Digital Systems Laboratory Manual

Vineesh V.S.
Piyush Soni
Arjun Vishnoria
Raunak Gupta
Soumik Sarkar
Avirup Mullick
Madhumita Date
Madhav Desai
Department of Electrical Engineering
IIT-Bombay, Mumbai

February 22, 2016

Chapter 1

Overview

This manual has been designed to guide a student through a laboratory course in Digital Systems Design. It may be viewed as a series of experiments which are of two types:

- Familiarization with tools and instruments: students should use these to familiarize themselves with the important technology to be used in the experiments.
- Experiments using the tools and instruments: these experiments are designed to take a student through different aspects of digital design: from CMOS inverter characterization up to complex Register-Transfer-Level (RTL) descriptions.

The manual is to be followed in sequence.

1. In Chapter 2, the student is introduced to the digital storage oscilloscope.
2. In Chapter 3, the student characterizes a CMOS inverter to understand its DC characteristics as well as its transient characteristics.
3. In Chapter 4, the student is introduced to the Krypton CPLD board which will be used in subsequent experiments.
4. In Chapter 5, the student implements a simple combinational circuit on the Krypton CPLD board and verifies it using switches and LED's on the board.
5. In Chapter 6, the student is introduced to the use of a scan-chain based test scheme.

6. In Chapter 7, the student implements a more complex combinational circuit on the Krypton CPLD board and verifies the functionality using the scan-chain based tester.

7. In Chapter 8, the student implements a finite state machine on the Krypton CPLD board and verifies the functionality using the scan-chain based tester.

8. In Chapter 9, the student implements a Register-transfer-level (RTL) description on the Krypton CPLD board and verifies the functionality using the scan-chain based tester.

9. In Chapter 10, the student implements a RTL circuit with SRAM on the Krypton CPLD board and verifies the functionality using the scan-chain based tester.

10. In Chapter 11, the student is introduced to the use of a logic analyzer.

Some of the advanced experiments in RTL may need two weeks to complete. Additional experiments and modifications to suggested experiments are of course possible, depending on the instructor.

The manual was prepared with contributions from a group of students and staff at the Wadhwani Electronics Laboratory in the Department of Electrical Engineering at IIT Bombay.

Chapter 2

Familiarization: The Digital Storage Oscilloscope (DSO)

Oscilloscopes are electronic instruments used to observe time-varying electrical signals. Plots of signal voltage values can be displayed as a function of time. Two types of oscilloscopes are usually used:

- Cathode-ray oscilloscope (CRO): A CRO uses analog circuits to sense and display information on a screen. Typically, it does not have the ability to store information about captured waveforms.
- Digital storage oscilloscope (DSO): A DSO typically digitizes and stores captured signals in an internal memory. The stored signals can then be analyzed and displayed on a screen. This offers great flexibility in analysis of the captured waveforms.

2.1 The Tektronix TDS 200 series Digital Storage Oscilloscope

We will describe the Tektronix TDS 200 series DSO[1]. A representation of its front panel is shown in Figure 2.1.

This DSO has the following features:

- Handle signals of frequency upto 60 MHz
- Can display two signals simultaneously

[1]Some other DSOs available in the lab are *TDS1002*, *TDS1002B* and *GDS-1072A-U-GW INSTEK*

Figure 2.1: TDS 210/220 two channel digital real-time oscilloscope

- Cursors with readout
- Autoset for quick setup

2.1.1 First look: verify that the instrument is working correctly

Let's do a quick functional check to verify that instrument is working correctly. See Figure 2.1 to identify the connection points on the DSO front-panel.

- Turn on the device and wait until the display shows that all the self tests passed.
- Take the signal probe (Eg: TDS2200). This probe has two attenuation settings, 1X and 10X. Set the attenuation button at the tip of the probe to 1X.
- Now, attach the probe tip to the *PROBE COMP* 5V connector and the probe reference lead to the *PROBE COMP* ground connector.
- Plug the probe into *Channel 1* on the oscilloscope and push the *AUTOSET*[2] button. On the display you should see a square wave of approximately 5V peak-to-peak at 1KHz, as shown in Figure 2.2. This

[2] ① in Figure 2.1

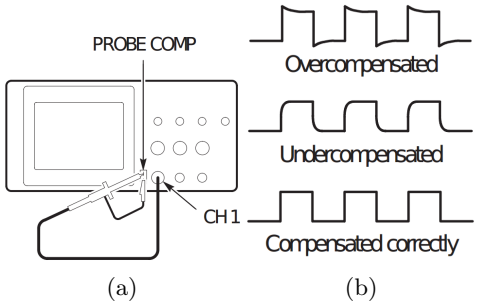

Figure 2.2: Testing the probe

ensures proper probe compensation[3]). Inform the lab staff if you observe something very different.

2.2 Capture a signal using the DSO

We will construct a test circuit and capture its waveforms using the DSO. The test circuit is shown in Figure 2.3.

Connect the output of our circuit (node 2) to *Channel 1* of the DSO as shown in Figure 2.3[4]. Are you able to see a stable waveform? If not you may need to use a triggering mechanism to capture the waveform.

2.3 Use triggering to observe stable waveforms

If the signal being observed is not stable[5], we can use a trigger to tell the oscilloscope the point from which the display of information should start. Note that the signal must be approximately periodic. There are three types of triggering.

- *Auto*: In this mode, DSO will display stable waveform in the presence of a valid trigger and unstable waveforms when no valid trigger is given
- *Normal*: DSO will display stable waveforms in the presence of a valid trigger and will retain the previous stable waveform in the presence of invalid trigger

[3]Probe compensation is a method to match the probe impedence to the input channel of the DSO

[4]The ground wire of the probe has to be connected to node 3

[5]Either moving waveforms or multiple waveforms displayed on screen

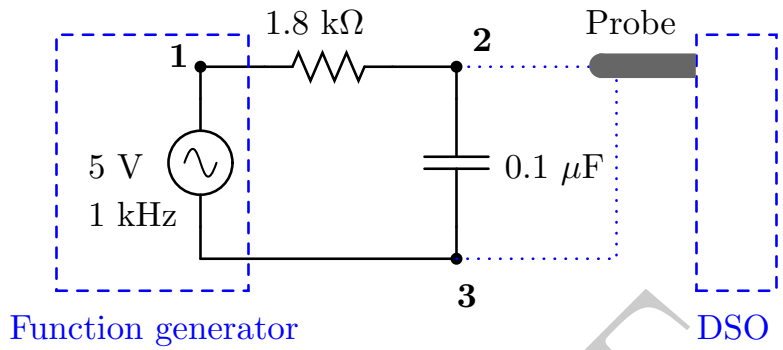

Figure 2.3: Test circuit with nodes for observing the signals

- *Single*: In this mode, waveform is acquired each time when the *RUN/STOP*[6] button is pressed and a trigger condition is detected

Use the following steps to obtain a stable waveform.

- Push the *MENU* button[7] in the *TRIGGER* section
- Select *Auto* in the *Mode* section[8]
- Select *CH1* in the *Source* section since we have connected the output of the circuit to channel 1
- Adjust the *LEVEL* knob[9] and keep the trigger level somewhere between the peaks of the input waveform. You will be able to see a small arrow on the right side moving up and down with your *LEVEL* knob
- You should be able to see a stable waveform on the display. Try out *Normal* and *Single* triggering modes by varying the trigger level and frequency of the input sine waveform (in function generator)

Type of triggering and various sources used for triggering are discussed in the Appendix

[6] ② in Figure 2.1
[7] ③ in Figure 2.1
[8] Mode button can be found from the set of buttons shown at the right side of the DSO panel. Labelled ⑫ figure 2.1
[9] ④ in Figure 2.1

2.4 Measure signal parameters

To measure signal frequency, period peak-to-peak amplitude etc. of the output waveform follow the steps given below

- Push the *AUTOSET* button footnoteIt automatically sets the trigger levels and displays the waveforms
- Push the *MEASURE* button[10] to go to the Measure menu
- Push the top menu box button to select *Source*
- Select *CH1* for the first three measurements
- Push the top menu box button to select *Type*
- Push the first *CH1* menu box button to select *Freq*
- Push the second *CH1* menu box button to select *Period*
- Push the third *CH1* menu box button to select *Pk-Pk*

2.5 Use cursors to take measurements

Using this one can vary the position of two cursor lines, get the voltage and time values corresponding to the position of the cursors, difference between them etc.

- Push the *CURSOR* button to go to the Cursor menu
- Push the top menu box button and select *Type* as *Voltage*[11]
- Set the *Source* option to *CH1*
- Vary the position of the two horizontal cursor lines using the *CURSOR 1* and *CURSOR 2* knobes
- Observe the voltage values corresponding to cursors under *Cursor 1* and *Cursor 2* the difference between them under *Delta*
- Push the top menu box button and select *Type* as *Time*
- Vary the cursors using the previous knobs and observe the time values

[10] ⑤ in Figure 2.1

[11] Two horizontal lines will appear if you set *Type* as *Voltage* and two vertical lines will show up if you set it as *Time*

2.6 Display two waveforms

There are two channels in the DSO which can be controlled independently

- Take another probe and plug it to the *Channel 2* on the oscilloscope
- Connect its tip to the node 1 in the example circuit
- Push *AUTOSET* button
- Push *CH 1 MENU*[12]
- Push the *Coupling* button on the right side of the display and see the various coupling options
- Push *Invert* button to invert polarity of the signal
- Push *CH 2 MENU*[13] and repeat the above two steps

You can also get a plot of one signal versus the other (its called XY mode). Follow the steps below to get a plot with channel-1 voltage on the X-axis and channel-2 voltage on the Y-axis

- Push *CH 1 MENU* and use *Coupling* button to change the coupling to *Ground*
- Use the *POSITION*[14] knob of channel-1 to bring the ground line of channel-1 to the center of the display
- Change the coupling back to *AC*
- Push *CH 2 MENU* and use *Coupling* button to change the coupling to *Ground*
- Use the *POSITION* knob of channel-2 to bring the ground line of channel-2 to the center of the display
- Change the coupling back to *AC*
- Push the *DISPLAY* button[15]

[12] ⑦ in Figure 2.1
[13] ⑧ in Figure 2.1
[14] ⑨ in Figure 2.1
[15] ⑩ in Figure 2.1

- Push the *FORMAT* button and select *XY* format
- You will see a ellipse shaped plot on the display
- Change the amplitude and frequency of the input sine wave. What do you observe?

Change the *Format* back to *XT* when you are done

2.7 Perform mathematical operations

DSO has the capability to perform some basic mathematical operations with the input signals.

- Push *MATH MENU* button[16]
- Push the *Operation* button to change the operation to +, - and *FFT*

2.8 Appendix

2.8.1 More about triggering

- *Trigger sources*: The source for triggering can be AC line, External sources or one of the input oscilloscope channels
- *Type of triggering*: Two types of triggering exists - edge triggering and video triggering. In edge triggering, an edge occurs when the trigger input passes through a specified voltage level in the specified direction. Video triggering is used on fields or lines of standard video signals

2.8.2 Data acquisition

On acquiring analog data, the oscilloscope converts it into a digital form. The 3 different acquisition modes include: Sample, Peak Detect and Average.

- *Sample*: The signal is sampled at equal intervals to reconstruct the waveform and is more accurate. But sampling will not include the narrow pulses in between the sampling intervals

[16] (11) in Figure 2.1

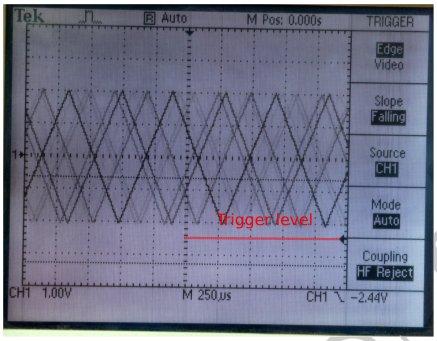

Figure 2.4: Untriggered waveform

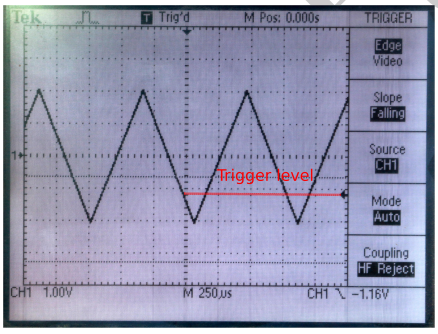

Figure 2.5: Displayed waveform with trigger level set below zero

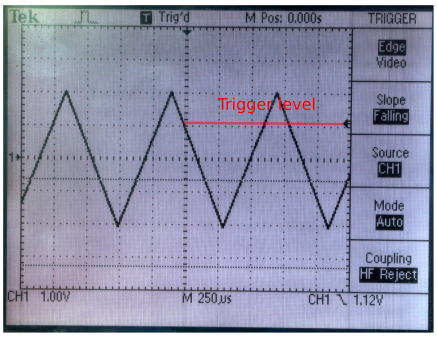

Figure 2.6: Dispayed waveform with trigger level set above zero

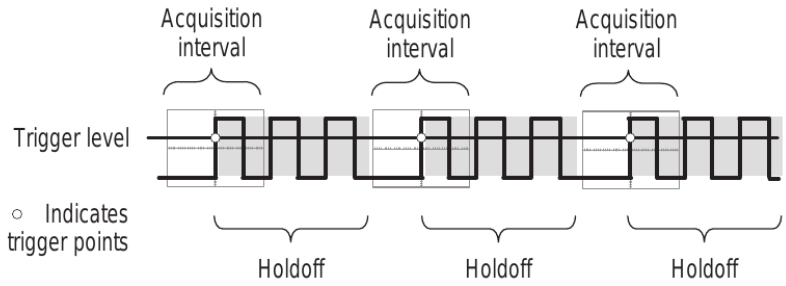

Figure 2.7: Holdoff- triggering not done

- *Peak Detect*: Detects the peaks over a specified interval and these are used to display the waveform. Detecting the peaks will enable the detection of narrow pulses
- *Average*: In this mode the average of several samples are acquired and displayed.

2.8.3 Hold off

Hold-off time is the period that follows each acquisition. Triggers are not recognized during the hold-off time. Sometimes for complex signals like a digital pulse train, it might be required to use hold-off. Ref Fig 2.7 .

Bibliography

[1] http://mmrc.caltech.edu/Oscilliscope/TDS210%20Digital%20Os%20User%20Manual.pdf (Accessed: 03-09-2015)

[2] www.tek.com/datasheet/tds200-series (Accessed: 04-09-2015)

[3] Photo credits: Anil R. Gawai, WEL, IIT Bombay

Chapter 3

Experiment: Characterization of a CMOS Inverter

The CMOS inverter is the simplest static complementary CMOS logic gate. It is also representative of all static complementary logic gates, in the sense that the qualitative behaviour of an arbitrary static complementary gate is captured by that of the inverter.

In this experiment we will characterize a CMOS inverter. To characterize a CMOS inverter, we will do a DC (or steady state behaviour) characterization by measuring its transfer characteristics and output characteristics. We will also do an AC (or transient behaviour) characterization and measure the inverter delay and its dependence on the power supply, and will observe the currents drawn from the power supply.

3.1 Components required and pre-requisites

- Familiarization with DSO.
- IC $MM74C04$ - 4
- Breadboard.
- Decoupling capacitors -0.1 μF.
- 1 Ω Resistor for switching current.
- 1.2 KΩ Resistor for output characteristics.

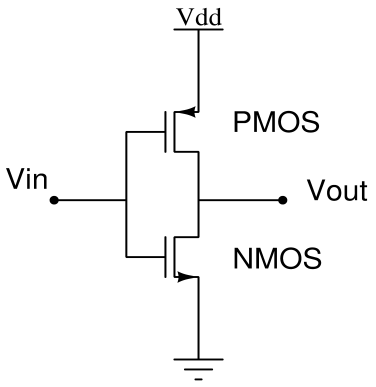

Figure 3.1: CMOS inverter

Condition	**PMOS**	**NMOS**
$0 \leq V_{in} < V_{tn}$	Linear	Cutoff
$V_{tn} \leq V_{in} < V_{DD}/2$	Linear	Saturation
$V_{in} = V_{DD}/2$	Saturation	Saturation
$V_{DD}/2 < V_{in} \leq V_{DD} - \|V_{tp}\|$	Saturation	Linear
$V_{in} \geq V_{DD} - \|V_{tp}\|$	Cutoff	Linear

Table 3.1: Different regions of operation of the MOSFETs in the inverter

- 20 KΩ potentiometer for output characteristics.
- Opamp as buffer - TL072.
- Connecting wires, voltmeter, Ammeter.

3.2 DC Transfer Characteristics

The physical structure of a CMOS inverter consists of a PMOS and NMOS in series as shown in Figure 3.1. The drains and gates of the PMOS and NMOS are tied together, and the sources of the PMOS and NMOS are connected to V_{DD} and *gnd* respectively. When the input voltage changes from 0 V to V_{DD}, the complementary NMOS and PMOS devices are forced to operate in different regions as shown in Table 3.1. In this table V_{tn} and V_{tp} are the threshold voltages of the NMOS and PMOS transistors respectively. By convention, $V_{tp} < 0$. Sometimes we will use V_t to refer to either V_{tn} or $-V_{tp}$ depending on the context.

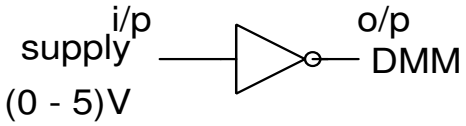

Figure 3.2: Test circuit (without power supply)

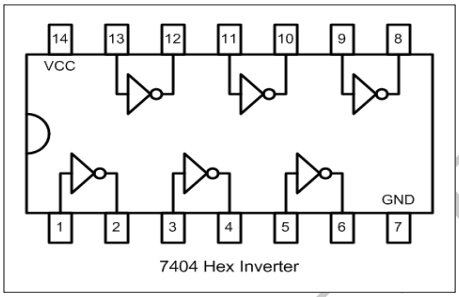

Figure 3.3: Pin diagram of the MM74C04

The inverter transfer characteristic is the plot of the output voltage as a function of the input voltage. As we vary the input voltage from $0V$ to V_{DD}, the output voltage will change from V_{DD} to $0V$. The point in the transfer characteristic when the input and output voltages are equal is called the *switching point* of the inverter.

3.2.1 Experiment setup

Connect pin 7 and pin 14 of the MM74C04 to V_SS (ground, that is, $0V$) and $V_{DD} = 5V$ respectively. Use one of the inverters in the IC and vary the input voltage from 0 to 5V (see Figure 3.2) and note down the output voltage. Plot the inverter transfer characteristics. Take more readings in the transition region.

From the switching point, we can estimate the β ratio of the two tranistors as follows. Since, at the switching point, the same current flows through PMOS and NMOS:

$$I_n = I_p$$

$$I_n = \frac{\mu_n C_{ox} W (V_{in} - V_t)^2}{2L} \tag{3.1}$$

$$I_p = \frac{\mu_p C_{ox} W (V_{in} - V_{DD} - V_t)^2}{2L} \tag{3.2}$$

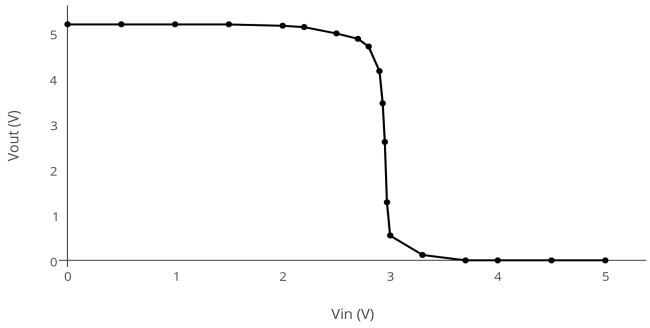

Figure 3.4: Form of the transfer characteristics

where V_t is negative for PMOS. After solving these equations, we get switching point V_{sw} as:

$$V_{sw} = \frac{\sqrt{\beta_p}(V_{DD} - V_t) + \sqrt{\beta_n}V_t}{\sqrt{\beta_p} + \sqrt{\beta_n}} \tag{3.3}$$

$$\text{where, } V_t = |V_{tp}| = V_{tn} \text{ and } \beta = \mu C_{ox} W/L$$

From this and your measurements, you can estimate the ratio β_P/β_N.

3.3 Output Characteristics

The output characteristics give us information about the current sourcing/sinking capability of the inverter. We find how the output voltage varies with the current drawn from or sunk into the inverter. Note that when the output of the inverter is high, the output voltage will fall as the current drawn from the inverter increases. When the output of the inverter is low, the output voltage will rise as the current sunk by the inverter increases.

3.3.1 Experimental setup

The experimental setup for the two cases (output high and output low) is shown in Figure 3.5 and Figure 3.6 respectively.

A 1.2 KΩ resistor is used just to avoid the short circuit between V_{DD} and V_{SS} when the potentiometer resistance is zero. The values of resistances

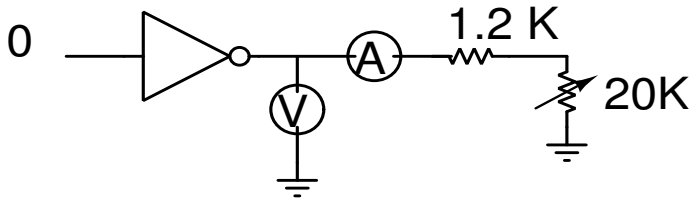

Figure 3.5: Output characteristics when output is high (input = $0V$, Supply voltage = $5V$)

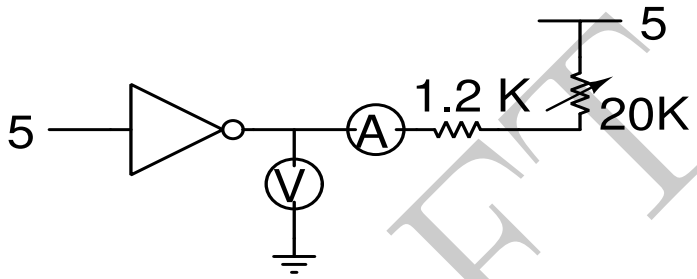

Figure 3.6: Output characteristics when output is low (input = V_{DD}, Supply voltage = $5V$)

are chosen depending on I_{OL} and I_{OH} values from datasheet, which is 1.75 mA for $MM74C04$.

Connect pin 7 and pin 14 of the MM74C04 to V_{SS} and V_{DD} respectively, as you did earlier. Use one of the inverters as shown in Figure 3.5 and vary the 20 KΩ potentiometer in steps while ensuring that the magnitude of the current $|I_{out}|$ (sourcing/sinking) is not more than $2.75mA$. At each step, note the output current and voltage using the ammeter and voltmeter and plot the output characteristics in both the cases.

3.4 Delay characterization of the inverter

The delay of an inverter can be modeled as

$$d_{abs} = k_0 + k_1 C_{load} \tag{3.4}$$

where k_0 and k_1 are constants, C_{load} is the load capacitance being driven by the inverter, and d_{abs} is the delay measured in seconds.

Usually, it is convenient to express the load C_{load} as a multiple of the

input capacitance C_{in} of the inverter. Then, we can write Equation 3.4 as

$$d_{abs} = k_0 + \tau_{inv} \frac{C_{load}}{C_{in}} \tag{3.5}$$

where the delay is measured in seconds.

Instead of thinking in terms of seconds, we can think in terms of multiples of the parameter τ_{inv}. The delay can be expressed in multiples of τ as

$$d_{inv} = p_{inv} + \frac{C_{load}}{C_{in}} \tag{3.6}$$

where $p_{inv} = k_0/\tau$ is the parasitic delay of the inverter (measured in τ units). The quantity τ is just the slope of the of the line in Equation 3.5,

Let

$$h \ = \ \frac{C_{out}}{C_{in}} \tag{3.7}$$

Then, the delay of the inverter (in τ) units is just

$$d_{inv} = \ p_{inv} + h \tag{3.8}$$

We will find τ and p_{inv} in this experiment.

For a general static complementary gate X, the delay (in τ_{inv} units) is

$$d_X = p_X + g_X h \tag{3.9}$$

and $g_X > 1$. That is, the inverter is the most efficient static complementary gate, and in general, g_X is a measure of the logical complexity of the gate X.

3.4.1 Experiment setup

Using the tools at our disposal, we will not be able to measure the delay of a single inverter directly. Instead, we construct a 17 stage ring oscillator. This ring-oscillator will produce a periodic waveform whose period is

$$\sum_{i=1}^{17} (d_{rise}(i) + d_{fall}(i)). \tag{3.10}$$

Thus, measuring the period of this ring oscillator will give us the value of $d_{rise} + d_{fall}$ for an inverter (assuming that all inverters in the ring are identical). We will construct the ring oscillator using several IC's of type

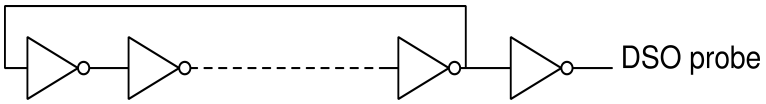

Figure 3.7: Ring oscillator circuit with default load (=2) (supply not shown)

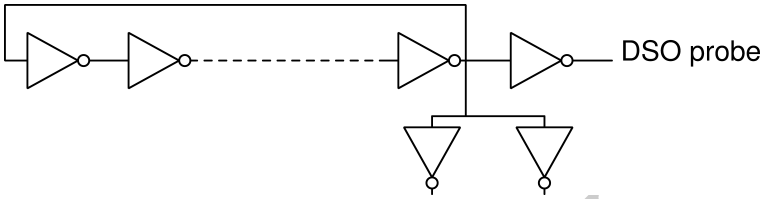

Figure 3.8: Ring oscillator circuit with additional 3-inverter load load (total-load=4) (supply not shown)

MM74C04, as shown in Figure 3.7. The output of one of the inverters in the ring can be connected to an extra load as shown in Figure 3.8.

For each IC, connect pin 7 and pin 14 of all ICs to V_{SS} and V_{DD} as before. Make the 17 stage ring oscillator by connecting inverters back to back as shown in Figure 3.7. The output of the ring oscillator can be loaded by multiples of single-inverter loads by connecting inverters as shown in Figures 3.7 and 3.8 (supply connections for load and ring oscillator are not shown). Connect the DSO probe as shown in the Figures (at the output of the buffer), so as to avoid loading on the ring oscillator due to the probe capacitance.

Vary the number of loads by connecting more inverters to the output (note that if no additional inverter is connected to the ring oscillator output, this corresponds to a load of 1 unit (inverter) at the output). Note that the period of oscillation will be

$$\tau \times (34p_{inv} + (32 + (2 \times (1 + AdditionalOutputLoad)))) \tag{3.11}$$

when measured in seconds (we have assumed that the rise and fall delays of an inverter are the same).

Measure the period of oscillation by varying the load from 1 to 7 units and calculate the delay. Plot the period of oscillation versus the load. From the slope of this plot, find τ. From the value of τ and y-intercept, find p_{inv}. Take a snapshot of the waveform at each load point.

Some precautions when using the breadboard:

- Keep the ICs as close as possible, use short wires and place decoupling capacitors (0.1 μF) between Vcc (pin 14) and gnd (pin 7) of each IC.

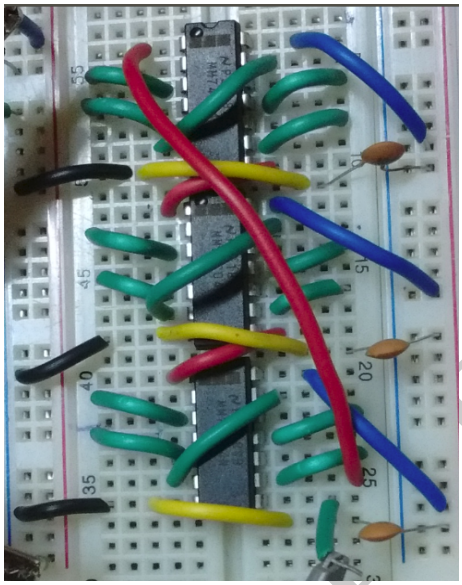

Figure 3.9: Connections on the breadboard

- Set the attenuation on probe to $1X$ (ie no attenuation) and make sure that attenuation of DSO is also set to $1X$.

3.5 Delay variation with supply voltage:

The delay of an inverter varies as the supply voltage is varied. As a first approximation, the delay of an inverter is proportional to

$$C_{load}V_{DD}/I_{ON} \tag{3.12}$$

where I_{ON} is proportional to $(V_{DD}-V_t)^2$. Thus, as V_{DD} is varied, the period of ring oscillator is proportional to

$$\frac{V_{DD}}{(V_{DD}-V_t)^2} \tag{3.13}$$

Therefore the time period decreases with the increase in supply voltage. Energy (E) dissipated in each inverter is proportional to V_{DD}^2, which increases with increase in supply voltage. Also the average power dissipated is proportional to $CV_{DD}^2 \times Frequency$, and also increases as supply voltage increases.

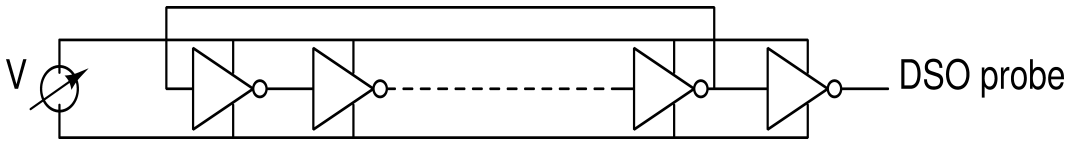

Figure 3.10: Test circuit for measuring the delay as a function of supply voltage

3.5.1 Experiment setup

Use the same setup as in Figure 3.7, vary the power supply (from 3 to 6 V), and measure the ring oscillator period at each voltage. Plot the period with power supply voltage, and check that the dependence is as noted in Equation 3.13. Take a snapshot of the waveforms at $4, 5, 6V$.

3.6 Current drawn by the ring oscillator

Whenever an inverter output switches from low to high, current is drawn from the power supply. The current drawn is approximately a triangular pulse whose width is essentially the delay of the gate. Therefore the current drawn by each inverter in the ring oscillator is a triangular pulse train with frequency equal to frequency of oscillation. The current drawn by the ring oscillator is a summation of the individual currents drawn by the inverters (note that the inverters switching events do not all happen at once). We can connect a small resistor in series with ground path to observe this in practice.

3.6.1 Experiment setup

Connect a small resistor in path of gnd in the ring oscillator as shown in Figure 3.11. The voltage across resistor will be proportional to the switching current. To measure this voltage an op-amp (TL072) buffer is used.

- Choose the value of R as 1Ω, so that the voltage is the switching current.
- Connect one pair of DSO probe at the output of buffer and other at output of ring oscillator, as shown in Figure 3.11. Connect the ground of DSO probes to the closest ground possible in the circuit from the two measurement points. Set the attenuation on probe to $1X$ (ie no attenuation) and make sure that attenuation of DSO is also set to $1X$.

- *Observe both waveforms together on DSO*, so that the DSO triggers correctly using the larger amplitude output voltage.
- Measure all voltages with respect to the common ground point as shown in Figure 3.11.
- Measure the frequency of voltage across resistor waveform manually (if needed), using the *cursor option* (cursor button) of the DSO, The DSO might show an incorrect frequency, because the amplitude is small.

3.7 Laboratory assignment report submission

Snapshot of DSO submitted should contain frequency, peak-peak amplitude of both the channels on the display. Compile all your readings, plots, observations and DSO snapshots in a pdf file and submit it on the online portal.

1. Plot the transfer characteristics of inverter and tabulate the input and corresponding output voltages. Find the switching point on the transfer characteristic.
2. Plot the output characteristics of the inverter in the two cases: output-high and output-low. Tabulate the input and corresponding output voltages for both the cases.
3. Plot the oscillation period of the ring oscillator as a function of the load at the output. Find p_{inv} and τ. A snap-shot of the DSO (get help from Laboratory staff if you do not have a camera) showing the ring-oscillator output with load = 2 should also be included in the report.
4. Plot the ring oscillator period as a function of the power supply voltage (varied from $3V$ to $6V$). Record your observations in a table. How does the delay vary with the power supply voltage?
5. Observe the power-supply current drawn by the ring oscillator. Measure the peak and average value. Include a snapshot of the DSO measurement.

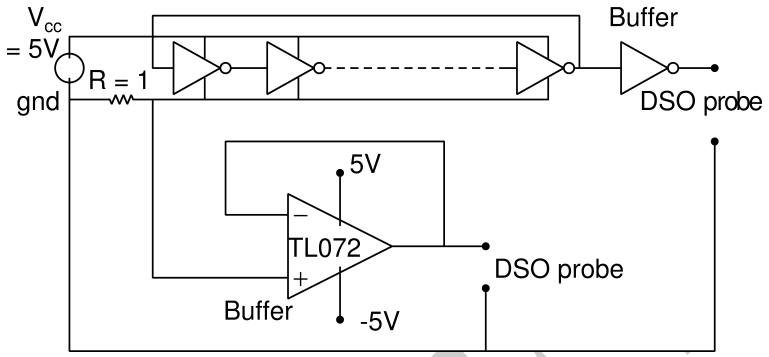

Figure 3.11: Circuit diagram for switching current measurement

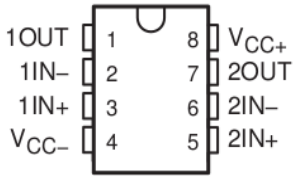

Figure 3.12: TL072 pin diagram

Bibliography

[1] Datasheet of TL072 operational amplifier:

http://wel.ee.iitb.ac.in/wel12/Components%20Records/analog%20ic_datasheet/tl072.pdf

Chapter 4

Familiarization: The Krypton CPLD kit

The Krypton kit [1] has been designed at IITB to serve as a platform for a digital electronics laboratory course. The components of the Krypton kit are

- The Krypton CPLD card, which uses an Altera MAX V 5M1270ZT144C5N CPLD.
- An SRAM daughter card (with a Hynix 62256A SRAM).
- A 4x4 keypad.
- 8 pin Strip cables.
- A USB cable for programming the CPLD.
- Krypton card USB programming driver (software).
- Supporting files for validation tests (software, documentation).

These components are described in more detail in Section 4.1.

In order to validate these components, we describe a series of tests, each of which involves connecting components to the CPLD card, programming the CPLD with a specific bit-stream, and validating the observed results with those that are expected. These tests are described in Section 4.2.

4.1 The components of the Krypton Kit

We describe the important components of the Krypton kit.

4.1.1 The Krypton board

The Krypton board contains a single Altera CPLD(MAX V - 5M1270ZT144C5N) and some other peripherals. A photograph of the board is shown in Figure 4.1.

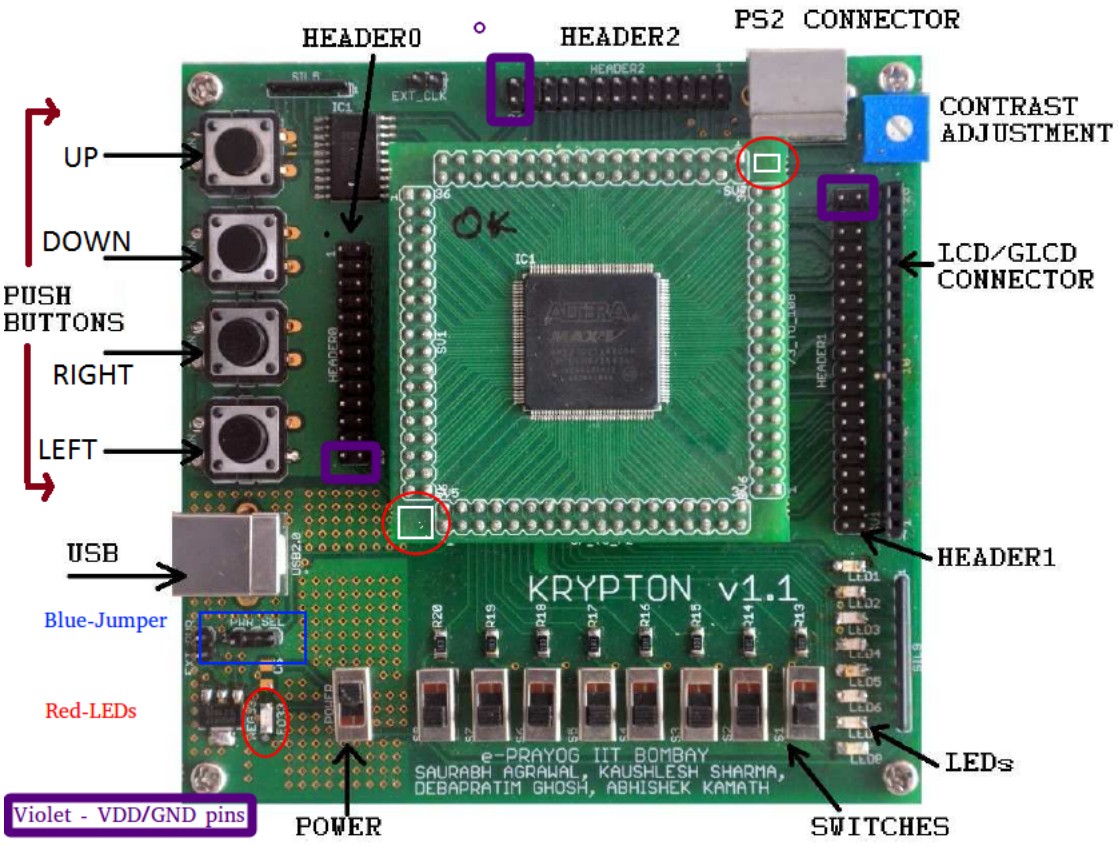

Figure 4.1: Krypton Board Top View. When connected to a host computer, the three LED's circled in red should be illuminated. Note the VDD/VSS pins in each header.

The important elements of the board are

- The CPLD.
- USB connector.
- 4 push-buttons.

- 8 on-board switches.
- 8 on-board LEDs.
- 20-pin connector for LCD panel.
- Three connection HEADERs (HEADER-0 with 26 pins, HEADER-1 with 40 pins, HEADER-2 with 26 pins) to connect external devices.
  - In each header, one pin is tied to VDD and one pin is tied to VSS as follows: In HEADER-0, pin-26 is tied to VDD and pin-25 is tied to VSS, in HEADER-1, pin-40 is tied to VDD and pin-39 is tied to VSS and in HEADER-2, pin-26 is tied to VDD and pin-25 is tied to VSS. **Do not connect other signals to these pins!**
- 5V power connector.

The board is normally powered through the USB connection, but can also be powered from an external power supply (5V) through the power connector. A jumper (shown in Figure 4.1) needs to be set to use one of the two power sources.

4.1.2 The LCD panel card

The LCD panel card offers a means of displaying information. The 16-pin LCD card is to be connected to the 20-pin LCD connector. When connecting the 16-pin LCD card into the 20-pin LCD connector, pins 1-16 of the connector are to be used. The connection is shown in Figure 4.2.

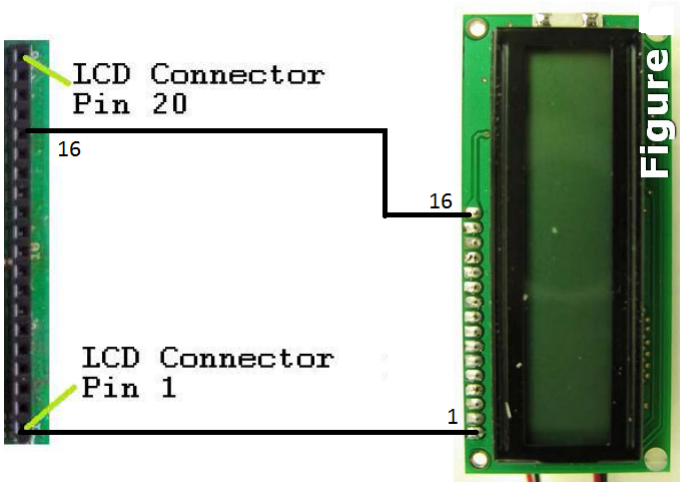

Figure 4.2: Connection of LCD Display

4.1.3 The SRAM daughter card

The SRAM daughter card is used to connect external SRAM to Krypton CPLD card. On the CPLD card, we use HEADER 2 to connect the SRAM daughter card. The SRAM daughter card has a 26 pin socket which fits on the 26 pin HEADER 2 of the CPLD card. The exact connection of daughter card to HEADER 2 is shown in Figure 4.3. To make sure your connection is proper, One LED should be lit on the SRAM daughter card.

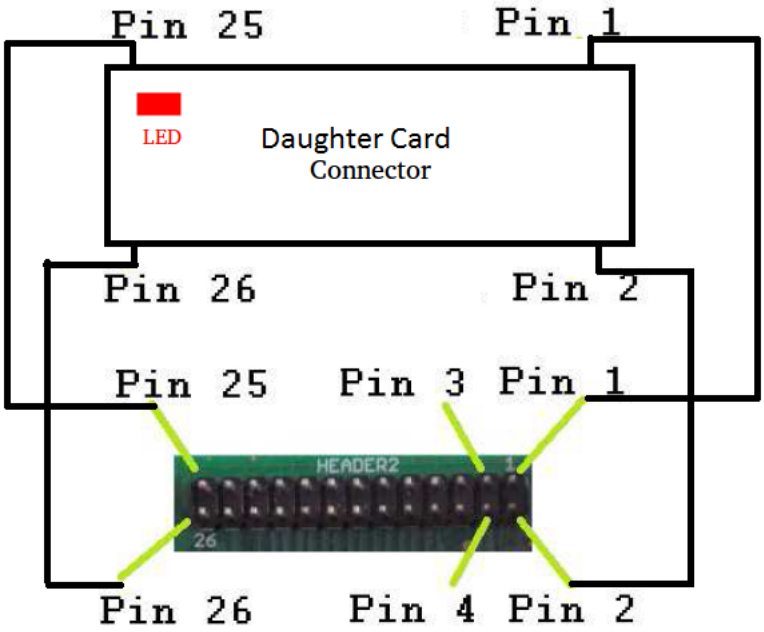

Figure 4.3: Connection of daughter card to HEADER 2 of kit, rectangle in red displays an LED

4.1.4 4x4 Keypad

A 4x4 keypad can be connected to the Krypton CPLD card as an input device. The keypad has an in-built pull up resistor, and no external pull up resistor is needed. The keypad has four row pins as inputs and four column pins as outputs (8 I/O pins). It is to be connected to the Krypton card at HEADER 0. Eight pins of keypad are connected to eight pins of HEADER 0 as shown in Figure 4.4.

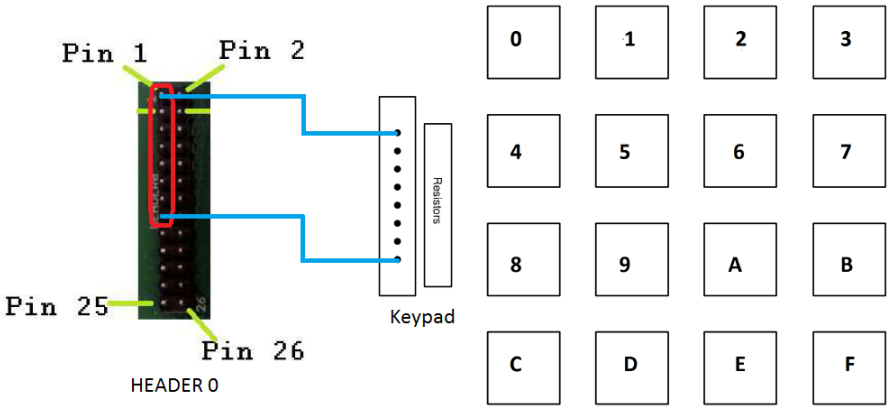

Figure 4.4: Connection of Keypad to HEADER 0 of kit

4.1.5 8-pin strip cable

This cable can be used to make connections between HEADERs and between HEADERs and external devices. An example of the use of this cable is shown in Figure 4.5, where the strip cable is used to make a connection between two rows of HEADER0.

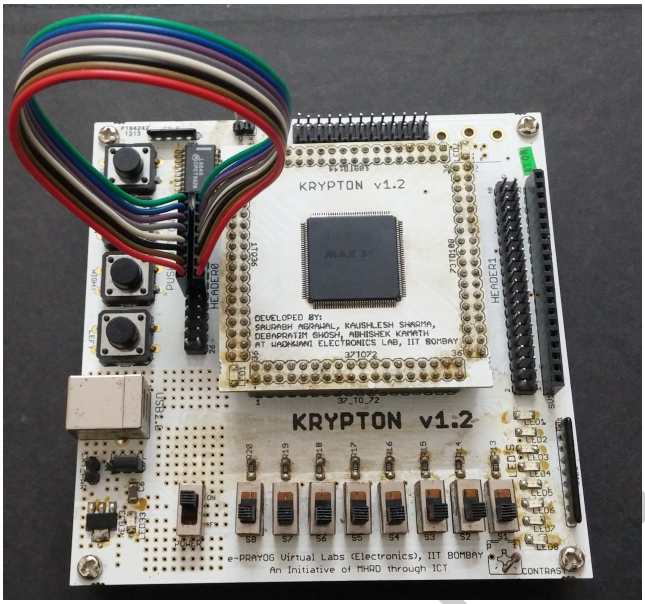

Figure 4.5: Connection between rows of HEADER0 through Strip Cables

4.1.6 USB programming cable

The Krypton CPLD card needs to be programmed using a USB connection. A standard USB cable is used to make a connection between the Krypton CPLD card and a host PC (running Linux Ubuntu 12.04/14.04). The Krypton driver is then used to program the CPLD card.

4.1.7 The Krypton card driver for Linux

You will need to have the Krypton card driver for Linux installed on the host PC which you will use to program the card. This driver can be downloaded and installed by following the instructions in [3].

4.1.8 Validation test files

All validation files are of .svf format. The test files can be downloaded from [4]. These validation files are to be used to program the CPLD card in order to test various aspects of the Krypton kit.

4.2 Validation tests

The validation tests provided in the kit are as follows

- use the host computer to program the Krypton board.
- push-button, LED, switch, clock test.
- LCD test.
- SRAM test.
- Header connectivity test.
- Keypad interfacing test.

We describe these tests in more detail.

4.2.1 Test the USB connection between the host computer and the Krypton card

A USB cable is connected between host PC and Krypton CPLD card. When the CPLD board is powered on, three LED's in the board should light up (see Figure 4.1 for the location of these LED's). If the three LED's are lit up, we proceed to check whether the CPLD card can be programmed successfully.

The test circuit that we use is a simple combinational circuit with 3 inputs A, B, C, connected to switches 1, 2 and 3. Each input value can be set to either 0 or 1 using switch. The output is $B(A + C)$ and is displayed at LED-1. Use the Krypton CPLD driver to load the test file called Program_Load.svf file into Krypton Board. You can then use the switches to check whether the output is as expected.

To load the svf file into the Krypton CPLD card:

- To remove the unwanted modules, type following command[1]

```
sudo -i
lsmod
```

  A list of detected modules is seen. We need to remove two of them as follows:

[1] You will need super-user permission to do this. If you do not have this permission, contact a teaching assistant.

```
rmmod ftdi_sio usbserial
```

- At the command prompt, type the command

  ```
  jtag
  ```

- Now you will be presented with a command console. Type

  ```
  cable ft2232
  ```

  An indication should now appear that the driver is connected.

- Now type :

  ```
  detect
  ```

  which displays the detected CPLD device (Details like IR&Chain length, Manufacturer, Device ID, Stepping etc).

- Load the svf file into the CPLD device by typing

  ```
  svf <full-path-of-svf-file> progress
  ```

  Thats it, the board is programmed!

More details can be found in reference [2].

4.2.2 Test the LED's, switches, push-buttons and Clocks

The Krypton CPLD card has several elements built into it:

- Two clock signals, one of 1Hz and other of 50Mhz. These are connected to CPLD IO pin 18 and pin 89 respectively.
- 8 Switches, 8 LEDs and 4 push-buttons with hardware debounce.

To test these elements, load the On_board_peripheral.svf file into Krypton board. Initially all switches are to be kept off. For each test, exactly one of the switches is to be turned on. The tests are as follows:

- Switch s1: The eight LED's will be lit up to display an 8-bit count which is updated every $0.04194s$.

- Switch s2 : The eight LED's will be lit up to display an 8 bit count which is updated every second.
- Switch s3 : All LEDs are lit up. There are four push buttons. You can check each of them by pressing them one at a time and checking the observations which should be as follows:
  - If push-button **up** is pressed, LED 1 and 5 will turn off.
  - If push-button **down** is pressed, LED 2 and LED 6 will turn off.
  - If push-button **right** is pressed, LED 3 and LED 7 will turn off.
  - If push-button **left** is pressed, LED 4 and LED 8 will turn off.
- Switches s4,s5,s6,s7,s8 (one at a time): The corresponding LED (that is LED-4 for switch s4 etc.) will turn on.

4.2.3 Test the LCD card

Connect LCD display to LCD connector. The LCD connector is to be plugged into board directly as shown in Figure 4.2. Use the test file LCD.svf. After loading the program switch S1 has to turned on. You should see eventually see activity on the LCD which ends with "KRYPTON" displayed on the LCD. The process will take about 45 seconds. Push the **up** key to reset the LCD display.

4.2.4 Test the CPLD card headers

There are three headers in the Krypton CPLD card (HEADER-0, HEADER-1, HEADER-2). Every HEADER has 2 rows of pins. We will check each header by writing to each pin and reading from each pin. We test one header at a time by connecting its two rows using strip cables. Before starting the test, we program the CPLD card using the svf file IO.svf. Now, switches are used to run the tests. Note that for testing each header, you will to connect its two rows using strip cables. **Note that two pins in each header are tied to VDD and VSS as described in Section 4.1.1. Be careful about these!** The switch settings for the different tests are describe below. After changing a switch setting, you will need to press and release the reset key (push-button **up**) to start the test.

- Switch 1 : HEADER0 Data sent from first row to second row
- Switch 2 : HEADER0 Data sent from second row to first row

- Switch 3 : HEADER1 Data sent from first row to second row
- Switch 4 : HEADER1 Data sent from second row to first row
- Switch 5 : HEADER2 Data sent from first row to second row
- Switch 6 : HEADER2 Data sent from second row to first row

In each case, the output displayed on LED-8 down to LED-1 will be 0x66 (01100110 LED-8 is the most-significant bit) if the test has passed.

4.2.5 Test the keypad

The keypad is connected to HEADER 0 as described in Figure 4.4. Load the Keypad.svf file into the CPLD card. Now start presssing keys on the keypad. The current key being pressed is displayed on LED-5 to LED-8 (the hex-code for the key will be displayed with LED-5 being the most-significant bit (MSB)), and the last key that you pressed is displayed on LED-1 to LED-4 (LED-1 is the MSB).

4.2.6 Test the SRAM

We test the SRAM by ensuring that all memory elements can be correctly written into and read from. The SRAM (Hinex 62256) is a high-speed, low power and 32K x 8-bit (32KB) CMOS Static Random Access Memory. The SRAM card is connected to the Krypton CPLD card as shown in Figure 4.3. Only 13 bits of the address are actually used so that this SRAM card provides access to 8KB of memory.

Load the SRAM.svf file into CPLD card. Press the **up** push key (this is used as a reset) on the CPLD card. When the reset key is pressed, LED-1 to LED-8 should display 0xAA (10101010). Now release the reset key. This will run the test and the test status will eventually be displayed on LED-1 to LED-8. If the test has passed, LED-1 to LED-8 should display 0x66 (01100110 in binary, LED-8 is the most-significant bit). If the test has failed, you will either see 0x69 (01101001) which indicates that a 0 could not be correctly read from or written to a memory location, or you will see 0x96 (10010110) which indicates that a 1 could not be correctly read to or written from a memory location. If both the 0-read/write and 1-read/write fail, you will see 0x99 (10011001).

Bibliography

[1] Krypton getting started with User Manual available at

http://wel.ee.iitb.ac.in/teaching_labs/WEL%20Site/ee214/resources/development_boards/krypton/Krypton%20Usermanual.pdf

[2] UrJTAG PPT available at

http://wel.ee.iitb.ac.in/teaching_labs/WEL%20Site/ee214/resources/development_boards/krypton/Using%20Quartus%20and%20UrJTAG%20for%20Krypton.pdf

[3] Krypton drivers

http://wel.ee.iitb.ac.in/teaching_labs/WEL%20Site/ee214/resources/development_boards/krypton/Kypron%20drivers_CDM20817.zip

[4] Krypton kit validation test svf files

http://wel.ee.iitb.ac.in/teaching_labs/WEL%20Site/ee214/resources/development_boards/krypton/Kypron%20test_files.zip

Chapter 5

Experiment: Combinational logic implementation on the Krypton kit

We will first demonstrate the implementation of a priority encoder using the Krypton CPLD kit, and verify that the implementation is correct by using the on-board switches and LED's in the kit. The circuit to be implemented is an 8-to-3 encoder which has 8 inputs signals x7,x6,x5,x4,x3,x2,x1,x0, and produces 3 bit encoded output s2,s1,s0 and a signal bit N indicating whether the bits on s2,s1,s0 are valid or not. If all the input bits to the encoder are 0, then N=1 and s2,s1,s0 are dont-cares. If at least one of the input bits to the encoder is 1, then N=0, and the bits s2,s1,s0 indicate the binary code for the lowest index I for which the corresponding input xI is 1. So when multiple input bits are 1, the encoded bit s2,s1,s0 represent the binary representation of lowest index I such that xI is 1.

In implementing this circuit, we go through the following steps:

- Design the logic network corresponding to the specification.
- Describe the logic network using VHDL.
- Write a test-bench (in VHDL) to verify the logic network.
- Simulate the test-bench with the logic network in order to verify the correctness of the implementation.
- Synthesize the logic network for the KRYPTON CPLD. This produces programming files as well as a post-synthesis logic netlist which you can then simulate.

- Simulated the synthesized network to verify the correctness of the synthesis.
- Download the programming files onto the KRYPTON CPLD card.
- Test the correctness of the implementation on the KRYPTON CPLD card using the switches and LED's on the CPLD card.

5.1 Logic Design

Based on the truth table, we construct the Boolean functions for the outputs s2,s1,s0 and N as follows:-

$$
\begin{aligned}
N &= \overline{x7 + x6 + x5 + x4 + x3 + x2 + x1 + x0} \\
s0 &= = x1.\overline{x0} + x3.\overline{x2}.\overline{x1}.\overline{x0} + x5.\overline{x4}.\overline{x3}.\overline{x2}.\overline{x1}.\overline{x0} \\
&\quad + x7.\overline{x6}.\overline{x5}.\overline{x4}.\overline{x3};\ .\overline{x2}.\overline{x1}.\overline{x0} \\
s1 &\quad x2.\overline{x1}.\overline{x0} + x3.\overline{x2}.\overline{x1}.\overline{x0} + x6.\overline{x5}.\overline{x4}.\overline{x3}.\overline{x2}.\overline{x1}.\overline{x0} \\
&\quad + x7.\overline{x6}.\overline{x5}.\overline{x4}.\overline{x3}.\overline{x2}.\overline{x1}.\overline{x0} \\
s2 &= x4.\overline{x3}.\overline{x2}.\overline{x1}.\overline{x0} + x5.\overline{x4}.\overline{x3}.\overline{x2}.\overline{x1}.\overline{x0} + x6.\overline{x5}.\overline{x4}.\overline{x3}.\overline{x2}.\overline{x1}.\overline{x0} \\
&\quad + x7.\overline{x6}.\overline{x5}.\overline{x4}.\overline{x3}.\overline{x2}.\overline{x1}.\overline{x0}
\end{aligned}
$$

5.2 Implementation in VHDL

The eight inputs are treated as separate scalars x7,x6,x5,x4,x3,x2,x1,x0, and the three output bits are treated as scalars s2,s1,s0 and N. The interface is as shown in Figure 5.1.

We implement the four output boolean logic functions for s2,s1,s0 (or bit_vector **s**) and N using concurrent signal assignment statements.

5.3 VHDL Implementation of the Priority Encoder With Scalar Interface

The Boolean functions for the four outputs are directly implemented as four concurrent assignments.

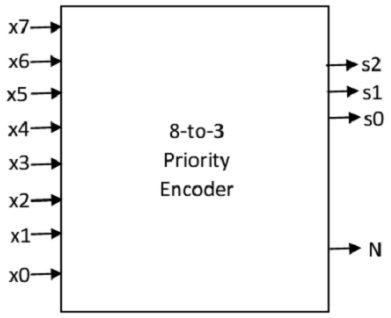

Figure 5.1: 8-to-3 Priority Encoder With Scalar Interface

5.3.1 VHDL code of PriorityEncoder

```
1   library ieee;
    use ieee.std_logic_1164.all;
3   entity PriorityEncoder is
      port(x7,x6,x5,x4,x3,x2,x1,x0:in bit;
5        s2,s1,s0,N:out bit);
    end PriorityEncoder;
7   architecture comb of PriorityEncoder is
    begin
9      N  <= not(x7 or x6 or x5 or x4 or x3 or x2 or x1 or x0);
       s0 <= (x1 and not x0) or
11               (x3 and not x2 and not x1 and not x0) or
                 (x5 and not x4 and not x3 and not x2 and
13                  not x1 and not x0) or
                  (x7 and not x6 and not x5 and not x4
15                    and not x3 and not x2 and not x1
                      and not x0);
17     s1 <= (x2 and not x1 and not x0) or
                 (x3 and not x2 and not x1 and not x0) or
19               (x6 and not x5 and not x4 and not x3 and
                     not x2 and not x1 and not x0) or
21               (x7 and not x6 and not x5 and not x4 and
                   not x3 and not x2 and not x1 and not x0);
23     s2 <= (x4 and not x3 and not x2 and
                        not x1 and not x0) or
25             (x5 and not x4 and not x3 and not x2 and
                                  not x1 and not x0) or
27             (x6 and not x5 and not x4 and not x3
                   and not x2 and not x1 and not x0) or
29             (x7 and not x6 and not x5 and not x4 and not x3
                   and not x2 and not x1 and not x0);
31  end comb;
```

5.4 Concept of a test-bench

For every entity described in VHDL, a self-checking test-bench is necessary to verify whether the entity is working correctly or not. The testbench is basically another VHDL file which applies a set of input combinations to the entity being tested (termed as the design-under-test (DUT)), and checks whether it produces the correct and expected output or not.

In our case, the DUT is a 8-to-3 Priority Encoder as discussed above. Our testbench applies all the possible 256 input combinations and checks whether the output is as expected or not. If the output is not as expected, it reports that particular input combination along with the expected output and produced/actual output. It also keeps a count of the number of input combinations for which the test fails. If the test passes for all possible 256 input combinations, we can conclude that our implementation of the test-bench is correct.

Note: it is not possible to exhaustively test a combinational circuit if the number of inputs is very large. In such cases, at the minimum, we choose a set of tests such that every output bit takes a value of 0 in at least one test and a value of 1 in at least one test.

We have implemented the Testbench in two ways.

- The working operation of the priority encoder is modelled in Python programming language, and the expected output is written in a text file. Basically this file is a text file which contains all the 256 lines (corresponding to 256 possible input combinations). Each line consists of an 8-bit input vector x, and the corresponding expected 3-bit output vectors s, and the output bit N, each field separated by a blank space. In the VHDL description of the test-bench, this text file is read one line at a time and the corresponding input bit vector x, output bit vector s and output bit N are separated out. Then the input bit vector x is applied to the component DUT and the output s and N are matched with the one in the file. If it matches, it goes to the next line until the end of the file is reached. If the outputs doesnt match, it reports an error indicating the input vector, the expected output and produced output. It also keeps a count of the number of times the test fails.

When the test is completed, it prints the number of successes and failures.

- In the second form, we have generated all possible 256 input combinations inside the architecture of VHDL testbench file itself, and applied it to the component DUT. The expected output is also computed within the test-bench and compared with the observed output.

5.4.1 VHDL code of the testbench

To test the entity PriorityEncoder, we implemented the testbench in two possible ways as was described above. In one implementation, we generated a text file outputs.txt which contains all the inputs and corresponding expected outputs and matched each line of this file with the output generated by the entity PriorityEncoder. In the other implementation, the input combinations are generated inside the testbench code itself and the outputs are matched with the expected output. We present both the implementations of the test-bench.

Testbench using a file to match the results

```
1   library ieee;
    use ieee.std_logic_1164.all;
3   use std.textio.all;

5
    entity test_match is
7   end test_match;

9   architecture testbench of test_match is
      signal x7,x6,x5,x4,x3,x2,x1,x0,s2,s1,s0,N:bit:='0';
11
    component PriorityEncoder
13    port(x7,x6,x5,x4,x3,x2,x1,x0:in bit;
           s2,s1,s0,N:out bit);
15  end component;

17  function bitvec_to_str(x:bit_vector) return String is
    variable L:line;
19  variable W:String(1 to x'length):=(others=>' ');
    begin
21    write(L,x);
      W(L.all'range):=L.all;
23    Deallocate(L);
      return W;
```

```
25      end bitvec_to_str;

27      begin
        process
29        file f:text open read_mode is "outputs.txt";
          variable x:bit_vector(7 downto 0);
31        variable s_temp:bit_vector(2 downto 0);
          variable N_temp:bit;
33        variable L:line;
          variable fail,success:integer:=0;
35        variable s:bit_vector(2 downto 0);
        begin
37        while not endfile(f) loop
            readline(f,L);
39          read(L,x);
            read(L,s_temp);
41          read(L,N_temp);
            x7<=x(7);
43          x6<=x(6);
            x5<=x(5);
45          x4<=x(4);
            x3<=x(3);
47          x2<=x(2);
            x1<=x(1);
49          x0<=x(0);
            wait for 10 ns;
51          s(2):=s2;
            s(1):=s1;
53          s(0):=s0;
            wait for 0 ns;
55          if not (x="00000000") then
               assert(s=s_temp)
57             report "Error. Input is " &  bitvec_to_str(x) &
                       " Expected " &  bitvec_to_str(s_temp) &
59                     " Produced " & bitvec_to_str(s)
               severity error;
61          end if;
            assert(N=N_temp)
63            report "Error in N. Input is "& bitvec_to_str(x) &
                     " Expected " & bit'image(N_temp) & " Produced
    "
65                   & bit'image(N)
              severity error;
67          if (x="00000000") then
              if not(N=N_temp) then
69              fail:=fail+1;
              end if;
71          elsif not(s=s_temp and N=N_temp) then
              fail:=fail+1;
```

```
        end if;
      end loop;
      success:=256-fail;
      assert false report "Test completed. " &
                integer'image(success) &
                  " successes. " & integer'image(fail)& "
    failures. "
        severity note;
      wait;
    end process;
    dut:PriorityEncoder
      port map(x7=>x7,x6=>x6,x5=>x5,x4=>x4,
                  x3=>x3,x2=>x2,x1=>x1,x0=>x0,
                     s2=>s2,s1=>s1,s0=>s0,N=>N);
    end testbench;

```

The testbench code uses a function called bitvect_to_str which converts a bit_vector to String. This is required because in the report statement, we indicate the input combination, the expected, and produced output if an error occurs. Since these are of the type bit_vector, and the report statement only accepts a String, we need to convert bit_vector to String in order to report it.

Let us examine this testbench in detail. A small portion of the outputs.txt file is shown below.

```
10011100 010 0  <----
10011101 000 0
10011110 001 0
10011111 000 0
10100000 101 0
```

Suppose we are currently examining the line indicated by arrow.

```
readline(f,L);
```

This reads the entire line from the file and stores in L.

```
read(L,x);
read(L,s_temp);
read(L,N_temp);
```

This section reads each word from the line L one after the another and stores it in different bit vectors. So x gets the input combination 10011100, s_temp gets 010 which is the expected output and N_temp gets 0 which is the expected value of N.

Then from the bit_vector x, all the bits are assigned to signals x7, x6, x5, x4, x3, x2, x1, x0 which are mapped to input ports of the entity PriorityEncoder. The entity produces the output bits s2,s1,s0,N as described in the architecture of the entity. The output bits s2,s1,s0 are stored in a bit_vector s. Now two assert statements are used to check whether s_temp (containing the expected output of s2,s1,s0 from file) matches with s (containing the produced output s2,s1,s0 from DUT) and N_temp (containing the expected output N from file) matches with s (containing the produced output N from DUT). At the end it reports that the test has completed along with the number of successful and failed test cases.

One special case occurs when the input is 00000000. The corresponding output s2,s1,s0 should be dont cares, meaning that the value contained by them is meaningless. The outputs.txt file contains 000 as output for this case. Therefore we should not check whether s2,s1,s0 matches expected output for this case. So in the code for this input, only the N bit is checked.

5.4.2 Testbench which generates the inputs and expected outputs and does the check

This approach is more efficient when the number of patterns to be applied is very large. When calculating the expected output you should use an algorithm that is different from the one used in the DUT for computing the output (Why?).

```
1  library ieee;
   use ieee.std_logic_1164.all;
3  use ieee.numeric_bit.all;
   use std.textio.all;
5
   entity test_self is
7  end test_self;
```

```
9     architecture testbench of test_self is
        signal x7,x6,x5,x4,x3,x2,x1,x0,s2,s1,s0,N:bit:='0';
11
        component PriorityEncoder
13        port(x7,x6,x5,x4,x3,x2,x1,x0:in bit;
            s2,s1,s0,N:out bit);
15      end component;

17      function bitvec_to_str(x:bit_vector) return String is
           --  AS BEFORE ---
19      begin
           --  AS BEFORE ---
21      end function bitvect_to_str;
      begin
23
        for i in 0 to 255 loop
25        x:=bit_vector(to_unsigned(i,8));
          wait for 0 ns;
27        x7<=x(7);
          x6<=x(6);
29        x5<=x(5);
          x4<=x(4);
31        x3<=x(3);
          x2<=x(2);
33        x1<=x(1);
          x0<=x(0);
35        wait for 10 ns;
          s(2):=s2;
37        s(1):=s1;
          s(0):=s0;
39
          if (x0='1') then
41            assert(s="000" and N='0')
              report "Error. Input is " &  bitvec_to_str(x) &
43                   " Expected S=000 N=0 Produced S=" &
                     bitvec_to_str(s)&" N="&bit'image(N)
45            severity error;
              if not(s="000" and N='0') then fail:=fail+1;
47            end if;
          elsif (x1='1') then
49            assert(s="001" and N='0')
              report "Error. Input is " &  bitvec_to_str(x) &
51                 " Expected S=001 N=0 Produced S=" &
                    bitvec_to_str(s) & " N=" & bit'image(N)
53            severity error;
              if not(s="001" and N='0') then fail:=fail+1;
55            end if;
          elsif (x2='1') then
```

```
57              assert(s="010" and N='0')
                report "Error. Input is " &  bitvec_to_str(x) &
59                   " Expected S=010 N=0 Produced S=" &
    bitvec_to_str(s) &
                     " N=" & bit'image(N)
61                   severity error;
                if not(s="010" and N='0') then fail:=fail+1;
63              end if;
            elsif (x3='1') then
65              assert(s="011" and N='0')
                report "Error. Input is " & bitvec_to_str(x) &
67                    " Expected S=011 N=0 Produced S=" &
    bitvec_to_str(s) &
                    " N=" & bit'image(N)
69                  severity error;
                if not(s="011" and N='0') then fail:=fail+1;
71              end if;
            elsif (x4='1') then
73              assert(s="100" and N='0')
                report "Error. Input is " & bitvec_to_str(x) &
75                    " Expected S=100 N=0 Produced S=" &
    bitvec_to_str(s) &
                    " N=" & bit'image(N)
77                   severity error;
                if not(s="100" and N='0') then fail:=fail+1;
79              end if;
            elsif (x5='1') then
81              assert(s="101" and N='0')
                report "Error. Input is " & bitvec_to_str(x) &
83                    " Expected S=101 N=0 Produced S=" &
    bitvec_to_str(s) &
                    " N=" & bit'image(N)
85                   severity error;
                if not(s="101" and N='0') then fail:=fail+1;
87              end if;
            elsif (x6='1') then
89              assert(s="110" and N='0')
                report "Error. Input is " & bitvec_to_str(x) &
91                    " Expected S=110 N=0 Produced S=" &
    bitvec_to_str(s) &
                     " N="&bit'image(N)
93                   severity error;
                if not(s="110" and N='0') then fail:=fail+1;
95              end if;
            elsif (x7='1') then
97
                assert(s="111" and N='0')
99              report "Error. Input is " & bitvec_to_str(x) &
                    " Expected S=111 N=0 Produced S=" &
```

```
    bitvec_to_str(s) &
                    " N="&bit'image(N)
                     severity error;
            if not(s="111" and N='0') then fail:=fail+1;
            end if;
        else
            assert(N='1')
            report "Error in N. Input is " & bitvec_to_str(x) &
                   " Expected N=1 Produced " & bit'image(N)
                   severity error;
            if not(N='1') then fail:=fail+1;
            end if;
        end if;

      end loop;
      success:=256-fail;
      assert false report "Test completed. " &
                  integer'image(success) & " successes. " &
                  integer'image(fail)&" failures."
      severity note;
      wait;
    end process;
    dut:PriorityEncoder
    port map(x7=>x7,x6=>x6,x5=>x5,x4=>x4,
                    x3=>x3,x2=>x2,x1=>x1,x0=>x0,
                          s2=>s2,s1=>s1,s0=>s0,N=>N);
    end testbench;

```

In this testbench the inputs are generated inside the code using a loop which runs from 0 to 255, each time converting the loop variable i to a bit_vector x.

```
x:=bit_vector(to_unsigned(i,8));
```

Then from the bit_vector x, all the bits are assigned to signals x7, x6, x5, x4, x3, x2, x1, x0 which are mapped to input ports of the entity PriorityEncoder. The entity produces output bits s2,s1,s0,N as described in the architecture of the entity. The output bits s2,s1,s0 are stored in a bit_vector s. Then the output is checked to see whether it matches the expected output or not. The different cases are quite obvious which is represented here in the form of a pseudo code for convenience.

```
    if x0 ==1
    s=000 and N=0
```

```
    else if x1==1
    s=001 and N=0
    else if x2==1
    s=010 and N=0
    else if x3==1
    s=011 and N=0
    else if x4==1
    s=100 and N=0
    else if x5==1
    s=101 and N=0
    else if x6==1
    s=110 and N=0
    else if x7==1
    s=111 and N=0
    else
    s=xxx and N=1


```

So for each input, the outputs are compared and checked to see if there are any errors or not. At the end it reports test completion along with the number of successful and failed test cases.

5.4.3 Python Code for creating the test trace file used in the test-bench (first form)

```
def tobinary(n,width):
  s=''
  for i in range(width):
     s=s+str(n%2)
     n=n/2
     s=s[::-1]
return s

f=open('outputs.txt','w')
f.seek(0)
f.truncate()
for i in range(256):
  x=tobinary(i,8)
  s='000'
  if x=='00000000':
    N='1'
  else:
    N='0'
  if x[7]=='1':
```

```
        s='000'
    elif x[6]=='1':
        s='001'
    elif x[5]=='1':
        s='010'
    elif x[4]=='1':
        s='011'
    elif x[3]=='1':
        s='100'
    elif x[2]=='1':
        s='101'
    elif x[1]=='1':
        s='110'
    elif x[0]=='1':
        s='111'
    f.write(x+" "+s+" "+N+"\n")
f.close()

```

This code generates the file named outputs.txt which contains the testbench data.

5.5 Results obtained by simulating the TestBench together with the PriorityEncoder

Here we show the output of the testbench when applied to the DUT.

5.5.1 Case 1: No errors

```
testbench_match.vhd:60:16:@2560ns:(assertion note): Test
completed. 256 success. 0 failure.
```

5.5.2 Case 2: Output bit s2 is intentionally stuck at 0

```
testbench_match.vhd:48:24:@170ns:(assertion error): Error.
Input is 00010000 Expected 100 Produced 000
testbench_match.vhd:48:24:@330ns:(assertion error): Error.
Input is 00100000 Expected 101 Produced 001
```

```
 testbench_match.vhd:48:24:@490ns:(assertion error): Error.
Input is 00110000 Expected 100 Produced 000
 testbench_match.vhd:48:24:@650ns:(assertion error): Error.
Input is 01000000 Expected 110 Produced 010
 testbench_match.vhd:48:24:@810ns:(assertion error): Error.
Input is 01010000 Expected 100 Produced 000
 testbench_match.vhd:48:24:@970ns:(assertion error): Error.
Input is 01100000 Expected 101 Produced 001
 testbench_match.vhd:48:24:@1130ns:(assertion error):
Error. Input is 01110000 Expected 100 Produced 000
 testbench_match.vhd:48:24:@1290ns:(assertion error):
Error. Input is 10000000 Expected 111 Produced 011
 testbench_match.vhd:48:24:@1450ns:(assertion error):
Error. Input is 10010000 Expected 100 Produced 000
 testbench_match.vhd:48:24:@1610ns:(assertion error):
Error. Input is 10100000 Expected 101 Produced 001
 testbench_match.vhd:48:24:@1770ns:(assertion error):
Error. Input is 10110000 Expected 100 Produced 000
 testbench_match.vhd:48:24:@1930ns:(assertion error):
Error. Input is 11000000 Expected 110 Produced 010
 testbench_match.vhd:48:24:@2090ns:(assertion error):
Error. Input is 11010000 Expected 100 Produced 000
 testbench_match.vhd:48:24:@2250ns:(assertion error):
Error. Input is 11100000 Expected 101 Produced 001
 testbench_match.vhd:48:24:@2410ns:(assertion error):
Error. Input is 11110000 Expected 100 Produced 000
 testbench_match.vhd:60:16:@2560ns:(assertion note): Test
completed. 241 success. 15 failure.

```

5.6 Experimental procedure

- Implement the priority encoder and its test-bench as described above. Prepare two text files.
- Use a VHDL simulator to simulate the test-bench with the priority encoder instantiated as the DUT. You may need to generate a trace file if you use the test-bench which needs the trace file. You may use either the simulator in the Altera Quartus tools, or use GHDL (see Section 5.8 for details on how to use GHDL).
- If the simulation indicates that the priority encoder implementation works for all 256 input combinations, proceed to the implementation.

You will need to synthesize the priority encoder description using the Altera tools targeting the Krypton CPLD. Synthesis instructions are summarized in Section 5.9.

  - Map the eight inputs to the priority encoder to the pins on the Krypton CPLD which are connected to the eight switches on the Krypton CPLD card.
  - Map the four outputs of the priority encoder to the pins on the Krypton card which are connected to the LED's on the Krypton CPLD card.

- Do a post-synthesis simulation of the net-list produced by the synthesis tool to confirm that the synthesis tool has not made an error.
- If the post-synthesis simulation has passed, download the synthesized bit-stream to the Krypton CPLD using the programming cable. Download instructions are summarized in Section 5.10.
- Confirm that your circuit is working on the CPLD card by trying the following nine combinations: all 0-s, eight possible combinations in which exactly one of the inputs is 1. Use the switches to set the values, and observe the results on the LED's.

You will need to summarize your results at each step in preparing a report for your experiment.

5.7 Issues

The procedure that you have used for testing your design in hardware is not very satisfactory, because it cannot be scaled to circuits with a large number of inputs and outputs. In such cases, we need an automated mechanism to apply inputs and observe outputs. You will be introduced to such a mechanism which you will then use to verify more complex circuits.

5.8 Installing and using GHDL

GHDL is a free VHDL simulator which was developed by Tristan Gingold. You can use it on a Linux machine or a Windows machine.

5.8.1 Installing ghdl on Windows

Go to the following website.

`http://sourceforge.net/projects/ghdl-updates/`

Download ghdl-0.31-mcode-win32.zip and follow the following instructions.

1. Create a ghdl directory where desired, e.g. C:\ghdl
2. Unzip ghdl-0.31-mcode-win32.zip into C:\ghdl
3. Open a command shell and cd into the resulting installation directory, C:\ghdl\ghdl-0.31-mcode-win32
4. Run the batch files named $set_ghdl_path.bat$ and $reanalyze_libraries.bat$
5. Now set *PATH* and *GHDL_PREFIX* in the Windows SYSTEM or USER environment variables by following steps:
   - Go to Start→ Settings→ Control Panel→ System→ Advanced system settings→ Environment Variables
   - Prefix the existing PATH with **C:\ghdl\ghdl-0.31-mcode-win32\bin;**
   - Add a new environment variable **GHDL_PREFIX=c:\ghdl\ghdl-0.31-mcode-win32\lib**

Installation check on Windows

Once **ghdl** is set up, issuing the following command:

```
C:\textbackslash ghdl\textbackslash ghdl-0.31-mcode-win32>ghdl -v
```

This should print the version should see

```
GHDL 0.31 (20140108) [Dunoon edition] + ghdl-0.31-mcode-win32.patch
Compiled with GNAT Version: GPL 2013 (20130314)
mcode code generator
Written by Tristan Gingold.

Copyright (C) 2003 - 2014 Tristan Gingold.
GHDL is free software, covered by the GNU General Public License.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

C:\textbackslash ghdl\textbackslash ghdl-0.31-mcode-win32>ghdl --dispconfig
```

This should print the configuration information and you should see

```
command line prefix (--PREFIX): (not set)
environment prefix (GHDL_PREFIX): C:\Ghdl\ghdl-0.31-mcode-win32\lib
default prefix: C:\Ghdl\ghdl-0.31-mcode-win32\lib
actual prefix: C:\Ghdl\ghdl-0.31-mcode-win32\lib
command_name: C:\Ghdl\ghdl-0.31-mcode-win32\bin\ghdl.exe
default library pathes:
C:\Ghdl\ghdl-0.31-mcode-win32\lib\v93\std\
C:\Ghdl\ghdl-0.31-mcode-win32\lib\v93\ieee\
```

Alternative way to install on Windows

If you are too lazy to perform the steps described above, there is a quick way to install an older version of GHDL. Go to the following link.

`http://ghdl.free.fr/download.html`

and scroll down to the bottom of the page. Then inside Binaries (Windows), download **ghdl-installer-0.29.1.exe**. After downloading, just run the .exe file and it will do the rest for you.

5.8.2 Installing GHDL on Linux

We describe the installation procedure onto an Ubuntu system. GHDL has been an essential tool for VHDL simulations on Linux systems, especially Ubuntu and its derivatives. But, recently it has been removed from the official Ubuntu repositories and cannot be installed directly. There is an unofficial PPA hosting these packages, and you can use the following commands to install ghdl from there.

1. sudo add-apt-repository ppa:pgavin/ghd
2. sudo apt-get update
3. sudo apt-get install ghdl

Thanks to Peter Gavin for compiling and hosting these packages. You can get more details about the PPA on this link.

`https://launchpad.net/~pgavin/+archive/ubuntu/ghdl`

5.8.3 Using GHDL

If you are trying ghdl for the first time, the following commands would be useful for simulating your VHDL codes.

1. To compile all the VHDL files in a directory

   ```
   ghdl -a *.vhd *.vhdl
   ```

2. After compilation, use the following command to create an executable,

   ```
   ghdl -m top_entity_name
   ```

   This produces an executable called top_entity_name. You can run this executable directly.

3. Finally, to run the simulation and generate the waveforms,

   ```
   ./top_entity_name --stop-time=XXXns --wave=waveform.ghw
   ```

   or

   ```
   ghdl -r top_entity_name --stop-time=XXXns --wave=waveform.ghw
   ```

   The generated waveforms can be viewed using **gtkwave**, which you need to install in your machine.

5.9 Synthesis using Quartus tools

The Quartus synthesis tools provide an integrated synthesis and simulation environment for Altera FPGA's and CPLDS (KRYPTON is an Altera CPLD).

1. Open the Quartus II IDE. A pop-up will appear asking you to either create a new project or open an existing one. Click on *Create New Project*. Alternatively, you may click on *File* $\longrightarrow$ *New Project Wizard*.

2. An Introduction page opens up. Click on *Next*.

3. This opens up Page 1. Here, you need to specify a working directory for your project. Click Browse (...) to create a new folder for this project. Next specify the project name and top level design entity. Important: This is a very critical step in your design. Top level design entity refers to the name of the entity in your VHDL code which you wish to implement. By default the entity name follows the same name as the project. In our case, the top-entity name is PriorityEncoder. After setting the names, click *Next*.

4. This opens up 'Page 2'. Click on *'Empty Project'* and click on *'Next'*.

5. This opens up Page 3. This page allows you to include any existing VHDL program files as part of your project. We add the name of the file in which you have implemented the PriorityEncoder. Click *Next*.

6. On Page 4, you are asked for the family and device settings, i.e. the target CPLD on which you wish to implement your design. Important: This is a very crucial step.. do not select the wrong device!

7. Click on the *Family* drop-down list and select **MAX V**. The device window shows a long list of devices available in the MAX V family.

8. On the right side of the window, you can filter out the device list by selecting *Package* as **TQFP**, *Pin Count* as **144** and *Speed Grade* as **5**. You will see that the list is now quite short. Select the device **5M1270ZT144C5** and click *Next*.

9. In the Page 5 window, you will be asked to select a simulation tool. Again, this is not mandatory and can be skipped. Click *Next*, and on Page 5 you will be shown a project summary. You may use this to review your settings, and can go back to rectify any mistake. Once confirmed, click *Finish*. The project is now created.

10. Go to *Assignments* $\longrightarrow$ *Settings*. A new window will appear. On the left tab, under *EDA Tool Settings*, click on *Simulation*. On the right side click on *Compile test bench* and click on *Test Benches*. A new window will appear. Fill up the boxes as shown in the Figure 5.2. Then under *Test bench and simulation files*, add the name of the file in which you have implemented the test-bench. Click on *OK* and exit the settings.

11. Run *RTL Simulation* and check waveforms in ModelSim (something like Figure 5.3).

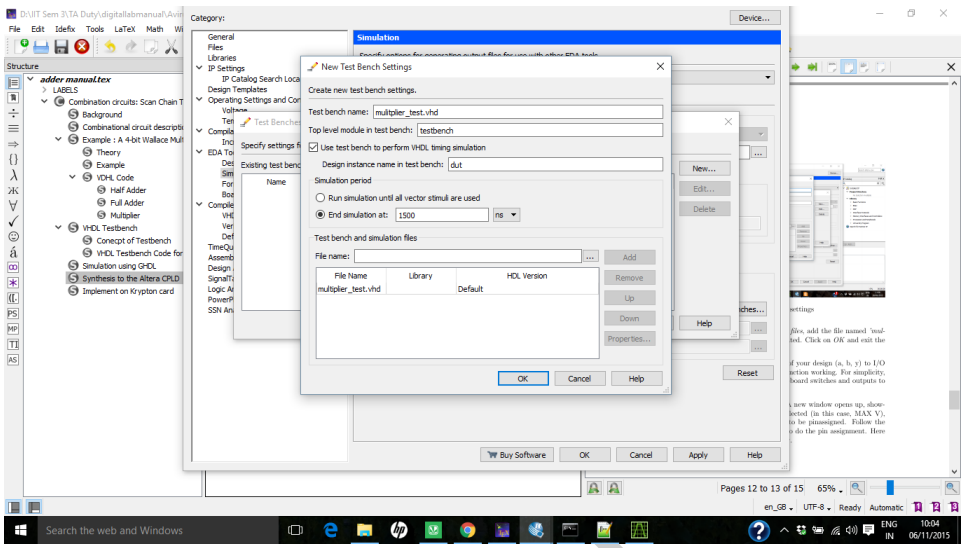

Figure 5.2: Testbench settings

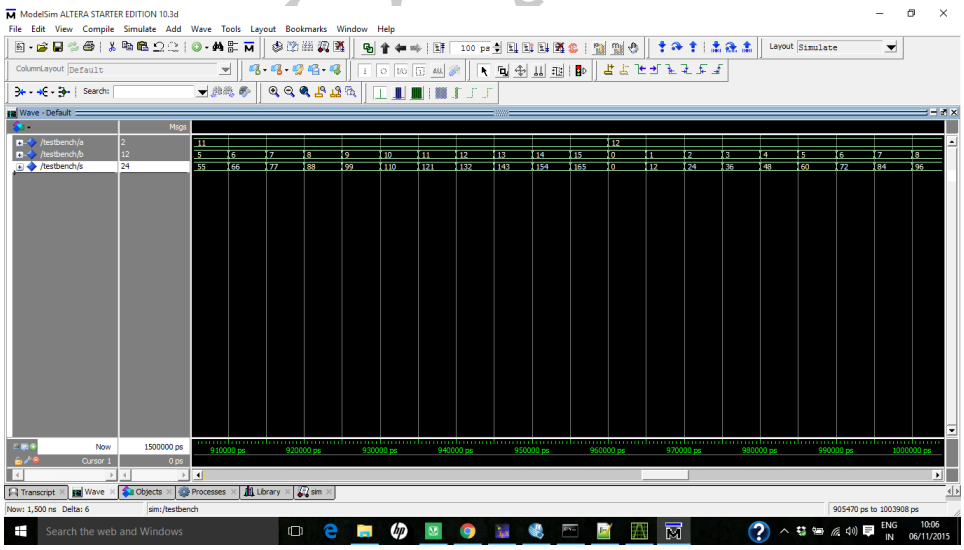

Figure 5.3: RTL Simulation ModelSim waveform

12. To confirm the post-synthesis correctness, run *Gate Level Simulation* from *Tools* $\longrightarrow$ *Run Simulation Tool* $\longrightarrow$ *Gate Level Simulation* and check the waveforms in ModelSim.

13. To finish the implementation, you need to now assign the port pins of your design (a, b, y) to I/O pins on the CPLD to verify the logic function working. For simplicity, we will assign the input lines to the on-board switches and outputs to on-board LEDs using the pin assignments described in Chapter 4.

14. Go to *Assignments* $\longrightarrow$ *Pin Planner*. A new window opens up, showing you the schematic of the device selected (in this case, MAX V), and below, the signal lines that need to be pinassigned. Follow the information given in Krypton manual to do the pin assignment.

15. Once the pin assignment is complete, compile the design. Go to *Processing* $\longrightarrow$ *Start Compilation*. This starts the compilation process, and errors in the code, if any, are shown on the post-compilation report.

16. Now go to *Tools* $\longrightarrow$ *Programmer*. A programmer window will open. You should see the project output file *PriorityEncoder.pof* in this window. If not, you have messed up somewhere. Call for help.

17. In the programmer window, go to *File* $\longrightarrow$ *Generate (JAM, SVF...)*. A new window will open. Select the programming file type as Serial Vector Format (SVF). Browse to the directory where you wish to store the .svf file. The filename need not be the same as the project name. Click on *Generate*. The programming file is now ready.

5.10 Download the SVF file onto Krypton card

Assume that you have your programming file created from Quartus II. Connect the programming cable to the Krypton card and download the SVF file to the card (Follow the instructions in Chapter 4) using the urjtag utility. Once the card is programmed, you are free to play around with it. Set the state of the switches S7-S0 as you wish and observe the results on LED7-LED0. All the best.

Chapter 6

Familiarization: A scan-chain based circuit tester

Testing a complex digital system using switches and LED's is not possible for two reasons: complex systems can have many inputs and outputs, and to test them, one has to apply several test patterns. An automated mechanism for testing complex systems is necessary. Such a mechanism is called a tester. We will describe a simple tester that you can use to test your design on the Krypton CPLD card.

The mechanism uses three steps:

- The design that you wish to test (the DUT) is to be implemented in the Krypton card by connecting a scan-chain to it. The scan-chain is used to send the input pattern to the DUT and to extract the response of the DUT. This is done by instantiating the VHDL description of the DUT with a scan-chain component which will be provided to you.
- A test-pattern file needs to be created on a host computer. The test-pattern file specifies the input patterns and the expected outputs. You can generate the test pattern file using a VHDL simulation of the DUT together with a testbench.
- The host computer needs to be connected to the Krypton card using a microcontroller bridge which allows the host to download patterns to the Krypton card and extract the response from the Krypton card.

In this document we describe, using an example, the three step mechanism outlined above. In Section 6.1, we describe how the scan-chain is to be interfaced to the DUT at the VHDL description level. In Section 6.2,

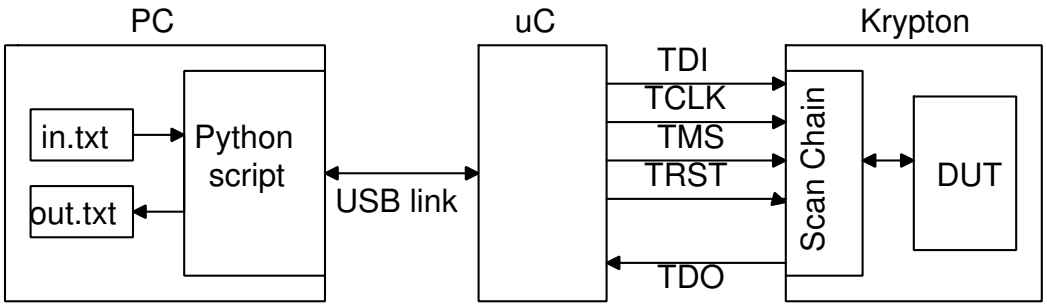

Figure 6.1: Tester Architecture

we describe the format of the test-pattern file. In Section 6.3, we describe the way in which the microcontroller bridge card can be used to connect the host computer to the Krypton card. Finally, in Section 6.4, we illustrate the test procedure from the host computer.

6.1 Scan Chain Insertion

The user has to write a top level entity which uses the DUT and `Scan_Chain` module as components. The top level entity will have only these two components communicating with each other. It will have 5 interface signals (1 bit each) `TDI`, `TMS`, `TCLK`, `TRST` and `TDO`; and should be connected to pins of the CPLD using Pin Assignment feature of Altera-Quartus.

Suppose you wish to test a particular design of addition of two 16-bit vectors. with the following VHDL description.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity DUT is
   port (
      clock, reset: in std_logic;
      A, B: in std_logic_vector(15 downto 0);
      Sum: out std_logic_vector(15 downto 0));
end entity;

architecture Behave of DUT is
  signal sumvector: std_logic_vector(15 downto 0);
```

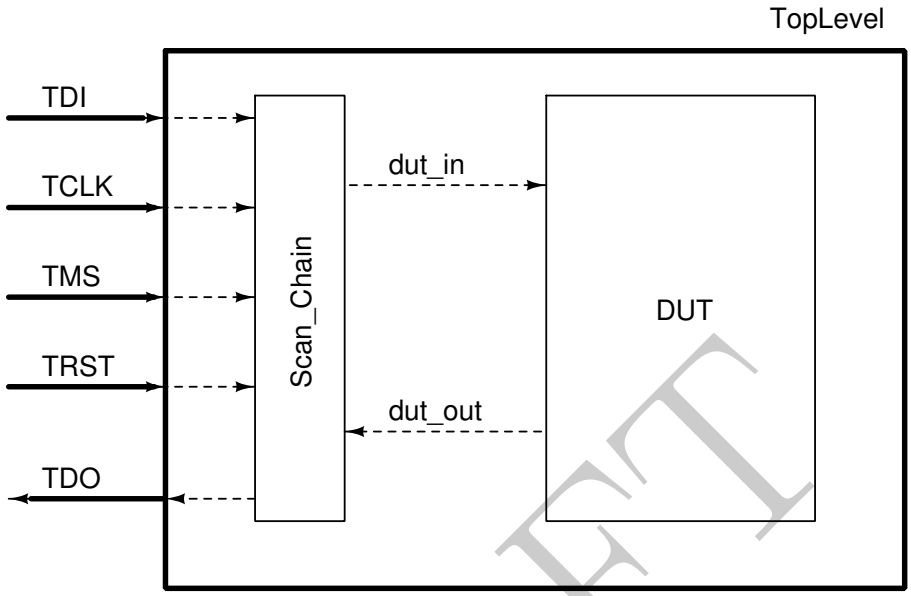

Figure 6.2: Top Level entity to be implemented into the Krypton CPLD


```
begin
  process(clock, reset)
  begin
    if (reset = '1') then
      sumvector <= std_logic_vector(to_unsigned(0, sumvector'length));
    elsif (clock'event and clock = '1') then
      sumvector <= std_logic_vector(unsigned(A) + unsigned(B));
    end if;
  end process;
  Sum <= sumvector;
end Behave;
```

This DUT has 34 inputs and 16 outputs. To test it we will use a scan chain.

The scan chain module has two configurable parameters (`in_pins` & `out_pins`) indicating the number of input and output bits to the DUT, which can be generic mapped. It also has one output (`dut_in`) and one input (`dut_out`) that should be connected to the DUT. Internally it contains an FSM, one input scan register and one output scan register. The entity of the scan chain module is given below:

```
entity Scan_Chain is
  generic (
    in_pins : integer; -- Number of input pins
    out_pins : integer -- Number of output pins
  );
  port (
    TDI : in std_logic;  -- Test Data In
    TDO : out std_logic;  -- Test Data Out
    TMS : in std_logic;  -- TAP controller signal
    TCLK : in std_logic;  -- Test clock
    TRST : in std_logic;  -- Test reset
    dut_in : out std_logic_vector(in_pins-1 downto 0);  -- Input for the DUT
    dut_out : in std_logic_vector(out_pins-1 downto 0);  -- Output from the DUT
  );
end Scan_Chain;
```

The scan-chain and the DUT are instantiated together in a top-level entity as follows

```
entity TopLevel is
  port (
    TDI : in std_logic;  -- Test Data In
    TDO : out std_logic;  -- Test Data Out
    TMS : in std_logic;  -- TAP controller signal
    TCLK : in std_logic;  -- Test clock
    TRST : in std_logic;  -- Test reset
  );
end TopLevel;

architecture Struct of TopLevel is
   -- declare DUT component
   component DUT is
   port (
      clock, reset: in std_logic;
      A, B: in std_logic_vector(15 downto 0);
      Sum: out std_logic_vector(15 downto 0));
   end component;
   -- declare Scan-chain component.
   component Scan_Chain is
   generic (
```

```
      in_pins : integer; -- Number of input pins
      out_pins : integer -- Number of output pins
   );
   port (
      TDI : in std_logic;  -- Test Data In
      TDO : out std_logic;  -- Test Data Out
      TMS : in std_logic;  -- TAP controller signal
      TCLK : in std_logic;  -- Test clock
      TRST : in std_logic;  -- Test reset
      dut_in : out std_logic_vector(in_pins-1 downto 0);  -- Input for the DUT
      dut_out : in std_logic_vector(out_pins-1 downto 0);  -- Output from the DUT
   );
   end component;
   -- declare I/O signals to DUT component
   signal clock, reset: std_logic;
   signal A, B, Sum: std_logic_vector(15 downto 0);
   -- declare signals to Scan-chain component.
   signal scan_chain_parallel_in : std_logic_vector(33 downto 0);
   signal scan_chain_parallel_out: std_logic_vector(15 downto 0);
begin
   scan_instance: Scan_Chain
       generic map(in_pins => 34, out_pins => 16)
       port map (TDI => TDI,
                 TDO => TDO,
                 TMS => TMS,
                 TCLK => TCLK,
                 TRST => TRST,
                 dut_in => scan_chain_parallel_in,
                 dut_out => scan_chain_parallel_out);

   dut_instance: DUT
      port map(clock => clock, reset => reset, A => A, B => B, Sum => Sum);

   -- connections between DUT and Scan_Chain
   reset <= scan_chain_parallel_in(33);
   clock <= scan_chain_parallel_in(32);
   A <= scan_chain_parallel_in(31 downto 16);
   B <= scan_chain_parallel_in(15 downto 0);
   scan_chain_parallel_out <= Sum;
end Struct;
```

6.1.1 Synthesis Project using Quartus

- Open Quartus.
- Go to *File -> Open New Project Wizard*.
- Specify the working directory for the project.
- Write the name of top-level entity in the name of the project. (TopLevel in the present example)
- Click "Next".
- Ensure the Project type is *Empty Project*. Click "Next".
- Add all the VHDL files that you created for this project. The files include all the VHDL files related to your DUT, the TopLevel entity, the Scan Chain entity (*Scan_Chain.vhd*) and the Scan Register entity (*Scan_Reg.vhd*). The Scan Chain and Scan Register VHDL files are provided to you. Click "Next".
- In the Family & Device Settings, select Family -> MAX V, Package -> TQFP, Pin count -> 144, Core Speed Grade -> 5, and select device 5M1270ZT144C5. Click "Next" twice & "Finish".
- Now compile the design by pressing *Ctrl+L*.

After compilation, if there are no errors observed, we will do the pin assignment by going to the menu *Assignments -> Pin Planner*. The Pin Planner will open in a new window. You will see the port names of the top-level entity already specified in the Node Name column. Fill the values in the location column as given in Table 6.1.

Node Name	Location
TDO	PIN_3
TDI	PIN_5
TMS	PIN_7
TRST	PIN_21
TCLK	PIN_23

Table 6.1: Pin Planning

Compile the design again (*Press Ctrl+L*).

Go to *Tools -> Programmer*. A new window opens up and you should see the name of your top-level entity (*TopLevel.pof*). If you don't, go to "*Add File*" in the left panel, select the *.pof* file (*TopLevel.pof*). You can find it in the "Output Files".

To create a flashable file, go to *File -> Create JAM, JBC, SVF or ISC file* menu. Select *.svf* format from the *File Format* drop down menu. Do not change any other parameters and click "OK". You should see a *.svf* file generated in your project folder.

Flash the *.svf* file into CPLD using UrJTAG. Follow *UrJTAG User Manual* [2].

6.2 Test Input Format

The text file to be passed to the python script for test execution has to be written in following format. Specify only the values written in italics.

- SDR *inpins* TDI(*input*) *outpins* TDO(*output*) MASK(*maskbits*)

The *inpins* & *outpins* contain the number of input & output pins of the DUT, respectively, in **decimal** format. The attributes *(input)* & *(output)* contain the input vector to be applied and its expected output vector, respectively, in **hex** notation. The maskbits are used to specify if any of the output bits of the CPLD need to be compared with the expected output. A bit value '*1*' signifies that comparison is required. The mask bits are also specified in **hex** notation.

This instruction loads the input into the Scan Chain register inside the CPLD. Also, if any of the mask bits are set, it reads data from the Scan Chain register.

- RUNTEST *delay* msec

As the previous instruction loads the input and samples the output, this instruction is used to apply the input combination to the DUT and run the test for *delay* msec.

For example, the input text file for the design described in Section 6.1, and the waveform in Fig.6.3 will be written as follows:

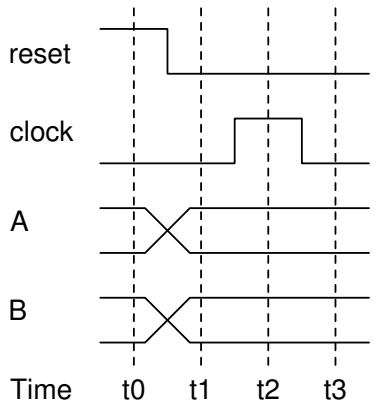

Figure 6.3: Example Waveform

```
# @t0
SDR 34 TDI(200000000) 16 TDO(0000) MASK(0000)
RUNTEST 1 MSEC
# @t1
SDR 34 TDI(051267A5C) 16 TDO(0000) MASK(0000)
RUNTEST 1 MSEC
# @t2
SDR 34 TDI(151267A5C) 16 TDO(0000) MASK(0000)
RUNTEST 1 MSEC
# @t3
SDR 34 TDI(051267A5C) 16 TDO(CB82) MASK(FFFF)
RUNTEST 1 MSEC
```

Note that you can use '#' for a single-line comment.

6.3 Hardware connections

The physical connections among the host PC, microcontroller board and the user module(CPLD) are specified as follows. The host PC is connected to the microcontroller board through USB cable. Use 8-Pin Female Connector for connecting microcontroller board to CPLD.

Two different microcontroller boards are available, *Tiva-C* and *Ptx-128*. You can use any one of the two.

6.3.1 Using Tiva-C microcontroller

Host-PC with Tiva-C

Tiva-C has 2 ports for USB connection. Make sure you use the *Device Port* as encircled in Fig.6.4.

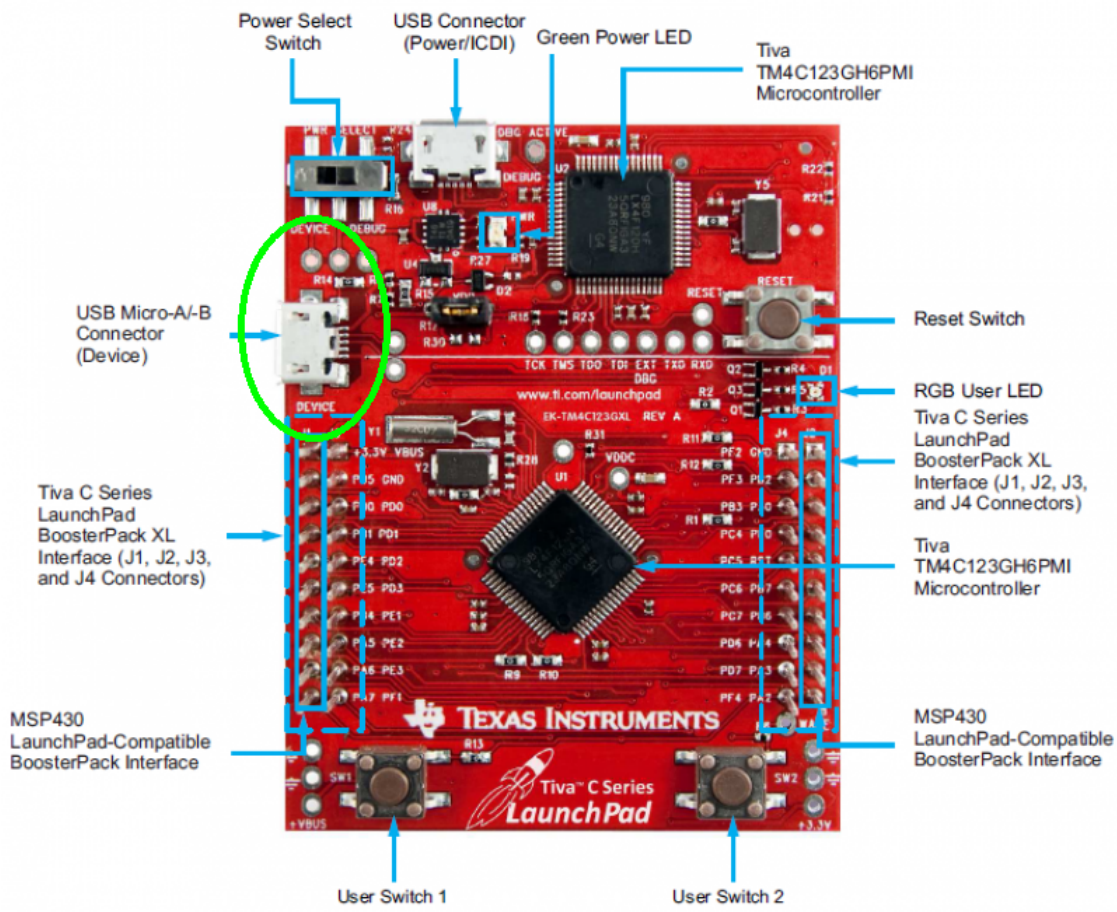

Figure 6.4: Tiva-C Evaluation Board (taken from [1])

CPLD with Tiva-C

Make the connections as shown in Table 6.2 and Fig.6.5.

CPLD Pin	Tiva-C Pin	Function
Header 0: 3	J1:PB5	TDO
Header 0: 5	J1:PB0	TDI
Header 0: 7	J1:PB1	TMS
Header 0: 13	J1:PB4	TRST
Header 0: 15	J1:PA5	TCLK
Header 0: 25	J2:GND	GND

Table 6.2: Hardware Connections

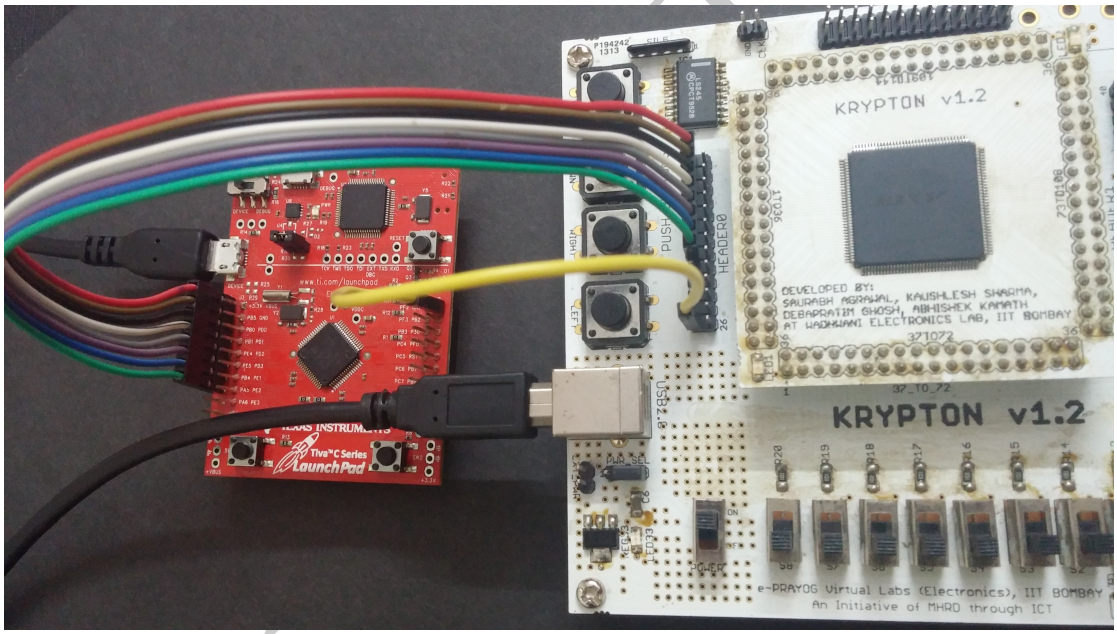

Figure 6.5: Hardware Connection between CPLD and Tiva-C using 8-PIN Female Connector

6.3.2 Using Ptx-128 microcontroller

Host-PC with Ptx-128

Use USB Cable for connection.

CPLD Pin	Ptx-128 Pin	Function
Header 0: 3	PORTC:PC0	TDO
Header 0: 5	PORTD:PD0	TDI
Header 0: 7	PORTD:PD1	TMS
Header 0: 13	PORTD:PD4	TRST
Header 0: 15	PORTD:PD5	TCLK
Header 0: 25	POWER_CONNECTOR:GND	GND

Table 6.3: Hardware Connections

CPLD with Ptx-128

Make the connections as shown in Table 6.3 and Fig. 6.6.

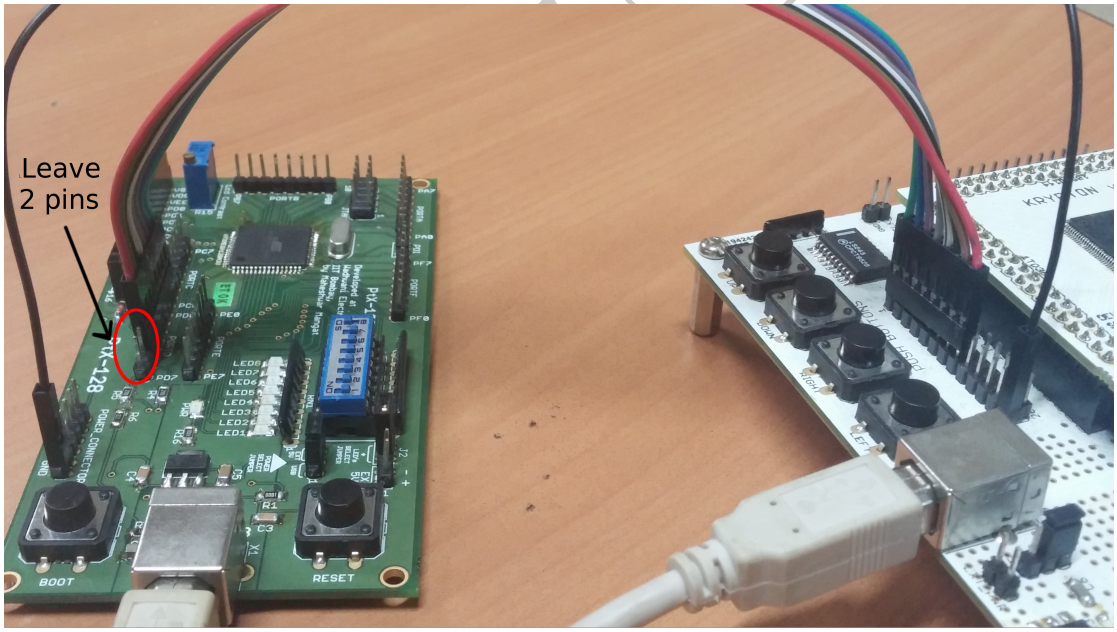

Figure 6.6: Hardware Connection between CPLD and Ptx-128 using 8-PIN Female Connector

6.4 Running the test

6.4.1 Installation

1. goto→ http://sourceforge.net/projects/pyusb/ or google Pyusb
2. Download 1.0.0 version and extract it to any folder
3. Open folder and you will see docs,usb,setup.py etc. file and folder.
4. Open terminal, goto the present directory and run following two commands:
   (a) Install Python and other dependencies:
   sudo apt-get install python libusb-1.0.0
   (b) Install PyUSB:
   sudo python setup.py install

6.4.2 Running the Python script

After the installation run the `scan.py` script with the following command.

For Tiva-C : `sudo python scan.py` *`input_file output_file`* *tiva*

For Ptx-128: `sudo python scan.py` *`input_file output_file`* *ptx*

Where the *input_file* should contain all the commands to be executed as described in Section 6.2, *output_file* should be an empty file for storing the results back.

6.4.3 Seeing Results

The *output_file* lists the results under the following headers

```
Expected Output    Received Output   Remarks
==========================================
```

The first column under "Expected Output" lists the outputs that are expected from the CPLD, the second column under "Received Output" lists the bits received from the CPLD, and the third column under "Remarks" specifies the result. If the Expected Output matches with the Received Output, the Remark is "Success", and if not, the Remark is "Failure".

For example, the correct result of the example input text file described in Section 6.2, and the waveform in Fig.6.3 will be as follows:

Expected Output	Received Output	Remarks
1100101110000010	1100101110000010	Success

Bibliography

[1] Texas Instruments, "Tiva-C Series TM4C123G LaunchPad Evaluation Board User's Guide", Texas Instruments, 2013.

[2] Debapratim Ghosh, "Using Quartus II and UrJTAG for Krypton", Wadhwani Electronics Laboratory, 2013.

[3] Titto Thomas, "Scan chain based testing mechanism for digital systems", Wadhwani Electronics Laboratory, 2015.

Chapter 7

Experiment: Implement an 8-Bit ALU using the KRYPTON CPLD card

Implement an 8-bit ALU with the following specifications. There are two 8-bit inputs X and Y, and it produces an 8-bit output Z. The operation to be performed is selected by a 2-bit operation code op_code. The functionality of the ALU based on the op_code bits is shown in Table 7.1. The numbers X, Y, Z are to be viewed as integers represented in 2-s complement form. The shifts are logical, so that a right shift does not do carry extension.

You will implement the 8-bit ALU in a systematic manner (as outlined in Chapter 5.

1. Logic design: Logic design will be done using two-input AND, OR and NOT gates only.

2. VHDL description of the ALU.

3. Testbench design and implementation. All possible input combina-

Op Code (op_code)	**Operation**	**Operands**	**Result**
00	Addition	X,Y	Z=X+Y
01	Subtraction	X,Y	Z=X-Y
10	Logical Right Shift	X,Y	Z=X$\gg$Y
11	Logical Left Shift	X,Y	Z=X$\ll$Y

Table 7.1: Op-codes for the ALU

tions are to be checked.

4. From the testbench, you will need to generate a trace file to feed to the scan-chain based tester in order to test your design (the format of the file is described in Chapter 6).

5. Simulation of the testbench with the ALU to check correctness.

6. Synthesize the ALU and simulate the post-synthesis logic network to confirm correctness.

At this point, you will diverge from the path taken in Chapter 5, and attempt to verify the hardware implementation of your ALU by using the scan-chain based tester described in Chapter 6.

1. Wrap your ALU entity by integrating it with the scan-chain entity as described in Chapter 6.

2. Synthesize the wrapped entity using the ALTERA Quartus tools (pin mapping as described in Chapter 6).

3. Program the KRYPTON CPLD using the program file generated by synthesis.

4. Connect the scan-chain test setup as described in Chapter 6.

5. Run the tester script with the reference trace file that you have generated earlier (use the correct script options depending on the type of micro-controller card that you are using).

6. Summarize the results.

Chapter 8

Experiment: Implement a simple string recognizer using the KRYPTON CPLD card

Implement a simple string recognizer which is given a sequence of alphabets as an input. The input and output alphabet consists of $\{a, b, c, d, \ldots, z\}$ (ie, the 26 lower-case letters in the English alphabet) together with the *blank-space* character. The behaviour of the string recognizer is as follows:

- If at any point k, the sequence of letters seen thus far contains one of the following sub-sequences, then the machine outputs the letter "y". Otherwise, it outputs the letter "n".

  ```
  bomb
  gun
  knife
  terror
  ```

  For example: suppose the input string is

```
bring the oil to my building
```

This string contains the subsequence "bomb"

```
bring the oil to my building
^          ^       ^  ^
```

Thus, the input and output sequences when lined up will look like:

```
bring the oil to my building
nnnnnnnnnnnnnnnnnnnnynnnnnnn
```

Design a Mealy machine for implementing this string recognizer.

1. First of all, note that you can do this by a parallel combination of simpler Mealy machines, each of which recognizes one of the four simple substrings specified.

2. Identify a suitable set of states for implementing the state machine(s). Note that in each machine, one of the states must be the reset state.

3. Write the next state and output functions in a table with the following format:

   ```
   present-state  input-symbol  next-state output-symbol
   ```

To implement the Mealy machine in VHDL (for eventual implementation on the KRYPTON CPLD card), we will encode the input and output alphabet. the 26 letters and blank-space are coded by using 5 bits X4,X3,X2,X1,X0 as follows:

```
letter   X4 X3 X2 X1 X0
a        0   0  0  0  1
b        0   0  0  1  0
c        0   0  0  1  1
d        0   0  1  0  0
..
w        1   0  1  1  1
x        1   1  0  0  0
y        1   1  0  0  1
z        1   1  0  1  0
space    1   1  0  1  1
```

The output symbols n, y can be coded with a single bit W as

```
symbol  W
n       0
y       1
```

In addition, the string recognizer has a reset and clock as is usual. Thus, the top-level entity for your string-recognizer could look like this

```
entity StringRecognizer
  port (X4,X3,X2,X1,X0: in std_logic; W: out std_logic;
        clk,reset: in std_logic);
end entity StringRecognizer;
```

The architecture of each FSM in your string recognizer will consist of a single process statement which implements the Mealy machine that you have designed. It is a direct translation of the state transition table to VHDL.

```
-- signals declared in architecture..
type FsmState is (S1,S2,...,SK);  -- list of states
signal Q: FsmState; -- Q need not be encoded into bits
------

process(Q, ... input-variables ...,clk,reset)
  variable nQ: FsmState;
  variable ... output-variables ... : std_logic;
begin
  -- default values.
  nQ := Q;
  ... initialize-output-variables ...

  -- calculate nQ, output variables
  case Q is
     when S1 =>
        -- nQ,output-variables
        -- depending on X's,Q.
     when S2 =>
        -- nQ,output-variables
        -- depending on X's,Q.
     ...
     when SK =>
        -- nQ,output-variables
        -- depending on X's,Q.
  end case;

  -- machine outputs.
  output-signals <= output-variables;


  -- next-state to state.
```

```
    if (clk'event and clk = '1') then
       if(reset = '1') then
          Q <= S1;  -- S1 is reset state
       else
          nQ <= Q;
       end if;
    end if;
  end process;

  -- rest of architecture....
```

You will have to write a test-bench for this design. Try the following sequence to the implementation to begin with:

```
he was in born in mumbai i knew that i felt good inspite of errors
```

This pattern hits all four key words:

```
he was in born in mumbai he knew that it felt good unite of errors
          ^^      ^  ^      ^^        ^  ^^   ^    ^^ ^^     ^^^^
          bo      m  b      kn        i  fe   g    un te     rror
```

The expected response (lined up with the input string) is:

```
he was in born in mumbai he knew that it felt good unite of errors
nnnnnnnnnnnnnnnnnnnnnynnnnnnnnnnnnnnnnnnnynnnnnnnnnynnnnnnnnnnnyn
```

8.1 Laboratory assignment

Complete the implementation of the Mealy FSM by going through the following steps (in your laboratory report, you should give a summary of the results obtained at each step):

1. If you use a parallel decomposition (you should), then for each constituent FSM:
   - Determine the number of states needed to implement the FSM, and design the state transition graph (next-state, output tables).
   - Describe the FSM in VHDL.
2. Assemble the individual process statements for the FSM's in a single entity architecture description.

3. Write a test-bench which reads a trace file and checks that the machine produces the expected outputs.

4. Simulate the test-bench with the entity/architecture and confirm that your implementation is functional. Use the test-bench to generate a trace file to be used with the scan-based tester.

   - IMPORTANT: clock and reset are also inputs to the system. The rising edge of the clock should be used to implement the delay element. Your trace file should ensure that when data is changed, clock is **not rising** (why?). Thus, your trace file should look like

```
clk   input-data   expected-output-data
0      x(0)           y(0)
1      x(0)           --
0      x(1)           y(1)
1      x(1)           --
..
0      x(k)           y(k)
1      x(k)           --
0      x(k+1)         y(k+1)
... etc. ...
```

5. If the verification of your design is successful, proceed to synthesize your design and simulate the gate-level netlist to confirm correctness of the hardware implementation.

6. Now, integrate your entity with the scan-based tester, and synthesize the integrated description (produce a programming file for the CPLD).

7. Download the program file to the Krypton card, and verify the implementation using the scan-based tester.

Chapter 9

Experiment: Implement a greatest-common-divisor circuit using the KRYPTON CPLD card

You will implement a greatest-common-divisor (GCD) computation circuit in two steps. First implement a divider and then use the divider to implement a GCD circuit.

9.1 An unsigned divider

Design an unsigned integer divider with the following interface

```
entity unsigned_divider is
    port(clk: in bit;
       reset: in bit;
       -- the two inputs
       dividend: in bit_vector(15 downto 0);
       divisor : in bit_vector(15 downto 0);
       -- the next two implement a ready-ready
       -- protocol to start the division
       inputs_ready: in bit;
       divider_ready : out bit;
       -- the two outputs
       quotient  : out bit_vector(15 downto 0);
```

```
        remainder : out bit_vector(15 downto 0);
        -- the output ready-ready handshake
        output_ready: out bit;
        output_accept: in bit;
        );
end entity unsigned_divider;
```

The input side information is exchanged using the inputs_ready/divider_ready pair, and the output side information is exchanged using the output_ready/output_accept pair.

The entire divider is to be constructed using only the following building blocks:

- FSM.
- Registers and D-flipflops.
- Multiplexors.
- AND, OR, NOT gates (equations are OK).

You are NOT ALLOWED to use any of the ieee packages or functions present in any of the ieee packages.

Implement the divider using the following steps:

- Choose the division algorithm?
- Write an RTL description of the system. First declare the registers, then describe the RTL algorithm (use the pseudo-code introduced in class) and verify that the description implements the algorithm.
- Infer a data-path and a control-path.
- Implement the control-path using an FSM.
- Implement the data-path by first implementing the data-path operators that you need and then hooking up the multiplexors, registers and operators.
- Verify (with a test-bench) using at least 16 random divisor-dividend pairs.
- Using the Quartus tools, synthesize the divider and simulate to check that the synthesized circuit works.

9.2 Use the divider to implement a GCD computation circuit

Use the divider to implement a circuit which accepts eight 16-bit numbers and outputs the greatest-common-divisor of these numbers. You can use the Euclidean algorithm to find the GCD of two numbers as the basis for the circuit. You will perform the GCD operation by using the observation that

$$gcd(a, b, c) = gcd(gcd(a, b), c)$$

so that you need not read in all the numbers in order to start the computation.

The entity description of the system is

```
entity System is
   port ( din: in bit_vector(15 downto 0);
          dout: out bit_vector(15 downto 0);
          start: in bit;
          done: out bit;
          erdy: in bit;
          srdy: out bit;
          clk: in bit;
          reset: in bit);
end entity;
```

The input data is accepted using the erdy/srdy pair. The system starts when the start bit is asserted, and indicates completion by asserting the done bit. When the done bit is asserted, the information at dout must be valid data.

Follow the same steps as in the previous question, except that for verification, do so with at least 16 distinct sets of eight numbers.

9.3 Check the GCD circuit on the KRYPTON CPLD board

Now that your GCD circuit works in simulation, put it into the Krypton CPLD card (after integrating it with the scan-chain logic) and confirm that it works.

Chapter 10

Experiment: RTL with SRAM

Chapter 11

Introduction to the use of a logic analyzer

A Logic analyzer is an instrument used to view multiple digital waveforms. The key features which distinguish it from a digital storage oscilloscope (DSO) are

- the availability of large number of input channels (32/63/128).
- advanced triggering capabilities based on input conditions, history etc.
- analysis of captured waveforms using high-resolution timing, sample & hold analysis etc.

A logic analyzer is not to be used for analog measurements since it does not have the required resolution.

11.1 The TLA5202B logic analyzer

We will describe the Tektronix TLA 5202B Logic Analyzer (most of the material here is based on Tektronix Logic Analyzer User Guide). A representation of its front panel is shown in Figure 11.1.

This logic analyzer has the following features:

- 68 channels
- 2 GHz, 256 Mb deep timing analysis
- 125 ps MagniVu high resolution timing acquisition

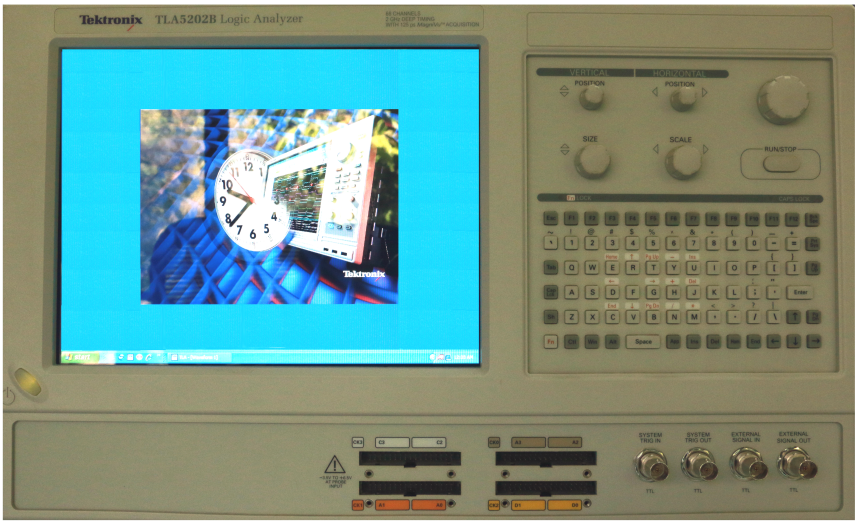

Figure 11.1: TLA 5202B Logic Analyzer Front Panel

Figure 11.2: Logic Analyzer Probes

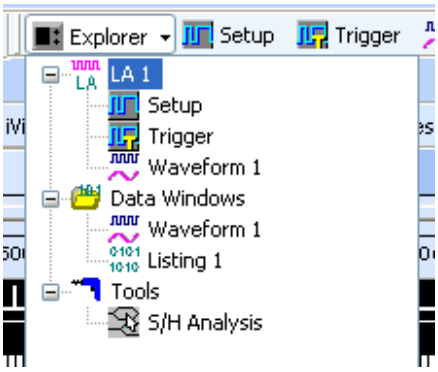

Figure 11.3: TLA explorer

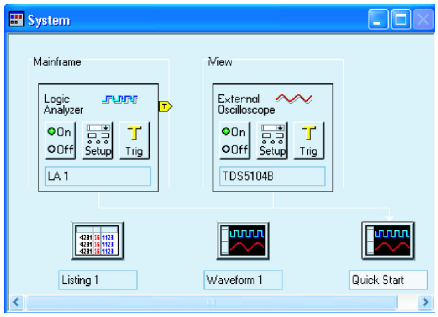

Figure 11.4: System window

Figure 11.5: Toolbar

- Connectorless probing system

Some applications of the logic analyzer include

- Digital hardware verification and debug
- Monitoring and measurement of digital hardware performance
- Single microprocessor or bus debug

11.2 Logic analyzer windows

TLA Explorer: It allows you to access the key windows from a tree structure
System window: It shows the block diagram representation of the modules and the available data windows
Toolbar buttons: These can be used to navigate between the windows quickly

11.3 Example circuit

We will be using a 4-bit synchronous counter with an *enable* (called as *COUNT* in the manual) and *clear* for explaining the features of the logic

analyzer. VHDL code of the above circuit is given in the Appendix 11.5. The probes of the logic analyzer are connected as shown below [1]

- CLR: A1(1)[2]
- COUNT: A1(2)
- Q(0): A1(3)
- Q(1): A1(4)
- Q(2): A1(5)
- Q(3): A1(6)

11.4 Basic steps for using a logic analyzer

- Configure the setup window to set up the logic analyzer channels, threshold voltages, clocking etc
- Configure the waveform window
- Acquire data
- Try out different triggering conditions

11.4.1 Configure the setup window

1. Open the setup window from the toolbar buttons (Figure 11.5).
2. Rename probes (Figure 11.6) using using the information given in the above section (Figure 11.3).
3. Group the four output bits of the counter (Figure 11.7) and create the bus COUNT_OUT (Figure 11.8).

[1] We will be using the probe *A1* of the logic analyzer

[2] The pin corresponding to the CLR signal of the adder has to be connected to the probe A1(1) of logic analyzer

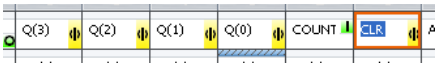

Figure 11.6: Assigning names to the probes

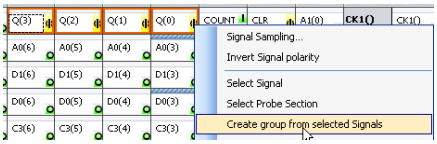

Figure 11.7: Grouping the signals

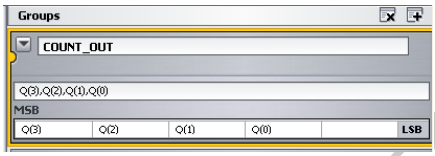

Figure 11.8: Assigning names to the group of signals

11.4.2 Configure the waveform window

- Open the setup window from the toolbar buttons (Figure 11.5).
- Right-click in the waveform label area and select Add Waveform (Figure 11.9).
- Select the signals related to the counter (Figure 11.10) and add it to the wave window
- Select the output signal group (COUNT_OUT) that we had created earlier
- Click on the *Value* button (Figure 11.11) to display the values of signals at the position of the markers (Figure 11.12).

11.4.3 Acquire data

- Enable *continuous data acquisiton* by clicking on the *Repetitive* option from the *System* dropdown menu (Figure 11.13).
- Click on the run button (Figure 11.14) to start data acquisition. Now you should be able to see the counter output waveforms (Figure 11.15).

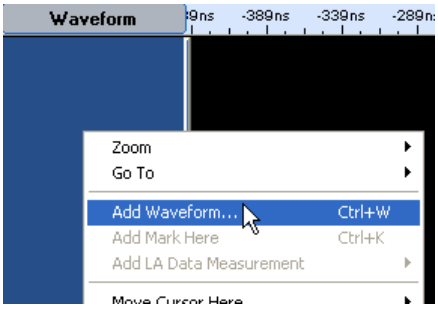

Figure 11.9: Add waveforms

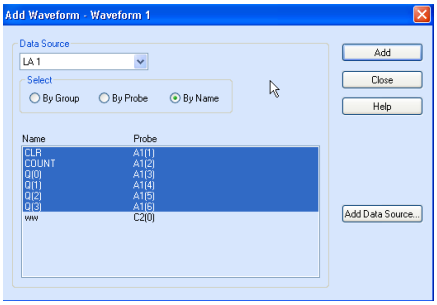

Figure 11.10: Selecting signals to be displayed

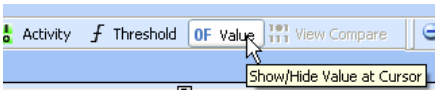

Figure 11.11: Value button

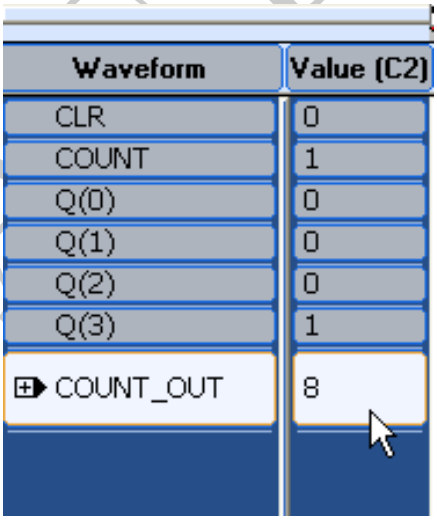

Figure 11.12: Displaying the value of signals at the position of the marker

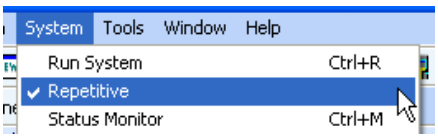

Figure 11.13: Enabling repetitive run

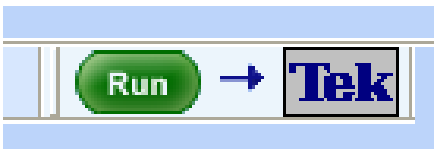

Figure 11.14: The Run button

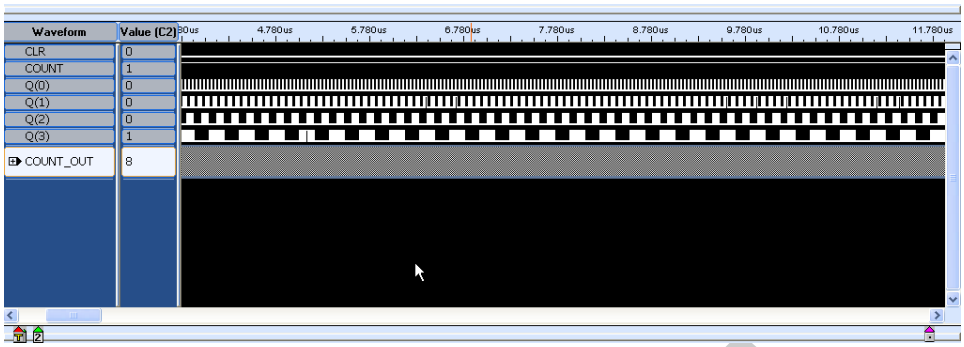

Figure 11.15: Counter signals displayed in the Waveform window

11.4.4 Try out different triggering conditions

- Trigger immediately (default)[3]
- Trigger on channel transition
- Trigger on a sequence of events

Trigger immediately

This is the default triggering condition (Figure 11.16). The signals will be captured if any of the signal values changes. We were using this triggering until now.

Trigger on channel transition

Here we will set the triggering condition as the rising edge of the signal COUNT (Figure 11.18). You will observe that the wave window will get updated only when the COUNT is going from LOW to HIGH

- Stop the data acquisition process[4] by clicking on the *STOP* button (Figure 11.17).
- Go to the trigger window
- Click on the *Trigger on channel transition (edge)*

[3] We are discussing only 3 out of many possible triggering conditions which can be set in the logic analyzer

[4] Only if its running.

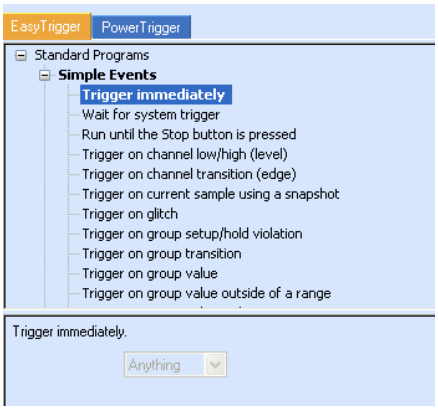

Figure 11.16: Trigger immediately

Figure 11.17: The Stop button

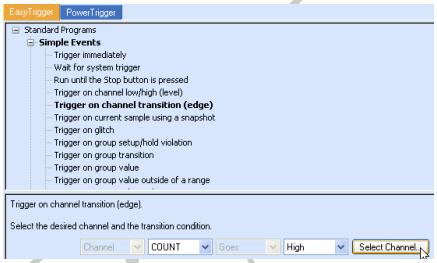

Figure 11.18: Trigger on the rising edge of COUNT

- Select the channel as *COUNT* and the level as *High* as shown in Figure 11.18.

Trigger on a sequence of events

Here the triggering will happen only on a sequence of events set by the user. Here we will set the condition *(COUNT=1) & (CLR=0)* as the trigger condition. This can be done using the following steps

- Go to the trigger window and click on the *Power Trigger* tab
- Click on the box named *If/else* in the *State 1* panel (Figure 11.19) and set the options as shown in Figure 11.20.
- Click on the box named *If/else* in the *State 2* panel (Figure 11.21) and set the options as shown in Figure11.22.

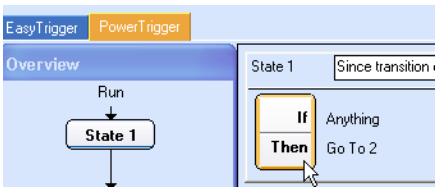

Figure 11.19: State 1 button of Power Trigger

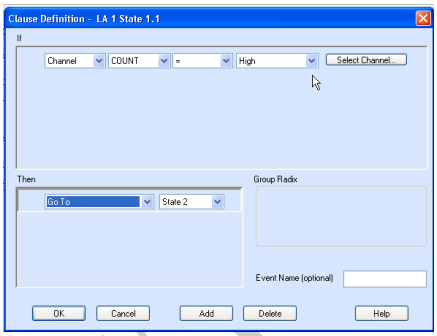

Figure 11.20: Trigger condition for state 1

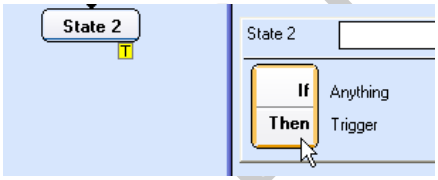

Figure 11.21: State 2 button of Power Trigger

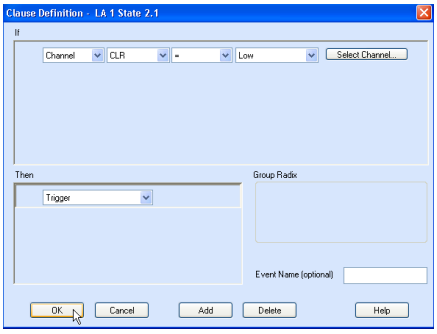

Figure 11.22: Trigger condition for state 2

11.5 Appendix

The VHDL code of the 4-bit counter referred in this document is given below

```
-- 4-bit adder
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity counter is
  generic(n: natural :=4);
  port(   clock:  in std_logic;
          clear:  in std_logic;
          count:  in std_logic;
          Q:  out std_logic_vector(n-1 downto 0)
          );
end counter;
architecture behv of counter is
  signal Pre_Q: std_logic_vector(n-1 downto 0);
begin
  process(clock, count, clear)
  begin
    if clear = '1' then
      Pre_Q <= Pre_Q - Pre_Q;
    elsif (clock='1' and clock'event) then
      if count = '1' then
        Pre_Q <= Pre_Q + 1;
      end if;
    end if;
  end process;
  Q <= Pre_Q;
end behv;
```

chapter–11/manual/code_snippets/counter.vhdl

Bibliography

[1] http://www.tek.com/datasheet/tla5000b-series (Accessed: 17-11-2015)

[2] Photo credits: Anil R. Gawai, WEL, IIT Bombay