This manual belongs to: ______________________

e-mail: ______________________

If found please return to owner

or leave in the electronics laboratory (1.8)

# Contents

# Organisation of the Laboratory

- These notes apply to the CS121 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 experiments are done according to the timetable displayed in the laboratory and that every effort is made to complete experiments within the scheduled time.
- The scheduled laboratory sessions are on TUESDAY and THURSDAY afternoons. 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 experiments 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 experiments within the scheduled time.
- A master timetable is displayed on the notice board in the laboratory which gives details of when you should attend the laboratory and what experiment is scheduled for each session. A timetable is also included in this manual.
- The laboratory is permanently staffed with a technician who is responsible for maintenance of equipment. He also supervises the laboratory outside formal laboratory hours. This means that if you do not complete an experiment within the session(s) scheduled for it, you can catch up by attending the laboratory when you have spare time and it is not in full use by other students. The technician may not be able to answer all your questions about the experiments however.
- The first session of the CS121 laboratory consists of several short introductory experiments designed to familiarise you with basic logic gates and some simple circuits. The rest of the laboratory comprises a larger set of similar experiments which are ‘wired’ inside a programmable gate array.
- Students work in groups of two. You may choose your own partner provided that you attend your first lab. session. Each group is assigned a group number which is solely for administration in this laboratory. Quoting the group number on all work handed in will assist with the running of the laboratory.
- *Experiment scripts must be read before coming to the laboratory session for that experiment.* Besides familiarising you with the work to be done, some experiments require preparation before the practical work is started. 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.

- **All 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 included at the back of this manual and should also be available in the laboratory.

- Each experiment requires the work to be checked by a demonstrator as the experiment proceeds. The experimental answer sheets indicate after which parts of the experiment a signature must be obtained. The completed and signed answer sheet must be handed in at the end of the experiment. No marks will be awarded unless this is done.
  The longer experiments contain ‘milestones’ for each session which should be signed off. These are to help you keep up to schedule.
  *The technician is not empowered to sign the answer sheets.* If you have to complete an experiment 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.

- The pro-forma answer sheets can be handed in by putting them into the laboratory “post-box”. Before handing in a pro-forma answer sheet a report card[1] should be filled in and stapled to it. Once the report has been marked the report will be returned. Remember to put your name and laboratory group on both your report and the card.

- **Remember that experiments should be completed and handed in at the end of the session scheduled for that excercise.** 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.

- Keep copies of your designs in the CS121 laboratory; some of these will be needed if you enrol in CS122. Please <u>do not</u> delete anything at the end of the course without first checking that this is okay with your (former) CS121 lab. partner.

- 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 experiments in the CS121 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 experiment 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).

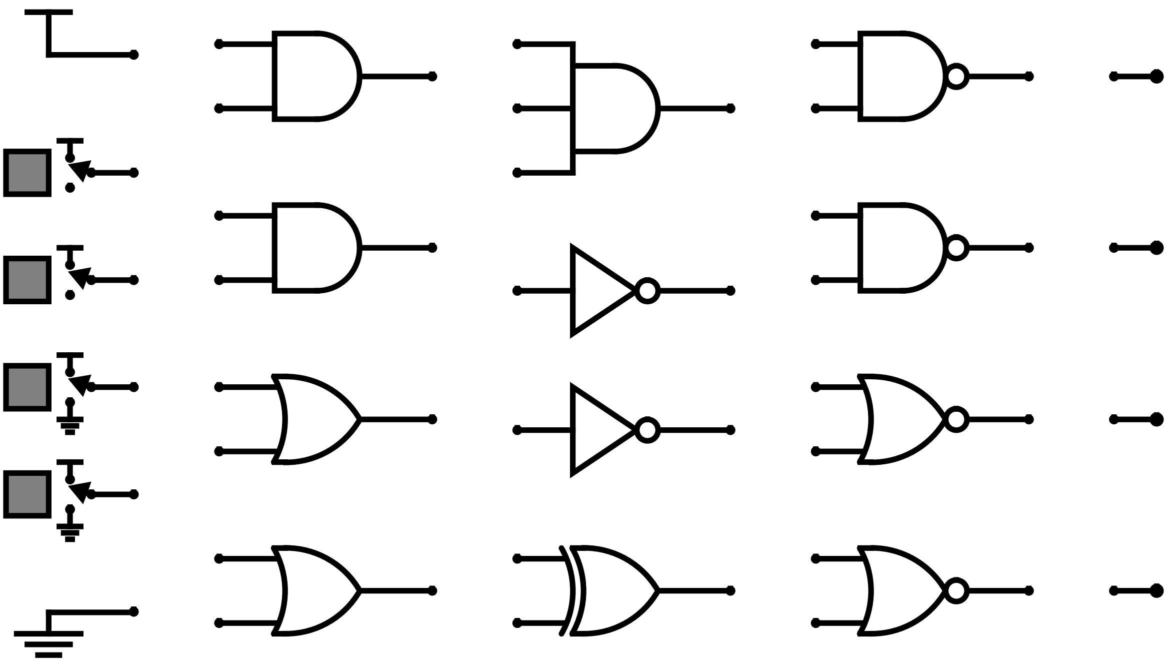


***Figure 1.1: Laboratory Patch Board***

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 experiment 121 - 1.

# Experiment 121 - 1: Simple Logic Design

## Aim

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

## Preparation

Since this experiment is designed as an introductory exercise, no written preparation is needed. However, you should read the experiment 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 experiment 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’ <u>and</u> the second input is ‘true’ (<u>and all</u> 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.

| A | B | C |
|---|---|---|
| F | F | F |
| F | T | F |
| T | F | F |
| T | T | T |

| A | B | C |
|---|---|---|
| F | F | F |
| F | T | T |
| T | F | T |
| T | T | T |

***Figure 2.1: Truth tables for 2-input AND and OR gates***

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.

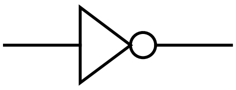


***Figure 2.3: The Inverter***

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.

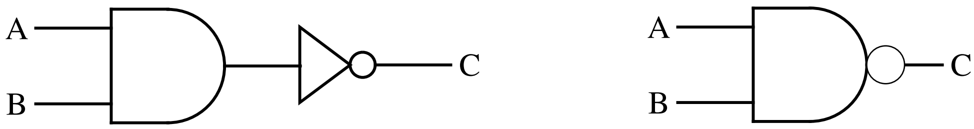



***Figure 2.4: Function and ISO symbol for a NAND gate***

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:

| | | |
|---|---|---|
| $A \,.\, B$ | for | A AND B |
| $A + B$ | for | A OR B |
| $\overline{A}$ | for | NOT(A) |

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:

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

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

**Practical**

**This experiment 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 experiment, 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".**

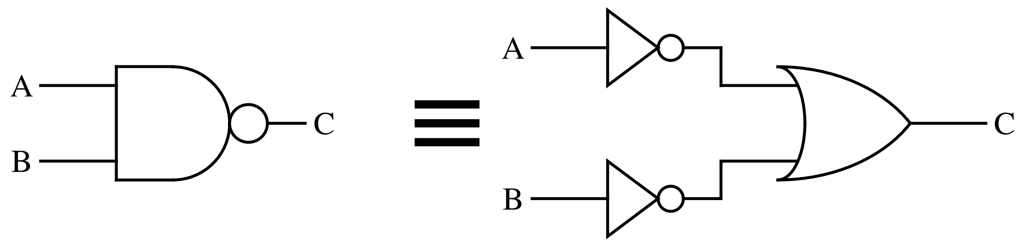
**This experiment is performed on the laboratory patchboards, described in an earlier section.**

**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:

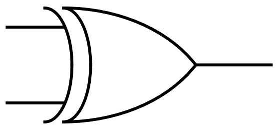
$$\overline{\mathbf{A.B}} \equiv \overline{\mathbf{A}}+\overline{\mathbf{B}}$$



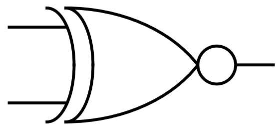

**Practical**

**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)
or
not-equivalent gate



Exclusive-NOR (XNOR)
or
equivalent gate

**The normal symbol for exclusive-OR is a "⊕"**

**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.**

## Signal Interpretation

A signal may be interpreted in a number of ways depending upon the way it is used. These may appear confusing at first, but they will soon become familiar.

So far the signals used in this experiment have been interpreted as Boolean values. For example the circuit in figure 2.5 evaluates a simple logic function where each signal is either TRUE or FALSE. Such signals are often classified as **control** signals.

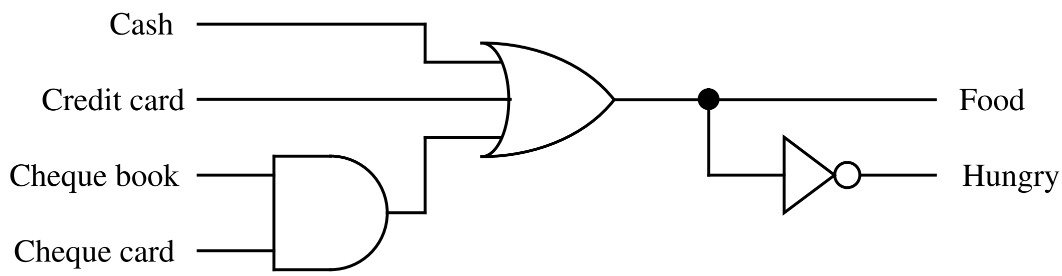



***Figure 2.5: A simple 'shopping' circuit***

As was mentioned earlier it is not always convenient to think of functions in this way. For example it is often necessary to represent *numbers* in calculations. Conventionally the signals are then interpreted as "0" and "1" and are often referred to as **data** signals. Several of these binary digits (**bits**) are frequently grouped together to form binary numbers.

This division into data and control is not absolute; sometimes the same wire may represent data in one part of a circuit and a control function in another.

When constructing circuits from real, electronic gates the logical states are normally represented by an electrical voltage. As there are only two states for a given signal (wire) these are simple referred to as ‘high’ and ‘low’ voltages[1]. These are often abbreviated to “H” and “L”, respectively. Usually “0” is represented by a “L” voltage and “1” represented by a “H”. However the ‘rules’ are less clear when dealing with control signals.

**Active high and active low**

The normal convention is to interpret a ‘high’ voltage on a control circuit as ‘TRUE’ (“active”) and a ‘low’ voltage as ‘FALSE’ (“inactive”); if a signal obeys this rule then it is known as an **active high** signal. However sometimes it is convenient to interpret the signals in the opposite sense, such that a signal is active when its corresponding wire is at a ‘low’ voltage; this is therefore known as an **active low** signal.

Sometimes the active state of signals may be confused; for example a signal that signifies ‘OPEN’ and is active high <u>may also be interpreted as</u> an active low signal ‘CLOSED’. There are several conventions for specifying the active levels of signals, but probably the commonest is to write active low signals with a line above them, thus:

$$\mathrm{OPEN} \equiv \overline{\mathrm{CLOSED}}$$

Others prefer to suffix the signal name with the active state:

$$\mathrm{OPEN.H} \equiv \mathrm{CLOSED.L}$$

and sometimes the convention that active low signals begin with the letter “n” (for **N**OT) is used, especially when only letters/numbers can be used in a name.

$$\mathrm{OPEN} \equiv \mathrm{nCLOSED}$$

The interpretation of signals as active high and active low can be very confusing at first, but soon becomes familiar with use. Remember:

| | | | | |
|---|---|---|---|---|
| Active High | high (H) | active | asserted | true |
| | low (L) | inactive | not asserted | false |
| Active Low | high (H) | inactive | not asserted | false |
| | low (L) | active | asserted | true |

## 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 2.6.

1. In this context “high voltage” is typically 5 volts, not a dangerous high voltage.

| INPUTS | | | OUTPUTS | |
|---|---|---|---|---|
| Digits | | Carry | Carry | Sum |
| $A_i$ | $B_i$ | $C_{i-1}$ | $C_i$ | $S_i$ |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

***Figure 2.6: Truth Table for a Simple Adder***

From the truth table of figure 2.6 it is possible to construct the logical equations needed to generate the functions $S_i$ and $C_i$. However, since this is the first experiment 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 2.7; such circuits are known as half-adders.

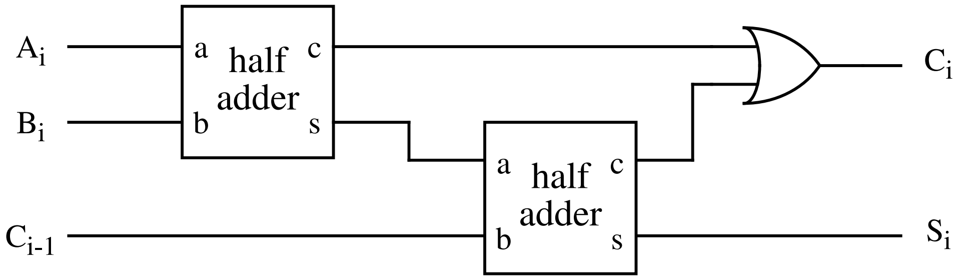



***Figure 2.7: A Full Adder built from two Half Adders***

Figure 2.7 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 depended 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 experiment 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 2.8. This method of drawing the circuit emphasises the essential difference between sequential circuits and combinatorial circuits, that is sequential circuits have a **feed-back** path from output to input.

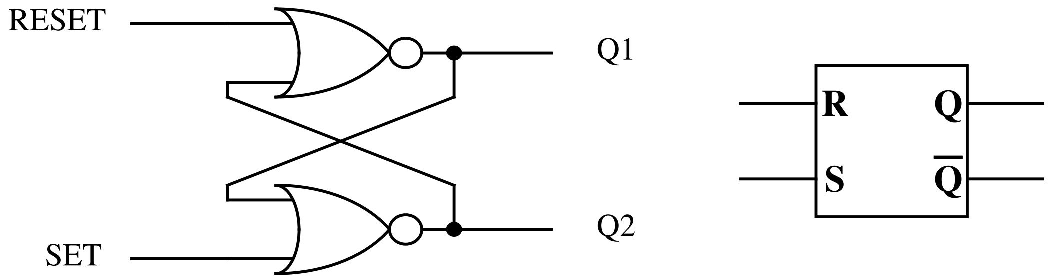



***Figure 2.8: S-R Flip-flop: Circuit and Symbol***

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

$$Q1 = \overline{RESET + Q2} = \overline{RESET} \cdot \overline{Q2}$$

$$Q2 = \overline{SET + Q1}$$

therefore

$$\overline{Q2} = SET + Q1$$

$$Q1 = \overline{RESET} \cdot (SET + Q1)$$

From these equations the behaviour of the device may be tabulated as shown in figure 2.9.

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, $Q1$ and $Q2$ 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.

| S | R | $Q_1$ | $Q_2$ |
|---|---|---|---|
| 0 | 0 | Q1 | Q2 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 [a] |

[a]This state is not normally allowed

***Figure 2.9: Tabulated Behaviour of a S-R Flip-flop***

**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.**

**7 What is the normally disallowed state of the input lines for this flip-flop?
What would happen if both the control lines were in this disallowed state and simultaneously deasserted (i.e. inactivated)? Think about how the circuit works. Do you expect the circuit's behaviour to be consistent?
You may be able to demonstrate this on the patchboard; if you cannot, ask a demonstrator.**

## Clocked S-R Flip-Flop

The circuit of figure 2.10 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.

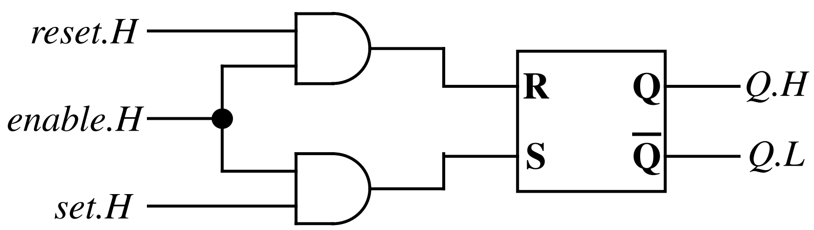



***Figure 2.10: Clocked S-R Latch Circuit***

## 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 2.11.

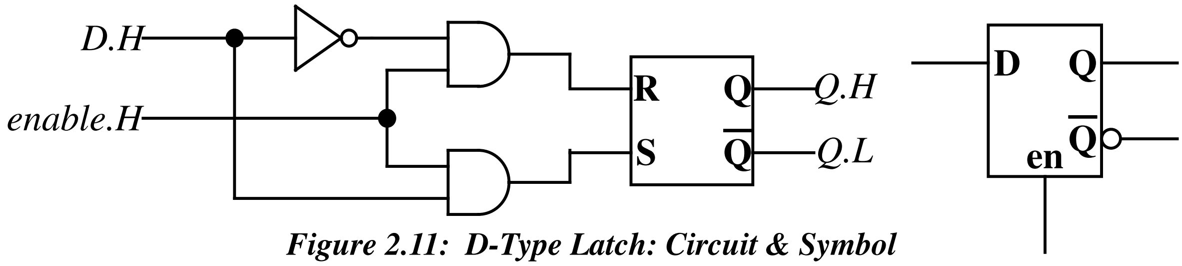



***Figure 2.11: D-Type Latch: Circuit & Symbol***

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**

**8 Construct and test a D latch *using the flip-flop design from part 6 of this experiment*. Document and demonstrate your circuit.**

**Don't Forget** to hand in your experiment 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 experiment. 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 is the ‘bottom’ layer of the hierarchy, although these could further be decomposed into transistors[2] and thence into bits of solid state physics.

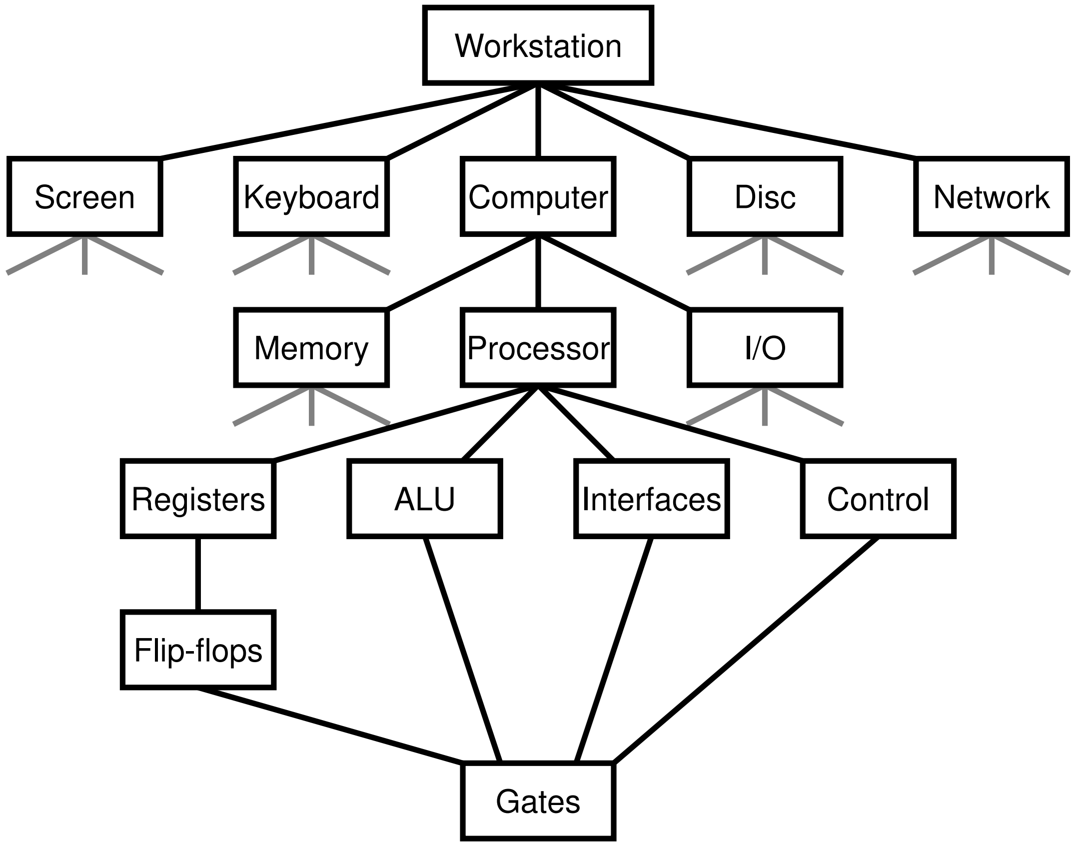



***Figure 3.1: Design Hierarchy***

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 *ViewDraw* 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 **Powerview** 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. A line taken from the bus is called a “breakout” and is normally drawn as shown in figure 3.2. The breakout must be labelled to identify it.

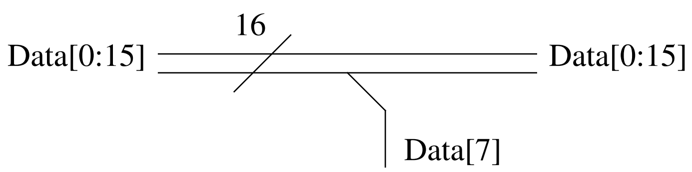



***Figure 3.2: A 16-bit bus***

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.

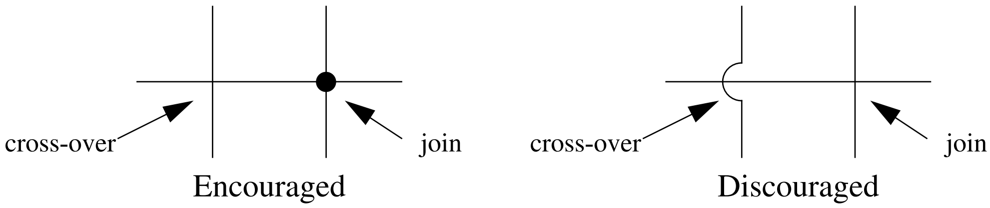



***Figure 3.3: Signal wires crossing***

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[1] 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 difficult in entering schematics[2] 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 Powerview software.

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

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

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

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

---

1. Not directly relevant to CS121 but worth remembering for the future.
2. ...or most other subjects

**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 red in Powerview**.

**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 red in Powerview**.

**break-out**: a point where a single net enters or leaves a bus.

**pin**: a place on a symbol where nets can be connected.
**Pins are cyan (light blue) in Powerview**.

**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 a 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 white in Powerview**.

**attribute**: an attribute is some miscellaneous value associated with some other item. They are as diverse as a propagation delay for a gate, a pin number on an integrated circuit or a date on a diagram.
Attributes are shown in yellow in Powerview.
There is no need to manipulate attributes in this laboratory.

**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 are green in Powerview**.

# Powerview Walk-Through

This exercise is intended as a first introduction to the Powerview software. This software can only be used in the "X windows" or "Openwindows" environments.

## Software Setup

Before Powerview 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:

/usr/info/courses/CS121/mk_cs121_env

Because people work in pairs it is necessary for both partners to run this script. This will create a shared directory, which either partner can then use.

All your Powerview work is contained in a directory ~/powerview, and the project directory required for this course is contained therein. An additional file 'Iview' will also be added to your home directory; this is used if the on-line help is invoked.

## Creating a Design

### Starting ViewDraw

Start Powerview by typing *pview* in a command shell window. The Powerview cockpit will be loaded into the machine, providing access to the editing and simulation tools. 'Double click' on the ViewDraw icon in the icon area in the cockpit, this will invoke the ViewDraw schematic editor. A 'ViewDraw file open' window should open; from here type the name '**halfadder**' at the 'Enter name' prompt. If the schematic already exists it may be selected by clicking the button with its name. After the file has been selected click on ACCEPT. *Note: filenames acquire an extension - typically '.1' - as the symbol identifier this should NOT be typed here*.

Two ViewDraw windows will now appear; the smaller one is the 'context' window and shows the whole schematic; a box on this indicates the area displayed in the main ViewDraw window; this is more meaningful after zooming in. These windows can be moved and iconified so put the smaller one somewhere convenient. It is usually most convenient to work in the larger window, but either may be used. Now you are ready to begin the design entry.

On the large window you should see a white rectangle. This represents the edge of your piece of 'electronic paper' upon which the design is drawn. Try not to let designs overflow this area as this may cause problems when it comes to printing out.

### 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 the following design.

- The first thing to do is place some of the components. Select Add ➪ Comp from the main Viewdraw menu bar; this will cause the ‘Add Component’ window to appear. Within this window there are two main areas. The upper half shows the set of components available in the current library (initially it should be blank as none has been created) and the lower area is a list of available libraries. These may be selected with the left mouse button. If necessary these areas may be scrolled.
  Select the X3000 library; a list of components now becomes available. Select the component AND2.1 (an AND gate with 2 inputs) and place this component on the schematic area by moving the cursor to the chosen position and pressing the middle mouse button. This function and gate is now selected, so to place another AND gate simply press the middle button over the schematic area, move the gate to the correct position and press the middle button. Add components OR2.1 and INV.1, place them so as to resemble figure 4.1.

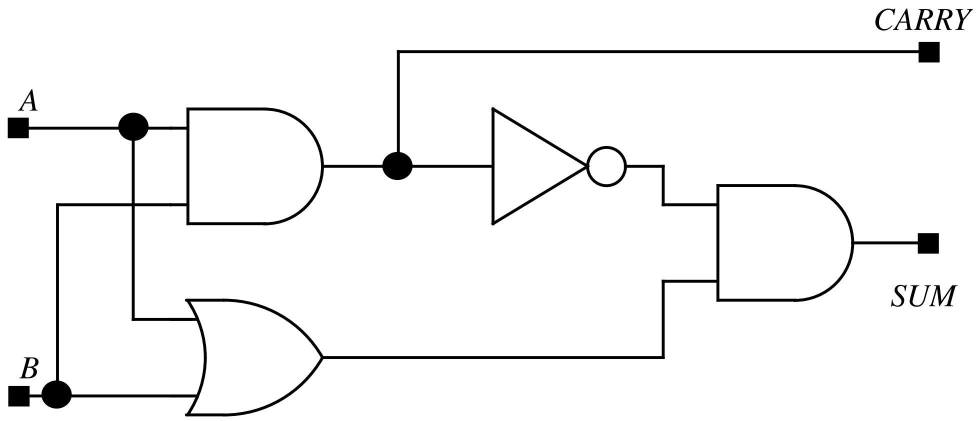



***Figure 4.1: Half Adder***

- Any erroneous editing may be corrected using the EDIT ➪ UNDO option which will undo the previous editing command. Only a single action can be undone.

- As these components will appear rather small it is useful to be able to zoom in on them. The various functions for altering the display are contained in the pull-down menu View, although there are other (more convenient) ways of executing them using the function keys on the keyboard. Experiment with In, Out, Zoom, Full, Pan and Scroll and familiarise yourself with what each does. Watching the context window whilst doing this will demonstrate its function. If you get ‘lost’ then View ➪ Full should get back to a recognisable state.

- If a component is in the wrong position it can be moved. Selecting Edit ➪ Move allows components to be ‘picked up’ and repositioned using the middle button. Before a component can be moved it must be ‘selected’ using the left mouse button; this should highlight its bounding box. Once Move mode has been entered components may be selected and moved at will. If an extra component has been added by mistake, select the component use Edit ➪ Delete to remove it.

- Now the components are positioned correctly join them up using Add ➪ Net. Place the cursor on the start point of which must be either a component pin (light blue) or an existing net then press the middle button. Move the cursor to the end point and press the middle button again. If necessary bends in the wire may be placed using the middle button. If the end of the net is to be left unattached press the middle button at the end point to place it then the right button to leave it 'floating'.
  These connections should all appear in red on the display. If the diagram becomes untidy remember that nets may be moved and deleted in the same manner as components.

- Labels ("A", "B", "CARRY" and "SUM") can now be added to the nets so they can identified easily. Select a net to add a label to and Add ➪ Label. This will make an Add Label window appear. Enter the name for the net in this window then click Accept. Move the resulting box to the correct position and click the middle button. This label is now attached to the selected wire. Label the rest of the nets by highlighting them with the left button and labelling them with the middle button.

- Save the design with File ➪ Write.

By this stage you will probably have already discovered some 'short cuts' for obtaining the desired result. For example pressing the right hand mouse button in the drawing window gives an alternative menu for some frequently used buttons. Also included in many of the menu entries is a keystroke which can be used to perform the same function. Many of these are chosen mnemonically, e.g. 'z' for zoom or 'u' for undo, although some e.g. 'i' ("instantiate") for Add ➪ Comp are less obvious.

**Creating a Symbol**

The circuit produced so far is only a fragment which may itself be used as a library component for larger circuits. However before it can be used it must be given a symbol to represent it on future schematics (e.g. figure 4.2).

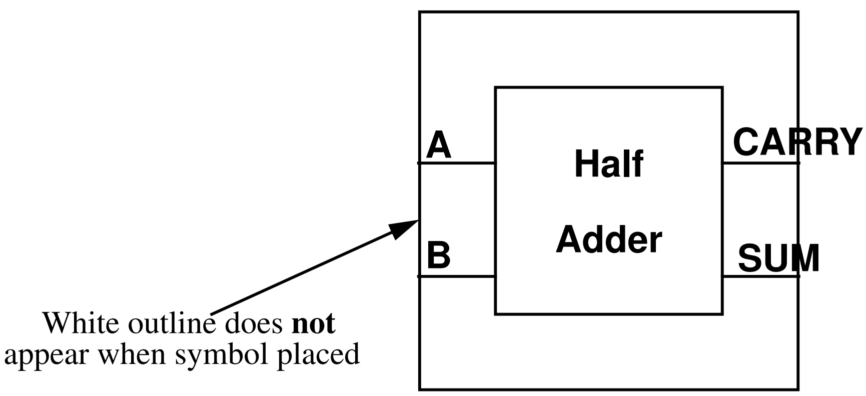



***Figure 4.2: A possible Symbol for a Half Adder***

- To design a symbol for the design another ViewDraw window must be opened. The easiest way to do this is to use File ➪ Level ➪ Push ➪ Symbol; this suspends the current activity and gives a blank sheet on which to work. The white box is the (nominal) bounding box for the design. The size of this box may be altered but the default size should be large enough for the simple 'halfadder' symbol.

- The most important features of the symbol are its *pins*. These specify the connection points and should have the same names (*labels*) as the connection points in the schematic. To place a pin select Add ➪ Pin, use the middle button at the start point on your symbol and draw out towards the boxed border, press the middle button to complete the placement. The outer (connection) end of the pin should 'snap' to the bounding box. Use the middle button for the remaining pins; you will need a total of four pins.
  Pins should be light blue.

- Label the pins as before remembering to use the same label names as in the schematic design. These labels will be matched between the two and any name inconsistency will cause errors.
  Remember that labels appear in white.

- The rest of the symbol is merely pretty graphics intended to make identification easier. When designing the symbol work inside the boxed area. Various functions are available under the Add ➪ Graphics menu options such as Arc, Box, Circle, Line and Text.

- Save the design with File ➪ Write. When a symbol is saved a window is created with information about the design and potential warnings. This specifies the design name, any 'unattached attributes' and the pin names. If the symbol is correct you should recognise all the name under the heading "pin information" and there should be nothing under the "unattached attributes" heading.

- To return to the schematic select File ➪ Level ➪ Pop.

- The half adder is now available for use in other designs however it may *NOT* be used in the half adder design since this will cause 'recursion'. It will be in the Primary library on Add Component window. This component will be used later in the construction of a full adder. However first it must be simulated to ensure that it will perform as expected.

## Simulating the Design

The design will be simulated using the ViewSim tool, the output will be displayed on the schematic and in a ViewWave window. The simulator has many facilities not used in this walk-through, which are covered by the on line documentation.

- Activate the vsm tool from the Powerview cockpit. This will process ('extract') the schematic and produce a file suitable for the simulator. Usually the default 'Design Name' will be correct, but check this on the top line of the window; this can be edited if required. Click the Accept button on the vsm window which appears and wait for VSM to finish and exit with no errors or warnings (this will be displayed on the cockpit message area).

- Activate the ViewSim tool from the cockpit. Check the defaults and click Accept on the ViewSim start-up window. The Viewsim tool will now start on the current design. This opens a ViewSim window which has a number of options and menus). Small “X”s will be placed on all the inputs and outputs on the schematic. The “X” indicates the current value is undefined. If any “?”s appear then the simulator has not been able to identify a node. This can happen if the schematic has been changed since vsm has been run. Try quitting the simulator and re-extracting the design. If this fails consult a demonstrator.

- Select the A, B, SUM and CARRY signals on the schematic. Note that it must be the <u>nets</u> which are selected, not just the labels. Normally selecting an item deselects the previous one, but if the Control key is used in conjunction with the left mouse button several items can be selected. Selecting a single segment of each of the nets is sufficient. From the ViewSim window select Trace ➪ In Wave Window... and click Accept on the window which appears; this will start a ViewTrace tool which will display the selected signals. Resize the ViewTrace window and rearrange the windows so the three tools are visible.

- To simulate the circuit the inputs must be set up. First select the signal ‘A’ (left button only) on the schematic and then Set ➪ L (i.e. ‘Low’) and Accept in the ViewSim window. This makes an instantaneous change to the circuit and some (simulated) time must pass before the result appears. To see the effect of this on the circuit press the Sim button (or the Sim ➪ Sim menu entry) in the top left of the ViewSim window. This allows 100ns to pass.
  The logic levels will be displayed as either 0, 1 or X in boxes on the schematic and as waves in the ViewTrace window. Signals which have been ‘set’ externally have boxes around the value whereas those derived from gates in the circuit do not. Note that the output of the first AND gate is defined (even though one of its inputs is not) but the output of the OR gate is undefined; think why this is so.

- Select the net B from the schematic and Set ➪ L, Accept this as above and press the Sim button again to see the effects. All the nets in the circuit should now be defined. On the ViewTrace window notice that the outputs do not change at the same time as the inputs but a small time later. This is the **propagation delay** of the circuit. It is relatively unimportant for this circuit because the inputs are only changing every 100ns, but it is significant in high-speed circuits.
  The default delays assumed for the simulator are 1ns per gate. This represents “some” delay; the propagation delays in the final circuit may be greater.

- Practise with the simulation. Remember that if an output from a gate is set high or low with the ‘set’ command it will be forced to the assigned level in preference to the simulator’s calculated level, this can be remedied with Set ➪ Release which allows a signal to adopt its ‘natural’ state.

- Characterise your circuit. Does it perform as expected?

At this point proceed to exercise 121 - 2 which extends this design further.

## A few useful extra features in PowerView

- When adding components the component can be rotated using F9 and made visible using the left button.
- Groups of items can be highlighted using the left button to draw a box around the area to be selected. Individual items from this group can be de-selected by pressing SHIFT and the left mouse button on the item to be removed from the group. In a similar way items may be added to the group using CONTROL and clicking the left mouse button on the item. These groups may be moved or deleted as shown above.
- A 'Fast' mode is available which 'intelligently' selects the action the user wants to perform:

    - On blank areas of the schematic a component is added;
    - over a component or wire the item is moved;
    - over a pin a wire is added.

  Over the editing area the right button will produce a menu with the main editing functions listed above. The Button Action bar will specify the action of the buttons for which ever mode the system is in. The 'fast' mode does not allow nets to be started from other nets, thus it is not possible to join two nets together in fast mode since acting on a net in fast mode will move it.
- When creating symbols a default 'bounding box' (100 x 100 units) is supplied. This is large enough for simple symbols, but is cramped for symbols with more pins. To change the size of this use: Change ➪ Block ➪ Size ➪ Z-WxH and type in the new width and height.
- When drawing both schematics and symbols it is sometimes convenient to change the drawing grid. This parameter is available; it is hidden under Props ➪ Value ➪ GridSpace (you will need to scroll to find this last entry). It is sensible to keep the gridspace to a multiple of 5 or 10 in most cases.
- Remember that changes to the schematic cannot be simulated without remaking the netlist and reloading it into ViewSim.
- ViewTrace is not really needed for these simple simulations but has nevertheless been introduced here. It is much more useful for sequential circuits when it is desirable to be able to view the history of the circuit as well as its current state. You may like to experiment with other display options, such as the 'Monitor' facility.

# Powerview

Powerview is a Computer Aided Design (CAD) system produced by Viewlogic Incorporated. It is a package containing a range of ‘tools’ which allow - among other things - the entry of schematic diagrams, and the simulation of the resulting design. The tools of greatest interest to this laboratory are those which allow the entry of circuit diagrams via the schematic capture programme **ViewDraw** and the simulation of these using **ViewSim**.

Once the schematic is complete and checked to ensure its correctness, an EPLD (Electrically Programmable Logic Device) can be configured to implement the design. The use of an EPLD allows any design faults found at this late stage to be corrected without redesign of the physical hardware around the EPLD; if it doesn’t work first time the EPLD can be reprogrammed with a corrected design. This will be done later on in the CS121 course using circuits based on the Xilinx X3030.

The following section contains a brief description of Powerview and some of its facilities. This is followed by a step-by-step walk-through as a first-time introduction to the system. Ideally the walk-through should be used for familiarisation before attempting to use some of the more complex functions.

## Software Setup

Before Powerview 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:

/usr/info/courses/CS121/mk_cs121_env

Because people work in pairs it is necessary for both partners to run this script. This will create a shared directory, which either partner can then use.

All your Powerview work is contained in a directory ~/powerview, and the project directory required for this course is contained therein. An additional file ‘Iview’ will also be added to your home directory; this is used if the on-line help is invoked.

## Using Powerview

Typing *pview* will change to the powerview subdirectory and invoke Powerview. The Powerview ‘cockpit’ will then appear; from here the Powerview tools may be run.

### The Powerview Cockpit

The ‘cockpit’ is shown below in figure 5.1. It is the main Powerview control area; from here projects can be created, opened and destroyed, tools can be started and suspended, and general information about the current tool, such as the current libraries in use, is displayed. The individual elements in the cockpit are described below.

The top line of the cockpit window contains various pull-down menus. Of these only the ‘red square’ is used frequently.

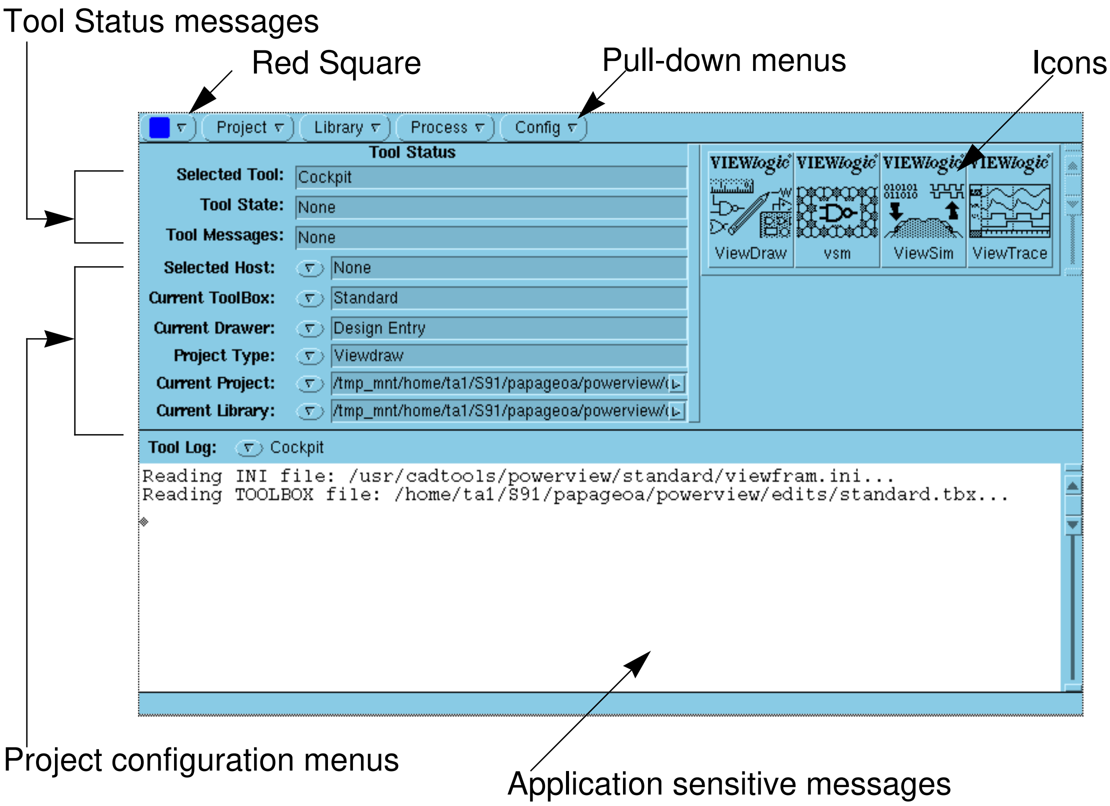



***Figure 5.1: Powerview cockpit***

The RED SQUARE menu includes Quit, a help facility under Help ➪ ViewDoc and customization under the Customize ➪ Edit Preferences item.

The help displays a 'hypertext' document from which subjects may be selected by double clicking with the left button on the associated square. Pages may be stepped through using the forward and back a page icons, the scroll bar may be used to traverse the document and the text search icons allow specific keywords to be searched for. The control panel for the help menus is shown in figure 5.2.

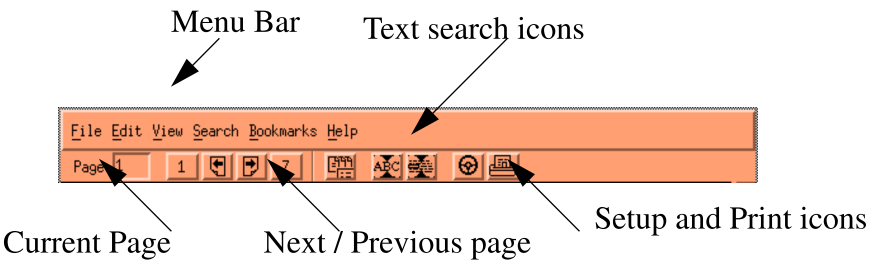



***Figure 5.2: Powerview Reference Manual***

The red square menu is available on all Powerview tools, and is the first item on the menu bar. The other pull down menus give control over projects, libraries, tool boxes and configuration options. These are activated by pressing the left mouse button whilst over the chosen item.

- The Project menu provides the facilities to Create, Set current, Remove, Delete, Copy, Move and Restore projects. For the CS121 course you will only need one project which has already been created. Certain of these options can destroy all your work to date; using copy can fill your file space quickly. **If you must use these, use them with CAUTION**.
- The Library menu selects which library your designs will be available from. The libraries in /usr/cadtools/... are read only. The Show displays all files in the library which are of the type selected.
- The Process menu allows tools to be 'started' and 'stopped'; brief information about a tools function and use are available. The TEF options on the menu refer to Tool Encapsulation Files which specify how a tool is initialised. These are stored in /usr/cadtoools/... and cannot be edited.
- The Config menu allows the re-initialisation of the cockpit with Init Toolboxes. Editing the environment variables could damage your powerview setup and is NOT recommended.

Icons for each of the available tools in a 'drawer' in the toolbox are shown in the icon area. The standard toolbox has been set up with only four tools: ViewDraw, VSM, ViewSim and ViewTrace. These are the only tools required for CS121/122 and each is described below. The icon selected for the Process menu has its name on a darker background, clicking on the name with the left mouse button selects the tool. To start the tool either select its name as outlined then start it from the Process menu or simply double click with the left button on the tool.

The Tool Status messages display information about the currently active tool and its status. This area cannot be edited; it only reflects the status of the other tools, changes should be made from the process menu.

- Selected Tool: shows the currently selected tool.
- Tool State: this show one of: 'started', 'stopped' or 'none'.
- Tool Messages: messages relating to the tool are shown here.

The Project configuration menus in the tool/project configuration area allow the following changes to be made.

- Change of tool box; the standard tool box contains all the tools necessary for the CS121/122 courses.
- Change of drawer; the design entry drawer has the tools necessary for the CS121/122 courses. Other drawers contain tools for (e.g.) ASIC (Application Specific Integrated Circuit) and PCB (Printed Circuit Board) design.
- Type of project; Viewdraw is the circuit design project, Viewplace is the PCB design project type. Only Viewdraw project is needed for the lab exercises.
- Current Project; this is the pathname for the project currently being worked on.

- Current Library shows the component library currently used by the project.

Application sensitive messages are specific to the tool selected from the Tool Log button, they include information about which setup files were used in starting a tool, most error messages from the tool and an exit code (0 if it exited safely).

## ViewDraw

The ViewDraw tool is schematic capture program which allows the user to prepare a schematic diagram. This schematic diagram may be evaluated with other tools in the Powerview package. ViewDraw provides the facilities to make use of library parts when designing circuits; any schematic may be added to a local 'primary' library so it may be used as a component in future designs. ViewDraw normally provides two windows: the main editing window in which most of the work is done and another generally smaller context window which shows the entire current worksheet; the currently displayed area in the ViewDraw window is boxed on the context window. The context window may be closed or resized. The main ViewDraw screen is shown in detail in figure 5.3.

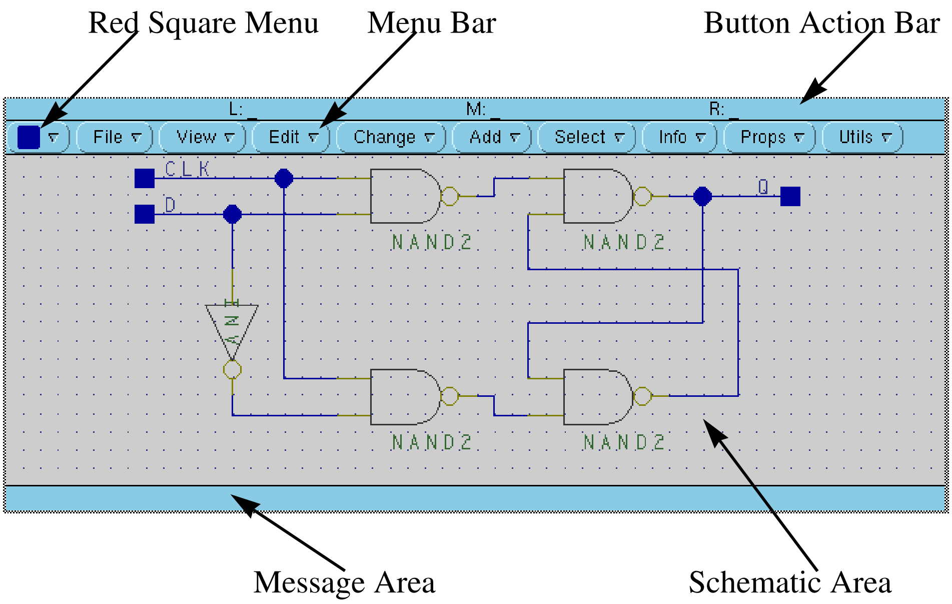



***Figure 5.3: ViewDraw Editing Window***

The Red Square menu is the same as for the Powerview cockpit.

The Button Action Bar show the action that will be performed by the mouse buttons. The button action bar is sensitive to mouse position and the currently selected function for example in 'zoom' the bar shows L: select M: zoom R: menu-schematic.This is generally useful for knowing what mode you are in and what the next mouse button press will do.

The Menu Bar provides access to all the pull down menus, which allow the user to select what the next action will be. These are summoned with the left or right mouse buttons. The available menus are:

- File: This menu provides the file access functions as follows:
  - Open which opens a new ViewDraw window
  - Level which changes the current level in the schematic hierarchy and is used to make for user defined components
  - Read rereads the file currently being edited and
  - Write which saves the current schematic.
- View: The Zoom, Pan and Refresh items are available from this menu, allowing areas of the schematic to be seen in greater detail.
- Edit: The standard Move, Copy, Delete and Undo functions provided. there is also a Buffer which allows cutting and pasting to other schematics. XForm, is the transform option so components may be reflected, scaled and rotated.
- Change: This menu changes parameters of non-schematic items on the sheet including Label visibility and Schematic sheet size.
- Add: The next drawing action is chosen from this menu. Schematics components Net, Bus, Component may be selected from this menu, as may the Graphics components Arc, Box, Circle, Line and Text.
- Select: Items may be selected by name from the select menu. A list of selectable types is given., Comp, Label, Name and Value choosing one of these will produce a window in which the desired item name or type may be entered.
- Info: Much information on available components, symbols, specific components and other ViewDraw items is available from this menu.
- Props: Configuration of the ViewDraw display Modes and Values allows most of the display parameters on the schematic to be changed. Examples of changeable parameters are BusWidth, GridSpacing and allowing Nets to start in space.
- Utils: The check item generates a description of the schematic which may be used by other programs, the file generated is a wirelist file with the extension wir.The other facilities on this menu are concerned with the PCB programs and are not needed in schematic capture.

Keyboard accelerators are available for most commands, these are used for a single action and do not change the editing mode as an example ‘d’ will delete the highlighted component but not change the editing mode to delete mode. The accelerators are listed on the menus and are simply typed from the keyboard.

The keyboard accelerators for the viewing functions are:

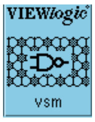
- F5: View Refresh redraw the schematic the display
- F6: View Pan centres the schematic area on the current cursor position.

- F7: View In zoom the schematic one stage centred on the cursor.
- F8: View Out shrink the schematic one stage centred on the cursor.
- F9: View Zoom defines a box which will become the view area.
- F10: View Full to view the entire schematic.

The Fast-Draw mode allows the software to ‘intelligently’ guess the action required, dependant upon the item closet to the cursor. This will work for the basic functions however if fast-draw does not allow an action it may be performed from the menu command.

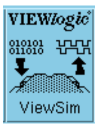
The Schematic Area is where the schematic diagrams are constructed. Components, wires, buses and other schematic components may be placed in this area. In the schematic area the right mouse button brings up a schematic menu which allows the adding of schematics components, general editing commands and the zoom function.

The Message Area gives the X and Y co-ordinates of the mouse at the last action, also given are the change in position between the last two actions. Errors and messages are also reported here, if an action fails check this area to see if there is an explanation. The message area briefly outlines the action performed by the menu option selected.




## VSM

A saved schematic is not in a form the simulator can use directly. VSM is a program which converts a schematic net to a simulator input. The complete circuit hierarchy is stored in file, including any custom library parts. The VSM output file has the extension ‘.VSM’ and is used in the ViewSim tool. The netlist produced by VSM is the circuit description used by the simulation program ViewSim to model the circuit and contains the entire definition necessary for the simulation. A new netlist needs generating if *ANY* changes have been made to the schematic since the last netlist.




## ViewSim

ViewSim (Fig. 5.4) is a tool which allows the simulation of circuits and aids debugging before any physical hardware is made. ViewSim requires a ‘.VSM’ file to be generated from the schematics before it will execute. The circuit simulation can be displayed as logic levels marked on the schematic diagram at a given time or graphically as a set of wave forms. The former is probably more useful when debugging combinatorial circuits whilst the latter allows a view of the history of the circuit and is useful in sequential systems; it also allows ‘glitches’ and propagation delays to be more readily observed. The inputs to the circuit may be setup ‘by hand’ on the schematic or a command file may be used as an input to ViewSim which specifies the signals to set up and their timings.

The Simulation Area reports information about the status of the system and responses to actions such as reporting the simulation time after any command which alters it. Signals being watched can have their values and changes displayed in this area. A scroll bar gives access to

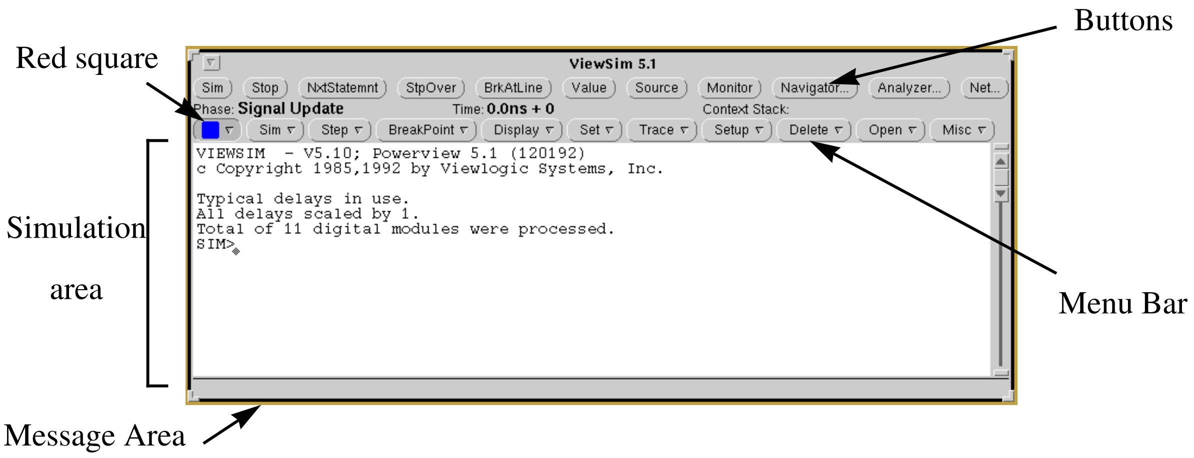



***Figure 5.4: ViewSim Window***

messages generated earlier in the simulation session.

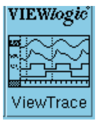
The Message Area displays information about the currently selected menu item and reports any errors which may have occurred.

The Menu Bar shows all the available menus which allow the setting up of signals and the display on the ViewDraw or ViewTrace. The simulation can be cycled through one step at a time or run to conclusion or a breakpoint using these the menus.

- Red Square: The menu is as the standard configuration used on the Powerview tools and has been covered in the cockpit section.
- Sim: This menu sets the simulation cycle time, it can only go forward or be reset. Run, Sim and Cycle advance the time by varying degrees; Continue starts the simulation after a halt from breakpoints or a Stop command. Restart resets the simulation time to zero and clears all setup waveforms and clocks.
- Step: This menu requires the debugger to be active and is not used in the CS121 course.
- Breakpoint: Simulation can be stopped at a particular event or time using the breakpoint commands. The Set Break ➪ On Signal sets which action on which signal will cause a break. Delete allows breakpoints to be removed.
- Display: This menu is used to give information about the simulation, such as current time step size or which signals are being watched. Much of the ViewSim simulation information can be displayed from this menu.
- Set: Signal values can be set high, low or undefined using this menu. Selecting any of the options produces a window in which several signals may be selected by entering the signal names or signals may be selected from the schematic. A value which has been ‘set’ appears with a box around it on the schematic. This value is taken in preference to any value the circuit may calculate. The effect of ‘set’ may be cancelled by ‘release’ing the particular network.

- Trace: Output from viewsim may be viewed on the ViewDraw schematic diagram, on a ViewTrace window or output to a log file as text. Several of these options may be active at once.
- Setup: Complex wave forms may setup, the signals to be Watched or Traced may be selected, time step size may be changed, bus values may be allocated and circuit specific timing checks such as Transport and Inertial delays.
- Delete: Items setup on the setup menu may be deleted, these include Waveforms, Patterns, Clock, Break Points, Watched signals and Traced signals.
- Misc: These are the file access command which are Net to load a new circuit to simulate, in the form of a '.VSM' file. Execute allows execution of either a '.CMD' file or a '.LOG' file which can contain prepared commands and signal setups which may be regularly used. Type outputs a '.LOG' file and Help prompts for a topic help is needed on.

Above the Menu Bar is a set of Buttons which provide a selection of the common items from the menus.




## ViewTrace

Viewtrace (fig. 5.5) is a waveform displayer and editor. It allows the history of logic values at selected points in the circuit to displayed as a waveform with time along the horizontal axis. This enables the relationship between signals to be seen clearly, showing which signals are affected by other signals, which signals change together or which signals have a time lag between them and another signal changing. Short periods of incorrect signal value often called glitches may also be spotted with ease. The simulation allows many signals and buses to be examined. Command files may be generated for use with ViewSim to provide a prepared input for a circuit so it does not require re-entering each time the simulation is executed.

The Button Action bar specifies the action of each mouse button at the current time, taking into account any current modes, and the mouse cursor position.

In common with the other Powerview packages the Message Area is at the bottom of the application window and gives general information about system status, such as the simulation time or menu action.

The Graphical Simulation Area is the area where the waveforms are displayed. The time is marked along the base and the signal names up the left hand side. The axis on the left is marked digital to show the signals are all digital; analog analysis is available but not used in the CS121/122 courses. The waves displayed may be highlighted for editing and the display area zoomed if further detail is needed on any critical timings.

Several editing and trace control options may be chosen from the Menu Bar. These options facilitate file, viewing, editing, axis manipulation, signal manipulation, selection and setup operations in addition to the Red Square menu.

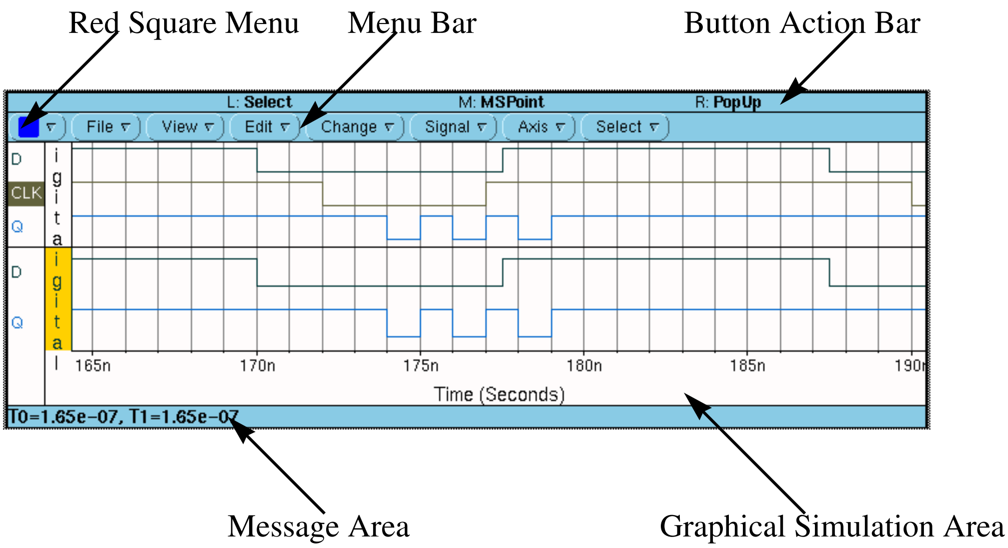



***Figure 5.5: ViewTrace window***

- Red Square menu: Performs all the normal customization, help and quit functions, also allowing plotting, macros and text editing.
- File: Files may be opened which generates an additional Viewtrace window, read; to add to the currently loaded information and written out as waveforms with either the current name or a new name.
- View: The zoom, refresh, pan, zoom in, zoom out and scroll functions are available, to allow specific areas to be examined in more detail; zooming out can sometimes provide a good overview of which signals are active.
- Edit: Signals can be edited with cut and paste operations, areas of signal may be deleted, new signals added and sections of signals repeated, this allows command files to be set up for ViewSim. Editing may not be performed whilst simulation is occurring in the ViewTrace window.
- Change: Signal Name, Colour, Symbol and Radix may be changed, as may information on the axis such as grid spacing. Text may be added, the information displayed may be specified and SnapMode may be changed from here.
- Signal: Which signals are to be displayed and on which axis is determined from this menu, as are some digital operations and logical operations which may be performed on the signals.
- Axis: Manipulation of axis is allowed, axis may be Split, Joined, Added and removed. Added axis display signal selected at the time of creation. These signals are the same as the original and operating on one will have the same effect on the other.
- Select: Signals or text may be selected by name from this menu. They may also be de-selected.

The different Powerview tools interact in the following manner. As can be seen in figure 5.6 all tools can be started from the cockpit, and the tools interact with one another in various ways, using either direct communication or passing information through different files.

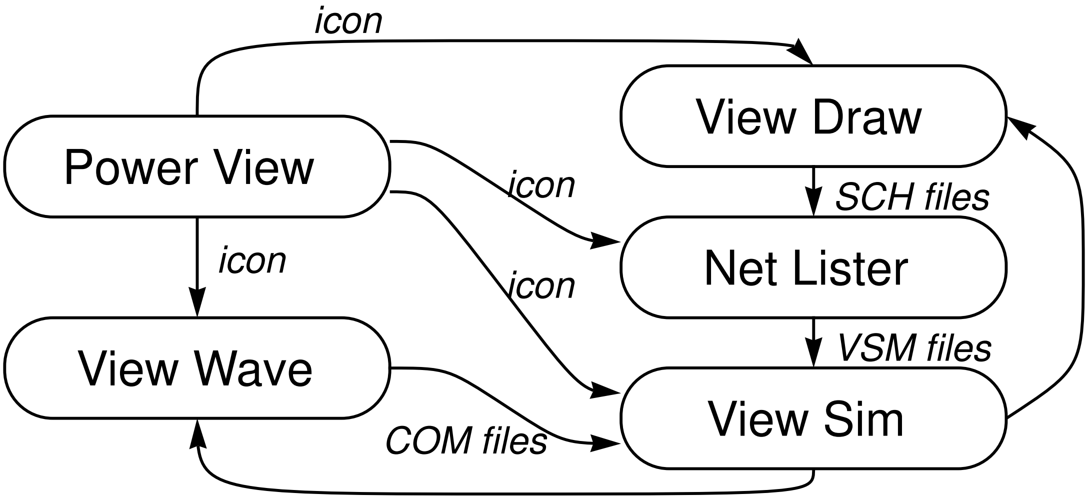



***Figure 5.6: The Interaction of the Powerview Tools***

# The Library Elements Available

All the designs in this laboratory are built from library elements. Most of these are held in a directory which should be available as “**x3000**” if your set up is correct; the ‘local’ cells are contained in a different library - “**englib**”.. You may not modify or add to this library but you can build you 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

- [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}
- [INV] inverter
- [TBUF] tristate buffer

- [VCC] logic high
- [GND] logic low
- [DFF] D-type flip-flop, including an enable input (CE) and an asynchronous reset (RD) - both active high

## Input and output

- [IBUF] input buffer
- [OBUF] output buffer
- [OBUFT] output tristate buffer
- [INFF] input cell with flip-flop
- [INLAT] input cell with latch
- [OUTFF] output cell with flip-flop
- [OUTFFT] output cell with flip-flop and tristate
- [IPAD] input pad
- [OPAD] output pad
- [BPAD] bidirectional pad

## Miscellaneous

- [CLB] configurable logic block - the basic Xilinx structure
- [IOB] input/output block - the Xilinx I/O cell
- [OSC] clock oscillator circuit
- [TITLE] title box

## Local ‘macrocells’

- [keyboard] represents the keyboard switches and other push buttons, including the input pads and buffers. Note that the input signals are **low when a switch is pressed**.
- [key_timing] the keyboard timing encoder; this circuit produces a pulse which is high for a single clock period shortly *after* the input goes high. It is intended to for the keyboard interface to allow time for the key encoding to become valid before signalling that it is ready.
- [clocks] this block includes the on-board clock oscillator and a ripple counter performing a clock division. Outputs are provided at appropriate frequencies for use in other circuits.
- [display] this block contains all the outputs for the 7-segment displays. This display must be multiplexed. The eight outputs to **the seven segments and the decimal point are active low** whilst **the two enable outputs are active high**.
- [bargraph] the bargraph comprises ten individual LEDs. The inputs are **active low** (for LED on).
- [mu_title] a pro-forma title box for neat documentation.

# Experiment 121 - 2: Simple Logic on the Gate Array

## Aim

This experiment is intended as a "hands on" introduction to the schematic capture and gate array layout software and to provide some familiarisation with the inside of the ERA gate array. 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 Powerview 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 Powerview 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 **Powerview Walk-Through** section in this manual. Follow the instructions here carefully.

Note that no physical wiring is needed for this or any of the other gate array experiments; 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. 2.7).**

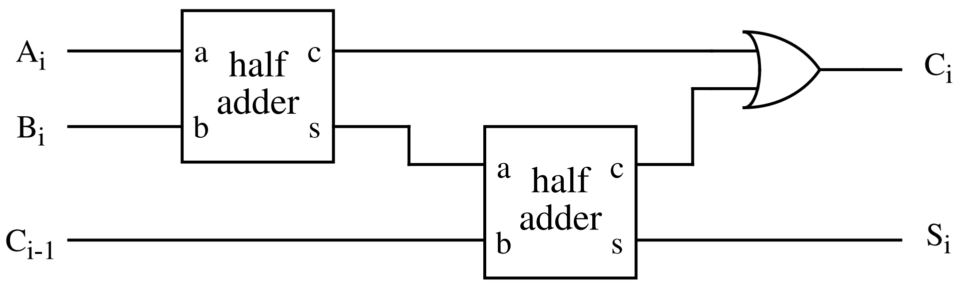



***Figure 6.1: A Full Adder built from two Half Adders***

**As an experiment try selecting one of the half adder symbols and executing the function File ➪ Level ➪ Push ➪ Schematic (also available by pressing the**

**right hand button over the main display and selecting Level ➪ Push Schematic). This will allow you to enter the previously prepared schematic.**

***Note** that the labels 'inside' the components ({a,b,c,s} in figure 2.7) are* local *to that component and therefore not implicitly connected 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 ViewTrace. 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

| D3 | D2 | D1 | D0 | S1 | S0 | Q |
|---|---|---|---|---|---|---|
| X | X | X | 0 | 0 | 0 | 0 |
| X | X | X | 1 | 0 | 0 | 1 |
| X | X | 0 | X | 0 | 1 | 0 |
| X | X | 1 | X | 0 | 1 | 1 |
| X | 0 | X | X | 1 | 0 | 0 |
| X | 1 | X | X | 1 | 0 | 1 |
| 0 | X | X | X | 1 | 1 | 0 |
| 1 | X | X | X | 1 | 1 | 1 |

***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.

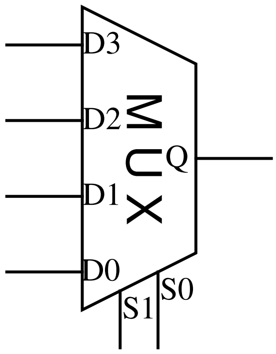



*Figure 6.3: Typical Symbol for a 4:1 Multiplexer*

**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 experiments. (The process for altering the size of a symbol is described on page 28.)**

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

## Plotting

To print a diagram from ‘ViewDraw’ first select ‘Plot Setup ...’ under the red square menu. This produces a window with various options which should be set as follows:

| | | |
|---|---|---|
| **DEVICES** | **POSTSCRIPT** | |
| **PAPER SIZES** | **A4** | |
| **OUTPUT** | **SPOOL** | **Output e1_lw** |
| **EXTENTS** | **<choose as appropriate>** | |

Most of this information should already be correct. When this is correct ‘Accept’ it; the diagram will be sent to the laser printer. (The printer set-up routines are painfully ‘buggy’; fortunately they are rarely needed.)

If the printer setup is already correct a plot can be initiated simply by selecting ‘Plot Go’ under the red square. This sends the diagrams to the printer in the lab., which is available during scheduled lab. sessions.

## Joint reports

To save time and paper you may wish to submit a joint report with your partner. If you are doing this please:

- include **both** people's answer sheets/comment forms
- **one** copy of the diagrams
- **two** blue record cards - one for each person - *both at the front of the report*

# Karnaugh Maps

Various procedures have been developed for simplifying logical expressions as an aid to the logic design process; one of the more useful techniques is the Karnaugh map.

Consider the truth table shown figure 7.1 which defines a function F of four logical variables A, B, C, D. This figure also shows how these variables can be represented on a map of sixteen locations. Each of the sixteen possible combinations of the inputs A, B, C, D has an associated location and the value of the function F for that combination of inputs is placed in this location. An “X” is placed at any location where the input combinations either do not occur or the value of the function does not matter.

| A | B | C | D | F |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | X |

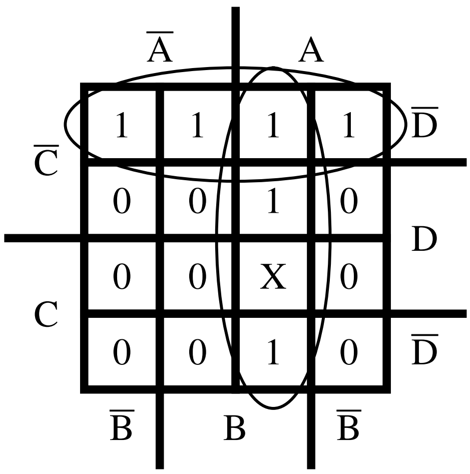



***Figure 7.1: A Truth Table and its corresponding Karnaugh Map***

Once the Karnaugh map has been created terms that are independent of one or more inputs are found. This is done relatively easily by grouping together adjacent “1”s and “X”s that can be used to complete groups of ones.

Such groups should be:

- rectangular
- a power-of-2 wide and high (i.e. 1, 2, 4, etc.)
- as large as possible

It does not matter about:

- overlapping
- ‘wrapping around’ the edges of the map
- including or excluding all the “don’t care” (“X”) cases

For example, by taking the X ("don't care") term as a 1, the group of four terms running vertically to the right of centre in the map is covered simply by the expression A = 1 and B = 1 or A AND B = 1. The 1s on the top row of the map can be identified by the expression $\overline{C}$ = 1 and $\overline{D}$ = 1 or $\overline{C}$ & $\overline{D}$. By ORing of these two expressions together an equation can be derived that covers *all* cases where F must be a 1; since both expressions will produce a 1 for the combination 1 1 0 0 the overlap at this point is unimportant. Thus, the function can be expressed by the following equation:

$$F = A \,.\, B + \overline{C} \,.\, \overline{D}$$

For "sum of products" expressions, such as that derived above, an electrical implementation can be made as shown in figure 7.2.

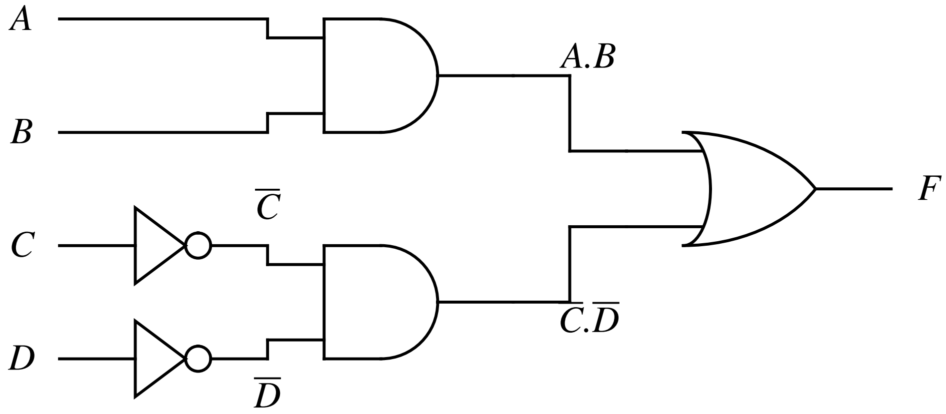



***Figure 7.2: Implementation of F = A.B + $\overline{C}.\overline{D}$***

In Powerview this can be drawn as shown in figure 7.3 (among other ways).

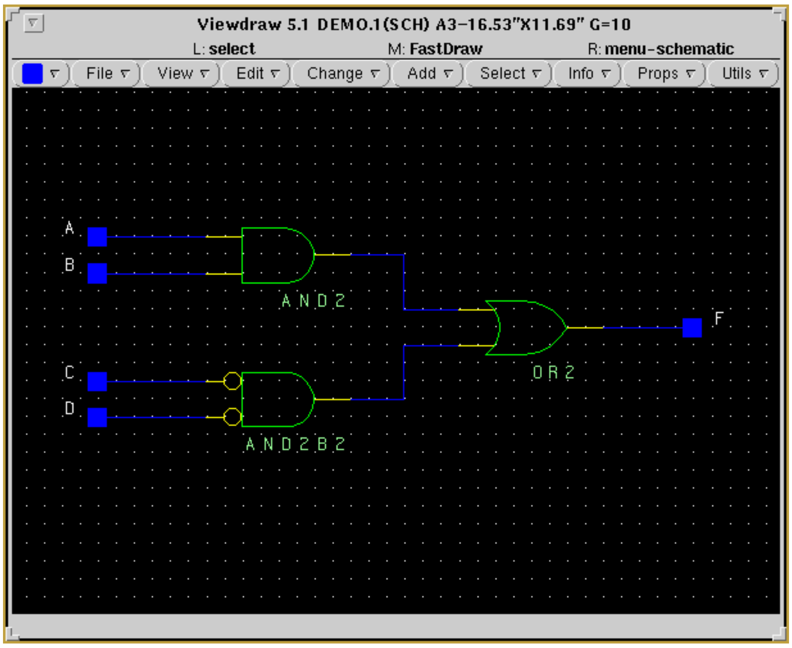



***Figure 7.3: Powerview implementation of F = A.B + $\overline{C}.\overline{D}$***

# Experiment 121 - 3: A Seven Segment Decoder

## Aim

This experiment develops the Powerview work to produce a macroscopic block which can be used in later designs. This is more complex than earlier designs but similar principles still apply.

This is a long experiment. You should do the design work *before* coming into the laboratory.

## 'Milestone'

You should aim to have demonstrated a working simulation of the decoder by the end of the first session. You should at least demonstrate a simulation of one of the segments.

## 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. Examples of this occur in the translation of bit patterns representing functions into the signals which control an arithmetic unit, and the translation of character codes into the signals that drive a printing 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.

In achieving this translation there are generally two main objectives:

1. To keep the translation logic as cheap as possible by minimising the number of integrated circuits used.

2. To translate the data as quickly as possible by minimising the number of levels of gating through which any signal has to pass.

Seven segment displays are commonly used to display the digits from 0 to 9 on pieces of electronic equipment such as calculators, VCRs, 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. 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.

## 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. Thus the individual counters produced in this experiment count in BCD. 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. However BCD numbers are less compact and harder to perform arithmetic on, and so are infrequently used in real applications.

## Segments to Light

Figure 8.1 shows the segments which must be lit to display each of the digits 0-9.

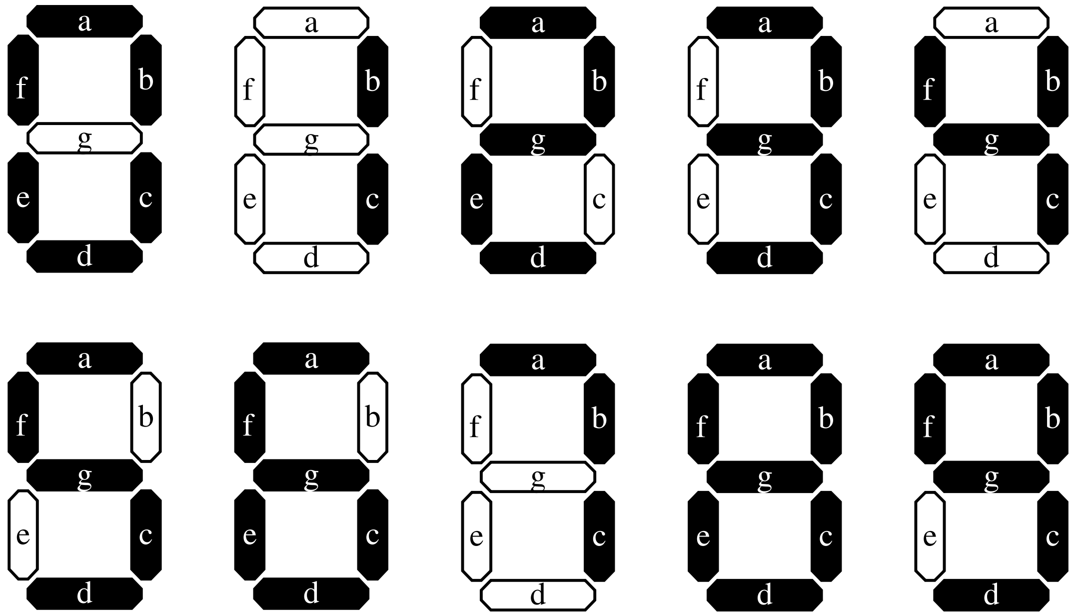



***Figure 8.1: Seven segment patterns for the digits 0..9***

## Example Solution For The Display Driver

The truth table for segment *b* is shown in figure 8.2. The solution shown circles all the 1’s to produce a sum of products for *b*. Alternatively, the 0’s could have been circled to produce a sum of products for $\overline{b}$.

### Practical

**1 Design and implement a seven segment decoder using the x3000 component library. This should accept a 4-bit number as an input and should output signals to illuminate the relevant segments of the display. There is no need to take into consideration numbers greater than nine. The outputs of the circuit should be <u>active-low</u> i.e. an output should be low when its corresponding segment is to be lit up.**

| N | W | X | Y | Z | b |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 0 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 1 |
| 5 | 0 | 1 | 0 | 1 | 0 |
| 6 | 0 | 1 | 1 | 0 | 0 |
| 7 | 0 | 1 | 1 | 1 | 1 |
| 8 | 1 | 0 | 0 | 0 | 1 |
| 9 | 1 | 0 | 0 | 1 | 1 |
| - | 1 | 0 | 1 | 0 | X |
| - | 1 | 0 | 1 | 1 | X |
| - | 1 | 1 | 0 | 0 | X |
| - | 1 | 1 | 0 | 1 | X |
| - | 1 | 1 | 1 | 0 | X |
| - | 1 | 1 | 1 | 1 | X |

| | $\overline{W}$ | | W | | |
|---|---|---|---|---|---|
| $\overline{Y}$ | 1 | 1 | X | 1 | $\overline{Z}$ |
| $\overline{Y}$ | 1 | 0 | X | 1 | Z |
| Y | 1 | 1 | X | X | Z |
| Y | 1 | 0 | X | X | $\overline{Z}$ |
| | $\overline{X}$ | X | X | $\overline{X}$ | |

$$\text{Segment}_b = Y \, . \, Z + \overline{Y} \, . \, \overline{Z} + \overline{X}$$

N.B. This result is for an *active-high* output

***Figure 8.2: Truth Table and Karnaugh map for segment b***

**Keep the schematics as neat as possible; if you prefer each segment may be designed separately and then grouped together by using an extra level of design hierarchy.**

**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). *Only attempt to do this if you have plenty of time.***

**After producing the design create a symbol for it and place it on a separate diagram. Use this to simulate and test the circuit. When your design is correct, show it to a demonstrator.**

**2 Include in your documentation an example of the design process (truth tables/ logic equations/Karnaugh maps/etc.) you have used. One example segment (not the one above!) is sufficient.**

**3 To interface the circuit to the 'outside world' some library cells are provided which implement the Xilinx LCA's peripheral interface and ensure that the pins appropriate to the laboratory board are connected. These are in the library called "englib" and are wired up in the same manner as your own circuits or the basic circuit elements.**

**These cells are described in the section on the Powerview libraries. The contents of these cells may be viewed with 'level' 'push' 'schematic' if you are curious. The cells relevant to this exercise are:**

**"display" is the interface to a twin 7-segment display on the experimental circuit board.**

**"keyboard" is a collection of the various keys available on the test board.**

**Connect your circuit to the seven segment display symbol provided in the "englib" library. This symbol has eight connections on the left-hand side, one for each segment and one for the decimal point. *Unused inputs on a circuit should always be connected to a logic level*; in this case the decimal point should be wired inactive (i.e. 'high'). In addition there are two 'enable' inputs on the display symbol. These are active high and so one should be chosen and connected to a 'high'[1] (which turns that display on) whilst the other is turned off ('low'[2]).**

**To simulate test input choose four buttons from the keyboard, available in the same library, and use these as inputs. Remember that the keys are also active-low and it may be convenient to invert these signals for test purposes.**

## 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 saved the design from Powerview ('file' 'write'). From a shell window type:

xxx <filename>

**Note**: do not include the ".1" extension in the filename.

The compilation process will take several minutes. As there are a limited number of licences for the software it is possible (though unlikely) that the compiler may abort; if so wait for about a minute and try again. If the compiler fails because of errors and you are unable to decipher the error messages consult a demonstrator.

When compilation is complete the design must be downloaded. Consult a demonstrator to obtain one of the circuit boards; they will connect it for you. Check that the yellow LED is lit on the circuit board; if it is not ensure that the board has power and that the "serial/EPROM" switch is set to "serial", then press the red button on the circuit board. The yellow light should now be on. Then type:

xld <filename>

The programme will download in a few seconds and the design should run automatically. Whilst the programme is loading the green LED should come on. On completion the yellow and green LEDs should both go out and the circuit should be operating.

**Test the implementation of the design on the circuit board. If it is functioning show it to a demonstrator, otherwise find out what is wrong using ViewSim.**

**Attach your schematic diagram(s) to the answer sheet for handing in.**

1. In the x3000 library logic 'high' is called "Vcc"
2. called "GND"

# Experiment 121 - 4: Counters

## Aim

To investigate time dependent circuits. This experiment 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.

To illustrate a practical use for **multiplexing**. Multiplexing is a technique where more than one signal shares a single wire.

## Preparation

Read the experiment description thoroughly. Ensure that you are familiar with the function and operation of D-type flip-flops. In addition prepare the logic design for the counter; there will not be time to do this in the laboratory.

## 'Milestone'

You should be able to show logic equations for the modulo-10 counter by the end of your fourth CS121 laboratory session.

## D-type flip-flops

Two types of storage element - S-R flip-flops and transparent latches -have already been met in these laboratories. A third sort of flip-flop - the D-type flip-flop {figure 9.1} - is introduced in this experiment.

Unlike a latch, which is controlled by a voltage level, a D-type flip-flop is enabled by a **transition** (or **edge**) on the control signal. This edge will always be same direction for a particular device but - depending on the device - may be low⇒high[1]or high⇒low[2]. The edge which causes the transition is known as the **active edge**; the other transition which occurs between active edges is known as the **inactive edge**.

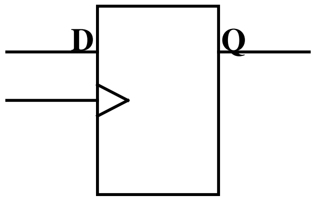



***Figure 9.1: Symbol for a D-type flip-flop***

This response to an electrical *edge* rather than a level means that the flip-flop captures an instantaneous 'snapshot' of its input {figure 9.2}.

1. sometimes called a **positive edge** or **rising edge**
2. sometimes called a **negative edge** or **falling edge**

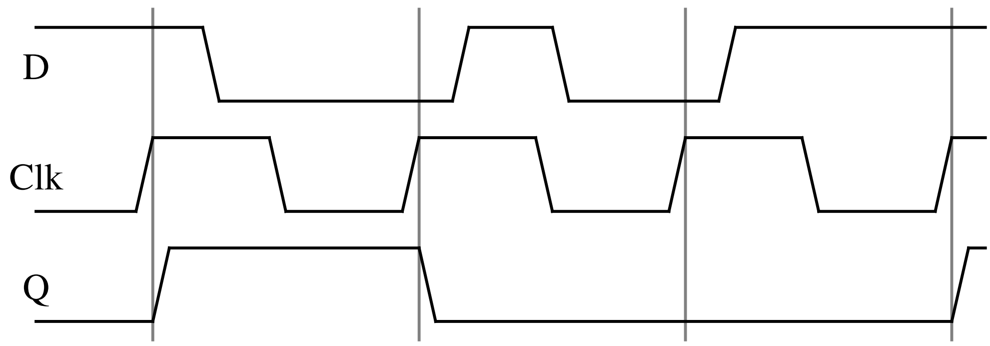
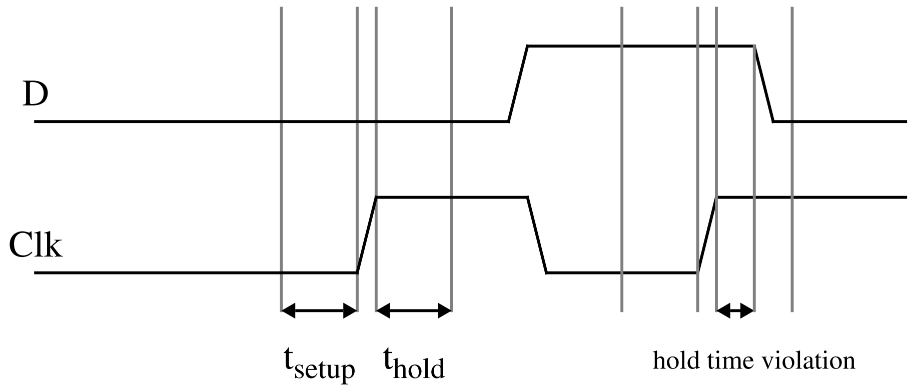



***Figure 9.2: Behaviour of a D-type flip-flop***

The control input of a D-type flip-flop is usually known as the **clock input**. This is normally indicated by a triangular marking next to the input connection (see fig. 9.1). A falling edge triggered flip-flop will normally have a 'bubble'[1] on the clock input, indicating the inversion.

Although an ideal D-type flip-flop changes its **state** instantaneously in response to the clock input, a real circuit merely changes very fast. In general when D-type flip-flops are used care must be taken to ensure that the input does not change just before or just after an activating clock transition. The times before and after the control edge are known as the **set up** and **hold** times respectively and violation of these limits will cause the circuit to function unpredictably. These restrictions should not affect this laboratory, but you should be aware of them.




## Introduction to Counters

Counters are simple but useful **Finite State Machines** (FSMs). These are circuits which change their **state** each time they receive a **clock** signal. In this context "state" refers to the information stored in the circuit, thus a single D-type flip-flop has two possible states, corresponding to it storing a "0" and storing a "1". In a finite state machine the state is contained as a particular configuration of a number of bits. For instance a counter which cycles though sixteen states must contain (at least) 4 bits, as $2^4$=16.

Counters may be loosely subdivided into **up** counters and **down** counters. As might be

1. Small circle indicating inversion; see figure 9.3 for an example.

expected an ‘up counter’ will produce a sequence of values 0,1,2,3, ... and a ‘down counter’ will produce a sequence such as 5,4,3, ... Normally when a counter reaches its terminal value it starts back at its initial value. Thus a 4-bit *binary*[1] up counter would produce a sequence:

0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0,1,2,3,4 ...

This sequence has sixteen discrete states and so it is also referred to as a “modulo 16” counter. A counter need not cycle through all its possible states; a 4-bit counter could be made which cycled through (for example) thirteen states; this would be a modulo 13 counter. Note that this would also contain three other ‘forbidden’ states.

Regardless of the direction of counting there are basically two families of counters which are used in digital circuits: these are **synchronous** counters and **asynchronous** (or ‘ripple’) counters. The difference between these is that all the bits synchronous counters change state at the same time, whereas in an asynchronous counter only one bit is clocked; this bit changing in turn *may* cause the next bit to change - the change of state therefore “ripples” through the circuit.

## Ripple Counters

Ripple counters are the simpler of the two varieties of digital counter. Each stage of a ripple counter is a single flip-flop which changes state each time each time there is a **falling edge**[2] on its clock input. This type of flip-flop is sometimes known as a **toggle** flip-flop {figure 9.3}. The

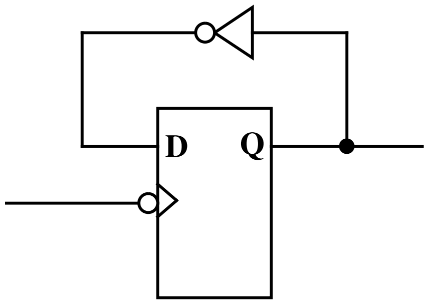



***Figure 9.3: Toggle flip-flop***

logic and timing diagrams for a 3-bit asynchronous up counter is shown in figure 9.4. In this figure it is assumed that each flip-flop has complementary outputs.

A single toggle flip-flop divides the input clock frequency by two; that is it produces a falling edge on its output for every two falling edges on its input (the other causes a rising edge). If several toggle flip-flops are connected together in series each stage performs a division by two, thus the input frequency may be divided by 2, 4, 8, 16, 32 and so on.

Because the clock signal ‘ripples’ from flip-flop to flip-flop the outputs from the various stages change at different times. This means that - even though it may be functioning correctly - the counter’s outputs may not always represent the expected ‘value’. This problem increases at high frequencies as is demonstrated in figure 9.5. There is therefore an upper limit to the fre-

1. In this context binary refers to a counter which cycles through *all* its possible states.
2. for an up counter; a rising edge for a down counter

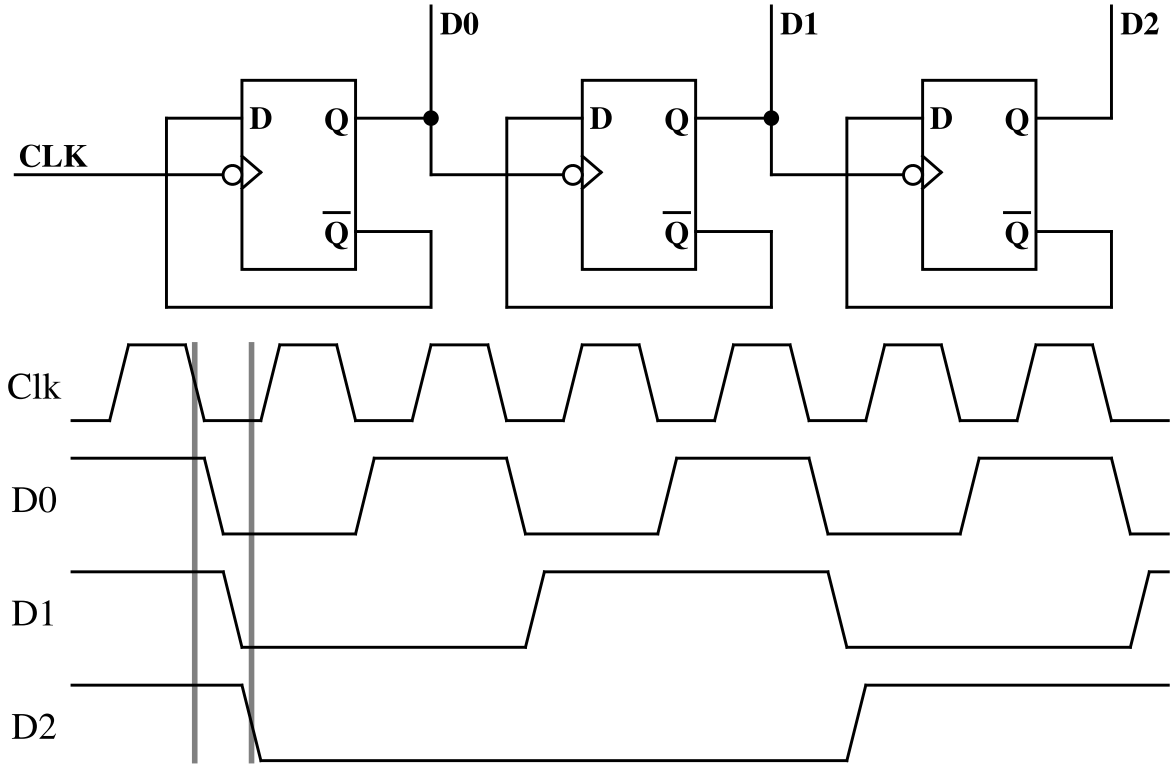



*Figure 9.4: Diagrams for a 3-bit ripple counter*

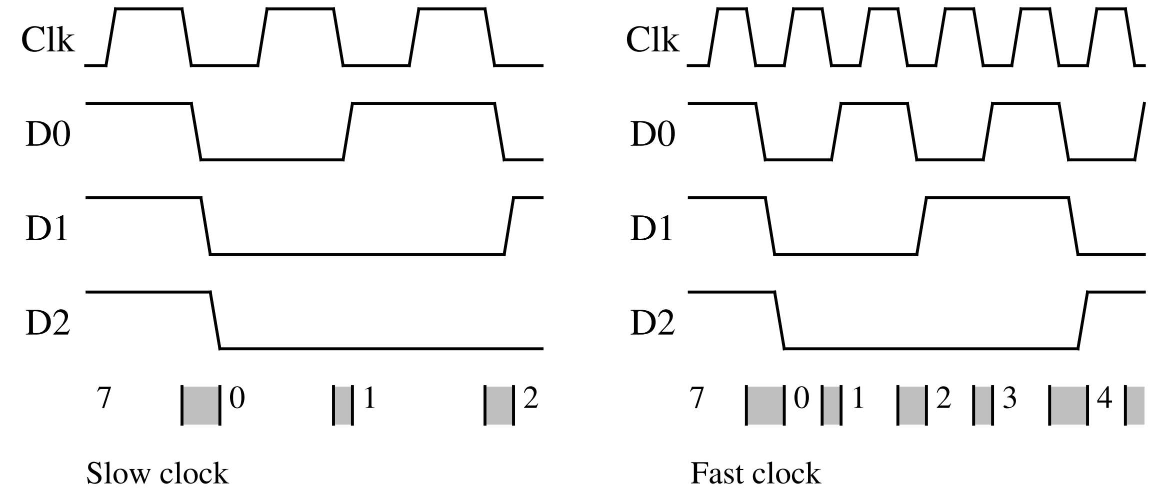



*Figure 9.5: Timing of a 3-bit ripple counter*

quency of such counters which is governed by both the speed of the switching circuit and the number of bits in the counter.

## Synchronous Counters

Although useful, ripple counters suffer from certain drawbacks. Firstly the outputs change at different times; for instance if a counter moves from 01111111 to 10000000 it passes through seven intermediate stages which may take a long time and cause the output to adopt undesirable configurations. Secondly a ripple counter can only (easily) count in powers of 2 (2, 4, 8, etc.).

Synchronous counters overcome these problems. In a synchronous counter all the output flip-flops are clocked simultaneously, thus all the outputs change at the same time[1]. Counting is achieved by using logic to control the inputs of the flip-flops. An example of this, describing a three bit counter is shown in figure 9.6. Note that in this and all the following descriptions the least significant bit is labelled *A*.

$$A := \overline{A}$$

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

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

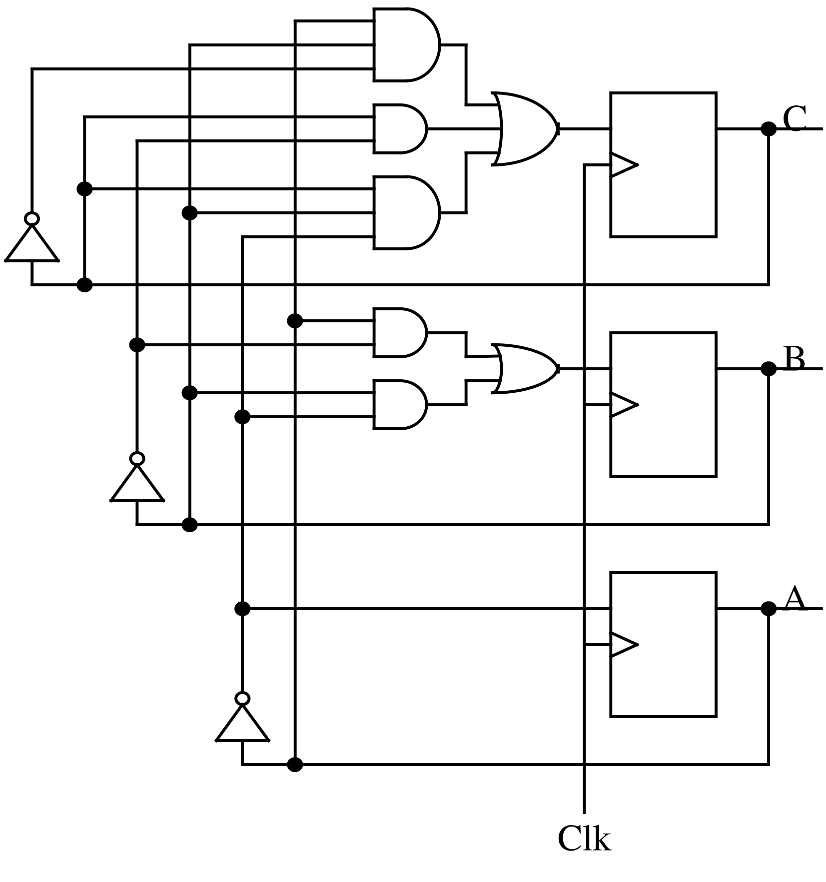



***Figure 9.6: Circuit diagram of a 3-bit Synchronous Counter***

Remember that in a synchronous counter the flip-flops do not change until the clock edge, at which point they all change simultaneously. The propagation delays through the flip-flop and the combinatorial logic ensures that the inputs do not change immediately after the clock edge which helps to satisfy the operating conditions (i.e. hold time) of the flip-flop.

1. Slight differences may be encountered due to different propagation delays, but these are negligible

By altering the logic a synchronous counter may be made to count in cycles of an arbitrary number of states. Thus a four bit counter may be used to count in cycles of from 9 to 16 states (less than nine states can be represented but *require* less than four bits). Thus a 4-bit counter could be made that was (say) modulo 11.

The most common 4-bit counters are modulo 16 and modulo 10, representing the two most common counting systems in computing. These are also known respectively as **binary** and **decade** counters.

## Xilinx flip-flops

The D-type flip-flops for the following designs are explicitly given in the *X3000* library. These are called **DFF** and are slightly dissimilar from models which may have been met before, so they are described in figure 9.7 below.

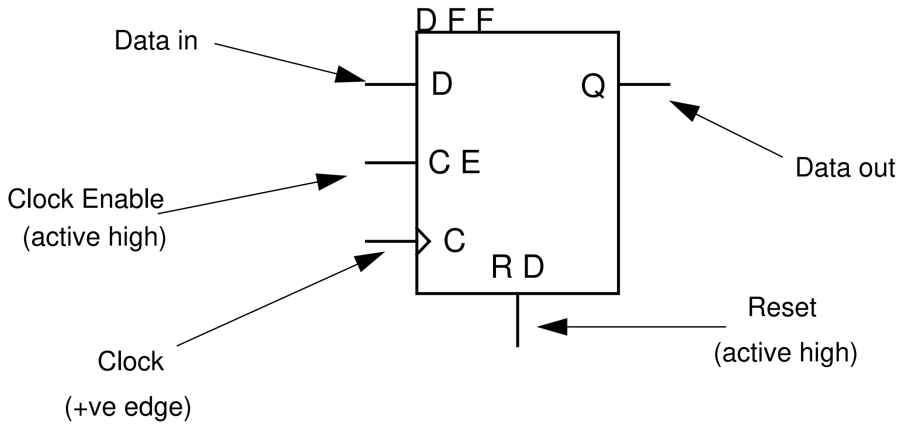



***Figure 9.7: X3000 D-type flip-flop (DFF)***

- The flip-flop will only respond to an active clock edge if Clock Enable is high.
- The active high Reset forces the output (Q) to “0”; this is an *asynchronous* function - i.e. it does not require a clock transition to operate.

**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 about 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 other 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 experiment. For neatness this counter should be designed as a separate schematic with its own symbol, as should the 7-segment decoder.**

**In order to determine the correct logic functions for the counter it is necessary to determine the inputs for the *next* state from the current one. The following state transition table (when completed) may prove useful in this.**

| Current state | Next state |
|---|---|
| 0 0 0 0 | 0 0 0 1 |
| 0 0 0 1 | 0 0 1 0 |
| 0 0 1 0 | |
| 0 0 1 1 | |
| 0 1 0 0 | |
| 0 1 0 1 | |
| 0 1 1 0 | |
| 0 1 1 1 | |
| 1 0 0 0 | |
| 1 0 0 1 | 0 0 0 0 |

**From this table the logic functions may be derived for the inputs to each flip-flop. This in turn allows the logic gates to be defined in a style similar to that shown in figure 9.6. It is probably beneficial to test the counter using the simulator.**

## Simulating a Clocked Circuit

A clocked circuit such as a counter may be simulated by manipulating the signals individually as is usually done for purely combinatorial logic. However it is very tedious to set the clock high, step, set the clock low and step for every cycle.

ViewSim allows signals to be designated as clocks and can perform this tedious repetition automatically. A clock is specified with the function Setup ➪ Clock... which creates a small window. Into this enter the required signal name (this will default if there is a signal selected on the schematic) and the pattern that the clock should take. The pattern ("Transition sequence" is specified by a string of "0"s and "1"s, *separated by spaces*. A pattern "0 1" specifies a single clock pulse; if "0 1 0 1" is entered the pattern will contain two pulses. 'Uneven' clocks may be produced using sequences such as "0 1 0" which is only 'high' for one third of its period. This variation is known as the **duty cycle** or the **mark-space ratio** (33% or 1:2 respectively in the preceding example). 50% duty cycle clocks are by far the most commonly encountered however.

When a clock has been defined it can be used for a simulation run. The command Sim used previously will advance the clock by a single **phase**, so that two or more commands will be required to cycle through the whole clock sequence. This allows changes to be made to other signals during the simulation. Selecting Sim ➪ Cycle will run a whole clock cycle, making life easier when no other input is required.

Unfortunately the clock must be re-affirmed after executing Sim ➪ Restart.

*Note that there is a 'problem' with the simulator when it comes to the initialisation of the counter. In the real circuit it is adequate to ensure that the counter is in - or will end up in - one of the legal states for the counter to function correctly. This is because the counter is always in* one *of its possible states and its next state can always be predicted. However under the simulator the circuit is initialised into an* unknown *state and the next state is therefore also unknown (i.e. no change). It is therefore necessary to connect and use the* reset *signal on the flip-flops for simulation purposes. Nevertheless, simulation is recommended for debugging.*

*A flip-flop can (apparently) be cleared by using* Set ➪ L... *to force its output low. This is not a good way of doing things; it does not clear the flip-flop and it can mask problems in the circuit. If you force someting to a value remember that it will remain at that value until told otherwise (e.g. with* Set ➪ Release...*).*

### Practical

**For the physical implementation another entry in the "englib" is useful; this is "clocks", an on-board oscillator and clock divider which provides several output frequencies. The numbers on the pins on the right hand side of the symbol indicate the *approximate* clock frequency in hertz**

**For neatness include a figure description on each schematic. This serves an important documentation function in labelling the diagram and giving it a version number and a date so that differing versions can be distinguished. A typical template for this can be found in "englib" called "mu_title" and it is usual to place this in the lower right hand corner if the figure. Remember to fill in the empty boxes!**

## Glitches

A **glitch**[1] is a brief, unwanted level on a signal. Glitches are caused when two gate inputs change at the same time, or very nearly so. Thus if one input to an AND gate goes low → high at about the same time (say, less than the propagation delay of the gate) as the other input goes high → low then there is a chance of a high-going glitch at the output.

In most cases these do not matter, but when dealing with clock signals the glitch can cause the spurious clocking of flip-flops which can alter the state of the circuit at the wrong time. This is caused by 'gating' clock signals. For example in the following practical exercise it is possible

1. Also known as a "runt pulse"

to produce a solution to the problem by ANDing the input clock with a carry signal. This can cause a problem similar to that shown in figure 9.8. Here the ‘En’ signal is caused to change by the ‘Clk’ signal and so will trail the clock by a short propagation delay. The output from a subsequent AND gate will also de delayed slightly, but the skew may (or may not) cause a glitch at this point. This could cause an unwanted clock edge if the output is used to trigger edge-sensitive devices (D type flip-flops, counters etc.).

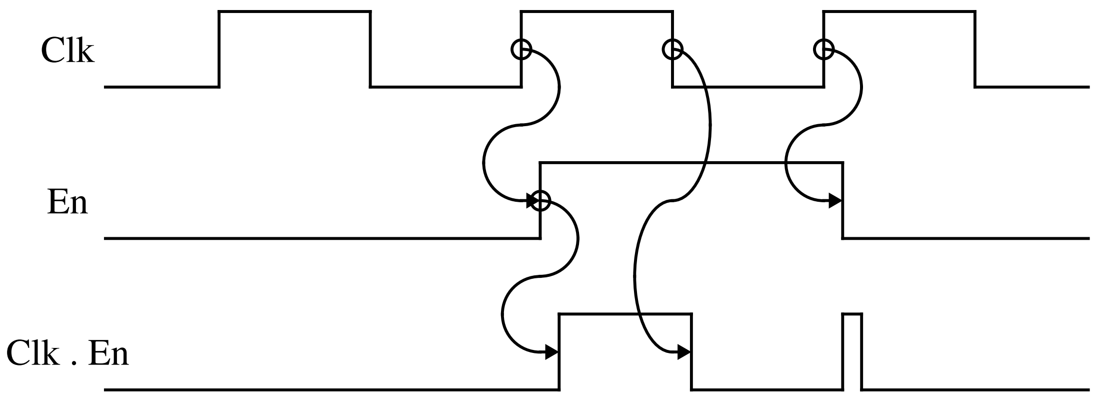



***Figure 9.8: Glitch production by gating a clock signal***

The best rule is *never* to gate clock signals.

## Time-Domain Multiplexing

Multiplexers were described in an experiment 121 - 2; it is now time to utilise a component produced in that session.

Both of the displays on the lab. board share the same 8-bit bus to set up their bit patterns. If it is required to display all the numbers from 00 to 99 then in most cases the digits will be different. To illuminate different digits on both displays ‘simultaneously’ they must actually be flashed alternately at a high enough frequency to fool the eye; this is **time-domain multiplexing**.

Time-domain multiplexing is not as intimidating as it sounds. It is a common technique for being in two (or more) places at once. It is the same as ‘time-slicing’ in an operating system where short bursts of different programmes are run interleaved with each other in an attempt to make it appear that several things are actually happening concurrently.

**Practical**

**2 Extend the counter circuit from part 1 of this exercise so that it cycles through the states 0 to 99, displaying the numbers in the process.**

**First modify the decade counter definition so that it includes 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. 9.9}. 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”).**

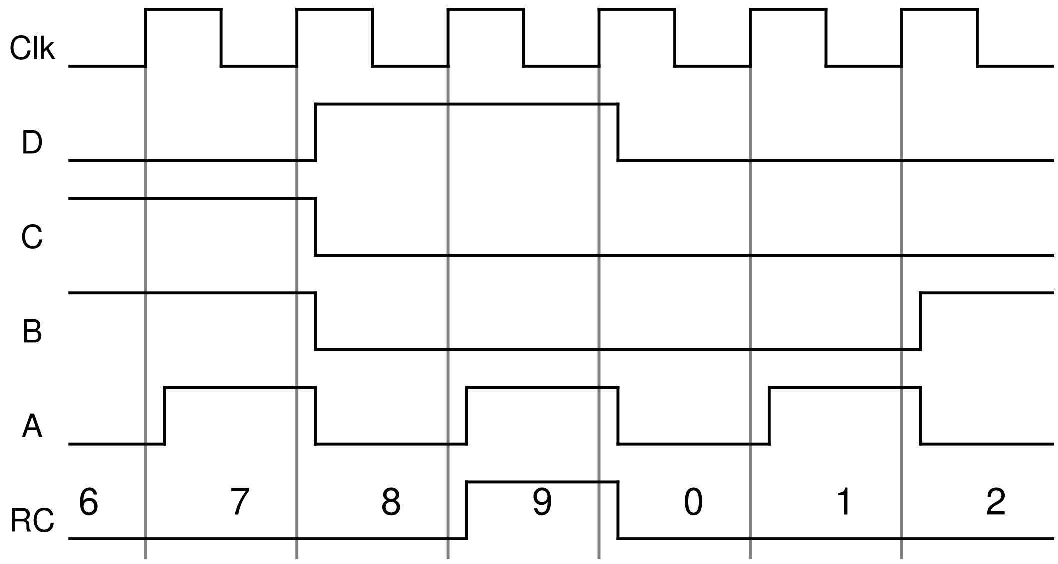



*Figure 9.9: Timing Diagram for a Decade Counter with a Ripple Carry*

**In order to display two digits ‘simultaneously’ they must be time-domain multiplexed. This can be done either before or after the digits are decoded. If the multiplexing occurs before the decoder however only four signals need to be multiplexed (the decoder inputs) and only a single decoder is required whereas multiplexing after the decoder requires two 7-segment decoders and twice as many multiplexers. Each input to the 7-segment decoder must be connected to the output of a 2:1 multiplexer which is fed by the two possible inputs (left and right digits). The control signal on the multiplexer should be fed by a clock signal to swap between the two displays.**

**To ensure that only one of the displays is lit at any instant the multiplexer signal should also go to the display enable signals. Because the displays alternate one of the displays will be enabled directly by this signal whereas the other must be enabled by its inverse.**

**If you are unsure of your design for this consult a demonstrator.**

*If this multiplexing is too slow then the displays will flicker visibly. Conversely if the frequency is too high the displays will ‘smear’ together because of the finite switching speed of the circuitry. In practice a frequency of about 1kHz works quite well. Note that when both displays are in use they appear dimmer than when just one is lit.*

## Finite State Machines

A finite state machine is a system that may adopt one of a number of **states**. It may change from one state to another in response to various inputs. Usually the change in state is controlled by a single clock signal.

The decade counter in the previous experiment is an example of a finite state machine; it has ten possible (legal) states and on each active clock transition it changes into the subsequent state. Thus - for example - from state “7” it would change into state “8” at the next clock edge.

This behaviour may be depicted on a **state diagram** {figure 9.10}.

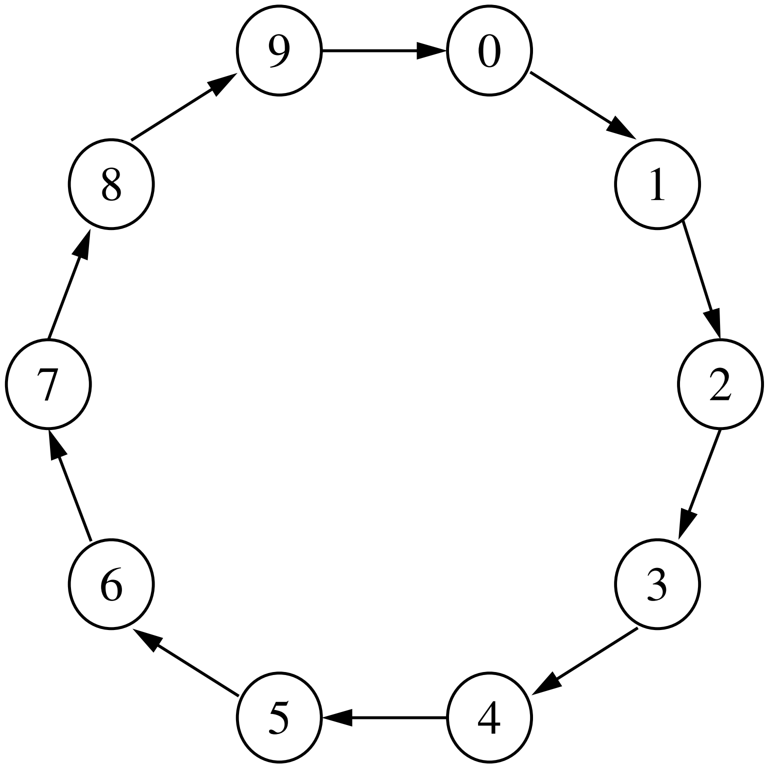



***Figure 9.10: State Diagram for a Modulo-10 Up Counter***

The simple counter lacks one element of a general state machine; it does not have any control inputs and therefore cannot make decisions. Imagine adding an *enable* input to the counter. It would now change from state “7” to state “8” at the next clock edge only if the **enable** input was active, otherwise it would stay in state “7”. The state diagram for this circuit is shown in figure 9.11.

In general 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 - not shown on the preceding figures - 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.

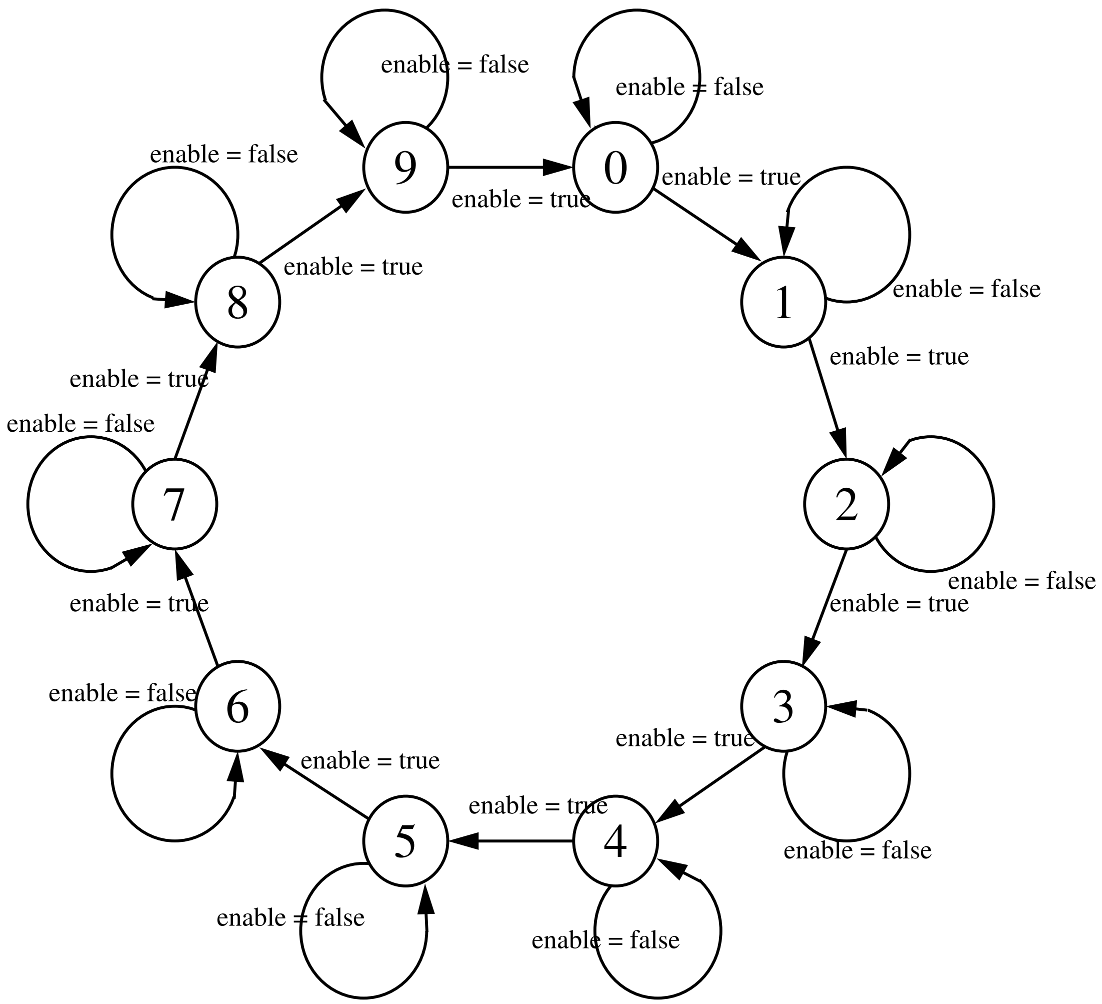



***Figure 9.11: State Diagram for a Modulo-10 Up Counter with*** **enable**

**Practical**

**3 Sketch the state diagram for a modulo 5 up/down counter where the control input *direction* is high for counting up and low for counting down (only a single control signal need be considered). Abbreviate the signal naming appropriately.**

# Boolean Logic Identities

The identities given in this section may be useful in manipulating logic equations in the electronics laboratory during the course of the year.

There are 3 basic logical operators: AND, OR and NOT

AND is represented by a period or dot: “.”

OR is represented by a plus sign: “+”

NOT is represented by an overscore, e.g. NOT A = $\overline{A}$

$$\overline{\overline{A}} \equiv A$$

$$A + (B+C) \equiv (A+B) + C$$

$$A \,.\, (B+C) \equiv (A.B) + (A.C)$$

$$A + (B.C) \equiv (A+B) \,.\, (A+C)$$

| | |
|---|---|
| $A \,.\, T \equiv A$ | $A + F \equiv A$ |
| $A \,.\, F \equiv F$ | $A + T \equiv T$ |
| $A \,.\, A \equiv A$ | $A + A \equiv A$ |
| $A \,.\, \overline{A} \equiv F$ | $A + \overline{A} \equiv T$ |

| | |
|---|---|
| $A + A.B \equiv A$ | $A \,.\, (A+B) \equiv A$ |
| $A + \overline{A}.B \equiv A+B$ | $A \,.\, (\overline{A}+B) \equiv A.B$ |
| $A \,.\, B+\overline{A} \,.\, B \equiv B$ | |

Where: A, B and C are signals
T is ‘TRUE’
F is ‘FALSE’

## Alternative notation

Sometimes the following symbols for logical operations are encountered:

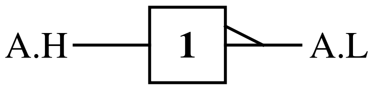
$\wedge$ ≡ ‘AND’ $\qquad$ $\vee$ ≡ ‘OR’ $\qquad$ $\veebar$ ≡ ‘XOR’

These can be useful to avoid confusion in circumstances where both arithmetic and logic functions occur in proximity.

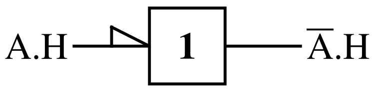
# Dependency Notation

Dependency notation is not a design technique, rather it is a method of documentation. It has been introduced because the usual means of documenting a design, the logic diagram, is used to represent on the same sheet of paper both the logical design and the electrical implementation. In any top-down design approach, whether software or hardware, the logical design should not be unduly influenced by the implementation. It is desirable that the documentation should reflect the dichotomy between the two, and it is this that the dependency notation approach attempts to achieve. These notes have been included as an addendum and are intended to clarify the use of the dependency notation and to outline its general philosophy. They are intended to read in conjunction with the script for experiment 121 - 1.

In brief dependency notation dictates that **active low** signal paths must start on a triangle. They must also end on a triangle *unless a logical inversion is implied*. **Active high** signals start without triangles. Thus the following diagram indicates a logical signal $A$ being translated from an active high to an active low representation:




whereas




represents the logical inversion of $A.H$ to produce $\overline{A}.H$. Notice the two different representations of an 74HCT04 “inverter”.

A common reaction on first encountering dependency notation is to be confused by how to deal with the logical *NOT*. Often this is a hang-over from the confusion introduced by the positive logic convention. In most cases logical *NOT* is not what is actually required at all. As an example, a common requirement is for an active low $RESET$ signal to be distributed throughout the system. In the dependency notation there is no confusion - the signal is labelled $RESET.L$, and the reader of the circuit diagram can immediately deduce that the signal is a $RESET$ signal and is asserted low. In the positive logic convention however, this signal would be labelled $\overline{RESET}$. The reader of the circuit diagram is invited by this notation to interpret the signal as $NOT\ RESET$ - which is precisely the opposite of what is intended. In practice, of course, the reader undergoes some mental contortion and *usually* arrives at an understanding of the actual operation of the circuit - but why make life difficult in the first place?

Furthermore, humans find the concept of a negated variable difficult to manage. At a trivial level, the action of a signal $LAMP\_OFF$ is easier to comprehend than a signal NOT $LAMP\_ON$. The point of this discussion is that negated signals are only rarely needed and should not be distributed around the system. This is not an absolute prohibition - rather a matter of style. If the circuit designer really is sure that he wants a signal called NOT $LAMP\_ON$ he is free to use one. However, the signal still needs to have a suffix to denote the voltage level of the signal when asserted. Thus the signal would be written $\overline{LAMP\_ON.L}$ or $\overline{LAMP\_ON.H}$

In view of the remarks above, it is valid to ask if negation is needed at all.

The answer is yes, but that logical negation arises from the synthesis of logical expression and the documentation should reflect this. The negation occurs naturally at the input of the gate implementing the Boolean expression, and should not be an intrinsic signal property. The point is elaborated later, but first let us see how logical *NOT* arises in dependency notation.

Figure 10.1 shows two implementations of an OR function $C=A+B$, where $A$, $B$, $C$ are logical variables. In dependency notation, the presence of a triangle denotes an active LOW signal so that the first representation tells us that the inputs $A$, $B$ are active high, whereas the second indicates that the inputs $A$, $B$ are active low.


***Figure 10.1: Gates Implementing Logical OR***

However, the assumptions regarding the asserted levels of the input variables are associated with the logic gate implementing the logic function, whereas the actual signals put on the inputs are determined elsewhere. Let us see what happens if there is an inconsistency in the choice of asserted logic levels. Suppose for example that $A$ is the signal $OPEN.H$ and $B$ is the signal $STOP.L$. Consider the case of the first OR function. If the signal $OPEN$ is TRUE then it is asserted HIGH, corresponding to the logical input A being asserted. Thus, $A = OPEN$. On the other hand if the signal $STOP$ is TRUE then it is asserted LOW corresponding to $B$ being not asserted. Thus $B = \overline{STOP}$. The actual operation of the gate in terms of the logical variables is therefore:

$$C = OPEN + \overline{STOP}$$

Similar reasoning shows that the second OR function implements the operation:

$$C = \overline{OPEN} + STOP$$

In general the rule is that if a signal with one asserted level is taken to a gate having a different assumption regarding the asserted voltage level, the signal will be logically inverted by the gate. Further, if a consistent notation is maintained so that active low signals having a '.L' suffix start on a triangle, and active high signals start from a source having no triangle, we can formulate the following rule:

A signal is logically inverted *at its destination* if there are non-matching triangles at source and destination.

We are now left with the problems of how to introduce logical negation wherever we need it, and the slightly thornier problem of how to document it. Let us specify a very simple problem. As part of the control of a washing machine, it is required to activate the spin-drier motor if the spin function is selected, but - for safety reasons - not if the door is open. Thus we need to form the Boolean expression:

$$MOTOR = SPIN \,.\, \overline{OPEN}$$

Assume that we are constrained to use a simple AND gate, that we are supplied with two signals *SPIN.H*, *OPEN.H* and the motor interface needs an active high signal to start the spin-drier. If the signals are taken directly to the inputs of the gate, the resulting output will be

$$\text{MOTOR} = \text{SPIN} \;.\; \text{OPEN}$$

This is not what is required, and the asserted voltage level of the signal *OPEN* needs to changed - this is simply done with the aid of an inverter or voltage translator. Of course this is no different to what a designer using the positive logic convention would do. Care however needs to be taken with the documentation, and there are three subtly different representations as shown in figures 10.2, 10.3 and 10.4.

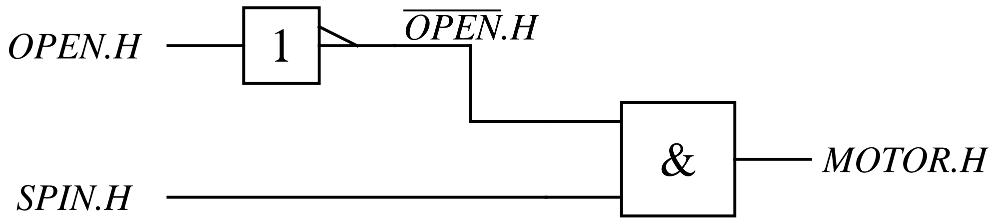



***Figure 10.2: Not Recommended***

The documentation of figure 10.2 has the superficial advantage that the logical negation is explicitly shown in the signal name. There are however two significant disadvantages:

1 The notation is no longer consistent in that the active high signal ($\overline{OPEN}$) starts on a triangle.

2 We have generated a logical signal ($\overline{OPEN}$) which, strictly speaking, is not necessary since the negation is part of the Boolean expression which we are trying to synthesise. In the simple case shown above this objection is of minor importance. However, if the output of the inverter is taken to other points in the circuit, we are distributing around the system a negated variable which, as pointed out earlier, is undesirable.

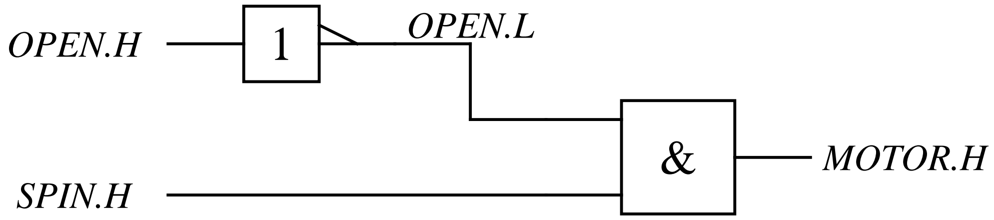



***Figure 10.3: Recommended Documentation***

The logic diagram of figure 10.3 removes both drawbacks, and is the recommended method. The documentation is consistent in that the active low signal starts on a triangle and, by applying the dependency notation for negation, it can be seen that the Boolean function required (including the negation) is synthesised at the gate. It does have the disadvantage, at least for those used to working with the positive logic convention, that the negation is implicit rather than explicit.

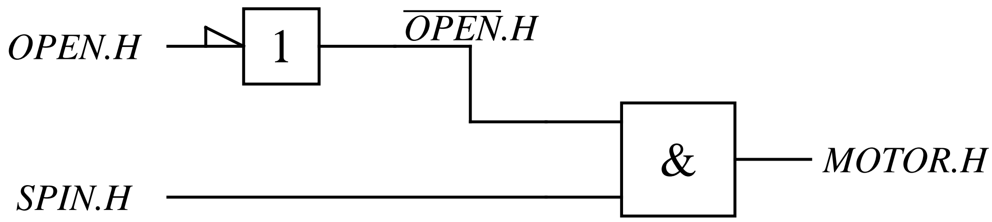



***Figure 10.4: Not Recommended***

The circuit of figure 10.4 gets round the first objection to figure 10.2. However, the second objection still remains. Furthermore, the negation has now effectively been moved away from the AND gate to the input of the inverter.

It should be noted that all three logic diagrams above represent exactly the same hardware, and thus produce exactly the same results. Remember that in the documentation for a real system, the pin numbers and board positions of the gates should be included.

# Experiment 121 - 1: Answer Sheet

**Name:**

**Group:**

**Experiment date:**

## Part 1

Fill in the missing columns.

NOR gate

| INPUTS | | OUTPUT |
|---|---|---|
| F | F | |
| F | T | |
| T | F | |
| T | T | |

## Part 2

NOR gate equivalent circuit

## Part 3

Fill in the missing columns.

XOR gate

| INPUTS | | OUTPUT |
|---|---|---|
| F | F | |
| F | T | |
| T | F | |
| T | T | |

An exclusive-OR gate is also known as a “not-equivalent” gate because:

## Part 4

XOR gate equivalent circuit

Demonstrator’s signature for parts 1 .. 4:

## Part 5

Fill in the missing SUM and CARRY columns.

| INPUTS | | CARRY | SUM |
|---|---|---|---|
| 0 | 0 | | |
| 0 | 1 | | |
| 1 | 0 | | |
| 1 | 1 | | |

Complete the logical expressions:

SUM =

CARRY =

Full adder design:

Demonstrator's signature for part 5:

## Part 6

Flip-flop circuit diagram:

## Part 7

What is the normally disallowed state for this flip-flop? :

What happens if both control lines are simultaneously deasserted? :

Is this behaviour consistent? :

## Part 8

Circuit diagram for D latch:

Demonstrator's signature for part 7 and 8:

## Questionnaire

Please fill in the following short form by circling the appropriate number.

Length of experiment: (1 = too long)- 1 2 3 4 5 - (5 = too short)

Interest of experiment: (1 = boring)- 1 2 3 4 5 - (5 = interesting)

Value of experiment: (1 = no value)- 1 2 3 4 5 - (5 = very useful)

Any other comments:

**Don't Forget** to hand in these experiment progress sheets.

# Experiment 121 - 2: Answer Sheet

**Name:**

**Group:**

**Experiment date:**

## Parts 1 & 2

Demonstrator's signature for parts 1 & 2

## Part 3 & 4

Demonstrator's signature for parts 3 & 4

Attach schematics of your circuit, including any symbols for them, to this answer sheet before handing it in.

## Questionnaire

Please fill in the following short form by circling the appropriate number.

Length of experiment: (1 = too long)- 1 2 3 4 5 - (5 = too short)

Interest of experiment: (1 = boring)- 1 2 3 4 5 - (5 = interesting)

Value of experiment: (1 = no value)- 1 2 3 4 5 - (5 = very useful)

Any other comments:

**Don't Forget** to hand in this experiment progress sheet.

# Experiment 121 - 3: Answer Sheet

## Name:

## Group:

## Experiment date:

## Part 1

Demonstration of simulation: pass/fail

Demonstrator's signature for part 1

## Part 2

Example derivation of one segment (use other side or attach loose sheet).

## Part 3

Demonstration of circuit: pass/fail

Demonstrator's signature for part 3

Attach schematics of your circuit, including any symbols for them, to this answer sheet before handing it in.

## Questionnaire

Please fill in the following short form by circling the appropriate number.

Length of experiment: (1 = too long)- 1 2 3 4 5 - (5 = too short)

Interest of experiment: (1 = boring)- 1 2 3 4 5 - (5 = interesting)

Value of experiment: (1 = no value)- 1 2 3 4 5 - (5 = very useful)

Any other comments:

**Don't Forget** to hand in this experiment progress sheet.

# Experiment 121 - 4: Answer Sheet

**Name:**

**Group:**

**Experiment date:**

**'Milestone'**

Demonstration of design methodology: pass/fail

Demonstrator's signature

## Part 1

Demonstration of design: pass/fail

Demonstrator's signature for part 1

There is no need for schematics for this part of the experiment.

## Part 2

Demonstration of design: pass/fail

Demonstrator's signature for part 2

Attach schematics of the circuits <u>used in part 2</u> to this answer sheet before handing it in.

## Part 3

State diagram for a modulo-5 up/down counter:

## Questionnaire

Please fill in the following short form by circling the appropriate number.

Length of experiment: (1 = too long)- 1 2 3 4 5 - (5 = too short)

Interest of experiment: (1 = boring)- 1 2 3 4 5 - (5 = interesting)

Value of experiment: (1 = no value)- 1 2 3 4 5 - (5 = very useful)

Any other comments:

**Don't Forget** to hand in this experiment progress sheet.