This manual belongs to: ____________________

e-mail: ____________________

If found please return to owner

or leave in the electronics lab.

Contents

Organisation of the Laboratory

- These notes apply to the CS122 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.
- **You should work individually in this lab.** This does not preclude you from discussing your ideas with other people, but you will be expected to be able to be able to explain your design when it is demonstrated.
- 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 sessions 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.
- This laboratory is intended to ‘follow-on’ from the CS121 laboratory in the first semester. The CS121 lab. manual may therefore provide useful reference material for these experiments. Many of the designs produced in the earlier lab. may be of use here, so this may be the point at which you discover that good documentation is not simply something imposed to annoy.
- All the CS122 practical work is performed using the Powerview tools and the Xilinx gate array; these should be familiar from CS121, as should certain basic design techniques such as logic minimisation, state machines etc.
- The emphasis in the CS122 laboratory is shifted from basic technique towards actual logic *design*. For this reason the problems are not described in detail, but given as a *specification* to be fulfilled. It is up to the designer (i.e. you) to decide how to solve the problem and produce working hardware which fulfils the specification. In some areas the problems are (either deliberately or accidentally) vague, in which case sensible and workable decisions must be made to fill in any ‘holes’.

- *Experiment scripts must be read before coming to the laboratory session for that experiment.* Besides familiarising you with the work to be done, most experiments require preparatory work 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.

- As before the manual includes some extra sections which are intended to help with design technique and fill in further information about the design process.

- **This laboratory is assessed.** It is therefore important to ensure that your work is marked.

- 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.
  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 blue report card should be filled in and stapled to it.

- **Remember that experiments should be completed and handed in at the end of the session(s) scheduled for that exercise.** This should not be difficult if some preparation work is done.

- Please report any script errors or faulty equipment.

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

Register Transfer Level (RTL)

Up to this point practicals have been almost entirely concerned with gate level design. Different blocks, such as counters, multiplexers and a seven segment decoder have been designed but these have been put together in a largely preprescribed manner.

The CS122 experiments are different in that they require design at a higher level. This level is often known as **Register Transfer Level** or **RTL** as it is primarily concerned with the transfer of data between blocks of about the complexity of a register. A number of the most commonly encountered blocks are described below; some of these should be familiar, others may be new.

When designing at RTL the internal details of a block can be ignored; it is only necessary to know the *function* of a block (what it does) and its *interface* in terms of its inputs (what goes in) and outputs (what comes out). This is very similar to a procedure in a programming language where the calling process is unaware of the details of the function but simply invokes it as appropriate.

When designing circuits of the complexity of the following experiment it is usually easier to use a 'top-down' design style. In this case the highest level reached need only be RTL as the finished design is still reasonably simple. When designing larger circuits more levels of hierarchy may be introduced above this in order to keep the complexity manageable.

Buses

RTL blocks are generally connected together by **buses**[1]. A bus is a group of signals - usually with a similar function - which have been collected together for convenience. Thus your seven segment decoder has two buses, a 4-bit input bus and a 7-bit output bus. Buses are merely a convenience which save drawing all the individual connections and thus keep the design representation simpler and easier to understand. In addition to the buses there are usually (though not always) some *control* signals such as clocks and enables which must be wired individually.

Buses may comprise simple signals flowing from an output to one or more inputs; such signals are called **unidirectional**. In other cases they may be driven from more than one set of outputs (see **tristate buffers** below), in which case they are known as **bidirectional** buses as data can 'flow' either way along the wire. The data bus in a computer is a good example of a bidirectional bus (fig. 1.1); at one time it may be used for reading memory (data is output from the

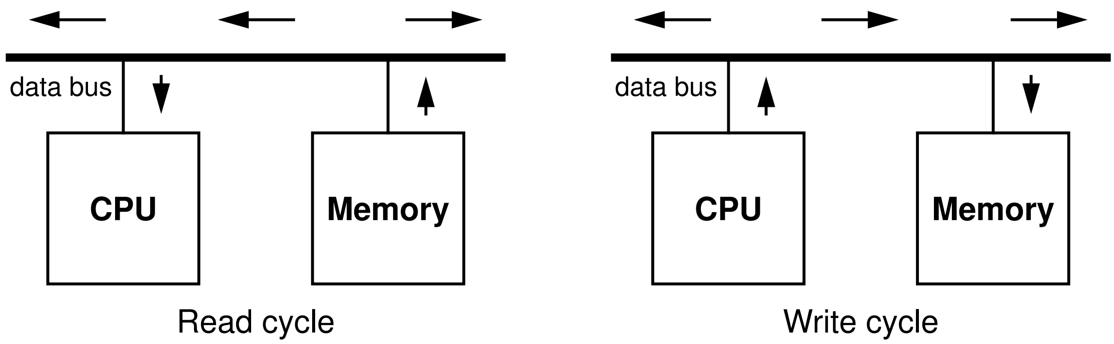


Figure 1.1: Example use of a bidirectional bus.

1. ... or "busses" if you prefer ...

memory and input to the processor) and then ‘turned around’ so that the processor **drives** the bus for a write operation (the memory outputs having been ‘turned off’).

The implementation of buses in Powerview is described in the following section.

Some Basic RTL Blocks

Some of the basic RTL blocks are described below. Note that - in practice - there may be variation between the implementations of these blocks (for example some registers will have a *reset* signal whereas others may not) but in each case there is a common and useful underlying function.

Registers

Registers are probably the simplest blocks and give their name to the class. A register is a storage device and is a collection simpler storage elements. There are two common types of register: the edge-triggered or D-type register which is made up of D-type flip-flops: and the level-sensitive or transparent register which comprises transparent latches. There are many ways of referring to these and it is only through familiarity and experience that you will realise which is appropriate for a given application.

In all cases a register is a collection of one or more of its basic storage elements. A register can contain any number of bits, so that - for example - a 32-bit register will contain thirty two storage elements. Only the data inputs and outputs of the register are independent; any other functions such as the clock, reset, enable etc. are *common* to all the storage elements. Thus an 8-bit register will have eight inputs and eight corresponding outputs but only a single clock (or enable) pin.

Counters and FSMs

Counters have already been encountered. A counter is a register with some extra logic to allow it to change its value as controlled by a set of input signals. Options on counters include up, down, up/down counting, clearing, enable/disabling and presetting (the ability to load an externally supplied value into the register). All counters will have a clock input which specifies *when* it should count.

A counter is an example of a **Finite State Machine** (FSM) whose generic structure is shown in figure 1.2. Finite state machines are commonly used for sequencing and control logic.

Multiplexers

Multiplexers or data selectors have already been met. They provide a means of supplying a set of inputs with data from one of a set of sources. Because multiplexers are typically controlled with a binary value they are usually met in the categories 2-to-1, 4-to-1, 8-to-1 etc. although this need not be the case.

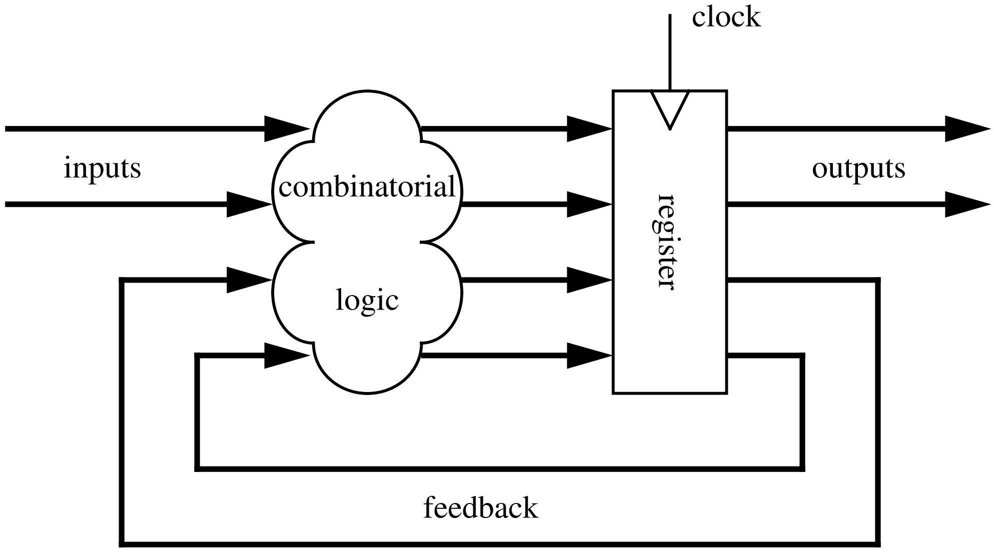
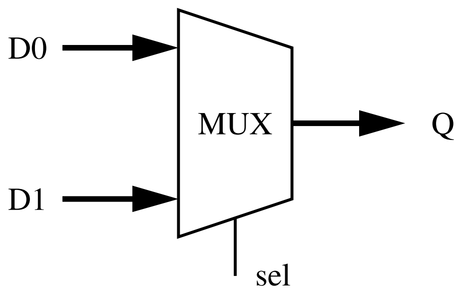


Figure 1.2: A typical Finite State Machine (FSM) structure

“Multiplexer” is a long word and tends to clutter diagrams; it is often shortened to ‘MUX’. Multiplexers are usually drawn as shown in figure 1.3:



D1	D0	sel	Q
X	0	0	0
X	1	0	1
0	X	1	0
1	X	1	1

Figure 1.3: Symbol and Truth Table for a 2:1 Multiplexer

Often several signals are multiplexed in parallel, thus the generic multiplexer may be used to select from sets of signals (i.e. buses) rather than single wires. Such devices are referred to as - for example - an “octal 2:1 multiplexer” meaning there are actually *eight* individual 2:1 multiplexers which are all switched together. The multiplexing function is sometimes provided by means other than gates such as ‘tri-stating’ all but one of a set of outputs (see below).

Buffers

Buffers are circuits which do not significantly affect the data but are needed to reduce the *fanout* of a given bus. They are primarily used to reduce *loading* and so speed up signals. In this laboratory the use of buffers is largely ignored although the fanout of some signals may be very great. For example the input signals to the seven segment decoder may go to ten or twenty gates inside the block as well as to other RTL blocks; it is usual to buffer the signals into such blocks (internally) to avoid this large ‘hidden’ load.

A buffer may be inverting or non-inverting (an inverting buffer changes the *sense* of the data) and may be tri-statable allowing the data to be disabled. The term ‘buffer’ is used to refer to both an individual signal line and to an instance of many such buffers (fig. 1.4).

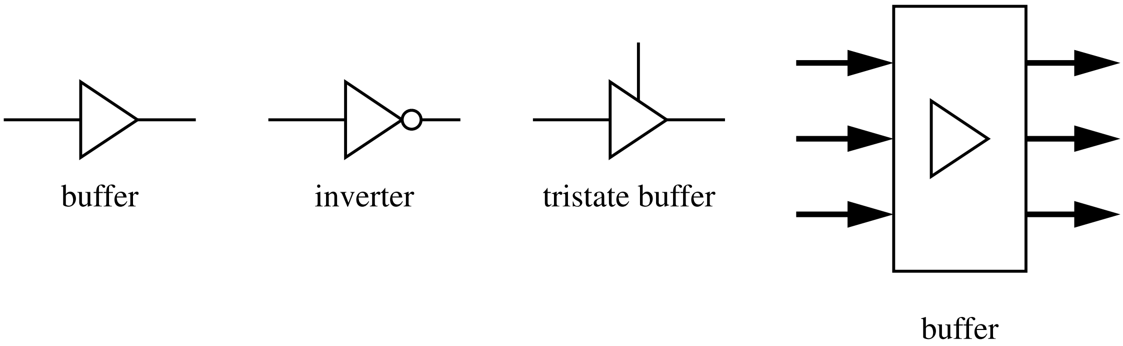


Figure 1.4: A Selection of Buffers

Tristate[1] buffers are special digital circuits whose outputs may be set into three discrete states. As well as being able to produce “0” and “1” (like a normal output) they can go into a third state, usually referred to as “Z” or “high impedance”. This is equivalent to turning the output off, effectively disconnecting the gate from the network. The gate is then said to be **disabled**.

Unlike most outputs tristate outputs may be connected together. When doing this however it is important to ensure that only one output is ever enabled at any time. This can be used, for example, as another method of creating a multiplexer circuit. Signals treated in such a fashion are commonly used for data and address signals; however *extreme care* must be taken if they are ever used in a control circuit due to the risks involved in switching between the different signal sources when the wires may be briefly disconnected from *all* outputs and therefore subject to glitching.

Adders

An adder is a block which takes two input buses and, treating them as binary numbers, adds them and places the result on an output bus.

ALUs

An ALU is an Arithmetic/Logic Unit (fig. 1.5). It performs functions such as addition (an adder is a subset of an ALU), subtraction, comparison, bitwise operations (AND, OR, exclusive-OR) and so forth. In addition to its buses it requires a number of control bits - which form its *function code* - to determine what operation to perform at a given time. There is at least one ALU in every computer processor.

1. Note: there are a limited number of tristate (10) *signals* allowed on the XM3030 device used in the lab.

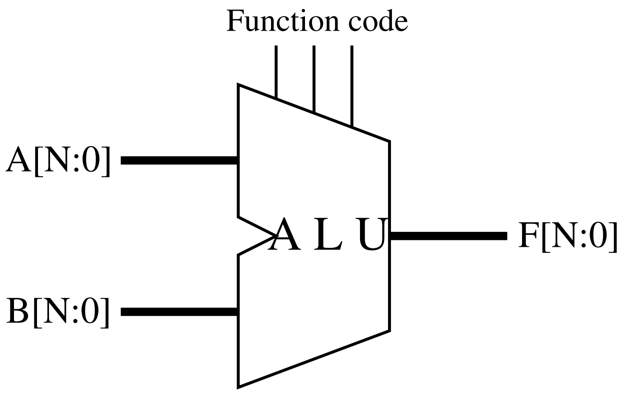


Figure 1.5: Typical Arithmetic/Logic Unit (ALU)

ROMs

Read Only Memory (ROM) is probably familiar as part of a computer's store and as such it is often used for holding fixed programmes. However (usually much smaller) ROMs are also a useful component in logic design.

A ROM translates an input pattern (on the *address bus*) into an output pattern (on the *data bus*). ROMs are useful for *translation* functions when arbitrary sequences of numbers are required. For added flexibility RAMs may also be used, although these are considerably more complex and must also be initialised before use as they lose their contents when the power is switched off.

PLAs

A PLA is a Programmable Logic Array. These are used like ROMs but provide greater flexibility. The detailed use of PLAs is beyond the scope of this course.

Specialist blocks

Sometimes blocks other than those mentioned here are required for specialist functions. For example the seven segment decoder is clearly a block in RTL but has a very specialist function. These may also be included in a design without regard to their contents. It is quite permissible to sketch out a block - knowing its function - without knowing what its internal composition is and this often provides a useful level of abstraction when creating a design.

In Powerview it is possible to define a *symbol* without defining its *schematic*. This may then be used to draft an RTL design. After the higher level design is thought to be correct the function of each block can be added at the schematic level. This top-down approach may prove useful in the final experiment and is encouraged, although individual design style is a matter of personal taste.

More on Powerview

The following sections comprise a random set of independent notes intended to extend some of the (hopefully) familiar features of Powerview. None of these is necessary to complete the laboratory, but learning about at least some of them may help save time later.

Projects

Powerview allows the definition of several “projects”. Each project is a self contained entity which has its own set of default values, can refer to its own set of libraries etc. Project manipulation is performed using the Powerview Cockpit.

A new project should be created for CS122 to avoid the modification/deletion of schematics by former partners from CS121; this is done automatically by the set-up script provided.

Schematics, Symbols and Sheets

Most, if not all, of the designs done so far have been ‘bottom-up’; i.e. a schematic has been created from known components and then represented by a symbol. It is also possible to design ‘top-down’ in Powerview, by creating a symbol first.

When a ViewDraw window is opened a dialogue box appears offering a set of existing schematics. Below these buttons is a line labelled “Enter name:” for starting new schematics, and below this are two buttons “Schematics” and “Symbols”. “Schematics” is the default option, but if “Symbols” is pressed instead the symbol editor is entered immediately; this in turn allows a new ‘empty’ symbol to be created. This symbol may be used on other schematics although (of course) it must be complete to the gate level before simulation etc. can take place.

In large designs it is not uncommon to use a model of the schematic written in a Hardware Description Language (HDL) to allow simulation before the circuit is built in detail. There is neither the necessity nor the time to do this in this laboratory however.

You have probably observed that your designs typically acquire a “.1” suffix; this is the sheet number. “Sheets” form a ‘horizontal’ layer in the design hierarchy, so that if a design is too crowded to fit neatly onto a single sheet, extra sheets can be used which are logically part of the same diagram. It is unlikely that you will need more than one sheet per schematic, but if necessary these are available via the Level ➪ Push ➪ Sheet command.

Transforming Symbols

Symbols which have already been placed on a schematic may be transformed by first selecting them and then invoking the relevant command from the Edit ➪ Xform ➪ menu. Edit ➪ Xform ➪ Reflect allows the symbol to be ‘mirror-imaged’ in a (temporary) line subsequently drawn with the middle button. Edit ➪ Xform ➪ Rotate allows the symbol to be rotated 90° anticlockwise using the middle button; the mouse position specifies the centre of rotation. The use of Edit ➪ Xform ➪ Scale should be obvious[1].

1. Use this with caution to avoid routing nets ‘off-grid’.q

Moving things between Schematics

Sometimes it is desirable to be able to cut a section from an old schematic and paste it into a new one. This can be done using entries in the Edit ➪ Buffer ➪ menu which allows cutting and copying[1] to and pasting from a buffer which can transfer elements between designs. Elements which are cut/copied are those selected when the command is executed.

Deleting Old Designs

Sometimes you may find that you have old, superseded designs in your project directory, which you would like to get rid of. This can be done from Powerview, but it is not one of the software's best functions.

To do this select Project ➪ Delete ➪ Schematic or Project ➪ Delete ➪ Symbol as appropriate (you will probably need to do both) from the Powerview Cockpit. This gives access to a menu which allows design names to be selected. Within a design the particular sheets must then be selected and can then be removed.

Remember to use this function with care.

Signal Names

Every symbol and every net on a schematic has a unique name. This is normally assigned automatically when the symbol/net is created. This name can be seen in the bottom left corner of the viewdraw window when a signal is *select*ed. When a net (or symbol) is *labelled* the automatic name is replaced with the assigned label.

All nets which have the same label are electrically connected. This means that (for example) the label 'clock' could be given to several net fragments on a schematic to connect them together rather than having 'wires' trailing across the picture. *Be careful not to give signals the same name if they should not be connected together.*

If it is necessary to identify a net on the schematic by name it is possible using the command Select ➪ Name... and entering the desired label name. This is useful when trying to find accidental short circuits which may be detected during compilation (e.g. "multiple source pins on signal ...").

Re-importing Altered Designs into ViewSim

There is no need to quit from ViewSim if a design has failed its simulation. The design can be edited in ViewDraw (and <u>saved</u>) run through VSM and then imported into ViewSim using the Net... command.

1. like 'cut' but does not delete the original

Buses ...

... in ViewDraw

Buses are merely a convenient method of representing bundles of wires. They confer two advantages:

1 they allow the value of several signals to be grouped together; thus instead of writing:

D7 = 1, D6 = 0, D5 = 1, D4 = 1, D3 = 0, D2 = 1, D1 = 1, D0 = 0

we can write:

D[7:0] = $B6_{16}$

2 Bundling signals with a similar function together helps to keep diagrams neat and easy to understand.

In *ViewDraw* buses are placed with the Add ⇨ Bus command. They appear as thick red lines. A bus must be labelled with the signals it contains.

Note:- *unlike normal nets, which can be left unlabelled, all buses must be labelled with a unique name, which also specifies the bus width.*

To separate individual signals from a bus, a *breakout* is required. In some CAD packages this requires a special symbol, but in *ViewDraw* all that is needed is to connect the net to the bus, and label it with the appropriate signal name (fig. 2.1). A bus is like an array in a programme,

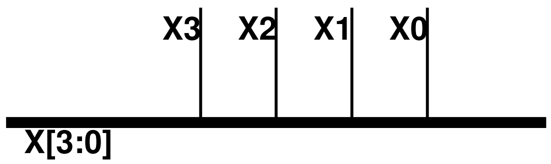


Figure 2.1: Example of bus 'breakouts'

and the labelling scheme reflects this, in that it is typically expressed as a name followed by a range of values as expressed elsewhere in this section[1]. Powerview allows ranges to be specified from high to low (e.g. "D[7:0]") or low to high ("D[0:7]").

If a range of values is attached to a single net as a label the names will be *expanded* and names will be supplied individually (without prompting) until the range has expired. For example if a bus is created it may be labelled "A[3:0]"; it will then retain this label signifying that it comprises four signals {A3,A2,A1,A0}. If, however, an individual wire has the same label specified it will adopt the first specified value only (i.e. "A3"). Subsequently the labels "A2", "A1" and "A0" may be placed on other wires automatically by first *select*ing the relevant wire and then pressing the middle mouse button. This confers no special properties to the net, but it can save considerably on typing, especially with large buses.

Note that although buses are typically used as an 'array' of signals, Powerview will allow any collection of signals to be grouped into a bus. Thus a bus could be labelled "A,B,C,D"; it would then comprise the four individual signals. This can be useful (for example) when tidying

1. Other CAD packages may use different syntax, but the principle remains the same.

up control signals into a single control bus. Connections are always made in the same order; thus if (for example) a pin of a symbol is labelled “A,B,C,D” and is connected to a bus labelled “D[3:0]” then “A” will correspond to “D3”, “B” to “D2” and so forth.

The concept of the bus can be extended to pins on symbols. Thus an entire bus can be connected to a single pin (appropriately labelled) on a symbol. This allows much simpler diagrams at the RTL, where the many individual signals are not shown at all on higher level schematics.

Once a bus has been created its value is displayed in a similar way to that of any other node. However for compactness and legibility the value on a bus is normally represented in hexadecimal (hex) rather than as binary; thus 5A\H is used instead of 01011010. Because the bits are interpreted in the same order as they are expressed in the label it is therefore sensible to label a bus D[7:0] rather than D[0:7]; the functionality is unchanged but the legibility is enhanced.

... in ViewSim

The value on a bus can be set by manipulating the signals individually, but it is often more convenient to treat the bus as a **vector**[1]. This allows the bus to be set up with a single command.

Unfortunately *ViewSim* does not comprehend that *A[7:0]* is a vector, and has to be told of this explicitly[2]. To do this use the command Setup ➪ Vector... which opens a dialogue box which prompts for a vector name and the nodes that it is to represent. Thus the name “*A*” could be assigned to *A[7:0]* (or alternatively to *A7 A6 A5 A4 A3 A2 A1 A0*). If this were done then all eight values may be assigned with a single command Set ➪ Assign...

This command allows the vector (e.g. *A*) to be assigned to a specified value. The value may be in binary, hex, octal or decimal; the **radix** is controlled by the suffix {\b, \h, \o, \d} although the \b suffix may be omitted as binary is assumed by default.

... in ViewTrace

Normally if a bus is selected when *ViewTrace* is invoked, the relevant bus is *expanded* when it is displayed, i.e. all the individual signals are extracted. This is caused because the name passed to ViewTrace is in the form *BUS[N:0]*. This is visible in the dialogue box when the Trace ➪ In Wave Window command is invoked. However if *vector names* are substituted on the line specifying the “Nodes/Vectors” to be traced the displayed buses will be condensed into a single trace. This should appear as shown in figure 2.2, where the number indicates the numerical value of the bus. The normal default radix is hex; this may be altered using Change ➪ Signal ➪ Radix... in the ViewTrace window

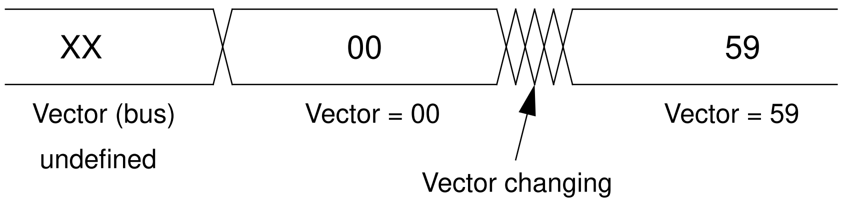


***Figure 2.2: Example of vector (or bus) waveform on* ViewTrace**

1. “Vector” is a much abused word in computing; in this context it simply means a set of values (bits).
2. Yes, it is very sad.

Clocks and Running Simulations

Input design ...

The CS121 laboratory introduced the simulator and explained how it is possible to force individual values onto signals and how to generate clocks for stepping (for example) a finite state machine through a number of iterations. Whilst useful for debugging simple circuits - where often a single simulation run is adequate to find any faults - this can be tedious for more complex circuits. If a long sequence of input combinations[1] is needed, and the circuit's complexity means that there is a high probability of faults, the test may have to be rerun several times on amended designs before the result is correct.

As well as being boring, trying to duplicate an input sequence many times can be error prone. It is therefore better to be able to store the sequence as a programme and run it when required.

Powerview has the capability of taking its input from a 'command' file, as well as individual changes on the display. This allows a preprogrammed sequence to be 'run' on the design. Such sequences can be run to completion so that the output can be examined after many steps; alternatively the simulation can be stepped through so that potential problems can be spotted as they occur.

A command file can be generated as text, which is quick to do once the syntax is understood, or by using the *ViewTrace* waveform editor to draw the desired waveforms.

...using text

A sequence of normal *ViewSim* commands can be stored in a file that can be executed during a simulation session. Command files can be created with any text editor but must be stored in the project directory (~/powerview/CS122, in this case). All the command files must have the extension "*.cmd*".

All the legal *ViewSim* commands can be put in the command file. During the execution all the commands in the command file are interpreted by *ViewSim* and executed in the same order as they have been stored. If there are any errors in the command file warnings and error messages are displayed in the *ViewSim* window.

Some of the possible commands are described below:

Stepsize alters or displays the current "stepsize" value. The stepsize determines the time step when the **Simulate** command is executed.

`stepsize` – Display the current simulation time increment to 300ns (the default stepsize is 100ns).

`stepsize 300ns` – This command changes the simulation time increment to 300ns.

1. also known as "**test vectors**"

Display shows nodes in the circuit and the node values. The nodes are specified after the **Display** command using a space as a separator.

`display *` – this command displays all the nodes of the circuit.

Network reads a flattened wirelist (*.VSM* file) into *ViewSim*. Issuing the **Network** command during a *ViewSim* simulation session flushes the old simulation data base before reading the new network file. The current simulation time is rolled back to 0.0. This command is very useful if you have made some corrections in your circuit and you want to simulate a new version of your design (after the compilation with the use of *VSM* compiler). There is no need in quitting *ViewSim* and running it again for a new simulation session.

`network D_flip-flop.vsm` – This command loads the wirelist of the D_flip-flop design for the simulation.

Wave invokes the *ViewWave* to display the simulation results in the graphically.

`wave D_flip-flop.wfm d clk reset q nq`
– this command directs the simulation results of the D_flip-flop design (nodes *d*, *clk*, *reset*, *q*, *nq*) to stream *D_flip-flop.wfm* that serves as an interface to the *ViewWave*.

N.B. each time the file is run the "*.wfm*" file grows; it is a good idea to delete this file after use.

Vector establishes a shorthand notation for a node, or set of nodes. The following examples define vectors `a` and `in`:

```
vector a a[0:3]
vector in d clk
```

Radix specifies the output radix for a set of vectors. There are four options for the radix base: `hex, dec, bin, oct`. The default base is `hex`.

`radix dec a in` – this command changes the output radix for the *a* and *in* vectors to decimal.

The **L**, **H**, **X**, **A** and **R** commands are used to assign values to signals and vectors.
L sets a list of signals low
H sets a list of signals high
X sets a list of signals to unknown
A assigns a value to a (previously defined) vector
R "releases" a list of signals (i.e. they are "tristate" or in state "Z")

`h reset` – set the signal reset high
`l d clk` – set the signals *d* and *clk* to low.

Sim advances the simulation by a time specified by “stepsize”.

Watch places nodes or vectors on an internal list that is accessed by the **Display** and **Print** commands. Once a node or vector is placed on the watch list, its values are automatically displayed whenever the **Display** command is executed. The watch list is automatically printed after every **Simulate**, **Run** or **Cycle** command.

`watch in q nq` – this command places the vector *in* and the nodes *q* and *nq* in the watch list.

Pattern defines a sequence of input patterns for a node or vector. The patterns specified by the **Pattern** command are applied to the correspondent vectors/nodes of the circuit only with the issuing of the **Run** command. The **Run** command applies each pattern and simulates for stepsize time units, until the longest pattern sequence has been exhausted. Shorter pattern sequences are repeated until they match the size of the longest pattern sequence. The following example demonstrates setting patterns using the **Run** command to automatically simulate these patterns:

```
| This is the command file D_flip-flop.cmd for the
| simulation of the D_flip-flop design
network D_flip-flop.vsm
stepsize 200ns
vector in d clk
| Invoking the graphical interface
wave D_flip-flop.wfm reset d clk q nq
pattern reset 1 0
pattern in 00 10 11 00 01
watch in reset q nq
run
| the end
```

The simulation results can be seen in the *ViewWave* graphical window and in the *ViewSim* window as follows:

```
time= 200.0ns reset=1 in=00 q=0 nq=1
time= 400.0ns reset=0 in=10 q=0 nq=1
time= 600.0ns reset=0 in=11 q=1 nq=0
time= 800.0ns reset=0 in=00 q=1 nq=0
time= 1000.0ns reset=0 in=01 q=0 nq=1
Simulation stopped at 1000.0ns
```

Test patterns for the **Pattern** command can be read from a file stored in the project directory. In such a file each test vector must be placed on one line.

```
| Here is an example of the file D_flip-flop.tst
| which contains 5 test vectors
00
10
11
00
01
```

As a result, the command `pattern in 00 10 11 00 01` can be replaced by the following command: `pattern in < D_flip-flop.tst`.

The last example of the `D_flip-flop.cmd` file can be written in another way:

```
network D_flip-flop.vsm
stepsize 200ns
wave D_flip-flop.wfm reset d clk q nq
watch in q nq
h reset
sim
l reset d clk
sim
pattern d 1 0
clock clk 0 1
cycle
run
```

... using *ViewTrace*

To do this open a *ViewTrace* window, using either the icon in the cockpit or File ➪ Open from an existing *ViewTrace* window. This creates a dialogue box which offers various ‘flavours’ of waveform file, either for the current design (if there is one) or all the designs in the current project. There will probably be a few files here which are left over from previous simulation runs, but the first time this process is used to produce input it is likely that a new file must be created. Enter an identifiable name and ‘accept’ it. It is not necessary to add a file extension at this point although a “*.cmd*” extension may be included if desired. The file name need not be the same as the design name, although it may be convenient to do this. However it is quite likely that more than one input file may be needed to test a single design.

A new *ViewTrace* window should now appear, which will be blank[1]. New waveforms can be entered into this window by using the command Edit ➪ CreateSignal... This opens a small dialogue box which allows a signal name to be entered. This may represent a single signal, or it may be a bus. In any case the number of bits must be entered (the default is “1” for an individual net) and there is a choice of radix; normally buses would be displayed in binary or hex[2].

1. except for a little logo
2. Remember that to display BCD the display should be hexadecimal; 0110 1001 is “69” in both hexadecimal and BCD, but is “105” in decimal.

When entering signals remember that their names should correspond to the names in the schematic diagram.

When the first signal is created a default time of 100ns is allocated and that signal should appear as an ‘unknown’ for that time (note also that axes have been created and labelled). If the signal is a bus its value should be superimposed on the graphical trace. More signals can be created as desired at this point. Note that other *ViewTrace* functions such as Change ➪ Signal ➪ Name... and Change ➪ Signal ➪ Radix... are available in case of errors. Signals may be selected for this by ‘clicking’ the left button on them, as in other Powerview programmes. The selected signals’ labels are highlighted at the left hand side.

The value of a signal may also be edited. To do this the portion of the signal to be changed must first be selected. The default **resolution** on the time axis is 50ns, so a portion of signal will only be selected if the left button is pressed and the mouse ‘swept’ along a large proportion of the desired signal. When a section of signal is selected it may be set to a value using Edit ➪ SetValue...; the settings allow a “0”, “1”, “X” (unknown) or “Z” (tristate, disabled) or a value to be entered in the case of a bus. ‘Accept’ing the value should cause the display to change to the chosen value.

There are two limitations with the default set up: the time allowed is too short (only 100ns) and the resolution is too coarse (only two phases of 50ns each). These settings may be changed as described below.

To alter the length of time of the simulation select Edit ➪ TimeLimit... This allows the time the simulation ‘begin’s and ‘end’s to be edited. Initially it is sensible to begin at time zero. The default end time is ‘1e-7’seconds (or 100ns); this can be edited in the dialogue box to allow a longer simulation run.

If a finer resolution is required the relevant axis must first be selected, so ‘click’ the left button at the bottom of the window, somewhere near the label “Time (Seconds)”; the area along the bottom of the display should be selected and change colour (to blue). This is necessary because several axes of differing resolutions are possible in the same window. The function Change ➪ Axis ➪ GridSpace can then be used; again a dialogue box is produced with the value (rather inconveniently) in seconds. The initial value is “5e-8” or 50ns; 1e-8 (or 10ns) is probably more convenient, but this depends on both the design being simulated and the designer’s own preferences.

Note:- *time values may be entered more conveniently in the relevant units,*
so that typing “10ns” (or just “10n”) is equivalent to “1e-8”.
The postfix “u” is used for µs in the absence of a “µ” key.

The methods outlined above allow the entry of any values which may be desired. However a couple of extra functions deserve a brief mention as they can make the entry of large vectors less tedious. For example if a clock signal is being defined a single cycle of the clock may be produced, ‘select’ed and the function Edit ➪ Repeat... invoked. This will repeat the highlighted section a selectable number of times (not including the original copy). Similarly sections of waveforms may be edited by other commands such as Cut and Paste all available under the Edit ➪ menu.

When the waveform design is complete it must be saved as a command file for ViewSim. To do this select File ➪ Write... and select ‘Format: CMD’ in the resulting dialogue box.

When designing waveforms remember that all circuits have some **propagation delay**, and flip-flops (and hence registers) require **set-up** and **hold** times. Thus the inputs to a register should be changed well in advance of the clock signal which latches them. The definition of “well in advance” is open to interpretation, and *for the purposes of these simulations* all these times are set to 1ns, so that a simulation of a signal passing through four gates and arriving at a register should be allowed at least 5ns. It is better to err on the side of caution in such tests, so a clock frequency of 10MHz (100ns period) is probably about right, with the input signals changing shortly[1] after the active clock edge. In most logic devices (including the Xilinx gate arrays) it is possible to meet the hold times for a flip-flop even if its output is fed back to its input; indeed this has already been observed in the earlier counter experiment in CS121.

Running tests

When a command file has been generated it can be used as input in a simulation. To do this, first start *ViewSim* on the design in the normal way. The test file can then be imported using Misc ➪ Execute ➪ CmdFile... This prompts for the name of the file for execution and then (assuming it exists) loads and executes it. Alternatively (and rather more easily) the command file name may simply be typed (with or without its “*.cmd*” extension) in the *ViewSim* window.

Command files produced using *ViewTrace* have a different format and do not run automatically[2]. Execution of the file can be initiated by the command Sim ➪ Run which will cause the input to run to completion. Alternatively Sim ➪ Sim or Sim ➪ Cycle can be used to advance the simulation in smaller time increments. Sim ➪ Sim advances time by one ‘step’; this defaults to 100ns (as shown by Display ➪ Stepsize), but may be altered with the Setup ➪ Stepsize... command. Sim ➪ Cycle moves time forwards by one clock cycle, and is probably the most generally useful step.

Note that entries in the Step ➪ and Breakpoint ➪ menus are not very useful in these laboratories. Finally don’t forget that the simulation can be reset to its initial conditions with the command Sim ➪ Restart.

Viewsim VHDL Window

In the default configuration, when *ViewSim* is started, there are two ‘panels’ created in the *ViewSim* window. The upper one of these is concerned with simulation using VHDL and is not used in this laboratory. This can be removed to save space on the display if required.

VHDL is VHSIC[3] Hardware Description Language. A hardware description language is a means of specifying the behaviour of a component in a form like a programming language rather than as a schematic diagram.

When *ViewSim* is started a ‘dialogue box’ appears with several options; the VHDL Source Window can be turned off at this point. The chosen status is saved and becomes the future default setting.

1. 10ns, 50ns, something like that.
2. I don’t know why.
3. *VHSIC (Very High Speed Integrated Circuit) was an US DoD research project.)*

Pass Through-Pins

When creating symbols for certain sub-circuits it can be an advantage to allow them to be "butted up" adjacent to each other. To facilitate this, Powerview allows more than one pin to have the same label, and hence be connected to the same network.

This is simply a drawing convenience, but it can make documentation much clearer. Powerview refers to such a pin as a "Pass-thru pin" [sic]. They are used - for example - when constructing arrays of memory devices where most of the signals are common to all the devices, with only *chip_select* signals distinguishing between them (Fig. 2.3).

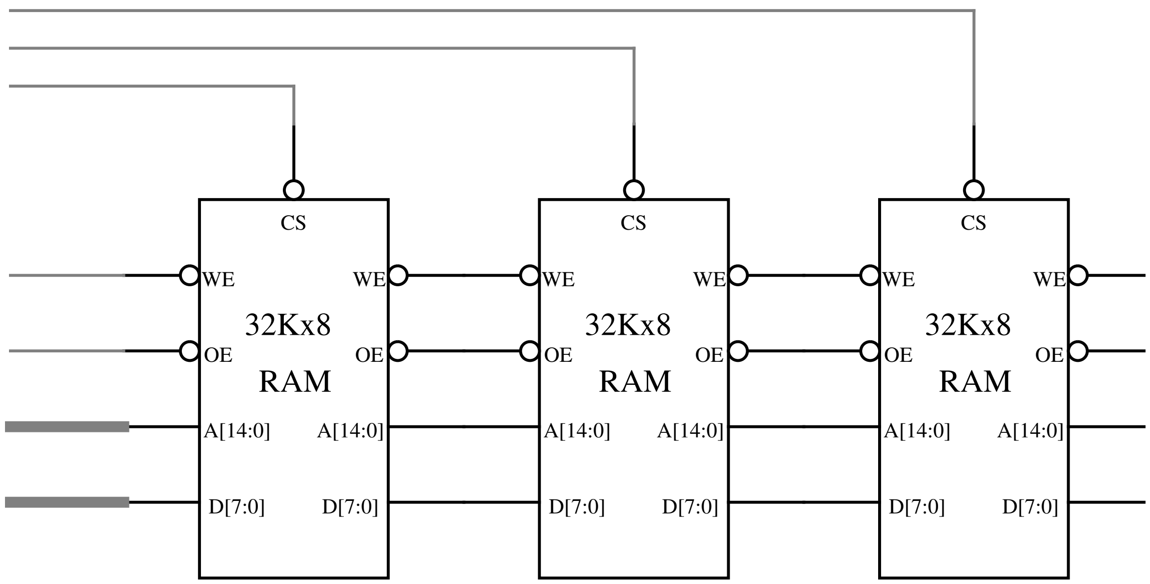


Figure 2.3: Example of Pass-through pins

Help

Remember that help is always available by using the manual via ■ ➪ Help ➪ ViewDoc and sometimes in the individual utility programmes (e.g. Misc ➪ Help... in *ViewSim*).

Experiment 122 - 1: Hi-Lo - a simple computer game

Aim

To use some of the skills and sub-circuits acquired in preceding experiments to design and implement a ‘real’, complex design. Unlike the other experiments in this laboratory ‘exact’ instructions are not given and there is a considerable emphasis on *design* rather than *implementation*.

Preparation

This is not an easy experiment. You *<u>must</u>* give some thought to the design before attending the laboratory.

This experiment should take about two laboratory sessions.

You will already have most of the necessary building blocks from preceding experiments in CS121. Try to decide what other functions you will need and how they might be implemented. The following sections are intended to stimulate ideas rather than explain exactly how to construct the circuit.

If you are short of time then implement a subset of the circuit; for example the scoring function can be omitted without affecting the rest of the circuit. Alternatively if you have both the time and the inclination you are welcome to extend and elaborate the design. If you do wish to change the scope of the experiment you are encouraged to discuss what is feasible with a demonstrator.

Specification

‘Hi-Lo’ is a very simple game in which the player must attempt to guess a secret number in as few tries as possible. After each guess the player is told whether the guess was correct or whether it was too high or too low. The player can then use this as a clue to making subsequent guesses. In this case the number should lie in the range 0 to 99 inclusive.

The guess should be entered using the number keys on the keypad. It will be necessary to type two numbers for each guess. It is an advantage to have a separate ‘guess’ key so that a mistyped number can be corrected. The guess should be displayed as it is typed. Each guess should be counted in order to keep score.

One button should be used to start a new game. This button must select a new secret number and clear the score; it may also initialise other functions.

To make demonstration easy the number buttons should be used at face value. The “*” button should be used to initialise the circuit and the “#” button used for entering guesses. Score may be kept on the bargraph and ‘hi’ and ‘lo’ conditions indicated by lighting up the left or right decimal points respectively (although other novel solutions are permissible). The indication of a correct guess is left to the discretion of the designer; one simple way might be to light *both* decimal points, a more complex solution could flash the display.

The Problem

The first exercise to undertake is to form an algorithm for the specification. The design can be broken down into a number of smaller problems.

Clock Generator

The clock generator is provided in the 'cs121lib' library. The reset input is active high and may be tied inactive (low). The unit uses *ripple counters* to divide the clock so that the outputs have a *phase difference* (i.e. they do not all change simultaneously).

Random Number Generator

There are many ways to generate 'random' numbers with a computer, almost all of which really generate *pseudo-random*[1] numbers (in this case this does not really matter).

The number to be generated here should be a BCD number between 0 and 99. Probably the simplest way to do this is to use the counter from experiment 121 - 4. If the counter is clocked sufficiently fast but only enabled whilst the 'initialise' button is held down it should be impossible to know how far the count has got.

Alternative random number generator designs are welcomed, however remember that the number must be in BCD.

Guess Entry

The entered guess must be stored somewhere. As each digit is a 4-bit BCD value it is suggested that a 4-bit latch may prove a useful library cell. Two of these (at least) will be required in the design.

Two methods of data entry can be considered. Either the number typed can always be sent to one of the digits (traditionally the least significant) thus forcing the number already there to scroll to the other or the typed numbers may be sent alternately to the two stored positions. Choose whichever of these solutions seems more appropriate.

When a button is pressed its value must be coded into a 4-bit BCD number. When - and only when - this value is valid a *strobe* output can be asserted to indicate this. There is a timing problem introduced by this since the bits within the BCD value will all change at fractionally different times.

To assist with the timing of these functions a timing block is included in the library. This should be fed by an OR of all the number input buttons (i.e. any button pressed) and a clock signal of your choice. Its output is a single high-going pulse, one clock period long and between two and three clock periods after the key is pressed (there is some uncertainty since the button press is not *synchronised* with the internal clock). This can be used as an enable on other D-type flip-flops (in the same manner as the 'ripple carry' introduced in the previous experiment) or as a clock pulse. The first of these designs would be **synchronous** since the same clock is used for the keyboard and latches. The other design is **asynchronous** because the clock on one block causes the clock on the next block. Nowadays synchronous designs are usually preferred but for this experiment choose a style that suits you.

1. i.e. an apparently random but repeatable sequence

Guess Display

Most of the guess display unit should be obtainable from the second part of experiment 121 - 4. Again a two digit BCD number is input which must be displayed on the twin 7-segment display. This should not present any difficulties.

Comparison

The two values (‘secret number’ and ‘guess’) must be compared to see if they are the same or - if not - which is greater. There are several ways of doing this.

The simplest method is probably to use an 8-bit comparator built hierarchically from smaller comparator cells. In a single bit comparison the values are equal if both the inputs are “0” or both the inputs are “1”. One is greater than the other only if it is “1” and the other is “0”. Note that it is only necessary to produce a ‘greater than’ and an ‘equal’ output because ‘less than’ can be deduced from these.

In a two bit comparator the result will be ‘equal’ if both the individual bits are ‘equal’. The result is ‘greater than’ if the most significant bit (MSB) is ‘greater than’ or the MSB is ‘equal’ and the least significant bit (LSB) is ‘greater than’. This can be extended indefinitely. Note that such a comparator is suitable for both BCD and binary representations.

Alternatively a more ‘direct’ combinatorial logic approach may be used. In any case it is suggested that, for simplicity, each 4-bit digit is dealt with separately and combined at the top level of the comparator schematic.

Scoring

The score should be indicated on the LED bargraph. This can be done by counting the number of guesses with a counter and decoding the state to a strip of lights. Alternatively a **shift register** may be used where each LED has its own flip-flop and each guess causes the illuminated state to move one place further. In either case the display must be cleared when a new random number is selected (the asynchronous clear on the flip-flop cell is convenient for this). Be careful that the clear operation switches the LEDs *off* rather than on as the bargraph outputs are active low.

Putting it together

Some thought must be given to how to connect the individual blocks in order to achieve the desired effect. It will probably be necessary to include some “**glue**” logic on the top level schematic; this comprises any remaining gates which are needed to interface the RTL blocks together. Usually this is just a few gates and in some cases may be nothing at all.

It is also important to plan a clear layout of the schematics in order to provide legible documentation. The top level schematic should show the configuration and interaction of the major functional blocks, interconnected by signals and buses as appropriate. This should not be too crowded with symbols; it is better to use more hierarchy than to cram too much into one ‘module’ (compare with *structured programming*).

Remember that as well as a ‘bottom-up’ design method a ‘top-down’ approach can be used; it is possible to design a symbol which (initially) has no schematic and use this as a component of the top level schematic. The ‘innards’ of the symbol can be filled in later.

Practical

1 **Implement the Hi-Lo game described in the specification above. You may deviate from the specification if you can improve the circuit, but the *function* of the game must be maintained. Try to keep your schematics neat by careful layout and appropriate use of hierarchy.**

Demonstrate the working design.

2 **Your design probably contains *hidden state*; the 'secret number' should certainly be hidden in normal operation. However this makes the design difficult to test. Suggest how you might modify the design to allow the 'secret number' to be examined for test purposes. You do not have to implement test functions (unless you want to).**

3 **When you play the game you employ your own algorithm to solve the puzzle. Describe (briefly - in words) how this algorithm operates. What is the *maximum* number of guesses that should be required? What is the significance of this number? (This is a bit of "information theory"!)**

Consider how you might implement the solving algorithm in hardware or software; you may even wish to build this on the Xilinx gate array[1].

1. Just for your own satisfaction, there are no bonus marks for this.

Synchronous and Asynchronous Design

A synchronous circuit has all its state changes controlled by a global clock. An asynchronous circuit does not, so that different events may trigger different operations.

Your Hi-Lo game implementation is probably asynchronous; this is possible because the subsystems do not interact in a critically dependent way. Possible conflicts in behaviour are avoided because of the way the game operates; thus the behaviour is undefined if guesses are being entered whilst the random number is still being chosen. In this circuit the human arbitrates the functions and ensures that possible conflicts do not arise.

For many large and complex circuits the dependencies between subcircuits are very difficult to model if they are operating asynchronously. For example it is difficult to compare two numbers unless they are both guaranteed to remain stable until the decision is made. If one number changes in the middle of the comparison the wrong decision may be made.

To prevent such things from occurring it is normal to make complex circuits synchronous. This means that all changes are governed by a fixed, global clock signal. Thus, in the preceding example, the numbers to be compared would be known to have settled into a valid state some time after a clock edge (depending on propagation delays etc.) and they will remain stable until the next active clock edge, which occurs after a known time.

Another advantage of synchronous circuits is that (usually) the propagation delays are such that a register may supply its own inputs (via some combinatorial logic). An example of this is seen in most pictures of FSMs (e.g. figure 1.2); this is also useful in processing elements, such as in the next experiment.

For these reasons the final experiment should also be seen as practice in purely synchronous design techniques.

A Design Example: A Stopwatch

This section is intended as a guide to the development process typical in many circuit designs. A simple design is specified and then carried through to one of many possible implementations. This design is similar in complexity to the subsequent laboratory exercise.

Specification

The design required is a stopwatch. This is a counter, capable of being started and stopped as desired. It must also be capable of displaying “split times”, i.e. freezing the display whilst still keeping time internally.

The stopwatch is controlled by two buttons, here labelled “A” and “B”. When pressed button “A” starts the timer if it is stopped, and stops it if it is running. Button “B” is used to hold the split time when the watch is running, but resets the timer if it is stopped.

Datapath

The first part of the design usually involves the so-called “datapath”. This comprises the major ‘highways’ of the design. It is essential to ensure that these will accommodate all the required operations. At this stage it is not vital to consider *how* the datapath is controlled.

The datapath can be designed at RTL. It will usually comprise most of the ‘gates’ of a finished design, *although it is **not** the most time consuming design phase*.

By examining the specification the following components may be identified:

- An input ‘clock’
- A display
- Two input buttons
- A counter

Note that the frequency of the clock and the size of the counter were missing from the specification; in this example they are not relevant, although in a real circuit this would be a serious omission.

A little further thought reveals another necessary component; if the stopwatch has to hold a steady ‘split time’ whilst the counter is running, a register is required for this purpose.

Having identified these components it is necessary to connect them to create the required functionality. Figure 3.1 attempts to illustrate this. Note that - in this case - the input buttons are purely for *control*, ad so are not required for the datapath design. This would not be true if a value was being entered.

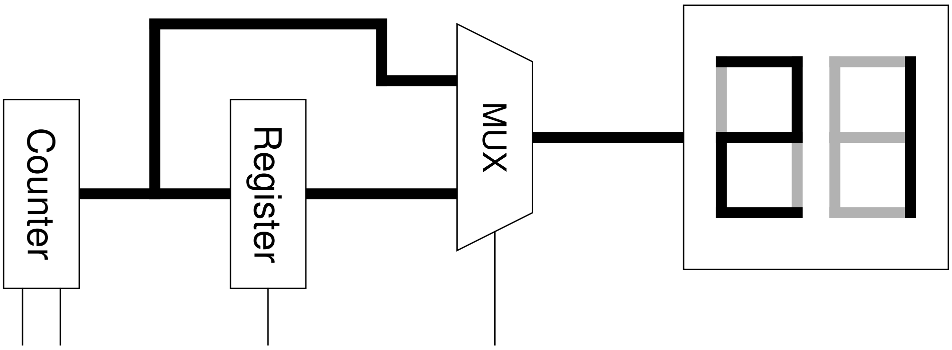


Figure 3.1: Example Stopwatch Datapath Design

This design allows the following operations:

- the counter's contents may be displayed
- the register may be loaded from the counter
- the contents of the register may be displayed

This is sufficient for all the specified operations. Note that another component - a multiplexer - has been added to the design to fulfil the specification.

Omitting signals such as the clock which is distributed to all the blocks which may require it, there are several control signals needed for the circuit to operate. These are:

- Counter enable
- Counter reset
- Register enable
- Multiplexer control

Ensuring that each of these signals is in the correct state at the correct time is the job of the control circuitry.

State Machine

The stopwatch can be in any one of a number of states at a given time. When devising the state diagram it is a good idea to specify as few states as possible to make the later implementation easier. In this system four states can be identified:

- Stopped, displaying counter
- Running, displaying counter
- Running, but displaying 'split time'
- Stopped, displaying 'split time'

Note that no mention has been made of the values in the datapath at this point.

These states can be numbered arbitrarily, although the choice of numbers may make implementation easier. In this case we will choose to make the 'stopped' states even numbers and the 'running' states odd numbers:

0 Stopped, displaying counter

1 Running, displaying counter

2 Stopped, displaying 'split time'

3 Running, but displaying 'split time'

The next job is to specify how the stopwatch controller moves from state to state; this may be done with a state diagram (figure 3.2).

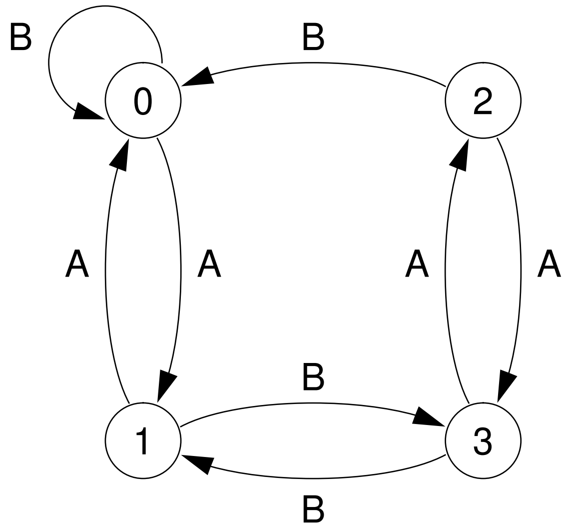


Figure 3.2: State Diagram for Stopwatch Controller

The following assumptions have been made:

- the values of the data are not part of the control state
- if no button is pressed the control state does not change
- each button press causes a single state transition

Notice that a 'sensible' assumption has been required (the effect of button "B" from state 3, which was missing from the specification) to complete this state diagram. Once this has been added the state diagram is complete in that the action of every type of input is defined for every state.

This state diagram can now be converted into a state transition table (fig. 3.3). Because there are exactly four states the state can be encoded in two bits (2^2=4)[1]. These bits have been

1. If five to eight states were needed three bits would be required, and so forth.

labelled S1 and S0. The state diagram specifies the next state, in this case as N1 and N0. A

S1	S0	A	B	N1	N0
0	0	0	0	0	0
0	0	1	0	0	1
0	0	0	1	0	0
0	1	0	0	0	1
0	1	1	0	0	0
0	1	0	1	1	1
1	0	0	0	1	0
1	0	1	0	1	1
1	0	0	1	0	0
1	1	0	0	1	1
1	1	1	0	1	0
1	1	0	1	0	1

Figure 3.3: State Transition Table for Stopwatch Controller

further assumption has been made that the buttons are mutually exclusive, i.e. they cannot both be pressed at once.

Note: this transition table includes the condition for remaining in the same state in the absence of an active input; alternatively this could be done using a 'clock enable'.

From the state transition table, logic equations can be defined:

$$N1 = \overline{S1} \,.\, S0 \,.\, B + S1 \,.\, \overline{B}$$

$$N0 = \overline{S0} \,.\, A + S0 \,.\, \overline{A}$$

These can then be implemented as a finite state machine (FSM) as shown in figure 3.4. Note that, as only a single state transition can result from a single button press, the input signals A and B must only be active for a single clock cycle.

This state machine 'remembers' the state of the control logic and performs the action appropriate for each button press at every point.

Up to this point no control signals have been derived for the datapath. However these are now relatively easy to produce. For example the multiplexer control signal should adopt one value when the counter is displayed (states 0 & 1) and the other value when the 'split time' is required (states 2 & 3). In this case only the state bit S1 (or - possibly - its inverse) is required. similarly the counter 'enable' signal can be derived solely from S0. These are simplified by the choice of state numbering determined earlier; if the states had different numbers the output derivation may be different, although the principle is the same.

The other two control signals - the counter clear and the register enable - are transient events

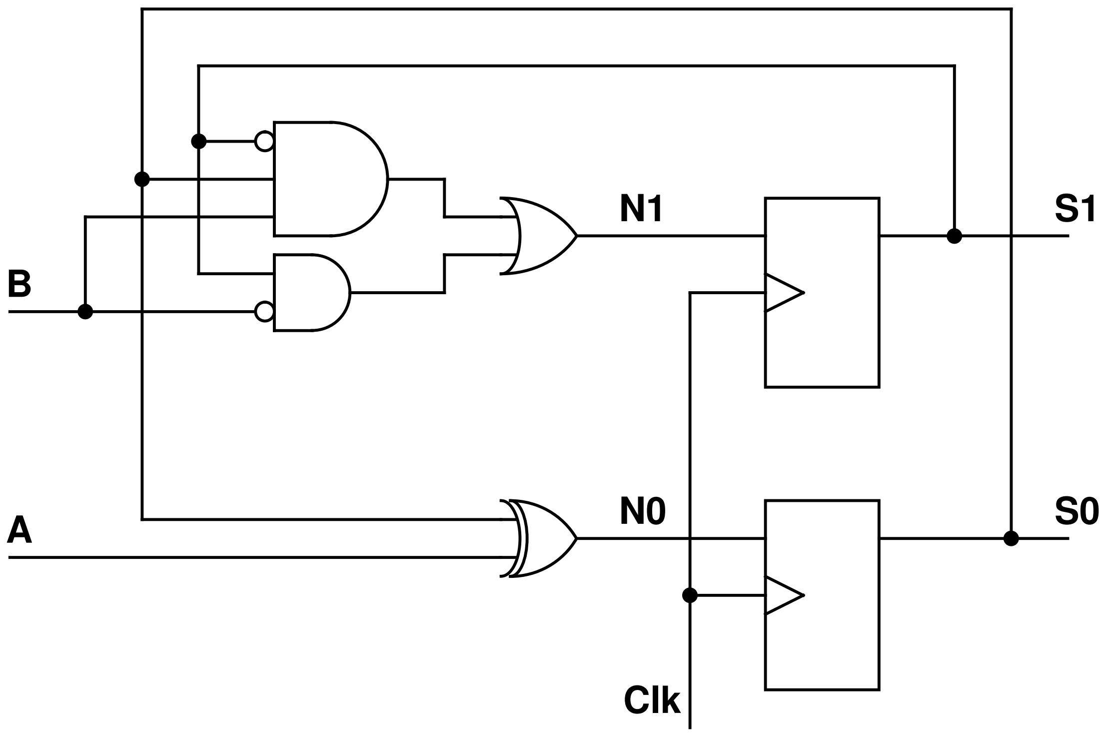


Figure 3.4: Finite State Machine (FSM) for Stopwatch Controller

and do not depend solely on the current state. To illustrate their derivation the transition table in figure 3.3 can be extended to include the various output signals (fig. 3.5). Here the counter

S1	S0	A	B	N1	N0	CR	CE	RE	M
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0	0	0
0	0	0	1	0	0	1	0	0	0
0	1	0	0	0	1	0	1	0	0
0	1	1	0	0	0	0	1	0	0
0	1	0	1	1	1	0	1	1	0
1	0	0	0	1	0	0	0	0	1
1	0	1	0	1	1	0	0	0	1
1	0	0	1	0	0	1	0	0	1
1	1	0	0	1	1	0	1	0	1
1	1	1	0	1	0	0	1	0	1
1	1	0	1	0	1	0	1	0	1

Where: CR = Counter Reset
CE = Counter enable
RE = Register Enable
M = Multiplexer control

Figure 3.5: Full State Machine Table for Stopwatch Controller

reset and register enable signals only appear when the button input signals are active. As these are (externally) guaranteed active for only a single clock cycle, so the output signals are only active for a single clock cycle. This gives the final set of output equations:

$$N1 = \overline{S1} . S0 . B + S1 . \overline{B}$$

$$N0 = \overline{S0} . A + S0 . \overline{A}$$

$$CR = \overline{S0} . B$$

$$CE = S0$$

$$RE = \overline{S1} . S0 . B$$

$$M = S1$$

from which the remaining control logic can be derived with a few gates. The control logic can then be appended to the datapath to produce the final design.

Experiment 122 - 2: A Calculator

Aim

To provide further practice with RTL design and to produce some significant synchronous state machines. There is also some further illustration of how computer arithmetic may be performed.

Preparation

Although this exercise may appear easy there are a number of design subtleties which should cause some consternation. You should complete the datapath design in about one lab. session and allow *at least* one session for debugging the control logic.

As in experiment 122 - 1 most of the subcircuits required should already be available. Again detailed instructions are not given, although some is guidance provided in the following sections. Make sure that you read the specification thoroughly and ensure that your implementation covers all the requirements. You are at liberty to extend the functionality if you wish to do so. If you are short of time then implement only the addition and subtraction functions and omit multiplication and division. Discuss and agree any other changes with one of the lab. demonstrators.

Choice is given as to whether the calculator should be decimal or hex. These each present slightly different problems, although hexadecimal will probably prove easier to implement overall. To avoid tedious logic design a hexadecimal 7-segment decoder is provided in the standard library although you are welcome to use your own design. If you need further advice you should consult a demonstrator before starting any detailed work.

Specification

The requirement is for a simple desk calculator. This may be either decimal or hexadecimal and should be capable of addition, subtraction, multiplication and division. Numbers should be entered on the keypad and the display should display the number entered or the running total as appropriate. The following keys should provide the following functions:

0 .. F	The appropriate hexadecimal digit
+,-	Addition and subtraction respectively
#	“=” button

The remaining key (“*”) is reserved to assist with other functions; its use is left to the designer. The separate global ‘reset’ button may be used as a clear function. This will reset all the flip-flops in the design to “0” (“L”); it is part of the chip architecture and does not require wiring on the schematic.

It is acceptable to neglect operator precedence, so that calculations are performed individually, ‘left to right’.

The machine should function in the same fashion as a typical pocket calculator. As an example, the following key sequence should produce the specified result:

Key(s)	Display	Notes
“Reset”	00	(or just “ 0”)
“2”, “3”	23	
“+”	23	
“5”	05	(or “ 5”)
“#”	28	
“-”	28	
“B”	0b	(or “ b”)
“+”	1d	
“4”, “1”	41	
“#”	5E	

Note that care must be taken to ensure that the sequence “#”, “+” in the middle of a calculation produces the same effect as just “+”. (There are a number of other similar cases.)

Finally, the whole design should be fully synchronous. This means that all the flip-flops that you place, all through the hierarchy, should have a common clock signal. This clock should run continuously; if a flip-flop is not to change on a particular clock edge[1] it should be disabled by ensuring that its clock enable (*CE*) signal is inactive (low). To reduce the *skew* on the clock distribution network a global clock **driver**[2] (*gclk*) is available in the *X3000* library. This symbol can only be used once per design, and its output can only be connected to flip-flop clock inputs.

Computer Arithmetic

Addition and Subtraction

Addition has already been covered in a previous lab. (CS121 - 2). When binary two’s complement notation is used subtraction is simple as it is the addition of the two’s complement of the number. i.e.:

$$A - B = A + (-B)$$

Remember that the two’s complement of a number is the binary NOT of that number, plus 1. Also don’t forget that the adder can have a carry input.

Multiplication

Multiplication is not an easily tractable problem for a computer (or calculator). However there are several methods of tackling the problem depending on the speed required and the **silicon area**[3] which can be dedicated to the problem. Four potential solutions for selection are presented below:

1. which will be a very common occurrence
2. see also later section on the Xilinx FPGA
3. i.e. the number of transistors or gates

1 **Combinatorial logic**. It is possible to treat multiplication as a combinatorial design problem. In this case the logic would require 16 inputs (two 8-bit numbers) and 8 outputs (the result need only be modulo 8 as that is all that can be displayed). This approach gives the fastest multiplication times but presents a not inconsiderable design effort and will use a large number of gates. For this reason it is not recommended here.

A possible solution to this problem is to use a ROM look-up table to give the result; this is very simple, but would require 64Kbytes of ROM and is not feasible in this laboratory. The size of such a ROM expands rapidly (exponentially) with the number of input bits.

Combinatorial logic multiplication is employed where speed is essential, such as in **Digital Signal Processors** (**DSP**s). It is normally too expensive for general purpose microprocessors.

2 **Repeated addition**. This is a very simple method for multiplication. For example, if the numbers 15 and 24 are to be multiplied, start with a 'running total' of zero and add 15 to it 24 times.

This will produce the correct result, and requires little more hardware than an adder and a counter. The disadvantage is that it can be extremely slow, especially as the numbers get larger (multiplying 32-bit numbers could take thousands of millions of cycles). This may be a suitable solution here however.

3 **Shift-and-add**. This is a straightforward approach to multiplication which uses no more hardware that the repeated addition method, but can complete in a number of cycles proportional to the number of bits in the number, rather than the magnitude of the quantity represented.

Consider performing the decimal multiplication below:

$$\begin{array}{ccccccl} & & 4 & 5 & 6 & & \text{(multiplicand)} \\ & \times & 1 & 2 & 3 & & \text{(multiplier)} \\ \hline & 1 & 3 & 6 & 8 & & \\ & 9 & 1 & 2 & & & \\ 4 & 5 & 6 & & & & \\ \hline 5 & 6 & 0 & 8 & 8 & & \text{(product)} \end{array}$$

Here one number (456) is a) multiplied by the least significant digit of the other (3), b) multiplied by the next most significant digit (2) and the result shifted left (i.e. multiplied by 10) and c) multiplied by the last digit (1) and the result shifted left by two places (i.e. multiplied by 100). Finally d) all the partial results are added to give the correct answer.

This complex operation should already be quite familiar! However if this is subdivided into stages much of the complexity can be lost. Remember that is makes no difference if the shift (x10) is before or after the multiplication. Note that as one number is shifted to the left the other (as individual digits) is shifted right, thus preserving the position of the digits in the result.

Keeping a running total in an **accumulator** means that only two numbers ever have to be added at any stage (figure 4.1).

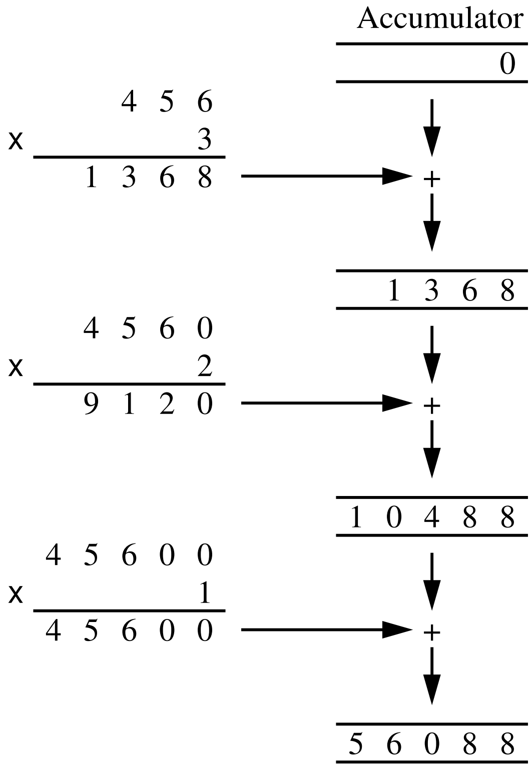


Figure 4.1: Example of decimal 'long multiplication'

The only remaining complex operation is now the multiplication itself. In decimal this can be mind-taxing; in binary multiplication by a single digit can only be by zero or by one, both of which are trivially easy! In fact multiplication is an AND function[1].

The binary implementation of this is exemplified in figure 4.1.

There are various ways of shifting different parts of the calculation to provide different variations of this algorithm, but the underlying principle is always the same. A shift register is easy to produce, being a register where each input bit is selected (by a multiplexer) as being either from a 'normal' input or from an adjacent bit (more or less significant, depending on the required direction of the shift.

In the example above, two 4-bit numbers are multiplied to give an 8-bit result. A 4-bit (modulo 16) result can be obtained by allowing the most significant bit to be

1. which is why an AND operation is sometimes referred to as a "product" and has the symbol "."

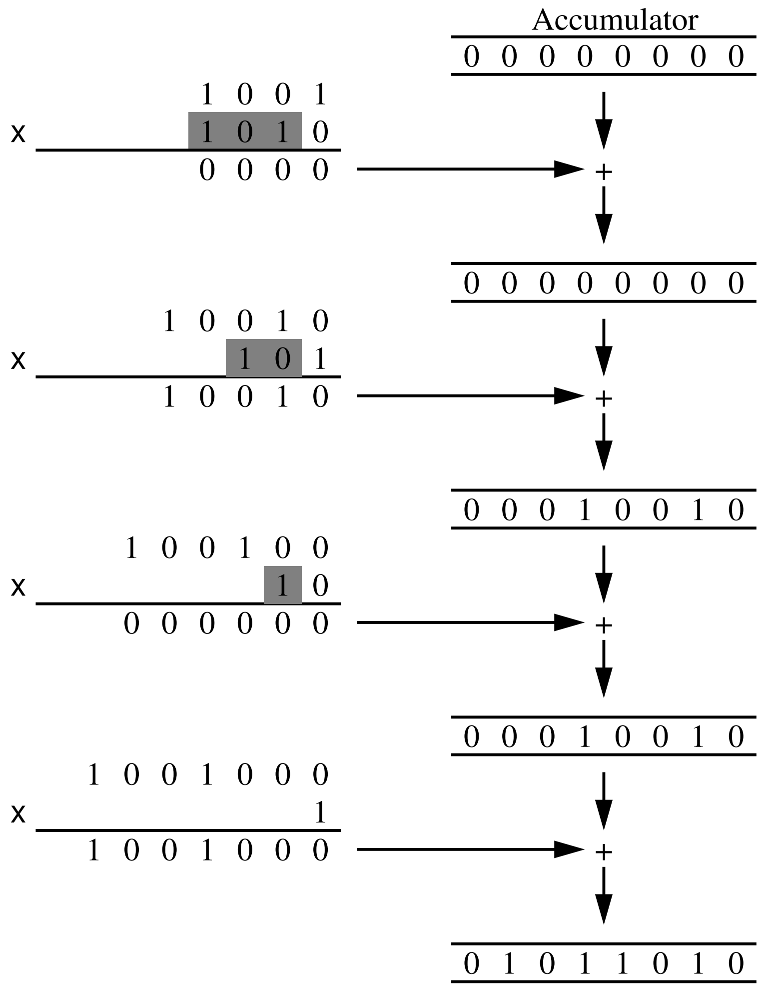


Figure 4.2: Example of binary 'long multiplication'

lost during a right-shift operation as this can no longer affect the lower bits of the result.

In a 4-bit operation the algorithm has 4 iterations. However this can be reduced in some circumstances (not in the preceding example) by terminating the operation when the multiplier becomes zero. This is because all further operations can only add zero to the accumulator. This is known as **early termination**. This is a useful technique because, even if it does not reduce the number of cycles performed, it guarantees termination without the need for an iteration counter, as the multiplier is bound to be zero eventually as "0"s are shifted in.

N.B. This is normally the algorithm of choice when implementing multiplication in software with a microprocessor.

4 **Booth's algorithm**. This is an optimization of a 'shift-and add' algorithm and is useful for dealing with signed numbers. As the numbers used in this experiment are unsigned, and Booth's algorithm is more complex than the previous example, it presents little advantage in this laboratory. A description is therefore not given here; interested parties should be able to find references.

Division

Division is - in principle - similar to multiplication; where multiplication is a repeated addition division is a repeated subtraction. It may be performed using (very complex) combinatorial logic, explicit repeated subtraction or an iterated 'shift-and-subtract'. The options are slightly more limited than for multiplication however, because the most significant bit of the result must be calculated first. Two possible methods are outlined (in brief) below.

1 **Repeated subtraction**. Set a counter to zero. Subtract the divisor from the dividend. If the result is positive, increment the counter and repeat the from the subtraction. The counter is the quotient when the loop terminates.

This will produce the correct result but, like the 'repeated addition' multiplication can be very slow.

3 **Shift-and-subtract**. This is best illustrated graphically (fig. 4.1):

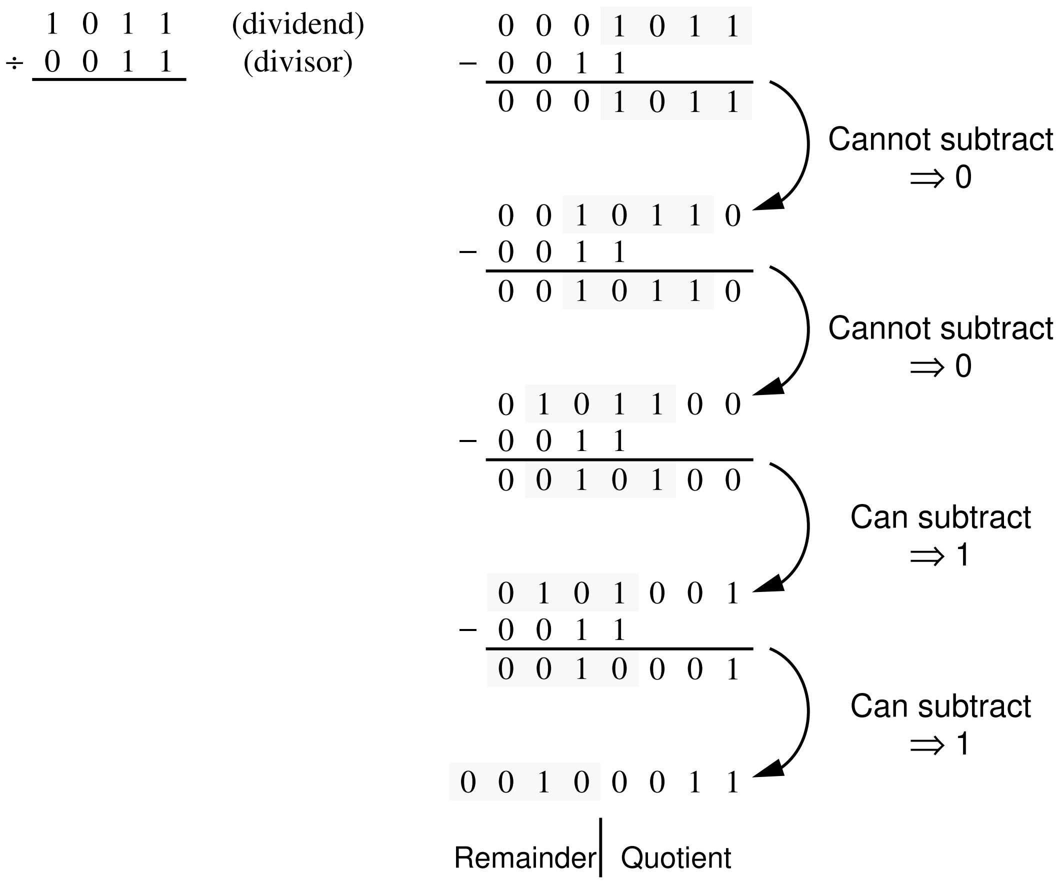


Figure 4.3: Example of binary 'long division'

The dividend and divisor are aligned as shown in figure 4.1. A subtraction is then attempted[1] and, if the result is positive (i.e. the subtraction "will go"), the result is retained; otherwise the subtraction is ignored. The dividend is then left-shifted and a bit - which indicates whether the subtraction was possible - is appended to the dividend. This process is repeated for a number of times equal to the number of bits in the dividend. At this point all that is left of the dividend is the remainder, whilst the quotient has been accumulated in the other bits.

This is analogous to a normal 'long division', where subtraction of the divisor is attempted from successive parts of the dividend, starting at the most significant end; in binary division there can only be two possible results for each digit rather than the ten available in decimal division. Using the same (shift) register for the dividend and the quotient is an optimization which, whilst unnecessary, represents a significant hardware saving.

Note that, unlike the multiplier, there can be no early termination and the iterations must be counted explicitly.

Hints

The following sections are intended to provide some hints on a possible solution to the problem. **You should attempt at least a paper design of your own before reading these sections**. Remember that if your design seems to be overcomplicated it can probably be simplified.

Overall

In a design of this complexity it is beneficial to decompose the problem in a 'top-down' fashion.

There is a great similarity between the architecture of a calculator and that of a computer processor. As such the design may be conveniently split into a datapath its associated control logic, and a number of interface circuits. The interface circuits will provide the I/O such as the display and keyboard interfaces and clock distribution (most of this should be available from earlier exercises).

Datapath

There is a great similarity between the architecture of a calculator and that of a computer processor. Thus the datapath will form a logical cycle where data from certain registers will be passed through an arithmetic unit and back to the registers again.

More specifically there will probably be an accumulator - which is used to hold the accumulated result of the calculations - and at least one temporary register to hold the other operand for the current calculation. The arithmetic unit will be primarily an adder, possibly with some means of forming the two's complement of one of the inputs to enable subtraction.

Note that the accumulator must be accessible in two 4-bit halves to allow it to be loaded one digit at a time from the keyboard.

1. may be treated as a comparison.

If multiplication is attempted it may be done purely in the arithmetic unit, but it is probably more efficient to use the existing adder and registers if an iterated cycle is to be performed. This will increase the control complexity of the circuit slightly, but it will reduce the datapath complexity. In general this is a better compromise because the control logic is a set of individual gates, whereas the data bits are repeated many (in this case eight) times.

The design should be synchronous so all the registers should be clocked by the same signal. If - as is likely - certain registers should not change at certain times they should be disabled using the clock enable signal.

The completed datapath should be a (fairly simple) network of RTL blocks connected by buses and with a number of individual control signals to the various multiplexers, register enables and so forth. For a calculator design including multiplication there should probably be less than a dozen control signals to generate; if this is omitted then there should be less than half a dozen.

When the datapath is believed to be correct, step through some test sequences on paper to ensure that all the required operations are possible. This will also help to identify the states of the control lines during the execution of each operation. Pay particular attention to the functioning of sequences such as:

“2” “+” “3” “#” “+” “4” “#”
and
“2” “+” “3” “+” “4” “#”

to ensure that they can both behave identically.

Control

Start with a state diagram. Work to minimise the number of states the logic cycles through and define the behaviour at all points. Follow the behaviour through by hand for various key sequences as a test. The rest of this section is designed as guidance if you get stuck and presents one possible solution.

Before considering the different states, the inputs can be simplified. Disregarding “reset” (and functions such as memories) for the moment, there are three ‘classes’ of keys which can be pressed: numbers, functions and ‘equals’ (“#”). Each of these has a different effect in each state.

There are a number of control states which the calculator can be in; more than there may at first seem. Disregarding any states which may pertain whilst a multiplication or division is ‘in progress’ there are certainly four possible states:

0 a ‘reset’ state

1 a state in which numbers are being entered with no function pending

2 a ‘function pending’ state (e.g. following the entry of “+”)

3 a state in which numbers are being entered with a function still pending

During the transitions between some of these states certain actions should be performed. The two major actions are *execution* of the pending function (which itself must be encoded and stored somewhere until it is needed) and the *clearing* of (the upper digit of) the display when a new number is entered. Note that in the sequence "2", "+", "3", "#", the display should read "02" (or " 2") until the "3" is entered, at which point it should change to "03" (" 3"); *the display should not read "23" at any time*.

Construct a state diagram with these four states and note the transitions between them for each class of keystroke. Note also the places where operations occur when the state changes; you may choose to make these separate states in their own right, which subsequently pass control back to one of the above listed states unconditional of any input. For the purposes of this diagram it is possible to ignore the fact that - in most cases - despite the free-running clock, the state does not change because no button has been pressed.

Practical

Implement the calculator as specified above. You must complete at least the data entry, display, addition and subtraction functions and should be able to make an attempt on multiplication; division is a (fun) bonus due to time limitations. Remember that the design should be <u>fully synchronous</u>.

Remember that although the control logic for such a circuit is fairly small its operation is relatively complex. Allow *at least 50%* of the design time for control logic design (and simulation and debugging).

Remember: simulation speeds up the debugging process.

Keep your schematics neat and legible. All diagrams should be easy to follow; make use of the design hierarchy rather than overcrowding an individual schematic. Partitioning should be done in a logical manner. Every sheet should have at least a title, the designer's name and the last revision date clearly legible.

Demonstrate the working design. You will be expected to be able to explain how your calculator functions to a demonstrator.

Include a state diagram of the control circuit in your documentation.

Possible Extensions

If time permits consider what extensions may be implemented. For example one or more memories may prove useful, as might an 'automatic constant' facility. Try suppressing the 'leading zero' when small numbers are entered/displayed. How about negative numbers or full operator precedence? The bargraph may be used to display 'flags' to indicate conditions such as a range overflow. If you find yourself out of ideas, ask!

Reference

Clements, A. "The Principles of Computer Hardware 2nd edition"
Oxford University Press, 1991
Especially section 4.9

The Xilinx FPGA

The designs produced in this laboratory are implemented on a Logic Cell Array (LCA) made by Xilinx Inc. This is a proprietary form of a chip generically known as a Field Programmable Gate Array (**FPGA**) (Fig. 5.1). This consists of an array of programmable cells known as Configurable Logic Blocks (**CLB**s) which can be connected as appropriate to the design. Each CLB can perform the function of one or more gates.

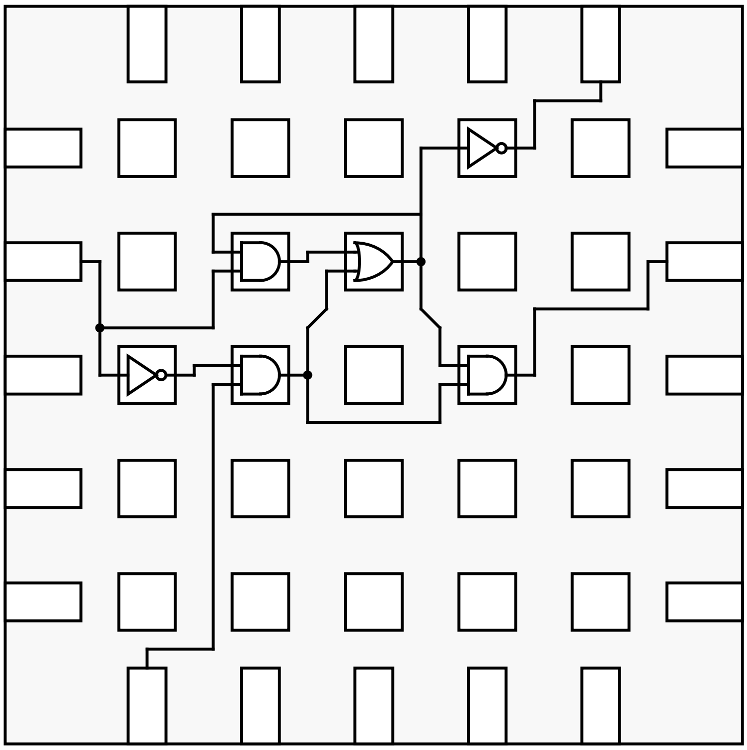


Figure 5.1: Hypothetical FPGA

The particular device used is an XC3030. This is a “3000” series device (as opposed to the earlier “2000” series or the new “4000” series chips) and is (supposedly) capable of containing up to about 3000 ‘equivalent gates’. It is a 10x10 array, and so contains 100 CLBs in addition to a number of I/O blocks which are used to control the inputs and outputs from the chip **core** to its **pins**. Other devices in the “3000” series range from the XC3020 (2000 ‘equivalent gates’ or 64 CLBs) to the XC3090 (9000 ‘equivalent gates’ or 320 CLBs). The XC3030 also has 80 Input/Output Blocks (IOBs) but the parts used in the laboratory are contained in an 84 pin package and so (after power supplies and control pins have been subtracted) only 66 of these have access to a physical pin. All the pin assignments are contained in various of the symbols in the appropriate library.

In the “3000” series devices each CLB comprises a combinatorial logic function generator and two D-type flip-flops. The logic block can produce two independent logic functions of up to four inputs each or a single function of five inputs (hence the largest gates available in the library have five inputs). The logic is derived using a look-up table stored in RAM (Random Access Memory) rather than using actual gates in order to increase its flexibility, thus *any* function is possible. The gates drawn on the schematic are reduced by the compiler (when possible) so that several gates may be fitted into a single CLB. For example it is possible to fit a seven segment decoder into four CLBs with room to spare.

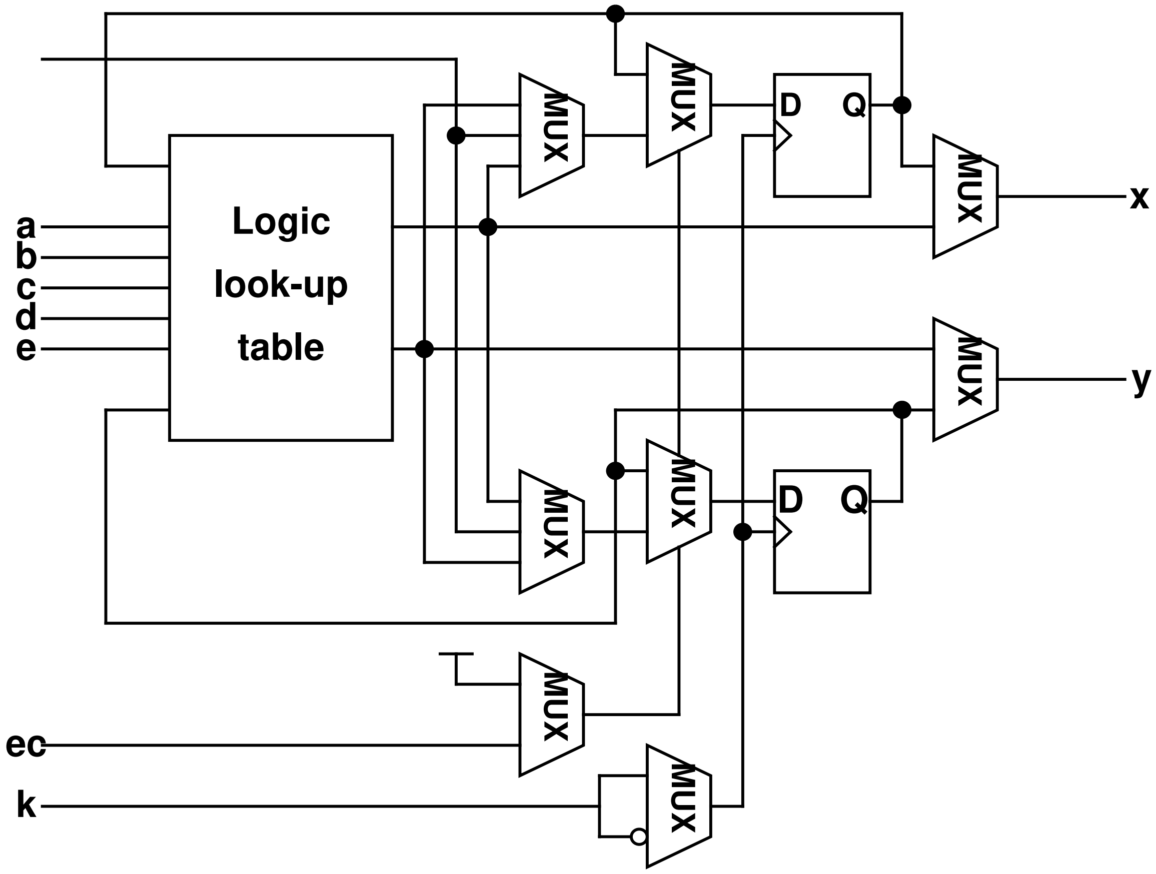


Figure 5.2: Xilinx CLB (reset circuitry omitted)

The interconnections between the CLBs are also programmable, although there are a limited number of connections available. The compiler will attempt to 'place' the CLBs for each function such that the amount of wiring necessary is reduced. *In extremis* it is possible to **place and route** the device by hand, but this can be extremely time consuming. On the Xilinx devices the connections may be 'local' (between adjacent CLBs), 'longlines' (distant on the same row or column) or via internal buses which may use tri-state signals. The routing decisions are normally made by the compiler.

The timing characteristics of the compiled circuit depend on the particular layout selected. Thus the timings which can be derived from *Viewsim* will not accurately reflect the characteristics of the final circuit. In practice the circuits will be much slower that simulation may indicate. This is a general problem with any 'chip' design, and an accurate timing model requires that the **layout** is **back-annotated** to allow implementation details to be taken into account. As speed is not a great issue in any of the designs in CS121/122 this has been neglected, but it can be very important in 'real' circuits.

As many 'real' designs are synchronous across the whole 'chip' it is important that, at least, the clock signal is distributed without **skew**, i.e. each clock pulse arrives simultaneously at all the

CLBs. For this purpose the Xilinx devices contain a specialised clock distribution network which may be used for such designs. This is not essential in these lab.s as the clock frequencies are quite low, but preferably should be employed in synchronous designs such as the calculator in experiment 122-2.

The Xilinx devices are RAM programmable; thus the chip's programme is held in RAM, just like a typical software programme. This means that the chip can be configured and reconfigured many times for use in different designs or to correct errors in a particular design. In a 'real' circuit the disadvantage of this is that the configuration is lost when the power is switched off. The normal method of loading such a device is not through a serial line from a computer but from a small, serial-access EPROM[1] or EEPROM[2]. Other manufacturers make similar programmable devices which use different programming methods. For example the Actel Corporation produce FPGAs which retain their programme permanently, however these devices are OTP (One Time Programmable) and therefore would not be suitable for this laboratory. Other devices use EPROM or EEPROM on board the gate array, but such devices require a special eraser/programmer in order to download their configurations.

1. Erasable Programmable Read Only Memory; erasure is by exposure to ultra-violet (UV) light and hence the package will contain a quartz window to expose the chip.
2. Electrically Erasable Programmable Read Only Memory (sometimes EEROM or "E^{2}ROM"; like EPROM but erasure is performed electrically and therefore there is no need for a window in the package.

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 = $\overline{A}$

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

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

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

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

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

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

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

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 - “**cs121lib**”.. 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.

Glossary of Terms

Accumulator: A register which is used repeatedly in an arithmetic function to 'accumulate' the result.

Active: A signal which is indicating a 'true' condition (see also *asserted*, etc.).

Active edge: The *transition* of a signal (such as a *clock*) which can cause a circuit to change its *state*. See also *inactive edge*.

Active high: A signal whereby a 'high' logic level indicates that the condition is *active*.

Active low: A signal whereby a 'low' logic level indicates that the condition is *active*.

ALU: Arithmetic/Logic Unit.

AND: see *Boolean logic*.

Asserted: A signal which is indicating a 'true' condition (see also *active*, etc.).

Asynchronous: Having no fixed relationship in time. Asynchronous signals may change in any order or even simultaneously.

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 yellow in Powerview.

Back-annotation: The extraction of physical data from a *layout* in order to increase the accuracy of a *simulation*.

BCD: see *Binary Coded Decimal*.

Bidirectional: A *net* (or, more usually, a *bus*) which has more than one output connected to it; data thus 'flow' both ways along at least some of the wires. e.g. a processor data bus. See also *unidirectional*.

Binary: base 2, and counting systems etc. derived therefrom.

Binary arithmetic: Addition, subtraction etc. of *binary arithmetic values*.

Binary arithmetic values: Numbers with each digital signal being a single base two digit with a value of "1" or "0".

Binary Coded Decimal (BCD): A representation of numbers where four bits are used to encode the values 0..9 only. This facilitates easy translation to decimal, but makes storage less efficient (values are 'wasted') and arithmetic functions (etc.) more complex.

Binary Counter: a *counter* which is modulo 2^N.

Bit: contraction of Binary digIT. A variable which can have one of two values. The state which may be indicated by a single digital signal.

Boolean algebra: The textual manipulation of *Boolean Logic*.

Boolean logic: The combination of the *Boolean logic values* using simple functions such as AND, OR and NOT (fig. 6.1).

In1	In2	Out
0	0	0
0	1	0
1	0	0
1	1	1

AND

In1	In2	Out
0	0	0
0	1	1
1	0	1
1	1	1

OR

In1	In2	Out
0	0	0
0	1	1
1	0	1
1	1	0

XOR

In	Out
0	1
1	0

NOT

Figure 6.1: Basic Boolean logic functions

Boolean logic values: Values with the two states equating to “TRUE” and “FALSE”; these are commonly abbreviated to “T” and “F” respectively.

Booth’s algorithm: A computer multiplication algorithm.

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

‘Bubble’: Small circle on component input or output indicating an inverted (or *active low*) signal.

Buffers: ‘Gates’ which perform no logic function but which increase the *drive* (and hence the *fanout*) of a signal. May also be used to provide *tristate* signals.

Buried register: A *register* in a *chip* whose outputs are not directly accessible from the ‘outside’.

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.

CAD: Computer Aided Design.

‘Chip’: Short for “silicon chip’; an *integrated circuit*.

Circuit diagram: See *schematic diagram*.

CLB: Configurable Logic Block; the basic component of a *Xilinx FPGA*.

Clock: A regular signal which is used as a time reference for other signal changes in *synchronous* circuits.

CMOS: Complementary Metal Oxide Semiconductor

Combinatorial logic: An implementation of *Boolean logic* where all the outputs depend solely on the current inputs. (see also *sequential logic*)

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

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

Core: 1) The part of an integrated circuit which contains the functional *gates* (see also pad ring).
2) The main memory of a computer (historical).

Counter: a simple finite state machine which follows a repetitive pattern, usually cycling (upwards or downwards) through a set of N states (hence a "modulo N" counter).

Cycle: A complete operational period of a *clock*.

D latch: sometimes used for *transparent latch*.

D-type flip-flop: An edge-triggered storage device which has a single data input. The output adopts this state in response to an *active edge* of a *clock* signal.

Data selector: sometimes used for *multiplexer*.

Decade counter: A modulo 10 *counter*.

De Morgan's theorem: NOT(X AND Y) ≡ NOT(X) OR NOT(Y)
NOT(X OR Y) ≡ NOT(X) AND NOT(Y)

Dependency notation: A notation occasionally used for *schematic diagrams*. Its main characteristic is that signal names are given for the *active state* and the symbols indicate the particular active state of the signal.

Digital Signal Processor (DSP): A specialised computer or processing system used for processing signals (e.g. telephone systems, CD players).

Disable: To stop a function from occurring. Examples include: stopping data entering a latch; preventing a flip-flop from changing; and stopping data from reaching a tristate bus.

Drive: 1) The act of outputting a signal onto a net.
2) The capacity of an output to '*fanout*' to input loads.

DSP: See *Digital Signal Processor*.

Duty Cycle: the proportion of time a signal spends in its active state. Usually used to refer to very regular signals such as *clocks*. (See also *mark-space ratio*).

Edge: The change from one logic level to another (see also *transition*).

EEPROM: Electrically Erasable Programmable Read Only Memory (sometimes EEROM or "E^{2}ROM"; like an *EPROM* but erasure is performed electrically and therefore there is no need for a window in the package.

Enable: To allow a function to occur. Examples include: allowing data into a latch; permitting a flip-flop to change; and allowing data onto a tristate bus.

EPLD: Electrically Programmable Logic Device

EPROM: Erasable Programmable Read Only Memory; erasure is by exposure to ultra-violet (UV) light and hence the package will contain a quartz window to expose the chip.

Equivalent gate: alternative for *exclusive-NOR gate*. (See *Boolean logic*.)

Exclusive-OR (XOR): see *Boolean logic*.

Exclusive-NOR (XNOR): see *Boolean logic*.

Falling edge: A *transition* from a "H" to a "L". See also *rising edge*.

Fanout: The number of inputs driven (or which can be driven) by an output.

Feedback: Any system where (at least part of) the input is derived from the current *state* of the system.

Finite State Machine (FSM): A type of *sequential* circuit which employs *feedback* in order to follow a predefined set of *state* transitions, governed by its current state and (usually) some external inputs (fig. 6.2).

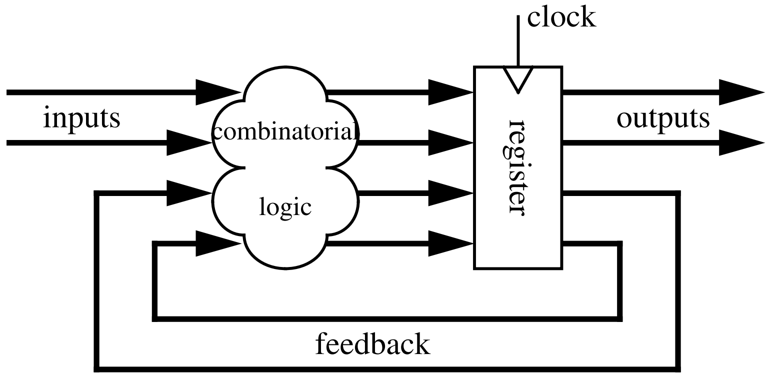


Figure 6.2: A typical Finite State Machine (FSM) structure

Flip-Flop: Any of a number of *bit* storage devices. Most commonly, but certainly not exclusively, a "D-type". See *D-type flip-flop*, *JK flip-flop*, *toggle flip-flop*, *RS flip-flop* and *transparent latch*.

FPGA: Field Programmable Gate Array. A large, user programmable circuit.

FSM: See *Finite State Machine*.

Gates: Simple circuits which provide basic *Boolean logic* functions.

Gate level: Usually the bottom level of a design hierarchy.

Glitch: A very short and unwanted pulse on a signal, typically caused by *race hazards* in networks of *gates*. Glitches are also known as “runt pulses”.

Glue: A slang term for the odd bits of logic used to hold *RTL* blocks together.

GND: Common abbreviation for *ground*. (see also *Vcc*, *Vdd*, *Vss*).

Ground: Zero volts. This is normally the ‘reference voltage’ from which other measurements are taken. In most (but not all) logic families this is a “0” or a logic low (“L”).

Hardware Description Language (HDL): A means of specifying the behaviour of a component in a form like a programming language rather than as a schematic diagram.

Hexadecimal (Hex): Base 16. Digits are represented as 0..9 then A..F. Hexadecimal is convenient as it is compact yet very easy to convert into *binary* (4 bits per digit).

HDL: See *Hardware Description Language*.

Hold time: The time data must remain stable after the *clock edge* for correct function of a *flip-flop*. See also *set-up time*.

IC: See *Integrated Circuit*.

Inactive: A signal which is indicating a ‘false’ condition.

Inactive edge: A *transition* of a signal (such as a *clock*) which cannot cause a circuit to change its *state*. See also *active edge*.

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

Integrated Circuit (IC): A ‘silicon *chip*’ containing from one to several million *gates*.

JK flip-flop: An edge-triggered storage device which has separate inputs allowing it to be set, cleared, toggled or not respond on a subsequent *clock* edge.

Karnaugh map: A graphical tool used for combinatorial logic minimisation.

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.

Latch: A *storage element*; often used as shorthand for '*transparent latch*' although sometimes used more generally.

Layout: The physical placement of *gates* on a *chip* or of chips on a *PCB*.

LCA: Logic Cell Array; A *Xilinx FPGA* design.

LED: see *Light Emitting Diode*.

Light Emitting Diode (LED): A form of diode which emits light when it is conducting electricity. LEDs are commonly used as indicator lights as they are simple, bright, low power devices. Available in various colours (typically red, amber, yellow and green, as well as infra-red) although blue LEDs are difficult to produce and therefore expensive.

Logic diagram: Sometimes used for *schematic diagram*.

Logic equations: A textual means of representing a combinatorial logic function.

Logic level: 1) A 'high' or a 'low' voltage.
2) The actual voltage values interpreted as 'high' or 'low'.

Mark-space ratio: The ratio of the time that a regular signal (such as a *clock*) is 'active' to the time that it is inactive. See also *duty cycle*.

Momentary Action: (usually referring to a switch) the action (output change) only occurs when there is some input stimulus (e.g. a push button).

Multiplexer: A circuit which allow one of a selection of signals or buses access to a single resource. Often abbreviated to "MUX".

Multiplexing: The act of using multiplexers.

MUX: Common shorthand for "*multiplexer*".

NAND: see *Boolean logic*.

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.

NOR: see *Boolean logic*.

NOT: see *Boolean logic*.

Not-equivalent gate: alternative for *exclusive-OR gate*. (See *Boolean logic*.)

OR: see *Boolean logic*.

OTP: One Time Programmable.

Pad ring: The periphery of a *chip* which connects the core to the *pins*.

Patchboard: A circuit board which allows the rapid prototyping of circuits by simply plugging together components and wires (also known as a 'breadboard').

PCB: Printed Circuit Board. An stiff, insulating (often fibreglass) board with a pattern on conducting metal tracks etched onto its surface. Used for supporting and connecting electronic components. Easy to mass-produce.

Phase: 1)The relationship of two signals.
2)One half of a clock *cycle*.

Pin: 1) A place on a *symbol* where *nets* can be connected.
Pins are cyan (light blue) in Powerview.
2) A metal contact on the outside of an integrated circuit.

PLA: Programmable Logic Array. A customisable form of a *sum-of-products* structure.

'Place and Route': The act of choosing a position on a chip (circuit board etc.) for *gates* is known as 'placement'. 'Routing' is the subsequent connection of these components. The two operations are related. This process is often automatic nowadays. See also *layout*.

PLD: See *Programmable Logic Device*.

Product of Sums: A logic structure comprising an *AND* (product) *gate* whose inputs are the outputs of a set of *OR* (sum) *gates*. Analogous to - though more rarely encountered than - a *sum of products*.

Programmable Logic Device (PLD): A *chip* which may be customised (programmed) by the end-user.

Propagation delay: The delay between the inputs to a *gate*/circuit changing and its output changing in response.

Race hazard: A situation where signals which should arrive in a predetermined order are delayed by *gates*/wiring so that they may arrive in the wrong sequence. To be avoided! See also *skew*.

Register: A circuit component comprising a number of *flip-flops*.

Register Transfer Level (RTL): A 'level' in the design hierarchy which corresponds to the plugging together of blocks of about the complexity of a *register*. Usually one level above '*gate level*'.

Ripple Counters: An asynchronous counter where one counter output changing may cause the next counter in the chain to toggle by clocking it directly.

Rising edge: A *transition* from a "L" to a "H". See also *falling edge*.

ROM: Read Only Memory.

RS flip-flop: (sometimes SR flip-flop) a simple storage device which is Reset (to “0”) or Set (to “1”) by *active* pulses on the relevant input signal.

RTL: See *Register Transfer Level*.

Schematic: Shorthand for *schematic diagram*.

Schematic capture: A means of entering circuit components and wiring into a *CAD* system simply by drawing them on the screen as a *schematic diagram*. Also known as *schematic entry.*

Schematic diagram: a diagram showing the *components* of a circuit and their interconnections. Also known as a ‘*circuit diagram*’ or just as a ‘*schematic*’.

Schematic entry: See *schematic capture*.

Sense: The sense of a signal determines whether it is *active high* or *active low.*

Sequential logic: A system constructed from *Boolean logic* elements where the outputs depend both on the current inputs and the past history of the system. The system therefore contains some *storage element*(*s*) which holds its current *state*. (See also *combinatorial logic*.)

Set-up time: The time data be remain stable before the *clock edge* for correct function of a *flip-flop*. See also *hold time*.

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.

Shift register: A register which - in addition to normal register function - can also move each of its bits to an adjacent bit in response to the correct stimuli.

Simulation: Testing a circuit (or other) system without recourse to building it.

Skew: The time difference between two signals; may be caused by differences in *propagation delays*. See also *race hazard*.

SR flip-flop: see *RS flip-flop*.

State: The state of a circuit is effectively the pattern of values held within its internal storage elements; thus a modulo-8 counter would have eight possible states (usually encoded as combinations the individual states of three flip-flops. This is only applicable to sequential circuits; combinatorial circuits have no inherent state.

State diagram: A graphical means of representing the states of a system and the possible transitions between them (fig. 6.3).

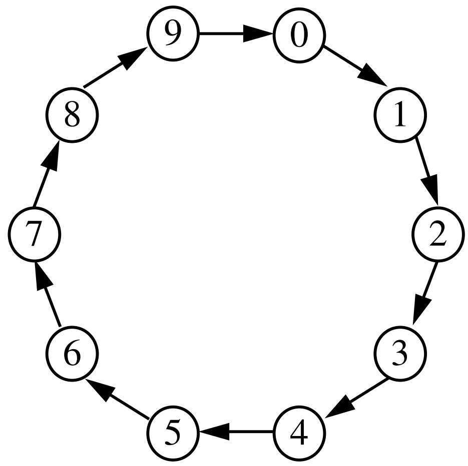


Figure 6.3: State Diagram for a Modulo-10 Up Counter

Storage element: Any circuit or component which contains *state* (e.g. a *latch*, *flip-flop* or memory).

Sum of Products: A logic structure comprising an *OR* (sum) *gate* whose inputs are the outputs of a set of *AND* (product) *gates*. A commonly encountered structure in *Programmable Logic Devices* (*PLD*s).

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.

Synchronous: All signal changes are referenced to a global clock signal.

T-type flip-flop: occasionally used for *toggle flip-flop*.

Time-Domain Multiplexing (TDM): The sharing of a resource by allocating ‘time-slices’ to different clients.

Toggle: to move backwards and forwards between two states. Also used as an abbreviation for something which has this behaviour (e.g. a toggle switch or a *toggle flip-flop*).

Toggle Flip-Flop: A *flip-flop* which moves into the ‘other’ *state* every time it receives a *clock* input. Typically used as a “divide by two”. (Occasionally referred to as a T-type flip-flop).

‘Top-down’ design: A design philosophy where the whole design is successively broken into a hierarchy of smaller elements until implementable blocks are obtained.

Transistor: An electronic switching/amplification component. Transistors are the fundamental components of almost logic *gates*.

Transition: 1) The change from one logic level to another (see also *edge*).
2) The change from one *state* to another.

Transition table: A graphical means of representing the behaviour of a sequential circuit; similar to a *truth table* but (some of) the outputs will form subsequent inputs.

Transparent latch: A level sensitive storage circuit which, when enabled passes its input(s) to its output(s) (i.e. is 'transparent').

Tristate output: A type of logic output which - in addition to the normal two states "H" and "L" ("0" & "1") - can adopt a third state, usually referred to as "Z" or "high impedance". This switches the output 'off' effectively disconnecting it from the circuit.

Truth table: A graphical means of representing a combinatorial logic function.

TTL: Transistor Transistor Logic. A bipolar logic family now becoming largely obsolete.

Unidirectional: A *net* (or, more usually, a *bus*) which has only one output connected to it; data can thus only 'flow' one way along the wires. e.g. a processor address bus. See also *bidirectional*.

Vcc: Typical label for a "1" or logic high ("H"). Derived from "collector voltage" in *TTL*; the name persists even though the derivation does not.
(See also *GND*, *Vdd*, *Vss*).

Vdd: Typical label for a "1" or logic high ("H"). Derived from "drain voltage" in *CMOS* logic. (See also *GND*, *Vcc*, *Vss*).

VHDL: *VHSIC* Hardware Description Language. A hardware description language is a means of specifying the behaviour of a component in a form like a programming language rather than as a schematic diagram.

VHSIC: Very High Speed Integrated Circuit; an US DoD research project.)

Vss: Typical label for a "0" or logic low ("L"). Derived from "source voltage" in *CMOS* logic. (See also *GND*, *Vcc*, *Vdd*).

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

Xilinx: A US corporation who design and manufacture *FPGA*s.

XNOR: exclusive-NOR; see *Boolean logic*.

XOR: exclusive-OR; see *Boolean logic*.

Experiment 122 - 1: Answer Sheet

Name:

Group:

Experiment date:

Part 1

Demonstration of design: pass/fail

Demonstrator's signature for part 1

Attach schematics of your circuit to this answer sheet before handing it in.

(To save paper sub-schematics for blocks submitted for earlier CS121 exercises - such as the seven segment decoder - may be omitted).

Part 2

How might the design be made more testable?

Part 3

What is your game solving algorithm?

What is the maximum number of guesses required?

Why?

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 122 - 2: Answer Sheet

Name:

Group:

Experiment date:

Demonstration of design: pass/fail

Demonstrator's signature

Attach schematics of your circuit to this answer sheet before handing it in.

(To save paper sub-schematics for blocks submitted for earlier exercises may be omitted)

Please include state diagram overleaf or on separate sheet.

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.