets output delay on pins or output ports relative to a clock signal.

  The advanced commands that set design constraints are

- *group_path*

  Groups a set of paths or endpoints for cost function calculation. This command is used to create path groups, to add paths to existing groups, or to change the weight of existing groups.

- *set_false_path*

  Marks paths between specified points as false. This command eliminates the selected paths from timing analysis.

- *set_max_delay*

  Specifies a maximum delay target for selected paths in the current design.

- *set_min_delay*

Specifies a minimum delay target for selected paths in the current design.

7.4 Commands for Analyzing and Resolving Design Problems

The commands for analyzing and resolving design problems are:

- *all_connected*

  Lists all fanouts on a net.

- *all_registers*

  Lists sequential elements or pins in a design.

- *check_design*

  Checks the internal representation of the current design for consistency and issues error and warning messages as appropriate.

- *check_timing*

  Checks the timing attributes placed on the current design.

- *get_attribute*

  Reports the value of the specified attribute.

- *link*

  Locates the reference for each cell in the design.

- *report_area*

  Provides area information and statistics on the current design.

- *report_attribute*

  Lists the attributes and their values for the selected object. An object can be a cell, net, pin, port, instance, or design.

- *report_cell*

  Lists the cells in the current design and their cell attributes.

- *report_clock*

  Displays clock-related information on the current design.

- *report_constraint*

  Lists the constraints on the current design and their cost, weight, and weighted cost.

- *report_delay_calculation*

  Reports the details of a delay arc calculation.

- *report_design*

  Displays the operating conditions, wire load model and mode, timing ranges, internal input and output, and disabled timing arcs defined for the current design. x

report_hierarchy

Lists the children of the current design.

- *report_net*

  Displays net information for the design of the current instance, if set; otherwise, displays net information for the current design.

- *report_path_group*

  Lists all timing path groups in the current design.

- *report_port*

  Lists information about ports in the current design.

- *report_qor*

  Displays information about the quality of results and other statistics for the current design.

- *report_resources*

  Displays information about the resource implementation.

- *report_timing*

  Lists timing information for the current design.

- *report_timing_requirements*

  Lists timing path requirements and related information.

- *report_transitive_fanin*

  Lists the fan-in logic for selected pins, nets, or ports of the current instance. x

  report_transitive_fanout

  Lists the fan-out logic for selected pins, nets, or ports of the current instance.

7.5 Script file to synthesize the FIFO design top down

```
#analyzing and elaborating the low level design

analyze -f verilog full_flag.v
elaborate full_flag
check_design
```

```
analyze -f verilog empty_flag.v
elaborate empty_flag
check_design

analyze -f verilog write_pntr.v
elaborate write_pntr
check_design

analyze -f verilog read_pntr.v
elaborate read_pntr
check_design

analyze -f verilog memory.v
elaborate memory
check_design

analyze -f verilog aasd.v
elaborate aasd
check_design

analyze -f verilog fifo_top.v
elaborate fifo_top
check_design

#ungrouping the design hierarchy
ungroup -all -flatten

#generating write clock and read clock
create_clock wrt_clk -period 5
create_clock rd_clk -period 25

#setting false path for the signals crossing the clock domain
set_false_path -from [get_clocks wrt_clk] -to [get_clocks rd_clk]
set_false_path -from [get_clocks rd_clk] -to [get_clocks wrt_clk]

#to constraint the area
set_max_area 29790 -ignore_tns

#compiling the design
compile
check_design

#generating the SDF file
write_sdf fifo.sdf

#reporting the area and timing
report_area >> fifo_area_new_1.rpt
report_timing >> fifo_timing_new_1.rpt
```

7.5.1 Area report for top down synthesis.

```
****************************************
Report : area
Design : fifo_top
****************************************

Information: Updating design information... (UID-85)
Library(s Used:saed90nm_max
(File:/opt/ECE_Lib/SAED_EDK90nm/Digital_Standard_Cell_Library/synopsys/models/saed90nm_max_pg.db)
Number of ports:              62
Number of nets:             1726
Number of cells:            1684
Number of references:         20

Combinational area:       15767.195304
Noncombinational area:    14713.237717
Net Interconnect area:    2577.459163

Total cell area:          30480.433021
Total area:               33057.892184
```

7.5.2 Timing report for top down synthesis

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : fifo_top
****************************************

Operating Conditions: BEST   Library: saed90nm_max
```

Wire Load Model Mode: enclosed

Startpoint: rpt/read_address_reg[1]
(rising edge-triggered flip-flop clocked by rd_clk)
Endpoint: mem/rdata_reg[2]
(rising edge-triggered flip-flop clocked by rd_clk)
Path Group: rd_clk
Path Type: max

Des/Clust/Port	Wire Load Model	Library
fifo_top	35000	saed90nm_max

Point	Incr	Path
clock rd_clk (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
rpt/read_address_reg[1]/CLK (DFFARX1)	0.00	0.00 r
rpt/read_address_reg[1]/Q (DFFARX1)	0.59	0.59 f
U1744/Q (NBUFFX2)	0.21	0.80 f
U796/Q (NBUFFX2)	0.15	0.94 f
U1621/Q (MUX41X1)	0.08	1.03 r
U1725/Q (MUX41X1)	0.08	1.11 r
U1723/Q (MUX21X1)	0.06	1.17 r
mem/rdata_reg[2]/D (DFFX1)	0.00	1.17 r
data arrival time	1.17	
clock rd_clk (rise edge)	25.00	25.00
clock network delay (ideal)	0.00	25.00
mem/rdata_reg[2]/CLK (DFFX1)	0.00	25.00 r
library setup time	-0.03	24.97
data required time	24.97	
data required time	24.97	
data arrival time	-1.17	
slack (MET)	23.80	

Startpoint: wpt/write_address_reg[0]
(rising edge-triggered flip-flop clocked by wrt_clk)
Endpoint: mem/memory_reg[28][6]
(rising edge-triggered flip-flop clocked by wrt_clk)
Path Group: wrt_clk
Path Type: max

Des/Clust/Port	Wire Load Model	Library
fifo_top	35000	saed90nm_max

Point	Incr	Path
clock wrt_clk (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
wpt/write_address_reg[0]/CLK (DFFARX1)	0.00	0.00 r
wpt/write_address_reg[0]/Q (DFFARX1)	0.58	0.58 f
U1544/Q (NBUFFX2)	0.23	0.81 f
U760/Q (NBUFFX2)	0.15	0.96 f
U895/Q (MUX41X1)	0.11	1.07 f
U831/Q (MUX41X1)	0.07	1.13 f
U830/Q (MUX21X1)	0.05	1.18 f
U829/Q (AO22X1)	0.47	1.65 f
U768/Q (NBUFFX2)	0.14	1.80 f
U1251/Q (AO22X1)	0.07	1.87 f
mem/memory_reg[28][6]/D (DFFX1)	0.00	1.87 f
data arrival time	1.87	
clock wrt_clk (rise edge)	5.00	5.00
clock network delay (ideal)	0.00	5.00
mem/memory_reg[28][6]/CLK (DFFX1)	0.00	5.00 r
library setup time	-0.03	4.97
data required time	4.97	
data required time	4.97	
data arrival time	-1.87	

slack (MET)	3.10

7.6 This is the script file to synthesize the FIFO design bottom up

This is the script file to compile the top module using the Characterized lower level modules

```
#reading the lower level compiled files

read_file -f ddc full_flag_bot_up_II.ddc
read_file -f ddc empty_flag_bot_up_II.ddc
read_file -f ddc write_pntr_bot_up_II.ddc
read_file -f ddc read_pntr_bot_up_II.ddc
read_file -f ddc memory_bot_up_II.ddc
read_file -f ddc aasd_bot_up_II.ddc

analyze -f verilog fifo_top.v
elaborate fifo_top

check_design

create_clock wrt_clk -period 5
create_clock rd_clk -period 25

#setting false path for the signals crossing the clock domain
set_false_path -from [get_clocks wrt_clk] -to [get_clocks rd_clk]
set_false_path -from [get_clocks rd_clk] -to [get_clocks wrt_clk]
```

```
#compiling the full flag design

analyze -f verilog full_flag.v
elaborate full_flag
check_design
create_clock -period 5 -name sys_clock
compile
check_design
write -f ddc -o full_flag_bot_up.ddc
quit
```

```
#compiling the empty flag design

analyze -f verilog empty_flag.v
elaborate empty_flag
check_design
create_clock -period 10 -name sys_clock
compile
check_design
write -f ddc -o empty_flag_bot_up.ddc
quit
```

```
#compiling the write pointer design

analyze -f verilog write_pntr.v
elaborate write_pntr
check_design
```

```
create_clock clock -period 5
compile
check_design
write -f ddc -o write_pntr_bot_up.ddc
quit
```

```
#compiling the read pointer design

analyze -f verilog read_pntr.v
elaborate read_pntr
check_design
create_clock clock -period 10
compile
check_design
write -f ddc -o read_pntr_bot_up.ddc
quit
```

```
#compiling the memory design

analyze -f verilog memory.v
elaborate memory
check_design
create_clock wclk -period 5
create_clock rclk -period 10
compile
check_design
write -f ddc -o memory_bot_up.ddc
quit
```

```
#compiling the synchronizer design

analyze -f verilog aasd.v
elaborate aasd
check_design
create_clock clock -period 5
compile
check_design
write -f ddc -o aasd_bot_up.ddc
quit
```

7.6.1 Gates files created using compiling lower level module

FIFO _TOP_GATES

```
`timescale 1ns/1ns
module fifo_top_gates ( wrt_clk, wrt_rst, wrt_inc, full_flag, rd_clk, rd_rst, rd_inc,
     empty_flag, write_data, read_data, read_add, write_add, write_pointer,
     read_pointer );
 input [15:0] write_data;
 output [15:0] read_data;
 output [4:0] read_add;
 output [4:0] write_add;
 output [5:0] write_pointer;
 output [5:0] read_pointer;
 input wrt_clk, wrt_rst, wrt_inc, rd_clk, rd_rst, rd_inc;
```

```
 output full_flag, empty_flag;
 wire   \wq2_rpntr[0] , wrt_rst_syn, \mem/N68 , \mem/N67 , \mem/N66 ,
      \mem/N65 , \mem/N64 , \mem/N63 , \mem/N62 , \mem/N61 , \mem/N60 ,
      \mem/N59 , \mem/N58 , \mem/N57 , \mem/N56 , \mem/N55 , \mem/N54 ,
      \mem/N53 , \mem/N36 , \mem/N35 , \mem/N34 , \mem/N33 , \mem/N32 ,
      \mem/N31 , \mem/N30 , \mem/N29 , \mem/N28 , \mem/N27 , \mem/N26 ,
      \mem/N25 , \mem/N24 , \mem/N23 , \mem/N22 , \mem/N21 ,
      \mem/memory[0][0] , \mem/memory[0][1] , \mem/memory[0][2] ,
      \mem/memory[0][3] , \mem/memory[0][4] , \mem/memory[0][5] ,
      \mem/memory[0][6] , \mem/memory[0][7] , \mem/memory[0][8] ,
      \mem/memory[0][9] , \mem/memory[0][10] , \mem/memory[0][11] ,
```

EMPTY_FLAG_GATES

```
`timescale 1ns/1ns
module empty_flag_gates ( read_pointer, write_pointer, empty_flag );
 input [5:0] read_pointer;
 input [5:0] write_pointer;
 output empty_flag;
 wire   n3, n4, n5, n6, n7, n8, n9;

 NOR4X0 U3 ( .IN1(n3), .IN2(n4), .IN3(n5), .IN4(n6), .QN(empty_flag) );
 XOR2X1 U4 ( .IN1(write_pointer[4]), .IN2(read_pointer[4]), .Q(n6) );
 XOR2X1 U5 ( .IN1(write_pointer[1]), .IN2(read_pointer[1]), .Q(n5) );
 XOR2X1 U6 ( .IN1(write_pointer[0]), .IN2(read_pointer[0]), .Q(n4) );
 NAND3X0 U7 ( .IN1(n7), .IN2(n8), .IN3(n9), .QN(n3) );
 XNOR2X1 U8 ( .IN1(write_pointer[2]), .IN2(read_pointer[2]), .Q(n9) );
 XNOR2X1 U9 ( .IN1(write_pointer[3]), .IN2(read_pointer[3]), .Q(n8) );
 XNOR2X1 U10 ( .IN1(write_pointer[5]), .IN2(read_pointer[5]), .Q(n7) );
Endmodule
```

FULL_FLAG_GATES

```
`timescale 1ns/1ns
module full_flag_gates ( write_pointer, read_pointer, full_flag, clk_en );
 input [5:0] write_pointer;
 input [5:0] read_pointer;
 input clk_en;
 output full_flag;
 wire   n9, n10, n11, n12, n13, n14, n15, n16;

 NAND2X0 U10 ( .IN1(clk_en), .IN2(n9), .QN(full_flag) );
 NAND4X0 U11 ( .IN1(n10), .IN2(n11), .IN3(n12), .IN4(n13), .QN(n9) );
 NOR3X0 U12 ( .IN1(n14), .IN2(n15), .IN3(n16), .QN(n13) );
 XOR2X1 U13 ( .IN1(write_pointer[1]), .IN2(read_pointer[1]), .Q(n16) );
 XOR2X1 U14 ( .IN1(write_pointer[3]), .IN2(read_pointer[3]), .Q(n15) );
 XOR2X1 U15 ( .IN1(write_pointer[2]), .IN2(read_pointer[2]), .Q(n14) );
 XNOR2X1 U16 ( .IN1(write_pointer[0]), .IN2(read_pointer[0]), .Q(n12) );
 XOR2X1 U17 ( .IN1(write_pointer[4]), .IN2(read_pointer[4]), .Q(n11) );
 XOR2X1 U18 ( .IN1(write_pointer[5]), .IN2(read_pointer[5]), .Q(n10) );
Endmodule
```

EMPTY_FLAG_GATES

```
`timescale 1ns/1ns
module empty_flag_gates ( read_pointer, write_pointer, empty_flag );
  input [5:0] read_pointer;
  input [5:0] write_pointer;
  output empty_flag;
  wire   n3, n4, n5, n6, n7, n8, n9;

  NOR4X0 U3 ( .IN1(n3), .IN2(n4), .IN3(n5), .IN4(n6), .QN(empty_flag) );
  XOR2X1 U4 ( .IN1(write_pointer[4]), .IN2(read_pointer[4]), .Q(n6) );
  XOR2X1 U5 ( .IN1(write_pointer[1]), .IN2(read_pointer[1]), .Q(n5) );
  XOR2X1 U6 ( .IN1(write_pointer[0]), .IN2(read_pointer[0]), .Q(n4) );
  NAND3X0 U7 ( .IN1(n7), .IN2(n8), .IN3(n9), .QN(n3) );
  XNOR2X1 U8 ( .IN1(write_pointer[2]), .IN2(read_pointer[2]), .Q(n9) );
  XNOR2X1 U9 ( .IN1(write_pointer[3]), .IN2(read_pointer[3]), .Q(n8) );
  XNOR2X1 U10 ( .IN1(write_pointer[5]), .IN2(read_pointer[5]), .Q(n7) );
Endmodule
```

READ_PNTR_GATES

```
`timescale 1ns/1ns
module read_pntr_gates ( clock, reset, incrementer, flag, read_pointer, read_address
 );
  output [5:0] read_pointer;
  output [4:0] read_address;
  input clock, reset, incrementer, flag;
  wire   N2, N3, N4, N5, N6, n11, n14, n17, n20, n22, n24, n36, n37, n38, n39,
        n40, n41, n42, n43, n44, n45, n46, n47;
  wire   [5:0] b_next;

  DFFARX1 \b_next_reg[0]  ( .D(n24), .CLK(clock), .RSTB(reset), .Q(b_next[0]),
      .QN(n43) );
  DFFARX1 \b_next_reg[1]  ( .D(n22), .CLK(clock), .RSTB(reset), .Q(b_next[1]),
      .QN(n44) );
  DFFARX1 \b_next_reg[2]  ( .D(n20), .CLK(clock), .RSTB(reset), .Q(b_next[2]),
      .QN(n45) );
  DFFARX1 \b_next_reg[3]  ( .D(n17), .CLK(clock), .RSTB(reset), .Q(b_next[3]),
      .QN(n46) );
  DFFARX1 \b_next_reg[4]  ( .D(n14), .CLK(clock), .RSTB(reset), .Q(b_next[4]),
      .QN(n47) );
  DFFARX1 \b_next_reg[5]  ( .D(n11), .CLK(clock), .RSTB(reset), .Q(b_next[5]),
      .QN(n42) );
```

WRITE_PNTR_GATES

```
`timescale 1ns/1ns
module write_pntr_gates ( clock, reset, incrementer, write_flag, empty_flag, wclk_en,
      write_pointer, write_address );
  output [5:0] write_pointer;
```

```
 output [4:0] write_address;
 input clock, reset, incrementer, write_flag, empty_flag;
 output wclk_en;
 wire   N3, N4, N5, N6, N7, n16, n17, n19, n21, n23, n25, n27, n39, n40, n41,
      n42, n43, n44, n45, n46, n48, n49, n50, n51, n52, n53;
 wire   [5:0] bnext;

 DFFX1 wclk_en_reg ( .D(n16), .CLK(clock), .Q(wclk_en) );
 DFFARX1 \bnext_reg[0]  ( .D(n27), .CLK(clock), .RSTB(reset), .Q(bnext[0]),
     .QN(n49) );
 DFFARX1 \bnext_reg[1]  ( .D(n25), .CLK(clock), .RSTB(reset), .Q(bnext[1]),
     .QN(n50) );
 DFFARX1 \bnext_reg[2]  ( .D(n23), .CLK(clock), .RSTB(reset), .Q(bnext[2]),
     .QN(n51) );
 DFFARX1 \bnext_reg[3]  ( .D(n21), .CLK(clock), .RSTB(reset), .Q(bnext[3]),
     .QN(n52) );
 DFFARX1 \bnext_reg[4]  ( .D(n19), .CLK(clock), .RSTB(reset), .Q(bnext[4]),
     .QN(n53) );
```

MEMORY_GATES

```
`timescale 1ns/1ns
module memory_gates ( wdata, waddr, w_clken, wclk, raddr, rclk, rdata );
 input [15:0] wdata;
 input [4:0] waddr;
 input [4:0] raddr;
 output [15:0] rdata;
 input w_clken, wclk, rclk;
 wire   N10, N11, N12, N13, N14, N15, N16, N17, N18, N19, \memory[31][15] ,
      \memory[31][14] , \memory[31][13] , \memory[31][12] ,
      \memory[31][11] , \memory[31][10] , \memory[31][9] , \memory[31][8] ,
      \memory[31][7] , \memory[31][6] , \memory[31][5] , \memory[31][4] ,
      \memory[31][3] , \memory[31][2] , \memory[31][1] , \memory[31][0] ,
      \memory[30][15] , \memory[30][14] , \memory[30][13] ,
      \memory[30][12] , \memory[30][11] , \memory[30][10] , \memory[30][9] ,
      \memory[30][8] , \memory[30][7] , \memory[30][6] , \memory[30][5] ,
      \memory[30][4] , \memory[30][3] , \memory[30][2] , \memory[30][1] ,
      \memory[30][0] , \memory[29][15] , \memory[29][14] , \memory[29][13] ,
      \memory[29][12] , \memory[29][11] , \memory[29][10] , \memory[29][9] ,
      \memory[29][8] , \memory[29][7] , \memory[29][6] , \memory[29][5] ,
      \memory[29][4] , \memory[29][3] , \memory[29][2] , \memory[29][1] ,
      \memory[29][0] , \memory[28][15] , \memory[28][14] , \memory[28][13] ,
      \memory[28][12] , \memory[28][11] , \memory[28][10] , \memory[28][9] ,
      \memory[28][8] , \memory[28][7] , \memory[28][6] , \memory[28][5] ,
```

AASD_GATES

```
`timescale 1ns/1ns
module aasd_gates ( in, clock, reset, aasd );
 input [15:0] in;
 output [15:0] aasd;
 input clock, reset;
```

```
 DFFARX1 \aasd_reg[15]  ( .D(in[15]), .CLK(clock), .RSTB(reset), .Q(aasd[15])
      );
 DFFARX1 \aasd_reg[14]  ( .D(in[14]), .CLK(clock), .RSTB(reset), .Q(aasd[14])
      );
 DFFARX1 \aasd_reg[13]  ( .D(in[13]), .CLK(clock), .RSTB(reset), .Q(aasd[13])
      );
 DFFARX1 \aasd_reg[12]  ( .D(in[12]), .CLK(clock), .RSTB(reset), .Q(aasd[12])
      );
 DFFARX1 \aasd_reg[11]  ( .D(in[11]), .CLK(clock), .RSTB(reset), .Q(aasd[11])
      );
 DFFARX1 \aasd_reg[10]  ( .D(in[10]), .CLK(clock), .RSTB(reset), .Q(aasd[10])
      );
 DFFARX1 \aasd_reg[9]  ( .D(in[9]), .CLK(clock), .RSTB(reset), .Q(aasd[9]) );
 DFFARX1 \aasd_reg[8]  ( .D(in[8]), .CLK(clock), .RSTB(reset), .Q(aasd[8]) );
 DFFARX1 \aasd_reg[7]  ( .D(in[7]), .CLK(clock), .RSTB(reset), .Q(aasd[7]) );
 DFFARX1 \aasd_reg[6]  ( .D(in[6]), .CLK(clock), .RSTB(reset), .Q(aasd[6]) );
```

7.6.2 Area report for top level module

****

Report : area

Design : fifo_top

****

Library(s) Used:

gtech (File: /usr/synopsys/A-2007.12-SP5/libraries/syn/gtech.db)

saed90nm_max (File: /opt/ECE_Lib/syn_lib_2.2/DATABASE_SAED90NM_08_September_2008/synopsys/models/saed90nm_max.db)

Number of ports: 62

Number of nets: 77

Number of cells: 9

Number of references: 7

Combinational area: 7748.808294

Noncombinational area: 0.000000

Net Interconnect area: undefined (No wire load specified)

Total cell area: 7748.808294

Total area: undefined

7.6.3 Timing report for top level module

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : fifo_top
****************************************

 # A fanout number of 1000 was used for high fanout net computations.

Operating Conditions: WORST    Library: saed90nm_max
Wire Load Model Mode: top

  Startpoint: rpt/read_address_reg[0]
              (rising edge-triggered flip-flop clocked by rd_clk)
  Endpoint: mem/rdata_reg[0]
            (rising edge-triggered flip-flop clocked by rd_clk)
  Path Group: rd_clk
  Path Type: max



  Point                                                   Incr       Path
  ---------------------------------------------------------------------------
  clock rd_clk (rise edge)                                0.00       0.00
  clock network delay (ideal)                             0.00       0.00
  rpt/read_address_reg[0]/clocked_on (**SEQGEN**)         0.00       0.00 r
  rpt/read_address_reg[0]/Q (**SEQGEN**)                  0.00       0.00 f
  rpt/read_address[0] (read_pntr_size6)                   0.00       0.00 f
  mem/raddr[0] (memory_width16_addr5)                     0.00       0.00 f
  mem/B_7/Z (GTECH_BUF)                                   0.00       0.00 f
  mem/C1243/S0 (fifo_top_MUX_OP_32_5_16)                  0.00       0.00 f
  alt42/U170/Q (MUX41X1)                                  0.63       0.63 f
  alt42/U167/Q (MUX41X1)                                  0.60       1.23 f
  alt42/U166/Q (MUX21X1)                                  0.29       1.52 f
```

mem/C1243/Z_15 (fifo_top_MUX_OP_32_5_16)		0.00	1.52 f
mem/rdata_reg[0]/next_state (**SEQGEN**)		0.00	1.52 f
data arrival time	1.52		
clock rd_clk (rise edge)		25.00	25.00
clock network delay (ideal)		0.00	25.00
mem/rdata_reg[0]/clocked_on (**SEQGEN**)		0.00	25.00 r
library setup time		0.00	25.00
data required time	25.00		
data required time	25.00		
data arrival time	-1.52		
slack (MET)	23.48		

Startpoint: wpt/write_address_reg[0]
(rising edge-triggered flip-flop clocked by wrt_clk)
Endpoint: mem/memory_reg[0][0]
(rising edge-triggered flip-flop clocked by wrt_clk)
Path Group: wrt_clk
Path Type: max

Point	Incr	Path
clock wrt_clk (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
wpt/write_address_reg[0]/clocked_on (**SEQGEN**)	0.00	0.00 r
wpt/write_address_reg[0]/Q (**SEQGEN**)	0.00	0.00 f
wpt/write_address[0] (write_pntr_size6)	0.00	0.00 f
mem/waddr[0] (memory_width16_addr5)	0.00	0.00 f
mem/B_2/Z (GTECH_BUF)	0.00	0.00 f
mem/C1242/S0 (fifo_top_MUX_OP_32_5_16_1)	0.00	0.00 f
alt45/U170/Q (MUX41X1)	0.63	0.63 f
alt45/U167/Q (MUX41X1)	0.60	1.23 f
alt45/U166/Q (MUX21X1)	0.29	1.52 f
mem/C1242/Z_15 (fifo_top_MUX_OP_32_5_16_1)	0.00	1.52 f

mem/C1241/Z_0 (*SELECT_OP_2.16_2.1_16)		0.00	1.52 f
mem/memory_reg[0][0]/next_state (**SEQGEN**)		0.00	1.52 f
data arrival time	1.52		
clock wrt_clk (rise edge)		5.00	5.00
clock network delay (ideal)		0.00	5.00
mem/memory_reg[0][0]/clocked_on (**SEQGEN**)		0.00	5.00 r
library setup time		0.00	5.00
data required time	5.00		
---			
data required time	5.00		
data arrival time	-1.52		
---			
slack (MET)	3.48		

Observation from area report of Top-down and Bottom-up synthesis:

Bottom-up scan insertion has several advantages, including:

• Identifying scan design rule violations early in the design flow;

• Reducing the risks of time-consuming, top-level reoptimization steps;

• Facilitating parallel development and sign-off of scan modules.

Top-down scan insertion has the virtue of simplicity, but has implications for large multi-million gate designs due to the capacity and run-time limitations of synthesis tools. However, with designs reaching millions of gates and sub-modules also increasing in relative size, synthesis tools have reached their capacity limit for top-level scan synthesis, even with the bottom-up approach. Hence, some level of abstraction is needed at the sub-module level, and integration of these models at the top-level enhances capacity as well as performance for large designs. Synopsys has introduced the hierarchical scan-synthesis methodology with test models, which capitalizes on the proposed IEEE 1450.6 Core Test Language (CTL) standard to perform scan synthesis for the next generation of complex designs and SoCs

REFERENCES

[1] Michael John Sebastian Smith [June 1997], "Application-Specific Integrated Circuits", Addison-Wesley Publishing Company, VLSI Design Series. [April 2011]

[2] http://users.ece.gatech.edu/~hamblen/DE2/DE2%20Reference%20Manual.pdf [Jan 2011]

[3] Himanshu Bhatnagar ~Advanced ASIC chip Synthesis~ Using Synopsys Design Compiler, Physical Compiler and Primetime. [March 2011]

[4] http://en.wikipedia.org/wiki/Application-specific_integrated_circuit [Feb 2011]

[5] http://www.tutorial-reports.com/hardware/asic/overview.php [Feb 2011]

[6] http://www.utdallas.edu/~zhoud/Lecture 1 .pdf [April 2011]

[7] http://iroi.seu.edu.cn/books/asics/Book2/CH01/CH0 1.1 .htm [Oct 2011]

[8] SYNOPSYS Design Compiler User Guide [December 2011]

[9] http://www.csun.edu/~glaw/ee420.htm [Jan 2012]

[10] http://www.csun.edu/~rmehler/ee527.htm [March 2011]

[11] http://www.synopsys.com/Tools/Implementation/RTLSynthesis/CapsuleModule/hss_bkgrd.pdf

[12] Naveen Kumar Project 2011 "POWER OPTIMIZATION IN ASIC DESIGN USING SYNOPSYS DESIGN COMPILER" [March 2012]