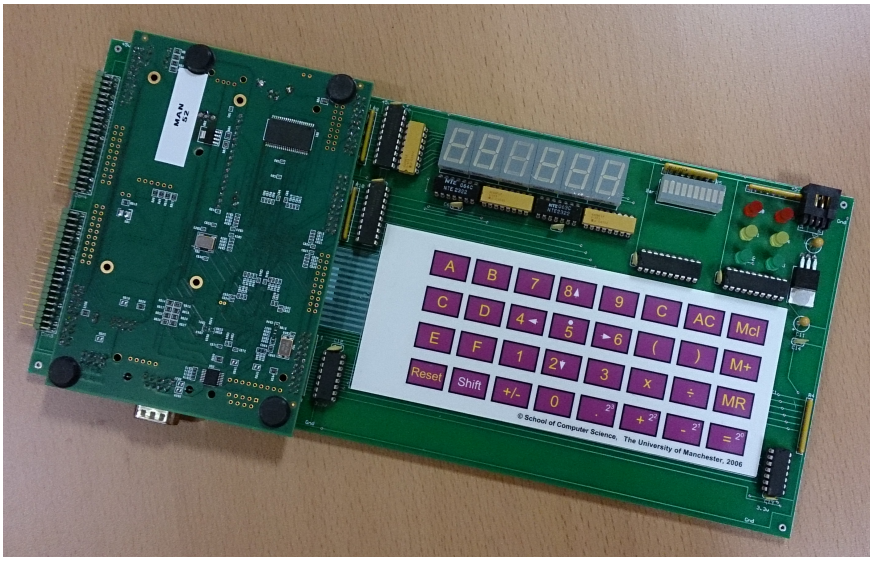
COMP12111

Fundamentals of Computer Engineering



Laboratory Manual 2013/2014

This lab manual belongs to:

……

Lab group

……

Version 2013 - Paul W Nutter

Contents

Lab Organisation

These notes apply to the COMP12111 engineering laboratory associated with the corresponding lecture course. Please read the lab manual carefully before you attempt each exercise. It is important that exercises are attempted 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!

- Laboratory sesssions are timetabled according to laboratory group:

  - Group W– Thursday 3pm – 5pm
  - Group X – Wednesday 9am – 11am
  - Group Y – Tuesday 2pm – 4pm
  - Group Z – Thursday 1pm – 3pm

- **Half of the assessment for this course is based on the laboratory work.** It is therefore important to ensure that your work is completed and submitted in order to maximise the marks you can achieve for the course unit.

- Practical exercises for which work is assessed are highlighted in gray boxes in this manual.

- Demonstrators (who are typically post-graduate students undertaking research projects) and members of staff will be available during all lab sessions to provide assistance where required. Always attempt exercises before seeking help, but *as soon as* you get into difficulty ask for assistance. If you do not ask for help soon enough, you may fail to complete the exercises within the scheduled time – you may then fall behind.

- This manual starts with some background information followed by detailed information about each of the five exercises. **It is IMPORTANT that you READ the background material before you start each exercise**. You should also make use of the lecture material which supports the lab.

- Make sure you read each exercise before you attend the lab. There is not much practical *preparation* you can do for labs but you need to *understand* the exercise and the concepts it is trying to reinforce. Thus, if you do not read the exercise and leave this work until you attend the laboratory, you will not complete the practical work within the scheduled sessions and be left needing to catch up. In addition to the practical instructions the laboratory manual contains sections of relevant descriptive material which should form useful lecture revision.

- The first lab exercise consists of several short introductory exercises designed to familiarise you with basic logic gates and some simple circuits – for this you will not need a computer. The rest of the laboratory comprises a larger set of similar exercises which are 'wired' inside a field programmable gate array – these will involve the use of CAD tools.

- A number of exercises come with the interface to a module defined for you, either in the schematic or the verilog module header. It is important that you **DO NOT CHANGE** any

of the interface signals, do not rename them, change the bus size, or add more inputs/outputs. If you do so, you may find it difficult to submit your work.

- Exercise 1 is completed using a pro-forma answer sheet (available in the lab or via the course website) – this must be posted in the postbox at the front of Tootill 1 by the deadline. The remaining exercises are submitted using the submit mechanism. To submit any exercises for this lab you must run the submnit script

  12111submit

  and follow the instructions given. More detail is given later on this process.

- **Fixed deadlines to submit work are applied in this laboratory.** Each exercise has a deadline associated with it. The deadline for a lab exercise is **11pm on the day of your lab**. Apart from optional extra exercises, which have a deadline of the START of the last timetabled lab (the marking lab).

- Failure to submit your work by the deadline will result in work being treated as being submitted late – LATE penalties will be rigorously applied for any work submitted after a deadline, unless we know of any mitigating circumstances. Extensions are available to the **START of the next lab** if required (so if your lab starts at 3pm, the extended deadline is until 3pm), but it is up to you to ask for an extension, which are **NOT** automatic. Extended deadlines will only be granted if you are seen to be making progress with the exercise you request the extension for. In the case of exercises handed in using the submit mechanism, the hand-in date is the date you submit your work via the 12111submit submission mechanism (see next section).

- Keeping to the schedule gives more time for the laboratory demonstrators to help you, if you need it. Please manage you time carefully and DO NOT use time in this lab to complete other labs – if you do so, you will be asked to leave and your attendance will not be counted.

- You must **ONLY ATTEND YOUR OWN** scheduled lab session. There are a limited number of computers in Tootill 1, and a limited amount of demonstrator time. It is unfair to demand demonstrator time in someone elses lab session! Anyone found in the wrong lab will be asked to leave.

- Please do not bring food and drink (apart from bottled water) into the laboratory – spills are inevitable and greasy fingers make keyboards and screens unpleasant for everyone.

- Please report any faulty equipment so that it can be repaired.

- If you have suggestions for improving the laboratory or the manual, please e-mail these to

  - Paul Nutter (p.nutter@manchester.ac.uk),
  - Vasilis Pavlidis (vasileios.pavlidis@manchester.ac.uk)
  - Tom Thomson (Thomas.Thomson@manchester.ac.uk).

Submission of Lab Work

The submission of your work for this lab differs slightly to the process used in other labs. Here, you submit your work in two ways depending on which exercise you are handing in:

Exercise 1

For exercise 1 you are provided with pro-forma answer sheets to complete (available in the Tootill 1, or can be downloaded from the course website). As you progress through the each part of exercise 1 you will be required to demonstrate your work to a lab demonstrator and have your answer sheet signed off – failure to have designs signed off will result in marks being lost when your work is marked. Once completed you must attach a completed blue report card – which can be found at the front of the lab – completed with your name, group {W, X, Y, Z) and exercise number, to your answer sheet. Submit your answer sheet by posting it in the post-box at the front of the lab.

Exercises 2-5

To submit your work for exercises 2-5 you must use the 12111submit mechanism.

Marks for each of these exercise come from two sources:

- Automatic marking that tests the operation of your designs to check they work as expected and that the testing/simulation of your designs is sufficient.
- A formal demonstration.

The breakdown of marks between these two marking mechanisms varies between exercises and is given in the next section.

For the automatic marking you are required to submit your work by the deadline for that exercise. This also serves to notify Arcade of the date and time your work was submitted so that late penalties can be applied.

The automatic marking looks at two aspects of your work:

- It checks whether you design works by testing your design against our own test stimulus, and
- It tests your stimulus against our own design to check that your test coverage is sufficient for you to have sufficiently tested your own design.

For the formal demonstration you must show your design working on the experimental boards in the laboratory, as well as answering any questions in the lab manual (there will be space provided on the demonstration answer sheet for this.

To submit an exercise you must run the 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 12111submit script you will be presented with 3 options:

1) Submit design

Use this option if you want to submit your work for marking. You will be asked to identify the exercise you wish to submit, i.e. ex2a, and then you will be presented with 3 options:

a) type "submit" to submit your work first the first time
b) type "submit-again" to submit your work again (be careful as work submitted again after the deadline will be considered late)
c) type "submit-diff" to check your current design against that which you have previously submitted

When you submit your work the script will then look in your Cadence directory for the files it needs for submission, check that these compile, and then copy them over to the marking server for marking. Some exercises have multiple parts (such as exercise 2) so you will need to run the 12111submit script for each of these exercises as you complete them. However, you demonstrate all of them together.

Late submission of work will be detected by this mechanism. Once you have submitted your work it will be automatically marked and you will receive an email containing your mark and feedback. This feedback is provided after the extended deadline for an exercise has passed.

You only need to submit the exercises you have completed.

2) Print exercise marking feedback sheet

Enter this if you need to print out the demonstration marking sheet that you will need for one-to-one demonstrating in your next lab. Please note: there is one marking feedback sheet for each exercise, regardless of the number of parts to an exercise, so you only need to run this option once for each exercise, once you have completed them. Optional exercises have their own demonstration marking sheet.

After submission during the next lab you should arrange your formal demonstration. To tell us you need to demonstrate your work, please write your name and machine number on the white board at the start of the lab – the white board should have lists associated with exercise. Once you have demonstrated your work you will receive a marking token and a mark that you must submit by running the 12111submit script, entering the exercise number and then "submit-mark".

3) Enter demonstration mark

This is used by the demonstrator to submit your demonstration mark during your demonstration lab. A secure token is required (which are provided to demonstrators for each lab session) to submit your demonstration mark.

Laboratory Timetable

The lab consists of 5 exercises. The following table list the deadline week for each exercise – the deadline is **11pm** on the day of the lab session in the week listed. If you finish an exercise early then move onto the next exercise.

If you fail to finish by a deadline, then an extension **MAY** be granted, but extensions are **NOT** automatic. If you would like to request an extension you must request it by the **end** of the lab where there is a deadline. The extension is then until the **BEGINNING** of your next lab.

Date (w/c)	Week	Exercise	Arcade Deadline	submit	Deadline Week*	Marks	Total
	1.2 (B)	Exercise 1					
	1.3 (A)	Exercise 1	1.2D	-	1.3	30	**30**
	1.4 (B)	Exercise 2					
	1.5 (A)	Exercise 2	2.2D	ex2a, ex2b, ex2c	1.5	20[a] 10[f]	**30**
	Reading Week						
	1.7 (B)	Exercise 3	3D	ex3	1.7	30[a] 10[f]	**40**
	1.8 (A)	Exercise 4					
	1.9 (B)	Exercise 4	4.2D	ex4a, ex4b	1.9	30[a] 10[f]	**40**
		Exercise 4x	4.3D	ex4x	1.12	5[a] 5[f]	**10**
	1.10 (A)	Exercise 5					
	1.11 (B)	Exercise 5	5.2D	ex5	1.11	30[a] 10[f]	**40**
		Exercise 5x	5.3D	ex5x	1.12	0[a] 10[f]	**10**
	1.12 (A)	Demo/Marking					
						200	**200**

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

Exercises 4x (ex4x) and 5x (ex5x) are optional extra exercises that you should only attempt if you have time. They have few marks available for them and consequently you will not be penalised too heavily if you do not attempt them. The deadline for submission of the extra exercises is the beginning of the last scheduled laboratory (the marking lab).

Schematics & Schematic Capture

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, the interconnections between them. However, for larger circuits some means of managing this complexity must be introduced.

Large electronic designs 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).

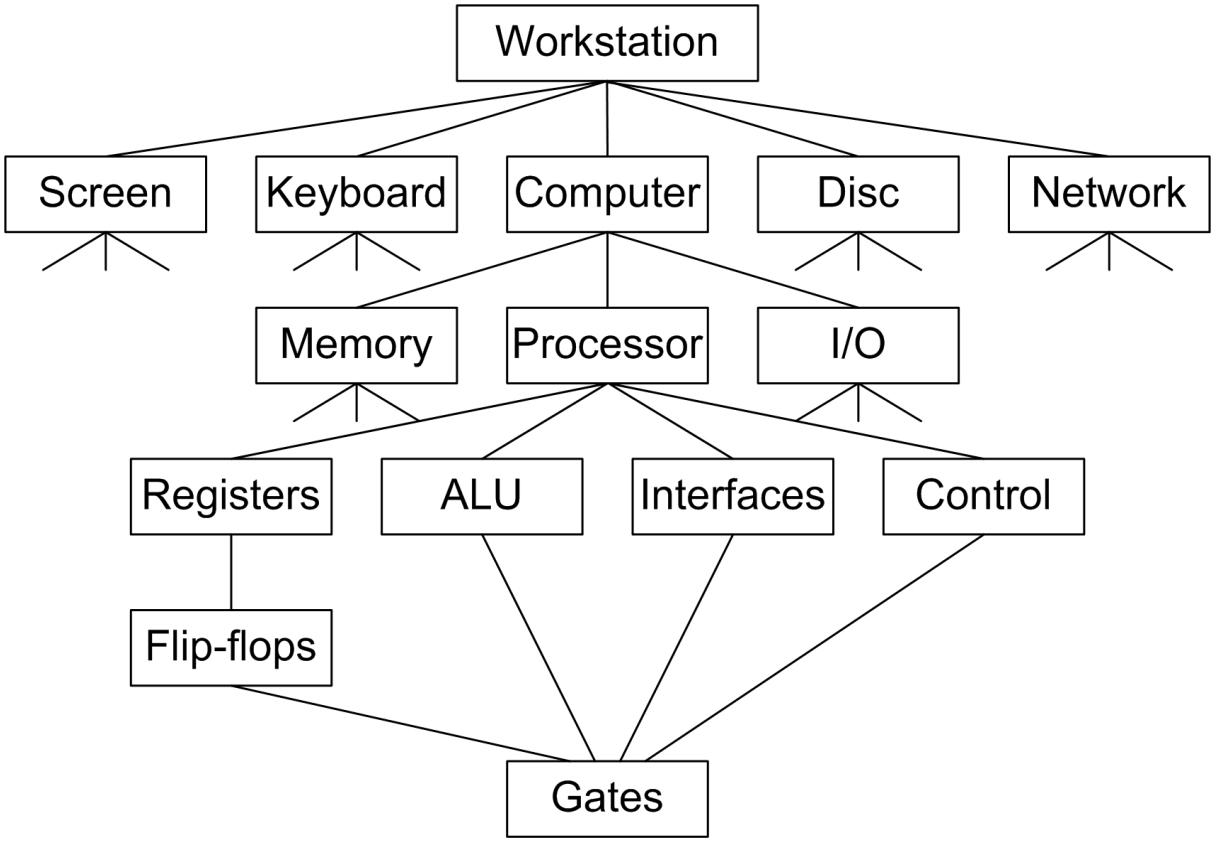


Figure 1.1: Design hierarchy

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**). This allows the schematic diagram to be entered directly into the computer, which maintains the hierarchy and allows 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, and is described in the next section.

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 debugged before it is built. The schematic can even be **compiled** directly into a real circuit automatically, allowing an implementation on a physical device without the need for physical 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 into a **bus**. Buses may be abbreviated to a thickened line or double lines with the width of the bus flagged. Wires (or smaller buses) can be 'broken out' of the bus by appropriate labelling, as shown in figure 1.2.

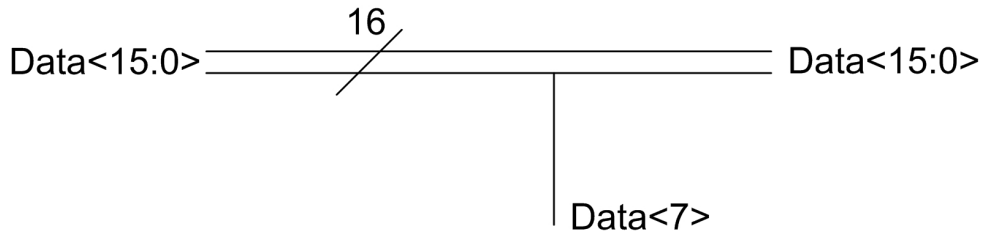


Figure 1.2: A 16-bit bus

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 1.3. 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.

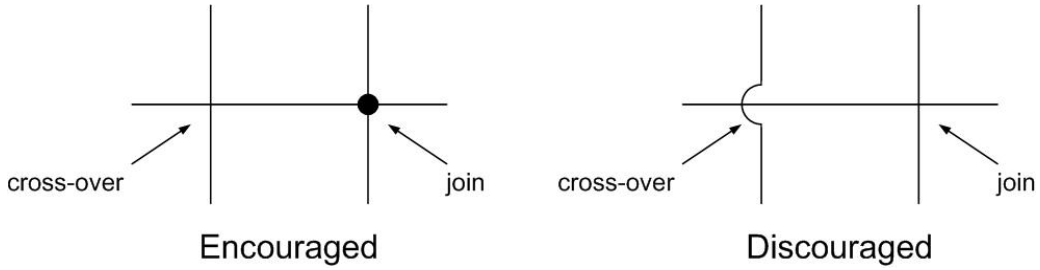


Figure 1.3: Signal wires crossing

On designs using multiple devices[3] always label the pin number of the device close to the signal path joining the symbol. A part number should also be given, preferably just above or below the symbol. Finally, a reference number should be given to identify the physical placement of the device on the circuit board.

Power supply pins need not be indicated but remember that all gates must be powered from somewhere.

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. (This is 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.

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 in Cadence uses angular brackets, '<' and '>', so that convention will be adopted for the remains of this section.

Figure 1.4 shows how buses may be depicted on a schematic. It is normal (not compulsory) to use narrow lines for single wires and thick ones for buses; this makes the schematic a bit easier to read. Whatever is used the network (wire/bus) should be labelled to show its width (i.e. the number of signals contained).

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

[3] Not directly relevant to COMP12111 but worth remembering for the future

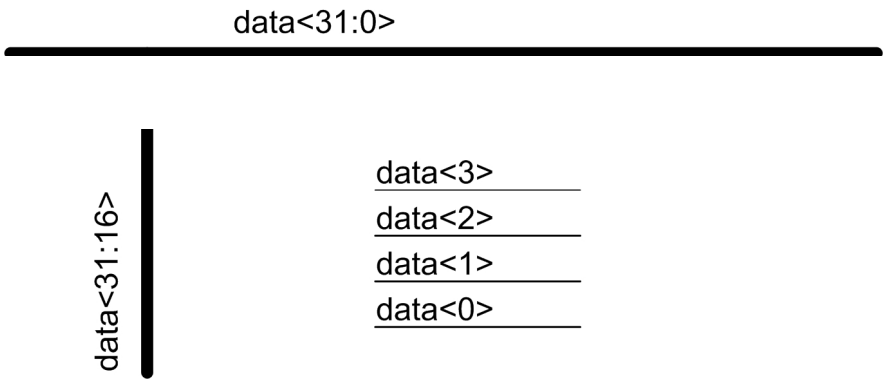


Figure 1.4: Example bus signals

Using wire names to make connections

Whilst it is often convenient to show the buses/signals physically connected in the schematic it is not necessary. In the Vituoso schematic capture tool any set of wires/buses with the same wire name are connected. Thus, in figure 1.4 the individual signals shown (`data<0>`, `data<1>`, `data<2>` and `data<3>`) are also present in the horizontal bus section (`data<31:0>`), as is the vertical bus (`data<31:16>`) - connections are made "invisibly".

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 1.5 illustrates connections between buses inferred using wire names. Here, we have three input pins and 4 output pins, each has a wire/bus 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.

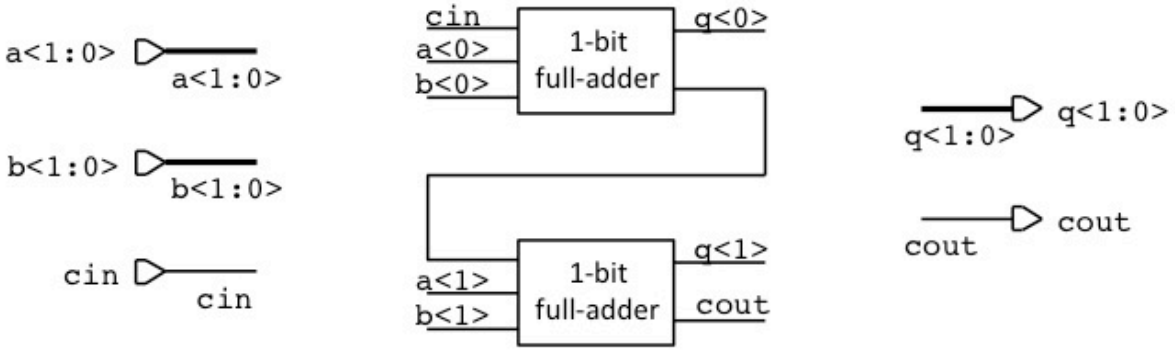


Figure 1.5: Use of wire names to infer connections between buses/wires

You can use wire names (Add ↵ Wire Name …) to signify connections between buses. 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.

Instantiating multiple 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 1.6), 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.

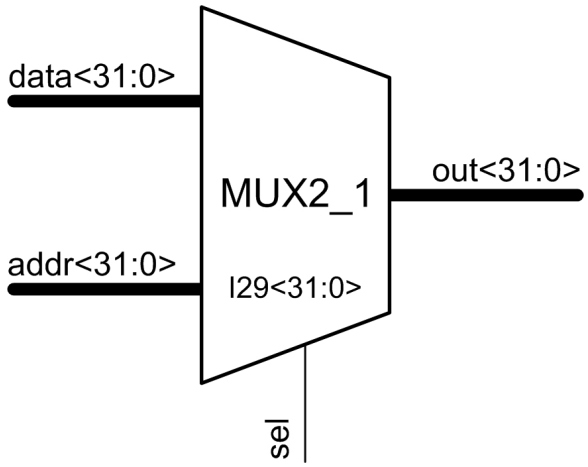


Figure 1.6: How to make a 32-bit 2:1 multiplexer

Jargon

The greatest difficult in entering schematics[4] 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**.

In addition, the following Cadence Design Architect default colours are useful to know:

Library components are green.

Any items that are currently selected are outlined in white.

[4] … or most other subjects

Library Elements Available

All the designs in this laboratory are built from library elements. Most of these are held in a directory that is available as "**spartan3**"; 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

Verilog: a Brief Introduction

Verilog is a Hardware Description Language (HDL). Its initial use in this laboratory is initially to provide a **simulation** environment to control and test schematic designs. Later we will use it to describe real hardware and **synthesize** it 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[5] 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. 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.

Although the first few 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 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 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 above 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`)

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

- 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`)

```
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

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. 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:

```
‘b0110_0011      // Binary
‘h63             // Hexadecimal
’d99             // Decimal
```

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

```
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

When simulating a clocked circuit it is possible to use a stimulus file given in figure 2.3.

```
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

However, this may get a bit tedious as the extent of the simulation grows though. 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 statements. The first `initial` simply initialises the clock to zero (otherwise it would be undefined) at time 0s. The "`always`" statement loops, repeatedly 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. The last `initial` statement contains the various input values that have been set to change every 100 ns, i.e. every clock period. It can now assume that the clock is running in the 'background'. The first "`$stop;`" statement encountered (there is only one in this example) will stop the whole simulation. The effect of this file is shown in figure 2.5.

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

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 activating a reset – which is not shown on the symbols – for the first 100 ns 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.

```
initial Clk= 0;

always
  begin
  #50
  Clk = ~Clk;
  end

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

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

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

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

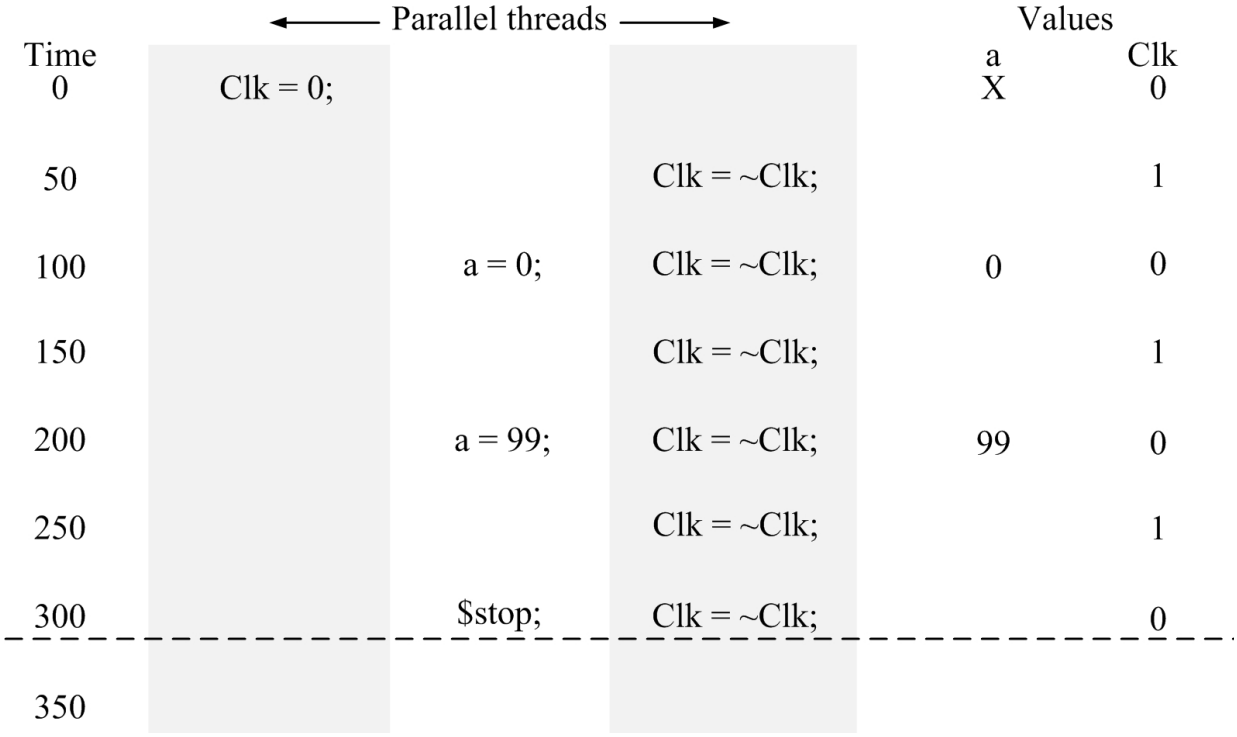


Figure 2.5: Effect of parallel threads

Synthesizable Verilog

Although so far we have only looked at using Verilog for producing stimulus files for simulation, it 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.

An example module syntax for a modulo-7 counter is given in figure 2.6.

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

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

assign zero = (count == 0);

endmodule
```

Figure 2.6: Verilog modulo-7 counter module

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

initial count = 0;

always @ (posedge clock)
…
endmodule
```

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

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. Hence for simulation purposes, we will have to initialise the value for count using an `initial` statement, as shown in figure 2.7.

In figure 2.7 the `initial` statement will be ignored when synthesized as it doesn't do anything. However, it is important that `count` is initialised to a value otherwise the simulation will not work as expected. (When using the Xilinx tools the value will always be initialised to zero.)

Logic

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

Simple combinatorial logic

Use continuous assignment instead of gates, 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.

```
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

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.

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, hence `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

This is more detail than needed for this course, but Verilog modules can be nested, as illustrated in figure 2.11. Here, we instantiate a instance (called "instance_name") of the module "fred" to be used in another design.

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

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

Figure 2.11: Nested Verilog modules

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.

Exercise 1: Simple Logic Design

Aim

This exercise is an introduction to the basic building blocks used in digital systems - logic gates - and introduces some of the conventions that may be used to describe digital systems. Simple circuits are used as illustrations of both **combinatorial** logic and **storage** elements. You will need to apply some of the concepts you have been introduced to in lectures, such as DeMorgan's theorem.

NOTE: You do not need a computer for this exercise, just a patch board to build you circuits on using the wires provided!

Preparation

Since this exercise is designed as an introductory exercise, no written preparation is needed. However, you should read the exercise script and the sections of this manual on Boolean logic identities. You should also ensure that you understand the conventions used for describing digital logic.

Duration

You have two lab sessions to complete Exercise 1.

The submission deadline is 11pm on the day of your scheduled lab in week 1.3.

Deliverables

You will need a copy of the answer sheet for this exercise, these are available in the laboratory, or can be downloaded from the course unit website. Complete the answer sheet and have exercises signed off by a demonstrator where required – failure to have designs signed off as working will result in marks being lost.

Once you have completed the exercise complete a blue card (available in the lab) and attach to your answer sheet and post in the "post-box" at the front of the lab. Don't forget to attach any circuit diagrams where requested.

Feedback

You should receive feedback for this lab exercise (in the form of a marked answer sheet) by the lab of week 1.6.

Assessment

This exercise counts 30 marks towards the total of 200 marks available for the lab (15%).

The Laboratory Patch Boards

The first lab exercise is concerned with building circuits out of their basic elements. These basic elements are called gates and have a set of one or more inputs and a single output. The most common type of gates have their own symbols so that they can be recognised easily.

For the first exercise all the required gates are provided on the patch board provided in the laboratory. Figure 3.1 illustrates the layout of the patch board and identifies the logic gates available. The patch board conceals the 'dirty details', such as the need for power supplies, and displays the state of each input and output connection on a small light emitting diode (LED), which when lit represents a logic '1', and when not lit represents a logic '0'.

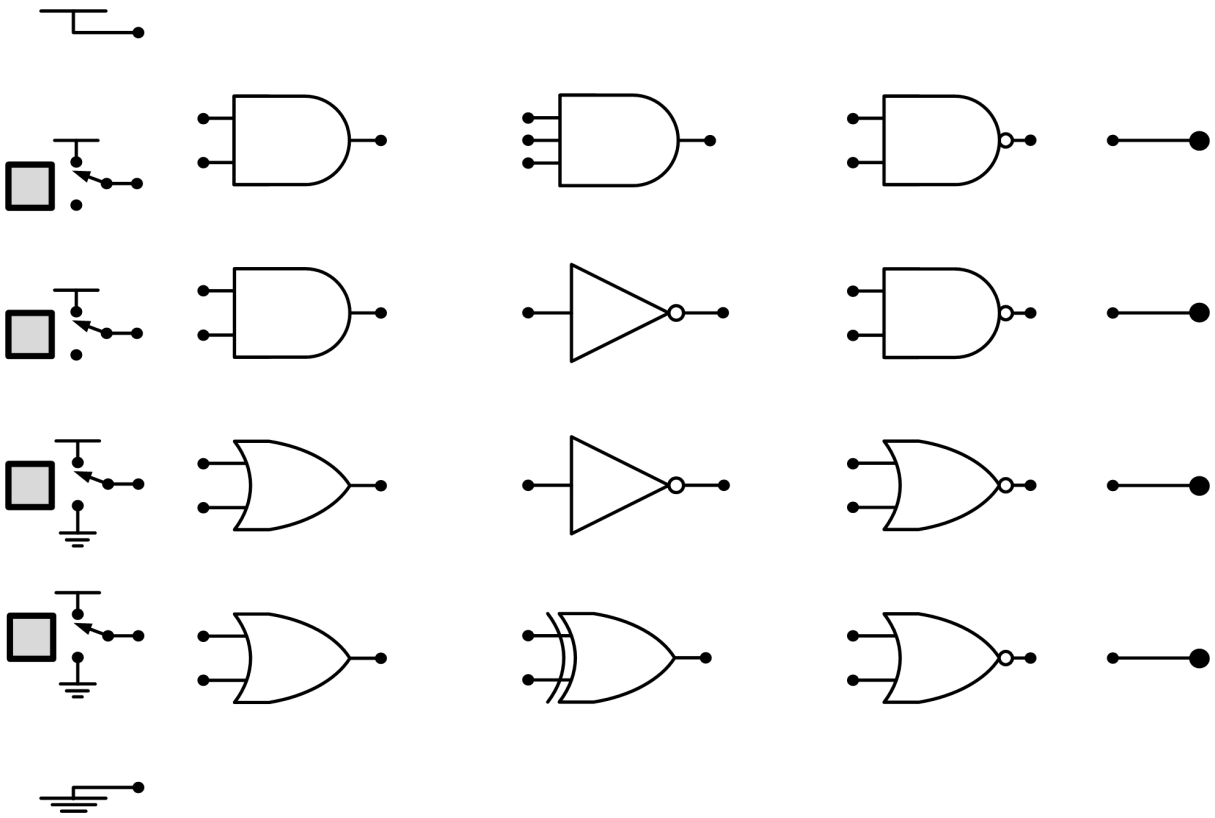


Figure 3.1: Laboratory Patch Board

Circuits are constructed by connecting the inputs (from the switches to the left) to gates using the wires provided. More complex logic functions can be implemented by connecting gates together. The state of the output of an implemented logic function can be view by using the output LEDs to the right of the patch board. Unused inputs are held inactive on the patch board, but - in general - unused inputs should be tied to a logic level – VCC – '1' – top left of the patch board, or GND – '0' – bottom left of the patch board – depending upon the logic function being implemented.

At the left-hand size of the board are four push buttons. The upper two switches are **momentary action**, whilst the lower two are **latching**. The switches are used to select the 'state' of the input binary signals.

Digital Circuits

Digital electrical circuits possess two well-defined states. There are (at least) three ways of viewing these states; these are as:

Boolean logic values with the two states equating to "TRUE" and "FALSE", abbreviated to "T" and "F" respectively.

Binary arithmetic values with each digital signal being a single base two digit with a value of "1" or "0" – "1' generally represents TRUE, '0' – generally represents FALSE – this is the convention we will be adopting in the laboratory.

Voltage levels with the values 'high' ("H") voltage and 'low' ("L") voltage.

Generally, T = 1 = H and F = 0 = L, this is called positive, or active-high, logic. Note that sometimes the terms **active** or **asserted** are used for "TRUE" and the terms **inactive** or **not asserted** are used for "FALSE".

A single digital signal is not capable of anything that is useful on its own. To do useful things two or more signals must be combined. In digital electronics simple combinations are performed by logic **gates**, while more complex functions can be built by combining gates into **circuits**. There is a wide variety of ways in which signals can be combined, only a few of which will be encountered during the course of this laboratory.

Basic Logic Functions

Three basic logic functions are introduced here, although these may already be familiar. They are called "**AND**", "**OR**" and "**NOT**".

"AND" and "OR" are methods of combining two (or more) logical inputs to implement a logical output that conforms to the required Boolean function. In the case of an AND function the output is 'true' if, and only if, the first input is 'true' and the second input is 'true' (and all other inputs are true, if they exist). Conversely the OR function is 'true' if the first input is 'true' or the second input is 'true' (or any of the other inputs are true, if they exist).

The NOT function has only a single input; it is 'true' if the input is not 'true' (i.e. 'false') and vice versa. With these three simple functions any computer can be built.

Representation of Digital Functions - Truth Tables

A truth table is a tabular means of representing the value of the output of a function for all possible combinations of its inputs. The table is split in two; the left-hand side enumerates all possible input values, and the right-hand side defines the output values for the input conditions listed on the same row. For example, two truth tables for a 2-input AND gate are shown in figure 3.2. The logical inputs to the function are labelled A and B and the logical output is labelled C – you can use any name or symbol to represent your input and output signals. There is no functional difference between the two truth tables shown, the only difference is that one uses a Boolean logic representation, T and F, while the other uses a binary arithmetic representation, 0 and 1. In this lab we will be using the binary arithmetic representation.

Inputs		Output
A	B	C
F	F	F
F	T	F
T	F	F
T	T	T

Inputs		Output
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Figure 3.2: Truth tables for 2-input AND gate

Truth tables are useful for defining a function exactly. However, for even small circuits the number of inputs can be sufficient to need a large truth table to list **ALL** permutations of the inputs; each extra input doubles the size of the table. A possible way to overcome this problem is to use a representation method that omits some of the detail. Representations that do this include logic diagrams and logic equations. In both of these methods details are omitted by only showing/naming functions without enumerating their details. Note: a one-to-one correspondence should exist between any logic diagram and logic equation representation of a circuit.

Representation of Digital Circuits - Logic Diagrams/Schematics

When drawing logic diagrams (schematics), signal interactions are represented by symbols that indicate the function performed. The symbols illustrate connections for all the input and output signals. For complex functions these are often just rectangular boxes with the name of the function appended for identification. However, simple gates such as AND and OR are so common that they have their own, standard, readily recognisable symbols, as shown in Figure 3.3.

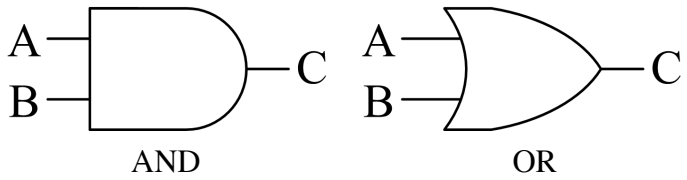


Figure 3.3: ISO[1] Symbols for AND and OR functions

Whenever these symbols appear, then a logical AND or a logical OR function has been implemented. We don't really care how the function is implemented inside the symbol – this is unnecessary detail and depends on many factors, such as the technology used (abstraction!) – we are simply interested in the logic function it will perform. The figure illustrates two-input gates where in each case the inputs are labelled "A" and "B", and the output is labelled "C" (these letters are just names for the signals; they could be called anything). Gates with more than two inputs are also possible.

The inverter or "NOT" gate is a device that swaps a signal from one state to the other, thus a 'TRUE' becomes NOT('TRUE') (or 'FALSE') or - interpreting the signals using the arithmetic representation - a '1' becomes NOT('1'), or '0', and a '0' becomes NOT('0'), or '1'. The usual symbol for an inverter is shown in figure 3.4.

[1] ISO – International Organization for Standardization (http://www.iso.org)

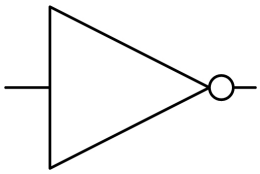


Figure 3.4: The Inverter

The ‘bubble’ (the little circle at the output) in this symbol is generally used to indicate an inversion.

Sometimes gates perform both a logical function and an inversion; thus as an example there is a ‘NAND’ gate (an abbreviation of ‘Not AND’) which performs the same function as an ‘AND’ gate followed by inversion, ‘NOT’, as illustrated in Figure 3.5.

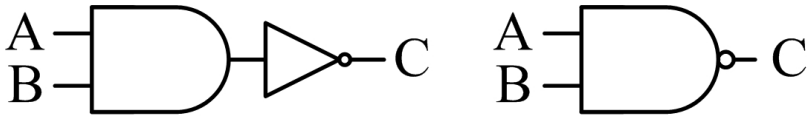


Figure 3.5: Function and ISO symbol for a NAND gate

Similarly a ‘NOR’ gate can be defined. What do you think the symbol for this could be?

Connections between inputs and outputs of gates can be represented by lines on the circuit diagram connecting inputs and outputs of gates together – this allows us to produce more complex logical functions. Sometimes it is necessary to be able to identify connections, the easiest way of doing this is to name the connection. Drawing all the lines necessary to show all the connections on a logic diagram can produce a very messy diagram. One way to overcome this is the name the connection and omit the line that explicitly shows it. In this case the signal name is placed on the diagram adjacent to any device inputs or outputs to which it connects. Any inputs or outputs identified by a common name are assumed to be electrically connected.

Representation of Digital Functions – Logic Equations

Another method of representing logic is to write out the Boolean algebra that represents the function. This is a very compact method of representing a logic function.

The basic symbols are:

A . B	for	A AND B
A + B	for	A OR B
$\overline{A}$	for	NOT(A) (or sometimes A’)

The rules of Boolean algebra are similar to normal algebra; AND functions are evaluated before OR functions, and parentheses, “(“ and “)”, may be used to specify evaluation order. Sometimes the NOT ‘overline’ is used to encompass part or all of an equation, thus:

$\overline{A.B}$	means	NOT(A AND B)

Some rules of Boolean algebra are included in the section ‘Boolean Logic Identities’ at the end of this manual.

Practical

Simple Gates

This exercise is marked on the basis of the pro-forma answer sheet provided in the laboratory (Tootill 1). Fill in the sheet as you complete each part of the exercise, ensuring as you do so that you demonstrate your working circuits to a lab demonstrator, who will sign off the relevant sections in your answer sheet to confirm your circuit works. When you have completed all parts of this exercise attach a blue card – which can be found at the front of the laboratory – to your answer sheet, making sure you fill in the required details – name, lab group, exercise number, and date submitted. Post your answer sheet in the "post-box" at the front of the laboratory. Failure to attach this card will result in your work not being marked.

This exercise is performed on the laboratory patch boards, described in an earlier section.

1. **Identify a 'NOR' gate on the patch board and verify its behaviour. Complete the truth table on your answer sheet.**

   Q1.1 What is the Boolean expression for the NOR gate without using a OR expression (+).

De Morgan's Theorem

De Morgan's theorem allows AND functions to be translated into OR functions and vice versa.

Briefly it states that:

$$\overline{A.B} = \overline{A} + \overline{B}$$

so the NAND (AND with output inverted) can be expressed as an OR gate with the inputs inverted, i.e.

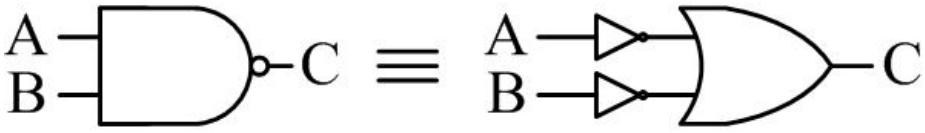


Figure 3.6: NAND or OR relationship using De Morgan's theroem

In addition, it states that a NOR gate can be implemented using AND gates:

$$\overline{A+B} = ?$$

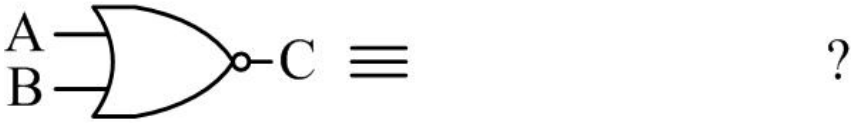


Figure 3.7: NOR or AND relationship using De Morgan's theroem

What do you think the NOR representation using AND gates is? You could take a guess from looking at the NAND example.

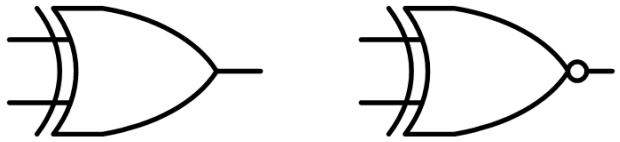
Practical

Simple Gates continued

2. **Construct and test a NOR gate on the patch board. You MUST NOT USE a NOR or an OR component.**

   Sketch your circuit on your answer sheet, illustrating your IMPLEMENTED circuit, i.e. identifying the actual gates on the patch board you have used to implement your design.

Figure 3.8 illustrates the symbol for the exclusive-OR (XOR) and exclusive-NOR (XNOR) gates. An exclusive- OR gate is similar to an (inclusive) OR gate, but it 'excludes' its output from being true when both of the inputs are true. The XOR gate has its own symbol: ⊕.



Exclusive-OR (XOR) Exclusive-NOR (XNOR)

Figure 3.8: The symbols for the exclusive-OR and exclusive-NOT gates

Practical

Simple Gates continued

3. **Identify the XOR gate on the patch board and investigate its behaviour. Complete the truth table for the XOR gate on your answer sheet.**

   Q3.1 Write down the Boolean expression for the XOR gate using AND (.) and OR (+) symbols.

   Q3.2 The XOR and XNOR gates are known as "not-equivalent" and "equivalent" gates. Explain why.

4. **There are various methods of constructing XOR gates from the basic gates: the inverter, AND, and OR. Construct an XOR gates using logic gates other than the XOR gate on the patch board.**

   Sketch your circuit on your answer sheet, illustrating your IMPLEMENTED circuit, i.e. identifying the actual gates on the patch board you have used to implement your design.

   Show a demonstrator your working XOR gate and have your answer sheet signed off.

A Simple Binary Adder

As an example of the use of logic circuits in computers consider the addition of two binary numbers.

If we have two single bit variables (let's call them A and B) and add them together we will produce a 2-bit sum. The sum can take a value between 0 and 2, so we need two bits to represent this, a sum bit (called S this example) and a carry bit (called C_o in this example). This can be seen in the truth table in figure 3.9.

It is clear that simple Boolean relationships between the variables A and B can be used to determine the sum, S, and carry, C_o, signals. A simple circuit that encapsulates this behaviour is the **half-adder**.

A	B	S	C_o
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 3.9: Truth table for a half-adder

Practical

Half-adder

5. **Build, document and test a half-adder circuit using the gates available on the patch board.**

   Sketch your circuit on your answer sheet, illustrating your IMPLEMENTED circuit, i.e. identifying the actual gates on the patch board you have used to implement your design.

   Q5.1 Write down Boolean expressions for the sum and carry signals for the half-adder.

In order to add operands with more than one bit, i.e. two *n*-bit binary numbers, you must take into account carries between bit positions; this involves the addition of a carry in on top of the addition of the pairs of digits. So you must add pairs of bits from the two numbers a bit at a time starting from the least significant bit (lsb) to the right. For each bit you must take into account the two binary values you are adding along with a carry in from the previous bit calculation to produce a binary sum value and a binary carry out (to the next bit). For example, consider the sum

$$\begin{array}{r} 0\,1\,1\,1 \\ +\,0\,1\,1\,0 \end{array}$$

How do we calculate the final sum?

We calculate the sum and carry components starting from the lsb (bit 0):

- 1 + 0 (+ carry in of 0 to the lsb) gives a sum of 1 and a carry into the next bit of 0.
- 1 + 1 + 0 (carry in from the previous bit) gives a sum of 0 and a carry into the next bit of 1
- 1 + 1 + 1 (carry in from the previous bit) gives a sum of 1 and a carry into the next bit of 1
- 0 + 0 + 1 (carry in from the previous bit) gives a sum of 1 and a carry out of 0.

Hence:

```
  0 1 1 1
+ 0 1 1 0
  1 1 0 1      Sum components
0 1 1 0        Carry components
```

We can encapsulate the behaviour of this addition process using a truth table to represent the sum and carry out values for all possible combinations of the two input bits and the carry in, as shown in figure 3.10. The logic function being implemented here is known as a **full-adder**.

Inputs			Outputs	
Digits		Carry	Sum	Carry
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 3.10: Truth table for a full-adder

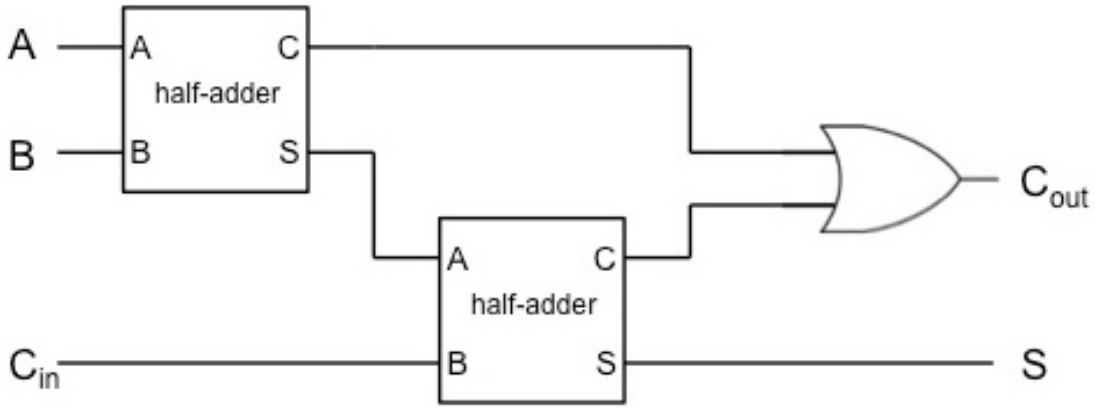


Figure 3.11: The full-adder circuit built from two half-adders and an OR gate

From the truth table of figure 3.10 it is possible to construct the logical equations needed to generate the functions S and C_{out}. However, since this is the first exercise a slightly easier approach will be presented. The three input, two output full adder circuit can be produced using two half-adders, as shown in figure 3.11.

Figure 3.11 also exhibits a characteristic common in engineering design, that of hierarchical structure. The circuit is built from 'black boxes' whose function and interfaces are known. These 'black boxes' are reused to generate a more complex circuit. This is similar to building software from functions and procedures, which are - in turn built from other procedures. This technique will be used extensively in later laboratories.

Practical

Full-adder

6. **Build, document and test a full-adder circuit from two half-adders using the patch board. There are enough gates for this on the patch board, although some ingenuity may be required.**

   Sketch your circuit on your answer sheet, illustrating your IMPLEMENTED circuit, i.e. identifying the actual gates on the patch board you have used to implement your design.

   Show a demonstrator your working full-adder circuit and have your answer sheet signed off.

Flip-Flops

The full-adder circuit is an example of purely **combinatorial** design. The values of the outputs of combinatorial circuits depend only on the instantaneous value of the inputs. Digital circuits may also be constructed where the value of the output depends on both the current and past values of the inputs. Such circuits are known as **sequential** logic circuits. These store information internally about their history; this is known as the **state** of the circuit.

The basic building block of a sequential circuit is the latch, or the flip-flop as it is often referred to. The latch is an edge triggered device, whereas a flip-flop is an edge-triggered device often controlled by an external clock signal. The rest of this exercise is an investigation of flip-flops and some of the ways in which they can be used.

Set-Reset (S-R) Latch

The S-R flip-flop (or R-S flip-flop as it is also known) is the simplest form of flip-flop. It has two control lines, *reset* and *set*, and two outputs, and can be constructed from NOR gates as shown in figure 3.12. The circuit emphasises the essential difference between sequential circuits and combinatorial circuits, that is, sequential circuits have a **feedback** path from output to input.

In this example the output changes when S or R are '1', this is called **active-high** since the action (setting or resetting the output) occurs when a '1', or high signal, is seen on one of the inputs.

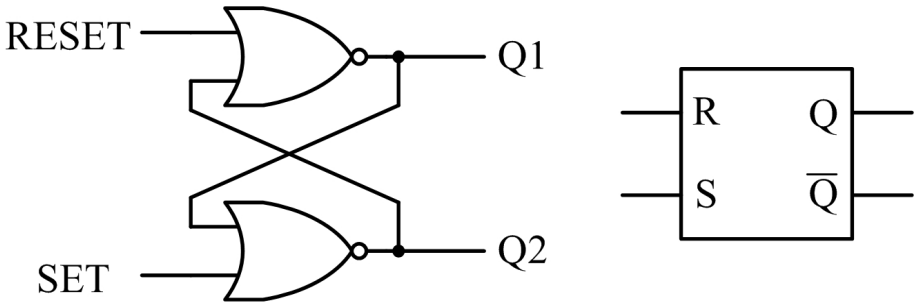


Figure 3.12: S-R latch: Circuit and symbol

Practical

Latches & Flip-flops

7. **Construct an S-R latch based on the circuit given in figure 3.12. Investigate the circuit's behaviour and complete the transition table in the answer sheet.**

   Q7.1. What effect does asserting the set and reset inputs independently have on the state of the outputs?

   Q7.2. What do you notice about the operation of the circuit when set and reset are high at the same time? Is this behaviour valid? Explain your answer.

   Q7.3. If I produced an identical circuit but with the NOR gates replaced by NAND gates, how would the flip-flop operate? Produce a transition table to illustrate the operation of the new design.

Gated S-R Latch

The circuit of figure 3.13 has an additional *enable* input so that the S and R inputs are active **only** when the *enable* is high. When the *enable* is low the state of the flip-flop is "latched" and cannot change until the *enable* goes high again. The addition of the *enable* input changes the R-S flip-flop from an element used in asynchronous systems to one useful in synchronous systems.

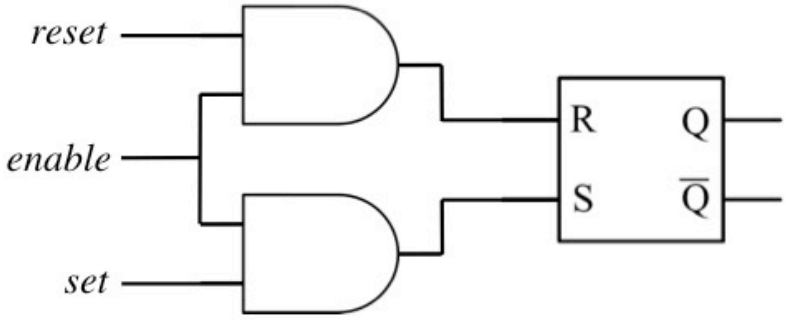


Figure 3.13: Clocked S-R latch circuit

enable	data	Q	$\bar{Q}$
1	0	0	1
1	1	1	0
0	x	last Q	last $\bar{Q}$

Figure 3.14: Flip-flop design transition table

Practical

Latches & Flip-flops continued

8. **Construct and test the latch design based on that given in figure 3.13, which has two inputs: an enable input, that only allows the state of the output of the flip-flop to change when it is high, and a data input that sets the output of the flip-flop to '0' if the data input is 0 (and enabled), or sets the output to '1' if the data input is '1' (and enabled). A transition table for the flip-flop is given in figure 3.14.**

   Sketch your circuit on your answer sheet, illustrating your IMPLEMENTED circuit, i.e. identifying the actual gates on the patch board you have used to implement your design.

   Show a demonstrator your working flip-flop circuit and have your answer sheet signed off.

Hand-in

Make sure you get your answer sheets signed by a demonstrator and hand them in, along with any required circuit diagrams.

Don't forget to fill in and attach a blue hand-in card with your answer sheet.

You have now completed Exercise 1.

Exercise 2: Binary Adders in Verilog

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, an aim is to emphasise logic hierarchy, as systems are built from "black boxes" which are in turn constructed from simple gates. The logical design itself should present no difficulties.

Preparation

This is the first use of the Cadence CAD 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 a design and test it to confirm it is working as expected. There are no marks associated with the tutorial; however, you must complete the tutorial as the half-adder design you design will be used later in the exercise.

Note that no physical wiring is needed for this or any of the other gate array exercises; the patch board and its contents is replaced either by a simulator or by a single integrated circuit.

Please note: 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.

Duration

You have two lab sessions to complete Exercise 2.

The deadline is 11pm on the day of your scheduled lab in week 1.5.

Deliverables

1) Submit your work via the script 12111submit before the deadline. There are three exercises that must be submitted individually: ex2a, ex2b, ex2c. Make sure you print out a copy of the marking sheet for exercise 2 before the demonstration lab.
2) In your next scheduled lab arrange a demonstration session by writing your name and machine number of the whiteboard.

Feedback

You will receive your automated feedback at the end of week 1.7 after the extended deadline has passed.

Assessment

This exercise counts 30 marks towards the total of 200 marks available for the lab (15%), 20 marks from the automated marking and 10 marks from the demonstration.

Cadence Tools Tutorial

In this tutorial you will be introduced to the Cadence CAD design suite. It is not assessed, but we recommend you work through the tutorial in order to familiarise yourself with the software tools you will be using in the later exercises. This software can only be used in an "X windows" environment. The various design, simulation and run tools do have different names, but in the following sections the tools will usually be referred to generically as "Cadence" for convenience.

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 the Appendix. When you run the Cadence tools for the first time you will be asked to confirm that you agree to the conditions set out by the end user agreement; failure to do so will result in you not being allowed access to the tools.

Software Setup

Before Cadence can be run for the first time its directory structure must be created and certain system variables must be set up. Most of these are performed automatically by running the script:

```
mk_cadence COMP12111 <return>
```

This will create a directory, ~/Cadence, and the project directory required for this course is contained therein.

WARNING! DO NOT do anything to the files/directories in this directory; all these files are managed by Cadence – doing so may cause you problems when using your design in Cadence! You have been warned.

ANOTHER WARNING! DO NOT run this command again or it will erase all your hard work! You have been warned – again!

Creating a Design

Starting Cadence

Start Cadence by typing

```
start_cadence COMP12111 <return>
```

in a command shell window. (You will have to agree to the EUA before you proceed for the first time.) The Cadence Integrated Circuit Design and Simulation window (*icds*) will eventually start, as shown in figure 4.1. 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, but you will not normally need to do that.

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 4.1).

Notes:

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.

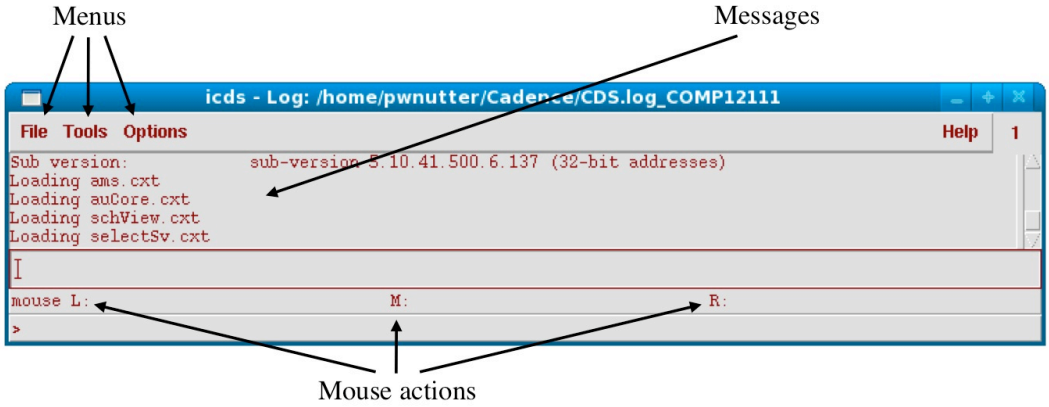


Figure 4.1: Cadence icds

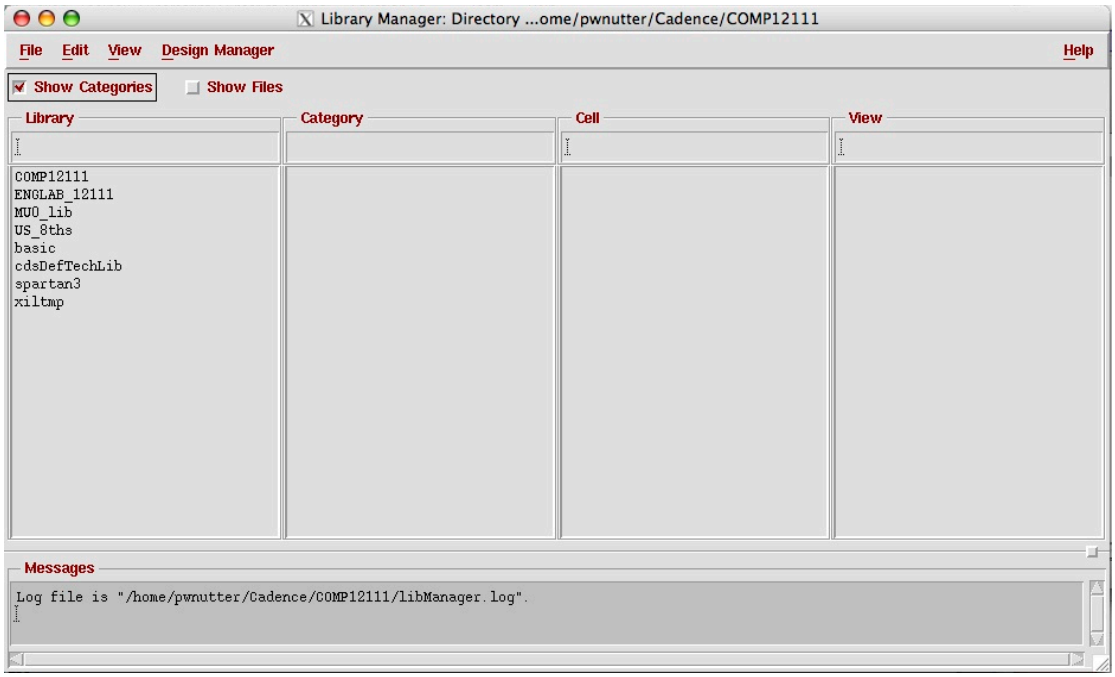


Figure 4.2: The Cadence Library Manager window

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 *icds* (Tools ↵ Library Manager ….) and is shown in figure 4.2. So please make sure you use it to open designs.

Designs are identified as cells in a particular library. You will place you designs in the library COMP12111, although you will use designs from the spartan3 library 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.

Closing Down

When you have finished, close down Cadence before logging off. If you simply close down your X windows session without closely Cadence down properly it can leave files locked which will cause you problems in your next session. You can close the session tidily from icds with File ↵ Exit ….

If you have problems running Cadence and the *icds* window fails to display, then it is likely you have exited Cadence incorrectly. To enable Cadence to start, you must check the /Cadence directory and delete the file CDS.log_COMP12111.cdslck if present (i.e. rm CDS.log_COMP12111.cdslck). This should enable Cadence to run. However, you may also have problems opening any schematics for editing that may have been open at the time Cadence was not shut down properly. In this case Cadence will ask whether you want to open the files as read only and will not let you edit them. If this occurs ask a demonstrator to delete the lock file – DO NOT attempt to do it yourself!

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.

Creating a Component and Editing the Schematic

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. In this tutorial you will enter and simulate a new design.

The first thing to do is to open a design using the Cadence Library Manager (in icds click Tools ↵ Library Manager…), select the Cell name "**halfadder**" 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 4.3, in which you can create your design.

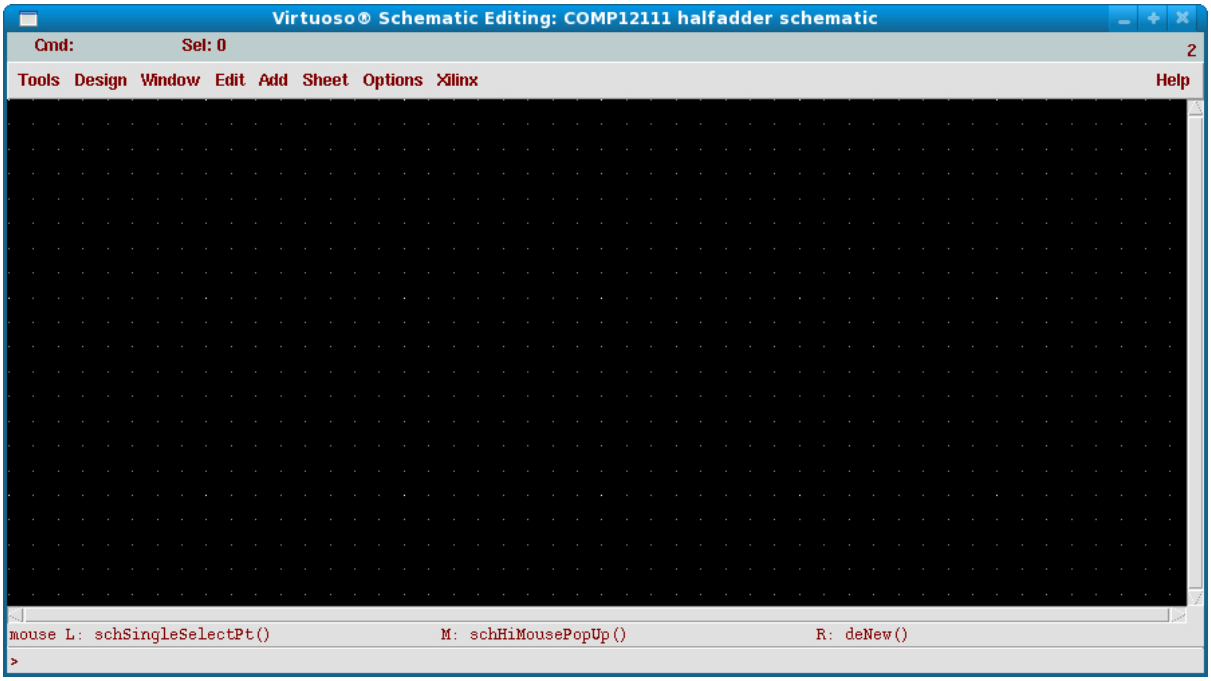


Figure 4.3: Virtuoso schematic editor

You are going to design and test a half-adder, the circuit for which is shown in Figure 4.4.

The first thing to do is place some components (in this case logic gates). 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: Add ↵ 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 spartan3 library (click on the name spartan3 in the "Library" column), which should produce a display of the list of available gates in the "Cell" column, as shown in Figure 4.5.

A list of gates should appear in the "Cell" column of the Library Browser window. This will need to be scrolled to gain access to all the components.

You may also see 'Categories'; these are classifications, for example grouping all the AND gates together.

1. Select the component AND2 (an AND gate with 2 inputs) using the LMB and place this component 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.

   Cadence will have placed one AND gate, and will now be ready to place another. Move the cursor and then place another AND2 gate. You could continue placing AND gates if more were needed, but in this case you only want two. Press the 'Esc' key to stop placing AND2 gates.

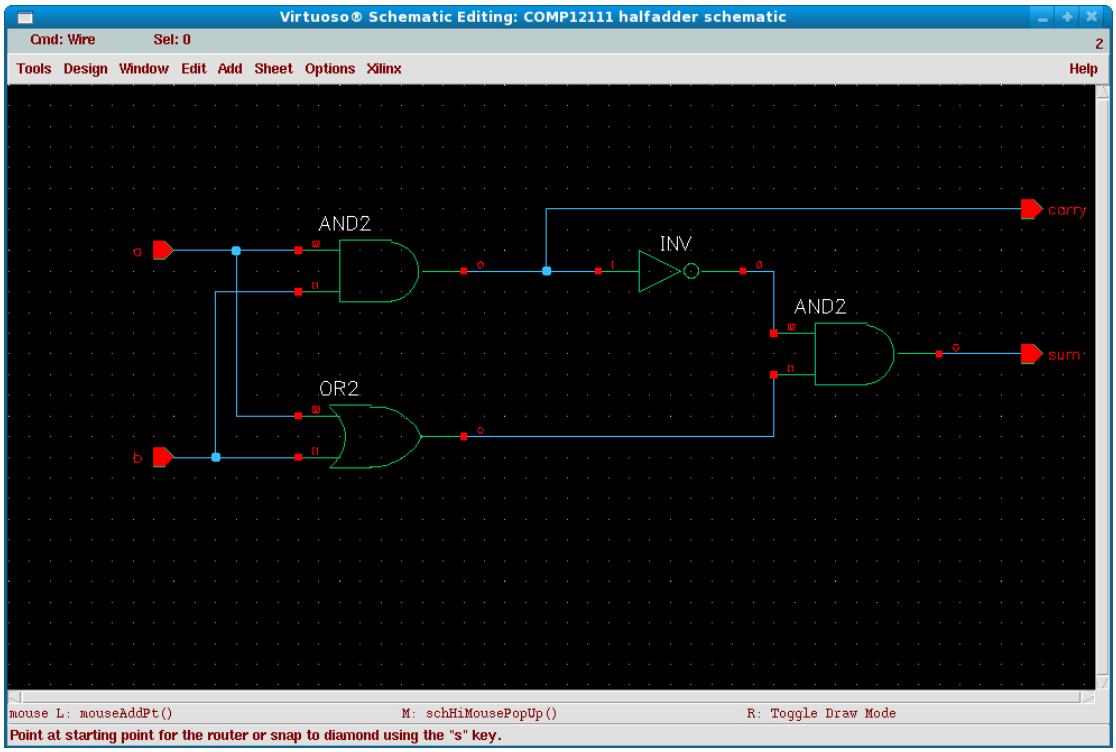


Figure 4.4: The half-adder schematic

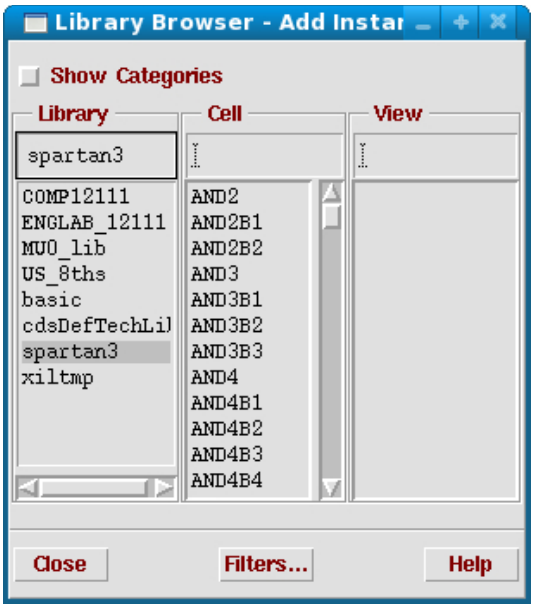


Figure 4.5: spartan3 library components listed in the library browser window

2. Add components OR2 and INV, placing them in the positions shown in Figure 4.4 (you haven't placed the inputs, outputs or wires yet). You may need to zoom out to do this sensibly; the view control functions are accessible via the Window ↵ Zoom or Window ↵ Pan submenus. **Follow any instructions** that appear at the bottom of the Virtuoso Schematic Editing window.

   Components can be moved when they are selected (white). To move a component, select it (LMB) and then select MMB ↵ stretch. 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.

Top tip: You can unselect everything by clicking on the background of the schematic, and you can select everything in a region by clicking and dragging the LMB.

3. Now you are going to add the inputs and outputs to the halfadder. Inputs and outputs are called 'pins'. Select Add ↵ 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 (on the keyboard). Place the two pins one after another, approximately as illustrated in Figure 4.4. Each name you place in the list will produce a pin of the appropriate function, which you place on the schematic one after another.

Top tip: 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.

Note – in this exercise we are asking 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.

4. Follow the same procedure to place two **output** pins, 'carry' and 'sum', remember to select the pin direction as 'output'.
5. Now the components are positioned correctly they must be joined with wires. Wires must run connections to pins on components (red squares), to input or output pins or to other wires. Select Add ↵ Wire (narrow), or press 'w' on the keyboard, and then place wires to connect the components as shown in Figure 4.4. Point at the start position of the wire, click the LMB to start a wire, click once at each place where the wire changes direction

and at the end of the wire. If you wish to end a wire without connecting it, double click the LMB the end point.

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'. If you take the trouble to find the 'Move' operation, you will find that connectivity is broken. For legibility it is a good idea to try to maintain most wires with "Manhattan" geometry (i.e. keep segments either vertical or horizontal, as in Figure 4.4).

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 the schematic design of the half-adder. Save the schematic: Design ↵ Check and Save, or 'X'.

   Look in the *icds* window and confirm that the design checked and saved correctly – you should see messages saying:

   > Schematic check completed with no errors.
   >
   > "COMP12111 halfadder schematic" saved

   If not, there may be something wrong with your design. Look at it again; potential problems will be highlighted on the schematic, and there will be error messages in the *icds*. If you can't see the fault, ask a demonstrator 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).

Symbol Design

You are now going to build a symbol for the half-adder. The symbol is the object that will appear when you wish to place a half-adder in a more complex system, such as a full-adder (which can be made using two half-adders).

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 Design ↵ 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 4.6.

@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 (e.g. in a full adder design). You might choose to modify this by replacing “@partName” with “Half Adder”.

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.

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.

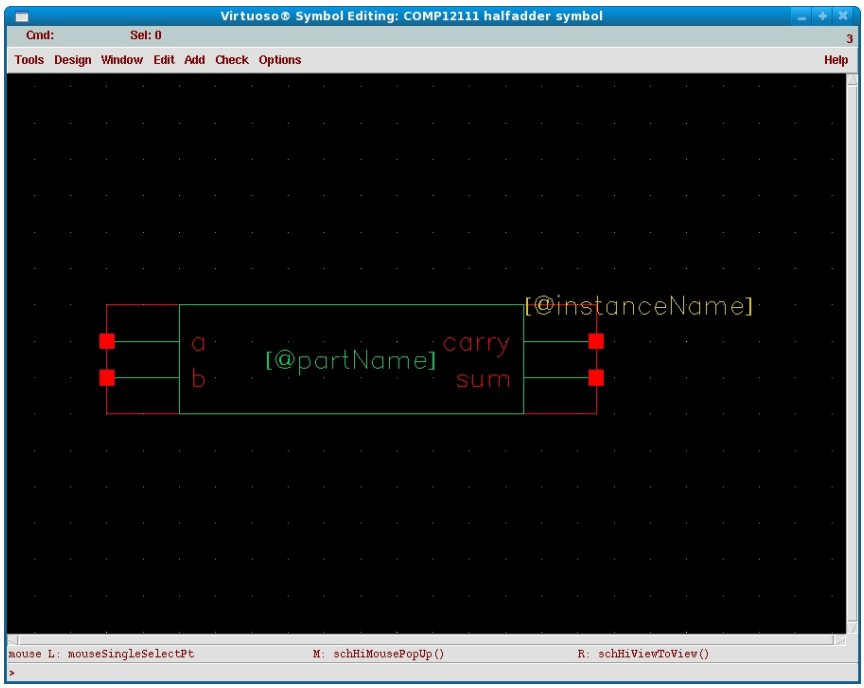


Figure 4.6: A possible symbol for a half-adder

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

   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 would be ***very unwise***. To test components they must be simulated to make sure they meet their specification. **Simulation and testing is a very important part of the design process, you should make sure you always simulate your designs to check they are working correctly.**

   You did write a specification before you started designing, didn’t you? Answer: “Yes, I completed a truth table for a half-adder”.

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 a specification, which should be created before the design begins. In the case of the half-adder the specification is encapsulated in the truth table, 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.

1) Starting the Simulator and preparing the design for simulation

1. Open the schematic design of the system you want to simulate (the half-adder 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 4.7.
3. Make sure that the correct "Cell" and "View" are selected. In this case the "cell" is "halfadder" and the "View" is "schematic". The run directory, which is used to store simulation files will be set for you.
4. Click the 'Initialize Design' button and then the 'Generate Netlist' button. You may see a popup window asking if it is OK to continue, if so click 'Yes' or 'No'.

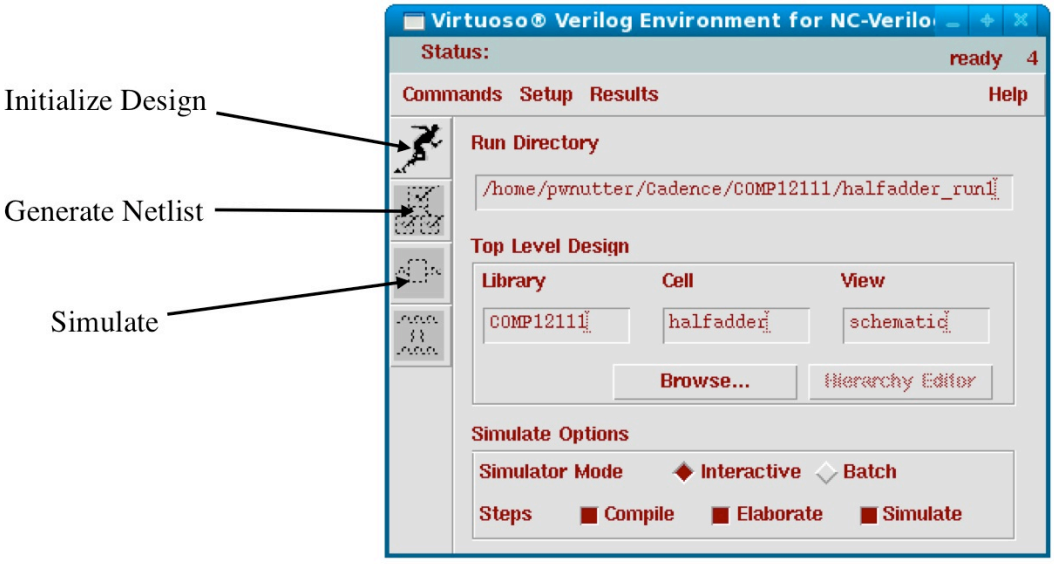


Figure 4.7: Virtuoso simulation window

2) Telling the simulator what input values to drive your design with.

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.

1. Select Commands ↵ Edit Test Fixture. This will first open an "Edit Fixture" window, with file name "… /stimulus.v", select OK. A text editor window will open to allow you

to edit the file named 'stimulus.v', in which you can place code to describe the sequence of inputs that you want to drive the half-adder with. **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 (for example `http://www.asic-world.com/verilog/veritut.html`), but for this lab. you only need a very limited subset of the language (two operations).

```
// Basic half adder test
// Inputs a,b
// Outputs sum, carry

// Initialise inputs to zero
     a = 0;
     b = 0;
     #20  // Wait for 10ns to allow the adder to settle

// Cycle through the other input combinations,
// pausing at each input state
     b = 1;
     #20
     b = 0;
     a = 1;
     #20
     b = 1;
     #20

$stop;    // Stop simulating
```

Figure 4.8: Commands to include in the half-adder Verilog script

`#x` is the instruction to wait for a fixed time (specified in ns, or 10^{-9}s) where `x` is an integer number, so to wait for 20ns the instruction is #20.

To force inputs (or wires) to particular values, equate the pin name to the desired value (0 or 1 are the possible values for logic gates). So `a=1;` will set the input '`a`' to the value '`1`'. To test the half-adder you want to cycle through each one of the possible input combinations, waiting for a time at each so that the resulting output can be observed. A sequence of Verilog instructions that achieves this is shown in Figure 4.8.

2. Save the file that you have edited.

3) Selecting the signals that you want to see.

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 locations in the design at which you want to observe the behaviour of the system. 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. This should (after some delay) open two windows, the "SimVision Design Browser" and the "SimVision Waveform Viewer" for the halfadder. Find the

"Design Browser", which should look as shown in Figure 4.9 (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 4.9 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.

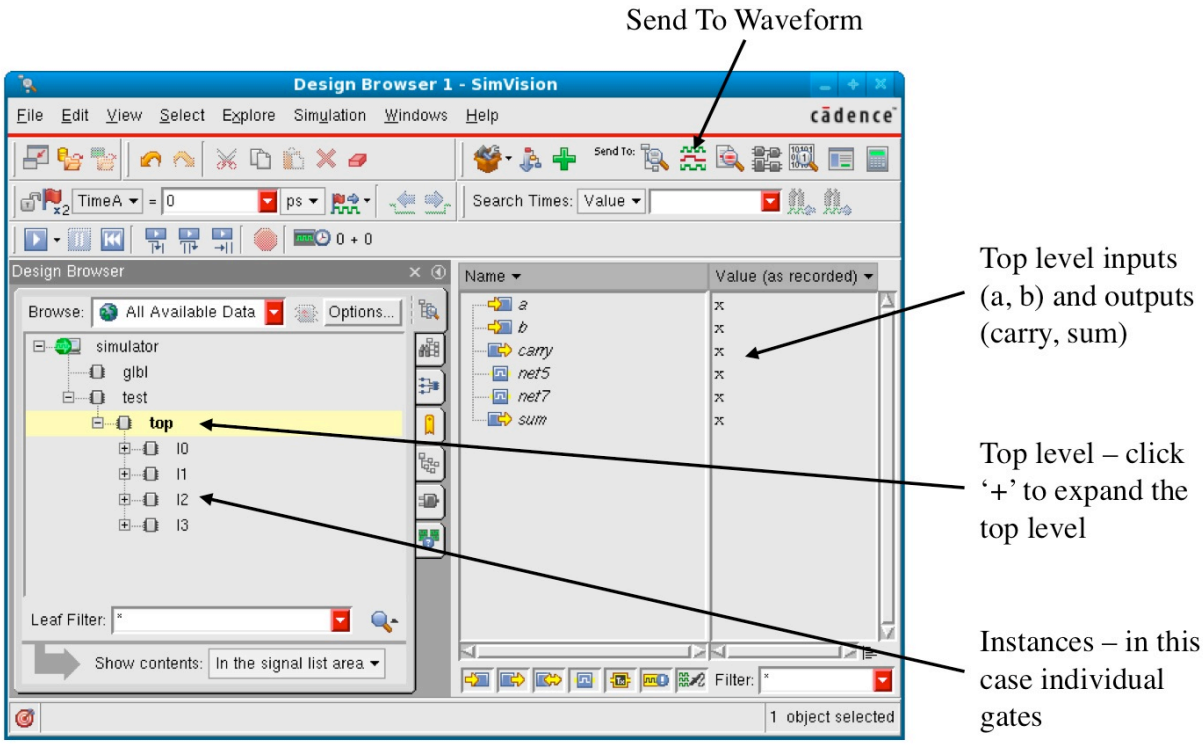


Figure 4.9: SimVision Design Browser

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 outputs of individual gates. For now you can just compare the inputs and outputs of the half-adder with the truth table. Select all of the inputs and all of the outputs of the top-level design (*a, b, sum, carry*) in the right hand pane. To select more than one input, hold down 'Ctrl' as you click.

4) Observing the behaviour of the circuit.

1. Click the "Send to Waveform" button (see figure 4.9).
2. Find SimVision waveform viewer, as shown in figure 4.10 (it should already be open).
3. When the SimVision window opens, there will be no traces visible. Click the "Run" button, as illustrated in figure 4.10, and the waveforms (or traces) should appear. 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.

Compare the simulated behaviour of your circuit before the outputs change. Does it perform as required? If not, do you know why? Can you fix it? There is nothing to hand in for this exercise as it isn't assessed. However, you will be using your half-adder design in the following exercise!

Notice that when the inputs change there is a delay before the outputs change. Why is this? Would that happen with real gates? At the last transition, when *a* is high ('1') and *b* changes from '0' to '1' the *carry* output changes to its final value significantly earlier than the sum output. Why is that?

The values of the sum and carry outputs are shown red and simultaneously '0' and '1' for a short time at the start of the simulation. Why is that?

To investigate the reasons for the different delays between the outputs, you need to be able to look at what is happening within the half-adder circuit, maybe at the inputs and outputs of individual gates.

Figure 4.10: SimVision waveform viewer

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 4.9). This would allow you to select any outputs you might be interested in and pass them to the waveform viewer. It is probably better to observe the status of wires than gate outputs, as described below.

You can identify specific wires (nets). For this you need to name the nets. Open the schematic of the half-adder and select a wire that doesn't already have a name (note that wires attached to pins, even indirectly, will already have adopted the pin name). Select the submenu Add ↵ Wire Name. This will cause an 'Add Wire Name' window to open. Name the wire in the 'Names' text entry box, and press enter (or return) on your keyboard. The text name for the wire should appear in gold on the screen, with a small gold square nearby. Move the text to a convenient position with the square over the wire you wish to name, and click the LMB.

1. Check and save the design in the schematic editor
2. Close the simulator.
3. Restart the simulator, and repeat the process described above, except that you should not need to recreate the stimulus file.
4. When simulating, add the outputs of appropriate gates to the waveform viewer, then Run the simulation again.
5. You can now trace the effect of changing the input b from '0' to '1' through the half-adder in the waveform viewer.

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**.

To exit Cadence select File ↵ Exit from the *icds* 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.

This section should have given a brief introduction to some of the capabilities (and quirks!) of the Cadence software. The descriptions are far from comprehensive; some more facilities are introduced in later sections and it is probable that you will discover other ways of using the software that suit you better.

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!).
- **Do not delete any of the code that is provided; just include your code in the appropriate place.**

Testing your Designs by Simulation

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 in the design of ANY system, 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 4.11. Input values are provided by means of a Verilog stimulus file – see later.

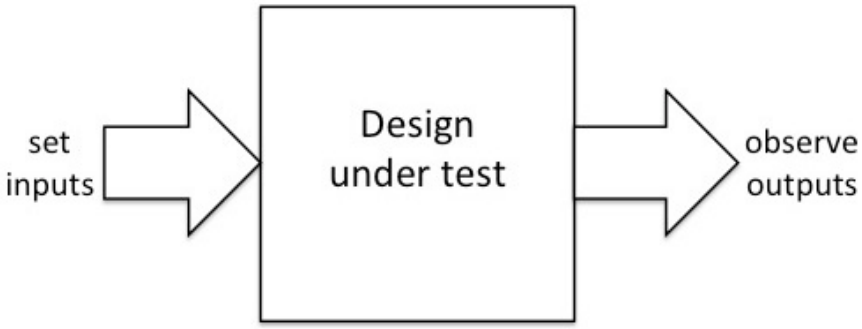


Figure 4.11: Design under test – set inputs and observe outputs

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.

As an example, consider the half-adder design that you have just completed. As it 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 in Figure 4.8. 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.

The next stage of the lab will involve you using your half-adder design to complete a design for a full-adder. Do you need to test your design exhaustively in this case? The answer is no. You are confident that your half-adder design works (as you have tested it exhaustively), so there is no need to test it again. You only need to ensure that the additions in your full-adder design work correctly. So in the case of the full-adder (see figure 4.4) you should be checking that when the sum output of the first (top) half-adder changes, it affects the output from the second (lower) half-adder (sum and carry out) in the correct manner. In addition, you should also check that the carry out of the first half-adder sets the carry out of the full-adder as expected.

You should ALWAYS test any of your designs using simulation as part of the overall design process.

Recap

For future reference:

To create a new design:

- From the *icds* menu select File ↵ New ↵ Cellview…; this will open the *Create New Component* window as shown in Figure 4.3.
- Select the library "COMP12111", type the name you want to use for the new component into the "Cell Name" text box and click 'OK'.

To create a symbol:

- In the Schematic Editor select Design ↵ Create Cellview ↵ From Cellview. This will open a "Cellview from Cellview" window.
- 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.

To simulate:

- Open the schematic design of the system you want to simulate.
- Select Xilinx ↵ Simulation. This will open the "Virtuoso Verilog Environment for NC Verilog Integration" window as shown in Figure 4.7.
- Make sure that the correct "Cell" and "View" for the design you re working on are selected. Select "Browse" if this isn't the case and select your design. Check tat the "run Directory" is set to the correct design (<*design*>_run1).
- Click the 'Initialize Design' button and then the 'Generate Netlist' button. You may see a popup window asking if it is OK to continue, if so click 'Yes' or 'No'.

3. Edit and produce a simulation file: select Commands ↵ Edit Test Fixture. Save the file that you have edited.

3. In the "Virtuoso Verilog Environment for NC Verilog Integration" window, click the "Simulate" button. This should (after some delay) open two windows, the SimVision Design Browser and the SimVision Waveform Viewer.

4. The Design Browser 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.

6. Select the signals you wish to view in the Waveform viewer. To select more than one input, hold down 'Ctrl' as you click. Click the "Send to Waveform" button.

7. In the SimVision waveform viewer, click the "Run" button and the waveforms (or traces) should appear.

Practical

ex2a: The Full-Adder

1) **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.**

   A full adder can be made from two half-adders and an OR gate as shown in figure 4.12. It has three inputs: *a*, *b*, and a carry in, *cin*, and two outputs: a sum, *s*, and a carry out, *cout*.

   a) **Use the half-adder symbol you produced in the Cadence tutorial to produce a full-adder circuit using the cell fulladder.**

   b) **Create a symbol for your full-adder.**

   c) **Simulate your full-adder and observe the output waveform to confirm it is working correctly. Remember to choose a suitable range of input values in your test stimulus to enable a complete test of the full-adder operation.**

   d) **Submit your design as ex2a.**

ex2b: A 4-bit Adder

2) **A multiple bit adder can be implemented by instantiating the required number of full-adders. Figure 4.13 illustrates the example of a 2-bit adder. Extend this to create a schematic for a 4-bit adder. A design adder_4bit has been provided for you for this exercise.**

   Notice that buses have been used in this design with inputs *a*<3:0> and *b*<3:0>, and output *s*<3:0>, each 4-bits wide. 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. You will have to use wire names (see section 5) in order to infer connections from the input/output buses to the individual input pins on your full-adder symbols.

   a) **Implement a 4-bit adder using your 1-bit adder design.**

   b) **Create a symbol for your 4-bit adder.**

   c) **Simulate your 4-bit adder and observe the output waveform to confirm it is working correctly. Ensure that you check a suitable number of input combinations to identify sensible operation of the adder. Including setting the value of the carry in.**

   d) **Submit your design as ex2b.**

The following question will require answering in your marking feedback sheet:

Q2.1 The ripple adder is so-called because its speed of operation is limited by the time it takes for the carry to ripple from the lsb to the msb. To determine this delay you would add F to 0, with a carry in of 1, this will result in the carry rippling from one bit to the next until it emerges from the msb. Using the tools determine the time it takes for the carry to ripple through in this case.

ex2c: A 16-bit Adder

3) Following the design of a 4-bit adder, you can now extend the design to produce a 16-bit adder.

a) Using your 4-bit adder design produce a 16-bit adder design. A cell adder_16bit has been provided for this exercise.

b) Create a symbol for your 16-bit adder.

c) Simulate your 16-bit adder and observe the output waveform to confirm it is working correctly. Ensure that you check a suitable number of input combinations to identify sensible operation of the adder.

d) Submit your design as ex2c.

The following questions will require answering in your marking feedback sheet:

Q3.1 Determine the ripple carry delay for your 16-bit adder. How does this compare to the delay through your 4-bit adder.

Q3.2 What modifications can be made to the design in order to produce a general 16-bit adder/subtractor unit? Think about this and briefly discuss it in your marking sheet.

Hand-in procedure

Run the submit command "12111submit" for each exercise (ex2a, ex2b, and ex2c) to submit your files for marking. Print out the demonstration marking sheet using the labprint option available under 12111submit.

Formal demonstration in the next laboratory

You have now completed Exercise 2.

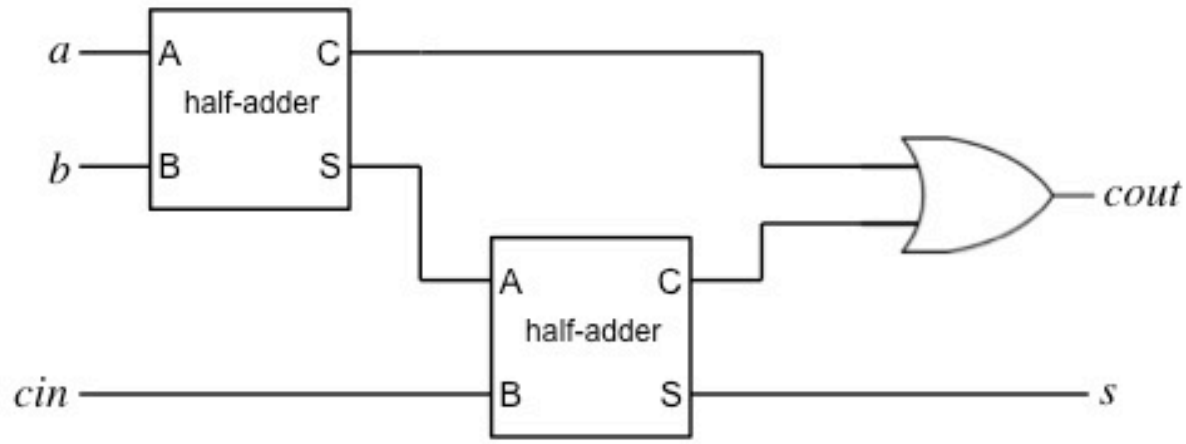


Figure 4.12: Full Adder built from two half-adders

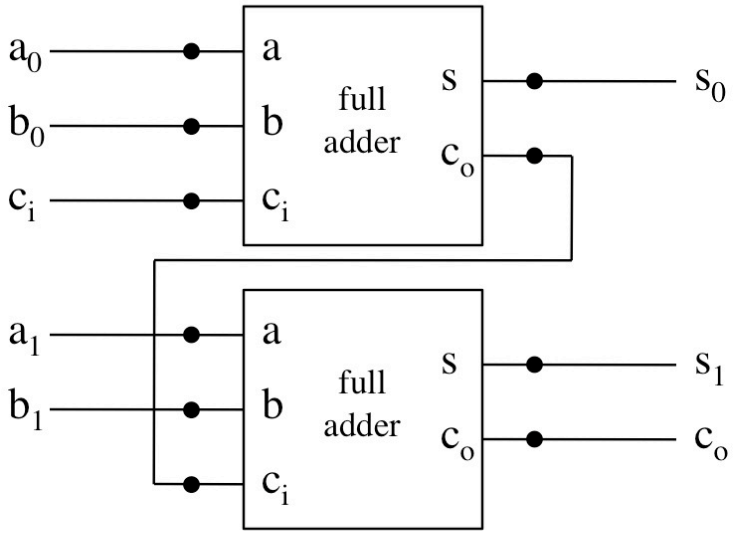


Figure 4.13: 2-bit adder

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 **Design ↵ 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 **Design ↵ 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.

Printing

To print an open schematic or symbol, select 'Design ↵ Plot ↵ Submit…'. This produces a dialogue box. In the bottom right is a button 'Plot Options', which opens another dialogue box with various options, which should be set as follows:

Plotter Name	**lfprt**
Paper Size	**A4**
Orientation	**automatic**
Mail Log To	*not specified*
Scale	*set as required*

Most of this information should already be correct. When it is correct, you can click 'Apply' and your settings will become defaults for that design. Click 'OK', on the 'Plot Options' box, and then click 'OK' on the 'Submit Plot' box to print.

Note: lfprt is any one of the printers in the resource centre. If you need to change the printer setup you can do this on the 'Plot Options' box.

To print waveforms from the simulator, display the required waveform in the waveform window, then in the waveform window select File ↵ Print Window. The default settings should be sensible, but check before you print, especially the destination printer, which should be *lfprt* to send to the least busy printer in the student resource centre.

Keyboard shortcuts in Cadence

You will have noticed that many menu and submenu items have single characters to their right. These are single keyboard keystrokes that will have the same effect. For example:

]	Zoom In by 2
[	Zoom Out by 2
w	Add Wire
p	Add Pin
i	Add Instance

… and many, many more. Build up your own list of useful shortcuts.

Exercise 3: The Seven Segment Decoder

Aim

To design a seven segment decoder in Verilog and test its operation on 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 an implementation on an FPGA using the experimental boards available in the laboratory.

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.

Duration

You have one lab session (but two weeks due to reading week) to complete Exercise 3.

The deadline is 11pm on the day of your scheduled lab in week 1.7.

Deliverables

1) Submit your work via the script 12111submit before the deadline. There is one exercise that must be submitted: ex3. Make sure you print out a copy of the marking sheet for exercise 3 before the demonstration lab.
2) In your next scheduled lab arrange a demonstration session by writing your name and machine number of the whiteboard.

Feedback

You will receive your automated feedback at the end of week 1.8 after the extended deadline has passed.

Assessment

This exercise counts 40 marks towards the total of 200 marks available for the lab (20%), 30 marks from the automated marking and 10 marks from the demonstration.

Introduction

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. 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 concerned.

Seven segment displays are commonly used to display the digits from 0 to 9 on items of electronic equipment such as calculators, digital watches, etc. 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'). The normal set of patterns for this is shown later.

Because there are only ten possible digits, the number can be represented as a 4-bit code, since:

$$\log_2(10) = 3.32$$

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 is often represented as four bits. However, to make an easily readable display these four bits must be converted into a seven-bit code for the display. This is the function of a seven segment decoder.

Starting a Verilog source file

Previously we have generated schematic and symbol files; Verilog uses a third type, which is basically a plain-text file.

To open a Verilog file use the Cadence Library Manager, right click on the functional view to a design and select open. 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. Just like the previous process (from schematics) you can edit the symbol later if you wish.

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 interconnect these as a schematic at a higher level of the hierarchy using symbols.

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.

Combinatorial circuits in Verilog

Each 'unit' of a design in Verilog is referred to as a *module*; this is equivalent to a single schematic. There are a number of ways to define circuits in Verilog but the most appropriate here is a '`case`' statement; an example for you to work from is shown in figure 5.1.

```
// Comments as to function of module, author, date, etc.
module sevensegmentdecoder(input       [3:0] bcd,
                           output  reg [7:0] segments);

// Internal variable declarations – if any

always @ (bcd)      // "Sensitivity list"
  case (bcd)
    0:    segments = 8'b0011_1111;

// …           Fill in the rest!

    default:   segments = 8'b0000_0000;
  endcase

endmodule
```

Figure 5.1: Example Verilog code

Points to note:

- 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 output bus is eight bits wide. Segment 'a' is the least significant bit; the most significant is the decimal point, which will be set to 0, and hence 'off'.
- 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 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 'case 0' clause is shown; I wonder why? – because you will be doing the work to populate the rest!
- The number assigned to '`display`' 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).

Segments to Light

Figure 5.2 shows the segments that must be lit to display each of the digits 0-9.

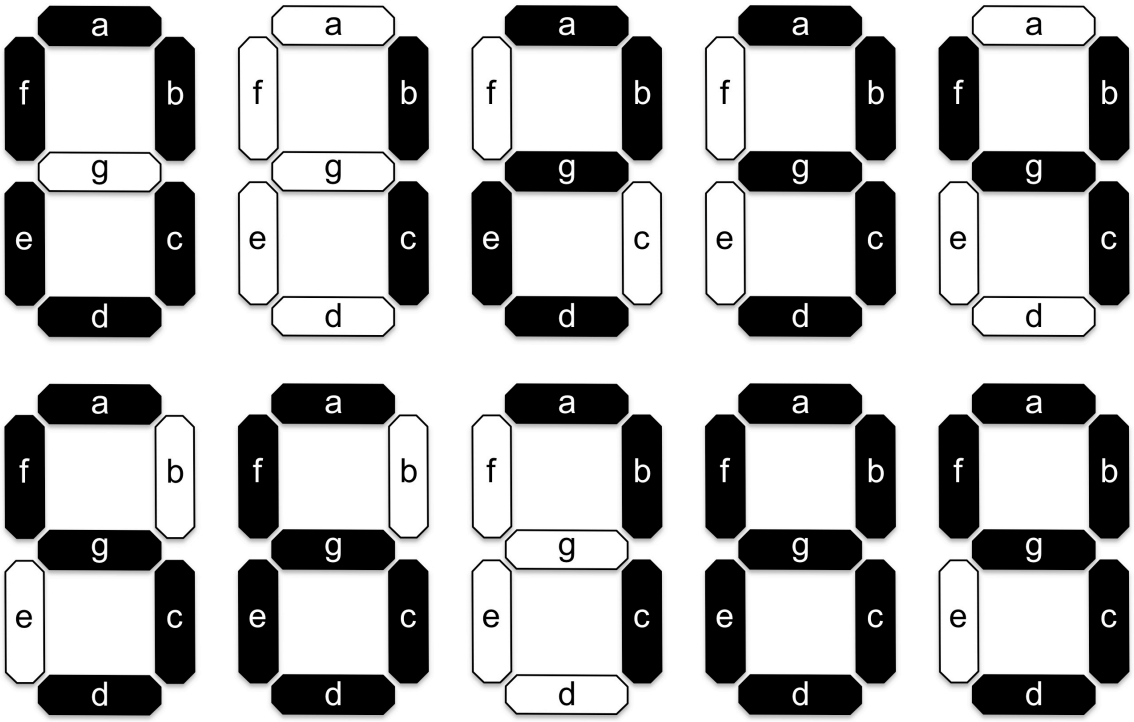


Figure 5.2: Seven segment patterns for the digits 0-9

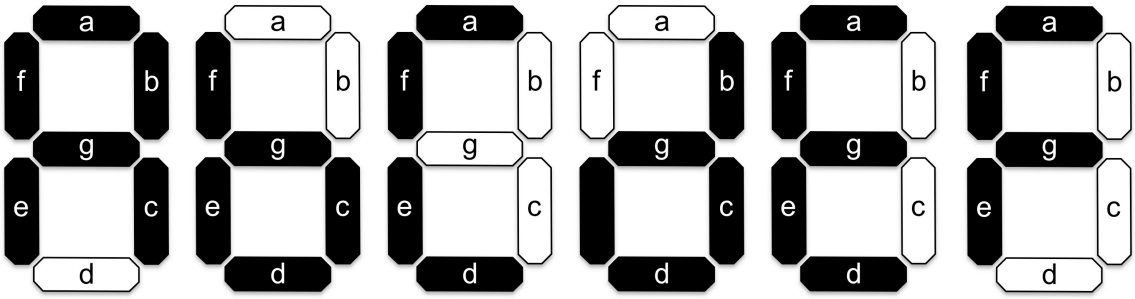


Figure 5.3: Seven segment patterns for the digits A..F

Practical

Seven Segment Decoder

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 value 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 (incomplete) module definition provided for you in order to implement the seven segment decoder. DO NOT change the input/output signals defined. Once completed save and close the Verilog editor, 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 *icds* and use: 'Tools ↵ Verilog ↵ Integration ↵ NC-Verilog…', this will open up a form where you must browse 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) Extend the design to give a full decode of all the possible input patterns to hexadecimal digits (i.e. 0-9 and A-F). The usual patterns for the additional 'numbers' are shown in figure 5.3.

d) Simulate your modified design to ensure it works as expected. You will need to extend your test stimulus to do this.

e) To interface the circuit to the 'outside world' a cell named "sevensegmentdecoder_test" with view "schematic" has been provided for you, instantiate the symbol for your seven segment decoder in this schematic. You will also need to instantiate the cell 'Board' from the library called "ENGLAB_12111" which contains the interface definitions that allow you to connect to the various components available on the experimental board. Figure 5.4 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. The inputs and output of 'Board' are described opposite.

Connect your seven segment decoder symbol 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.

Test the implementation of the design on the circuit board – see the following hints for advice on how to download a design to the experimental board. If it is functioning correctly the binary value selected using the switches should be decoder to a character on the seven segment display.

f) Submit your design as ex3.

Hand-in procedure

Run the submit command "12111submit" for the exercise to submit your files for marking. Print out the demonstration marking sheet using the labprint option available under 12111submit.

Formal demonstration in the next laboratory

You have now completed Exercise 3.

Board Inputs and Outputs

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

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 on 'Board' are (located to the left hand side):

Clk_50MHz	: 50 MHz free-running clock
Clk_1MHz	: 1 MHz free-running clock
Clk_1kHz	: 1 kHz free-running clock
Clk_1Hz	: 1 Hz 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.

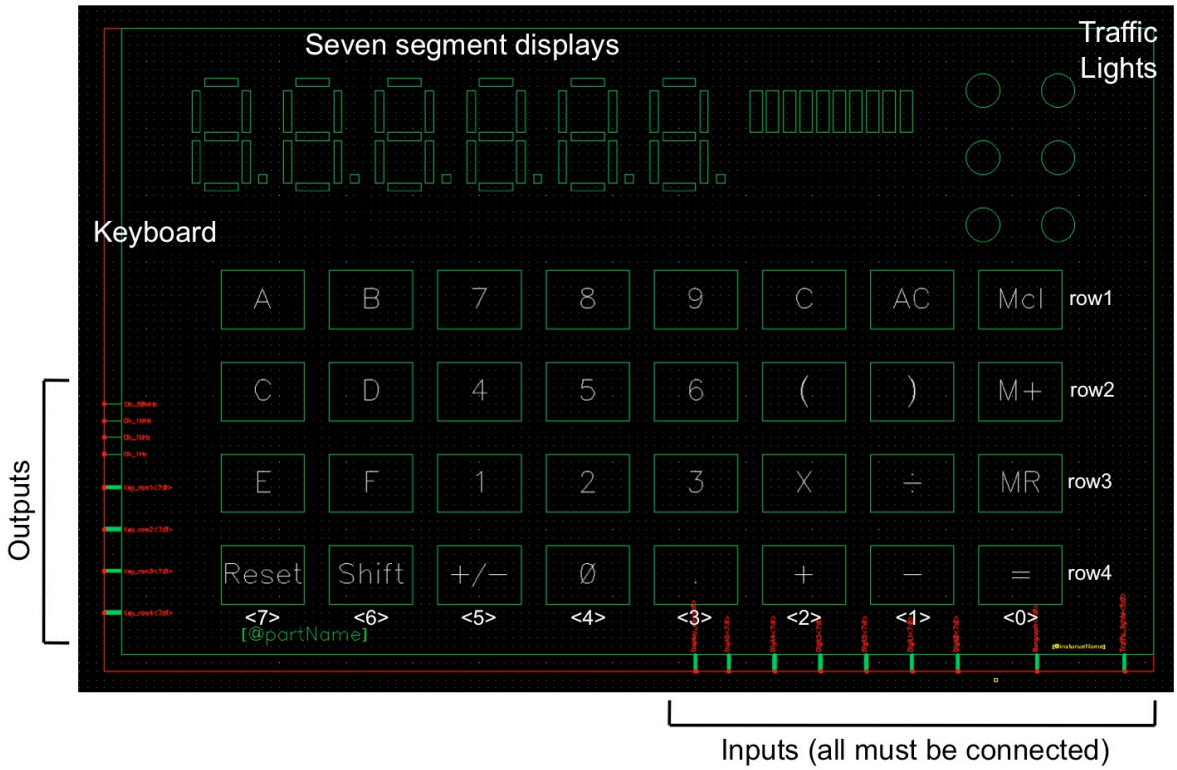


Figure 5.4: Symbol for the "Board"

Hints:

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 spartan3 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 spartan3 library provides a logic '1'.

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

Design compilation

When the design is complete it must be compiled into a physical circuit layout and then downloaded into the Xilinx gate array on the lab experimental board. To compile the design first 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 compilation 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. (Errors at this point can be difficult to diagnose; you may need to consult a demonstrator if you experience problems.)

Design download

The real test is to see the design running. You will need a lab experimental board which has a keyboard and some displays attached and hosts a system that includes the FPGA. This should be connected to your computer via a serial line (blue connector) and powered on (make sure it is!).

Downloading can be invoked from the Cadence simulator with 'Commands ↵ FPGA Load'. (This is for convenience – it is not part of the Cadence tools). 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.

Printing Verilog source files

A way of printing files neatly is available as follows:

- From the *icds* window select File ↵ Open
- Select the functional view of the appropriate cell
- Change Mode from Edit to Read
- In the text viewer that appears, click Design ↵ Print

This sends the print to one of the lfprt printers in a font size that should be legible if you've kept your lines to a sensible length.

Binary coded Decimal (BCD)

A numerical representation, which is sometimes encountered in computing, is a format known as Binary Coded Decimal or, more concisely, BCD. This is a format where a group of four bits is used to represent a decimal digit in the range 0-9. The other six states are not used and are 'illegal'.

BCD is used because it is convenient for display purposes; there is no need to translate decimal numbers to binary and then back to decimal again, which involves multiplication and division if multiple digits are represented. However, BCD numbers are less compact and harder to perform arithmetic on, and so are infrequently used in real applications.

Exercise 4: Finite State Machines & Counters

Aim

To investigate time dependent circuits. This exercise introduces **finite state machines** 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.

Duration

You have two lab session to complete Exercise 4. The deadline is 11pm on the day of your scheduled lab in week 1.9.

Deliverables

1) Submit your work via the script 12111submit at the end of the lab before the deadline. The two exercises that must be submitted individually: ex4a, ex4b. Make sure you print out a copy of the marking sheet for exercise 4 before the demonstration lab.
2) In your next scheduled lab arrange a demonstration session by writing your name and machine number of the whiteboard.

Feedback

You will receive your automated feedback at the end of week 1.10 after the extended deadline has passed.

Assessment

This exercise counts 40 marks towards the total of 200 marks available for the lab (20%), 30 marks from the automatic marking and 10 marks from the demonstration. An additional 10 marks are available if you complete the extra optional exercise.

Finite State Machines in Verilog

A counter is a simple finite state machine (FSM). In the lecture notes you were given an example structure of a Verilog FSM. The Verilog code in Figure 6.1 gives an example of a simple 2-bit counter (0➔1➔2➔3➔0 …). In this example each "count" represents the state of the machine.

The internal variables "`current_state`" and "`next_state`" have been used rather than 'exposing' the output "`digit`" directly; this is a useful technique when the output is not identical to the internal state. "`next_state`" is determined when "`current_state`" is updated at each positive edge of the clock.

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

// internal variables
reg       [1:0] next_state;        // Internal variable
reg       [1:0] current_state;     // declaration

initial
begin
// you may need to initialise some values for the simulation
// to work – maybe one of the internal variables!
end

// define the next state – no inputs here
always @ (current_state)           // on a change of
   case (current_state)            // current_state
      0:  next_state = 1;          // update next_state
      1:  next_state = 2;
      2:  next_state = 3;
      3:  next_state = 0;
      default: next_state = 0;     //Catch anything missed
   endcase

// on a clock edge perform the state assignment
always @ (posedge clock)
     current_state <= next_state;

// assignment the output – digit
always @ (current_state)
     digit = current_state;        // When current_state
                                   // changes, so does the
endmodule                          // output - digit
```

Figure 6.1: Verilog code for a simple finite state machine

There may be several control inputs to the state machine. These input bits, together with the bits defining the current state, are used to determine the next state. In the example given above the counter is free running and the output will always change on a rising edge on the input signal `clock`. An additional consideration is that there may be other 'forbidden' states in the system. For example if we modified the code above so that it only counts from 0 – 2 (0➔1➔2➔0 …), what happens if the current state becomes 3 (somehow), which is an invalid count in the design and hence a forbidden state. Normally these states should either move directly or indirectly into one of the legal states; it is important to avoid 'closed loops' of one or more forbidden states. 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.

Points to note:

- the sequential '`always`' is just sensitive to '`clock`' because other inputs *only* have influence at that time.
- '`digit`' is a 2-bit register that is both read and written to in this statement.
- 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 is sensitive to a clock.
- `current_state` will only be updated on a rising edge of `clock`, immediately on the value of `current_state` changing the value of `next_state` will be evaluated as `current_state` is in the sensitivity list of the `always` block that updates `next_state`.

Practical

ex4a: Modulo-10 Counter

Design a Verilog module for a modulo-10 counter as defined in the specification given below.

Specification

- **The counter should counts from "0" to "9" (and cycles) at a rate determined by an external clock.**
- **The counter should only increment when the input *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***
- **The output *digit* should reflect the current count**
- **The output carry should be high whenever the value of the count is 9, it should be 0 otherwise**

You should design your counter as a finite state machine adopting the structure given in Fig 6.1. An input *clock* is used to control the rate of the counter.

The following files have been provided for you:

- **cell "counter09", view "functional" – module description of the mod-10 counter**
- **cell "counter09_test", view "schematic" – test schematic for testing the operation of your mod-10 counter using the lab experimental board.**

The counter should run autonomously and should display *only* the digits 0-9 in the correct order. For neatness and to promote reusability you should design your counter as a separate module with its own symbol. The inputs and outputs have been defined for you, do not change the names provided.

You MUST simulate your design to confirm it is working correctly, checking that the `enable` and `reset` signals operate as specified in the specification. In order to simulate you will need to initialise your counter into a known state; this can be done by toggling the *reset* signal (0->1->0) in your stimulus file.

For the physical implementation of your modulo-10 counter, the component 'Board' in the "ENGLAB_12111" has a number of clock outputs that may be used. Produce a higher-level schematic, using the cell "counter09_test", where your counter is connected to a seven segment display on the lab board via the seven segment decoder you produced earlier. Your counter should increment/decrement every second. Test your modulo-10 counter using the experimental boards in the laboratory.

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 section on *Verilog: A Brief Introduction*.

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 6.2 gives two examples of `always` blocks, one where an input signal, `signalin`, acts synchronously, and another where the input signal acts asynchronously.

In the first example the output `anoutput` is assigned the value 1 if `signalin` = 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 `signalin` in the sensitivity list of the `always` block the output `anoutput` is assigned a value whenever the input, `signalin`, changes, regardless of the clock – it is acting saynchronously.

Traffic Lights

Traffic lights are used universally to control the flow of traffic at junctions. Figure 6.3 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 finite state machine to control a traffic light system where each state corresponds to one of the eight possible light configurations in the sequence.

Note that states 1 and 5 are the same – both sets of lights are on red – however for simplicity these will be treated to be different states.

```
always @ (posedge clock)
if(signalin == 1)
    anoutput = 1;
  else
    anoutput = 0;


always @ (posedge clock, posedge signalin)
if(signalin == 1)
    anoutput = 1;
  else
    anoutput = 0;
```

Figure 6.2: Synchronous and asynchronous blocks

The delay between state transitions differ depending upon the state. For example, the delay from state 0 to state 1 will differ from state 2 to 3.

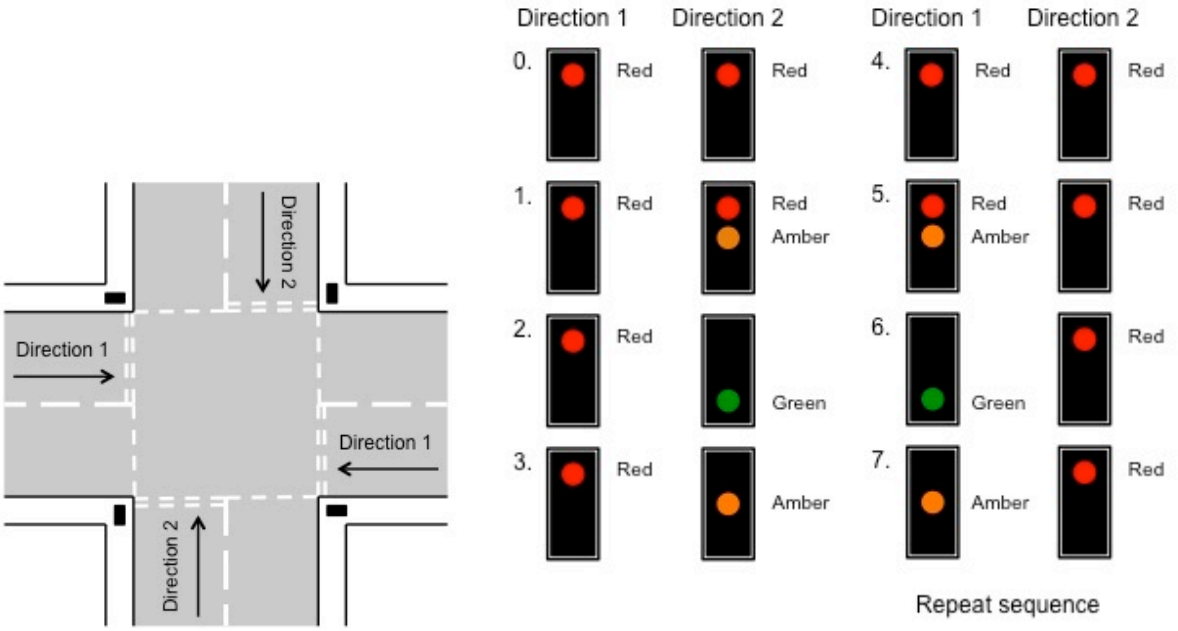


Figure 6.3: Traffic light sequence for a simple crossroads

Practical

ex4b: Traffic Lights

Design, develop and demonstrate a working traffic light system using the lab experimental boards according to the specification given below.

Specification

- **The sequence of the traffic lights should follow that of a UK crossroads (see figure 6.3)**
- **The duration of each sequence should be controllable (see figure 6.4)**
- **The traffic lights should reset to an initial starting state on the application of a *reset* signal (asynchronous)**

The following files have been provided for you:

- **cell "trafficlight", view "functional" – module description of a finite state machine for controlling the traffic light sequence**
- **cell "trafficlight_test", view "schematic" – test schematic for testing the operation of your traffic light system using the lab experimental board.**

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**
- **count – 4-bit value providing the external counter value, used to determine whether a required period of time has passed**
- **reset_count – reset the external counter to zero**
- **enable_count – enable the external counter**

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 6.5. The design "trafficlight" will need modifying to give the appropriate light sequence for each state as well as the appropriate delay, as given in the table in figure 6.4.

To test your design using the experimental board available in the laboratory you must use the schematic "trafficlight_test" using the symbol "Board" to gain access to the components on the experimental board. Using the design "trafficlight" as your controller and the mod-10 counter you developed previously ("counter09") to provide a counter to the controller, complete the design so you are able to demonstrate a working 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 6.6.

The clock for the fsm must be connected to a higher clock frequency that that of the counter (1s) so you can ignore the delay associated with the fsm clock.

Hand In

Run the submit command "12111submit" for each exercise (ex4a and ex4b) to submit your files for marking. Print out the demonstration marking sheet using the labprint option available under 12111submit.

Formal demonstration in the next laboratory.

You have now completed Exercise 4.

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 6.4: Traffic light sequence and delay for each state

Bit	LED
0	Green right
1	Amber right
2	Red right
3	Green left
4	Amber left
5	Red left

Figure 6.6: LED connections in "Board"

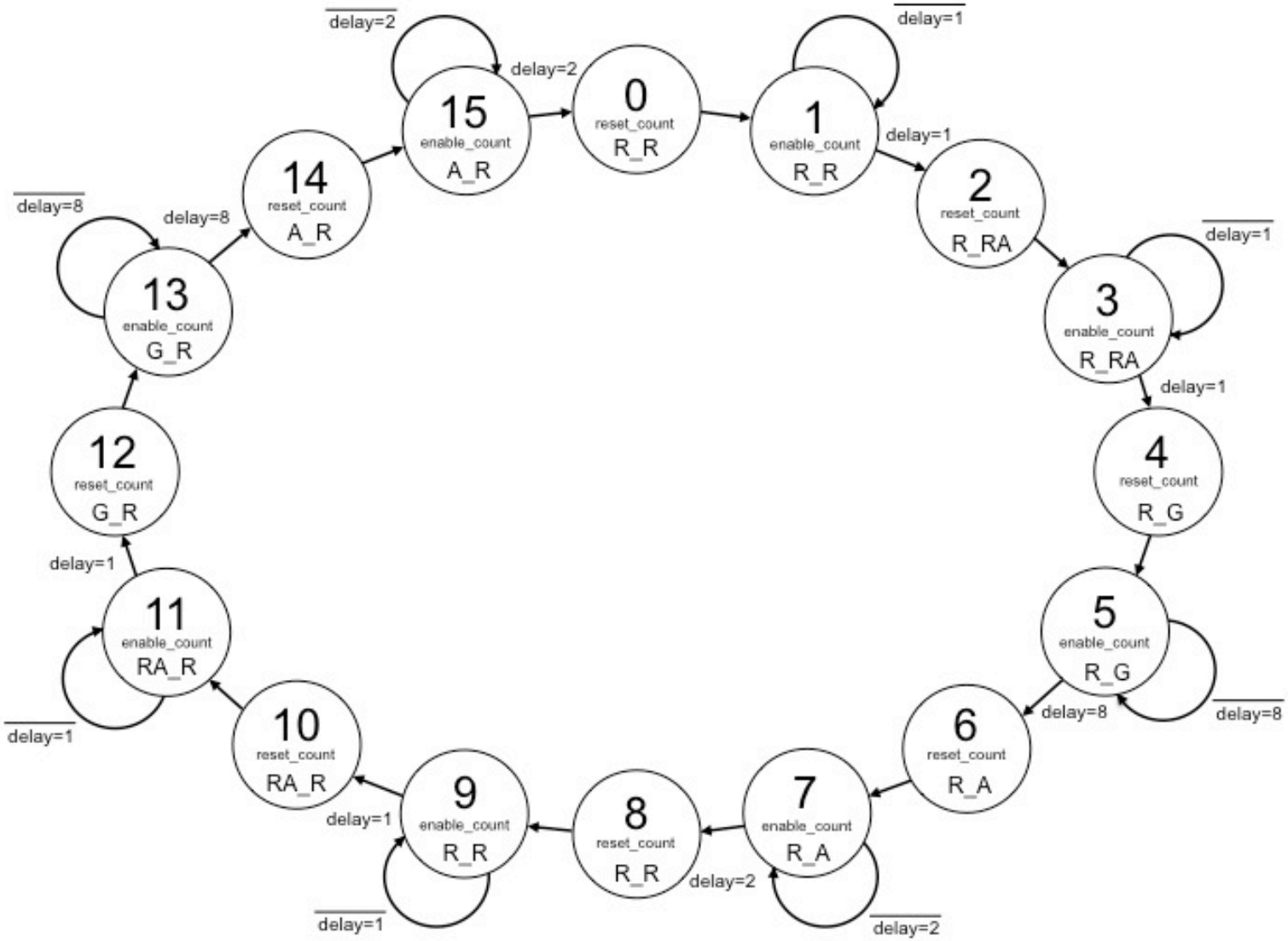


Figure 6.5: Traffic Light Controller State Transition Diagram

Adding readability … and maintenance

Although numbering states {0, 1, 2, …} works fine, there are a couple of poten 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, change all the references to ‘state 3’ to ‘state 5’ whilst not changing any oth code.

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

Similarly, output codes may be clearer is enumerated, so that names can then be u

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

```
‘define   R__R      6’b100_100;    // ‘_’ in a number
‘define   RA_R      6’b110_100;    // ignored
‘define   G__R      6’b001_100;
‘define   A__R      6’b010_100;
‘define   R_RA      6’b100_110;
‘define   R__G      6’b100_001;
‘define   R__A      6’b100_010;


. . .
if (state == ‘ALL_STOP) lights = ‘R__R;  // Example on
. . .
```

When the compiler encounters “`R__R” it looks up and substitutes from the definition (“100100” in this case). If the definition is changed, all the refere changed automatically.

Note the use of the ‘backtick’ character (‘`’) indicating both the definition an macro.

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

‘Traditionally’ such definitions are given in upper-case (capital) letters. This ha meaning for the compiler but is a programmers’ convention.

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 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.

Exercise 4x: Optional

Aim

To develop more complex counter design and use hierarchy in design.

This exercise follows on directly from exercise 4 and allows you to obtain extra marks for that exercise. You must only attempt if you finish the regular exercises before the deadline. There a few marks available for the exercise (5% of the total lab mark), so there is little penalty if you do not complete it.

Preparation

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

Duration

The deadline for the optional extra exercises is the start of the lab in week 1.12.

Deliverables

1) Submit your work via the script 12111submit at the end of the lab before the deadline. The exercise that must be submitted is ex4x. Make sure you print out a copy of the marking sheet for the optional exercise 4 before the demonstration lab.
2) The optional exercises can be demonstrated at the same time as exercise 4 if you have completed it. It can also be demonstrated in the last scheduled lab session (the marking session). Remember to write your name and machine number of the whiteboard.

Feedback

You will receive your automated feedback at the end of week 1.12.

Assessment

This exercise counts 10 marks towards the total of 200 marks available for the lab (5%).

Practical

ex4x: 000 – 999 Counter

You are required to design a counter that counts from 000 to 999 following the specification given below.

Specification

- **The counter should counts from "000" to "999" (and cycles) at a rate determined by an external clock.**
- **The counter should only increment when the input *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***

The following files have been provided for you:

  - **cell "counter000999", view "functional" – module description of your counter**
  - **cell "counter000999_test", view "schematic" – test schematic for testing the operation of your mod-10 counter using the lab experimental board.**

The counter should be constructed using instances of your modulo-10 counter (hierarchy!). Once again the input/output pins have been defined for you, do not change these.

Your design for a modulo-10 counter includes an output carry that goes high when the current value of digit is 9. You can use this output to control the behaviour of the next digit. This output may used to enable the clock on a second (tens) counter. Note that, because the counter's outputs trail the input clock (due to the counter's propagation delay) the 'ripple carry' is active at the time the counter is clocked from state "9" to state "0"; this is the same time that the next counter higher must be clocked (e.g. from "19" to "20").

Simulate your design to confirm it is working correctly.

A schematic has been provided to test your counter design using the lab experimental board, cell "counter000999_test". Use this to test your counter design using the experimental board by display the 3-digit count on seven segment display.

Hand In

Run the submit command "12111submit" for the exercise (ex4x) to submit your files for marking. Print out the demonstration marking sheet using the labprint option available under 12111submit.

Formal demonstration in the last scheduled lab.

You have now completed the optional part of Exercise 4.

Exercise 5: MU0 – A Microprocessor System

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.

Duration

You have two lab sessions to complete Exercise 5. The deadline is 11pm on the day of your scheduled lab in week 1.11.

Deliverables

3) Submit your work via the script 12111submit at the end of the lab before the deadline. The two exercises that must be submitted individually: ex5. Make sure you print out a copy of the marking sheet for exercise 5 before the demonstration lab.
4) In your next scheduled lab arrange a demonstration session by writing your name and machine number of the whiteboard.

Feedback

You will receive your automated feedback at the end of week 1.12 after the extended deadline has passed.

Assessment

This exercise counts 40 marks towards the total of 200 marks available for the lab (20%), 30 marks from the automatic marking and 10 marks from the demonstration. An additional 10 marks are available if you complete the extra optional exercise.

The microprocessor system

In this exercise you are asked to finish a partially completed MU0, simulate it to confirm it operates as expected and then download the working design onto a field programmable gate array (FPGA) on the experimental boards in the lab – this will implement a working processor!

You have seen the MU0 microprocessor in lectures and looked in detail at its operation. 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).

MU0 Overview

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.

Control Logic

The control logic is all provided for you and has been constructed in schematic form. It provides all the control signals you will need for this exercise.

The control logic states are shown in Figure 7.1; states alternate unless the processor is halted.

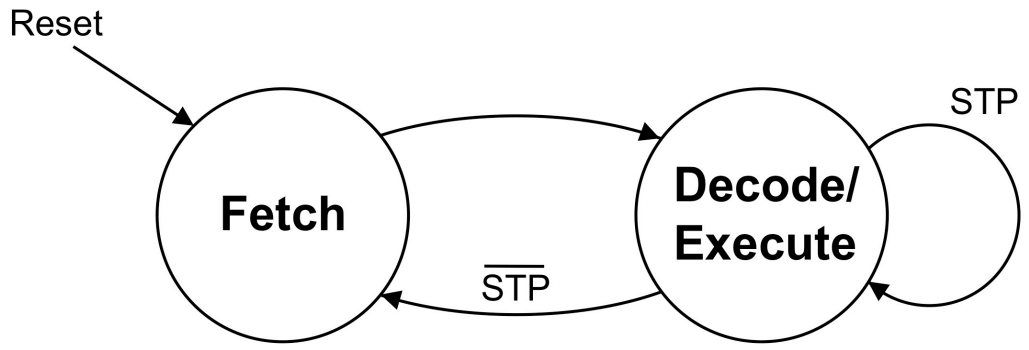


Figure 7.1: MU0 control state diagram

The control logic requires the following inputs:

- Clock : global clock
- Reset : system reset
- F<3:0> : the upper 4 bits of the fetched instruction
- N : the MSB (sign bit) of the accumulator
- Z : a Boolean indication that the accumulator contains zero

The control logic provides the following outputs:

- PC_En : Programme Counter write enable
- Acc_En : Accumulator write enable
- IR_En : Instruction Register write enable
- X_sel, Y_sel : ALU input selects
- Addr_sel : Address output select
- ALU_fn<1:0> : ALU function select: **01** : X + Y, **11** : X – Y, **10** : X + 1, **00** : Y
- Halted : status: the processor is halted

The Processor Interface

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

- **Clock**: input
- **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 processor fetches (from the memory), decodes and executes instructions. Each fetch is from the 'next' address unless a branch instruction is taken: branches define a new fetch address. The address of the instruction to be fetched is held in the program counter as in a common processor.

The programmers' model has two registers:

- a 16-bit Accumulator
- a 12-bit Program Counter

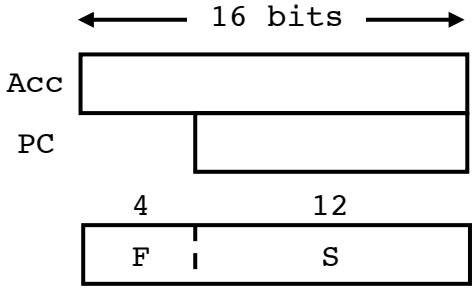


Figure 7.1 MU0 Registers

There is one 16-bit instruction format consisting of a 4-bit code representing the instruction, and a 12-bit value representing an address in memory.

The instructions are given in the table in figure 7.3. The instruction act on values (or operands) store in memory.

Reset initialises (resets) the internal state (registers) of the processor. A consequence is that the PC is set to 0000 on reset going high. On reset going low execution starts again by fetching the instruction from address 000.

Memory interface

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.

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.

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 is accumulator is +ive
0110	JNE S	Jump to instruction address S is accumulator is not 0
0111	STP	Halt execution
1000 - 1111	-	Reserved

Figure 7.3: MU0 instructions

Implementation

As illustrated in the lectures, the implementation of the MU0 may vary. The following assumes that each instruction occupies two clock cycles, the first fetching the instruction and the second executing what was fetched – as illustrated by the control state diagram in figure 7.1 – the two phase being:

Fetch: read memory at address PC; hold result in Instruction Register (IR); increment PC;

Decode/execute: various operations depending on instruction fetched.

Figure 7.4 illustrates one possible implementation of the datapath in the MU0. It is important to understand the operation of the datapath to ensure the multiplexers and control signals are connected properly as in this exercise you will be required to finish off the MU0 datapath by adding the missing components.

There are three 2:1 multiplexers in the MU0 datapath:

- one controlled by X_sel that selects between the accumulator or program counter to form the input X to the ALU – 16 bit input/output.
- one controlled by addr_sel that selects between program counter and the address section of the instruction register to form the address to memory – 12 bit input/output.
- one controlled by Y_sel that selects between the data to the ALU or the address section of the instruction register to form the input Y to the ALU – 16 bit input/output.

The status of the multiplexer control signals is determined by the current state and instruction, as shown in the cut-down state transition table for the MU0 shown in the table of figure 7.5.

In the case of the addr_sel multiplexer: from figure 7.5 it can be seen that for an instruction fetch (state 0) addr_sel = 0, so the output of the multiplexer should contain the contents of the program counter, as this is the address of the next instruction to be fetched. The address portion of the instruction stored in the instruction register (IR) should only be selected (addr_sel = 1) when executing instructions. If this multiplexer is configured incorrectly then it is clear that the next instruction will not be fetched as required and the processor will not work.

In the case of X_sel: the value of X_sel is 1 for an instruction fetch only (state 0) as it is during the fetch state that the PC is incremented for the next instruction, hence it is important that that the program counter is fed to the X input of the ALU at tis stage of the control.

We will leave it for you to think about the Y_sel multiplexer operation and to determine the correct wiring for your multiplexers.

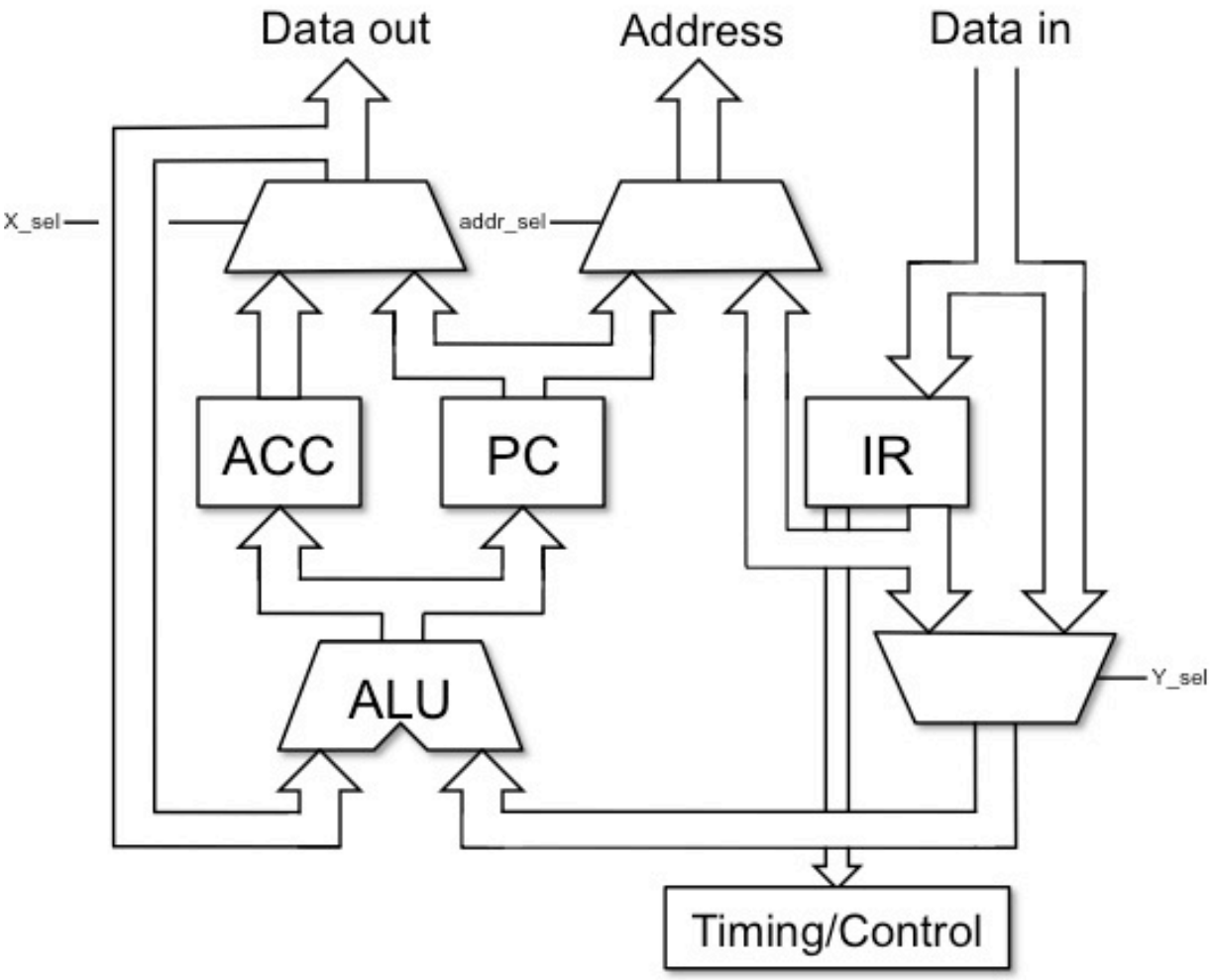


Figure 7.4: Possible MU0 datapath

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

Figure 7.5: MU0 State Transition Table

Practical

This practical is divided into stages. First you must complete and simulate the MU0 to confirm it is working, then you must implement it in hardware using the experimental boards available in the laboratory.

For this exercise, designs can be found in the MU0_lib library.

Completing & Simulating the MU0

Construct and simulate an MU0 microprocessor. There is an incomplete MU0 schematic preloaded in the library as a starting point.

Figure 7.4 illustrates the MU0 datapath, you are provided with an initial datapath layout to save time; you will need to modify this. If you open the datapath (MU0_lib -> MU0_datapath), you will notice that some of the constituent parts are not provided, compared to Figure 7.4: these include registers and multiplexers.

If you open the MU0_ALU schematic (MU0_lib -> MU0_ALU) you will notice that the 16-bit adder in the ALU is missing.

The goal of this first exercise is to add the missing components in these two schematics.

16-bit registers can be constructed simply by modifying the 'Instance Names' of single, edge-triggered flip-flops. Several flip-flop variants are available in the library; a key to their names can be found on page 45 of this manual. You will need to implement 3 registers: the accumulator, the program counter, and the instruction register. Your registers should take clock and reset signals, along with the appropriate register enable signal: Acc_En, PC_En, and IR_En.

You have already produced a 16-bit adder, "adder_16bit", earlier that can be used to complete the design of the MU0 ALU.

You will also need to design the missing multiplexers. The cells have been provided for you, "mux_2to1_12bit" and "mux_2to1_16bit" both with view "functional". You should complete the functional description for a 2:1 16-bit MUX and a 2:1 12-bit MUX, and simulate to confirm correct operation. Figure 7.6 gives some example code for implementing a single bit 2:1 mux, you can use this as a basis for constructing your 12-bit and 16-bit versions.

Once you have completed your multiplexers you can add these to the ALU datapath - take care that the multiplexer selects are the 'right way up' – this may take some thinking, as problems with this exercise usually relate to the control signals for the multiplexers selecting the wrong inputs. 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.

Once you have completed the MU0 datapath the processor can be simulated to test its operation and confirm that your completed processor works. To enable the simulation to be performed a memory model must be provided to supply data and instructions to the processor for testing.

Two memory models are provided: `memory_1` **and** `memory_2`**; for simulation**

purposes you should use `memory_1`, which is a RAM written in Verilog and will pre-load an image so that the processor has something to run. A test RAM image is stored in a pre-prepared file called `MU0_test.mem`, this is loaded into memory when the design is compiled. The code for this test is given below, and available at `$COMP12111/MU0_examples/MU0_test.s` (Appendix C lists the code for `MU0_test.s`). It is intended to test the processor's operation. If it works correctly the processor should halt at a STP instruction, i.e. the halted signal should go high after a short period.

A cell "MU0_test" with view "schematic" has been provided for testing the operation of your processor, add to this schematic your completed processor (""MU0") and the memory module `memory_1`, as shown in figure 7.7, making the appropriate connections between the processor and memory. Use this to simulate and therefore test the processor's operation. If the processor fails to do the prescribed behaviour then it is likely you have one or more multiplexers connected incorrectly (see previous exercise).

For the simulation you will need to edit the stimulus file to define the operation of the Clk and Reset signals.

Hand In

Run the submit command "12111submit" for the exercise (ex5) to submit your files for marking. Print out the demonstration marking sheet using the labprint option available under 12111submit.

Formal demonstration in the last scheduled lab.

```
module mux2to1 (input               select,
                input    [1:0]      d,
                output   reg        q);

always @ (select, d)     // whenever select or d change
begin                    // so does the output
     case(select)        // use a case statement to
     0: q = d[0];        // determine the assignment.
     1: q = d[1];
end

endmodule
```

Figure 7.6: Verilog module for a simple 2:1 MUX

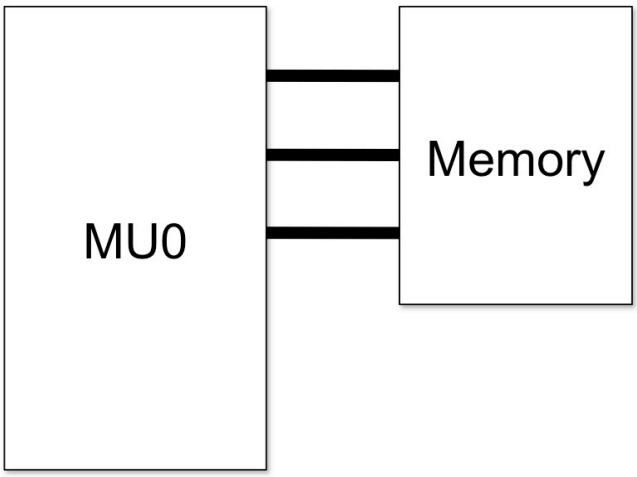


Figure 7.7: Processor simulation subsystem

Verilog memory model

A memory model (`memory_1`) is provided in your MU0_lib library that can be connected directly to the processor model. This is suitable for simulation purposes and is shown in figure 7.8, with some explanations below.

```
module memory  (input                    Clk, Wen,
                input        [11:0]      address,
                input        [15:0]      write_data,
                output   reg [15:0]      read_data);

// internal variables
reg  [15:0]    mem [12'h000:12'hFFF];

initial $readmemh("<filename>", mem);

always @ (negedge Clk)
   if (WEn) mem[address] <= write_data;
   else     read_data    <= mem[address];

endmodule
```

Figure 7.8: Memory model Verilog module

- The key element is the declaration of "`mem`", which is a 4 K array of 16-bit words. The array size has been specified in hexadecimal and *ascending* order to make loading values more 'obvious'. (This is a fairly typical convention.)
- The memory array is preloaded from a file by '`$readmemh`' before simulation starts. This is also a (rare) example of an '`initial`' statement which can be compiled under some circumstances – the memory contents are included in the FPGA bitfile and downloaded to the chip.

- To operate with the synchronous MU0 processor – and for easy FPGA implementation – memory state changes use a clock. Reads and writes here use the negative (falling) edge of the clock so, for example, **read data will appear in the middle of a clock cycle**. Remember this when looking at traces. [Strictly speaking this compromises the synchronous model, however it is a convenient way of providing a simple timing model here.]
- Reads and writes are simultaneous, thus if '`REn`' and '`WEn`' were both asserted in the same cycle the read data would be the 'old' contents of the memory. This should not occur in your simulation.

For download to the FPGA board a different, compatible memory is used; this provides a "dual-port" memory, allowing display of its contents while the processor is running.

Implementation of the MU0

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 '`MU0_system`', as shown in figure 7.9, for you to modify.

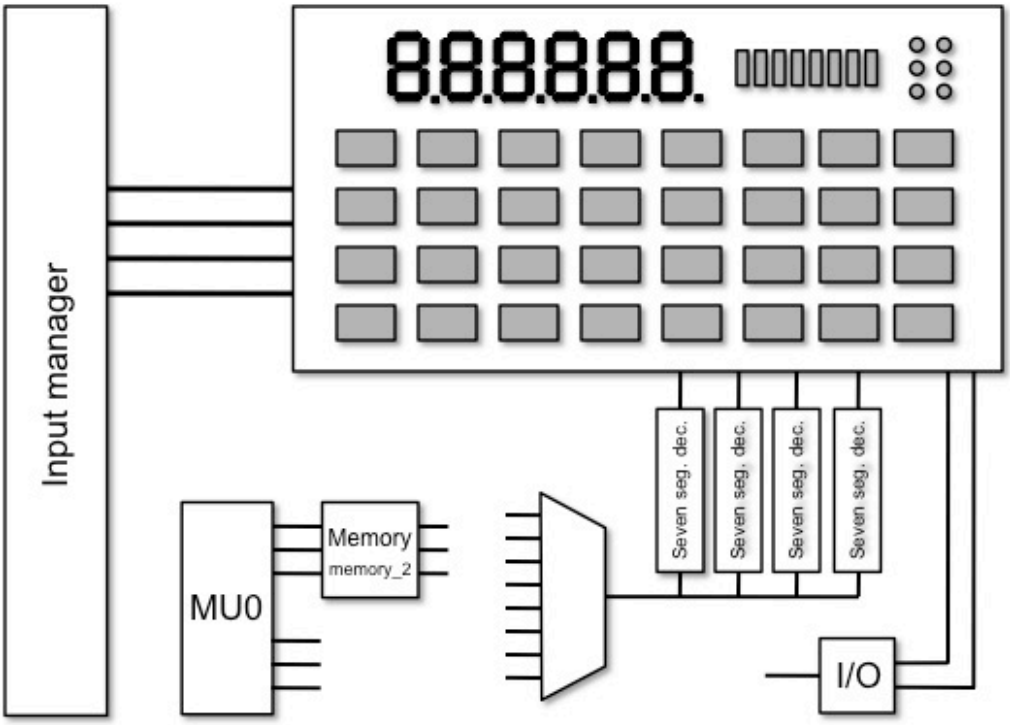


Figure 7.9: Suggested microprocessor test layout – MU0_system

- '`Input_manager`' provides an interface to the clocks and keyboard, allowing stepping and running, and an entry system to allow memory to be displayed.
- Some pre-placed components provide an output interface so that different buses can be displayed. Latches provide output to various LEDs. Although this is not necessary, it makes the system more 'user friendly'. Provision to display the MU0's internal state is provided but to do this extra pins exposing the register contents would have to be added to the processor.
- Memory - '`memory_2`' is a Verilog memory that 'bolts' directly onto the processor.

It differs from '`memory_1`' in being a dual-port memory, which allows its contents to be viewed for debugging purposes even while the processor is using it. It also has different (default) contents, loading a demonstration programme from your own user space.

- Four of the displays are driven using seven segment decoders. If your seven segment decoder you produced in the earlier exercise can display hexadecimal values you can substitute it in the appropriate places. If not, you could upgrade your own design or use the '`decoder_7_seg`' module provided. (This last is implemented as a schematic, for comparison.)

Most of figure 7.9 is simply test and display logic; the important parts are the processor (`MU0`), memory (`memory_2`) and (in this case very limited) input/output.

Display subsystem

The display module allows you to use the keyboard on the lab experimental board to step through the operation of the processor, and display different data values on the seven segment display. However, you must wire its ports up correctly for this to work as described!

The clock can be controlled with the following keys:

Shift - provides a single clock pulse
\+ - starts a regular clock
– - stops the regular clock
+/- - toggles regular clock between 1 Hz and 10 Hz.

Pressing number keys 0-5 shows one of the internal buses (if connected). There are some 'expected' connections that are assumed for labelling the display (figure 7.10). 16-bit buses/ registers are shown as 4 hex digits, 12-bit ones as 3 digits.

Key	Bus	Label	Bits	Provided
0	Address	A	12	Yes
1	Input data	di	16	Yes
2	Output data	do	16	Yes
3	PC	Pc	12	No
4	IR	Ir	16	No
5	Accumulator	Ac	16	No

Figure 7.10: Recommended display wiring

Pressing number keys 0-5 shows one of the internal buses (if connected – you may have to bring some buses out of the MU0 symbol if you would like to view the contents of buses). There are some 'expected' connections that are assumed for labelling the display. 16-bit buses/ registers are shown as 4 hex digits, 12-bit ones as 3 digits.

In addition, a memory location can be viewed:

- Press M+ - this prompts 'A?' for an address
- Enter the three digit hex number (labelled 'd')
- Press '=' - for data display (labelled 'd')

This is updated as the memory changes.

Software

The initial memory contents of your MU0 system is downloaded into the FPGA with the design. It must therefore be available when the design is compiled.

'`memory_2`' initially expects a file called '`MU0_demo.mem`' in your Cadence compilation directory (`~/Cadence/COMP12111/xilinx_compile/`). The Xilinx compilation software is quite sensitive to the format of this file so, unfortunately, it cannot contain comments and must be *exactly* the same size as the RAM (i.e. 4 Kwords). A default file should already be present; there is an assembler (described later) that can output files in this format.

`$COMP12111/MU0_examples/MU0_GCD.s` is an example binary file that contains object code for the programme shown in figure 7.11. This uses all but one instruction type and can be used to exercise your design. It is a greatest common divisor programme; it calculates the highest common factor of a pair of inputs. This is done by subtracting the smaller from the larger repeatedly until they become equal.

```
                LDA  initial_A      ; Initialisation
                STA  A              ;
                LDA  initial_B      ;
                STA  B              ;


loop            LDA  A              ; Main loop
                SUB  B              ; A – B (compare)
                JNE  unfinished     ;
                JMP  finished       ;
unfinished      JGE  A_GE_B         ; A >= B
                LDA  B              ;
                SUB  A              ;
                STA  B              ; B := B – A
                JMP  loop           ;
A_GE_B          STA  A              ; if (A >=B) A := A – B
                JMP  loop           ;

finished        LDA  A              ;
                STA  Result         ; Store result
                STP                 ; Finished – halt

initial_A       DEFW 0x1C94         ; Values to work with
initial_B       DEFW 0x7850         ;
Result          DEFW 0x0000         ;
A               DEFW 0x0000         ; Temporary variables
B               DEFW 0x0000         ;
```

Figure 7.11: MU0 demonstration programme (MU0_GCD.s)

Practical

Implementing the MU0

Complete the MU0 system shown in figure 7.9, i.e. the "MU0_system" schematic using your MU0 processor and adding any necessary missing components. Compile and download the MU0 system to the FPGA on the experimental board. For illustration purposes wire up the 'debug' seven segment displays. In order to view progress you can then follow the value on the processor's address bus. More information can be obtained if you modify the processor to 'pin out' its internal register states. Provision has been made to display these by wiring them to the appropriate buffers.

One useful output indicator is the 'halted' signal that indicates the processor has executed a `STP` instruction – you observed this in the previous exercise. This could be connected, for example, to the seven segment decimal points to indicate that the programme has completed.

In your demonstration show a demonstrator the default programme running; the 10 Hz clock works best for this.

Q10.1 What programme is implemented on the experimental board when using `memory_2`?

Hand in

There is no submission for this part of the exercise. You will demonstrate your working MU0 in the demonstration lab.

You have now completed Exercise 5.

Exercise 5x: Optional

Aim

To develop further your MU0 design.

This exercise follows on directly from exercise 5 and allows you to obtain extra marks for that exercise. You must only attempt if you finish the regular exercises before the deadline. There a few marks available for the exercise (5% of the total lab mark), so there is little penalty if you do not complete it.

Preparation

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

Duration

The deadline for the optional extra exercises is the start of the lab in week 1.12.

Deliverables

1) There is no submission for the optional part of exercise 5. However, you must make sure you print out a copy of the marking sheet before the demonstration lab.
2) The optional exercises can be demonstrated at the same time as exercise 5 if you have completed it. It can also be demonstrated in the last scheduled lab session (the marking session). Remember to write your name and machine number of the whiteboard.

Feedback

There is no submission for this exercise, feedback will be given in the demonstration lab.

Assessment

This exercise counts 10 marks towards the total of 200 marks available for the lab (5%).

Software Development

An assembler `MU0asm` is available which will produce MU0 object code from a symbolic source. Example source code can be found in: `$COMP12111/MU0_examples/` (`/opt/info/courses/COMP12111/MU0_examples/`).

Type `MU0asm <filename>` to generate a file `<filename>.mem` in your Cadence compilation directory (`~/Cadence/COMP12111/xilinx_compile/`). This will generate a memory image file which can be loaded into Verilog simulation or compiled into a FPGA bitfile for downloading. Adding the "`-l`" option list will also generate a list file which can be used as a reference for debug purposes. Modifying the filename in 'memory_2' to "<filename>.mem" will include this in a subsequent compilation.

Practical

The following exercises offer extra work that you might choose to explore if you have time to do so – DO NOT rush the earlier exercise in an attempt to do these exercises! They are independent and should be attempted individually. The exercises are deliberately sketchy in their description.

Extra Option 1

Implement a complete MU0 processor as a Verilog module.

This is not as intimidating as it sounds if you proceed logically:

- **Develop (and simulate!) the control state machine.**
- **Use this and the IR state to fetch and execute instructions, i.e. assign the register state changes.**
- **Use the state/IR information to set the values for the output controls.**

Note: a simple implementation might not have obvious datapath/control separation, looking (a bit) like a Java programme.

Extra Option 2

Write, debug and demonstrate a programme written in MU0 assembly code. [Nice code examples will be added to the examples directory for future years.]

Hand In

There is no submission for this exercise, just print out the demonstration marking sheet using the labprint option available under 12111submit.

Formal demonstration in the last scheduled lab.

You have now completed the optional part of Exercise 5.

Verilog reduction operators

Occasionally it is useful to examine the state of a whole bus. A pertinent example would be detecting if the MU0 accumulator was zero or not for a JNE instruction. In this example, the desired operation is a NOR of all the bits in the accumulator.

This could be expressed as:

```
assign zero = ~(acc<15> | acc<14> | acc<13> | … | acc<0>);
```

But this is rather tedious. Instead a 'reduction operator' can be used: these are unary operators which act in the appropriate manner on all the bits in a multi-bit operand.

The example above can be shortened to:

```
assign zero = ~|acc;
```

Reduction operators for all the basic logic function are provided:

Operation	**Signals**
AND	&my_signals
NAND	~&my_signals
OR	|my_signals
NOR	~|my_signals
XOR	^my_signals
XNOR	~^my_signals

Appendices

Appendix A

Boolean Logic Identities

Zero and Unit Rules

$$A \bullet 1 = A$$

$$A + 1 = 1$$

Complement Relations

$$A \bullet \overline{A} = 0 \qquad A + \overline{A} = 1$$

$$\overline{(\overline{A})} = A$$

Idempotence

$$A \bullet A = A \qquad A + A = A$$

Commutative Laws

$$A \bullet B = B \bullet A \qquad A + B = B + A$$

Absorption Rules

$$A + A \bullet B = B \text{ (note: } \bullet \text{ has precedence over +)}$$

$$A \bullet (A + B) = A$$

$$A + \overline{A} \bullet B = A + B$$

Distributive Laws

$$A \bullet (B + C) = A \bullet B + A \bullet C$$

$$A + B \bullet C = (A + B) \bullet (A + C)$$

Associative Laws

$$A + B + C = (A + B) + C = A + (B + C)$$

$$A \bullet B \bullet C = (A \bullet B) \bullet C = A \bullet (B \bullet C)$$

De Morgan's Theorem

$$\overline{A + B + C} = \overline{A} \bullet \overline{B} \bullet \overline{C}$$

$$\overline{A \bullet B \bullet C \bullet \ldots} = \overline{A} + \overline{B} + \overline{C} + \ldots$$

Appendix B

Cadence End User Agreement

Appendix C

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 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
```