E BLOCKS2

Field-Programmable Gate Array (FPGA)

CP3811

MATRIX

FPGA Solution

Course notes

Contents

Chapter 1: About this Course

Introduction to this package

This course contains a complete guide to using Quartus and to developing designs in block diagram format, in VHDL and in Verilog.

Our intention is that, with just a basic understanding of digital electronics, you will be able to teach yourself to program FPGAs in block diagram mode or using a high-level descriptive language.

This assumes a basic understanding of combinational and sequential logic. Note that Quartus itself is primarily a design tool for CPLD/FPGA technology, and is not a great package for learning Digital Electronics.

To help you understand the transition between conventional digital logic simulation and design in VHDL and Verilog, the course includes a number of designs for the Proteus circuit simulator.

How to use this course

This is a self-study resource. Whilst a teacher may introduce each topic, the intention that students use this resource to teach themselves.

The remit is to enable you to:

    use Quartus as a programming tool for FPGA technology

    design complex digital systems using block diagrams in the Quartus environment

    program using the VHDL language

    program using the Verilog language

In working through this course, use the left hand menu to work through the topics in the order they are presented.

The path the course follows is:

    setting up Quartus

    using Quartus for simple designs

    using advanced features of Quartus

    introducing Very High Level Descriptive Languages (VHDL)

    introducing behavioural VHDL

    exploring combinational logic using VHDL

    exploring sequential logic using VHDL

We strongly recommend that you learn either Verilog or VHDL, even though it is possible to use block diagrams for most of this work.

Example projects

Within the course, there are a number of sample projects, using Block Diagram Files (bdf), Verilog and VHDL Code files etc.

The example files are in the resource examples folder, you should copy these to your own working directory.

Network users
Users on a network, or restricted users on a standalone computer, may not have full access to the default Project files location. In this case, it is recommended that a copy of the Project files folder is available for each user, in their "My Documents" folder.

What you need

To get started you will need the following:

a computer with Windows operating system;

an BL0580 FPGA Solution;

an internet connection;

the FPGA course resource (zip) from the MatrixTLS.com website

You will also need access to a conventional circuit simulator such as Tina, Multisim, or Proteus.

FPGA solution

Great for teaching modern digital system design

The BL0580 FPGA Solution featured in this course includes a range of E-blocks2 boards and accessories:

E-Blocks:

BL0167 LED board

BL0145 Switch board

BL0138 Keypad board

BL0114 Combo board

BL0093 FPGA programmer / development board.

Accessories:

HP2666 Power supply

terasIC blaster USB programmer

The importance of ALT – TAB and printing

This course has been developed for use on-screen rather than in a document. The advantage of this approach is that it allows each topic and sub-topic to be presented to students in small manageable chunks - an approach that would result in a lot of waste paper in a conventional paper manual.

This approach also allows the use of hyperlinks to gain direct access to files, while the full course structure is visible at all times. When discussing program listings, this approach allows the program to be presented on one side of the screen whilst the text on the other side can be scrolled up and down. However some sections of the course are best printed out and followed on paper.

If you hold the ALT key down and press the TAB key at the same time, you toggle between the applications that are currently open on your computer desktop. This is useful when working through instructions in Quartus.

Chapter 2: About PLD technology

Introduction

In this section you will learn about:

- The relationship between PLA's/ CPLDs / FPGA's;
- The main providers of PLA's/ CPLDs / FPGA's ;
- The FPGA we use and its characteristics;
- The importance of PLA's/ CPLDs / FPGA's.

The importance in the electronics industry, and hence in education, of programmable logic devices (PLDs) and other reconfigurable logic devices is not immediately apparent.

In most instances, a microcontroller could perform the tasks traditionally carried out by a PLD. However the advent of FPGA technology has brought a new dimension to this argument.

This document explains the architecture of programmable logic devices, how they have evolved, and why these devices will become more important over the next few years.

In the beginning...

PLDs (Programmable Logic Devices) stem from the original technology used to make Programmable Read Only Memories which was developed at the start of the computing boom. A variation of this technology led to the creation of Programmable Array Logic devices (PAL), which is where we start this discussion.

PAL s are simple logic blocks that provide a configurable combinational logic block. This does not seem that useful but when you consider that a single PAL might have replaced several TTL chips you can understand why their use was adopted all those years ago.

PAL architecture

To understand how this works lets look at a simple diagram, shown below.

This represents the internal workings of a simple PAL with two inputs and 3 outputs. The terms A and B are inverted and buffered and made available to a series of AND gates with fusible inputs. The outputs are fed in turn to fixed OR gates and then to the appropriate outputs.

The mechanisms and constructions in silicon inside the device that allow this to take place are beyond this discussion. Here we are interested only in how the system works.

This architecture, and a suitable system to lets us alter the state of the fuses, allows the outputs X, Y and Z to be any combinational logic variation of the two inputs A and B.
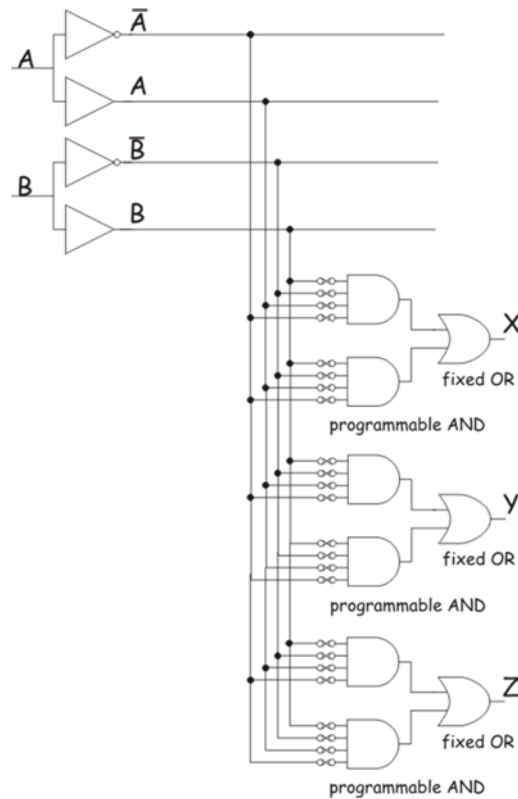
Fig 2.1: PAL architecture.

PAL Example

Assume we want the functions:

    X = (A!.B!) +(A.B!)
    Y = (A.B!) +(A.B)
    Z = (A!.B)

(Here we use the '!' notation to denote inversion - it's just easier to type than a bar over the top of the letter.)

This new diagram would show you how the fuses would be orientated in the PAL.

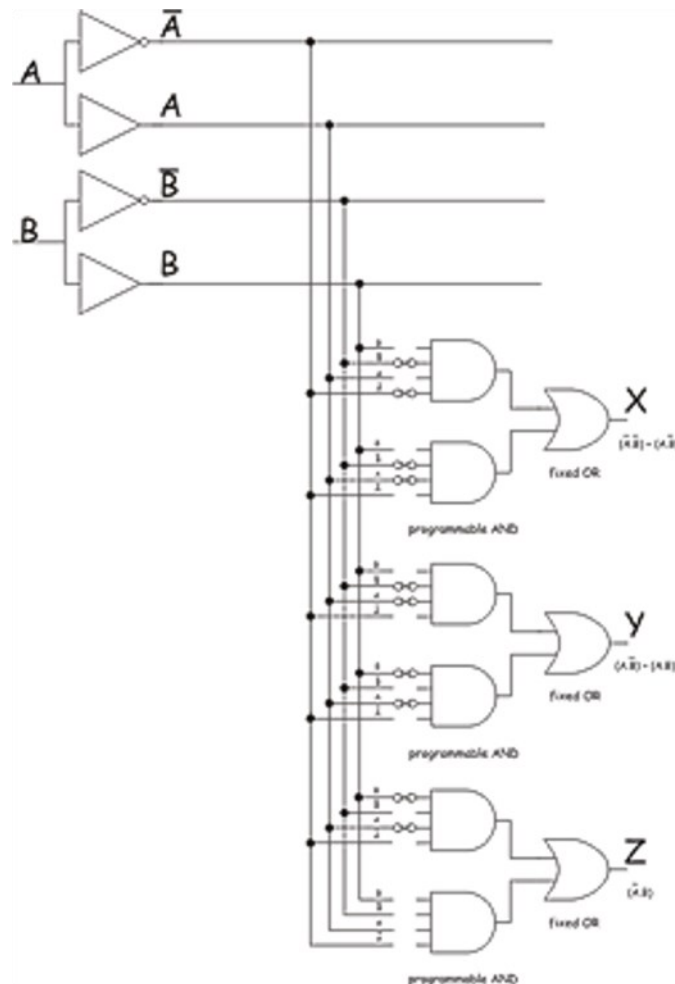Note that the gate inputs float high when the fuse is blown.

Fig 2.2: PAL Example

PALs and PLAs

A Programmable Logic Array device differs slightly from a PAL in that both the AND terms and the OR terms are programmable. This results in a slightly more flexible device than a PAL but a device that is slightly more costly and complicated to program since more fuses need blowing. We won't look at the detail here!

And so to PLDs

The addition of a single bit register in each output path allows the construction of state machines (in particular counters and shift registers). The resulting devices are very flexible and are generally referred to as Programmable Logic Devices or PLDs. The flexibility is greatly increased by feeding the outputs of the register bits back into the array itself allowing the construction of fairly complex designs.

To give increased flexibility, both the AND and OR sections of the circuit are programmable. The real circuits get rather complicated at this time but the following schematic shows the overall architecture.
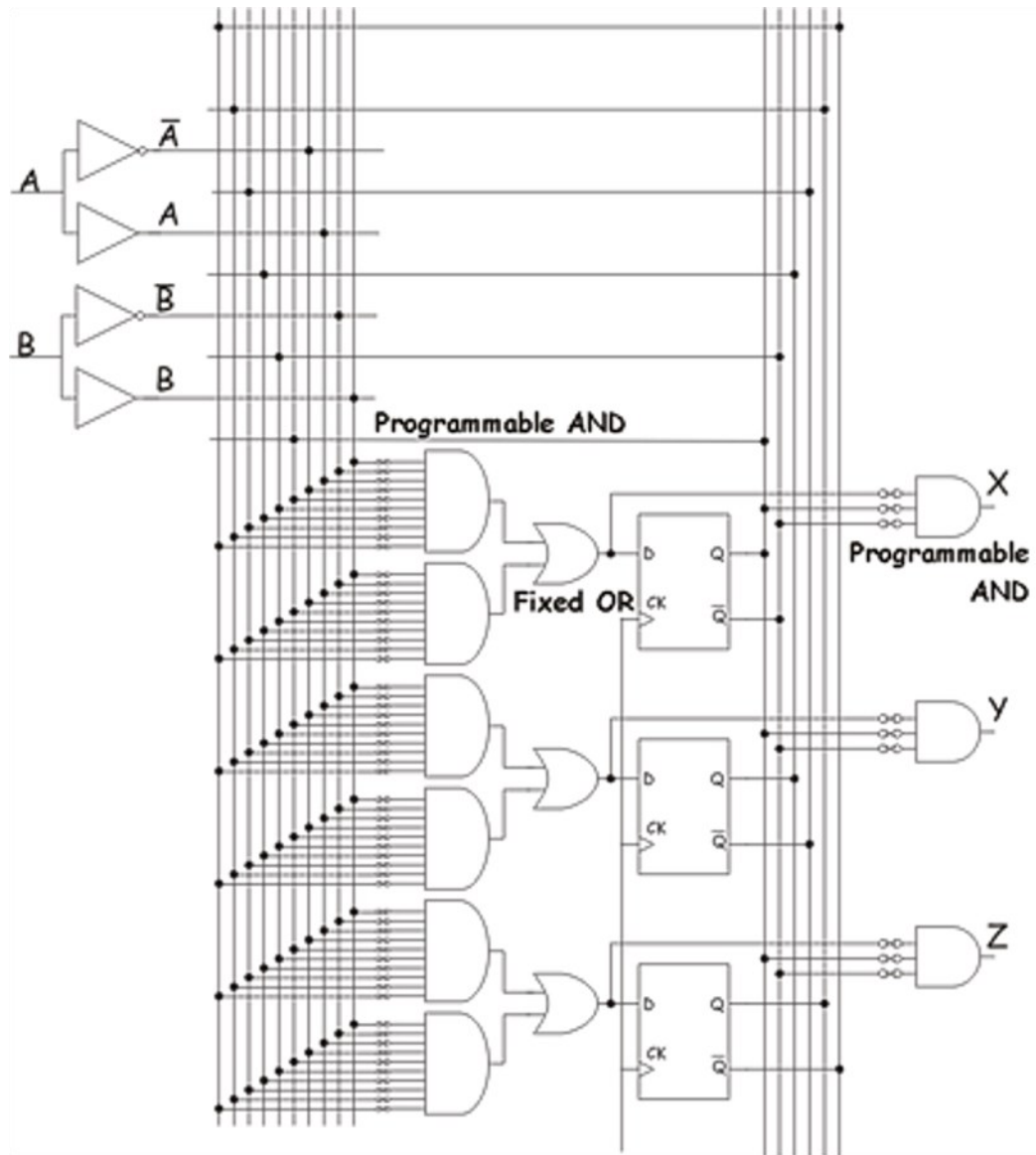
Fig 2.3: PLD architecture

Here you can see that registers have been added to the 'fixed OR' gates and that their outputs are fed back into the central bus in the device. All the registers operate off the same clock signal.

The outputs can be selected from the OR gate or from the outputs of the D-type. In a real PLD, the combinational part of the device would be vastly more complicated (and flexible) than that shown here. Both the AND and the OR parts of the circuit would be programmable. You would also find that the outputs would be tri-state, controlled by further fuses, so that they can be declared as either an input or an output in the programming process.

From the diagram, you can see that a wide range of circuits can be built from such a device.

Circuit representation

As the architecture inside a PLD gets more complicated, it becomes increasingly difficult to draw a traditional circuit diagram of the device. Instead we use a schematic representation of the device, as in the diagram here:
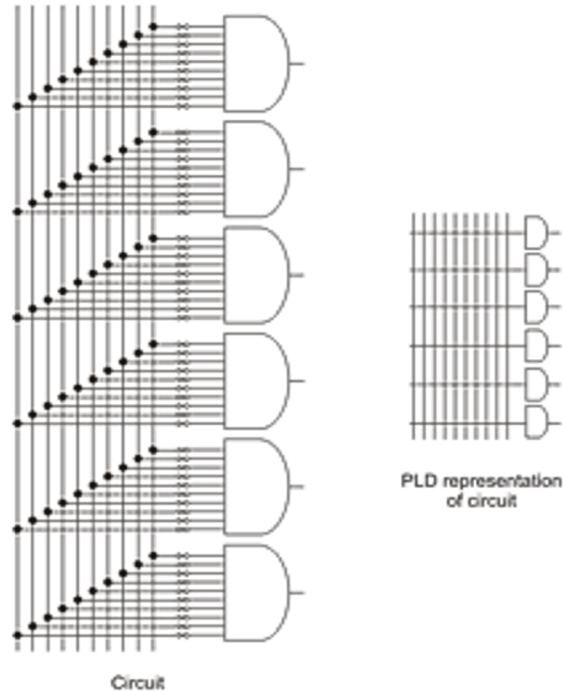

Fig 2.4: PLD circuit representation

The inputs and fuses on the AND gates have been simplified to a single wire crossing each of the available lines inside the PLD. There are no junction 'dots' connecting the inputs to the available lines.

Designers understand that a connection can be made between any input and any available line by blowing appropriate fuses. This new representation allows 'schematics' for vastly more complex devices to be condensed to relatively simple diagrams.

A real device

At this stage, we look at a real device. The 22V10 is a generic design, made by several man-ufacturers. It contains a large combinational logic block feeding 10 registers, or D-type flip-flops, whose outputs are fed back into the combinational logic block.

Fig 2.5: 22V10 chip

Showing the full circuit diagram in a conventional way gets very messy, and to understand its operation, we need to make some short cuts in how we draw the diagram. On the functional diagram, given below, the multiple input AND gates are shown by simply drawing a line across the appropriate logic lines and having a single input AND gate at the end of the line. Similarly the OR gate is shown spanning the eight inputs.

The actual circuit of each register block is slightly more complex than a simple D-type flip-flop but this will do for our purposes.

Notice that the output from the flip-flop is shown on both the top right and also the bottom left of the flip-flop.
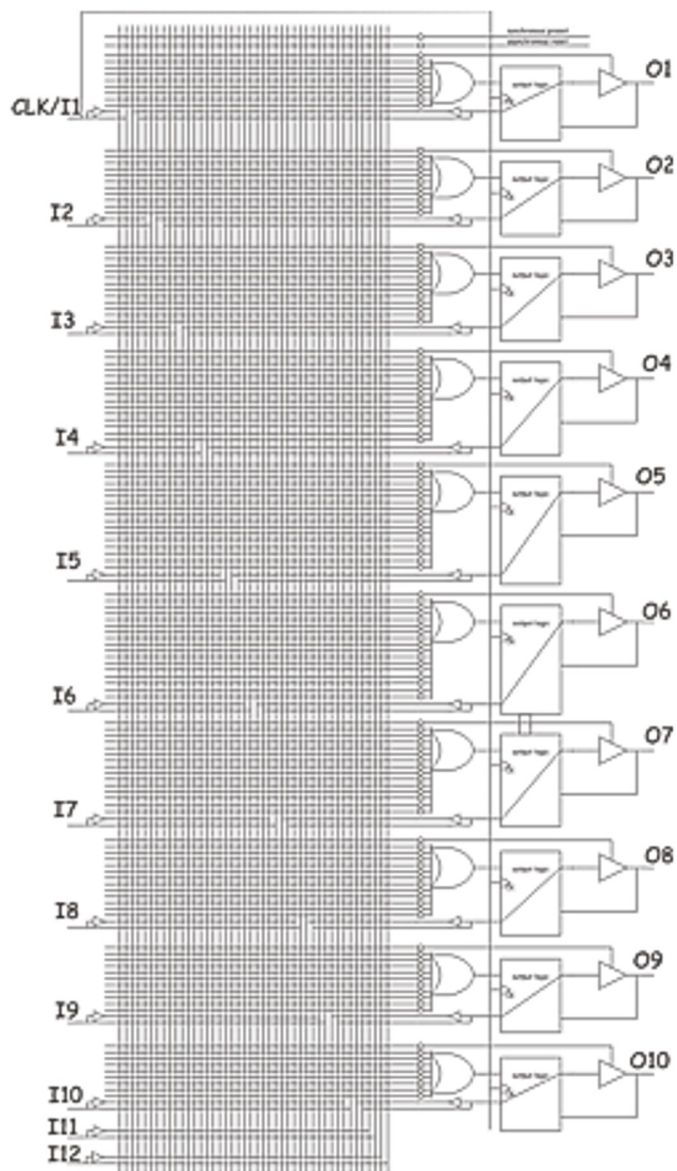
Fig 2.6: A block schematic of a 22V10

The spare lines on the right hand side of the combinational logic block are used for synchronous reset and preset lines. You can assume that these are fed to all the output logic blocks. Notice that the number of AND lines on the OR gates vary from eight to sixteen.

Packages and logic levels

22V10s are available in a number of packages including DIL (24 pin,) PLCC and surface mount variations. They operate off 5V and are CMOS and TTL compatible.

An example

To explain how this works, here is a representation of the fuses left intact (the dots,) and the input signal paths (the thick lines,) for a design where O2 will represent the ANDed inputs) I8, I9, I10, I11, I12:
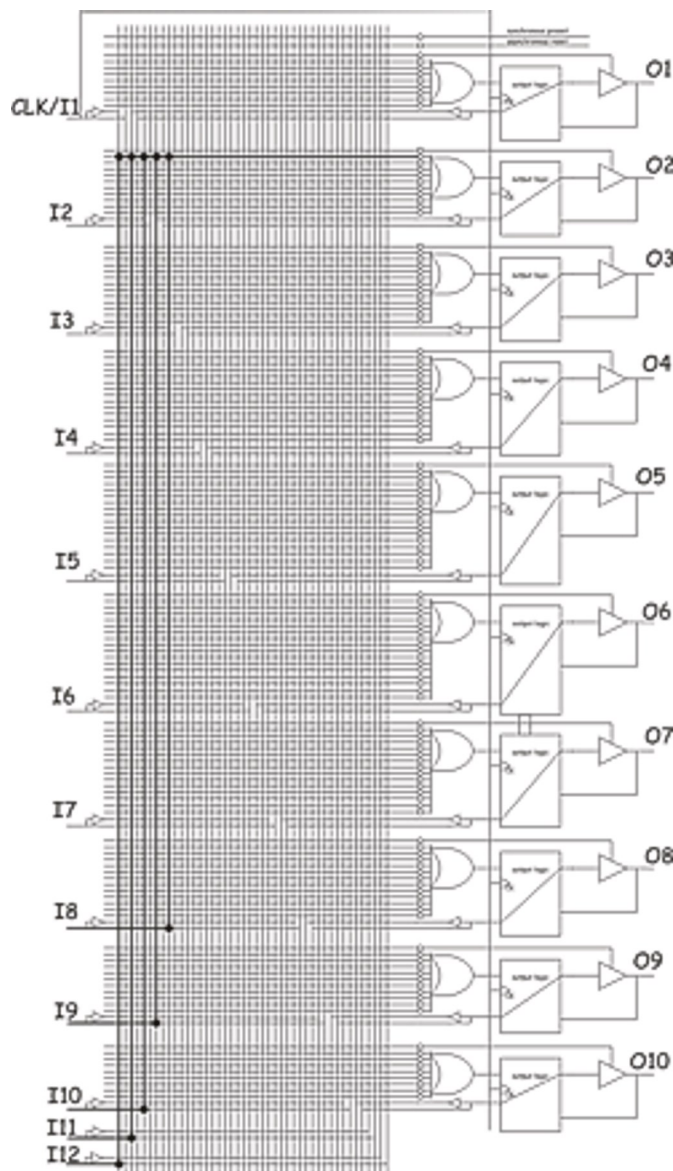

Fig 2.7: 22V10 Example

Exercise

This exercise demonstrates how flexible the device is and helps to explain the structure of the device

Use the 22V10 diagram to develop an 8-bit counter that resets at a count of 10. You should print out a copy of the 22V10 - copy it from your browser and paste it into Word first - and mark with a dot the connections you want on the combinational logic lines, i.e. the fuses that are left intact after programming.

In formulating the answer to this task you may find the following table useful:

	O2	O3	O4	O5
Z	0	0	0	0
Z+1	1	0	0	0
Z+2	0	1	0	0
Z+3	1	1	0	0
Z+4	0	0	1	0
Z+5	1	0	1	0
Z+6	0	1	1	0
Z+7	1	1	1	0
Z+8	0	0	0	1
Z+9	1	0	0	1

Fig. 2.8

where:
Z is the state at the time of the first pulse;
Z+1 is the state at the time of the second pulse etc.

It will help to formulate the Boolean equations for each output state, before tackling the diagram.

For example: you can see that O2 is simply the inverse of its previous state.

This can be written as:

$$O2 (Z+1) = !O2$$

This implies that all you need for the O2 line is an inverter between the O2 output and the flip flop feeding O2.

A Karnaugh map may help you to simplify the problem.

Why use a PLD?

A question that comes up often is: "Why should we use a PLD when a microcontroller is so easy to program and is so flexible?"

In the 1980's, when PLDs first came onto the market, the development path for microcontrollers was very inelegant. A microcontroller was a high volume, high commitment item with even 'One Time Programmable' devices not yet available. The use of PLDs was easy to justify then.

Today, reprogrammable microcontrollers are the preferred choice for many of the applications that PLDs used to perform. PLDs still have one great advantage over

microcontrollers - speed. Propagation times through a PLD are of the order of a few nano-seconds, which makes them about 100 times faster than the average microcontroller. As a result, PLDs are still useful in applications where timing is important, such as bus control and data multiplexing.

Complex PLDs

Even though devices like the 22V10 allow designers to mop up a large amount of 'glue' logic, in the form of 74xxx ICs, there are applications where more complex logic blocks are needed.

Hence Complex PLDs, (CPLDs,) were developed. These are basically a number of PLDs formed in a single package, with an even larger programmable interconnect block between the PLDs themselves. Combining PLDs in this way has advantages - reducing component count on the circuit board, increasing reliability and increasing the speed of the overall circuit.

These days CPLDs are available with up to over five hundred logic elements in packages of over two hundred pins.

As the complexity increases, we need yet another way of representing the architecture inside the CPLD, to help us to understand the device's structure and the functionality. We use a block approach.

Look at the following diagram that represents the inner workings of an Altera 7000 CPLD device.

At the top of the diagram on the next page, you can see the various clocks and output enable lines which are multiplexed and fed to all the macrocells. The word 'macrocell' is used to refer to a programmable register logic block. You might also find this referred to as a 'logic element'. The macrocells themselves are bunched together in blocks of sixteen, each of which is associated with up to sixteen I/O (input / output) lines. The heart of the device is the PIA – Programmable Interconnect Array – which consists of a vast grid of signal lines that allows each macrocell to access all other lines in the device.

This basic structure is expandable. This allows manufacturers, like Altera, to make different versions of the device with different numbers of macrocells. As this number increases, so does the area of silicon required to implement it, and the size of the package which houses the device. The result is that the price increases as the number of macrocells increases.

At this point the actual internal architecture of the device becomes less important (unless you are going to design CPLDs). The design software, Quartus, used to design and program these devices, shelters users from having to know too much about the architecture of the device.
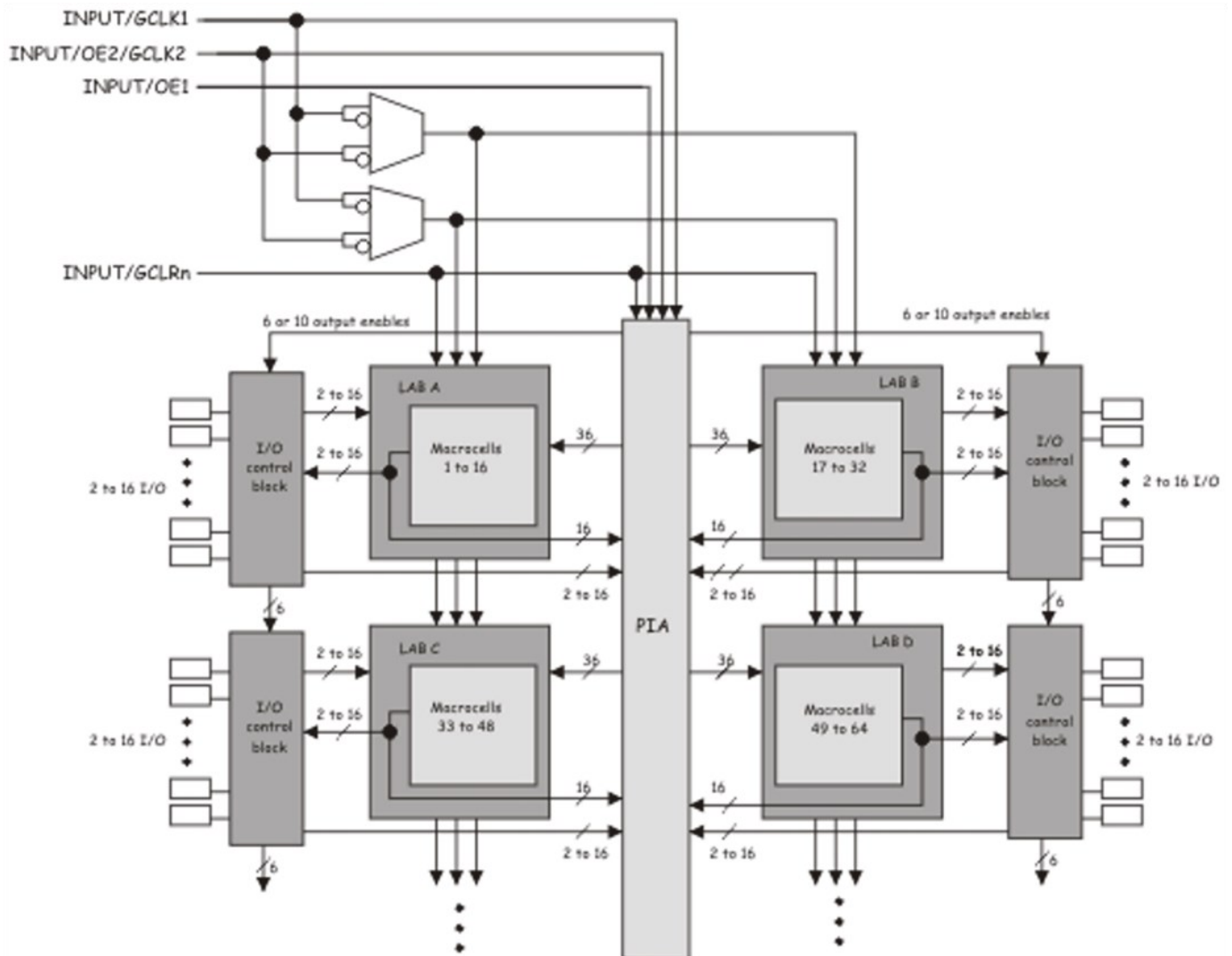
Fig. 2.9: Complex PLD

FPGAs

Field Programmable Gate Arrays, (FPGAs,) take this architecture a step further. In addition to circuitry for simple state machine generation, each logic block within an FPGA also contains memory, which can be configured for a variety of topologies. We come back to look at the architecture of an FPGA later on, but here are a few key differences between FPGAs and CPLDs.

The most remarkable difference is the scale of the two devices. Whereas a CPLD contains a few hundred macrocells, an FPGA contains a few thousand. The Altera FPGA device shipped with the E-blocks development kit contains over 10,000 logic elements. FPGA's with millions of macrocells are available.

Even the most complicated of the designs in this course uses less than 1% of the resources of the FPGA.

Whilst the architecture of CPLDs and FPGAs is different, the software shields the user from these differences. The same software, Quartus, is used for both CPLD and FPGA design. As far as a user is concerned, the difference between an FPGA and a CPLD is mainly that the FPGA is bigger. You can do more with it, but it costs more.

A key difference between CPLDs and FPGAs is that CPLDs are based on PROM-like technology, which allows the devices to retain their configuration when power is removed. On the other hand, FPGAs are based on Static RAM technology, which means that the contents need refreshing repeatedly. As a result, FPGAs need reprogramming each time the device is powered up, either directly from a PC, from a microcontroller or from static RAM.

One final difference between PLD technology and FPGA technology is that the FPGA operates at a lower voltage. Internally, the FPGA we use operates from a 1.5V supply which is buffered up to 3.3V for interfacing to the outside world.

Why use FPGAs?

Firstly, we need to restate that, in implementing a design, there is little difference between programming a CPLD and an FPGA. The software takes care of it all for you.

When looking at the question 'Why use CPLDs?' earlier on, the only real reason for preferring a CPLD to a microcontroller, was speed. As microcontrollers themselves are getting faster, that argument is weakening. However when asking the question 'Why use FPGA's?' the answer is very different.

The argument will revolve around a new design methodology called 'System On Chip' or SOC. This refers to the capability to put an entire digital electronic system into an FPGA. For example, a designer can now implement an entire microcontroller, a digital signal processing system, a block of memory, a key pad interface, a graphical LCD display interface, and a USB interface on one single chip.

To do this, the designer uses software to define different parts of the FPGA to perform the relevant functions. The same software is used to link the design elements together. It can even implement a program in pseudo-ROM that the microcontroller can run.

The traditional method of implementing such a design in silicon would be to either link several integrated circuits together on a circuit board, or to develop an ASIC, (Application Specific Integrated Circuit).

The first solution, developing the system on a circuit board, would increase cost, decrease reliability, and increase design time. Developing an ASIC takes months and months, is extremely costly and is only therefore practical where large volumes are involved.

Fig. 2.10: FPGA system.
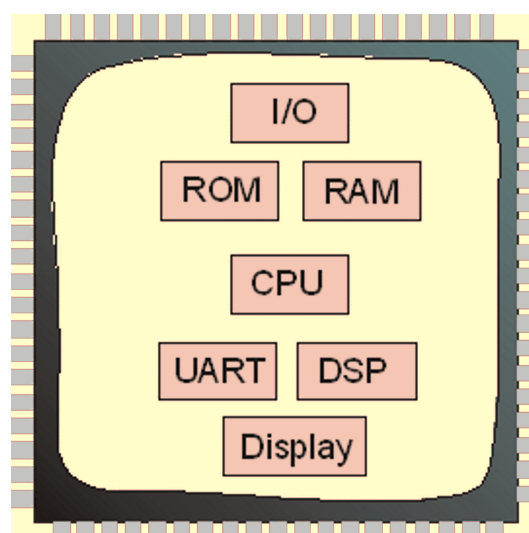
Using System On Chip designs has several advantages:

- design cycles are short;

- the resulting system is totally reconfigurable;

- the design can be changed in production;

it allows the same intellectual property to be reused many times.

A disadvantage is that it is still fairly costly, but as the technology matures, the costs are falling to a point where more and more ASIC designs are changing over to FPGA. Equally, with designs that previously used a microcontroller with a few support devices, these are now being replaced by FPGAs.

System On Chip is such an important technology that it alone justifies learning about this technology and how to program these devices.
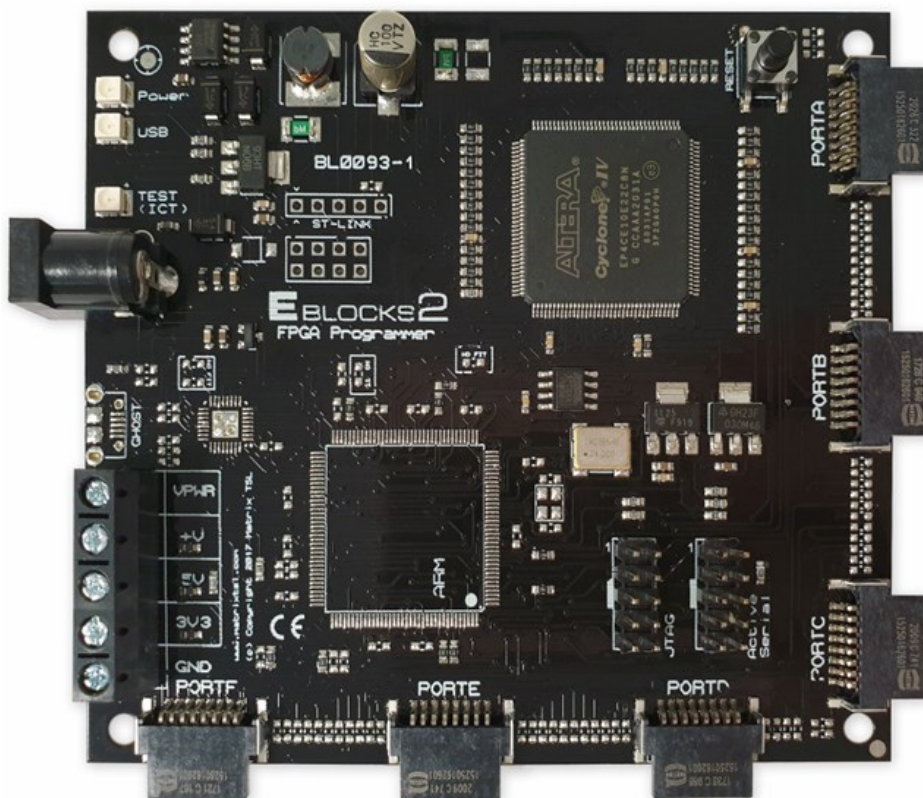
Which FPGA?

Fig. 2.11: FPGA E-blocks2 BL0093

The FPGA device used is Altera's EP4CE10E22C8 FPGA which contains over ten thousand logic elements. The FPGA E-blocks board provides 5 full E-blocks ports to interface to other E-blocks boards, from simple LED and switch boards through to more complex boards like internet interfaces, IrDA communication systems, internet and Bluetooth boards.

Chapter 3: Getting started

Introduction

The function-limited version of Quartus which is available free of charge is more than sufficient for this course. Please note that you may want to print out the next section and work through it whilst installing Quartus.

Installation

The Quartus design software is available via the Intel FPGA web site.
The free version is currently called Quartus Prime Lite

The download first requires the setting up of an account with Intel, which is explained on the Intel web site.

Also ensure that you add the support package for Cyclone IV.

Run the Quartus set-up program to begin the installation process and follow the on-screen instructions. Ensure that all drivers are installed such as for the USB Blaster.

Drivers and cables

The instructions in the previous section should enable you to get Quartus working.

The next step in the installation is to add the hardware. This makes use of software inside Quartus that allows programming via the computer's USB port, and uses the 'USB Blaster' device.

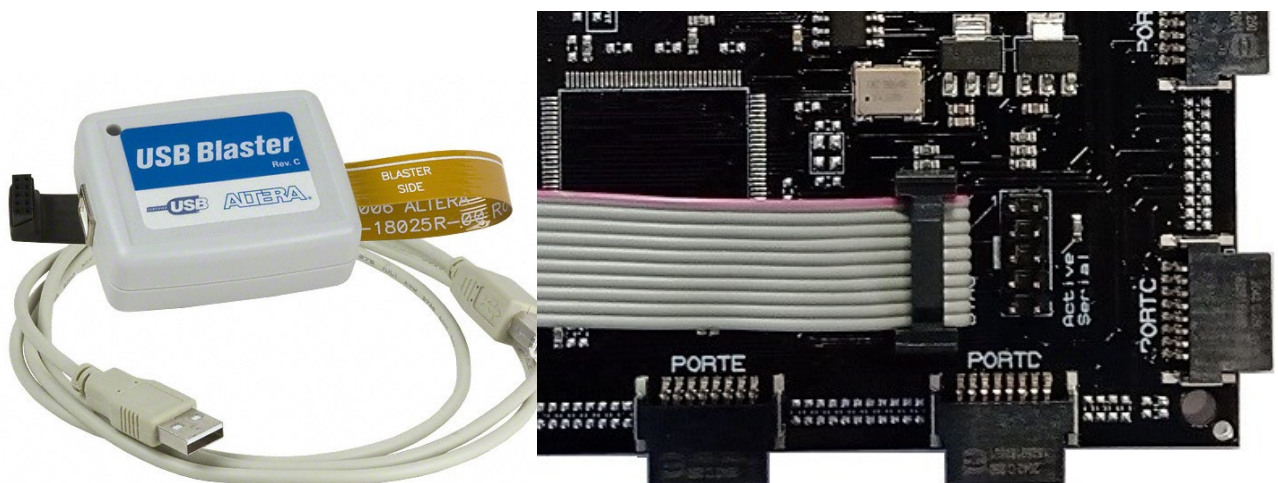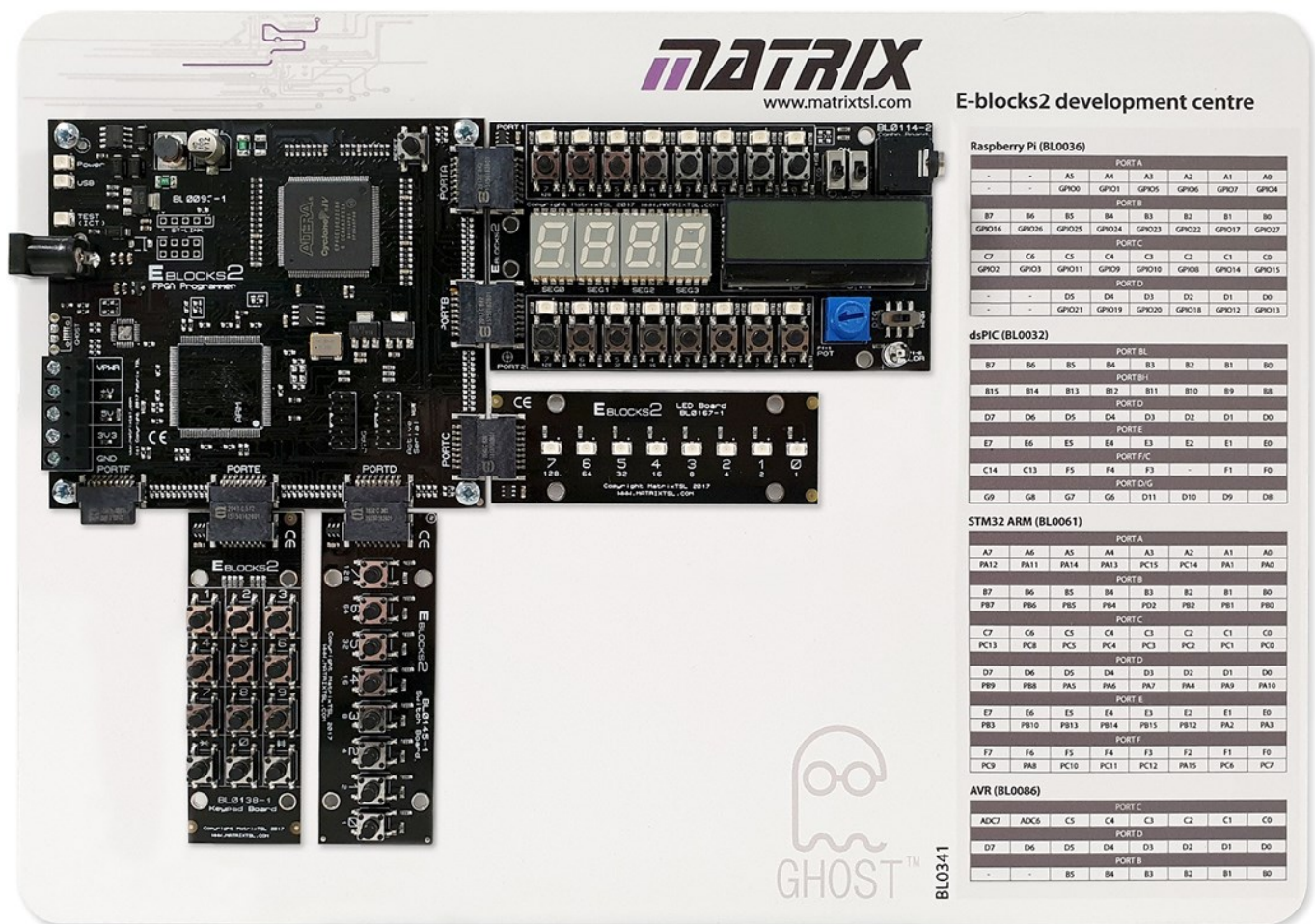Fig 3.1: USB Blaster.

Some operating systems may require a driver to enable Quartus to use the ''USB Blaster'.

This should have been installed along with Quartus, if not install the driver from

<Path to Quartus installation>\drivers\usb-blaster directory.

Setting up your hardware

The basic set up of E-blocks for most tutorials in this course is shown in the diagram:

Fig. 3.2: E-blocks2 layout.

Pin connections for hardware

With the E-blocks2 set up in the format described on the previous section, the pin connections are as follows in the next two pages.

Port A pin	FPGA pin	Switch	Character
0	110	0	0
1	111	1	1
2	112	2	2
3	113	3	3
4	114	4	
5	115	5	
6	119	6	
7	120	7	

Port B pin	FPGA pin	LED	Segment
0	73	0	a
1	74	1	b
2	75	2	c
3	76	3	d
4	77	4	e
5	80	5	f
6	83	6	g
7	84	7	dp

Ports A and B used with BL0114 Combo board Fig. 3.3
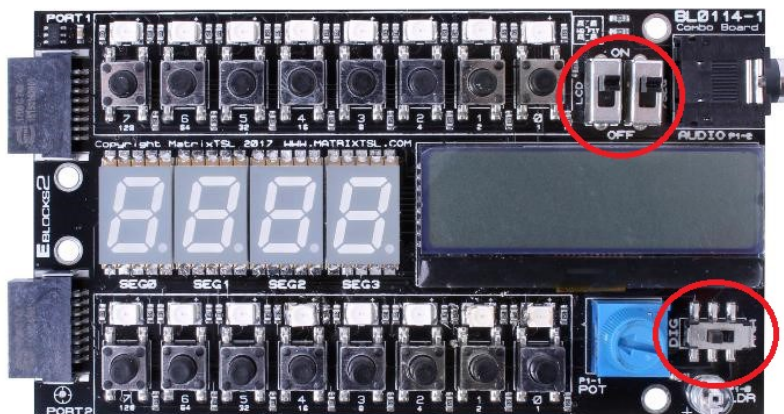
The exercises use the 4 digit display of the BL0114 Combo board and thus require the selection switches to be set as follows:

- LCD – Off
- 7 seg – On
- Dig/Ana - Digital

Port C pin	FPGA pin	LED
0	54	0
1	55	1
2	58	2
3	59	3
4	60	4
5	64	5
6	65	6
7	66	7

Port C used with BL0167 LED board Fig. 3.4

Port D pin	FPGA pin	Switch
0	38	0
1	39	1
2	42	2
3	43	3
4	44	4
5	46	5
6	49	6
7	50	7

Port D used with BL0145 Switch board Fig. 3.5

Port E pin	FPGA pin	Keypad
0	128	Column 1
1	129	Column 2
2	132	Column 3
3	133	
4	135	Row A
5	136	Row B
6	137	Row C
7	138	Row D

Port E used with BL0138 Keypad board Fig. 3.6
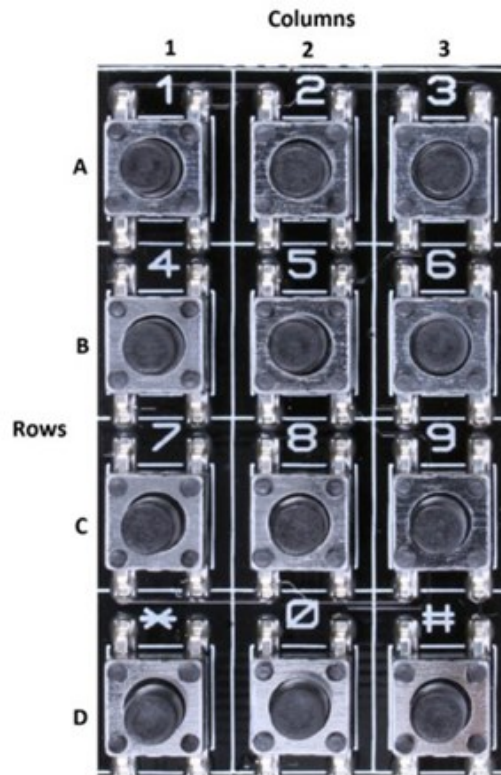
BL0138 Keypad board layout

Port F pin	FPGA pin	
0	98	
1	99	
2	100	
3	103	
4	104	
5	105	
6	106	
7	28	

Port F available for additional use

Getting your hardware working

In this section, you use Quartus to check that your hardware is working. Don't worry too much about understanding the design itself – you are concentrating on transferring the program.

Under the directory `Quartus` in the resource zip you will find a directory called `simpleand`.

Open this folder and locate a file called `simpleand.qpf`. Open this file - you will see a screen like this:
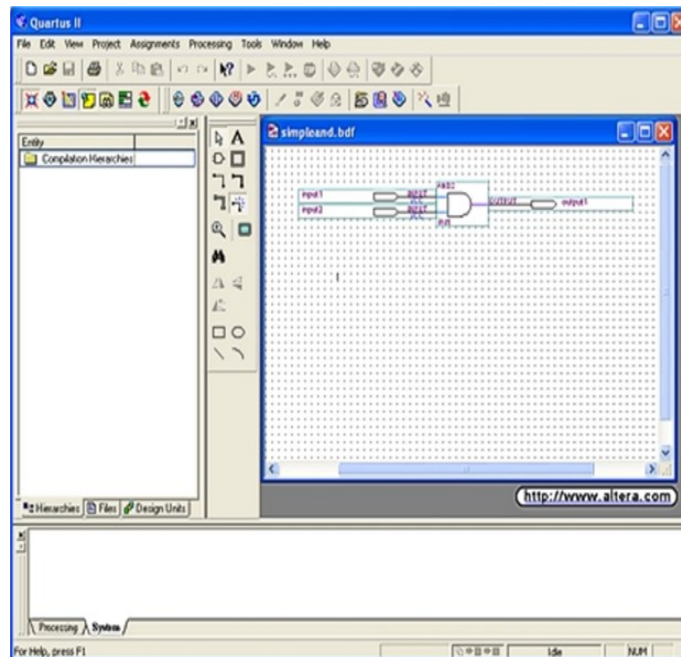
Fig. 3.7: Quartus IDE.

This project is a simple AND gate. Its purpose is to ensure that you become familiar with the Quartus software.

This design has already been compiled and is ready to be sent to the FPGA.

Configuring the programmer

From within Quartus, select TOOLS…PROGRAMMER
You should see a screen that looks like:
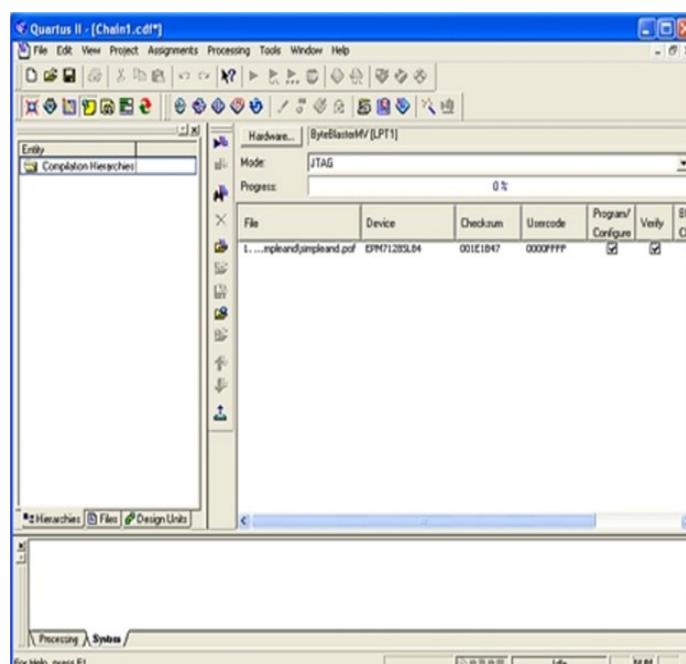
Fig. 3.8: Programmer screen

Firstly, select the type of hardware you have. For the board we provide, you will be using a USB cable, the USB Blaster.

To select this:

>   click on the HARDWARE button;

>   click ADD HARDWARE;

>   from the HARDWARE TYPE drop down box, select USB Blaster;

>   check 'USB Blaster' from the list of available hardware items;

>   your 'Currently selected hardware' should be 'USB Blaster'.

Next select the programming technique:

>   from the drop down box titled MODE, select JTAG.

If you have problems getting the programmer to work, make sure you have installed the latest programming drivers for your operating system.

Programming the FPGA

To the left of the programmer screen are a number of icons that control the programmer. If your file is not listed in the programmer screen, you need to select the file you want to program, and the program settings.

To do this:

>   click on the OPEN icon;

>   navigate to the original simpleand directory, and select the file simpleand.sof;

>   click the check boxes under PROGRAM and VERIFY.

You are now ready to program your device.

To start, click on the START PROGRAMMING icon

Hint – if you are not sure of the function of an icon, hold your mouse cursor over it. A small pop-up hint appears, displaying the icon's function.

Once you press the button, the program is sent to the FPGA.

You can now test the AND gate. Pressing both switches SW0 and SW1 illuminates LED D0. This is a two input AND gate.

Chapter 4: Getting to know Quartus Introduction

Whether you are instructor or student, you need to respect Quartus. Quartus is a phenomenal design tool, used extensively in industry, very capable and very flexible. Its power and flexibility mean that it can also be quite complex to use for the novice,.

Whilst it has learning potential for digital electronics, it is not perfect for all the phases of learning. This is particularly true for basic digital electronics, where students will struggle if they have to learn electronics and how to use Quartus, both at the same time.

We recommend that you use Quartus only when you have got a grasp of digital electronics.

How to learn Quartus

When looking at the materials supplied with Quartus, we found a number of problems for novices:

the documentation supplied is very complex;

the examples supplied are very complex;

the assumed knowledge is high.

As a result, learning Quartus takes time and risks being frustrating.

For these reasons, we have provided some simple step-by-step projects that you can use yourself. These start at a very simple level and work up to much more complex designs, equivalent to many logic chips.

We recommend that you review the Quartus tutorial and documentation at a later stage. The examples may be complex but the manuals do give a very good picture of the design flow and the hierarchy within Quartus. The Quartus Handbook and other support materials are available from the Altera website (www.altera.com).

A note on Wizards

The instructions for entering information in wizards should be used with care. When guiding you through wizards, we indicate only the main selections to make – please leave other items unchanged.

Using this document

This document contains instructions for designing a project from scratch. We have included quite detailed instructions so that you can follow the design flow. You can go through this document step-by-step and enter the design now, or you can skim through and try the exercises at the end coming back to this document when/if you get problems.

Document conventions

Within this document we will use the following conventions:

Courier will be used for anything that must be typed as it appears, such as a file name:

```
C:\quartus\bin\quartus.exe
```

CAPITALS will be used for menu commands which you need to click on such as FILE..NEW

Note: Quartus does not allow spaces in file names. For this reason we use underscores in file names throughout these tutorials.

Design flow

Before getting into your first design, it is worth considering all of the elements in it and in the Quartus package. You need to be aware of the different types of files you will create in producing the design.

The design flow stages within Quartus can be simplified to:

- design your circuit;

- compile your design;

- design your simulation;

- simulate your design;

- program your PLD.

This is vastly simplified. The Quartus documentation gives a more exhaustive list, but this will do for now. It goes without saying that this is not a simple linear process – you may find problems with your design, and need to revisit certain stages as a result.

In using Quartus, it may seem that there are a large number of steps in making even the simplest design. Understanding the top level design will give more meaning to the individual steps in the design process.

Each step results in the creation of two types of file - those you create as part of the design, and those Quartus creates containing results of the compilation and simulation processes.

At the top of the hierarchy is the Quartus project - a set of files that stores information about your design. Within that, the main files are the Quartus Project file (file extension '.qpf') and the Quartus Settings File ('.qsf').

You will find that there are a large number of other files created – a simple traffic light project results in around 37 different files in the design directory and about 20 files in the db subdirectory, most of which you never need to worry about. This can look a little daunting but is a result of the very modular approach Quartus takes to design flow, which increases its flexibility.

Files you will create

The processes in the design flow outlined above create a large number of files, with a variety of file extensions. It is more important to appreciate that these divide into two main types - simulation files and design files - than to know the specifics of each file.

When designing projects using a conventional block diagram approach, you create *.bdf files: **b**lock **d**iagram **f**iles. These contain the core of your design. You may end up with several of these in one design – Quartus allows you a hierarchy of these to simplify the design process.

Each design you produce should be simulated to verify that the design does what it is supposed to. Initially, this simulation does not take place in 'real time', and so cannot be used to test the effects of the design on LEDs and 7-segment displays and other hardware.

This course does not cover simulation. There is a wide variety of tutorial material on simulation on the internet.

It is important to understand that when using Quartus, you are working with projects - collections of files. You never really work with individual files as you might do with other design tools.

The Quartus GUI

Before you start, you need to identify parts of the Quartus design environment. When you first load Quartus and select FILE….NEW BLOCK DIAGRAM FILE, your screen will resemble this (depending on the version of Quartus used, and the way it is set up):
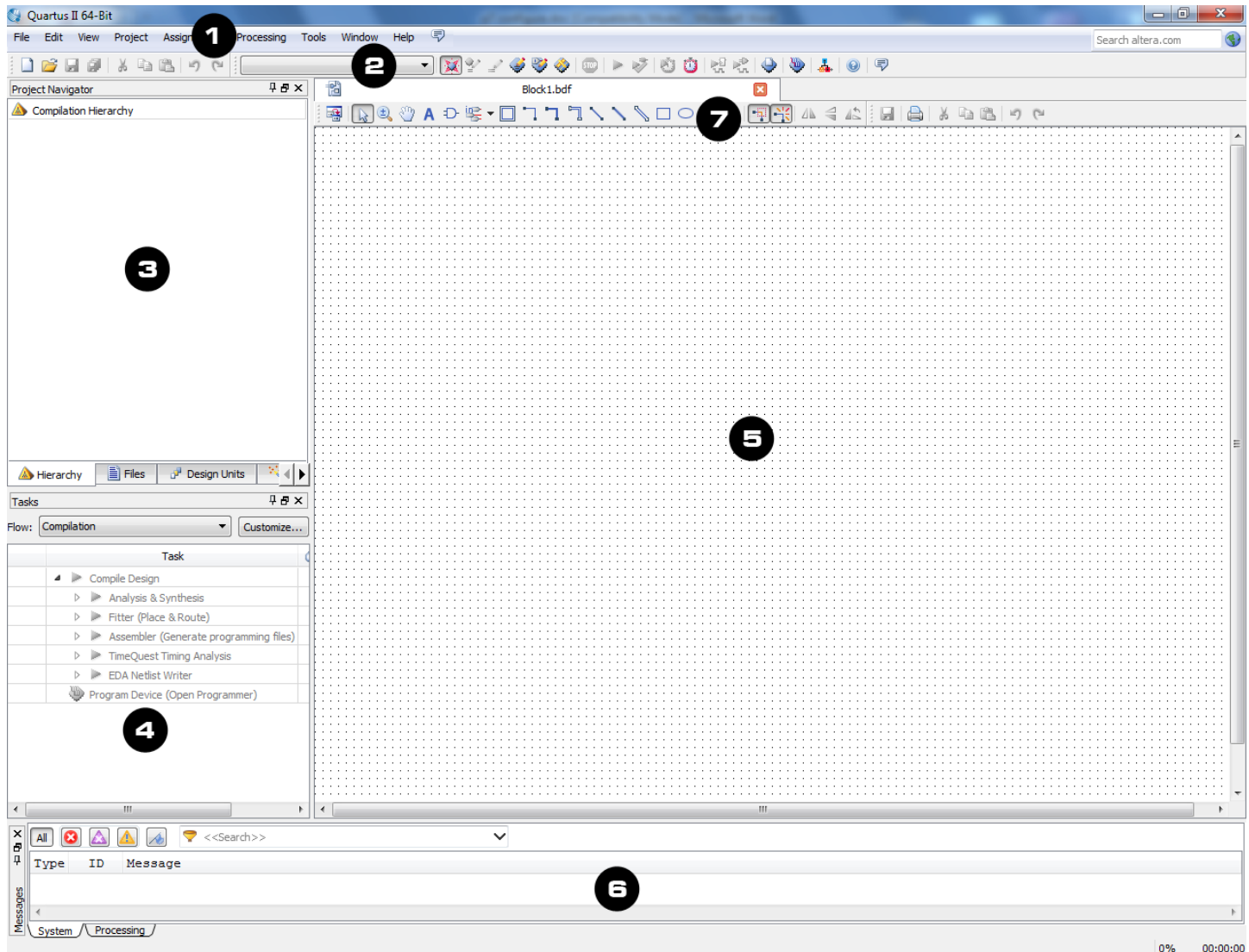
Fig. 4.2: Quartus GUI

The areas numbered in this image are:

1. a standard Windows tool bar, giving access to all functions within Quartus;

2. two icon fields, which give quick access to major functions within Quartus; (All of these are also accessible via menus. Throughout the course, we refer to the menu commands rather than to the icons.)

3. the Project Navigator, which allows you to see the hierarchy of your project and the files it contains;

4. the Status window, showing the status of various operations (such as simulation and compilation);

5. the design area, where you actually enter the design information;

6. the Message window, where Quartus displays error and warning messages during

simulation and compilation tasks;

7. the Block and Symbol Editor toolbar.

With the exception of item 7 you can make these items visible and invisible by selecting VIEW….UITILITY WINDOWS.

Introduction to design

In creating a new design from scratch, you need to:
run Quartus and choose toolbar options, etc.;
create a Project;
choose your preferred method of design entry: schematic, HDL, etc.;
enter a basic design using schematic, Verilog and VHDL;
choose and use appropriate compiler options;
choose and use appropriate simulator options;
download your design to the Matrix FPGA board;
test your design.

Next, you will work through all of the points mentioned above, except simulation.
The steps that follow use the same numbering to help you keep track. You may wish to print out each section as you go through it.

Step 1: Run the Quartus software.

The default installation requires you to run Quartus by clicking START\PROGRAMS\ALTERA\13.0SP1\QUARTUS\QUARTUS 13.0SP1 WEB EDITION, (or similar). Alternatively, there may be an icon on the desktop.

It may take several seconds to load, and it will try to check with the Intel website to see if there are any updates, so be patient!
Once it has loaded your screen should look something like the image below.
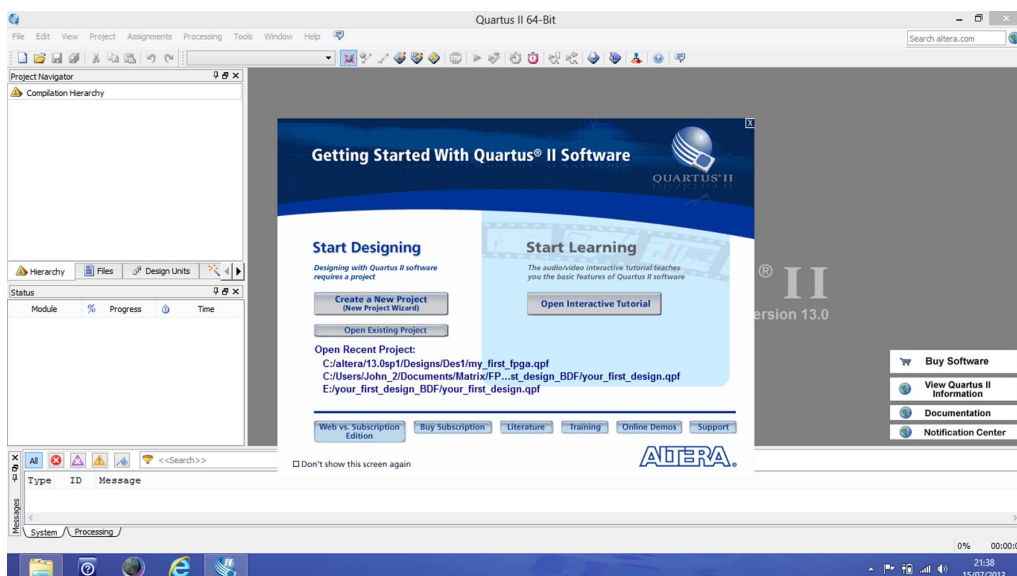

Fig. 4.11: Quartus opening screen.

The large (main) window is used for working on design files. The two smaller windows on the left are the Project Navigator (top) and Status windows. You can choose whether these are displayed using the View/Utility Windows menu.

The Toolbars can be customized so that only a few of the many command buttons are displayed. Your personal preferences may not agree with these settings, but these notes assume that you can click the buttons listed below. You can also access all the facilities via the main menus, and such menu routes will be given.

In order to reproduce the arrangement shown, use the TOOLS/CUSTOMIZE… facility. Turn off all the toolbars apart from Standard Quartus then use the Commands tab to drag the following command buttons onto that toolbar, from the given 'Actions':

    Applications - Programmer
    Assignments - Settings, Pin Planner, Assignment Editor
    Compiler - Start Analysis and Synthesis, Start Analysis and Elaboration, Start Compilation
    Edit - Undo
    File - New, New Project Wizard, Save All
    Help - Search
    Project - Set as Top-Level Entity

To remove any unwanted command buttons, simply drag them off the toolbar.
Then click OK (twice).

Step 2: Create a New Project

Click the New Project Wizard button ![icon] (FILE/NEW PROJECT WIZARD) and take a quick look at the information on the first page that opens.

Then click NEXT.

You need to specify the location and name of your project, and the folder (directory) containing it. The target directory will be set to the default. However, for this project, create a folder called `first` and then select it as the target directory. You can create it somewhere else if you wish.

You also have to think up a name for your project, and a name for the 'top entity' in it.
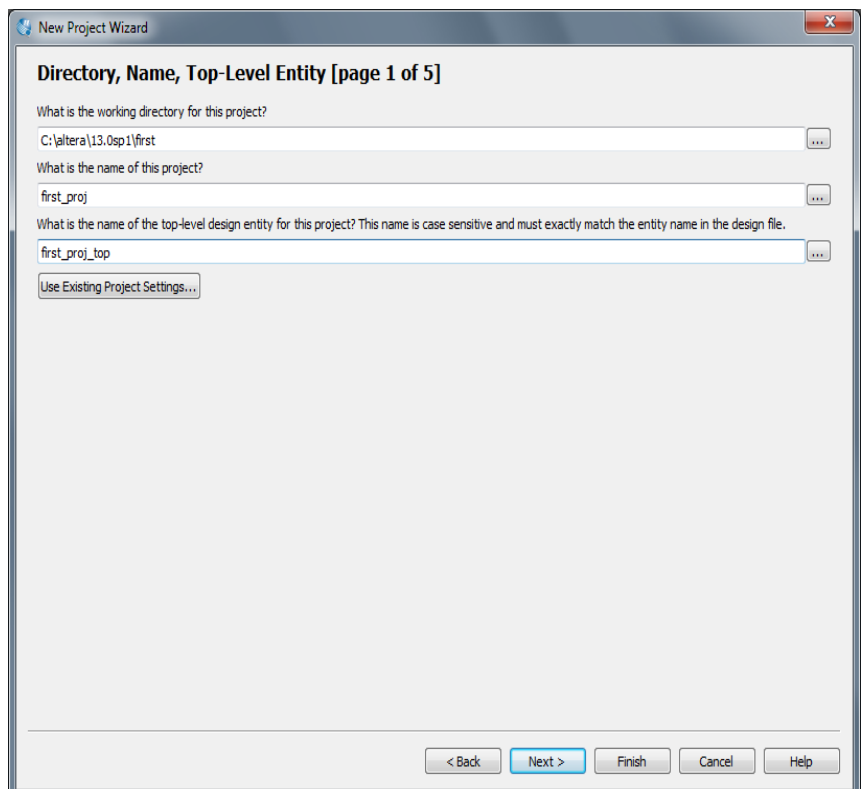
Fig. 4.12: Page 1 of New Project Wizard.

When you click NEXT, Quartus asks if you wish to create the `first` folder for your project, if it does not already exist Click on Yes', and page 2 of 5 of the Wizard opens.

There are no design files to add to your project, so click Next to get to page 3.

On page 3, you select the family of the specific device you wish to use. The chip on the Matrix FPGA board is a member of the Cyclone IV E family, so select that option in 'Device family'. In the 'Target device' section, click on 'Specific device selected in 'Available devices' list'. Scroll down the 'Available devices' list and click on the 'EP4CE10E22C8' device used on the Matrix FPGA board. Then click NEXT.

You are not going to use any EDA (Engineering Design Automation) tools apart from Quartus itself, so leave all the options to 'None' and click NEXT.

Page 5 confirms your choices. Check that you have selected the EP4CE10E22C8 device and that the other files and folders are correct. Click BACK if you need to change anything, otherwise click FINISH.

What exactly have you created? Use Windows Explorer to verify the following.

  The `first` folder has been created, for example in the `C:/altera/13.0sp1` folder.

  Inside that, there is a Quartus Project File (.qpf) and a Quartus Settings File (.qsf) in the `first` folder.

  There is a db folder within the `first` folder.

  There are two files inside that folder.

If you were to close Quartus now, the next time you run the program, your project could be called back by clicking FILE > RECENT PROJECTS > `C:/altera/13.0sp1/` `first_proj.qpf`.

Step 3: choosing your design entry method

In this first exercise, you are going to create a new design, using the schematic drawing entry method. (Later you will learn about Verilog and VHDL design entry methods.)

Click the New button   or select FILE/NEW… and BLOCK DIAGRAM/SCHEMATIC FILE option and click OK. A new file opens in the main window together with the Block and Symbol Editors toolbar. The Quartus software names the new file `Block1.bdf`.

Step 4: enter your design

You are about to create a design comprising just one AND gate!

Click the ⟨Symbol Tool⟩ button in the Block and Symbol Editors toolbar. The Symbol dialogue window allows you to select different symbols. Click on the folder or the symbol next to it to open the 'libraries' folder, then open the 'primitives' folder. Inside that, open the 'logic' folder. Select the 'and2' device, and click OK.

Click once in the schematic file drawing area to place one instance of a 2-input AND gate.

Click the ⟨ ⟩ button ('Selection Tool') or press the 'Esc' key to release the AND gate from your mouse. Select and delete any spare AND gates, that were inadvertently created.
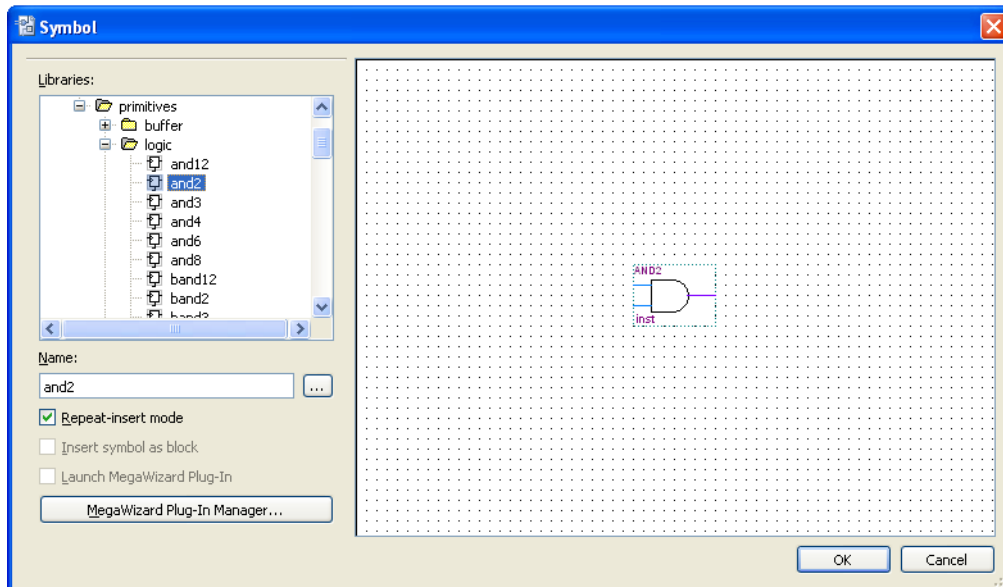
Fig. 4.13: Schematic symbol selector.

As well as the AND gate, your design requires input and output 'pins'. These allow you to specify physical characteristics of the design, such as the pin numbers of the chip to use. Use the Symbol Tool again but this time select the 'libraries/primitives/pins' folder. Select the input pin device, and click OK. Place an input pin to the left of each input connection of the AND gate.

Next, in the same way, select an output pin device from the 'libraries/primitives/pins' folder. Place one to the right of the AND gate output connection, and add two more as shown below.
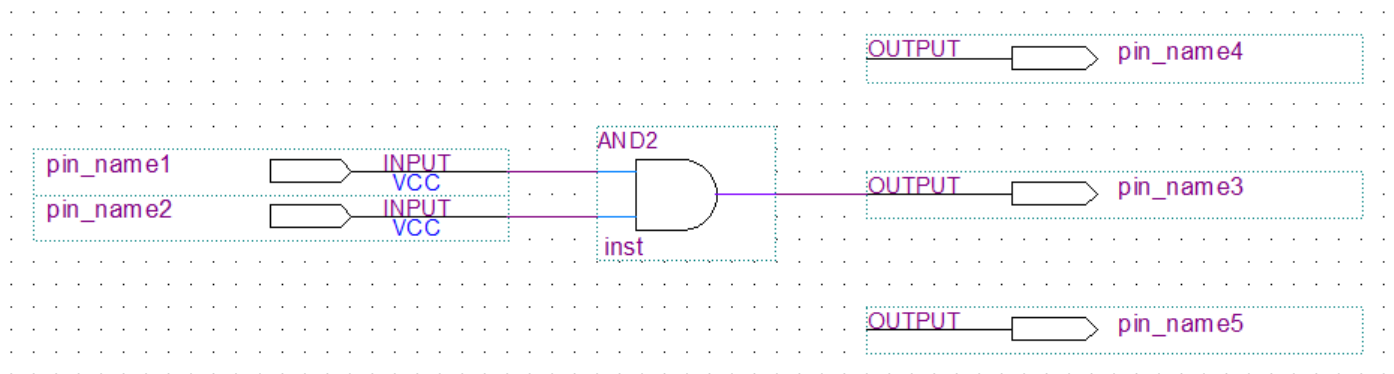
Fig. 4.14: AND gate design.

Now you can wire up the two input pins, and the three output pins. To do so, you click on one connection point, hold the mouse button down, drag the wire to the other and then release the mouse button. Wires are laid out automatically with just vertical or horizontal

runs ('orthogonally').

You can click on the Orthogonal Node Tool ⌐ or [⌐] just hover carefully over the component connection point to get into wiredrawing mode. Your drawing should start to look like:
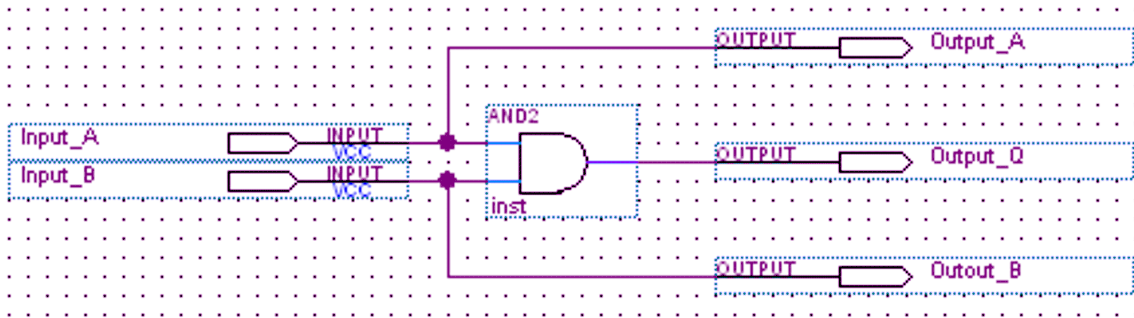
Fig. 4.15: Partially completed AND gate design.

Edit the names of the input and output pins by double-clicking them. Use the names given in the diagram.

Now select FILE > SAVE AS… A dialogue box should open, expecting you to name the file `first_proj_top.bdf`, since this was the information given in the New Project Wizard, and to save it in the `first` folder. If these are the settings in the dialogue, go ahead and click SAVE, otherwise enter the required file and folder names and then save the file.

Step 5: set the compiler options and analyse the design.

At this stage, your design needs to be analyzed. This checks for errors, but also allows the software to recognize the input and output names you entered on the schematic.

 This forms part of the compiling process, but before you run the compiler you need to check that the system will analyse the right things for you. Even though, in this, your first design, there is only one file, you still need to make sure that Quartus looks at it in the right way!

Click the Settings [✎] button (or click the ASSIGNMENTS/SETTINGS… menu). The Settings – `first_proj_top` dialogue opens.

Click the GENERAL option in the 'Category:' list and ensure that the top-level entity is `first_proj_top`.

Next, check that the 'Files' category contains just the `first_proj_top.bdf` Block Diagram/ Schematic type file. Now click OK.

Click the 'Start Analysis and Elaboration' button ✔ via the PROCESSING/START/ menu (Processing > Start > Start Analysis and Elaboration).

After a few seconds activity, the Status window should show the following:

Fig. 4.16: Analysis and Elaboration.

If there are any problems with your design, the compiler will stop sooner, and some error

messages will appear in the Message Window below the Main Window in the Quartus screen. Attend to the messages and re-analyse.

If you take a look at the files in your first and first/db folders, you should find that there are now almost fifty files. (You may need to refresh the Explorer window in order to see all the new files, if you have had it open while these operations have been taking place.)

If there are any problems with your design, the compiler will stop sooner, and error messages will appear in the Message Window below the Main Window in the Quartus screen. Attend to these messages and re-analyse.

If you take a look at the files in your first and first/db folders, you should find that there are now almost fifty files. (You may need to refresh the Explorer window in order to see all the new files, if you have had it open while these operations have been taking place.)

Step 6: simulate the design.

This is the stage at which to simulate the design at RTL level. This step allows you check the design logic. You have to devise a set of input signals and then let the software calculate how your design will respond. If the results coincide with the design specification then you start to feel confident that the design will actually work.

As the systems we are looking at are relatively simple, we take the alternative approach of loading the design into the FPGA chip and testing the design on the hardware directly.

Step 7: fit your design and download it to the Matrix TSL FPGA board

There are actually three stages within this step.

    a. Decide which physical pins on the FPGA chip to use.

    b. Get the Quartus software to 'fit' your design into the FPGA chip.

    c. Download the fitted design to the hardware.

Step 7a: adding the pin information

The hardware referred to in these notes comprises the Matrix E-blocks2 boards:

- BL0093 FPGA board

- BL0145 Switch board connected to Port D

BL0167 LED board connected to Port C

With this arrangement, pressing one of the switches generates a logic 1 signal on the corresponding FPGA pin, as shown in the table below. The connections to the LEDs are also given.

Switch	FPGA pin	FPGA pin	LED
SW7	50	66	D7
SW6	49	65	D6
SW5	46	64	D5
SW4	44	60	D4
SW3	43	59	D3
SW2	42	58	D2
SW1	39	55	D1
SW0	38	54	D0

Fig. 4.17.

We use switch SW7 (pin 50) for Input_A and switch SW5 (pin 46) for Input_B.

For the outputs, we use LED D7 (pin 66) for Output_A, LED D5 (pin 64) for Output_B and LED D0 (pin 54) for signal Output_Q.

To make these pin assignments, first click the 'Pin Planner' icon (or ASSIGNMENTS/PIN PLANNER....)

Double click in the first row of the 'Location' column (for Input_A.) Enter the number '50'. The software completes the entry as 'PIN_50'. Click ENTER on the keyboard.

Go down the 'Location' column, adding the pin numbers given above for the other input and outputs. The 'Pin Planner' looks like the following diagram.

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate
input_A	Input	PIN_50	3	B3_N0	PIN_50	2.5 V		8mA (default)	
input_B	Input	PIN_46	3	B3_N0	PIN_46	2.5 V		8mA (default)	
output_A	Output	PIN_66	4	B4_N0	PIN_66	2.5 V		8mA (default)	2 (default)
output_B	Output	PIN_64	4	B4_N0	PIN_64	2.5 V		8mA (default)	2 (default)
output_Q	Output	PIN_54	4	B4_N0	PIN_54	2.5 V		8mA (default)	2 (default)

Fig. 4.18: Pins assignment.

Close the Pin Planner.

When asked, click to save the file.

Verify that the pin information has been added to the schematic.

Step7b: Fit

Fitting is part of the compilation process. To start this, click on PROCESSING/START COM-PILATION... . The progress of the various stages of compilation is shown in the 'Status' window.
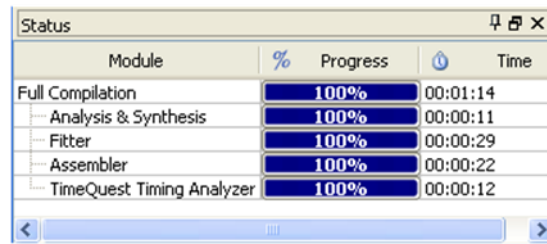
Fig. 4.19: Complete compilation

Hopefully, the message 'Full Compilation was successful' will appear after some seconds of activity.

Step 7c: Download

You are now at the stage where you have a design file (`first_proj_top.sof`) ready to send to the target board.

In order to do so, the USB Blaster driver must be installed on your computer. Connect the USB cable from your computer to the Matrix FPGA board via the USB Blaster.

Apply power to the board using the mains power supply provided, with the voltage set to 7.5V.

Now click the Programmer button (or select TOOLS/PROGRAMMER ...). The Programmer window should open. Tick the check boxes so that you end up with a window similar to that below.
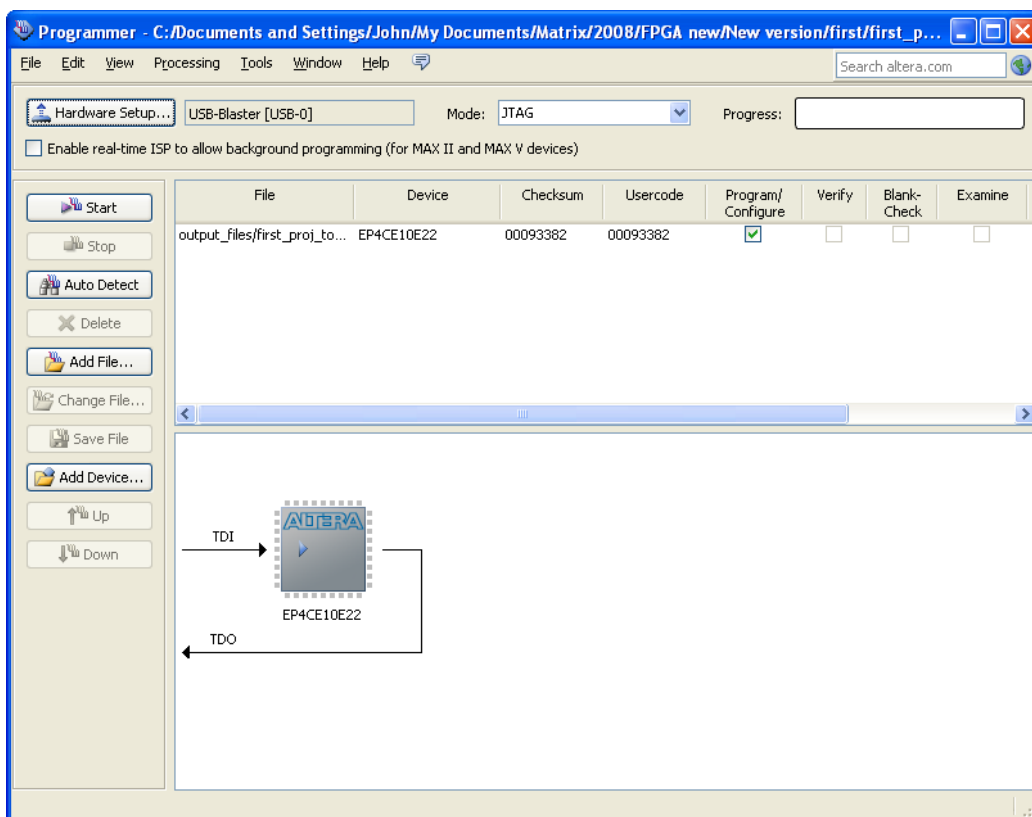
Fig. 4.20: Programmer ready to download.

Now hit the button, (or select START). The Progress bar should fill, and eventually show 100% completed.

Step 8: test the design

The moment of truth - verify that the FPGA chip behaves as a two-input AND gate! Pressing switch SW7 should make LED D7 light up, switch SW5 should light LED D5, and pressing both switches together should make LED D0 light up as well.

Note:
The design is still volatile. If you remove power from the FPGA board, your design is lost. A later section shows how to program the design permanently into the FPGA.

Optional step: Amend the design, re-compile and download

Getting your first design compiled, downloaded, and running on the target hardware takes some effort, but now you should be able to alter the design very easily.

If you have closed down Quartus, you need to launch it again, open the `first_proj` project then, if necessary, open the `first_proj_top.bdf` file. Select the AND gate and delete it. Now repeat Step 4 shown earlier, but select an OR gate ('or2') from the 'libraries/ primitives/logic' folder and place it in the drawing to occupy the space left by the deleted AND gate. Save the BDF file, and re-compile it.

Now run the programmer again, for the now modified `first_proj_top.sof` file.

Verify that you have an OR gate by watching the LEDs on the target board as you press S7 and S5 - either (or both) should cause LED D0 to light up.

Try replacing the gate in your existing design with a NAND gate (nand2 in the 'libraries/ primitives/logic' folder.) Save the design file, run a full compilation, use the programmer tool to download the design. Then verify the NAND gate action - the only way to turn off LED D0 is to press both S7 and S5.

Permanent transfer

The programmer will download the `first_proj_top.sof` file to the FPGA hardware, allowing it to be tested, but does not install the code permanently.

To do so, the file must be converted to a different format, 'JTAG indirect configuration file' (.jic) format.

On the main Quartus screen, click on FILE/CONVERT PROGRAMMING FILES... .

In the dialogue box that opens:

- select 'JTAG Indirect Configuration File' (.jic) from the 'Output programming file' type list;

- set the configuration device to 'EPCS4';

- set the mode to 'Active Serial';

- click on 'Flash Loader' and then click on the 'Add Device...' button;

- select the 'Cyclone IV E' device family and then select 'EP4CE10' from the 'Device name' list;

- click on 'SOF data' and then click the 'Add File...' button;

    The 'Select Input File' dialogue box opens.

- Navigate to the `first_proj_top.sof` file, in the output_files folder, select it and click on OPEN.

The dialogue box now resembles the one shown below
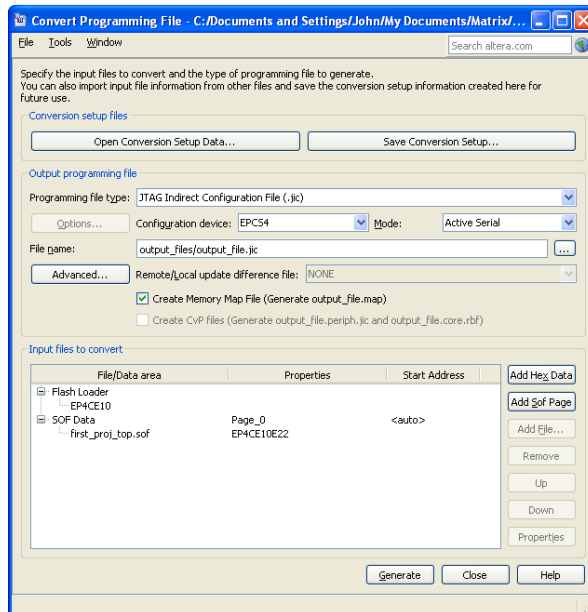
Click the GENERATE button.

The file `output_file.jic` has been created in the output_files folder.

Next launch the programmer, by clicking on TOOLS/PROGRAMMER. If necessary, add the `output_file.jic` to the programmer. Check the 'Program/Configure' and 'Verify' boxes.

The dialogue box resembles the one shown below:
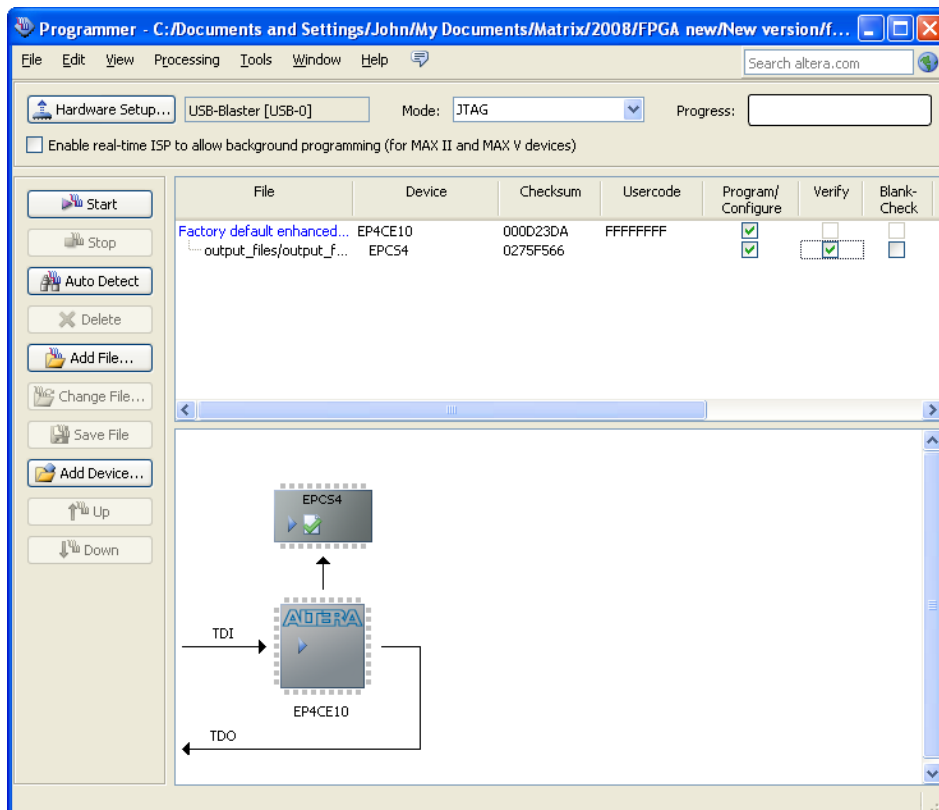
Click on START to transfer the file to the FPGA hardware.

Once the transfer is complete, press the RESET button on the FPGA board. The program is

now written into the FPGA. When you disconnect the board from the computer, the program still runs on the board. If you disconnect the power supply, and re-attach it, the program is retained, and will still run.

A second design - the PIN checker

The AND gate example served to identify the processes involved in creating a FPGA design. You can now practice those skills on a more complicated schematic design.
The aim:
  a security system requires the user to enter the correct PIN (personal identification number,) in order to be able to use a device;
  the PIN takes the form of a single 4-bit binary number, entered on four switches;
  an LED lights if the PIN is correct;
  a different LED lights if the PIN is wrong.
The design shows how two standard logic ICs, the 7485 magnitude comparator and the 7432 quad 2-input OR gate can be mimicked by the FPGA.
The Quartus code to do so is found by adding symbols for the 7485 and 7432 by scrolling down the 'altera/13.0sp1/quartus/libraries/others/maxplus2' library. The 'VCC' and 'GND' symbols (for +5V and 0V respectively,) are found in the 'altera/13.0sp1/quartus/libraries/primitives/other' library.

The 7485 can be cascaded to create an 8-bit comparator, by using the ALBI, AEBI and AGBI inputs (meaning 'A less than B', 'A equal to B' and 'A greater than B' respectively.) As we are not using it in this mode, we need to supply appropriate signals to these inputs, i.e. 0V to ALBI and AGBI, and +5V to AEBI.

We have inserted the correct PIN, as '1001' by connecting inputs B3 and B0 to +5V, and inputsB2 and B1 to 0V.

Assuming that the Quartus program is already running, work through the sequence that follows.
  Use the New Project Wizard to create a project called PIN_check, in the directory .../PINcheck, with a top-level file called PIN_check_top.
  Assign the design to the correct FPGA device - the EP4CE10E22C8.
  Create a new Block Diagram/Schematic file, and save it as PIN_check_top.
  Add components, and configure them, to produce the design shown below:
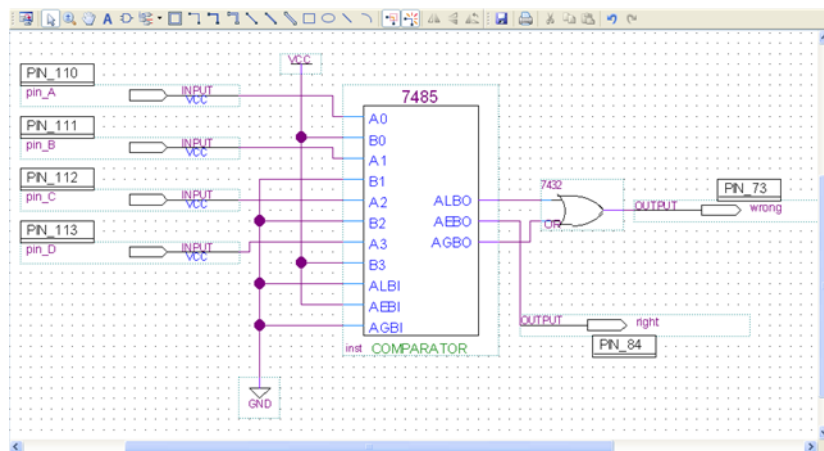
- Start the compilation process by running 'Analysis and Elaboration'.
- Then use the Pin Planner to allocate the FPGA pins shown in the diagram. This uses switches SW0, SW1, SW2 and SW3 to input the PIN ABCD, and shows the result on LEDs D0 (wrong) or D7 (right.)
- Once compilation is completed successfully, run the Programmer to download the design to the FPGA.
- Test it by trying different PINs entered on the switches.
- Further work:
  go back and change the correct PIN in the design, and retest it;
  modify it to use an 8-bit PIN. (You need to add a second 7485, and connect its ALBI, AEBI and AGBI inputs directly to the first 7485's ALBO, AEBO and AGBO outputs. Connect the OR gate to the outputs of the second 7485 in the same way as above.)

Summary so far

This exercise introduced you to the Quartus software for developing designs for FPGA devices.

You have seen how to:
- run Quartus;
- set up a project;
- enter a schematic design;
- compile the design;
- download the design to the Matrix FPGA target board.

In the next section, you enter the design using the Verilog hardware description language, rather than by using a schematic diagram.

Chapter 5: Advanced Quartus features

Introduction

In this section, we introduce further projects and exercises. It is designed to teach you the capabilities of Quartus, and project management within Quartus, rather than digital electronics design. You may want to come back to this section later, when you are developing more advanced projects.

The discussion in this section focuses only on the use of Block Diagram Files. However, Verilog or VHDL files can be incorporated easily into the design philosophy described here. If your projects are relatively simple, you may want to skip this chapter and continue to chapter 6, which concerns the design of FPGA programs using Verilog and VHDL.

Introduction

In this section you learn:
- how to compile a traffic light design from basic design elements;
- how to design all the elements from scratch.
- We have split this section into two parts:
- the first part looks at how all the parts of Quartus work. For this section, we provide all the design elements. You need to put them in the right place and enter the correct information within Quartus, to 'glue' the design together.
- the second part concentrates on entering an actual circuit. (If you get stuck, you should refer to the design files in part one.)

Putting a project together

One of the more difficult aspects of learning how to use Quartus, is understanding the structure of a project, and the settings that are required.

Here, we concentrate on putting a project together from all its elements. In the next section, we look at how to construct the elements themselves.

We will put together a traffic light controller that consists of three circuit blocks:
- a two bit counter (a four-state machine);
- a combinational circuit to decode the red, amber and green signals;
- a top level circuit block that ties these together.

As you have seen, a simplified design flow of a project is as follows:
- design your circuit;
- compile your design;
- design your simulation;
- simulate your design;
- program your FPGA.

Of course, this will rarely be as linear. You usually need to reiterate this process, and modify your design, before the design is right.

your design, before the design is right.

This section aims to help you to understand this design flow, and the hierarchical nature of Quartus. Pay special attention to getting the compilation and simulation settings right as that is usually where problems start.

Hardware settings
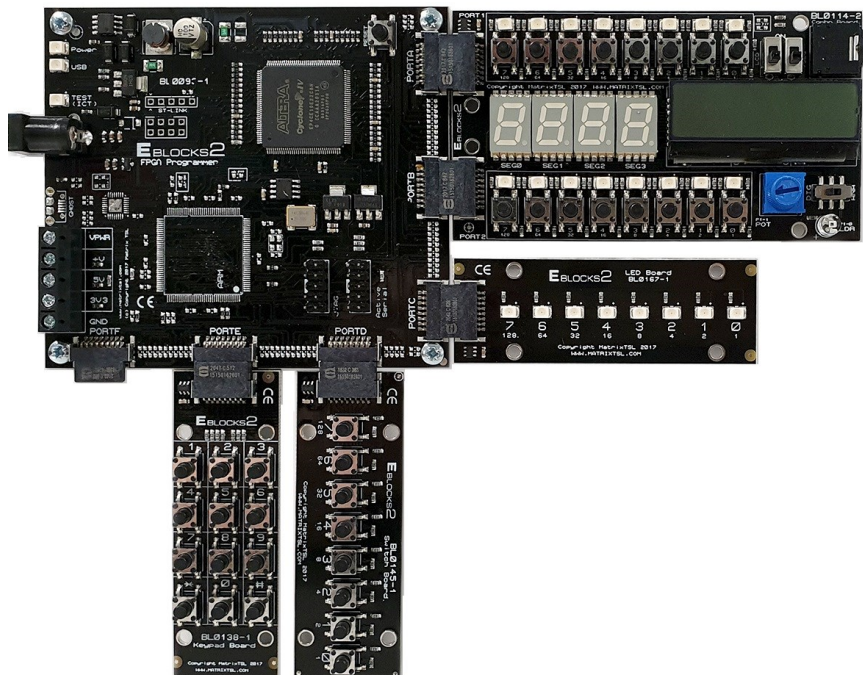
We will use the same E-blocks configuration described earlier:

Fig. 4.1: E-blocks layout.

The 7-segment display is on Ports A and B, the switch board is on Port D, the LED board on Port C and the keypad board is on Port E.

Instructions

To do this:

    Make a new directory called 'empty_traffic'. (Available in the resource zip)
    Copy the following files to this directory:
            `Traffic.bdf;`
            `Combinationalpart.bdf;`
            `2_bit_counter.bdf;`
    (Ask your supervisor for copies of these files if you do not have them.)
    Open Quartus and select FILE...NEW PROJECT WIZARD.
        Select the name of the project and the top level entity as 'traffic'
    NEXT
    Select the three BDF files: ADD ALL, NEXT
Set the Family to 'Cyclone 4E',and select the chip 'EP4CE10E22C8', and then click on

FINISH

Next, examine the three *.bdf files. To do this open the Project Manager window and select the FILES tab. You should be able to see these three files. Open each one in turn, by clicking on it, and look at the overall circuit. Hopefully you will recognize that the design of this circuit consists of two low level circuit blocks (`combinational-part.bdf` , and `2_bit_counter.bdf`) and a high level circuit that is comprised of these two low level circuits.

Compile your design by selecting PROCESSING…START COMPILATION.

You have now compiled a complete project. (You may decide to practise this exercise again to become familiar with the design flow.)

If you want to you can assign pins and send the program to the development board. However, for it to work, it needs a clock signal. This will be addressed in the next section.

A new task

Now, you have a better understanding of how the various design elements combine to make up a Quartus project file. We repeat the exercise but design all the individual elements of the project, to show you how to enter designs.

Here you will learn:

how to design a sequential logic circuit;

how to simulate and test this circuit;

how to configure Quartus to make a more complex project.

We recommend that you work carefully through this section to ensure that you are familiar with the settings in Quartus. Remember to follow the text carefully to avoid mistakes. This may seem laborious – but it is worth doing!

New brief

A traffic light controller consists (in its simplest form) of a two stage counter and some combinational logic to decode the status of the red, green and amber lights. (Notice that we are basing the design on the British traffic light system. If you are not in the UK, then you can vary this design to mirror your own national system later.)

Our aim is to teach you how to use Quartus. We assume that you are already aware of how traffic lights work. If not, then please refer to our Digital Electronics CD ROM where this design is explained further.

We are going to split this design into two stages:

first, we design the circuit elements – a simple two bit counter and some combinational logic;

once these are working, we design a higher level block that ties the two together as a complete circuit.

This is not the cleanest way of designing a circuit! Really, we should start with blocks, declare the inputs and outputs to the blocks and then design the blocks themselves.

However this technique is one that needs practising!

Building the two bit counter

Our traffic light design consists of a two-bit counter made from JK flip flops. To keep the design neat, we will separate the counter components from the combinational logic components.

To build the counter section, follow these instructions:

Start Quartus and set up a new project – `traffic.quartus`.

Start a new Block Diagram File and design a two-bit counter. Use JK flip flops from the storage symbol section. Call the input `counterclock`, and the two outputs `counter0` and `counter1`. Save the design as `2_bit_counter.bdf`. You should have a design that looks like:
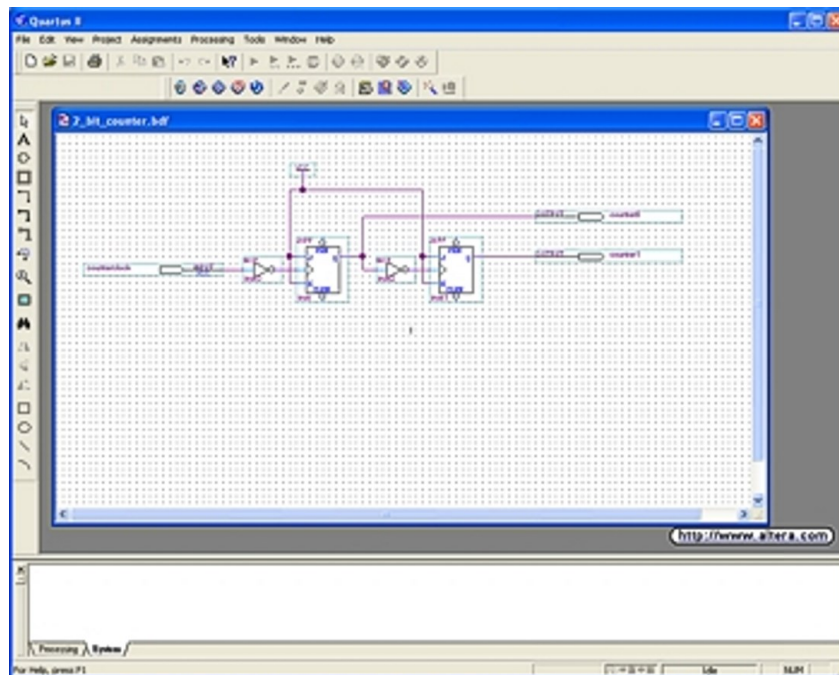
Fig. 5.1: 2 bit counter.

For this project, we build the combinational logic separately from the counter circuit. We could get the whole circuit in one Block Diagram File but it is good practice to break the design into separate elements. When doing so, we need to specify the hierarchy of each element – just as you would in a software program, where you define functions and sub routines. You need to let Quartus know where the file `2_bit_counter.bdf` sits in the hierarchy of your design, for the compiler. To do this, select PROJECT….SET as TOP LEVEL ENTITY.

Make sure your design compiles, by selecting: PROCESSING…START COMPILATION. (You will not be able to find the nodes of your design unless you compile first.)

Testing the two bit counter

This section shows you how to simulate the two-bit counter.

Build a Vector Test Waveform that clocks the input at regular intervals. Make sure that the two outputs count upwards. Save this as `2_bit_counter.vwf` .

You should have something that looks like:
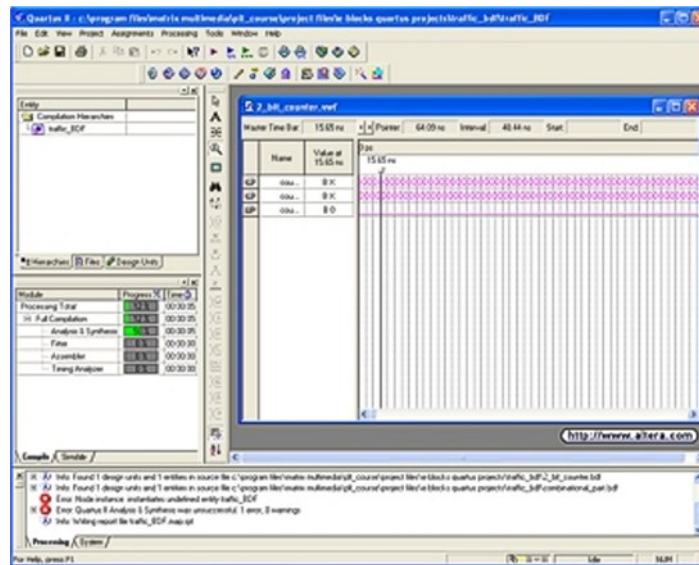
Fig. 5.2: Testing the counter.

You can use this to test the two-bit counter.

Building the combinational sections

Next, we move on to design the combinational logic part of the circuit.
To do this:
1. Make a new Block diagram file with inputs labelled `counter0` and `counter1` and outputs `red`, `amber` and `green`. The following screen image shows how to connect them:
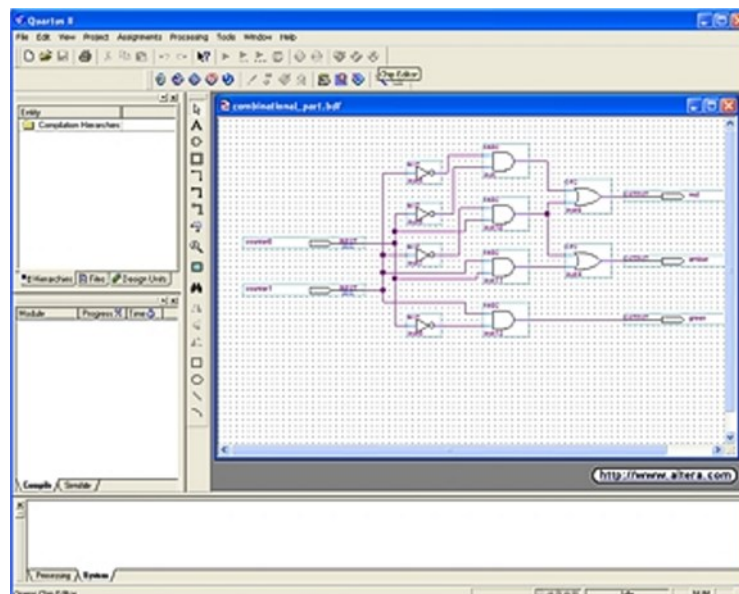
Fig. 5.3: Building the sections.

2. Set the compiler focus on to this object to add it to the hierarchy. To do so, select PROJECT….SET AS TOP LEVEL ENTITY.
3. Compile this part of the design

Creating a top level design

We are going to create sections that Quartus can recognize, for each of the two circuits we have designed. We can then design a top level design circuit that refers to each of these new parts. In practice, we will be creating two new symbols - one for the 2-bit counter and one for the combinational logic block.
To do this:

Open the 2-bit counter.

From the FILE menu select CREATE/UPDATE….SYMBOL FILE FROM CURRENT FILE.
Quartus will create a new project symbol from the 2-bit counter file. You can check this by clicking on the Symbol tool, expanding the project library by clicking on the '+' next to 'Project' and you will see your new symbol listed there.

Repeat this for the combinational logic circuit.

Now we can create the top level design.
To do this:

Create a new BDF and save it as traffic.bdf.

Click on the symbol tool and expand the Project library. Place a '2_bit_counter' symbol and a 'combinational part' symbol.

Insert an input clock and output pins.

SAVE the file.

Select PROJECT…SET AS TOP LEVEL ENTITY

You should end up with:
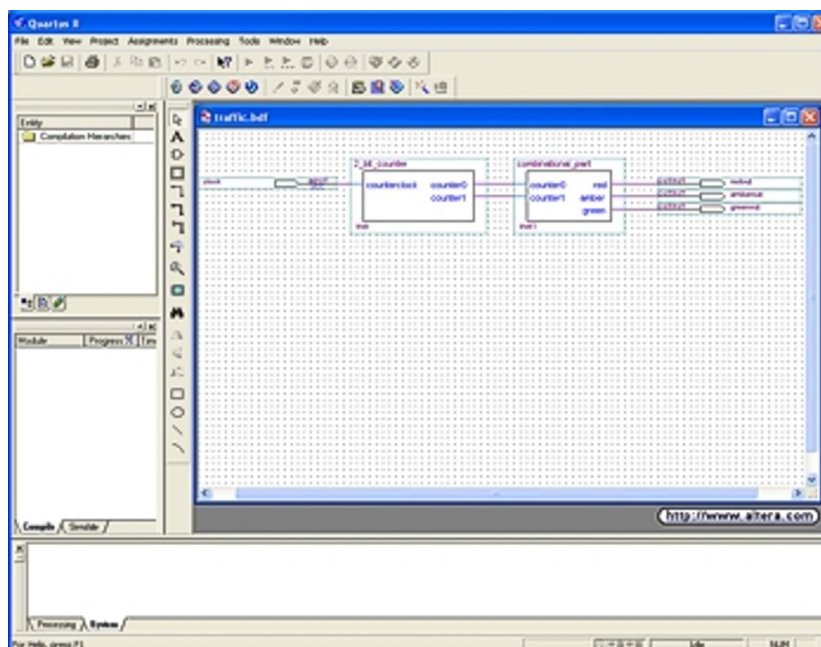
Fig. 5.4: Top level design.

The next step is to compile this design to check for errors.

Simulating the whole project

Finally, we can simulate the whole design to check it all works.

To do this:

1. Select FILE …NEW… OTHER FILES….VECTOR WAVEFORM FILES.
2. You should know how to set up the waveform file – refer to earlier sections if in doubt!
3. Save as `traffic.vwf.`
4. Update the Assignments setting for this simulation.
5. Simulate your file.

Open the Project Navigator screen. You can see that this project consists of:

```
Device design files
    Traffic.bdf
    Combinationalpart.bdf
    2_bit_counter.bdf
Software files
    none
Other files
    2_bit_counter.vwf
    traffic.vwf
```

Exercises

Exercise 1

Load the traffic project into Quartus. Make sure you can compile and simulate it. Look at this project in detail to reinforce your understanding of Quartus.

Exercise 2

Build this project again from scratch, to make sure that you understand all the elements of Quartus. (If necessary, refer to earlier instructions.)

Exercise 3

As you can see, there are some errors in the combinational logic part of the design which result in the traffic light output sequence being incorrect.

Redesign the circuit so that the sequence is as follows:

- Green
- Amber
- Red
- Red and amber
- Green
- etc....

Exercise 4

The traffic light combinational logic circuit is quite inefficient as it uses more gates than are necessary. Redesign the it to use fewer gates. Verify that the circuit works using simulation.

Exercise 5

In practice, a crossroads could consist of four sets of traffic lights, running two different cycles of lights.

Assuming that they are generated by a simple 4-stage state machine (like the original design,) make a table that shows the output sequences.

Then design a circuit that produces the following sequence of outputs:

Counter0	Counter1	Red 1	Amber 1	Green 1	Red 2	Amber 2	Green 2
0	0	off	off	on	on	off	off
1	0	off	on	off	on	on	off
0	1	on	off	off	off	off	on
1	1	on	on	off	on	off	off

Prove your design through simulation.

Exercise 6

Assuming that the board is supplied with a 20MHz clock, create a suitable circuit to divide the clock frequency down to 1 Hz and use this as a clock source for the two-bit counter.

Take this design one stage further by assigning pins for the hardware and downloading the design into the FPGA.

Introduction to 'top-down'

In the previous section, we looked at building Quartus projects 'bottom-up', starting from individual design elements, assigning symbols for each of these and then tying the whole design together in a top level design.

In this section, we examine the other approach – 'top-down'. In this, we start by creating the topology of the design and then develop circuitry for each of the blocks in it.

This section does not contain step-by step instructions for the project. If you find progress difficult, then go back to previous sections and practise your design building there first.

It is recommended that, in addition, you take a look at Altera's own tutorial 'My First FPGA Design', which offers a very good overview of the 'top-down' approach. (This is available to download from the Altera website.)

Top down approach

In this section, we examine:

   the top down approach to logic design;
   how Quartus assists the design of more complex projects;
   how to incorporate 74xxx elements into your design.

This project starts to demonstrate the real benefits of FPGAs. For projects consisting of only a few flip-flops and some discrete logic, the benefits of FPGAs are slim. They come into their own when projects become larger and more complex. This project will give you an appreciation of how and why.

From now on, we assume that you have got to grips with the Quartus interface and the way Quartus works. We no longer take you through menus step-by-step. Instead, we concentrate on explaining the systems in place.

A counter design

Load the project file `two_digit_counter.quartus`.

This is quite a large project, which compiles into 44 macrocells. It is structured in a way that allows for easy design generation and updating.

Make sure the Project Navigator window is open, and explore the elements in the project. The top level entity is `two_digit_counter.bdf` which looks like:
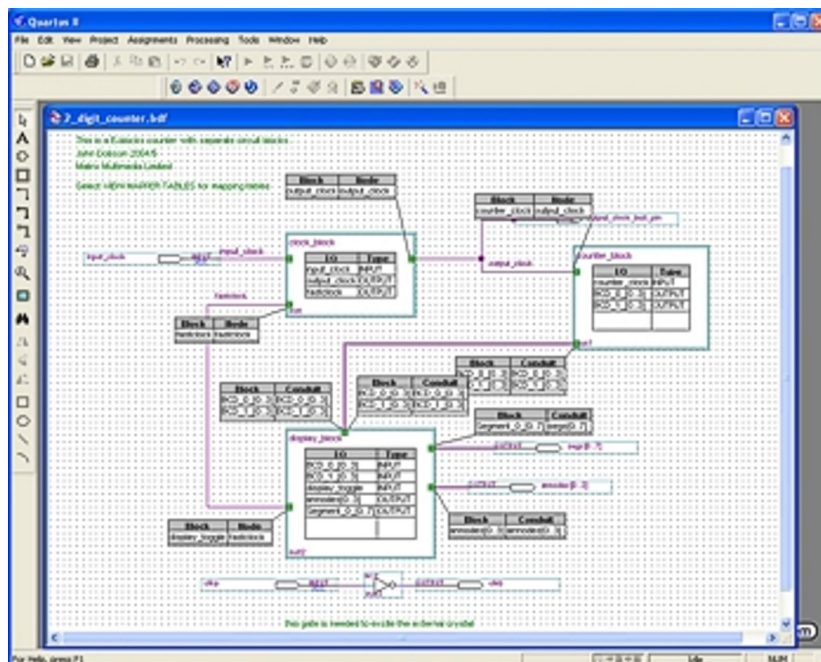
Fig. 5.6: Counter design.

Notice that:

- The whole project is contained under the BDF file 2_digit_counter.bdf .
- In 2_digit_counter.bdf, you can see that the project consists of three sub elements: display_block, counter_block, and clock_block.
- In the 2_digit_counter.bdf file,we have used the shorthand wiring notation '[0..7]' to allow us to wire buses together easily.
- This shorthand notation can also be used for input and output pins to avoid cluttering up your designs.
- When wiring blocks together with conduits, you need to specify the bus lines in the conduit and how these connect to the block elements at each end of the conduit.
- The easiest way to access any part of the design is to double click on each block to reveal the sub circuit it represents.
- This block approach allows newcomers to the design to see how the design works.
- Where a conduit enters a block, there is a small green symbol. Right mouse clicking on this block allows you to map the conduit connections to the block connections.
- There is an option to allow you to view/hide conduits. Check your VIEW menu if you are having trouble here.

- The naming of all conduits and signals is meaningful to aid understanding
- The pin assignment editor knows when you have used an input and output bus and puts pin connections on automatically to aid debugging
- In display_block, and counter_block we have used standard 74xxx parts.
- Where you come across a 74xxx part, you can double click on it in the Project Manager window to see its elemental form. These are 'off the shelf' parts supplied by Altera and can be found in the OTHERS… section of the symbol library.
- The disadvantage of using 74xxx components is that relating the datasheet to the symbol can be tricky.

Top down design

The 'top-down' design method is important for several reasons:
- it encourages you structure the design in sections;
- it allows a number of designers to work on a project at the same time, each being responsible for one of the sections of the design;
- it is easier to debug;
- it facilitates development in manageable 'chunks', likely to be reusable in future designs;
- it is easier to document.

What's in the design so far?

The design files consist of:
- Device design files
- Counter_block.bdf
- 2_digit_counter.bdf
- clock_block.bdf
- display_block.bdf
- \..\..~\7447.bdf
- \..\..~\7490.bdf
- Software files
- none

Fill in the blanks

The project in the directory 'Partial_Two_Digit_Counter on the CD ROM is a variation of the 2_digit_counter project that has had the clock block removed.
You need to:
- copy this directory to an appropriate place on your C: drive;
- create a new block to replace the clock circuitry;
- edit the top level block design file (Two_digit_counter.bdf) to include this new file;
- add output pins for the dual 7-segment display;

- recompile the design;
- verify (through simulation or otherwise) that the design works.
- There are several ways in which you can replace the clock circuit:
- you can design a clock circuit using individual flip-flops;
- you could use a 74 series counter;
- you could try using VHDL, (if you were feeling ambitious).

Note - you can copy the file `clock_block.bdf` from the original `Two_bit_counter` directory if you want to save time!

You should practice creating the `clock_block.bdf` from the block symbol if possible.

Light-chaser - Megafunctions

Megafunctions are modules, designed by Altera for use in FPGA designs. They are installed by default in the QuartusII suite, in the '`libraries/megafunctions`' folder, when you down load the Quartus software. Their function is to save the user time and effort when creating a FPGA design.

We will access them using the 'MegaWizard Plug-in Manager'.
The task is to design a light-chaser, where one LED is lit, and then a short time later, the LED to its left is lit, and so on. This behaviour repeats over and over again.

The design requires a clock signal to make the light-chaser pattern move to the next stage. We will use a phase-locked loop (PLL), configured by the 'MegaWizard Plug-in Manager'. The clock frequency from this will be too high, so we follow it with a counter, whose role is simply to generate a lower frequency clock signal.

It is followed by a shift register. In this, the stored data moves one position to the left on each clock pulse. Its outputs will be connected to the LEDs, and will show the light-chaser effect.

Assuming that the QuartusII program is already running, start the new design as follows.

- Use the New Project Wizard to create a project called chaser, in the directory .../chaser, with a top-level file called chaser_top.
- Assign the design to the correct FPGA device - the EP4CE10E22C8.
- Create a new Block Diagram/Schematic file, and save it as chaser_top.

Light-chaser - Using the Megafunctions PLL

You just created the chaser_top.bdf file. The task now is to add the components described in the previous section.
First, the phase-locked loop:
- Click on the 'Symbol' icon and then on the 'MegaWizard Plug-in Manager...' button.

The first page of the wizard appears.

The action to create a new megafunction is already selected, so click NEXT.

On page 2a of the wizard:
- expand the 'I/O' folder by clicking on the '+' next to it;
- select the megafunction called 'ALTPLL';
- check that the 'Cyclone IV E' family is chosen;
- select 'Verilog HDL' as the output file type;
- name the output file by adding pll on the end of the directory name given;
- click NEXT.
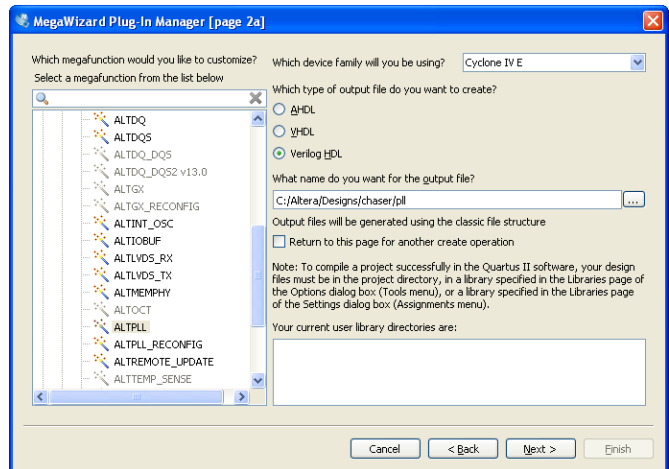
On page 3:
- confirm that the 'Currently selected device family' is still Cyclone IV E;
- leave the selected speed grade as '8';
- change the frequency of the 'inclk0' input to 5 MHz;
- leave 'PLL type' and 'Operation mode' at their default settings;
- click NEXT.
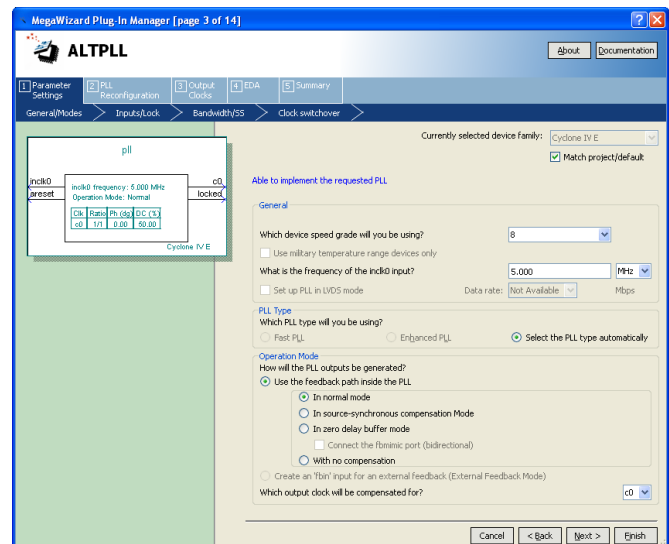
On page 4:
- uncheck all options;
- click NEXT, and then click tab 3 'Output Clocks', which takes you to page 8.
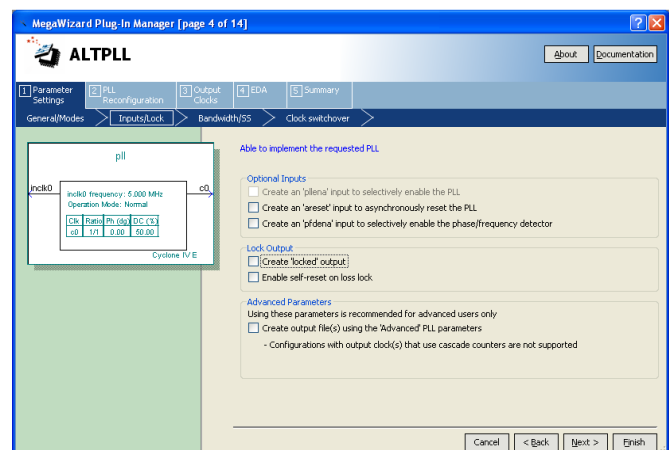
On page 8:
- change the 'clock division factor' to 1000;
- leave all other settings unchanged;
- click FINISH, (as the remaining pages do not require attention.
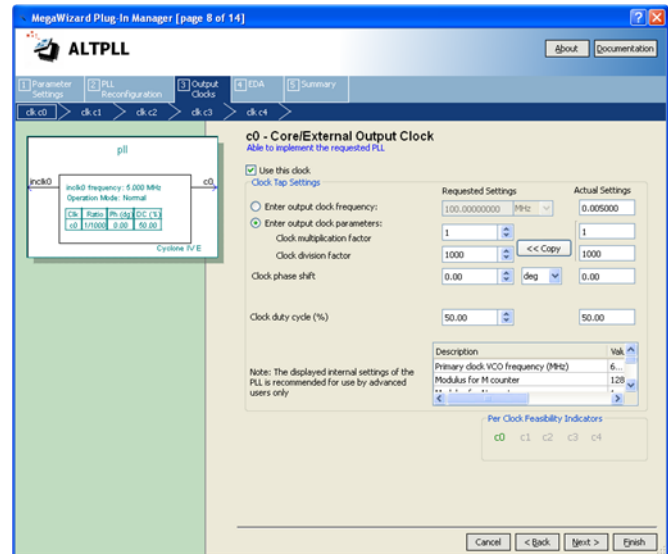
- The wizard displays a summary of the files it has created:
- click FINISH again.
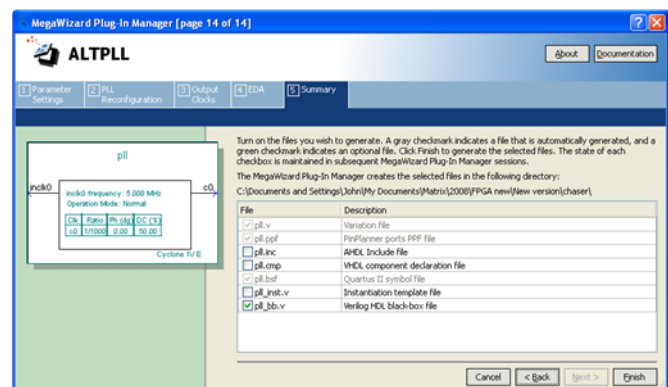
Quartus generates a symbol for the phase locked loop:
- click OK;
- drag an image of the symbol onto the main workspace, and click again to release it;
- press escape to prevent any further instances of the image.
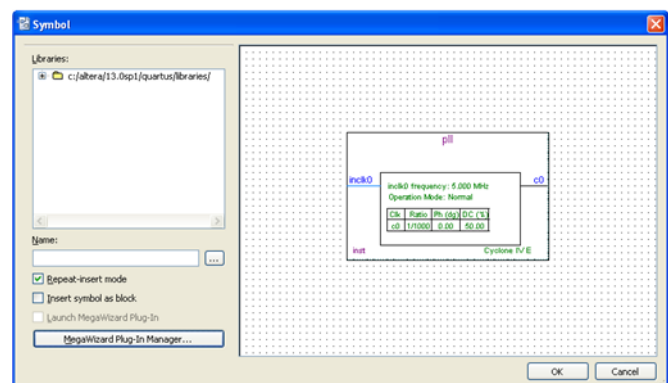
Light-chaser – Verilog counter

The output clock frequency from the phase-locked loop is too high for our purposes. We are going to follow it with an 8-bit counter, purely to lower that frequency.

The method chosen is to write the code for the counter in Verilog. (VHDL could have been used equally well.)

To do this:

- create a new file, and choose a Verilog HDL Design File, (FILE / NEW / VERILOG HDL..);
- save it as counter.v; (FILE / Save As...) and then click SAVE;
- type the following code into the main workspace for the Verilog file:

```
//This counter has a single clock input and an 8-bit output port
module counter (input clock , output reg [7:0] count_out);
always @ (posedge clock)
begin
     count_out <= #1 count_out + 1;
end
endmodule
```

- save the Verilog file, (FILE / SAVE);
- click on FILE / CREATE/UPDATE / CREATE SYMBOL FILES FOR CURRENT FILE;
- click on the chaser_top.bdf tab, and then on the 'symbol' icon, ⊳ ;

- click on the counter symbol, in the Project Library;
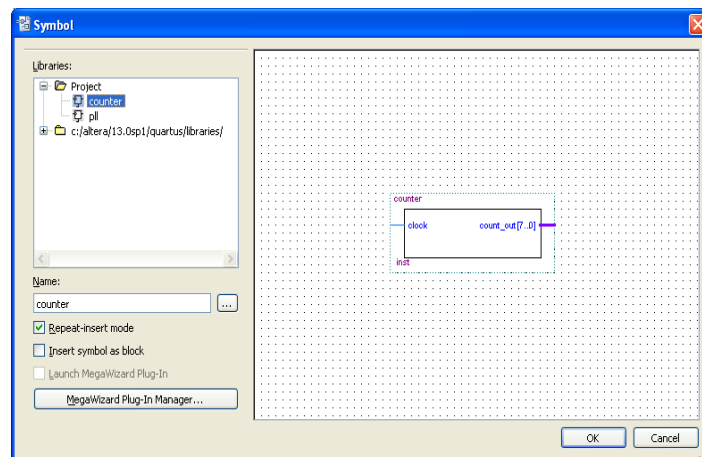
- click OK, and place the symbol so that its clock input is in line with the 'c0' output of the phase-locked loop;
- Save the project, (FILE / SAVE PROJECT).

Light-chaser - shift register

The next item in the design is the shift register. This is created using the 'MegaWizard Plug -in Manager...'. The sequence is like that used to create the PLL.
However, this time:

- on page 2a of the wizard, open the 'Memory Compiler' and select 'LPM_SHIFTREG'. Add "shift" to the end of the directory name for the file, and click NEXT.

on page 3, make the selections shown:
- an 8-bit 'q' output;
- shift left;
- a data output and a serial data output;
- a serial data input;
- click NEXT.
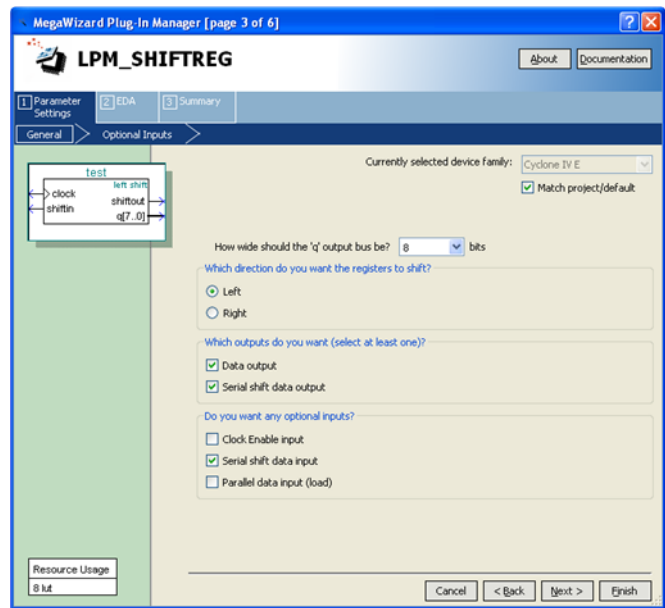
on page 4:
- add an asynchronous 'set' input;
- configure it to set the data to '1';
- click FINISH.
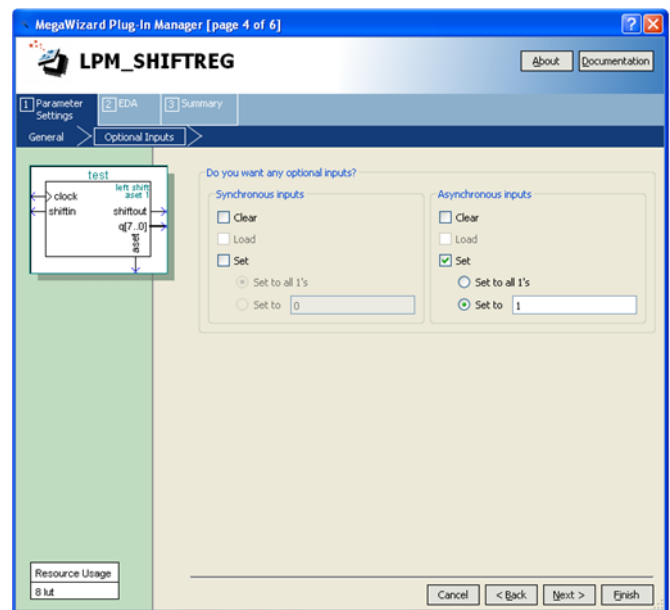
- The wizard then displays a summary of the files created:
- Click FINISH.
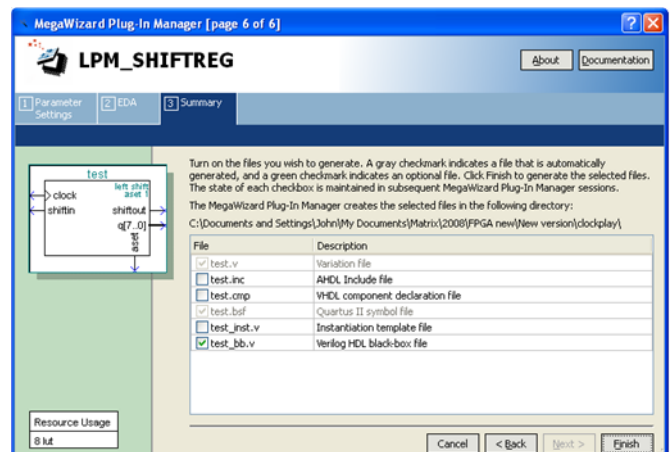
Quartus generates a symbol for the shift register:

- click OK;
- drag an image of the symbol onto the main workspace, and click again to release it;
- press escape to prevent any further instances of the image.
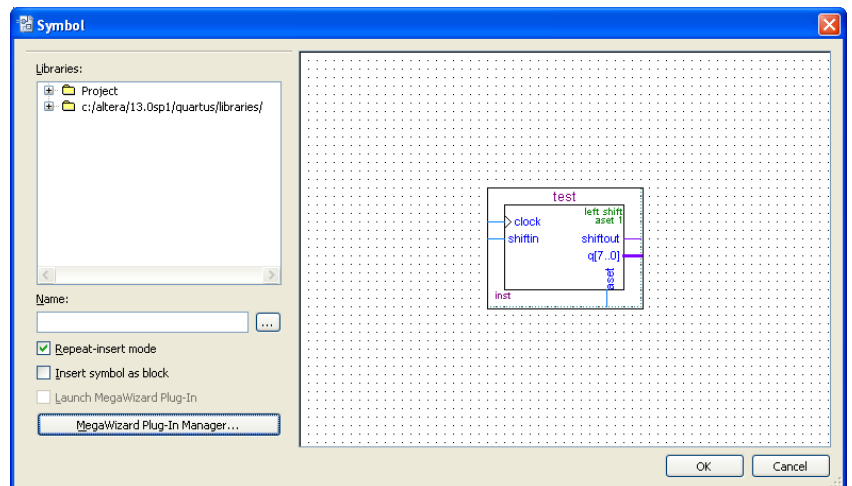
Light-chaser - finishing touches

Finally:

make the additions shown in the diagram:
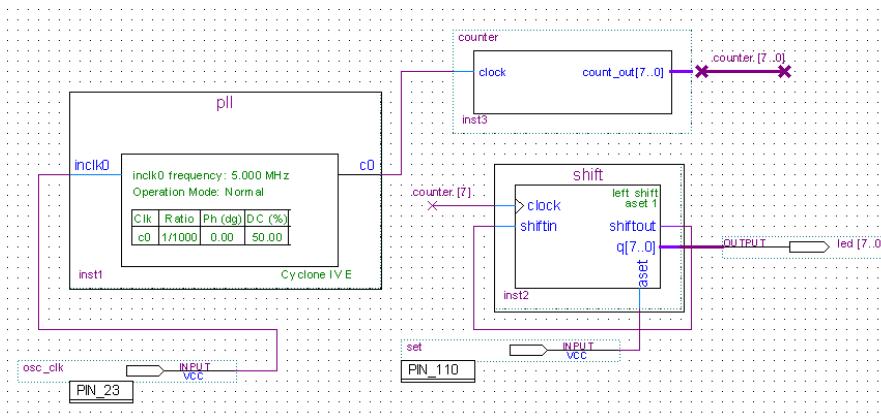
- move the symbols into the positions shown;
- rename them inst1, inst2 and inst3, by double-clicking on each label in turn;
- add two input pins, and change their names to those shown;
- add an output pin, and name it led [7..0] . This creates an 8-bit wide bus to connect to the LEDs;
- hover over the output of the counter and draw a bus line from it, right-click on it to open its properties, and name it counter [8..0];
- hover over the shift register clock input, add a connection, and name it counter [7]to select the most-significant bit (lowest frequency) of the counter output;
- connect the serial output of the shift register to its serial input, so that the pattern repeats;
- make the other connections shown.
- Run the 'Analysis and Elaboration' tool.
- Use the 'Pin Planner' to allocate the pins shown to the inputs and outputs. The LEDs are connected via FPGA pins on Port C (see the table on page 22).
- Compile the design.
- Run the programmer and download it to the FPGA.
- Test the design:
- press switch SW0 to set the LEDs to show the binary number 00000001;
- release the switch to see this pattern shift to the left repeatedly, and eventually re-enter at the right-hand end of the LED board.

Chapter 6: Descriptor languages

Introduction

In this section, we look at both Verilog and VHDL. If you intend to use a descriptor language in developing further projects, then it is likely that you will make a choice to either learn VHDL or Verilog. Fortunately, Quartus Web Edition can cope with either, so you are not limited by the software. So which do you choose?

Your choice is likely to be dictated by factors other than technical issues. As far as we are aware, there is little difference in terms of performance between Verilog and VHDL. There is little difference in terms of difficulty in learning them. Other factors, like the language of choice of your company, or the experience of others around you, are likely to be more important.

Historically VHDL tends to be the design language of choice for European countries, whereas Verilog tends to be used more in North America.
In the following sections, we include design examples in both languages. We suggest you study this section for both Verilog and VHDL so that you have an appreciation of both languages, to make an informed choice of one language for the rest of the course.

Verilog design entry

In this exercise, you replace a schematic design file with an equivalent Verilog one.

In outline, you are going to:

1. run Quartus;

2. open an existing project;

3. create a new Verilog design file and add it to the project;

4. remove the schematic design file;

5. compile the design;

6. download the design and test it.

Step 1: run Quartus
If it's not already running, run Quartus Web Edition.

Step 2: open existing project
Open the project `Three_Input_AND` which consists of a simple three input AND gate.

You may want to copy the project directory to a new location, to prevent overwriting the original files.

Step 3: create a new Verilog design file
Click the ⬜ button (or File\New.) then select Verilog HDL file.

Type (or COPY and PASTE) the code given here (but omit the comments, that follow the '/' symbol).

```
module Three_Input_AND (Input_A, Input_B, Output_A, Output_B, Output_Q);
  // declare inputs and outputs
  input Input_A, Input_B;
  output Output_A, Output_B, Output_Q;

  // define how the design works
  assign Output_A = Input_A;
  assign Output_B = Input_B;
  assign Output_Q = Input_A & Input_B;
endmodule
```

Use FILE\SAVE AS to save your new file as Three_input_AND_verilog.v in your first folder.

Let's look in some detail at this listing.

Notice that Verilog keywords become coloured blue as you have type them in. Make sure to type them in lower case - Verilog is case sensitive.

The first line has the essential keyword module followed by the name for the module, and by the list of all input and output signals, enclosed in parentheses. A module is a self-contained listing that describes some hardware. All Verilog designs start with this keyword. The end of the design is indicated by the endmodule keyword.

The module name should match the project top-level entity name, since this module will be the top (only) file in the project hierarchy. The module name can be as long as you like, but must not have any spaces or punctuation, or use & signs, etc. Stick to letters, numbers and the underscore character. Don't start with a number!

On the next two lines the input and output signals are 'declared'. This is required by the language, and helps the software catch design errors. For instance, the compiler will not allow you to send output data to a signal that has been declared as an input. Signal names follow the same rules as module names. In this text, upper and lower case letters have been used to help readability, but remember that Verilog is case sensitive. Signal Input_A is a different one from signal Input_a .

The assign statements show how data flows through the module. Outputs are placed to the left of the = sign. Thus, assign Output_A = Input_A; means that signal Output_A is connected to, and receives data from, signal Input_A .

The & symbol indicates that signal Output_Q gets its data by AND-ing signals Input_A and Input_B together.

The Boolean operation symbols are shown below.

AND	&
OR		
NOT	!
XOR	^
XNOR	~^

Fig. 6.1

There are no special symbols for NAND or NOR. You would implement NAND, for example, by using assign Output_Q = !(Input_A & Input_B);

Each statement in Verilog is completed with a semicolon. The last line of the listing, however, has no punctuation.

Complete step 3 by choosing a different Boolean function from the last one you used in your schematic-based design. This way you will be sure that your design really has changed!

Step 4: reorganize the project files
In order to change the design from `3_input_AND.bdf` to `3_input_AND_verilog.v`, you need to remove the former from the project, and add the latter.

Use the Settings button [icon] or the Assignments/Settings menu to open the Settings dialogue window. Click the Files category then use the Add and Remove facilities to ensure that the only file in the project is `Three_input_AND_verilog.v`

Note - the Project Navigator window will still show the bdf file in the hierarchy. However, the correct file will show after the compiler has run - see the next step.

Step 5: Compile
This uses the same process as before. The compiler settings file (`Three_input_AND.csf`) still has the pin location information, so you don't need to enter the pin connection details again.

Run a full compilation of the design.

If you made any mistake entering the Verilog text, the compiler will probably fail to complete, and some red error messages will be written to the Messages window at the bottom of the Quartus screen. Some description of the error will be given, together with the number of the offending line. If you double-click the error message you will be taken to the relevant line within the text file editor.

Examine the text carefully, looking for the mistake. When you find it, correct it and run the compiler again. Enter a deliberate mistake, just to check out this facility!

Step 6: Download and test
Connect the target board to the USB port of the computer and apply power to it.

Click the [icon] button (Tools/Programmer) to run the programmer software. Tick the Program/Configure and Verify check boxes and then press [icon] the button to start programming.

Once programming is finished, manipulate switches SW7 and SW5 to check that your new Verilog design behaves as expected.

Automatic conversion of schematic to HDL

The Quartus software provides a means of converting a schematic design (BDF) into an HDL design.
You can find this under the File/Create/Update... menu item. Then choose Create HDL Design File for Current File. This works only if the current file is a schematic, of course.
Once you click this option, a further dialogue box pops up, from which you can choose VHDL or Verilog.
You may choose to return to the BDF file and use this facility to create a Verilog version of

the design. However, you may want to rename your existing Verilog design file to prevent it from being overwritten by the one which Quartus will generate.
The layout of the automatically generated file is slightly different from the one given above, but there are no other differences.

Exercises

Once you have a Verilog design that compiles and downloads correctly, it is easy to modify it.
Try altering the behaviour to NAND, XOR, or some other logic function.
Edit your Verilog code, run a full compilation, download and test.

VHDL design entry

In this exercise, you replace your existing schematic or Verilog design file with an equivalent VHDL one.
In outline, you are going to:

    run Quartus;
    open an existing project;
    create a new VHDL design file and add it to the project;
    remove the schematic or Verilog design file;
    compile the design;
    download the design and test it.

Step 1: run Quartus
If it's not already running, run Quartus Web Edition.

Step 2: open existing project
Open the project `Three_input_AND` which consists of a simple three input AND gate. You may want to copy the project directory to a new location, to prevent overwriting the original files.

Step 3: create a new VHDL design file

Click the [ ] button (or File\New.) then select VHDL. Type the code given below
(but omit the comments given after the '--' symbols).

```vhdl
ENTITY Three_input_AND IS
PORT
(
    -- declare inputs and outputs
    Input_A, Input_B, Input_C:                       IN BIT;
    Output_A, Output_B, Output_C, Output_Q:     OUT BIT
);

END Three_input_AND;

ARCHITECTURE how_it_works OF Three_input_AND IS
BEGIN
    -- define how the design works
    Output_A <= Input_A;
    Output_B <= Input_B;
    Output_C <= Input_C;
    Output_Q <= Input_A AND Input_B AND Input_C;
END how_it_works;
```

Use File\Save As.to save your new file as `Three_input_AND.vhd` in your first folder.

Let's examine this listing.
The first part of the file defines the connections (ports) to and from the design.
The words ENTITY, IS, PORT, IN, OUT and END are VHDL reserved words and the editor
automatically colours them blue. They are given here in capitals simply to emphasize them
as keywords.

Note that the name of the entity needs to match the name of the file itself
(`Three_input_AND.vhd`). The entity name can be as long as you like, but must not have
any spaces or punctuation, or use & signs, etc. Stick to letters, numbers and the
underscore character. Don't start with a number!

Ports are defined by giving a list of port names, then a colon, then the mode (IN or OUT)
then the signal type (BIT). All this is a requirement of the VHDL language, and is there to
help the software catch design mistakes. For instance, the software will not allow you to
send data to a signal that has been declared as an IN, or to set a BIT type signal to any
value other than '1' or '0'. Other data types will be introduced later, when they are needed.

Identifier names for ports follow the same rules as for entities. In this text, upper and
lower case letters have been used to help readability, but remember that VHDL ignores
case. Input_A is the same as Input_a .

Note the absence of a semi-colon on the last port definition before the closing bracket, and the presence of one after the closing bracket. You need to type the examples very carefully to get all the semi-colons in the right place!

The second part defines the operation of the design. The words ARCHITECTURE, OF, BEGIN and END are more VHDL reserved words. The name how_it_works is user supplied. The symbol <= ('gets') shows what is connected to what.

The reserved word OR is used to define an OR function. Other Boolean operators are AND , NAND , NOR , XOR , XNOR and NOT .

To complete step 3, choose a different Boolean function from the last one you used so that you can be sure that your new VHDL design really is different from the one already programmed into the CPLD device!

Step 4: reorganize the project files

In order to change the design from `Three_input_AND.bdf` to `Three_input_AND.vhd` you need to remove the former from the project and add the latter.

Use the Settings button [✎] or the Assignments/Settings. menu to open the Settings dialogue window. Click the Files category then use the Add and Remove facilities to ensure that the only file in the project is `Three_input_AND.vhd`

Note that the Project Navigator window will not update yet but the correct file will show in the hierarchy after the compiler has run - see the next step.

Step 5: Compile

This uses the same process as before. The compiler settings file (`Three_input_AND.csf`) still has the pin location information, so you don't need to enter the pin connection details again.

Press the [🗐] button (Tools/Compiler Tool) followed by the Start Compilation button, to run a full compilation.

If you have made some mistake entering the VHDL text, the compiler will probably fail to complete, and some red error messages will be written to the Messages window at the bottom of the Quartus screen. Some description of the error will be given, together with the number of the offending line. If you double-click the error message, you will be taken to the relevant line within the text file editor.

Examine the text carefully, looking for the mistake. When you find it, correct it and run the compiler again.

Enter a deliberate mistake, just to check out this facility!

Step 6: Download and test

Connect the target board to the USB port of the computer and apply power to it. Click the [🖑] button (Tools/Programmer) to run the programmer software.

Tick the Program/Configure and Verify check boxes and then press the [▶] button to start programming.

Manipulate switches SW7 and SW5 to check that your new VHDL design behaves as expected.

Automatic conversion of schematic to HDL

The Quartus software provides a means of converting a schematic design (BDF) into an HDL design.

You can find it under the File/Create/Update... menu item, then choosing Create HDL Design File for Current File. This only works if the current file is a schematic, of course. Once you click the option, a further dialogue box pops up, from which you can choose VHDL or Verilog.

You may choose to return to the BDF file and use this facility to create a VHDL version of the design. However, you may want to rename your existing VHDL design file to prevent it from being overwritten by the one which Quartus will generate.

The automatically generated file is slightly different from the one given above. Instead of using BIT type signals, STD_LOGIC type have been introduced. This requires that libraries are used, and these are declared on the first few lines of the design listing.

The other difference is that the software has chosen a different name for the architecture. Apart from changes to the layout, there are no other differences.

Exercises

Once you have a VHDL design that compiles and downloads correctly, it is easy to modify it. Try altering the behaviour to NAND, XOR, or some other logic function. Edit your VHDL code, run a full compilation, download and test.

Take the code in the example above and start a new project in Quartus. Here are a few hints and reminders:

Remember that Verilog is case sensitive, (and VHDL is not!) If you encounter problems, make sure that you are spelling files and declarations with the correct case.

Make sure the MODULE/ENTITY declaration name is the same as the top level entity.

Remember that you can't declare pins before you have compiled!

Summary so far

In this section, you have seen how a High Level Descriptor language can be used for designing digital electronics systems in a FPGA.

You should now be able to make a choice of studying further using either Verilog or VHDL, and in the sections that follow we will build on your knowledge of these languages.

Please note that you should now make a choice of studying either VHDL or Verilog. All future exercises will be given in both languages, but we recommend that you study only one.

Chapter 7: Behavioural descriptions

Introduction

Many syllabuses in Digital Electronics emphasise the use of simulation software to gain an insight into the behaviour of digital circuits. Whilst the simulation tools of Quartus give an excellent representation of the waveforms expected in a FPGA-based circuit, other SPICE-based circuit simulators, like Proteus, Tina or Multisim, give more interactive animations, and can simulate a wider variety of circuits.

The aims of this chapter are:

  to introduce you to more complex digital electronics circuits

  to reinforce your understanding of how these circuits work using a traditional SPICE simulator

  to show you how you design these circuits using the Verilog and VHDL descriptor languages.

An additional aim is to introduce the idea of a 'behavioural' description of a circuit in Verilog and VHDL. When defining a circuit, it is possible to describe just the overall action of the circuit, without having to specify the detailed internal logic. This feature gives the descriptor languages a powerful advantage over schematic entry when designing more complex circuits.

Topics covered in this chapter

On completing this chapter you will be able to:

  draw and simulate basic digital circuits using your circuit simulator;

  investigate the behaviour of a de-multiplexer using your simulator;

  use behavioural terms in Verilog and VHDL to describe hardware.

The new Verilog constructs in this chapter are:

- always
- if ... else ...
- reg
- begin ... end
- The new VHDL constructs in this chapter are:
- WHEN ... ELSE ...
- PROCESS
- IF ... THEN ... ELSE ...

What is a behavioural description of hardware?

Suppose that you want a circuit that allows a signal A through to its output Q if, and only if, a third signal 'Control' is HIGH. Otherwise you want Q to stay LOW .

This is a behavioural description of the design: "Q receives signal A if Control is HIGH , otherwise it stays LOW ." This kind of language is allowed in Verilog and VHDL, and saves you the effort of working out how to build such a circuit.

A solution to this design problem is a simple AND gate, of course. A 'circuit diagram' or 'structural' description of the hardware would be: "Q is output of an AND gate which has

inputs A and Control".

For simple circuits the behavioural description may take up more lines of code than the structural description. For complicated circuits, being able to describe just the behaviour saves a lot of time and effort over having to work out a circuit solution. It is one of the main reasons for using a hardware description language rather than drawing circuit diagrams.

Using a simulator to investigate the behaviour of a two-way demultiplexer
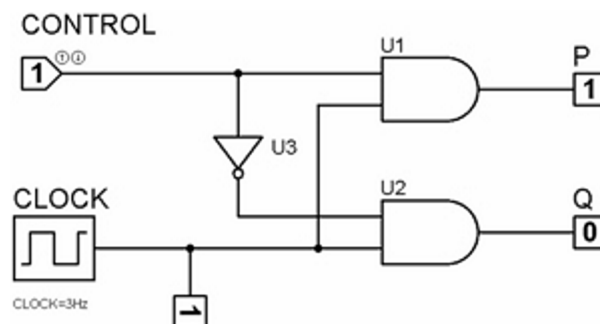
Fig. 7.1: Two-way demultiplexer circuit.

Run your circuit simulator and enter the design shown in the diagram above.

The CLOCK should generate a square wave signal that goes HIGH and LOW three times a second.

Make sure you understand the operation of this circuit, as we will be looking at the Verilog and VHDL equivalents in the next few screens.

Behaviour: the circuit sends the CLOCK signal to output P if the CONTROL signal is high but to output Q if it's LOW .

Verilog versions of the two-way demultiplexer

'Circuit diagram' version

The code below gives the Verilog equivalent of the circuit diagram.

```verilog
module two_way_dmpx (clock, control, P, Q);
    input clock, control;
    output P, Q;

    assign P = control & clock;
    assign Q = !control & clock;
endmodule
```

'Behavioral' version

The equivalent code written in 'behavioral' language is:

```verilog
module two_way_dmpx (clock, control, P, Q);
    input clock, control;
    output P, Q;
    reg P, Q;

    always @ (control, clock)
    if (control == 1)
    begin
        P = clock; Q = 0;
    end
    else
    begin
        Q = clock; P = 0;
    end
endmodule
```

This 'behavioural' version of the code reflects the description given above Figure 2.1: "
Behaviour: you should find that the circuit will send the CLOCK signal to output P if the
CONTROL signal is high but to output Q if it's LOW ."
In fact, this description is ambiguous since it doesn't say what happens to the output that
doesn't get the CLOCK signal. The assumption is that it goes LOW .
In Verilog (and VHDL) you have to be specific about this kind of thing. Hence you have to
say, "If the CONTROL is HIGH then the CLOCK signal goes to P (and Q goes LOW)
otherwise the CLOCK goes to Q (and P goes LOW)".
Note the use of the double-equals sign on the sixth line of code. This shows the condition
that is tested in the if (<condition>) ... else ... clause. In this case, it tests whether the
CONTROL signal is equal to logic 1. Notice that the condition itself has to be enclosed in
brackets.
The begin and end keywords are required to 'bracket' statements together.
The if construct has to be placed within an always @ structure. This structure is really
designed to deal with sequential logic designs, and a consequence of this is the need to
declare signals P and Q as reg (register), even though no registers will be used.
The 'Event Control' list following the always @ keywords must contain all the signals that
can affect the outputs.

VHDL versions of the two-way demultiplexer

Circuit-diagram version

In VHDL the 'circuit diagram' mode of the design would be:

```
ENTITY two_way_dmpx IS
PORT
(
    control, clock: IN BIT;
    P, Q:           OUT BIT
);
END two_way_dmpx;

ARCHITECTURE data_flow OF two_way_dmpx IS
BEGIN
    P <= control AND clock;
    Q <= NOT control AND clock;
END data_flow;
```

'Behavioral' versions

There are two ways of writing 'behavioral' VHDL code for the design. One would be:

```
ENTITY two_way_dmpx IS
PORT
(
    control, clock:  IN BIT;
    P, Q:            OUT BIT
);
END two_way_dmpx;

ARCHITECTURE behavior_a OF two_way_dmpx IS
BEGIN
    P <= clock WHEN control = '1' ELSE '0';
    Q <= clock WHEN control = '0' ELSE '0';
END behavior_a;
```

The code says that output P gets the clock signal when control is '1' (HIGH) , but Q gets it when control is '0' (LOW) .

Notice that you need single quotation marks around the 1 and the 0 . If you leave them out, the values 1 and 0 would be considered to be integers, and BIT-type signals such as P and Q cannot take integer values. VHDL is very strict about such rules.

The other point to make is the need for the ELSE clause for each signal. It is easy to assume that output P will 'naturally' go LOW when the control signal ceases to select it. In VHDL, the assumption is that P will stay in its last state, which could be HIGH or LOW. The ...ELSE '0'; part of the instruction explicitly states that signal P should go LOW when it is no longer selected.

The second way in which VHDL can describe the behaviour of the two-way demultiplexer circuit is with an IF construct. The way the language works means that this has to be placed within a structure called a PROCESS . As far as the current example is concerned, a 'process' is just a chunk of code that can contain an IF construct.

```
ENTITY two_way_dmpx IS
PORT
(
   control, clock:    IN BIT;
    P, Q:               OUT BIT
);
END two_way_dmpx;

ARCHITECTURE behavior_b OF two_way_dmpx IS
BEGIN
    PROCESS (control, clock)
    BEGIN
        IF control = '1'
        THEN
            P <= clock;
            Q <= '0';
        ELSE
            P <= '0';
            Q <= clock;
        END IF ;
    END PROCESS ;
END behavior_b;
```

The PROCESS keyword is followed by a list of signals that can affect the process. This 'sensitivity list' must contain all the relevant input signals if you are designing a combinational logic circuit.

As in the previous design, you have to be explicit about the behaviour of all signals under all conditions - what should Q do when P has the clock, and vice versa.

Indentation of code

It is very good practice to pay attention to the layout of your code. It makes it look as if you know what you are doing! Indentation is purely an aid to readability, to show the structure of the design, but try to be consistent and use indentation intelligently.
The style adopted in these notes is:
  keywords such as BEGIN , THEN and ELSE get their own line;
  the line underneath them is indented (two columns);
  ELSE and the various END lines are 'outdented' (two columns);
  all other lines line up with the one above.
Using this scheme all the ENDs line up vertically with their respective BEGINs, or equivalent keyword, and instructions are grouped together clearly.

Difficulties with Quartus

When starting a new Quartus project, with a VHDL or Verilog file as the top level entity, you may encounter an error message "node instance not instantiated". We believe this is a bug within Quartus.

The solution is either to create temporarily a BDF as the top level entity, or to place your VHDL/Verilog file in a project that has already compiled successfully. We have created a `3_input_voter` project which you can use as a starting point. You will see this project in the next section.

Quartus simulation of two-way demultiplexer

The results of simulating any of the Verilog or VHDL designs, given earlier, are shown below.
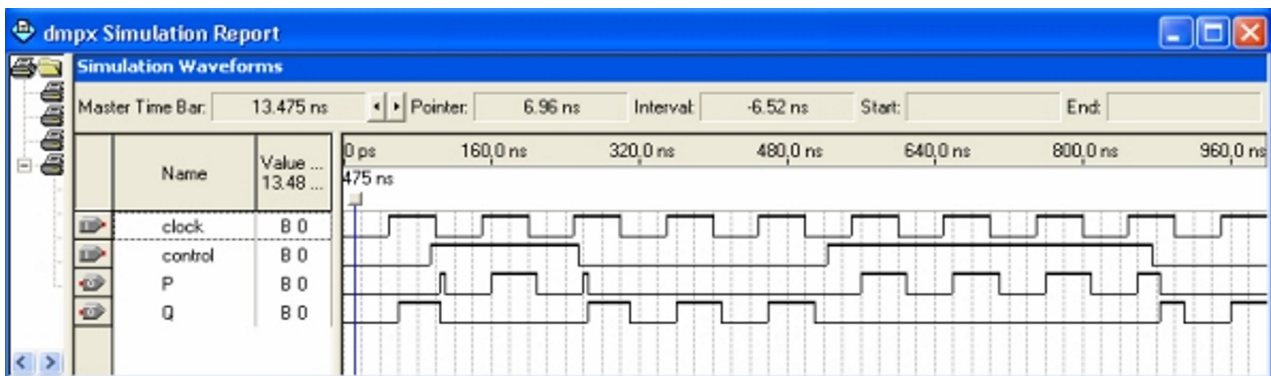
Fig. 7.2: Simulation result for two_way_dmpx design.

To begin with, Q gets the clock signal because control is LOW .
Later, however, when control goes HIGH , P gets it.
At times when they are not receiving the clock signal, P and Q go LOW .

Exercise 1: build a two-way demultiplexer

OK - you've read all about it. Now use Quartus to create a new project and use one of the five designs given above to turn the E-blocks FPGA board into a two-way demultiplexer.

Use one of the buttons on the switch E-block as a clock signal, and another for the control. Use LEDs on the LED E-block to show the status of outputs P and Q.

Go back to earlier chapters if you need to remind yourself on how to use Quartus. For the simulation part of the design process, note that the VWF (Vector Waveform File) comprises just the clock and control signals. Create the clock waveform as described in previous chapters using the Overwrite Clock with a period of 100ns.

For the control signal use the waveform-editing tool -   just slide this tool along the waveform to move the edges to where you want them. Remember to save the VWF file, and give the file name as the input for the simulation run.

Exercise 2: design a two-way multiplexer
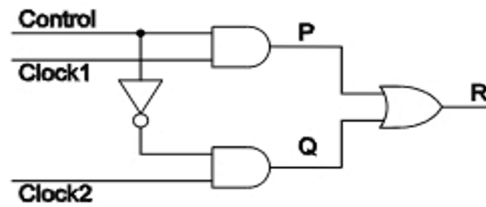
Fig. 7.3: Two-way multiplexer.

The job of a multiplexer is to choose one input signal from several, and pass it through to its output.

In this example output R will receive Clock1 when the Control is HIGH, but Clock2 when Control is LOW . You may want to use a circuit simulator to verify this action, setting Clock 1 to 1Hz and clock 2 to 3Hz.

Now design a Verilog and/or VHDL solution to fit the chip on the FPGA board.

A neat solution uses the format of the behaviour OF two_way_dmpx VHDL code, but all five ideas given above will work. Solutions are given at the end of this chapter.

<u>**Summary**</u>

This chapter has introduced a slightly more complex digital circuit, and the use of behavioural language in Verilog and VHDL to describe it.

You have studied a two-way demultiplexer, which takes a single clock signal and sends it to one of two outputs.

You have also studies a circuit which does the opposite - selecting one of two input signals and routing this through to the output.

In the next chapter, several more 'combinational logic' circuits will be introduced, with Verilog and VHDL descriptions to compile and download to the FPGA board.

For these circuits, it is recommended that you also use a traditional circuit simulator to gain an understanding of the circuit before transferring it to VHDL/Verilog.

Chapter 8: Combinational logic using HDL

Introduction

Now that you have learned the basic concepts of logic design using VHDL or Verilog, your next step is to expand your knowledge in terms of the range of circuits, HDL commands and HDL structures.

In this section, we examine the topic of combinational logic in more detail. Again you may wish to use a conventional circuit simulator to draw the circuits and simulate them to aid understanding before proceeding to design it in VHDL or Verilog.

The previous chapter introduced a couple of circuits with special names: a 'multiplexer', and a 'demultiplexer'. They switch one of several input signals to a common output, or switch a common input to one of many outputs.

It is important to know what is meant by such terms, and this chapter introduces more circuits designed to provide solutions to problems that crop up in a variety of designs.

The title of this chapter is 'Combinational logic'. The outputs of combinational logic circuits depend entirely on the combination of HIGH and LOW logic levels present on the inputs. This contrasts with the situation in 'sequential' circuits, where the outputs depend on the past history of the input signals as well as their current states.

All the circuits considered in this chapter are combinational, and the chapter reviews one of the classic ways of simplifying such circuits – the Karnaugh map.

Topics covered in this chapter.

The circuits considered are:

- decoders
- encoders
- comparators
- parity checkers
- gray-to-binary and binary-to-gray converters
- adders, subtractors and multipliers
- There is also an assignment based on driving seven-segment displays.
- Some new Verilog and VHDL constructs are covered:
- bit vectors, such as D[3..0]
- input/output signals, i.e. inout
- the 'case' statement
- the 'for' loop
- the declaration and use of VHDL libraries

First, however, we look at a voting machine.

Design of a 3-input voting machine

A circuit is required whose output goes HIGH if two or more of its three inputs are HIGH. One way to design this is to start by listing all the possible input combinations, and state what the output should do in every case. This is called a *truth table* . This is fine for small designs like this one, but gets a bit tedious when there are a lot of inputs. A circuit with *n* inputs would need a table with 2^n rows. Thus, our 3-input system will need 2^3 (that is 8) rows. A 4-input system would need 16, and so on.

We name the input signals A, B and C, and the output Q.

A	B	C	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Fig. 8.1 - Truth table for voting machine

The Boolean expression derived straight from this table is:
Q = A.B.!C+A.!B.C+!A.B.C+A.B.C
From this we may draw a circuit:
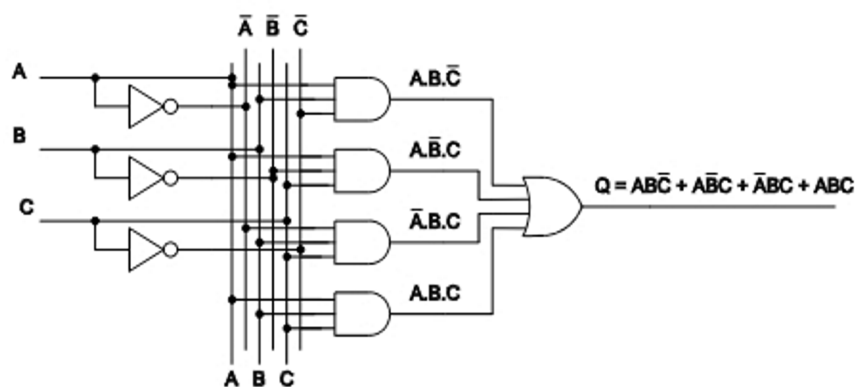
Fig. 8.2: Implementation of Q = A.B.!C+A.!B.C+!A.B.C+A.B.C

The circuit uses inverters to create the 'NOT' versions of the input signals, AND gates to create the AND terms of the expression, and a four-input OR gate to OR the terms together.

Using a Karnaugh map to simplify the circuit

The data from the truth table can be transferred to a Karnaugh map below:

Fig 8.3: Karnaugh map for voting machine.

The method of getting the data from the truth table to the map is quite easy: the AB value gives the position across the map, and the C value gives the vertical location. For instance, the shaded row in the table in the previous section, corresponds to the shaded cell in the Karnaugh map. The ABC value in the table is 011 so, in the Karnaugh map, go across to the 01 column and down to the 1 row. The Q value in the truth table is 1 so that's the value that goes into the cell.
Important! When constructing a Karnaugh map remember to put the numbers around the outside in Gray-code order, not binary. Further details on Gray code are given later in this chapter, but for now just note that the decimal equivalents for the columns are 0, 1, 3, 2.
After completing all the cells we find the 1's are quite nicely bunched together, indicating some simplification is possible. The technique is to group the 1's into rectangular 2's (or 4's if possible), and then read off the Boolean expressions for the groups. In this case the four 1's can be grouped into three 2's as shown below.
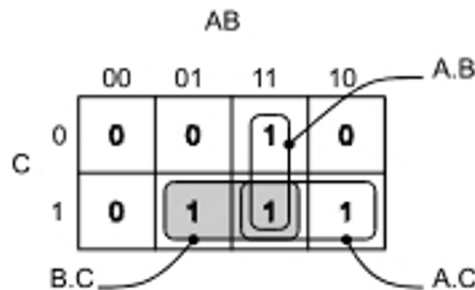
Fig. 8.4: Groupings for voting machine.

In order to attach Boolean expressions to the cell groups, consider which variables define the group (are constant for the group,), and what their values are in the group. In the grey group, for instance, B is HIGH and so is C in both cells making up the group, so the group is represented by the single term, B.C.
The simplified expression for the machine can now be deduced.
It is:
Q = A.B + A.C + B.C
The simplified circuit is thus:
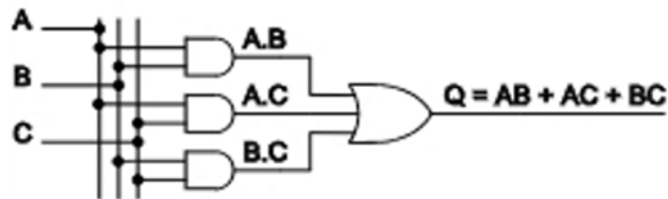
Fig. 8.5: Simplified version of voting machine.

This is clearly a lot simpler than the earlier circuit.

Construct the circuit in your circuit simulator and check it fulfills the requirements of the original truth table.

Verilog implementation of voting machine

The code below gives the un-simplified version of the machine:

```verilog
module voter (A,B,C,Q);
    input A,B,C;
    output Q;

    assign Q = A&B&!C | A&!B&C | !A&B&C | A&B&C;
    // Q shows if two or more inputs are HIGH
endmodule
```

The Quartus software is clever enough to work out that it can simplify the circuitry. If you enter the code and run the synthesize/analyse tool and then take a look at the resulting equations file (file extension .eqn) you will see that the circuit of Figure 8.5 is used.

This code shows how to embed a comment into a listing. The // at the start of a line indicates that what follows on that line is an explanation of what you, as a designer, are trying to do. A multi-line block of comment can be enclosed with /* at the start and */ at the end. Comments are vital if the intention of the design is at all unclear.

There is a structure in the Verilog language that allows you to enter the information in a format that mirrors the original truth table. This is the 'case' statement, and it is used in the example below. Like the 'if' construct, you need an 'always' structure to 'hold' a 'case' statement.

```verilog
module voter (A,Q);
    input [2:0] A;
    output Q;
    reg Q;

    always @ (A)
    case (A)
        3'b 011: Q = 1;
        3'b 101: Q = 1;
        3'b 110: Q = 1;
        3'b 111: Q = 1;
        default: Q = 0;
        // Q shows if two or more inputs are HIGH
```

```
        endcase
    endmodule
```

The 'case' statement has been placed within an 'always @' structure so, as earlier, the output, Q , has to be declared as a 'reg' , even though this is a purely combinational circuit, and no registers are used.

Another requirement is also something seen before. The variables that can affect the output have to be placed in the event list following the 'always @' keywords.

The input variable has been changed from three separate signals (A, B and C) to a 3-bit variable, A. In detail, the three inputs are A[2], A[1] and A[0]. The syntax to declare such a variable is: input [2:0]A; Having done this we can think of A having a value such as binary 101.

In fact, the numbers have been given in 3-bit binary, hence the declarations like 3'b 110. The 'case' statement itself needs the name of the variable, A, (in brackets) immediately after the 'case' keyword, and then the lines underneath set out the effects in a case by case list. The keyword 'default' means you don't have to list every single case; you list the 'interesting' ones, and then say what happens in all other situations.

Entering the design using this technique has a number of benefits:

  the required behaviour is clearer;

  it saves you time;

  it may avoid your making a mistake.

The Quartus software takes care of any possible circuit simplification.

VHDL implementation of voting machine

The code below implements the un-simplified version of the machine:

```
ENTITY voter IS
PORT
(
    A, B, C: IN BIT;
    Q: OUT BIT
);
END voter;

ARCHITECTURE first OF voter IS
BEGIN
    Q <= (A AND B AND NOT C) OR (A AND NOT B AND C) OR ( NOT A AND B AND C)
OR (A AND B AND C);
    --Q shows if two or more of the inputs are HIGH
END first;
```

The Quartus software is clever enough to work out that it can simplify the circuitry. If you enter the code and run the synthesize/analyse tool, then take a look at the resulting equations file (file extension .eqn) you will see that the circuit of Figure 8.5 is used.

This code shows how to embed a comment into a listing. The -- at the start of a line

indicates that what follows on that line is an explanation of what you, as a designer, are trying to do. (A multi-line block of comment requires the -- at the start of every line.) Comments are vital if the intention of the design is at all unclear.

There is a structure in the VHDL language that allows you to enter the information in a format that mirrors the original truth table. This is the 'CASE' statement. Like the 'IF' construct, you need a 'PROCESS' structure to 'hold' a 'CASE' statement.

```
ENTITY voter IS
PORT
(
    A: IN bit_vector(2 DOWNTO 0);
    Q: OUT bit
);
END voter;

ARCHITECTURE second OF voter IS
BEGIN
    PROCESS (A)
    BEGIN
        CASE A IS
            WHEN "011" => Q <= '1';
            WHEN "101" => Q <= '1';
            WHEN "110" => Q <= '1';
            WHEN "111" => Q <= '1';
            WHEN OTHERS => Q <= '0';
            --Q shows if two or more of the inputs are HIGH
        END CASE ;
    END PROCESS ;
END second;
```

The 'CASE' statement has been placed within a 'PROCESS' structure and, as in chapter 2, the sensitivity list following the 'PROCESS' keyword has to contain all the variables that can affect the output.
The input variable has been changed from three separate signals (A, B and C) to a 3-bit variable, A. In detail, the three inputs are A(2), A(1) and A(0).

The syntax to declare such a variable is: A: IN bit_vector(2 DOWNTO 0);
Having done this, we can think of A having a value such as "110". Notice that this is a string of bits, and so you have to use the double-quote symbol ("). This contrasts with the single quotes (') around bit values like '1'. There's reason in there somewhere!

The 'CASE' statement itself needs the name of the variable, A, immediately after the 'CASE' keyword, and then the lines underneath set out the effects on a case by case list. The keyword 'OTHERS' means you don't have to list every single case - you list the 'interesting' ones, and then say what happens in all other situations. You can think of the => symbol as

"then" and <= as "gets". For instance, the first case is read "When A is "011" then Q gets '1'".

Entering the design using this technique has a number of benefits:
   the required behaviour is clearer;
   it saves you time,
   it may avoid you making a mistake.
The Quartus software takes care of any possible circuit simplification.

Exercises

Exercise 1
Use any of the voter code examples to turn the E-blocks FPGA board into a 3-input voting machine. Use switches 0, 1 and 2 as inputs, and LED D0 as output.
You should use the voter project as a starting point for this work. Remember to copy this project folder to a separate directory to prevent overwriting original material.

Exercise 2
Adapt your design to accept four inputs.
There should be two outputs: one to show if a majority of the inputs are HIGH , and another to show if there is a dead heat (just two inputs HIGH). Some solutions are given at the end of this chapter.
The pin information table of chapter 1 is reproduced here for convenience.

Switch	FPGA pin	FPGA pin	LED
SW7	45	55	D7
SW6	44	54	D6
SW5	41	52	D5
SW4	40	51	D4
SW3	39	50	D3
SW2	37	49	D2
SW1	36	48	D1
SW0	35	46	D0

Fig. 8.6

Decoders

Decoders take a coded input and generate a 'straightforward' output.
A typical example is the 74HC138 3-to-8 decoder. The input is a 3-bit binary code, while the output comprises 8 lines, one of which goes LOW .
For example, if the binary input is 011 then output number 3 goes LOW (and all the others are HIGH).

The IEC symbol for the 74HC138 is shown in the Proteus design example below.
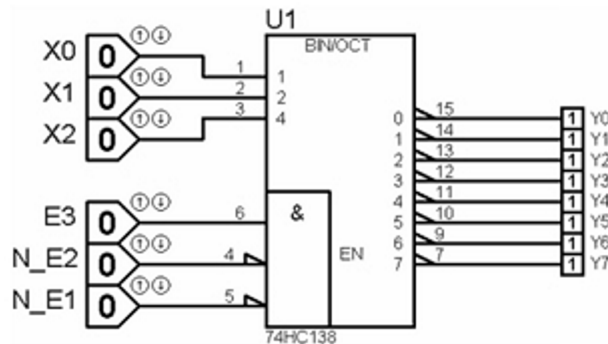
Fig. 8.7: 74HC138 3-to-8 decoder.

The 3-bit input signal is X; this comprises three individual inputs, X2 (MSB), X1 and X0. The 8-bit output is Y, comprising signals Y7…Y0.

The figure shows none of the outputs as LOW . This is because an enable signal is needed for this to happen. This enable signal has signal E3 HIGH while signals N_E2 and N_E1 are both LOW.

On your circuit simulator, run the animation (file 3_8_Dec.dsn) and click on E3 to make it HIGH . Now click on the X input signals and confirm that the Y output that goes LOW corresponds to the value of the binary input on X.

Verilog implementation of a 3-to-8 decoder

```
module decoder (E3, N_E2, N_E1, X, Y);
//Verilog version of 74138
    input [2:0] X;
    input E3, N_E2, N_E1;
    output [7:0] Y; reg [7:0] Y;

    always @ (X, E3, N_E2, N_E1)
    if (E3 & !N_E2 & !N_E1)
        case (X)
            0: Y = 8'b 11111110;
            1: Y = 8'b 11111101;
            2: Y = 8'b 11111011;
            3: Y = 8'b 11110111;
            4: Y = 8'b 11101111;
            5: Y = 8'b 11011111;
            6: Y = 8'b 10111111;
            7: Y = 8'b 01111111;
        endcase
    else
        Y = 8'b 11111111;
endmodule
```

The first thing to point out is the need for separate declaration lines for the 3-bit input, X,

and the single-bit input signals. If you try to put them all on one line, all the signals are regarded as 3-bit. In other words, the [2:0] qualifier belongs to the 'input' keyword, not the X variable.

Secondly, notice the nested case structure within the 'if'. This says, "If the chip is enabled, then the Y output depends on the X input." The else clause implies that if the chip is not enabled, then all the Y outputs go HIGH. The enable condition is met only when E3 is HIGH and both N_E2 and N_E1 are LOW.

VHDL implementation of 3-to-8 decoder

```
ENTITY decoder IS
PORT
(
    E3, N_E2, N_E1: IN bit;
    X: IN bit_vector(2 DOWNTO 0);
    Y: OUT bit_vector (7 DOWNTO 0)
);
END decoder;

ARCHITECTURE behaviour OF decoder IS
--VHDL version of 74138
BEGIN
    PROCESS (E3, N_E2, N_E1, X)
    BEGIN
        IF (E3 AND NOT N_E2 AND NOT N_E1)='1'
        THEN
            CASE X IS
                WHEN "000" => Y <= "11111110";
                WHEN "001" => Y <= "11111101";
                WHEN "010" => Y <= "11111011";
                WHEN "011" => Y <= "11110111";
                WHEN "100" => Y <= "11101111";
                WHEN "101" => Y <= "11011111";
                WHEN "110" => Y <= "10111111";
                WHEN "111" => Y <= "01111111";
            END CASE ;
        ELSE
            Y <= "11111111";
        END IF ;
    END PROCESS ;
END behaviour;
```

This design includes an example of a nested 'CASE' structure within the 'IF' . This says, "If the chip is enabled then the Y output depends on the X input." The 'ELSE' clause implies that if the chip is not enabled then all the Y outputs go HIGH . The enable condition is met only

when E3 is HIGH and both N_E2 and N_E1 are LOW .

Notice that, unlike Verilog and many programming languages, VHDL requires the ='1' part at the end of the 'IF' statement. Why? VHDL is very strongly 'typed'. This means that one type of data cannot be used in place of another. The 'IF' condition has to be a Boolean data type: TRUE or FALSE. Variables such as E3, and N_E2 can only take binary values: '1' and '0'. In most programming languages, these two types are interchangeable, but not in VHDL. You have to ask the question "Is E3 = '1'?" to which the answer, TRUE/FALSE, is of the correct data type.

Encoders

A decoder has a coded input and a 'straightforward' output; an encoder has a coded output and a 'straightforward' input. An example of an encoder is the CMOS CD4532, 8-input 'priority' encoder. The input arrives on one of its 8 input lines, and the output is a 3-bit binary number, indicating which of its input lines is active. The IEC symbol is shown in the Proteus design example below.
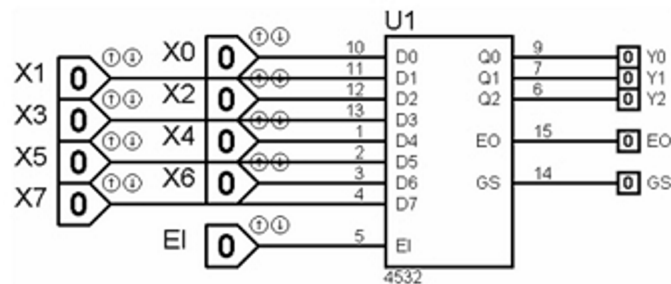
Fig. 8.8: 4532 8-to-3 encoder.

The 8 input lines are labelled X7…X0 and connect to the D7…D0 pins of the IC. The 3-bit output signal is labelled Y (Y2…Y0) and is generated by the Q2…Q0 pins of the IC.

There is an additional input that is used to enable the whole chip. It is labelled EI (Enable Input). If this is LOW then all the outputs are LOW , irrespective of the X inputs. This is the situation shown in Figure 8.8.

Thinking ahead to the Verilog and/or VHDL code to implement a 4532, the first part of the description will be "If EI is HIGH then (put the description of the rest of the behaviour in here) else all the outputs are LOW."

Run the animation on your circuit simulator and click EI so that it is HIGH. Now investigate how the X inputs control the Y outputs. Verify that if X'N' is HIGH then the value of the Y output is 'N'.

The statement given above is not quite the whole truth! What happens if two or more of the X inputs are HIGH? Verify that the input with the highest priority 'wins'. This feature explains the name given to the device - "priority encoder".

The other two outputs need some explanation. They are used when several 4532 chips are used together to encode larger numbers of input signals. The EO goes HIGH when there is no input (i.e. all the D inputs are LOW). The GS signal does the opposite. It goes HIGH when

there *is* an input (i.e. one or more of the D inputs are HIGH). Use the simulation to verify these statements.

Verilog implementation of 8-to-3 priority encoder

```verilog
module encoder (EI,X,Y,EO,GS);
//Verilog version of 4532 IC
    inout EI;
    input [7:0] X;
    output [2:0] Y;
    output EO, GS;
    reg [2:0] Y;
    reg EO, GS;
    assign EI = 1;

    always @ (EI,X)
    if (EI)
        if       (X[7]) begin EO = 0; GS = 1; Y = 7; end
        else if (X[6]) begin EO = 0; GS = 1; Y = 6; end
        else if (X[5]) begin EO = 0; GS = 1; Y = 5; end
        else if (X[4]) begin EO = 0; GS = 1; Y = 4; end
        else if (X[3]) begin EO = 0; GS = 1; Y = 3; end
        else if (X[2]) begin EO = 0; GS = 1; Y = 2; end
        else if (X[1]) begin EO = 0; GS = 1; Y = 1; end
        else if (X[0]) begin EO = 0; GS = 1; Y = 0; end
        else           begin EO = 1; GS = 0; Y = 0; end
    else               begin EO = 0; GS = 0; Y = 0; end
endmodule
```

The EI input has been declared in this implementation as an 'inout' type. This allows it to be set to a particular value within the code itself. This is needed since there are only 8 switches on the E-blocks switch board, and there are actually 9 inputs. In a 'real' situation EI would be declared as an input , and the line 'assign EI = 1;' would be omitted from the design.

As pointed out earlier, the use of nested if 's describes the behaviour of the encoder. The order in which the bits of the X input are tested is important to get the correct priority action. For instance, if X[7] is HIGH then the other bits are ignored - this is how the 'else if' structure works.

To test this design on the FPGA board, you need to assign all the IO pins from pin 35 to pin 45 to the X inputs, and connect the E-blocks switch board to the Port A connector. Pins 46,

48 and 49 can be used for the Y output, and pins 54 and 55 for GS and EO. EI can be assigned to any unused IO pin (e.g. 56).

The EO and GS signals are probably not important in a FPGA implementation. As mentioned above, they are needed to expand the circuit to cope with more than 8 inputs. However, if you wanted to encode a larger number of signals, you would simply redesign the code for wider bit vectors for X and Y, and lengthen the list of else if clauses. They have been included here to illustrate how the language can be used.

VHDL implementation of 8-to-3 priority encoder

```vhdl
ENTITY encoder IS
PORT
(
    EI: INOUT bit;
    X: IN bit_vector(7 DOWNTO 0);
    Y: OUT bit_vector(2 DOWNTO 0);
    EO, GS: OUT bit
);
END encoder;

ARCHITECTURE behaviour OF encoder IS
--VHDL version of 4532 IC
BEGIN EI <= '1';
    PROCESS (EI, X)
    BEGIN
        IF (EI)='1'
        THEN
            IF      X(7) = '1' THEN EO <= '0'; GS <= '1'; Y <=
"111";
            ELSIF X(6) = '1' THEN EO <= '0'; GS <= '1'; Y <= "110";
            ELSIF X(5) = '1' THEN EO <= '0'; GS <= '1'; Y <= "101";
            ELSIF X(4) = '1' THEN EO <= '0'; GS <= '1'; Y <= "100";
            ELSIF X(3) = '1' THEN EO <= '0'; GS <= '1'; Y <= "011";
            ELSIF X(2) = '1' THEN EO <= '0'; GS <= '1'; Y <= "010";
            ELSIF X(1) = '1' THEN EO <= '0'; GS <= '1'; Y <= "001";
            ELSIF X(0) = '1' THEN EO <= '0'; GS <= '1'; Y <= "000";
            ELSE                          EO <= '1'; GS <= '0'; Y <=
"000";
            END IF ;
        ELSE
            EO <= '0'; GS <= '0'; Y <= "000";
        END IF ;
    END PROCESS ;
END behaviour;
```

The EI input has been declared in this implementation as an 'INOUT' type. This allows it to be set to a particular value within the code itself. This is needed since there are only 8 switches on the Matrix E-blocks switch board, and there are actually 9 inputs. In a 'real' situation EI would be declared as an 'IN', and the line EI <= '1'; would be omitted from the

design.

As discussed earlier, the use of nested IF's describes the behaviour of the encoder. The order in which the bits of the X input are tested is important to get the correct priority action. For instance, if X(7) is HIGH then the other bits are ignored - this is how the 'ELSIF' structure works. Note the spelling of the keyword 'ELSIF' .

To test the design on the FPGA board, assign all the IO pins from pin 35 to pin 45 to the X inputs, and connect the E-blocks switch board to the Port A connector. Pins 46, 48 and 49 can be used for the Y output, and pins 54 and 55 for GS and EO. EI can be assigned to any unused IO pin (e.g. 56).

The EO and GS signals are probably not important in a FPGA implementation. As mentioned earlier, they are needed to expand the circuit to cope with more than 8 inputs. However, if you wanted to encode a larger number of signals, you would simply redesign the code for wider bit vectors for X and Y, and lengthen the list of else if clauses. They have been included to show how the language can be used.

Comparators

In digital electronics, a comparator is used to check one binary number against another. Outputs tell whether the two numbers are the same, or indicate which one is bigger.
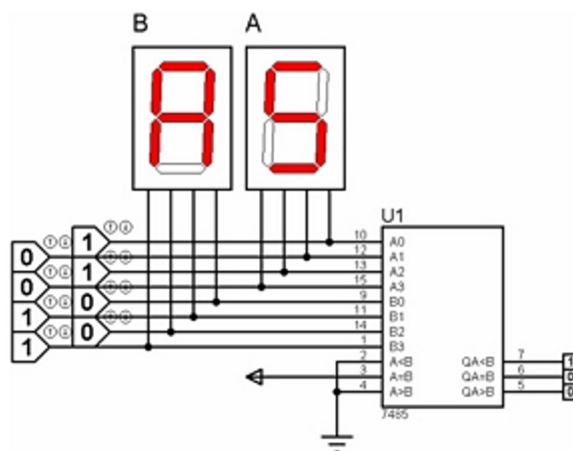
Fig. 8.9: Comparator showing that 0101 is less than 1010.

In the circuit shown above, the TTL 7485 4-bit comparator has two inputs, A (comprising A3 down to A0) and B (B3 to B0). These are set respectively to 0101 and 1010. The seven-segment displays show the equivalent hex values (5 and A). The outputs confirm that input A is less than input B.

Enter this design into your circuit simulation package and check that the outputs indicate whether A < B, A = B or A > B.

Verilog implementation of a comparator

```
module comparator (A,B,A_EQ_B);
    input [3:0]A,B;
    output A_EQ_B;
    reg A_EQ_B;
```

```
        always @ (A,B)
        if (A == B)
            A_EQ_B = 1;
        else
            A_EQ_B = 0;
    endmodule
```

This implementation uses the 'if...else...' syntax to determine the state of the output signal. As in previous examples, this syntax has to be embedded within an 'always' structure, which requires the output to be declared as a register.
A neater way of achieving exactly the same result is shown below.

```
    module comparator (A,B,A_EQ_B);
        input [3:0]A,B;
        output A_EQ_B;
        assign A_EQ_B = (A == B);
    endmodule
```

VHDL implementation of a comparator

```
    ENTITY comparator IS
    PORT
    (
        A,B: IN BIT_VECTOR (3 DOWNTO 0);
        A_EQ_B: OUT BIT
    );
    END comparator;

    ARCHITECTURE a OF comparator IS
    BEGIN
        A_EQ_B <= '1' WHEN A = B ELSE '0';
    END a;
```

Both of these examples compare two 4-bit input signals, A and B. Signals with more bits could easily be implemented simply by re-defining the width of the inputs.
If you want an output that shows when A > B, you would simply replace the A == B (Verilog) or A = B (VHDL) test with A > B .

Parity checker

The parity of a binary number can be even or odd: it's even if there are an even number of 1's in the number (and odd otherwise). Thus the number 10100101 has even parity, but 00000010 has odd. The truth table for a four-bit parity checker is shown below:

	A	B	C	D	Q
0	0	0	0	0	0
1	0	0	0	1	1

2	0	0	1	0	1
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	0
10	1	0	1	0	0
11	1	0	1	1	1
12	1	1	0	0	0
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	0

Fig. 8.10 Truth table for 4-bit odd parity checker

The 1's in the Q column indicate that the corresponding number, formed from A, B, C and D, has odd parity.

Notice that the 5-bit number A,B,C,D,Q always has even parity. As a result, Q can be thought of as an *odd parity checker* or as an *even parity generator* .

The Karnaugh map for Q is:

Fig. 8.11: Karnaugh map for odd parity checker.

No grouping of 1's is possible. However, this checkerboard pattern is a characteristic of the exclusive-OR gate. The logic of exclusive-OR is 'one or the other but not both'.

The truth table for an exclusive-OR reflects this meaning:

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 8.12: Truth table for exclusive-OR

… and the Karnaugh map shows the checkerboard:
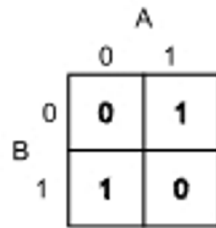
Fig. 8.13: Karnaugh map of exclusive-OR

The circuit symbol for exclusive-OR is:
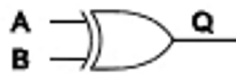
Fig. 8.14: Exclusive-OR symbol

…and the circuit for the 4-bit odd parity checker/even parity generator is:
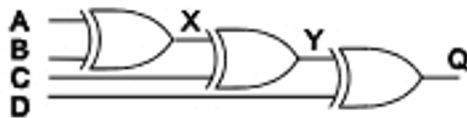
Fig. 8.15: Simplified circuit.

It's not obvious that this circuit will work.

Convince yourself that it does by completing the table below and checking that it agrees with the one above.

	A	B	X	C	Y	D	Q
0	0	0		0		0	
1	0	0		0		1	
2	0	0		1		0	
3	0	0		1		1	
4	0	1		0		0	
5	0	1	1	0	1	1	0
6	0	1		1		0	
7	0	1		1		1	
8	1	0		0		0	
9	1	0		0		1	
10	1	0		1		0	
11	1	0		1		1	
12	1	1		0		0	
13	1	1	0	0	0	1	1
14	1	1		1		0	
15	1	1		1		1	

Fig. 8.16: Truth table for the Simplified circuit.

Exercise

Devise a Verilog and/or VHDL implementation of Figure 3.15

Gray-to-binary converter

This circuit is another example of the use of an exclusive-OR gate.
Gray code is used to reduce errors in mechanical encoding systems. In Gray code, when moving from one value to the next, only one digit changes,. This is very different from binary where many digits may change at the same time. For instance, changing from 7 to 8 in binary means changing from 0111 to 1000 - all the digits change. In Gray code 7 is 0100 and 8 is 1100 - only the left-most digit changes.
For a mechanical system, this is useful in that an error in one of the digits will only produce a small error in the coded value. Suppose there is an error in the left-most digit, caused by a slight misalignment of a sensor. This causes 0100 to be mistaken for 1100. In a binary-coded system the output would read 4 when it should be 12, but using Gray code it would show 7 instead of 8: a much less serious problem.

So how does Gray code work? The table below compares 4-digit Gray with 4-bit binary.

	Gray				binary			
	G3	G2	G1	G0	B3	B2	B1	B0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	1	0	0	1	0
3	0	0	1	0	0	0	1	1
4	0	1	1	0	0	1	0	0
5	0	1	1	1	0	1	0	1
6	0	1	0	1	0	1	1	0
7	0	1	0	0	0	1	1	1
8	1	1	0	0	1	0	0	0
9	1	1	0	1	1	0	0	1
10	1	1	1	1	1	0	1	0
11	1	1	1	0	1	0	1	1
12	1	0	1	0	1	1	0	0
13	1	0	1	1	1	1	0	1
14	1	0	0	1	1	1	1	0
15	1	0	0	0	1	1	1	1

Fig. 8.17: Gray code and Binary truth tables

Counting in Gray code and in binary

Counting in Gray code:

To get from one value to the next, change the right-most digit that will give you a new pattern.

Thus, starting with 0000, you get the next value by changing digit G0, the right-most digit. This gives 0001. Now you have to change digit G1, to get 0011. Now you can change digit G0 again: 0010.

The table shows that only one digit changes between adjacent values.

Suppose that you have a mechanical position sensor that generates Gray code, but you need to change this into binary. You need an electronic system that has four inputs, G3, G2, G1 and G0 and generates four outputs, B3, B2, B1 and B0 according to the table of Figure 8.17.

Generating output B3 is easy: simply join it to input G3 with a piece of wire.

Output B2:

For the first 8 rows of the table, it is the same as input G2.

For the last 8 rows, B2 is the complement of G2.

We need a device that will invert input G2 *when G3 is HIGH* . The exclusive-OR gate is just such a device.

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 8.18: Exclusive-OR truth table.

Fig. 8.18 gives the truth table for the exclusive-OR. Output Q is the same as input B when A is LOW , but the complement of B when A is HIGH .

Output B1 is sometimes the same as input G1 and sometimes its complement. G1 needs to be inverted when output B2 is HIGH .

Similarly, using signal B1 to invert input G0 will create output B0.

The final circuit is shown below.
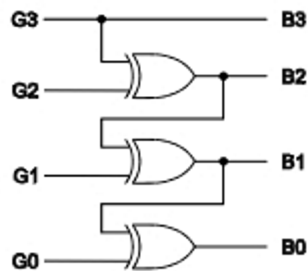
Fig. 8.19: Gray to binary converter.

Verilog implementation of Gray-to-binary conversion

The code is a straightforward implementation of the circuit.

```
module gray2bin (G,B);
    input [3:0] G;
    output [3:0] B;
    assign B[3] = G[3];
    assign B[2]= G[2] ^ B[3];
    assign B[1] = G[1] ^ B[2];
    assign B[0] = G[0] ^ B[1];
endmodule
```

The last three assignments are an example of the kind of thing that can be implemented in Verilog using a 'for' loop. Each line is identical, except that the bit number changes each time.

The code below shows how an 8-bit Gray to binary converter can be implemented using a loop structure.

```
module gray2bin_for (G,B);
    input [7:0] G;
    output [7:0] B;
    reg [7:0] B; integer N;

    always @ (G)
    begin
    B[7] = G[7];
    for (N=6; N>=0; N=N-1)
        B[N] = G[N] ^ B[N+1];
    end
endmodule
```

The loop has to placed with an 'always' structure, so the output has to be declared as a 'reg' , as we have seen before. The loop control variable, N , has to be declared as an integer .

The loop itself starts with the keyword 'for' followed by three statements defining how the loop behaves. The first statement gives the starting value for the loop variable; the second states the conditions under which the loop should continue; and the third states how the loop variable changes each time the loop 'executes'.

As a paper exercise, write out the six lines of code that the loop generates. Check that the last three lines are identical to the code at the top of this page.

VHDL implementation of Gray to binary conversion

Again, the code is a straightforward implementation of the circuit. Signal B has to be declared as an 'INOUT' type however, since it is used both as an output and as an input - it appears on both the left and right side of the 'equations'.

```
ENTITY gray2bin IS
PORT
(
    G: IN BIT_VECTOR (3 DOWNTO 0);
    B: INOUT BIT_VECTOR (3 DOWNTO 0)
);
END gray2bin;

ARCHITECTURE a OF gray2bin IS
BEGIN
    B(3) <= G(3);
    B(2) <= G(2) XOR B(3);
    B(1) <= G(1) XOR B(2);
    B(0) <= G(0) XOR B(1);
END a;
```

The last three assignments can be implemented in VHDL using a 'FOR' loop. Each line is identical, except that the bit number changes each time.

The code below shows how an 8-bit Gray to binary converter can be implemented using a loop structure.

```
ENTITY gray2bin_for IS
PORT
(
    G: IN BIT_VECTOR (7 DOWNTO 0);
    B: INOUT BIT_VECTOR (7 DOWNTO 0)
);
END gray2bin_for;

ARCHITECTURE a OF gray2bin_for IS
BEGIN
    PROCESS (G,B)
    BEGIN
        B(7)  <= G(7);
        FOR N IN 6 DOWNTO 0 LOOP
            B(N)  <= G(N) XOR B(N+1);
        END LOOP ;
    END PROCESS ;
END a;
```

The loop has to placed with a 'PROCESS' structure. The sensitivity list normally contains just the input signals, since these are the ones that affect the outputs. However, in this example, signal B is needed as well since it appears on the right-hand side of the equations, just like an input signal. The loop control variable, N , does not need to be declared. The loop itself starts with the keyword, 'FOR' , followed by the name of the loop variable and the range that the variable is to take as the loop 'executes'.

As a paper exercise, write out the six lines of code that the loop generates. Check that the last three lines are identical to the code at the top of this page.

Adders subtractors and multipliers

Complex arithmetic on data is best done using microprocessors or digital signal processor chips. However, there are dedicated chips (such as the TTL 74283) that will add numbers together.

Details about how positive and negative numbers are represented in binary form, and about how these somewhat complex devices work are given in Rice (2001).

These notes emphasize the use of Verilog and VHDL to implement designs in a FPGA chip, so here we look at the way you can implement adders and subtractors with great ease using these languages. Even a multiplier is possible!

Verilog adder

The code below shows how to add two 4-bit numbers, A and B, to form a 5-bit answer, Q.

```verilog
module adder (A,B,Q);
    input [3:0] A,B;
    output [4:0] Q;
    assign Q = A + B;
endmodule
```

Easy! The work done by the Quartus software to make it so easy is impressive.
It is worth compiling, downloading and testing this design on the FPGA board
(switch board attached to Port A and LED board to Port B). Use pins 35 - 39 for A
(0) to A(3), 40 - 45 for B(0) to B(3) and 46 - 51 for Q(0) to Q(4).

Verilog subtractor

This is identical to the adder, except that the '−' sign is used instead of '+'.
Some familiarity with 2's complement representation of negative numbers is needed to appreciate the results of this design.
For instance, if A is 0 (none of the switches SW3 to SW0 pressed) and B is 1 (SW4 pressed) then the output should be '−1'. In 5-bit 2's complement this is 11111.
If you press all of the switches SW7 to SW4 the calculation is 0 − 15. This should come to −15, of course, i.e. 10001 in 5-bit 2's complement arithmetic.

Verilog multiplier

The code for this is shown below. Assuming inputs are positive 4-bit numbers, the output needs to be 8-bit. The maximum calculation is 15 X 15 = 225. In binary this is 1111 X 1111 = 11100001.

```verilog
module mult (A,B,Q);
    input [3:0] A,B;
    output [7:0] Q;
    assign Q = A * B;
endmodule
```

The simplicity of the code belies the complexity of this design. It takes a significant fraction of the total resources of the FPGA chip to execute even this small (4-bit x 4-bit) calculation.

VHDL adder

The code shows how to add two 4-bit numbers, A and B , to form a 5-bit answer, Q.

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.ALL;

ENTITY arithmetic IS
PORT
(
    A,B: IN UNSIGNED(3 DOWNTO 0);
    Q: OUT UNSIGNED(4 DOWNTO 0)
);
END arithmetic;

ARCHITECTURE adder OF arithmetic IS
BEGIN
    Q <= ('0' & A) + ('0' & B);
END adder;
```

The first new feature of this design is the need to use the IEEE VHDL library for its numerical processing features. The basic language does not have these, so you have to 'bolt these on' using libraries. The first line of the code declares the library and the second the particular facilities within the library that are to be used.

The second feature is that the input and output signals have been declared as UNSIGNED , rather than as BIT_VECTOR . This is required if arithmetic operations are to be used.

The third feature is the need to generate 5-bit numbers from the 4-bit numbers A and B. With its strict type rules, VHDL will only add numbers where the output has the same width as the input. The ('0' & A) object is a 5-bit number with 0 as its MSB and the four bits of A as its remaining digits.

Finally, the '+' sign has been used to do the addition.

It is worth compiling, downloading and testing this design on the FPGA board (switch board attached to Port A and LED board to Port B). Use pins 35 - 39 for A (0) to A(3), 40 - 45 for B(0) to B(3) and 46 - 51 for Q(0) to Q(4).

VHDL subtractor

This is identical to the adder, except that the '–' sign is used instead of '+'.

Once again, some familiarity with 2's complement representation of negative numbers is needed to appreciate the results of this design.

For instance, if A is 0 (none of switches SW3 to SW0 pressed) and B is 1 (SW4 pressed,) then the output should be –1. In 5-bit 2's complement this is 11111.

If you press all of the switches SW7 to SW4, the calculation is 0 – 15. This should come to –15, which is 10001 in 5-bit 2's complement arithmetic.

VHDL multiplier

The code for this is shown below. Assuming that the inputs are positive 4-bit numbers, the output needs to be 8-bit. The maximum calculation is 15 X 15 = 225. In binary this is 1111 X 1111 = 11100001.

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.ALL;

ENTITY arithmetic IS
PORT
(
    A,B: IN UNSIGNED(3 DOWNTO 0);
    Q: OUT UNSIGNED(7 DOWNTO 0)
);
END arithmetic;

ARCHITECTURE mult OF arithmetic IS
BEGIN
    Q <= A * B;
END mult;
```

The simplicity of the code belies the complexity of this design. It takes a significant fraction of the total resources of the FPGA chip to execute even this small (4-bit x 4-bit) calculation.

Summary

This chapter has covered the use of Verilog and VHDL to implement a number of combinational logic design ideas – solutions to problems that crop up frequently in digital systems. Some new Verilog and VHDL constructs have been introduced that allow the language to describe the designs. The Quartus software has been used to compile the designs into code that can be downloaded into the FPGA chip on the Matrix E-blocks board.
The assignments that follow gives you a chance to develop some of these ideas.
In the chapter that follows these, we concentrate on sequential logic - counters and other systems - where the output depends on the past sequence of input signals.

Chapter 9: Combinational logic assignment

Introduction

The basic E-blocks2 FPGA kit comprises the FPGA board itself, a switch board with 8 switches on it, an LED board with 8 LEDs on it, and a seven-segment display board.
The idea of this assignment is that the switches will control the information on the seven-segment display, using the FPGA as the interface.
The display has four seven-segment characters. The segments of each character are labelled 'a' to 'g' , 'a' being the top one, 'b' to 'h' the other outside ones going clockwise, and 'g' the middle one. There is also a decimal point segment (dp) on each character.
Each display is a 'common anode' device, which means that it needs a HIGH on its anode pin, and LOWs on its cathodes in order to illuminate a segment. Although the anodes are separate, the cathodes of all four displays are joined together. This means that all four will display the same pattern if their anodes are HIGH simultaneously.
The relevant pin numbers are:

Anode for char	Port A pin	FPGA pin	segment	Port B pin	FPGA pin
0	0	110	a	0	73
1	1	111	b	1	74
2	2	112	c	2	75
3	3	113	d	3	76
			e	4	77
			f	5	80
			g	6	83
			dp	7	84

Fig. 9.1: Tables of pin connections.

The table assumes you have plugged the display board into connectors Port A and Port B of the FPGA board.
Thus, to use the right-hand display to show the number 7 you would need to make FPGA pin 113 go HIGH and pins 73, 74 and 75 go LOW. This would make segments a, b and c of the right-hand display light up resulting in a '7' being shown there.

Task 1

Use Verilog or VHDL to describe a system where switch SW7 controls whether the left-hand character of the display board lights up, SW6 the next one, SW5 the next, and SW4 the right-hand one.
The binary number present on switches SW3 … SW0 determines the actual (hexadecimal) character displayed.
Compile and download your design to the FPGA board.
Demonstrate your design to the lecturer.
Advice:

- Split the switch input data into two 4-bit signals: the bits from SW7 … SW4 could make up a signal named 'position', and the bits from SW3…SW0 a signal named 'value'.
- Declare two output signals: one 4-bit for driving the anodes of the displays (call it 'anodes') and an 8-bit one for the cathodes (call it 'segments').
- The first part of this task is now quite straightforward: in Verilog you could write "assign anodes = position"; in VHDL "anodes <= position". You need to make sure the right pins are assigned to the individual bits of these two signals, of course.
- To sort out which segments light up depending on the value of value use the 'case' structure.
- Remember that an individual bit needs to be LOW in order to turn that segment on. Allocate pins to the various bits so that segment[7] controls segment a and segment[0] controls the decimal point. This makes it easier to think about the bit pattern needed to create any particular character on the display.
- For instance, to show the number 7, the bit pattern would be 00011111, in order to turn on the first three segments, namely segments 'a', 'b' and 'c'.
- This problem is similar to Exercise 2.

Task 2

Redesign the code so that the character displayed reflects the gray code value on switches SW3…SW0, rather than the binary value.

Referring to Figure 8.17, this means that the display should show '2' for an input of '0011', and 'E' for '1001', for example.

Compile, download and demonstrate your design to the lecturer.

Task 3

Look up the manufacturer's data sheet for the 74LS47 - typing "74LS47" into Google is a good start! This chip does a similar job to Task 1, but has a few more inputs.

The RBI (Ripple-blanking input) can be used to turn all the segments off when the input data is 0000. LT (Lamp test) will force all the segments to light up, irrespective of the data input.

The 7447 is a standard part in the Quartus library. Create a new Block Design File. Add a 7447 from the MAXPLUS2 section of the Symbol library. Add inputs and outputs to complete a circuit. Use FILE…CREATE/UPDATE to create a Verilog/VHDL equivalent of this part.
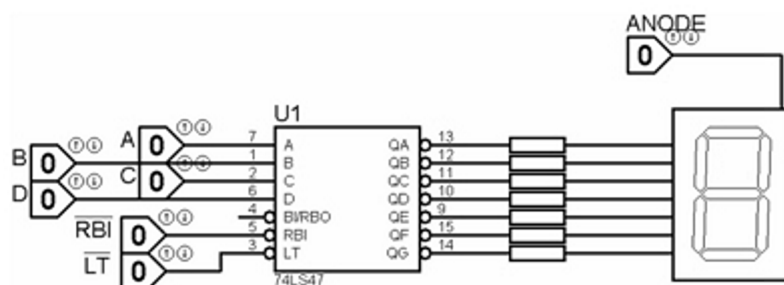
Fig. 9.2: A 74LS47 circuit

Task 4

Like Task 2, the aim here is to show the Gray code of the input data on the display. However, because of the restrictions of the 74LS47, you only need to deal with a 3-bit input, so leave input D permanently LOW. An input on CBA of '011' should show '2' on the seven-segment display, for example, while '111' should show '5'.
Demonstrate your design to the lecturer.
Advice: use a three-bit version of Figure 8.17 as part of your design

Task 5

The assignment is to write a short report on one of your Verilog or VHDL designs. It should include program listings and printouts of the design. Explain how you tested it and include your test results. Compare the FPGA and MSI approaches in terms of chip count, cost, flexibility, etc.

Coverage - with respect to British BTEC syllabus

This assignment covers half of Outcome 3 of the Digital and Analogue Devices and Circuits unit (DADC), and the whole of Outcomes 1 and 3 of the Combinational and Sequential Logic unit (CSL).

Assessment and grading criteria

DADC (Pass): "Design and construct combinational. digital electronic circuits using logic devices" and "Test digital electronic circuits" - complete Task 1 satisfactorily.

CSL (Pass): "Design and build circuits using combinational logic" and "Design and evaluate a digital system" - Tasks 1, 3 and 5 completed satisfactorily.

Merit: All tasks completed satisfactorily. Report is written clearly, using technical and non-technical language appropriately. Report is a stand-alone document, giving background to assignment as well as outlining the process undertaken. Evidence of problem solving using appropriate methods is presented.

Distinction: All tasks completed satisfactorily. Evidence of ability to work independently, but also to ask for advice and discuss best approach when appropriate

Chapter 10: Sequential logic

Introduction

The previous sections examined a number of combinational logic circuits – circuits whose outputs depend purely on the current combination of HIGH and LOW logic levels on their inputs.

In this chapter, an extra ingredient is added - memory. The current output of a sequential logic circuit depends not only on the current inputs, but also on what has happened to the circuit in the past. Counters and other systems that move from one state to another under the control of a clock are examples of such circuits.

The basic memory element that allows sequential logic circuits to be built is the 'Set/Reset bi-stable latch'. A slightly more complex device is the 'flip-flop'. This comes in two varieties: the D-type and the JK.

Topics covered in this chapter.

After briefly considering the behaviour of the SR latch and the two flip-flop types, the circuits considered in this chapter are:

- asynchronous up counter
- asynchronous BCD up counter
- synchronous up counter
- synchronous up/down 0 – 5 counter
- synchronous up/down BCD counter
- sequence detectors
- flashing car turn indicators

There is also an assignment, designed to cover certain criteria of the UK Edexcel BTEC Higher National units in Digital and Analogue Devices and Circuits, and Combinational and Sequential Logic.

The following new Verilog and VHDL constructs are covered:

- posedge, negedge, falling_edge, rising_edge
- wire, signal
- parameter, variable
- integer data type (VHDL)
- user-defined data type (VHDL)
- concatenation

The SR latch

A schematic of a ⎯ RESET ⎯ Set-Reset latch is shown below:
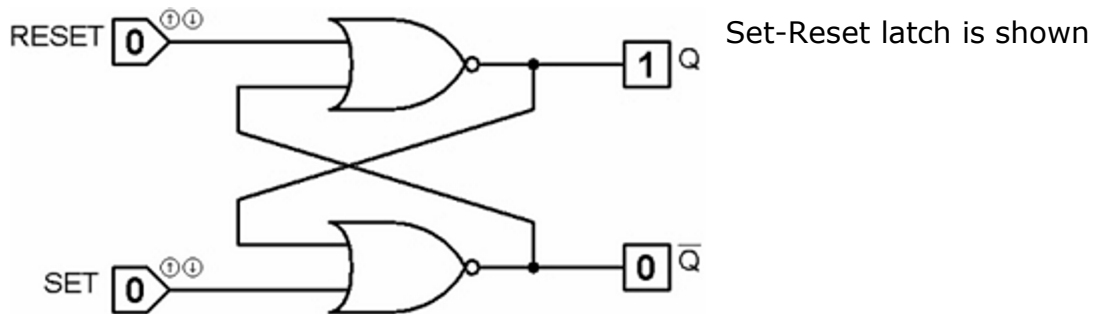
Fig. 10.1: Set-Reset latch made from NOR gates (in its *set* state)

To appreciate what this circuit does, you need to simulate it in your circuit simulation package. When you first run the simulation nothing much seems to happen. The outputs at start up are undefined. The circuit cannot 'make up its mind' as to what the outputs should do.

Your simulation package should allow you to set the initial Logic state of the device so that the Q output goes HIGH. Now click the Set input again, so that it is LOW. What do the outputs do? You should find that they remain unchanged, with the Q still 1 and !Q still 0. The circuit is now *remembering* that it has just been *set*. Next, click the Reset input a couple of times (so that it goes HIGH then LOW). You will have *reset* the circuit.

So that's what you can do to this circuit. You can *set* it by toggling the Set input up and down, or you can *reset* it, by toggling the Reset input up and down. Normally you leave both Set and Reset inputs LOW: it then *remembers* whether it is set or reset.

Note that the term 'set' is defined as 'making the Qoutput go HIGH (and!Q go LOW)'.
The term 'reset' is defined as 'making the Q output go LOW (and !Q go HIGH).'
What happens if you try to *set* this circuit and *reset* it at the same time? You should find both outputs go LOW . This doesn't do any damage to the circuit, but is not normally useful.

The D-type flip-flop
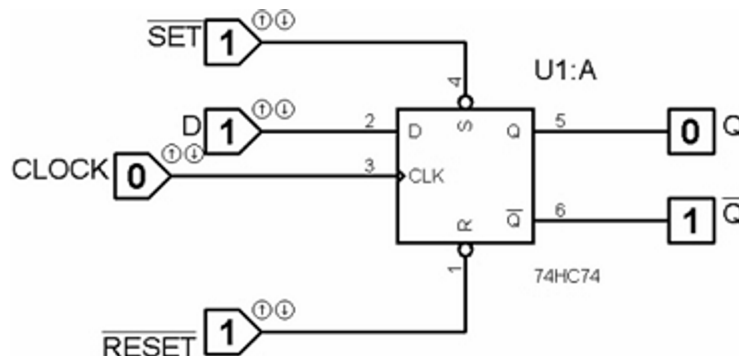
Fig. 10.2: The D-type flip-flop

With four inputs, the D-type flip-flop is slightly more complex than the Set-Reset latch.
Two of the inputs are labelled SET and RESET, so you might expect them to behave in the

same way as the SET and RESET inputs of the previous circuit. Well, they do, but they are active when they are *LOW* . This is why their names have bars over.

Simulate this simple circuit in your simulation package. Verify that toggling the !SET input LOW then HIGH again will *set* the Q output to 1, and that doing the same to the !RESET will *reset* Q back to 0.

Leave the !SET and !RESET inputs *in* active (HIGH) and click the CLOCK input to make it go HIGH . Verify that whatever data is on the D input is transferred to the Q output. Try this with the D input set to both possible states (1 or 0).

The D-type flip-flop is used to remember (store) a single binary digit - a '1' or a '0'. Banks of them can store numbers comprising as many bits as there are flip-flops. Such a bank of D-types is called a register, and you will see how they are used later in this chapter.

The JK flip-flop
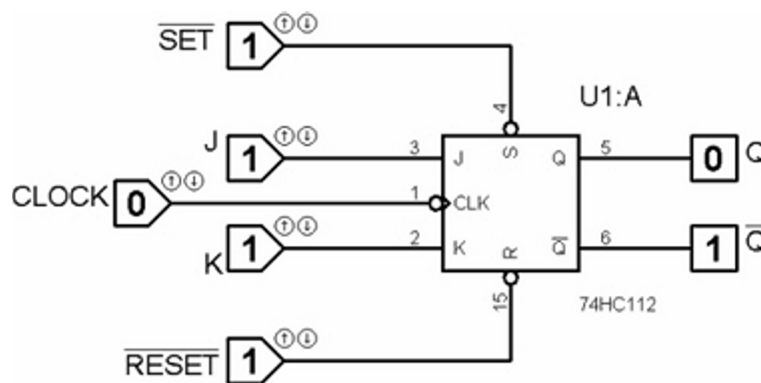
Fig. 10.3: The JK flip-flop

With five inputs, the JK is slightly more complex than the D-type flip-flop or the Set-Reset latch.

The !SET and !RESET inputs behave in the same way as they do for the D-type. They are active-low control signals.

Develop this circuit in your simulation package. Verify that toggling the !SET input LOW then HIGH again will *set* the Q output to 1, and that doing the same to the !RESET will *reset* Q back to 0.

Leave the !SET and !RESET inputs active (HIGH) and make sure the J and K inputs are also both HIGH.

Now try toggling the CLOCK input HIGH and LOW a few times. What do the outputs do? Verify that they also *toggle* each time the CLOCK changes from HIGH to LOW .

There are other things you can do with a JK flip-flop. If you make the J and K inputs the complement of each other (one HIGH and the other LOW,) then the device behaves just like a D-type, with data on the J input transferred to the Q output on the (falling) edge of the CLOCK . If you leave the J and K inputs both LOW, then the outputs remain unchanged from their present state, whatever you do to the CLOCK.

Use the Proteus software to verify these statements.

We will only be considering the toggle mode of the JK flip-flop in these notes, so remember that the outputs toggle on the falling edge of the CLK , provided both J and K are HIGH (and the !SET and !RESET inputs are both inactive - HIGH .)

Asynchronous up-counter

The first application of a JK flip-flop we will look at is a binary counter. Before thinking about the circuit, consider the waveforms required at the outputs of a binary counter.
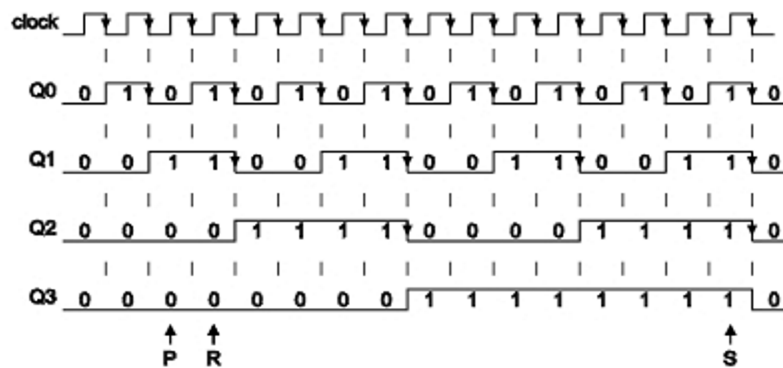
Fig. 10.4: Waveforms for a binary counter

The diagram shows the outputs Q3, Q2, Q1 and Q0 of a 4-bit counter. The count increases by one each time the clock input changes from HIGH to LOW. Output Q3 is taken as the most-significant-bit (MSB). Starting at a count of '0' at the left hand side of the diagram, it reaches '2' at point P (0010), '3' at R (0011) and '15' at S (1111). On the negative edge of the next clock pulse, the counter 'rolls over' back to 0.

When does output Q0 change state?

     Answer: on each negative edge of the clock. The negative edges of the clock signal have been emphasized by the little arrows on the waveform.

When does Q1 change state?

     Answer: on each negative edge of Q0.

When does Q2 change state?

     Answer: on each negative edge of Q1.

When does Q3 change state?

     Answer: on each negative edge of Q2.

Circuit implementation

Having analyzed the waveforms in this way, it is fairly easy to devise a circuit that will generate them. The key point is to use a JK flip-flop in 'toggle' mode for each output, but taking care over where each one gets its clock input. The Q0 flip-flop can get its clock from the system clock input. For the Q1 flip-flop clock input, use the Q0 signal; the Q2 flip-flop should use Q1, and the Q3 flip-flop should use Q2.
Figure 10.5 shows this set-up.
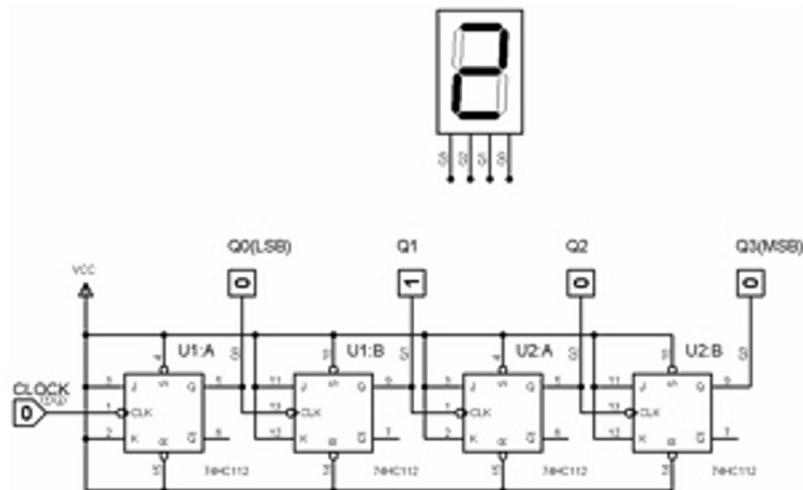
Fig. 10.5: 4-bit binary counter built from JK flip-flops.

You can see that all the flip-flops have their J and K inputs tied HIGH (to the VCC rail.) so they will toggle each time they get a negative edge on their CLK input. You can also see where each flip-flop gets its CLK signal from. For U1:A, it is the CLOCK signal itself. For U1:B, it is Q0. For U2:A, it is Q1 and for U2:B it is Q2, as discussed earlier.

The S and R connections are also all tied HIGH so that they are inactive, and do not affect the circuit operation.

The circuit shows the situation after two HIGH-LOW edges of the system CLOCK . This corresponds to point P on the waveforms diagram of Figure 10.4. Note that Q3 is the MSB, but it is convenient to draw the circuit with the MSB on the right.

Enter this design in your circuit simulation package. Verify that the count increments by one each time the system CLOCK changes from HIGH to LOW .

Verilog implementation of 4-bit asynchronous counter

The code below implements the circuit of figure 10.5

```
module asynch_counter (Clk, Q);
    input Clk;
    output [3:0] Q;
    reg [3:0] Q;

    always @ ( negedge Clk)
        Q[0] = !Q[0];
    always @ ( negedge Q[0])
        Q[1] = !Q[1];
    always @ ( negedge Q[1])
        Q[2] = !Q[2];
    always @ ( negedge Q[2])
        Q[3] = !Q[3];
endmodule
```

The four 'always' statements represent the four flip-flops. The event control list for each one uses the keyword 'negedge' to signify that the negative edge of the signal controls the event that follows.

Statements such as Q[0] = !Q[0]; are the way you make sure a signal toggles. Thus the first 'always' statement causes Q[0] to toggle on every negative edge of Clk. The second one causes Q[1] to toggle on every negative edge of Q[0] , and so on.

Try this out. Use pin 35 for the Clk and pins 46, 48, 49 and 50 for Q[0] , Q[1] , Q[2] and Q[3] and connect the switch board to connector Port A and the LED board to Port B of the E-blocks FPGA board. You should find that the system counts up in binary each time you release switch SW0.

In fact, if you watch the LEDs carefully, you may notice a problem - sometimes they count two, or three increments at a time. The problem lies not with the counter, but with the switch. Mechanical switches bounce, giving several pulses in quick succession, when they should just give one. A solution to this problem is given in the next section.

Debounce circuit

One solution to the switch-bounce problem is to use the SR latch circuit of Figure 10.1. We will need to use two switches from the switch board, one to set, and the other to reset, the latch. The switches will still bounce, of course, but the spurious pulses will only reinforce the first, 'proper' signal. The circuit of Figure 10.6(*bis*) shows the details, including the names of the signals involved in the new code.
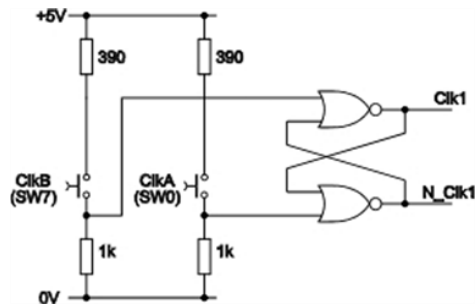
Fig. 10.6(*bis*): Using an SR latch to de-bounce switches.

```vhdl
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL, IEEE.Numeric_Std.ALL;
ENTITY asynch_counter IS
PORT
(
    ClkA,ClkB: IN STD_LOGIC;
    Q: INOUT UNSIGNED(3 DOWNTO 0)
);
END asynch_counter;
ARCHITECTURE second OF asynch_counter IS
SIGNAL Clk1, N_Clk1: STD_LOGIC;
BEGIN
    --de-bounce the switches
    Clk1 <= NOT (ClkB OR N_Clk1);
    N_Clk1 <= NOT (ClkA OR Clk1);
    --use Clk1 as clock for first flip-flop
    PROCESS (Clk1, Q)
    BEGIN
        IF RISING_EDGE (Clk1)
        THEN
            Q(0) <= NOT (Q(0));
        END IF ;
        IF FALLING_EDGE (Q(0))
        THEN
            Q(1) <= NOT (Q(1));
        END IF ;
        IF FALLING_EDGE (Q(1))
        THEN
            Q(2) <= NOT (Q(2));
        END IF ;
        IF FALLING_EDGE (Q(2))
        THEN
            Q(3) <= NOT (Q(3));
        END IF ;
    END PROCESS ;
END second;
```

The code uses the 'SIGNAL' keyword to declare the two internal signals Clk1 and N_Clk1, which are 'wired up' to the ClkA and ClkB inputs. Clk1 is then used to clock the Q0 flip-flop. Assign pin 45 to ClkB , compile and download to the FPGA board. Clicking and releasing SW0

and SW7 alternately will generate nice clean clock pulses for Q0, and the system should now count reliably.

VHDL implementation of 4-bit asynchronous counter

The code below implements the circuit of Figure 10.5.

```vhdl
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL, IEEE.Numeric_Std.ALL;
ENTITY asynch_counter IS
PORT
(
    Clk: IN STD_LOGIC;
    Q: INOUT UNSIGNED(3 DOWNTO 0)
);
END asynch_counter;

ARCHITECTURE first OF asynch_counter IS
BEGIN
    PROCESS (Clk, Q)
    BEGIN
        IF FALLING_EDGE (Clk)
        THEN
            Q(0) <= NOT (Q(0));
        END IF ;
        IF FALLING_EDGE (Q(0))
        THEN
            Q(1) <= NOT (Q(1));
        END IF ;
        IF FALLING_EDGE (Q(1))
        THEN
            Q(2) <= NOT (Q(2));
        END IF ;
        IF FALLING_EDGE (Q(2))
        THEN
            Q(3) <= NOT (Q(3));
        END IF ;
    END PROCESS ;
END first;
```

The four 'IF' statements represent the four flip-flops. The 'FALLING_EDGE' function does what it says - checks the falling edge of the signal. Note that this is defined in the IEEE.Std_Logic_1164.ALL library, which is why this library is declared at the top of the code. Note also that VHDL will not detect the FALLING_EDGE of a BIT, which is why Clk has to be declared as STD_LOGIC type.

The other library (IEEE.Numeric_Std.ALL) is needed so that Q can be declared as a 4-bit number. Note that the various bits of Q are used both on the right and left side of the <= assignments, which is why Q has to be declared as an INOUT type.

Statements such as Q(0) <= NOT(Q(0)); are the way you make sure a signal toggles. Thus,

the first 'IF' statement causes Q(0) to toggle on every falling (i.e. negative) edge of Clk. The second one causes Q[1] to toggle on every negative edge of Q[0], and so on.

Try this out. Use pin 35 for the Clk and pins 46, 48, 49 and 50 for Q [0] , Q[1] , Q[2] and Q[3]. Connect the switch board to connector Port A and the LED board to Port B of the Matrix FPGA board. You should find that the system counts up in binary each time you release SW0.

Debounce

One solution to the switch-bounce problem is to use the SR latch circuit of Figure 10.1. We need two switches from the switch board - one to set and another to reset the latch. The switches still bounce but the spurious pulses only reinforce the first 'proper' signal. The circuit of Figure 10.6 shows the details, including the names of the signals involved in the new code.
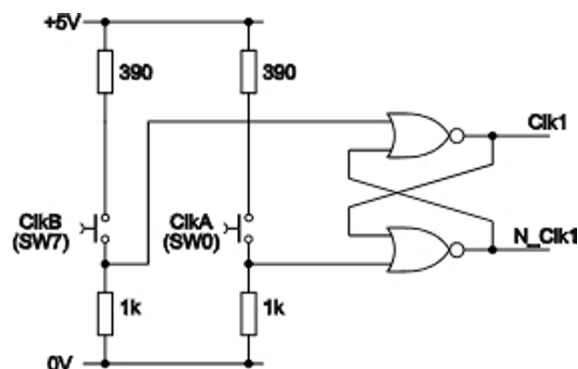
Fig. 10.6: Using an SR latch to de-bounce switches

The code uses the 'wire' keyword to declare the two internal signals Clk1 and N_Clk1 , which are 'wired up' to the ClkA and ClkB inputs. Clk1 is then used to clock the Q0 flip-flop.

Assign pin 45 to ClkB , compile and download to the FPGA board. Clicking and releasing SW0 and SW7 alternately will generate nice clean clock pulses for Q0, and the system should now count reliably.

Hence:

```
module asynch_counter (ClkA,ClkB,Q);
    input ClkA,ClkB;
    output [3:0] Q;
    reg [3:0] Q;
    wire Clk1, N_Clk1;
    //Two NOR gates to de-bounce the switches assign
    Clk1 = !(ClkB | N_Clk1);
    assign N_Clk1 = !(ClkA | Clk1);

    //Use Clk1 to clock the first flip-flop
    always @ ( posedge Clk1)
        Q[0] = !Q[0];
    always @ ( negedge Q[0])
        Q[1] = !Q[1];
    always @ ( negedge Q[1])
        Q[2] = !Q[2];
    always @ ( negedge Q[2])
        Q[3] = !Q[3];
endmodule
```

Asynchronous BCD up-counter

BCD stands for Binary-Coded-Decimal. It is a binary counter that counts only as far as 9. On the next clock pulse, it , resets to 0, instead of counting on up to 15 .

One way to achieve this is to generate a reset signal when the count reaches decimal 10, and use this to force the count back to 0. The circuit of Figure 10.7 does this. When the counter reaches 10 (binary 1010) outputs Q3 and Q1 will be HIGH (and Q2 and Q0 LOW). Signals from Q3 and Q1 feed the NAND gate to generate the active-LOW signal which clears all four flip-flops.
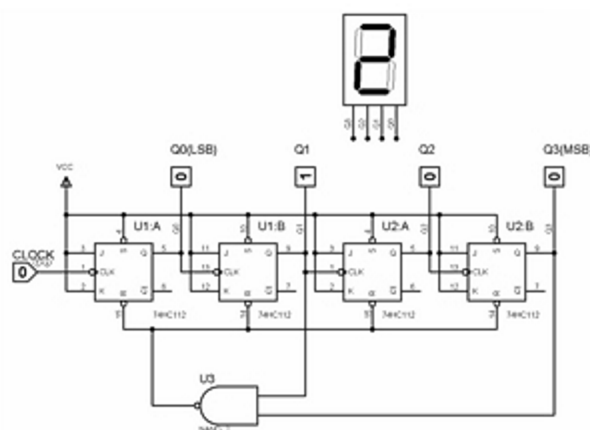
Fig. 10.7: BCD up counter (reset on 10).

This design is provided on the CD-ROM. Verify that it counts 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, ... The reset happens so quickly that you will not notice the brief time when the count actually reaches 10.

Verilog implementation of asynchronous BCD up-counter

The listing below implements the circuit of Figure 4.7

```verilog
module asynch_BCD_counter (ClkA,ClkB,Q);
    input ClkA,ClkB;
    output [3:0] Q;
    reg [3:0] Q;
    wire Clk1, N_Clk1, N_Reset;
    assign Clk1 = !(ClkB | N_Clk1);
    assign N_Clk1 = !(ClkA | Clk1);
    assign N_Reset = !(Q[3] & Q[1]);

    always @ ( posedge Clk1, negedge N_Reset)
    if (!N_Reset)
        Q[0] = 0;
    else
        Q[0] = !Q[0];
    always @ ( negedge Q[0], negedge N_Reset)
    if (!N_Reset)
        Q[1] = 0;
    else
        Q[1] = !Q[1];
    always @ ( negedge Q[1], negedge N_Reset)
    if (!N_Reset)
        Q[2] = 0;
    else
        Q[2] = !Q[2];
    always @ ( negedge Q[2], negedge N_Reset)
    if (!N_Reset)
        Q[3] = 0;
    else
        Q[3] = !Q[3];
endmodule
```

As before, each flip-flop is implemented by an 'always' statement. However, the active-low reset on each flip-flop has to be included in the event control list - hence the addition of the 'negedge N_Reset' . The action of the reset is implemented using the 'if (!N_Reset)' structure. If the N_Reset signal is LOW, then make the Q output go LOW otherwise toggle the Q output.

The N_Reset signal itself is declared as a 'wire' , and implemented as a NAND gate using an appropriate assign statement.
The use of ClkA and ClkB has been included in order to de-bounce the input switches from the switch board.
Compile and download this design to the FPGA board, using the same pin assignments as for the binary counter.

You should find it counts up to 1001 (9) but then, instead of resetting back to 0 the display shows 0100 (4) at the next clock pulse. The reason for this is shown in the timing diagram of Figure 10.8.

Fig. 10.8: Timing diagram showing problem with BCD counter

At 900ns the Clk goes LOW. After a 10ns delay this affects Q[0], which changes from HIGH to LOW. This negative edge is, of course, the Clk input for Q[1], so after a slightly shorter delay than before (just 8ns), Q[1] changes from LOW to HIGH.

We now have HIGHs on both Q[3] and Q[1], so this causes the NAND gate to make N_Reset go LOW. This happens 26ns after the original negative edge of the input Clk. After just 1ns (at 927ns), signals Q[1] and Q[3] are reset LOW. So far, so good - this is what is supposed to happen. The count has reached decimal 10, the reset signal has been generated, and all the outputs are LOW . However, there is now a race. The negative edge on Q[1] at 927ns causes Q[2] to toggle (at 935ns). Meanwhile, the LOW on N_Reset disappears at 935ns, allowing Q[2] to remain HIGH , generating the 0100 (4) display seen on the FPGA board.

If you build this circuit from 'traditional' ICs it works satisfactorily, so why not with the FPGA? It is just the case that the propagation delays of the gates inside the FPGA chip turn out to be unsatisfactory for this design.

This is the kind of problem that occurs when using 'asynchronous' designs. So much depends on whether one delay is greater than another, that it is often difficult to pinpoint the cause of such problems. In the next example we look at 'synchronous' design techniques, which eliminate such considerations.

VHDL implementation of asynchronous BCD up counter

The listing below implements the circuit of Figure 10.7

```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL, IEEE.Numeric_Std.ALL;
ENTITY asynch_BCD_counter IS
PORT
(
    ClkA,ClkB: IN STD_LOGIC;
    Q: INOUT UNSIGNED(3 DOWNTO 0)
);
```

```
END asynch_BCD_counter;

ARCHITECTURE a OF asynch_BCD_counter IS
SIGNAL Clk1, N_Clk1, N_Reset: STD_LOGIC;
BEGIN
    Clk1 <= NOT (ClkB OR N_Clk1);
    N_Clk1 <= NOT (ClkA OR Clk1);
    N_Reset <= NOT (Q(1) AND Q(3));
    PROCESS (Clk1, N_Reset, Q)
    BEGIN
        IF (N_Reset = '0')
        THEN
            Q(0) <= '0';
        ELSIF RISING_EDGE (Clk1)
        THEN
            Q(0) <= NOT (Q(0));
        END IF ;
        IF (N_Reset = '0')
        THEN
            Q(1) <= '0';
        ELSIF FALLING_EDGE (Q(0))
        THEN
            Q(1) <= NOT (Q(1));
        END IF ;
        IF (N_Reset = '0')
        THEN
            Q(2) <= '0';
        ELSIF FALLING_EDGE (Q(1))
        THEN
            Q(2) <= NOT (Q(2));
        END IF ;
        IF (N_Reset = '0')
        THEN
            Q(3) <= '0';
        ELSIF FALLING_EDGE (Q(2))
        THEN
            Q(3) <= NOT (Q(3));
        END IF ;
    END PROCESS ;
END a;
```

As before, each flip-flop is implemented by an 'IF' statement. The action of the reset is implemented using the 'IF (N_Reset = '0')' structure. If the N_Reset signal is LOW, then make the Q output go LOW, otherwise toggle the Q output. The N_Reset signal itself is declared as a 'SIGNAL' , within the architecture section of the listing. The line 'N_Reset <= NOT(Q(1) AND Q(3));' is equivalent to the NAND gate of the circuit diagram. The use of ClkA and ClkB has been included in order to de-bounce the input switches from the switch board.

Compile and download this design to the FPGA board, using the same pin assignments as for the binary counter. You should find it counts up to 1001 (9), but then, instead of resetting

(to 0), the display shows 0100 (4) at the next clock pulse. The reason for this is shown in the timing diagram of Figure 10.8(*bis*).
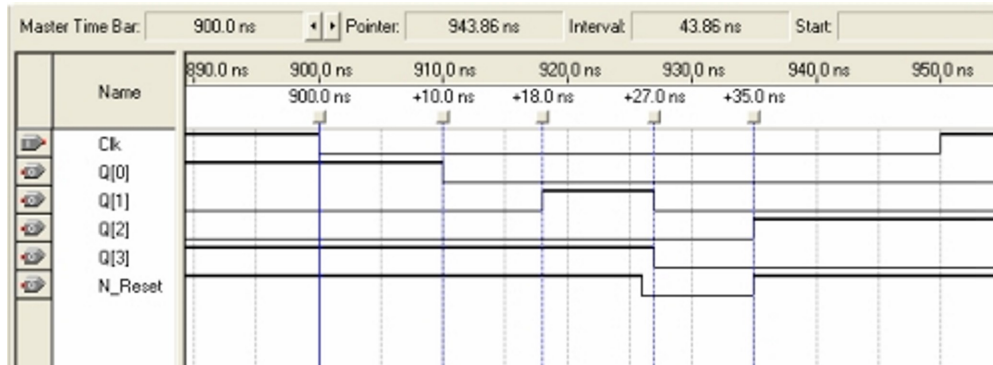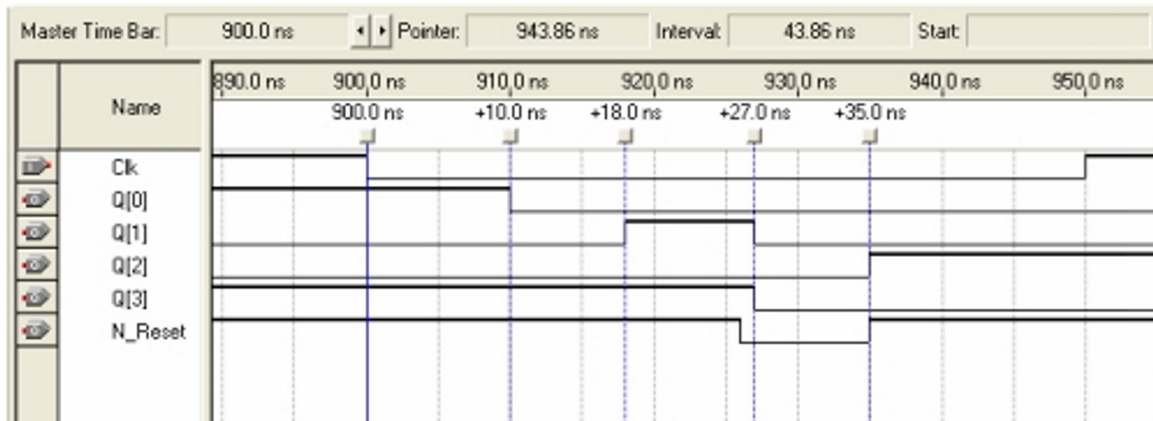

Figure 10.8(*bis*): timing diagram showing problem with BCD counter

At 900ns the Clk goes LOW. After a 10ns delay this affects Q[0], which changes from HIGH to LOW. This negative edge is, of course, the Clk input for Q[1], so after a slightly shorter delay than before (just 8ns), Q[1] changes from LOW to HIGH.

We now have HIGHs on both Q[3] and Q[1], so this causes the NAND gate to make N_Reset go LOW. This happens 26ns after the original negative edge of the input Clk. After just 1ns (at 927ns), signals Q[1] and Q[3] are reset LOW. So far, so good - this is what is supposed to happen. The count has reached decimal 10, the reset signal has been generated, and all the outputs are LOW . However, there is now a race. The negative edge on Q[1] at 927ns causes Q[2] to toggle (at 935ns). Meanwhile, the LOW on N_Reset disappears at 935ns, allowing Q[2] to remain HIGH, generating the 0100 (4) display seen on the FPGA board.

If you build this circuit from 'traditional' ICs it works satisfactorily, so why not with the FPGA? It is just the case that the propagation delays of the gates inside the FPGA chip turn out to be unsatisfactory for this design.

This is the kind of problem that occurs when using 'asynchronous' designs. So much depends on whether one delay is greater than another, that it is often difficult to pinpoint the cause of such problems. In the next example we look at 'synchronous' design techniques, which eliminate such considerations.

Synchronous counters

The counters of the previous sections have been described as 'asynchronous'. What's asynchronous about them? Well, although all the outputs appear to change simultaneously when you watch the simulation closely, there is a short delay between Q0 changing and Q1 changing state. Figure 10.8 indicates that this delay is only 8ns, but the outputs are slightly out of synch with each other. This can lead to design problems, as experienced over the FPGA implementation of the BCD counter.

The outputs of synchronous counters, on the other hand, all change at exactly the same instant. You can tell whether a system is synchronous by checking where each flip-flop gets its clock. If they all use the *same* clock signal, then all the flip-flop outputs will change

together, and the system will be synchronous.

How do you design synchronous systems? An elegant technique called a 'finite state machine' is used, illustrated below.
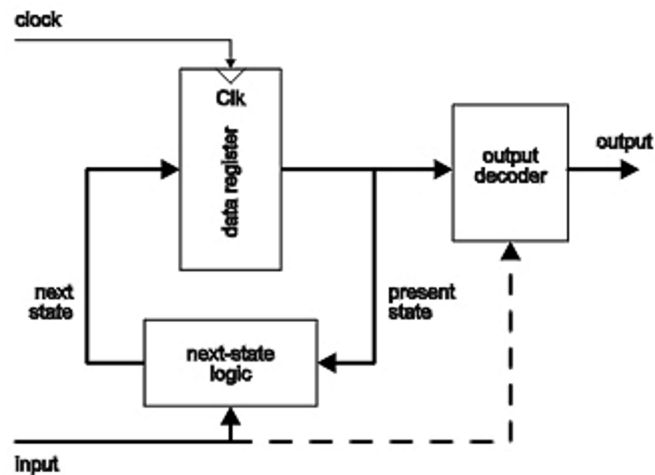
Fig. 10.9: General-purpose finite state machine

If you ignore the dotted line, Figure 10.9 illustrates what's known as a ' Moore machine', part of a 'Mealy machine'. (E. F. Moore and G. H. Mealy were pioneers in the design of sequential logic.) The heavy lines represent data buses, i.e. collections of individual signals grouped together.

At the top of the diagram there's a data register. As mentioned in the section on D-type flip-flops, this is a device for remembering a number. Each time the clock signal goes HIGH , the data register accepts a new binary number from its input, and presents it at its output.

The number held in the data register represents the 'present state' of the system. When the data register is clocked, the 'next state' is read in. Where does the next state come from? The 'next-state logic' block generates it.

This block looks at the present state of the system, and works out what the next state should be, taking into account any possible input signals to the whole system. The behaviour of the whole system is thus governed almost entirely by the way the next-state logic does its job.

The next-state logic block is *combinational*, so, although it might be tedious work, it's relatively easy to design, using the techniques like Karnaugh maps covered earlier.

The output decoder is another combinational logic block. It looks at the current internal state of the system and generates the required outputs. The number of individual signals in the output bus will probably be different from the number of signals making up the present-state, next-state buses.

Synchronous binary up counter

As a first example, we design a 3-bit binary up counter. For this example, there are no external inputs to the next-state logic, and the output decoder is not required, so the result is:
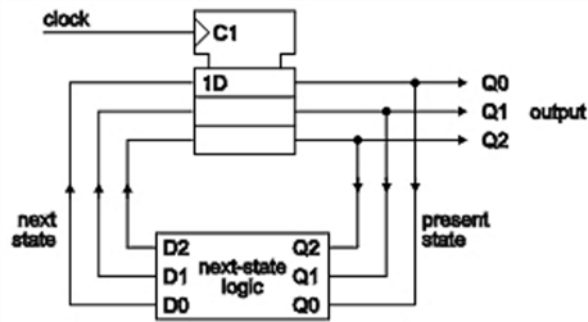
Fig 10.10: 3-bit finite state machine

To turn this general-purpose finite state machine into a binary counter, the circuitry within the 'next-state logic' block has to be designed appropriately, and the first step in this process is to write down the truth table. Note that the Q signals are inputs, while the D's are outputs. Effectively, three truth tables are needed – one for each output – but they are combined in the table below.

	present state			next state		
	Q2	Q1	Q0	D2	D1	D0
0	0	0	0	0	0	1
1	0	0	1	0	1	0
2	0	1	0	0	1	1
3	0	1	1	1	0	0
4	1	0	0	1	0	1
5	1	0	1	1	1	0
6	1	1	0	1	1	1
7	1	1	1	0	0	0

Fig. 10.11: Truth table (present state – next state table) for binary counter FSM

How have the entries in this table been derived? Easy! Just look at any given present state, work out what the next state should be (in this case, just count up by one), and write that state down on the right hand (next state) side of the table. Thus row 0 shows that the next state after 000 is 001. The only (slightly) tricky one is row 7: the next state after 111 is 000.

The next task is to use circuit minimization techniques to create circuits for each of the three outputs. By just looking at the truth table you can sometimes spot solutions. If you can't, you just work through Karnaugh maps, etc to arrive at your circuit. Looking at the truth table for D0, it is easy to see that it is simply the complement of Q0, so a single NOT gate will suffice to generate the D0 signal.
You may recognize the 0110 pattern for D1 as exclusive-OR: D1 = Q1 Å Q0.

The 0001 pattern for D2 is AND, while the 1110 pattern is NAND. We need to AND Q1 and Q0 while Q2 is LOW , but then NAND them when it is HIGH . The conditional inverter properties of the exclusive-OR can be utilized here, together with an AND gate. The complete circuit is as follows:
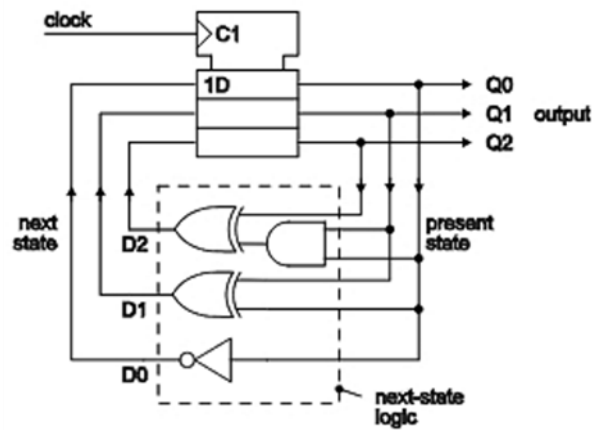
Fig. 10.12: 3-bit binary up counter

The circuit may be re-drawn as shown below:
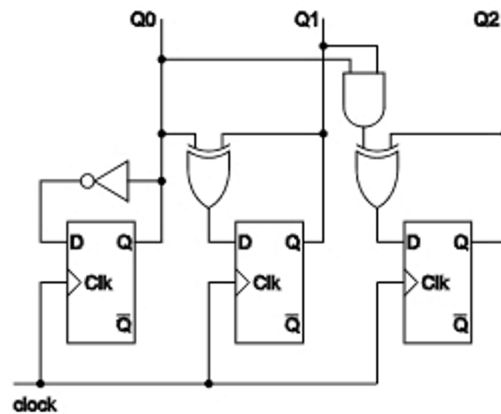
Fig. 10.13: 3-bit binary up counter re-drawn

The number of bits in the counter can easily be extended. Think about the circuit like this: the feedback from each flip-flop's output to its own input causes the flip-flop to toggle. The first (Q0) section has a simple inverter, so it toggles at every rising edge of the input clock.

The other sections have controlled inverters in the form of the exclusive-OR gates. These feed back inverted forms of their output only when the preceding outputs are all HIGH . The circuit below shows how a four-bit counter can be implemented using Proteus.
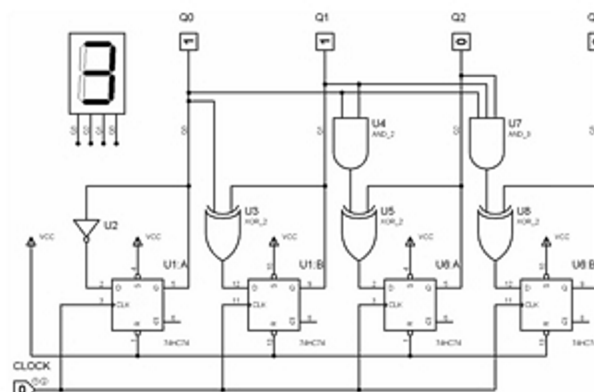
Fig. 10.14: 4-bit synchronous binary up counter

Verilog implementation of synchronous binary counter

The listing below implements the design of Figure 10.14, the 4-bit synchronous binary up counter.

```verilog
module synch_counter (ClkA,ClkB,Q);
    input ClkA, ClkB;
    output [3:0] Q;
    reg [3:0] Q;
    wire [3:0] D;
    wire Clk1, N_Clk1;
    assign Clk1 = !(ClkB | N_Clk1);
    assign N_Clk1 = !(ClkA | Clk1);
    assign D[0] = !Q[0];
    assign D[1] = Q[0] ^ Q[1];
    assign D[2] = (Q[0] & Q[1]) ^ Q[2];
    assign D[3] = (Q[0] & Q[1] & Q[2]) ^ Q[3];

    always @ ( posedge Clk1)
        Q = D;
endmodule
```

The first two assign statements implement a de-bounced clock, so that you can check out the action of this design on the Matrix FPGA board. This is the same as for the asynchronous designs, previously investigated.
The other four are effectively the combinational logic making up the 'next-state logic' of the 3-bit finite state machine and the 3-bit binary up counter.
The 'always' structure implements the bank of four D-type flip-flops making up the data register for this design. All the D inputs get clocked to the Q outputs on the rising edge of Clk1 .
Using suitable pin assignments for ClkA and ClkB (e.g. 35 and 37), compile this design and download it to the FPGA board. Verify that the LEDs count up in binary.

VHDL implementation of synchronous binary counter

The listing below implements the design of Figure 10.14, the 4-bit synchronous binary up counter.

```vhdl
LIBRARY IEEE;
 USE IEEE.Std_Logic_1164.ALL, IEEE.Numeric_Std.ALL;
 ENTITY synch_counter IS
 PORT
 (
    ClkA,ClkB: IN STD_LOGIC;
    Q: INOUT UNSIGNED(3 DOWNTO 0)
 );
 END synch_counter;

 ARCHITECTURE a OF synch_counter IS
 SIGNAL Clk1, N_Clk1: STD_LOGIC;
 SIGNAL D: UNSIGNED (3 DOWNTO 0);
```

```
BEGIN
    Clk1 <= NOT (ClkB OR N_Clk1);
    N_Clk1 <= NOT (ClkA OR Clk1);
    D(0) <= NOT Q(0);
    D(1) <= Q(0) XOR Q(1);
    D(2) <= (Q(0) AND Q(1)) XOR Q(2);
    D(3) <= (Q(0) AND Q(1) AND Q(2)) XOR Q(3);
    PROCESS (Clk1)
    BEGIN
        IF RISING_EDGE (Clk1)
        THEN
            Q <= D;
        END IF ;
    END PROCESS ;
END a;
```

The first two <= statements implement a de-bounced clock, so that you can check out the action of this design on the Matrix FPGA board. This is the same as for the asynchronous designs, previously investigated.

The other four are effectively the combinational logic making up the 'next-state logic' of the 3-bit finite state machine and 3-bit binary up counter.

The 'PROCESS' structure implements the bank of four D-type flip-flops making up the data register for this design. All the D inputs get clocked to the Q outputs on the rising edge of Clk1 .

Using suitable pin assignments for ClkA and ClkB (e.g. 35 and 37), compile this design and download it to the FPGA board. Verify that the LEDs count up in binary.

Synchronous 0-5 up-down counter

We now tackle a slightly more ambitious design - a counter that counts up or down in the range 0 – 5. In addition to the clock input, this design has a control input that determines whether the count increments or decrements at each clock pulse.
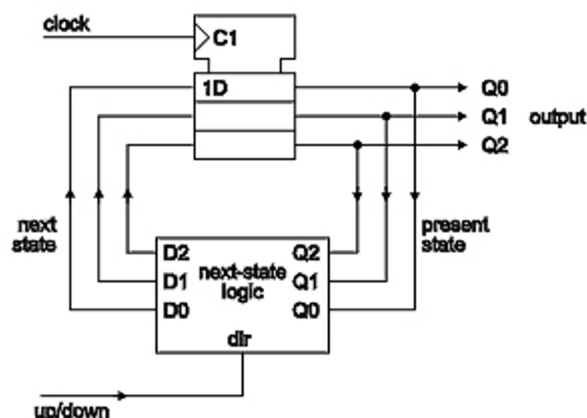
Fig. 10.15: 3-bit FSM with control input

As before, the process will be:
• write out the present state / next state table;

- devise circuitry to create the next state from the present state;
- implement/test the design in Proteus;
- implement the design in Verilog and/or VHDL and test on the Matrix FPGA board.

There are four inputs to the next-state logic, so the truth table has 16 rows.

	dir	present state			next state		
		Q2	Q1	Q0	D2	D1	D0
0	0	0	0	0	1	0	1
1	0	0	0	1	0	0	0
2	0	0	1	0	0	0	1
3	0	0	1	1	0	1	0
4	0	1	0	0	0	1	1
5	0	1	0	1	1	0	0
6	0	1	1	0	x	x	x
7	0	1	1	1	x	x	x
8	1	0	0	0	0	0	1
9	1	0	0	1	0	1	0
10	1	0	1	0	0	1	1
11	1	0	1	1	1	0	0
12	1	1	0	0	1	0	1
13	1	1	0	1	0	0	0
14	1	1	1	0	x	x	x
15	1	1	1	1	x	x	x

The entries in this table are calculated as follows.

When the direction signal (dir) is LOW, the counter is required to count down so the next state is just one less than the present state. When the present state is 0, however, the next state must be 5, since this is the maximum value the output is allowed. This particular situation is shown on row 0 of the table.

Rows 1 to 5 show the other normal count-down states. Since the count is restricted to values between 0 and 5, rows 6 and 7 should never happen. Bearing this in mind, we don't care what the next state is and this fact is indicated by the x's in the table.

Rows 8 to 12 show normal counting up. Row 13 shows the count rolling over from 5 back to 0, and rows 14 and 15 are 'impossible' so don't care rows.

Now, we have to devise minimized functions for D0, D1 and D2.

D0 is again easy - just make it the complement of Q0.

D1 and D2 are less obvious.

This time, we will make use of Karnaugh maps to derive solutions as shown in Figures 4.17 and 4.18 below.
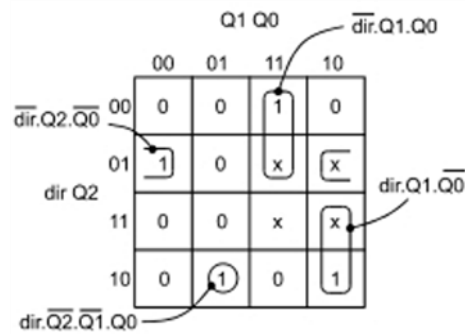
Fig. 10.17: Karnaugh map for: D1=!dir.Q2.!Q0+!dir.Q1.Q0+dir.Q1.!Q0+dir.!Q2.!Q1.Q0
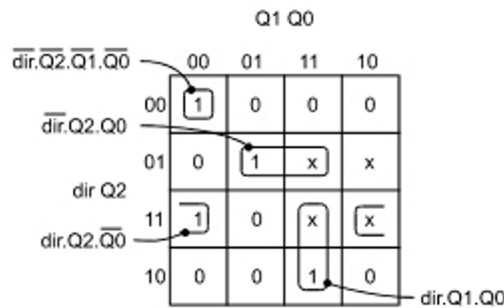
Fig. 10.18: Karnaugh map for: D2=!dir.Q2.Q0+dir.Q2.!Q0+dir.Q1.Q0+!dir.!Q2.!Q1.!Q0

The resulting circuit is shown below.
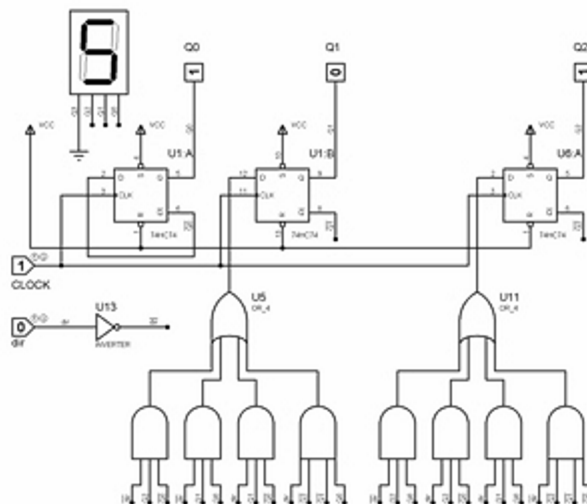
Fig. 10.19: Proteus implementation of synchronous 0 – 5 up/down counter

Using the Proteus design, verify that the design works correctly. It should count up when the dir control is HIGH (and down when it's LOW). The counter should count just in the range 0 – 5.

Verilog implementation of up/down 0 – 5 counter

The listing below implements the circuit of Figure 10.19, the synchronous 0 – 5 up/down counter.

```verilog
module synch_ud_counter (ClkA,ClkB,dir,Q);
    input ClkA, ClkB, dir;
    output [2:0] Q;
    reg [2:0] Q;
    wire [2:0] D;
    wire Clk1, N_Clk1;
    assign Clk1 = !(ClkB | N_Clk1);
    assign N_Clk1 = !(ClkA | Clk1);
    assign D[0] = !Q[0];
    assign D[1] = (!dir & Q[2] & !Q[0])
        |(!dir & Q[1] & Q[0])
        |(dir & Q[1] & !Q[0])
        |(dir & !Q[2] & !Q[1] & Q[0]);
    assign D[2] = (!dir & Q[2] & Q[0])
        |(dir & Q[2] & !Q[0])
        |(dir & Q[1] & Q[0])
        |(!dir & !Q[2] & !Q[1] &!Q[0]);

    always @ (posedge Clk1)
        Q = D;
endmodule
```

The code has the same structure as for the up-counter. The ClkA and ClkB signals are used in an SR latch to create a de-bounced clock, Clk1. This is used to clock the data register, allowing the three bits of D to appear on the output Q.
The D signals are generated using equations taken straight from the Karnaugh maps.
Using pin 45 for signal dir, compile and download this design to the Matrix FPGA board.
Verify that if SW7 is pressed, the LEDs count up in the range 0 – 5 (000 – 101 binary), otherwise they count down.

VHDL implementation of up/down 0 – 5 counter

The listing below implements the circuit of Figure 10.19, the synchronous 0 – 5 up/down counter.

```vhdl
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL, IEEE.Numeric_Std.ALL;
ENTITY synch_ud_counter IS
PORT
(
    ClkA,ClkB,dir: IN STD_LOGIC;
    Q: INOUT UNSIGNED(2 DOWNTO 0)
);
END synch_ud_counter;

ARCHITECTURE a OF synch_ud_counter IS
```

```
            SIGNAL Clk1, N_Clk1: STD_LOGIC;
            SIGNAL D: UNSIGNED (2 DOWNTO 0);
            BEGIN
                Clk1 <= NOT (ClkB OR N_Clk1);
                N_Clk1 <= NOT (ClkA OR Clk1);
                D(0) <= NOT Q(0); D(1) <= ( NOT dir AND Q(2) AND NOT Q(0))
                    OR ( NOT dir AND Q(1) AND Q(0))
                    OR (dir AND Q(1) AND NOT Q(0))
                    OR (dir AND NOT Q(2) AND NOT Q(1) AND Q(0));
                D(2) <= ( NOT dir AND Q(2) AND Q(0))
                    OR (dir AND Q(2) AND NOT Q(0))
                    OR (dir AND Q(1) AND Q(0))
                    OR ( NOT dir AND NOT Q(2) AND NOT Q(1) AND NOT Q(0));
            PROCESS (Clk1)
            BEGIN
                IF RISING_EDGE (Clk1)
                THEN
                    Q <= D;
                END IF ;
            END PROCESS ;
        END a;
```

The code has the same structure as for the up-counter.
The ClkA and ClkB signals are used in an SR flip-flop to create a de-bounced clock, Clk1.
This is used to clock the data register, allowing the three bits of D to appear on the output Q.
The D signals are generated using equations taken straight from the Karnaugh maps.
Using pin 45 for signal dir, compile and download this design to the Matrix FPGA board.
Verify that if SW7 is pressed the LEDs count up in the range 0 – 5 (000 – 101 binary),
otherwise they count down.

Behavioural description of counters

In previous chapters, we saw how to use behavioural language to describe the required
operation of circuits. The de-multiplexer, for instance, could be described using the
following Verilog code:

```
if (control == 1)
    begin
        P = clock;
        Q = 0;
    end
else
    begin
        Q = clock;
        P = 0;
    end
or the following VHDL code:
BEGIN
    P <= clock WHEN control = '1' ELSE '0';
```

```
        Q <= clock WHEN control = '0' ELSE '0';
    END ;
```

Counters can be described very neatly in behavioural form. You just say, for example

Q = Q + 1; (Verilog)
or
Q <= Q + 1; (VHDL)

In these statements, Q would be a multi-bit signal, of course. Counters of size 8, 16, 32-bit or whatever you like can be implemented with just that simple line of code.

The following examples show how to describe a 4-bit BCD up-down counter, similar in action to the 3-bit 0 – 5 up-down counters studied earlier. You can hand over all the hard work of truth tables, Karnaugh maps, etc. to the compiler.

Verilog implementation of synchronous BCD up-down counter

```verilog
module synch_BCD_ud_counter (ClkA,ClkB,dir,Q);
    input ClkA, ClkB, dir;
    output [3:0] Q;
    reg [3:0] Q;
    wire Clk1, N_Clk1;
    assign Clk1 = !(ClkB | N_Clk1);
    assign N_Clk1 = !(ClkA | Clk1);
    always @ (posedge Clk1)

    if (dir)
        if (Q < 9)
            Q = Q + 1;
        else
            Q = 0;
    else
        if (Q > 0)
            Q = Q - 1;
        else
            Q = 9;
endmodule
```

The two 'assign' statements create a de-bounced clock (Clk1), as before.

The 'always' structure says that if the 'dir' (is HIGH), then Q becomes one bigger, provided it is less than 9. If it is 9 it becomes 0 on the next positive edge of Clk1. On the other hand, (if dir is LOW,) Q becomes one less, provided it is greater than 0. If it is 0 then it becomes 9 on the next positive edge of Clk1.

Enter this design, compile and download to the E-blocks FPGA board.
Verify that you get a 0-9 up/down counter.

VHDL implementation of synchronous BCD up-down counter

```vhdl
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL, IEEE.Numeric_Std.ALL;
ENTITY synch_BCD_ud_counter IS
PORT
(
    ClkA,ClkB,dir: IN STD_LOGIC;
    Q: INOUT INTEGER RANGE 0 TO 15
);
END synch_BCD_ud_counter;

ARCHITECTURE a OF synch_BCD_ud_counter IS
SIGNAL Clk1, N_Clk1: STD_LOGIC;
BEGIN
    Clk1 <= NOT (ClkB OR N_Clk1);
    N_Clk1 <= NOT (ClkA OR Clk1);
    PROCESS (Clk1)
    BEGIN
        IF RISING_EDGE (Clk1) THEN
            IF (dir = '1') THEN
                IF Q < 9  THEN
                    Q <= Q + 1;
                ELSE
                    Q <= 0;
                END IF ;
            ELSE
                IF Q > 0 THEN
                    Q <= Q - 1;
                ELSE
                    Q <= 9;
                END IF ;
            END IF ;
        END IF ;
    END PROCESS ;
END a;
```

The first two assignments in the architecture create a de-bounced clock (Clk1), as before. The 'PROCESS' structure says that, on each rising edge of Clk1 , if 'dir' is HIGH then Q increments by one, provided it is less than 9. If it is 9 it becomes 0. On the other hand, (if 'dir' is LOW,) Q decrements by one, provided it is greater than 0. If it is 0, then it becomes 9 on the next positive edge of Clk1 .

Notice that the Q port signal has been declared as an 'INTEGER' type in the range 0 to 15. This allows it to be set to values like 0 and 9. A 'BIT_VECTOR' type cannot be used, because VHDL doesn't allow the '+' sign to be used with such types. If you use the UNSIGNED type, you have to say "0000" instead of 0 and "1001" instead of 9.

Enter this design, compile and download to the E-blocks FPGA board. Verify that you get a 0-9 up/down counter.

Another state machine: a sequence detector

The synchronous counters discussed above have been called 'state machines' - the state of the machine being the value of the counter. Referring to the general purpose finite state machine, the output of the circuit is simply the output of the data register. There has been no need for an output decoder.
In our next design, things are different. The machine will move from one state to another, but the output will only change when the fourth state is reached. What do the states represent? The design is looking out for the sequence 1-0-1 in a stream of input data. Differing states show how close we are to finding that sequence.
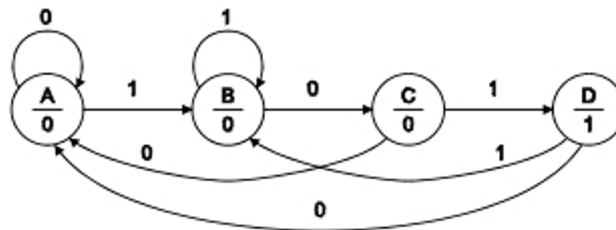
Fig. 10.20: State diagram for 1-0-1 sequence detector

In Figure 10.20, the different states are labelled A, B, C and D. The output from the machine is shown as the 0 or 1 beneath the state label. The input to the machine is shown as the 0 or 1 attached to the various arrows, and the arrows themselves show how the machine must move from one state to another.
State A is the initial state - nothing has happened so far. The value below the line shows that the output of the system is 0 - the sequence has not yet been detected. If the input signal is 0, then the system just stays in state A, as indicated by the curved arrow. If it's a 1, then we go to state B, since this could be the beginning of a 1-0-1 sequence. The system still outputs a 0 in state B, of course.
Having reached state B, if a 0 comes along the system moves to state C, since we are getting nearer to the sought-for sequence. If a 1 arrives while in state B, you might think the system should go back to state A but this latest 1 could itself be the start of a 1-0-1 sequence so the correct thing to do is stay in state B.
From state C, a 1 signifies that the 1-0-1 sequence has been detected, so the system transitions to state D, and outputs a 1 to signal the fact. A 0 at state C is disaster (the sequence must have been 1-0-0) so we go right back to state A.
The options from state D are back to A if a 0 arrives, but B if it's a 1.
Note that this system will only output a single 1 during a sequence such as 1-0-1-0 -1. The second 101 'overlaps' the original 101 and does not count as a separate instance of the sequence.

It can take a great deal of effort to get the state diagram correct for a complex system, but from there on the steps are fairly straightforward. The first is to create a present-state/next-state table. This just lists what the next state is, given the logic level on the input signal, P.

input (P)	present state	next state
0	A	A
0	B	C
0	C	A
0	D	A
1	A	B
1	B	B
1	C	D
1	D	B

Fig. 10.21: Present state/next state table for 1-0-1 detector

Row 1 shows that the system stays in state A if the input signal is 0, while row 5 shows it going from state A to state B if the input is 1.

The next step is to allocate codes to the states. For this example, binary coding will be used: A = 00; B = 01; C = 10 and D = 11. A 2-bit system is sufficient to code this 4-state machine, and two D-type flip-flops will be used. Labelling the MSB flip-flop Q1, with input D1, and the LSB flip-flop Q0, with input D0, gives a new version of the present-state/next-state table.

input	present state		next state	
P	Q1	Q0	D1	D0
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	1
1	0	1	0	1
1	1	0	1	1
1	1	1	0	1

Fig. 10.22: Present state/next state table encoded in binary

We can now devise circuitry to generate the D1 and D0 signals. The table above shows that D0 can be generated directly from the input signal, P.

D1 isn't quite so straightforward but a Karnaugh map may help.

$$D1 = \overline{P}.\overline{Q1}.Q0 + P.Q1.\overline{Q0}$$
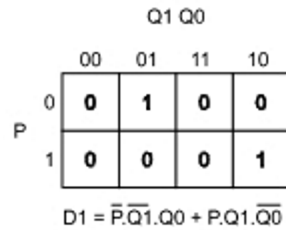
Fig. 10.23: Karnaugh map for D1

It turns out that the Karnaugh map reveals that no minimization is possible.

We now have most of the information required to create a circuit that will obey the state diagram of Figure 10.20. Two D-type flip-flops are needed, and the circuitry to drive the D inputs (the next-state logic) has been derived above. The only additional task is to make an output signal that goes HIGH when machine state D is reached. This is the 'output de-coder' of Figure 10.9 and Figure 10.24.
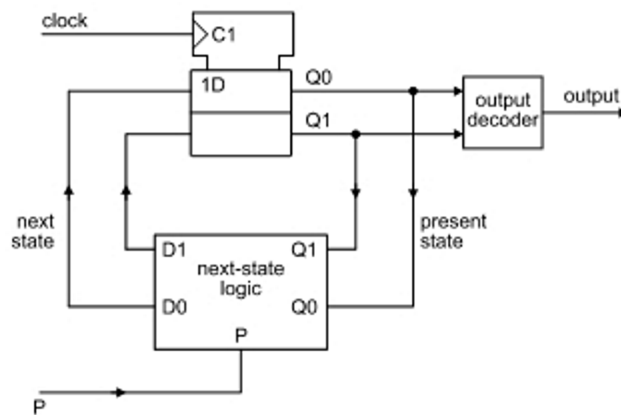
Fig. 10.24: State machine for 1-0-1 sequence detector

Designing the output decoder is easy. Remember, the output only goes HIGH when state D is reached, and state D is represented by both Q1 and Q0 going HIGH . A 2-input AND gate will do the job.

The circuit of the detector is given in Figure 10.25.
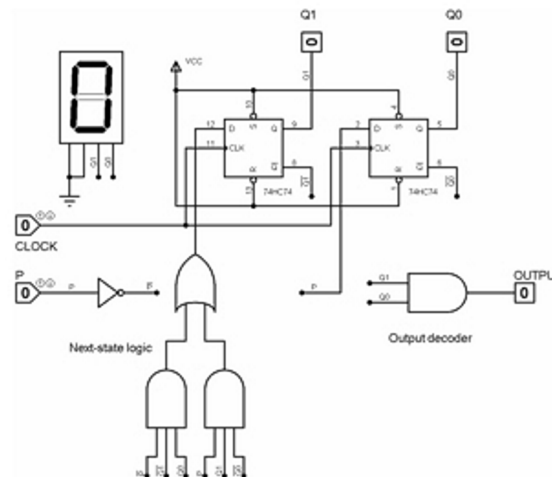
Fig. 10.25: Implementation of 1-0-1 detector

The diagram shows the detector in state A. The seven-segment display indicates the states

of internal signals Q1 and Q0, and we chose 00 to represent state A. Similarly, state B is shown as 1, state C as 2 and state D as 3.

Build the circuit in figure 10.25 in your circuit simulation package and run it. Verify that the only way to get the OUTPUT to go HIGH is to enter the sequence 1, 0, 1 on the P input, clocking the CLOCK (HIGH and LOW) at each point along the sequence.

Verilog implementation of 1-0-1 sequence detector

We can implement the circuit of the 1-0-1 sequence detector in Verilog code as shown below.

```verilog
module sequence_101 (ClkA,ClkB,P,Output);
    input ClkA, ClkB, P;
    output Output;
    wire Clk1, N_Clk1;
    wire [1:0] D;
    reg [1:0] Q;

    //Clock de-bouncer
    assign Clk1 = !(ClkB | N_Clk1);
    assign N_Clk1 = !(ClkA | Clk1);
    //Next-state logic
    assign D[0] = P;
    assign D[1] = (!P & !Q[1] & Q[0]) | (P & Q[1] & !Q[0]);
    //Output decoder
    assign Output = Q[1] & Q[0];
    //Data register
    always @ ( posedge Clk1)
        Q = D;
endmodule
```

The first two assignments create a de-bounced clock, Clk1 . The next two implement the next-state logic, and the last the output decoder. The two D-type flip-flops are implemented within the 'always' structure. D itself is declared as an internal 'wire' , and Q as an internal 'reg'.

With pins 35, 37, 45 and 46 assigned to signals ClkA , ClkB , P and Output respectively, compile and download the design to the Matrix FPGA board (with switch board attached to connector Port A and LED board to Port B). To get LED D0 to light up you will need to press and hold switch SW7 down, then click-and-release switch SW2 then click-and-release switch SW0. Then release switch SW7 and repeat the SW2/SW0 clicking. Then press/hold SW7 and do the SW2/SW0 thing.

Behavioural Verilog code for the 1-0-1 sequence detector

The theme of much of this course is that a FPGA development system, such as Quartus, will allow you to describe the required behaviour of a design, and will work out how to implement it.

So, how do you describe the 1-0-1 sequence detector?

Well, you have to describe the 1-0-1 sequence detector state diagram. There isn't any special syntax for this. You just use the case structure that has been mentioned earlier. The use of a parameter to name the states is new, however.

```verilog
module sequence_101 (ClkA,ClkB,P,Output);
    input ClkA, ClkB, P;
    output Output;
    wire Clk1, N_Clk1;
    parameter [1:0] A = 0, B = 1, C = 2, D = 3;
    reg [1:0] current_state, next_state;
    //Clock de-bouncer
    assign Clk1 = !(ClkB | N_Clk1);
    assign N_Clk1 = !(ClkA | Clk1);

    //Next-state logic
    always @(current_state,P)
    case (current_state)
        A: if (P) next_state = B; else next_state = A;
        B: if (P) next_state = B; else next_state = C;
        C: if (P) next_state = D; else next_state = A;
        D: if (P) next_state = B; else next_state = A;
    endcase

    always @ ( posedge Clk1)
        //Data register
        current_state = next_state;
        //Output decoder
        assign Output = (current_state == D);
endmodule
```

Note:

The first 'always' is used for the combinational logic, comprising the next-state logic, whilst the second one is used for the data register.

The output decoder is implemented in the final assign statement.

Although this listing is longer than the previous one, it does save you the chore of working out the next-state logic and, of course, it is easier to see how it should behave.

Chose some different pin assignments, compile, download and verify operation.

VHDL implementation of 1-0-1 sequence detector

The following shows how to implement the circuit of the 1-0-1 sequence detector in VHDL code:

```vhdl
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL;
ENTITY sequence_101 IS
PORT
(
    ClkA,ClkB,P: IN STD_LOGIC;
    Output: OUT STD_LOGIC
);
END sequence_101;

ARCHITECTURE circuit OF sequence_101 IS
SIGNAL Clk1, N_Clk1: STD_LOGIC;
SIGNAL Q, D: STD_LOGIC_VECTOR (1 DOWNTO 0);

BEGIN
    --Clock de-bouncer
    Clk1 <= NOT (ClkB OR N_Clk1);
    N_Clk1 <= NOT (ClkA OR Clk1);
    --Next-state logic
    D(0) <= P;
    D(1) <= ( NOT P AND NOT Q(1) AND Q(0))
        OR (P AND Q(1) AND NOT Q(0));
    --Output decoder
    Output <= Q(1) AND Q(0);
    PROCESS (Clk1)
    BEGIN
        --Data register
        IF RISING_EDGE (Clk1)
        THEN
            Q <= D;
        END IF ;
    END PROCESS ;
END circuit;
```

The first two '<=' statements create a de-bounced clock, Clk1 . The next two implement the next-state logic, and the last the output decoder. The two D-type flip-flops are implemented within the 'PROCESS' structure. D and Q are declared as internal 2-bit 'SIGNAL's.

With pins 35, 37, 45 and 46 assigned to signals ClkA , ClkB , P and Output respectively, compile and download the design to the E-blocks FPGA board (with switch board attached to Port A and LED board to Port B).

To get LED D0 to light up, you will need to:

- press and hold down switch SW7;
- click-and-release switch SW2;
- click-and-release switch SW0;
- release switch SW7;
- repeat the SW2/SW0 sequence;
- then press/hold switch SW7;
- go through the SW2/SW0 sequence again.

Behavioural VHDL code for the 1-0-1 sequence detector

The theme of much of this course is that a FPGA development system, such as Quartus, will allow you to describe the required behaviour of a design, and will work out how to implement it.

So, how do you describe the 1-0-1 sequence detector?

Well, you have to describe the 1-0-1 sequence detector state diagram. There isn't any special syntax for this. You just use the case structure that has been mentioned earlier. The use of an 'enumerated data type' to name the states is new, however.

```vhdl
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL;
ENTITY sequence_101 IS
PORT
(
    ClkA,ClkB,P: IN STD_LOGIC;
    Output: OUT STD_LOGIC
);
END sequence_101;

ARCHITECTURE states OF sequence_101 IS
TYPE State_type IS (A,B,C,D);
SIGNAL Clk1, N_Clk1: STD_LOGIC;
SIGNAL Current_state, Next_state: State_type;
BEGIN
    --Clock de-bouncer
    Clk1 <= NOT (ClkB OR N_Clk1);
    N_Clk1 <= NOT (ClkA OR Clk1);
    --Next-state logic
    PROCESS (Current_state,P)
    BEGIN
        CASE Current_state IS
            WHEN A => IF P = '1' THEN Next_state <= B; ELSE Next_state <= A;
END IF ;
            WHEN B => IF P = '1' THEN Next_state <= B; ELSE Next_state <= C;
END IF ;
            WHEN C => IF P = '1' THEN Next_state <= D; ELSE Next_state <= A;
END IF ;
            WHEN D => IF P = '1' THEN Next_state <= B; ELSE Next_state <= A;
END IF ;
```

```
        END CASE ;
    END PROCESS ;

    PROCESS (Clk1)
    --Data register
    BEGIN
        IF RISING_EDGE (Clk1) THEN
            Current_state <= Next_state;
        END IF ;
    END PROCESS ;
    --Output decoder
    Output <= '1' WHEN Current_state = D ELSE '0';
END states;
```

The 'TYPE' definition on the first line of the 'ARCHITECTURE' body allows the programmer to define their own data type. Here we have defined the State_type to consist of possible values A , B , C and D.

On the next line we have defined two signals, 'Current_state' and 'Next_state' , to be this data type. In other words, 'Current-state' can only take the 'enumerated' values A, B, C or D.

The first 'PROCESS' is used for the combinational logic comprising the next-state logic, whilst the second one is for the data register. The output decoder is implemented in the final line of the body of the 'ARCHITECTURE' .

Although this listing is longer than the previous one, it does save you the chore of working out the next-state logic and, of course, it is easier to see how it should behave.
Chose some different pin assignments, compile, download and verify operation.

Flashy turn indicator: a Mealy type Finite State Machine

The 1-0-1 detector is a Moore type state machine, since the output depends purely on the state of the machine.
This final example is a Mealy type, since the output will depend on both the state of the machine and the input signals. It looks at controlling a more ambitious turn indicator for a vehicle. The rear light cluster consists of two rows of six bulbs on either side. When turning left or right, a 'flashy' light sequence takes place. When braking, a different arrangement of lights comes on.
The state diagram, showing this behaviour, is shown below. In this and the discussion that follows, TL means the turn-left sequence, and TR the turn-right sequence. The labels on the transition arrows show that the TR signal controls turning right, while TL is for turning left. Transition arrows without labels are the default, i.e. neither turning right nor left.
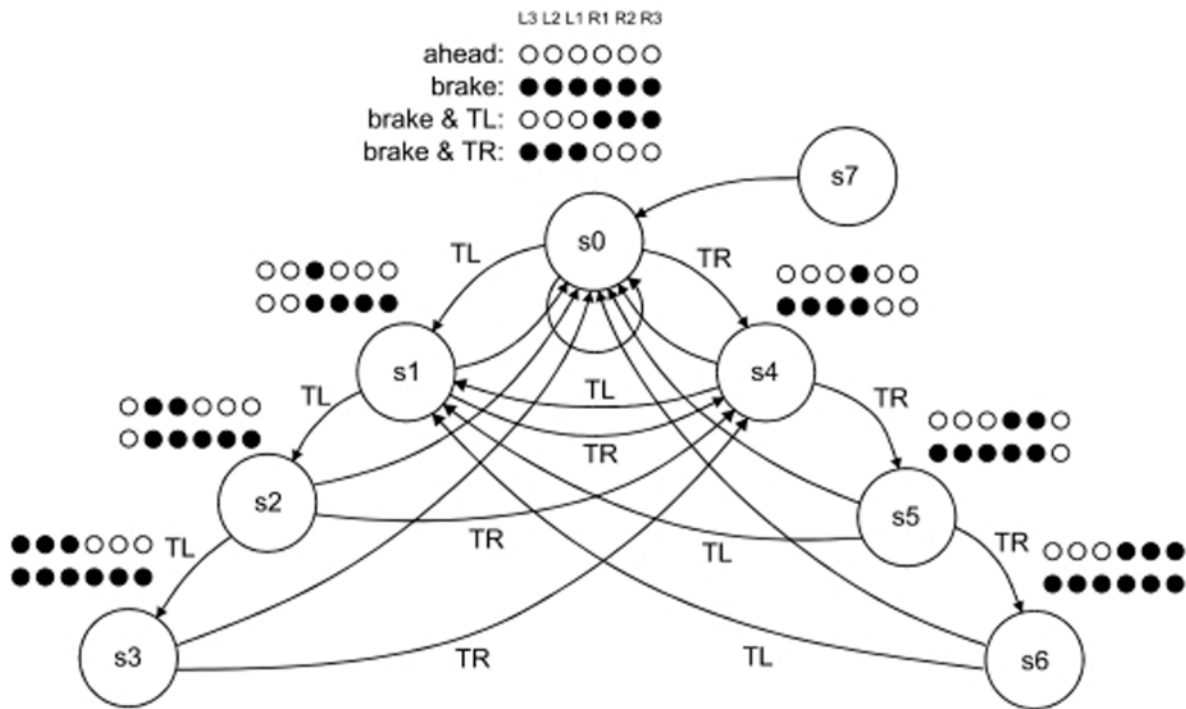
Fig. 10.26: State diagram for 'Flashy' turn indicator

Figure 10.26 illustrates the various states and the required transitions.

For each state, the blobs show how the six output indicator lamps should behave. The different rows of blobs show what should happen given different input conditions. The top row shows what should happen if the brake pedal is *not* pressed. When turning right, for example, the TR signal will be active and so the machine will cycle through states s0, s4, s5, s6, s0... - the right-hand set of indicator lamps will progressively light up. When the brake signal *is* active, all the left indicators light up as well, as shown in the bottom row of blobs.

State s0 is the 'straight-ahead' state and is slightly more complicated. The top row of blobs shows that, normally, no lights should light up. When the brake pedal is pressed (second row of blobs), they should all do so. The third and fourth rows show what should happen when braking and turning left or right respectively.

Seven states are needed for this system, and they will be coded using a 3-bit encoder. The spare eighth state (s7) is not used but has an unconditional transition to s0 just in case the state is entered when the system turns on.

The block diagram for the system is given in Figure 10.27.
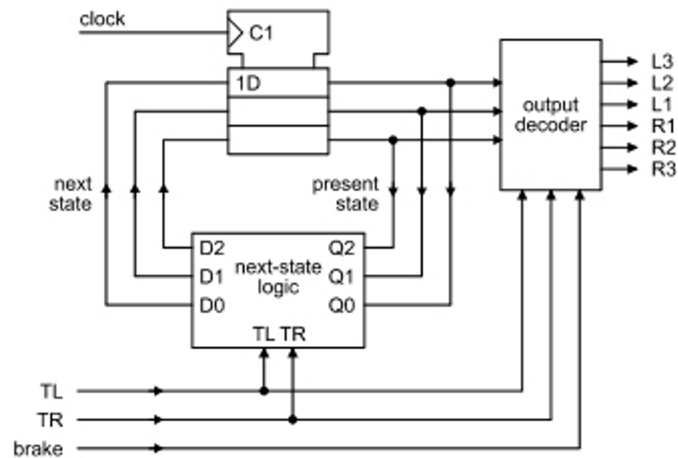
Fig. 10.27: Block diagram for flashy turn indicator

The design of the next-state logic and the design of the output decoder are tackled separately.

Design of the next-state logic:

There are five input signals, so the next state table will have 32 rows. It is shown in Figure 10.28 below, with the states encoded in binary order (s0 = 000, s1 = 001, etc.).

Row 0 shows state s0 staying in s0 when neither TL (the left-indicator sequence) nor TR (the turn-right indicator sequence) is active. The next seven rows show the system returning to state s0, from whatever state it is in when both TL and TR cease to be active.

In row 8 the TR signal is active, so the system moves from s0 to s4. Rows 9, 10 and 11 show the transitions from the turning left states (i.e. s1, s2 and s3) to the first of the turn right states (s4). The remaining rows up to row 23 are interpreted in similar fashion. The last eight rows have 'don't-care x symbols' since we assume that it will not be possible to generate a TR *and* a TL signal at the same time.

	Turn		Present			Next		
	TL	TR	Q2	Q1	Q0	D2	D1	D0
0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0
2	0	0	0	1	0	0	0	0
3	0	0	0	1	1	0	0	0
4	0	0	1	0	0	0	0	0
5	0	0	1	0	1	0	0	0
6	0	0	1	1	0	0	0	0
7	0	0	1	1	1	0	0	0
8	0	1	0	0	0	1	0	0
9	0	1	0	0	1	1	0	0
10	0	1	0	1	0	1	0	0
11	0	1	0	1	1	1	0	0
12	0	1	1	0	0	1	0	1
13	0	1	1	0	1	1	1	0
14	0	1	1	1	0	0	0	0
15	0	1	1	1	1	0	0	0
16	1	0	0	0	0	0	0	1
17	1	0	0	0	1	0	1	0
18	1	0	0	1	0	0	1	1
19	1	0	0	1	1	0	0	0
20	1	0	1	0	0	0	0	1
21	1	0	1	0	1	0	0	1
22	1	0	1	1	0	0	0	1
23	1	0	1	1	1	0	0	0
24	1	1	0	0	0	x	x	x
25	1	1	0	0	1	x	x	x
26	1	1	0	1	0	x	x	x
27	1	1	0	1	1	x	x	x
28	1	1	1	0	0	x	x	x
29	1	1	1	0	1	x	x	x
30	1	1	1	1	0	x	x	x
31	1	1	1	1	1	x	x	x

Fig. 10.28: Encoded next-state table for flashy turn indicator.

Using techniques not covered in this course, it can be shown that minimized expressions for D2, D1 and D0 are:

D2 = TR.!Q2+TR.!Q1
D1 = TR.Q2.!Q1.Q0+TL.1q2.!Q1.Q0+TL.!Q2.Q1.!Q0

$$D0 = TR.Q2.!Q1.!Q0+TL.Q2.!Q1+TL.!Q0$$

The design of the next-state logic is now complete.

Design of the output decoder

The output decoder has six inputs, so requires a 64-row table to be fully defined. Making use of don't care x symbols on both the input and output sides, it is shown in Figure 10.29.

	Present			Brake	Turn		Next Left			Next Right		
	Q2	Q1	Q0	b	TL	TR	L3	L2	L1	R1	R2	R3
0	0	0	0	0	x	x	0	0	0	0	0	0
1	0	0	0	1	0	0	1	1	1	1	1	1
2	0	0	0	1	0	1	1	1	1	0	0	0
3	0	0	0	1	1	0	0	0	0	1	1	1
4	0	0	0	1	1	1	x	x	x	x	x	x
5	0	0	1	0	x	x	0	0	1	0	0	0
6	0	0	1	1	x	x	0	0	1	1	1	1
7	0	1	0	0	x	x	0	1	1	0	0	0
8	0	1	0	1	x	x	0	1	1	1	1	1
9	0	1	1	0	x	x	1	1	1	0	0	0
10	0	1	1	1	x	x	1	1	1	1	1	1
11	1	0	0	0	x	x	0	0	0	1	0	0
12	1	0	0	1	x	x	1	1	1	1	0	0
13	1	0	1	0	x	x	0	0	0	1	1	0
14	1	0	1	1	x	x	1	1	1	1	1	0
15	1	1	0	0	x	x	0	0	0	1	1	1
16	1	1	0	1	x	x	1	1	1	1	1	1
17	1	1	1	x	x	x	x	x	x	x	x	x

Fig. 10.29: Output decoder truth table for flashy turn indicator

Figure 10.29 reflects the output requirements (the blobs) as shown on the state diagram (Figure 10.26). Row 0 reflects the situation while in state s0 - so long as the brake isn't pressed, none of the output indicators should light up, even if you are turning. Rows 1, 2 and 3 show the situation if the brake is pressed and the various possible turns (no turn, turn right or turn left) are taking place. Row 4 takes care of state 0, brake pressed, and the impossible turning left and turning right situation.

Rows 5 to 16 enumerate states s1 to s6, and the two possible levels of the brake signal for each case. For these states the logic levels on the TL and TR signals are irrelevant, as far as the output signals are concerned.

The last row deals with state s7, which should never be reached.

The following expressions can be derived for L3 to R3:

L3 = !Q1.!Q0.b.!TL+Q1.Q0+Q2.b
L2 = !Q0.b.!TL+!Q2.Q1+Q2.b
L1 = !Q2.Q1+!Q2.Q0+b.!TL+Q2.b
R1 = Q1.b+b!TR+Q0.b+Q2
R2 = !Q2.b.!TR+Q2.Q0+Q0.b+Q2.Q1+Q1.b
R3 = !Q2.b.!TR+!Q2.Q0.b+Q2.Q1+Q1.b

The design-work is now complete.

Figure 10.30 shows the circuit. (The next-state logic and output decoder circuit blocks are not shown.)
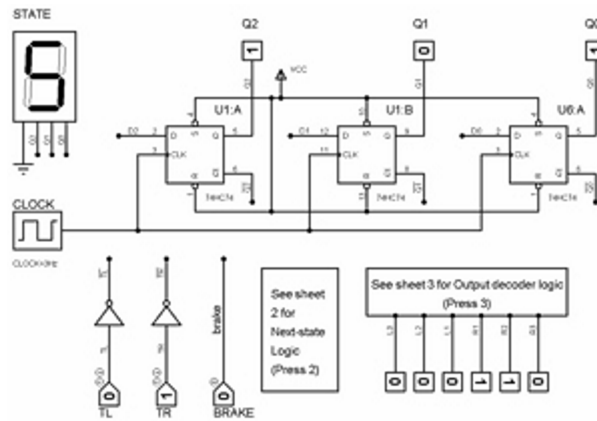
Fig. 10.30: Circuit diagram of Flashy turn indicator

The diagram shows the system in state s5 (second of the turn right states) without the brake pedal pressed. Simulate this circuit and verify that the circuit behaves in accordance with the state diagram, Figure 10.26.

The Proteus design uses the inbuilt CLOCK (set to oscillate at 3Hz) to drive the next-state flip-flops. An equivalent signal would be useful in a FPGA implementation, and the Matrix FPGA board provides such a facility.

This starts with crystal oscillator, made from a 25MHz crystal, a couple of resistors and two capacitors, connected to a logic inverter as shown below.
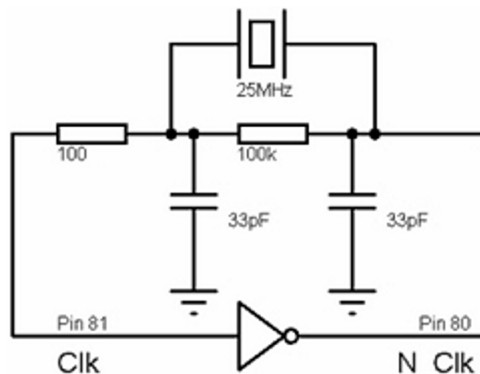
Fig. 10.31: Crystal oscillator

There are plenty of inverters inside the FPGA chip and the following code examples incorporate this.

The output of this circuit (N_Clk) is then divided by two, 23 times to give an approximately 3Hz signal for the next-state flip-flops.

Verilog implementation of flashy turn indicator

```verilog
module flashy (Clk,N_Clk,TL,TR,brake,L3,L2,L1,R1,R2,R3);
    input Clk, TL, TR, brake;
    output N_Clk;
    output L3,L2,L1,R1,R2,R3;
    reg L3,L2,L1,R1,R2,R3;
    reg [2:0] current_state, next_state;
    parameter [2:0] s0 = 0, s1 = 1, s2 = 2, s3 = 3, s4 = 4, s5 = 5, s6 = 6,
s7 = 7;
    reg [22:0] Q;

    assign N_Clk = !Clk; //to form 25MHz xtal oscillator

    //divide 25MHz by 2^23 = 2.98 Hz
    always @ ( posedge N_Clk)
        Q = Q + 1;
    //use 2.98Hz clock to go from one state to next
    always @ ( posedge Q[22])
        current_state = next_state;
    always @(current_state,TL,TR)
    //define state machine
    case (current_state)
        s0: if      (TL) next_state = s1;
            else if (TR) next_state = s4;
            else         next_state = s0;
        s1: if      (TL) next_state = s2;
            else if (TR) next_state = s4;
            else         next_state = s0;
        s2: if      (TL) next_state = s3;
            else if (TR) next_state = s4;
            else         next_state = s0;
        s3: if      (TR) next_state = s4;
            else         next_state = s0;
        s4: if      (TR) next_state = s5;
            else if (TL) next_state = s1;
            else         next_state = s0;
        s5: if      (TR) next_state = s6;
            else if (TL) next_state = s1;
            else         next_state = s0;
        s6: if      (TL) next_state = s1;
            else         next_state = s0;
        s7:              next_state = s0;
    endcase

    always @ (current_state, brake, TL, TR)
```

```verilog
//output decoder logic
    case (current_state)
        s0: if (brake == 0)                    {L3,L2,L1,R1,R2,R3} = 6'b000000;
            else if ({brake,TL,TR} == 3'b100)  {L3,L2,L1,R1,R2,R3} = 6'b111111;
            else if ({brake,TL,TR} == 3'b101)  {L3,L2,L1,R1,R2,R3} = 6'b111000;
            else if ({brake,TL,TR} == 3'b110)  {L3,L2,L1,R1,R2,R3} = 6'b000111;
            else                               {L3,L2,L1,R1,R2,R3} = 6'bxxxxxx;
        s1: if (brake == 0)                    {L3,L2,L1,R1,R2,R3} = 6'b001000;
            else                               {L3,L2,L1,R1,R2,R3} = 6'b001111;
        s2: if (brake == 0)                    {L3,L2,L1,R1,R2,R3} = 6'b011000;
            else                               {L3,L2,L1,R1,R2,R3} = 6'b011111;
        s3: if (brake == 0)                    {L3,L2,L1,R1,R2,R3} = 6'b111000;
            else                               {L3,L2,L1,R1,R2,R3} = 6'b111111;
        s4: if (brake == 0)                    {L3,L2,L1,R1,R2,R3} = 6'b000100;
            else                               {L3,L2,L1,R1,R2,R3} = 6'b111100;
        s5: if (brake == 0)                    {L3,L2,L1,R1,R2,R3} = 6'b000110;
            else                               {L3,L2,L1,R1,R2,R3} = 6'b111110;
        s6: if (brake == 0)                    {L3,L2,L1,R1,R2,R3} = 6'b000111;
            else                               {L3,L2,L1,R1,R2,R3} = 6'b111111;
        s7:                                    {L3,L2,L1,R1,R2,R3} = 6'bxxxxxx;
    endcase
endmodule
```

In the listing on the previous page, the Clk and N_Clk signals have been used to form the inverter part of the crystal oscillator, shown in Figure 10.31. The output (N_Clk) acts as the clock for a 23-bit binary counter. The MSB of this counter (Q(22)) drives the state machine from one state to the next.

As the comment in the code suggests, dividing the 25MHz signal of the crystal oscillator by two, twenty-three times results in a suitably slow signal to make the output display change at a visible rate.

The way the machine changes from one state to another is listed in the third of the 'always' constructs. The code is derived directly from the state diagram of Figure 10.26. The states themselves (s0 , s1 etc) are defined in the parameter statement in the top part of the listing.

The final 'always' construct implements the output decoder logic. This is the circuitry that controls which lamps light up. It depends on the state of the machine and the logic levels on the brake , the TL and the TR signals. The table of Figure 10.29 has been used to create this code. Notice the use of curly braces {} to group signals together. This is referred to as 'concatenation'. It allows the states of the six individual lights to be defined by just one 6-bit binary number.

Using the pin assignments shown below, and with the LED board plugged into Port C and the switch board into Port D, compile and download the design to the Matrix FPGA board. Verify that pressing switch SW7 gives the turn left signal, switch SW5 gives turn right and switch SW3 acts as the brake.

Clk	38
N_Clk	66

brake	43
TR	46
TL	50
R3	54
R2	55
R1	58
L1	59
L2	60
L3	64

VHDL implementation of flashy turn indicator

```vhdl
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL, IEEE.Numeric_std.ALL;
ENTITY flashy IS
PORT
(
    Clk, TL, TR, brake: IN STD_LOGIC;
    N_Clk: INOUT STD_LOGIC;
    LIGHTS: OUT UNSIGNED (5 DOWNTO 0)
    --LIGHTS comprises L3, L2, L1, R1, R2, and R3 of Figure 4.26
);
END flashy;

ARCHITECTURE states OF flashy IS
TYPE State_type IS (s0,s1,s2,s3,s4,s5,s6,s7);
SIGNAL Current_state, Next_state: State_type;
SIGNAL Q: UNSIGNED (22 DOWNTO 0);

BEGIN
    N_Clk <= NOT Clk;
    --inverter used to create 25MHz oscillator
    PROCESS (N_Clk)
    --divide 25MHz by 2^23 for 2.98Hz
    BEGIN
        IF RISING_EDGE (N_Clk) THEN
            Q <= Q + 1;
        END IF;
    END PROCESS;

    PROCESS (Q(22))
    --use 2.98Hz clock to go from one state to next
    BEGIN
        IF RISING_EDGE (Q(22)) THEN
            Current_state <= Next_state;
        END IF;
    END PROCESS;

    PROCESS (Current_state, TL, TR)
```

```vhdl
BEGIN
    CASE Current_state IS
    --define the state machine (Figure 4.26)
    WHEN s0 => IF    TL = '1' THEN Next_state <= s1;
               ELSIF TR = '1' THEN Next_state <= s4;
               ELSE              Next_state <= s0;
    END IF;
    WHEN s1 => IF    TL = '1' THEN Next_state <= s2;
               ELSIF TR = '1' THEN Next_state <= s4;
               ELSE              Next_state <= s0;
    END IF;
    WHEN s2 => IF    TL = '1' THEN Next_state <= s3;
               ELSIF TR = '1' THEN Next_state <= s4;
               ELSE              Next_state <= s0;
    END IF;
    WHEN s3 => IF    TR = '1' THEN Next_state <= s4;
               ELSE              Next_state <= s0;
    END IF;
    WHEN s4 => IF    TR = '1' THEN Next_state <= s5;
               ELSIF TL = '1' THEN Next_state <= s1;
               ELSE              Next_state <= s0;
    END IF;
    WHEN s5 => IF    TR = '1' THEN Next_state <= s6;
               ELSIF TL = '1' THEN Next_state <= s1;
               ELSE              Next_state <= s0;
    END IF;
    WHEN s6 => IF    TL = '1' THEN Next_state <= s1;
               ELSE              Next_state <= s0;
    END IF;
    WHEN s7 =>                   Next_state <= s0;
    END CASE;
END PROCESS;

PROCESS (Current_state, brake, TL, TR)
VARIABLE Controls: UNSIGNED (2 DOWNTO 0);
BEGIN Controls:= brake & TL & TR;
CASE Current_state IS
    WHEN s0 => IF brake = '0'       THEN LIGHTS <= "000000";
               ELSIF Controls = "100" THEN LIGHTS <= "111111";
               ELSIF Controls = "101" THEN LIGHTS <= "111000";
               ELSIF Controls = "110" THEN LIGHTS <= "000111";
               ELSE                       LIGHTS <= "XXXXXX";
    END IF;
    WHEN s1 => IF brake = '0'       THEN LIGHTS <= "001000";
               ELSE                       LIGHTS <= "001111";
    END IF;
    WHEN s2 => IF brake = '0'       THEN LIGHTS <= "011000";
               ELSE                       LIGHTS <= "011111";
    END IF;
    WHEN s3 => IF brake = '0'       THEN LIGHTS <= "111000";
```

```
                    ELSE                              LIGHTS <= "111111";
          END IF;
          WHEN s4 => IF brake = '0'          THEN LIGHTS <= "000100";
                    ELSE                              LIGHTS <= "111100";
          END IF;
          WHEN s5 => IF brake = '0'          THEN LIGHTS <= "000110";
                    ELSE                              LIGHTS <= "111110";
          END IF;
          WHEN s6 => IF brake = '0'          THEN LIGHTS <= "000111";
                    ELSE                              LIGHTS <= "111111";
          END IF;
          WHEN s7 =>                                  LIGHTS <= "XXXXXX";
          END CASE;
      END PROCESS;
  END states;
```

In the listing above, the Clk and N_Clk signals have been used to form the inverter part of the crystal oscillator, shown in Figure 10.31. The output (N_Clk) acts as the clock for a 23-bit binary counter. The MSB of this counter (Q(22)) drives the state machine from one state to the next. As the comment in the code suggests, dividing the 25MHz signal of the crystal oscillator by two, twenty-three times results in a suitably slow signal to make the output display change at a visible rate.

The way the machine changes from one state to another is listed in the third of the 'PROCESS' constructs. The code is derived directly from the state diagram of Figure 10.26. The states themselves (s0 , s1 etc) are defined in the 'TYPE' statement in the top part of the listing.

The final 'PROCESS' construct implements the output decoder logic. This is the circuitry that controls which lamps light up. It depends on the state of the machine and the logic levels on the brake , the TL and the TR signals.

The code has been derived directly from table Figure 10.29. A 3-bit 'VARIABLE' has been declared within the 'PROCESS' called 'Controls' . This has then been defined as comprising the signals brake, TL and TR, joined together using the & symbol. This is referred to as 'concatenation'. It allows the states of the three individual signals to be defined by just one 3-bit binary number. The individual lights L3, L2 ... R3 cannot be combined in this way since they are outputs. The 6-bit signal LIGHTS has been used instead. Thus LIGHTS (5) is L3 , LIGHTS(4) is L2 and so on.

Using the pin assignments given below, and with the LED board plugged into Port B and the switch board into Port A, compile and download the design to the Matrix FPGA board. Verify that pressing switch SW7 gives the turn left signal, switch SW5 gives turn right and switch SW3 acts as the brake.

Clk	38
N_Clk	66
brake	43
TR	46
TL	50
R3	54
R2	55
R1	58
L1	59
L2	60
L3	64

Summary

This chapter covers a lot of ground, from basic latches and flip-flops to quite complex state machines.

Using a string of JK flip-flops, it shows how to make an asynchronous counter. When loaded into the FPGA hardware and clocked using push switches, the problem of switch bounce appears. This can be solved using two switches and a Set-Reset latch.

An asynchronous BCD counter is tackled next, using the reset input on each flip-flop to reset the whole counter when it reaches decimal 10. Circuit simulations of this work, but subtle timing problems become apparent when the same design is loaded into the E-blocks FPGA board.

Next, a different approach is taken, in which all the flip-flops were clocked by the same signal, reducing the possibility of timing problems. Synchronous design techniques using the idea of a state machine are introduced. This powerful technique uses combinational logic design processes to ensure that the next state of the machine is the one required. Binary, 0 to 5 and 0 to 9 (BCD) up and down counters are designed and implemented both in the circuit simulator and on the FPGA hardware.

The state-machine technique is extended to detect a 1-0-1 sequence in a data stream, and then to design a flashy turn indicator for cars. An oscillator for the turn indicator is built into this last design.

Some new Verilog and VHDL concepts are introduced and the power of the development system to automate much of the design process is demonstrated. The software will design the circuit to implement the required behaviour, provided the designer can describe how the system should progress from one state to the next, and how the outputs should behave, depending on the state of the machine and external inputs.

That completes the course, although there are still two assignments to tackle, to demonstrate your understanding of the concepts and techniques involved.

The author of these notes obtained much of the information about Verilog and VHDL from the book 'HDL Chip Design' by Douglas Smith, ISBN 0-9651934-3-8. Getting designs to compile is not always straightforward, and the error messages generated by Quartus are sometimes confusing. Very careful attention to syntax is needed, and VHDL is particularly strict about data types.

The general digital electronics topics covered, and more, are discussed further in the book 'Combinational and Sequential Logic' by Martin Rice, ISBN 0-582-43164-6.

Chapter 11: Assignment - Modulo-sixty counter

Introduction

With 60 seconds in a minute, and 60 minutes in an hour, there is a need for a counter that will count in 60's. This is normally made from a decade (BCD) counter and a six's counter. You have seen in detail how a 0 to 5 counter can be designed. Your task in this assignment is to design, simulate, implement and test a BCD counter and then use it, to-gether with a 0 to 5 counter, to count 0, 1, 2....58, 59, 0.
Read the section 'What you should hand in' next!

Task 1

Using the 'present-state / next-state' state-machine technique, design a 4-bit, synchro-nous BCD up counter.
First draw the required truth-table and then use Karnaugh maps to devise the four circuits needed to drive the D-inputs of the four D-type flip-flops.

Task 2

Enter your design into a suitable circuit simulation package, using devices from the 74HC logic series.
Demonstrate your design to your lecturer.

Task 3

Implement a BCD up-counter on the E-blocks FPGA board using Verilog or VHDL. Use a behavioural description of a BCD counter, similar to that given earlier in the course
Use two switches to generate a de-bounced clock signal and demonstrate your counter to your lecturer.

Task 4

Implement a modulo-60 counter on the E-blocks FPGA board by combining your BCD counter with a 0 to 5 counter.
Use the falling edge of Q3 on the BCD counter to act as the clock for the 0 to 5 counter.
Compile, download, verify your design and demonstrate to your lecturer.

Task 5

Add two output decoders to your circuit. Use one to drive the seven-segment display board, as in the assignment earlier. This should show the digit value of the 0 to 59 coun-ter. Use the other one to drive the individual LEDs on the LED board, so that just one LED lights up at a time, to indicate the tens value of the counter - no LED for counts from 0 to 9; D0 for 10 to 19; D1 for 20 to 29; ... D4 for counts 50 to 59.
Test and demonstrate your design.

Task 6

Use the 25MHz oscillator to generate an accurate 1Hz signal to drive your counter. The code should reflect the following behaviour - if the present count is 24,999,999 then the next count should be 0; otherwise the next count is one more than the present value. Test and demonstrate your design.

What you should hand in

You should hand in a written report on this assignment.

The grading criteria give more information as to what is expected.

For task 1 include:

- the truth table;
- the Karnaugh maps;
- the minimized expressions for the D inputs.
- For task 2 include:
- a printout of the schematic;
- an explanation of:
- the circuit works, with details of how next-state logic derives the next state from the current one;
- how the D-type flip-flops transfer the next state to the output at each clock pulse.
- For task 3 include:
- a Verilog or VHDL listing of your code;
- a comparison and discussion of the different approaches of tasks 1,2 and 3.

For tasks 4, 5 and 6 include:

- the Verilog or VHDL code;
- short comments in your code to explain the operation of the design.

Coverage

This assignment covers Outcome 3 of the Digital and Analogue Devices and Circuits unit (DADC) and Outcomes 2 and 3 of the Combinational and Sequential Logic unit (CSL).

Grading criteria

DADC

Pass: "Investigate digital electronic circuits" - complete tasks 1, 2 and 3 satisfactorily.

CSL

Pass: "Design and build circuits using sequential logic" and "Design and evaluate a digital system" - complete tasks 1, 2 and 3 satisfactorily.

Merit: Complete most tasks satisfactorily. Report is written clearly, using technical and non-technical language appropriately. Report is a stand-alone document, giving background to assignment as well as outlining the process undertaken. Evidence of problem solving using appropriate methods is presented.

Distinction: Complete all tasks satisfactorily. Evidence of ability to work independently, but also to ask for advice and discuss best approach when appropriate.