Physical Computing

The Principles of Computing for Physicists

First Edition

Physical Computing

The Principles of Computing for Physicists

First Edition

Ed Daw
PHY31001, The University of Sheffield

Self Publishers Worldwide
Seattle San Francisco New York
London Paris Rome Beijing Barcelona

This book was typeset using LaTeX software.

Preface

Science and Engineering all about making connections between nature and mathematics. A successful scientific theory makes a connection between the behaviour of some real world hardware in an experiment and the properties of an abstract mathematical system that reproduces what is seen in that experiment. The starting point is observations about the real world hardware, and the end point is a mathematical model exhibiting the same behaviour, and perhaps predictive power. In engineering it is the same process in reverse. You start with an abstract idea that you want a real world system to replicate, and your end point is the realisation of that real world system - a machine. So science and engineering are both about relationships between nature and mathematics. In order to do either effectively, it is sometimes best to start from the simplest building blocks. Conceptually, the simplest abstract system is one that can only be in two states, say true or false. A two state system like this is called a bit. Bits turn out to be the basic currrency of computers that are some of the most sophisticated machines that engineers have produced, and have started the information revolution. In this book, we will be mostly concerned at first with bits and how to represent them using electronic circuits. We will catch glimpses of what those electronic circuits can do for science, and how they can be used as tools for building better abstract models of reality, also known as scientific theories.

The manipulation of these representations of bits using circuits is called digital signal processing (DSP). In classical digital signal processing, ranges of voltages on wires are used to represent the values of bits. For example, in 3.3V CMOS logic, the convention is that any voltage between 2.0 V and 3.3 V represents logic 1, or true, any voltage between 0.0 V and 0.8 V represents logic 0, or false. Just as the bits themselves are represented by classical voltage ranges, the operations on the bits are represented by circuits. For example, the simplest abstract digital operation I can think if is NOT, where if the input is a 0 then the output is a 1 and vice versa. An implementation of a NOT in a digital circuit must generate an output between 0.0 V and 0.8 V from an input between 2.0 V and 3.3 V, and vice versa. Though of course digital circuits have become incredibly sophisticated, we shall show that all of them are made up of three basic circuit elements - gates, registers and oscillators. So the fundamental building blocks of classical computers are fewer in number than the fundamental

building blocks of the Universe in particle physics!

In the first part of this book, we will learn to design and build digital circuits out of these three basic building blocks. The tasks that can be carried out by these circuits will be very simple, but I hope that by understanding the basic principles of how these blocks are combined, you will be able to see how these ideas can be generalised to far more complex systems. Forty years ago you would have needed a soldering iron, breadboard, or wire wrapping tool to assemble your circuits. We will instead use a more modern approach, and that is to make use of a hardware description language (HDL). This language enables us to describe the properties of the system we would like to represent. We then use computer software that translates our hardware description language code into a file called a bitstream that is used to configure a flexible device called a field programmable gate array (FPGA). Once programmed, the FPGA implements the digital circuit that represents the abstract system we described with our HDL code. This FPGA chip can be obtained housed on a development board with a variety of inputs and outputs that allow us to verify that the digital circuit is faithfully representing the operation of the digital circuit we described in our HDL code. You will also learn how exist half way between the abstract system and its hardware representation by simulating the behaviour of the digital circuit before you actually program the hardware device. In industry, hardware description languages are also used to design more specialised chips such as application specific integrated circuits (ASICs) and even entire processor cores. So, learning how to program FPGAs with hardware description languages (HDLs) is a useful and transferable skill.

In the second part of this book, we will learn how to make use of HDL code that others already wrote to describe sophisticated circuits combined with HDL code that we wrote ourselves. An analogy is in computer programming, where you rarely write every line of the code you are using yourself; instead you make use of libraries and routines that were written by others. In particular, we will implement a soft core processor called MICROBLAZE on our FPGA, and we will learn how to interface our own HDL code to the microblaze core. MICROBLAZE is a fully functional processor core, so in order to operate correctly it will need programming. We will learn to program our microblaze core in C, a high level computer language that is particularly suitable to working directly with digital hardware. We will learn the C programming that is needed for this task as we go along.

In the last part of the book, we consider an important generalisation of digital signal processing, called quantum computing. I said at the beginning that the basic currency of DSP is the bit, that can take two different values. What happens if we try and represent a quantum system using classical bits? For example, we might have a system consisting of an electron for which the z component of the spin can be either up or down, so that the states are $|\uparrow\rangle$ and $|\downarrow\rangle$. However, notice that in quantum mechanics $\left(|\uparrow\rangle + |\downarrow\rangle\right)/\sqrt{2}$ is also a possible state of this system. Therefore, classical bits are not suitable for representing the state of a quantum spin. This example leads us to the more general abstract

concept of a qubit, a system that can be in any linear superposition of two orthogonal states. It turns out that qubits can be manipulated using quantum circuits to perform computing tasks that are impossible with classical digital signal processing. We shall in the last part of this course learn more about the building blocks of quantum computers, and some of the problems encountered in trying to realise them in practice.

In summary, this book is part practical and part theoretical. The practical aim is to teach you the fundamentals of hardware description languages and computer architecture. These are skills you can use later for very practical purposes, like getting a job. The theoretical part is acquiring some useful knowledge that is a little off the beaten track, particularly for pure scientists, and understanding where we might be going next. The world of digital signal processing is colliding as we speak with the world of quantum mechanics, so this is an exciting time to be alive.

This book is very short for a good reason. Although the abstract concepts discussed above form a body of knowledge that will stand the test of time, the details of the hardware description language, the environment (called a software development kit or SDK) in which the HDL is developed, and the names and performance level of the FPGAs and the development boards on which they are housed change from year to year. Should this book become popular, I fully expect to have to re-write it in a few years. It will give me pleasure if the book is that successful, and because it is short I might have the energy for the task of re-writing it. Furthermore, a short book is more likely to be cheap enough for students to buy, and possible for them to read in the limited time that they have. In the meantime, I hope to make it clear as I go along which particular details I anticipate will change with time. I hope that because the book is short and fairly simple it will be a useful starting point for the readers own interests and curiosities, and not too difficult for me to update as things evolve. I hope you enjoy what is within these pages !

Ed Daw, 1st February 2021

Table of Contents

Chapter 1

Starting point

1.1 What is this course?

A course at University is intended to take the student on a journey between some starting point, a body of knowledge and skills which it is presumed they already have, to some finish point, where the student has learned some new material with some intended purpose. Many courses make the mistake of assuming the starting point is no knowledge at all! This is the safest bet for an academic, but it has two immense disadvantages. First, ignoring the student's existing knowledge means you end up re-teaching material, costing you precious time. Second, the student's existing knowledge may be in some context that they have never questioned. Pointing out that there is a wider world out there that they have not explored may pique their interest and motivate them to devote energy and time to working in your course. An under-appreciated reality of teaching is that it is in part the art of seduction. Students will not work for you unless you can succeed in motivating them to do so.

This course will attempt to raise your enthusiasm level by pointing out that what you probably know about computing is almost certainly confined to an artificial and some would say Utopian environment. Learning how to do computing outside this environment, in the real 'physical' world is liberating and fun. Furthermore, many of the applications of computers, or more generally of digital circuits and algorithms, underpin some of the worlds most interesting machines and experiments, from spacecraft to gravitational wave detectors. We will in this course explain the senses in which these statements are true, and we will learn some basic techniques in what I call 'physical computing'.

1.2 Analysis of a simple program

Most of you will have done a course in computer programming using a high-level language such as PYTHON. Below is a code listing that I lifted from an

online programming course for undergraduates.

```
import numpy as np
from matplotlib import pyplot as plt

ys = 200 + np.random.randn(100)
x = [x for x in range(len(ys))]

plt.plot(x, ys, '-')
plt.fill_between(x, ys, 195, where=(ys > 195), \
facecolor='g', alpha=0.6)

plt.title("Sample Visualization")
plt.show()
```

The program produces this graphical output.

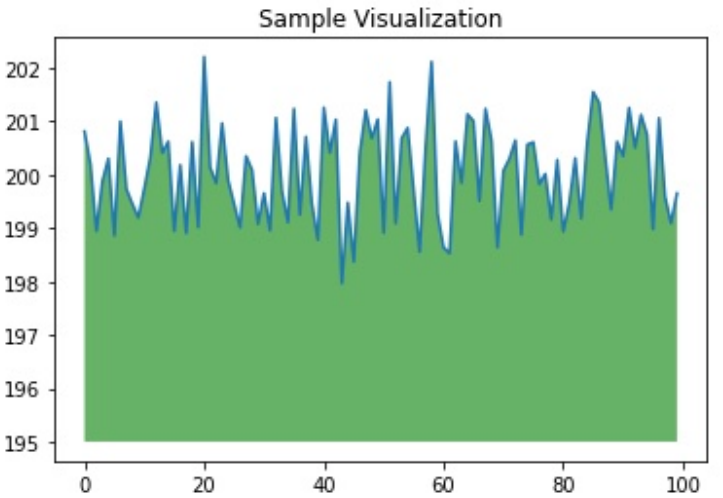


Figure 1.1: Graphical output of the PYTHON program

Let us analyse this program. First, its job is to make a plot of some random numbers. It probably isn't important exactly how long the code takes to produce this plot, as long as it isn't so long that it could have been done more efficiently without the computer. It also probably isn't critical that the program takes the same amount of time to produce this output every time it is run.

The program makes use of some imported packages `numpy` and `matplotlib`, for mathematical functions and graphical output respectively. It is somebody else's problem to implement these library functions. The `matplotlib` plotting function will only work if the computer has a way of outputting graphics. When

I ran this code, the graphics appeared in my web browser, but this was because I was using Jupyter notebook; other python environments would have put the graphics in a separate window.

Other observations about the program might seem obvious, but perhaps only because you have always taken them for granted. First, the program executes by default from top to bottom. Occasionally, there is a command that causes the code to go into a loop, so for example when the values of `ys` are generated, the command `randn` generates 100 random numbers, so this random number generator loops around and runs 100 times before the program moves to the next line of code. Similarly for the next line, which generates 100 values of `x`. All high-level programming languages feature loops, which cause code to execute multiple times. Another high-level programming concept is a conditional, meaning code that executes under some condition, but not if that condition is not satisifed. Loops and conditionals are so ubiquitous that you take it for granted that programs will utilise them. For that matter, you also take it for granted that code will execute from top to bottom.

Other things you might take for granted are particular to PYTHON. Notice how variables such as `x`, `ys` and `facecolor` represent data of different types, and that the type of data they represent is set automatically by PYTHON when the variables are defined. You don't generally have to declare what type of data a particular variable is going to be set to before that variable is used, PYTHON figures things out on the fly. In this respect, PYTHON differs from other high level languages like C or C++; in those languages you need to DECLARE a variable, including its data type, before you use it. You might think about whether this characteristic of figuring out variable types when they are first used is always a good thing, or whether perhaps it might occasionally cause problems.

The structure of high level computer languages generally comes about from them being intended for programming computers of a particular type. For example, most computers revolve around a very small number of processor cores which do all the calculations. Of course on a modern computer this may be less true than it used to be; there are often many cores on a CPU, and that doesn't include the GPU chip, where there are hundreds or thousands of cores that are more special purpose. In general, however, the architecture we imagine in a computer is that of a processor doing all the calculations, and input data and commands being piped in to this processor in sequence, and output form the processor coming out, so to speak, of the other side.

However, things do not have to be this way. In a more general calculating circuit, you can imagine the input data feeding in to the circuit through multiple parallel paths into a variety of different elements that all do things to subsets of the data at once. Such a circuit could be vastly more efficient than the gridlock that occurs when everything has to be stuffed in to a small number of processor cores. However, there is a price to pay, which is that the calculations in all these elements must be coordinated with each other, so that the outputs of the elements may combined and fed into yet more elements so that

more calculations can be done, but on the right numbers. This way of doing things is potentially far more powerful than a conventional single processing unit architecture, but the programming of such a machine is vastly more complex as well. And, consider, the arrangement of the calculating elements for a particular computation may also need to be optimised for the efficiency of that computation. For the next computation, you may want to arrange the circuit differently.

So, in this course, we will consider the construction of digital circuits that do calculations. These circuits are literally wires connecting very fundamental building blocks. Until the 1980s. digital circuits were assembled using literal physical wires or circuit boards, connecting these fundamental blocks together. If you wanted a new circuit, you had to build it out of hardware. In the 1980s, however, large scale integration of digitial hardware led to a more abstract way of doing things. You described the digital circuit that you wanted using a new type of language, a Hardware Description Language (HDL). By a wierd stroke of luck I programmed early circuits of this type, which were called PALs, or programmable arrays of logic, using an early HDL called ABEL, when I was doing work experience at school. I had no idea that I would wind up using it's direct descendent, called VHDL, a far more advanced hardware description language, in my physics research more than 35 years later. Life is strange sometimes. Anyway, the hardware description language literally permits you to build a digital machine. You are building hardware, using a language.

Having built the hardware, in simple cases, you just pipe the data in, and the desired output appears at the other end of your digital machine. This has two great advantages. First, it is extremely fast. There is no operating system or other environment to slpw down the process. Second, often the calculation takes exactly the same amount of time every time it is executed. This makes this type of machine suitable for use in situations where the output of the calculation is used to control the state of something critical, like a physics experiment, where unexpected delays cause glitches which may cause the machine or experiment to stop working.

In more complex cases, your machine may grow to resemble a computer, and you may need to write more code, this time perhaps in a high level language, to program it. But because you have far more control over the architecture of your machine than yo do with an ordinary computer, you are empowered to make a far more interesting, useful and powerful calculation engine. I hope you will enjoy learning physical computing, as I have enjoyed it. It's physics, but not perhaps as you have experienced it before.

Chapter 2

Logic and Integers

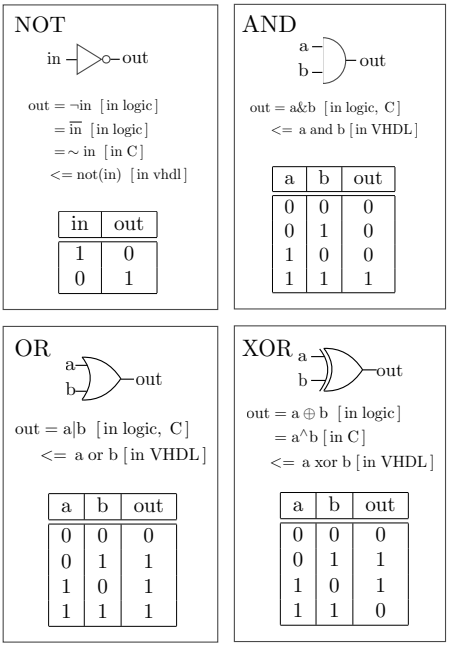
2.1 Logic Gates

We start our physical computing course where all beginners courses on digital circuits begin, with logic gates. A logic gate is an abstract operation having a single digital output and one or more digital inputs. There are seven logic gates that you will see on schematics. The simplest of them is a NOT gate, or inverter, mentioned in the preface, whose output is 1 when its single input is 0, and 0 when the input is 1. The next simplest gates each have two inputs. The two input AND gate has output 1 only when both its inputs are 1, the two input OR gate has output 1 when either or both of its inputs are 1, and the two input XOR (exclusive or) gate has output 1 when either one, but not both, of its inputs are 1. That's four so far. The other three gates consist of the AND, OR and XOR gates with an inverter on the output, forming the NAND, NOR and XNOR gates. The six two-input gates generalise to more than two inputs. For example, a 3 input XOR gate has output 1 when any one, but not more than one, of its inputs is 1.

An alternative to these written descriptions of functionality is to tabulate the outputs for each possible combination of inputs. Such tables are called truth tables. The truth tables for the four gates NOT, AND, OR and XOR are shown in Figure 2.1 along with the symbols representing the gates. The figure caption tells you how to draw the other three gates NAND, NOR and XNOR.

2.2 De Morgan's Laws

It turns out that you can build AND gates out of OR gates and NOT gates, or OR gates out of AND gates and NOT gates, as a consequence of a useful piece of maths that comes from the theory of sets, called De Morgan's laws. In the theory of sets, if A and B are each sets, then $A \cap B$ is called the intersection of A and B, and means all the elements that are in both set A and set B, and



in	out
1	0
0	1

a	b	out
0	0	0
0	1	0
1	0	0
1	1	1

a	b	out
0	0	0
0	1	1
1	0	1
1	1	1

a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

Figure 2.1: Truth tables and symbols for four of the seven logic gates commonly encountered. The other three, NAND, NOR and XNOR are obtained by adding a not at the output of AND, OR and XOR, respectively. The symbols are the same as for AND, OR and XOR except for the addition of a small circle like that on the right side of the NOT gate. Sometimes you will also see a small circle on one of the input leads before the gate body. For example, if this circle before the body of the AND gate appears on the b input, it means 'a and not b.'

$A \cup B$ is called the union of A and B and means all the elements that are set A or set B. So there is a direct correspondence between the logical AND and the intersection, and between the logical OR and the union. What about NOT? In set theory the symbol for all the elements not in set A is the complement of A, or $\overline{A}$. There are two De Morgan's Laws, and both can be stated either in set notation or in logic notation. The logical statements are

$$\neg(a\&b) = \neg a|\neg b \tag{2.1}$$
$$\neg(a|b) = \neg a\&\neg b. \tag{2.2}$$

In words, Equation 2.1 states that not(a and b) is the same as (not a) or (not b), and Equation 2.2 states that not(a or b) is the same thing as (not a) and

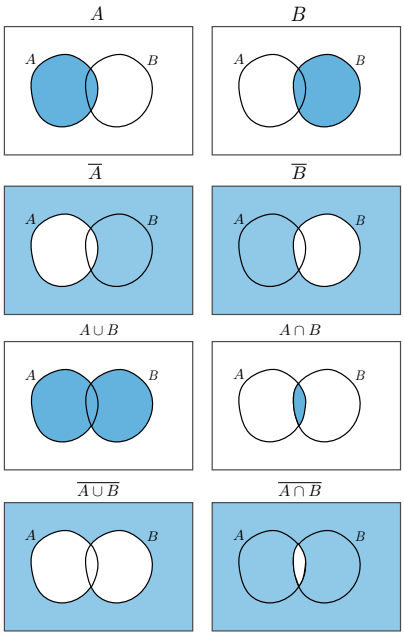


Figure 2.2: Venn diagrams illustrating De Morgan's laws.

(not b). The same two laws can be stated in set notation as

$$\overline{A \cap B} = \overline{A} \cup \overline{B} \tag{2.3}$$

$$\overline{A \cup B} = \overline{A} \cap \overline{B}. \tag{2.4}$$

If you like Venn diagrams, you can prove De Morgan's Laws with them. This is done in Figure 2.2. The rectangle represents all objects in some space, everything inside the left hand ellipse is in set A, and everything inside the right hand ellipse is in set B. The complements of A and B are shown in the second row. The union and intersection of A and B are in the third row. The fourth row shows the complements of the union and the intersection of A and B. The union of $\overline{A}$ and $\overline{B}$ is the areas that are blue in either of the diagrams in the second row. This is everything except the thin white sliver common between both A and B, in other words it is the complement of the intersection of A and B. This proves De Morgan's first law, Equations 2.1 and 2.3. The intersection of $\overline{A}$ and $\overline{B}$ is the areas that are blue in both of the diagrams in the second row. This is everything that isn't in either of the white ellipses of A or B, in other words it is the complement of the union of A and B. This proves De Morgan's second law, Equations 2.2 and 2.4.

De Morgan's laws simplify the job of chip designers since essentially all of the logic gate elements on a chip may be fabricated out of either OR gates or

AND gates, plus inverters. Figure 2.3 is a third statement of De Morgan's laws in terms of the symbols for the gates themselves.

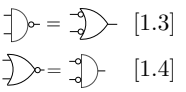


Figure 2.3: De Morgan's laws in terms of the equivalence between different types of logic gate.

2.3 Parallel Buses

We aren't going to get very far if we only ever deal with one bit at a time. The simplest way to streamline manipulations of multiple bits is to perform the same operation on many bits in parallel. A collection of bits in parallel is called a bus, and the number of bits in the bus is called the bus width. Frequently buses consist of 8, 16 or 32 bits in parallel. There are other kinds of buses other than parallel, and we will discuss some of these presently, but for now just imagine an assembly of signals in a set of wires. Buses are ordered. In other words, the position of a particular bit of a bus is well defined. Usually we think of a bus as an ordered set of bits from left to right, with the rightmost bit called the least significant bit, or LSB, and the leftmost bit called the most significant bit, or MSB. This is in preparation for use of parallel buses to represent integers, which we will discuss in Section 2.4.

The logic operations defined in Section 2.1 were so far discussed as operating on a single bit, but they can also operate on buses containing many bits, so long as each input and output has the same bus width. Used in this way, the logic operations NOT ($\neg$), AND ($\&$), OR ($|$), XOR ($\oplus$), NAND ($\neg\&$), NOR ($\neg|$) and XNOR ($\neg\oplus$) are examples of 'bitwise' operations, in that they operate on each bit separately and do the same thing to all the bits. So, for example,

$$\neg\, B00010000 = B11101111 \tag{2.5}$$

and

$$B10101010 \,|\, B00010000 \;=\; B10111010. \tag{2.6}$$

Here I have used the notation of a leading B to mean that the sequence of digits following represents a bus of bits - just a way of expressing whether the bits in a bus are high or low, 1 or 0, or equivalently true or false.

There are other binary operations that can only meaningfully act on parallel buses of bits. Bit shifting operations, in particular, involve shifting the values in all the bits in a bus to the left (in the direction of the most significant bit) or to the right (in the direction of the least significant bit). As a consequence, either the rightmost or the leftmost bit falls off the end of the bus and is lost,

and a gap appears at the other end. In a logical bit-shift operation, the gap is filled with a zero. In an arithmetic bit shift, the gap is filled with a copy of the value in the bit that must moved along to create the gap. Finally, you could possibly place the value in the bit that fell off the other end of the bus in this position. All three types of bitshift operations can sometimes be needed. Where the bus is representing logical rather than arithmetical data, logical bit shifting is the most common. An exercise in Appendix **??** will show you how to execute logical bit shifts on buses representing logical data in the hardware description language VHDL.

The use of binary operations to simplify and speed up computer code is an interesting subject. There is a nice book by Warren [3] where you can read about logical consequences of De Morgan's laws and mathematical tricks involving bitwise logic operations combined with bit shifts, particularly in Chapters 1 and 2, which I will make available to you on Collaborate.

2.4 Logic and Integer Arithmetic

2.4.1 Logic and the Foundations of Mathematics

De Morgan's laws are an example of the connection between the theory of sets and logic. This is natural because both sets and logic are closely connected to the foundations of mathematics. The drive to build a logical foundation for mathematics was an area of intense activity at the turn of the 20th century. The famous series of books, Principia Mathematica, by Russell and Whitehead [4], an attempt to build a consistent and complete logical foundation for mathematics, famously collided with the philosophy of Kurt Gödel who proved that the task is impossible. Modern formulations of axiomatic set theory, or the logical foundations of mathematics, are based on Zermelo-Fraenkel set theory, and it is in this context that modern developments in the logical foundation of mathematics are usually discussed. The subject of the logical foundations of mathematics is fascinating. If you are interested, and there are many popular books on the subject which will stretch your mind, especially if you try and do the puzzles [1, 2] !

Fortunately we do not need to construct a complete logical basis for mathematics, only to work out some ways of using logic gates to do simple arithmetic, which is much easier. Nonetheless, we shall quickly discover that in using logic to do mathematics you have to be careful to avoid encountering difficulties. These difficulties shall be discussed in Chapter 3.4.

2.4.2 Binary Representation of Positive Integers

Let us first consider representation of integers. Bits naturally lead to the use of binary numbers to represent integers. Any sensible representation will have the integer zero represented by any number of 0s, and the integer one represented by a single 1. If the integers are all positive or zero, then all we need is ordinary

binary numbers, so the numbers 0 to 15 can be represented by three binary digits. In ascending order, $B0000$, $B0001$, $B0010$, $B0011$ $B0100$, $B0101$, $B0110$, $B0111$, $B1000$, $B1000$, $B1001$, $B1010$, $B1011$ $B1100$, $B1101$, $B1110$, $B1111$. If we want to represent any larger number than 15, we will need more bits in our binary representation than four. The largest unsigned integer representable using an N bit parallel bus is $2^N - 1$.

2.4.3 Hexadecimal Notation

I am already getting tired of writing all those zeros and ones, and in digital signal processing it is common to represent collections of binary numbers using hexadecimal, or base 16. Because people commonly have ten figures, we have developed naturallly to have symbols for each of the counting numbers between 0 and 9, and think of larger numbers as groups of 10, or groups of 100, and so on. However, suppose we had actually had 16 fingers! We would then have naturally developed separate symbols for the first 16 integers 0 to 15 that are also each representable by four binary bits. Table 2.1 shows the first 16 numbers from the set of positive integers and zero, together with their decimal and hexadecimal representations.

decimal	binary	hexadecimal
0	B0000	0x0
1	B0001	0x1
2	B0010	0x2
3	B0011	0x3
4	B0100	0x4
5	B0101	0x5
6	B0110	0x6
7	B0111	0x7
8	B1000	0x8
9	B1001	0x9
10	B1010	0xa
11	B1011	0xb
12	B1100	0xc
13	B1101	0xd
14	B1110	0xe
15	B1111	0xf

Table 2.1: Decimal, binary and hexadecimal representation of zero and the first fifteen positive integers.

The hexadecimal notation is extensible to numbers represented by many more bits. For example, on computers integers are commonly represented as 32 bit binary numbers, so that the unsigned integer 1 would be represented as B00000000000000000000000000000001. In hexadecimal this is 0x00000001,

which is still a lot of leading zeros, but you can see how the reduction in the number of symbols by a factor of four when using hexadecimal notation instead of binary is attractive.

2.4.4 Binary Counters and Oscillators

While we are looking at this table, it makes obvious another useful feature of binary numbers, and that is the behaviour of the binary digits. Scan down the column of binary numbers and look only at the least significant bit (LSB). Notice that as the count proceeds, the least significant bit oscillates back and forth between 0 and 1 with a period of two counting numbers. Now look at the second-least significant bit. This bit also oscillates back and forth between zero and one, but the period of this oscillation is four counts. The third least significant bit also oscillates periodically with a period of 8 counts, and the most significant bit (MSB) has a period of the full 16 counts. This means that if we can cause a bus to count upwards from zero, each subsequent bit can serve as an oscillator with a period of twice that of the previous bit. It turns out to be very useful indeed to be able to make oscillators with different periods in digital devices, and we shall see the power of this in Chapter 3.4.

2.4.5 Overflows in Binary Arithmetic

We have already realised that we need more than four bits to represent a positive binary number greater than 15. What is the convention for dealing with overflows? What happens when you add 1 to 0xf ? We will operate with the following convention, following digital signal processing conventions: *overflows are disregarded*. If you require more bits than you have designed into your system to get the right answer, it's just tough - you should have used more bits. While this might sound harsh, in fact the convention that overflows are ignored is very useful indeed.

The first reason for this is to do with counters. As alluded to before, if we count up in binary, each successively more significant bit behaves like an oscillator with half the period of the bit to the right. What about the most significant bit? What does it do? It starts out as 0, then half way to the maximum capacity of the bus, it switches to 1. What happens when it reaches full capacity? This is when all the bits are 1. The next number in the sequence, *ignoring the overflow* is all zeros. In other words, the counter just resets to zero, and then keeps counting. A counter that starts at zero, then gets to the maximum number that can be represented with the number of bits, then resets to zero automatically and starts again, is very useful indeed!

There's another very useful application of ignoring overflows. It means that a binary number of N bits containing all 1s is adjacent in the sequence of binary numbers to a binary number containing all 0s. This comes in very handy when representing negative integers with a system called *twos complement* which we will discuss in the next section.

To summarise this section, in a binary representation of numbers, overflows are ignored. If you have 4 bits, then all numbers are represented using 4 bits, and if you end up running off the end, you neglect any overflows and instead your sequence wraps around back to the other end.

2.4.6 Negative Integers

What about if we have negative integers? These are almost always represented using a method called twos complement. Suppose we have 4 bits in our binary representation. So far we have thought of these as representing the unsigned integers between 0 and 15. However, recall that the binary number $B1111$, neglecting overflows, is adjacent to the binary number $B0000$. How about using $B1111$ to represent -1? Following the same logic, $B1110$ is adjacent to $B1111$, so it must be -2. Carrying on with this argument, we reach a binary four-bit twos complement representation for the signed integers in the range $[-8, +7]$,

decimal	binary	hexadecimal
0	B0000	0x0
1	B0001	0x1
2	B0010	0x2
3	B0011	0x3
4	B0100	0x4
5	B0101	0x5
6	B0110	0x6
7	B0111	0x7
-8	B1000	0x8
-7	B1001	0x9
-6	B1010	0xa
-5	B1011	0xb
-4	B1100	0xc
-3	B1101	0xd
-2	B1110	0xe
-1	B1111	0xf

Table 2.2: Decimal, binary and hexadecimal symbols for the twos complement representation of the signed integers in the range -8 to $+7$ inclusive.

You might notice that moving from -1 to -8 is a binary count upwards but with the symbols 1 and 0 reversed. This means that there is a simple logical way to deduce the binary representation of $-x$ from the binary representation of $+x$. This is

$$-x = \neg x + 1. \tag{2.7}$$

Let us check that this works. For example, the binary representation of 3 is $B0011$. We take the bitwise not of this binary sequence to get $B1100$, then we

add one to this to get $B1101$. From Table 2.2 this represent -3. Try a couple of others. Note that it won't work for -8, because +8 cannot be represented in twos complement signed integer arithmetic using only four bits.

2.4.7 Integer addition and multiplication

I am about to teach you how to add and multiply binary numbers by hand as you might have learned to do in school for decimal numbers. Of course, there are calculators that have a binary mode - indeed, yours probably does. You can do your binary calculations there, or check your manual calculations using them. There are also online binary and hexadecimal calculators - see, for example `calculator.net/binary-calculator.html`.

When you were at school you were probably taught about column addition as a way of adding together two numbers each of which had multiple non-zero digits. You basically write out the two numbers one above the other with each succesive digit aligning vertically with the other number to be added. You then add the digits in the corresponding columns starting with the least significant digit and moving across until you have done the most significant one. The interesting case is when the sum of the two digits exceeds the capacity of the symbols available from a single digit. In this case you 'carry' the difference to the next column, usually writing the carried number below the output to remind you to add it on when you get to the next column. If there are any carry digits left at the end, these are added on the left.

It's almost the same thinking about binary addition, except the 'columns' instead of being 1, 10, 100 etc are 1, 2, 4, $\cdots$. In both cases, the M^{th} column represents $(\text{base})^{(\mathrm{M}-1)}$. The only other modification is that you methodically *ignore the overflows*. Let us do some examples. For a start, $2+3$ or $B0010 + B0011$. Figure 2.4 shows how this is carried out using column addition. In

$$\begin{array}{r} 0010+ \\ 0011 \\ \hline 0101 \\ \hline {\scriptstyle 1} \end{array}$$

Figure 2.4: Adding 2 and 3 in binary.

this example, the second column addition results in a carry of $B1$ to the third column, and the third column then just contains that $B1$, so that the overall answer is 1 times 2^2 + 1 times 2^0, or five.

In the next example, shown in Figure 2.5, we carry out $-3+7$. The four bit binary representation of +7 is $B0111$. To obtain the binary representation of -3 we start with +3 which is $B0011$. We first do a bitwise not, leading to $B1100$, then we add a least significant bit, $B0001$, to obtain $B1101$, and that's our four-bit binary representation of -3. Finally, we carry out the column addition. Notice that when we get the answer we truncate at 4 bits, discarding

the overflow 1. This leads to the correct binary representation of the answer, $B0100$, which is $+4$. For a final example of addition, let us consider $2 - 4$. We

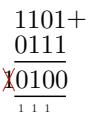


Figure 2.5: Addition of -3 and $+7$ in binary

do this subtraction by considering it as $2+(-4)$, and then working out the twos complement representation of -4. To do this, start with $+4$, which is $B0100$, perform a bitwise not to obtain $B1011$, and add one LSB to obtain $B1100$. The addition is then carried out as shown in Figure 2.6. There turn out to be no carry bits necessary in this sum, and the resulting binary $B1110$ is -2 in decimal, as can be found out by subtracting 1LSB to obtain $B1101$ and doing the bitwise not to obtain $B0010$, which is $+2$, hence the original $B1110$ must be -2. This is also confirmed by Table 2.2.

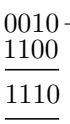


Figure 2.6: The addition of $+2$ and -4.

Next we move on to multiplication, which is carried out using the columnwise long multiplication technique you were taught in school. An example is shown in Figure 2.7. The two inputs are $B1111$, which is the 4-bit twos complement

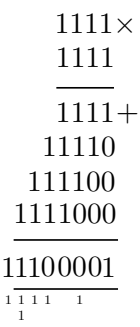


Figure 2.7: Multiplication of the four bit representation of -1 by itself.

representation of -1. We expect to obtain $-1 \times -1 = +1$. We proceed as we do in decimal long multiplication. First the least significant bit of the second

number by all digits of the first one, placing the result on the top output row. Some notation if you haven't encountered it: a 'leading' digit is one to the left of all the digits visible in a set of digits; a 'trailing' digit is one to the right of those digits. Now add a trailing zero to the LSB of the next row, then proceed to multiply the second least significant bit of the second input number by the first row, and place the binary result to the left of the trailing zero. Repeat with the other two bits of the second input number. You will now have four rows containing binary numbers with one more bit in each successive row. Add the four rows up.

When adding the four rows up we will encounter an oddity of addition that isn't often encountered in decimal column addition, although it can also happen. In the later columns, there end up being in some cases four or more 1s to add. In these cases the result has more than two digits in binary. For example, in binary $B1 + B1 + B1 + B1 = B100$. In these cases you end up adding carry digits to other columns than that immediately to the left of the one you are working on. In the case of a $B100$ result in a column addition, you'd need to add a carry 1 to the column two to the left of the current. And so on. In some cases there will end up being multiple carry 1s in a column. This happens in in column 7 of the example.

In fact, you only need to carry out sufficient columns of the multiplication and subsequent addition to get the result for the four least significant bits of the long multiply, because all higher bits are going to be discarded. The result, truncating to four bits, is $B0001$, so that in 4-bit twos complement binary $-1 \times -1 = +1$. So, everything seems consistent.

2.5 Sign-extension

It often happens that you want to multiply two numbers represented in some N-bit binary representation, and it is clear that the result is going to require more than N bits to represent it. When this happens, it is necessary to add more bits to the left of the MSB in each of the input numbers. If you are using an unsigned representation, you just add zeros. However, if you are in a twos complement signed representation, and the input number is negative, then the leftmost bit will be 1. It turns out that if you pad negative numbers, those with a leftmost 1, with 1's to the left, then this maintains the binary number as having the same numerical value in the expanded representation with more bits, just as padding a positive binary number with zeros to the left maintains its value. This procedure of padding based on the value of the most significant bit is called sign extension. The same rule applies to real numbers with fractional parts, which we discuss next.

2.6 Fixed point arithmetic with real numbers

Real numbers that are not integers, can be multiplied and added together using the same rules as described above, with three additional new elements. The representation we arrive at after adding these elements is known as fixed point representation. It is somewhat of a misnomer, because there is no point in a binary sequence, and because the position of 2^0 does in fact move around between columns as we shall see. The terminology makes sense in the context of the popular floating point representation of numbers, where the position of the 2^0 column is recorded in a second binary number, the exponent. Floating point numbers are in practice tricky to manipulate digitally due to the large number of special cases that are necessary to represent all the possible numbers in a range, so a discussion of them is beyond the scope of this book. You can read about floating point representation of numbers in [3], chapter 17.

The first rule concerns how to represent the fractional portion of numbers. We are free to interpret any one of a sequence of bits as representing the number $1 = 2^0$. When representing integers, the rightmost bit has been 1, but if we were to make, say the third-rightmost bit equal to $1 = 2^0$, then the bit to the right of it would be $2^{-1} = 0.5$ and the LSB would be 2^{-2}. In this representation, the smallest positive number that could be represented by the string of binary numbers would be 0.25, Let's say we have six bits in our number system, and let us say we wish to represent positive as well as negative real numbers. The largest positive number is represented by 011111, which is $4+2+1+0.5+0.25 = +7.75$. To find the representation of -7.75, do $\neg B011111 + 1\text{LSB}$, which is $B100000+1\text{LSB} = B100001$. The binary string for -0.25 in this representation is $B111111$, since the smallest negative number is all 1s. If we add this to the representation of $-7.75 = B100001$ and discard the overflow, we obtain $B100000$, so the most negative number representable in six bit twos-complement binary with 2^0 represented by a 1 in the third-from-rightmost bit column is -8.

The second rule concerns how to work out the representation of the answer. In addition, you ensure that the columns where $1 = 2^0$ is written are the same in both inputs, and you follow the same rule as for addition of signed integers, discarding any overflows in the result. For multiplication, you line up the 2^0 columns as before. In the result, the rule is that you add the number of columns to the right of 2^0 in the two inputs, and the sum of these numbers of columns is the number of columns to the right of $2^0 = 1$ in the result. This exact same rule is used to position the decimal point in the answer to the product of two decimal numbers.

2.7 A fixed point multiplication example

Let us say we wanted to compute -4.25^2 in binary. In the same 6-bit fixed point representation discussed in Section 2.6, $+4.25$ is $B010001$. We find -4.25 by doing $\neg(B010001) + 1\text{LSB}$, which is $B101110 + 1\text{LSB} = B101111$. Next,

we recognise that we will probably need some more bits, so let's just add another six bits to the right by sign extension, so that we end up with a 12 bit representation of -4.25, which is $B111111101111$ = 0xfef. Not wanting to do a tortuous long multiplication, I use a web-based binary calculator to do the heavy lifting, taking only the 12 least significant bits of the result, and I obtain $B000100100001$ = 0x121. Following the rule that you add the number of digits to the right of 1 in each of the inputs, there are $2 + 2 = 4$ digits to the right of 1 in this number, so that the LSB represents $2^{-4} = 0.0625$, which is 1/16. The digits other two non-zero digits are 2 and 16, so the answer is $-4.25^2 = 16 + 2 + 0.0625 = +18.0625$, which is correct.

2.8 Summary of Chapter 1

In this chapter we have learned about the core logical operations available on digital platforms, and how the fundamental operations of arithmetic can be built up. We have learned how to represent positive and negative real numbers using fixed point twos complement representation. This will later on allow us to use our digital platforms to do digital signal processing, which is one of the most important applications of digital signal processing units. Hopefully you see how, in principle, complex numerical calculations, at least those involving addition and multiplication can be based on simple logical operations on strings of binary bits. You can now begin to appreciate what is inside a computer. However, there are important fundamental elements still to introduce. Fortunately, there are not that many more - in fact, we will meet them both in Chapter 3.4.

Chapter 3

Sequential Circuits

3.1 Motivation

In chapter 2.4 we investigated logic gates and more general devices possessing truth tables. We can imagine that a great many problems can be solved purely using logic; given enough input data bits, and enough logic gates, a configuration of logic gates could be found that could produce a correct result. However, there certainly exist questions that cannot be answered by logic alone. For example, how long is it between flashes of that light? That one can't be done with a pure logic circuit because logic alone cannot wait or measure time. Or how about solving the equation $\tan(x) = x$?. Or how about any circuit that executes tasks in responds to commands issued by humans?

In these cases, we are going to need more elements in our armoury than just logic gates. The thing that these examples have in common is that solving them requires a circuit that can work with intermediate results. In the first case, if you have an oscillator, you can detect its pulses, and each time you detect a pulse, you can increment a counter, so long as you have some way of memorising where the count has got to. You can then gate the counter to start when one light pulse is detected and stop on the detection of the next one. In the second case, there are many numerical methods of solving nonlinear equations like $\tan(x) = x$, such as the Newton Raphson method. These methods rely on starting at some initial guess, then using an iterative procedure to converge on the right answer, within some required tolerance. But in order to work, these techniques need to store intermediate results, and logic gates cannot store anything. In the final case, the circuit has to ask for a command, wait for it to be entered, memorise it, act on it, and then perhaps ask for another command. There are plenty of intermediate results and some waiting to do.

3.2 Trouble with logic

As a first step in seeing how we might introduce some of the necessary elements into our circuit, let's consider a troublesome logical circuit, shown in various representations in Figure 3.1.

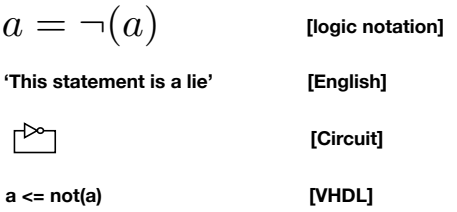


Figure 3.1: An internally inconsistent circuit in various representations.

This is the kind of statement that Kurt Gödel was worried about in connection with the programme of building a complete and self-consistent logical framework for mathematics [1, 2]. You can immediately see the problem with it - if it is true that the statement is a lie, then it must be false, and there is a contradiction.

3.3 Oscillators

In fact, this statement is rather close to the science of oscillators, as you can see by inserting the simple words, 'a moment ago'. If you say 'a moment ago, this statement was a lie', then it's fine because a moment ago it can have been a lie, so it is now a true statement that it was a lie before. A moment from now, it will be false that a moment ago the statement was a lie, actually it was true then, and so forth. In science terms, if we insert a time delay in the loop with the not gate, there is a consistent signal that can exist indefinitely in the circuit. In Figure 3.2 a 1 ns time delay has been inserted, resulting in a 2 ns period square wave oscillator.

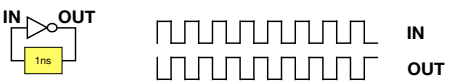


Figure 3.2: An oscillator made from a not gate and a time delay.

All oscillators can be modelled as closed loops with waves travelling around them satisfying Nyquist's criterion, which is that if you break the loop and inject an periodic signal into the broken end in a forwards direction, if the signal arriving at the other broken end behind you with the same phase and amplitude as the injected signal, then the circuit, when reconnected, will support that oscillation indefinitely. In this case it works because the 1 ns delay is half a

period, so it flips the sign of the square wave, and the not gate flips it back again, so the round trip phase shift is a full cycle.

Oscillators are going to play an important role in digital circuits, but they are difficult to fabricate on conventional semiconductor wafers, so commonly the oscillator is an external device mounted next to the digital processor, such as a crystal or MEMs oscillator. These cost litle and have a frequency stability of tens of parts per million. If you need a far more stable oscillator, commercial reference oscillators based on atomic transitions in caesium or rubidium routinely achieve frequency stabilities of parts in 10^{11}. Intermediate performance is achieved by quartz oscillators phase locked to the 1 pulse-per-second carrier transmitted by the GPS satellite system. Digital signal processing circuits like FPGAs also often incorporate phase locked loops, which can generate an internal square wave oscillator on the chip which is phase locked to the internal reference, even if the required frequency isn't the same as that of the reference.

3.4 Registers

The value of oscillators is that they enable timing of processes, because the regular oscillation of an incoming square wave from the oscillator can be used to drive a counter. However, counters require some form of memory to store the current value that the count has reached. Logic gates cannot be made to store anything on their own, so a new type of component is needed. The simplest form of storage available in a logic circuit is a register, or bank of registers, which can store a binary number for some defined period of time. The hardware circuit that enables registers to store information on an FPGA is called a D flip-flop.

A diagram of a D flip-flop is shown in Figure 3.3.

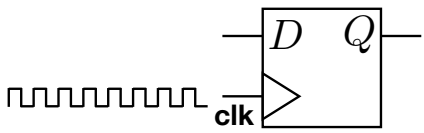


Figure 3.3: Standard schematic representation of a D flip-flop circuit.

The flip-flop has three ports, two input and one output. One of the input ports is dedicated to an oscillator input signal. Often these signals are referred to as clocks, and `clk` will often be the name of the incoming signal, but really it is just a square wave oscillator. The other input signal I will call D, and the output signal is Q. For now let us think of D and Q as single bit signals. If we wish to store more than one bit, a bank of D flip-flops is connected in parallel to the same oscillator, and one flip-flop handles each bit.

The function of the flip-flop can be thought of classically as follows. When the rising edge of the oscillator input is detected, the digital signal present at

D at that time is copied to Q, where it remains persistent until the next rising edge of the clock input, when the cycle repeats. The process of reading the signal at D and setting the Q output equal to that signal is called a register transfer. Because the signal is held at the Q output for a clock period, this provides a storage mechanism for digital data.

Chapter 4

Serial Buses

Chapter 5

Data Converters

Chapter 6

State Machines

Chapter 7

Instruction Sets and Computers

Chapter 8

Programming in C

Chapter 9

Buses again

Chapter 10

A Command Line Interpreter

Chapter 11

Qubits and Quantum Computing

Appendices

Appendix A

Hardware for the Sheffield course

A.1 Introduction

We will develop our hardware and software in an integrated develpment environment called Vivado/Vitis, and deploy to a develpment board. The development environment will be running on your loaned Lenovo ThinkPad i7 PC and the test hardware consists of a BASYS3 development board. These are connected together with a USB to micro-USB cable.

The Lenovo ThinkPad runs Ubuntu, a dialect of the LINUX operating system, which is an example of the UNIX family of operating systems that has been around since the mid 1970s. The choice of LINUX rather than Windows is for many reasons, but here I highlight one of them. Microsoft has a habit of inducing Windows to install updates at random times. Experience taught us that too often students were sitting in the lab waiting for a windows update to complete, and there was no obvious way to stop this happening. In UNIX, updates are very much under the control of the administrator of the system. In the case of your laptops this is Mitch, and we only update in the off-season! There are other reasons but this is the main one.

The PCs are quite nice and in particular quite fast because they have several intel i7 processor cores, and the software we are running to program the BASYS3 boards can make use of these in parallel. They do have some annoyances. The main one is the rather small screen. If at home you have an HDMI monitor and cable, then you can use an ordinary HDMI cable to connect the laptop to an external screen, which makes using the computers more comfortable. You can also plug in your own mouse via USB. The other annoyance is the 'PgUp' and 'PgDn' (page up and page down) buttons being located right next to the cluster of arrow buttons on the keyboard, which means that sometimes you find yourself taking forays into bits of your code you didn't intend to visit.

There is nothing that can be done about this; I only flag it in case you don't understand what is happening, and advise you to be conscious of exactly where your finger is when trying to use the arrow keys.

The BASYS3 boards are designed for teaching purposes are are therefore easy to use relative to other development boards we have tried. However, the hardware on these boards is quite cheap, and occasionally fails. Please do not leave the connecting cable plugged in to the USB port on the board when you put it in your bag to carry it around! This is a great way to destroy the rather fragile USB connector on the board. The other failure point is the 16 LEDs and switches along the bottom edge of the board, below the BASYS3 logo. The switches wear out quite quickly, and the LEDs (rather surprisingly) also fail quite often. In todays project we will check all the switches and LEDs on your boards.

You will all be learning to use LINUX, which many of you will never have met. This should not be a handicap on the course. An important difference between LINUX and Windows is that in LINUX the command line interface, accessed by opening the terminal program, is a very powerful environment. Windows users mostly abandoned the DOS prompt and direct issue of commands to the operating system many years ago, although amongst engineers, scientists and developers DOS still does get used, in spite of its many shortcomings. In LINUX it is common to make use of commands issued directly into a terminal for tasks like creating directories, moving files, and communicating with other machines on the internet using various communication protocols (ssh, sftp, git and other things that we shall learn about). You'll find yourself using the terminal window to get things done. This is a useful transferable skill into many lines of work and post degree research.

The laptops have all been set up with the same environment and software. You will find that you do not have permission to install your own software on the system or make major changes to the way your computer is set up. It is not in your interest to make major changes to the system - for example trying to repartition the disk and install a windows operating system on a separate partition. We therefore make a rule against attempting such hacks, and Mitch will not be happy, and will tell me (and I will be even less happy) if it becomes clear that somebody has broken this rule.

One final point. This is a course that teaches practical skills. As with other practical skills – riding a bike, playing a musical instrument, etc, real facility comes with practice. The labs give you a total of about 33 hours of time officially dedicated to this. However, because I am lending you the hardware, you can also practice outside labs, and are strongly encouraged to do so. In about 5 years of teaching the course, several of our students have used skills gained on the course to get very good jobs in companies. I am very much hoping to have one of them talk to you all about their experience in industry using things that we taught on this course. The course is largely what you make of it. If you work hard, you are gaining some genuinely useful skills for your CV and future career, if that is what you decide you want.

A.2 Brief Tour of the Laptop

The first job is to turn on the laptop, and log in under the physics user. Your username will remain 'physics' for the duration of the course.

The first job is to connect the laptop to the campus wifi. It will not connect by default – Mitch has wiped the hard drive and it is essentially equivalent to a fresh-out-of-the-box system.

With the exception of the very first instruction, the instructions below are identical to the University's seen at `https://students.sheffield.ac.uk/it-services/wireless/connect#linux`. These are to be followed with one exception. The root certificate download can be found on the laptop already at `/etc/ssl/certs/QuoVadisOVRootCertificate.crt` :-

1. Click on the downwards arrow in the top right hand corner of the desktop and select 'Wifi Not Connected' then 'Select Network', then select eduroam, then press the 'Connect' button at the bottom of the selection window.

2. *NOTE - you do not have to follow this instruction because the file is already on your machine at the path above.* Download the QuoVadisOVRootCertificate.crt certificate to your machine from `https://www.sheffield.ac.uk/polopoly_fs/1.950471!/file/QuoVadisOVRootCertificate.crt`.

3. When in range of eduroam click on it to enter settings.

4. Use the following settings:

   - Wireless Security: WPA2 Enterprise
   - Authentication: PEAP
   - CA Certificate: Browse to where you saved the certificate and select
   - Inner Authentication: MSCHAPv2
   - Username: Your UoS username followed by @sheffield.ac.uk
   - Password: Your UoS password

   You should then be able to connect to eduroam

Figure A.1 shows the Ubuntu window manager you should see when you have logged in. Down the left hand side there are a set of icons in a toolbar that run different pieces of software. Yours are not quite the same as mine as I have added a few extras. On your machines, there are seven that will prove useful during the course. By hovering with your mouse pointer over the icon you can see what these applications are. Locate the following applications:

1. Firefox Web Browser

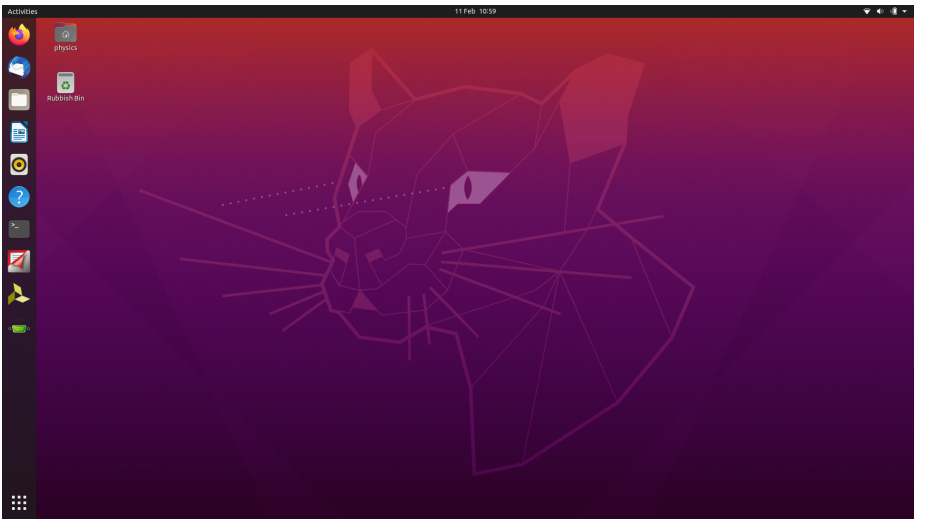


Figure A.1: A blank screen upon logging in as physics to Ubuntu

2. Files
3. Terminal
4. Serial port terminal
5. Vivado 2019.2
6. Xilinx Vitis IDE 2019.2
7. Documentation Navigator

You can single-left-click on any of these items to run the corresponding applications. If you run something and want to stop, just as in Windows, use the red 'x' in the top right hand corner of the window. Most windows can be resized by dragging on the corners, minimised by clicking on the underscore also in the top right corner, and expanded to full-screen mode by clicking on the rectangle in the same place. If you minimise a window belonging to one of the sidebar applications, it can be recovered by clicking again on the icon. You can also re-open a minimised icon by clicking on 'Activities' in the top left hand corner. The full set of installed applications can be seen by clicking on 'nine dots' icon in the lower left corner. You can drag across the mouse pad to go up and down the full set of icons. Clicking on it again takes you back. If you are alone and want some background music or talk, the 'RythmnBox' application plays podcasts.

The two icons in the top left of the main window are short-cuts to the Files application running on your home directory and the contents of the rubbish bin

Firefox should need no explanation. The 'Files' application is like the Windows file manager, and it allows you to navigate through the hierachy of folders on the Ubuntu machine. If you right click on a file you are given an option to move to the recycle bin, just as in Windows. In fact, Ubuntu is trying to emulate the Windows operating system. The big difference between Linux and Windows is the use of the Terminal application. Windows has the DOS prompt, but DOS is so arcane that few but the brave use it very much. In Linux the Terminal 'shell' is the BASH shell, which has quite a lot of functionality and is quite sophisticated. However, you shouldn't actually need to learn too much UNIX/Linux to do this course. If you are interested, there are lots of tutorials on line in learning the BASH shell in UNIX/Linux. The serial port terminal will be useful later in the course when we are communicating with microprocessor cores on the BASYS3 board. Vivado and Vitis are the two graphical user interfaces that we will use to develop our applications. We will spend most of our time using Vivado and Vitis.

Before you can use Firefox you will need to connect your laptop to a wireless network. The icon to do this is in the top right hand corner of the screen and looks like a segment of fruit. If you click on this icon it will show you a list of available wireless networks - you should check and see that there is one that you can log in to. After logging in, the segment will be partially filled in, indicating the quality of your connection.

You should be able to use Firefox to log on to MUSE and from there with the google mail tool and Blackboard you should be able to use your laptop to take part in the lab projects without need for your own personal devices. The camera built in to the laptop and the built in microphone both work with blackboard and we will need to communicate with each other freely to make a success of the course, within the limits imposed by bandwidth.

To shut the laptop down, the small downwards arrow in the very top right pulls down a menu from which you can select the Power Off/Log Out option, and there you can do all the usual things - turn off, restart, log out, etc. In these things Ubuntu again behaves much like other popular operating systems.

There are also several other equipment items. These are, all told

1. Cardboard box and packing material (please preserve in good condition)
2. Laptop with power supply (two connected cables)
3. Gel filled protective laptop case
4. USB to micro-usb adaptor cable
5. Basys3 FPGA development board
6. PMODDAC2 two channel digital-to-analog converter (DAC) - this is small!
7. some cables
8. a crystal earpiece

Obviously try not to lose anything and tell me if you do, or if anything gets broken. Try NEVER to keep drinks anywhere they can spill on the laptop. Especially on the table directly next to them. Think of them as lab equipment. You wouldn't drink in a lab, so don't keep drinks in the vicinity of this machinery.

One final thing about the laptops. If you want to dump the whole screen to a picture file, then at any time you can just press the PrtScr button which is on your keyboard two keys to the right of the space bar, and a '.png' format bitmap of the entire screen is written to the Pictures subfolder of your home directory. Try this, and then use the Files application to go to this directory, double clicking on the icon to view it. If you just want to print one window, make sure this window is selected, then hold down the Alt key and again press PrtScr (Alt-PrtScr), and a '.png' file just of the highlighted window is put in the same place. This could be useful if something isn't working right and you want to send myself or Mitch graphics that are useful in explaining your problem.

Appendix B

Lab Exercise 1 - Logic in VHDL

B.1 Creating a new Vivado project

VHDL is a hardware description language for expressing the abstract functionality of digital circuits. VHDL can be compiled into an actual wiring diagram for a specific hardware device, which in this course will be a field programmable gate array, or FPGA. The FPGA we will be using in this course is from the Artix7 series made by Xilinx. Specifically the part number of the Artix7 FPGA on your BASYS3 boards is **xc7a35tcpg236-1**. This is a part number that you will need every week in the lab.

The compiler that tries to turn your VHDL code into a hardware specification is actually a command line tool, but like many such tools it is usually used by way of a graphical user interface, sometimes called a software development kit (SDK). SDKs are all over the engineering world. In this course we will be using two SDKs, Vivado and Vitis. Vivado is the SDK for building the hardware, and when the hardware incorporates a microprocessor, Vitis is the SDK that we will use to program that microprocessor in the C language.

A few practical details. You may have noticed that the chapters of the book correspond to the material taught in lectures, and the appendices correspond to the labs. This is deliberate. The lecture material I consider to be implementation independent; it would be the same if we were using a different HDL made by a different company for different chips, or a different SDK. The appendices are specific to our hardware. The idea is that I may have to re-write the appendices many times, but I am hoping not to have to do too much rewriting of the chapter materials.

So, let's get started. For our first project we will build hardware that allows us to check that all the switches and LEDs on the BASYS3 board are working. There are two reasons for doing this. First, they might not be. The components

of the BASYS3 are fairly cheap and it is not uncommon for the switches in particular to wear out and stop working. The second reason is that this is one of the simplest projects we can do, so it is a good one to start with.

You will need to have read Appendix A before starting these instructions. Start up Vivado using the left hand tool shortcut. After some delay the window shown in Figure B.1 will appear. Note that some people become impatient and click the icon twice in which case two copies of Vivado will open. Just close one of them if this happens and use the other one.

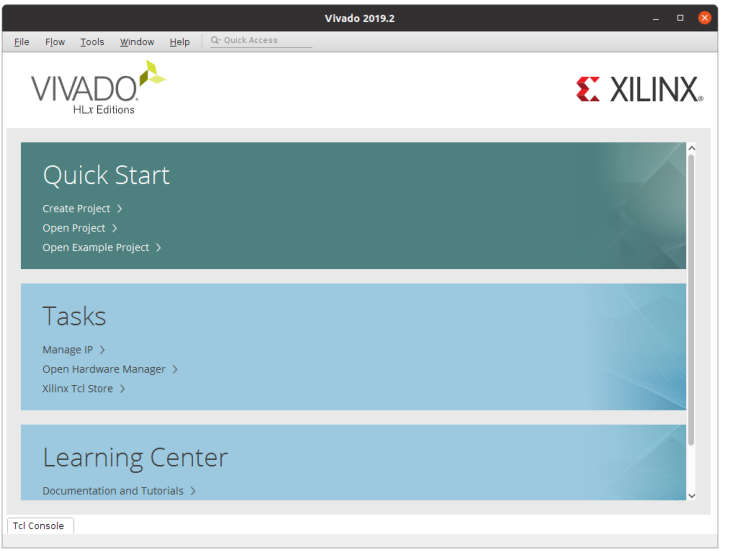


Figure B.1: Startup screen for the Vivado software development kit

You'll want to create a project, so left-click (henceforth whenever I ask you to click without specifying which button, I mean left-click) on 'Create Project'. This brings up a subwindow that says 'Create a New Vivado Project'. On the first page you are asked to give the project a name, and to specify its location. Let's call the project `check_leds_and_switches`. It is an old habit of mine to use underscores instead of spaces in file and directory names. I recommend that you pick up this habit. Unix-like computers tend to place special meaning on whitespace characters like space, tab, etc and things can unexpectedly fail to work if you have files and folders that have spaces in their names. Change the project name by clicking in the box and typing the new name. For the project location, the default is your home directory but we don't want this directory to become too cluttered, so let us make a new place to keep our Vivado projects. Click on the small rectangle to the right of the Project location box, with three dots in it. This brings up a 'Choose project location' window, shown in Figure B.2.

The default location is your home directory. In this window, we can cre-

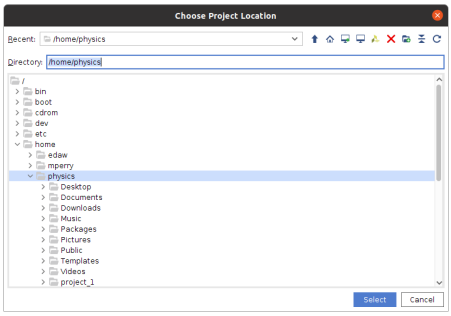


Figure B.2: Window to choose project location. The 'create folder' icon is the third from the right of the small icons in the top right hand corner of the window.

ate a sub-directory of the currently selected directory (you'll see that physics, your home directory, is highlighted), by clicking the third icon from the right in the row of small icons in the top right hand corner of this window. Type `vivado_projects` in this window and you will see a new folder with this name appears and becomes highlighted. Click Select. This returns you to the 'New Project' window, and you will see that the project location is now `/home/physics/vivado_projects`. Because you have the box checked that says 'Create project sub-directory', Vivado will create a further sub-directory called `check_leds_and_switches` within the Vivado projects directory. By using a new directory for every Vivado project we create, we can keep ourselves organised as we do more work. You are now done picking a project name and directory, so you can click the blue Next button ato the bottom, or use the shortcut alt-n. As an aside, in Vivado and many other software packages having graphical-user-interfaces (GUIs) every button that has a label with the first capital letter underlined has a keyboard shortcut of alt-(that key). No need to use shift because alt-n is the same as alt-N by convention.

On the second page displaying in this window you keep the default which is 'RTL Project'.

On the third page you are asked to add sources. The sources are VHDL source files, and we are going to build ours from scratch, so you don't need to add any at this stage. So, Next (or alt-n) again.

This brings you to the fourth page, where you are asked to add constraints. A constraint is a file that we will discuss further. It's basic functionality is to associate pins on the FPGA chip with ports, which are wires into and out of the circuit you are going to define in your VHDL code. We will always start with a standard constraint file that is supplied by Digilent, the makers of the BASYS3 board that we have lent you. You will find this file in your home directory, and it is called `Basys-3-Master.xdc`. You can add this constraint file now, or later on once the project is open. Let's add it now. Click on the '+' icon close to

the top left of the window you are in, and select 'Add Files...'. This will bring up a browser window, which is shown in Figure B.3.

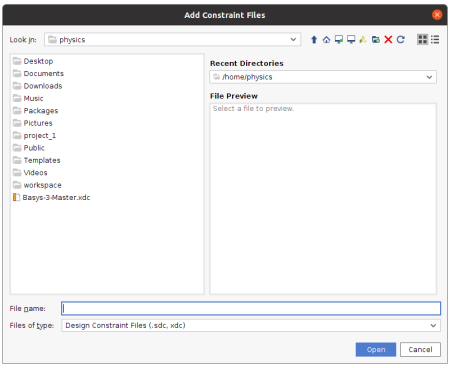


Figure B.3: The browser window for selecting a constraint file.

The browser starts in your home directory by default, which is the location of the constraint file. Click on it, and the contents at the head of the file should appear in the 'File Preview' pane to the right, and the file name should also appear in the box second from bottom of the window. Click OK, or just hit the Return key, and this should close the browser window. Finally, be sure to check the box in the bottom left hand corner that says 'Copy constraints files into project'. This causes Vivado to duplicate the constraints file into a subdirectory of your project folder, so that you can re-use the generic one in your home directory for future projects. You don't have any more constraint files to add, so hit Next (or alt-n).

The next part of project specification is to select the Default Part, or sometimes you will instead want to select the Default Board. For today, we're just going to select the part. You'll see a long list of parts appear in a table in the lower part of this window. The list is so long that scrolling up and down it is a chore. The parts on this list are FPGA devices that is supported under your Vivado software license. One of these devices is the FPGA on the BASYS3 board, and as already mentioned the part number is **xc7a35tcpg236-1**. Click in the Search window, (or alt-S) and type in the part number, and the list of parts in the table will narrow to just 1. Then, click on the one remaining row, and this part should be highlighted in blue. Note that in this table various statistics about the chip are given. It has 236 pins for input/output, 106 IOBs (input output buffers), 20800 LUT (look-up table) elements, 41600 FlipFlops, 50 block RAMs, 90 DSPs (digital signal processors), and you can use the horizontal scroll bar at the bottom to move across the table to see that it has 2 Gigabit transceivers and 2 GTPE2 transceivers, one PCIe (pci express module), 5 MMCM (multi-mode clock module), a minimum operating voltage of 0.95V, and a maximum operating voltage of 1.05V. Click next (or alt-n). This completes the specification of the project. Check that a new RTL project with

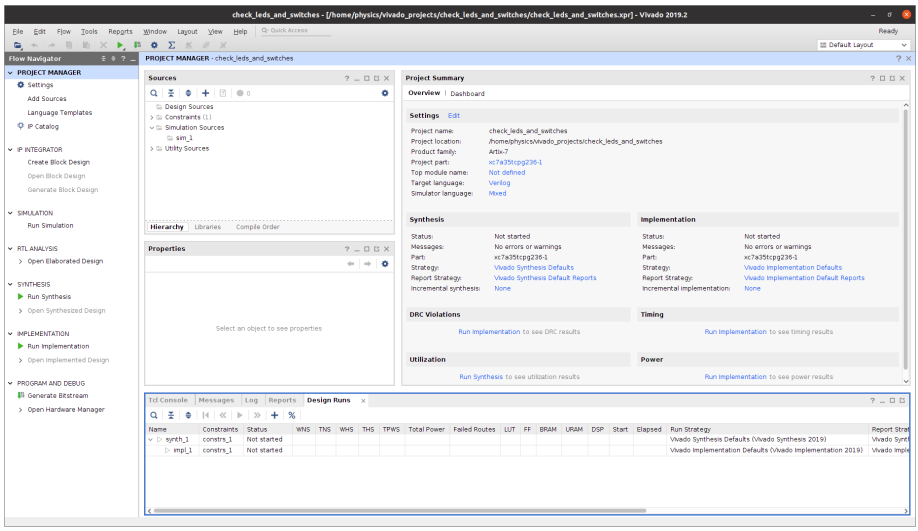


Figure B.4: A new Vivado project window on start-up.

the correct name of `check_leds_and_switches` will be created. Check that the default part is the one in bold above, of the Artix-7 family, package cpg236, and speed grade -1. Then, alt-f or click Finish. A small window indicating that some work is being done will appear and after that the Vivado window will change to the top window of the project. It might be best to use the 'square' icon in the top right of the window to maximise it to full screen. The window should appear as shown in Figure B.4.

B.2 Organisation of the Vivado project window

The general layout is that there is a left hand 'Flow Navigator' toolbar. This bar contains commmands, or directories (with a '>' on the left) containing more commands. At the top of this 'Flow Navigator' on the right there is an underscore, which when you click on it causes the flow navigator to disappear. If this happens, you can make it reappear by clicking on the tall slim rectangle all the way on the left with 'FLOW NAVIGATOR' written in it sideways. This should bring it back. The other three icons expand and collapse the contents of the 'Flow Navigator', and give you some text help on what this navigator is for.

The rest of the window consists of a set of panes, each of which have tabs in them. The biggest pane currently just has a 'Project Summary' in it, which we will come to in a minute. To its left there are two smaller panes stacked vertically on top of each other. The top one says 'Sources' and the bottom one says 'Properties'. Each of these three panes has a cross in the top right

hand corner. If you click on this, the pane disappears. If this happens, you can make it reappear by clicking on 'Window' in the top left of the Vivado project window and selecting the name of the pane. Try it with one of them.

At the bottom of the window is a long but squat pane with five tabs in it labelled 'Tcl Console', 'Messages', 'Log', 'Reports' and 'Design Runs'. Explore by clicking on them one-by-one.

'Tcl Console' contains an enticing window with the words 'Type a Tcl command here'. This is what is visible of the command line program underlying the graphical user interface (GUI). If you were an expert, you could run the entire SDK without ever using a mouse from this window. Indeed, many professional engineers probably do exactly that. As a point of interest, Tcl was a scripting language that was very popular in the 1990s and early 2000s, then fell out of favour, but not before the entire engineering community had bought heavily into it, rather in the same way that astronomers have gone a bundle for PYTHON. The Tcl language came with some rather nice GUI widgets so the package was called Tkl/Tk, and many GUIs, including earlier versions of Vivado, were built out of Tk widgets. So, that's why it's called a Tcl command. You'll never need it but it is nice to understand why things are the way they are. If you are curious, when we get to actually running commands, you can open up this Tcl Console window and view the actual command line tools that you have invoked by clicking on the symbols. This is why I know that you could run the whole program from the command line - all the GUI does is generate Tcl commands corresponding to the mouse clicks it detects, and then displays the results in a graphical format. It is literally a graphical interface to a command line, or shell, based tool.

The 'Messages' tab is where you go to look for errors, warnings and information that are fed back to you after you run a command. You'll see that there are currently three messages of class info - if you uncheck the box they disappear, and if you recheck the box they come back. This is useful because if you run something and it goes wrong there may be 107 warnings of no interest and a single error that needs rectifying. By unchecking the 'warnings' and 'info' boxes you can view the error and do something about it.

'Log' shows all the text returned by all the commands you run in a single window. It is probably not a very efficient way of getting this information - you are better off with the 'Messages' tab.

'Reports' is where you can view technical data about the products resulting from your design. Things like what fraction of the hardware resources on the chip are being used appear here.

'Design Runs' is where you can view progress when you launch a command that takes a few minutes to complete. One of the reasons that I lend out these laptops rather than having students install the software on their own machines is that Vivado and its underlying tools are actually very compute-intensive. These laptops were the best hard disk / processor / core combination I could find for the money. They are quite fast machines, but you'll still find yourself waiting for things, especially when the projects get more elaborate later in the

course. You're probably best off leaving the 'Design Runs' tab open for now.

Each of the panes has a small icon in the top right hand corner to the left of the cross icon that hides the pane, which looks like a rectangle with an arrow in it. If you click this, it pulls the pane out into a separate window that you can make very large. Try this with one of them. If you want this window to go back into the Vivado project window, then next to the 'X', the icon in this window now looks like a right angle with an arrow pointing southwest away from it. Click this and the window turns back into a pane. The next icon along in each pane is an empty rectangle. If you click this, that pane expands to fill a much bigger space, obscuring the other panes, but not the Flow Navigator. This is another way to concentrate on what is in one of the panes at the expense of the others. To reverse this action, click on the 'double square' icon that appears in the same place as the single square when you maximise the pane in this way. This will restore the pane you maximised to its previous size. There is also an '_' icon which turns any pane into a tall thin rectangle at the edge, which you can retrieve by clicking on that rectangle. Try that too. Finally, each pane contains a '?' icon, which expands to some help on what that pane is for. In previous versions of the software this wasn't actually any use at all. However, it has improved, so do have a read of some of this help text.

All in all, I recommend that you spend some time exploring this SDK and how it is organised. Because you are all working remotely, if something wierd happens in the GUI, it will be far, far easier if you can make things right for yourself and not have to ask us to do it for you.

B.3 Which hardware description language ?

There are two hardware description languages supported in Vivado - VHDL and Verilog. Verilog is probably used more extensively in professional engineering, but I think VHDL is a better language for first time hardware developers, because it makes very explicit certain concepts that are fundamental to the architecture of computers, particularly registers, which we will discuss next week. `Verilog` tries to look more like an ordinary high-level programming language like C, but I think that for a student this is actually counter-productive. So, we will use VHDL. If you look in your Project Summary window, however, you'll see that it says, on the left, amongst many other things, that the 'Target language' is Verilog. This strange choice of words means - 'I am expecting you to code in Verilog here'. We need to fix this or there's going to be trouble. So, click on the blue 'Edit' in this pane, click on the arrow next to Verilog in the window that appears, and select VHDL instead, followed by OK. Now you are set up to compile VHDL code in Vivado.

As an aside, after I started learning about VHDL for myself about a decade ago, I realised that it reminded me of a work experience placement I had when I was at school, at a company called Lucas Aerospace. The engineers there sent me off to write code for a PAL, which stands for programmable array of logic,

in a language called ABEL. I ended up concluding that the PAL the engineer had selected didn't have enough gates on it to implement the functionality that the engineer wanted, so it wasn't a roaring success. Well, it turns out that a few years later Xilinx bought ABEL, and used it as the precursor to developing VHDL. So, without realising it, I have been programming in hardware description languages since I was fifteen! And, it turns out that ABEL is still in use, in some quarters, 35 years later. All that happened is the devices got bigger - PALS became CPLDs (complex programmable logic devices) which then got even larger and became FPGAs, and now we have SoCs, which are systems-on-a-chip. What's next? Maybe UoS, which instead of University of Sheffield could be Universe-on-Silicon ?

B.4 A first look at the constraint file

In the 'Sources' pane, there is a folder icon with Constraints (1) written next to it. Click on the '>' next to that folder. The folder hierachy should expand and the file Basys-3-Master.xdc that you incorporated as a constraint file should appear. Double click on this file. It opens in the largest pane (which I'll just call the main pane from now on) as a new tab. There should now be two tabs in the main Pane, the constraint file and the 'Project Summary' tab that was there before. Use the mouse to toggle between views of the two tabs. Go back to the constraints tab. Scroll up and down inside the file and take a look at its contents.

First, note that every single entry in this file, with the exception of a couple at the end, starts with the '#' character. This is the comment character in constraint files, so before we start doing things every line in this file is commented out. However, if you look down this file whilst also looking at the board you'll begin to notice a correspondence between the constraint file entries and the hardware. Here are some observations I would make.

- The file entries are divided into blocks with space lines between the blocks.
- The first block below the four comment lines at the top is to do with a Clock signal. We won't worry about clocks this lab.
- After this there is a block entitled Switches. The lines in this block are in pairs. The first line of each pair contains a letter-number combination. In the first pair it is V17, in the second it is V16, etc.
- There are 16 of these pairs in the switches block.
- There are 16 switches on the board. If you look at the board, can you find the letters V17, V16, and the other similar pairs printed on the board in the vicinity of the switches?
- Moving past the switches block, the next block says LEDs. This block too has pairs of lines, and there are 16 pairs in all.

- Like the switches, the LED lines are associated with alphanumeric codes, like U16 and E19. Look at the board and locate the LEDs, close to the switches, that have these alphanumeric codes next to them.

We're actually only going to use the LEDs and switches today, so you don't need to worry about the remainder of the constraint file. However, if you want to take a quick look, the remaining blocks also correspond to physical devices connected to the FPGA. For example, the 3 blocks below the title 7 segment display, with 7 pairs of lines then a space then a pair of lines on its own, then four more pairs of lines, are all related to the four digit alphanumeric LED display on the board. The five pairs labelled Buttons correspond to the cluster of five pushbuttons - notice the names of the ports are btnC (central), btnU (up), btnL (left) ...etc, corresponding to the physical arrangement of the buttons on the board. The four groups labelled Pmod Header JA, JB, JC, and JXADC have to do with the four 12-pin expansion connectors, or PMOD connectors, on the two short sides of the board. The section labelled VGA Connector corresponds to the pins of the micro-D connector on the back side, which is designed to control an analog RGB monitor. The remaining blocks are associated with an RS232 serial interface that can be connected to the micro-USB connector next to the on switch, the full sized USB port in the top right corner, and some SPI flash memory connected to the FPGA.

B.5 Modifying the constraint file

Remember from the first lecture (see Chapter 2) when I discussed buses? In this constraint file, the LEDs and switches are each configured as 16 bit wide buses. You have a bank of 16 switches and a bank of 16 LEDs. Each LED and each switch is wired to a different physical pin on the FPGA hardware. The FPGAs have a grid of pins that cover the whole back side of the chip, except for a square in the centre that for this chip has no pins. The pins are arranged in a square grid. These alphanumeric codes are literally the grid references of the pins that are connected to the devices in question. After these alphanumeric codes are square brackets containing syntax like 'get ports {sw[0]}'. A port is VHDL syntax for the name of an input and output in the hardware description language. Where an input or an output is part of a bus, the square brackets specify which bit this port is. So, in the case of this port, it's the least significant bit, bit 0, of the 16 bit bus connected to the switches. Looking down the rest of the switches, you can see that each pair of lines refers to a different bit in the bus, from the LSB at the top to the MSB at the bottom. The second line in each pair specifies what type of voltage standard the hardware is - in this case LVCMOS33 means low voltage CMOS 3.3V. These lines are used when Vivado is doing timing simulations on the signals propagating across the FPGA.

You are going to need to uncomment all the lines in the switch and LED blocks. To do this, use the left mouse button to highlight the entire block, or as much of it as you can at once. The lines you highlight will turn blue. At the

top of the Basys-3-Master.xdc tab, there are 11 small icons. The third from the right is '//'. If you click on this it will uncomment the highlighted lines. They should change from grey to black and blue, and the comment characters at the beginning of the lines should disappear. Carry on with this until you have uncommented all the lines in the switch and LED blocks, except the first line in each block, where the double comment character is a way of telling you that this is just a title.

After uncommenting all the switch and LED blocks, you'll need to save changes. Notice that after you started altering the constraint file, an asterisk (star) appeared next to the file name at the top of the tab. This means the file has unsaved changes. To save these changes, hold down the Ctrl key and press s (ctrl-s). The asterisk should disappear when your changes are saved. Your constraint file is now ready to use.

B.6 Creating a source file

Our next job is to create our VHDL source file. To do this look in the 'Flow Navigator' sidebar and click 'Add Sources'. A window appears, with three options, the middle of which is 'Add or create design sources'. Click the button next to this option to select and then click Next. The ensuing window looks a lot like the one where you added the constraint file earlier. In fact, it's the same window, only this time, we don't want to add an existing file, we want to create one from scratch. To do this, click the plus sign towards the top left corner and select 'Create File'. A sub-window appears. The file type should be VHDL, but you need to give the file a name. Click in the box next to File name. It can be anything you want as long as it ends in .vhd, only don't include any spaces in the name. How about 'switch_to_led.vhd'. That is what I called mine, anyway. Hit OK. This should return you to the 'Add Sources' window, and your file should be visible as shown in Figure B.5

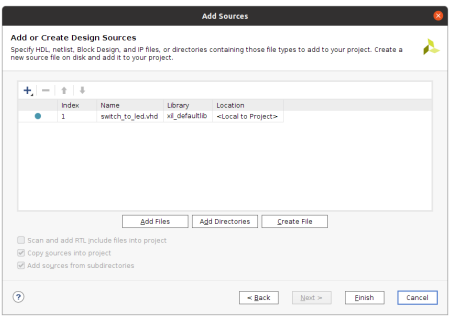


Figure B.5: The 'Add Sources' window after you have named your source file.

You can now click Finish, only it's not over yet. A new window appears called 'Define Module'. You don't actually have to use this, but it is quite useful

to do so as it gets your source code off to a good start. In this window there is a table called I/O Port definitions. In our module, there will be an input port bus for the signals from the 16 switches and an output port bus for the signals going to the 16 LEDs. So, in the top left cell just under port name, click in the box and write sw, which if you recall was the name of the 16 ports for the switches in the constraint file. Check the box to confirm that this input is a Bus. Next click on the zero in the MSB column. The zero is now blue and you can just enter 15, or click the up arrow until the zero increments to 15. As long as it ends up as 15, that's OK.

Next click in the next row down directly below where you wrote sw. The row fills with entries. Click again and then you can enter led as the port name. Change the 'in' to 'out' on this row as the switches are outputs, check the Bus box again, and change the MSB to 15. Click OK to finish. In the Sources pane, you should now be able to see your file under 'Design Sources' with whatever filename you gave it. Double click on the file name and your file will open up in the main pane. Now you have three tabs in the main pane - Project Summary, Basys-3-Master.xdc, and your VHDL source file.

B.7 Modifying the source file

The first thing you'll notice about the source file when you open it is that most of the lines begin with two dashes in a row. This is the comment character in VHDL. It is a bit annoying that the VHDL comment character is different from that in the constraint file. It's because the constraint file syntax is actually Tcl syntax, and I believe the double dash comment character originates in the ABEL language I used when I was 15! The file lines are numbered, and everything up to and including line 21 is basically superfluous. I suggest that you delete these lines. Line 22 is the first important one, where it says 'library IEEE;'. You can also remove the remaining comments after 'use IEEE.STD_STD_LOGIC_1164.ALL' and 'entity switch_to_led is'. The code should now look like this.

```
----------------------------------------------------
--- WARNING - THIS CODE WILL NOT BUILD UNTIL -------
--- YOU MAKE THE ADDITION GIVEN ON THE NEXT PAGE ---
----------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity switch_to_led is
    Port ( sw : in STD_LOGIC_VECTOR (15 downto 0);
           led : out STD_LOGIC_VECTOR (15 downto 0));
end switch_to_led;

architecture Behavioral of switch_to_led is

begin


end Behavioral;
```

Here is an explanation of the code. The first couple of lines declare that you will be using logical data types. The four lines starting with entity switch_to_led is and ending with end switch_to_led describe the input and output ports to your logic circuit. In this case, we have a 16 element input bus whose port name is sw, and a 16 element output bus whose port name is led. Notice that sw and led match the port names in the constraint file. It is very important that this is so, otherwise the code will not compile correctly. Notice that the ports are defined between a pair of round brackets, and the two port declarations are separated by a semicolon. Notice also that the last port declaration has a semicolon OUTSIDE the round bracket that encloses the ports, but no semicolon inside this bracket. All this syntax is correct, and the ports are already correctly defined because we used the GUI window during the create source sequence to specify our ports ahead of time.

VHDL differs from python in that it does not use whitespace characters like spaces, tabs or newlines to convey information. A command is terminated by a semicolon. You can have as much whitespace as you like in a single statement, so long as use a semicolon to tell the compiler when that statement is finished. Personally I think this is profoundly sensible. But, it does mean you have to be careful to remember the semicolons.

What is still missing is what connects the input ports to the output ports. All we want to do to start with is connect each of the 16 switches to the corresponding LED. To do that we add one line as follows:

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

entity switch_to_led is
    Port ( sw : in STD_LOGIC_VECTOR (15 downto 0);
           led : out STD_LOGIC_VECTOR (15 downto 0));
end switch_to_led;

architecture Behavioral of switch_to_led is

begin
    led <= sw;
end Behavioral;
```

The architecture part of the code is where you define how your circuit is connected together between the ports. In this case, it is very simple. We just connect the 16 bit input bus from the switches to the 16 bit output bus going to the LEDs. The <= operator is the 'assignment' operator that makes the connection. It is formed from a less than sign followed by an equals sign. Do not get confused and write just = in this context. That means something else in VHDL. Think of <= as wires connecting two objects. Use ctrl-s to save your file.

Vivado has an automatic syntax checker. If you have made the file correctly, the file name should remain in bold in the Design Sources window. If however you have made a syntax mistake, this will be indicated in your 'Sources' pane as shown in Figure B.6.

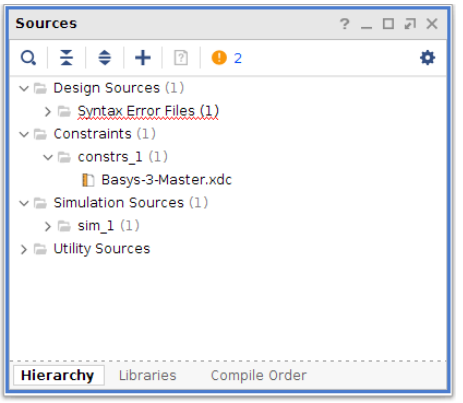


Figure B.6: Appearance of the 'Sources' pane when Vivado has found a syntax error in a VHDL source file.

Even if you didn't make any errors, try inserting a rogue character in your VHDL file, then using ctrl-s to save, so that you trigger the error checker. If

you click on the '>' next to Syntax Error Files, your file is there, and it is not highlighted. Now fix the error and ctrl-s again, and your file is restored to Design Sources and is listed in boldface. Vivado's editor also spots syntax errors in-line by underlining them in red. If you see this kind of red underlining in your file, it probably means you should look for a mistake.

B.8 Synthesising your design

You are now ready to run synthesis on your design. In this step, Vivado and its underlying compiler determine the optimal set of logical entities available on the chip to implement your circuit. Of course, this design will use very few resources, but it's our first one, so no surprises there. Click 'Run Synthesis' in the 'Flow Navigator' bar on the left. When you do this, a small 'Launch Runs' window appears. This is shown in Figure B.7.

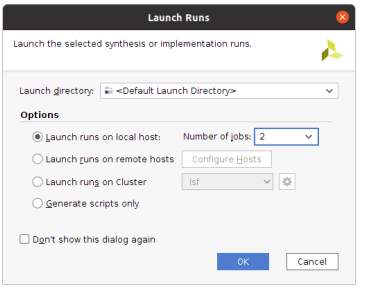


Figure B.7: The 'Launch Runs' window

This window isn't so important for this simple job, but for larger jobs you can save significant time here. Vivado is capable of operating multiple CPU cores at once. In this window you can select a number of jobs up to 4. This means that you can use up to the 4 cores that your laptops i7 pentium processor has. Where you are building multiple modules at once, this will literally divide your build time by four. So, it's well worth being aware of this functionality. In this case just hit OK.

If all goes well, you will see a small 'Synthesis Completed' window like the one in Figure B.8.

You next want to launch the 'Run Implementation' step. This step figures out how to route the wires between the input and output pins on the chip and the required logical elements. Just select OK to start this process. Again, the default of 2 parallel jobs is OK for this build. You'll have to wait a short while for this to complete. If everything goes well, you should see another similar window, 'Implementation Completed'. This time you want to skip opening the implemented design, so just select 'Generate Bitstream', and then click OK (or hit return). Once the bitstream has built, a 'Bitsream Generation Completed'

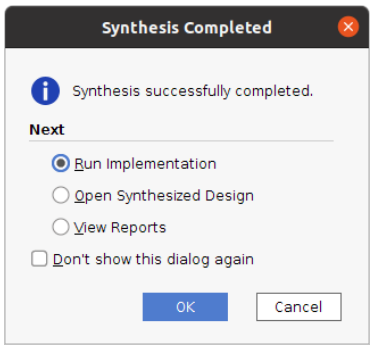


Figure B.8: The 'Synthesis Completed' window

window appears. In this window, you just hit cancel, because we need to get the board hooked up before we launch on the hardware.

B.9 Launching on hardware

We are now ready to launch on the hardware. The first step is to connect the BASYS3 board to the computer. Before we do this, however, there are a few checks. On this board, in the top left and top right corners there are some blue 'jumpers' connected across pairs of pins. The jumper in the top left (with the switches at the bottom) should be connected between the middle of the three pins and the one towards the inside of the board, with the letters 'USB' written on the board next to it. This jumper sets the board to be powered from your laptop through the USB cable. The other jumper position instead draws the board power from the two pins below the power switch labelled EXT and GND. The power switch itself should be in the 'OFF' position, with the switch slid towards the inside of the board.

The other jumper next to the big USB port should be set in the middle position it can occupy, connecting the middle two of the four pins. The board can be programmed in three ways, and the way we will be using today is to program the board directly from an external device by way of a so-called JTAG chain. A JTAG chain is a standard protocol for programming multiple chips through a serial interface. The other two programming methods are to program the board from an on-board flash memory chip (QSPI) and from a USB stick plugged in to the large USB port (USB).

The computer connects to the board using the usb to microusb cable that I supplied you with. Ensure the board is not on a metallic surface that could short out the pins. Note that the micro USB plug connects to the BASYS3 with the flat and larger side of the D connector UPWARDS. Do not force it in because these connectors are not all that robust! It should go in quite easily. Connect the board up to any USB port on the laptop and turn it on at the

switch. A red light in the top left hand corner of the board should light up, and an orange light next to the large USB port should come on for a second or so, and then go out.

Back in Vivado, at the very bottom of the 'Flow Navigator', click on the arrow next to 'Open Hardware Manager'. Click on 'Open Target', and select 'Open New Target...'. A window will appear called 'Open New Hardware Target'. Click next. On the next screen, you want to accept the default that your target is connected to your local machine, so click next again. The window should now show your Digilent device as a hardware target. This is shown in Figure B.9. Click Next and then Finish.

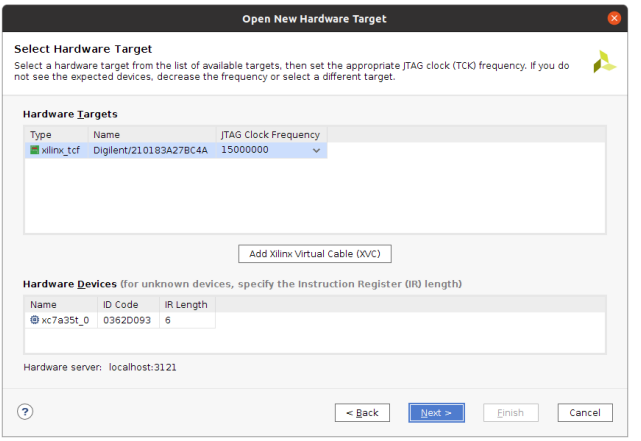


Figure B.9: The hardware window with the digilent board and device visible.

B.10 Programming the device

You should now be able to program the board. At the bottom of the 'Flow Navigator' sidebar, click on 'Program Device' and select the xc7a35. In the window that appears, you can see the bitstream file, switch_to_led.bit, which is used to program the chip. Click program, and after a few moments, the green LED in the top right hand corner of the board indicates that programming is 'done'. Operate all the switches and verify that the corresponding LED lights up when any switch is thrown. Make a note of any that do not work.

B.11 The hardware manager

Once you have the hardware device programmed, your Vivado window will have changed to a state where the 'hardware manager' is running and visible. The appearance of the Vivado window in this state is shown in Figure B.10.

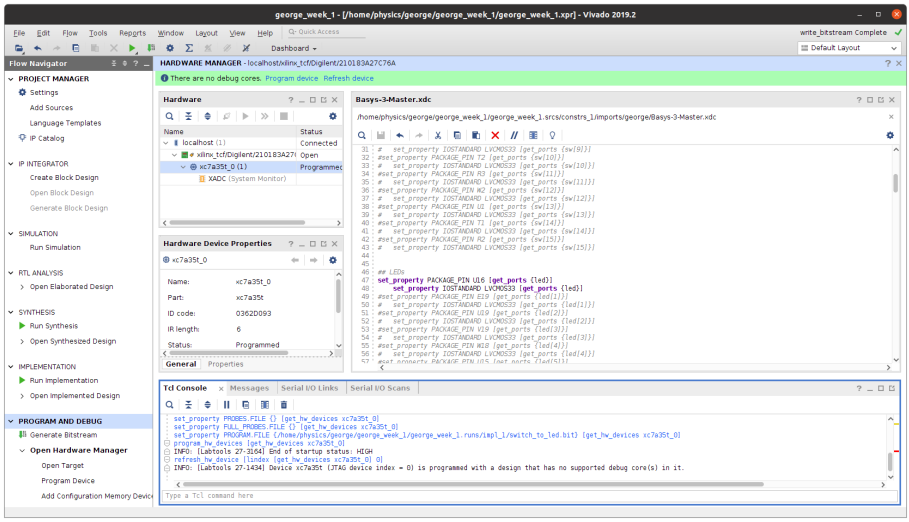


Figure B.10: The Vivado window when the hardware manager is open

In this state, the chip is being monitored in real time by Vivado. The Artix7 has an on board temperature sensor and analog-to-digital converter, so it can measure its own temperature. This is so that commercial applications can put in safeguards for overheating - which is why your mobile phone will probably switch itself off if you leave it on a sunny windowsill in the summer. If you double click on XADC in the Hardware pane, then click 'OK' in the'New Dashboard' window that appears, this will bring up an updating strip chart of the measured temperature of the FPGA in centigrade. You can put your finger on the chip and after a while you will start to see it warm up.

If you turn off or disconnect the board with the hardware manager on, a window will come up informing you that it has lost the connection to the device. This can also happen if your USB cable isn't very well connected at either end, or when you pick the board up with the cable hanging off the connector. It's fine just to click OK if this happens. If you wish to reconnect to the board, just turn it back on, plug the cable back in, or fix whatever the hardware issue was, then at the top of the hardware manager board you can click on 'Open target', then select 'Auto Connect'. It should re-connect to the board when you do this. You will then have to re-program the board using the instructions in Section B.10.

Once you are done with studying your hardware, you may want to start over with a new project, or make modifications to the one you have open. Before doing this, click the 'X' in the hardware manager pane to close it, and then confirm that you want to close it with 'OK'. This will take you back to the more regular appearance of the Vivado window so you can proceed with further modifications, or close the project and create a new one.

B.12 Other project ideas

Here are some other projects you might try. Some of them are very modest modifications to the project already described. Others are more substantial new work. Remember that whenever you modify the constraints file or any of the VHDL code in your projects, you must use ctrl-s to save the changes to your files, at which point the asterisks next to their names should disappear, then redo the process of synthesis, implementation and bitstream generation. Note that you can take a short-cut by clicking on 'Generate Bitstream', which will cause it to synthesise and implement as well, after it prompts you to confirm that it's okay for that to happen. This saves you effort, but it also means that if something goes wrong it may be less obvious to you at which stage it failed.

One additional piece of advice. In my experience more than half of student mistakes have to do with syntax errors in the constraint file, usually with the various square brackets and curly braces. Double-check that you haven't left any stray braces lying around as they are hard to spot in the generally cluttered syntax of these files. If your constraint file is faulty, this may cause the build to fail at the last stage when Vivado is generating the bitstream. This is a clue that can help you go back and give the constraint file extra scrutiny.

For the projects that require quite major changes to the VHDL code and constraints, you might be better off starting again with a new project. To do this you can either close the project you are currently working on using `File->Close project` from the file menu, or much as you can do in WORD or EXCEL, you can instead use `File->Project->Save As...`. The latter will create a copy of your current project, allowing you to select the project location, which should certainly be `/home/physics/vivado_projects`, and a new project name, which should suit whatever the modified project is going to do. This way your current functioning project remains intact, but you avoid having to set up a new project from scratch, since you can now go on to modify the copy without losing the original.

After all that, here are some ideas for other projects you could usefully work on.

1. Change to

   ```
   led <= not(sw);
   ```

   so that the LEDs are off when the switches are on.

2. Make the bottom 8 switches control the top 8 LEDs and vice versa. To do this, use the downto mechanism to refer to a subset of the lines of a bus, so

   ```
   led(7 downto 0) <= sw(15 downto 8);
   led(15 downto 8) <= sw(7 downto 0);
   ```

3. Change the constraint file so that only two of the switches and one LED are enabled. Do this by commenting out all the others. But, also, name

the switches and the LED as separate discrete ports instead of elements of a bus. So, for example, where the constraint file currently says

```
set_property PACKAGE_PIN W16 [get_ports {sw[2]}]
        set_property IOSTANDARD LVCMOS33 [get_ports {sw[2]}]
```

you should modify each of the two switches left so it is a discrete one-element bit rather than a bus element, as follows

```
set_property PACKAGE_PIN W16 [get_ports {swone}]
        set_property IOSTANDARD LVCMOS33 [get_ports {swone}]
```

and similarly for the two lines corresponding to the other switch. In your VHDL code, you need to declare the switch as a separate port, so if the new names of your two switches are swone and swtwo and your single LED is just named led, then your VHDL code port section becomes

```
entity switch_to_led is
    Port ( swone : in STD_LOGIC;
           swtwo : in STD_LOGIC;
           led : out STD_LOGIC);
end switch_to_led;
```

Notice that once the inputs and outputs are no longer buses but single bits, the data type changes from STD_LOGIC_VECTOR to STD_LOGIC. Now that you have simpler discrete inputs and outputs, try the logic gates, so that in the architecture portion of your code you could write

```
architecture Behavioral of switch_to_led is
begin
    led <= swone xor swtwo;
end Behavioral;
```

Check that the logic gates have the correct behaviour.

4. Work out a simple set of logic gates that implement a one bit adder with carry. The functionality should be, when both the inputs are zero, the output is zero and the carry is zero. When one of the inputs is one and the other is zero, the output is one but the carry is zero. When both the inputs are one, the output is zero and the carry is one. Both the output and the carry are displayed on LEDs. Design your logic to implement this simple adder, and build and test your design on your board.

5. Implement a one bit adder, but this time it needs to have three inputs, one for each of the two inputs to the one bit adder you already designed, and a third input to accept any bits carried over from another column. You'll need to figure out a truth table for the two outputs, the out for this bit and the carry out for the next one, from the three inputs, then enable three switches and two leds to test whether your logic implements the truth table, reprogram the board and test your hardware.

Appendix C

Lab Exercise 2 - Sequential Circuits

C.1 Creating a project for a sequential circuit

Circuits implementing a sequence of instructions are called sequential. To define the sequence, and indeed to have any control over the evolution of our circuit with time, we need to implement a well controlled clock. The underlying circuitry was discussed in Chapter 3.4. The core element is a flip-flop whose gate input is connected to an oscillator.

Create a new VHDL project in a sub-directory of the `vivado_projects` of your home directory. Copy the constraint file template into the project. Don't forget to check the box specifying that a copy of the constraint file should be made in the project. You can add the constraint file either when directed to specify constraints at the start of the process, or using the `add sources` button once the project is created. Don't forget to specify the correct FPGA part at the project creation stage. Remember to change the target language from Verilog to VHDL in the main pane of the project once it opens.

Note that the name of the project can start with a number, so `32bit_counter` is a perfectly valid project name. You can also start the names of VHDL source files with numbers. However, you cannot use a number for the name of an entity in a VHDL source file. This is worth mentioning, because if you decide to give a VHDL source file a name that starts with a number, the default name of the entity in the source file will be the same, and this will cause a syntax error. You can either not use numbers for VHDL source files, or you can rename the entity in the file once you have created it, in both the `entity` and `architecture` statements.

C.2 32-bit Counter

The first project of Lab 2 is a 32 bit counter. The upper 16 bits of the counter will be converted to binary and displayed on the LEDs. You will need to edit your constraints file in the project and uncomment the lines defining the on-board clock based on the 100 MHz MEMs oscillator attached to pin W5. These lines are shown below, after the // button in the main Vivado pane was used to remove a single # comment character from the front of each line. Note that I have inserted a backslash character '\' half way through the `create_clock` statement. This is just so the text doesn't run off the page. The syntax of the constraint file is based on the Tcl language, and backslash is the line continuation character in Tcl. At any rate, later on the code seems to build and run fine with this modification. But, as I said, it's only there so I can fit the relevant lines of the constraint file within the page width of the book

```
# Clock signal
set_property PACKAGE_PIN W5 [get_ports clk]
        set_property IOSTANDARD LVCMOS33 [get_ports clk]
        create_clock -add -name sys_clk_pin -period 10.00 \
        -waveform {0 5} [get_ports clk]
```

To continue with the project you will need to use the 'Add Sources' menu item in the 'Flow Navigator' to add a VHDL source. Follow the guide from last week if you have forgotten what to do. When given the opportunity to specify ports, make sure there is an 'in' port called clk to match the constraint file and an out port called led, which is a 16 bit wide bus. Note that the led elements will need un-commenting in the constraint file as well.

When your VHDL file appears, double click on it and take a look. The comments above the first library statement can be deleted - they are boilerplate intended for Engineers who need to record who wrote the code, the history of modifications, etc. Important in a large development team but not when you are writing your first small programs. This code is going to be using numeric data types, so you need to uncomment the IEEE numeric libraries that are by default commented out in the template. The first three lines of your code should look like this.

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

This has the effect of allowing logical and numeric data types within your program. Next follows the entity statement, where you specify the input and output ports for your part.

```
entity thirty_two_bit_counter is
    Port ( clk : in STD_LOGIC;
           led : out STD_LOGIC_VECTOR (15 downto 0));
end thirty_two_bit_counter;
```

C.3 Nodes and signals

Next comes the first portion of the architecture statement, where you describe the wiring inside the part. We immediately encounter some examples of nodes, which in VHDL are called signals.

```
architecture Behavioral of thirty_two_bit_counter is
  signal counter_d, counter_q : UNSIGNED (31 downto 0);
  signal counter_in_logic: STD_LOGIC_VECTOR (31 downto 0);
begin
-- rest of architecture specification code here
```

Recall that the purposes of nodes is to supply points to which the ports of two or more gates, flip flops or other devices inside your circuit can attach. In this case, the two nodes on the upper line are for connecting to the D and Q ports of a flip-flop, and to connect to whatever operations are going to occur between Q and D. These are of type UNSIGNED, which is the VHDL type for an unsigned integer. Our binary counter is going to count from zero up to $2^{32} - 1$, then reset to zero automatically, because overflows are ignored. The other node, called `counter_in_logic` is needed because we need to cast the value of the counter from `UNSIGNED` to `STD_LOGIC_VECTOR` so that we can extract a subset of the bits and display them on the LEDs. There are two reasons why this is necessary. First, VHDL is a strongly typed language. There is no operator that can extract a subset of the bits of an integer. Second, recall that VHDL insists that the inputs and outputs of a device are represented by logical data types. So, even if our board had 32 LEDs and we could represent all the bits, the code would not compile if we were to try and wire the counter directly to the LED output.

C.4 Processes

The next part of the code is written to induce VHDL to connect some of the flip-flops on the board as registers. In order to do this, there must be an if statement, which you haven't met in VHDL, and this if statement must be inside a process. Since the outer of these statements is a process, let us start by explaining what processes are for.

The natural tendency in VHDL is for statements to correspond directly to wires that are connected in a circuit. Outside a process you cannot write two logically inconsistent statements. So, for example, if a node 'myvar' is of data type `STD_LOGIC`, then this code would not be allowed outside a process.

```
myvar <= '1';
myvar <= '0';
```

This code literally instructs VHDL to connect the same node to logic level 1, which is at least 2.0 V, and to logic level 0, which is at most 0.8 V. This would

cause a short circuit, and VHDL will not allow it. The purpose of processes is to permit data structures where there is a necessity for one instruction to supersede another one. In a process, if the same node is wired up several different ways in successive lines, then the last line takes precedence. So for example, this code is allowed

```
process begin
  myvar <= '0';
  myvar <= '1';
end process;
```

The result would be that `myvar` would be wired to logic 1, and not to logic 0. What is the point of this construction? For a start, it is useful if you are going to use a conditional assignment, such as an if statement. For example, consider the following code

```
process(a) begin
  if (a='1') then
    myvar <= '0';
  else
    myvar <= '1';
  end if;
end process;
```

If the node or port `a` is in logical state 1, then `myvar` is wired to 0, otherwise `myvar` is wired to 1. The outcome depends on the state of `a`, and therefore the node or port `a` is included in the dependency list. The dependency list is a comma separated list of nodes or ports (I could say variables, but I'm trying to help you to avoid confusing hardware description languages and high level languages like PYTHON), in round brackets next to the process statement. Any nodes or ports whose value affects the outcome of the statements in the process is included in this list. You can see why conditionals like if...then...else are always wrapped in a process statement in VHDL. Their effect is to set the value of a node or port different ways depending on some other condition. We will meet other conditional statements later.

Note also that, bearing in mind the earlier point about the last statement taking precedence, the following code is equivalent to the conditional example above

```
process(a) begin
  myvar <= '1';
  if (a='1') then
    myvar <= '0';
  end if;
end process;
```

Here the first assignment of `myvar` as 1 can be though of as a default. However, if it turns out that `a` is 1, then the later assignment of `myvar` to 0 supersedes the default. I think this way of using if...then conditionals is good practice, because if you set a default assignment for a node or port inside the process before the conditional, then the node or port is assigned to some value, whatever happens with the conditional. In VHDL, where it is crucial to avoid leaving hanging wires connected to nothing, the existence of a default for all nodes or ports is an important protection against hard-to-diagnose faults.

Now, consider the process and conditional statement that follows next in the code for the 32 bit counter.

```
process(clk, counter_d) begin
  if(clk'event and clk='1') then
    counter_q <= counter_d;
  end if;
end process;
```

This is a special combination of a process and a conditional that is used specifically to set up memory registers. First, note that the outcome of the code depends on `clk` because it is used in the if...then statement. Also, the assignment of `counter_q` depends on the state of `counter_d`. Hence both `clk` are in the dependency list. Second, note the special purpose syntax inside the if statement. The first part, `clk'event` is a special instruction to look for edges in the `clk` port, times when the `clk` port is transitioning between states. The second part of the conditional checks that the value of `clk` after the transition is detected is 1. The combination of these two conditionals is enough to induce VHDL to wire `clk` to the gate port of a flip-flop, since this is the only circuit element on the chip that can detect edges.

Since the hardware device being used is a flip-flop, we know that the only thing it can do is copy the value of its input, or D port, to its output, or register, or Q port, so the contents of this if...then statement have the effect of ensuring that 32 single bit flip-flops will have their input D ports connected to the 32 bits of `counter_d`, and their output Q ports connected to the 32 bits of `counter_q`. When rising clock edges appear, a momentary snapshot will be taken of the state of `counter_d`, and copied to `counter_q`, where it will remain persistent until the next rising clock edge.

Any number of register updating statements can be included in this special purpose process and conditional statement. But, no other operations should be put in this process! This is because the flip flop copy operation can happen almost instantaneously on the detection of an edge in the gate port, but all other operations, such as logical or arithmetical operations, take too long to be implemented on the clock edge like this. If you have other operations to perform, put them in a separate process.

Another good rule of thumb is that only inside this special process should you use the `d` port of a flip flop as the input on the right of the <= operator, and the `q` port of a flip flop as the output on the left of the <= operator. Outside

this special process, you should always be setting the `d`, or input port, based on the value of the `q`, or output port. This means that all the time consuming operations have 10 ns for their outputs to settle; the only operations that need to happen much faster are the copies inside the flip-flops, and these are fast by design.

C.5 Counting

To actually count upwards, we will use the + operator, which implements binary addition on all the integer data types we will use in the course. Note that + just adds binary sequences. The input and the output should have the same number of bits, and any overflows will be discarded. The code to implement the counting is very simple

```
counter_d <= counter_q + 1;
```

Since `counter_q` is the currently stored value of the counter, and will remain static for 10 ns between rising clock edges, all this statement does is ensure that the appropriate logic is in place between the node `counter_q` at its input and the node `counter_d` at its output such that `counter_d` is one greater than `counter_q`. At the next rising edge in `clk`, `counter_d` is copied to `counter_q` incrementing its value by 1 and the cycle continues.

C.6 Casting and selecting a subset of bits

The next two lines introduce us to casting and a further operation on logical data types, the selection of a subset of bits.

```
  counter_in_logic <= std_logic_vector(counter_q);
  led <= counter_in_logic(31 downto 16);
```

Note that since these lines are outside any process or conditional construction, they represent permanent wires between nodes and ports. In the upper line, the node `counter_in_logic` is connected to the output of a cast operation, with the currently stored value of the counter at the input. The cast operation is necessary because VHDL is a strongly typed language. You need to specify exactly what data type each node or port has. We want a node of the `STD_LOGIC_VECTOR` type with the same bus width as the counter, so we employ the `std_logic_vector()` operator, which casts a node or port into the `std_logic_vector` data type. In the lower line, we use the `31 downto 16` construct to select a subset of the bits in `counter_in_logic` to assign to the 16 `led` ports.

Lastly we must end the process, with

```
  end Behavioural;
```

Type in the code carefully, making sure you understand the concepts as you do so, as this code contains several new concepts. Ask myself or Mitch if you are unsure about anything. Then, run synthesis, implementation, and bitstream build in turn. At each stage, there may be errors or critical warnings - if the build fails you can check for these in the Messages tab of the lower pane. Once you have built your bitstream, follow the same procedure as last week to attach your board, launch the hardware server in a terminal window, open the hardware manager, connect it, program the board, and see if the lights count in binary.

C.7 Gating your counter with a pushbutton

Let us next make a counter that only counts up when you have a pushbutton pressed, and stores the previously attained value when the counter is released, starting to count from that same value when the pushbutton is pressed again.

Close the hardware manager then use the `File->Project->Save As` option to save the project as a new name, say `gated_counter`. Make sure the directory of the new `gated_counter` project is in a subdirectory of `vivado_projects`. There is no need to include run results in the new project, although it does no harm if you do.

Open up the constraint file, and scroll down until you find the lines referring to Buttons. There should be five sets of two line entries, each referring to one of the cluster of five pushbuttons arranged as central, up, left, right, down. Uncomment the two lines referring to the central button, then save the modified constraint file.

C.8 Conditional assignments

We have already discussed if...then statements and stated that these have to be inside process statements. It turns out that there is one type of conditional which operates slightly differently, and as a consequence should never be inside a process. These are called conditional assignments. Here is an example of a conditional assignment. In this case c is a 2 bit `std_logic_vector` and a and b are 32 bit signed integers.

```
a <= b+1 when c="01" else
     b+10 when c="10" else
     b+100 when c="11" else
     b;
```

Note the defining feature of conditional assignments, that the conditional exhausts all the possibilities for the controlling logic. In this case, a 2 bit parallel bus can only take the four values $B00$, $B01$, $B10$ and $B11$. The code specifies an outcome in each of those eventualities. In fact, this code is somewhat overengineered since the last else means that there is a default, which covers all

eventualities other than those above. It is good practice always to include a default in conditional assignments.

Using this new construct, you should be able to modify the line

```
counter_d <= counter_q + 1;
```

so that the effect is the one desired in this project - that when the pushbutton is pressed (this will make the corresponding port logic 1), the counter continues to count up, but when the button is released, the count will stop and retain its value. Try and get this working without more detailed instructions.

C.9 Counting button pushes

Now we are ready to try something significantly more challenging. Consider the humble pushbutton. Even if you press it for as short a time as you physically can, the duration of that press is going to be many thousands of 10 ns clock cycles, as you can verify by watching the LEDs when you push the button as briefly as you can in the previous project. Given that, how do we build a code that increments a counter by exactly one every time a button is pressed?

Let's take a first guess at how to handle this task. Let us build a detector for changes of button state from zero to 1. After all, you might think that when you press a button it may stay in state 1 for a long time, but maybe it only transitions from 0 to 1 once. Couldn't we use this to achieve our goal?

To try this idea, create a one bit register called `bstore`. This should have an input and an output node just like the counter register, and it should be implemented with a flip flop in just the same way. Connect the input node to the button. You can detect edges by comparing the current button state to the output node, which after all represents the button state last clock cycle. If the current button state is 1 and the bstore output is 0, this means that the button has just changed state from 0 to 1. If this is true, then increment the counter. Otherwise, leave the counter in its previous state.

See if you can get this working, and see if it works as well as you wanted it to, or if there are problems. If there are problems, then try to work out what is causing those problems. Here are a couple of hints. The register copy from `d` to `q` for bstore can be in the same process and if...then statement as that for the counter. You can reduce the number of bits in the counter from 32 to 16 and put all of them on the LEDs. You will no longer need the `counter_in_logic` node as you can cast `counter_q` directly to the `led` outputs. Another hit is that conditional assignments can have logical combinations of several conditions after the `when` keyword and before the `else`. Finally, don't forget to modify the dependency list for the register to include `bstore_d`.

C.10 Bouncing buttons

If you play with your button-triggered counter for a while, you will probably find that, though most of the time it works fine, occasionally there will be one of two effects. The first effect you will sometimes see is that you will press the button once, but the count will increment by more than one unit. The second effect you will see sometimes is that there will be one or more counts at the moment you release the button. These problems are caused by the physical structure of the button itself. Figure C.1 is a slightly exaggerated illustration of the problem.

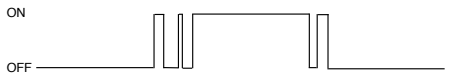


Figure C.1: The bounce effect in a pushbutton.

The button is not mechanically perfect, and now and again, both at the pressing of the button, and the release, there are multiple state changes. This effect is called 'bounce' in engineering, and mechanisms to prevent it are called debouncing circuits. As our final project today, and the most challenging of them, we will think about how to make a debouncing circuit for our pushbutton.

There are in fact several ways to do it, and the better and clearer ones require programming constructs we have not studied yet, such as state machines. However, with what we already have we can try and make a button debouncer.

The first thing is to recognise that we are most of the way there. Our buttons appear to be of high enough quality, that the number of bounces is rarely more than one, and usually zero. A good question is, what makes multiple switches of state a bounce? The answer is that the multiple switches have to occur within a short time interval. What we want in a debouncer is a way of ignoring further changes of state that occur less than a certain short amount of time after an initial change of state is detected. As can be seen from Figure C.1, the same effect happens both on press and release, so this mechanism should be triggered both on the detection of a transition from 0 to 1 when the button is pressed, and on the detection of a transition from 1 to 0 when it is released.

One way to implement this is with a second counter. We have one counter that is now just counting button presses. This counter needs to carry on doing its job. We are going to need a second counter that counts driven by the clock, as it was before in the very first project. However, it only needs to start counting up when a change of button state is detected. Let's say this counter is zero initially and the button is unpressed. Now our current mechanism detects a change of button state from 0 to 1. This increments the counter. It is no longer at zero. Let's say the next value of the counter depends on its current value. If the counter is at zero and no change of button state has been detected, the counter stays at zero. If the counter is at zero and a change of button state has been detected, the counter increments. If the counter is anything other than

zero, it increments anyway, no matter what the state of the button. Thus, the counter keeps on counting up. Once the counter reaches its maximum capacity, it naturally resets to zero, at which point we're back to the zero state. Crucially, the counter not being zero implies that we are within a time interval, settable by a judicious choice of the number of bits in the counting variable. Only when a change of button state from 0 to 1 is detected, and in addition the counter is at zero, do we increment the second counter that is counting how many times we have pressed the button.

The last thing to consider is what happens when the button is released. The counter is at zero, having cycled round after the button was pressed. Now we detect that the button state is zero but it was 1 last time. We increment the counter to 1, and it's exactly the same as in the button press, any further button changes of state do not result in our button press counter being incremented. Long after the button state has settled back down to unpressed again, the counter again cycles around back to zero, and we are back where we started.

See if you can implement all of this. It's a bit more challenging, but it just requires judicious use of conditional assignments and one additional counter. Note that the operators $>$ and $<$ for greater than and less than work with numeric data types in VHDL.

Appendix D

Lab Exercise 3 - The Seven Segment Display

D.1 Introducing the seven segment display

The seven segment display on the BASYS3 board is the first device we have encountered that is addressed serially. The display is connected to the FPGA by a total of 12 pins. There are four pins that control which segment of the display is currently being addressed, and the other eight pins control the seven bars of the digit and the decimal point.

We leave the names of all pins connected to the display as they were in the master constraint file. The four control pins are sometimes called anodes, because you can think of each of these four pins as connecting to the positive terminals of all 8 LEDs associated with a single digit. As defined in the constraint file, `an[0]` is associated with the rightmost digit, `an[1]` with the digit second from right, and so on. The signals on these pins are active low, which means that when they are set to 0, that connects the LEDs of the corresponding digit to the other 8 pins on the FPGA which control what pattern of LEDs are lit up for that digit.

So, for example, if I write `"0000"` to the anode signals, this means that all four digits will display the same pattern of LEDs lit up as is represented by the signals on the other eight pins. In our first exercise with the LEDs, we will light up all four digits with the same patterns, set from eight of the switches on the board.

The other eight pins from the display control what is actually visible from the segments of the digit and the decimal point. As for the anodes, all eight signals are active low, meaning that you have to set them to 0 to light the corresponding segment. The names of the eight lines are `seg[0]` through `seg[6]` for the seven segments of the LED display, and `dp` for the decimal point.

D.2 Exercise 1

Create a project that permanently sets the anode signals to all zeros, so that all four digits are being addressed simultaneously all the time. Connect eight of the BASYS3 switches to the seven segments and the decimal point. Build the project and experiment with lighting up the bars in different combinations, determining which bars on the seven segment display connect to which elements of the `an[]` array.

D.3 Exercise 2

Now modify the project so that four more of the switches on the BASYS3 control which of the digits are connected to the switches that were previously controlling all the digits at once.

D.4 Slow cycling

In last weeks lab we built a 32 bit counter that has a count rate, dictated by the external clock period, of one count every 10 ns. However, in order to use this display, we wish to count much more slowly, and furthermore to use those slow counts to determine which digit is being addressed, so that we can address them all in turn. At first, we want to switch between the digits so slowly that we can see that switching taking place. Therefore, use the formula worked out in class to work out which counter bit has a period of approximately one second. Say this bit is n. Then bits n and $n + 1$ form a two bit counter that counts from 0 to 3 and then resets and does it all again. You can then use a conditional assignment to associate the values of the binary counter with anode port patterns that select each digit in turn. For example, when the counter reads `"00"` that could translate to an anode bit pattern of `"1110"` to select the rightmost digit only.

D.5 Exercise 3

Make a fast counter with 32 bits. Do not forget to uncomment the lines corresponding to the external clock in the constraint file, and to uncomment the numeric data type library at the top of your VHDL code. You will have to make a `std_logic_vector` signal having 32 bits to cast the clock into logic form. Create also a two bit wide signal that is wired to the bits of the clock corresponding to a least significant bit count rate of approximately one Hz. Use a conditional assignment to assign the anode pins to select a different clock digit, from right to left, as the counter increments. Now you should be able to use 8 switches to address each of the four digits of the display in turn, and at a one second

switch rate between digits you should be able to see, manually, how the four digits are addressed sequentially.

D.6 Exercise 4

You should now calculate which bits of the counter will cause the digit scan rate to be about a quarter of a millisecond per digit. Change the two bits used to assign the anode pin values to the two bits that have this switch rate. When you re-build the code, all four digits should again appear lit, but this time you know that the four digits are being addressed sequentially, rather than all at the same time.

D.7 Exercise 5

Next by working out the specific LED bar patterns associated with these letters, and using a conditional assignment to set the bar patterns on the display dependent on which digit is currently being addressed, write the word $F00d$ in the display.

D.8 Exercise 6

Work out the bit patterns in the anode bars to display the sixteen hexadecimal digits, `0-9` and `a-f`. Your BASYS3 board has four 7 segment displays on it, each of which can display any of these sixteen digits. It also has 16 switches, which you can think of as 4 groups of 4. Each group of 4 switches has 16 different possible configurations, 0000 through 1111. Write a VHDL program that allows you to display any combination of four hexadecimal digits by setting the switches accordingly. For example, to display the word `F00d`, your switch configuration would be `1111000000001101`.

Appendix E

Lab Exercise 4 - Joining and Sharing Components

E.1 Motivation and Overview

In Appendices B, C and D we used as examples a binary counter, a de-bounced push-button, and writing to a seven segment LED display. In this chapter, we shall use these three projects as components of a device that counts how many times you have pressed a push-button and displays the count in hexadecimal on the display. We shall do this in several different ways, and as an aside we shall learn how to use, and then how to set up, a shared repository for code using the sharing infrastructure of `git` .

E.2 Using a git repository

A public git repository containing VHDL code for a debounced button, a sixteen bit counter, and a driver for the four digit display on the BASYS3 board can be found at `https://github.com/eddaw1/phyip`. Go to this repository. You will find yourself in the main directory and see before you several folders, including three that I will want you to use in this lab. These are `bgate`, which contains IP for a debounced button, `sixteencount`, IP for a sixteen bit binary counter triggered by an active-high logic pulse, and `sevenseg`, a code for driving the seven segment display. Descend into each subdirectory in turn. Note that the `..` link in the subdirectory takes you back up a level to the main `phyip` directory. Within each subdirectory, you will see two further subdirectories, called `src` and `gui`. These contain, respectively, the VHDL source code and a file that allows Vivado to generate a graphical user interface (GUI) widget corresponding to the IP device. We will first use the `port map` mechanism to write code connecting the pieces of IP into a project.

E.2.1 Connecting up IP using `port map`

The `port map` mechanism allows you to connect input and output ports of the IP elements together to form a larger program, rather like calling functions within the body of a program in a high level language. To connect together the elements from my IP catalog, you'll need to start a new Vivado project in the usual way. Specify the usual constraint file, uncomment the lines associated with the clock, the 7 segment LED display (decimal point, anodes, and segments), and the central button on the 5-button pad. Now within the project, you are going to create four new VHDL files, and copy the VHDL code from my IP repository corresponding to the debounced button, the counter, and the display driver into three of these three VHDL files. The fourth file is for the port map statements to link them together. You can do this by creating empty VHDL files, then copying and pasting the code form my repository into the empty files. Call the files `sevenseg.vhd`, `counter.vhd`, `debouncedbutton.vhd` and `toplevel.vhd`. You can add all three files at once using the `Add Sources` link in the top left of the Vivado GUI. There is no need to use this link three times separately for the three files, though it does no harm if you do. You do not need to specify any particular inputs or outputs to these files.

Now go in to `counter.vhd`, and completely empty it out. On the git repository web site. Descend into the `sixteencountsrc` subdirectory, and click on `sixteencounter.vhd`. This will then show you a source listing of the VHDL for the IP. Click on the `raw` button in the top right corner of the code pane, and you will see an ascii dump of the code. Empty the `counter.vhd` file in your Vivado project. Back in the git repository, highlight the code, crtl-c to copy it, and in VHDL ctrl-v to paste it into the `counter.vhd` file in your Vivado project.

Repeat this exercise, pasting the contents of the git repository file `bgate/src/32bit_counter.vhd`, which contains the debounced button, into the Vivado project `debouncedbutton.vhd` file, and the contents of the git repository file `sevenseg/directlight.vhd` into the git repository file `sevenseg.vhd`. Three out of the four VHDL files in your Vivado project should now be populated. Save them all. As you do this, check for the appearance of syntax errors in the files. It is quite easy to make a mistake and forget to delete the last line or the first line that the newly created VHDL file came with when you paste in the contents of the git repository, for example.

Next you need to create the port map file that connects everything together. Go in to `toplevel.vhd`. Using the notes from seminar 6, yesterday, create three port maps that connect together the three pieces of IP with internal signals to interconnect between the three IP blocks and ports for inputs and outputs. The only input ports should be clk and the button, and the output ports should be the anodes, segments, and decimal point, of the 4 digit LED display.

E.3 Your first VIVADO GUI

The `port map` mechanism is, as I said in the lecture, rather cumbersome. Though it gets the job done and sometimes not having to mess about with a graphical user interface can be faster, and more efficient, there is no way of really visualising the functionality of the code you are writing. The Vivado GUI mechanism, where you represent each piece of IP with a graphical widget and join them with wires, is a natural way to overcome these problems. We can exploit this mechanism to build up the same project as we just went through with `port map`, graphically.

The instructions for doing this are again in the notes for seminar 7 from yesterday. Starting at slide 20, open a terminal and use `git clone` to copy my entire git repository onto your laptop. Then follow the instructions on Slides 21-30 to build up a connected block diagram connected to the same ports as in the port map version. Finally create an overarching master vhdl file and build and run your counter. Specific instructions that mirror the contents of seminar 7 are given in Section E.3.1.

E.3.1 Drawing a block design

Once the main VIVADO window is open, look in the top left corner for 'Create Block Design'. The default name of design_1 is fine. Hit OK. We are going to want to add our own IP blocks to this design, so we need to tell Vivado where to look for them. They should all be in the my_ip local git repository copy. Right-click on the open design window, and select 'IP settings'. In the left-hand menu of the window that appears, flip open the option labelled IP by clicking on the arrow, and select 'Repository'. A small box should appear, labelled IP repositories, which is initially empty. Click on '+' and navigate to my_ip. When you select this directory, you should see this appear in the 'Add repository' window, and there should be at least 3 IP blocks, one for each of your mini projects. Click OK and OK again to close the IP settings window.

You next need to add one copy of each of the debouncer, the sixteencounter and the seven segment display. Click on the '+' button to add IP. An enormous list will appear, containing many interesting possibilities - a very large stack of pre-configured IP arrives with the Vivado package. Fortunately there is also a search engine that should permit you to find your three pieces of IP using the names they had when you exported them as IP - probably these will be the same names as those of the VHDL source files they were created from. As you add these IPs to the project you will see their gui representations appear in the window, and things should look very much as they do in Figure E.1.

You should play with the GUI window a bit. If you left click and drag diagonally in the window, it either magnifies or zooms out on the gui window. If you find yourself lost, you can click on the third icon at the top from the left, which is 'zoom to fit' and will fit whatever graphics you have imported into your view in the window. You should rearrange them so that the debounced

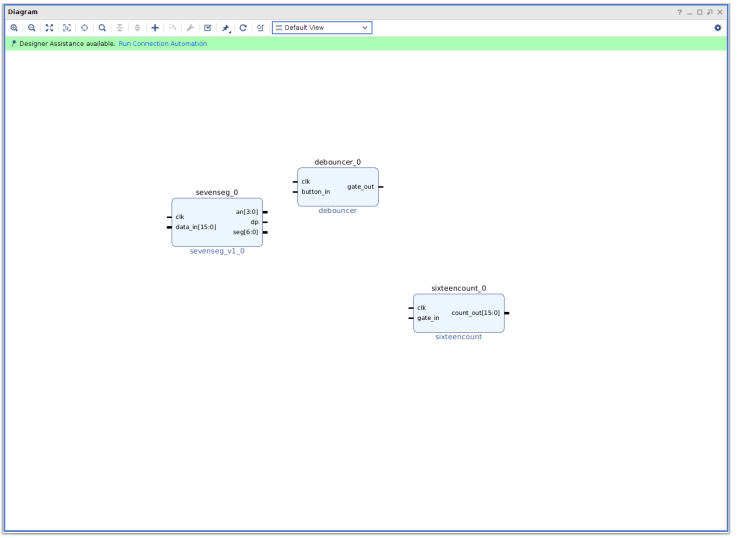


Figure E.1: The gui window before wiring but after the IP elements of our design are added.

button is on the left, the counter is in the middle and the display driver is on the right. Then wire the internal connections by dragging wires between ports until it looks roughly as it does in Figure E.2.

We have now connected all the internal wires, those that don't join to any ports - pins on the actual chip. All three of our widgets require a clock input. If you hover over the clk input to the debouncer, and right click then select 'Make external', it will create a wire to an external input. Repeat this for button_in, and on the seven segment display for both output ports (mine has an additional ouptut for the decimal point, so I've connected that too, but you needn't worry about this. Now you can also join wires from the clock inputs to the other two widgets to the same input port for the clock as for the debouncer, and it should create neat wires connecting all three clock inputs to the same input port. Things will look very much like they do in Figure E.3.

One slightly annoying habit of this gui is adding _0 after the name of each port that is made into an external input or output. These either have to be added by hand in the constraint file, or they have to be edited in the gui. The latter is less fussy. Simply click on each port in turn, and a window at the lower left of the GUI pane should appear called 'External Port Properties'. The names I suggest are clk, btnC, an, and seg. These will then match the corresponding names in the constraints file. Finally, edit the constraints file and make sure each port that you are going to use is uncommented. Save the constraint file. Save the block design in the file menu. And, make sure the constraint file is saved.

One remaining step before we build. The gui is in fact just a graphical user

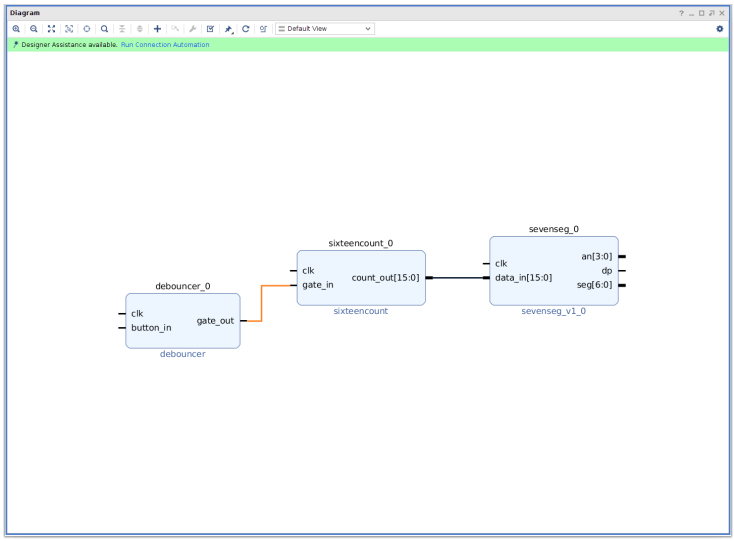


Figure E.2: The gui window after internal wires but before external wires.

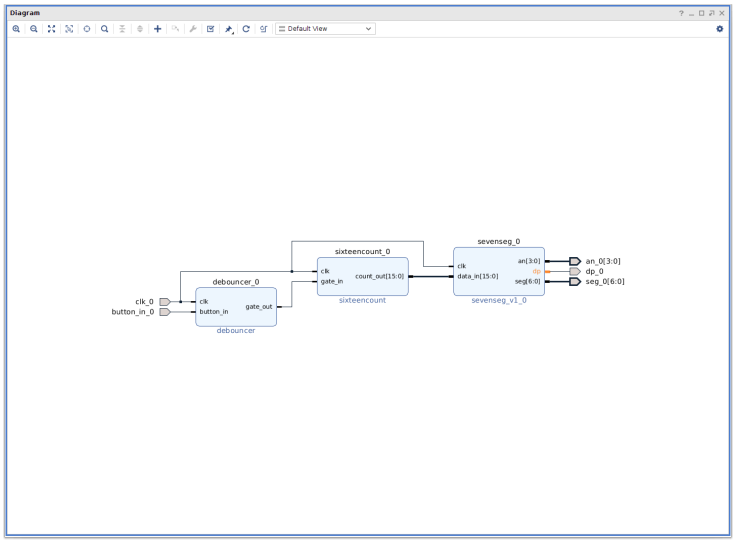


Figure E.3: The gui window with all wires in place.

interface to VHDL, and therefore we need an overarching VHDL file into which this gui fits. To create one, go to the 'Sources' tab in the top left of the main Vivado window, and highlight design_1, your block design. Right click on it and select 'Create HDL wrapper'.

You are now ready to build the project. Click 'Run Synthesis' as usual. This will take a little longer than usual because each of the IP sub-blocks needs to get built before the project as a whole is put together. This is where having multiple cores on your laptop is useful - one core can be deployed for each IP build in parallel, which speeds things up noticeably. If you are curious about how things are progressing, go to the 'Design runs' tab in the slim pane at the bottom, and open the sub-area called 'Out-of-Context module runs', and at the bottom of that hierachy you should be able to see things building.

Once the synthesis has finished, you should be able to proceed as usual with Run Implementation and Generate Bitstream. Hopefully when you write the bitstream, you will be able to use the central pushbutton to increment the digital display counter. Though this was a simple project, you can see that the graphical method gives you a much more visual picture of what is going on in the code. The power of this method is fully exploited when building large systems on the chip that include processors, memory, input terminals, and shared memory with your own modules.

If you get done with this exercise, and successfully build your debounced button driven counter, you might want to try one of the following exercises.

- Create a git repository using the instructions in Section E.4 below.
- Use the built in IP called DSP48E1 in Vivado to make a simple adder for two numbers that explicitly uses the DSP slice.

E.4 Starting your own git repository

A git repository is the current standard for sharing and team development of code. You can, and we will learn how to, store your created IP in a git repository, then retrieve it and use the copy to deploy into a new git project. Others can (with your permission) also download your developed IP, and many users can also collaborate on the same code, even at the same time using a mechanism called branching. Here we will only go as far a simple example, a repository containing just the IP for your 32 bit binary to hex display converter. So, the first job is to create your own account on github, if you don't have one. Visit the URL `https://github.com`, and follow the instructions to open a new account. in the left hand vertical menu bar that appears, to the right of the word 'Repositories', there should be a button labelled 'New', which you click, and are then prompted to provide a name for your repository. I suggest you call it phyip. Once you have created it, you can return to github home by clicking on the cat silouette in the top left corner. Then you can click on phyip under Repositories to enter the new repository. It is currently empty.

The next step is to ensure that you can check your code into the repository. To do this we will create an encryption key pair between your laptop and your git repository. On the laptop type in the following:

```
ssh-keygen -t rsa -b 4096 -C "physics"
```

Accept the default location for the file, which is in your home directory under a hidden subdirectory called .ssh (all directories and files whose name begins with a dot are normally hidden, but you can see them using ls -a). Also when prompted for a passphrase, leave it blank, and again when asked to re-enter it. Now if you invoke

```
cd /home/physics/.ssh
ls
```

you should be able to see two new files in there, `id_rsa.pub` and `id_rsa`. These are your public and private keys. The basic rules of key encryption is that the private keys never leave your machine, but the public keys can be shared with anyone.

In the web page for github, inside your new phyip project, at the top there are a horizontally justified set of tabs, including a settings tab which you should click. Close to the bottom of the left justified vertical list of options, you should see 'deploy keys'. Click this option, and then click AddDeployKey. This should bring up an empty box. In the terminal window you have open in your .ssh directory, type more id_rsa.pub. Several lines of seemingly random alphanumeric characters starting in ssh-rsa and ending in physics should appear. If you left-click-drag over the whole thing, then middle-click in the AddDeployKey box in your github deploy keys window, then the whole string should get copied over. Next give it the title 'physics' and click 'Add Key'. This gives your computer remote access to the git repository from the terminal command line. Your new repository is now ready to drop your code onto.

Next, cd into the `/home/physics/my_ip` directory. A subdirectory for your display IP should already be there. The next job is to create a generic readme file called README.md. To do this type nano README.md. This should open a blank text editor window in your terminal. Type in a sentence of explanation of what the repository is for. Now do ctrl-O to write the file, and ctrl-X to exit the editor.

You are now ready to make the local directory into a git repository and push to your github account. You should still be in the `my_ip` directory, and this is going to be the top directory of your git repository. From there, do the following

```
git init
# the < and > symbols should not be included, they just prompt
# you to replace the carets and their contents with something
# personal to you.
```

```
git config --global user.email "<your email>"
git config --global user.name "<your name>"
git add --a
git commit -a
```

When you do git commit, you'll be put into a nano editor window, and there you type some description of what you just did to the code - in this case something like repository creation is appropriate. Use ctrl-O (accept default filename) and ctrl-X to exit. Now to create a shortcut to your github repository. In the command below, put your github username where I have inserted the carets.

```
git remote add origin git@github.com:<username>/phyip.git
git branch -M main
git push -u origin main
```

Now if you refresh your browser window in the github site, you should be able to see entries for the README.md file and the displayhex IP directory. Congratulations! Your first developed HDL code is now on git. If you want, you can make it public and others can download and use it. We'll have a go at cloning a git repository back to your machine and making use of it next.

E.5 Use of github to store VHDL source for use with `port map`

For the purpose of storing VHDL source for use with the `port map` mechanism, it suffices to store only the VHDL source that corresponds to the device you have created. For example, you may wish to store devices for the pushbutton debouncer and the LED display. It will be important to organise things so that our git repository does not get cluttered. Let us therefore agree on an organisational framework for our repository. Within the phyip project, let us create a subdirectory called portmapsources. Within this subdirectory we will create a further subdirectory for each piece of VHDL that is intended to be used with the `port map` mechanism. So, within the `my_ip` subdirectory on your laptop (so you will need to `cd my_ip` in a terminal window, make new subdirectories for each piece of VHDL code you want to store.

```
cd my_ip
mkdir debouncebutton
mkdir leddisplay
git add debouncebutton
git add leddisplay
```

The `git add` commands don't do anything to the remote repository - nothing happens remotely until you invoke `git push`. Next `cd` into each of the

subdirectories in turn, and in that subdirectory create a `README.md` file using `nano README.md`. This file should contain what the VHDL code does, your name and email. I would advise against putting a date of creation because git is much better at keeping track of that than you are likely to be. Dates written in to files become defunct almost immediately and it is impossible to keep track of all of them.

Now it is time to go and find the VHDL code that you used in the Exercise of Section E.2.1. The actual VHDL file you created is in the directory containing the Vivado project where you tested your portmap under

```
/<project name>.srcs/sources_1/imports/<username>/<filename>
```

Use cp to copy this file into the correct subdirectory of your git repository, the same directory where you created the `README.md` file that tells you where it is. Now `cd` back to `my_ip` and in that directory invoke:

```
# adds all the new files to the git project
git add -A
# commits the current version of the files - you'll
# need to do this each time you modify them.
git commit
```

Once again you'll be taken into the nano editor - write some text explaining what you just added to the repository, then do ctrl-O (and accept the default file name) to write the file and ctrl-X to exit the editor. Nothing you have done so far will affect the repository on github. However, now we are ready to push the new version to the github repository.

```
git push -u origin main
```

If all has gone well, your VHDL code should now be safely backed up to the repository.

If this seems like an awfully convoluted way of backing up a file, then you are right - it is! But, the advantages of this method of keeping a backup quickly becomes apparent when you are developing software in a team, or are interested in publishing what you have written.

Git supports an otherwise problematic side of software development called version control. Suppose there were 100 developers working on a massive `VHDL` project, and all of them were modifying some subset of the files all the time. How would you keep track of these things? Git supports branching, where you can create a 'branch' of the repository, check out the code, make changes and check back in without actually merging your changes in a way that affects other developers. Whenever the repository managers agree that everyones changes are going to be combined together, then the repository manager can carefully merge everyones changes back onto the master repository. And, when everything works again (usually after some problems with incompatibilies - you can't escape that one!), the master repository can be 'tagged', so that if someone breaks the

master branch, it is possible to rewind to the last time it worked and thereby fix it.

Another advantage is that github supports a secure web interface, which we have been using already to view what is in our repository. In fact, if you go back to your github repository view on the web and use the browser refresh, you should be able to see the new folder structure and its file contents on the web interface. You can also edit the files there, but bear in mind that there is no way to test your edits by building the VHDL code into bitstream files and programming the board remotely, so any edits done on the web interface to the repository should be done with extreme care. One neat thing about the github web interface is that github 'knows' the syntax for a wide range of languages, including VHDL! So if you use the web interface to view your source code you'll find the comments, commands and keywords are colour coded for easier code browsing.

Finally, Git supports controlled public release of your software. You can let the entire internet look at your source, but nobody that you don't want to be altering your code is allowed write access. So, if you think your debounced pushbutton works particularly well, you can launch it onto the web and who knows? Perhaps it will be in a cellphone somewhere before you know it!

E.6 Packaging for the VIVADO GUI

The `port map` paradigm is perfectly functional and flexible enough to support quite sophisticated large project integration from modular small pieces. The main limitation is that it doesn't lend itself to quick debugging. It is hard to see for any remotely complicated module, based on the port map syntax, exactly how things are wired together, and it is easy to make mistakes, as you may already have found. Also, you have to know exactly how to connect everything, down to the individual wire. This isn't so bad when you have a 3 port device such as the debouncer, but what happens when your device is a microprocessor? You shouldn't have to figure out exactly hwo to wire everything! It would be nice if modules could come with some built in native intelligence so they in some sense 'know' how they should be wired up. For example, we shall soon see that we can grab a module that is a fully functional microprocessor called microblaze. This processor will want to be connected to memory. The memory is also available - we don't have to write it (thank goodness). We also, it turns out, don't have to know exactly how to connect the memory to the processor. This can all be coped with in vivado using a hand graphical user interface that they supply with the package. We will now learn how to deploy our IP so that it can be used with this package. It is surprisingly easy. After that, we'll learn how to deploy our graphics-ready widget to github so that others can use it.

Reopen the VIVADO project where you previously created and tested the `port map` to the 3 port debounced button core. The VHDL for this core itself is in a subdirectory of that project, called

```
/<project name>.srcs/sources_1/imports/<username>/<filename>
```

It is this VHDL file that we want to package up as a GUI widget that can be used in the vivado graphical developer. In the VIVADO project (you should previously have verified that the button debouncer works - you can just as easily carry out this exercise using the 4 digit LED display driver, so long as you have a working submodule that was successfully accessed using the `port map` protocol.

In the open VHDL project, first ensure that the hardware manager is closed. In the 'tools' menu at the top, select 'Create and Package new IP...'. In the sub-window that appears, have a read of the verbage, then click Next. The default in the next window is to package the whole project. But we don't want to do this, because the whole project interfaces the debouncer to a specific button on a specific board. What we want to export is just the VHDL that debounces a generic button, potentially on a generic board, even perhaps using an FPGA made by a different manufacturer, or perhaps this VHDL code will be used to design an application specific integrated circuit (ASIC) - a much more specialised (and expensive) proposition! So, instead we check the box that says 'Package a specified directory', and again click Next. We then navigate to the directory containing the VHDL code for the debouncer. Mine is called `debouncer.vhd` and it is in the directory above. So click in the '...' box, and navigate to the directory containing that file. There is no need to check the 'package as a library core' button, though we may come back to what this button is for later. Click 'Next'.

You next want to create a directory where the packaged IP will eventually be written. We want this to be in the local copy of our git repository. So, in the shell,

```
cd /home/physics/my_ip
mkdir -p guiwidgets/debouncedbutton
```

Leave the contents of the 'Project location' box at the default; this is where Vivado creates the temporary project it uses to build the GUI. Click 'Next' again. Then, click 'Finish'.

Vivado will then launch a completely separate project to package up the debouncer into a GUI widget. It might take a minute or two and the computer fans may come in. It seems to have quite a lot to do - who knows what. Anyway, what you see next should look like Figure E.4.

The thing to look at here is the main pane, containing the 'Package IP - debouncer' tab. This contains some fields that you might care about if you were going to sell your widget to someone. Here we will leave most of them as defaults. The one thing we will change is the Display name, because these _v1_0 labels will appear in all our GUIs if we don't. Alter debouncer_v1_0 to just plain debouncer.

Next, click on the menu item to the left of this tab called 'customisation GUI'. this is what your tool will look like when you incorporate it into a graphical project. The GUI widget for my debouncer is shown in Figure E.5. Inputs

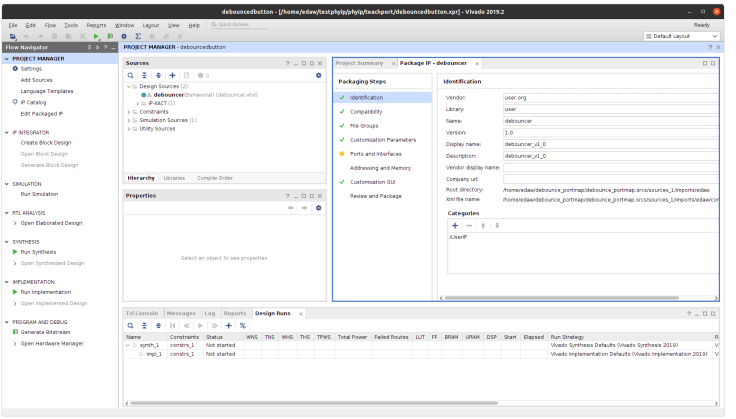


Figure E.4: The sub-project containing the source for the widget export. Widgets are called IP by Vivado - they are thinking that you are going to become a developer and sell these things...maybe you will!

are on the left, outputs are on the right. There is lots of flexibility built in to this gui creation tool, since it is probably also used to create the IP for processors, memory, sophisticated bus devices, etc. But for us it is sufficient to create something very simple.

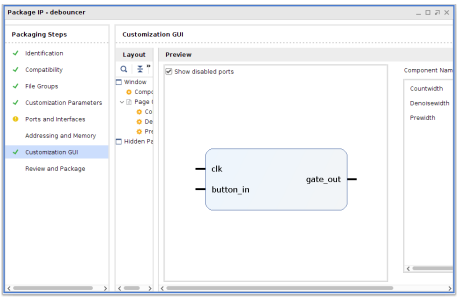


Figure E.5: This is the graphical representation of the debouncer widget. Inputs are on the left, outputs are on the right.

If all seems well, then click on the 'Review and Package IP' item in the vertical bar to the left of the GUI sketch, and select package IP. It should say that it has finished successfully, and ask you if you want to close the project - select Yes. You can now also close the project containing the port map 'container' that you used to test port mapping.

Now to put your newly created IP in the git repository. In a terminal window, go into the sources directory where the VHDL code you just created is located.

```
cd .../<project name>.srcs/sources_1/imports/<username>
ls
```

In addition to `debouncer.vhd` you should now be able to see a file called `component.xml` and a directory called `xgui`. This whole structure needs to be copied into your local copy of the GIT repository

```
cp -R * /home/physics/my_ip/guiwidgets/debouncedbutton/.
```

Next move to the local copy of the git repository, add the new files, commit, and push to the git repository

```
cd /home/physics/my_ip
git add -A
git commit
# enter what you are changing in nano,
# then ctro-O, return, ctrl-X
git push -u origin main
```

You should now refresh your browser view of the git repository and go and take a look at the files you have uploaded under the new directory path to guiwidgets that should have appeared. One nice thing about VHDL is that when it exports IP cores like this, only a very minimal set of files is dumped to the IP core directory. These files include the VHDL source, a description of functionality in the ascii text format XML, and a script in a language called Tcl that is used by Vivado in incorporating your IP core into other projects. You don't have to worry about exactly what these files are, but you might wish to take a look at them in the web git interface, just for interest.

Appendix F

Lab Exercise 5 - Using XADC

In todays lab we will use one of the IP modules built in to Vivado. Like many chips, the Artix7 FPGA has some built in capability for analog to digital conversion. This is intended to read out analog signals used as diagnostics for the chip. The two quantities usually probed by the ADCs are the temperature of the chip and the voltages at the supply rails. However, if we choose to we can route the ADC inputs to external ports of the BASYS3 board and use them to measure external signals. You will have noticed the four 12-way connectors, two mounted no each end of the board. These are called PMOD (peripheral module) connectors. Digilent make many peripherals for the PMOD standard; the DAC card that I supplied with the BASYS3 is one of them. However, one of these PMOD connectors can also be used to input signals onto the inbuilt ADC. The connector is labelled JXADC and it is to the left of the four digit display.

When using the JXADC port for analog input, the wiring is as follows. The four pins (in two pairs) nearest to the JXADC label, labelled with '3V3' and 'GND' are supply rails used to power expansion connectors. The other 8 pins can be thought of four banks of two pins each, arranged on the connector in four vertical pairs. For the purposes of analog input, the upper of this pair is the + input and the lower one is the − input.

Analog signals come in different varieties. In physics labs, most cables carrying signals are so-called BNC coaxial cables. These are the black lab cables you connect to oscilloscopes, usually they have RG58 written on the side, which is a class of cable having standard characteristics. Signals on such cables are single ended, which means that the inner conductor at the core of the coax carries the signal, and the outer conductor which forms a braided sheath around it is connected to ground. Typically the inner conductor can carry a voltage which is either at a positive or negative voltage with respect to the outer grounded conductor. Coaxial RG58 cables of the type found in physics labs can carry

signals at high frequencies, up to a few GHz and are very mechanically robust.

It is also quite common to have differential signals. Here again you have two conductors, but instead of one of them being connected to ground, they are both floating, and referred to as the + (non-inverting) and − (inverting) input. The signal is the difference in voltage $V_+ - V_-$. Ideally the non-inverting input and the inverting input would always be carrying equal magnitude but opposite sign voltages. The common mode voltage, which is the average of the two voltages would be zero. If you want really low noise, you then twist the two signal carrying conductors together and surround them with a grounded sheath. This way of carrying signals is called twisted pair and ground. The advantage of this arrangement is that it keeps common mode signals to a minimum, and the grounded sheath can have a break in it which breaks any circuit that may be created in the ground path between two connected instruments.

In low cost digital circuits, such as on the BASYS3 board, there is rarely any negative voltage rail, and so it is not possible to have a negative voltage read out on the board. So, the XADC inputs have two possible modes, single ended and differential. In single ended mode, you need to connect the lower '-' input of the pair carrying the signal to GND. The upper input carries the signal. To be read out correctly, this voltage must be positive. It cannot be greater than 3.3V because that is the positive voltage rail for the chip, but it may also turn out later that in reality the voltages that can be read out are between 0V and some other positive voltage that is lower than +3.3V. Between 0V and the full scale voltage, the analog input will be converted to a 12 bit binary unsigned output, where binary zero corresponds to 0V and all ones corresponds to the maximum voltage that can be read out.

You can also configure XADC to read a differential signal but beware - because no negative voltages can be read out, the only way to have a differential signal read is to add a common mode voltage big enough so that the biggest negative voltage anticipated as a signal on either wire, plus the common mode voltage added to both signals, will not be negative. So, for example, suppose you have a differential signal where either the inverting or the non inverting wire can potentially have a voltage of $-0.5\,V$ relative to ground. You then need to add a common mode voltage to both inputs of at least +0.5V relative to ground, otherwise your signal will be distorted by the ADC.

F.1 Analog Input, LED Output

Start a new Vivado project called xadctoled. Import the usual BASYS3 constraint file. Do not edit the constraint file yet. As usual, specify the `xc7a35tcpg236-1` device. Click on the link close to the top of the left hand Flow Navigator labelled IP Catalog. A new tab should appear in the large project pane with the top of a very long list of built in IP. Type XADC into the Search box in this pane and select XADC Wizard by double clicking on it. A large IP customisation block appears, shown in Figure F.1. The xilinx IP blocks are highly

configurable. There are many controls and five tabs to go through. Fortunately we will mostly select the defaults. Set things as follows:

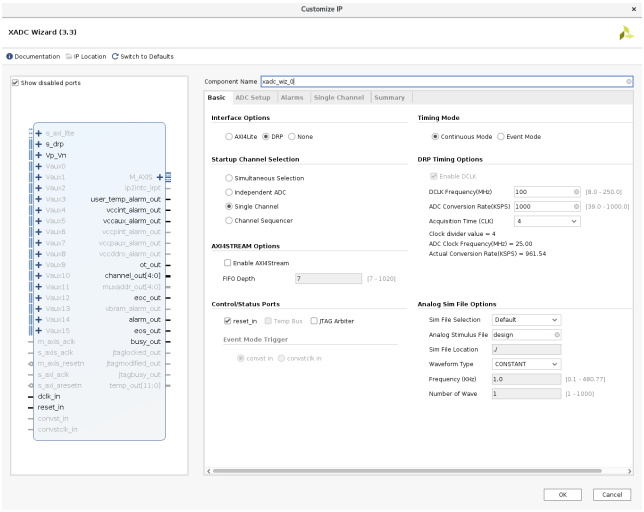


Figure F.1: The XADC wizard for customising the configuration of the internal 12 bit analog to digital converter.

1. In the Basic tab, leave everything as it is. Click on the ADC Setup tab.

2. In the ADC Setup tab, again leave everything as it is. Select the Alarms tab.

3. In the Alarms tab, we won't be using the XADC to set alarms on overtemperature or supply voltage. You should uncheck all four of the alarms that are on by default. Notice that as you do this, some of the ports on the IP block for XADC become greyed out. Click on the Single Channel tab.

4. We will an external pin connected to one of the internal analog to digital converters as the input, so change single channel from its default of TEMPERATURE to one of the options lower down, called `VAUX14`. Leave everything else alone. Select the Summary tab.

5. Your summary should read that you have selected the DRP interface, the XADC is operating in single channel mode with the AXI4Stream interface false and continuous timing. The input clock is 100MHz, there is no channel averaging and no external MUX (multiplexing). If all this looks good, click OK in the bottom right hand corner.

6. In a window that appears called 'Generate Output Products', accept all the defaults by just clicking Generate. You'll also need to click OK when it announces that it has created an 'out of context run'

Next we will connect up the ADC to the LEDs using our own VHDL top level file and the port mapping technique that we used last week. Create a new VHDL file and call it xadctoled.vhd. For ports you will need a clk input and the LED outputs (all 16 of them). You will also need to enable ports L3 and M3, which correspond to the second port along from the lower end of the connector nearest to the switches, enabled and named as follows

```
set_property PACKAGE_PIN L3 [get_ports {vauxp14}]
        set_property IOSTANDARD LVCMOS33 [get_ports {vauxp14}]
set_property PACKAGE_PIN M3 [get_ports {vauxn14}]
        set_property IOSTANDARD LVCMOS33 [get_ports {vauxn14}]
```

In the VHDL source code you will need to make a port map to the `xadc_wiz_0` device. Here is most of the VHDL code you will need.

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

entity xadctoled is
    Port ( clk : in STD_LOGIC;
           led : out STD_LOGIC_VECTOR (15 downto 0);
           vauxp14 : in STD_LOGIC;
           vauxn14 : in STD_LOGIC);
end xadctoled;

architecture Behavioral of xadctoled is
  signal enable, ready: STD_LOGIC;
  signal drpaddress: STD_LOGIC_VECTOR(6 downto 0);
  signal dataread: STD_LOGIC_VECTOR(15 downto 0);
  signal databuffer_d, databuffer_q: STD_LOGIC_VECTOR(15 downto 0);
begin
  process(clk, databuffer_d) begin
    if(clk'event and clk='1') then
      databuffer_q <= databuffer_d;
    end if;
  end process;

  -- select data channel
  drpaddress <= "0011110";

  -- port map for the ADC
  myadc: entity work.xadc_wiz_0
  port map (
    daddr_in => drpaddress,
    dclk_in => clk,
    den_in => enable,
```

```
    di_in => "0000000000000000",
    dwe_in => '0',
    busy_out => open,
    vauxp14 => vauxp14,
    vauxn14 => vauxn14,
    vn_in => '0',
    vp_in => '0',
    alarm_out => open,
    do_out => dataread,
    eoc_out => enable,
    eos_out => open,
    channel_out => open,
    reset_in => '0',
    drdy_out => ready
    );

    -- WARNING: THIS CODE WILL NOT RUN WITHOUT MODIFICATIONS.
    -- here I have left out some code that you will have to
    -- figure out for yourself. First, whenever the ready node
    -- goes to '1', you want to copy the dataread node to the
    -- databuffer defined above in the clk process, but when
    -- ready is '0' you just want that buffer to retain whatever
    -- it was already storing. Secondly, the buffer output needs
    -- to be connected to the LEDs.

    end Behavioural;
```

Once you have incorporated the above code in your project, you should see the `adc_wiz_0` IP modify to be a subelement of the main xadctoled program. You should also be able to synthesise, implement, and generate your bitstream.

F.2 Running XADC

To test your ADC, deploy your code on the board. You will need to wire up the XADC connector to test it. I suggest that you start by just using a couple of wires to connect the input pins of the ADC directly to the voltage rails. A diagram showing the relevant pins of the JXADC connector, viewed looking in to the connector port, is shown in Figure F.2

Connect the V_- terminal to either of the GND terminals with a wire, and connect one end of the other wire to the V_+ terminal leaving the other end unconnected for now. Connect the other end of this second wire first to the GND terminal. Now the analogue voltage at both ADC inputs should be 0V. However, you should see that about 7 or 8 of the LEDs are lit. The lowest 4 bits are always on because the ADC output is actually only 12 bits, so the four least significant bits are not carrying any information, they are just always

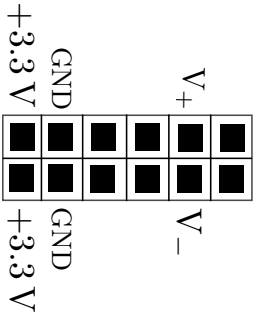


Figure F.2: The pins of the JXADC connector, viewed from the direction looking inwards in to the connector port, showing ground, supply voltage, and the non-inverting and inverting inputs for ADC channel `VAUX14`.

high. There are then about 3 or maybe 4 more bits that should be zero, but appear to be lit anyway. This is because these bits are full of noise. ADCs usually have noise in about their lowest three or four bits. They are imperfect devices. Now connect the V_+ terminal to +3.3 V instead. Now all the LEDs should light up.

The next experiment is to connect the potentiometer to the ADCs. Do this carefully. Leave V_- connected to GND, but remove the lead connected to V_+. Connect the two end leads of the potentiometer to +3.3 V and GND, and the middle potentiometer lead to V_+. You should now be able to vary the voltage on V_+ between 0V and 3.3V. What do you see?

Using a voltmeter, if you are in the lab, you should be able to measure the voltage on the signal pin of the potentiometer, and determine the voltage at which all the LEDs are first lit. Hint: it's not 3.3V. The full scale deflection of the ADC occurs at a different voltage.

F.3 A tone generator

You should all have crystal earpieces. These can be driven directly by a digital output of a BASYS3. In this project you will generate the 'A' below middle C, audible through your crystal earpieces. The frequency of this tone, well known to string players because it is traditionally used to tune violins, violas and cellos, is 440 Hz. You therefore need to generate a square wave at this frequency. To do this, calculate how many clock cycles there are in half a period of such a square wave. Make a counter that has enough bits to reach at significantly higher than this number of counts, maybe 4 bits higher. Now, arrange so that when this

counter reaches a half period, a `STD_LOGIC` register is modified to the `not` of whatever it was before. The output of this register will therefore go through a full cycle in one period of the wave. By connecting this register output to one of the other PMOD ports (see the constraint file to find a suitable one - the other three PMOD ports are labelled JA, JB and JC), you should be able to connect the crystal earpiece between this port and ground and hear the tone you have made.

F.4 A tuneable tone generator

Now for the clever bit. Make a tone generator whose frequency is controlled by the potentiometer. You have a 12 bit number which you can control with a knob. Make this number control the count that your counter has to reach before resetting, otherwise it's the same as the tone generator project above. You now have made your board convert a signal from analog to digital, and then back from digitial to analog. Tone generators that have a varying pitch with voltage are in fact used quite a lot in diagnostic equipment. For example, in a leak checker, an operator squirts helium gas at some pipework. Rather than having a display that shows the helium level reaching the sensitive detector, the helium level is used to control a variable pitch speaker, so that the operator can focus on squirting the helium in all the right places and just 'hear' when they have found the leak.

F.5 Graphical design tool practice

The XADC widget can also be used in the graphical designer. Make a new project with an empty graphical design, and add the XADC wizard as IP to the graphical design. Make the same choices of settings as in the port map project. Once you have your XADC design element in the graphical design, you'll need to create your own IP to contain the 16 bit register to store the output, and also wire in the constraints file correctly to connect the external ports. I'm deliberately leaving these instructions vague so that you are challenged to go back to last weeks work and recall how to use the graphical tool. See me or Mitch if you are unsure, though.

Appendix G

Lab Exercise 6 - Writing a Serial Driver

In this lab exercise we will write a driver for the PmodDA2 module, a two channel digital to analog converter that plugs in to any one of the PMod expansion ports on the BASYS3 board.

As we discussed last Tuesday, even the quite simple serial protocol that is used to drive the PModDA2 requires a reasonably complex set of elements to work. And, at the moment, we can't use a great deal of lab diagnostics. Consequently, we shall develop the code for driving the PModDA2 using the *behavioural simulator* tool incorporated into VIVADO.

G.1 The Pmod Driver - Ingredients

You will need the following ingredients

1. A fast counter with 2 bits that counts from 0 to 3 then resets. The lower bit should change state every rising edge of the `100 MHz` clock that is built in to the `BASYS3` board.

2. A slow counter capable of counting from 0 to 16 inclusive, that increments its value every 40ns.

3. A signal that is the not of the most significant bit of the fast counter. This signal will be used as the `DACCLK` signal to act as the clock for the BASYS3.

G.2 Stage 1 - making the fast counter

Create a 2 bit counter that is item 1. You should know how to do this from projects in previous weeks, but ask if you get stuck. Create a single bit signal

that is the not of the most significant bit of this counter, and connect it to the correct pin of the JA PMOD connector (upper row). You'll have to look at the reference document for the PmodDA2 to figure out which pin this is. Once you've got this to the state where you think it should work, check that you can go through the synthesis, implementation, and bitstream generation stages successfully.

If you are in the lab, connect the oscilloscope to the pin on JA where you think you connected the DACCLK signal. If you probe between this pin and a neighbouring ground on one of the oscilloscope channels, you should be able to see a 25MHz sinewave. Using oscilloscopes requires practice. Ask if you can't get it to work.

G.3 Stage 2 - Simulating the fast counter

We're going to run a behavioural simulator on your code containg the simple 2 bit counter. For this bit it is important to ensure that the internal clock registers initialised to values. The syntax to do this on a pair of register signals is. To initialise two bit signals to zero you would add, for example, `:= to_unsigned(0,2);` after the `(2 downto 0)` in the `signal` statement for the registers. Note the colon before the equals sign; this is part of the code too.

bf Be sure you have run the script that Mitch will supply at the beginning of the lab; it makes some changes to your laptops that fix some problems we had that were causing the simulator to fail on the laptops. Once you have run this code, you will have to quit out of vivado and restart your laptop to ensure that the changes propagate through to all the places where they are needed.

Start the simulator by finding the `Run Simulation` item in the Flow Navigator on the left of the VIVADO window. Select 'Run Behavioural Simulation.' Figure G.1 shows what the whole VIVADO project window should look like after you start the simulator. The simulation appears in a new tab of the main window. If you have done things right, all the signals except clk (in the column on the left) should be green. If you click on the underscores in all the other panes in the VIVADO window, this will make the simulation pane bigger so it is easier to see the output, so go ahead and do this. You can get the other tabs back by clicking on the small elongated rectangles that the other tabs collapse to around the edges when you use the underscore buttons. Now the simulation should occupy the whole VIVADO window as shown in Figure G.2.

The `clk` signal is orange because its state in the simulation is undefined. To set this signal as a 10ns clock, you right-click over the 'clk' in the simulation and select 'Force Clock'. In the window that appears, you set the Value radix as binary (this just tells the simulator that you want to display its value after simulation as binary), the leading edge and trailing edge signals to 1 and 0 respectively, leave the Starting time after offset at zero ns, leave the cancel box blank, leave the duty cycle at 50%, and set the period of the clock to 10ns (matching the clk signal on the BASYS3 board). Then select OK.

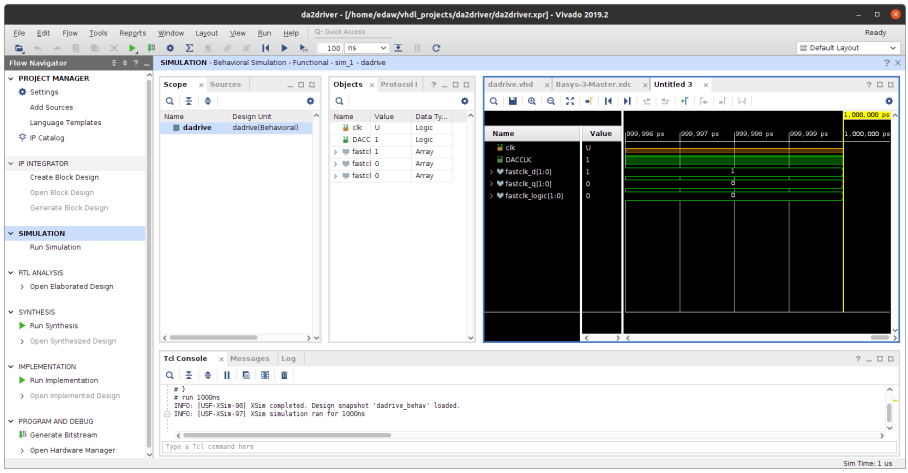


Figure G.1: The VIVADO screen after starting the simulation

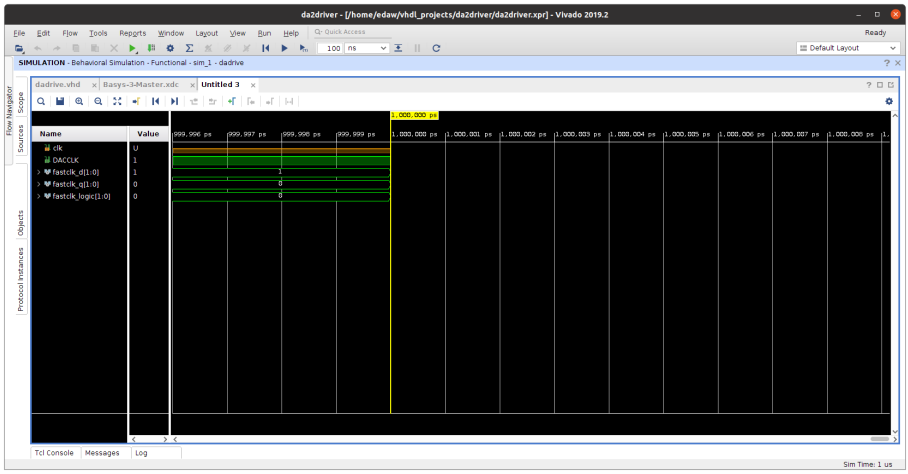


Figure G.2: The VIVADO screen after minimising the other panes

To run the simulation for, say, 200ns, go to the box in the middle of the top of the Vivado window, where the time 100 ns is the default, and change to 200, leaving the ns as it is. Now, immediately to the left of the 200, there is a small arrow to th eright with a (T) under it. This is a timed simulation. Click on this arrow once. The simulator output should resemble Figure G.3. It doesn't look very interesting because the display shows only the last 10ps of the simulation. You will need to zoom out using the magnifying glass with a minus sign icon in the top left of the simulation pane. You can also use the horizontal scroll bar below the simulation output pane to position the piece of the simulation you are interested in appropriately. Note that the simulation tool by default does nothing until 1000 ns, displayed as `1,000,000 ps`, so if you only ran the simulation once the time period you are interested in is between `1,000,000 ps` and `1,200,000 ps`. Click on the > to the left of the signal for the output of the clock storage register so you can see the display of the binary bits of the clock as well as the decimal value.

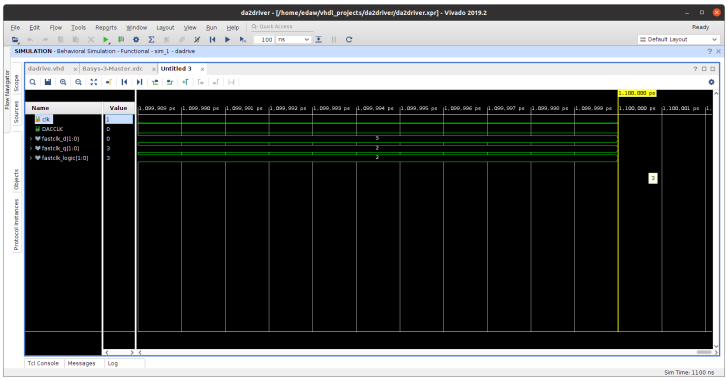


Figure G.3: Simulation results before rescaling

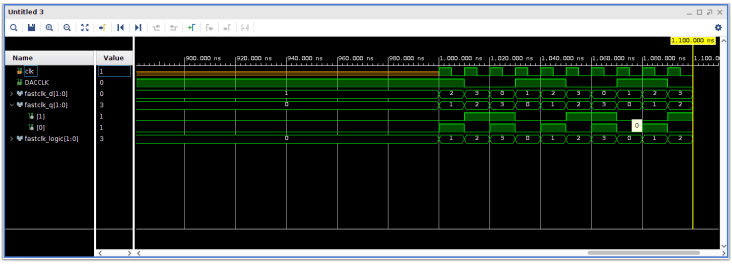


Figure G.4: Simulation results after rescaling

Notice in the simulation results that the DACCLK signal is in antiphase with the most significant bit of the 'Q' output of the clock register, in my

simulation called `fastclk_q`. Notice also that the falling edge of the DACCLK signal, which will be the signal to the PModDAC2 board to read the next data bit, occurs when the `fastclk_q` signal is transitioning between 1 and 2. If everything looks right, close the simulator. If things don't look right, try and figure out what you haven't done right in your VHDL code, or ask for help. Note that if you alter your VHDL code and want to re-simulate, you'll need to close the simulation using the blue cross in the top right hand corner, and re-start it again.

G.4 Slow clock and alignment

We have a total of 16 bits of data to send, and we also need to rest the DAC chips by setting $\overline{\text{SYNC}}$ to high for a single period. Therefore we need a slow clock that counts from 0 to 16 and then resets and does it all over again. Because this is 17 cycles, we cannot use the natural overflow of a binary counter to do the reset, and we will have to do it using a conditional assignment.

Figure out how many bits a binary counter needs to be able to hold the numbers 0 to 16 inclusive, and create a slow clock buffer that turns over when the fast clock transitions from 3 to 0. It should reset to 0 after it reaches 16. See if you can figure out how to code this up in your project. Don't forget to initialise your new registers to zero in the signal statement and include the appropriate register signal in the sensitivity list for the clk edge process. Don't forget that if you use a conditional assignment for setting the next value of the slow counter, Then, re-simulate and see if your simulation output looks like that in Figure G.5.

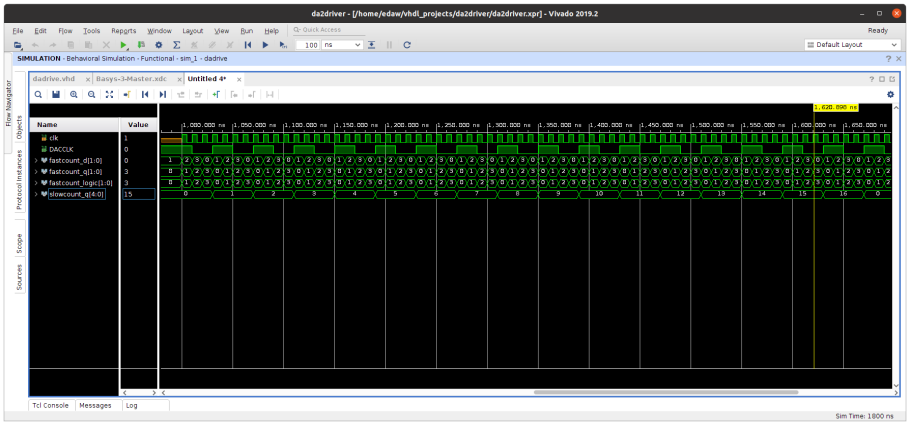


Figure G.5: Simulation results with the slow counter added

Notice that the falling edges of the DACCLK signal occur half way thorough the cycles of the slow counter, and that the slow counter counts from 0 to 16

inclusive then resets. Check that your simulation output has this behaviour and ask for help / debug it yourself if it does not.

G.5 Enable signal

Next is the $\overline{\text{SYNC}}$ signal. This signal should be high when the slow counter is zero and low otherwise. Implement this in VHDL and check with the simulation.

G.6 A ramp waveform generator

A ramp waveform rises linearly from zero to a maximum voltage, then resets quickly to zero, and the cycle repeats. If we drive our DAC with a counter, then we can make a ramp generator. The DAC has 12 bits, so there are 2^{12} or 4096 counts to go from all 12 bits zero to all 12 bit ones. If you want the ramp to have a period of 1 second, then the count must increment roughly every 1/4096 seconds or 0.24 ms. This is about 24,400 10 ns clock cycles. So, one way of making the ramp of roughly the right period is to have another fast counter with the right number of bits to reset roughly every 24,400 clock cycles. The required number of bits is $\log_2(24400) = \log_{10}(24400)/\log_{10}(2) \sim 15$ bits. Therefore add to the code a counter with 15 bits that is incremented every cycle. Also create a 12 bit data buffer, again using an unsigned data type, and increment this data buffer by 1 only when the 15 bit counter is zero. This way, the 12 bit counter should contain a digitised 12 bit ramp waveform with a period of order one second.

Create two external ports, called `DATA1` and `DATA2`. Wire `DATA2` permanently to '0' as we are not going to use it. `DATA1` will be the serial port output of our ramp generator. Refer to the PmodDA2 reference guide under 'Interfacing with the Pmod' for the contents of the 16 successive bits read. The bits are fed in from MSB first to LSB last.

Therefore, `DATA1` must be 0 when the slow counter register output is less than or equal to 4 (when the slow counter register is zero, that's when the SYNC signal is high; when it's between 1 and 4 inclusive, that's when the unused 4 bits of data are expected at the DAC. On slow counter values 5 to 15 inclusive, bits 11 to 0 of the databuffer register output should be connected to `DATA1`. So the number of the bit you are sending is 16 minus the value of the counter, for counter values starting at 5 and ending at 16. A convenient way of writing the code would be to use the value of the counter itself to index the array of bits making up the data buffer.

Using a logic vector value as an index for an array is not something we have covered yet in VHDL. This is the mechanism. The value of the counter is cast using `to_integer`. VHDL presumably chooses an appropriate bit width for the hardware to select which bit. The syntax is shown below in the code listing. The code listing assumes that the buffer containing the data is called `datacounter_q`, and the buffer containing slow counter is called `slowcount_q`.

You will also need a signal with 12 bits that is a cast to `std_logic_vector` of `databuffer_q`, so that you can extract single bits from the data. Don't forget that bitwise operations in VHDL only work on logic or logic vector data tyles. Let's call the 12 bit wide `std_logic_vector` signal `datalogic`

```
-- cast contents of databuffer_q to a signal for a logic vector
-- of the same length.
datalogic <= (std_logic_vector)databuffer_q;
-- slowcount_q has 5 bits and you are trying to set it to 5, hence
-- two 5s. The second line contains the array indexing. You could
-- also use a huge conditional assignment and deal with each
-- possibility for the slow counter separately. This is more typing,
-- but just as efficient code-wise.
DATA1 <= '0' when slowcount_q < to_unsigned(5,5) else
            datalogic(16-to_integer(slowcount_q));
```

Once you think you have the `DATA1` output behaving correctly, and the ramp being generated with the right period in `datacounter_q`, use the simulator again to check its functionality. This time you'll need to simulate for enough cycles so that the value of `databuffer_q` has a chance to increment a few times. Once it gets a value far from zero, you'll be able to examine the value of the bits appearing serially in `DATA1` to see if the stream of bits matches the value of the databuffer. So, for example, if databuffer was `0x3a` then this is $3 \times 16 + 10 = 58$. In binary this is $B000000111010$ which is $2 + 8 + 16 + 32 = 58$. I ran a long simulation (just adjust the simulation duration to something like 20ms) then zoom out on the far end (right hand end) of the simulation, and click somewhere in the window. A vertical yellow line appears. Now use the magnify-+ button to zoom in on the neighbourhood of that line. Note the value of the data buffer, and then see if the bit pattern in DATA1 matches up. In Figure G.6 I have shown my simulation at around 18.95 ms after the start.

G.7 Connecting up the PMOD chip

If you are at home, you won't easily be able to check that the hardware connected to the PMOD port actually works as advertised. However, there is one way you can do this - if you reduce the number of bits in the ramp generator counter from 15 bits to 6 bits, then an earphone connected between pin 1 (analog output 1) and pin 5 (ground) on the output of the PMOD card connected to the BASYS3 (you can just plug it in then build and deploy the code if the simulator predicts it will work correctly), will make an audible tone.

For a more challenging readout, use 15 ramp generator counter bits, not six as for the earpiece output. You can then import the ADC from last weeks project, wire it up to the LEDs, and connect two cables, one between the PmodDA2 ground and the inverting input to channel 2 of the XADC input,

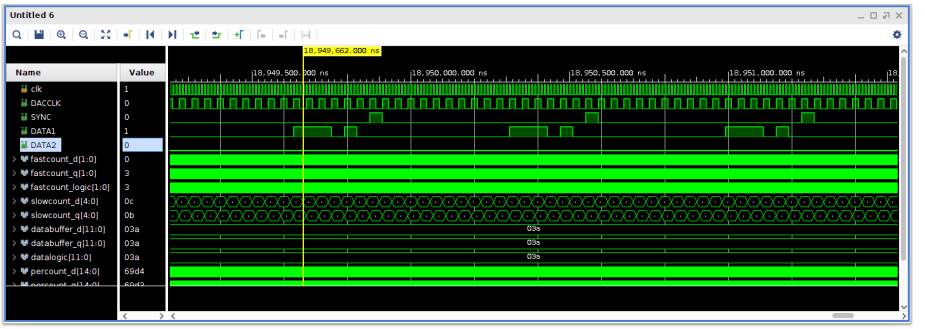


Figure G.6: Simulation results showing the serial port. You can see that the value stored in datalogic is `0x03a`, which as hexadecimal for 58. The value in `DATA1` has its least significant bit just before the SYNC line goes high. The first few bits from LSB to MSB are $0, 1, 0, 1, 1, 1, 0, 0$ and the rest of the bits are zero. This indeed corresponds to decimal 58, so this appears to be working.

and the other between the PmodDA2 analog output 1 (pin 1) and the non-inverting input to channel 2 of the XADC. You can then internally connect the digital output from XADC to the LEDs and watch them light up to represent the ramp signal as it evolves.

Finally, if you are in the lab, you can connect an oscilloscope probe between pin 1 and pin 5 on the PModDA2 output connector, and see if you can record the ramp on an oscilloscope trace. If you are going to do this, you might want to reduce the number of bits for the counter to 6 so that the ramp period is less than 1/100 of a second, as the oscilloscope will be easier to handle if the waveform is faster.

Appendix H

Lab Exercise 7 - Embedded Processor Cores

In this exercise we will install a microblaze soft processor core on the BASYS3 board, then program the core in the 'C' language that we started to learn in the last seminar.

H.1 Hardware installation

First, the hardware. Start Vivado as usual, and select 'Create project'. Everything follows as it did for previous exercises at the start. Skip adding any source or constraint file. When you get to the 'Default part' page, you will notice that near the top of the window there are two tabs, labelled 'Parts' and 'Boards'. The 'Parts' one is lit up by default. You need to instead select the 'Boards' tab. When you do this, you should be able to scroll down and highlight 'Basys3', then click next and finish. The project summary should now look roughly as in Figure H.1, up to the paths to things, which will be different for you.

Next select 'Create Block Design' and accept the default name (project_1). In the empty diagram press the button to add IP, and in the menu that appears search for MicroBlaze. You want the one at the top that is just MicroBlaze on its own. Double click on it and it shows up in your window. Next, you'll need a UART interface, so you can connect your laptop to the processor to either make input or see output. So, click on the '+' button at the top of the design window. The same menu appears again. This time, you do a search for UART. This stands for 'Universal Asynchronous Receiver Transmitter' - it's a protocol for two devices talking to each other where there is no handshaking, ie, both devices can send at any time no matter what the other device is doing. You want AXI Uartlite. Double click on this and it will also appear in your block diagram.

You will need one other piece. This is the 'system clock'. This is board

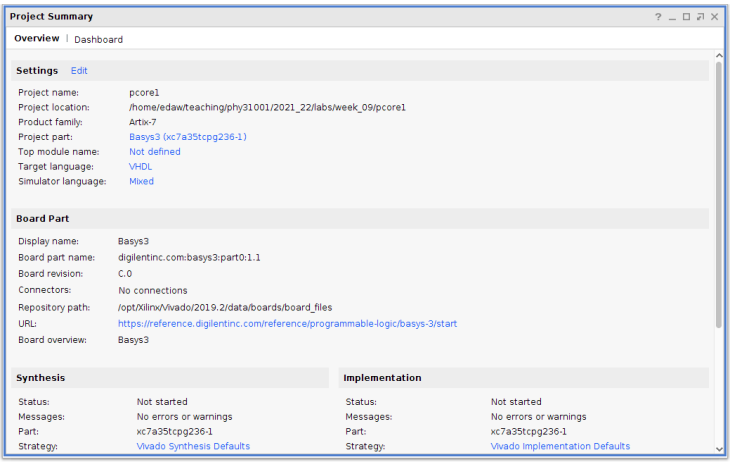


Figure H.1: The project summary after setting up for use with the Basys3 board.

dependent, because the clock is supplied by an external crystal, which in the case of the BASYS3 board is a MEMs device that produces a single ended output, meaning that the clock signal is just a square wave on a single wire. The board tools in Vivado assume a professional-grade differential clock signal, so you need to defeat the tendency of the connection automation tools to make everything differential. TO do this, go to the 'Board' tab under where it says BLOCK DESIGN (other tabs are sources, design, signals...). In the board tab you'll see features of the board rather than the FPGA. Drag the top one labelled System Clock onto the block diagram.

At this point you have three pieces of hardware but no idea how to connect them together, or indeed how to custommise them where necessary. For the first 'Hello World' project, you probably don't need to change anything. However, the default amount of memory assigned to the processor core is tiny, so I'd like you to increase this. Highlight the microblaze core (it turns orange), and click on the blue link above it in the window labelled 'Run Block Automation'. In the window that appears, change the local memory from 8 kB to 128 kB. Now your programs are unlikely to run out of memory when deployed to the core. Change nothing else and click 'OK'. This will induce the appearance of several more blocks in the design, a 'Clocking wizard', a unit called 'Processor System Reset', a debug module, and some 'Microblaze local memory'. Your diagram should now look something like that in Figure H.2

You'll now notice that there are some remaining unconnected subsystems. Most significantly, there are no connections to the clock on the board or the USB uart (serial port) output. However, the remaining link at the top of the window, called 'Run Connection Automation' can take care of this. This is because the board support package for Basys3 contains much information about non-FPGA

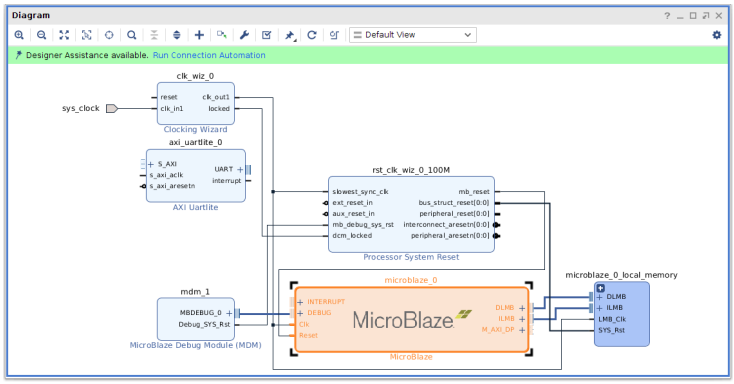


Figure H.2: The block design after block automation, but before connection automation.

components on the board, so that using this support package, Vivado can ensure that the various components needed to support the microblaze core are found and connected up correctly. Click on the 'Run Connection Automation' link and check the 'All Automation' box, to connect everything together. More wires and blocks should appear. A notable one is the AXI Interconnect. AXI is the bus that is used as a general purpose connecter to peripherals, in the same way as the PCI and PCI express buses are used to connect memory, graphics cards, network cards, etc, to the microprocessor inside a PC. It is needed here to connect the UART controller to the Microblaze processor core.

There are a couple of final steps before we are done with this diagram. First, check that everything that needs to be connected, is connected. A rudimentary diagram checker is built in to Vivado - it's the box with a tick mark in it above the diagrm window. Click on this, and you should see a window appear where it says 'Validation succesful....'. Click 'OK'. If you don't see this, then check you have carried out all the steps described above. There is also a tool for tidying up your block diagram so that it looks less like a plate of spaghetti. This is the small arrow going around in a circle icon that is above the block design. Click this and nothing changes with the wiring, but things are moved around so that they look a lot tidier. See Figure H.3. You'll notice that some more extra modules have appeared.

One more thing that is worth looking at. The AXI bus is more complicated than the simple parallel and/or serial buses with which you are already familiar. It's an example of a memory mapped bus. This is needed when there is the potential for more than two devices to be connected together by a bus. The obvious potential for data to end up arriving at the wrong device is guarded against by associating a memory map with the bus. Each device on the bus occupies a pre-determined set of addresses in this memory map, so that if you want to communicate with a particular device, you just read from, and/or write

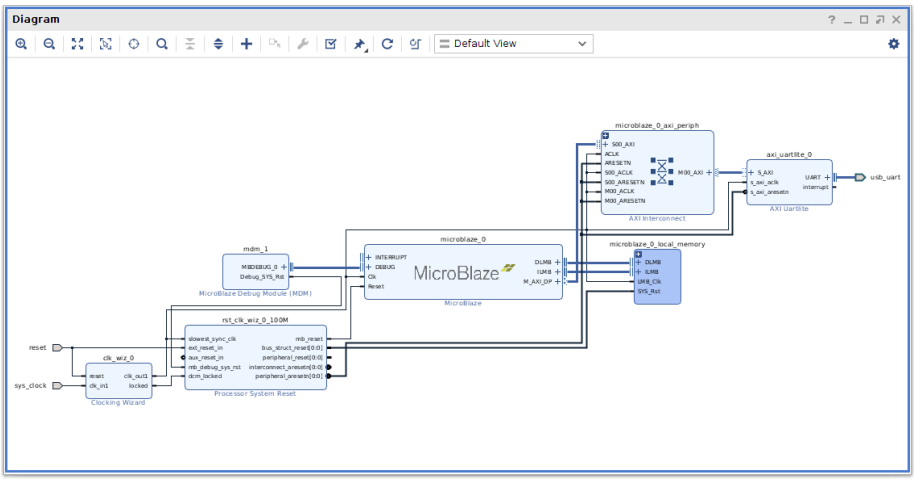


Figure H.3: The block diagram after connection automation has been run.

to, addresses associated with it. If you click on the 'Address Editor' tab above the diagram, you'll see that there are 32 bit address buses associated with data and instructions for each of the memory controllers and the UART device. This is shown in Figure H.4.

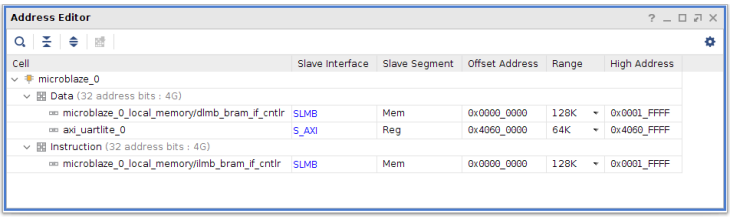


Figure H.4: Address space of our soft core hardware

If you wanted to, you could edit both the Offset address, which is the address where the memory mapped to the device starts (sometimes called the base address of the device), and also the size of the space it occupies. In this case, we want to leave it alone! Needless to say when we come to adding shared memory, this will have a base address on the bus too, which we will need to know to write programs that use this shared memory. Save the block design by using 'Save block design' under the File menu at the top left of the Vivado window.

H.2 Hardware build

So far we have been laying out the hardware to support the soft core. Now we need to build it. Under the hood, all of the blocks that we have been dealing with are in fact VHDL modules connected together with port map syntax. The compilation is of the VHDL, not of the graphics. Consequently, like every VHDL project, we need a top level VHDL program which contains the port map statements that actually connect everything together. There will also be some numerical inputs added, specifying, for example, the memory size for the MicroBlaze core, and anything else in the set of components that we have added which is in need of customisation. To create this file, go to the pane directly under where it says BLOCK DESIGN - design_1. One of the tabs reads Sources. Click on this tab to select it. Under 'Design Sources (1)', you should see an entry for design_1. Right-click on this and in the ensuing window select 'Create HDL Wrapper...'. Select the default of allowing Vivado to manage wrapper and auto-update; this means that if you make changes to the diagram, those changes will automatically result in corresponding modifications to the wrapper. Click 'OK'. After a short pause, the hierachy under Design Sources should have a top level object called design_1_wrapper. This is actually a VHDL file. You don't need to know what is in it, but double-click on it if you are interested. Your diagram is still behind the code that appears, it's just in a tab called Diagram.

From now on, hardware wise, it's back to how we are uesd to doing things. Click on 'Run Synthesis'. Make sure you enable all 4 cores on your machine for synthesis as this time Vivado has to build all the components in your diagram, including the soft core processor, and this will take some time. If you click on the 'Design Runs' tab in the bottom window and expand all the items in the list, you'll get progress bars for each of the sub-builds as they occur. You should see more than one object being built at the same time. The view in this window with everything expanded is shown in Figure H.5

After synthesis, if all goes well, then you should be able to go ahead and run implementation, then generate bitstream, in the usual way. When you have generated the bitstream, you will need to click on 'Cancel' because we don't want to open the hardware manager quite yet.

H.3 Exporting your hardware design

Finally, we need to export the bitstream to the software development kit, where it will be combined with the program to be run on the MicroBlaze core. To do this, in the File menu in Vivado, select 'Export' and then 'Hardware'. In the window that appears, check the box that says 'Include bitstream' and hit 'OK'. Notice that the name of the file it produces is 'design_1_wrapper.xsa'. This is the file we will need to point to when we launch Vitis, the software development kit where we will produce our 'C' program. Leave Vivado open.

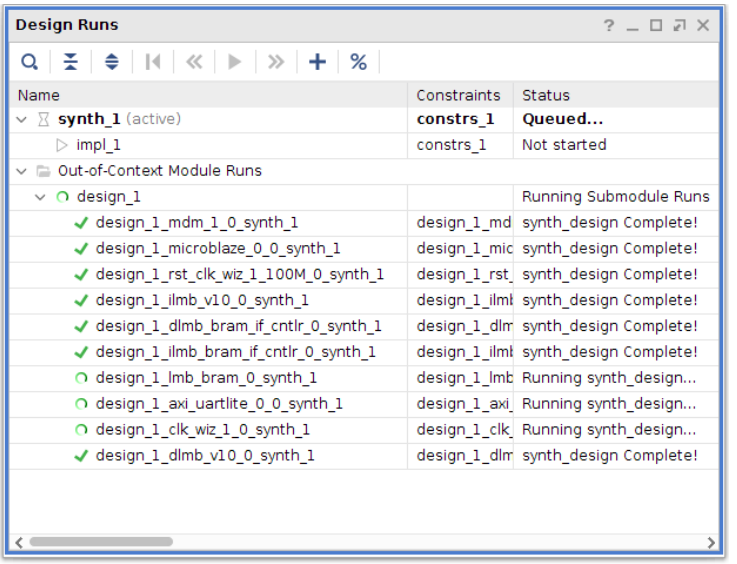


Figure H.5: The Design Runs tab during the soft machine build. Notice that three blocks are being built at once, one by each of three cores in the CPU. This build is why I bought nice-ish laptops for the course.

H.4 'Hello World' program

Launch Vitis using the red stylised 'V' icon in the left hand toolbar on your laptop. When asked for a workspace directory, browse to the top directory of the project you just created in Vivado. It doesn't have to be there, but it seems sensible to keep everything in the same place. Click 'Launch'. A window should appear with various options. Select 'Create Application Project'. The project name should be (for want of a better idea) hello_world. Click 'next'. At the top of the window that follows, there is a tab called 'Create a new platform from hardware'. Select this tab, then press the + button in the ensuing screen. Navigate using the file browser to the .xsa file from your hardware project, and then press the 'Select' button in the top right of the screen. This .xsa file should be added to the list of available platforms, and be highlighted. The platform selection screen should resemble Figure H.6. Click on 'Next'. The next screen has the defaults correct - we are compiling for microblaze_0 (the only processor core in your hardware - there could in general be more than one), in a standalone operating system (you are building code for the 'bare metal' chip - you don't have an operating system kernal running your code at all), and you are going to develop your application in C. The default application template is, as it always is, a program conveniently called hello world. Click finish on the bottom right.

The main IDE (integrated development environment) window now appears.

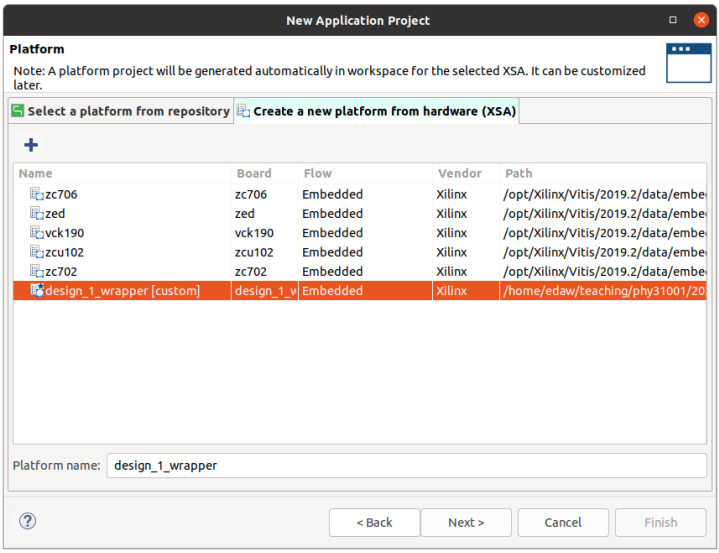


Figure H.6: The choice of platforms after your newly created hardware platform is added.

This is the environment where you will write your code. It appears similar to Figure H.7.

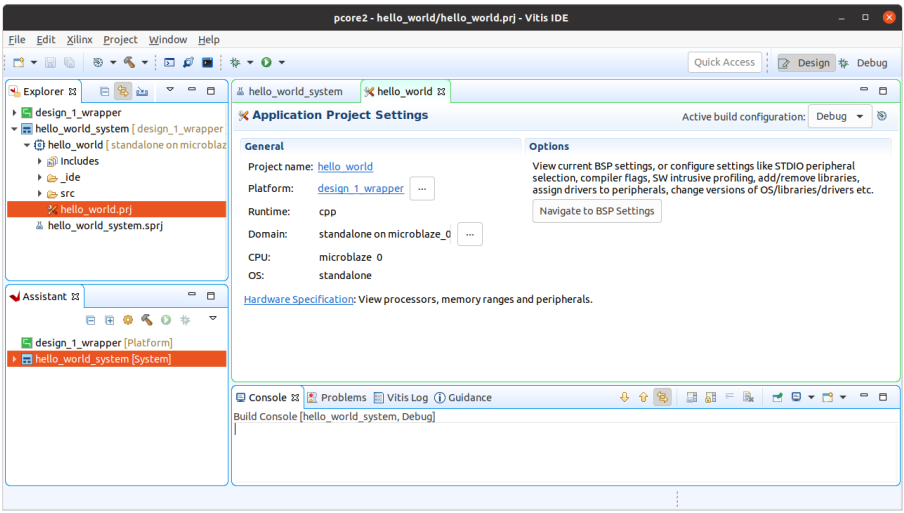


Figure H.7: The IDE when you first enter it for a new system

The SDK project uses a standard variety of graphical development environ-

ment known generically as Eclipse. Eclipse graphical interfaces are ubiquitous in software engineering. The interface is based around a file system that looks very like a standard PC directory structure. The top level directories are on the left in the Explorer window. They are design_1_wrapper, which contains code relevent to the hardware. For now we'll leave that one shut. Then there is the folder hello_world_system, which contains everything having to do with the code we are about to develop for the board. Each folder contains an arrow that you can click on to reveal the folder contents, which may be source files or sub-folders or both. Open the hello_world_system folder, and you will see a subfolder called hello_world. Open this subfolder, and you will see several sub-sub-folders. One of them is called src (as usual short for source code), so open that one. In this folder you will see a file called helloworld.c . This contains the source code that will allow the processor core in your hardware system to greet you. Double click on helloworld.c to open it and view its contents.

The first 46 lines are an extended comment that contains things like the name of the developer, when it was written, a contact email, and a summary of what the code does. It's a good idea to get into the habit of providing this information, especially if you think you might one day develop software in a team. Lines 48 to 61 contain the actual code, reproduced below.

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"

int main()
{
    init_platform();

    print("Hello World\n\r");

    cleanup_platform();
    return 0;
}
```

The three include statements cause the preprocessor to reproduce the contents of those files at the top of the code before compilation. They typically contain further preprocessor commands, which are in turn executed, and function declarations as discussed in class. For example, a function print() is used in the code - the declaration of print() will be in one of the headers. Note that print() is NOT a commonly used C function - this is one that xilinx have added themselves. The more common C print command is printf. Xilinx have written their own printf(), which is a more sophisticated print command, and it is declared in xil_printf.h. We will use it in a while.

It is pretty straightforward to see by eye what this function does. The init_platform() and cleanup_platform() statements are doing some special functions that must be required by microblaze. We don't need to know what they

do right now, but we could if we wanted to go and read the source for these functions to find out. The only line which we need to know about for now is the print line, which is obviously going to attempt to print something. Just so we know that we are not cheating, edit the file substituting your name for World. After this edit, use ctrl-s to save the file.

Now highlight the hello_world_system folder, and click on the hammer icon at the top. This will compile the project, producing an executable that can eventually be loaded onto our hardware. A small pane at the bottom with a tab called Console shows you the build commands and progress. You can click on the rightmost icon in this small pane to maximise the window and see what it contains. It compiles two C programs, one called helloworld.c (the one we just examined) and the other called platform.c. The latter one is probably the 'microkernel' that initialises the processor, connects it to the memory where are program is stored, and then positions the stack pointer that carries out the commands at the command corresponding to the start of the main() function. After these two C programs have been built, then a second code called the linker joins them together and makes the executable, which is called hello_world.elf . This is the program - the 1s and 0s that define the instructions and data that are fed the microprocessor state machine.

If the build worked, then you will see at the bottom, in blue, the time at which the build finished, and how long the build took in milliseconds. This was measured using the clock built in to your laptop to do the timing. To put this window back into the small pane at the bottom, click on the rightmost icon in the window again, which should now be two small overlapping rectangles.

Now we are ready to run your first C program. Go back into Vivado, open the target manager, and programm the Basys3.

Having programmed it, start the application called 'Serial port terminal' in the left hand tool bar - the icon looks like a 9 pin d-sub connector (the old standard connector for serial cables). Once you have started it, click on the 'Configuration' pull-down at the top and select port. You need to change the Port to /dev/ttyUSB1 and the Baud Rate to 9600. Other than that, leave everything the same, then click Okay.

Now finally go back to your Vitis project window. Highlight the 'hello_world_system' folder in the Explorer pane. Now at the top of the window, find the rightmost icon, which is a green circle with a white 'go' arrow in it. Press on this button. A progress bar will appear whilst your hello_world.elf program is transferred to the microblaze core. You should then see Hello (your name) appear in your UART terminal window. You have turned your Artix7 into a computer and it has run its first program. Well done.

This is a laborious process. It clearly isn't worth it just to have the computer say hello, but you can imagine that we can now make our computer do more sophisticated things. For a start, we can try out some of the C programming that I introduced in the last seminar. Then we can perhaps move on to getting the computer to interact with programmable logic.

H.5 More C Programming

First I'd like you to use the more sophisticated xil_printf() command instead of the simple print(). This is because xil_printf() allows us to print variables as well as just text. For now however, just make the replacement of the print statement with xil_printf as follows:

```
xil_printf("Hello World\n\r");
```

Press the hammer again to rebuild, then the green arrow to re-run. You should get another Hello statement. I don't care at this point if you use World or your name. You may by now be wondering what the `\n\r` charcters at the end mean. These are speical non- printing characters, called 'new line' and 'carriage return'. To understand why they are needed, you need once to have used an old fashioned mechanical typewriter. I am sure none of you have. However, on an old typewriter, in order to type two lines of text, you needed to type the first line, and then return the carriage that contained the paper to the beginning of the line, and move a roller that advanced the paper upwards by one line. There, that's two operations. That old mechancial reflex-action has ended up finding its way into modern ascii text files! Furthermore, whether you need `\n\r` or just `\n` is believe it or not system dependent! On Linux machines you just need `\n` and on microblaze processors you need `\n\r`. It's a mad world sometimes.

Next I want you to make some slightly more extensive modifications to the code, as below.

```
int main()
{
    /* declare a 32 bit unsigned integer variable called count */
    u32 count;
    init_platform();

    /* loop over count values between 0  and 9 inclusive */
    for(count=0;count<10;++count) {
            /* display greeting and current count value */
            xil_printf("Hello Ed %u\n\r",count);
    }

    cleanup_platform();
    return 0;
}
```

I have added some comments next to the new things I have inserted to explain what they do. You should put comments in your code as well. The first addition is the declaration of a variable called count. This is at the top of the main() function. The variable is of type u32, which is Xilinx's 32 bit unsigned

integer data type. This variable is used as the counting variable in a for loop. The for loop starts with count=0, and cycles whilst count is any value less than 10, incrementing by 1 at the end of each iteration.

Finally, I have altered the xil_printf statement which is now inside the body of the loop. I have included a %u after Hello Ed (my name, of course). This is called a format field specifier. It tells the print function to expect to print an unsigned integer there. Then, after the end of the double quotes that enclose the text and format field specifier, there is a comma, followed by the name of the variable to be printed.

The xil_printf() function can take an arbitrary number of arguments, where each argument is a new variable to be printed. So, for example we could write instead the following inside the loop:

```
xil_printf("2 times %u is %u\n\r",count,2*count);
```

This time there are two format field specifiers in the xil_printf(), and two corresponding variables to be printed. The first is still just count, and for the second I've built some simple multiplication arithmetic into the print statement, so that the effect is to print out the first 10 elements of the 2 times table (including 2 times 0). Note that the format field specifier is the same both times, because both things to be printed are integers.

You can play around more with making the arithmetical operation more complicated, but don't push your luck. You will find you can't do more sophisticated maths than add, subtract, multiply and perhaps divide. xil_printf() may also have trouble printing out floating point numbers. After all, the processor is super-compact and running on a multi-purpose FPGA, not a special purpose CPU.

Now lets make a conditional statement inside our for loop. Modify the loop to the following code:

```
for(count=0;count<10;++count) {
    /* test to see if it's 5, if it is, print yipee 5 */
    if(count==5) {
            xil_printf("Yipee, count=%u\r\n",count);
    } else {
            xil_printf("count=%u\r\n",count);
    }
}
```

This code will print count=(value) every cycle of the for loop, but will print something different when the value is 5. It might occur to you that you can achieve the same effect with slightly simpler syntax, as below.

```
for(count=0;count<10;++count) {
    /* test to see if it's 5, if it is, print yipee 5 */
    if(count==5) {
```

```
            xil_printf("Yipee, ");
    }
    xil_printf("count=%u\r\n",count);
}
```

This saves an else statement. It exploits the fact that every cycle we are displaying the value of count. The only thing that is added is the Yipee, . If we use xil_printf without the \n\r for the yipee, then the next xil_printf statement will complete the line of text.

If you prefer to spell out your numbers with letters, we can implement a case statement that does this as follows:

```
    for(count=0;count<4;++count) {
            xil_printf("count is ");
            /* switch statement to act depending on count */
            switch(count) {
            case 0:
                    xil_printf("zero");
                    break;
            case 1:
                    xil_printf("one");
                    break;
            case 2:
                    xil_printf("two");
                    break;
            case 3:
                    xil_printf("three");
                    break;
            default:
                    xil_printf("out of range");
            }
            xil_printf(".\r\n");
    }
```

Note that this time I have cut the for loop off above 3, because I didn't want to write out ten case statements! Case structures are often a good alternative to multiple if statements, in the case where there is a large set of possibilities and you want to address all of them. The break; statement is a bit of an oddity of the language. You need one at the end of each case. It is also a good practice to have a default action if none of the cases are satisfied, otherwise the behaviour of your code can be ill-defined. Note also that I include text before and after the case structure so that in each case I get a complete sentence.

Now that we are getting more complex code, you may find that you start to make mistakes. For example, you might miss a semicolon. When you try to compile this will generate errors. For each error, the code editor will put a red 'x' next to the line number where the problem originated. If you hover above

this 'x', a message will appear telling you the error message this line resulted in. This can be a useful way of figuring out what you did wrong.

Appendix I

Lab Exercise 8 - Asynchronous Shared Memory

The lab exercise in Appendix H was a simple "Hello, world!" program. Whilst this made successful use of the soft Microblaze core on the FPGA, it is pretty clear that this kind of computer is in all ways inferior to the laptops I lent you at the beginning of the course! To make successful use of soft cores, they must be able to interact with devices that we build in programmable logic on the FPGA chip. As alluded to in the seminars, there are two main categories of interaction. These are the exchange of data, which can be achieved using shared memory, and the sending of signals, which can be achieved using interrupts. In this exercise, we will use shared memory.

Shared memory is exactly what it says on the tin. Computers operate through the storage of programs and data in memory. This memory is accessed using a memory map. Each memory element has an address, and the addresses are usually expressed as numbers written, at the time of writing, as 32 bit binary numbers. Rather than writing out 32 1's and 0's, which would be arduous and error-prone, we adopt the only-slightly-better convention of representing these 32 bit numbers 8, 4-bit 'words', where a 4 bit word is represented as a single digit in hexadecimal, for example `0xC0000000`. In binary, this particular address is `b11000000000000000000000000000000`; you can see why binary representation is inconvenient. At least some of the memory addresses available to the processor can be used for the storage of data; you can write data to those addresses using a mechanism called pointers, which I will explain in a subsequent section, and read the data back from that same address into your program.

Shared memory provides a set of memory addresses to be written to, and read by, both the processor core and the programmable logic. The word asyn-

chronous means that the shared memory is protected from a potential problem called a race condition. A race condition arises if both the processor and the programmable logic try to write to the shared memory at the same time, or if one tries to write whilst the other one simultaneously tries to read. If this happens, one of the two accessors is allowed to go first, and the other waits until it has finished. This queuing is handled by the shared memory hardware, and it is not possible to predict which of the processor the shared memory will perform its operation first. If it matters, then you must exercise tighter control of writing and reading to shared memory. This can be accomplished, for example, by the use of interrupts.

I.1 New 'IP' Devices

The hardware I would like you to build today starts with the same layout that we used in the previous lab. You have a microblaze core connected to memory and a UART interface connected to a terminal. But, this time, we add some additional elements that are used to configure some shared memory. These make use of several new parts from the Xilinx IP catalogue, described below.

I.1.1 Constant

This element simply outputs a constant value. You can find it by using the "+" button in the diagram and searching for 'Constant'. If you right-click on the block and select 'Customize block', there are two parameters you can set, the width of the constant in bits, and the value of the constant. The value of the constant can be written in decimal, or in binary using `0b1111` for 15, for example, or in hexadecimal using `0xf` for the same number in hexadecimal.

I.1.2 Concat

This element allows you to join together two or more buses to make one big one that has width equal to the sum of the widths of the input buses. You can customise the block to set the number of buses you wish to concatenate and the width of each of the buses.

I.1.3 Slice

This element allows you to extract a sub-range of the bits from a bus. For example, if you have a 32 bit bus coming in, you might want an output bus that is bits 0 to 15 inclusive of the input bus. Customising slice allows you to specify the width of the input bus, and the subset of the bits from the input bus you wish to write to the output bus. The width of the output bus is automatically the number of bits in the range you specify, which is inclusive of the elements specified at both ends.

I.1.4 Block Memory Generator

This element generates the shared memory. It arrives by default with one input port. There are many configuration paramters on customisation. The most important one is in the 'Basic' tab and it is 'Memory Type'. We will always set this to 'True Dual Port RAM'. This is the only element that you will need to specify. When the customised block appears in your diagram, you can click on the '+' icon next to either of the 'BRAM_PORT' inputs to see what is included in these buses. Each one has various inputs and outputs. We will typically operate the BRAM with port A connected to the processor system by way of a BRAM controller (to be described next), so that we won't need to break out the sub-components of this port, and port B connected to the programmable logic, so we will need to define the inputs and outputs to the sub-elements of that port.

I.1.5 AXI BRAM Controller

Shared memory is connected to the processor system using a general purpose, and quite sophisticated, bus called AXI4. Both the shared memory and the controller for the UART port that connects to your terminal are on this same bus, and once you have more than one device connected to the AXI port, this in turn leads to a requirement for yet another piece of IP called AXI SmartConnect. Fortunately, if you put both the UARTlite and the AXI BRAM Controller IP on the same diagram, 'Run Connection Automation' will automatically connect everything up using AXI SmartConnect and it should all work.

Using these new pieces of IP, and the Microblaze processor we used last time, we will build the hardware necessary for the project.

I.2 Building the Hardware

The full hardware configuration we are going to build today is shown in Figure I.3. Build the diagram up using the instructions below. Some of the blocks that you will add will need to be customised. Customisation of the various blocks is described in Section I.1.

1. Open the hello world project from last time. The block diagram should look like that in Figure I.1. Exit Vitis.

2. Check that it still works! Run the hello world program, open the terminal, and verify that you get the greeting. Note that the constraint file should have the clock and reset buttons uncommented, and that the names of those ports in the constraint file should match the names appearing in the diagram.

3. In vivado, from the File menu, select Project, then 'Save As...'. Select a folder to put it in, and ensure that the option to create a new subfolder

for the project is checked, then name your project, perhaps 'sharedmemory'. This should automatically migrate you into the new 'sharedmemory' project area, leaving the hello world project unchanged.

4. Import the 'Block Memory Generator', customising as described in Section I.1.4. Break out port B using the '+' icon so you can see all the sub-ports of this bus.

5. Import the 'AXI BRAM Controller'. Customise the AXI Bram controller to reduce the number of BRAM interfaces to from 2 to 1. This turned out to be critical to getting it to work, so be sure to do this.

6. Move both the new IPs down to the lower part of the diagram, below the Microblaze processor. This will make it easier to wire in the constants later.

7. Select 'Run Connection Automation'. This will bring up a set of checkboxes for the buses you want to connect. Select all except `BRAM_PORTB`. Now click on the button at the top of the diagram pane that looks like a step with a circular arrow inset. This should neaten up the diagram, and it should look like that shown in Figure I.2. It does not matter if everything is not in exactly the same place, so long as the wires are connected to all the peripherals.

8. In the file menu, save the block design. You never know when things might crash.

9. In the constraints file, un-comment the switches and LEDs and save the file.

10. Wire the `clkb` port on the block memory generator to any of the other wires connected to `clk` in the diagram. Get two Constants from the IP catalog, and set them to single bit binary 1 and binary 0, respectively. Wire the binary 1 to the `enb` (enable b) port, and the binary 0 to the `rstb` (reset) port.

11. Save the block design again.

12. Get a `Slice` IP. Customise it so that the highest 16 bits are extracted from a 32 bit buffer. Connect its input to `doutb` from the block memory. Right click on its output and select 'Make external'. Change the name of the external port to `led` to match the led outputs in the constraint file.

13. Get a `Concat` IP. Customise so that it concatenates together two 16 bit buses into a 32 bit one. You will need to toggle the switches for the widths of the input ports from auto to manual and set them both to 16. There should now be two input ports to the Concat, called `In0` and `In1`. The `In0` port should be made external and renamed to `sw` to match the switches. The `In1` port should be connected to a sixteen bit wide constant

whose value is zero. Connect the 32 bit wide output bus from the `Concat` IP to `dinb` on the Block Memory Generator.

14. Save your diagram again.

15. Import another constant. This should be set to 32 bits wide and a value of 0. Wire this constant to the `addrb` input to the block memory generator. This defines which memory address is being connected to, relative to the base address of the block memory. We are using the very first address, so the memory offset into the address space of the block memory is zero.

16. The last port to wire on the block memory is the `web` port. This is short for write enable. It has 4 bits, each of which corresponds to one of the four bytes in the 32-bit-wide field of data to and from the device. We are here always writing the lowest 16 bits from the switches into the block memory, and writing the highest 16 bits from the block memory to the LEDs. So, we set these bits to `0b0011`, so that the lowest 16 bits are 'write enabled'; we can write the memory corresponding to these bits into the shared memory device.

17. Save your diagram, which should now have the same connections as shown in Figure I.3.

18. Run synthesis, run implementation and build the bitstream file as usual. This takes a while because there are many IP cores in this project. Be patient. However, be warned that when I tried this Vivado quit half way through. This left a bunch of zombie processes running - if this happens to you, you can see the zombie processes in a shell window by invoking `ps -ef | grep Vivado`. In this position, having no idea whether these processes were actually doing anything useful, I decided to restart my laptop and re-open vivado. Vivado attempted to re-launch the build, but I decided to terminate this and re-start the synthesis process. This did finally build everything, but I'm telling you because it was a bit arduous and stressful, so you don't worry if the same thing happens to you.

19. As for the hello world example, Use File, Export, Export hardware to export the hardware description file. To avoid confusing it with the other ones, rename the file shared_memory, rather than design_1_wrapper.

20. Connect the board to your laptop and power it on, ready to be programmed.

I.3 C code for the shared memory

You should start a new Vitis project based on the hello world template just as you did last time. However, the hello world code should be replaced by a copy of the code segment below.

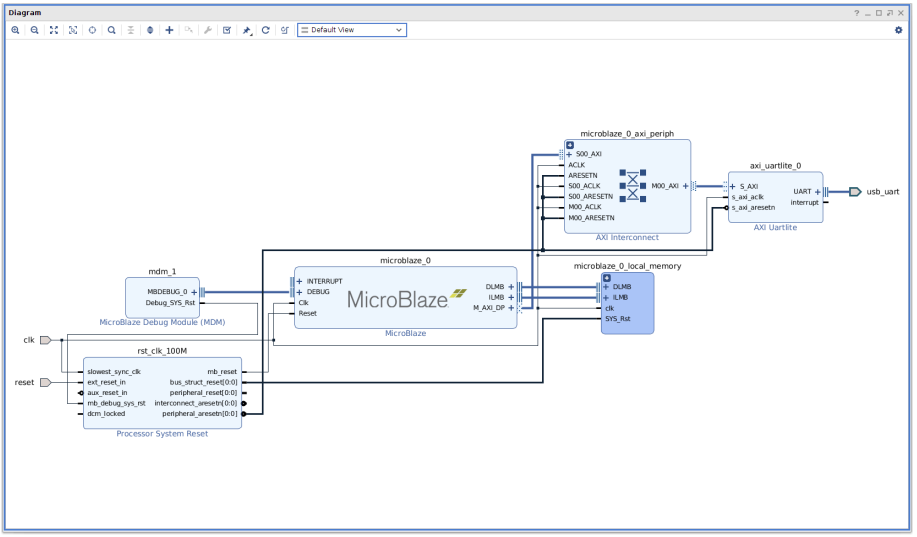


Figure I.1: The diagram for the 'Hello World' project.

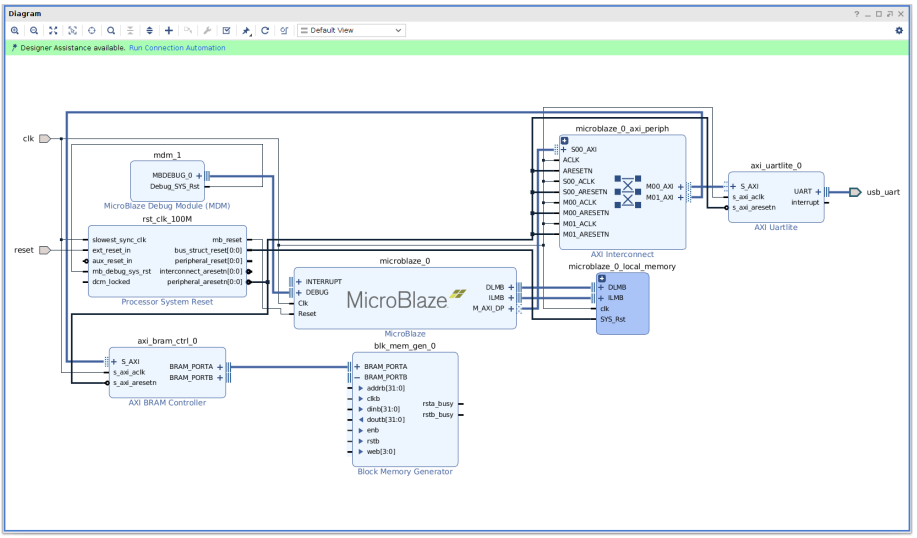


Figure I.2: The shared memory hardware after the shared memory and controller are connected using block automation.

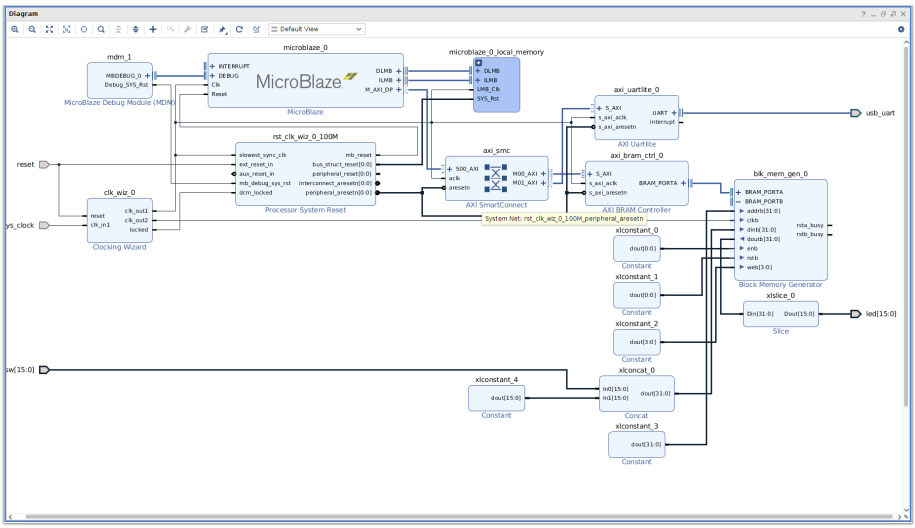


Figure I.3: The block diagram for the shared memory exercise.

```
#include <stdio.h>
#include <unistd.h>
#include "platform.h"
#include "xil_printf.h"
#include <xparameters.h>

int main()
{
        u16* pdataval=(u16*)XPAR_BRAM_0_BASEADDR;
        u32 counter=0;
    init_platform();

    pdataval[1]=127+32768;
    while(1) {
            xil_printf("count %u, value %u\r\n",counter++,pdataval[0]);
            sleep(1);
    }

    cleanup_platform();
    return 0;
}
```

This code contains several elements requiring explanation. First, the new include file `<xparameters.h>`. This file is produced by Vivado, and defines the memory map of the hardware we just built. This memory map includes an

address for the start of the shared memory block. This is needed here, because we are going to both read from and write to the shared memory in this code. The address is defined as a preprocessor 'constant', written `XPAR_BRAM_0_BASEADDR`. It's actual value in the code is `0xC0000000`.

In the `main` program, we first define a couple of constants, but we do it using a new and interesting syntax in the very first line,

```
u16* pdataval=(u16*)XPAR_BRAM_0_BASEADDR;
```

This means that pdataval is the *address of* a variable of type u16. This is the first example that you have met of the use of a pointer. A pointer in C is just an address in the memory of the computer. The value of this address is `0xC0000000`, and it corresponds to the address in memory where the shared memory starts. The other variable is just a counter used in the loop below.

The next interesting thing is this line:

```
pdataval[1]=127+32768;
```

pdataval is an address at the short integer, 16 bits in size. pdataval[0] is the *value at* this memory location. We just set this to 127+32768. The binary bit pattern of this is `0b1000000001111111`. This is what your LED outputs should display. This is your first write from the processor system to the LED outputs.

Next we enter an infinite loop, using the standard code for setting up infinite loops, `while(1)`. Everything inside the curly braces after this statement repeats until the board is switched off. This code prints the value of a counter, followed by the value at address (or pointer) pdataval. This value is set from the switches in the hardware. So, if you change the switches, you should see the value printed out change in your terminal when this code is running.

Here are the instructions for the rest of the project.

1. Launch vitis. Either make the base directory the same as your vivado project directory containing your .xsa file, or choose a new empty directory.
2. Make a new project using your .xsa file to specify the hardware.
3. Make a hello world project.
4. In vivado, use the hardware manager to program the board.
5. Open the serial comms program, open a connection to port USB1 at 9600 baud as you did last week.
6. Run the hello world program as you did last week, check that Hello World appears in the terminal.
7. Now replace the C code in helloworld.c with the code we just described.
8. Highlight the system directory and rebuild with the hammer icon.

9. Highlight the directory under system containing your software application.

10. Launch on the system. The LEDs should after a short delay display the bit pattern `0b1000000001111111`. The terminal should update once a second, and display the bit pattern set on the switches.

Appendix J

Lab Exercise 9 - Commanding the Lights

J.1 Processor System to Programmable Logic

In Appendix H we learned to embed a microblaze processor on an Artix 7 FPGA. The processor was connected via a UART interface to a terminal window. We used C to write simple programs that could display things in that window. In this lab we will do several things. First, we will learn how to use C to detect and store keystrokes from the keyboard connected to the terminal. A sequence of keystrokes will be stored in memory, until a special keystroke, the 'return' key, is detected. Then the series of stored keystrokes will be interpreted as a number. Second, we will learn to create a shared memory between the processor system (PS), the computer we embedded on the FPGA, and the programmable logic (PL) that we can arrange for ourselves alongside the processor. Finally, we will learn to read out the shared memory in programmable logic and display its contents with the LEDs on the BASYS3 board. In both the storage of a sequence of characters in memory on the FPGA, and the writing of numbers to shared memory, we will need to make use of a facility for direct addressing of memory in C called pointers. Pointers are the main reason why we choose to use C as opposed to PYTHON or some other higher level language in this course.

The structure of this lab will be as follows. First, you should certainly carefully read through the notes from the seminar in Week 10. In that class I taught you how to build a terminal program on the microblaze processor. We will use that program with very little modification to allow us to send numbers that we will type in on the keyboard to shared memory. Second, you will use VIVADO to build a slightly more complex computer that incorporates a shared memory module and the wiring to connect the data output of the shared memory to our 16 LEDs. Next you will build the command line interpreter from

Lecture 10 and run it on your modified hardware. The presence of the extra hardware for shared memory and LEDs should not prevent the command line interface program from working as I showed you in class. Finally, you will add some code to this command line interface so that when you type in a number, that number is written to shared memory and is mirrored on the LEDs on the BASYS3 board.

J.2 Modified Hardware

Figure H.3 shows the basic hardware configuration that was built in Appendix H. We must first add some hardware elements to this block diagram for the shared memory facility and the LED outputs. If you have not yet built your embedded system, follow the instructions in Section H.1 of Appendix H until you have this block diagram built. Now press the '+' button and search for a module called "AXI BRAM Controller". This is the unit that will interface the shared memory to the bus of the microblaze processor. Once this block has appeared, double click on it, and reduce the number of AXI BRAM interfaces from the default of 2 to 1. Next, add a second hardware element called "Block Memory Generator". Once this block has appeared, double click on it too, and alter it to "True Dual Port RAM". Also check the small box for "Common Clock". Now click on 'run connection automation' and select 'all automation' to wire the shared memory into the rest of the hardware.

On the left hand side of the Block Memory Generator, you should see that one of the two BRAM ports is wired to the AXI BRAM Controller, but the other one is unwired. If you click on the '+' sign next to this unwired controller, it will expand to reveal 7 ports. We will need to connect all of these up to use the shared memory. Sometimes an unused port may disappear from the GUI interface. If this happens, close and reopen the expansion by clicking on '-' then '+' again, and the port should reappear. The first and easiest wire is the reset port, labelled 'rstb'. Wire this to 'reset' on the left hand side of the diagram. Next, we need to wire up the clock input, labelled 'clkb'. I have found it best to create a dedicated clock port to wire this to. To do this, double click on the 'Clocking Wizard' module, select the 'Output Clocks' tab and check the second unused output, then click OK. A clock output port should appear, and you should wire this to 'clkb' on the Block Memory Generator.

There are several ports on the Block Memory Generator that need constant inputs of various descriptions. These are labelled 'addrb', 'dinb', 'enb' and 'web'. The simplest of these is 'enb', which is a simple enable line, and should be set to '1'. To do this, use '+' and search for 'Constant'. When this block appears, wire the output to 'enb'. The default output for 'Constant' is a single bit with value 1, so you don't need to change anything. Now get another 'Constant', but this time double click on it, and change its data so its Width is 32 bits and its value is 0. This one should be wired to both 'addrb' and 'dinb'. The 'addrb' allows you to select the starting byte in the address space of the

shared memory for the data we will be reading. Since we will simply use the very first byte of the address space, this offset is set to zero. The 'dinb' bus allows you to write to shared memory from programmable logic. We won't be doing this, so we just set this input to zero as well. We use the same 32 bit wide zero to wire to both inputs to save space in the diagram. The 'web' input enables on-the-fly reconfiguration of shared memory, but we won't need this, so this needs to be connected to another constant, equal to zero, four bits wide. Connect that one up as well.

Next to connect the output data. Note that this is 32 bits wide, but we only have 16 LEDs. We shall use these LEDs to display the contents of the lowest 16 bits of the data in the first 32 bit space of addresses in the shared memory. To access the lowest 16 bits of a 32 bit field, we import a block called 'Slice'. This block should be moved to under the Block Memory Generator. Double click on it and set 'Din from' to 15 and 'Din Down To' to 0. 'Dout Width' should automatically adjust to 16. Wire the input of Slice to the doutb output of the Block Memory Generator'.

Next, we need to connect up the LEDs. You'll need to add a constraint file to the Vivado project. Use the fully commented constraint file as usual, and only uncomment the 16 LEDs. Now right click on the Dout output of Slice, and select 'Make External'. The port name will be Dout_0 by default. Click on it, and in a window to the left of the GUI, a box next to Name should say Dout_0. Click in this box and change the name to led, and hit return. The port name should change to 'led'. Now press the button at the top of the gui window with the circle-arrow in it to reorganise the diagram. It should now look like the one in Figure J.1. Click on the box with a tick in it and let Vivado run a hardware check on your diagram as a rudimentary check that you haven't left anything critical out or left any loose wires hanging around.

If you have not already done so, create an HDL wrapper for your hardware using the instructions in Section H.2. You should now be able to carry on following the instructions in this section to run synthesis, run implementation and generate the bitstream. Just continue with the instructions until you have exported the project. One interesting aspect is to look at a tab that should have appeared in the main project pane in Vivado called 'Address editor'. This shows a rough memory map of the address space of buses connected to the microblaze processor. Notice that axi_bram_ctrl_0 occupies address `0xC0000000`. We will encounter this address again when we write the code.

J.3 Software for the Light Controller

The software can mostly be re-used from the terminal program that was developed for Seminar 6. See page 14 of the notes from this seminar. The program is reproduced below. You will need to go through the notes of the seminar to make sure you understand the use of pointers/addresses to write data to memory and read it back.

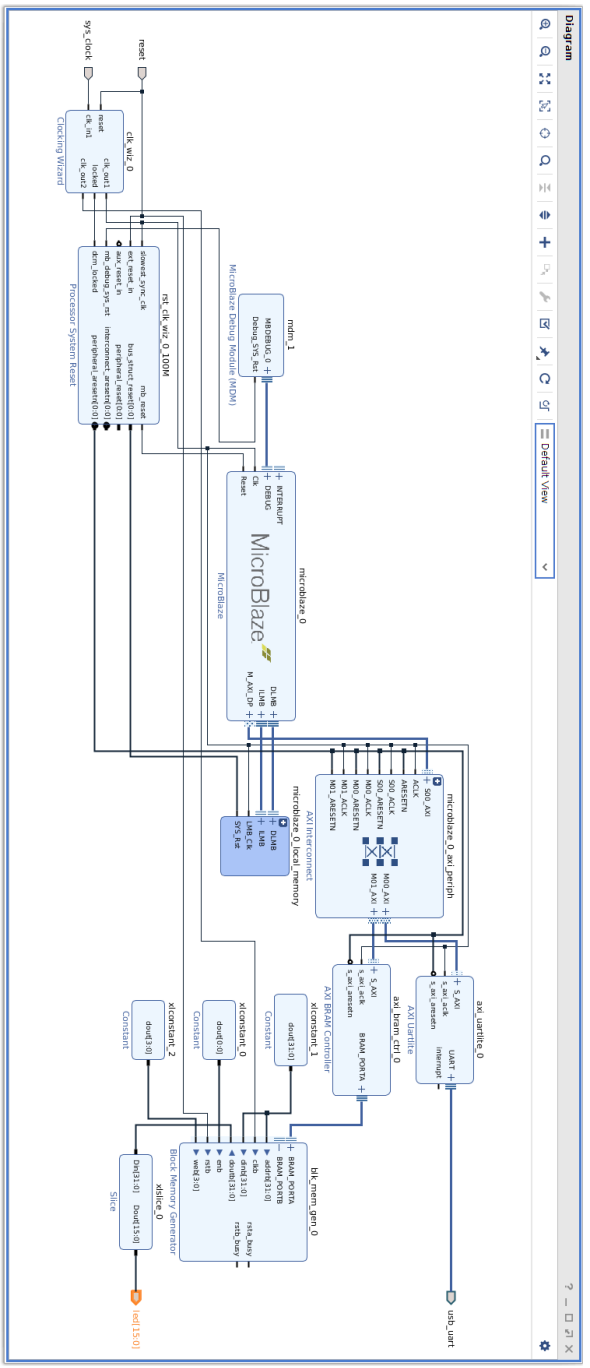


Figure J.1: Hardware for the shared memory project

```
#define CMD_BUF_LEN 20
int main()
{
  u32 charcount;
  unsigned char inp;
  char cmdbuffer[CMD_BUF_LEN];
   while(1) {
    xil_printf("\r\n$ ");
    charcount=0;
    while(charcount<CMD_BUF_LEN) {
       xil_printf("\r\n$ ");
          charcount=0;
          while(charcount<CMD_BUF_LEN) {
                inp=XUartLite_RecvByte(XPAR_UARTLITE_0_BASEADDR);
                if(inp!=(unsigned char)'\r') {
                  xil_printf("%c",(char)inp);
                  cmdbuffer[charcount]=(char)inp;
                  ++charcount;
                } else {
                  cmdbuffer[charcount]='\0';
                  xil_printf("\r\nCompleted command was %s",cmdbuffer);
                  charcount=CMD_BUF_LEN;
                }
          }
          if(inp!='\r') {
                  xil_printf("\r\nERROR: command buffer full");
          }
        }
    return;
}
```

The key statement allowing the detection and ingestion of keystrokes from the terminal keyboard is `inp=XUartLite_RecvByte(XPAR_UARTLITE_0_BASEADDR)`. This is a call to wait for a key press on the keyboard, and load the character corresponding to that key into the variable `inp`, which is of the somewhat obscure data type `unsigned char`. This character data is written to a memory address, or pointer `cmdbuffer+charcount`. The variable `charcount` is incremented each time a character is stored, so that the sequence of keystrokes ends up in an array. Once the return key is hit, the termination character is appended to the end of the character array. Currently in the code nothing is done with the character string.

You should open Vitis, select the base address of your project as the working directory, and following the instructions in Appendix H import the hardware file, which is a file called `<project name>.xsa`, into the IDE. Start a new hello world project as we did before, but now replace the code in `hello_world.c`

with the above code for the terminal emulator, ensuring that you also have the correct set of include files, given on page 13 of the notes from seminar 6:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xil_types.h"
#include "xparameters.h"
#include "xuartlite.h"
```

After saving the modified source code you should build the project, start the terminal, configure the terminal following the instructions in Section H.4, and run the code, checking that the terminal prompt appears in the terminal window, and that you can enter commands up to 20 characters long. After you press return, the command should be echoed back to you on the terminal, and if you exceed 20 characters this should be detected and induce an error. It's a simple, decently robust, terminal, but it has no interpreter.

J.4 The simplest possible interpreter

We wish in the simplest possible way to interpret this character string as an unsigned integer, and to write this integer to shared memory so that its value can be displayed with the LEDs. First we will need a variable of an appropriate type whose location is the address of the shared memory. The address we require is `0xC0000000` which if you recall was in the memory map in hardware. We could hard code this address, but this isn't a very robust approach. Better than that, it turns out that the header file `xparameters.h` defines a preprocessor string which corresponds to this address. Were we to alter our hardware, and the base address of the shared memory were to change, the file `xparameters.h` would be altered to contain the modified base address, and there would be no need to update our C code. Using the Explorer pane in IDE, open the folder labelled `design_1_wrapper`. Navigate to `xparameters.h`, which is at the path shown below. Line 552 of this file contains the `#define` for a preprocessor variable `XPAR_BRAM_0_BASEADDR` to the required address in the memory map.

```
microblaze_0/standalone_domain/bsp/ \
microblaze_0/include/xparameters.h
```

Note that when you open this file in the IDE the contents are displayed in a new tab, but the source you are modifying has not disappeared, and is in another tab behind which you can return to and carry on coding. To associate this particular address with a 16 bit variable, insert the following variable declaration immediately following the opening curly brace of the main program.

```
u16 *pled=XPAR_BRAM_0_BASEADDR;
```

The next task is to insert a line of code that interprets the input keystrokes as a number. A simple way of doing this is to use the `atoi` function built in to C, which essentially takes a character string and attempts to interpret it as an integer. This integer is then cast to an unsigned short integer of the xilinx inbuilt type `u16`, and this number is stored at the memory location that is mapped to the shared memory in hardware. On the other, programmable logic side of the fence, the data at this memory location is wired to the LEDs. The line you need to alter in the C code is the line where the command was echoed back to you, which currently as follows.

```
xil_printf("\r\nCompleted command was %s",cmdbuffer);
```

Replace this command with the following invocation of `atoi`.

```
*pled=(u16)atoi(cmdbuffer);
```

This command interprets the string of characters at the address `cmdbuffer` as an integer, and then casts that integer into data type `u16`, storing the result at the address pled, which is `0x0000000C` in the memory map, the memory location shared with the programmable logic.

Recompile your code and after recompiling, you should see that numbers you typed in are displayed in binary on the 16 LEDs. An additional nice feature is that if you type in anything that is less than 20 digits long but isn't a number, the behaviour of `atoi()` is to return zero. So, none of the LEDs light up. This is a reasonable, though not very revealing, failure mode.

J.5 Coding challenges

Here are some other projects that you might try to enhance the terminal based LED controller.

First, you could use the driver you previously built for the 4 character 7 segment display to display the number you enter in hexadecimal on the LED display instead. This is a matter of creating a graphical 'IP' from your LED controller, making the exported control visible to your graphical designer in Vivado, adding the controller to the hardware, connecting it to the clock (you'll need to create a further clocking wizard output to connect it), modifying the constraint file so that the LED ports are available on the diagram, and wire the input to your LED display driver to the `doutb` port on the shared memory. Make the outputs of your LED display driver external, and change the names of the external ports to match their names in the constraint file. You will need to rebuild your hardware, and re-export it to your project directory. Vitis will almost certainly spot that the `xsa` file has been updated, and ask you whether you want to update the software project. You will then need to rebuild everything in VITIS, both the `design_1_wrapper` folder and and the application folder (highlight the directory and then hit the hammer button in each case. Re-launch your bitstream in Vivado onto the BASYS3 and run the Vitis project.

Secondly, notice that in hardware we dumped the upper 16 bits of the 32 bits of shared memory. We could instead wire these upper 16 bits to the 4 digit display, whilst keeping the lower 16 bits connected to the LEDs. Now you will need software that can set the value of either of these 16 bits of shared memory. In C code, you can make use of pointer arithmetic. For example, to set the values of the unsigned shorts in the two 16 bit memory segments, we could do

```
*pled=3;
*(pled+1)=7;
```

which sets the lower 16 bits to contain 3, and the upper 16 bits to contain 7. However, you will also need a slightly more sophisticated controller in C so that you can type in a number and have your software know whether you want it to appear on the display or the LEDs. You can either cheat or do it properly. I would cheat first. Exploit the fact that integers can be negative, and put something like this in your C code:

```
/* declaration at the top of main */
int posneg;

/* in the body of the code */
if(posneg<0) {
  *pled=(u16)(-posneg);
  } else {
  *(pled+1)=(u16)posneg;
  }
```

Following appropriate modifications to the hardware to connect the upper 16 bits to the display, this hack effectively allows you to use negative inputs to set the LEDs and positive inputs to set the display. It is, however, not very user friendly - who ordered the sign of numbers dictating where they end up? You may want instead to do it more properly. In this case, you need actual commands and a primitive interpreter for those commands. You'd like to be able to enter 'leds=value' and have the LEDs display the value, or 'display=value' and have the LEDs display the same thing. In this case you need to analyse the entered string for the occurrence of the '=' symbol. Everything before this symbol is interpreted as the command, and everything after this symbol is interpreted as the value.

A command for finding a particular character in a string in C is `strstr()`, which returns a pointer to the start of the first occurrence of that character or character sequence. If `strstr` can't find the character or sequence at all, it returns `NULL`. You replace the value at the pointer to the equals sign with the string termination character, and run atoi on an address one past this inserted termination character, to cast the remainder of the string as an integer. You then read the command, which is the portion of the string at the address `cmdbuffer` before the newly inserted termination character, as a command,

and see whether it contains the sequence led. If it does, you write the value to `*pled+1`, and if it doesn't you write value to `*pled`.

```
/* at the top */
char* pvalue;
/* in the body, find the equals sign */
pvalue=strstr(cmdbuffer,'=');
/* replace the equals with a string termination */
*(pvalue)='\0';
/* read the command and act accordingly, using atoi */
/* to interpret the characters at the address following */
/* the newly inserted termination as a number, casting */
/* it to u16 and writing to the appropriate shared */
/* memory location */
if(strstr(cmdbuffer,"led")!=NULL) {
  *(pled+1)=(u16)atoi(pvalue+1);
  } else {
  *(pled)=(u16)atoi(pvalue+1);
  }
```

A final comment on this code. I have completely abandoned making it robust to syntax errors. What does the code do if there is no equals sign in the string? Probably something undesirable. Also, any command that doesn't contain the character sequence led is interpreted as a directive to send the data to the display. If there was some third command, this would not work. You might want to error trap the return value of null into pvalue so that nonsense entries are dealt with robustly in the code, and further error trap the scenario where the command doesn't contain led or display. Having done that, congratulations you've written your first (very primative) command line interpreter on an embedded system!

Bibliography

[1] Douglas R Hofstadter. *Gödel, Escher, Bach: an eternal golden braid*. Penguin books. Basic Books, New York, NY, 1979.

[2] Raymond Smullyan. *Forever Undecided: A Puzzle Guide to Gödel*. Oxford Univ. Press, Oxford, 1987.

[3] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012.

[4] Alfred North Whitehead and Bertrand Arthur William Russell. *Principia mathematica; 2nd ed.* Cambridge Univ. Press, Cambridge, 1927.