About these notes

These notes form part of the teaching materials for the first year course unit COMP12111: Fundamentals of Computer Engineering (formerly COMP10211).

This course introduces digital logic and its application in computer organisation and design. The major emphasis is on practical design work and in the laboratory state-of-the-art computer-aided design tools are used to support the design of digital hardware systems. In the lab you will produce designs, simulate them and then implement them using the electrically reconfigurable gate arrays found on the experimental boards in the laboratory.

The lectures initially support the laboratories but progress to a wider overview of the design and interaction of computer hardware systems. Ultimately a complete - if simple - computer is described as constructed from simple gates.

Organisation of these notes

These handouts contain copies of the lecture slides as well as "extra" information relevant to the material being taught. All material is examinable. Further information can be found on the course eLearning site.

eLearning Resources

Further information, such as copies of these workbooks, copies of the slides, extra notes, corrections to the notes, etc, can be found on blackboard through my.manchester.

The Blackboard site for the course also includes quizzes, worked examples etc. It also includes videos illustrating key concepts covered in this part of the course unit. These videos are in addition to the podcasting of lectures undertaken by the University, which are available through the University Blackboard page for this course unit.

Further help and advice can be found in the School of Computer Engineering Wiki

https://wiki.cs.manchester.ac.uk/engineering/

Assessment

This course unit is assessed by a formal examination and laboratory work (50:50 split). The examination lasts 1hr 30mins and has two parts – section A and section B. Both sections contain two questions, you must answer both questions 1 and 2 in Section A (which is compulsory). You must choose to answer one questions from two in section B.

This course has been running in its current form since the 2008-2009 academic year. Hence, when looking at past papers (all of which are available of the University intranet) take care when looking at papers prior to 2008. If the question looks unfamiliar, the chances are the subject matter is no longer covered. No sample answers are given for past papers. However, I will be happy to discuss any questions relating to the material I cover in lectures, providing some attempt to answer them has been made beforehand.

If you are stuck …

I'm happy to answer any questions relating to the module (well, my course material certainly). If you want to speak to me either catch me at the end of lectures, in the lab, or come and see me in my office. If you are planning on visiting me in my office then, if possible, please email me to check my availability, that way I'll make sure I set aside enough time to answer your questions. If you knock on my door out of the blue, then I may not have the time to see you!

Further information, such as copies of these workbooks, copies of the slides, extra notes, corrections to the notes, etc, can be found on the course unit Blackboard site.

Finally, if you notice any mistakes, then please let me know … I'm not perfect and my notes definitely aren't … ☺.

References

I acknowledge the following references which I have used to prepare these notes:

1. COMP10211 Lecture Notes (pre-2010) by Dr Jim Garside, School of Computer Science, The University of Manchester.
2. "Principles of Computer Hardware", A Clements, 4th Edition, Oxford University Press, ISBN 978-0-19-927313-3
3. "Digital Design with RTL Design, VHDL, and Verilog", F Valid, 2nd Edition, Wiley, ISBN 978-0-470-53108-2
4. "The Verilog hardware description language", Thomas and Moorby, 5th Edition, Springer, ISBN 978-0-387-84930-0.
5. "Digital Design", M M Mano, 3rd Edition, Prentice Hall, ISBN 0-13-035525-9
6. "Fundamentals of Logic Design", C H Roth Jr., 4th Edition, PWS Publishing, ISBN 053495472-3.

MANCHESTER 1824
The University of Manchester

COMP12111: Fundamentals of Computer Engineering

Part 2
CAD & Verilog

Notes:

Lecture Aims

The aims of these lectures are:

- to discuss how to handle complexity in design
- to discuss the role of CAD tools and how we will use them in the lab.
- to introduce Hardware Description Languages (HDLs)
- to discuss the syntax of the Verilog HDL – module structure, declarations, assignments etc.
- to investigate how we implement FSM designs in Verilog

Quiz-time

Q. How much does it cost to design and produce a modern processor?

A. Around $1,000,000
B. $1,000,000 - $500,000,000
C. $500,000,000 - $1,000,000,000
D. $1,000,000,000+

Notes:

Quiz-time

Q. Today's processors contains billions of transistors. CAD tools have aided this development because:

A. allows us to manage the design hierarchy
B. by providing tools for testing
C. by automating stages in the design process
D. all of the above … and more

Notes:

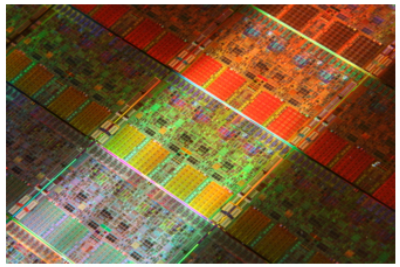
MANCHESTER 1824

The University of Manchester

Complexity and Design

In the early days of digital design, the designer invariably had to design digital systems by drawing schematics on paper.

… this approach is relatively straight-forward for very simple designs containing a limited number of transistors (1000s)



However, this approach is completely impractical when it comes to designing more complex systems, such as a microprocessor, that may have many millions of gates!

How do we design such complex systems?

… the answer, we use computer aided design (CAD) tools

Notes:

Complexity and Design

Early microprocessors were designed and physically laid-out by hand (literally). With a couple of thousand transistors (the basic building blocks of gates) this is possible, if tedious and error prone. With designs (again, literally) a million times this size this is not a realistic proposition, even if the added problems introduced by lithography of features smaller than the wavelength of light were not present.

A Brief History of 'Computer' Chips

- In the beginning the first microprocessor (Intel 4004) had a 4-bit datapath processor; nothing else would fit on the 2300 transistor chip.
- Soon afterwards the datapath was expended to a more useful 8 bits. Typically, the address space was 16-bits (64Kbytes).
- Memory chip sizes grew to make populating all 64 Kbytes affordable.
- Datapaths expanded to 16-bitl address spaces of 20-, 24- and then 32-bits appeared.
- RISC processors led expansion to 32-bit datapaths. 'Pipelining' becomes common to increase processor speed.
- Processor speeds began exceeding memory speeds leading to demand for on-chip cache.
- Lots of architectural expansion such as on-chip floating point units.
- More levels of cache hierarchy.
- 'Systems-on-chip' – one piece of silicon integrates most/all a user's needs.
- Multicore processors.
- …

That's not a complete story, but it gives you a clue. For example, "microcontrollers" – systems-on-chip containing a processor, some memory and a limited set of peripherals have been around since 1975. However, until recently, compromises had to be made which limited the processor architecture (and therefore performance) to leave space for the other functions.

To give some idea of size, an ARM microprocessor occupies:

- about 0.075mm^2 in a 65nm technology (≈ 13 processors per mm^2)
- about 0.036mm^2 in a 45nm technology (≈ 28 processors per mm^2)

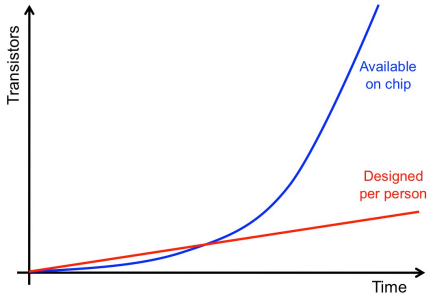
Designer Productivity

'Moore's Law states that the states that the number of transistors on a chip will double about every two years; this is an exponential growth in the chip capacity.

Even with improved productivity due to better tools there is a serious 'design gap' developing between what can be designed and what can be built.

Chips are being filled with:

- More complex functions
  - limited by design shortage
- Higher levels of abstraction – whole systems on chip
  - but many systems are now on single chips
- Replication of simple elements
  - more memory/cache
  - multicore processors



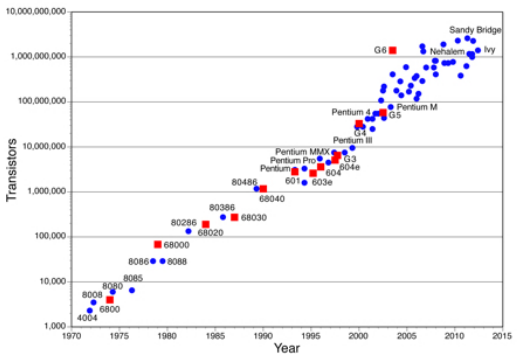
MANCHESTER 1824

The University of Manchester

Why do we need CAD tools?

Modern complex digital systems can have billions of transistors, the basic building blocks of digital logic gates.

- Moore's Law exhibits exponential growth in the number of transistors
- Designer productivity grows very little
- tools needed to increase productivity



Notes:

Computer Aided Design (CAD) Tools

Most effort in designing any complex engineered system goes into the managing the actual complexity of the design.

Computers are good at:

- Doing many, repetitive, tasks
  - Each individually simple
  - Nested layers of different designs
- Rapid processing
- Doing exactly what is defined for them
- Not making any error, and **never** getting bored.

Therefore, CAD tools are highly suitable as an aid in designing and testing complex logic designs (such as modern processors!).

Computers are tools – give them boring, mechanistic tasks because that's what they're good at. Here are some applications not all of which you will use.

- Design Entry
  - replicate a design many times without errors
  - manage a hierarchy and maintain files, giving easy, uncluttered access to relevant figures (largely hidden from you in Cadence)
- Simulation
  - many interconnected gates, each simple but on the whole complex
  - apply numerous rules to calculate timing, eyc.
- Translation/Compilation
  - follow algorithms to alter a design from one form – say something straightforward for a human – to another.
  - link tools together without errors
- Layout
  - apply the numerous rules and constraints needed to achieve a physical realisation
  - 'extract' the results of the realisation to improve simulation estimates
- Test
  - estimate 'test coverage' – percentage of signals that have been changed and observed
  - generate test input patterns
- Analysis
  - find (e.g.) critical paths which limit a circuit's speed
  - feedback aid to designer

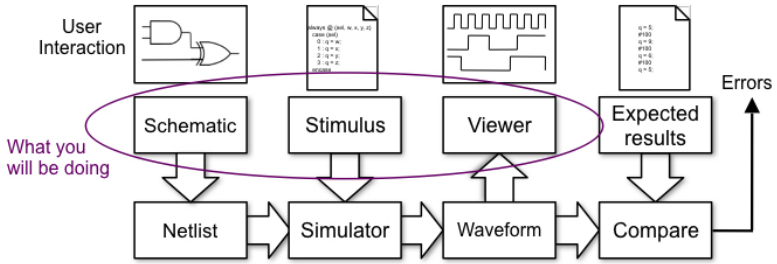
MANCHESTER 1824

The University of Manchester

Design Tools

It would not be possible to manage modern IC designs without CAD tools.

… Cadence is a 'design framework' that contains numerous CAD tools.



- The tool flow has (largely) been automated

but

- The user must provide input and feedback
  - spot problems/errors and adjust the input accordingly

Notes:

In the lab …

In our laboratories we stick to using a few important, high-level tools:

- Schematic capture and symbol editor (Virtuoso)
- Simulator (NCSim)
- Viewer (SimVision)
- Logic Synthesizer (XST)
- Place and Route (Various tools)

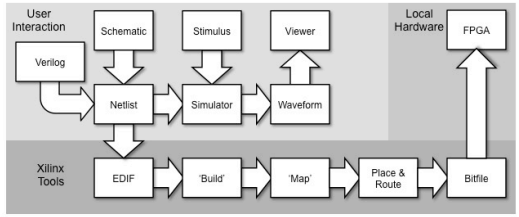
These are sufficient to design, debug and 'compile' quite complex designs, providing we do not want to meet challenging timing parameters etc. These can be implemented in programmable chips called FPGAs (Field Programmable Gate Arrays) and we will do that in later labs.

Some of the things we *don't* do …

The designs could also be implemented directly in silicon. However, in a modern chip there are many, many details to look after which we won't look at here. You could spend an entire 3-year degree programme covering topics relevant to the design of a complete microprocessor. Examples include: silicon layout, design rule checking, timing extraction, test coverage, and thermal design.

Laboratory design flow

The use of tools in the lab is shown in the figure opposite:



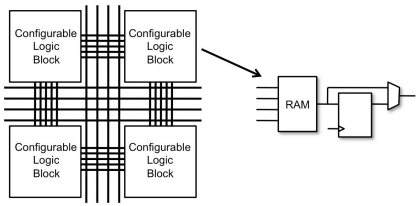
The compilation chain works as follows:

- EDIF (Electronic Design Interchange Format) is used as a common description to move the design between different toolsets.
- The 'Build' process breaks down the input description into a set of primitive components suitable for the target device.
- The mapping process groups primitives together into logic blocks; at this point resources are still regarded as unlimited.
- Place and route (PAR) takes each logic block and finds a suitable place for it on the target device. This is influenced by how the blocks need to be interconnected. It then wires them together as required. Place and route may iterate, trying several placements to see which turns out 'best'.
- Once the design has been mapped onto a physical device a 'bitfile' is generated which is suitable for communicating the programming information to the FPGA. (The FPGAs in the lab contains ~200,000 gates and require 131Kbytes of bitfile.)

FPGAs

Field Programmable Gate Arrays are commodity silicon chips which are configurable 'in the field' (i.e. in the lab!) to implement a user's design. Consequently, a working 'custom' chip can be produced in a few minutes rather than the couple of months it takes to make a 'real' custom chip. The advantages of this approach should be obvious. The disadvantages from custom chips are the slower operation, lower capacity (fewer gates/chip) and higher power. None of these will be much of a problem in our labs!

Modern FPGAs are large enough to implement whole systems. FPGAs are sometimes used as prototyping systems before building a real ASIC (Application Specific Integrated Circuit) as they can 'simulate' a design's operation much faster than a software simulator and therefore allow more tests to be performed.



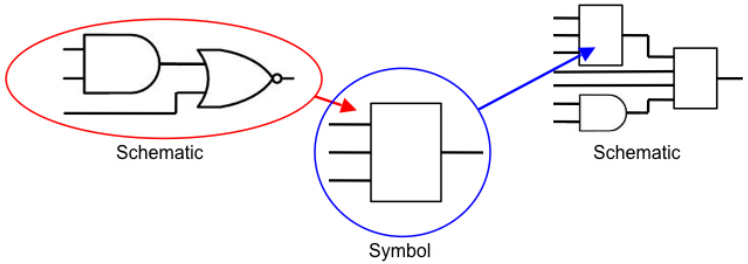
A typical FPGA comprises numerous logic blocks with programmable interconnections between them and the pins of the device. Each logic block comprises a small RAM – which can implement several gates-worth of combinatorial logic – and, optimally, a D-type flip-flop as an output register.

MANCHESTER 1824

The University of Manchester

Schematics and Symbols

A symbol is an abstract representation of a design/schematic …



- Library of 'primitives' provide the basic building blocks
  - AND, OR, NOT, D-type, etc.
- Used to make schematics
  - with 'sensible' choice of functions and I/O
- Schematic abstracted by a symbol

Version 2016 COMP12111: CAD & Verilog

Notes:

Schematics and Symbols

Schematic Capture

Schematic capture is a means of entering hardware designs graphically.

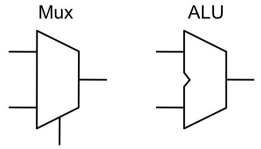
You have already seen some examples of logic functions drawn as gates, connected by wires. This has long been a standard way to represent, design and document circuits. By using a 'drawing' the designer has a two-dimensional surface to lay out the function, so –often – the signals flow causally from left to right whilst gates placed above and below each other are evaluating in parallel. [This is only an approximate convention, but it makes a simple model.)

Schematics are very good at showing interconnections, where wires and buses can be followed around a diagram. They are therefore appropriate for the higher levels of a design where reasonably complex blocks are interconnected (and abstracted).

Schematics can be used to interconnect gates at lower levels. This is sometimes useful. However, the detailed design of complex gate-level circuits can be tedious and other tools may be used. We will meet one shortly.

Symbols

A symbol is an abstract representation of any hierarchy it contains. It is largely just a drawing, and many symbols end up as (rather boring) rectangular boxes. The major exception to this are gates which have shapes as have been seen previously. There are one or two higher level 'shapes' which tend to be used; the two commonest are the multiplexer and the ALU.



The important features of a symbol are that:

- it is associated with the underlying design – usually by name
- it has pin connections which link to the underlying circuit

Schematic Capture Process

To manage a large design some hierarchy is essential.

- Basic library elements are used as components to assemble larger functional blocks in *schematic* form.
- The block is then abstracted as a 'black box' with a known function, some inputs and some outputs.
- The block is represented by a *symbol*.
- The symbol can then be used – repeatedly if necessary – in subsequent schmatics.

What makes a sensible 'size' schematic?

You have to discover this for yourself, but here are some guidelines.

- The function of the block should be easy to describe and understand; this does not necessarily mean it is simple.
  - e.g. 2-to-1 multiplexer, 32-bit adder, Pentium quad core processor.
- 'Cut' the design at 'natural' boundaries, typically minimising the number of input and output connections.
- Its complexity should be non-trivial but easily comprehensible.
  - Guide: it should fit on a 'page' without cramming if printed
  - Guide: about a dozen elements is typical
  - If 'too complicated' introduce more hierarchy
- Remember, other people have to be able to read it too.

Q.

Think about how you would draw a schematic for the full adder when built from half adders.

Quiz-time

Q. Testing is one of the KEY elements in the design process

A. true
B. false

Notes:

Quiz-time

Q. Symbols are an abstraction of a design. In the lab you have designed a full adder, which you have tested extensively, and then used its symbol in a 4-bit adder design. Do you have to test extensively again?

A. no – no testing is necessary
B. not extensively, but some tests are still required
C. yes – every possible input signal combination should be tested

Notes:

Quiz-time

Q. If we test EVERY possible state that a simple ARM processor can reside in at a rate of 1 every 0.000000001s (that's 1,000,000,000 states tested a second), how long would it take to test the **whole** design?

A. a few hours
B. a few days
C. a few years
D. a long time …

Notes:

Quiz-time

Q. The Intel Pentium chip (circa 1994) had a bug in it associated with floating point operations, how much did this cost Intel to remedy at the time?

A. around $1 milion
B. around $50 million
C. around $100 million
D. around $500 million

Notes:

MANCHESTER 1824

The University of Manchester

How do you test a design works?

As a designer you have created a new (non-trivial) deign and entered in a CAD system. Your design is almost certainly going to contain some **errors** …

You can remove errors by:

- thinking hard – best at finding 'odd' special cases
- running tests – best at finding misconceptions, typos, oversights, etc.

… **both** approaches are recommended.

A simulator uses an automated and independent view of what you produced.

- it obeys *exactly* what you wrote
- it does not do what you *meant* to write, but didn't

Simulation is **essential** in any sizeable design!

- get into the habit of verifying before you build the circuit
- ultimately, it will save you time!

Notes:

Simulation

In a design with any degree of complexity there is little chance that it will work 'first time'. Omissions in the specification, misconception in translation, typographical errors, etc. conspire to yield a design that probably does not-quite-the-right-thing.

This is true in software too. However, software can often be 'debugged' directly by running it and looking at the results. When making silicon this is not a realistic option because each attempt may cost in excess of £1M and a couple of months delay.

Silicon should be right the 'first time'. Whilst there are still occasional errors in some chips, it comes very close to achieving this aim. That is because designs are simulated for all conceivable circumstances before being sent for fabrication.

Chips are made of logic gates that have simple functions, such as AND and OR. These are interconnected in some way to achieve a higher-level function. Both the function and the interconnection are easy to describe, it is the number of gates and the resultant complexity which is difficult to manage.

Computers are good at following simple rules without getting bored and making mistakes. Therefore simulating what a circuit *will actually do* in a given circumstance is quite straightforward for a CAD tool to do.

Typically, simulation is used for three main purposes:

- Logic verification – ensuring the circuit does what you want it to do.
- Timing closure – finding if the circuit is fast enough; allowing the exploration of alternatives to increase the speed.
- Power estimation – making sure there is adequate power supplied.

In fact the gates are made of transistors that have their own properties. However, the manufacturer will typically supply gate models, which have been characterised (by simulation) so that your simulator has something reliable to work with.

In the lab …

Our labs are concerned with logic design and verification. Our designs will be quite small and not time critical so we shall only simulate to verify the functionality of the logic.

In some cases it will be possible to simulate a circuit *exhaustively*, i.e. to try every possible combinations of inputs. If you read ahead you might find out how to automate this process, so the tool will generate input patterned for you. However, as design sizes increase this soon becomes impossible.

Simulation Abstraction

In the labs we just use **digital** simulation to verify the **functional** correctness of a design. The timing model for the gates is quite approximate because we don't really need to know.

This keeps the results simple. For larger designs it also allows the use of abstraction which makes the simulation fast.

Other types of simulation are possible. **Analogue** simulations of the behaviour of each transistor and wire gives much more precise results for timing but takes much more computing power to do.

Power simulation is also important before a chip is completed.

Just to bandy about some inconceivably large numbers:

- There are probably about 10^{80} atoms in the universe
- The universe is about 4×10^{17} seconds old
- A simple model of the ARM processor can adopt 10^{154} states

If memory states are considered too, then that number gets significantly bigger.

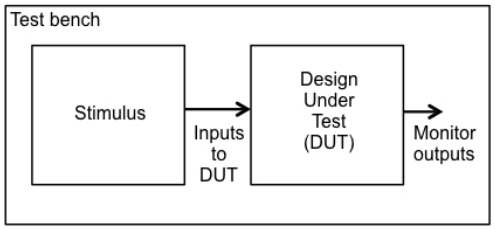
Even testing billions of patterns per second, the universe is not going to last long enough to cover a noticeable percentage of all the possible cases!

MANCHESTER 1824

The University of Manchester

Simulation

You have produced a design and you wish to simulate its behaviour. To do this you use a test bench …



The goal of simulation is change the inputs in a pre-defined manner and see that the design behaves as expected by monitoring any output signals.

Notes:

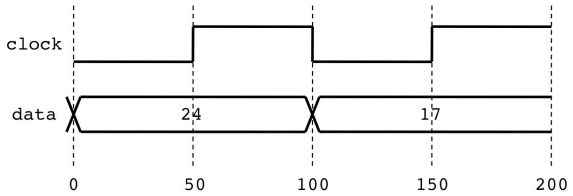
Stimulus File

The stimulus is provided via a stimulus file, which, in our case, is actually written in Verilog. Interactive simulation is possible. However, real simulation runs soon get long and quite repetitive as a test case is tried, a problem is identified, a fix attempted and the same case tried again. It is therefore useful to be able to record and replay inputs in a file.

An example stimulus file is

```
initial           // run once
  begin
    clock = 0;    // assign a value
    data = 24;
    #50           // wait a period of time (50ns)
    clock = 1;
    #50
    clock = 0;
    data = 17;
    #50
    clock = 1;
    #50
    $stop;        // stop the simulation
  end
```

The stimulus in the slide will generate this output:



Explanation:

- `initial` means 'run the following statement once when the file is invoked'
- `begin` and `end` bracket the statements in between into a single statement
- `clock = 0;` assigns a value at the current time
- `#50` advances the time by 50ns
- `$stop;` causes the simulator to stop running.

When the simulation is run the simulator allows time to pass but tracks the behaviour specified by the stimulus file. The circuit is now 'driven' to obey specific commands.

As systems get more complex the stimulus file tends to grow and become more elaborate. number of parallel processes may be included too, but we won't go there yet!

In the lab …

The stimulus files are written in Verilog.

You only need to use a few simple commands:

- thingy = 23; Set a variable (circuit input called 'thingy' here) to a value
- #100 Allow time to pass
- $stop; Stop the simulation

It is possible to do more elaborate functions in the stimulus file, such as changing behaviour in response to the test circuit's outputs, but we do not need to, yet.

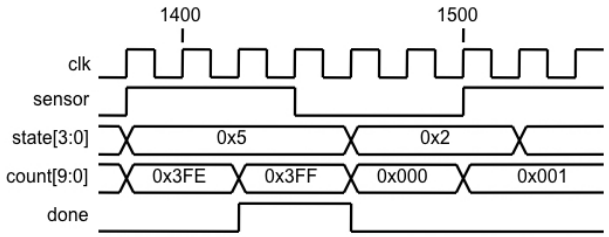
One simple enhancement may be to make a clock generator; see the lab manual for details.

MANCHESTER 1824

The University of Manchester

Waveform Viewer

Shows timing from the simulation. However, there could be many signals, possibly millions, in a typical design. Thus, we again need to manage complexity.



- Choose *which* signals to view (what do you want to know and test?)
- Can move through – and zoom - time
- Preserve designer's hierarchic model
- Keep buses, as appropriate

Notes:

Waveform Viewer

A useful output from a simulator is a waveform view (i.e. a timing diagram) that displays a set of signals evolving over the course of a simulation run. This allows you to read forwards and backwards in time to identify how and when something went wrong.

To be optimistic for a moment, it can also show you that something is functioning as it should. This should be the eventual result but it may take a few simulation runs to find and fix design errors before you get there.

The great advantage of simulation is that it increases observablity of the 'insides' of a circuit. You can spot a potential problem and trace back through the logic to fins exactly which signal went wrong first. This is not available once the design is in chip because only the signals on the outside are accessible (and only then if you are looking at them at the relevant time).

A suggestion to make life easier …

Every signal and every instance of a component has a name. You will have named certain signals because you wanted to identify them with I/O pins, but under most circumstances the names will be automatically generated. Thus, in Cadence, instances will be I0, I1, … and wires ('networks') net1, net2, net3, …

The simulator and viewer preserve the names and design hierarchy, which means that many of the signals available for selection will be hard to identify.

The simplest way of preventing confusion is to give meaningful names to all the signals by labelling every signals (Cadence: `Create` ↲ `Wire Name` …). This can be tedious when creating the schematic but may make things easier later. It's also possible to give your own names to instances (Cadence: `Edit` ↲ `Properties` ↲ `Objects` …) which makes the hierarchy easier to negotiate.

This is recommended behaviour; another 'good habit' that pays off increasingly as design complexity expands.

Additional Tools in the Lab.

As shown in the design flow there are a number of tools which you will use, but most of these are invoked by automated scripts and therefore 'hidden' from you.

There are only two other tolls that you may encounter.

Text Editor

A standard text editor is used for preparing test-based files. These are Verilog files and are used for simulation stimulus and for hardware descriptions – coming next.

Library Manager

The library manager allows you to browse and open designs for editing.

It should also be used if you need to copy, delete or rename designs. You don't (need to) know how the designs are stored. Leave the file management to the appropriate tool, **don't** try to hack from a shell because you'll probably miss some subtlety in the way the files are stored and referenced.

MANCHESTER 1824

The University of Manchester

Hardware Description Languages

Describing systems using schematics can be somewhat tedious. Hardware Description Languages (HDLs) are intended to make certain jobs easier.

- HDLs permit a textual description of hardware
- Higher level abstraction than schematics
  - Alleviate the need for user to perform detailed logic design/minimisation
  - Easy to parameterise and therefore maintain
  - More readable – in most circumstances
- They are not the best solution to everything.
  - Interconnectivity can be hard to express
  - Parallelism in the system less obvious than on a schematic

… the appropriate tool for the appropriate job.

There are two widely used HDLs – VHDL and Verilog.

Notes:

Hardware Description Languages

We have already met a little Verilog for generating stimuli for the simulation. We shall now look at some more of its capabilities which can be used for more elaborate stimuli but can also be used to define actual hardware systems.

HDLs describe an algorithm as a set of text statements, somewhat similar to a programming language (such as Java).

- This can be a Good Thing in that:
  - it hides some of the 'dirty' details
  - makes hardware design more accessible to software folk!
- This can be a Bad Thing if:
  - the difference are not appreciated
  - code is written to look like a serial programme.

Some Pros and Cons of using HDLs

Add your own opinions if you like!

Advantages

- Expresses some functions (such as addition, comparison) simply
- Suited to RTL design
- Performs logic synthesis – no need to produce gate-level designs (schematics)
- Code *may* be easier to read – if well written, structured, commented, etc.

Disadvantages

- Not guaranteed to synthesize to the most efficient circuit
- Easy to write something that produces excessive hardware
- Parallelism often harder to 'see' – especially to people used to thinking in terms of software design!

MANCHESTER 1824

The University of Manchester

Verilog

Verilog was originally intended as a description language for *modeling* systems. However, it was extended to enable it to be used to synthesise logic circuits.

Verilog is particularly suited for RTL descriptions.

We will use Verilog for:

- Stimulus files, for testing designs with a simulator
  - Not synthesizable (can not be translated into hardware)
- Hardware descriptions for building into running chips
  - Synthesizable (can be translated into hardware)

What follows is:

- ENOUGH to make things with and give a flavour of a typical development process
- NOT a complete description of the language; it has many more features

Notes:

Verilog v. Java

A HDL and a general-purpose software programming language have different functions and therefore different characteristics, strengths and weaknesses. Writing a windowed graphics application would be difficult (impossible?) in Verilog, but describing a hardware system in Java would also be hard, and synthesizing the result (into actual implemented hardware) would be almost impossible.

Verilog

Strengths

- Easily tailored to support hardware needs, such as different variable (bus) sizes.
- Inherently parallel at low level.
- Quite expensive; a lot can be done with a little code.

Weaknesses

- Multiple ways to express the same thing – some not very reliable.
- Low level; wires and buses always remain 'exposed'.
- Easy to write something which produces 'bloated' hardware.

Java

Strengths

- Expressive and sophisticated.
- Error detection and reporting good.
- Scales well for describing larger tasks.

Weaknesses

- Serial – not a good match for hardware design.
- Easy to write something that (if it could) would produce 'bloated' hardware.

Similarities

- Syntax of expressions is almost identical
- Conditional operations use 'same' syntax
  - if (<condition>) <statement1>; else <statement 2>
- Code is easily subdivided into units with private variables and well-defined interfaces
  - Methods and classes in Java
  - Modules in Verilog
- Units often *instantiated* multiple times in the programme.

Differences

- Bracketing different
  - 'begin' and 'end' in Verilog
  - '{' and '}' in Java
- Loops function differently
  - Verilog control is external – runs statements on demand
  - Java loops are command driven
- Parallelism
  - Verilog is inherently parallel
  - Java is basically serial
- Sophistication
  - Verilog is quite primitive – easy to abuse
  - Java contains lots of protection and facilities such as inheritance to make the programmer's life easier (!)

On the last point, remember software can 'crash' at run-time and give out error reports; a silicon chip can't do that, it just won't work!

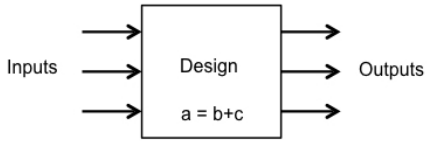
MANCHESTER 1824

The University of Manchester

Verilog - Modules

The module is the basic building block for describing digital systems – it is the equivalent of a schematic (or a symbol representing a design).

Consider a circuit design (here abstracted to a symbol):



The design:

- performs some function
- some inputs
- some outputs

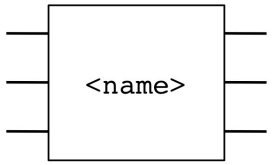
The module is the textual description of this design, it has inputs, outputs and produces some function.

The module is analogous to the class in OOP

Notes:

Module Syntax

Verilog can be confusing because there are sometimes some alternative syntaxes which work equally well. What follows is probably the clearest way of expressing a module.



A module is a logic function akin to the schematics you have seen previously, and it can be expressed by a symbol in the same way.

A Verilog module can therefore be used within a schematic, with the symbol being used in a high-level schematic and with the low-level content being described in Verilog.

A module begins with the keyword "`module`"; this is followed by the *name* of the module, which should reflect its function. A list of declared input and output variables follows.

A module ends with the keyword "`endmodule`".

The body of the module consists off declarations of any internal signals (there is none in the example on the slide) and statements that define the logic function.

Note: All the statements are potentially 'active' all the time.

There is no implicit ordering of the statements here.

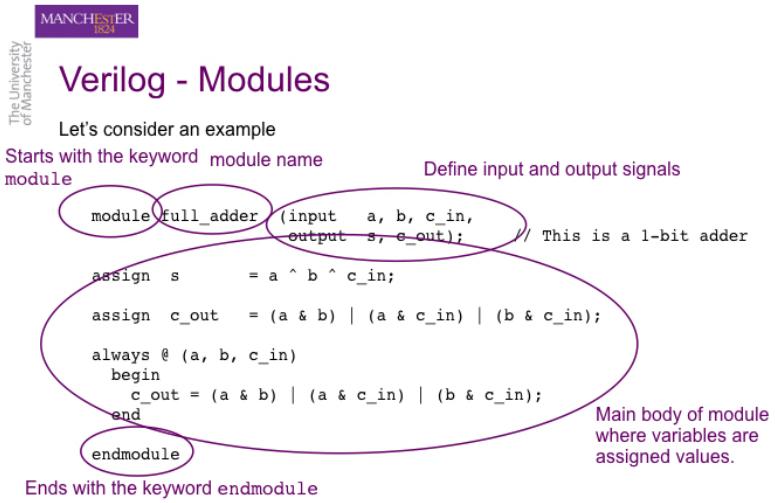
In the `full_adder` example any input changes will become apparent immediately at the outputs. The result is just the same as building the logic as a schematic.

Basic 'rules'

Some basic rules for producing modules:

- Keep the module's function straightforward and understandable
- 'Whitespace' has no effect on the syntax. It is advisable to indent for clarity in the same way you would in a programming language.
- Document appropriately with comments
  - /* <comment goes here >*/, or
  - // <comment goes here> ↵

Note that the module on the slide is rather trivial and larger functions are more typical.



Notes:

New style v. Old style

The module description illustrated in the slide is an example of the old style of module definition.

In the old style the input and output variables were defined within the body of the module, unlike the new style where they are defined in the header itself.

The full adder example using the old style:

```
module full_adder  (a, b, c_in, s, c_out);

input   a, b, c_in;
output  s, c_out;

// same as before

endmodule
```

and in the new style:

```
module full_adder  (input    a, b, c_in
                    output   s, c_out);

// same as before

endmodule
```

Both arevalid, and will be recognised by the CAD tools in the lab. However, we teaching you the new style.

MANCHESTER 1824

The University of Manchester

Verilog - Declarations

As in other programming languages, you must define the **type** of a variable being assigned a value.

- `wire` – which declares a connection between components (a net) – `wire` is the default type.
- `reg` – which declares something which may be a register but may just be a simple wire connection, this can be somewhat confusing and takes some getting used to.

Either of these may be declared as a bus of any width

```
wire        test;               // One bit wire
wire        a, b, c;            // Three more one bit wires
wire [15:0] mybus;              // Sixteen bit bus
reg  [63:0] storage;            // 64- bit register
```

If a whole bus is being used then it can be referred to by just its name

```
mybus                   // A 16-bit quantity
mybus[7:0]              // The eight LSBs of the bus
```


Notes:

Verilog – Declarations

The 'types' declared in Verilog only have to match the way they are assigned:

- `wire` – always a wire
  - used in "`assign`" continuous assignment
    output of (simple) **combinatorial** logic functions
  - used in higher level interconnection (not described here).
- `reg` – may be a wire of a register
  - used in blocking assignments (more later)
    guideline: use for (complex) **combinatorial** logic output
  - used in non-blocking assignments (more later)
    used for **sequential** (latched) outputs

Examples are given below.

There are no other types that we care about in the language; it is up to you to distinguish between integers, Booleans etc.

A Boolean is a single bit value where:

- '0' is regarded as 'FALSE'
- '1' is regarded as 'TRUE'

Beware:

- Verilog is case-sensitive
- Verilog allows individual wires (not buses) to be declared *implicity*

Thus:

```
wire        thing, wotsit;
assign      Thing  = 0;
assign      wotsit = thing;
```

will be accepted, but probably doesn't do what you wanted it to do.

Verilog Literal Values

By default, numbers are decimal. The numerical base can be specified as follows:

`'b` - prefixes a binary number
`'h` - prefixes a hexadecimal number
`'d` - prefixes a decimal umber (but it may be omitted in this case)

A (decimal) number in front of this specifies the number of bits in the number, i.e. the **bus width**.

Thus,

`16'h1234`

specifies a 16-bit hexadecimal number, this is 0001001000110100 in binary.

Q.

1. Express the numbers 0111000011100, 123456, C3DA as 16-bit .values.

2. Express the following values in binary: 'd54, 12'd54, 16'h0FFC.

MANCHESTER 1824

The University of Manchester

Verilog - Operators

As in other programming languages, Verilog has a number of operators than can be used.

Arithmetic operators:

- add '+', subtract '-', multiply '*' , divide '/' and modulus '%'.

Bit-wise logical operations:

- negation (complement) '~', AND '&', OR '|' and XOR '^'

Logical operations:

- negation (complement) '!', AND '&&', OR '||' and NOT equal '!='

Shift operations:

- shift right '>>', shift left '<<', concatenation {.}

Relational operations:

- greater than '>', less than '<', equal to '==', not equal to '!=', greater than or equal to '>=', less than or equal to '<='

Notes:

Verilog Operators

Verilog understands some simple arithmetic operations.

Operator Type	Symbol	Operation Performed
Arithmetic	+	addition
	–	subtraction
	*	multiplication
	/	division
	%	modulus
Logic (bit-wise or reduction)	~	negation (complement)
	&	AND
	|	OR
	^	XOR
Logical	!	negation
	&&	AND
	||	OR
	!=	NOT equal
Shift	>>	shift right
	<<	shift left
	{ , }	concatenation
Relational (comparison that return a Boolean value)	>	greater than
	<	less than
	==	equality
	!=	inequality
	>=	greater than or equal
	<=	less than or equal

(As a bitwise operation on a 1-bit value is the same as a logical operation these tend to be used interchangeably.)

Memories

It's not of tremendous use in the labs, but for interest, it is possible to declare memories too. For example:

```
reg [15:0] mem [12'h000:12'hFFF];
```

declares a memory of 4K, 16-bit words, such as may be used for our MU0 system.

This should be clearer when you have seen some of the later chapters of this course.

MANCHESTER 1824

The University of Manchester

Verilog – Control Statements

Like other programming languages Verilog includes control statements to control the flow of code execution.

The most common (and useful) is the `if` … `else` statement that have seen elsewhere.

```
if (<conditional statement>)
  begin
    // multiple statements
  end
else
  begin
    // multiple statements
  end
```

The conditional expression can be a Boolean expression which if true will result in the first set of statements being executed. Otherwise (else) the second set of statements will be executed.

Notes:

Control Statements

Why do we need control statements? How would you make decision without them? You need to be able to compare values and determine the difference between them, or whether they match a specified requirement etc.

In other programming languages you will use control statements such as:

`if … else`

`switch`

`while`

`do while`

`for`

`goto`

and more.

These statements are useful in programming as they allow you control the order in which code is run, rather than executing each line of code in sequence.

In Verilog we have similar control statements, for similar reasons. However, the only one that is really useful (because if can be easily translated into hardware) is the `if … else` statement. The `for` loop is also used in Verilog, but mainly for creating test stimulus and NOT for describing hardware.

MANCHESTER 1824

The University of Manchester

Verilog – if … else example

The if … else statement allows us to control the execution of code, just as in any other programming language.

Consider a design where the hardware is required to test an 8-bit input bus, sig_in, and if it is zero asserts an output signal, is_zero:

```
if (sig_in == 00000000)      Test Condition
  is_zero = 1;               If true then …
else
  is_zero = 0;               If false then …
```

Each condition can have many assignments, in which case they must be encapsulated within `begin` and `end` statements (like { and } in Java).

Notes:

`if … else` control statement

The `if … else` statement is one of the most important control statements and is used to control the execution of statements. The `if … else` statement must reside in an `initial` or an `always` block.

A simple Verilog description for D-type latch can use an `if … else` statement:

```
module D_type (input       D, En,
               ouput reg   Q);

  always @ (*)
    if (En == 1)
      Q = D;
    else
      Q = Q;

endmodule
```

Here, when `En` is 1 the output `Q` is set to the value of `D`, otherwise it remains unchanged. The keyword always will be discussed later, along with an explanation of how we assign values to variables in Verilog,

Q.

In the following code example what will happen if `En` = 0?

```
always @ (*)
  if (En == 1)
    Q = D;
```

You can omit the `else` in the expression and the behaviour will be the same in some cases! As a rule I suggest you always include the `else` statement, particularly when you are generating combinatorial logic, if you don't, strange things will happen, and probably not what you wanted to happen!

You can also have multiple `if … else` statements:

```
if (X == 00)
  // do something
else if (X == 01)
  // do something else
else        // this will catch all other values of X
  // do something else entirely
```

begin and end

Within the `if … else` statement you only need a begin and end if you have more than one assignment to a variable (more soon). The same applies to the contents of `initial` and `always` blocks … more soon.

Quiz-time

Q. Is there an error?

A. Yes
B. No

```
module test(input        signal,
            output  reg  done);

always @ (signal)
  if(signal == 0)
      done = 1;
  else if (signal == 1)
      done = 0;
  else
      done = `hX;

endmodule
```


Notes:

Quiz-time

Q. Is there an error?

A. Yes
B. No

```
module test(input        signal,
            output  reg  done)

always @ (signal)
  if(signal == 0)
      done = 1;
  else if (signal == 1)
      done = 0;
  else
      done = `hX;

endmodule
```

Version 2016 COMP12111: CAD & Verilog

Notes:

Quiz-time

Q. Is there an error?

A. Yes
B. No

```
module ltest(input        signal,
             output  reg  done);

always @ (signal)
  if(signal == 0)
      done = 1;
  else if (signal == 1)
      done = 0;
  else
      done = `hX;

endmodule
```


Notes:

Quiz-time

Q. Is there an error?

A. Yes
B. No

```
module test(input        signal,
            output  reg  done);

always @ (signal)
  if(signal = 0)
      done = 1;
  else if (signal = 1)
      done = 0;
  else
      done = `hX;

endmodule
```


Notes:

Quiz-time

Q. Is there an error?

A. Yes
B. No

```
module test(input        signal,
            output  reg  done);

always @ (signal)
  if(signal = 0)
      done = 1;

endmodule
```

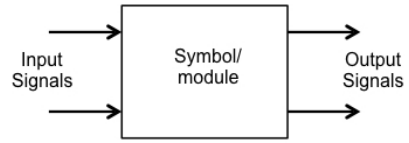

Notes:

MANCHESTER 1824

The University of Manchester

Verilog Assignments

The job of the module is to define the behaviour of the circuit/design – it must determine the value of any output signals.



The value of output signals are changed by assigning values to them.

In Verilog there are 3 forms of assignment, each with their own properties:

- continuous assignment
- blocking assignment (`a=b`)
- non-blocking assignment (`a<=b`)

The assignment you use depends on the type of circuit you are defining and how you define it.

Notes:

Verilog Assignments

Remember, the symbol is a means of abstraction in digital design that allows the designer to simplify the design. The symbol has input signals that are used to determine the state of the output signals. You can think of the module as the same (in fact a symbol in a schematic may simply be an instantiation of a module in the design).

Thus, in the same way the module must define the output signals depending on the state of the input signals. To do so the Verilog code must assign values to the output signals, or must be able to make decisions as to what the value of the outputs should be, so like other programming languages Verilog has control statements such as if … else in order to control the flow of code execution.

How do we assign values to output signals within the module? There are three ways we can do this, each of which will affect the implementation.

MANCHESTER 1824

The University of Manchester

Verilog - Assignments

Continuous Assignment

Continuous assignments are used to model combinatorial logic in a concise way.

Use the keyword 'assign'

```
assign  <variable> = <logical statement>;
```

For example:

a is given by c ORed with d, result ANDed with b

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

? is a query statement – if sel is 0 then q = x, otherwise q = y

```
assign q = (sel == 0) ? x : y;
```

Note: continuous assignment is only allowed on `wire` data types – so in the examples above, `a` and `q` must be declared as `wires` for the `assign` to be valid.

Continuous assignments are performed in the body of a module. They are not performed in `initial` or `always` blocks (more later).

Notes:

Continuous Assignments - Some More Examples

Continuous assignment using the assign keyword can be used to implement combinational logic, where the expressions are very simple using Verilog operators.

```
assign  s     =  a ^ b ^ c_in;
assign  c_out =  ( a & b) | (a & c_in) | (b & c_in);
```

This is a full adder made 'directly' rather than from half adders. The correspondence from the expressions to gates should hopefully be clear!

```
wire    [31:0]  x, y, z;    // 32-bit values
assign  z = x + y;
```

This is also an adder. Although it lacks the carry signals it is considerably clearer to use if you want to add quantities such as 32-bit numbers.

```
wire    [15:0]  x, y, z;
wire    match;
assign  match = (p == q);
```

`match` is the output of a comparator which evaluates if (the 16-bit buses) `p` and `q` are the same.

The 'query' operator

The query operator, ?,can be used to implement a *decision* statement, when using continuous assignment. For example the continuous assignment statement:

```
assign  q = (sel == 0) ? x : y;
```

means

```
if (sel == 0)    assign  q = x;
else             assign  q = y;
```

This is a 2-to-1 multiplexer. Note that `x` and `y` could be single wires of buses (declared as '`wire`' and not '`reg`').

Verilog - Assignment

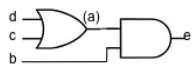
Blocking Assignment

A sequence of statements are executed 'in order' – top to bottom.

1) c + d

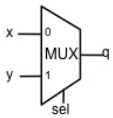
```
a = (c || d);
e = a && b;
```

2) AND the result with b




```
if (sel == 0) q = x;
else          q = y;
```

can be used with if … else statements



Use blocking assignments for:

- assigning variables in test ('stimulus') files
- describing complex combinatorial logic

The variable assigned must be a `reg` even though there may be no state holding involved.

Can only be used in an `initial` or `always` block (more later).

Notes:

Blocking Assignments

Blocking assignments allow the reassignment of a variable throughout a block of code. Consider:

```
count = count + 1;
if (count == 10)  count = 0;
```

The statements are 'executed' top to bottom in sequence. Thus:

- If count has an initial value of 8, its final value will be 9;
- If count has an initial value of 9, its final value will be 0;
- If count has an initial value of 10, its final value will be 11;

All assignments to a particular variable should be contained within the same 'block' of code. The block of code is 'executed' whenever some pre-specified conditions are met. There are two ways this can happen:

`initial`

- This code is 'executed' once, starting at 'time zero'.
- This is typically used for stimulus files.
- This is not (generally) synthesizable into hardware.

`always @ (<signal list>)`

- The code is 'executed' every time any signal in `<signal list>` changes. This can be synthesized into hardware.
- Normally all input signals (i.e. those which appear on the right hand side of any statement) should be included in `<signal list>` to ensure the block is combinatorial.
- If a signal is omitted from `<signal list>` then it can change without affecting the outputs, implying some state holding within the logic. This is legal, and occasionally useful, but not recommended for beginners!

A More Complex Example

The multiplexer on the slide is quite simple and can be written compactly as a continuous assignment. I choose to illustrate the use of a conditional `if … else …` operator, with a syntax that probably looks quite familiar.

However, consider a four input multiplexer:

```
always @ (sel, w, x, y, z)
  case (sel)
    0: q = w;
    1: q = x;
    2: q = y;
    3: q = z;
  endcase
```

(Note : `sel` is assumed to be a 2-bit bus here.)

The operation of the 'case' statement is similar to the 'switch' in Java, although the syntax is slightly different – we will look at this again soon.

- the keyword is `case`, not `switch`
- the cases are identified numerically
- each case executes a single statement
  - there is no `break`
  - if more functionality is needed, `begin` and `end` brackets can be used
- Case statements cannot be used in 'continuous assignment'. They are probably the most useful construct to require blocking assignments.

Verilog – the `always` block

We use always blocks to describe assignments that occur when something happens, such as a variable (or variables) changing, or with respect to some timing (a clock edge).

Syntax:

```
always @ (<condition>)
begin
  // statements
end
```

The `always` keyword does exactly what it says – perform these assignments *always*.

Execution of the assignments in the always block depends on the `<condition>` more later ....

Within an `always` block we can perform assignments using:

- blocking assignments
- non-blocking assignments

… you **NEVER** have continuous assignments within an `always` block.

Notes:

`always` blocks

As the name clearly suggests, and `always` blocks executes **always** from time 0 – think of it like an infinite loop. You can compare this behaviour to the `initial` block that executes **once** from time 0. The `always` block provides the designer with a high level of abstraction in order to implement more complex logic expression compared to continuous assignments.

A module can have any number of `always` blocks (and `initial` blocks), all of which are executed at the **same time**. Circuits are concurrent in nature; things don't happen in series, like they do with software code, things happen in parallel.

We will see soon that a *sensitivity list* is often provided together with the `always` keyword to determine *when* the always block should be evaluated. The general format is:

```
always @ (<sensitivity list>)
begin
  // procedural statements
end
```

More about the sensitivity list later.

Procedural statements can be implemented using

- blocking, and
- non-blocking

assignments.

Quiz-time

Q. Consider the code sequence:

initially a=0, b=1, c=2, what are the values of a, b, and c after the block has been executed?

```
begin
   a = 1;
   b = a;
   c = b;
end
```

A. a = 1, b = 2, c = 2
B. a = 0, b = 2, c = 2
C. a = 1, b = 1, c = 1
D. a = 0, b = 0, c = 0

Notes:

Quiz-time

Q. Consider the code sequence:

If count is initially 8, what will its value be after the block has been executed?

```
begin
   count = count+1;
   if (count == 10)
     count = 0;
   else
     count = count;
end
```

A. 8
B. 9
C. 0
D. 10

Notes:

Quiz-time

Q. Consider the code sequence:

If count is initially 9, what will its value be after the block has been executed?

```
begin
   count = count+1;
   if (count == 10)
     count = 0;
   else
     count = count;
end
```

A. 0
B. 9
C. 10
D. 11

Notes:

Quiz-time

Q. Consider the code sequence:

If count is initially 10, what will its value be after the block has been executed?

```
begin
   count = count+1;
   if (count == 10)
     count = 0;
   else
     count = count;
end
```

A. 0
B. 9
C. 10
D. 11

Notes:

Quiz-time

Q. What will the following code do?

```
always@(x)
 begin
  // assignments
 end
```

A. always do the assignments in the block
B. only do the assignments in the block when x changes
C. only do the assignments in the block when x = 0
D. only do the assignments in the block when x = 1

Notes:

Quiz-time

Q. What will the following code do?

```
always@(a,b)
 begin
  // assignments
 end
```

A. always do the assignments in the block
B. do the assignments when a and b change
C. do the assignments when a or b change
D. never do any assignments

Notes:

Quiz-time

Q. What will the following code do?

```
always@(posedge signal)
 begin
  // assignments
 end
```

A. always do the assignments in the block
B. do the assignments when signal = 1
C. do the assignments when we see a rising edge on signal
D. do the assignments whenever signal changes

Notes:

MANCHESTER 1824

The University of Manchester

The Sensitivity List

The condition in the brackets is called the sensitivity list

```
always @ (sel, w, x, y, z)        // sensitivity list
begin

   // do something

end
```

The statements listed in the always block are 'executed' whenever any signal in the sensitivity list changes value.

Such an `always` block will "generally" end up being synthesized into a block of combinatorial logic.

Rather than worrying about identifying which variables should be in the sensitivity list, we can use the * to specify all of them:

```
always @ (*)
```


Notes:

Sensitivity List

In the case of the following example:

```
always @ (a, b, c)
begin

// assignments

end
```

The always block will be executed whenever one of the variables in the sensitivity list changes state, the action is the OR of the variables.

Whenever a variable in the sensitivity list changes, the statements within the `always` block will be executed. It is important to make sure the variable you expect are listed in the sensitivity list, otherwise the statements will be evaluate as you expect!

If all input variables control the operation of the `always` block then use the * in the sensitivity list to force the `always` block to execute whenever any of the signals change.

Booleans

Remember Verilog does not enforce strict types.

- '0' is regarded as 'FALSE'
- '1' is regarded as 'TRUE'
  (in the case of a bus, anything non-zero represents 'TRUE')

In the code example on the slide `En` has been treated as a Boolean value, i.e.

```
if (En) …
```

rather than

```
if (En == 1) …
```

This is legal syntax and may make the code easier to read. Whether this is 'acceptable' or 'good practice' … you decide.

MANCHESTER 1824

The University of Manchester

Verilog – case statement

The `case` statement is useful when operation is determined by a number of options, and it can be used to replace a long list of `if … else` statements.

Following on from the previous example:

```
always @ (sel, w, x, y, z)
  case (sel)                          // case statement
    0: q = w;
    1: q = x;
    2: q = y;
    3: q = z;
  endcase
```

The value of `q` depends on the current value of `sel`

… if `sel=0`, `q=w`, else if `sel=1`, `q=x`, else if `sel=2`, `q=y`, else if `sel=3`, `q=z`.

Try to use a `default` case …

```
always @ (*)
  case (sel)                          // case statement
    0: q = w;
    1: q = x;
    2: q = y;
    3: q = z;
    default: q = w;
  endcase
```

If `sel` is not equal to 0, 1, 2, or 3

Notes:

'case's and 'default'

It is normal to include a `default` clause that captures any cases not explicitly stated. This avoids the synthesizer adding extra latches when they are not needed.

In the previous example of a 4-input multiplexer (assuming `sel` was a 2-bit value) all the cases are covered explicitly. However, a 5-input multiplexer (requiring a 3-bit select input) might look something like this:

```
always @ (sel, a, b, c, d, e)
  case  (sel)
    0: q = a;
    1: q = b;
    2: q = c;
    3: q = d;
    4: q = e;
    default: q = 0;
  endcase
```

Even if you know (or believe?) `sel` cannot adopt the value '6', the logic synthesizer does not. This code defines the output for **ALL** possible cases.

Adding `default` is harmless even if all the cases are already specified.

Strange Behaviour

Let's see what this subtly different statement does:

```
always @ (w, x, y, z)
  case (sel)
    0: q = w;
    1: q = x;
    2: q = y;
    3: q = z;
  endcase
```

This is evaluated when any of the data inputs change. However, we can change '`sel`' without invoking the statement.

Set the following inputs:

```
sel = 0;
w   = 1;
x   = 3;
y   = 5;
z   = 7;
```

`q` should be 1.

Change `w` to 2 and `q` becomes 2 as well.

Now **change `sel`** to 3; you might expect to select input `y` and therefore `q` becomes 5, but – in practice – **`q` remains 2** because the `always` statement has not been executed.

Now change `z` to 8; the statement is reinvoked and suddenly `q` becomes 5.

This could be what you wanted to happen, but it is unlikely!

If such a statement is synthesized, logic will be generated to do what you asked for. This will require the inclusion of various latches to hold previous values, so the circuit will get larger (slower, hotter) as well as (probably) not working.

So take care!

Quiz-time

Q. For the code shown …

Initially sel = 0, w = 1, x = 3, y = 5 and z = 7. So q = ?

```
always@(w,x,y,z)
case(sel)
  0: q = w;
  1: q = x;
  2: q = y;
  3: q = z;
endcase
```

A. 1
B. 3
C. 5
D. 7

Notes:

Quiz-time

Q. For the code shown …

Initially sel = 0, w = 1, x = 3, y = 5, z = 7 q = 1. If we change w to 2, q = ?

```
always@(w,x,y,z)
case(sel)
  0: q = w;
  1: q = x;
  2: q = y;
  3: q = z;
endcase
```

A. 1
B. 2
C. 3
D. 0

Notes:

Quiz-time

Q. For the code shown …

Initially sel = 0, w = 1, x = 3, y = 5, z = 7
q = 1. If we change sel to 2, q = ?

```
always@(w,x,y,z)
case(sel)
  0: q = w;
  1: q = x;
  2: q = y;
  3: q = z;
endcase
```

A. 1
B. 3
C. 5
D. 7

Notes:

Quiz-time

Q. For the code shown …

If we change sel to 4. What happens?

```
always@(sel)
case(sel)
  0: q = 1;
  1: q = 2;
  2: q = 3;
  3: q = 4;
endcase
```

A. the hardware breaks
B. the output q remains unchanged
C. q is set to 0
D. q becomes undefined

Notes:

Quiz-time

Q. For the code shown …

What happens if sel = x?

```
always@(sel)
case(sel)
  2'b00: q = 1;
  2'b01: q = 2;
  2'b10: q = 3;
  2'b11: q = 4;
  default: q = 0;
endcase
```

A. q becomes undefined
B. q = 0
C. q remains unchanged
D. q = x

Notes:

Quiz-time

Q. For the code shown …

What is q if a = 2?

```
if (a == 'b00)
  q = 2;
else if( a == 'b01)
  q = 0;
else if( a == 'b10)
  q = 3;
else if( a == 'b11)
  q = 2;
else
  q = 0;
```

A. 0
B. 1
C. 2
D. 3

Notes:

Quiz-time

```
always@(signal)
if ((signal < 10) || (signal > 50))
  out_of_range = 1;
```

Q. For the code shown, what will happen if signal is in the range 10 <= signal <= 50?

A. out_of_range = 0
B. nothing, out_of_range remains unchanged

Notes:

The Sensitivity List … continued

We can also make the `always` block *sensitive* to the *timing* of other signals, such as an external clock.

To do so we can specify the *edge* of the signal for which we want the `always` block to react to.

We can specify in the sensitivity list whether the always block is sensitive to …

- the rising edge/transition of a signal (0 -> 1) using `posedge`
- the falling edge/transition of a signal (1 -> 0) using `negedge`

For example:

```
always @ (posedge clock)  // when we see a rising edge of
                          // the clock …
begin
   // do something
end
```

This will invariably result in a sequential circuit being synthesized in hardware.

Notes:

Timing in the sensitivity list

Time dependent behaviour can be initiated in the sensitivity list by using the keywords

- `posedge` – execute on the rising (0->1) transition of the signal following the `posedge` keyword
- `negedge` – execute on the falling (0->1) transition of the signal following the `negedge` keyword

For example:

```
always @ (posedge clock)
begin

// assignments

end
```

will execute the `always` block whenever there is a rising edge on the input signal `clock`.

In this example:

```
always @ (posedge clock, negedge reset)
begin

// assignments

end
```

the `always` block will be executed on a rising edge on the input signal `clock`, or a falling edge on the input signal `reset`.

General Coding Guidelines

The following guidelines are useful to bear in mind when you come to write your own Verilog code:

- when modelling combinatorial logic use continuous assignment or blocking assignments
- when modelling sequential systems, flip-flops, registers, etc, use non-blocking assignments
- if your design contains both combinatorial and sequential in the same `always` block then use non-blocking assignments
- **NEVER** mix blocking and non-blocking assignments in the same `always` block – the compiler may report an error for this
- **NEVER** assign values to the same variable from more than one `always` block – this will lead to undesirable behaviour.

MANCHESTER 1824

The University of Manchester

Verilog - Assignment

Non-blocking Assignment

Non blocking assignments in a block of code occur in parallel – all at the same time.

The non-blocking assignment is identified by the '<=' symbol

Consider the following code:

```
a <= 1;
b <= a;
c <= b;
```

All three assignments are performed at the same time. So what values are the variables a, b, and c assigned?

In this case a will become 1, b will become the **old value** of a, and c will become the **old value** of b.

Notes:

Non-blocking Assignments

Non-blocking assignments assign a variable once per execution of that block. Consider:

```
count <= count+1;
if (count == 10) count <= 0;
```

This is BAD!

If count has an initial value of 8, the final value will be 9;

If count has an initial value of 9, the final value will be 10;

If count has an initial value of 10 the final value will be *undefined*, because both statements should assign a value simultaneously and these values will be different.

Any non-blocking assignments should be mutually exclusive, thus, this is legal code:

```
if (count == 9) count <= 0;
else            count <= count + 1;
```

Some Verilog you probably won't need

This is not supposed to be a complete introduction to Verilog. This section is simply for interest. However, you might like to know the syntax for some other operations.

Dividing Buses

Specify the bits concerned with the bus name.

```
wire  [31:0] my_bus;
wire   [7:0] one_byte;

assign one_byte = my_bus[15:8];
```

Concatenating Signals

The wires/buses are listed in order within '{','}'.:

```
wire  [31:0] my_bus;
wire   [7:0] byte_0, byte_1, byte_2, byte_3;

assign my_bus = {byte_3, byte_2, byte_1, byte_0};
```

Replicating Signals

e.g. to AND all the bits in a bus with copies of a single bit.

```
wire  [31:0] my_bus, result;
wire         one_bit;

assign result = my_bus & {32{one_bit}};
```

Note:

```
assign result = my_bus & one_bit;
```

will compile, but will **not** work correctly: `one_bit` will be extended to

```
0000_0000_0000_0000_0000_0000_0000_000'one_bit'
```

Verilog - Assignment

Non-blocking Assignment

Non blocking assignments **ALWAYS** occur in always blocks …

```
always @ (posedge clock)
   begin
     a <= 1;
     b <= a;
     c <= b;
   end
```

on the positive rising edge of the signal clock do …

… and are always associated with events that occur with respect to a clock

So non-blocking assignments are always associated with the description of **sequential** circuits.

Notes:

Some Pitfalls

You need to be careful when to use blocking, =, and non-blocking ,<=, assignments. Generally, you use blocking for implementing combinatorial circuits, and non-blocking for implementing sequential circuits.

There are some pitfalls you must be careful about:

1) Not assigning every variable that can be assigned in a combinatorial (`always@(*)`) logic block.

Consider the following example

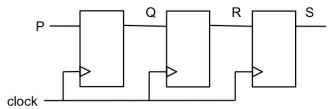
```
always@(*)
  if(X)
    Y = 0;
```

What happens if X ≠ 1? Then Y will not be assigned when the block is executed. The tools will implement this code in hardware using a latch, which is not what you want, and your design is no longer combinatorial in nature.

2) Using blocking, =, assignments in a sequential (`always@(posedge clock)`) block.

Consider the following example

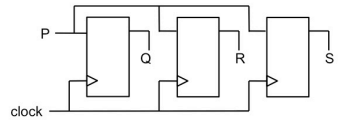
```
always@(posedge clock)
begin
  Q <= P;
  R <= Q;
  S <= R;
end
```



Shift Register

In this code the three assignments are made at the same time, this would result in a what is known as a shift register, as shown.

If you mistakenly used blocking, =, assignments instead, then the assignments would be done in order and would result in a parallel loading register being produce, as shown – not what you expected!



Parallel loading Register

3) Similar to point 2) make sure you use blocking, =, assignments in combinatorial always blocks (`always@(*)`) and not non-blocking, <=, as the results may not be what you expect!

4) Incomplete sensitivity lists

If you inadvertently missed out a variable in the sensitivity list for a combinatorial logic block, then this may result in the block not operating as expected. The result may be that latches will be introduced into your design (like point 1). Some tools will actually recognise this as a problem and give you a warning at compilation, other tools may not. So what should you do? Always use * as the sensitivity list when defining combinatorial logic always blocks.

Quiz-time

Q. Consider the code sequence:

If count is initially 8, what will its value be after the block has been executed?

```
always @ (posedge clock)
begin
   count <= count+1;
   if (count == 10)
     count <= 0;
end
```

A. 8
B. 9
C. 0
D. 10

Notes:

Quiz-time

Q. Consider the code sequence:

If count is initially 9, what will its value be after the block has been executed?

```
always @ (posedge clock)
begin
   count <= count+1;
   if (count == 10)
     count <= 0;
end
```

A. 0
B. 9
C. 10
D. 11

Notes:

Quiz-time

Q. Consider the code sequence:

If count is initially 10, what will its value be after the block has been executed?

```
always @ (posedge clock)
begin
   count <= count+1;
   if (count == 10)
     count <= 0;
end
```

A. 0
B. 10
C. 11
D. arrghhh – computer says no!

Notes:

Quiz-time

Q. Is the code shown correct?

```
always @ (*)
begin
   avariable <= 1;
   bvariable <= 2;
end
```

A. No
B. Yes
C. I don't know

Notes:

Quiz-time

Q. Is the code shown correct?

```
always @ (*)
begin
   avariable <= 1;
   bvariable = 2;
end
```

A. No
B. Yes
C. Not sure

Notes:

Quiz-time

Q. So we have been introduced to 3 ways of assigning values to variables in Verilog. When do we use continuous assignments, “assign”?

A. To model any type of logic?
B. To model combinatorial logic?
C. To model sequential logic?

Notes:

Quiz-time

Q. So we have been introduced to 3 ways of assigning values to variables in Verilog. When do we use blocking assignments, "="?

A. To model any type of logic?
B. To model combinatorial logic?
C. To model sequential logic?

Notes:

Quiz-time

Q. So we have been introduced to 3 ways of assigning values to variables in Verilog. When do we use non-blocking assignments, "<="?

A. To model any type of logic?
B. To model combinatorial logic?
C. To model sequential logic?

Notes:

MANCHESTER 1824

The University of Manchester

Design Partitioning

A single `module` can have numerous `assign` and `always` blocks.

- use these to partition your design sensibly
  - keep each block simple
  - keep one function in one place

but

- only assign a variable in one block
- DON'T mix blocking, '=', and non-blocking, '<=', assignments within an `always` block (you will actually get an error message from the compiler)
- strive for **legibility**
  - write your code in such a way that someone else can read it and can understand the function it implements.
  - use appropriate comments to describe what your code does!

Notes:

Aids to Legibility

These remarks are aimed at Verilog but the principles should be applied to all code that you write.

Commenting Code

- Comments can be added to the 'end' of any line using '//' – a line return will end the comment.
- Sections of code can be commented out using '/*' … '*/' bracketing.
- Suggestions:
  - Use the former syntax routinely
  - Use the latter for 'temporary' changes, such as commenting a block of code to experiment with an alternative.
- Comments should add something to the meaning conveyed in the code:
  - Good comment:

```
total <= total + 1;    // inc. count of cars
```

  - Poor comment:

```
total <= total + 1;    // add one to 'total'
```

Text Substitution – pre-compiler macros

Rather than just using numbers to identify states, readability can be aided by using names. Verilog is not particularly good as assisting this, but it is possible with text substitution.

```
`define READY     3'b000
`define DEC       3'b100
`define INC       3'b111
`define CMP       3'b001
…

 case (state)
 'READY: if (out)     state <= `DEC;
         else if (in) state <= `INC;
 'INC:   state <= 'CMP;
…
```

(Note: make sure you use the correct tick `, and not ', in your definitions.)

This is a pre-compiler directive, the compiler will substitute values for your macros when compiling. It allows you to easily change the values of variables used in your code.

Notes:

- Verilog uses ` when defining and using these substitutions (make sure you use the correct one!)
- A widely used convention is that such substitutions are upper case
  - Just a user convention, but …
  - not just used in Verilog; common in (e.g.) C programming

Is it = or <=?

In general you should use blocking assignments, =, to implement combinatorial logic blocks (`assign` and `always @ (<sensitivity list>)`) and non-blocking assignments, <=, to implement sequential (latched) blocks `always @ (posedge <clock>)`.

You should NEVER mix blocking and non-blocking assignments.

Quiz-time

Q. For the code shown, a changes value to 1. What is the value of b after?

A. 0
B. 1
C. You can't be sure!

```
module demo(input      a,
            output reg c);

always @ (*)
  b = a;

always @ (*)
  b = -a;

endmodule
```


Notes:

Quiz-time

Q. For the code shown, a changes value from 0 to 1. What is the value of c after?

A. 0
B. 1
C. You can't be sure!

```
module demo(input      a,
            output reg c);

//define internal signals
reg b;

always @ (*)
  b = a;

always @ (*)
  c = b;

endmodule
```


Notes:

Quiz-time

Q. Is the code shown syntactically correct?

A. Yes
B. No

```
module demo(input      clk, a,
            output reg c);

//define internal signals
reg b;

always @ (*)
  b = a;

always @ (posedge clk)
  c <= b;

endmodule
```


Notes:

Quiz-time

Q. What is the value of `c` if initially `a` is assigned the value of 1 before the clock edge?

A. Undefined
B. Old value of `a`
C. 1
D. It remains unchanged

```
module demo(input      clk, a,
            output reg c);

//define internal signals
reg b;

always @ (*)
  b = a;

always @ (posedge clk)
  c <= b;

endmodule
```


Notes:

Notes:

Notes:

Notes:

Notes:

Parallelism

One of the more difficult concepts to grasp is the parallelism in Verilog – or any hardware design. The code and schematic below describe the same function:

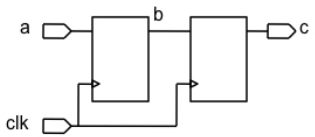
```
module  demo (input        clk, a,
              output  reg  c);

//define internal signals
reg b;

always @ (posedge clk)
  b <= a;

always @ (posedge clk)
  c <= b;

endmodule
```



- The two assignments in the two always blocks are synchronous , i.e. they happen simultaneously
- The (non-blocking) assignments can be written in any order – the result will be the same.

Notes:

Parallelism

All three of these modules have identical behaviour:

```
(1)   module  demo (input        clk, a,
                    output reg   c);

      reg  b;

      always @ (posedge clk)
        b <= a;
      always @ (posedge clk)
        c <= b;
      endmodule
```

```
(2)   module  demo (input        clk, a,
                    output reg   c);


      reg  b;

      always @ (posedge clk)
        begin
        b <= a;
        c <= b;
        end
      endmodule
```

```
(3)   module  demo (input        clk, a,
                    output reg   c);


      reg  b;

      always @ (posedge clk)
        begin
        c <= b;
        b <= a;
        end
      endmodule
```

Any of these examples will be fine syntactically! Note: that this is **not true for blocking assignments**.

This is syntactically correct and has the same effect as the examples above:

```
      always @ (posedge clk)
        begin
        c = b;
        b = a;
        end
```

This is syntactically correct and has a different result because 'b' is changed *before* it is read:

```
      always @ (posedge clk)
        begin
        b = a;
        c = b;
        end
```

For clocked circuits both of these forms are strongly discouraged. In general you should use non-blocking assignments when dealing with timed behaviour, i.e. the sensitivity list has a `posedge` or `negedge` in it!

MANCHESTER 1824

The University of Manchester

Verilog Synthesis

Not everything that can be written in Verilog can be synthesized into hardware.

- Example: #100
  - There is no reliable, technology-independent, temperature-stable way of making a 100 ns delay on a chip
  - If a delay is necessary it is down by counting clock pulses from a stable, external clock

Only a subset of the language is therefore available if writing circuit descriptions.

- Can use:
  - always, assignments, most operators (comparisons, +, -), if .. else, …
- Can't use:
  - initial, #nn, for, while, $display, some operators (*, /), …

Most of these are 'obvious' if you think about what is happening. If you need more functionality you will need to write it!

Notes:

Verilog for Simulation

`initial` blocks are useful for setting the initial value of variables, for example when testing:

```
initial
  begin
    a  =  5;            // Set at time = 0
    b = 0;              // Set at time = 0
    #100
    b = b + 1;          // Set at time = 100ns
    #100
    b = b + 1;          // Set at time = 200ns
    #100
    $stop;
  end
```

Multiple `initial` statements can be used:

```
initial  a = 5;         // Set at time = 0

initial
  begin
    b = 0;              // Set at time = 0
    #100
    b = b + 1;          // Set at time = 100ns
    #100
    b = b + 1;          // Set at time = 200ns
    #100
    $stop;
  end
```

We can also combine `initial` and `always` statements within the same code for simulation:

```
initial  a = 5;         // Set at time = 0
initial  b = 0;         // Set at time = 0

always
  begin
    #100
    b = b + 1;          // every 100ns
  end

initial                 // Stop the run after 300ns
  begin
    #300
    $stop;
  end
```

All three examples here perform the same action. If the counting was to be extended, the last would certainly be the most convenient (just after the "300" appropriately) but you should only use a form you feel comfortable with.

Some restrictions on synthesis

Multiplication & Division

Perhaps slightly surprising? However, there are numerous ways of performing a multiplication trading off hardware requirements and speed. The synthesizer doesn't know what you want. Division (by repeated subtraction) poses a similar dilemma.

Control Structures

Want to do something 100 times? In Java you might choose to use:

```
for (i = 0; i < 100; i = i+1) …
```

but in Java it's clear *when* this happens as there is a single point of execution. In fact it's a state machine. In Verilog we need to describe the process *explicitly*, for example

```
always @ (posedge clock)
  begin
  if (start)
    if(i == 0)
      active <= 1;
    else if (i == 100)
    begin
      active <= 0;
      i <= 0;
    end
    else    i  <= i + 1;
  end
```

(`active` is asserted for 100 clock cycles)

Quiz-time

Q. Within a module variables (outputs from the module) are assigned values using continuous assignments (“assign”) or within always blocks?

A. True
B. False

Notes:

Quiz-time

Q. A variable declared as a “reg” will always be implemented as a register?

A. True
B. False

Notes:

Quiz-time

Q. The assignments shown will synthesize into the same block of hardware?

```
assign q = (sel == 0) ? x:y;

if (sel == 0)  q = x;
else           q = y;
```

A. True
B. False

Notes:

Quiz-time

Q. The always block will be executed when p and q change?

```
always @ (p, q)
begin
  // statements
end
```

A. True
B. False

Notes:

Quiz-time

Q. Non-blocking assignments can be used in ANY always block?

A. True
B. False

Notes:

Quiz-time

Q. A module must contain a single initial block, a single always block and a single continuous assignment?

A. True
B. False

Notes:

MANCHESTER 1824

The University of Manchester

Quiz-time

Q. An initial block can be synthesized into hardware?

A. True
B. False

Notes:

Verilog Synthesis

Logic synthesis is the process of transforming a HDL description into an optimized netlist of gates that perform the required operation. The way this translation is performed depends upon the target hardware. Special software tools perform this process, largely independent of the user.

How will the logic synthesizer interpret the various constructs we have looked at into hardware.

`assign`

The assign statement describes combinatorial circuits. If the `assign` statement has Boolean equations then it will be synthesized into the appropriate logic gate circuit. For example:

```
assign X = control ?  I1 : I2
```

would synthesize to a 2:1 mux, with `control` as the signal select.

`always`

The `always` statement may produce a combinatorial or a sequential circuit. If the sensitivity list does not contain any clock (`posedge` or `negedge`) then the resulting hardware will be combinatorial. For example:

```
always @ (I1 or I2 or control)
  if (countrol) X = I1;
  else          X = I2;
```

is again an implementation of a 2:1 multiplexer. Remember, whenever any of the variable listed in the sensitivity list change, then the always block will be evaluated according to the Boolean expression in the brackets.

The use of an always block with a `posedge` or `negedge` clock will always produce a sequential circuit.

```
module flip_flop (input      clk, reset, d,
                  output reg q);

always @ (posedge clk)
begin
  if (reset == 1)
    q <= 0;
  else
    q <= d;
  end
endmodule
```

is an example of a simple D-type flip-flop with synchronous reset (you could make the reset asynchronous by including it in the sensitivity list).

Some general guidelines for Verilog structure

I do want to set down any rules to how you should structure your Verilog; we aren't teaching you to become expert Verilog coders, we just want you to get a feel for the language and how useful it can be for implementing (relatively easily) complex hardware designs. The following list offers some guidance on how to structure your code:

- Use meaningful names for signals and variables
- Comment code, in a sensible way, to make it readable
- Don't mix level and edge sensitive elements in the same always block
- Avoid mixing positive and negative edge-triggered flip-flops
- Use parentheses to optimize logic structure
- Use continuous assign statements for simple combinatorial logic
- Use non-blocking (<=) for sequential and blocking (=) for combinatorial logic
- Do not mix blocking and non-blocking assignments in the same always block
- Remember that all non-blocking assignments in an always block are evaluated at the same time

Never have dangling else's if you expect the code to be combinatorial … otherwise the compiler may try to "fix" your design and introduces latches, which isn't what you want!

MANCHESTER 1824

The University of Manchester

CAD Tools: Summary and Conclusions

- Tools are there to make life easier
- Many tools exist – all have a function
  - e.g. simulation does save time, even on quite small designs
  - Toolflows can be quite complicated.
- Hardware languages
  - Powerful tool to aid design
  - Not as 'evolved' as software languages
  - Parallelism quite exposed.
- Verilog
  - Primitive in places
  - Sometimes confusing
  - Can work in your favour – accelerate the design process
  - Widely used
- Infeasible to make modern devices without using tools like these (& more).

Notes:

Summary

We have looked at CAD tools as an aid for the designer of digital circuits. We have looked at Hardware Description Languages and looked at some detail of the syntax of Verilog, the HDL you will be using in the lab.

Next …

We will take a look at designing sequential systems and register transfer level (RTL) design. We will look at how we can design simple digital systems for controlling simple operations.