University of Bristol
Department of Electrical and Electronic Engineering

DP, 2016/17
Version 2.10x17

EENG28010
Digital Design, Group Project

Assignment 2:
Design of a Peak Detector

Name:.................................... ....

Group: ....

Year:..

Part 1: Introduction to Assignment

1 Introduction

In this assignment you will work in a group of four and implement a design in a Field-Programmable Gate Array (FPGA). This document has three parts; the first covers the specifications and technical details of what you should implement, the second describes how you should approach the design and what strategy you should adopt to divide the work equally amongst your group members, and the third gives a tutorial introduction to using the Xilinx ISE software to synthesise your design and download it to the FPGA.

1.1 How to Read this Document

You should scan parts 1 and 2 to get an overview of what you need to do and how you should go about organising the work in a group. You can skip the details of the protocols in Part 1 in a first read and come back to it when you need to implement that particular section. You should not start work on the design until all members of the group have read Part 2 and have agreed on the division of work, and how to manage information sharing, record details of meetings and keeping track of each individuals responsibilities and their contributions.

Note:

- Please do not start coding your designs until you have read Section 5.6 in its entirety, and fully understand it. This section highlights common mistakes made in the past.

1.2 Learning Outcomes of Assignment 2

The learning objectives of Assignment 2 are to:

- To enable the student to gain experience in digital logic design using VHDL
- To introduce the student to prototyping digital logic in FPGAs
- To familiarise the student with good practice in design management and effective collaboration in a shared group project

By the end of this lab students should be able to:

- interpret a specification to design a digital system;
- structure a design into modules and define module interfaces;
- approach the testing and simulation of a design in a systematic manner;
- use VHDL design entry to produce complex working systems;
- use ModelSim to simulate and verify their work;
- use Xilinx to prototype designs in FPGAs;
- prepare and give a group presentation;
- assess the performance of partners through peer assessment.

2 Functional Specifications

2.1 Overview

The task is to implement a system where a module called a peak detector will communicate with a data source that provides bytes upon request, and identify and store the byte with the highest value and its adjacent bytes in the sequence of retrieved data. Different groups will interpret the value of a byte based

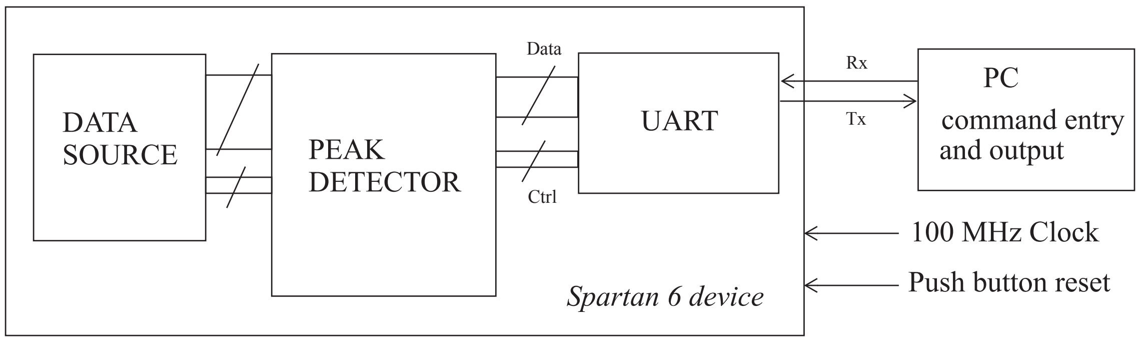


FIGURE 1. Peak detector overview

on their group no. differently, according to one of two's complement, signed magnitude or unsigned formats. The length of the sequence and the post-processing instructions to list the results will be communicated from a PC connected to the FPGA via a Universal Asynchronous Receiver and Transmitter (UART). The results will be displayed on the PC monitor and are communicated via the same UART link. The layout of the basic blocks in the design is shown in Figure 1.

The input typed into the keyboard and output received by the PC is captured and displayed in a VT100[1] terminal emulator, similar to the command terminal in Unix or DOS. The utility that emulates a VT100 terminal is called `PuTTY`. Configuration of `PuTTY` is covered in Section 3.2.

All keystrokes that are typed in the `PuTTY` terminal are captured by the *Receiver* (*Rx* for short) in the UART and communicated to the peak detector module. All outputs from the detector system are communicated to the PC terminal via the *Transmitter* (*Tx* for short) in the UART.

All of the logic in the design including the UART and data generator will be implemented in an FPGA device from Xilinx called *Spartan 6*. This device sits on a full development board which provides a 100 MHz clock as well as various I/O and other peripheral and control circuitry. The UART link is provided through a mini USB port and implements the RS232 protocol which is purely serial. The input and output serial lines are available at two I/O pins in the FPGA device. Details of the RS232 protocol are provided later in the document, and the VHDL code for both the *Rx* and *Tx* are provided.

The data generator, for which the code is also provided, has access to 500 bytes of data, and will provide a byte at a time upon request, with requests governed by a 2-phase handshaking protocol. Details will again be provided later in the document, but essentially, the data generator has an input control line and an output control line. When it sees a *transition* on the input control line (i.e. a '0' to '1' or '1' to '0' transition), it will provide a new byte on 8 parallel data lines, and change the value of its output control line; i.e. if it was '0' previously, it will transition to '1', and vice-versa. This transition on its output control line is a signal to the data acquisition module that data is valid. After the data has been latched, the consumer should effect a transition on the input to the data generator to request a new byte. Once all 500 bytes have been provided, the data generator will revert to the first byte in the sequence and keep cycling through the sequence.

The task of the peak detector is to process a number of words as typed in by the user in the terminal, and return the peak byte along with the three bytes that precede and follow it in the sequence, with the value interpreted according to one of two's complement, signed magnitude or unsigned formats. It should also return the index of the peak byte in the sequence. If multiple peaks are detected in a single data sequence, the first is to be retained. The design is to be implemented on the **Nexys 3** development

1. A VT100 terminal is a video terminal that was manufactured by DEC in the 70's and 80's, and allows character-based output.

board with a **Spartan 6** device running on a 100 MHz clock. It should use the 3 supplied modules with no modifications: a *Receiver* and *Transmitter* that implement the RS232 protocol, and a *Data Generator* that supplies a byte at a time under the control of a 2-phase handshaking protocol.

The functionality described above is described in more detail in the next few sections.

2.2 Commands

2.2.1 *Process Data Sequence*

Processing of a data sequence is initiated by the start command "`aNNN`" or "`ANNN`" typed in to the terminal, where "`NNN`" are 3 decimal digits, ranging from "`000`" to "`999`". This number is simply the number of words to be processed. For example, "`500`" means process ***500*** bytes. Hence the maximum number of words that can be processed is ***999***. Termination of a character string by hitting the `<ENTER>` key is ***not*** required. If the letter "`A`" or "`a`" is followed by 3 decimal digits in sequence, regardless of what keystrokes preceded "`A`" or "`a`", the command is valid. Examples of a valid start string are "`A500`", "`a067`" and "`hjhgjga945`". The last is valid as the last four 4 characters define a valid start command. However the string "`A56k7`" is not a valid command, as 3 decimal digits do not follow the character "`A`" in an uninterrupted sequence. For the same reason, neither is "`a55`".

Upon detection of a valid start command, the number of bytes specified should be processed without interruption. Each byte should be printed in ***hexadecimal*** format. The start command is valid at any point as long as data processing is not taking place. The behaviour when a user command is typed in the middle of processing a sequence is undefined.

2.2.2 *List Results*

Upon completion of processing of a data sequence, the following commands are valid:

- "`P`" or "`p`" to list the peak byte in hexadecimal format followed by the index of that byte from ***0*** to ***998*** in the sequence of bytes acquired from the data generator. The index should be printed as decimal digits;
- "`L`" or "`l`" to list in hexadecimal format the 3 bytes preceding the peak byte in the order received, the peak byte itself, and the 3 bytes following the peak byte in the order received, i.e. a total of 7 bytes.

These commands should not result in any action if a data sequence has not been processed prior to receiving them. The commands should always relate to the data sequence immediately preceding them, should multiple data sequences have been processed with multiple start commands.

2.2.3 *Reset*

Finally, push button `BTNU` on the board is to function as a synchronous reset. After the design has been downloaded to the FPGA, the reset button may be pressed to initialise the system. Afterwards, the system should function normally without the need to reset it, for example after every run. At any point if the reset button is pressed, the system should halt any current data processing and settle into the initial state.

Note:

- Push button "`BTNU`" is different from the pink coloured button marked "`RESET`" on the Nexys 3 board. When the `RESET` button is pressed, the FPGA will be loaded with the configuration stored in the on-board PROM. `BTNU` on the other hand can be configured to implement a user-defined function. In this case it should function as a synchronous reset that puts the state machine into its initial state.

2.2.4 *Summary*

The commands are summarised in Table 1.

TABLE 1. Peak Detector Commands

Command	**Description**
`ANNN or aNNN where NNN are 3 decimal digits.`	Start command to process NNN bytes of data. Print each byte processed in the terminal window in hexadecimal format, separated by spaces. For example bytes "11111111" and "10101001" in sequence are to be printed as "FF A9".
`L or l`	Print the 3 bytes preceding the peak in the order they were received, the peak byte itself and the 3 bytes following the peak in the order received. They should be printed in hexadecimal format and separated by spaces. For example, the output of typing this command after processing a data sequence may be "09 FA A0 FD BC 10 DE".
`P or p`	Print the peak byte itself, followed by a space and the peak index in decimal format. For example, the output of typing this command after processing a data sequence may be "FD 197".
`Push button BTNU (reset)`	Initialise the system when pressed at any point, including in the middle of a run. This is a synchronous reset and a reset command should not be required for the system to function normally, except after first loading the design.

2.3 Printing Output

Every keystroke typed into the terminal has to be echoed as it is typed.

After a valid command, the output should always start on a new line. In the VT100 terminal, a new line that starts at the first character position on the left hand side is achieved by a sequence of two bytes, a line feed "`\n`" and a carriage return "`\r`". The full ASCII code is described in Section 3.4.

2.4 Check Understanding of Specifications

Note:

- At this point, download the source archive for the Peak Detector. In the sub-folder "`fullSystem`" you will find three folders called "`unsigned`", "`twosComp`" and "`signedMag`". Descend to the folder relevant to y our group. and you will find a file named "`top.bit`", which is a bit file that can be used to configure the FPGA with the specific implementation. Configure the FPGA according to the procedure described below.

1. Make sure the board is connected to the PC via both the power up / configuration and UART USB cables, and that the power switch is on. The red LED next to the power switch should be lit.
2. On the host PC select **Start -> All Programs -> Xilinx Design Tools -> ISE Design Suite 14.2 -> ISE Design Tools -> 64-bit Tools -> iMPACT**.
3. A couple of windows will open and a dialogue box will open up asking if you want iMPACT to automatically load and save a project. Click "`Yes`".
4. Next it will ask if you want iMPACT to automatically identify and connect to the board. Say "`Yes`" again.
5. Finally, it will ask if you want to assign a configuration file. Say "`Yes`" and select the "`top.bit`" file supplied with the source bundle for Assignment 2. Click "`No`" if it asks you if you want to add a PROM file.

6. A symbol of a chip will appear in the main window. Right click this symbol and select "`Program`". Click "`Yes`" in the next dialogue box. Now the FPGA will be configured and you should see a "`Program Succeeded`" message.

Note:

- If you have any trouble with the above sequence, more details including screen-shots are given in Task 8.7 on page 33.

7. Open a `PuTTY` terminal as specified in Section 3.2.
8. Enter the different commands in the `PuTTY` terminal and check your understanding of the specifications. Make sure to type in all commands, and to check the operation for different length sequences by specifying various numbers. The data sequence it is processing is defined in the "`myPackage.vhd`" file available as part of the source file bundle supplied for this assignment.

Note:

- You do ***NOT*** need to implement the printing of an extra separation line after each command as implemented in the version provided to you. However if you wish, you can implement this formatting to improve the readability of the output.
- If you see each typed character twice for the provided implementation, it is due to the terminal settings. Check the terminal settings to make sure that typed characters are not set to be echoed.
- See how the implementation handles corner cases. If the peak index is such that there aren't enough bytes to populate 3 bytes either side of the peak, only the bytes corresponding to those valid indices are updated. For example, if the number of bytes requested is upto index 20 (i.e. 21 bytes from 0 to 20), and the peak is at index 18, an "`L`" command prints all 7 bytes, but only the 6 bytes at indices 15 (byte 0), 16 (byte 1), 17 (byte 3), 18 (byte 4 - the peak), 19 (byte 5) and 20 (byte 6) are valid, and the 7th byte, though printed, is to be disregarded as there isn't an index 21 in the retrieved sequence.
- This is simpler than printing variable numbers of bytes, and doesn't give misleading information as the number of valid bytes can be discerned from issuing the peak command. For example, in the above case, "`P`" would tell you that the peak index is 18, and hence there aren't enough bytes to populate 3 bytes after the peak.

3 Serial Communication with the Host PC

3.1 UART and RS-232 Protocol

The RS-232 serial communication protocol is a popular protocol for asynchronous transmission, when two communicating modules do not share the same clock. While it has been superseded for some applications by the USB standard, it is still widely used in others. Figure 2 illustrates the components used in the RS-232 serial connection in this assignment.

There are separate data lines for receiving and transmitting and hence the link is capable of full duplex transmission. While there is only one signal line for data transmission in each direction, the actual protocol includes some other signal lines, such as ground and control signals. These other signal lines do not need to concern us, nor issues related to the physical part of the standard, such as voltage levels, as the interface circuitry takes care of the actual signalling. What is available on the FPGA is access to

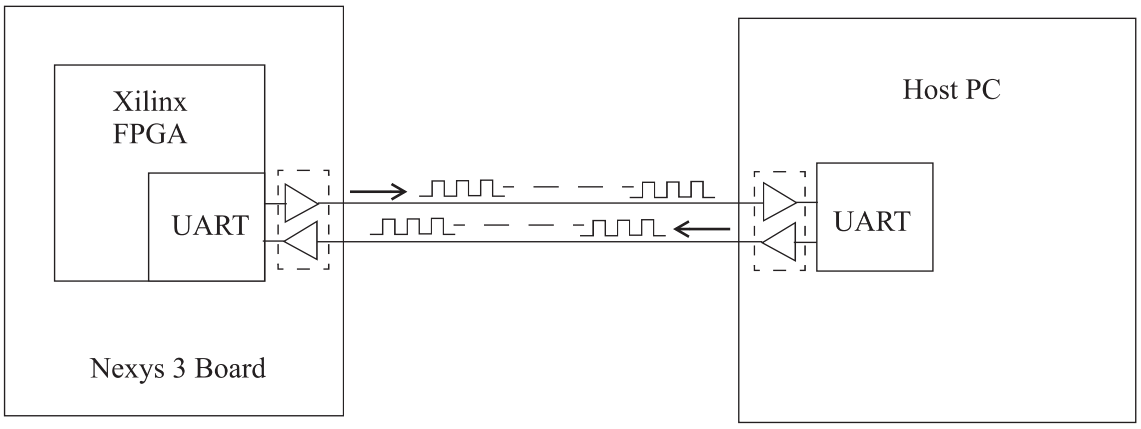


FIGURE 2. Serial communication Using R-232 protocol

two pins, one for transmitting and one for receiving. The data on these pins follows the protocol shown in Figure 3.

An RS232 frame consists of at least 10 bits, a start bit, a byte of data, and a stop bit. Additionally the standard also allows for a parity bit for error detection, but we will not use parity. For historical reasons, when the line is idle, the voltage is kept high, at logic ‘1’. To signify the start of a frame, the line is brought low. Afterwards, the line is driven high or low depending on the value of the bits in the byte. This is followed by a stop bit of logic ‘1’. The line is operated at a pre-agreed frequency, which is the baud rate of the line. This can be set to many values (for example Windows supports anything from 110 upto 921600), and we will work with a baud rate of 9600 symbols per second. As we are dealing with binary voltages, the baud rate is identical to the bit rate.

The actual serial connection between the development board and the PC is a micro USB cable. Wrappers on both sides allow us to simply concentrate on the RS-232 signalling protocol.

3.2 UART for the FPGA

You are provided with VHDL code for ***Rx*** and ***Tx*** modules that implement this protocol and are designed to run on a 100 MHz clock, to use on the FPGA side. These modules access two pins which are connected to the transmitting and receiving lines on the serial links. Their usage is demonstrated by a simple design that simply echoes whatever you type in the terminal emulation window. This whole source code bundle is in a zip archive named “`uart_rx_tx_stripped.zip`”. It also includes the pin configuration file and the Xilinx project file. You should be able to start a Xilinx project, synthesise, translate and map the design and generate a bit file to be downloaded into the FPGA as described in Section 8.

While you don’t need to understand the implementation details of the ***Tx*** and ***Rx*** modules to use them, it is strongly recommended that you study them as examples of good design practice. The ***Tx*** module

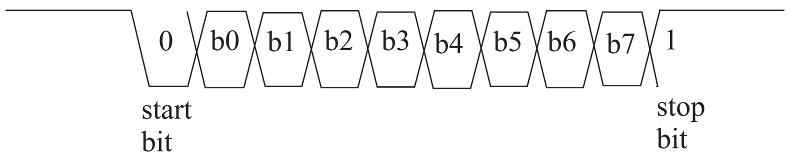


FIGURE 3. RS-232 Data Frame with no parity: Start bit of 0, followed by 8 data bits and a stop bit of 1

code has been downloaded from the Digilent (manufacturers of the Nexys 3 board) website[1]. The ***Rx*** module code was developed especially for this module, and illustrates the preferred way of describing a state machine, with a combinational logic block for determination of next state values, and a sequential state register block that updates its state on every clock cycle. Various other processes control timing and implement local controls, and manage the output signals. It also shows the sort of naming convention you should adopt, the mix of lower-case and upper-case letters for various objects to improve readability even though VHDL is case-insensitive, and the selective use of underscore ("_"). It also demonstrates modularity, with the task of a given process clearly defined. You should also note how each self-contained block of code (for example a process) is relatively short. If you find that you are writing pages upon pages of code in a single body, there may be a better (more efficient, more readable) way to implement your design.

3.3 UART on the PC

On the PC the UART hardware is standard. To open a serial link to the FPGA, you will use a terminal emulation[2] utility called `PuTTY`. This can be downloaded from http://www.putty.org/ (click on the first link). Once it is downloaded, double-click on it to open it, and you should see the window of Figure 4. Select the "`Serial`" category as highlighted and set the parameters as shown on the right. Do not click on "`Open`" yet.

Note:

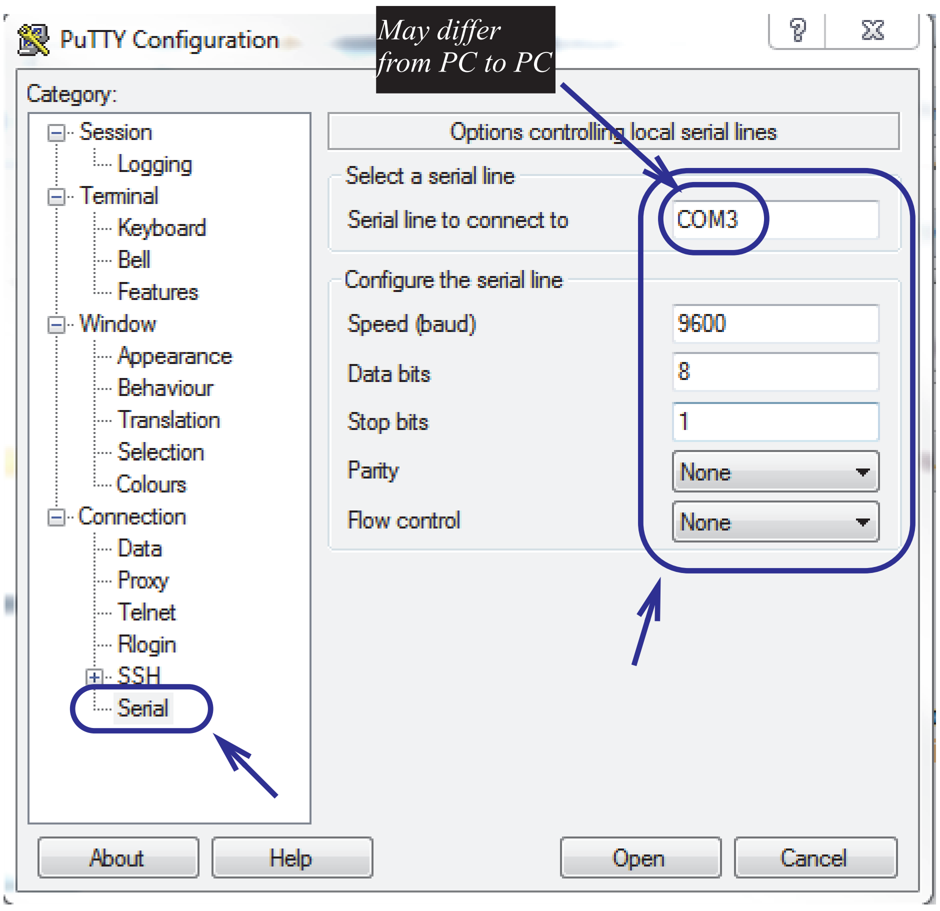
- A different COM port number may be allocated by your PC depending on what ports are in use when it detects and installs the USB driver for the serial port. When you first plug the board in, Windows 7 will auto detect new hardware and install the necessary drivers, informing you with a message in the notification area of your taskbar, at the bottom right of the screen. Click and expand this, and after installation, a green check mark will appear, telling you which port has been allocated for this link.

In order to avoid having to set the parameters each time you open the terminal, you can save these settings. Once you have edited the fields according to Figure 4, click on "`Session`" as shown in Figure 5, and give a suitable name (for example "`serial_hyp`"). Next time you open `PuTTY`, simply select the saved configuration, click on "`Load`" and your saved settings will appear. Afterwards click on "`Open`".

3.4 ASCII Printable Characters and Control Sequences for the VT100 Terminal

The VT100 terminal was a video terminal manufactured in the late 70's and early 80's by DEC that was very popular, and quickly became the de facto standard for terminal behaviour. The term is now commonly used for a terminal that *emulates* the behaviour of the original terminal. The VT100 terminal uses the 7-bit ASCII code for character representation. Clearly, 7 bits allow 128 characters to be represented. Of these the first 32 (from 0 to 31) are non-printable control codes and are used for various control actions. The remaining 96 (from 32 to 127) are the printable set available on most keyboards. Shown in Table 2 are the printable ASCII characters.

1. http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,897&Prod=NEXYS3&CFID=205834&CFTOKEN=29058160
2. A terminal emulator is a programme that emulates the functionality of the old text-based video terminals that were used with computers before the advent of modern monitors. The term VT-100 refers to one such video terminal made by Digital Equipment Corporation (DEC). It displayed ASCII characters, but also incorporated effects such as underlining, blinking etc. These are accomplished by transmitting a series of control sequences.



Options controlling local serial lines

Select a serial line

Serial line to connect to: COM3

Configure the serial line

Speed (baud): 9600

Data bits: 8

Stop bits: 1

Parity: None

Flow control: None

FIGURE 4. Configuring PuTTY for serial communication with the FPGA

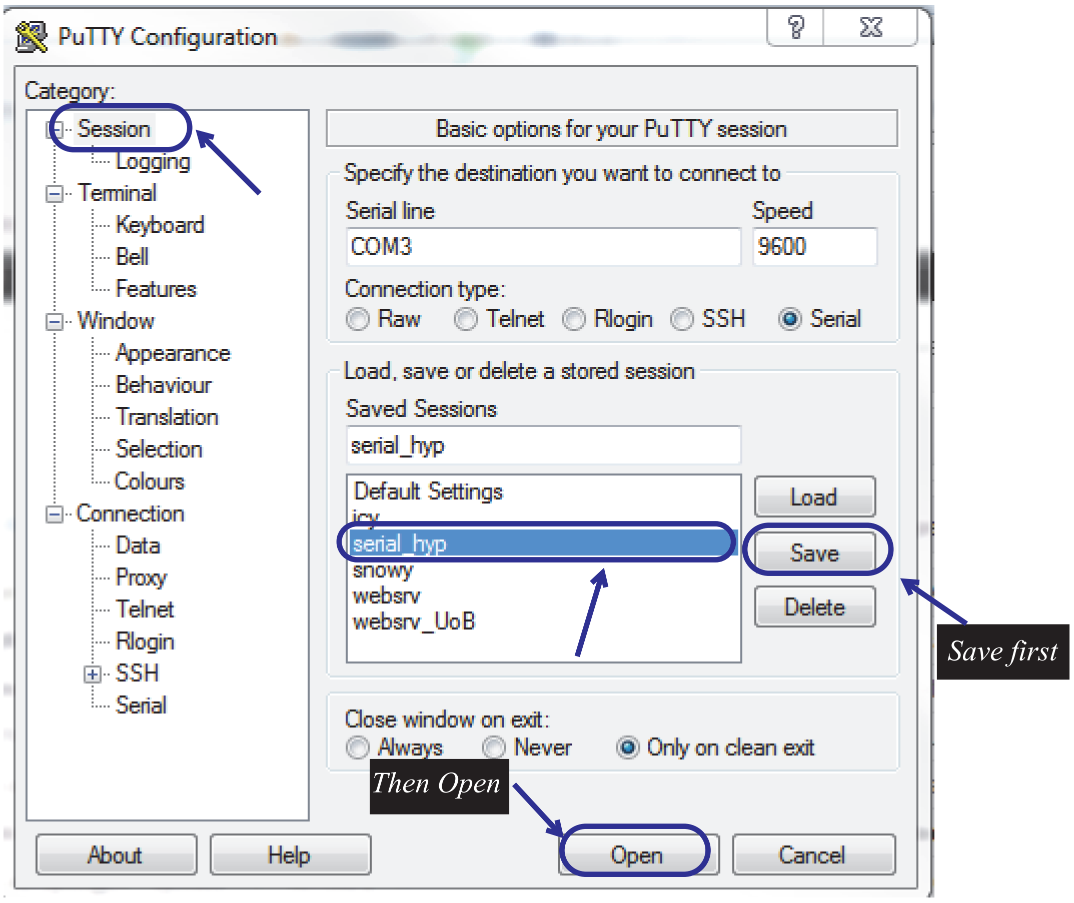
The control characters are given in Table 3. Specific sequences of these control characters can be used to control display characteristics in the VT100 terminal. Some of these sequences are used in the supplied demo project “`uart4`”, to turn underlining on, set the font colour to green, and to clear all formatting.

Note:

- Note that the binary format of the ASCII code can often be used to decode the represented characters. For example, the numeric value of the decimal digits 0 through 9 can be extracted simply by taking the binary value of the 4 least significant digits.

4 Two-Phase Signalling Protocol

Asynchronous communication between two modules that do not share a common clock on the same chip is effected by what is known as a *handshaking* protocol, which is a cycle of request and acknowledge between the two modules on dedicated control lines. Both four-phase and two-phase protocols are widely used. Four phase protocols are commonly return-to-zero, i.e. a control line conveys information by being taken high (logic ‘1’), and then returning to the low value (logic ‘0’). The receiving module therefore needs to monitor for the logic value on the control line. By contrast, two-phase protocols are non-return-to-zero, i.e. any transition, whether from low to high or high to low, conveys some information.



Serial line: COM3
Speed: 9600
Connection type: Raw Telnet Rlogin SSH [✓ Serial]
Saved Sessions: serial_hyp
Close window on exit: Always Never [✓ Only on clean exit]

FIGURE 5. Saving PuTTY settings for future sessions

We will use a two-phase handshaking protocol for this assignment for communication between the Data Generator and the Data Processor, which is implemented by two control signals “`ctrl_1`” and “`ctrl_2`” as shown in Figure 6. Rather than absolute values on the control lines, information is transferred by *transitions*. As shown, the data acquisition module makes a request by effecting a transition on its output control line `ctrl_1`. The data generation module monitors activity on this line, and fetches a new byte, makes it available on the data lines, and effects a transition on its output control line `ctrl_2`. This transition is the signal for the data acquisition module that the data available on the data lines are valid. When it needs a new byte of data, it makes another transition on its output control line, this time from ‘0’ to ‘1’. When the data generator is ready with the new word, it fetches the new byte. Until a new request is made, the old data has to be maintained on the data lines.

The data source supplied to you in the source code bundle for Assignment 2 will supply bytes without limit upon request. After 500 bytes the pattern will repeat.

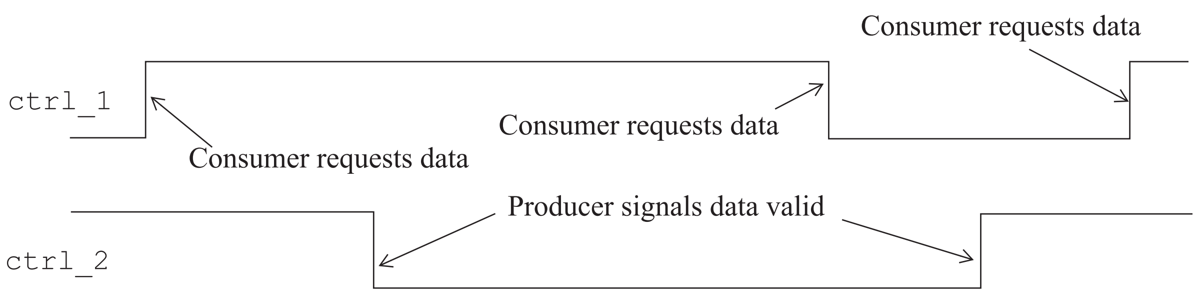


FIGURE 6. Two-phase handshaking protocol. The ctrl_1 line is driven by the data acquisition module, and the ctrl_2 line by the data generation module.

TABLE 2. Printable ASCII characters

DEC	HEX	BIN	Symbol	DEC	HEX	BIN	Symbol
32	20	00100000	(space)	80	50	01010000	P
33	21	00100001	!	81	51	01010001	Q
34	22	00100010	"	82	52	01010010	R
35	23	00100011	#	83	53	01010011	S
36	24	00100100	$	84	54	01010100	T
37	25	00100101	%	85	55	01010101	U
38	26	00100110	&	86	56	01010110	V
39	27	00100111	'	87	57	01010111	W
40	28	00101000	(	88	58	01011000	X
41	29	00101001	)	89	59	01011001	Y
42	2A	00101010	*	90	5A	01011010	Z
43	2B	00101011	+	91	5B	01011011	[
44	2C	00101100	,	92	5C	01011100	\
45	2D	00101101	-	93	5D	01011101	]
46	2E	00101110	.	94	5E	01011110	^
47	2F	00101111	/	95	5F	01011111	_
48	30	00110000	0	96	60	01100000	`
49	31	00110001	1	97	61	01100001	a
50	32	00110010	2	98	62	01100010	b
51	33	00110011	3	99	63	01100011	c
52	34	00110100	4	100	64	01100100	d
53	35	00110101	5	101	65	01100101	e
54	36	00110110	6	102	66	01100110	f
55	37	00110111	7	103	67	01100111	g
56	38	00111000	8	104	68	01101000	h
57	39	00111001	9	105	69	01101001	i
58	3A	00111010	:	106	6A	01101010	j
59	3B	00111011	;	107	6B	01101011	k
60	3C	00111100	<	108	6C	01101100	l
61	3D	00111101	=	109	6D	01101101	m
62	3E	00111110	>	110	6E	01101110	n
63	3F	00111111	?	111	6F	01101111	o
64	40	01000000	@	112	70	01110000	p
65	41	01000001	A	113	71	01110001	q
66	42	01000010	B	114	72	01110010	r
67	43	01000011	C	115	73	01110011	s
68	44	01000100	D	116	74	01110100	t
69	45	01000101	E	117	75	01110101	u
70	46	01000110	F	118	76	01110110	v
71	47	01000111	G	119	77	01110111	w
72	48	01001000	H	120	78	01111000	x
73	49	01001001	I	121	79	01111001	y
74	4A	01001010	J	122	7A	01111010	z
75	4B	01001011	K	123	7B	01111011	{
76	4C	01001100	L	124	7C	01111100	|
77	4D	01001101	M	125	7D	01111101	}
78	4E	01001110	N	126	7E	01111110	~
79	4F	01001111	O	127	7F	01111111	

TABLE 3. ASCII control characters that cannot be printed

DEC	HEX	BIN	Symbol	Description	DEC	HEX	BIN	Symbol	Description
0	00	00000000	NUL	Null char	16	10	00010000	DLE	Data Line Escape
1	01	00000001	SOH	Start of Heading	17	11	00010001	DC1	Device Control 1 (oft. XON)
2	02	00000010	STX	Start of Text	18	12	00010010	DC2	Device Control 2
3	03	00000011	ETX	End of Text	19	13	00010011	DC3	Device Control 3 (oft. XOFF)
4	04	00000100	EOT	End of Transmission	20	14	00010100	DC4	Device Control 4
5	05	00000101	ENQ	Enquiry	21	15	00010101	NAK	Negative Acknowl-edge
6	06	00000110	ACK	Acknowledgment	22	16	00010110	SYN	Synchronous Idle
7	07	00000111	BEL	Bell	23	17	00010111	ETB	End of Transmit Block
8	08	00001000	BS	Back Space	24	18	00011000	CAN	Cancel
9	09	00001001	HT	Horizontal Tab	25	19	00011001	EM	End of Medium
10	0A	00001010	LF	Line Feed	26	1A	00011010	SUB	Substitute
11	0B	00001011	VT	Vertical Tab	27	1B	00011011	ESC	Escape
12	0C	00001100	FF	Form Feed	28	1C	00011100	FS	File Separator
13	0D	00001101	CR	Carriage Return	29	1D	00011101	GS	Group Separator
14	0E	00001110	SO	Shift Out / X-On	30	1E	00011110	RS	Record Separator
15	0F	00001111	SI	Shift In / X-Off	31	1F	00011111	US	Unit Separator

Part 2: Methodology - How to Approach the Design

5 Design Organisation

The global architecture of the peak detector that you should adopt is shown in Figure 7. The ***Rx***, ***Tx*** and ***Data Generator*** modules are provided, and shown shaded in grey. Brief explanations of the interfaces and signal functions are given in the following sections. The detailed operation of the modules can be understood by examining the code.

5.1 UART Transmitter (*Tx*) Module

The functions of the signals in the ***Tx*** interface are described in Table 4.

TABLE 4. UART Transmitter (***Tx***) Interface

Signal	**Description**
`txNow`	Used to trigger a send operation. The ***Command Processor*** should set this signal high for a single clock cycle to trigger a send. When this signal is set high “`data`” must be valid. Should not be asserted unless “`txDone`” is high.
`data`	The parallel data to be sent. Must be valid in the clock cycle that “`send`” has gone high.
`CLK`	System clock. A 100 MHz clock is expected.
`txDone`	This signal goes low once a send operation has begun and remains low until it has completed and the module is ready to send another byte.
`txD`	Serial data link. This signal is connected to the external output pin, that in turn is connected to the serial data input of the receiver on the PC side.

5.2 UART Receiver (*Rx*) Module

The functions of the signals in the ***Rx*** interface are described in Table 5..

TABLE 5. UART Receiver (***Rx***) Interface

Signal	**Description**
`rxD`	Serial input data line. This will be connected to the external input pin that in turn is connected to the serial data output of the transmitter on the PC side.
`clk`	System clock. A 100 MHz clock is expected.
`reset`	Synchronous reset.
`done`	Used to signal to the receiver that data on the bus has been successfully read, and the register can be cleared. Should be set high by the ***Command Processor*** for 1 clock cycle. This signal is read by the ***Rx*** module to reset the data ready signal and also to check for overrun errors.
`data`	The received data is made available on this 8-bit wide bus. Data for current word is valid when “`valid`” is high.
`valid`	Goes high when all of the bits in a serial transmission of a word have been detected. It goes low when “`done`” is set high by the upper layer logic, to be ready to receive the next word and indicate completion of the current word.
`oe`	Goes high when an overrun error is detected; i.e. a new word is received, but upper layer logic has not signalled that previous byte has been read.
`fe`	The start and stop bits are not detected in order according to the RS232 protocol; i.e. a framing error has occurred.

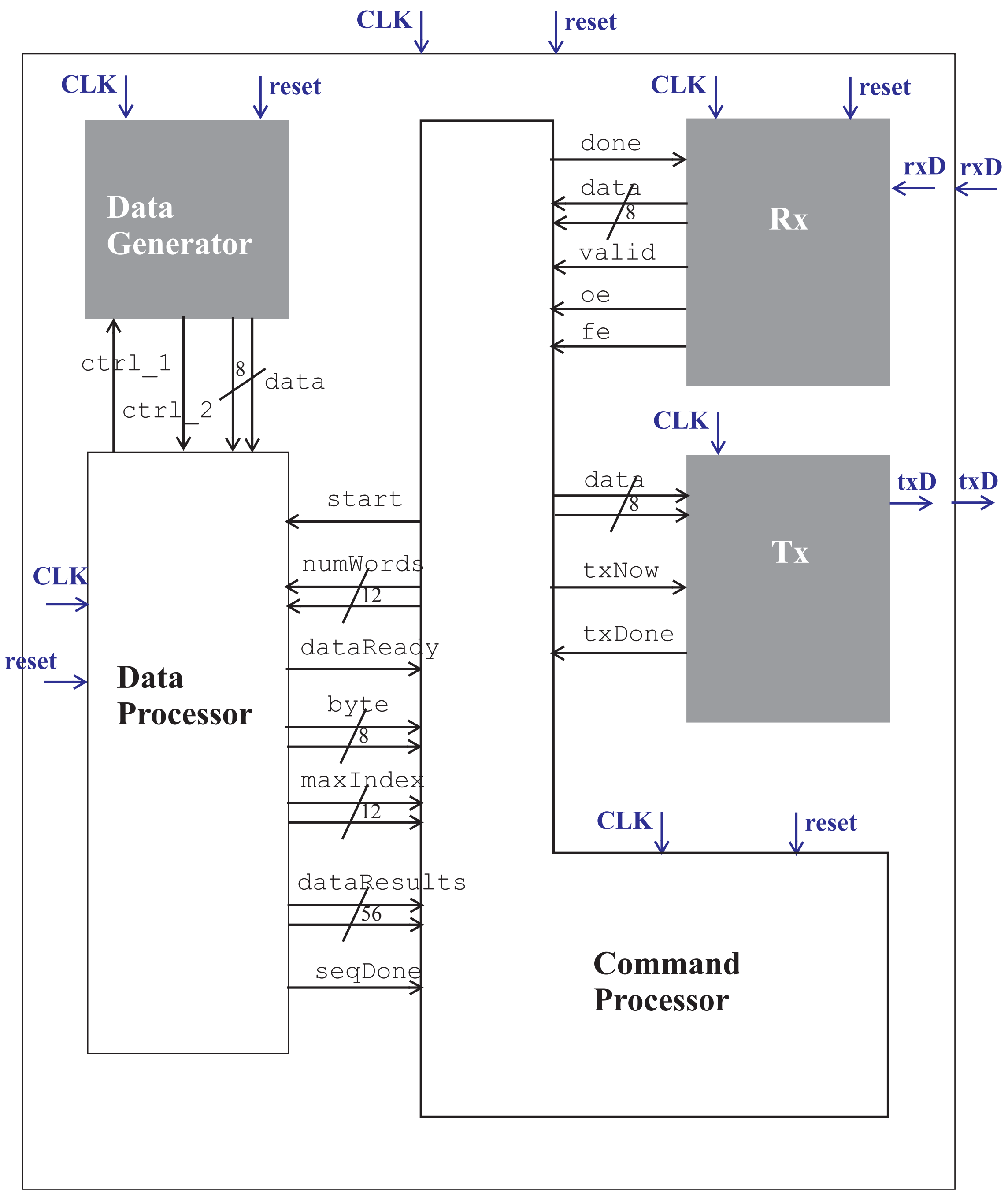


FIGURE 7. Block-level architecture of the peak detector module. The block shapes are not representative of the floor plan or layout.

5.3 Command Processor and Data Processor

One important consequence of the UART serial link protocol and the baud rate is that the serial signalling rate is much less than the clock frequency. The clock frequency of 100 MHz is over four orders of magnitude higher than the 9600 baud rate over the serial link. As a consequence, data retrieval needs to be halted periodically while the slow serial communication completes. This is effected by the "`start`" signal. Given in Table 6 are descriptions of this and other lines shared between the ***Data Processor*** and ***Command Processor***.

TABLE 6. Signals shared between the Command Processor and Data Processor

Signal	Description
`start`	Used to start and halt data retrieval by the ***Data Processor***, while the ***Command Processor*** communicates with the PC via the serial link, which operates at a much lower frequency than the clock rate. Goes high for one clock cycle to initiate data retrieval and then goes low. The value on the "`start`" signal should be checked at the beginning of every data retrieval cycle.
`numWords`	A 12 bit wide set of data lines that contain the number of bytes to process, in binary-coded-decimal (BCD) format. Each decimal digit is encoded in 4 bits.
`dataReady`	Asserted (active-high) by the ***Data Processor*** to signify that a new byte of data that has been supplied by the ***Data Generator*** is ready on the 8-bit "`byte`" line.
`byte`	Contains the latest 8-bits wide data word retrieved from the ***Data Generator***. Data is valid when "`dataReady`" is high.
`seqDone`	Asserted (active high) for one clock cycle when the number of bytes specified in "`numWords`" has been processed, and the 7 bytes comprising the peak and the 3 bytes either side of it are contained in "`dataResults`", which is 56 bits wide.
`dataResults`	A 56-wide set of lines that contain the 7 bytes comprising the peak in the middle (i.e. bit indices 31 down to 23), and the 3 bytes either side of it. Data is valid when "`seqDone`" is high.
`maxIndex`	Contains the index of the peak byte in BCD format.

5.4 Data Processor and Data Generator

The control lines "`ctrl_1`" and "`ctrl_2`" are shared between the ***Data Processor*** and ***Data Generator***, and are operated in accordance with the two-phase protocol described in Section 4.

5.5 Provided Code Artefacts

You are provided with four sets of code bundles in a single zipped archive. These are described below.

5.5.1 Demonstrator for UART

This code bundle is available in the sub-folder "`uart_rx_stripped`" and demonstrates the use of the UART. It comprises four VHDL sources files, a Xilinx project file and a Xilinx user constraints file:

- "`UART_RX_CTRL.vhd`": The source for the UART receiver.
- "`UART_TX_CTRL.vhd`": The source for the UART transmitter.
- "`control_unit_test.vhd`": The source for a simple controller that echoes text typed in the terminal window.
- "`test_top.vhd`": The source for the top component that instantiates and binds all of the three above components.

- “`uart_rx_stripped.xise`”: The Xilinx ISE project file for implementing the demonstrator.
- “`test.ucf`”: The user constraints file (UCF) that defines the pin configuration on the FPGA and is referenced by the project file.

The use of this archive is described in Section 8.

5.5.2 Data Processor

This code archive is available in the sub-folder “`dataProcessor`”. It contains three testbenches for the unsigned, signed magnitude and two’s complement implementations of the data processor that you can use to test the data processor component as a stand-alone unit. To check the data format your group should use, check Section 7.1.

The archive also contains the source for a package called “`common_pack.vhd`”. VHDL allows *packages* to be defined, which are commonly used as repositories of constants, types, functions, procedures and other objects that are referred to and used by multiple components. This package defines some global constants and data types as well as the data sequence used by the data generator. The source for the Data Generator is also provided.

To use this testbench, create a ModelSim project and add the testbench that is relevant for your group, and the Package and Data Generator source files. Then create a separate file named “`dataConsume.vhd`” with the same entity description for the Data Generator as used in the testbench that contains your implementation, and add it to the project.

5.5.3 Command Processor

This code archive is available in the sub-folder “`cmdProcessor`”. It contains a testbench that can be used to test the ***Command Processor*** as a stand-alone unit. The archive also contains the source code for the “`common_pack`” package. To use this testbench, create a ModelSim project and add the testbench and the Package source files. Then create a separate file named “`cmdProc.vhd`” with the same entity description for the ***Command Processor*** as used in the testbench that contains your implementation, and add it to the project.

5.5.4 Full System for Simulation

This code bundle is available in the sub-folder “`fullSystem/XYZ`” where the string `XYZ` is either `unsigned`, `signedMag`, or `twosComp`. You should choose the sub-folder relevant for your Group. These implementations use some Xilinx libraries. You should only look at this archive after you are familiar with the rest.

It includes synthesised versions of the ***Command Processor*** and the ***Data Processor*** as structural implementations. The testbench included here shows sequences of lengths 2 and 13 bytes being processed, with the `list` and `peak` commands being issued at the end.

The full archive comprises the following VHDL sources files:

- “`UART_RX_CTRL.vhd`”: The source for the UART receiver.
- “`UART_TX_CTRL.vhd`”: The source for the UART transmitter.
- “`common_pack.vhd`”: This package defines some global constants and data types as well as the data sequence used by the data generator.
- “`dataGen.vhd`”: The source for the Data Generator. You should scrutinise this code to understand how the single-phase protocol is implemented.

- “`cmdProc_synthesised.vhd`”: The Xilinx synthesised version of the Command Processor described as a pure structural implementation. This implementation uses some Xilinx libraries.
- “`cmdProc_wrapper.vhd`”: The synthesised version of the Command Processor has a slightly different entity description, with the custom data types `BCD_ARRAY_TYPE` and `CHAR_ARRAY_TYPE` being replaced with 2-D arrays of the `STD_LOGIC` type. This wrapper defines some conversion functions and instantiates the synthesised version as a component.
- “`dataConsume_synthesised_XYZ.vhd`”: The Xilinx synthesised versions of the Data Processor described as a pure structural implementation for unsigned, signed magnitude or two’s complement interpretation of the byte.
- “`dataConsume_wrapper.vhd`”: The synthesised version of the Data Processor has a slightly different entity description, with the custom data types `BCD_ARRAY_TYPE` and `CHAR_ARRAY_TYPE` being replaced with 2-D arrays of the `STD_LOGIC` type. This wrapper defines some conversion functions and instantiates the synthesised version as a component.
- “`tb_dataGenConsume.vhd`”: The testbench that instantiates all the components, and issues two start commands with 2 and 13 bytes respectively, followed by print list and print peak commands by driving the input line of the UART Receiver.
- “`top.bit`”: The bit file that can be directly downloaded to the FPGA to implement the full Peak Detector that you can use to check the specifications.

The testbench instantiates all the components and issues two start commands with 2 and 13 bytes respectively, followed by print list and print peak commands by driving the input line of the UART Receiver. Run the simulation for at least 160 ms to see the full simulation for these commands.

In order to replace the ***Command Processor*** (***Data Processor***) module in this testbench with your own you should delete the “`cmdProc_wrapper.vhd`” and “`cmdProc_synthesised.vhd`” (“`dataConsume_synthesised.vhd`” and “`dataConsume_wrapper.vhd`”) files from the ModelSim project, and add the file which you should name “`cmdProc.vhd`” (“`dataConsume.vhd`”) that contains the new source.

5.5.5 Synthesis and Implementation using Xilinx

In addition to the zip archive that includes all the artefacts for the peak detector design you can also find another zip archive named “`Netlist files for Xilinx.zip`”, that has two netlist files (post-synthesis `.ngc` files) for each of the data formats corresponding to the `unsigned`, `signedMag`, and `twosComp` specifications for both components. You can use the `.ngc` file of either the ***Command Processor*** “`cmdProc.ngc`” or the Data Processor “`dataConsume.ngc`” in conjunction with your developed code and all other necessary files (i.e. UART, etc.) to synthesize, place & route and finally generate the bit file (in `.bit` format) of your peak detector system in Xilinx. For example, a sub-group working on the Command Processor can use the “`dataConsume.ngc`” file along with the rest of the developed and supplied code to synthesise and test the whole system and vice versa.

5.6 Coding Style for Designing Hardware

For every design deliverable it is very important that you follow the design and coding guidelines described in Assignment 1. In particular, you have to:

1. Separate combinational and sequential elements[1] clearly, and follow prescribed templates for both;

1. Combinational logic refers to logic blocks where the output is at all times driver by the inputs. Sequential logic refers to logic blocks that hold the value of the output when not directly driven by the input, and comprise flip-flops. See Assignment 1, Section 5 for a detailed explanation.

- For combinational elements this means adding every signal that you read in a process to the sensitivity list, and assigning values in every single conditional branch of `if-then-else` and `case` structures. One way to accomplish the latter is to assign a default value at the beginning of a conditional branch structure and overwrite it only where necessary.
- For sequential elements this means using the provided template to infer a flip-flop, based on signal assignment under the control of a rising or falling clock edge. You should only use one type of flip-flop, either rising or falling and not both, throughout the design.
- For example, say you need an up-down counter with reset, enable and counting direction inputs. Clearly, this is a sequential block, as the count value needs to be saved. Define a process in the same architecture body, or a separate entity with its own architecture. An entity makes sense if you want to reuse it as a component across multiple designs. If you only want to use it once, a process makes sense and makes for less clutter. Whether or not you use a separate entity and architecture body, give the process a clear descriptive label, and make it synchronous, i.e. only clock need be in the sensitivity list. On every clock edge, either reset the count to zero, or if counting is enabled, increment or decrement count based on the counting direction. Do not be tempted to do anything other than implement the counter in this process.
- As another example, say you need a 3-way multiplexer. Again implement this as a process or in a separate architecture with its own entity depending on how you plan to reuse it. This is clearly a combinational element, and all data and select inputs need to be in the sensitivity list. If you have a behavioural if-then-else structure or a case statement, make sure to assign the output in every single conditional branch. Again, do NOT be tempted to do anything other than multiplex in this block.

2. Use the standard architecture comprising a data path or data paths and a controller or controllers as adopted in Section 6.4 of Assignment 1. The controller should be described as a finite-state machine and follow the supplied template exactly.
3. Avoid latches at all costs. Latches are inferred by assigning a value in a single conditional branch under the control of a level-sensitive condition. They can be unintentionally inferred by forgetting to assign values in all branches of a combinational process.
4. Avoid software-oriented styles of coding predicated on a single sequential thread. Remember that hardware is inherently concurrent.
5. Avoid long monolithic blocks of code that run into pages and pages. Organise your code into modular (self-contained) blocks, either as processes inside the same architecture, or separate entity-architecture bodies, where each block accomplishes one clearly defined task. Each modular block should be quite short, with a length on the order of what you can read on a single screen. If you need much longer code than can be read on a single screen, you should consider splitting the functionality into two or more blocks.
6. Synthesise your code often in Xilinx as described in Part 2 of this document, and pay heed to ***errors*** that make your code unsynthesisable and ***warnings*** related to latches and missing signals in sensitivity lists.
7. If you need to use arithmetic operators, use the `IEEE numeric_std` package and not the `std_logic_arith` package. The PDF available here: http://www.gstitt.ece.ufl.edu/vhdl/refs/vhdl_math_tricks_mapld_2003.pdf (tutorial by Jim Lewis) gives a very good summary of data types and conversion functions for arithmetic operators.
8. Try **not** to use any of the constructs **not** introduced in Assignment 1. In particular, be careful about using the `for` <parameter> `in` <range> `loop` construct. This is synthesisable for a static range, and if the loop contains no `wait` statements. However unless you understand what you are doing and know how it will be synthesised, please avoid. There is no penalty if you do use it properly.

Note:

- The example in Section 6.4 of Assignment 1 best describes how hardware should be implemented. This is the style you should follow closely. Also carefully read the supplied code for the UART receiver (***Rx***), which illustrates all the design principles and guidelines described here.
- Avoid at all costs artificial timing based on omitting signals from the sensitivity list as discussed in the example of Section 4.2 of Assignment 1. While this may work in a simulation, it will never work in actual hardware. Any synthesis tool will automatically add all signals that are on the right-hand side of a signal assignment or are read in `if-then-else` or `case` structures to the sensitivity list.

5.6.1 Common Mistakes

Implementing memory without following a flip-flop template. The most frequent causes of latches in submitted designs are when a latch is inferred by assigning to some signal in certain branches only, inside a state machine. In some cases this may be unintentional, and can easily be fixed by assigning in every conditional branch. These are cases where memory is not required to implement the functionality.

However in certain other cases, memory may need to be inferred for the given functionality. For example, consider the following code fragment:

```
...........
 CASE curentState IS
    ...........
    WHEN updateState =>
      tx_reg <= rx;
    WHEN inputReady =>
      -- tx_reg is not assigned in any other branch
      -- instead, its read here
      if (tx_reg = “11111111”)
        -- do something
.............
```

Let's assume that for this design to work, the value of `tx_reg` should only be updated in state `updateState`, and the value needs to be read in other states, i.e. memory is needed. Written in this form, a latch is implied, as the assignment is under the control of a level sensitive condition, and it is not assigned in every branch. A latch being implied is bad enough, but this is implication by stealth, it does not follow a latch template. This code is unlikely to work even in simulation. It is guaranteed not to work when synthesised.

Note:

- The above is a perfect example of how ***not*** to write VHDL code to be synthesised, and is typically the result of applying a software programming style and logic to hardware.

So how should you implement this instead? You should have a separate clocked process that assigns to `tx_reg` by reading the value of `currentState`. This is accomplished by the following code:

```
regSync : PROCESS (clk)
BEGIN
  IF clk'EVENT AND clk='1' THEN
     IF reset='1' THEN
        tx_reg <= X"FF"; -- assign a HEX value to std_logic_vector
     ELSIF currentState = updateState THEN
        tx_reg <= rx;
     END IF;
  END IF;
END PROCESS; -- regSync
```

Note how this has a synchronous reset, and everything that happens inside the process body is predicated upon a clock edge. Therefore, the process only needs to be triggered on a clock event, and does not need anything other than the clock to be in the sensitivity list.

6 Group Work

One of the three learning objectives of this assignment is related to group work, and learning how to effectively work within a group. You are expected to complete this assignment by working in a group of four. Section 6.1 suggests methods of distributing the work such that the work load for each student is more or less the same. You are expected to work as a group, report your interim progress as a group, and present your results at the end of the course as a group. Every group member has an opportunity to rate the contribution of the other group members using a form of peer assessment. The view your partners have of your work and contribution will count to your final mark.

6.1 Dividing Work Within a Group

6.1.1 Division of Work Among Group Members

The tasks in this design can be logically separated into UART related activity which communicates with the host PC (***Command Processor***), and transactions related to data acquisition and data processing (***Data Processor***). **Each 4-person group should split into two 2-person teams to independently develop each of these components**. You should stick carefully to the architecture of Figure 7, as this block-level organisation is a logical way of organising the design, and also provides a means of distributing the work equitably within the group.

Working within the framework of the interfaces described in Figure 7 allows different members of the team to develop various parts of the design independently. Once the interface is clearly defined and understood by everybody, the implementation details of each module are not necessary in order to design other modules which interface to them. All that is required is that a given module adheres to the agreed interface protocol. In order to test either the ***Command Processor*** or the ***Data Processor*** module independently of the other, black-box implementations are provided that can be instantiated within a software testbench, and also synthesised.

Once the two modules have the desired functionality, it should be a simple task to integrate them to produce the final working design.

The ***Command Processor*** is somewhat more complex to develop than the ***Data Processor***. Hence full functional compliance is required for the ***Data Processor*** for the interim submission, while only partial

Note:

- The test bench available in the source bundle for Assignment 2 will be used to check your code, and therefore you should follow the same port interface definitions.
- As the number of students in the class is not always divisible by 4, there may be one or two groups with three students each. These groups should contact one of the Lecturers to ask for further guidelines *if* they feel that they need it.

compliance with specifications is required for the ***Command Processor***. After the interim submission, the ***Data Processor*** team should take the lead in pursuing full integration of the two components and debugging, while the ***Command Processor*** team should complete the functional development of the ***Command Processor***. The details of dividing the work amongst team members after the interim submission is left to each group. See Section 7.2 for details of what is required for the interim submission.

6.1.2 Division of Work Among Team Members

It is very important that every team member takes an equal share of the development work. **This means that you should contribute equally to designing, developing, testing and debugging modules**, as well as to report writing and preparation of presentations. Please note that only one report is required in the entire project, as part of the interim submission, and one joint presentation at the end, and the majority of the deliverables are design artefacts.

Note:

- Successful team work will result from division of work based on required functionalities where one person is responsible for implementation of modules / parts of modules from simulation to synthesis.

Things to Avoid in Division of Work

The following strategies all lead to lopsided contributions and should be strictly avoided.

- Dividing the work based on simulation or synthesis e.g. one person takes care of simulation, the other takes care of synthesis. This makes no sense.
- Allocating one person to compose just the report and/or presentation.
- Allocating one person to do just debugging. While development and testing is often separated in industry, in this project you have been supplied with comprehensive test benches. You are very welcome to modify and enhance the supplied test benches, which should lead to increased understanding and appreciation of the intricacies of VHDL, but should not be at the expense of development.
- Not having clearly defined roles for each team member.
- Having one person loosely associated with both teams, for example with the role of troubleshooting issues in both modules. This is often an excuse for covering up the non-performance of one or more team members.
- Changing roles and responsibilities of a group member without due consultation within the group.

Note:

- The above are all examples from unsuccessful past attempts. Please ensure that every team-member has a development role and takes responsibility for a substantial portion of the design.

6.2 Working as a Group

The first scheduled session after Assignment 1 is complete should be the date of your first group meeting. At this meeting you should discuss the following:

1. Division of work within the group, i.e. who does what (see Section 6.1.1).
2. Means of communication within the group. Blackboard has a group space, where you can upload files, exchange messages etc., but you may prefer to use another platform. This is fine as long as everybody within the group is comfortable with the arrangement. However please make sure that all important communications are sent to and from your official University e-mail accounts.
3. Project plan and time line for meetings, internal and external deadlines, and milestones. The basic requirement in Assignment 2 is to develop the ***Command Processor*** and ***Data Processor*** separately based on the prescribed interface specifications, and then integrate the two. A clear milestone in this process is the interim submission, where each component has to meet a set of specifications. To meet this milestone, teams of two within the group have to work closely together, but there also has to be communication between the teams to ensure that the interface specifications have been clearly understood by both teams.

   It is highly recommended that a simple project plan be drawn up, with tasks, milestones, and meeting dates. Milestones or deliverables should be associated with each meeting date to monitor progress, even if the deliverable is only a progress update. This project plan should be part of the report submitted as part of the interim assignment.

After the division of work has been discussed and initial work plan drawn up, regular meetings should be held according to the agreed schedule. Please note that all scheduled sessions are mandatory, and should in the first instance serve as meeting dates. If your group has members that are from different cohorts, and it appears difficult to set up joint meetings, please contact a Lecturer for advice.

6.3 Contributing to the Group Effort

There are two essential aspects to each individual's contributions within a group, as described below.

6.3.1 Fulfil One's Own Responsibilities Within Group

Effective communication

You have to communicate effectively with your group members, including participating in group discussions, turning up at scheduled sessions, and responding to e-mails and other communications from group members in a timely manner. If you have any issue with an internal deadline or have trouble completing a task, it is your responsibility to let your group members know, and come to some alternate arrangement.

Carry out technical tasks

You have to complete the technical tasks that have been assigned to you within prescribed time frames including developing code for modules that are your responsibility, developing code for shared endeavours including test benches, and testing, analysing and trouble-shooting modules and code from other group members to facilitate integration.

6.3.2 Help Others Within Group Fulfil Their Responsibilities

All group members may not have the same level of proficiency in digital design and VHDL-based design entry, and thus different individuals may not work at the same speed at the beginning. If another group member is not as quick as you, the solution is not to take over all the work, but to give your teammate the opportunity to learn and catch up, and make a useful contribution. This is important for both you and your team-mate: for your team-mate, as they should learn the very important skills that are developed in this project, and for you, so that somebody else does not take credit for what you have done.

6.3.3 Dealing with Difficult Team / Group Mates

Efficient and professional collaboration within a group with diverse skill sets (eg: software or hardware backgrounds) is an essential skill for any aspiring Engineer. In the first instance it is your responsibility to treat every team mate with respect, come to an effective working arrangement and fulfil your own responsibilities. Team discussions, structural divisions of the design and other operational details will not be micro-managed, and are left to the initiative of each group.

However if any team or group find that a member or members do not fulfil their responsibilities and/or do not turn up at meetings and otherwise participate, or violate the covenant of respect, consideration and professionalism that affect their own ability to work effectively, the following procedure should be adopted:

1. Politely inform your team mate via his or her University e-mail address that in your opinion they are not contributing fairly or / and affecting your ability to contribute and suggest a meeting to discuss the issue. In the meeting address your concerns in a professional way, and try to come to a resolution.
2. If the above does not result in a satisfactory resolution, inform one of the course Lecturers as soon as possible. It is important that you let a Lecturer know of any issues early, as a fair and transparent process needs to be followed to ensure the views of all parties are heard. Quite often this involves a meeting with a Lecturer present, and prescribed timelines for subsequent work, with communications being copied to the Lecturer.

Note:

- As much as possible you should try to resolve issues internally, but it is very important that you bring any serious concerns to the notice of a Lecturer in a timely manner. For example, complaining about an under-performing team mate in the last week before the deadline will not leave time for any sort of satisfactory resolution.

7 Deliverables

7.1 Data Format According to Group

Each group has the same set of deliverables, but the way groups interpret data is different. Your group number will determine whether you treat the bytes as unsigned, signed magnitude or two's complement numbers. The remainder, after dividing your group number by 3, will dictate this choice as shown in Table 7. You can check which group you belong to by checking the group assignment Table on Blackboard.

TABLE 7. Data format for different groups

Remainder after dividing your group number by 3	**Data Format**
0	Unsigned
1	Signed Magnitude
2	Two's Complement

7.2 Interim Goal and Deadline

Two members of the group should develop the ***Command Processor*** module, which will interpret the specified commands typed into the command terminal, and print the requisite responses. The other two members should develop the ***Data Processor*** module that communicates with the supplied ***Data Generator*** module and detects the peak in a given sequence.

You are expected to start work on Assignment 2 from **course week 4** (***week 16 in the term timetable***). By the end of **course week 8** (***week 20 in the term timetable***) which coincides with the last week of term before the Easter break, every group should submit **two items of source code** corresponding to the data and command processors **and a report in PDF format**. Every student has exactly ***five*** scheduled sessions in this time period. You should attend each of these sessions, and also spend time on your own with your team members as needed to complete the required tasks.

Details of the three deliverables are given in Section 7.2.1 (***Command Processor***), Section 7.2.2 (***Data Processor***) and Section 7.2.3 (***Report***). Please check the Assignment 2 section in Blackboard for access to artefacts available for download. Template testbenches are provided for both the command and data processors. For the interim submission, full functional compliance for the ***Data Processor*** is required while only partial functional compliance is required for the ***Command Processor***. This is because the ***Command Processor*** is more complex to develop.

Note:

- The compliance of your source code submission with the specifications described under Section 7.2.1 and Section 7.2.2 will be very strictly checked. If your submission for the command processor or the data processor does not conform to specifications, you will not get any marks for functionality.
- Though the functionality is checked in a simulation, the code should be written in an RTL style that conforms to the synthesis guidelines and uses the templates for inferring combination and sequential logic described in Assignment 1. Marks will be awarded for good design practice, and the synthesisability of the code. A submission that does not adhere to the good design practices outlined in Assignment 1 and does not synthesise will likely get low marks, even though full functional compliance is met in a simulation.

Finally, please note that all deliverables pertaining to a group should be submitted as a **single submission**; i.e. one member of the group should submit everything in one go, as subsequent submissions will overwrite previous submissions.

7.2.1 Command Processor

The team working on the ***Command Processor*** unit should demonstrate functional compliance to the *process bytes* command in a simulation. Printing “`ANNN`” or “`aNNN`” where `NNN` is a three decimal digit sequence from 000 to 999 should result in the requested number of bytes being retrieved from the synthesised version of the ***Data Processor***, and printed. The printing needs to be carried out through a UART ***Tx*** module, and the output of the ***Tx*** module will be checked based on a baud rate of 9600 for the specified strings in hexadecimal format.

The reading of the command should be through the ***Rx*** module. The upper and lower case versions should be accepted, and wrong commands not result in any output.

Source code for the ***Command Processor*** that demonstrates this behaviour in a ModelSim simulation ***with the provided testbench*** should be uploaded to Blackboard.

7.2.2 Data Processor

The team working on the ***Data Processor*** unit should submit source code for the unit that demonstrates through simulation that any given number of words according to the specification can be processed, and the peak detected. The following will be checked:

- Correct retrieval of all seven bytes, including positioning with the peak in the middle;
- The value of the peak index.

Source code for the Data Processor that demonstrates this behaviour in a ModelSim simulation ***with the provided testbench*** should be uploaded to Blackboard.

7.2.3 Report

You have to produce a single report for each group of approximately 7-8 pages (a maximum of 8 pages), that has four sections:

- Description of the group composition and task division between group members (who does what).
- Description of the project plan, including a diagrammatic work plan such as a Gantt chart that shows the timelines for tasks, deliverables and milestones, as well as group meetings.
- Description of the architecture of the Data Processing module including a block-level sketch of the main logic components and a state diagram for the FSM.
- Description of the architecture of the Command Processing module including a block-level sketch of the main logic components and a state diagram for the FSM.

In summary, a professional report should be produced that describes your project plan and also clearly shows the architectures of your command and data processors with appropriate diagrams and descriptions. Even though different members of the group may contribute different parts of the report, it should read as a cohesive whole, and have consistent formatting and numbering of sections, figures and tables.

7.3 Final Goal and Deadline

After the interim submission, ideally you should have a ***Data Processor*** that has full functional compliance with specifications, and a ***Command Processor*** that has partial functional compliance. When you return from the Easter break, feedback from your interim submission will be available. In the two weeks after your return, you will have three scheduled sessions. In this time both teams within a group should work on developing the remaining functionality of the ***Command Processor*** and integration of the two units, taking the feedback into account.

By the end of **course week 10** (***week 22 in the term timetable***) a code archive that includes the bit file with full specified functionality to programme the FPGA should be uploaded to Blackboard according to the instructions available on Blackboard. The exact deadline will also be available on Blackboard.

Please note that no report is necessary, and the following aspects are checked in evaluating your design:

- Adherence to specifications when implemented on the board.
- Design excellence of the Command Processor.
- Design excellence of the Data Processor.

7.4 Group Presentations

In **course week 10** (***week 22 in the term timetable***) every group is required to present their design. Each group will have strictly 16 minutes to present their design followed by 4 minutes of questions for a total of 20 minutes. Each member must present for 4 minutes on average. The follow criteria will be considered in allocating a group and individual mark:

- Description of group work (how clear are the objectives, and the work carried out, how well is the overall presentation organised, do the different presenters convey a cohesive message)
- Description of individual contribution (how effectively do the individuals present their case, how clear is the contribution)

Part 3: Using the Software and Hardware

8 Introduction to Xilinx ISE

Recall the design flow introduced in Assignment 1, which is reproduced in Figure 8.

In this assignment, we will examine the entire design flow from design entry to implementation in a Xilinx FPGA. The Xilinx ISE Foundation software is at the heart of all processes and controls all aspects of the design flow. For different tasks in the design flow, specialised tools from different vendors can be used. These tools are embedded within ISE, which provides a convenient and integrated front end to everything. In this particular lab, we will use the ModelSim simulator which you used in Assignment1 for behavioural simulation, and native Xilinx tools for logic synthesis and physical design.

Note:

- The program help is a very good source of information, tutorials and general documentation as is the Xilinx website, and should be referred often.

The following sections give a step-by-step introduction to starting a design project, synthesising the design, mapping it and downloading it onto the FPGA.

8.1 Creating a Design Project

Task 8.1: Creating a new project in the ISE software

1. Download the source file bundle for Assignment 2 from Blackboard, save it on a local folder and decompress the archive. The relevant files for this exercise are to be found in the sub-folder named "`uart_rx_tx_stripped`".

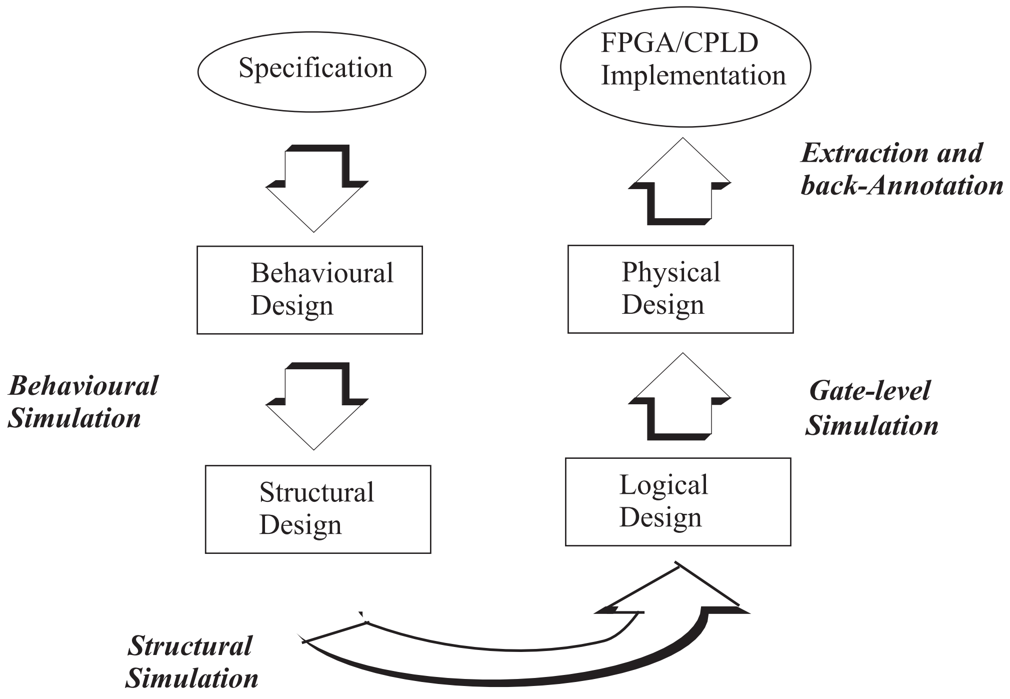


FIGURE 8. Digital Design in Programmable Logic

2. Start the program Xilinx ISE by clicking on **Start -> All Programs -> Xilinx Design Tools -> ISE Design Suite 14.2 -> ISE Design Tools -> 64-bit Project Navigator**. Close the welcome message that may open up.
3. Click on **New Project...** in the **Welcome to the ISE Design Suite** area on the left hand side as shown in Figure 9. and a new dialogue will open. In appropriate fields, specify a descriptive project name (such as Lab2_yourName) and also a folder where you will store the source and other files generated by the program. Select the top-level source type to be HDL and click on **Next**.
4. In the next dialogue, specify the **Product Category** as **All**, **Family** as Spartan6, **Device** as **XC6SLX16**, **Package** as **CSG324** and **Speed** as **-3**. These specify the target architecture, including the FPGA type, package type and speed rating. Select the Synthesis tool as **XST (VHDL/Verilog)**, and the **Simulator** as **ModelSim-SE VHDL**. Click **Next** and then **Finish** in the following dialogue box. An empty project will open.
5. Click on the "`Add Source`" icon to add an existing source (or "`Add Copy of Source`" if you want to create a copy of an existing source) as shown in Figure 10. You can also run this command from the menu item **Project -> Add Source**. Navigate to the sub-folder "`uart_rx_tx_stripped`" (to be found in the folder where you unzipped the source bundle) and add the "`test_top.vhd`" file.

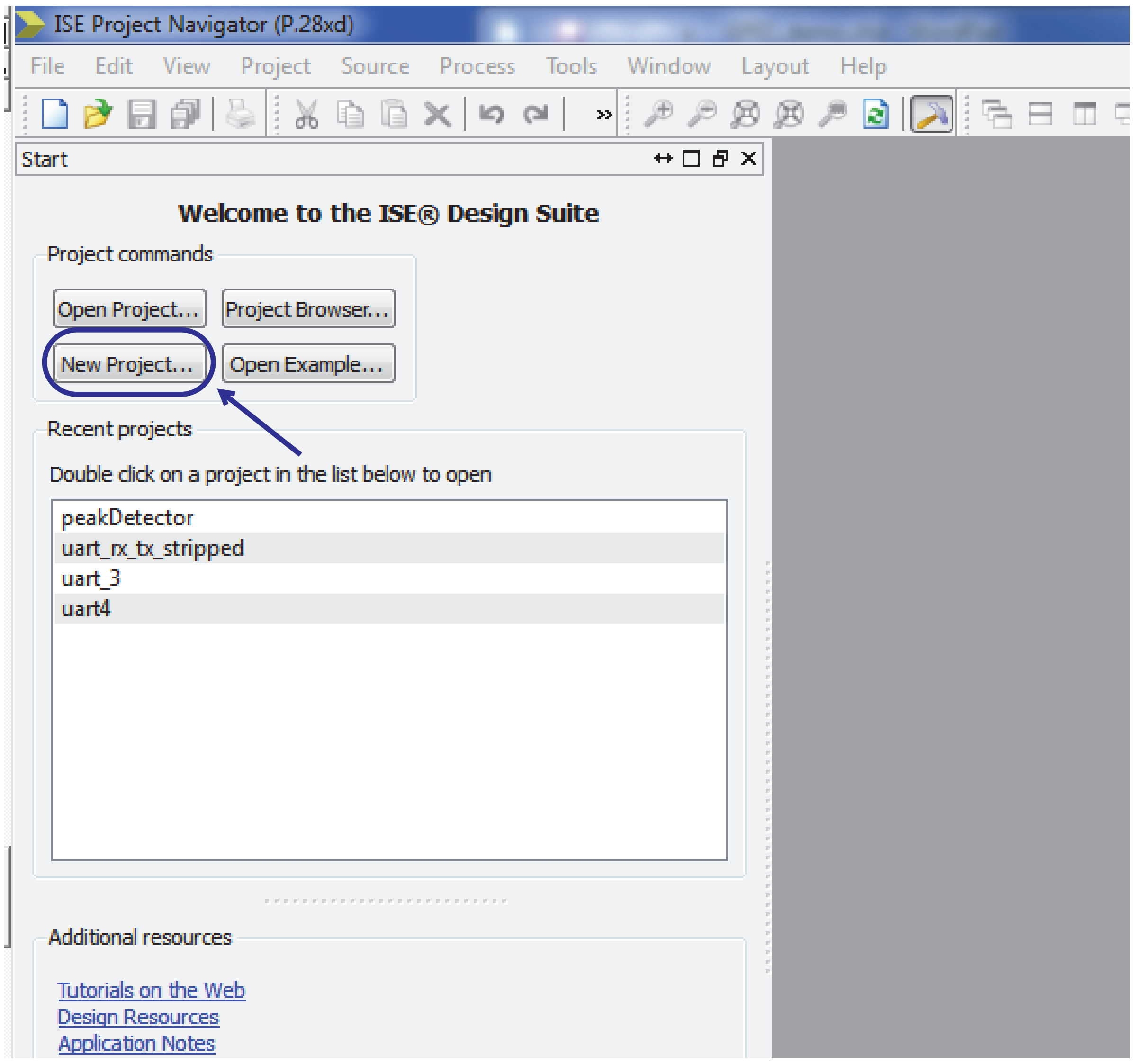
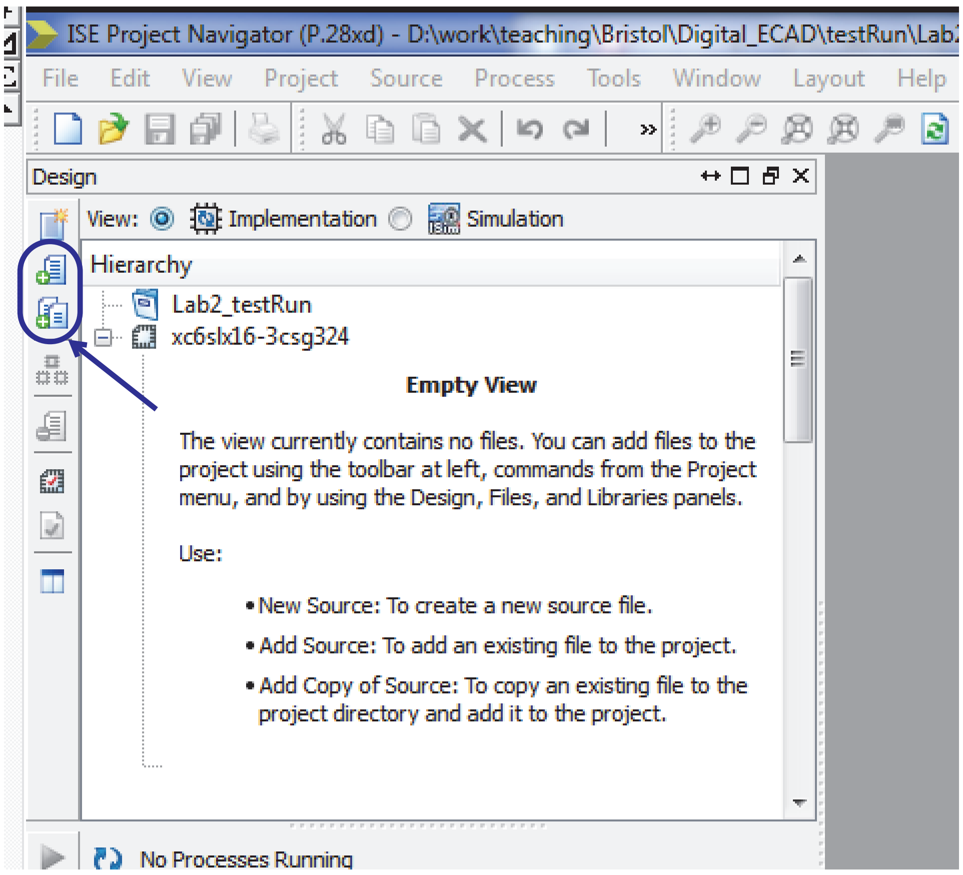


FIGURE 9. Start a new project in ISE Project Navigator


ISE Project Navigator (P.28xd) - D:\work\teaching\Bristol\Digital_ECAD\testRun\Lab
File Edit View Project Source Process Tools Window Layout Help

View: Implementation Simulation

Empty View

The view currently contains no files. You can add files to the project using the toolbar at left, commands from the Project menu, and by using the Design, Files, and Libraries panels.

Use:

- New Source: To create a new source file.
- Add Source: To add an existing file to the project.
- Add Copy of Source: To copy an existing file to the project directory and add it to the project.

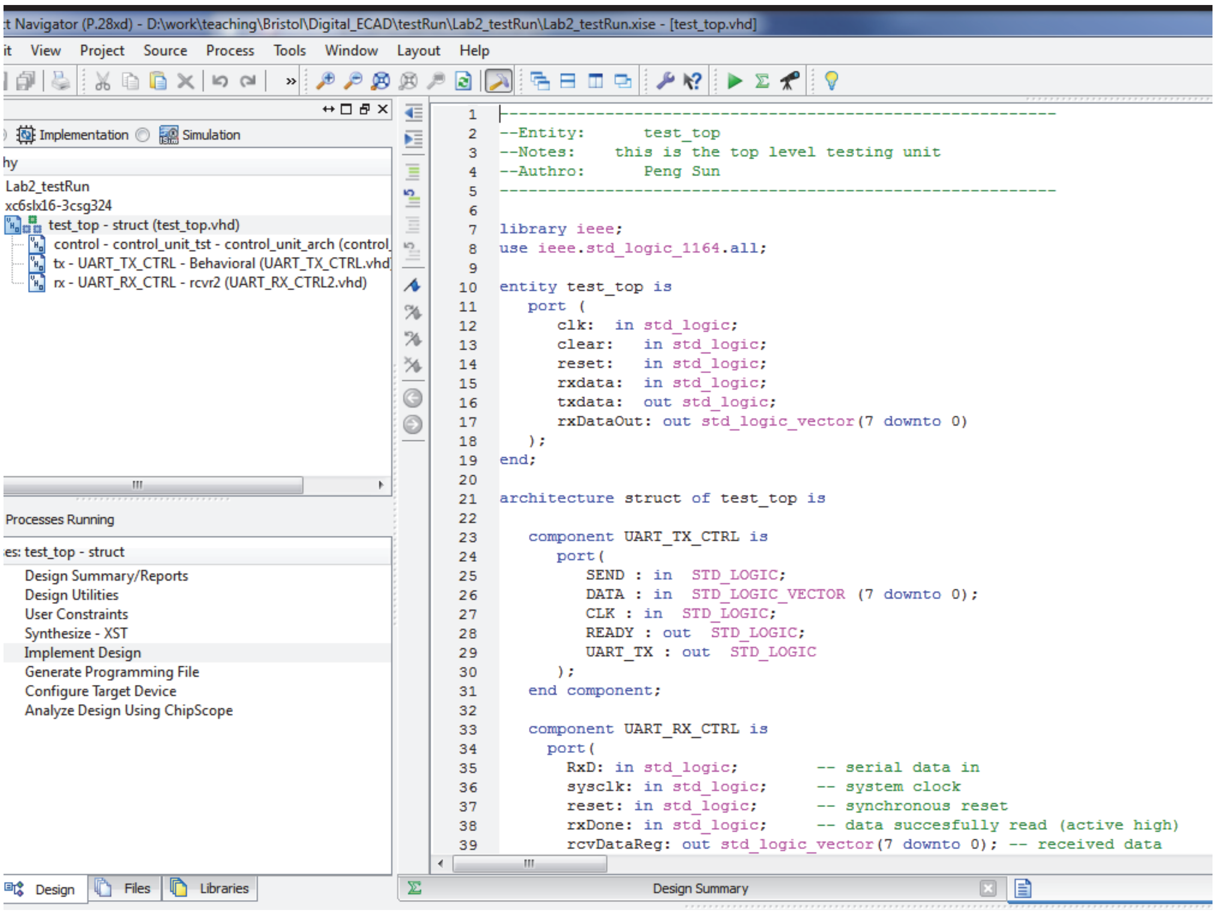
No Processes Running

FIGURE 10. Add source files to a project in the ISE Project Navigator

6. This file is the component at the top of the hierarchy which simply instantiates the components in the design. This is a purely structural description of the design. You can double-click on the component to view the source code in a new tab. Browse the code to see how the full design consists of three components, the Transmitter, Receiver and Controller wired together. These components will appear by name in the design hierarchy tab as shown in Figure 10, with orange coloured question marks against them as the source code corresponding to these modules haven't been added yet.
7. Add the sources corresponding to each of these modules from the source bundle, and the question marks will be replaced by "source code" icons. Browse the source code for each of these components.
8. A final file needs to be added to the project called "`test.ucf`" using the same procedure used to add source code. This file specifies how the I/O in the FPGA is to be configured. After adding it, double-click the file to open it and browse it. It contains only a few lines as we only have 4 external pin connections. Any line beginning with a "#" is a comment. The word "`net`" precedes identifiers used in the top level module of the design in "`test_top.vhd`". These names are all port identifiers. The word "`LOC`" is used to describe specific pins in the package.

Note:

- A full description of all the available pins and I/O is given in the Nexys 3 Board manual which is available from the Digilent website, and also saved on Blackboard for your convenience.



t Navigator (P.28xd) - D:\work\teaching\Bristol\Digital_ECAD\testRun\Lab2_testRun\Lab2_testRun.xise - [test_top.vhd]

Lab2_testRun
xc6slx16-3csg324
test_top - struct (test_top.vhd)
control - control_unit_tst - control_unit_arch (control
tx - UART_TX_CTRL - Behavioral (UART_TX_CTRL.vhd
rx - UART_RX_CTRL - rcvr2 (UART_RX_CTRL2.vhd)

es: test_top - struct
Design Summary/Reports
Design Utilities
User Constraints
Synthesize - XST
Implement Design
Generate Programming File
Configure Target Device
Analyze Design Using ChipScope

```
----------------------------------------------------------
--Entity:      test_top
--Notes:    this is the top level testing unit
--Authro:      Peng Sun
----------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity test_top is
   port (
      clk:   in std_logic;
      clear:    in std_logic;
      reset:    in std_logic;
      rxdata:   in std_logic;
      txdata:   out std_logic;
      rxDataOut: out std_logic_vector(7 downto 0)
   );
end;

architecture struct of test_top is

   component UART_TX_CTRL is
      port(
         SEND : in  STD_LOGIC;
         DATA : in  STD_LOGIC_VECTOR (7 downto 0);
         CLK : in  STD_LOGIC;
         READY : out  STD_LOGIC;
         UART_TX : out  STD_LOGIC
      );
   end component;

   component UART_RX_CTRL is
     port(
       RxD: in std_logic;         -- serial data in
       sysclk: in std_logic;      -- system clock
       reset: in std_logic;       -- synchronous reset
       rxDone: in std_logic;      -- data succesfully read (active high)
       rcvDataReg: out std_logic_vector(7 downto 0); -- received data
```

FIGURE 11. View of ISE Project Navigator

The two lines below the pin definition for the "`clk`" signal specify timing constraints. The line "`Net "clk" TNM_NET = sys_clk_pin;`" groups together all nets (wires) connected to "`clk`", and gives them the identifier "`sys_clk_pin`". The line "TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 100000 kHz;" specifies that this group of nets has to satisfy a period (timing) constraint of 100 MHz.

Note:

- The Project Navigator Interface is divided into four main sub windows, as seen in Figure 11. On the top left is the **Hierarchy** window which hierarchically displays the elements included in the project.
- Beneath the **Hierarchy** window is the **Processes for Current Source** window which displays available processes for the selected source.
- The third window at the bottom of the Project Navigator is the **Console** window which displays status messages, errors, and warnings, and which is updated during all project actions.
- The fourth window to the right is a multi-document interface (MDI) window for viewing text files (sources as well as design summary) and other output that may be generated depending on the tool invoked within ISE.
- Each window may be resized, undocked from Project Navigator or moved to a new location within the main Project Navigator window. These windows are discussed in more detail in the following sections.

8.2 Synthesising the Design

The synthesis tool performs three general steps (although all synthesis tools further break down these general steps) to create a netlist from the HDL description:

- Analyse / Check Syntax (Checks the syntax of the source code)
- Compile (Translates and optimises the HDL code into a set of components that the synthesis tool can recognize.)
- Map (Translates the components from the compile stage into the target technology's primitive components.)

The native Xilinx synthesis program is called **XST**. ISE allows you to specify synthesis programmes from other vendors as well, just as for behavioural simulation. You can set these options in the project properties. To change the properties at any time, select the targeted device (xc6slx16-3csg324), right-click and select **Design Properties**. In this assignment we will use **XST** so you do not have to change anything.

Task 8.2: Synthesising the design

1. Select the top module **test_top** in the Sources window and in the Processes window right click on the **Synthesize - XST** process and select "`Process Properties`". Change the **Optimization Effort** to **High** and tick the **Write Timing Constraints** box. Click **OK**.
2. Expand the **Synthesize - XST** process by clicking on the "+" icon to its left, when you can see the sub processes of the synthesis process, and you can monitor the progress. Double-click on **Synthesize -XST** and observe the messages in the Console window. Pay special attention to any warnings.

   If completed successfully, you should see a green check mark next to the **Synthesize - XST** process. If there are no errors but some warnings, you will see a yellow triangle with an exclamation mark inside it. In this case we have some warnings.
3. Click on the "`Warnings`" tab at the bottom of the console to load all only the warning. The first two warnings are easy to understand, because we actually never use the overrun and framing error outputs from the receiver in this simple design. Because they are never used, XST will not actually route these nets. To understand the next 6 warnings, you have to dig into the code of the transmitter. Because the stop and start bits are always defined as '1' and '0', XST does not need to load and store their values in a flip-flop or latch. Instead they can simply be always connected to VDD or Ground. These warnings therefore are not about issues we need to worry about, and will not affect the functionality of the design.

Note:

- You ignore warnings at your peril! Often they will result from syntactically correct but semantically incorrect code, such as implying a latch unintentionally by forgetting to assign outputs in all `if` or `case` branches of a combinational process. Do not ever proceed if you don't understand what each and every warning means, because it will usually result in your design not working.

4. In the MDI window, click on **Design Summary** tab. Under **Detailed Reports**, select **Synthesis Report** and spend some time reading it. Try to understand the meaning of all the messages.
5. Expand the **Synthesize - XST** process and double-click on **View RTL Schematic**. In the window that opens, select "`Start with a schematic of the top-level block`". A schematic of your design will open in a new window. You can descend into the top level module to view the implementation at different levels of hierarchy, and see how XST has mapped the behavioural code to the logic primitives available in the target architecture.

Note:

- The synthesis report gives many details of your design, including the critical path latency and the maximum clock period with which your circuit may be operated.

8.3 Implementing the Design in the FPGA

Design implementation is the process of translating, mapping, placing, routing, and generating a BIT file for your design. The design implementation tools are embedded within ISE.

Task 8.3: Setting implementation options

1. Click on **Edit -> Preferences** and in the **Preferences** dialogue box, Select **ISE General**. Change the **Property Display Level** from **Standard** to **Advanced**. Click OK.
2. In the **Processes** window, right-click **Implement Design** and select **Process Properties**. The **Process Properties** dialogue box provides access to the Translate, Map, Place and Route, Simulation, and Timing Report options. Select the **Post-Place & Route Static Timing Report Properties** category and change **Report Type** to **Verbose Report**.
3. In the **Place & Route Properties** category, change the **Place & Route Effort Level (overall)** to **High**. (This may already be set to high). This option increases the overall effort level of Place and Route during implementation.

Task 8.4: Mapping the design

Now that all implementation strategies have been defined (options and constraints in the UCF file), we can continue with the implementation of the design.

1. In the **Processes** window, expand I**mplement Data**, right-click on **Map** and select **Run** from the menu. Expand the **Implement Design** hierarchy to see the progress through implementation. The design is mapped into Configurable Logic Blocks (CLBs) and Input Output Blocks (IOBs). If you are unfamiliar with these terms, take a moment to browse the document on the Xilinx FPGA architecture.

Note:

Map performs the following functions:

- Allocates CLB and IOB resources for all basic logic elements in the design.
- Processes all location and timing constraints, performs target device optimisations, and runs a design rule check on the resulting mapped netlist.

Each step generates its own report including the **Translation Report** and the **Map Report**. The former includes warning and error messages from the translation process, while the latter includes information on how the target devices resources are allocated. For more information, refer to the **Xilinx *Development System Reference Guide***.

2. To view a report right-click any of these processes and select **View Text Report**. Check both the **Translation Report** and the **Map Report**.

Task 8.5: Timing analysis after mapping[1]

After the design is mapped, use the **Logic Level Timing Report** to evaluate the logical paths in the design. Because the design is not yet placed and routed, actual routing delay information is not available. The timing report describes the logical block delays and estimated routing delays. The net delays provided are based on an optimal distance between blocks (also referred to as unplaced floors).

Note:

- For a preliminary indication of how realistic your timing goals are, evaluate the design after the map stage. The actual total path delay after the design is routed may be much more than the estimated block delays, sometimes twice as much, or even more. For this small design, you will see when you compare the values later that the estimate is quite close.
- If your design is extremely dense, the **Logic Level Timing Report** provides a summary analysis of your timing constraints based on block delays and estimates of route delays that can help to determine if your timing constraints are going to be met. This report is produced after **Map** and prior to **Place and Route (PAR)**.
- For this exercise, timing constraints were defined earlier, and, as a result, the **Report Paths in Timing Constraints** option is selected. When **Implement Design** is run with the **Report Paths in Timing Constraints** option selected, a **Logic Level Timing Report** is generated with a period and path analysis for each constraint specified.

1. In the **Processes** window, expand the **Map** hierarchy. Double-click **Generate Post-Map Static Timing**. To open the **Post-Map Static Timing Report**, double-click **Analyse Post-Map Static Timing** and the report will open. Browse this report and try to understand the information contained in it.

Note:

- Even if you do not generate a **Logical Level Timing Report**, **PAR** still processes a design based on the relationship between the block delays and timing specifications for the design. For example, if a PERIOD constraint of `8 ns` is specified for a path, and there are block delays of `7 ns` and unplaced net delays of `3 ns`, **PAR** stops and generates an error message. In this example, **PAR** fails because it determines that the total delay (`10 ns`) is greater than the constraint placed on the design (`8 ns`). Therefore, you should always use the **Logic Level Timing Report** to determine timing violations that may occur prior to running **PAR**. Remember, every design has a time budget and all these design processes are time consuming!

Task 8.6: Placing and routing the design

The design can be placed and routed after the mapped design is evaluated. Your evaluation should have verified that block delays are reasonable given the design specifications (for this simple design that is easily so). The Flow Engine can perform a **Timing Driven Placement** which runs **PAR** with timing constraints specified from within the input netlist or from a constraints file. Else it can run a **Non-Timing Driven Placement** which runs **PAR** and ignores all timing constraints. Since timing constraints were defined earlier in the lab the **Place and Route (PAR)** process performs timing driven placement and timing driven routing.

1. You may skip this section on a first read. These details are necessary when including performance information of your design in your final group presentation.

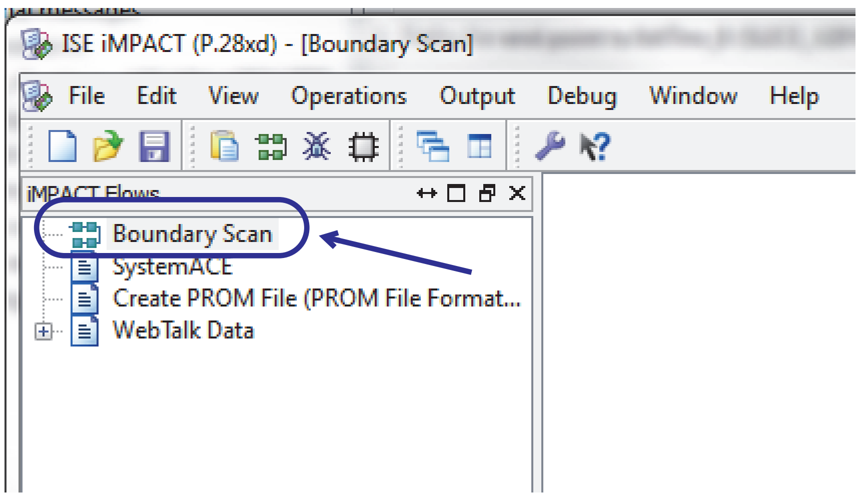


FIGURE 12. View of ISE Impact

1. To run **PAR** in the **Design Implement** hierarchy, double-click **Place & Route**.
2. Expand the **Place & Route** hierarchy, and you should see that timing and power reports are available to view. These commands open up new tools, and you can skip these in a first read. However these are powerful utilities that enable sophisticated timing and power analyses.
3. To ensure that the place and route process finished as expected, expand the **Generate Post-Place and Route Static Timing** button with a green check mark next to it**,** and double-click the **Analyze Post-Place and Route Static Timing** command. This post-layout timing report is generated by default to verify that the design meets your specified timing goals. This report evaluates the logical block delays and the routing delays. The net delays are now reported as actual routing delays after the Place and Route process.

Exercise 8.1: Compare post map and post place-and-route timing

1. What are the post synthesis, post **PAR** and post Map minimum clock periods and how do they compare?

Task 8.7: Configuring the FPGA device

Once we are confident about the design after analysing the timing reports, we can proceed to creating configuration data. Creating configuration data and uploading the bitstream to the target device is the final step in the design flow.

1. Right-click on **Generate Programming File** and select **Properties**. The **Process Properties** dialogue box opens. Select the **Startup Options** category and change the **FGPA Startup Clock** property from **CCLK** to **JTAG Clock**. Leave the remaining options in the default setting and click OK to apply the new properties.
2. Double-click **Generate Programming File** to create a bit stream of this design. The bit stream is created by the **BitGen** program which generates the "`test_top.bit`" file. This file is the actual configuration data. Verify that the file is in the project directory.
3. Expand **Configure Target Device** and double-click on **Manage Configuration Project (iMPACT)** and the ISE iMPACT programme will start. You can also start iMPACT as a standalone utility from Windows, by selecting **Start -> All Programs -> Xilinx Design Tools -> ISE Design Suite 14.2 -> ISE Design Tools -> 64-bit tools -> iMPACT**. The iMPACT utility allows us to configure the device in a variety of ways.

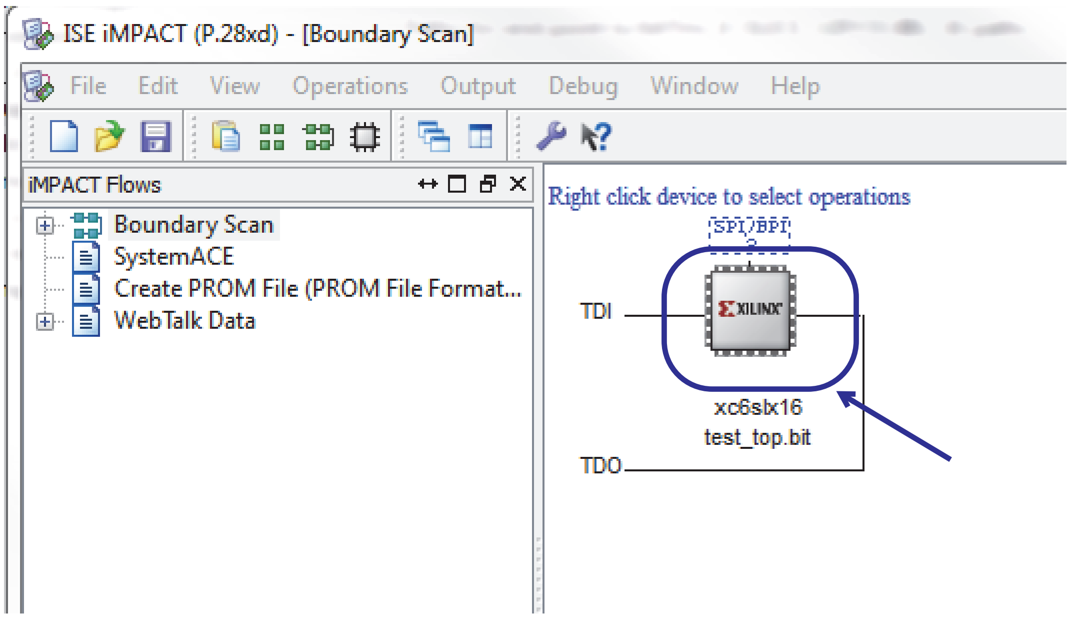

ISE iMPACT (P.28xd) - [Boundary Scan]
File Edit View Operations Output Debug Window Help
iMPACT Flows
Boundary Scan
SystemACE
Create PROM File (PROM File Format...
WebTalk Data
Right click device to select operations
SPI/BPI
TDI
TDO
xc6slx16
test_top.bit

FIGURE 13. Adding the bit file in iMPACT

4. In iMPACT, double click on **Boundary Scan** as shown in Figure 12.
5. Make sure the board is connected to the PC via both the power up / configuration and UART USB cables, and that the power switch is on. The red LED next to the power switch should be lit. Either right click in the blank window on the right and select **Cable Auto Connect** or select **Output -> Cable Auto Connect** from the top menu. The board should be detected by Impact with several messages appearing in the console.
6. Right click and select **Add Xilinx Device** and select the "`test_top.bit`" file. An icon of a Xilinx chip should appear as shown in Figure 13.
7. Right-click the FPGA device and select **Program**. Click OK in the next dialogue box. You should see a "Program Succeeded" message in iMPACT. Open a PuTTY terminal as specified in Section 3.3, and you should see whatever you type being echoed in the terminal.

9 Design and Verification Methodology

Xilinx has its built-in simulator, but we will continue to use the ModelSim simulator. It is very important that you verify the full functionality in a simulation before you move to Xilinx and start synthesising the logic.

9.1 Modularity

The way to proceed with your design is to pay strict attention to the principle of modularity. Essentially, when designing a component, ensure that it does one thing, and one thing only, and does it well. The notion of a component can be applied to any individual block of code. For example, assume you need a decoder block; this can be implemented in a single process body. This decoder block may reside in a file with other such components, but the functionality of the decoder block itself is precisely defined, and it is always a good idea to write a small test bench that tests the functionality of the decoder block separately. Debugging a large design is much easier if you have constituent components with clearly defined functionalities, that have been separately verified. If you have a large monolithic block of code with no clear structural lines, debugging it can be a very difficult task. ***See also Section 7.2***.

Note:

- For hints on good design practice and adhering to the principle of modularity, read the supplied source code for the Rx, Tx and Data Generator carefully.

9.2 Using the supplied code artefacts

9.2.1 Starting

All groups should start by implementing the stripped down version of the Rx and Tx and try to understand how it works. Scrutinise the code for the Rx, Tx and the Controller to look at examples of good design practice. The coding techniques adopted here include macro techniques such as modularity, division along clear structural lines and good naming conventions. They also demonstrate good coding techniques for describing the overall design as a data path and a controller defined as a finite-state machine, as well as templates for describing the combinational and sequential elements in the data path and finite-state machine.

9.2.2 Developing the Command Processor

The teams working on the Command Processor can start adding functionality to the controller in the stripped down version of the UART to recognise the requisite commands. You should try developing simple testbenches *of your own* to check for incremental additions before using the supplied testbench to test for the functionality required for the interim submission. To test against the requirements for the interim submission, you can use the testbench available in the "`cmdProcessor`" sub-folder of the code archive.

9.2.3 Developing the Data Processor

The teams working on the Data Processor should scrutinise the Data Generator source to understand how the single-phase protocol works and start developing the Data Processor. You should try developing simple testbenches to check for incremental additions before using the supplied testbench to test for the functionality required for the interim submission. You can work with the testbench relevant for your group available in the "`dataProcessor`" sub-folder of the code archive to test against the requirements for the interim submission.

9.2.4 Developing the full system

When working on the final phase of your design, the teams within a Group can use the supplied code archives available in the sub-folders under "`fullSystem/`" to ensure compliance with the specifications. To start with, create a ModelSim project and load all the files contained in the sub-folder ("`unsigned`", "`signedMag`" or "`twosComp`") that is relevant for your project.

The testbench instantiates all the components and issues two start commands with 2 and 13 bytes respectively, followed by print list and print peak commands by driving the input line of the UART Receiver. Run the simulation for at least 160 ms to see the full simulation for these commands.

In order to replace the Command Processor (Data Processor) module in this testbench with your own you should delete the "`cmdProc_wrapper.vhd`" and "`cmdProc_synthesised.vhd`" ("`dataConsume_synthesised.vhd`" and "`dataConsume_wrapper.vhd`") files from the ModelSim project, and add the file which you should name "`cmdProc.vhd`" ("`dataProc.vhd`") that contains the new source.

9.3 Simulator

It is possible to invoke ModelSim from within Xilinx, but at the initial design state, when you are developing the functionality of the peak detector, it is not necessary. Simply have a collection of test benches with which you can test the functionality of sub components and a test bench for the complete design. One you are satisfied with the functionality, you can start synthesising the logic in Xilinx.

Checking waveforms in simulations can be challenging when there is a big difference in the system clock rate and a signalling rate, as is the case with the UART. This means that you will have to simulate millions of 100 MHz cycles in order to transmit or receive a byte. One way to sidestep this issue is to change the timing constants in the supplied Rx and Tx implementations, so that transmitting and receiving words consume far fewer cycles. When transferring the code over to the Xilinx project, you can change the constants back to the original values to adhere to the specified baud rate.

Sometimes it is nevertheless useful to run the simulation with the final time constants that will be used for synthesis. With a simulation that runs for several million clock cycles (e.g.: a few hundred milliseconds worth of simulation time) you will find that the Xilinx waveforms load much faster if you omit the clock from the waveform viewer. The simulation does not actually take that long, but updating the waveforms on the screen does.

Recall from Assignment 1 that in ModelSim you can set break points on any executable line of code and run the simulation interactively, to check the behaviour of a given block of code. You can thus check the value of signals and variables at any given point in the simulation, step through individual lines of code, set up watches on signals and variables, and generally take advantage of most of the debugging tools available in a software programming environment that you will be familiar with.

Note:

- Verify the functionality of your code with simulations in ModelSim before attempting to download a design to the FPGA.

9.4 Post place-and-route timing simulations

After place-and-route, the length of the wires are known, and the signal propagation delay along a given path is known much more accurately than at the synthesis or map stages. It is common to annotate the original netlist with these delays to run a timing simulation and check for any timing violations. This is the last step shown as **Parasitic Extraction** and **Back Annotation** in Figure 8. For those of you who want to run this step, instructions on how to annotate the netlist with the extracted delays and run a simulation in ModelSim will be available on Blackboard in a separate document.