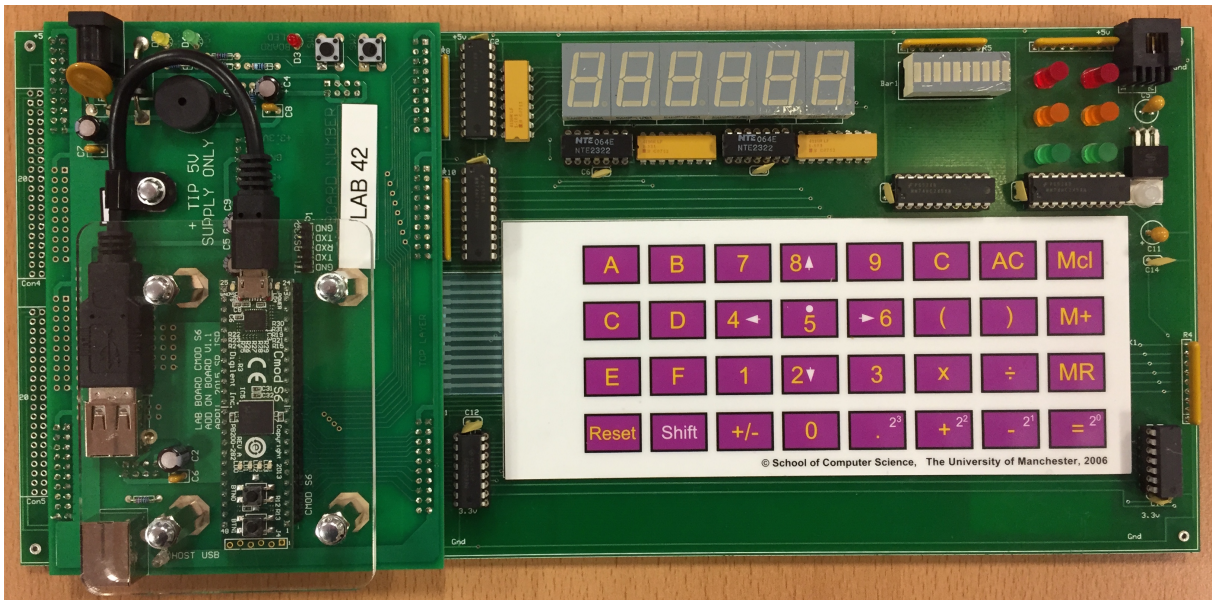
COMP12111

Fundamentals of Computer Engineering



Laboratory Manual
2016/2017

Contents

Laboratory Organisation

These notes apply to the laboratory associated with the course unit COMP12111: Fundamentals of Computer Engineering. Please read these notes carefully before you attend the laboratory as they contain important information regarding the organisation of the laboratory. It is important that exercises are attempted in the order given according to the timetable and identified deadlines – this is for your benefit in order to maximise the amount of marks you can achieve in the lab!

There are 10 scheduled laboratory sessions for you to attend, one per week starting from week 2, with each lasting 2 hours. You are given 2 weeks to complete most exercises, apart for exercise 5, where you only have 1 week. The full laboratory timetable, showing the deadlines associated with each exercise is given in Figure i.

There are a number of laboratory sessions scheduled each week, please make sure you **ONLY ATTEND YOUR SCHEDULED LABORATORY SESSION** as given in your personalised timetable. We do not have enough computers/equipment for more people to attend than those timetabled to do so!

Not every exercise will count towards the final assessment mark for the laboratory. However, some of the designs you create will be used in later exercises. For example, the 16-bit adder you produce in Exercise 1 will be needed for the design of MU0 in Exercise 4.

In this manual the submission point is highlighted by a grey box. Each exercise is preceded by detailed information about the exercise, plus some background information to help you. **It is important that you read the background material before you start each exercise.** You should also make use of the lecture material that supports each exercise!

If you do not prepare for each exercise before the lab (by at least reading the material that supports the exercise and familiarising yourself with the exercise) then you may find that you will fall behind and require an extension to the exercise. The exercises are designed so that the average student can complete them in the time allocated. In addition to the practical instructions, the laboratory manual contains sections of relevant descriptive material that should form useful lecture revision.

The exercises will involve the use of professional CAD tools that you would be more likely to use if you worked in the industry. Consequently, these tools are complex and can be difficult to use. However, exercise 1 takes you through the steps of producing and testing a design using the tools.

The laboratory counts for 50% of the overall mark for the course unit. A breakdown of the percentage contribution of each exercise to the overall laboratory mark in given in Figure i.

Design Interfaces

In all the exercises we provide you with the interface to your design. In the case of a schematic design we provide the input/output pins (where appropriate), and in the case of Verilog modules we provide the module header. It is **important** that you **do not change** the interface to any design. You should not add new pins, add new signals, rename signals, or change the size of any buses. If you do so, the submission mechanism for submitting your work will **not work**, which may result in your work being submitted late and may make it difficult to mark your work offline (where used).

Labos comptent pour moitié de la note

Week	Exercise	12111submit exercise	Deadline*	Demo Week	Arcade Deadline	Total (%)
1.1 (A)	No scheduled lab					
1.2 (B)	Exercise 1					
1.3 (A)	Exercise 1	ex1a, ex1b, ex1c	1.3	1.4	1.2aD, 1.2bD, 1.2cD	**Not assessed**[§]
1.4 (B)	Exercise 2					
1.5 (A)	Exercise 2	ex2	1.5	1.7	2.2aD 2.2fD	**20**
1.6 w/c	Reading Week – No scheduled lab					
1.7(B)	Exercise 3					
1.8 (A)	Exercise 3	ex3a, ex3b	1.8	1.9	3.2aD, 3.2bD 3.2fD	**35**
1.9 (A)	Exercise 4					
1.10 (A)	Exercise 4	ex4	1.10	1.11	4.2aD 4.2fD	**20**
1.11 (B)	Exercise 5	ex5	1.11	1.12	5fD[#]	**25**
1.12 (A)	Marking Session					

* – the deadline is before the END of your scheduled laboratory in the week given, the extended deadline is for the START of your next scheduled lab

a – automatic marking, f – face-to-face demonstration in lab

[§] – this work is not contribute to your mark for the lab. However, you must complete and submit the exercise. Feedback will be provided.

[#] – there is no off-line marking for this exercise, all marks come from the face-to-face demonstration

Figure i: COMP12111 Laboratory Timetable

Deadlines

The laboratory consists of five exercises. The deadline for submitting exercises is the **END** of the scheduled laboratory associated with the deadline (see Figure i).

Extensions are available but only if you can demonstrate that you have made sufficient progress on the exercise (or if you have mitigating circumstances to explain why you are behind).

If you would like to request an extension you must request it at the **END** of the laboratory when the call for extensions is made. The extension is then until the **START** of your next scheduled laboratory (see Figure i for details). Any work submitted after the deadline will be treated as being submitted **LATE** (please consult the 1st year lab manual to understand the implications of this.) Late penalties will be rigorously applied for any work submitted after the original deadline, unless we know of any mitigating circumstances, or an extension has been granted. Work submitted after the extended deadline will always be marked as late – there are no further extensions beyond the extended deadline, unless you have significant mitigating circumstances (illness on the day of the extended deadline will not be accepted).

It is vital that you keep to the deadlines for each exercise otherwise you are in danger of falling behind in the laboratory. The laboratory counts 50% of the overall mark, so the more exercises you complete and have marked; the more chance you have of passing the course unit.

The final submission deadline for **ALL** exercises (which will be recorded as **LATE** if the deadline has passed) is the **BEGINNING** of the **FINAL** scheduled laboratory in week 12. **Please note: the final lab session is for marking only - no exercises are to be completed during this lab** (we will run "submit-diff" to make sure the work demonstrated in the laboratory does not differ from what has been submitted).

Help in Laboratories

Teaching assistants (TAs) - who are typically post-graduate students - and members of staff will be available during all scheduled laboratory sessions to provide help and assistance where required. However, TAs are not there to tell you the answer, they are there to point you in the right direction when you are stuck. Always attempt exercises before seeking help, but do not wait too long before asking for help if you do get into difficulty. If you do not ask for help soon enough, you may fail to complete the exercises within the scheduled time and you will then fall behind.

Submission & Assessment of Lab Work

For most exercises the assessment marks come from two sources:

1. offline automatic marking of submitted work, and
2. a formal face-to-face demonstration in a scheduled laboratory session.

In some exercises marks are only awarded for demonstrating your work. However, you must still submit these exercises before the deadline.

When you submit your work via the submit script then Arcade is notified of the date and time when your work was submitted. Consequently, there can be no errors in recording the date/time your work was submitted, so please do not question any late flags.

The automatic marking looks at two aspects of your work:

1. it tests the functionality of your design, and
2. it tests the test stimulus that you have created to test the functionality of your design.

For the formal demonstration you must show your designs, in most cases working on the experimental boards, in the laboratory, as well as answering any questions on the labprint marking sheet.

The submission of your work for this lab differs slightly to the process used in other labs. Here, you submit your work using the lab submission script

12111submit

this uses the scripts "submit", "labprint" etc., that you may be familiar with from other labs in the first year.

On running the submission script you will be asked which exercise you would like to submit:

```
Please enter the exercise you require submitting/marking
Enter one of {ex1a ex1b ex1c ex2 ex2f ex3a ex3b ex3f ex4 ex4f
ex5 ex5f}
```

When submitting your work for offline marking you enter the shortened exercise number, i.e. for exercise 2, this will be ex2.

You will then be given four options:

1. `Type submit to submit your work for automatic marking`

You should use this option to submit your work for the first time. Enter 'submit'.

2. `Type submit-again if you want to resubmit it, but note that your time of completion will be the time of the resubmission, which could make your hand in late.`

You should use this option to submit your work again after you have previously submitted it. Enter 'submit-again'. Please note: if you run submit-again AFTER the deadline has passed, then you work will be treated as being submitted LATE.

3. `Type submit-diff if you want to compare your latest files with your previous submission.`

You should use this option to check whether the submitted work differs (or not) from the current working version. Enter 'submit-diff'. This is used by TAs to check that you are demonstrating the SAME work as you have submitted for offline marking. You can also use this to check whether you have actually submitted the exercise.

4. `Type labprint if you want to print an exercise marking feedback sheet`

You should use this option to print a labprint sheet for the demonstration. Enter 'labprint'.

The submit scripts look in your Cadence directory for the files needed for submission, check that these compile, and then copy them over to the marking server for offline marking (for submit and submit-again). Some exercises have multiple parts so you will need to run the 12111submit submit script for each of these exercises as you complete them. However, you should demonstrate all of them together. Late submission of work will be detected by this mechanism. You only need to submit the exercises you have completed.

After you have submitted your work you need to print out a labprint sheet to demonstrate your work in your next scheduled laboratory session. Run the 12111submit script again, enter the exercise, but this time enter 'labprint'. The labprint sheet for the submitted exercise should be sent to the print queue.

Please note: the printing of a labprint sheet is not evidence of the submission of an exercise. **Please remember to *submit* and *labprint* for all exercises**.

Feedback

For those exercises where there is offline marking, you should receive an email with a mark and feedback after the extended deadline has passed (usually the Friday). This email will come from 'Barry Englab' – a fictional member of the engineering team. Please make sure you look at the feedback, and particularly for exercise 1, make sure you correct any mistakes, as you will need a working 16-bit adder later.

You will also receive feedback from TAs during face-to-face demonstrations of your work in labs.

Demonstrating your Work

For each exercise you will be required to demonstrate your designs in the laboratory. To do so you will need to print out and complete the labprint sheet for the exercise. The labprint sheets also contain questions that you can attempt before you demonstrate your work. It is up to you if you choose to answer these questions, as answering them is not compulsory. If you choose not to answer them, then you will loose some marks, but not many. Some of these questions are deliberately challenging and the questions can vary between students. Please answer these questions **OUTSIDE** of the lab and not while you are waiting to demonstrate your work. You should be using this time more constructively by moving onto the next exercise.

To demonstrate you work in a lab session you must write your name and machine name clearly on the whiteboard at the front of the lab. This must only be done once the lab starts. Do not write your name on the whiteboard before the start of the lab as it will be cleaned at the start of every scheduled lab session.

Once you have demonstrated your work you will receive a marking token and a mark from the TA that you must submit by running the *12111submit* script, entering the exercise number followed by an 'f' for face-to-face marking - so for submitting your demonstration mark for exercise 2 you would enter 'ex2f'. Note, you only submit one face-to-face mark for exercises with multiple parts, such as exercise 1.

Please aim to demonstrate your work in your next scheduled lab session after the deadline has passed, see Figure i. Please do not leave your demonstrations until the last lab – there may not be enough time, and if you don't get work demonstrated by the end of this lab, it won't be!

Academic Malpractice

We don't accept any form of academic malpractice in this course unit.

What is academic malpractice? It is any activity used to gain an unfair advantage in academic work, such as plagiarism (copying and using the work of others) and collusion (sharing work too closely). More information on academic malpractice can be found in the "University Regulation on the Conduct and Discipline of Students":

http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=6530

Any work submitted for assessment in this course unit must be **INDIVIDUAL** work. Any student(s) found to be engaged in academic malpractice will be reported to the School Academic Malpractice officer, which may result in you having to attend a School hearing to explain your actions. In repeated cases you will be reported to the University.

If you are found guilty of academic malpractice then the penalty can range from having your work zeroed (in the first instance), to failing a course unit, or, in the worse case, failing your degree.

Please do not be an accessory to academic malpractice, do not share your work with any other student, and keep you work secure.

Laboratory Cleanliness

Please keep the laboratory clean. If you have any rubbish, then please place it in the bins provided, do not leave it lying around. Please note that **NO FOOD OR DRINK IS ALLOWED IN THE LABORATORY**, apart from bottled water. Spillages are inevitable and greasy fingers make keyboards and screens unpleasant for everyone. Anyone found with food and drink in the lab will be asked to leave.

Laboratory Time

DO NOT use your lab session for working on exercises for other laboratories, scheduled labs should only be used for working on COMP12111 exercises. If you are found working on other labs then your attendance will not be taken and you may be asked to leave. Extensions will not be granted to students working on exercises for other course units.

Cadence Lock Files

In this laboratory you will be using a commercial design framework called Cadence, which only runs in the Linux environment. Cadence contains a number of design tools, such as schematic entry and a waveform viewer, which aid the designed in producing complex digital circuit designs. Cadence is very powerful and is used by professional designers.

It is possible to run Cadence remotely, however we strongly recommend that you **DO NOT RUN CADENCE REMOTELY**. There are a number of reasons for this, one reason is that the Cadence tools run very slowly when run remotely. However, more of an issue is that Cadence is a multiuser environment (allowing multiple designers to work on the same design) and, as such, uses lock files to prevent files from being opened by multiple users. If you are running Cadence remotely and your connection fails (which commonly happens) then this may lead to lock files being left, which will then prevent you from running Cadence or even in some cases editing your designs.

The same problem can happen if you do not exit Cadence properly (i.e. close it down via the menu File ↲ Exit). If you simply close down your X windows session without closing Cadence down properly then lock files can remain.

If you have problems starting Cadence and it reports a log file initialization failure, i.e.

```
*WARNING* file /home/<username>/Cadence/CDS.log_COMP12111 No route to host
Failed to lock log file: /home/<username>/Cadence/CDS.log_COMP12111
*ERROR* Log file Initialization failure.
```

then these lock files need to be deleted manually. The lock files have the extension .cdslck and can be found in the Cadence directory, simply delete these files. Once deleted, Cadence should then be able to run. However, you may experience problems when opening designs (Cadence will report that you only open as read only), in this case you will need to contact a member of staff/teaching assistant to remove the lock files – **do not** attempt to do it yourself!

Please note: we will not accept the presence of lock files to be grounds for mitigation for the late submission of work. You have been warned!

Experimental Boards

In this lab you will be able to download your designs to experimental boards, where you can see you designs in action! The experimental board consists of a smaller main board, containing a field programmable gate array (FPGA), plugged into a larger daughterboard that contains a number of devices, such as a keyboard, LEDs, and seven segment displays, as illustrated in Figure ii.

You will learn how to use the features on the experimental board as you progress through the exercises.

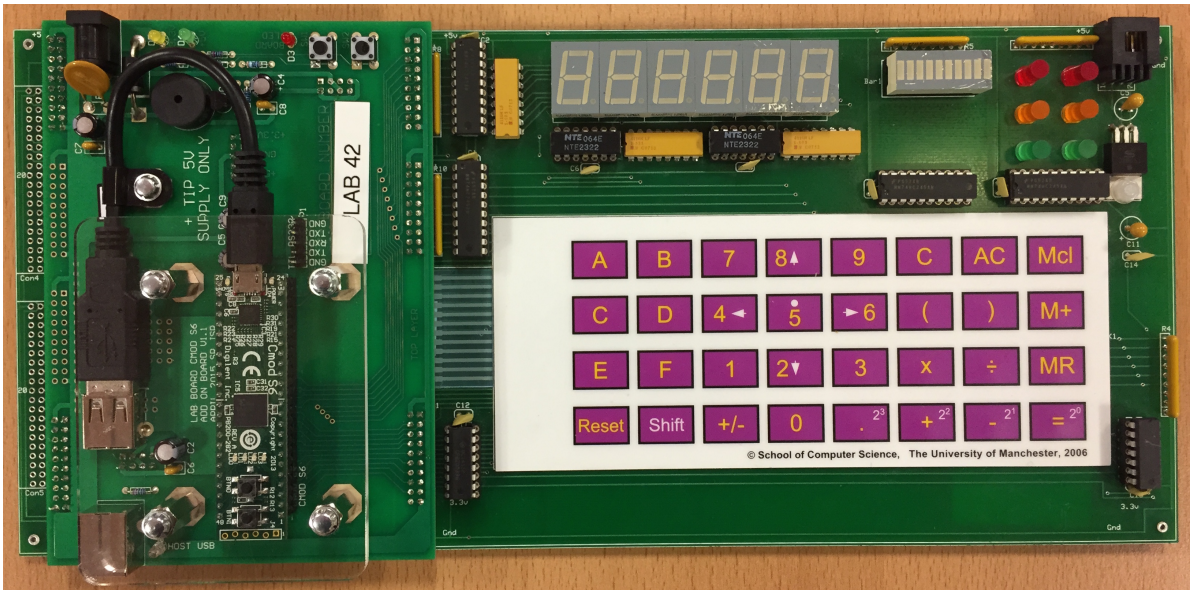


Figure ii: COMP12111 Experimental Board

Top Tips

Throughout this lab manual you will find hints and tips to help you with the exercises. These can be identified as the Top Tips boxes.

Faulty Equipment

Please report any faulty equipment to a member of staff/teaching assistant in the lab, or to Mr Steve Rhodes (corner office in Tootill 0) outside of a scheduled lab, so that it can be repaired.

Exercise 1: Binary Addition

Exercise Duration	2 lab sessions – weeks 1.2 and 1.3
Submission Deadline	End of your scheduled lab session in week 1.3. There are three submissions for the three parts of the exercise: ex1a, ex1b, ex1c.
Extended Deadline	Beginning of lab in week 1.4
Offline Marking	Yes
Demonstration Required	No
Feedback	Feedback from offline marking will be emailed in Week 1.4.
Assessment	This exercise does not count towards the final mark for the lab.

Aim

This exercise is intended as a hands-on introduction to schematic capture and the design tools you will be using for later exercises. In addition, the exercise should emphasise logic hierarchy, as systems are built from *black boxes* that in turn are constructed from simple gates. The logical design itself should present no difficulties.

Preparation

This is the first use of the Cadence software 'in anger' and - by necessity - must be performed in the laboratory using a PC booted to Linux.

The first part of the exercise is a tutorial in using the Cadence tools to enter designs, which you then simulate to test that the designs are working correctly; this is a standard approach to design that you must always adopt. There are no marks associated with this tutorial. However, you must complete the tutorial, as the half-adder design you produce will be used later in the exercise.

Cell designs, with schematic views, have been prepared for you for this exercise. Please do not change the name of the input/output pins, or add any further inputs/outputs.


This page
is
Empty

Cadence Tools Tutorial

A number of different methods of describing an electronic circuit exist, but by far the most common is the **schematic diagram**[1]. This is a picture of the circuit that shows the functional blocks and the interconnections between them. However, for larger circuit designs some means of managing this complexity must be introduced.

Large electronic designs, such as a processor, are extremely complex; indeed they are among the most complex systems produced by man. The reason why they can be produced at all is because they can be decomposed into successively smaller blocks in a controlled fashion. This is similar to the manner in which a software task is broken down into procedures and functions of manageable size; this then forms a hierarchical structure, as demonstrated in Figure 1.1. At the 'top' of the hierarchy is the system; this is decomposed into functional blocks which can usually be decomposed into smaller blocks and finally into basic gates. As far as this laboratory is concerned gates form the 'bottom' layer of the hierarchy, although these could further be decomposed into transistors[2] and then into bits of material (normally silicon).

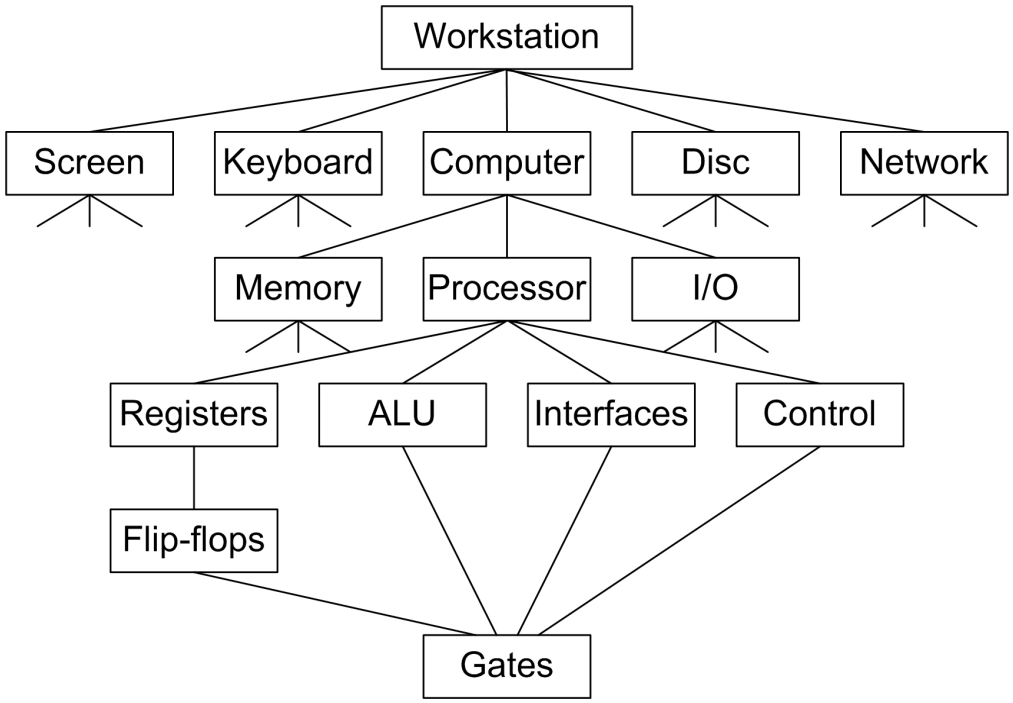


Figure 1.1: Design hierarchy

All complex electronic designs use design hierarchies. The lab exercises build these stage by stage, reusing the early, simple designs as part of later exercises. In addition, some 'library' components are already provided to assist with this process. During the exercises you will need to add your own parts to your own library in order to build a local design environment.

[1] Also known as a 'circuit diagram' or just a 'schematic'.

[2] Assuming we are using a particular electronic implementation. Almost all logic gates are currently electronic because they are cheap, fast, and convenient. There is no reason why gates cannot be made of other materials (matchsticks, Lego, etc.); indeed, pneumatic and fluidic gates are used in certain environments.

To assist with the design process of large circuits a number of computer aided design (CAD) tools have been devised. Arguably the most important weapon in this armoury is the programme that performs **schematic capture** (sometimes known as **schematic entry**), which allows the schematic diagram to be entered directly into the computer, maintaining the hierarchy and allowing relatively simple movement around the structure. The particular programme used in the laboratory is called **Virtuoso**, which is part of the **Cadence** design tool suite.

Once the schematic has been entered it may be used for a number of functions. Perhaps the most obvious is that it provides a (hopefully) neat means of printing documentation as a record of the design. However, the computer can also manipulate the design and - for example - may **simulate** its function so that it can be tested and debugged before it is implemented in physical hardware. The schematic can even be **compiled** directly into a physical circuit automatically, allowing the implementation of a design on a physical device without the need for wiring.

Drawing Layout

The actual layout of a schematic is an art that comes with experience. Clarity is a major objective and there are a number of guidelines available that may be broken with discretion. Some of these guidelines are implemented automatically when using the **Cadence** software. These comments apply equally to computer generated and hand drawn diagrams.

In general data, address and control signals should flow from left to right and from top to bottom of the page. Complex diagrams get crowded with signals and it is necessary to adopt certain conventions to reduce the complexity.

Where many wires are grouped together with a similar function, for example, where there are (say) sixteen wires representing a 16-bit binary number, then the signals may be grouped to form a **bus**.

It is easy to draw block diagrams without crossing lines, in order to give a neater appearance. However, this practice is not recommended for logic diagrams, and it best to allow lines to cross, where real connections are clearly indicated with a heavy dot, as shown in Figure 1.2. Sometimes lines that cross without connection are shown with a 'bridge' in the drawing; this style is now out of fashion and its use is discouraged (the CAD tools manage this for you anyway).

Individual signal paths need not be drawn continuously, especially if they are used in several drawings. If they are broken then they should be identifiable by sharing the same name and polarity. The output signals should have all their destinations flagged (by drawing number and - possibly - grid reference on that drawing). Input signals should have their source (only) flagged rather than a place somewhere in the middle of a chain of destinations that use the signal. This aids with tracing the source of a signal.

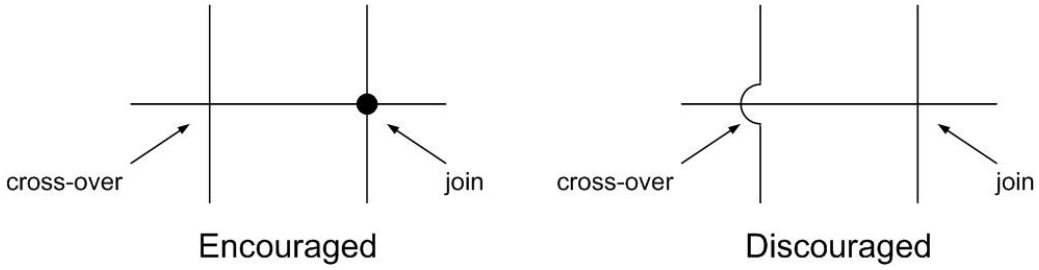


Figure 1.2: Signal wires crossing

To make a schematic even clearer one can also labels wires/buses to illustrate connections between different wire/buses without explicitly connecting them – see below.

Finally keep it neat. Remember that good documentation is important (if tedious).

Using Buses

In a programming language such as Java there are some basic *types* that can be used to define *variables*, two examples are *integers* and *Booleans*. A Boolean can only be TRUE or FALSE and so can be represented by a single bit. However, an integer can adopt any of a wide range of values as it is represented by several bits (typically 32 or 64).

When describing hardware it is often convenient to group bits together in a similar fashion. Such a grouping is called a **bus**. A bus can contain any number of signals which have the same name combined with an *index* that uniquely identifies each bit, (much like an *array* in a high-level programming language.) You will use buses extensively when you start using the CAD tools in later exercises.

It is not compulsory, but the normal convention for numbering bits in a bus is to express the range from a higher to a lower number separated by a colon. For example, a 32-bit bus might be called "`data[31:0]`", which we may use 4 bits, `data[11:8]`, elsewhere. The least significant bit of the bus is referred to as bit 0, the most significant bit of an *n*-bit bus is bit (*n*-1).

Note: using square brackets for buses, '[' ']', is the usual convention (particularly in hardware description languages such as Verilog). However, to confuse matters the Virtuoso schematic capture tool that you will use in Cadence uses angular brackets, '<' and '>', so that convention will be adopted for the remainder of this exercise.

Buses are particularly convenient in reducing wiring complexity, especially when crossing the boundaries of symbols.

Cadence

Cadence is an industry-standard tool that the University has access to via the Europractice framework. In order to use the software the University must adhere to an end user agreement (EUA) that states (amongst other things) that the tools must be kept confidential, must not be copied, and must not be used for commercial purposes. A copy of this end user agreement can be found in Appendix B. To enable you to gain access to the tools you should complete the form at

https://eua.cs.manchester.ac.uk/

failure to do so will result in you not being allowed to access the tools, and so will not be able to complete any of the lab!

In this tutorial you will be introduced to the Cadence design suite. The various design and simulation tools do have different names, but in the following the tools will usually be referred to generically as "Cadence" for convenience.

Software Setup

(Make sure that you have first agreed to the license agreement.) Before you start the exercises using Cadence you must run a script that will create the Cadence directory structure and system variables. In a terminal enter the following:

```
mk_cadence COMP12111 <return>
```

(`<return>` means press the enter/return key)

A directory, ~/Cadence, will be created, and within that a directory COMP12111 containing all the files you need to complete the laboratory.

WARNING!

It is important that you **DO NOT** change any of the files in the directory ~/Cadence/COMP12111. The files in this directory are managed by the Cadence tools. If you change these files (or delete them) manually then you may cause Cadence to break – this will take a lot of time and effort to fix.

DO NOT run the `mk_cadence` script again unless you want all your files (and your hard work) to be deleted. The script will ask if you are sure you want to delete the files.

Starting Cadence

You should start the Cadence design framework tools by typing the following in a terminal:

```
start_cadence COMP12111 <return>
```

The Cadence Integrated Circuit Design and Simulation window (*Virtuoso*) will eventually start, as shown in Figure 1.3. This provides access to the design and simulation tools and to components via the menus. It also provides a text report stating the actions that Cadence has performed, and occasionally error reports. This window can be moved, shrunk, grown, or iconified; growing it vertically will allow you to see more lines of text, as any error messages are listed in the dialog box.

The *Virtuoso* window is extremely useful as any errors in your designs will be reported in the dialog box.

In Cadence the actions associated with the three mouse buttons depend on which tool is in use and on the context (i.e. what you are doing right now). The current mouse button actions are always shown at the bottom of the window. When Cadence first starts these are blank (as shown in Figure 1.3).

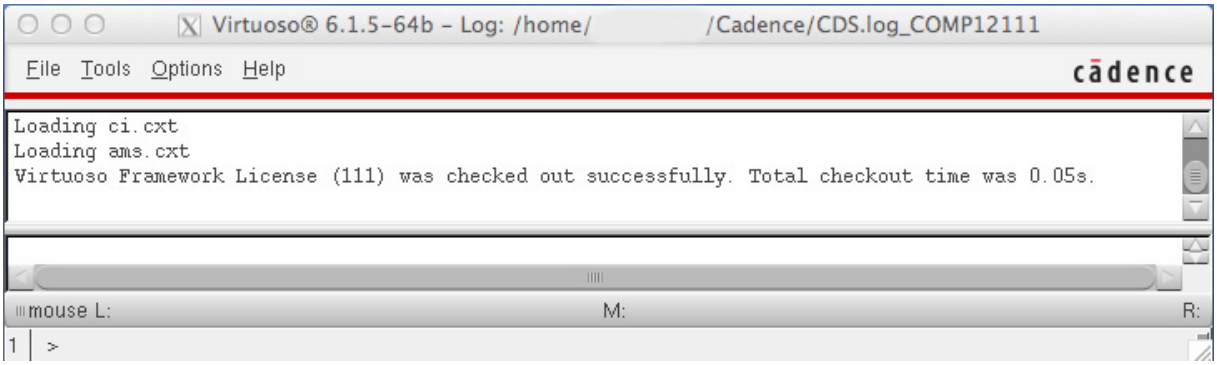


Figure 1.3: Cadence Virtuoso

In the following descriptions:
LMB means Left Mouse Button
MMB means Middle Mouse Button
RMB means Right Mouse Button

Mouse button actions (including selecting menu items) are shown in the Button Style, so File ↲ New ↲ Cellview … is a sequence of mouse clicks on menu items.

The other useful key to know is 'escape' (Esc). Which gets you out of most accidentally invoked functions and ends many repeated actions, such as placing gates or wires.

If things go wrong, there is an 'undo' facility that can reverse the last few commands.

Using the Library Manager to manage your designs

It is essential that you manage your designs using the library manager provided in Cadence. The library manager can be opened from the Tools menu in the *Virtuoso* (Tools ↲ Library Manager ….) and is shown in Figure 1.4. So please make sure you use it to open designs.

Designs are identified as cells in a particular library. You will place your designs in the library COMP12111, although you will use designs from the spartan6 and MU0_lib libraries in later lab exercises. Each cell can have a number of views, such as schematic, functional, and symbol. You will learn more about these later.

If you right click on a cell name in the cell list, a floating menu will open from which you can copy, rename and delete a design. Because Cadence maintains various internal data records, which may not be apparent from the directory structure, you should **only** access and manipulate your components or designs (i.e. edit, copy, or delete them) via the Cadence Library Manager. **DO NOT** attempt to move, copy or modify them directly from a command line or file manager.

To open a file for editing from the Cadence library manager window, simply right click on the view (schematic, symbol, functional) of the cell you wish to edit, and select "Open …" from the menu window that appears, or double click on the cell view you wish to open.

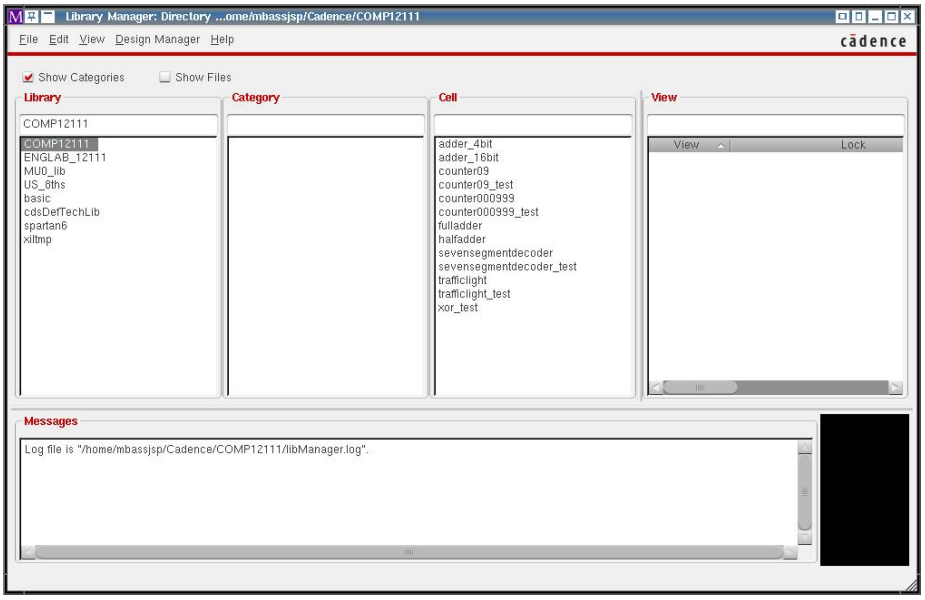


Figure 1.4: The Cadence Library Manager window

Exiting Cadence

To exit Cadence select File ↲ Exit from the *Virtuoso* window. **WARNING**: if you do not exit Cadence properly then it may result in lock files being left that will cause problems running Cadence in the future and/or problems editing your designs.

Creating Your First Designs

In this exercise you will gain familiarity with the tools by shown how to create two designs:

- An XOR gate,
- A half-adder.

As you progress through the design you will be introduced to the various aspects of the design process, such as simulation. Later you will be required to produce your own designs for a full-adder, 4-bit adder, and 16-bit adder using hierarchy as you progress. You will be required to submit these later three designs, for which you will receive feedback, but they do not count for the assessment for the course unit.

A Simple XOR Gate

Schematic diagrams consist of a number of component symbols connected by wires (or nets). They are normally constructed hierarchically such that a given component may consist of many sub-components. Schematic diagrams (often abbreviated to just "schematics") can be linked together by the use of labelled connectors.

The first thing to do is to open a design using the Cadence Library Manager (in Virtuoso click Tools ↲ Library Manager…), select the Cell name "**xor_test**" in the library COMP12111, right click on "schematic" in the View pane and select "Open". A new window "Virtuoso Schematic Editing" should open, as shown in Figure 1.5, in which you can create your design. As you can see from Figure 1.5 the schematic entry window has a familiar look and feel, with a menubar and a taskbar.

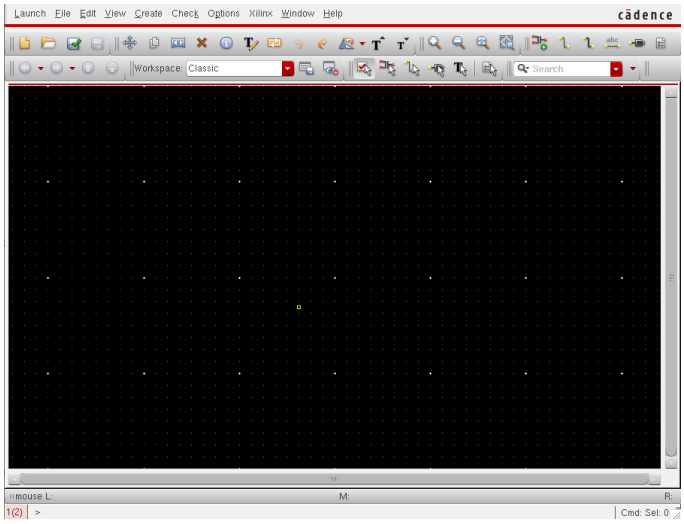


Figure 1.5: Virtuoso schematic editor

You are going to produce a design for an XOR gate and test it. You could simply insert an existing XOR gate as shown in Figure 1.6, however, as an exercise in design you are required to implement an XOR design without using an XOR (or XNOR for that matter!) logic symbol.

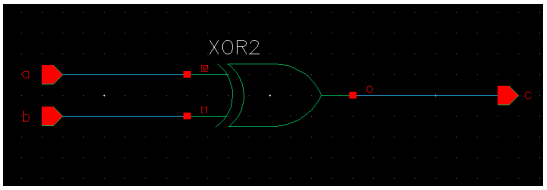


Figure 1.6: XOR test circuit

Figure 1.7 illustrates the truth table for the XOR gate.

1. Using the techniques you have been shown in the lectures produce a sum-of-products expression for the XOR function from the truth table.

Inputs		Output
a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

$Q = (A + \bar{B}) \times (\bar{A} + B)$

Figure 1.7: XOR gate truth table

2. Once you have produced a logical expression for the XOR function you can then translate this design into a hardware design. The first thing to do is place some components (a sum-of-product expression will often require AND and OR gates to be used). Components are stored in **libraries**. The library of basic gates that are used in this lab already exists and you can select and use gates from it. The easiest way to add a gate from this library is from the pull down menus: Create ↵ Instance… on the menu bar at the top of the Virtuoso window, this will open an 'Add Instance' dialog box. To select a component, click the Browse button to the

right of the Library text entry box, which will open the Library Browser window to enable you to browse all available components.

In the Library Browser, select the spartan6 library (click on the name spartan6 in the "Library" column), which should produce a Category list. You can then select a list of designs for different logical functions, as illustrated in Figure 1.8. For each logic gate you require in your design select the symbol view using the LMB and place it on the schematic area by moving the cursor until the gate is in the chosen position in the schematic editing window and then pressing the LMB.

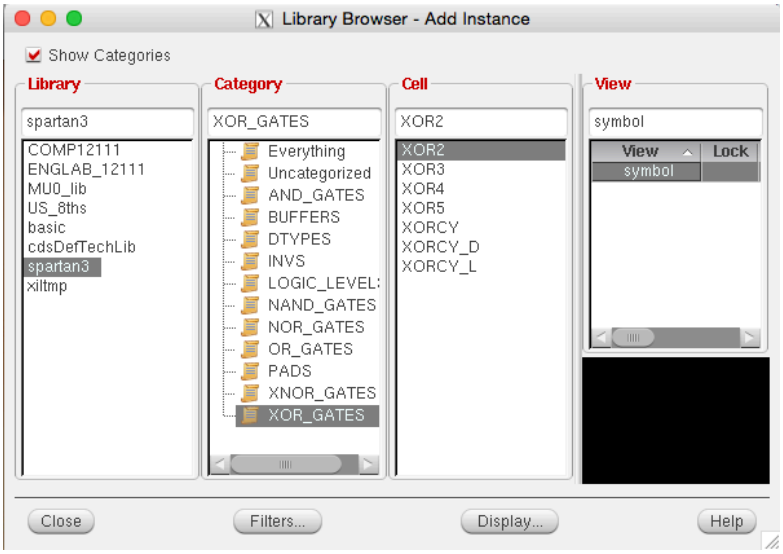


Figure 1.8: spartan6 library components listed in the library browser window

Components can be moved when they are selected (white). To move a component, select it (LMB) and then select RMB ↲ stretch then select the component again to move it. Note that everything selected will be moved as a group. The stretch command is very powerful, but difficult to get used to. Start by only moving a small number of items at once. If you begin something you didn't mean, just press Esc.

3. Now you are going to add the inputs and the output to your XOR design – these are how connections are made to your design from the outside world. Inputs and outputs are called 'pins', to add a pin select Create ↲ Pin…, and the 'Add Pin' window should appear. Set the 'Direction' to be 'input', and enter 'a b' (note the space between 'a' and 'b') in the 'Pin Names' text entry box and then press the Enter key (this will allow you to place two input pins, one called 'a' and one called 'b'). Place the two pins.

4. Follow the same procedure to place the output pin, 'c', but remembering to select the pin direction as 'output'.

5. Now the components are positioned correctly they must be joined with wires. Select Create ↲ Wire (narrow), or press 'w' on the keyboard, and then place wires to connect the components to implement your XOR design – that is connecting input/output pins to the inputs/outputs of logic gates, and connecting outputs of logic gates to inputs of logic gates where appropriate. Point at the start position of the wire, click the LMB to start a wire, and at the end of the wire. If you wish to end a wire without connecting it, double click the LMB at the end point of the wire.

   By default, once you begin placing wires, the tool will do this repeatedly. 'Escape'

or Cancel will terminate this, or the 'repeat' option can be switched off in the small pop-up box.

Connectivity is maintained when items are moved using 'stretch', whereas when using the 'Move' operation the connectivity is broken.

A wire must be placed onto a component pin (or vice versa) to make a connection. Merely moving one on top of the other does not make a connection; if this occurs a warning symbol appears at the appropriate position.

6. You have now completed a schematic design for the XOR gate. Save the schematic: File ↲ Check and Save, or press the icon (disc with a green tick on it) on the toolbar. Any issues with your design will be reported in the *Virtuoso* dialog box, so look in the *Virtuoso* window and confirm that the design checked and saved correctly – you should see messages saying:

   Schematic check completed with no errors.

   "COMP12111 xor_test schematic" saved

   If may be that *Virtuoso* reports some error, in which case there may be something wrong with your design. If so, check your design, potential problems will be highlighted on the schematic, and there will be error messages in the *Virtuoso*. If you can't see the fault, ask a TA before continuing.

   There are two types of problems you may reveal:

   - **Errors** - are definite problems: the circuit as drawn cannot be realised.
   - **Warnings** - are possible problems: the circuit can be built but you may have missed something. A typical 'warning' would be generated by a wire that just stops in space (making no connection).

Top Tip

Delete

Pressing 'Delete' or 'Del' on the keyboard or 'Edit ↲ Delete' on the menu will delete everything you have selected. Make sure that you have only selected the parts you really want to delete.

Any erroneous editing – including deletion – may be corrected using Edit ↲ Undo (or u on the keyboard), which will undo the previous command. Several successive actions can be undone.

Pressing 'esc' will cancel the current operation.

Top Tip

What's the difference between "Wire (narrow)" and "Wire (wide)"?

In reality nothing whatsoever. However, you should use "Wire (narrow)" when drawing single wires, and you should use "Wire (wide)" when drawing buses, so that you can easily differentiate between them in your design.

Simulating the design

Simulation is the process by which a designer verifies that a digital logic circuit does what is intended; it is an **essential** part of the design process. 'What is intended' is defined by the specification, which should be created at the start of the design process (whoops!). In the case of the XOR gate example the specification is encapsulated in the truth table shown in Figure 1.7, and so verification can proceed simply by generating all possible input combinations and checking that the outputs of the circuit match the desired outputs in the truth table.

The design will be simulated using the **Virtuoso Verilog Environment**, which will display the output in a separate window, showing graphs (or traces) of the values of selected pins and wires as a function of time.

Why Simulate a Design?

The role of simulation is to enable you to test your design to see that it works as expected, i.e. to test your design conforms to its specification. Testing is **ESSENTIAL** part of the design process and should be done when designing a new component, hardware or software, and helps to ensure that the product you deliver is working how the customer expects it to.

In hardware design testing is performed by assigning values to the **INPUTS** of your design and observing the **OUTPUTS** via simulation – as illustrated in Figure 1.9. Input values are provided by means of a Verilog stimulus file – see later.

By comparing the state of the output signals with the desired response (using the truth table) you will be able to determine whether the design is working as expected. In the case of sequential designs, the testing may involve the design being tested in a time dependent manner.

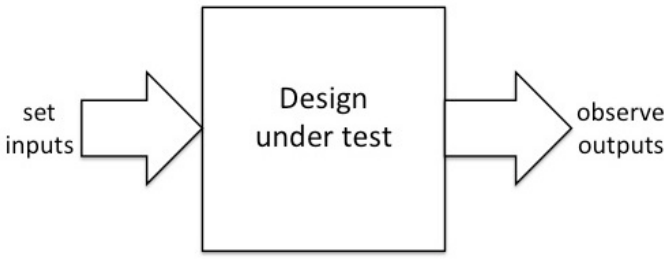


Figure 1.9: Design under test

Starting the Simulator and preparing the design for simulation

1. Open the schematic design of the design you want to simulate (xor_test in this case).
2. Select Xilinx ↲ Simulation from the menu, this will open the "Virtuoso Verilog Environment for NC Verilog Integration" window as shown in Figure 1.10. The simulation window will be completed for you, so you do not need to worry about specifying anything.
3. Click the 'Initialize Design' button (the one of the left hand side with a picture of a running man on it), you must **always** do this first. Then click the 'Generate Netlist' button below it (looks like a stack of boxes). You may see a popup window asking if it is OK to continue, if so select 'Yes'.

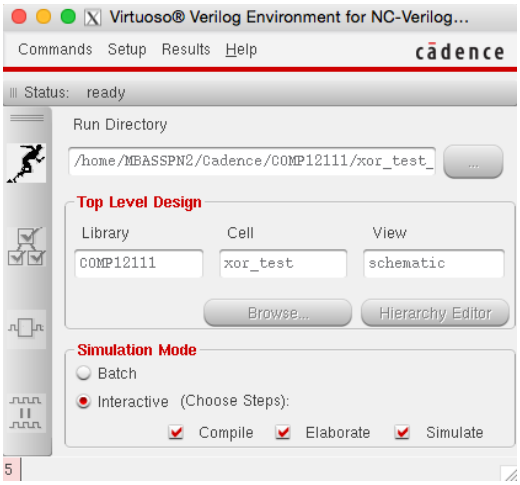


Figure 1.10: Virtuoso simulation window

Specifying the stimulus data

You now need to instruct the simulator to set the inputs to the `xor` design to different values in sequence, so that you can verify that the outputs are always correct.

1. Select Commands ↲ Edit Test Fixture. This will first open a text editor window in order for you to edit the stimulus information for testing your design, this file is named 'stimulus.v'. **Do not delete any of the code that is provided; just include your own code in the appropriate place**. The code is written in Verilog, a language created specifically for describing sequential processes in digital systems. There are many sources of information on Verilog if you wish to learn more[3], but for this lab. you only need a very limited subset of the language.

   ### What am I testing?

   When simulating a design you are providing values for the input signals, in the case of the `xor` gate the inputs `a` and `b`, and looking at the state of the output signals, `c` in this case, for each combination of the input signals. Comparing this with the specification (truth table) will help you determine whether your design works as expected. However, in the case of simulation we are looking at **timing waveforms**, which show the status of signals as a function of time.

   In this case we need to set the 4 possible combinations of the inputs a and b. We set the value of inputs by assigning values to them. However, we must also wait some time before we change the values to enable the output(s) to stabilize.

2. You now need to enter the stimulus (in order to test your design) into the stimulus.v file that you have opened. Figure 1.11 illustrates the required test stimulus. Enter this into your stimulus.v file below the line that says '// Enter your stimulus below this line'.

   You can see from the stimulus file that the inputs are assigned a value, and we are waiting some time, using the #<number> to specify a delay in ns, before assigning new values to the inputs. In the case of the half-adder initially we set both a and b to 0, wait 20ns, then change b to 1, wait 20ns, then change b to 0 and a to 1, wait 20ns, then change b to 1 and then wait a final 20ns. Here, we are testing every possible

[3] see http://www.asic-world.com/verilog/veritut.html

combination of the two input signals.

You will notice that the default stimulus file you are given has a line `$stop`, this tells the simulator to stop the simulation. It is always worth adding some delay before the `$stop` so that any input signals changes can have an effect before the simulation stops.

3. Save the file that you have edited, and close the window.

```
// Basic xor gate test
// Inputs a,b
// Outputs c

// Initialise inputs at the start of the simulation
      a = 0;      // set input a to 0
      b = 0;      // set input b to 0
      #20         // Wait for 20ns to allow the adder to settle

// Cycle through the input combinations,
// pausing after a change in the input to take into
// account the delays in the circuit and to enable the
// change in the output to be observed.

      b = 1;
      #20
      b = 0;
      a = 1;
      #20
      b = 1;
      #20
```

Figure 1.11: Stimulus data for testing the operation of the half-adder

Simulating the design and displaying the results

In a complex design there may be millions of gates, and it would be impractical to observe the logic level ('0' or '1') of all nodes in the system as it was simulated. The next stage of simulation is to select the signals that you want to observe the behaviour of. Cadence provides a 'Design Browser' that allows you to browse through the design and select the points of interest.

1. In the "Virtuoso Verilog Environment for NC Verilog Integration" window, click the "Simulate" button (third button down on the left hand side). This should (after some delay) open two windows, the "SimVision" Design Browser and the "SimVision" Waveform Viewer. Find the "Design Browser", which should look as shown in Figure 1.12 (it may well be hidden). If the Design Browser does not start, the most likely explanation is a syntax error in your Verilog simulation file, in which case a message window saying "NC-Verilog Compilation Step Failed..." will have appeared. Click Yes to see the error messages from the compiler, and go back to editing your Verilog simulator file (Stage 2).

   The Design Browser shown in Figure 1.12 allows you to browse into the hierarchy of the design in a manner similar to directories. Feel free to expand the top level and explore the instances below.

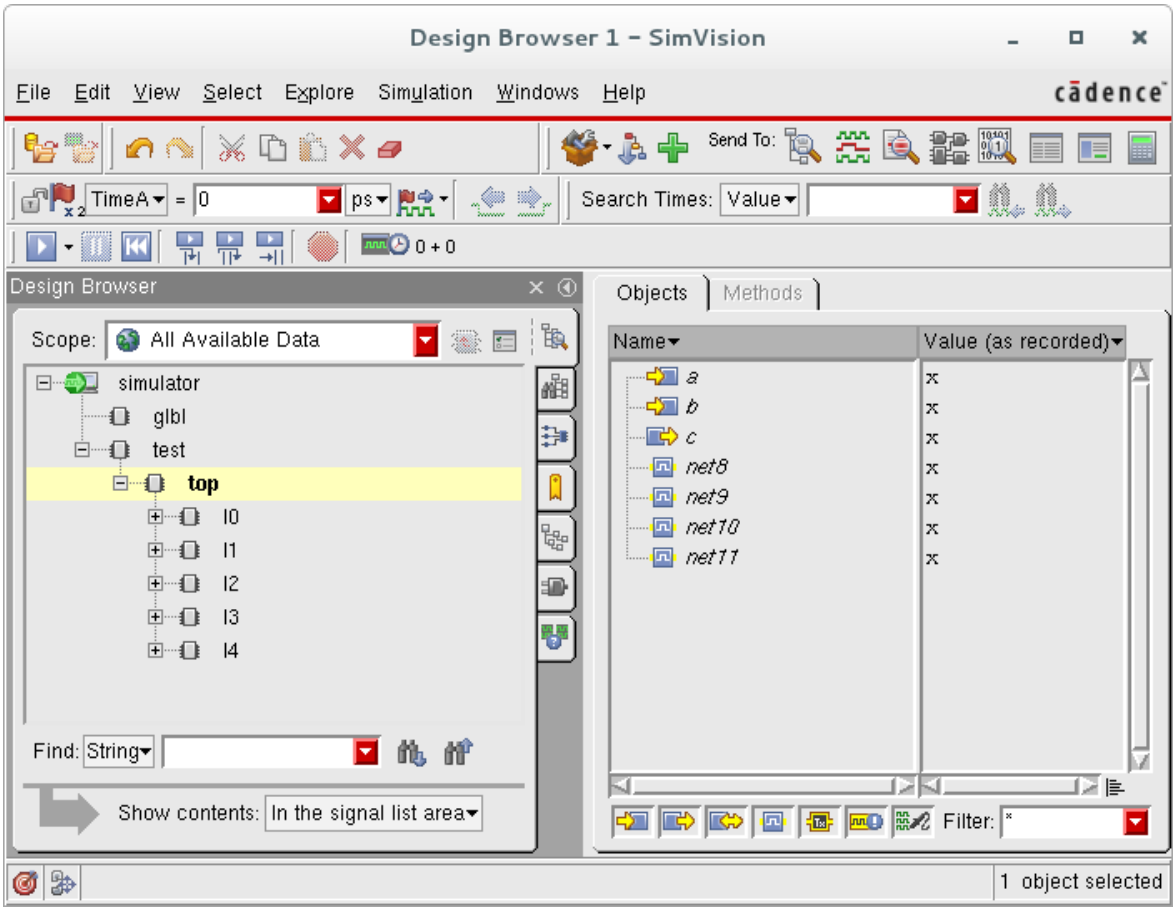


Figure 1.12: SimVision Design Browser

2. In future you will want to see the logic state at intermediate points in the design, so you will want to select other wires, or inputs and output(s) of individual gates. For now you can just compare the inputs and output of the xor gate with the truth table. Select all of the inputs and output of the top-level design (*a, b, c*) in the right hand pane (to select more than one input, hold down 'Ctrl' as you click), and click the send to waveform button (icon with the green and red horizontal traces on).

Observing the behaviour of the circuit

1. Find the SimVision waveform viewer, the signals you sent to the waveform viewer previously should be listed in the left-hand window.
2. Click the "Run" button (the "play" button – blue button with a white triangle on it's side) and the waveforms (or traces) should appear in the right-hand window, as illustrated in Figure 1.13. Note: you can use the scroll bar at the bottom of the waveform viewer window to change the time scale shown, so you can zoon-in (in time) on certain parts of the waveforms.
3. Compare the simulated behaviour of your circuit as the inputs change. Does it perform as required? Compare the behaviour with that suggested by the truth table in Figure 1.8. You should find it works as expected as you haven't had to do much to create this design. However, if there are problems, such as the output being undefined -'X', then there is a possibility you have made a mistake connecting the wires. Check your design if this is the case. Why are the signals red (or undefined) for the first 100ns? Notice that when the inputs change there is a delay before the outputs change. Why is this? Would that happen with a real gate?

You can identify individual gates (labelled in yellow/gold by their instance value) if you expand the top-level component in the left hand pane of the Design Browser (Figure 1.12). This would allow you to select any outputs you might be interested in and pass them to the waveform viewer. (You can't do much with this deign as it only has one gate.)

You can identify specific wires (nets), for this you need to name the nets.

4. There is nothing to hand in for this exercise as it isn't assessed. Close the simulator and waveform viewer.

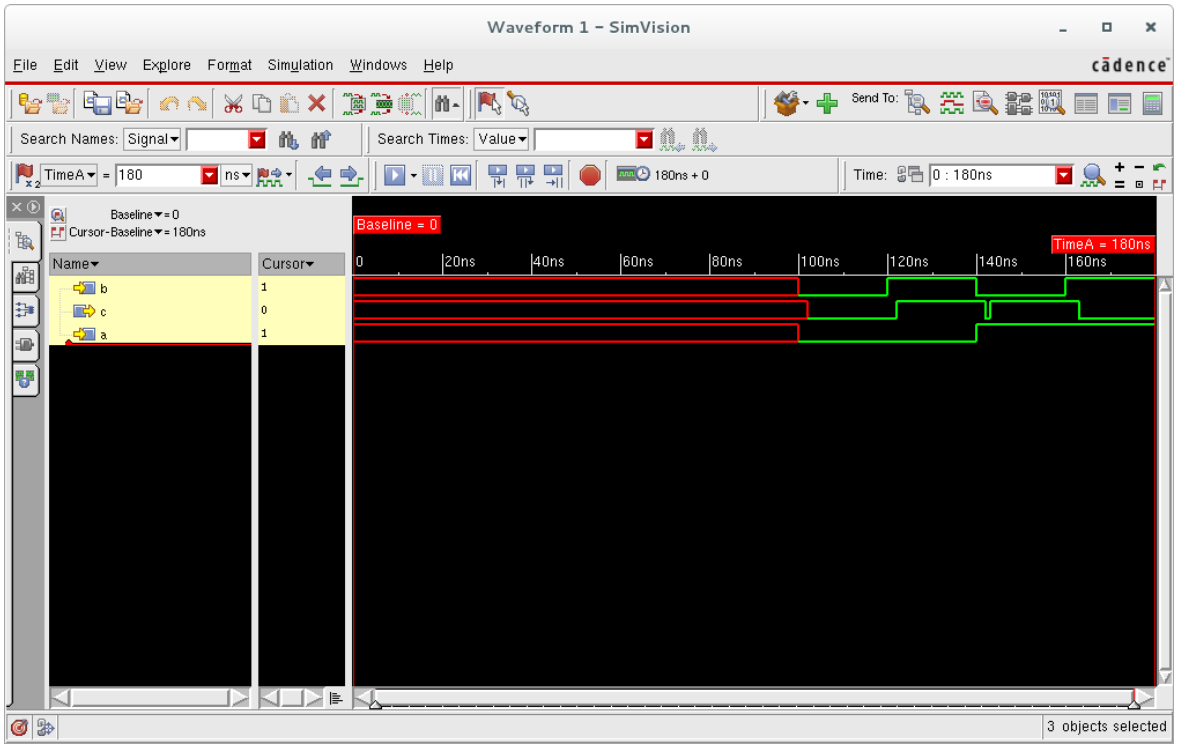


Figure 1.13: SimVision waveform viewer

The Halfadder

You have used Cadence to create a simple circuit consisting of an XOR gate, which you have tested to ensure the design works as expected. You will now design a more complicated circuit that contains a number of gates, which you will re-use (hierarchy) later in the exercise.

Open the "schematic" view of the Cell name "**halfadder**" using the Library Manager. A new window should open an empty design.

Next you are going to create the design of a halfadder as shown in Figure 1.14. Add instances of the gates used (AND2, OR2 and INV), add the input and output pins (inputs labelled "a" and "b", outputs labelled "sum" and "carry"), and add wiring as shown.

Check and save your design and check that there are no errors.

Note: in the XOR and half-adder designs we have asked you to place the input/output pins in the schematic. In later designs these will be provided for you. When choosing names for pins, name them by function where possible, but do make your design reusable in other, future circuits. Although the outputs can describe the relationship between the outputs and the inputs (as here), the function of the input pins sometimes

isn't obvious – what the numbers that are to be added are depends on what circuit the adder is placed in. So input pins are often given generic names such as 'a' and 'b', except where they have a clear function such as 'clock', 'enable' etc. Do not start pin names with numbers '1A' etc.

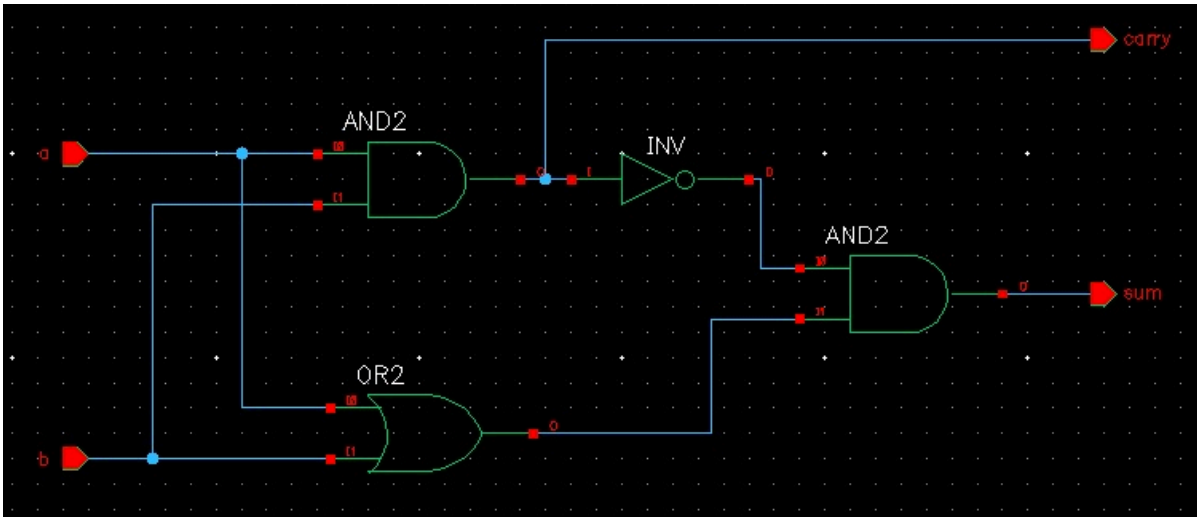


Figure 1.14: The half-adder schematic

Symbol design

You have now produced a design for a half-adder. However, if you need to use this design in another design then you have to produce a symbol for your design (this is abstraction), as it is the symbol that will use if you want to use the half-adder in a more complex system, such as a full-adder (which can be made using two half-adders – can you guess what's coming next?).

The symbol is not just a drawing. The symbol specifies the interface to the component, i.e. the inputs and the outputs. To do this, the symbol must have pins, whose name and connectivity matches the pins on the schematic diagram for the component.

Thus, if the halfadder schematic has two input pins 'a' and 'b', and two output pins 'sum' and 'carry', the symbol must show the same pins **with the same names.** That is how Cadence knows that a wire connected to the 'a' input of a half-adder symbol is meant to connect to the 'a' input pin of the halfadder circuit (as described by the schematic).

Symbol generation can be done semi-automatically:

1. In the Schematic Editor select Create ↲ Cellview ↲ From Cellview. This will open a "Cellview from Cellview" window.
2. Check that 'From View Name' reads "schematic", and 'To View Name' reads "symbol", and click OK. A window "Symbol Generation Options" should open, click OK.

   A symbol will automatically be generated that looks similar to Figure 1.15.\

   What do the labels mean? @partName is text that will appear on the symbol. The default is that the symbol text will be the same as the name of the design, so in this case "@partName" would be automatically translated to "halfadder" when the symbol is eventually used in another design. When the half-adder is used in another design, the

actual Instance Name will be inserted at the location specified by "@instanceName". The Instance Name is a unique identifier for every component in a design – e.g. it can be used to distinguish between multiple half-adders in a full-adder. Normally the first component you place in a design will be I0, the second will be I1, the third I2, etc. We recommend you don't change these names.

The red rectangle that passes through all the pins is the 'selection box'. This does not appear when the component symbol is included in another schematics, but it does define the bounding region of the symbol, which is used to select the component and to check for overlaps between components.

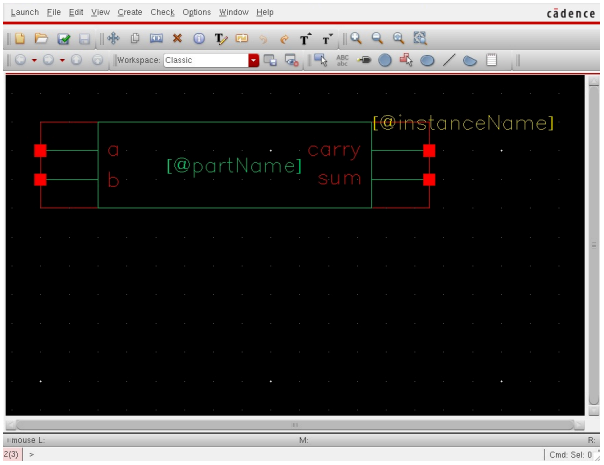


Figure 1.15: A possible symbol for a half-adder

3. You should now check and save your symbol design, as you did for the schematic, remembering to check the messages in *Virtuoso* to confirm success. To close the window select File ↲ Close All.

The half-adder could now be used in other designs (it may *not* be used in its own schematic, for obvious reasons, but you will want to use it in a full adder). However, it has not yet been tested, so using it now would be ***very unwise***. How do you know if it works? Testing your designs is a **vital** part of the design process (some may argue it is the most important!). To test a component operates as we expect it to you have to use simulation tools in order to make sure the design meets the specification.

You did write a specification before you started designing, didn't you? As a professional designer you should be answering: "Yes, I completed a truth table for a half-adder".

Simulating the design

Simulate the design as you did for the XOR gate - open the schematic and select Xilinx ↲ Simulation from the menu, this will open the "Virtuoso Verilog Environment for NC Verilog Integration". Click the 'Initialize Design' button, then click the 'Generate Netlist' button below it.

You now need to instruct the simulator to set the inputs to the half-adder to different values in sequence, so that you can verify that the outputs are always correct. Open the test fixture as you did for the XOR gate and add the same stimulus data (this design also only has two inputs). Save the file that you have edited, and close the window. Simulate the design by clicking the "Simulate" button in the "Virtuoso Verilog Environment for NC Verilog Integration" window to open up the design browser. Observe the waveforms to determine whether your design works as expected.

It probably seems very tedious to simulate simple circuits. However, as circuits get more complex this debugging aid becomes essential, and in hierarchical systems (such as the one you are about to build) it is vital that components are tested before being used. **Verifying your circuits by simulation before compiling them into hardware *will* save you time**.

As the half adder has a limited number of inputs then it is sensible to test its behaviour exhaustively, in this case you would be expected to vary each input independently for **ALL** possible combinations of the input signals, you have seen an example test stimulus. It is vital that **ALL** inputs are assigned a value initially as any undefined inputs may result in one or more of the outputs being undefined. So it is often best to initialise all inputs to zero at the start of your simulation.

Once you have tested the half-adder design exhaustively, and it works as expected then it should work in any future designs where you re-use it.

Top Tip

Summary of useful Verilog instructions for simulation

- You can add comments. Comments begin `//` and last for the rest of the line
- #x creates a delay of (waits for) x ns (10^{-9}s).
- `a = 0` sets the value of input '`a`' to `0`, `a = 1` sets the value of input '`a`' to `1` (it can also be `x` and `z`) – you can also assign a decimal value if the variable represents a bus, i.e. `a = 5;`
- You need a `$stop;` command at the end to tell the simulator to stop simulating once it has reached the end of the file (note: this is already there!).

The Full-Adder

The next stage of the lab will involve you using your half-adder design to complete a design for a full-adder as shown in Figure 1.17. A cell 'fulladder' with view 'schematic' has been provided for you for this exercise. From the Cadence library manager open the schematic for the cell fulladder, notice that the interface (the input/output pins) has been provided for you – DO NOT change these.

Create the full-adder design using the half-adder you produced previously following the design shown in Figure 1.17 and create a symbol for your full-adder.

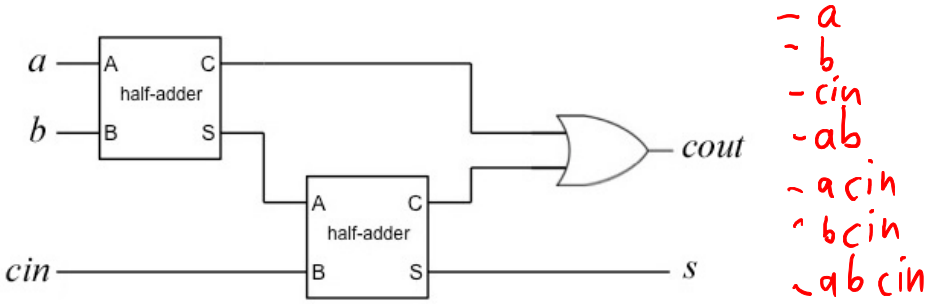


Figure 1.17: The full-adder circuit

Simulating the design

Simulate your full-adder and observe the output waveform to confirm it is working correctly.

Do you need to test your design exhaustively in this case? In this case, as there are so few inputs, then it is probably best to test all possible combinations of the input signals (eight test vectors).

ex1a: Full-Adder

Once you have completed your design and simulated it you can submit it for marking using the 12111submit script. Follow the instructions and submit the design as 'ex1a' and enter 'submit'. Please note: if the design fails to submit then this is probably because you have not simulated it.

Top Tip

Exploring the Hierarchy

When using a symbol within a schematic it is possible to explore the content of that symbol. As an exercise try selecting one of the half-adder symbols in your full-adder schematic and execute the function **Edit ↲ Hierarchy ↲ Descend Edit** …, (which is most easily obtained using the 'E' keyboard accelerator). This will allow you descend into the design of the half-adder where you could, if you like, modify its design. Note: if you change your half-adder design then it will affect both the half-adder instances in your full-adder schematic. To return to your schematic select **Edit ↲ Hierarchy ↲ Return** up the hierarchy. This function is useful with your own components (and, possibly, the local library "ENGLAB_12111") but the basic gates are 'bottom-level' models.

4-bit Adder

A multiple bit adder can be implemented by instantiating the required number of full-adders. Figure 1.18 illustrates the example of a 2-bit adder. Use this example as a basis for your design of a 4-bit adder that takes two 4-bit inputs and a carry in and produces a 4-bit sum and a carry out. You may want to look at the top tip box in the following pages where it is explained how to use wire names to aid readability of a design – you can use pin names to label wires/buses in your design as they are "seen" at this level of the hierarchy.. A design 'adder_4bit' with view 'schematic' has been provided for you for this exercise. Use this to produce a 4-bit adder by creating instances of your 1-bit adder symbol. Create a symbol for your 4-bit adder.

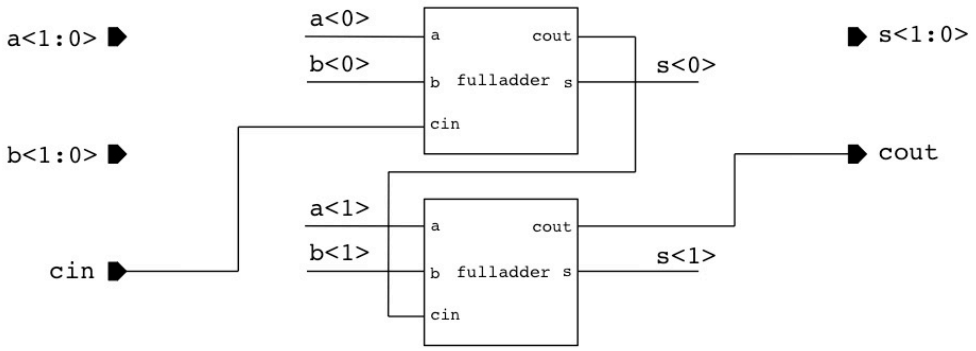


Figure 1.18: A 2-bit Adder

Simulating the design

Simulate your 4-bit adder and observe the output waveform to confirm it is working correctly. An incomplete stimulus file has been provided for you, however, you must complete it. For the half- and full- adders you tested all possible combinations of the input signals. However, you do not need to test as extensively for the 4-bit adder. As complexity of a design grows you may choose to be more selective in what you test as elements of the design may have already been tested so you should have that they work as expected.

Consider the simple 2-bit adder illustrated in Figure 1.18, what test strategy should you adopt for this design? The 2-bit adder consists of two full adders, which have been tested exhaustively, so we are confident that these operate correctly. What has been added here are the input and output connections, and the carry path from bit 0 to bit 1. So the testing strategy here would be to test the connections we have made at this level of the hierarchy.

The first test should always be to set all inputs to 0 and check that a 0 or 1 is observed on all the outputs, if not and a undefined, x, or a high impedance, z, are observed, then there is a problem with a connection somewhere in the schematic.

How do we test the inputs and outputs are connected correctly? One way to do this is to assert each input independently and check that we observe the correct sum signal at the output going high (as we know the full adder works). So the strategy would be:

1. Set $a_0 = 1$, $b_0=c_i=a_1=b_1=0$, and check that the expected output, s_0, goes high, if it doesn't, or a different output goes high, then we know either the input a_o, or the output(s), are connected incorrectly.
2. Repeat for $b_0 = 1$, $a_0=c_i=a_1=b_1=0$, testing s_0.
3. Repeat for $c_i = 1$, $a_0=b_0=a_1=b_1=0$, testing s_0.
4. Repeat for $a_1 = 1$, $a_0=b_0=c_i=b_1=0$, testing s_1.
5. Repeat for $b_1 = 1$, $a_0=b_0=c_i=a_1=0$, testing s_1.

You should always ensure you give enough delay after each change in the input(s) to allow the output to change accordingly.

All the input and output connections will now have been tested.

What about the carry path? The carry path will be exercised when two of the input to the first full adder are asserted, in this case a carry out will be generated which will feed into the sum of bit 1. We wouldn't want to set all three inputs though, as this would exercise both s_0 and c_0, and we should only be testing one action at a time. So to test the carry from bit 0 to bit 1 one test would be to set the inputs to $a_0 = b_0=1$, $c_i=a_1=b_1=0$ and check that just

the output s_1 goes high. If it doesn't, then we know the connection hasn't been made correctly.

Testing an n-bit adder simply involves extending these tests to all n inputs, and checking all n-1 carry paths between bits independently using the same process.

ex1b: 4-bit Adder

Once you have completed your design and simulated it you can submit it for marking using the 12111submit script. Follow the instructions and submit the design as 'ex1b' and enter 'submit'.

Please note: if the design fails to submit then this is probably because you have not simulated it.

A 16-bit Adder

Following the design of a 4-bit adder, you can now extend the design to produce a 16-bit adder. A cell "adder_16bit" has been provided for this exercise. Produce a design for a 16-bit adder based on the 4-bit adder you produced previously. Create a symbol for this design.

Simulating the design

Simulate your 16-bit adder and observe the output waveform to confirm it is working correctly. An incomplete stimulus file has been provided for you, however, you must complete it. Follow the advice given for the 4-bit adder to produce a suitable test stimulus to test the operation of your 16-bit adder.

ex1c: 16-bit Adder

Once you have completed your design and simulated it you can submit it for marking using the 12111submit script. Follow the instructions and submit the design as 'ex1c' and enter 'submit'.

Please note: if the design fails to submit then this is probably because you have not simulated it.

You have now completed Exercise 1.

Top Tip

Buses

For the 4-bit adder you may notice that the inputs, *a*<3:0> and *b*<3:0>, and output *s*<3:0>, are each 4-bits wide – these are buses. It is good practice to use buses for all inputs and outputs that contain more than a single wire, as this will make connectivity at a higher level in the hierarchy much easier. To make your schematic neater you should consider using wire names to infer connections from the input/output pins to the individual input pins on your full-adder symbols. There is no need to add a bus and label for each input/output pin as the pin names are 'seen' at this level of the design hierarchy.

Top Tip

Using Wire Names to Aid Readability

Whilst it is often convenient to show the buses/signals physically connected in the schematic, it is not necessary. In fact, connecting many wires/buses together in your schematic can often make it look messy and difficult to interpret. In the Vituoso schematic capture tool any set of wires/buses with the same name are connected by default. The tools take care of this for you. Thus, individual wires `data<0>`, `data<1>`, `data<2>` and `data<3>` can be taken from a bus `data<31:0>`.
The use of wire names is helpful (and recommended) to infer connections between wires/buses; the CAD software will make the connections when it synthesizes the design into hardware. Figure 1.18 illustrates connections between buses inferred using wire names. Here, we have three input pins, two of which are 2-bit buses, and 2 output pins, one of which is a 2-bit bus. Each has a wire connected to it with a label attached. The same wire names can be used to then infer connections between wires. For example, the bus input `a<1:0>` has a wire labelled with the same name (and bus width!). Bit 0 of this is connected to the top full-adder, as inferred by the label `a<0>`, the remaining bit of `a<1:0>`, i.e. `a<1>` is connected to one of the inputs for the bottom full-adder. Use of wires name thus makes the schematic much easier to read. When compiled the CAD software will make the connection between labelled wires for you.
In the Cadence schematic editor (more later) you can use wire names from the menu (Add ↲ Wire Name …). If you have a bus labelled as `data<31:0>`, you can connect a wire or bus to this bus by using wire names, for example, to connect a wire to bit 0 of `data`, you would label the wire `data<0>`; the CAD tools would then know that the wire is connected to bit 0 of data.

Top Tip

File Naming Convention in Cadence

Cadence is very particular about the names you give to your designs. Cadence only accepts file names with alphabetic and numeric characters (a-z, A-Z, 0-9), so **do not include spaces, dashes ('-') or any 'special' characters such as *,$,/ etc**. However, you **can** use underscore ('_'). Any special characters in filenames may seem to work at this stage but then may cause apparently inexplicable difficulties later when scripts are run to process designs. You **must** start a filename with an alphabetic character (and **not** a numeric character), much like a variable name in software. In most exercises designs will be provided for, please make sure you use these designs and do not change the filename otherwise the submit mechanism may not work.

Exercise 2: The Seven Segment Decoder

Exercise Duration	2 lab sessions – weeks 1.4 and 1.5
Submission Deadline	End of your scheduled lab session in week 1.5. Submission is ex2.
Extended Deadline	Beginning of lab in week 1.7
Offline Marking	Yes
Demonstration Required	Yes – run labprint for ex2
Feedback	Feedback from offline marking will be emailed in Week 1.7.
Assessment	20% of the total marks available for the lab. Offline marking 12%, face-to-face demonstration 8%.

Aim

To design a seven segment decoder in Verilog and demonstrate your design working using the experimental boards available in the lab.

Preparation

This exercise introduces Verilog as a means to specify the function of a combinatorial circuit. It also takes a design through to the implementation on an FPGA using the experimental boards available in the laboratory. Please read the background material on Verilog before you attempt the lab as it offers tips and advice on how to write your Verilog hardware descriptions and test stimulus.

This exercise has some 'creative' design input but is largely focused on introducing more tools. Make sure you *understand* the processes involved rather than just following the instructions; you will need them again later.

Verilog – A Quick Introduction

Verilog is a Hardware Description Language (HDL). Its initial use in this laboratory was to provide a **simulation** environment to control and test schematic designs. However, we will primarily be using it to describe real hardware that can be **synthesized** into logic circuits for downloading onto the experimental boards in the laboratory.

Figure 2.1 shows some rough equivalencies of different levels of the design hierarchy, compared with their software equivalents. A HDL is roughly equivalent to a programming language such as Java; most industrial hardware designers will use a HDL[4] most of the time. However, just like software, it is important to know what goes on 'underneath' if your design is to be efficient.

Hardware	**Software**
Gates	Machine Instructions
Schematics	Assembly Language
HDL	High-Level Language
Simulator	Debugger

Figure 2.1: Equivalencies of different levels of the design hierarchy

Simple Simulation

Simulation is the normal method of testing a design before it is implemented – and you have already had a go at simulating your designs in the previous exercise. When a silicon chip has been made – at a considerable cost – it is not possible to debug it, so it is important that it is 'right first time'. A simulator provides a means to test a design before it is made. It also allows *observability* of all parts of the system so that a fault visible on the 'outside' can be traced back inside the design, something that is not possible in a finished chip.

```
initial                 // The code below is run once
  begin
  #100                  // This inserts a delay of 100ns

  a = 0;                // Set variable "a" to zero
  #100                  // After 100ns
  a = 99;               // set variable "a" to ninety-nine
  #100                  // After 100ns

  $stop                 // stop the simulation at this point
  end
```

Figure 2.2: Simple Verilog simulation code

Although the simulations you do will be very simple, in general, simulations get quite complicated and – because designs rarely start off bug-free – the same simulation will be repeated a number of times. To facilitate this it is normal to generate a stimulus file in

[4] There are only two widely used HDLs at present. The other one is called "VHDL".

which the various input actions are recorded. The input sequence can then be invoked and the actions of the circuit displayed.

The simplest form of stimulus file will look something like the code given in figure 2.2, this is similar to what you produced for exercise 1, and so should be familiar. What is shown is a very small subset of what the language can do – there will be some more examples later – but it serves to illustrate how input signals can be set and delay inserted. The delays are necessary for any input changes to propagate through the circuit, and for you to see them in the waveform viewer!

Each time the code shown in Figure 2.2 is invoked it will:

- start at time zero (time=0s) – this is the role of the `initial` statement
- wait 100 ns (the variable "a" will be undefined at this time) (`#100`)
- set "a" to zero (`a = 0;`)
- wait 100 ns for any changes to propagate through the circuit (`#100`)
- set "a" to 99 (decimal) (`a = 99;`)
- wait 100 ns for any changes to propagate through the circuit (`#100`)
- stop the simulation and return control to the user (`$stop`)

The "`begin`" and "`end`" keywords act in the same way as '{' and '}' in Java, bracketing their contents into a single statement that is invoked by the keyword "`initial`".

There is an assumption that the variable "`a`" is declared elsewhere; in this case it is expected to be an input to the circuit being tested – it will probably be a bus in the design (as a single bit wire would only values of 0 and 1 to be represented). A method of adding comments to the file should also be apparent in the example!

For combinatorial circuits this form of stimulus file is usually adequate. Any number of inputs can be set and altered so any input combination can be investigated.

The only other thing it may be convenient to know at this stage is how to specify numbers in different bases. The default base is decimal. Bases can be specified as follows:

```
a = 'b0110_0011          // Binary using 'b
a = 'h63                 // Hexadecimal using 'h
a = 'd99                 // Decimal using 'd
```

Any number in front of the quote symbol is always in decimal and specifies the number of bits in the number, for example

```
a = 8'b0110_0011         // 8-bit binary value
```

will define an 8-bit binary value. An underscore '_' can be used to aid legibility for binary values.

It can be used with any base. It is not usually necessary but it is often considered good practice to match the size of the value to the size of the variable (bus) it is assigned to. Unlike Java, where integers will always be 32- or 64-bits, there is a wide selection of variable sizes in hardware.

Simulating a clocked circuit

In a clocked circuit we invariably have a clock input that we need to provide the stimulus for. Remember, a clock runs continuously at a constant rate (frequency). A possible stimulus file for generating a clock signal is shown in Figure 2.3. Where the clock signal input to the design under test (DUT) is called '`Clk`'. There is also an additional input to the DUT called '`a`' which is a bus.

```
// The following is for testing a design that has an input
// clock signal called 'Clk', as well as another input called
// 'a'

initial                 // The code below is run once
  begin
  #100                  // This inserts a delay of 100ns

  Clk = 0;              // Start with clock low
  a = 0;                // Se variable "a" to zero
  #50                   // Wait for half a clock period
  Clk = 1;              // Take clock high
  #50                   // Wait for half a clock period

  Clk = 0;              // Return clock low
  a = 99;               // Set variable "a" to ninety-nine
  #50                   // etc.
  Clk = 1;
  #50

  $stop;                // Stop the simulation at this point
  end
```

Figure 2.3: Verilog stimulus for a clocked design

Whilst the code shown in Figure 2.3 will insert the required delays to implement a clock signal that is varying at the correct frequency, this approach may get tedious (and excessive in code) as the extent of the simulation grows. Instead, a different, more code efficient, construct may be used to define a clock signal, as shown in Figure 2.4. This example exploits the capacity for *parallel programming* in Verilog. There are three code blocks: two `initial` blocks and an `always` block – remember these run at the same time starting at time zero. The first `initial` block contains a single statement and simply initialises the clock to zero (otherwise it would be undefined) at time 0s. The `always` statement runs repeatedly, it waits for 50 ns then inverts the clock ('~' is a binary 'not' operator; this assumes the clock is a 1-bit signal), thus completing a clock cycle over two iterations of the loop. These two blocks form the clock, and will run for the full duration of the simulation.

In the last `initial` statement any other input signals are changed and contains a number of statements. Here, it changes the value of the input '`reset`' at 100 ns intervals. The simulation then runs for 2000 ns until it stops when the simulator encounters the `$stop` statement The effect of this file is shown diagrammatically in Figure 2.5.

The advantage of this approach for generating the clock signal should be increasingly apparent as simulation files lengthen; it saves you a lot of typing!

```
// The following is for testing a design that has an input
// clock signal called 'Clk', as well as another input called
// 'reset'

// This is an example where we have two initial blocks and an
// always block that run concurrently starting at time = 0

// the following initial block initialises the clock

initial Clk= 0;        // initialise Clk to 0 at time 0

// the following always block creates the clock signal

always                 // always do the following
  begin
  #50                  // wait half a clock period
  Clk = ~Clk;          // invert the clock
  end

// the following initial block sets the other inputs to the
// design

initial                // The code below is run once
  begin
  #100                 // This inserts a delay of 100ns

  reset = 0;           // Set variable "reset" to zero
  #100
  reset = 1;           // Set variable "reset" to one
  #100
  reset = 0;           // Set variable "reset" to zero
  #2000                // run the simulation for a period of time

  $stop;               // Stop the simulation at this point
  end
```

Figure 2.4: More efficient Verilog stimulus code for defining a clock

WARNING: The gate and flip-flop models used in the laboratory are intended for the Xilinx FPGA. This device has a reset circuit that initialises all its flip-flops to zero when it is first configured. The simulation model reflects this by inserting a 100 ns delay at the start of simulation time. It is therefore not possible to alter any registers until after this 100 ns has elapsed. As a result, it is recommended that you begin your stimulus file with "#100" (as in the preceding examples) so that this time has expired before you try to exercise any sequential circuit. The template you are given includes this delay already.

Although the Xilinx FPGA has a reset process, it is still important for all synchronous designs to have a reset signal, the job of which is to place the FSM in a predefined state. This should always be tested as part of your testing strategy at the start of the test stimulus.

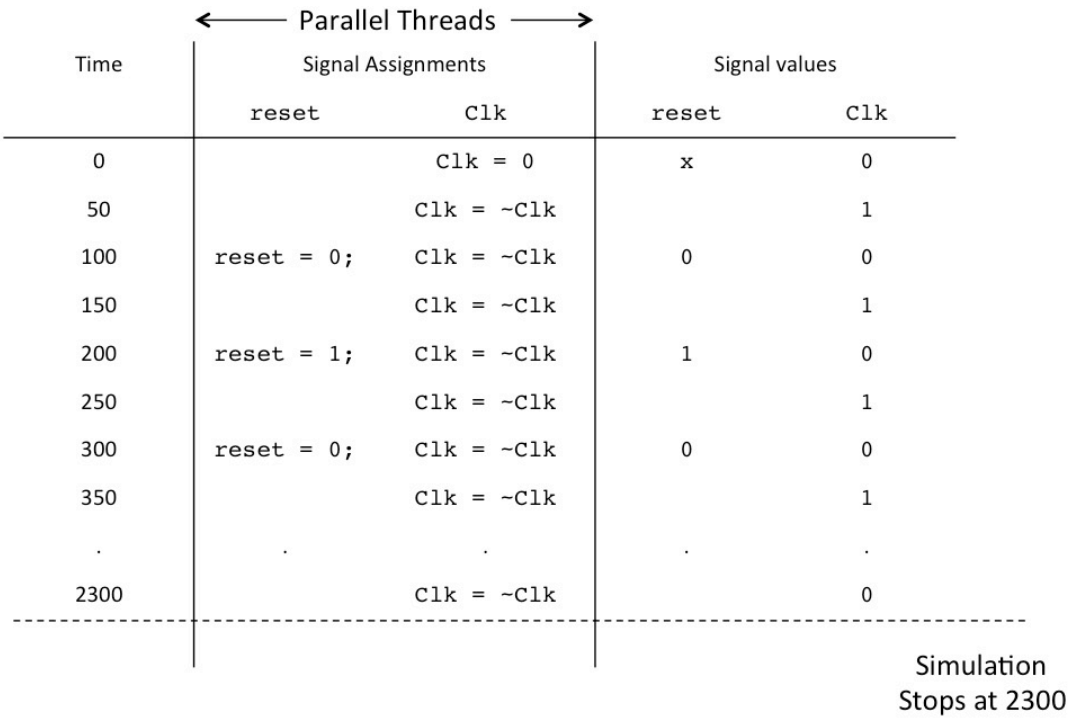


Figure 2.5: Effect of Verilog stimulus on variable assignment

Synthesizable Verilog

So far we have only looked at using Verilog for producing stimulus files for simulation. However, Verilog is a Hardware Description Language (HDL) and, as such, can be used to specify circuits that can then be synthesized into hardware. There are numerous ways in which this can be done – some 'cleaner' than others – and the syntax is not always as obvious as one would like. General points about Verilog are discussed in the lectures and therefore are not repeated here. This section is intended as a reminder and quick reference.

Modules

A Verilog module is similar to a symbol or a schematic: it has inputs, outputs and, possibly, may contain state. Only the defined I/O signals are visible from outside the module. It can be represented as a symbol and incorporated in to larger designs. The job of the module is to assign values to any output signals, this may be due to the current state of the input signals, as in a *combinatorial* circuit, or previous values of the input signals, as in a *sequential* circuit.

An example module syntax for a modulo-7 counter is given in Figure 2.6. In this example what is `count` initialised to? When synthesized in to hardware the register used to contain the value of `count` will be initialised to 0, so we don't need to worry. However, if you try to simulate this design, the value for `count` will be undefined and hence never changed, as it is never initialised to a known value. To solve this problem we could introduce a new input signal called reset that when taken high will initialise the value of count to zero. This will allow the count to initialised for simulation (by taking reset high in the test stimulus) and will allow the count to reset at any time. A modified example is shown in Figure 2.7. Reset is a global signal, like the clock, as such the reset action should be treated first before any other action is performed.

```
module fred(input                clock, enable,
            output reg [3:0]     count,
            output reg           zero);

always @ (posedge clock)
  if (enable)
    begin
      if (count == 6) count <= 0;
      else            count <= count + 1;
    end

always @ (*)
  if (count == 0)
    zero = 1;
  else
    zero = 0;

endmodule
```

Figure 2.6: Verilog modulo-7 counter module

The example shown in Figure 2.7 will allow the reset of count in both simulation and in hardware using the input signal `reset`. However, when will this occur? Will it be synchronous, i.e. synchronised with the clock edge, or will it be asynchronous, i.e. it can happen any time regardless of the value of clock? You have a think.

The use of a reset signal in synchronous designs is good practice, so we strongly suggest you always do so.

```
module updatedfred(input               clock, enable, reset,
                   output reg [3:0]    count,
                   output              zero);

always @ (posedge clock)
  if (reset)                  // reset condition, count set to 0
     count <= 0;
  else
    if (enable)               // otherwise if enabled increment
    begin
      if (count == 6) count <= 0;
      else            count <= count + 1;
    end

// the assignment of zero is straightforward so we can use an
// assign. In this case the variable (zero) needs to be declared
// as a wire

assign zero = (count == 0);

endmodule
```

Figure 2.7: Verilog modulo-7 counter module with count initialisation

Combinatorial Logic

Here are three general structures recommended as templates for anything you design:

Simple combinatorial logic

Use continuous assignment to represent simple logical functions, using the `assign` keyword as illustrated in Figure 2.8. The variable being `assigned`, `x`, must be defined as type '`wire`'.

```
wire        x;

assign      x = a & (b | c);
```

Figure 2.8: Continuous assignment

Complex combinatorial logic

If necessary, complex logic can use '`reg`' and blocking assignments '=', as illustrated in Figure 2.9.

If a variable is defined using the keyword '`reg`' it doesn't necessarily mean that it will be synthesized in hardware as a register, it could be synthesized as a simple combinatorial logic block (Verilog assignments can prove to be very confusing in this respect!).

A sequence of blocking assignments will be "executed" (assigned) in the order listed in the code.

```
reg   q;
reg   leap_year;

always @ (select, in0, in1, in2, in3)
   case (select)
      0:      q = in0;
      1:      q = in1;
      2:      q = in2;
      3:      q = in3;
   endcase

always @ (year)
   begin
   leap_year = 0;
   if (year[1:0] == 2'b00) leap_year = 1;
      // etc
   end
```

Figure 2.9: Example of the use of reg and blocking assignments

Sequential logic

State storage should generally use D-type (edge triggered) flip-flops, and use non-blocking assignment, as shown in Figure 2.10.

It is recommended that you use non-blocking assignments when the sensitivity list contains a clock source. In this example the `always` block is "executed" on a positive, rising edge on the clock signal, through the use of `posedge` on the signal `clk`.

A collection of non-blocking assignments within an always block will be executed at the **same time**, i.e. concurrently. Control statement, such as `if … else`, should be used to control behaviour.

```
reg [3:0]  count;

always @ (posedge clk)      // Modulo 16 up/down counter
  begin
  if (enable)
    if (up == 1)       count <= count + 1;
    else               count <= count – 1;
  end
```

Figure 2.10: Use of non-blocking assignments in Verilog

Structural Verilog

This is more detail than needed for this course, but Verilog modules can be nested within another Verilog module by creating an instance of the module being instantiated, as illustrated in Figure 2.11. Here, we instantiate an instance (called "a_new_fred") of the module "fred" to be used in the module it is created in.

```
wire       my_clock, my_en, zero_detect;
wire [3:0] count_out;

fred     a_new_fred   (.clock(my_clock),    // Instantiate a
                       .enable(my_en), //'fred'
                       .count(count_out),
                       .zero(zero_detect));
```

Figure 2.11: Nested Verilog modules

Top Tip

Verilog logic states

Note that each bit in a Verilog simulation can adopt one of *four* different states:

0	Logic '0'
1	Logic '1'
x	Undefined – the signal is a logic level but which is not known
z	Floating – the signal has no output driving it – it could be ANY value.

Background

In the logical design of digital systems, a common requirement is that an input bit pattern must be translated into a set of signals for controlling a particular hardware device, i.e. its job is decode an input data pattern to produce an appropriate output pattern – the circuit is called a **decoder**. In general the information is being represented in two different ways: the input code may be more compact in terms of bits in order to reduce storage requirements and the output code more redundant in order to suit the characteristics of the hardware it is controlling/connected to.

Seven segment displays were one of the first 'digital' displays and have been around for many years. They are commonly used to display the characters, such as the digits 0 to 9, on items of electronic equipment such as calculators, digital watches, etc, as illustrated in Figure 2.12.

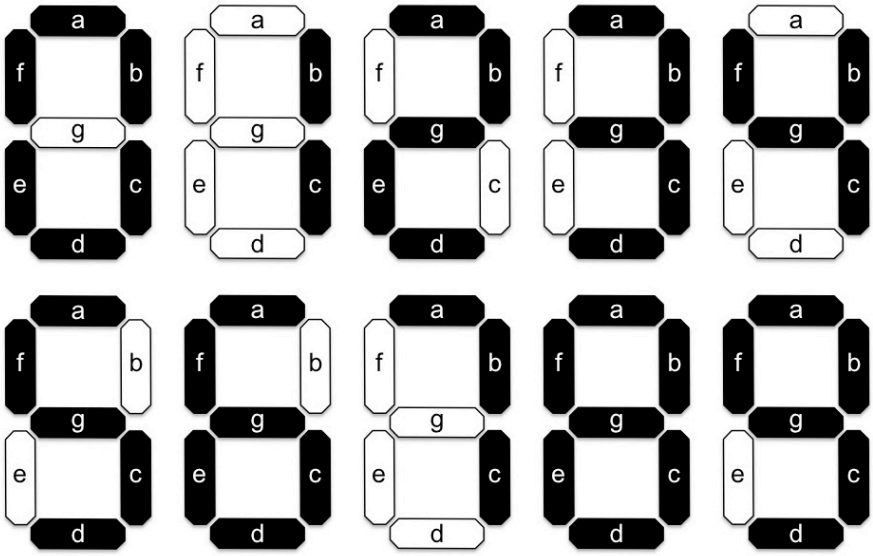


Figure 2.12: Seven segment patterns for the digits 0-9

To make an easily legible display these use a seven-bit code; there are seven segments and each represents one bit of information ('on' or 'off'). Because there are only ten possible digits, the number can be represented as a 4-bit code, since:

$$\log_2(10) = 3.32 \text{ or } 2^4 > 10$$

It is generally more convenient to use the minimum number of bits to represent any quantity because this reduces the wiring and circuitry required for both storing and operating on the digits. Thus, inside a machine a single digit (0-9) is often represented using a 4-bit code. However, to make an easily readable display these 4 bits must be converted into a 7-bit code for the display. This is the function of a seven segment decoder, it translates a 4-bit binary number (binary coded decimal, or BCD) into a 7-bit decimal code that allows that particular decimal number to be displayed on the seven segments of the display.

As 7-bits allows us to display more characters, we can actually display 16 (hexadecimal) characters on the display, so more characters can be displayed as illustrated in Figure 2.13.

You are required to implement a module to convert a 4-bit BCD code, representing the number/character to be displayed, to a 7-bit value for controlling the segments on the seven segment display. Hence, the module input is a 4-bit bus and the output is a 7-bit bus.

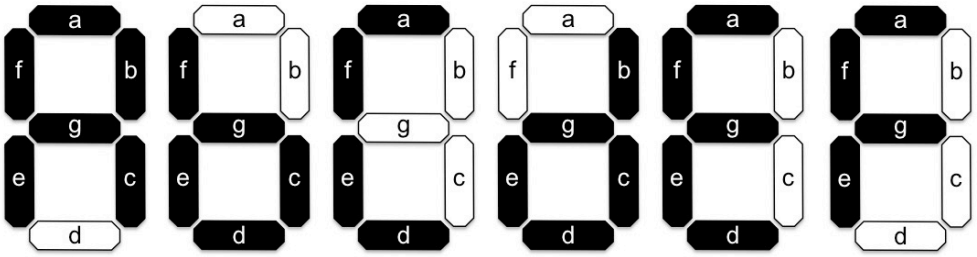


Figure 2.13: Seven segment patterns for the characters A..F

The seven segments of the display are labelled a – g, which correspond to bits in the 7-bit code, as shown in Figure 2.14.

Bit number	Segment
0	a
1	b
2	c
3	d
4	e
5	f
6	g

Figure 2.14: Mapping of segments to digits in the code

Starting a Verilog source file

Previously we have generated designs as schematic and symbol files. Verilog uses a third type, which, like a standard programme in any language, is a plain-text file.

To open a Verilog file from the Cadence Library Manager right click on the functional view of the file you want to design and select "Open". Please note, we have provided all the Verilog files you need for this exercise. This should open a text editor with the Verilog code displayed. When you close the editor it will perform a syntax check and report any syntactic errors you may have made. If there isn't already a symbol for the module it will offer to make you one automatically; this gets all the connection pins in place from your I/O list.

Modules specified in Verilog can be mixed freely with circuits designed as schematics. The most usual design approach is to construct most reasonably complex functions in Verilog and then place these as symbols in a schematic at a higher level of the hierarchy.

It is possible to include Verilog modules in other Verilog modules. However, the process of specifying the wiring is rather tedious and is not described here.

For the purposes of these laboratories the design flow requires a schematic at the highest level and offers the option of constructing basic modules either in Verilog or using gate-level schematics. So you will be designing an overall schematic for the system you will be designing, but this may include symbols containing Verilog description or gate-level schematics of components.

As we have already seen a module can be used to represent a particular "function" in

Verilog. Within the module we assign values to output signals. In the case of a combinatorial circuit the output will change whenever any of the input signals change. There are a number of ways to define the behaviour we require for our decoder in Verilog but the most appropriate here is a '`case`' statement; an example for you to work from is shown in Figure 2.15.

```
// Comments as to function of module, author, date, etc.

module sevensegmentdecoder(input       [3:0] bcd,
                           output reg [7:0] segments);

// Put your own comments in here, explain what it does

always @ (*)          // "Sensitivity list" - combinatorial
  case (bcd)
    0:     segments = 8'b0011_1111;
    1:     segments = 8'b0000_0110;

// …             You work out the rest!

  endcase

endmodule
```

Figure 2.15: Example Verilog code

Points to note:

- The decoder has an 8-bit output, yet you only need to create a 7-bit code for the display. The most significant bit (bit 7) is used to control the decimal point on the display. You can set this to 0, as is shown in the example in Figure 2.15.
- The module has a name (i.e. sevensegmentdecoder) following the keyword `module`, this is then followed by a list of input and output variables. Note: the name of the module MUST match the call name.
- Bus widths are specified using '[' and ']' characters (unlike '<' and '>' in a schematic).
- The 'variable types' are unusual – some would say eccentric! "`reg`" here does *not* mean that a register is used. For now, just copy the syntax here.
- Rather than '`initial`' – which evaluates once – this module uses '`always`' which evaluates every time there is a change in the "sensitivity list".
- In this case, every time "`bcd`", the input, changes the block below will be recomputed to give a new decoded output value for "`segments`". In practice this means that a combinatorial logic block will be produced.
- The '`case`' statement selects a following clause that matches the input.
- Here only the first two case clauses are shown; I wonder why? – because you will be doing the work to populate the rest!
- The number assigned to '`segments`' is an **8**-bit value, specified in **b**inary.
- '`_`' characters are ignored by the compiler but make the number easier to read.

- '`default`' is used to catch any cases that were not specified explicitly. If using BCD input then the 4-bit digit could have been wired (perhaps wrongly?) to an illegal value. This ensures that these are kept 'well behaved'.
- In this block only a single statement (the `case`) is specified and each case contains only one statement. If more statements are required '`begin`' and '`end`' can be used to group them together (like '{' and '}' in Java).

For this exercise you must produce a Verilog module for a seven segment decoder that translates a 4-bit binary-coded decimal (BCD) value into a code that allows the digits 0-F to be displayed on a human-readable display. A cell "sevensegmentdecoder" with view "functional" has been provided for you. Open this file.

a) Complete the functional description of the module provided to implement the seven segment decoder. DO NOT change the input/output signals defined. Once completed, save and close the Verilog editor and the Verilog parser will check the syntax of the Verilog code you have written. If there is a problem it will indicate the errors in your design, otherwise a symbol for your design should be created with the appropriate input and output pins.

b) Simulate your design to confirm it is working correctly. The easiest way to invoke the simulator is probably to start from the *Virtuoso* window: 'Tools ↲ Verilog Integration ↲ NC-Verilog…'. This will open up a form, press the 'Browse' button to select the design and view (functional) you wish to simulate. Once you have selected the design for simulation click 'Ok' and the NC-Verilog window should open.

c) Simulate your modified design to ensure it works as expected. You will need to create a test stimulus to do this. What are you testing? You are checking that the decoder works correctly, so you should be setting the input in the test file and checking that you observe the correct output when simulating.

d) To interface the circuit to the 'outside world' a cell named "sevensegmentdecoder_test" with view "schematic" has been provided for you. If you open this file you will notice that a symbol for the cell 'Board' from the library called "ENGLAB_12111" has been instantiated for you. This contains the interface definitions that allow you to connect to the various components available on the experimental board. Figure 2.16 illustrates the symbol for "Board" illustrating the input and output signals. This is the top level of the hierarchy so there are no input/output pins from this design, and hence no symbol. 'Board' is a hierarchical structure containing some other cells. These may be viewed by using 'Descend ↲ Edit' if you are curious. For illustrative purposes, some are defined as schematics, others as Verilog modules.

e) Instantiate you seven segment decoder design and connect it to one of the seven segment displays on the 'Board' symbol. Choose four buttons on the keyboard – we suggest using Key_row4<3:0> – and wire these to your inputs (remember to use wire names to make inferred connections between wires and buses in your circuit). Pressing combinations of these keys should produce the appropriate value to be displayed on the seven segment display. Please note: for the design to be synthesisable any inputs to the symbol board must be connected to something – generally ground. See the top tip at the end of the exercise for advice.

When you check and save your design you may see some warnings in the *Virtuoso* dialog box, this is perfectly fine, as you may have left some outputs unconnected.

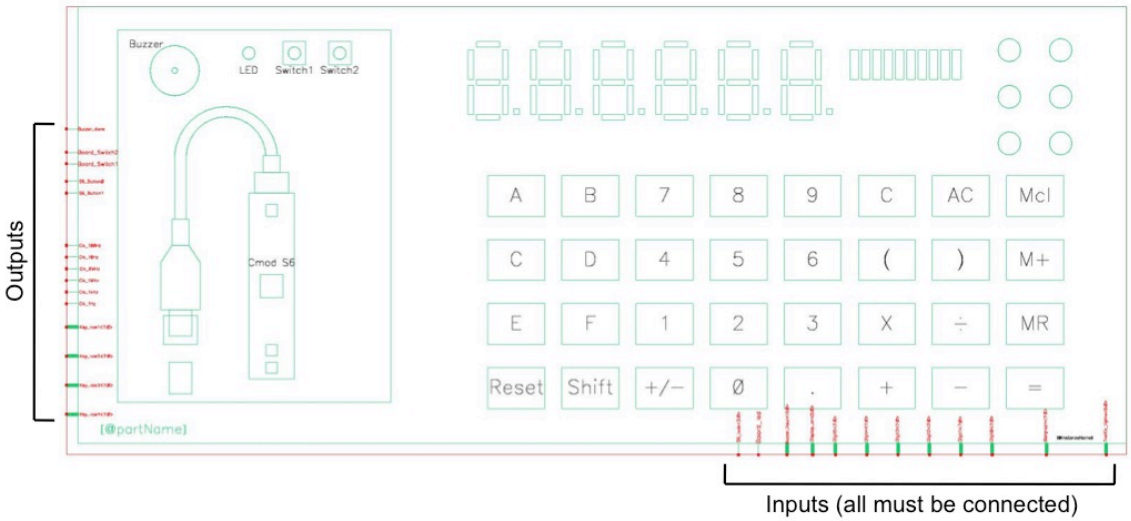


Figure 2.16: Symbol for the "Board"

Inputs and Outputs on Board

"Board" is at the top level of the design hierarchy. Any inputs/outputs connect to the real world via the physical board in the laboratory.

The inputs to the symbol 'Board' are (located at the bottom):

S6_leds<3:0>	: Corresponds to the four LEDs on the S6 FPGA board
Board_led	: Corresponds to the single red LED on the board
Buzzer_input	: Input for the buzzer on the board (see later description)
Display_en<5:0>	: Enables for each seven segment display. An enable must be set to '1' if any segments are to light. The index corresponds to the position (as below).
Digit5<7:0>	: Leftmost seven segment display. Each bit corresponds to one segment with segment 'a' as <0> etc. <7> is the decimal point.
Digit?<7:0>	: Other digits are similar to digit5.
Bargraph<7:0>	: Bargraph LEDs. <0> is at the left-hand side.
Traffic_lights<5:0>	: 'Traffic-light' LEDs. <0> is the right-hand green, <1> is the right-hand amber, and so on.

The outputs from the symbol 'Board' are (located to the left hand side):

Buzzer_done	: Output signal from the buzzer indicating that it has finished holding a note (see later).
Board_Switch1	: The state of switch 1 on the board (SW1)
Board_Switch2	: The state of switch 2 on the board (SW2)
Clk_8MHz	: 8MHz free-running clock
Clk_1MHz	: 1MHz free-running clock
Clk_1kHz	: 1kHz free-running clock
Clk_100Hz	: 100Hz free-running clock
Clk_10Hz	: 10Hz free-running clock

Clk_1Hz	: 1Hz free-running clock
Key_row1<7:0>	: The state of the top row of keys. Numbering is from right, so Key_row1<0> is the 'Mcl' key.
Key_row2<7:0>	: The state of the second row of keys, numbered as above.
Key_row3<7:0>	: The state of the third row of keys, numbered as above.
Key_row4<7:0>	: The state of the bottom row of keys, numbered as above.

Design compilation & download to the board

To test the implementation of your design you will need synthesise it into a physical circuit layout and then download it onto the Xilinx gate array on the lab experimental board. To synthesise the design ensure that you have checked and saved the design in Cadence. Then, if you have not already done so, invoke the simulator.

Initialize the design and generate the netlist as before (the topmost two buttons to the left of the Simulation Window), checking that no errors are reported at this stage. Then use 'Commands ↲ Xilinx Synthesis' to invoke the compiler. The synthesis process will take a few minutes – watch the window that opens to see that there are no errors and the compilation is "Done". If everything is okay this should have generated the bitfile that is used programme the FPGA.

The real test is to see the design running. You will need a lab experimental board that should be connected to your computer via a USB lead and powered on (make sure it is!).

Downloading can be invoked from the Cadence simulator with 'Commands ↲ FPGA Load'. There is no need to recompile each time you load unless the design has changed. The programme will take about 15 seconds to download and the design will run automatically.

ex2: Seven Segment Decoder

Once you have completed your design, simulated it, and downloaded it to the experimental board you can submit it for marking using the 12111submit script. Follow the instructions and submit the design as 'ex2' and enter 'submit'.

Print out a labprint sheet and answer the questions before you demonstrate the exercise in your next scheduled lab.

Please note: if the design fails to submit then this is probably because you have not simulated your seven segment decoder module.

You have now completed Exercise 2

Top Tip

Hierarchy & Schematics in Cadence

You will notice that the inputs and outputs of 'Board' are labelled in the symbol; this is so you know which input/output you are connecting to. However, these names exist within a different level of hierarchy from the design the Board symbol in instantiated in. Consequently, you can not use these names as wire labels in the your design. You will need to add a short wire to the input/output and label it if you want to use this approach to simplifying your design.

Top Tip

Unused connections

There will be some unused inputs on 'Board' in this exercise. In a real circuit it is important that all input signals are connected, so the compiler will reject the design unless these are wired to some signal. It is suggested that the majority are wired to 'GND', which is a logic '0', the symbol for which can be found in the spartan6 library.

As these pins are buses it is necessary to provide the appropriate number of drivers. For example, 8 GND signals for an 8-bit bus. The simplest way to do this is to instantiate a GND symbol and then use Edit ↲ Properties ↲ Objects… so that it represents eight parallel outputs. For example, if the instance has an Instance Name "I5", then editing it to "I5<7:0>" would produce the required 8 GND symbols configured as a bus. The outputs can then be connected to input bus pins as appropriate.

Don't forget to *enable* the appropriate display by wiring its Display_en signal 'high'. The symbol 'VCC' in the spartan6 library provides a logic '1'.

There is no need to connect unused outputs: these can be left 'floating'.

Top Tip

Simulation at the top level of the hierarchy

When creating a schematic with the symbol "Board" in it, we are at the top of the design hierarchy. There are no input signals we control, or output signals that we can observe. Hence, at this level of the design hierarchy we cannot simulate the performance of our design. However, if ALL the components used in the schematic have been tested effectively then there shouldn't be any problems (apart from mistakes made when connecting up components).

Top Tip

Instantiating multiple instances of components

Every item on a schematic will have a unique name. For example, if you instantiate (add an instance of) an AND2 gate it will have a *Cell Name* name ("AND2") but it will also have a *Instance Name* (e.g. I76) which will be assigned by the CAD tool.

Wires, too, are automatically named, with connected networks being given the same name. If you 'label' a net then this supersedes the automatically generated name, usually making the schematic easier to read and simulation results *much* easier to interpret. However it is possible – and sometimes desirable – to give your instances their own names too. That is not to suggest that every AND gate should be called Alice, Bob, Charlie, etc. but larger labelling blocks "ALU", "decoder", "accumulator" and so forth is often useful for readability.

The reason this is mentioned here is that it is possible to make arrays of components in the same way that a bus is an array of wires (Figure 2.17), for example, creating a register from flip-flops. You don't want to hand wire 32 instances of a D-type flip-flop to implement a 32-bit register.

In Cadence it is possible to replicate circuits to an arbitrary width when they are instantiated; the same basic symbol can be used in many different ways.

Note that the bus widths must match those expected by the instantiation or the tool will indicate an error. The exception is that if a single wire (such as 'sel' in figure 1.6) is connected it is replicated across all the copies of the symbol.

To achieve this in Cadence, first instantiate a single symbol as normal. Then select the symbol and Edit ↲ Properties ↲ Objects… to open a dialogue box. The appropriate width can then be appended to the Instance Name; for example if the name was "I29" this can be altered to "I29<31:0>" if 32 copies are desired, to form, in this case, a 32-bit 2:1 multiplexer that selects between two 32-bit buses.

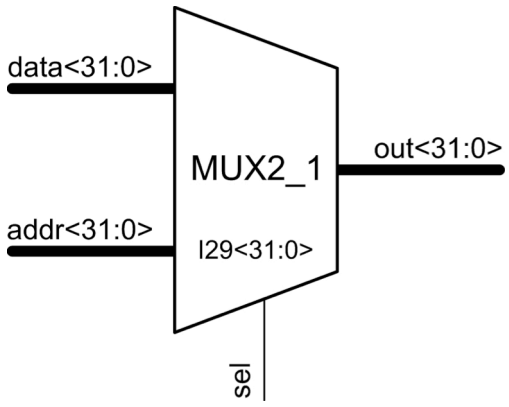


Figure 2.17: How to make a 32-bit 2:1 multiplexer

Exercise 3: Finite State Machines & Counters

Exercise Duration	2 lab sessions – weeks 1.7 and 1.8
Submission Deadline	End of your scheduled lab session in week 1.8. There are two parts for this exercise, so the submissions are ex3a and ex3b.
Extended Deadline	Beginning of lab in week 1.9
Offline Marking	Yes
Demonstration Required	Yes – run labprint for ex3 (Note: there is only one labprint sheet for the two parts of this exercise – ex3a or ex3b labrpint will print the same sheet).
Feedback	Feedback from offline marking will be emailed in Week 1.9.
Assessment	35% of the total marks available for the lab. Offline marking 20% (10% per exercise), face-to-face demonstration 15%.

Aim

To investigate time dependent, i.e. synchronous, circuits. This exercise introduces **finite state machines** (FSMs) in Verilog that are then used to build **counters**. Finite state machines are a type of circuit widely used in the timing and control of computers.

There is a little more of a 'design' element to this exercise than some of the preceding ones – it is not *just* about learning tools!

Preparation

Read the exercise description thoroughly. Make sure that you understand the functions described and especially the differences from the previous exercise.

Finite State Machines

The design of a finite state machine (FSM) has been discussed in detail in lectures. The general structure of a FSM is shown in Figure 3.1.

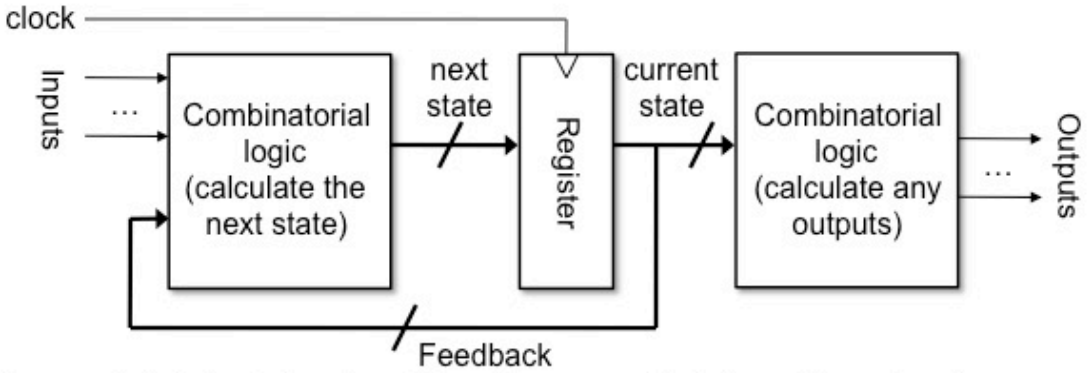


Figure 3.1: Verilog code for a simple finite state machine

The implementation of the FSM in Figure 3.1 has three functional blocks:

- a block of combinatorial logic to determines the **next state** from the **current state** and the state of any **input signals**,
- a register to hold the **current state**, at a rising edge of the clock signal **current state** is assigned the value of **next state**,
- a block of combinatorial logic to determine the state of any **output signals** from just the **current state**.

We could implement the FSM in Verilog as a single `module` however, to aid understanding we will be partitioning the design into three blocks to reflect the design shown in Figure 3.1. Note: this is just one solution, there are many ways you could implement a FSM, but by structuring it in this manner there is less chance of errors being introduced into the design.

Counters

A simple counter can be implemented as a FSM. Figure 3.2 illustrates the state transition diagram for a simple 2-bit counter, where the output at any instant of time is the value of the current state. At each rising edge of the clock we move from one state to the next, as a result the count increments. When the state/count reaches the maximum value of 3 then it repeats again from 0.

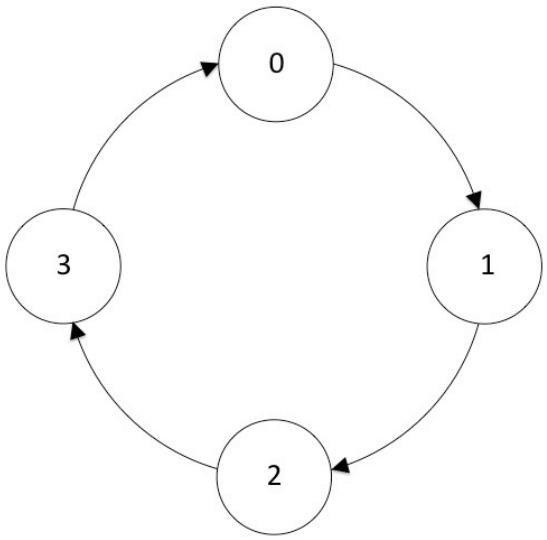


Figure 3.2: Simple 2-bit counter

Figure 3.3 illustrates a Verilog module implementation for this simple 2-bit counter. Here, `current_state` represents the current binary count, which is reflected at the output by the outputs signal `digits`. The

internal variables `current_state` and `next_state` have been used rather than 'exposing' the state values directly. Whilst for this example the value of `digits` is the same as `current_state`, this is a useful technique when the output is not identical to the internal state.

```
module counter   (input                clock,
                  output reg [1:0]     digits);

// internal variables
// we only have 4 states in this design, so 2-bits are enough
// to represent the state codes: 00, 01, 10, and 11

reg   [1:0] next_state;              // Internal variable
reg   [1:0] current_state;           // declarations


// next state logic – no inputs controlling the next
// state in this example, just current_state

always @ (*)                         // combinatorial
   case (current_state)              // next_state depends on
       2'b00: next_state = 2'b01;    // current_state
       2'b01: next_state = 2'b10;
       2'b10: next_state = 2'b11;
       2'b11: next_state = 2'b00;
       default: next_state = 'hx;    // Catch anything missed
   endcase                           // for testing – set to x!

// on a clock edge perform the state assignment
// if we had a reset signal, we would implement its
// functionality here

always @ (posedge clock)
      current_state <= next_state; // non-blocking!

// output logic assignment of digits
// we could use an assign here, i.e.
// assign digits = current_state;
// digits would need to defined as a wire in this case.

always @ (*)
      digits = current_state;      // When current_state
                                   // changes, so does the
endmodule                          // output - digits
```

Figure 3.3: Verilog implementation of a simple FSM

There may be several control inputs to the state machine. These inputs, together with the current state, are used to determine the next state. In the example given in Figure 3.3 the counter is free running and the output will always change on a rising edge on the input signal `clock`.

Points to note:

- There may be other 'forbidden' states in the system. In Verilog it is normal to ensure these by using a '`default`' case in the control logic to catch all the cases that were not

written explicitly. This is useful for simulation purposes for trapping issues. In this case it is best to set the variable to don't care, 'X', so that if the default case is entered, it will be clearly be evident in the waveform trace.

- The sequential '`always`' is just sensitive to '`clock`' because other inputs *only* have influence at that instance time (rising edge of `clock`).
- In the sequential always block the assignment operator '<=' is used, rather than '='. The difference between these can be subtle; the rule "use '=' for combinatorial assignment and '<=' for clocked assignment" is highly recommended. Generally, use '<=' if the sensitivity list contains a clock.
- '`current_state`' will only be updated on a rising edge of `clock`, immediately on `current_state` changing value, the value of '`next_state`' will be evaluated.
- A FSM should always have a reset signal that is used to initialise the FSM to a known state (usually 0). It is usual to introduce this behaviour in the sequential `always` block as this will translate very nicely into hardware (a register consisting of D-type flip-flops with a reset input).

Simulating a clocked circuit

A clocked circuit, such as a counter, may be simulated by manipulating the clock explicitly in the stimulus file. However, it is very tedious to set the clock high, step, set the clock low and step for every cycle. You may like to write a clock generator to make your simulation easier – see the earlier Verilog background information for advice on how to generate a clock.

Synchronous and Asynchronous Events

A synchronous system will cause events to occur in relation to a global clock signal, whereas in an asynchronous system, events will occur independently of a global clock signal. Figure 3.4 gives two examples of `always` blocks, one where an input signal '`control`' acts synchronously, and another where the input signal '`control`' acts asynchronously.

```
always @ (posedge clock)
if(control == 1)
    someoutput <= 1;
  else
    someoutput <= 0;


always @ (posedge clock, posedge control)
if(control == 1)
    someoutput <= 1;
  else
    someoutput <= 0;
```

Figure 3.4: Synchronous and asynchronous blocks

In the first example the output `someoutput` is assigned the value 1 if `control` = 1 when there is a rising edge on the input `clock` – this is synchronous in nature. In the second example, because of the inclusion of a dependency on `control` in the sensitivity list of the `always` block the output `some` is assigned a value whenever the input, `someoutput`, changes, regardless of the clock – it is acting asynchronously.

Top Tip

Global Signals – Clock and Reset

All sequential systems should have a clock signal (for obvious reasons). In addition, all sequential systems should have a reset signal that when high should initialise the system to a predetermined state. Both these signals are global signals, in that the are used throughout a complete system.

Where do these signals act in our design of an FSM? Well the clock is clearly only used in the current state assignment block (the register). What about reset? To answer this, we need to think about what reset does – it places the system in a known state. Whenever the current state changes then this results in both the outputs and the next state changing, as the current state forms an input to the combinatorial logic blocks that determine these signals. Consequently, a suitable place to act on the reset signal is also in the current state assignment block. In this case, reset should result in the current state of the system returning to an initial state. Applying the reset action in this block makes sense since a register is made of flip-flops and flip-flops often come with a reset input, which when high resets the output of the flip-flop to 0.

Top Tip

Signals and multiple always blocks – Don't do it!

The design of a Verilog module may contain many always blocks. Within these always blocks you will be assigning values to internal variables, or setting the output signals from the module. Consider the following example where a variable `asignal` is set in two always blocks:

```
always @ (posedge clk)
  asignal <= 0;

always @ (posedge clk)
  asignal <= 1;
```

What will happen here? Both assignments will be made at the same time and you will not be able to determine what the final value of `asignal` will be. Consequently, it is important that you only assign a variable a value in a single `always` block or `assign` statement, and never within multiple `always` blocks or `assigns`, as the behaviour will be undetermined.

The first task for this exercise is to design a modulo-10 counter as a Verilog module. The operation of the counter is defined by the following specification:

- The counter should count from "0" to "9" (and then restarts) at a rate determined by the external `clock`.
- The counter should only count when an input signal '`enable`' is high.
- If `enable` is low then the counter should hold the current value for the count.
- The counter should reset to 0 whenever the input '`reset`' goes high irrespective of the state of the `clock` (i.e. it is asynchronous) and the state of `enable`.
- An output '`digits`' should reflect the current count of the counter.
- An output '`carry`' should be high when the value of the count is 9, and should be 0 otherwise.

You should design your counter as a FSM adopting the general three functional blocks structure as shown in the schematic of Figure 3.1 and code example of Figure 3.3. However, do not simply copy the example, as the functionality is different to what is required for this exercise. For neatness and to promote reusability you should design your counter as a separate module with its own symbol.

The following file has been provided for you:

- cell "counter09", view "functional" – module description of the mod-10 counter

The inputs and outputs have been defined for you - do not change the names provided. As per the standard design process you MUST simulate your design to check that it is working correctly. As part of your test you should be checking the FSM works correctly for all the input signals, following on from the specification given above.

The counter can be implemented directly on the experimental board in the laboratory. The following file has been provided for you:

- cell "counter09_test", view "schematic" – test schematic for the mod-10 counter.

As for exercise 2 you can use the board symbol to interface your counter design to the hardware on the experimental board. Board has a number of clock outputs that may be useful in providing a clock to your counter. Produce a test schematic for your counter design by instantiating your counter09 module and connecting the inputs and outputs appropriately. In order to display your count you will need to interface to one of the seven segment decoders on the board. What design could you use to help you with this?

Test your design working on the experimental board.

ex3a: Modulo-10 counter

Once you have completed your design, simulated it, and downloaded it to the experimental board you can submit it for marking using the 12111submit script. Follow the instructions and submit the design as 'ex3a' and enter 'submit'.

Please note: if the design fails to submit (files not found message) then this is probably because you have not simulated your counter09 module.

Traffic Lights

Traffic lights are used universally to control the flow of traffic at road junctions. Figure 3.5 illustrates the layout of a simple traffic junction, and gives the traffic light sequence to control the flow of the traffic. We can design a FSM to control a traffic light system where each state corresponds to one of the eight possible light configurations in the sequence (labelled 0 to 7). Note that states 1 and 5 are the same – both sets of lights are on red – however for simplicity these will be treated as different states.

In a real implementation we may want different delays between state transitions depending upon the state. For example, the delay from state 0 to state 1 will differ from state 2 to 3, as we would expect one line of traffic to be on green longer than when both sets of traffic lights are on red. Figure 3.6 lists a set of possible delays for an implementation of this set of traffic lights.

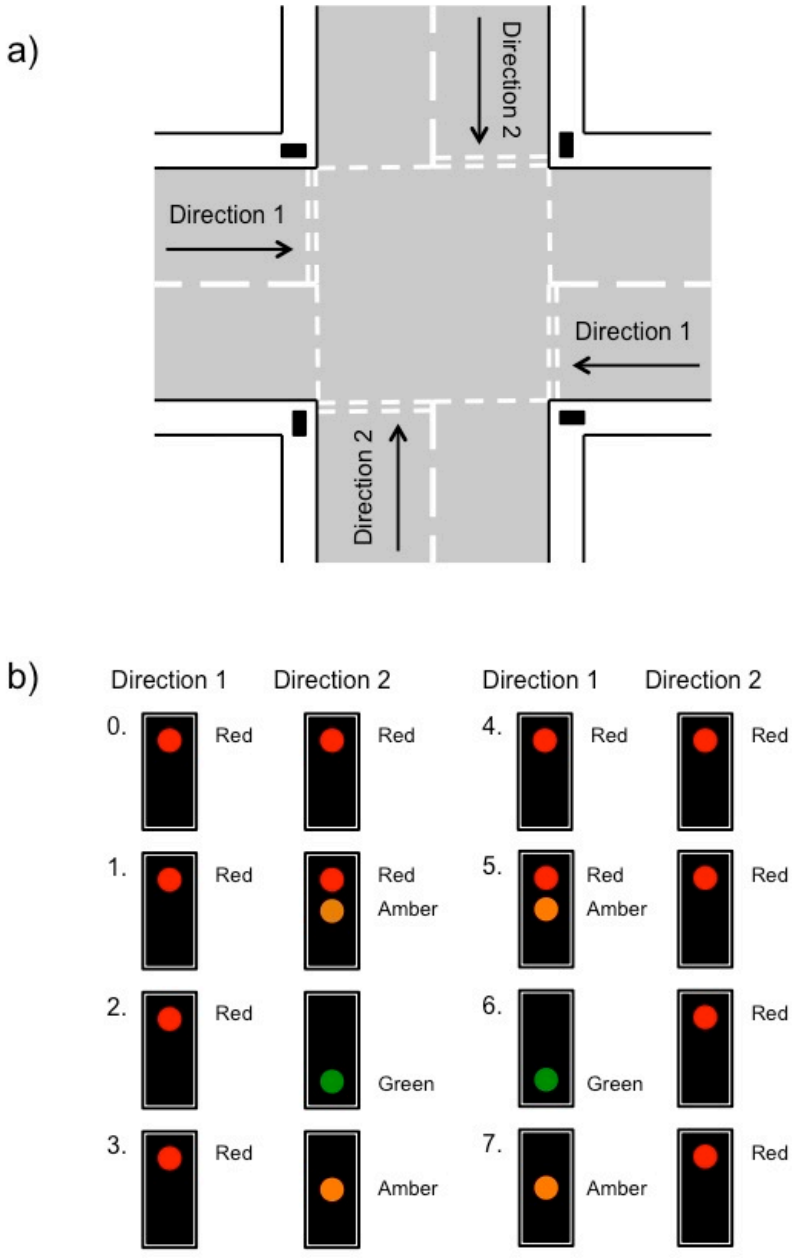


Figure 3.5: a) Traffic light junction, and b) corresponding traffic light sequence

Light Sequence	Delay (s)
R_R	1
R_RA	1
R_G	8
R_A	2
R_R	1
RA_R	1
G_R	8
A_R	2

Figure 3.6: Traffic light sequence and delay for each state

For this part of the exercise you are required to design and demonstrate a working traffic light system using the lab experimental boards according to the specification given below.

- The sequence of the traffic lights should follow that given in Figure 3.5 (and repeat).
- The traffic lights should reset to an initial starting state on the (asynchronous) application of a `reset` signal.
- The delay that the lights stay in each light sequence should be controllable and should be set to the values given in Figure 3.6.
- The delay for each state needs be determined using an external counter.

A module is required to encapsulate the required behaviour, and the following file has been provided for you:

- cell "trafficlight", view "functional" – module for the traffic light FSM

The Verilog module "trafficlight" has the following inputs and outputs:

Inputs

- `clock` – should be connected to a suitable clock source
- `reset` – asynchronous global reset to set the FSM to the initial state 0
- `count` – 4-bit value provided by the external counter, used to determine what period of time has passed since the counter was enabled
- `reset_count` – reset the external counter to 0
- `enable_count` – enable the external counter to start counting

Output

- `lightseq` – a 6-bit pattern for the current state of the traffic lights.

The state transition diagram for the traffic light controller is given in Figure 3.7. The design of the traffic light FSM in "trafficlight" has been partially completed for you, you will need to complete the design based on the requirement given above. The FSM for the traffic light controller has been structure in the three functional block approach you are familiar with.

Once again you should be simulating your design to ensure that it works correctly. To do this you will need to artificially set the value of the external counter (the input `count` to the FSM) to check that the FSM moves between states when the correct value of `count` is observed.

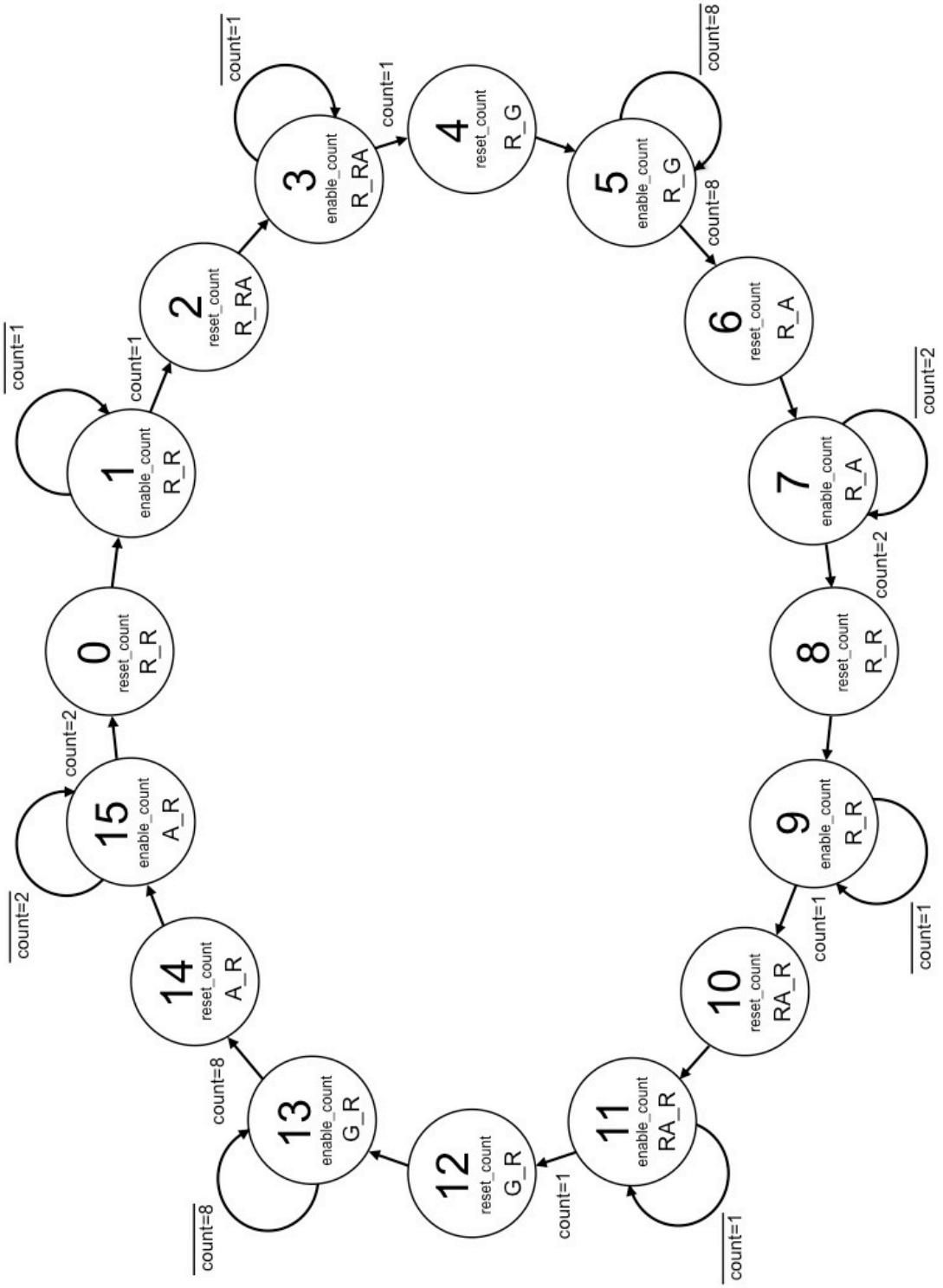


Figure 37: Traffic Light Controller State Transition Diagram

Once you have completed and simulated your design of the traffic light FSM controller, you can implement your design using the experimental boards available in the laboratory.

A test file has been provided for you to implement the traffic light controller using Board:

- cell "trafficlight_test", view "schematic" – test schematic for testing the operation of your traffic light system using the lab experimental board.

As in the previous test schematics Board has been instantiated to give you access to the features of the experimental board, such as clocks and LEDs. Instatiate "trafficlight" as your controller and the mod-10 counter you developed previously ("counter09") to provide a counter to the controller. You should connect up your design to Board in order to demonstrate your design. Board has a 6-bit input Traffic_lights<5:0> for controlling the on/off state of the LEDs on the lab experimental board; the LEDs are connected as shown in the table in Figure 3.8.

Bit	LED
0	Right-hand Green LED
1	Right-hand Amber LED
2	Right-hand Red LED
3	Left-hand Green LED
4	Left-hand Amber LED
5	Left-hand Red LED

Figure 3.8: LED connections in "Board"

Top Tip

Choice of FSM clock frequency

In the traffic light design you have two clocked FSMs – the counter and the trafficlight FSM. Which clocks should you use? The choice of clock frequency is important and depends on the application.

The counter clearly has to run with a 1s clock, as it is used to measure a delay in seconds. What about the FSM?

In this example it is important that the FSM runs at a faster rate than any control signals that govern its behaviour. So, in this case the counter FSM. So choose something greater than 1Hz.

Generally, the FSM clock is fast, and as a result the FSM may actually spend a lot of its time in a state waiting for something to happen. This is perfectly fine.

ex3b: Traffic Lights

Once you have completed your design, simulated it, and downloaded it to the experimental board you can submit it for marking using the 12111submit script. Follow the instructions and submit the design as 'ex3b' and enter 'submit'.

Print out a labprint sheet and answer the questions before you demonstrate the exercise in your next scheduled lab. There is one labprint sheet for this exercise.

Please note: if the design fails to submit (files not found message) then this is probably because you have not simulated your trafficlight module.

You have now completed Exercise 3.

Some more useful bits of Verilog syntax

More 'case' syntax

In Verilog it is possible to group several cases together within a case statement; to do this simply separate the cases with commas.

For example:

```
case (state)
  0, 1, 4:      . . .
  2, 3:         . . .
  . . .
```

Note that the cases do not need to be ordered numerically (although it often improves legibility if they are).

Concatenating signals

A bus can be made up by listing its constituents, separated by commas, in braces.

For example:

```
assign    six_bits = {3'b000, my_signal, 2'b10};
```

Partitioning the FSM

In practice a machine of this complexity may be split into two (or more) interacting FSMs, one controlling the state of the lights and another providing an indication of when the light state can change, with (potentially) different counts for each phase. This can be done with separate '`always`' blocks within the same '`module`'.

You might consider how this could be applied to your exercise. However do not do this unless you have plenty of time and are confident in your designs to date.

Top Tip

Adding readability … and maintenance

Although numbering states {0, 1, 2, …} works fine, there are a couple of potential problems with being so explicit.

- It is not always easy to remember what (e.g.) 'state 3' represents
- If the code is changed the state numbering might change. It is hard to, for example, change all the references to 'state 3' to 'state 5' whilst not changing any other '3's in the code.

It is good practice to *name* or **enumerate** the state identifiers instead. In that way 'state Fred' is easy to read and unique, and the actual value representing 'Fred' can be chosen arbitrarily.

Similarly, output codes may be clearer if enumerated, so that names can then be used.

Unfortunately, Verilog does not provide particularly good syntax for this. Here is an example of how outputs may be defined for some traffic lights.

```
`define   R__R      6'b100_100     // '_' in a number is
`define   RA_R      6'b110_100     // ignored
`define   G__R      6'b001_100

. . .
if (state == 'ALL_STOP) lights = 'R__R;  // Example only
. . .
```

When the compiler encounters "`R__R" it looks up and substitutes from the appropriate definition ("100100" in this case). If the definition is changed, all the references will be changed automatically.

Note the use of the 'backtick' character ('`') indicating both the definition and use of the macro.

This technique is *Good Practice*, not just in Verilog but in any programming language, hardware or software. It takes a little more effort at first but pays off in the longer term.

'Traditionally' such definitions are given in upper-case (capital) letters. This has no specific meaning for the compiler but is a programmers' convention.

Exercise 4: MU0 – A Microprocessor System

Exercise Duration	2 lab sessions – weeks 1.9 and 1.10
Submission Deadline	End of your scheduled lab session in week 1.10. Submission is ex4.
Extended Deadline	Beginning of lab in week 1.11
Offline Marking	Yes
Demonstration Required	Yes – run labprint for ex4.
Feedback	Feedback from offline marking will be emailed in Week 1.11.
Assessment	20% of the total marks available for the lab. Offline marking 10%, face-to-face demonstration 10%.

Aims

To reinforce the development processes encountered earlier.

To experience some higher-level system design and to produce a larger, more complex system.

To demonstrate that microprocessors are just finite state machines.

Preparation

Read the exercise description thoroughly – some basic information about the MU0 is provided. However, you should read through the accompanying lecture notes covering the operation and design of the MU0 processor.

MU0 Overview

In this exercise you are asked to complete an implementation of the MU0 processor, which we have already started for you. At the end of the exercise you will have produced a fully working processor!

You will encounter MU0 in more detail in lectures. To save time, the majority of the system has been provided for you. You will need the knowledge – and some of the circuits – you have acquired from previous exercises in order to complete the design. The exercise may seem daunting, but is in fact straightforward with regards to what you are required to complete.

The system illustrates the processor model, with a datapath and control, and the 'three box' computer model (processor, memory, I/O). The exercise uses a mixture of the techniques encountered earlier (hierarchy, schematics, Verilog, simulation, compilation).

The following is a brief overview of the MU0 processor and its operation with respect to this exercise. More information can be found in the lecture notes, which you should refer to for further information.

The Processor Interface

The input and output signals to the processor are defined as the *interface*, these are the signals that connect the processor to the outside world. The microprocessor has a limited, well-defined interface with the following signals:

- **Clock**: input clock to the processor
- **Reset**: input indicating that the processor should adopt a predefined initial state
- **Address** bus: 12-bit output to memory
- **Data out** bus: 16-bit output to memory
- **Data in** bus: 16-bit input from memory
- **Read control**: output that is asserted when the memory should be read
- **Write control**: output that is asserted when the memory should be written
- **Halt**: output indicating the processor has halted (optional)

The programmers' model has two registers as illustrated in Figure 4.1:

- a 16-bit Accumulator, `Acc`
- a 12-bit Program Counter, `PC`

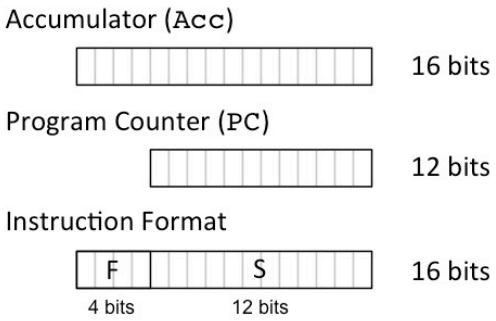


Figure 4.1: MU0 Registers & Instruction Format

The 16-bit instruction format consists of a 4-bit code representing the instruction, and a 12-bit value representing an address. A list of available instructions is given in Figure 4.2. The instruction act on values (or operands) held (stored) in memory.

F	Mnemonic	Function
0000	LDA S	Load accumulator from memory location S
0001	STA S	Store accumulator to memory location S
0010	ADD S	Add memory location S to accumulator
0011	SUB S	Subtract memory location S from accumulator
0100	JMP S	Jump to instruction address S
0101	JGE S	Jump to instruction address S if accumulator is +ive
0110	JNE S	Jump to instruction address S if accumulator is not 0
0111	STP	Halt execution
1000 -1111	-	Reserved

Figure 4.2: MU0 instructions

In the case of reset being asserted, the internal state (registers) of the processor are set to zero, resulting in the PC pointing at address 000 in memory. When reset goes low execution starts again by fetching the instruction stored at address 000.

Memory interface

To initiate memory access there are two memory control signals defined in the interface to MU0:

- **Wr** – write control - if the processor asserts the write control signal the data on the data output bus will be stored in the memory at the address indicated on the address bus.
- **Rd** – read control - if the processor asserts the read control signal data from the memory at the address indicated on the address bus will be returned on the data input bus before the next active clock edge.

MU0 Arithemtic Logic Unit (ALU)

The MU0 ALU performs a number of arithmetic functions depending on the value of the 2-bit control signal `M`. These operations are M<1:0>, the value of which determines the ALU operation that is performed on the two inputs to the ALU: X and Y, as shown in Figure 4.3.

M	Operation
00	Y
01	X+Y
10	X+1
11	X-Y

Figure 4.3: ALU operations as a function of M

The ALU performs subtraction through the addition of a negative number. Consequently, it contains additional logic to form the inversion of input Y when doing a subtraction. In order to form the two's complement inversion we need to add 1 to this value. We will leave it for you to figure out how to do this when you complete the ALU.

Control

A processor is a sequential system so it needs an FSM to control its behaviour. MU0 has a simple FSM as shown in the state transition diagram in Figure 4.4, which alternates between the Fetch and Decode/Execute states unless the processor is halted. The execution of an instruction is a two-stage process starting with the fetch phase, during with the processor **fetches** the instruction from the memory using the address held in the PC. This is then followed by the decode/execute phase where the fetched instruction is then **decoded** (by the control block) and the instruction **executed**.

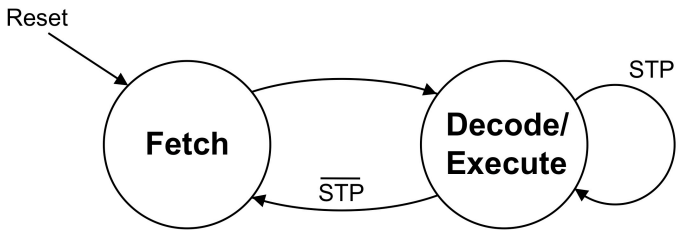


Figure 4.4: MU0 control state transition diagram

During the fetch phase the PC is incremented to point at the next instruction to be executed; unless a branch is taken, in which case the PC is overwritten with a new address during the decode/execute phase.

The fetched instruction is placed in a special register called the instruction register (IR) (which the programmer does not see) where it is decoded to determine what operation is required, and on what data, during the execute phase.

The control logic requires the following inputs:

- Clock : global external clock
- Reset : system reset
- F<3:0> : the upper 4 bits of the fetched instruction - decoded in the control to determine the required operation.
- N : the Negative flag (high when the accumulator content is negative), which is used when evaluating conditional branches.
- Z : the Zero flag, (high when the accumulator content is zero), which is used when evaluating conditional branches.

The control logic provides the following output signals to the processor datapath:

- PC_En : Programme Counter write enable – enables the PC to be overwritten on the next positive edge in Clock when high.
- Acc_En : Accumulator write enable – enables the Acc to be overwritten on the next positive edge in Clock when high.
- IR_En : Instruction Register write enable – enables the IR to be overwritten on the next positive edge in Clock when high.
- X_sel, Y_sel : select signals for the ALU input MUXs - determines where the inputs to the ALU come from.
- Addr_sel : Address output MUX select – determines where the address data comes from.
- M<1:0> : Determines the ALU function
- Halted : when high the processor is halted in the halted state

As the FSM only has two states, then the state code is only 1 bit, hence Fetch is 0, and Decode/Execute is 1.

Implementation of MU0

As illustrated in the lectures, the implementation of the MU0 may vary. The following example assumes the processor operation that is described above (and in the lectures) where each instruction takes two clock cycles: the first to fetch the instruction from memory and placing it in the IR; the second to decode and execute the instruction. Figure 4.5 illustrates one possible implementation of the MU0 datapath, along with the key control signals that control the operation of the various components of the datapath.

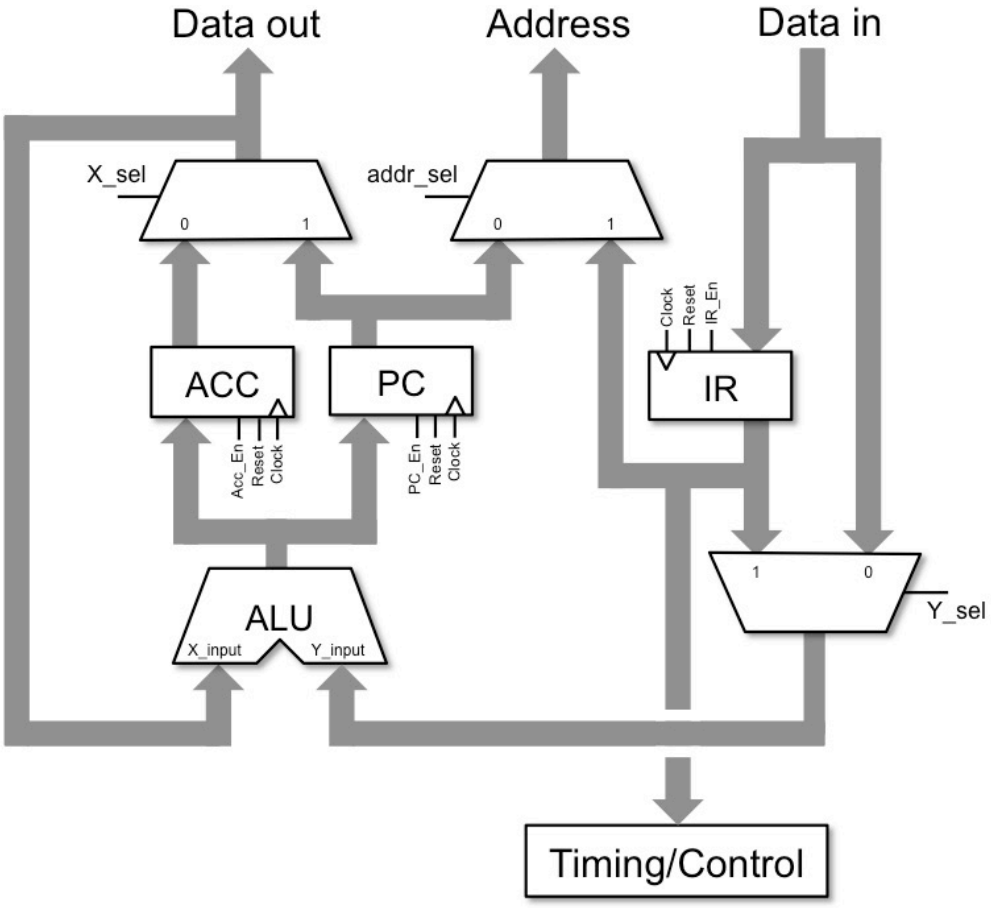


Figure 4.5: Possible MU0 datapath

Data movement through the datapath is controlled using three multiplexers, which do the following:

- X_sel selects between the accumulator (ACC) and program counter (PC) to form the X_input to the ALU – 16 bit input/output.
- addr_sel selects between PC and the address section of the instruction register to form the address to memory – 12 bit input/output.
- Y_sel selects between the data to the ALU or the address section of the instruction register (IR) to form the Y_input to the ALU – 16 bit input/output.

As the address bus is 12 bits and the data bus 16 bits, then we have a mixture of 12 and 16-bit buses in the MU0 datapath. As the PC is 12 bits, then we have to extend this to 16 bits

before it can be passed through the X_sel MUX, consequently we must prepend 4'b000 to the value of PC before the input to the MUX – you can see how this is achieved in the MU0 datapath schematic (see the next section).

The operation of the MUXs is determined by the current state of the FSM, as well as the type of instruction during the decode/execute phase. Figure 4.6 gives some detail of the state of these control signals as a function of the current state of the FSM and the instruction.

state	F[2:0]	Instr	X_sel	Y_sel	addr_sel
0	xxx	xxx	1	x	0
1	000	LDA	x	0	1
1	001	STA	0	x	1
1	010	ADD	0	0	1
1	011	SUB	0	0	1
1	100	JMP	x	1	x
1	101	JGE	x	1	x
1	110	JNE	x	1	x
1	111	STP	x	x	x

Figure 4.6: MU0 State Transition Table

Fetch phase

The PC holds the address of the *next* instruction in memory. In the fetch phase the memory address held in the PC must be placed on the address bus so that the data from memory (the instruction) is read into the IR. So addr_sel must select the value from the PC to be placed on the address bus in the fetch phase.

The value in the PC is incremented by applying the value in the PC to the X_input to the ALU and performing a +1 operation (M = 10). So X_sel must select the PC during this phase.

As the Y-input to the ALU is not used during an instruction fetch, then we do not need to worry what Y_sel is set to.

Decode/execute

During the decode/execute phase the operation of the datapath depends upon the instruction fetched. The values of the control signals addr_sel and Y_sel, depend on the type of instruction being executed. We can see from Figure 4.6 that the value of X_sel is independent of the instruction and should be set to 0 during the decode/execute phase.

During the instruction decode/execute phase any data placed on the address comes from the IR, as the address portion of the instruction (least significant 12 bits) provides the address of the data to be operated on. Consequently, addr_sel should be set to 1 during this phase.

The value of Y_sel depends on the instruction. In the case of all instructions apart from branch instructions the data to the Y_input of the ALU must come from memory – as here we are operating on data from memory for these instructions. However, in the case of a branch instruction (Jxx) the data to the Y_input of the ALU comes from the instruction, which is the least significant 12 bits of the IR, as it is the address of the new instruction (to go into the PC) if the branch is to be taken.

Completing MU0

Your task is to complete the MU0 processor datapath to produce a fully working MU0 processor - the control etc. is already completed for you.

You are provided with an incomplete MU0 datapath schematic in the MU0_lib library as a starting point. If you open the MU0 datapath schematic (MU0_lib -> MU0_datapath), you will notice that some of the constituent parts are missing, such as registers and multiplexers. To get MU0 working you will have to complete the datapath by adding these missing components and configuring them accordingly to provide the correct operation, as described above.

Registers can be constructed from D-type flip-flops. Several flip-flop variants are available in the library; a key to their names can be found in Appendix C. You will need to implement 3 registers: the accumulator (ACC), the program counter (PC), and the instruction register (IR). These registers need an enable input, which must be connected to the appropriate control signal (`Acc_En`, `PC_En`, or IR_En), they require a clock input, and an input to clear the contents of the register (acting as a reset). Choose an appropriate flip-flop that will satisfy these requirements from the library available.

You will also need to design the missing multiplexers as Verilog modules and instantiate their symbols in the MU0 datapath schematic. Cells have been provided for you: "`mux_2to1_12bit`" and "`mux_2to1_16bit`", both with view "functional". You should complete the Verilog module descriptions for these two MUX designs, and simulate them (writing your own test stimulus) to confirm they operate correctly. Figure 4.7 gives some example code for implementing a 2:1 MUX that selects between two single bit inputs, you can use this as a basis for constructing 12-bit and 16-bit versions.

```
module mux2to1   (input          select,
                  input          d0,
                  input          d1,
                  output reg     q);

// a mux is a combinatorial circuit, so no clock signal is
// required

always @ (*)             // whenever select or d change
begin                    // so does the output
  case(select)           // use a case statement to
    0: q = d0;           // determine the assignment.
    1: q = d1;
    default: q = 1`hx;   // default for testing
  endcase
end
```

Figure 4.7: Verilog module for a simple 2:1 MUX

Once you have completed your multiplexers you can add these to the ALU datapath - take care that the multiplexer is connected correctly – this may take some thinking, as problems with this exercise usually relate to the control signals for the multiplexers selecting the wrong inputs – see Figure 4.6 and the accompanying discussion.

Top Tip

How do I create a register?

Registers are simply a collection of flip-flops with the control signals connected together, and the inputs and outputs treated as buses. So one flip-flop is required for each bit of the bus – a 16-bit register requires 16 flip-flops. The Cadence tools can be an enormous help here by simplifying the design process by creating the register for you. If you create a multiple instantiation of a flip-flop symbol (as you did for the ground symbol in the previous exercises) then it will make the control signals common to all the flip-flops and assume that you have a bus input and a bus output.

If you open the MU0 ALU schematic (MU0_lib -> MU0_ALU) you will notice that the 16-bit adder in the ALU is missing. You have produced a 16-bit adder already, so you can easily finish off the ALU design! However, you must remember to connect the carry in to the 16-bit adder to something (we'll leave it to you to figure this out).

It is important to understand the operation of the datapath to ensure the multiplexers and control signals are connected properly, simulation helps greatly with this – remember you should be testing all designs that you create. So once you have completed the MU0 datapath the processor can be simulated to test its operation and confirm that it works correctly. A cell "`MU0_test`" with view "schematic" has been provided for testing the operation of your processor. To enable the processor to be tested a memory model must be provided to supply data and instructions to the processor for testing, so we provide you with a Verilog module memory_mu0, which you can find instantiated in the `MU0_test` schematic. If you open `MU0_test` you will see a statement beginning "$readmemh", this loads a pre-compiled programme into the implemented memory block. For the simulation of the MU0 the memory image `MU0_test.mem` is loaded into memory when the design is compiled, the code for this test is given in Appendix D, and available at

/opt/info/courses/COMP12111/MU0_examples/MU0_test.s.

Use this test program to simulate and therefore test the processor's operation. For the simulation you will need to edit the stimulus file to define the operation of the clock, `Clk`, and reset, `Reset`, signals – this is important!

If the processor fails to behave as expected then it is likely that one or more multiplexers connected incorrectly (see the top tip on the next page!).

Implementation of MU0 in hardware

Now you have a fully working and tested MU0 processor. The only things that remains is to see the design working in hardware!

Components are provided to help integrate your design on the lab experimental board available in the lab. To save time these have been set out in a design called '`MU0_system`'. `MU0_system` defines the components required to enable MU0 to work on the experimental boards in the laboratory, as well as interface MU0 to the hardware components found on the experimental board. Open `MU0_system` and then open the simulation window (Xilinx -> Simulation) and synthesise and download the design as you have done for previous exercises. Note: as `MU0_system` is more complex than anything

you have implemented before it may take a while to synthesise the design, so please be patient!

Top Tip

How do I know that MU0 is working?

The test program writes to memory locations a number of times before the halt signal goes high (34 clock cycles after reset goes inactive). Look out for the `Wr` signal going high before checking the memory address locations given in Figure 4.8.

Addr	Dout
`h22	`h0000
`h23	`h8000
`h24	`h0000
`h25	`hFFFF
`h27	`h1A55
`h28	`h1A55

Figure 4.8: Debugging MU0_test data

In order to provide a debug environment for MU0 and to allow you to update the code running on MU0 we provide a monitor environment called Perentie; this is similar to Komodo that you will have encountered in COMP15111.

To run Perentie simply run the command

start_perentie 12111

from the command line, this should open the a window to "Select Debugger Target", as shown in Figure 4.9.

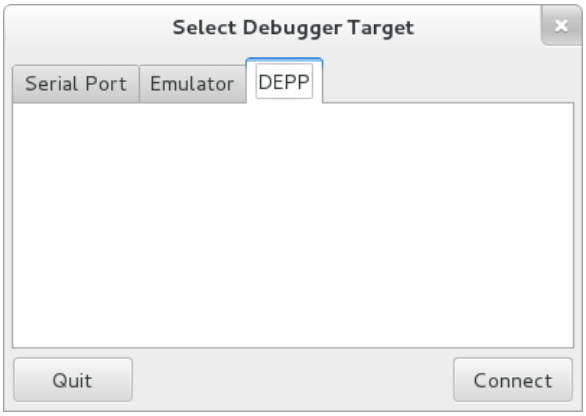


Figure 4.9: Select Debugger Target window

Navigate to the DEPP tab and click the "Connect" button. If successful, you should see the Perentie window as shown in Figure 4.10.

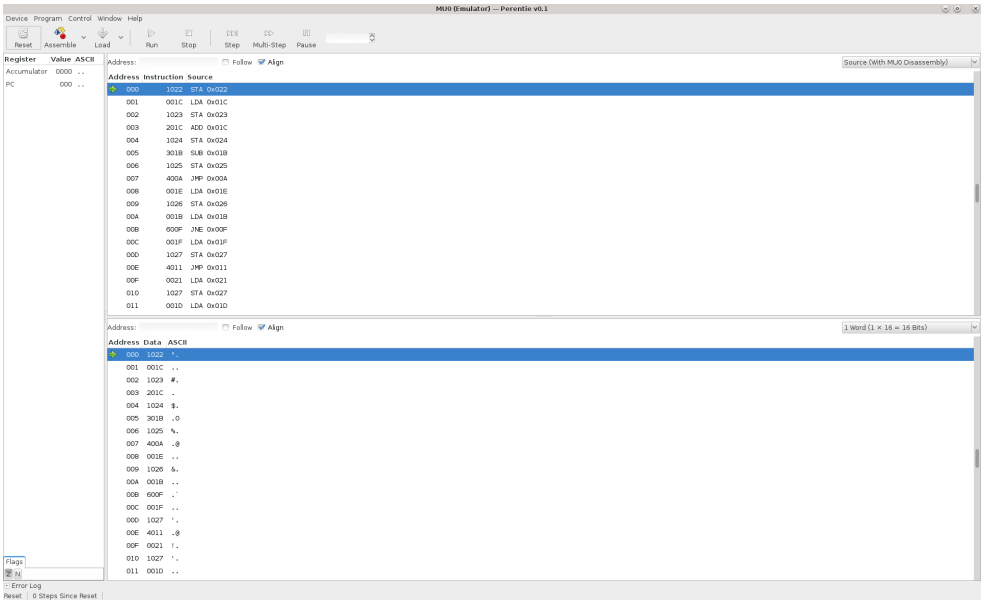


Figure 4.10: Perentie window

If you run Perentie before downloading the `MU0_system` design to the experimental board then Perentie will open up as illustrated in Figure 4.11. If so, close Perentie and download a working version of `MU0_system` to the experimental board before running Perentie again.

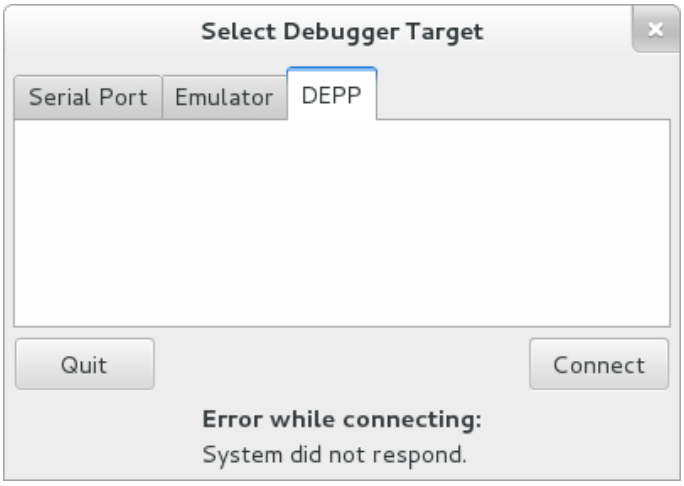


Figure 4.11: Board not connected/no power

In Perentie you can compile and run programmes for your working MU0 processor, view the contents of system registers and memory, update the working code, as well as step through the instructions in your programmes.

The predefined code in `memory_mu0` will have been downloaded to the FPGA with the MU0_system, which is initially the programme `MU0_demo.mem` that you used when testing your MU0 design. If you wish to load a different pre-compiled programme to the memory then you do this via the "Load" button in Perentie, simply click on the down arrow next to the button and select "Select Image File" then browse to the file (compiled files have the extension .kmd). If you need to assemble a file first then select the down arrow next to the "Assemble" button, select "Select Source File" and browse to the file (extension of .s). The file will then be compiled and any errors will be displayed in the

error log at the bottom of Perentie. If the code compiles successfully then a message "Complete. 3 passes performed." will be displayed in the error log. You can now load the compiled code (the .kmd file will have been saved to the same location).

If you select "Reset" to reset the processor, then "Run" in Perentie then you should see the demo programming code in Perentie. It is always good practice to reset the processor first using the "Reset" button. The demo code doesn't actually show you much running on the board as it just interacts with memory. However, you can use Perentie to observe the effect of different instructions on the system registers and memory. An additional demonstration programme can be found at

```
/opt/info/courses/COMP12111/MU0_examples/MU0_demo.kmd
```

which can you load directly into the memory in your working system via the "Load" button in Perentie. You should load this programme and run it for the demonstration. What programme is running on the experimental board?

ex4: MU0

Once you have completed your design, simulated it, and downloaded it to the experimental board you can submit it for marking using the 12111submit script. Follow the instructions and submit the design as 'ex4' and enter 'submit'.

Print out a labprint sheet and answer the questions before you demonstrate the exercise in your next scheduled lab. There is one labprint sheet for this exercise.

Please note: if the design fails to submit (files not found message) then this is probably because you have not simulated your MU0 design.

You have now completed Exercise 4.

Exercise 5: Programming MU0

Exercise Duration	1 lab session – week 1.11
Submission Deadline	End of your scheduled lab session in week 1.11. Submission is ex5.
Extended Deadline	Beginning of lab in week 1.12
Offline Marking	No
Demonstration Required	Yes – run labprint for ex5.
Feedback	N/A
Assessment	25% of the total marks available for the lab.

Aim

To do something with MU0 and create your own programme.

This exercise follows on directly from exercise 4, so you must ensure MU0 is working before starting this exercise.

Preparation

Read the exercise description thoroughly. We are deliberately leaving the details of programming MU0 vague as we want you to experiment!

Software Development

You have produced a fully working processor – MU0 – albeit with a very limited instruction set. Hence, it is possible to write software that you can run on the processor that can interact with the components available on the experimental board. You can create your own code using your favourite text editor, which you should save with a .s extension. You should compile your code via Perentie, which will produce a .kmd file if successful. You can then load this compiled file onto MU0 via Perentie.

Appendix E contains some hints and tips on writing code for MU0.

Using hardware on the experimental board

MU0 is connected to components on the lab board via the symbol "`board`" in MU0_system. Devices on the board are memory mapped so that you can make use of them in your code. This means that as well as accessing instructions and data held in physical RAM, addresses can also be used to access physical hardware connected to the processor. Figure 5.1 lists the components available on `board` and the associated memory address for accessing each.

Feature	Address
Traffic Lights	&FFF
Bargraph	&FFE
Buzzer input	&FFD
Board LED	&FFC
Display Enable	&FFB
Digit 5	&FFA
Digit 4	&FF9
Digit 3	&FF8
Digit 2	&FF7
Digit 1	&FF6
Digit 0	&FF5
S6 LEDS	&FF4
Buzzer busy	&FF3
Key Row 4	&FF2
Key Row 3	&FF1
Key Row 2	&FF0
Key Row 1	&FEF
Switches	&FEE

Figure 5.1: Board memory mapped devices

Traffic lights

The traffic lights are mapped to the 16 bits at address &FFF. Each bit is connected to a particular LED, as illustrated in Figure 5.2. Taking a bit high will switch the corresponding LED on; taking the bit low will switch it off. So to produce the required light pattern you simply write an appropriate 16-bit value to address &FFF, which will result in the corresponding light pattern being displayed on the LEDs. For example, to light just the

green LEDs on the right hand side of the board you would write the value &0009 to &FFF.

Bit number	15:6	5	4	3	2	1	0
LED	Not used	Left Red	Left Amber	Left Green	Right Red	Right Amber	Right Green

Figure 5.2: Traffic Light LED bit map for address &FFF

Bar Graph

The bar graph is controlled by the least significant 8-bits at address &FFE, bits 15:8 are not used. Bit 7 corresponds to the left hand segment, and bit 0 corresponds to the right hand segment.

Piezo Buzzer

The small board containing the FPGA chip contains a piezo buzzer, for which we have created a driver circuit. Control of the buzzer is achieved by writing to address &FFD, the control bits for which are listed in Figure 5.3.

Bit Number	Signals	Function
15	Mode	Controls the operation of the buzzer. Mode = 1 uses the buzzer in programmable mode and configures the buzzer circuit using the values specified for Note, Octave and Duration. If Mode = 0 then the buzzer uses direct control mode where the user can directly control the buzzer using bit 0 of Buzzer input.
14:12		Not used.
11:8	Duration	A value from 0 to 15 that determines the duration of the note in 1/10th of a second intervals. Only used in programmable mode (Mode = 1).
7:4	Octave	A value from 4 to 8 representing the octave. The higher the value, the higher the octave and thus the pitch of the notes. An increase by 1 in octave is equivalent to an increase by a factor of 2 of the note frequency. Values outside the range 4 – 8 will default to 8. Only used in programmable mode when (Mode = 1).
3:0	Note	A value from 0 to 11 representing the 12 semitones in an octave. See Figure 5.4 for corresponding note. Values outside this range default to 0 (note C). Only used in programmable mode (Mode = 1). If the buzzer is set to direct control mode (Mode = 0) then bit 0 is used to control the buzzer, in which case bits 3:1 are ignored.

Figure 5.3: Buzzer input bit map & functions

The buzzer can be controlled in two ways:

- Direct control
- Programmable mode

Code	Note
0	C
1	C#
2	D
3	D#
4	E
5	F
6	F#
7	G
8	G#
9	A
10	A#
11	B

Figure 5.4: Note code & corresponding musical note

Direct Control

In direct control the buzzer is driven directly by modulating bit 0 at address &FFD. To produce a tone you will need to apply a pulse width modulated input to the bit, the period and pulse width of which will determine the sound that is emitted. To use direct control bit 15 (Mode bit) of address &FFD must be set to 0.

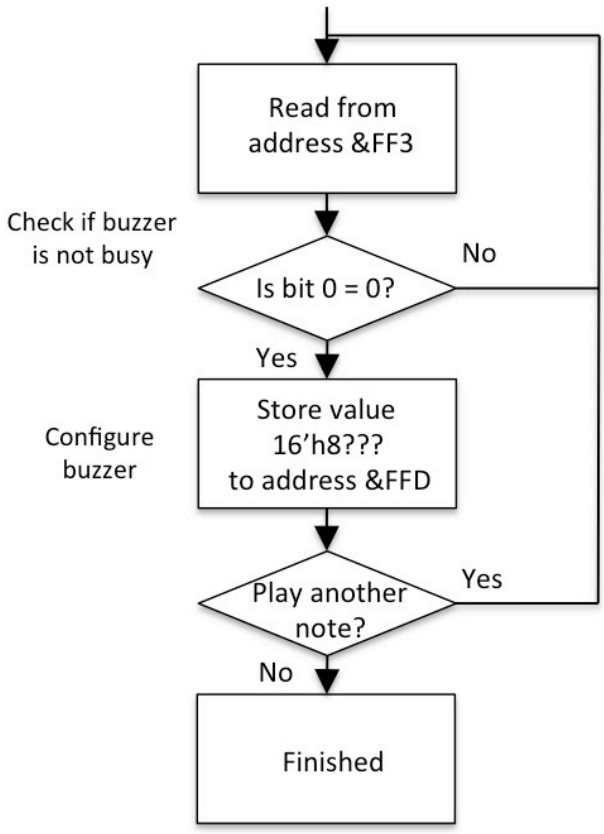


Figure 5.5: Using programmable mode

Programmable Mode

A buzzer circuit is provided that can be programmed to play a defined note for a defined duration, which is determined by writing the appropriate value to address &FFD. To enable programmable mode bit 15 (Mode bit) must be set to 1. There is no need to create a software delay to wait until the note has finished playing. However, you should check that the buzzer is ready by checking bit 0 of address &FF3 (Buzzer busy). If this is 1 then the note is playing. If it is 0 then the current note has stopped playing and buzzer circuit is ready.

The maximum error for the duration of the note held by the buzzer is 1/10 seconds. The maximum error for the frequency of the note is 0.37Hz(on the note C4).

Seven Segment Display

The seven segment displays on board can be used to display information (providing the decoders have been connected appropriately in the `MU0_system` schematic!). To use a seven segment display you must enable the display you wish to use, and then pass the appropriate data for that display to the appropriate address for the seven segment display you are using. There are 6 memory addresses listed in Figure 5.1 for controlling the seven segment displays. Address &FFB is used to enable individual displays as illustrated in Figure 5.6.

Bit Number	16:6	5	4	3	2	1	0
Key	Not used	Digit 5 (lhs)	Digit 4	Digit 3	Digit 2	Digit 1	Digit 0 (rhs)

Figure 5.6: Display enable at address &FFB

To enable digit 4 you would write the value &0010 to address &FFB. To enable all the displays, you would write the value &003F to &FFB.

To display the digit 8 on digit 2 you would enable digit 2 at address &FFB by taking bit 2 high, then write the value &0008 to address &FF7 to display the number '8'.

Keyboard

The four rows of the keyboard are mapped to memory addresses as given in Figure 5.1 - Key Row 4 (top), Key Row 3, Key Row 2, Key Row 1 (bottom). Each row is mapped to the least significant bits at the corresponding address with the left-most button being bit 7, and the right-most button being bit 0. When a button on the keyboard is pressed, the corresponding bit at the address corresponding to the row will go high (zero otherwise). You can determine which keys have been pressed by reading the from the appropriate memory address. Bit 0 represents the key to the far right of the keyboard row, bit 7 represents the key to the far left of the keyboard row.

The task for this exercise is to write and demonstrate a programme written in MU0 assembly code. Be creative as much as you can. Marks will be awarded according to the use of the different instructions, including branches/loops, the number of features used on the lab board, as well as the complexity and creativity of the code you produce.

ex5: Programming MU0

Once you have completed your programme place a copy of your .s file in the directory $home/Cadenced/COMP12111/ex5/. You can submit your code for marking using the 12111submit script. Follow the instructions and submit the design as 'ex5' and enter 'submit'.

Print out a labprint sheet and answer the questions before you demonstrate the exercise in your next scheduled lab. There is one labprint sheet for this exercise.

You have now completed Exercise 5

Appendices

Appendix A

Jargon

The greatest difficult in entering schematics[5] is often learning the jargon. To make things easier a list of terms is included here. Most (but not all) of these are used in the Cadence software. The colours described here are the default colours.

sheet: a piece of (possibly electronic) paper upon which things are drawn. Only really relevant when a schematic is so large that it spreads across more than one sheet.

schematic: a diagram showing the components of a circuit and their interconnections.

symbol: a pictorial representation of a circuit component used on a schematic. A symbol may represent something as simple as a gate, or as complex as a microprocessor. It is typically bound to a schematic of the same name.

component: that which is represented by a symbol. This is probably a whole hierarchy in itself.

instance: a single appearance of a symbol; for example, an AND gate may be *instantiated* many times in a single design.

net: a set of wires connecting various symbols. A net may be a single simple connection, or it may go to many different symbols. **Nets are light blue ("cadetBlue") in Cadence**.

bus: a collection of nets, usually with some familiar function; for example, a computer data bus may have thirty-two wires (32-bits) which are used to encode a binary number. A bus is basically a means of keeping drawings tidy. **Buses are light blue and thicker than single wires in Cadence**.

ripper: a point where a single net enters or leaves a bus (a.k.a. a "breakout").

pin: a place on a symbol where nets can be connected. **Pins on symbols are red squares in Cadence. Pins on schematics are red. The name of the pin on the symbol, and the name of the pin on the associated schematic must be the same**.

connector: a point on a schematic which is used to connect signals to other schematics.

label: an identifier. Every component and net in a design will have a label, although most will be assigned automatically. A label can be attached by hand to identify a specific item, Note that points in the same schematic with the same label will be connected, even if a net is not drawn between them. Labels are used to associate pins of symbols with the correct signals in the schematic. **Labels are the same colour as the component that they label in Cadence**.

text & graphics: these are merely features that clarify a schematic; they have no effect on the actual circuit. Symbols are simple graphical sketches with pins around them; only the pins have any electrical function. **Text and graphics are white in Cadence schematics**.

[5] … or most other subjects

Appendix B

Cadence End User Agreement

Please make sure you have read and understand your obligations contained within the Cadence End User Agreement. A copy of which can be found at:

http://www.europractice.stfc.ac.uk/eua/univs/cadence.pdf

Of particular importance are your obligations to keep the Cadence tools confidential and that the University's Cadence tools may NOT be used for any commercial purposes.

Please be aware that your usage of Cadence software will be recorded and these records will be kept for a minimum of 5 years

Appendix C

Spartan 6 Library Elements

All the designs in this laboratory are built from library elements. Most of these are held in a directory that is available as "**spartan6**"; the 'local' cells are contained in a different library – "**ENGLAB_12111**". You may not modify or add to this library but you can build you own library that uses these components. The following elements are available; you should not need to use any of the elements in the 'Input and output' or 'Miscellaneous' sections:

N.B. Do not use any of the elements from the library 'built in' directly

Basic Gates

Restrict yourself to the gates listed in this section. Any others you might see are for I/O support and will 'break' your design if used inappropriately.

- [AND2] : 2-input AND gate
- [AND2B1] : 2-input AND gate with one inverted input
- [AND2B2] : 2-input AND gate with two inverted inputs (functionally equivalent to a 2-input NOR gate)
- [AND3] : 3-input AND gate
- [AND3Bn] : 3-input AND gate with n inverted input(s) – n={1,2,3}
- [AND4] : 4-input AND gate
- [AND4Bn] : 4-input AND gate with n inverted input(s) – n={1,2,3,4}
- [AND5] : 5-input AND gate
- [AND5Bn] : 5-input AND gate with n inverted input(s) – n={1,2,3,4,5}
- [NANDxxx] : as AND gates but using NAND function
- [ORxxx] : as AND gates but using OR function
- [NORxxx] : as AND gates but using NOR function
- [XORn] : N-input exclusive-OR - n={2,3,4,5}
- [XNORn] : N-input exclusive-NOR n={2,3,4,5}
- [INV] : inverter
- [BUF] : tristate buffer – active low enable
- [VCC] : logic high
- [GND] : logic low

Flip-flops

- [FDxxx] : Positive (rising) edge-triggered D-type flip-flop. Any following letters indicate other control inputs, as follows:

C – asynchronous clear	(to '0')
P – asynchronous preset	(to '1')
R – synchronous reset	(to '0')
S – synchronous set	(to '1')
E – clock enable	active high

- [FDxxx_1] : As FDxxx but negative (falling) edge triggered.

Local 'macrocells' (ENGLAB_12111)

The only cell you need to place for I/O support is '**Board**'; any other cells in this library are contained inside that one.

The interfaces to Board are:

- Clk_???Hz : A free-running clock signal at the indicated frequency.
- Key_row?<7:0> : The key states for a row of keys. Row 1 is the top row. The bits are numbered from #0 at the right-hand side. A '1' state indicates the key is pressed.
- Display_en<5:0> : Enable signals for the six seven-segment displays. The displays are numbered from #0 at the right-hand side. An enable must be '1' for a display to illuminate.
- Digit?<7:0> : The seven segments of each display. The segments are numbered from #0 for segment a; #7 is the decimal point. A '1' illuminates the appropriate segment.
- Bargraph<7:0> : The bargraph LEDs. #0 is at the right-hand side. A '1' illuminates the relevant LED. *Note: the physical bargraph has ten LEDs – only the middle eight are connected.*
- Traffic_lights<5:0> : The 'traffic light' LEDs. A '1' illuminates the appropriate LED. These are numbered as follows:

#5	Left,	red
#4	Left,	amber
#3	Left,	green
#2	Right,	red
#1	Right,	amber
#0	Right,	green

Appendix D

MU0 Test Programme – MU0_test.s

```
; Simple MU0 verification programme
; JDG
; November 2008
; Modified Jan 2012 by JSP

; Begin program at reset address. Acc, pc, and ir should all
; be 0 after reset

          ORG  0

; Sore to memory and ALU tests

; Test store to memory
          STA  result1     ; Store acc into memory loc result1.
                           ;result1 = 0

; Test load accumulator from memory
          LDA  neg         ; Acc should be set to 'h8000
          STA  result2     ; Store value of acc to memory.
                           ; result2 = 'h8000

; Simple adder overflow test
          ADD neg          ; Acc should overflow to 0
                           ; ('h8000 + 'h8000)
          STA  result3     ; Store the addition result to memory.
                           ; result3 = 0

; Simple subtraction test
          SUB one          ; Acc should be 'hFFFF(0 - 1 = -1)
          STA  result4     ; Store the subtraction result to
memory.
                           ; result4 = 'hFFFF

; Test unconditional jump(JMP) - (always jump)
; If JMP passes result5 = 'h1A55, else result5 = 'hFA01

          JMP  jmp1ok      ; Pc should be set to jmp1ok
          LDA  jmperr1     ; If jump fails load error value
          STA  result5     ; If jump fails set memory to
                           ; failure value.

; Test conditional jump(JNE) based on the Z(zero) flag
; Relies on the JMP instr already being tested and working

; Test JNE for when Z flag is NOT set
; If JNE jumps when it should result6 = 'h1A55,
; else result6 = 'hFA02

jmp1ok    LDA  one         ; (Z)zero flag not set
          JNE  jmp2ok      ; Jump SHOULD be taken and execute the
                           ; "pass" reporting code
```

```
; error reporting code
          LDA  jmperr2     ; If JNE failed load the acc with
                           ; error value
          STA  result6     ; If JNE failed set memory to failure
                           ; value. result6 = 'hFA02
          JMP  fail1       ; If JNE failed, then jump over the
                           ; "pass" reporting code
; pass reporting code

jmp2ok    LDA  pass1       ; If jump taken load the acc with the
                           ; pass value
          STA  result6     ; If jump taken set memory to pass
                           ; value. result6 = 'h1A55

; If JNE jumps when it should NOT have then result7 = 'hFA03, else
; result7 = 'h1A55

; Test JNE for when Z flag is set

fail1     LDA  zero        ; (Z)zero flag set
          JNE  fail2       ; If JNE jumps here, it shouldn't have,
                           ; execute error reporting code
          JMP  jmp3ok      ; If JNE was ok, then execute the
                           ;"pass" reporting code
; error reporting code

fail2     LDA  jmperr3     ; If JNE jumped when it should not
                           ; have, then load error value
          STA  result7     ; If JNE fails set memory loc to
                           ; failure value. result7 = 'hFA03
          JMP  stop        ; If JNE failed, then jump over the
                           ; "pass" reporting code
; pass reporting code

jmp3ok    LDA  pass1       ; Load the acc with the pass value
          STA  result7     ; Set memory to pass value.
                           ; result6 = 'h1A55

; End of test for conditional jump(JNE) based on the Z(zero)
; flag

stop      STP              ; STOP - HALT program
done      JMP done         ; Just in case stop instr fails

; Definitions

one       DEFW 1           ; one
neg       DEFW &8000       ; -max
zero      DEFW &0000       ; zero

jmperr1   DEFW &FA01       ; jump fail values
jmperr2   DEFW &FA02
jmperr3   DEFW &FA03

pass1     DEFW &1A55       ; jump pass value
```

```
; result storage area

result1    DEFW &FFFF
result2    DEFW &0000
result3    DEFW &FFFF
result4    DEFW &0000
result5    DEFW &1A55
result6    DEFW &0000
result7    DEFW &0000
```

Appendix E

Writing Code for MU0

The following provides a brief overview of assembly language for MU0. The discussion here is not exhaustive and has been left deliberately vague so that you experiment with writing your own code.

In MU0, instructions are fetched from consecutive memory addresses, starting from address 0, unless a branch instruction updates the value of the PC, at which point fetching continues from the new address given by the branch instruction. The PC is always updated, by adding 1, to point at the next instruction during the fetch phase.

Commenting Code

It is essential that you comment your code; this makes it easier for you to understand your code, as well as helping others understand your code. Comments can be placed in your code by preceding the comment with a semicolon, ;, for example:

```
; The following code does something
```

treats all the text on that line following the semi-colon as a comment.

You can add comments after your code, for example

```
ADD &123  ; Add to the accumulator
```

It is important that your comments are useful and that they explain **why** you are doing, rather than just explaining **what** you are doing. For example, the comment

```
ADD  avalue    ; add the array index to the address in acc
```

offers more insight compared to

```
ADD  avalue    ; add the data at label avalue to the acc
```

which basically says what I can infer anyway from the instruction.

Precompiler Directives

ORG

`ORG` may be used to specify the address where the instruction/data following the `ORG` statement is placed in memory, i.e.

```
ORG  &100
```

You can use `ORG` to specifying the location of both instructions and data in memory, and you can use as many `ORG` statements as you like in your code.

`ORG` can be useful for partitioning your code and separating instructions from data.

Consider the example:

```
ORG   &00F

      ADD   &123
      STR   &123
      :           ; other code

ORG   &123

      DEFW &0000
      DEFW &0000

; end of code
```

will place the code following the `ORG &00F` statement starting at memory address F_{16}. So the `ADD` instruction will be placed at address F_{16} in memory, the `STR` instruction at address 10_{16}, with any following instructions being placed in consecutive memory locations. The data, defined using `DEFW` (more later), will be placed starting at address 123_{16}.

EQU

`EQU` is a precompiler directive that allows you to define constant values that are used frequently in code. These are particularly useful to aid legibility and to help with debugging code.

For example, consider the code

```
output     EQU   &FFF
```

will define a variable `output`, with a value FFF_{16} that you can use throughout your code. During compilation the compiler will replace all instances of `output` in the code with the value FFF_{16}.

Labels

Labels can be used to represent an address somewhere in memory, and can be used for specifying an address to jump to for a jump/branch instruction, or to point to data in memory (see `DEFW` later).

The format is:

```
<label>  <instruction or data>
```

So for example

```
           JMP   alabel
           :

alabel     ADD   &123
```

is used to specify the address for which the `JMP` instruction branches to.

MU0 Instruction Set

Load/Store Instructions

MU0 has two instructions that can be used to move data to/from the accumulator from/to memory:

`LDA <address>` - load the contents of the memory address specified into the accumulator, i.e. ACC := mem_{16}[<address>]

`STA <address>` - store the contents of the accumulator to the memory address specified, i.e. mem_{16}[<address>] := ACC

So, for example, if memory address &123 contains the value $FFFF_{16}$ then after executing the instruction

```
LDA &123
```

the accumulator will contain the value $FFFF_{16}$.

If the accumulator contains the value $65{,}536_{10}$ then after executing the instruction

```
STA &FFF
```

the memory address FFF_{16} will now contain the value $65{,}536_{10}$.

Arithmetic Instructions

MU0 has two arithmetic operations that operate on the contents of the accumulator and data stored at a specified memory address:

`ADD <address>` - add the contents of the accumulator to the contents of the specified memory address and store the result in the accumulator, i.e. ACC := ACC + mem_{16}[<address>]

`SUB <address>` - subtract from the contents of the accumulator the contents of the specified memory address and store the result in the accumulator, i.e. ACC := ACC – mem_{16}[<address>]

So, for example, if memory address &123 contains the value 15_{10} and the accumulator contains the value -5_{10} ($FFFB_{16}$) then after executing the instruction

```
ADD &123
```

the accumulator will contain the value $000A_{16}$,

If memory address &123 contains the value 15_{10} and the accumulator contains the value -5_{10} ($FFFB_{16}$) then after executing the instruction

```
SUB &123
```

the accumulator will contain the value $FFFB_{16}$, or -14_{16} (-20_{10}),

Branch Instructions

There are three branch instructions that can be used to interrupt the flow of instructions by overwriting the address of the next instruction held in the PC:

`JMP <address>` - always overwrite the PC with the address specified in the instruction, i.e. PC := <address>

`JGE <address>` - overwrite the PC with the address specified in the instruction if the current accumulator value is greater than or equal to zero, i.e. if ACC ≥ 0 PC := <address>, else PC := PC

`JNE <address>` - overwrite the PC with the address specified in the instruction if the current accumulator value is not equal to zero, i.e. if ACC ≠ 0 PC := <address>, else PC := PC

where `<address>` can be explicitly stated in the instruction, or can be a label (see later). For all branch instructions the PC is always changed, either by 1 (PC+1 During fetch) or to the value given by the instruction.

So, for example, after executing the instruction

```
JMP &123
```

the PC will be updated with the memory address 123_{16}, this will be then be the next instruction to be executed.

If initially the N flag is zero, then after executing the instruction

```
JGE link1
```

the PC will remain unchanged (stays PC+1) as the contents of the accumulator must be ≥ 0, since the negative flag is set to 0. However, if the value in the accumulator is negative, then the N flag is set, so after executing the same instruction the PC will be updated to the address corresponding to the label `link1`, which is the memory address from which the processor will continue to execute instructions.

If initially the Z flag is zero, then after executing the instruction

```
JNE &010
```

the PC will remain unchanged (stays PC+1) as the contents of the accumulator must be ≠ 0, since the zero flag is set to 0. However, if the accumulator is zero, then the Z flag is set, so after executing the instruction the PC will be updated with the address 010, which is the memory address from which the processor will continue to execute instructions.

In general, the result of `JGE` will depend on the state of the N flag, whereas the result of `JNE` will depend on the state of the Z flag.

Defining Data in Memory

MU0 operates on data that is stored in memory, so you cannot specify data directly in the instruction. Instead, instructions specify the address where the data is located in memory. You can refer directly to the address of the data in memory, i.e.

```
     ADD  &0FF
```

where the data has been reserved and defined using DEFW. However, this requires you to know where the data is located in memory – which is possible if you have positioned it using `ORG`. Alternatively, you can specify a location in memory using a label, again using `DEFW`, i.e.

```
<label>   DEFW <initial_value>
```

Consider the example

```
data DEFW &FFFF     ; allocate a word containing a value
```

which will define a word in memory at an address specified by the label `data` that initially contains the value $FFFF_{16}$. Here the actual address where the data is located in memory is not important.

To load data from a location specified in memory to the accumulator you can use the `LDA` instruction. For example:

```
     LDA  data      ; load data from memory into the acc
```

which will load the value $FFFF_{16}$ from the address specified by the label `data` into the accumulator.

Updating Data in Memory

To store data to a previously defined location in memory you would use a `STA` instruction, for example, the code

```
     LDA  zero
     STA  data
     STP

zero DEFW &0000
data DEFW &FFFF
```

will load the value 0000 from the address specified by the label `zero` into the accumulator and then store this value at the location specified by the label `data`, overwriting the previous stored value of $FFFF_{16}$ with the new value 0000.

Performing a comparison

MU0 has no compare instruction, so how do you perform a comparison? We can use the `SUB` instruction. As all MU0 instructions set the flags (since the values are based purely on what is held in the accumulator) then we can use the result of the `SUB` to make decisions. Consider the code example:

```
ORG &000

     LDA varA
     SUB varB
     STP

varA DEFW &0004
varB DEFW &0002
```

Here, first the data at the address specified by the label `varA` is loaded into the accumulator. Next, the data at the address specified by `varB` is loaded from memory and subtracted from the current value in the accumulator (`varA`) by the ALU, storing the result in the accumulator. This will result in the N and Z flags being set depending on the result produced by the `SUB` operation. Here the result will be 2, so neither flag will be set.

Let's consider another example. You are to produce some code that loads a particular value into the accumulator depending on the relationship between two numbers, `varA` and `varB`, that are held in memory:

- if `varA` = `varB` then we must load the value 0 into the accumulator
- if `varA` > `varB` then we must load the value 1 into the accumulator
- if `varA` < `varB` then we must load the value -1 into the accumulator

Here we are performing a comparison and depending upon the result we load a value into the accumulator. A code example may be:

```
; test code
; compare 2 values and load a value into the accumulator
; that depends on the difference the values

     LDA varA  ; load varA into the accumulator first
     SUB varB  ; subtract varB from varA - comparison
     JNE not0  ; if ACC != 0 then the values are not the
               ; same further tests from label not0

; if we get here the result is zero, i.e. they are the same

     LDA same  ; load 0 into accumulator
     JMP fin   ; jump to the end

not0 JGE notB  ; test if the result is positive
```

```
; if here then the result is negative so varB must be bigger

     LDA Bbig  ; load -1 into the accumulator
     JMP fin   ; jump to end

; if here then the result is negative so varA must be bigger

notB LDA Abig  ; load 1 into the accumulator

fin  STP       ; end of code example

varA DEFW &0002
varB DEFW &0002

same DEFW &0000
Abig DEFW &0001
Bbig DEFW &FFFF
```

where we can preload the values of `varA` and `varB`.

Implementing Delays

You may be required to introduce a delay in the code in order to wait before you execute the next instruction. How could you do this? One way is to create a simple counter that effectively counts through a defined number of instructions. For example, consider the following code:

```
      LDA  delay   ; load initial value into the accumulator
loop  SUB  one     ; subtract '1' from the accumulator
      JNE  loop    ; repeat until accumulator is zero

      LDA  avalue  ; carry on with the delayed instruction
          :

delay DEFW &000F
one   DEFW &0001
```

will loop through the `SUB` and `JNE` instructions until the accumulator reaches 0 from the initial value (in this case 15) loaded. Each instruction takes 2 clock cycles to execute, so for this example the total delay will be

(1 × `LDA` + 15 × `SUB` + 15 × `JNE`)×2.

so 62 clock cycles. Or ((2×delay value)+1)×2 for an arbitrary delay value specified at memory location `delay`.

You can have a look at the MU0 demo program:

```
/opt/info/courses/COMP12111/MU0_examples/MU0_demo.s
```

for ideas on how to write programs for MU0.