## Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organisation of the Laboratory</td>
<td>3</td>
</tr>
<tr>
<td>Patchboards</td>
<td>5</td>
</tr>
<tr>
<td>Exercise 10211 - 1: Simple Logic Design</td>
<td>7</td>
</tr>
<tr>
<td>Schematic Diagrams and Schematic Capture</td>
<td>15</td>
</tr>
<tr>
<td>Cadence Tools: Walk-Through</td>
<td>19</td>
</tr>
<tr>
<td>The Library Elements Available</td>
<td>33</td>
</tr>
<tr>
<td>Exercise 10211 - 2: Simple Logic on the Gate Array</td>
<td>35</td>
</tr>
<tr>
<td>Verilog: a Brief Introduction</td>
<td>39</td>
</tr>
<tr>
<td>Exercise 10211 - 3: A Seven Segment Decoder</td>
<td>35</td>
</tr>
<tr>
<td>Exercise 10211 - 4: Counters</td>
<td>51</td>
</tr>
<tr>
<td>Exercise 10211 - 5: Traffic Lights</td>
<td>55</td>
</tr>
<tr>
<td>Exercise 10211 - 6: MU0: a microprocessor</td>
<td>59</td>
</tr>
</tbody>
</table>
Organisation of the Laboratory

• These notes apply to the COMP10211 engineering laboratory associated with the corresponding lecture course. Please read them carefully; the laboratory is run as informally as possible, but with the large number of students involved it helps everybody if the following procedures are observed. In particular, it is important that exercises are done according to the timetable and that every effort is made to complete exercises within the scheduled time.

• The scheduled laboratory sessions are on Tuesdays (10.00 - 12.00) (Y), Wednesdays (9.00 - 11.00) (X) and Thursdays (1.00 - 3.00) (Z) and (3.00 - 5.00) (W). Demonstrators (who are typically post-graduate students undertaking research projects) and a member of staff will be available during these sessions to provide assistance. Always attempt exercises without first seeking help, but as soon as you get into difficulty ask for assistance from them. If you do not ask for help soon enough, you may fail to complete the exercises within the scheduled time.

• The laboratory is permanently staffed with a technician who is responsible for maintenance of equipment. Please report any faulty equipment so that it can be repaired. If available, he may also supervises the laboratory outside formal laboratory hours. This means that if you do not complete an exercise within the session(s) scheduled for it you can catch up by attending the laboratory when you have spare time and have demonstration signed off then. The technician may not be able to answer all your questions about the exercises however.

• The first sessions of the COMP10211 laboratory consists of several short introductory exercises designed to familiarise you with basic logic gates and some simple circuits. The rest of the laboratory comprises a larger set of similar exercises which are ‘wired’ inside a programmable gate array.

• Exercise scripts should be read before coming to the laboratory session for that exercise. There is not much practical preparation you can do but you need to understand the exercise and the concepts it is trying to reinforce. Thus, if you do not read the script 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.

• 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 marked. Pro forma answer sheets are available in the laboratory.
• Each exercise requires the work to be checked and marked by a demonstrator as the exercise proceeds. The exercise answer sheets indicate after which parts of the exercise a mark and signature must be obtained. When you are ready to have a section of an exercise marked you must assemble any paperwork required for marking (e.g. circuit diagrams) as specified for each exercise and have your work ready to demonstrate before asking for the work to be marked.

• If you have to complete an exercise outside of the allocated laboratory period then you may get the progress sheet signed by seeing a demonstrator during any scheduled laboratory; provided that there is space available to demonstrate your work.

• Once you have completed the exercise and all sections have been marked and signed the answer sheet along with all other marked material must be handed in. No marks will be awarded unless this is done. The answer sheets can be handed in by putting them into the laboratory “post-box”. Before handing in any written material a report card should be filled in with your name, group {W, X, Y, Z} and exercise number\(^1\) and stapled to it. Ensure that there is a card with your name (only) on it and that all material is stapled together; this will reduce the number of anomalies later.

• **Remember that exercises must be completed and handed in at the end of the session scheduled for that exercise.** This should not be difficult if a small amount of preparation is done.

• Keeping to the schedule gives more time for the laboratory demonstrators to help you, if you need it.

• Please report any script errors or faulty equipment.

• Please do not bring food and drink into the laboratory – spills are inevitable and greasy fingers make keyboards and screens unpleasant for everyone.

• If you have suggestions for improving the laboratory or the manual, please e-mail these to ‘jgarside’.

---

1. Most of the details on this card are no longer in use.
Patchboards

All the exercises in the COMP10211 laboratory are 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 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 a patch board (fig. 1.1). This conceals any ‘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).

![Gate symbols](image)

*Figure 1.1: Laboratory Patch Board*

Circuits are constructed by connecting the gates with wires. Unused inputs are held inactive on the patchboard, but - in general - unused inputs should be tied to a logic level.

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*. At the top and bottom of this column are logic ‘active’ and ‘inactive’ signals. There are some spare LEDs on the right hand side of the board for use as output indicators.

The gate symbols are introduced in exercise 10211 - 1.
Exercise 10211 - 1: Simple Logic Design

Aim

This exercise is an introduction to some of the basic building blocks used in digital systems. It introduces some of the conventions which may be used to describe digital systems. Simple circuits are used as illustrations of both combinational logic and storage elements.

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.

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”; in this manual these may be 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”.

- **Voltage levels** with the values ‘high’ (“H”) voltage and ‘low’ (“L”) voltage.

In the initial parts (1-4) of this exercise Boolean logic representation will be used. 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 much useful. To do useful things two or more signals must be combined. In digital electronics simple combinations are performed by 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. 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 the others, 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 others, 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 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 into two; the left-hand side which enumerates all possible values of the inputs and the right-hand side which defines what value the output will have when the inputs have the values listed on the same row. For example, the truth tables for a 2-input AND gate are shown in figure 2.1; the inputs are labelled A and B and the output is labelled C. There is no functional difference between the two truth tables shown, the only difference is that one uses a Boolean logic representation while the other uses a binary arithmetic representation.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

*Figure 2.1: 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 fully 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

When drawing logic diagrams signal interactions are represented by symbols which indicate the function performed. 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, readily recognisable symbols, as shown in figure 2.2.

*Figure 2.2: ISO Symbols for AND and OR functions*

Whenever these symbols appear a logical AND or a logical OR function has been implemented. The figure illustrates two-input gates where in each case “A” and “B” are the inputs and “C” is the output (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 which swaps a signal from one state to the other, thus a ‘TRUE’ becomes NOT(‘TRUE’) (or ‘FALSE’) or - interpreting the signals in a different fashion - a ‘1’ becomes NOT(‘1’) (or ‘0’). The usual symbol for an inverter is shown in figure 2.3. The ‘bubble’ (i.e. the little circle) in this symbol is used generally to indicate an inversion.

![Figure 2.3: The Inverter](image)

Sometimes gates perform both a logical function and an inversion; thus for example there is a ‘NAND’ gate (an abbreviation of ‘Not AND’) which performs the same function as an ‘AND’ gate followed by an inverter (fig. 2.4). Similarly a ‘NOR’ gate can be defined.

![Figure 2.4: Function and ISO symbol for a NAND gate](image)

Connections between functions can be signified by using lines on the logic diagram. Sometimes it is necessary to be able to identify connections. The easiest way of doing this is to name the connection. This name is usually shown on a logic diagram by placing it adjacent to the line forming 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 to 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** which represents the function. This is a very compact method of representing a logic function.

The basic symbols used are:

\[
\begin{align*}
A \cdot B & \quad \text{for} \quad A \text{ AND } B \\
A + B & \quad \text{for} \quad A \text{ OR } B \\
\overline{A} & \quad \text{for} \quad \text{NOT}(A)
\end{align*}
\]

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

\[
\overline{A \cdot B} \quad \text{means} \quad \text{NOT}(A \text{ AND } B)
\]

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

School of Computer Science

University of Manchester
Practical

This exercise is marked on the basis of the pro-forma answer sheet provided at the back of this manual. Fill in the sheet as you complete each part of the exercise, ensuring as you do that the relevant parts are signed by a demonstrator. When completed the answer sheets should be stapled together with a report card and placed in the laboratory “post-box”.

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

1 Identify a ‘NOR’ gate and verify its behaviour. Complete the truth table on the answer sheet.

De Morgan’s Theorem

De Morgan’s theorem allows AND functions to be translated into OR functions and vice versa. Briefly it states that:

\[ \overline{A \cdot B} \equiv \overline{A} + \overline{B} \]

![De Morgan's Theorem Diagram]

Practical

2 Construct and test a NOR gate on the patchboard without using either the NOR or OR components provided. Sketch a logic diagram of your circuit on the answer sheet.

3 The symbols below denote exclusive-OR and exclusive-NOR gates. An exclusive-OR gate is similar to an (inclusive) OR gate, but it ‘excludes’ its output from being true when both its inputs are true.

![Exclusive-OR (XOR) and Exclusive-NOR (XNOR) Symbols]

Exclusive-OR (XOR) or not-equivalent gate

Exclusive-NOR (XNOR) or equivalent gate

The normal symbol for exclusive-OR is a “\(\oplus\)"

Investigate and complete the truth table for the exclusive-OR gate. These gates are also known as “not-equivalent” and “equivalent” gates. Why?
4 There are various methods of constructing exclusive-OR gates from the simpler inverters and AND and OR gates. Devise one of these, construct and test it. Demonstrate that this is working and record the circuit diagram on the answer sheet.

Simple Adder

As an example of the use of logic circuits in computers consider the addition of two binary numbers. The simplest, most general, case is the addition of two 1-digit numbers and a carry-in from the previous case. The results of this operation produce a 1 digit answer and a carry out to the next stage; try a pencil and paper addition of two decimal numbers if unconvinced. In binary addition the digits are only allowed values of “0” and “1” and are called “bits” The truth table for binary addition is shown in figure 2.5.

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>OUTPUTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digits</td>
<td>Carry</td>
</tr>
<tr>
<td>C_i-1</td>
<td>C_i</td>
</tr>
<tr>
<td>S_i</td>
<td></td>
</tr>
<tr>
<td>A_i</td>
<td></td>
</tr>
<tr>
<td>B_i</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A_i</th>
<th>B_i</th>
<th>C_i-1</th>
<th>C_i</th>
<th>S_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2.5: Truth Table for a Simple Adder

From the truth table of figure 2.5 it is possible to construct the logical equations needed to generate the functions S_i and C_i. However, since this is the first exercise a slightly easier approach will be indicated. The three input, two output circuit can be split into two identical two-input, two output circuits as shown in figure 2.6; such circuits are known as half-adders.

Figure 2.6: A Full Adder built from two Half Adders

Figure 2.6 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. 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

5 Construct the truth table for a half adder circuit and deduce the logical equations for the two outputs required; the equations that you deduce should be obvious from the truth table. Build, document and test a full adder circuit from two half adders. There are enough gates for this on the patchboard, although some ingenuity may be required.

When drawing the schematic of this circuit draw the actual gates used in its construction.

Flip-Flops

The adder circuit is an example of pure combinatorial designs. The value of the outputs of these 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 a flip-flop. The rest of this exercise is an investigation of flip-flops and some of the ways in which they can be used.

Set-Reset Flip-Flop

The Set-Reset, or S-R, flip-flop is the simplest form of a flip-flop. It has two control lines. When the set line is asserted the output will go to a “1” while when the reset line is asserted the output will go to a “0” Once the output has been set to a particular value it will remain in this state until the other control line is asserted. An S-R flip-flop constructed from NOR gates is shown in figure 2.7. This method of drawing the circuit emphasises the essential difference between sequential circuits and combinatorial circuits, that is sequential circuits have a feedback path from output to input.

![Figure 2.7: S-R Flip-flop: Circuit and Symbol](image)

It is worth investing a little effort in the analysis of the circuit. The logical equations for the two outputs may be written:

---

University of Manchester

School of Computer Science
\[ Q_1 = \text{RESET} + Q_2 = \text{RESET} \cdot Q_2 \]
\[ Q_2 = \text{SET} + Q_1 \]
therefore
\[ Q_2 = \text{SET} + Q_1 \]
\[ Q_1 = \text{RESET} \cdot (\text{SET} + Q_1) \]

From these equations the behaviour of the device may be tabulated as shown in figure 2.8.

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q_1</th>
<th>Q_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Q1</td>
<td>Q2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0^a</td>
</tr>
</tbody>
</table>

\(^a\)This state is not normally allowed

*Figure 2.8: Tabulated Behaviour of a S-R Flip-flop*

This table is often called a truth table, although it should more correctly be called a transition table. It can be seen that, apart from the disallowed state, \( Q_1 \) and \( Q_2 \) are the same logical signal with opposite polarity. For this reason the outputs are often labelled \( Q \) and \( \overline{Q} \). Strictly this is incorrect and often causes confusion.

**Practical**

6 Using NAND gates construct a flip-flop which has active low set and reset lines. Verify the transition table and ensure that you understand how the flip-flop works. Reproduce the circuit diagram on the answer sheet.

**Clocked S-R Flip-Flop**

The circuit of figure 2.9 has an additional enable input so that the S and R inputs are active only when the enable is high. When the enable goes 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 S-R flip-flop from an element used in asynchronous systems to one useful in synchronous systems.

**D Latch**

A D latch combines the S and R inputs of an S-R latch into one input, thereby eliminating the \( S = 1, R = 1 \) conflict. When the clock is high the output follows the D (Data) input; when the clock is low the data is latched or stored. A circuit for a D latch is shown in figure 2.10.
Commercially available D latches are packaged in dual, quad, hex and octal forms, often having additional asynchronous set and reset inputs. They are sometimes known as transparent latches because - when the enable signal is active - the data flows from D to Q unchanged.

**Practical**

7 Construct and test a D latch using the flip-flop design from part 6 of this exercise. Remember that your S-R flip-flop is has active low inputs, so you can’t copy directly from figure 2.10. Document and demonstrate your circuit.

**Don’t Forget** to hand in your exercise progress sheet.
Schematic Diagrams and 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 which shows functional blocks and the interconnections between them and - on the small scale - should be familiar from the preceding exercise. 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 of manageable sizes. This then forms a hierarchical structure (Fig. 3.1). 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).

---

1. Also known as a ‘circuit diagram’ or just as a ‘schematic’.
2. Assuming we are using a particular electronic implementation. Almost all logic gates are currently electronic because these 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.
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 to this 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.

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 which 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 without the need for physical wiring.

### Drawing Layout

The actual layout of a logic diagram is an art which comes with experience. Clarity is a major objective and there are a number of guidelines available which 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, 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 3.2.

\[
\text{Data<15:0> ———— Data<15:0>}
\]

\[
\text{Data<7>}
\]

*Figure 3.2: 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 indicating real connections clearly with a heavy dot {Fig. 3.3}. Sometimes lines which cross
without connection are shown with a ‘bridge’ in the drawing; this style is now out of fashion and its use is discouraged.

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.

On designs using multiple devices 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).

**Jargon**

The greatest difficulty in entering schematics 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.

---

1. Not directly relevant to COMP10211 but worth remembering for the future.
2. ...or most other subjects
**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 similar 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. "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 the 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 which 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 ‘notes’ are white in Cadence schematics.*

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

**Library components are green.**

**Any items that are currently selected are outlined white.**
Cadence Tools: Walk-Through

This exercise is intended as a first introduction to the Cadence design suite. 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 is sensitive to the Window manager you use. It will work in AnotherLevelUp and in Fvwm95. Do not use other window managers as cadence may act unpredictably and you may find that not all mouse buttons work correctly all the time.

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 COMP10211
```

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

Creating a Design

Starting Cadence

Start Cadence by typing `start_cadence COMP10211` in a command shell window. The Cadence Integrated Circuit Design and Simulation window (icds) will eventually start (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 would allow you to see more lines of text, but you will not normally need to do that.

```
Figure 4.1: Cadence icds
```
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 which can reverse the last few commands.

Copying, Moving, Deleting:

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 tools, either accessing them directly from icds, or via the Library Manager. Do not attempt to move, copy or modify them directly from a command line or file manager.

Components can be opened or created directly from the File menu Tools → Library Manager…).

Closing down

When you have finished, close down Cadence before logging off. If you simply close down your X windows session 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…

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 walk-through you will enter and simulate a new design.
The first thing to do is open a design. When you have some designs to choose from this can be done using the Library Manager (in icds click Tools ↓ Library Manager…), but for creating a new design use File ↓ New ↓ Cellview… which will open the Create New File window as shown in Figure 4.2. Select the library “COMP10211”, type the name you want to use for the new component into the “Cell Name” text box and click ‘OK’. It is recommended that the first name you use is “halfadder”.

You can use alphabetic and numeric characters for a filename but do not include spaces, dashes (‘-’) or any ‘special’ characters such as *,$,/ etc. 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. It is sensible to use an alphabetic character to start a filename, not a number, much like a variable name in software.

A new window “Virtuoso Schematic Editing” should open, in which you can create your design (See Figure 4.3).

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 which 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. To select a component, click the Browse button to the right of the Library text entry box, which will open the Library Browser, enabling 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.

- Click on the component AND2 (an AND gate with 2 inputs) and place this component on the schematic area by moving the cursor until the gate is in the chosen position 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.

- 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.

Pressing ‘Delete’ or ‘Del’ on the keyboard or ‘Edit \(\downarrow\) 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 \(\downarrow\) Undo (or \(u\) on the keyboard), which will undo the previous command. Several successive actions can be undone.

- Now you are going to add the inputs and outputs to the halfadder. Inputs and outputs are called ‘pins’. Select Add \(\downarrow\) 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.

Note - 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. Don’t start pin names with numbers ‘1A’ etc.

- Follow the same procedure to place two output pins, ‘carry’ and ‘sum’.

- Now the components are positioned correctly they must be joined with wires. Wires must run connect either to pins on components (red squares), to input or output pins or to other wires. Select Add \(\downarrow\) 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 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.

You have now completed the schematic design of the halfadder. Save the schematic: Design \(\downarrow\) Check and Save, or ‘X’
Look in the icds window and confirm that the design checked and saved correctly – you should see messages stating:

Schematic check completed with no errors.

“COMP10211 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 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 which just stops in space.

**Designing the Symbol**

You are now going to build a symbol for the half adder. The symbol is the drawing 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, 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:

- In the Schematic Editor select **Design ↵ Create Cellview ↵ From Cellview**
- Check that ‘From View Name’ reads “schematic”, and ‘To View Name’ reads “symbol”, and click **OK**.
- If a ‘Symbol Generation Options’ window appears, you should not need to change anything, in which case just 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 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.

• You can modify the symbol in many ways - move pins, move text labels, replace the symbol with a different shape etc. using the menus edit and Add. Try Add Shape Polygon for example. You might want to move the Instance Name so that it will appear inside the symbol - if so, place the “@” of “@InstanceName” somewhere convenient.

• If you do modify the design of the symbol, you should ensure that the selection box is still the correct size. You can do this by Add Selection Box Automatic, which will resize it.

• 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.

• 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.

• You did write a specification before you started designing, didn’t you? Answer: “Yes, I completed a truth table for a half adder in part 5 of Experiment 10211-1”.

Figure 4.6: A possible symbol 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. ‘What is intended’ is defined by a specification, which should be created before 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

- Open the schematic design of the system you want to simulate (the half-adder).
- Select **Simulation**. This will open the “Virtuoso Verilog Environment for NC Verilog Integration” window as shown in Figure 4.7.

![Figure 4.7: Virtuoso Simulation Window](image)

- Click the ‘Initialize Design’ and then the ‘Generate Netlist’ buttons. You may see a popup window asking if it is OK to continue, if so click ‘Yes’ or ‘OK’

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.

- Select **Edit Test Fixture**. This will open a text editor window to allow you to edit a 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).

- #xx is the instruction to wait for a fixed time (specified in ns, or $10^{-9}$s), so to wait for 10ns the instruction is #10.
- 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 below in Figure 4.8.

- Save the file that you have edited.

**Summary of useful Verilog instructions:**

- You can add comments. Comments begin //
- #x creates a delay of (waits for) x ns
- a = x; sets the value of input ‘a’ to x.
- You need a $stop; command at the end to tell the simulator to stop simulating once it has reached the end of the file.
- Do not delete any of the code that is provided, just include your code in the appropriate place.

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.

- 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 (it may well be hidden) which should look as shown in Figure 4.9. 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)
// Basic half adder test
// Inputs a,b
// Outputs sum, carry

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

// Cycle through the other input combinations,
// pausing at each input state
   b = 1;
   #10
   b = 0;
a = 1;
   #10
   b = 1;
   #10
$stop;    // Stop simulating

Figure 4.8: Commands to Include in the Half-Adder Verilog Script

- The Design Browser (fig. 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.

- 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.
   - Click the “Send to Waveform” button (see Figure 4.9).
   - Find SimVision waveform viewer, as shown in Figure 4.10 (it should already be open).
   - When the SimVision window opens, there will be no traces visible. Click the “Run” button, and the waveforms (or traces) should appear.

Compare the simulated behaviour of your circuit to its specification. Does it perform as required?

Notice that when the inputs change there is a delay before the outputs change. Why is this?
Figure 4.9: SimVision Design Browser

- Send To Waveform
- Top level inputs (a, b) and outputs (carry, sum)
- Top level - click ‘+’ to expand the top level.
- Instances - in this case individual gates.

Figure 4.10: SimVision Waveform Viewer

- Run
- Reset
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.

In the Schematic diagram of your half adder, look at the instance values of the gates. 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. 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 \( \downarrow \) 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 LMB

- Check and save the design in the schematic editor
- Close the simulator.
- Restart the simulator, and repeat the process described above, except that you should not need to recreate the stimulus file.
- When simulating, add the outputs of appropriate gates to the waveform viewer, then Run the simulation again.
- 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.

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 which suit you better.

At this point proceed to exercise 10211 - 2 which extends this design further.
Buses

In a programming language such as Java there are some basic types which 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 which uniquely identifies each bit. [This is like an array in a high-level programming language.]

It is not compulsory, but the normal convention for numbering bits in a bus is to start from index 0 from the least significant bit. Thus a 32-bit bus might be called “data[31:0]”.

Note: using ‘[ ’ ’]’ is the usual convention, however Cadence uses ‘<’ and ‘>’ within the Virtuoso schematic capture tool, so that convention will be adopted for the remains of this section.

Figure 5.1 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).

![Figure 5.1: Example bus signals](image)

The names also serve to show how signals are connected. Thus the vertical bus in figure 5.1 carries only the more significant half of the horizontal bus. Whilst it is often convenient to show the buses connected (as above) it is not necessary; any set of wires with the same name are connected, thus the individual signals shown are also present in the horizontal bus section.

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

Every item on a schematic will have a unique name. For example, if you instantiate an AND2 gate it will have a Cell Name name (‘AND2’) but it will also have an View 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/when you ‘label’ a net 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 5.2). Using this technique 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.

![Figure 5.2: How to make a 32-bit 2:1 multiplexer](image)

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 5.2) 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** \(\rightarrow\) **Properties** \(\rightarrow\) **Objects**... to open a dialogue box. The appropriate width can then be appended to the View Name; for example if the name was “I29” this can be altered to “I29<31:0>” if 32 copies are desired.
The Library Elements Available

All the designs in this laboratory are built from library elements. Most of these are held in a directory which is available as “spartan3”; the ‘local’ cells are contained in a different library – “ENGLAB_10211”. You may not modify or add to this library but you can build your own library which 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 ‘builtin’ 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_10211)

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
Exercise 10211 - 2: Simple Logic on the Gate Array

Aim

This exercise is intended as a “hands on” introduction to the schematic capture and gate array layout software. A further 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. It is, deliberately, quite a short exercise. Before starting the practical described below you should also have read through the sections on Cadence and attempted the “walk-through” of the software. These should be used as guides until you feel familiar with the systems.

Before attempting to run the software for the first time some actions must be performed to enable it to run and modify your own libraries. These are covered in the subsection Software set up in the section the Cadence Tools: Walk-Through beginning on page 19 of this manual. Follow the instructions there carefully.

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

Practical

Perform the ‘walk-through’ in the preceding section. Use this as an opportunity to ‘play’ with the software and find out how it works (and where it doesn’t).

1 Using the half adder from the walk-through prepare a schematic for a full adder circuit and simulate its operation. A full adder can be made from two half adders and an OR gate, and should have three inputs A, B and Carry In, and two outputs Sum and Carry Out (fig. 2.6).

![Figure 6.1: A Full Adder built from two Half Adders](image)

As an exercise try selecting one of the half adder symbols and executing the function Design ➔ Hierarchy ➔ Descend Edit … function, (which is most easily obtained using the ‘E’ keyboard accelerator). This will allow you to edit either the
symbol or the schematic and then Design ↓ Hierarchy ↓ Return up the hierarchy. This function is useful with your own components (and, possibly, the local library “ENGLAB_10211”) but the basic gates are ‘bottom-level’ models.

Note that the labels ‘inside’ the components (\{a,b,c,s\} in figure 2.6) are local to that component and therefore not implicitly connected together on the higher level diagram.

A Multi-bit Adder

2 When the full adder is complete create a symbol for it. Use this to build a 4-bit binary adder. This should take two 4-bit numbers as inputs (\(A_{3,0}, B_{3,0}\)) and an input carry (which should be wired to ‘ground’) and produce a 4-bit output (\(S_{3,0}\)) and an output carry bit. Simulate this circuit. Find an input combination that causes a change at the least significant inputs to propagate up to the output carry and observe the propagation delay with SimVision. Notice how the carry ‘ripples’ along the chain of full adders; for this reason this design is known as a “ripple carry adder”.

Show this design to a demonstrator.

Multiplexers

Multiplexers are also known as data selectors, which is a good description of their function. A multiplexer is a device which selects its output from one of a number of inputs, according to an input code (i.e. a selection). For example a 4 to 1 multiplexer would have four data inputs and two selection inputs; its truth table is shown in figure 6.2

<table>
<thead>
<tr>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
<th>S1</th>
<th>S0</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 6.2: Truth table for a 4 to 1 multiplexer

Multiplexers are very common logic elements and typically have a symbol something like that shown in figure 6.3. Because ‘multiplexer’ is a long word it is often abbreviated to “MUX” and is sometimes omitted altogether in diagrams where the context makes the identity of the block clear.
Practical

3 Design, implement and simulate a 2:1 multiplexer. Create a symbol for this.

4 Create a quad 2:1 multiplexer from your previous circuit (i.e. four multiplexers with a common control signal as a single functional block).

Keep both the schematics and the symbols as neat as possible; you will be using this component in later exercises.

Show your circuit to a demonstrator and ensure that the progress sheet is signed off.

Attach your schematic diagram(s) to the answer sheet for handing in. You should include a complete set of all the work you have done, i.e.:

- 4-bit adder symbol
- 4-bit adder schematic
- full adder schematic
- half adder schematic
- quad multiplexer symbol
- quad multiplexer schematic
- 2:1 multiplexer schematic

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 selected
- 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 Ifprt to send to the least busy printer in the student resource centre.

**Keyboard shortcuts in Cadence**

You will have noticed that many menu and sub-menu 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.
Verilog: a Brief Introduction

Verilog is a Hardware Description Language (HDL). Its initial use in this laboratory was to provide a simulation environment to control and test schematic designs. Later we use it to describe some real hardware and synthesize it into logic circuits.

The table below 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 HDL1 most of the time. However, just like software, it is important to know what goes on ‘underneath’ if your design is to be efficient.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gates</td>
<td>Machine instructions</td>
</tr>
<tr>
<td>Schematics</td>
<td>Assembly language</td>
</tr>
<tr>
<td>HDL</td>
<td>High-level language</td>
</tr>
<tr>
<td>Simulator</td>
<td>Debugger</td>
</tr>
</tbody>
</table>

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 which 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.

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

---

1. There are only two widely used HDLs at present. The other one is called “VHDL”.

School of Computer Science  
University of Manchester
The simplest form of stimulus file will look something like this:

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

This is a very small subset of what the language can do – there will be some more examples later – but serves to illustrate how input signals can be set and pauses inserted. The pauses are necessary for any input changes to propagate through the circuit.

Each time the code above is invoked it will:

- wait 100 ns (the variable “a” will be undefined at this time)
- set “a” to zero
- wait 100 ns for any changes to propagate through the circuit
- set “a” to 99 (decimal)
- wait 100 ns for any changes to propagate through the circuit
- stop simulation and return control to the user

The “begin” and “end” keywords act in the same way as ‘{’ and ‘}’ in Java, bracketing their contents into a single statement which 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. 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
8'd99 // Decimal
```

Any number in front of the quote symbol is always in decimal and specifies the number of bits in the number. 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 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 something like this:

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

Clk = 0; // Start with clock low
a   = 0; // Set variable “a” to zero
#50  // Wait for half a clock period
Clk = 1; // Take clock high
#50  // Wait for other half of 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
```

This gets a bit tedious as the file size grows though. Instead, a different construct may be used:

```verbatim
initial Clk = 0;
always
begin
#50
Clk = ~Clk;
end

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

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
```

School of Computer Science
University of Manchester
This example exploits the capacity for parallel programming in Verilog. There are three statements. The first simply initialises the clock to zero (otherwise it would be undefined) at the start of the run. The second – the “always” statement – loops, repeatedly waiting for 50 ns then inverting 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 statement contains the various input values which 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 7.1.

![Figure 7.1: Effect of parallel threads](image)

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 which 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.

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.
Synthesizable Verilog

Although Verilog has so far only been used for simulation stimulus files it is a Hardware Description Language (HDL); it can be used to specify circuits which can then be synthesized. 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. This is 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 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 associated with symbol and incorporated in larger designs.

Module syntax can be summarised with the following modulo-7 counter:

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

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

assign zero = (count == 0);

endmodule
```

Note that all the statements are executed in parallel, as required.

Logic

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

Simple combinatorial logic

Use continuous assignment instead of gates.

```verilog
wire x;
assign x = a & (b | c);
```
Complex combinatorial logic

If necessary, complex logic can use ‘reg’ and blocking assignments.

```verilog
reg q;
reg_leap_year;

always @ (select, in0, in1, in2, in3)
case (something)
  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;
  // This could be completed but division is a bit hard!
end
```

Sequential logic

State storage should generally use D-type (edge triggered) flip-flops and use non-blocking assignment.

```verilog
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
```

Structural

This is more detail than needed for this course, but Verilog modules can be nested.

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

fred instance_name (.clock(my_clock), // Instantiate a ‘fred’
  .enable(my_en),
  .count(count_out),
  .zero(zero_detect));
```
Exercise 10211 - 3: A Seven Segment Decoder

Aim

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.

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.

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 pieces of electronic equipment such as calculators, digital watches, etc. To make an easily legible display these use a 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 below.

Because there are only ten possible digits the number can be represented as a 4-bit code, as:
\[
\log_2(10) = 3.32 \\
\text{or} \\
2^4 > 10
\]

It is generally more convenient to use the minimum number of bits to represent any quantity because this reduces the wiring and circuitry required for both storing and operating on the digits. Thus inside a machine a single digit 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.

Select ‘File ↓ New ↓ Cellview…’ and in the dialogue box set:

- View Name: “functional”
- Tool: “Verilog-Editor”

You will need a name for the design as well.
This should start a text editor and generate a very simple template. 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.

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.

**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 8.1.

```
// Comments as to function of module, author, date, etc.
module <name> (  
    input wire [3:0] digit,
    output reg [7:0] display  
);  

// Internal variable declarations - if any
always @ (digit)  // "Sensitivity list"
case (digit)
  0: display = 8'b0011_1111;
  // ... Fill in the rest here.
  default: display = 8'b0000_0000;
endcase

endmodule
```

*Figure 8.1: Some example synthesisable Verilog code*

(This fragment is available from “/opt/info/courses/COMP10211/Verilog_examples/Ex_seven_seg.v”. It can be ‘include’d from the editor.)

Points to note:

- The module has a name, followed by a list of inputs and outputs.
• Bus widths are specified using ‘[’ and ‘]’ characters. The output bus is eight bits wide. Segment a is the least significant bit; the most significant is the decimal point.

• 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 “digit” changes the block below will recompute. In practice this means that a combinatorial logic block will be produced.

• The ‘case’ statement selects a following clause which matches the input. Here only the ‘case 0’ clause is shown.

• The number assigned to ‘display’ is an 8-bit value, specified in binary. ‘_’ characters are ignored by the compiler but make the number easier to read.

• ‘default’ is used to catch any cases which 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 8.2 shows the segments which must be lit to display each of the digits 0-9.

![Seven segment patterns for the digits 0..9](image-url)
Practical

1 Design and implement a seven segment decoder which translates a 4-bit BCD code into a human-readable display. Create a symbol for this and instantiate your design in a higher level schematic, allowing it to be demonstrated on the lab FPGA board.

If you wish, 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 8.3.

You should simulate your circuit. The easiest way to invoke the simulator is probably to start from icds and use: ‘Verilog Integration ‹ NC-Verilog…’. From ‘Browse…’ you can select your design.

2 To interface the circuit to the ‘outside world’ create a new schematic and instantiate your decoder in this. You will also need to instantiate the cell ‘Board’ from the library called “ENGLAB_10211” which contains the interface definitions.

‘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 interfaces on ‘Board’ are:

Outputs (located on 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 the 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 8.3: Seven segment patterns for the digits A..F](attachment:image.png)
Inputs (located on the bottom):

- **Display_en<5:0>** Enables for each seven segment display. An enable must be ‘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.

Connect your circuit to a seven segment display symbol provided by the ‘Board’ component from the “ENGLAB_10211” library. Choose four buttons on the keyboard – we suggest using Key_row4<3:0> – and wire these to your input. Pressing combinations of these keys should not produce the appropriate display.

**Unused connections**

There will be some unused inputs on ‘Board’ in this exercise. In a real circuit is important that all inputs are defined and 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’.

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 a Instance Name “I5” this could be edited to “I5<7:0>”. The outputs can then be connected to input bus pins as appropriate.

Don’t forget to enable the appropriate display by wiring its display_enable signal ‘high’! The symbol ‘VCC’ 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. To compile the design first ensure that you have checked and saved the design from 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 on 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 appropriate window to see that there are no errors and the compilation is “Done”. If everything is okay this should have gener-
ated the bitfile which is used programme the FPGA. [Errors at this point can be difficult to
diagnose; you may need to consult a demonstrator if you have problems.]

**Design download**

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

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.

   *Test the implementation of the design on the circuit board. If it is functioning – if you have debugged the design by simulation it should work first time – show it to a demonstrator, otherwise find out what is wrong and fix it using the simulator.*

   *Get the pro-forma sheet signed and hand this in with a listing of the Verilog source code, a simulation trace of the module and a ‘top level’ schematic.*

**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 to one of the lfprt printers in a font size which 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 10211 - 4: Counters

Aim

To investigate time dependent circuits. This exercise introduces D-type flip-flops; these are then used to build up counters. Counters are examples of simple finite state machines, 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.

Revise the operation of the edge-triggered D-type flip-flops.

Sequential circuits in Verilog

The D-type, edge triggered flip-flop is usually the most ‘convenient’ storage element to design circuits with and Verilog is optimised for their use.

```verilog
module counter (input wire clock, input wire enable, output reg [3:0] digit);
    always @ (posedge clock) // “Sensitivity list”
        if (enable == 1)
            digit <= digit + 1;
endmodule
```

Points to note:

- The sequential ‘always’ is just sensitive to ‘clock’ because other inputs only have influence at that time.
  ‘enable’ is read when there is a positive edge on ‘clock’.
- ‘digit’ is a 4-bit register which 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.
• In this example there is no ‘else’ clause to the ‘if’; if it is omitted then no alternative action will take place (i.e. ‘digit’ will remain unchanged, here).

• This is a modulo-16 counter. There is no behaviour specified other than the possibility of an increment. However ‘digit’ is only four bits, so after reaching the value 15\textsubscript{10} it will ‘increment’ back to zero.

For illustration purposes, figure 9.1 shows a schematic with the same function as this code. This would be the expected outcome of synthesizing this Verilog.

![Figure 9.1: Circuit diagram of a 4-bit synchronous modulo-16 counter](image-url)
Practical

1 Design a circuit which displays the digits “0” to “9” repeatedly on a seven segment display. The circuit should run autonomously and should display only these digits in the correct order. The numbers should each be present for exactly 1 second so that they are legible whilst the entire sequence should be displayed in a short time. Only one of the displays should be used, the others should be blank.

The simplest way to perform this function is to use a synchronous modulo-10 or decade counter with its outputs connected to the seven segment decoder produced in the previous exercise. For neatness and to promote reusability this counter should be designed as a separate module with its own symbol.

Extend the function of the counter to include a ‘direction’ control bit (which can be wired to a keyboard button). The counter should then be an up/down counter, depending on the state of this bit.

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.

Practical

For the physical implementation ‘Board’ in the “ENGLAB_10211” has a number of clock outputs which may be used. The frequency should be apparent from the pin names.

2 Produce a counter which cycles through (and displays) the states 0 to 999. Build this using separate counters for each digit. You may have to modify your original design slightly, for example to include a ‘ripple carry output’. This is a signal which indicates that the counter is about to “wrap-around”. In the case of a decade counter it would be active only in state “9”. This output may used to enable the clock on a second counter (Fig. 9.2). 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”).

You should only need one type of digit counter for all purposes.

Demonstrate your circuit running and hand in a signed pro-forma answer sheet together with schematics, Verilog listings, a stimulus file and an example waveform plot from the 0-999 counter test.
Figure 9.2: Timing Diagram for a Decade Counter with a Ripple Carry
Exercise 10211 - 5: Traffic lights

Aims

To give an opportunity to run through a hardware development process from design to implementation.

To construct a small system exploiting a selection of the techniques from earlier exercises.

There is more design involved in this than most of the previous exercises.

Finite State Machines

A counter is a simple finite state machine (FSM). Here is another way of specifying a counter:

```verilog
module FSM (input wire clock, input wire enable, output wire [3:0] digit);

reg [3:0] state;

always @ (posedge clock)
if (enable == 1)
case (state)
  0: state <= 1;
  1: state <= 2;
  2: state <= 3;
  3: state <= 4;
  4: state <= 5;
  5: state <= 6;
  6: state <= 7;
  7: state <= 8;
  8: state <= 9;
  9: state <= 0; // Catch anything possibly missed
default: state <= 0;
endcase
assign digit = state; // Alias, for illustrative purposes
endmodule
```

Here an internal variable ("state") has been used rather than ‘exposing’ the output “digit” directly; this is a useful technique when the output is not identical to the internal state.

This is not a particularly ‘nice’ way to write a simple counter (although it would work just the
same) but may be a more useful type of structure when making a more complex FSM. In particular, if decisions need to be made in specific states this provides a straightforward way of representing the desired function.

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. An additional consideration is that there may be other ‘forbidden’ states in the system. For example a decade counter implemented as BCD will have four flip-flops and so can be in one of \(2^4=16\) possible states. It is important to consider what happens if the system ‘gets into’ one of the six ‘forbidden’ states (for example as a result of poor initialisation or a transient fault). 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.

Figure 10.1 shows a possible state diagram controlling a pedestrian crossing. Note that: a) several different states produce the same visible output, allowing the duration of (e.g.) the crossing time to be set, and b) there are alternative paths to state 1 so that the pressing of the ‘start’ button can be recorded whilst the traffic is given a chance to flow.

![State Diagram for a 'Pelicon' pedestrian crossing](image)

<table>
<thead>
<tr>
<th>Lights</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>G_R</td>
<td>0, 8, 11, 12</td>
</tr>
<tr>
<td>A_R</td>
<td>1</td>
</tr>
<tr>
<td>R_R</td>
<td>2</td>
</tr>
<tr>
<td>R_G</td>
<td>3, 4, 5</td>
</tr>
<tr>
<td>A_G</td>
<td>6, 7, 9, 10</td>
</tr>
</tbody>
</table>

*Figure 10.1: Possible State Diagram for a ‘Pelicon’ pedestrian crossing*
Practical

Design, develop and demonstrate a sequencer for EITHER a pedestrian crossing, based on the state diagram shown, OR a set of traffic lights such as would control a crossroads.

In the former case you might like to customise the timings, add a ‘wait’ indicator, implement the flashing amber/green indication, …

In the latter case should be two ‘sets’ of lights, interleaved in the usual sequence with short ‘all red’ periods as appropriate. When a set of lights is green it should stay that way for a longer time than – for example – it stayed amber. Input buttons can represent cars arriving on each road. In the absence of ‘traffic’ the lights should still run; however if an input is signalled the lights should switch sooner than they would otherwise, but not until a reasonable green period has elapsed. (We trust you to think of a sensible definition of “reasonable”!)

Process

• Sort out your state diagram first. You may wish to modify figure 10.1 to your own specification.

• You will want a top-level schematic with the ‘Board’ and a control module written in Verilog. (Tool: “Verilog_Editor” View Name: “functional”)

• Develop the control module systematically:
  - Decide on the input/output interface
  - Declare any internal variables
  - Code the FSM from your state diagram
    (Don’t forget the ‘default’ if you don’t specify every possible state.)
  - Assign outputs using the internal state and, possibly, input states
    (Could use continuous assignment; could use blocking assignment)

• Possibly add extensions
  - E.g. what states would illuminate the ‘WAIT’ sign at a crossing?
  - How might a flashing light combination be introduced?

• Test your design with the simulator.

• Add your module to the ‘top-level’ schematic to provide lights and a clock.
  - As an initial suggestion, the 1 Hz clock may be suitable

Hint

• More than one internal state may produce the same set of visible outputs.
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 (xyzzy)
    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.

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’.

There was an example of this technique in the lectures.

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.

Hand In

A signed demonstration sheet

A sketch of the state diagram you implemented

Your top-level schematic

Your Verilog listing(s)

Any other ‘interesting’ bits of the design you may have developed
Exercise 10211 - 6: An MU0 microprocessor system

Aims

To reinforce the development processes encountered earlier.

To experience some higher-level system design and some larger systems.

To demonstrate that microprocessors are just finite state machines.

The microprocessor system

In this exercise you are asked to construct a complete computer on the FPGA and use it to run a demonstration programme. To save time, a several of the symbols and some of their contents have been prepared already. You will need the knowledge – and some of the circuits – you have acquired from previous exercises.

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

Practical

This practical is divided into stages. Attempt as many parts as you have time for – hopefully at least the first two stages.

For this exercise, create new work in your MU0_lib working library.

Stage 1

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

To help with this you have lecture notes which show a datapath implementation. You are also provided with several (hopefully useful!) parts starting with the part MU0. This includes a control unit and some datapath elements. An initial datapath layout is given to save time; you might need to modify this.

Some of the constituent parts are not provided: these include multiplexers and registers. 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 p. 34 of this manual.

There is an ALU but its (16-bit) adder has been omitted.

You should be able to provide multiplexers from your previous exercises. Take care that the multiplexer selects are the the ‘right way up’.
Control logic

The control logic provided is constructed in schematic form. It is intended to provide all the control signals you will need although you may need to modify it if your datapath requires something slightly different.

The control states are shown in figure 11.1; states alternate unless the processor is halted.

The control logic requires the following inputs:

- Clock
- 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:

- $PC_\text{En}$ – Programme Counter write enable
- $Acc_\text{En}$ – Accumulator write enable
- $IR_\text{En}$ – Instruction Register write enable
- $X_{\text{sel}}, Y_{\text{sel}}$ – ALU input selects
- $Addr_\text{sel}$ – Address output select
- $ALU_{\text{fn}}<1:0>$ – ALU function select:  
  
<table>
<thead>
<tr>
<th>Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>$X + Y$</td>
</tr>
<tr>
<td>11</td>
<td>$X - Y$</td>
</tr>
<tr>
<td>10</td>
<td>$X + 1$</td>
</tr>
<tr>
<td>00</td>
<td>$Y$</td>
</tr>
</tbody>
</table>
- Halted – status: the processor has halted

Figure 11.1: MU0 control state diagram
Processor interface

The microprocessor has only a few interfaces and these are well defined.

- Clock
- Reset
- 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 indicating the memory should be read
- Write control: output indicating the memory should be written
- Reset: input indicating that the processor should adopt a predefined state
- 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 programmers’ model has two registers:
- a 16-bit Accumulator
- a 12-bit Programme Counter

There is one instruction format:

<table>
<thead>
<tr>
<th>F</th>
<th>Mnemonic</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LDA S</td>
<td>Load accumulator from memory location S</td>
</tr>
<tr>
<td>1</td>
<td>STA S</td>
<td>Store accumulator to memory location S</td>
</tr>
<tr>
<td>2</td>
<td>ADD S</td>
<td>Add memory location S to accumulator</td>
</tr>
<tr>
<td>3</td>
<td>SUB S</td>
<td>Subtract memory location S from accumulator</td>
</tr>
<tr>
<td>4</td>
<td>JMP S</td>
<td>Jump to instruction address S</td>
</tr>
<tr>
<td>5</td>
<td>JGE S</td>
<td>Jump to instruction address S if accumulator is positive</td>
</tr>
<tr>
<td>6</td>
<td>JNE S</td>
<td>Jump to instruction address S if accumulator is not zero</td>
</tr>
<tr>
<td>7</td>
<td>STP</td>
<td>Halt execution</td>
</tr>
<tr>
<td>8-15</td>
<td>—</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Reset halts execution. When reset is removed execution begins by fetching an 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.

Implementation

As illustrated in the lectures, the detailed implementation may vary. The following assumes that each instruction occupies two clock cycles, the first fetching the instruction and the second executing what was fetched.

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

Decode/execute: various operations depending on instruction fetched

Practical

The processor can be simulated on its own. However this relies on the simulation environment providing the correct values at the correct times. It is much easier to simulate a system which includes a memory model.
Two memory models are provided. For simulation purposes use memory_1. This 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; you can edit the component if you wish to include your own software. Software development is described later in this exercise.

Create a new schematic and integrate your processor with this memory (figure 11.3). Use this to simulate and therefore test the processor’s operation.

The following code, available as $COMP10211/MU0_examples/MU0_test.s is intended to test the processor’s operation. If it works correctly the processor should halt at the second STP with the accumulator value 0000.

```
Test code
ORG 0 ; Begin at reset address

LDA xxx ; Should be 8000
JGE fail ; If a positive number
JNE okay_1 ; Non-zero?
fail JMP fail ; Halt here
STP ; In case JMP fails
okay_1 STA yyy ; Test STA
SUB one ; Test SUB
JGE okay_2 ; Should take branch this time
JMP fail ; Failed to branch correctly
okay_2 LDA yyy ; Load modified value back
JNE okay_3 ; Loaded value should be non-zero
JMP fail ; Failed branch
okay_3 JGE fail ; Loaded value should be negative
ADD xxx ; Result should overflow to 0000
JNE fail ; Fail if result anything else
JMP done ; Test jump always taken
LDA one ; Indicate jump failure
STP ; Halt

one DEFW 1 ; Constant value
xxx DEFW &8000 ; Test value
yyy DEFW &0000 ; Location to modify/read back
```

This is not a perfect test but it does exercise every instruction at least once and conditional branches have to be both taken and not taken correctly to pass.
Verilog memory model

A memory model (memory_1) is provided in your MU0_lib library which can be connected directly to the processor model. This is suitable for simulation purposes and is reproduced here with some explanation.

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

reg [15:0] mem [12'h000:12'hFFF];
initial $readmemh("<filename>", mem);

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

endmodule
```

- 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.
Testbench

Components are provided to help integrate/display your design on the circuit board. To save time these have been set out in a design ‘MU0_system’ (fig. 11.4) for you to modify.

- ‘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_2’ is a Verilog memory which ‘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 decoder 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 11.4 is simply test and display logic; the important parts are the processor, memory and (in this case very limited) Input/Output.
The display module allows you to step and view the operation of the processor. (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 which are assumed for labelling the display (figure 11.5). 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 (scrolls)
- Press ‘=’ for data display (labelled ‘d’)

This is updated as the memory changes.

### Input and output

Some simple memory mapped I/O is provided in the upper address space. For simplicity (and laziness) the I/O repeats throughout the top quarter of the address space (i.e. C00 - FFF). Within this space, even output addresses correspond to the bargraph LEDs whilst odd addresses map to the traffic light LEDs. In each case the least significant bits are used.

These outputs are implemented as latches, in parallel with the RAM as it does no harm to write to the RAM at the same time.

An input port is provided which maps the six hex ‘letter’ keys to bits in the same word. Consider how this could be read by the processor.

**Hint:** this requires some address decode and a multiplexer to stop the RAM driving the input data bus at the same time as the I/O is being read.

<table>
<thead>
<tr>
<th>Key</th>
<th>Bus</th>
<th>Label</th>
<th>Bits</th>
<th>Provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Address</td>
<td>A</td>
<td>12</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>Input</td>
<td>di</td>
<td>16</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Output</td>
<td>do</td>
<td>16</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>PC</td>
<td>Pc</td>
<td>12</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>IR</td>
<td>Ir</td>
<td>16</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>Accumulator</td>
<td>Ac</td>
<td>16</td>
<td>No</td>
</tr>
</tbody>
</table>

*Figure 11.5: Recommended display wiring*
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/COMP10211/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) which can output files in this format.

$COMP10211/MU0_examples/MU0_GCD.s is an example binary file which contains object code for the programme shown in figure 11.6. 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.

```assembly
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 ; if (B > A) ...
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 11.6: MU0 demonstration programme*
Practical

Stage 2

Compile and download the MU0 system with your completed processor to an FPGA. 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 which indicates the processor has executed a \texttt{STP} instruction. This could be connected, for example, to the seven segment decimal points to indicate that the programme has completed.

Demonstrate the default programme running; the 10 Hz clock works best for this.

The following are some extra options which you might choose to explore. They are independent and can be attempted individually or in any order.

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.

Option 2

Modify the processor datapath and control to incorporate some of the optimisations presented in the lectures – or others of your own devising.

Take care: changes are likely to affect many parts of the system, possibly calling for changes in units such as the ALU.

Option 3

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

A signed demonstration sheet together with printed evidence of the latest stage you have completed.

e.g. schematics and simulation traces for stage 1
top level schematic (with any modifications), signed off demo for stage 2
Verilog listings (etc.) for option 1
etc.

Software development

An assembler MU0asm is available which will produce MU0 object code from a symbolic source. Example source code can be found in: $COMP10211/MU0_examples/ (/opt/info/courses/COMP10211/MU0_examples/).

Type MU0asm <filename> to generate a file <filename>.mem in your Cadence compilation directory (~/Cadence/COMP10211/xilinx_compile/). This will generate a memory image file which can be loaded into Verilog simulation or compiled into a FPGA bit-file 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.

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 that 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 functions are provided:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>&amp;my_signals</td>
</tr>
<tr>
<td>NAND</td>
<td>~&amp;my_signals</td>
</tr>
<tr>
<td>OR</td>
<td></td>
</tr>
<tr>
<td>NOR</td>
<td>~</td>
</tr>
<tr>
<td>XOR</td>
<td>^my_signals</td>
</tr>
<tr>
<td>XNOR</td>
<td>~^my_signals</td>
</tr>
</tbody>
</table>
Aiding readability … and maintenance

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

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

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

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

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

```verilog
'define R__R  6'b100_100; // ‘_’ in a number is ignored
'define RA_R  6'b110_100;
'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 only
```

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

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

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

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