Imperial College London

MSc in Analogue & Digital IC Design

Mastering Digital Design …
… in Verilog with FPGAs

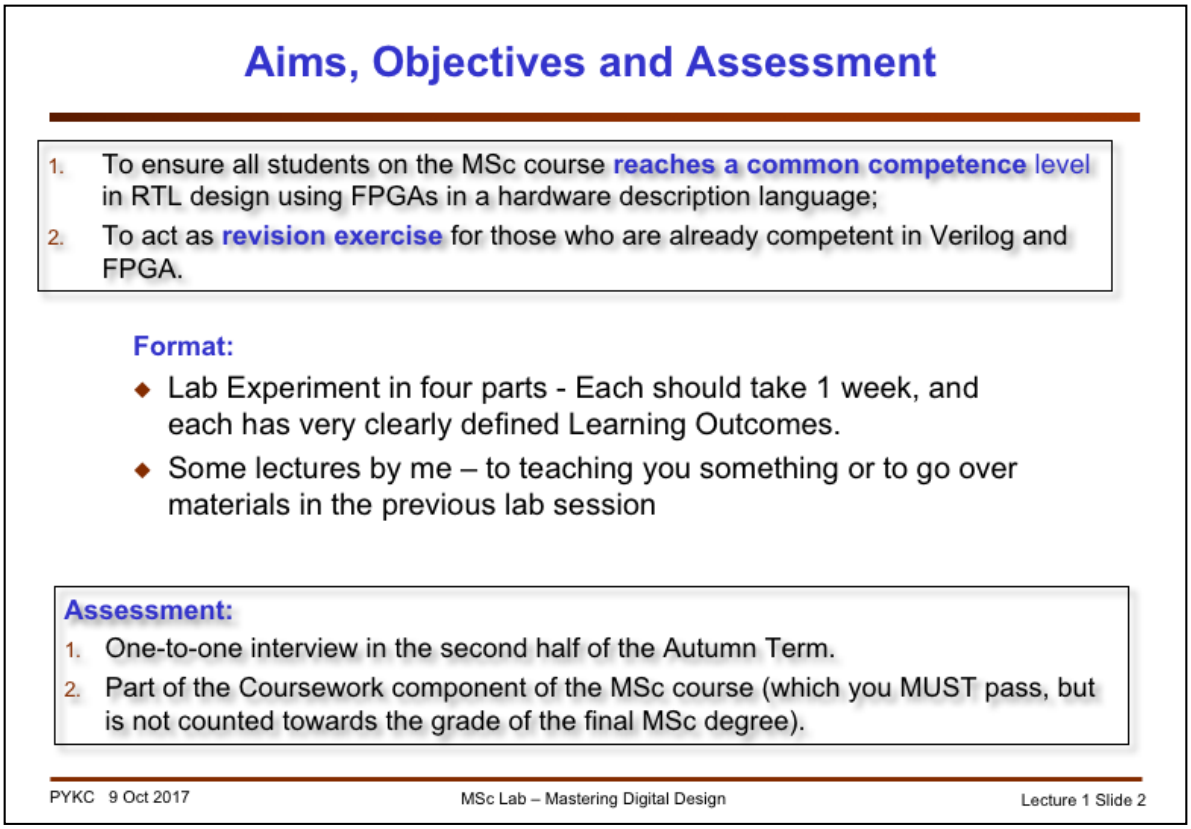
Peter Cheung
Department of Electrical & Electronic Engineering
Imperial College London

Course webpage: www.ee.ic.ac.uk/pcheung/teaching/MSc_Experiment/
E-mail: p.cheung@imperial.ac.uk

Welcome to this MSc Lab Experiment. All my teaching materials for this Lab-based module are also available on the webpage:

www.ee.ic.ac.uk/pcheung/teaching/MSc_Experiment/

The QR code here provides a shortcut to go to the course webpage.



This Lab Experiment is compulsory and its goal is to ensure that ALL students on this course get to a level of competence in digital design, Verilog and FPGAs as expected with our MSc graduates.

If you are already experienced with Verilog and/or FPGAs, you will find this experiment quite easy. However, if you have not done either in your UG degree programme, this is a great chance for you to catch up. This Laboratory served as a "levelling" purpose – make sure that all students on the course reach a common level and standard in digital design.

The learning outcomes for each of the four parts are:

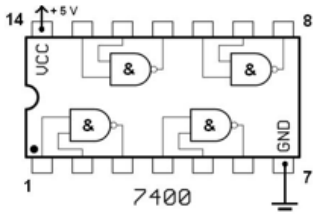
Part 1: Basic competence in using Intel/Altera's Quartus design systems for Cyclone-V FPGA; appreciate the superiority of hardware description language over schematic capture for digital design; use of case statement to specify combinatorial circuit; use higher level constructs in Verilog to specify complex combinatorial circuits; develop competence in taking a design from description to hardware.

Part 2: Use Verilog to specify sequential circuits; design of basic building blocks including: counters, linear-feedback shift-registers to generate pseudo-random numbers, basic state machines; using enable signals to implement globally synchronisation.

Part 3: Understand how digital components communicate through synchronous serial interface; interfacing digital circuits to analogue components such as ADC and DAC; use of block memory in FPGAs; number system and arithmetic operations such as adders and multipliers; digital signal generation.

Part 4: Understand how to implement a FIFO using counters as pointer registers and Block RAM as storage; implement a relatively complex digital circuit using different building blocks including: counters, finite state machines, registers, encoder/decoder, address computation unit, memory blocks, digital delay elements, synchronisers etc.; learn how to debug moderately complex digital circuits.

Old ways of implementing digital circuits



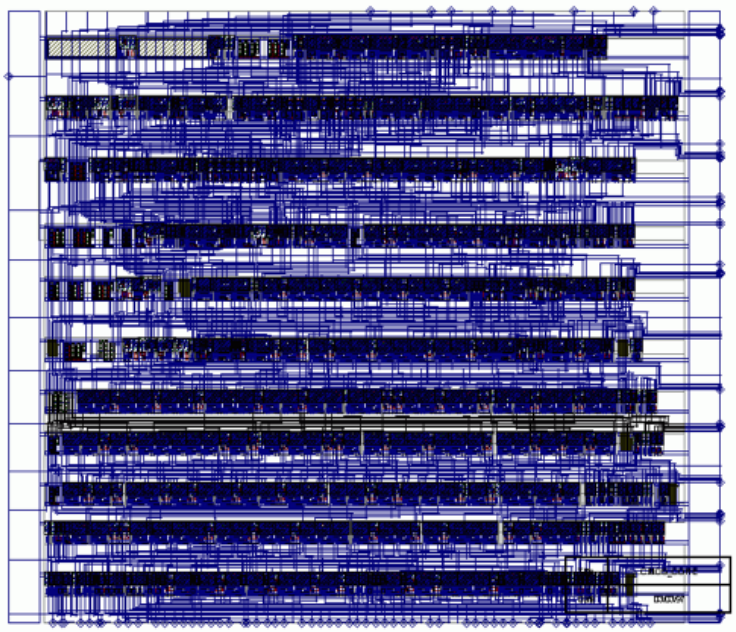
- Discrete logic – based on gates or small packages containing small digital building blocks (at most a 1-bit adder)
- De Morgan's theorem – theoretically we only need 2-input NAND or NOR gates to build anything
- Tedious, expensive, slow, prone to wiring errors

PYKC 9 Oct 2017 MSc Lab – Mastering Digital Design Lecture 1 Slide 3

You would have been taught at least how to implement digital circuits using gates such as the one shown here. You can still buy this chip with FOUR NAND gates in one package and this is known as discrete logic. We generally **do not** use these any more. It is slow, expensive, consumes lots of energy and very hard to use.

Nevertheless, it is good to learn about NAND and NOR gates because, using De Morgan's theorem, you could in theory design and implement a Pentium microprocessor using use two input NAND or NOR gates alone. It is therefore could be regarded as the building block of all digital circuits. Similarly, you could in theory build a car using only basic Lego blocks. Unfortunately such a car would not be very good.

Early integrated circuits based on gate arrays



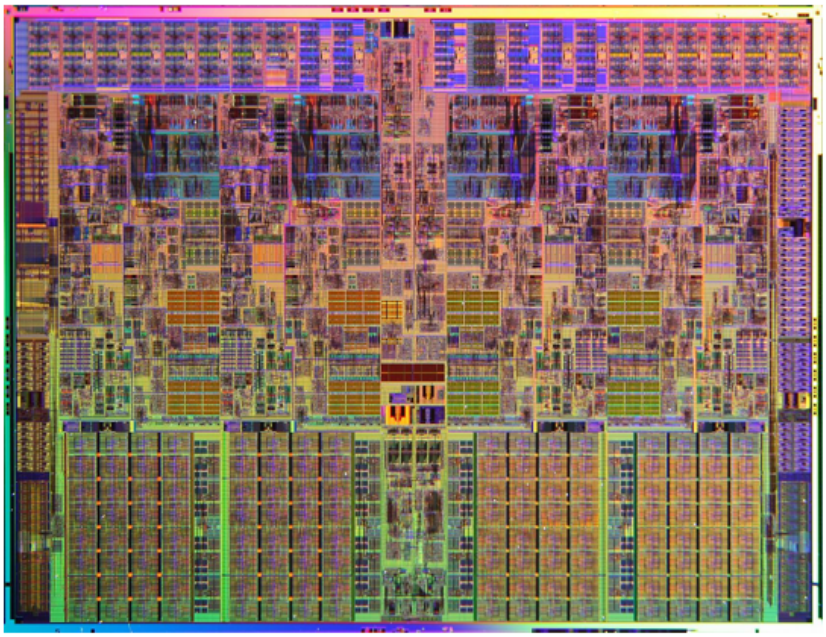
- Rows of gates – often identical in structure
- Connected to form customer specific circuits
- Can be full-custom (i.e. completely fabricated from scratch for a given design)
- Can be semi-custom (i.e. customisation on the metal layers only)
- Once made, design is fixed

PYKC 9 Oct 2017 MSc Lab – Mastering Digital Design Lecture 1 Slide 4

In early days of integrated circuits, designers started using rows of basic gates (shown as the dark stuff here arranged in rows). These are either completely customised (full-custom) or it is made with standard rows of gates but leaving the gates unconnected. For a specific design, the gates are connect through wires in the wiring channels. Therefore the customisation is only in the wiring metal layers and not the layers with transistors. This is known as “semi-custom” application-specific integrated circuits (ASICs).

Modern digital design – full custom IC

- Intel Core i7
- > ¾ billion trans.
- Very expensive to design
- Very expensive to manufacture
- Not viable unless the market is very large



PYKC 9 Oct 2017 MSc Lab – Mastering Digital Design Lecture 1 Slide 5

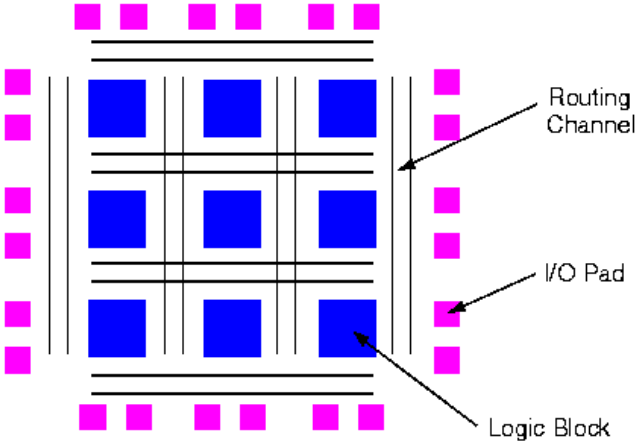
Of course you can also customise everything – each transistor and each wiring connect in a full-custom manner. Here is the layout of Intel i7 microprocessor (with 4 cores). Designing such a circuit is very expensive, highly risky, and once designed, it cannot be changed.

Most applications in electronic industry cannot afford to embark on such a design. This drives the rise of the Field Programmable Gate Array.

Field Programmable Gate Arrays (FPGAs)

- Combining idea from Programmable Logic Devices and gate arrays
- First introduced by Xilinx in 1985

- Arrays of logic blocks (to implement logic functions)
- Lots of programmable wiring in routing channels
- Very flexible I/O interfacing logic core to outside world
- Two dominant FPGA makers:
  - Xilinx and Altera
- Other specialist makers e.g. Actel and Lattice Logic



R1.1 p1 - p16

PYKC 9 Oct 2017 MSc Lab – Mastering Digital Design Lecture 1 Slide 6

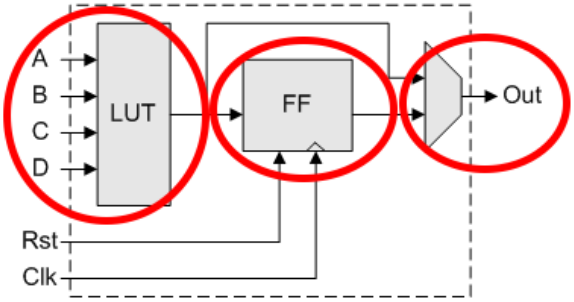
So what is an FPGA? You came across the idea of Programmable Logic Device in the first year, where the user can program what the logic gate does (be it a NAND or NOR or some form of SUM-of-PRODUCT implementation) or an adder, you as a user, can "program" the chip to perform that logic function. Now we can add another layer of user programmability – you can program how these logic gates are connected together! In that way, we have a general programmable logic chip. Unlike the microprocessor where the program is just the instruction to fix digital hardware, here you can program the hardware itself!

The first FPGA was introduced by Xilinx in 1985. It has arrays of logic blocks which are programmable. It is surrounded by PROGRAMMABLE ROUTING RESOURCES, which allows the user to define the interconnections between the logic blocks. It also has lots of very flexible input and output circuits (programmable for TTL, CMOS and other interface standards).

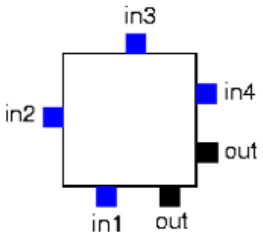
Nowadays, there are two major players in the FPGA domain: Xilinx and Altera (now part of Intel). These two domains 90% of the FPGA market with roughly equal share.

Configurable Logic Block (or Logic Element)

- Based around Look-up Tables (LUTs), most common with 4-inputs
- Optional D-flipflop at the output of the LUT
- 4-input LUT can implement ANY 4-input Boolean equation (truth-table)
- Special circuits for cascading logic blocks (e.g. carry-chain of a binary adder)



Logic Element (LE)



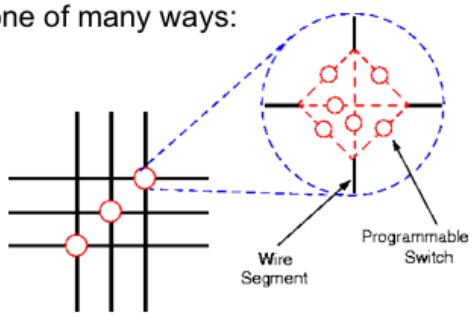
- Each logic block has pins located for easy access:

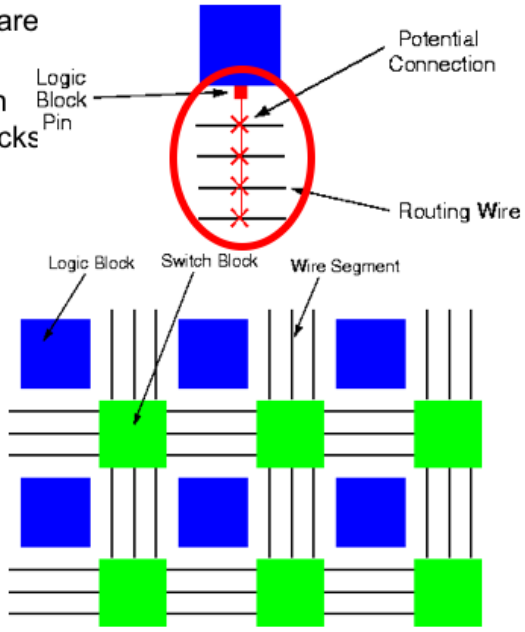
PYKC 9 Oct 2017 MSc Lab – Mastering Digital Design Lecture 1 Slide 7

Let us look inside an FPGA. Consider the logic block shown in blue in the last slide (Altera calls their logic block a **Logic Element** (LE)). Typically an LE consists of a 4-input Look-up Table (LUT) and a D-flipflop. Let us for now NOT to worry about how the 4-LUT is implemented internally. Just treat this as a 4-input combinatorial circuit which produces one output signal as shown here. The IMPORTANT characteristic is that the 4-LUT can be user defined (or programmable) to implement ANY 4-input Boolean function.
As we will see later, the lookup table is actually implemented with a bunch of multiplexers.

Programmable Routing

- Between rows and columns of logic blocks are wiring channels
- These are programmable – a logic block pin can be connected to one of many wiring tracks through a programmable switch
- Xilinx FPGAs have dedicated switch block circuits for routing (more flexible)
- Each wire segment can be connected in one of many ways:





PYKC 9 Oct 2017 MSc Lab – Mastering Digital Design Lecture 1 Slide 8

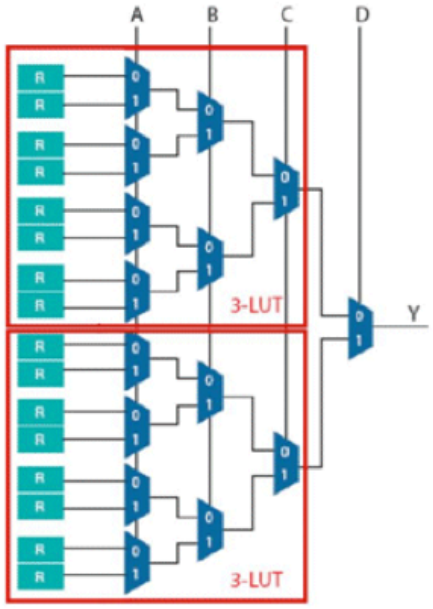
The Logic Elements are surrounded by lots of routing wires and interconnection switches. Typically a signal wire to the Logic Block or Logic Element can be connected to any of these wiring channels through a programmable connection (essentially a digital switch). Xilinx FPGAs also have dedicated switch blocks shown here. Horizontal and vertical wires can be connected through such as switch block with programmable switches (don't worry for now how that's done).

FPGAs have huge amount of these programmable resources and switches. Typically a very small percentage of these are being connected (i.e. ON) for a given application.

The main advantage and "power" of FPGA comes from the programmable interconnect – more so than the programmable logic.

The Idea of Configuring the FPGA

- Programming an FPGA is NOT the same as programming a microprocessor
- We download a BITSTREAM (not a program) to an FPGA
- This is known as CONFIGURATION
- All LUTs are configured using the BITSTREAM so that they contain the correct value to implement the Boolean logic
- Shown here is a typical implementation of a 4-LUT circuit
  - ABCD are the FOUR inputs
  - There is four level of 2-to-1 multiplexer circuits
  - The 16-inputs to the mux tree determine the Boolean function to be implemented as in a truth table
  - These 16 values are stored in registers (DFF)
  - Configuration = setting registers to 1 or 0



PYKC 9 Oct 2017 MSc Lab – Mastering Digital Design Lecture 1 Slide 9

Programming an FPGA is called "**configuration**". In programming a computer or microprocessor, we send to the computer instruction codes as '1's and '0''s. These are interpreted (or decoded) by the computer which will follow the instruction to perform tasks. The microprocessor needs to be fed these program codes continuously for it to function.

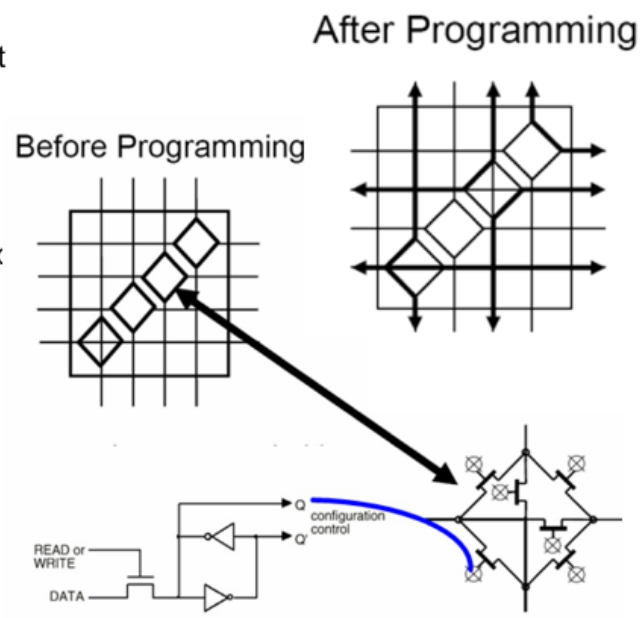
In FPGAs, you only need to **configure** the chip ONCE on power-up. You download to the chip a **BITSTREAM** (also bits in '1's and '0's), which determines the logic functions performed by the Logic Elements, and the interconnecting switches in order to connect the different LEs together to make up your circuit. Once the bitstream is received, the FPGA no longer needs to read the 1's and 0's again, very unlike a microprocessor which has to continually decoding the machine instructions. That's why we sometimes say that we **configure** an FPGA (instead of programming an FPGA, although the two words are used interchangeably).

What happens when you configure an FPGA? Let us consider the 4-input LUTs circuit. This is typically implement using a tree of four layers of 2-input to 1-output multiplexers. The entire circuit is behaving like a 16-to-1 multiplexer using the 4 inputs ABCD as the control of the MUX tree. For example, if ABCD = 0000, then the top most input of the MUX is routed to Y output.

In this way, ABCD forms the input columns of a truth table. For 4-inputs, the truth table has 16 entries. The output Y for each of the truth table entry corresponds to the input of the MUX. Configuration involves fixing the inputs to the 16-to-1 MUX by storing '1' or '0' in the registers R. Changing the 16 values stored, you can change to truth-table to anything you want.

Configuring the routing in an FPGA

- At each interconnect site, there is a transistor switch which is default OFF (not conducting)
- Each switch is control by the output of a 1-bit configuration register
- Configuring the routing is simply t put a '1' or '0' in this register to control the routing switches
- Bitstream is either stored on local flash memory or download via a computer
- Configuration happens on power up



PYKC 9 Oct 2017 MSc Lab – Mastering Digital Design Lecture 1 Slide 10

To configure the programmable routing, let us look at how the routing circuit works. Take Xilinx SWITCH BLOCK circuit (green blocks in slide 7). This block controls the connections between four horizontal channels and four vertical channels. The diamond shaped block is a potential interconnect site. Inside the switch block circuit, there are 6 transistor switches which are initially all OFF (or open circuit).

The gate input of EACH switch is controlled by the output of a 1-bit register (e.g. a 1-bit D-FF). If the register stores a '1', the routing transistor will have its gate driven high. Since the transistor is an nMOS transistor, it will become conducting. In this way, configuring the routing resources simply means that the correct '1's and '0's are stored in the registers that control these routing transistors.

As you would expect, typically an FPGA would have hundreds of thousands of these routing switches, most of these are OFF. Once programmed, the interconnections are made. The bold lines in the diagram above (after programming) shows the programmed connections.

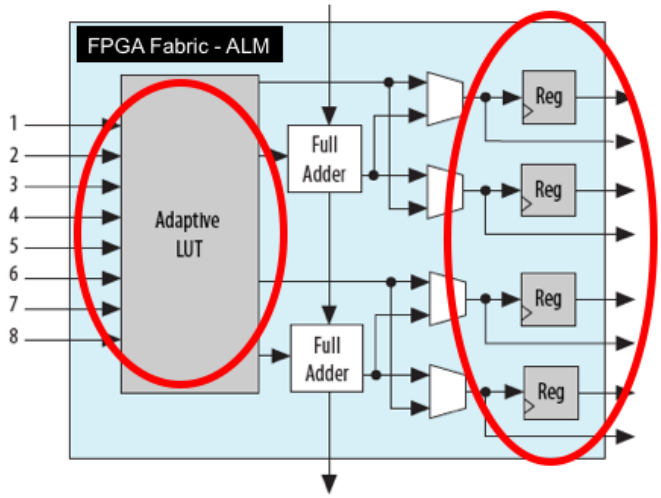
Bitsteam information used for configuration purpose are usually stored on a flash memory chip, which is download to the FPGA during power-up – similar to "booting up a computer". Once this is done, the FPGA is progammed to perform a specific user function (e.g. your design in the VERI experiment).

Alternatively you can send the bitstream to the FPGA via a computer connection to the chip. On the DE1-SOC board, it does both. Powerup DE1 will configure the Cyclone V FPGA chip to a "waiting" mode, which makes the DE1 board talk to the computer via the USB port while flashing the lights ON and OFF. You then send to the board a bitstream of your design via the USB port.

Cyclone V's Adaptive Logic Module (ALM)

- We use Altera's Cyclone V FPGA on this course
- It uses a more complex FPGA logic fabric known as Adaptive Logic Module (ALM)
- The device we use (5CSEMA5F31C6N) has 32,000 ALMs on one chip

- The logic element is more advanced than the original 4-LUT architecture
- The ALM can implement much larger logic functions, or can be broken into a number of smaller units



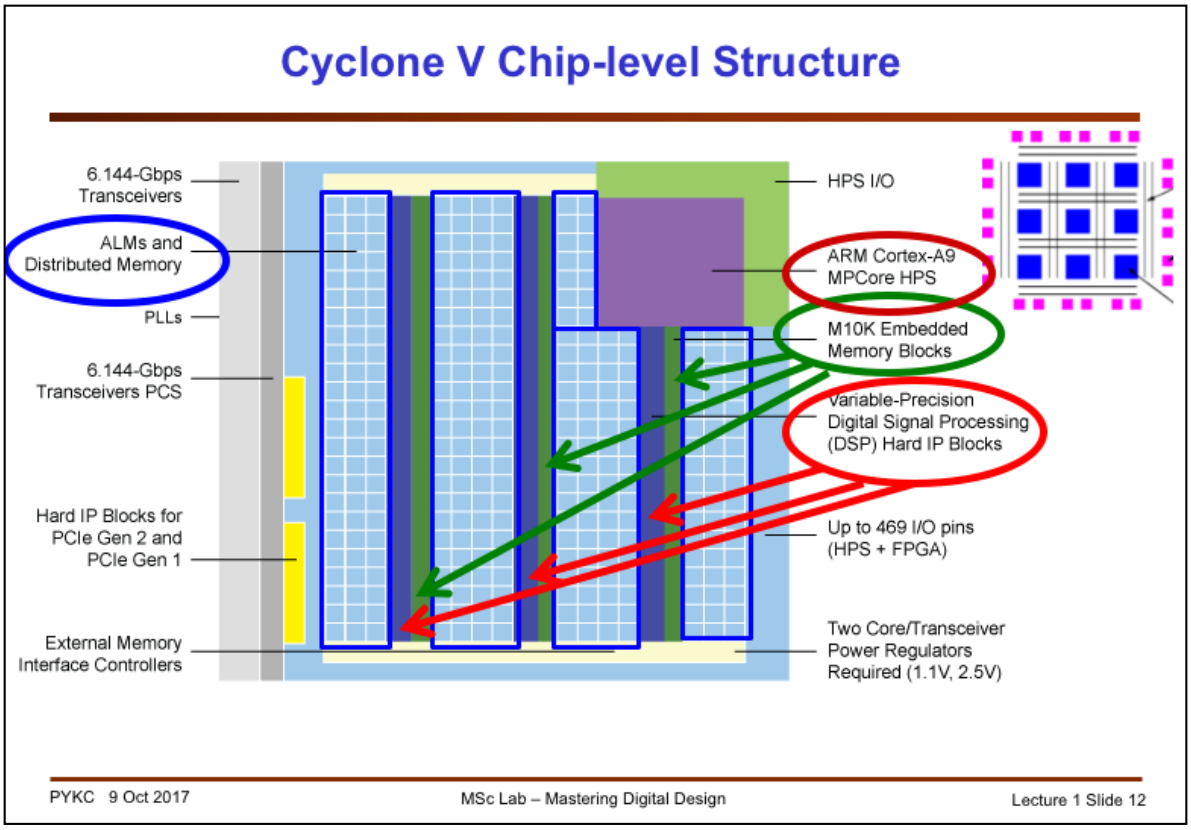
PYKC 9 Oct 2017 MSc Lab – Mastering Digital Design Lecture 1 Slide 11

Let us now look at the FPGA that you will use for this course. The Altera Cyclone V FPGA has a more advanced programmable logic element than the simple 4-input LUT that we have considered up to now. The call this a Adaptive Logic Module or ALM.

An ALM can take up to 8 Boolean input signals and produces four outputs with or without a register. Additionally, each ALM also can perform the function of a 2-bit binary full adder.

As a user of the Cyclone V FPGA, you don't actually need to worry too much about exactly how the ALM is configured to implement your design. The CAD software will take care of the mapping between your design and the physical implementation using the ALMs. It is however useful to know that as the technology evolves, more and more complicated programmable logic elements are being developed by the manufacturers in order to improve the area utilization of the FPGAs.

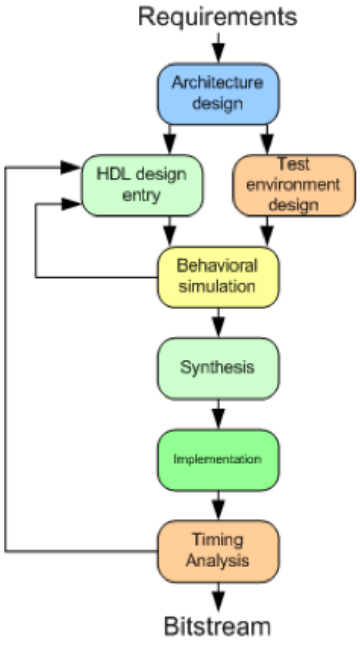
The Cyclone V on the DE1-SOC board has 32,000 ALMs, which could be estimated to be equivalent to 85K+ the old style LEs. Putting this in context, you could put onto this one chip 2,000 32-bit binary adder circuits!



The Cyclone V is much more than just an FPGA with a bunch of Logic Elements (or ALMs). Our chip in the DE1-SOC board has 32,000 ALMs, which is around 85K old style 4-input LUT LEs. On top of that, it also has over 4Mbit of embedded memory, 87 DSP blocks (to do multiply-accumulate operations needed for signal processing), and even a dual-core ARM microprocessor!

It has hard-logic to implement PCIe interface (to fast peripherals) and external memory interface to connect to external memory. It is a truly powerful chip onto which one could implement an entire digital electronic system. Therefore Altera call this Cyclone V System-on-Chip (SoC).

Design Tools – Altera Quartus II



- Quartus II – a comprehensive design tools for Altera FPGAs
- Special web edition free to download from (need registration):
  - http://dl.altera.com/?edition=lite
  - Features include (see introduction to Quartus II):
    - design entry
    - compilation from Hardware Description Languages (HDL)
    - synthesis
    - simulation
    - timing analysis
    - power analysis
    - project management

R1.3

PYKC 9 Oct 2017 MSc Lab – Mastering Digital Design Lecture 1 Slide 13

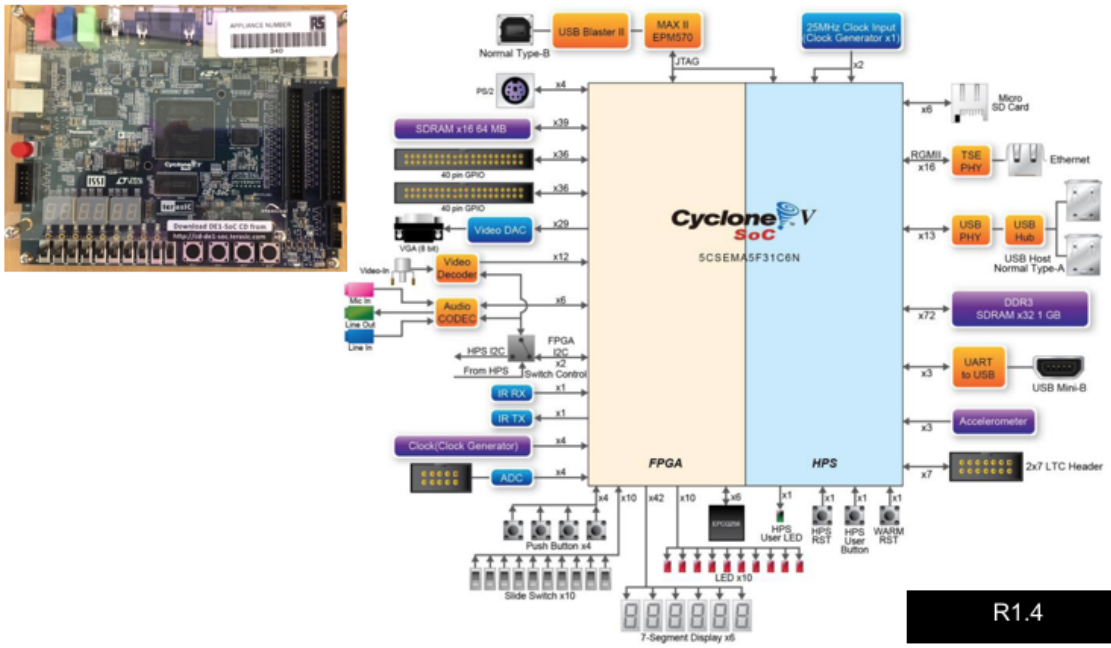
For this lab-based module, you will be designing circuits using the free version of the design suite known as **Quartus Prime Lite** from Intel/Altera. You can download your own copy onto your notebook machine, or you can use the versions that are installed in any PCs located anywhere in the department.

This very powerful design tool contains everything you need to design a complex digital system ON YOUR OWN COMPUTER! However, the software only runs on either a MS Windows or a Linux operating system. If you are using a Mac, you would need to run a Virtual Machine applications (such as Virtual Box) and install Windows or Linux before installing Quartus II software.

Beware that the software is very large – you need to have several GB of free disk space. The minimum required RAM is 4GB, and 8GB is recommended.

If your laptop is suitable, do download this software and play with it at home.

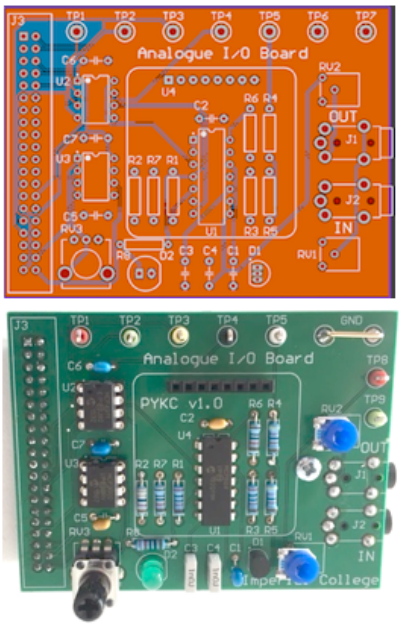
DE1-SOC Board



This slide shows you the functional blocks of the DE1-SoC board. This has everything you need test basic designs involving switches, 7-segment displays and even a VGA output.

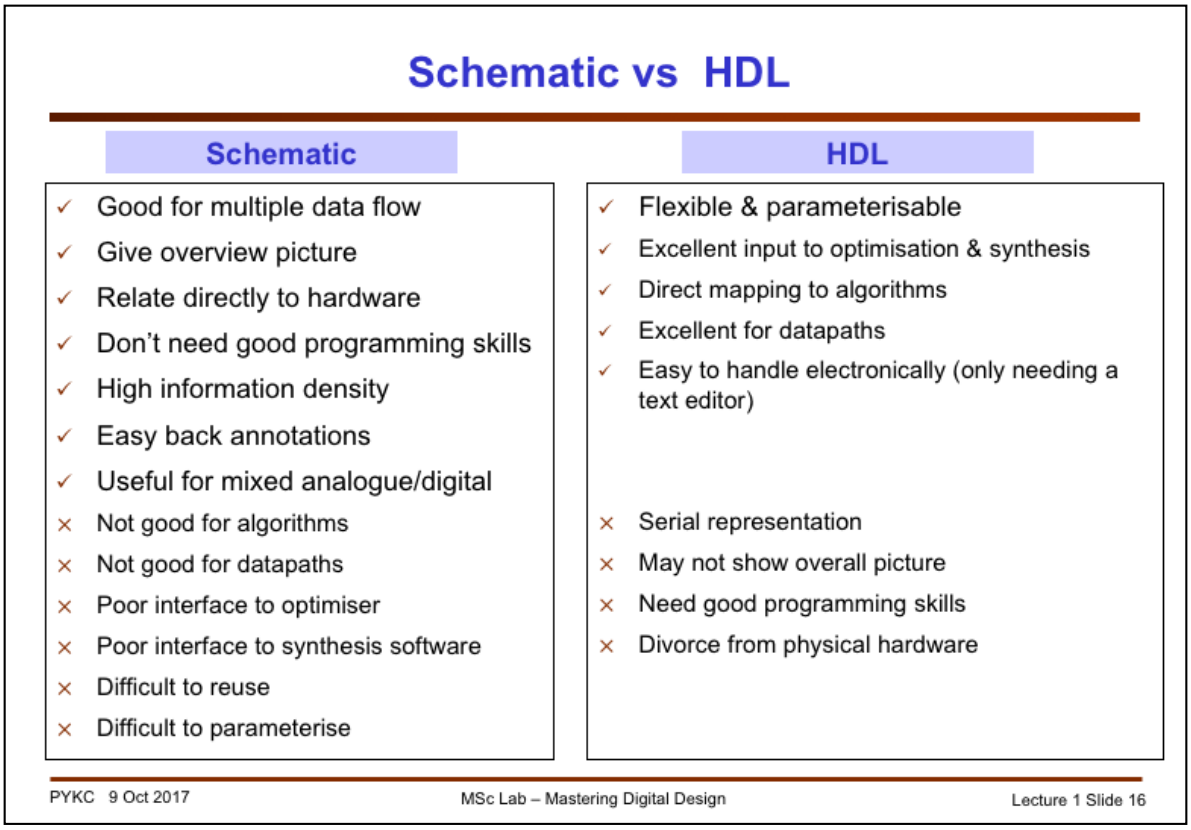
Add-on Board

- Provides analogue inputs and outputs
- Contains 2 channels ADC, one from microphone & one from a socket
- Has 2 channel analogue output with one driven by a DAC and another by a digital signal
- Includes built-in filter and operational amplifier
- Will be using this board in Part 3 & 4 of this Lab Experiment



PYKC 9 Oct 2017 | MSc Lab – Mastering Digital Design | Lecture 1 Slide 15

I also provide a purpose-built ADC/DAC board to support the lab experiment. This add-on board in only needed in week 3 onwards during the laboratory sessions. So for now, you can ignore it.



You are very familiar with schematic capture. In the first year, you used SPICE for simulating analogue circuits and Quartus II for your digital experiment where you create schematic circuits with gates.

However modern digital design methods in general DO NOT use schematics. Instead an engineer would specify the design requirement or the algorithm to be implemented in some form of computer language specially designed to describe hardware. These are called "Hardware Description Languages" (HDLs).

The most important advantages of HDL as a means of specifying your digital design are: 1) You can make the design take on parameters (such as number of bits in an adder); 2) it is much easier to use compilation and synthesis tools with a text file than with schematic; 3) it is very difficult to express an algorithm in diagram form, but it is very easy with a computer language; 4) you can use various datapath operators such as +, * etc.; 5) you can easily edit, store and transmit a text file, and much hardware with a schematic diagram.

For digital designs, schematic is NOT an option. Always use HDL. In this lecture, I will demonstrate to you why with an example.

Verilog HDL

- Similar to C language to describe/specify hardware
- Description can be at different levels:
  - **Behavioural level**
  - **Register-Transfer Level (RTL)**
  - **Gate Level**
- Not only a specification language, also with associated **simulation environment**
- Easier to learn and "lighter weight" than its competition: VHDL (3rd year optional course)
- Very popular with chip designers

- For this course, we will:
  - Learn through examples and practical exercises
  - Use two examples: 2-to-1 multiplexer and 7 segment decoder

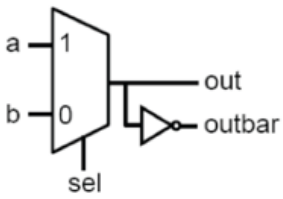
PYKC 9 Oct 2017 | MSc Lab – Mastering Digital Design | Lecture 1 Slide 17

I have chosen to use Verilog HDL as the hardware description language for the course. Verilog is very similar to the C language, which you should already know from your first year course. However, you must always remember that YOU ARE USING IT TO DESCRIBE HARDWARE AND NOT A COMPUTER PROGRAMME.

You can use Verilog to describe your digital hardware in three different level of abstraction:
1) Behavioural Level – you only describe how the hardware should behave without ANY reference to digital hardware.
2) Register-Transfer-Level (RTL) – Here the description assume the existence of registers and these are clocked by clock signal. Therefore digital data is transferred from one register to the next on successful clock cycles. Timing (in terms of clock cycles) is therefore explicitly defined in the Verilog once. This is the level of design we use on this course.
3) Gate Level – this is the low level description where each gate is described and how these are connected together is specified.
Verilog is not only a specification language which tells the CAD system what hardware is suppose to do, it also includes a complete simulation environment. A Verilog compiler does more than mapping your code to hardware, it also can **simulate** (or execute) your design to predict the behaviour of your circuit. It is the predominant language used for chip design.

Structure of a Module

- Verilog design contains **interconnected modules**
- A module has collections of low-level gates, statements and other modules
- Here is an example of a simple module that describes a 2-to-1 multiplexer:



sel	out
0	b
1	a

```
// Function: 2-to-1 multiplexer
module mux2to1 (out, outbar,
                a, b, sel);

    output  out, outbar;
    input   a, b, sel;

    assign  out = sel ? a : b;
    assign  outbar = ~out;

endmodule
```

- // to end-of-line is comment. Can also use /* … */ for multiline comments
- Declare and name module; list its ports; terminate with ';'
- Specify port as input, output (or inout if bidirectional)
- Express modules behaviour; each statement executes in parallel; ORDER DOES NOT MATTER

PYKC 9 Oct 2017 MSc Lab – Mastering Digital Design Lecture 1 Slide 18

This is a Verilog code module that specifies a 2-to-1 multiplexer. It is rather similar to a C function (except for the **module** keyword).

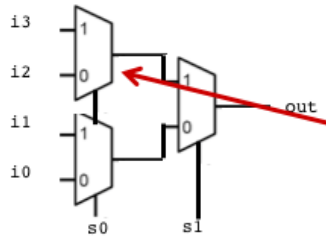
It is important to remember the basic structure of a Verilog module. There is a module name: mux2to1. There is a list of interface ports: 3 inputs a, b and sel, and 2 outputs out and outbar. Always use meaningful names for both module name and variable names.

You must specify which port is input and which port is output, similar to the data type declaration in a C programme.

Finally, the 2-to-1 multiplexing function is specified in the assign statement with a construct that is found in C. This is a **behavioural** description of the multiplexer – no gates are involved.
The last statement specifies the relationship between `out` and `outbar`. It is important to remember that Verilog describes HARDWARE not instruction code. The two **assign** statements specify hardware that "execute" or perform the two hardware functions in parallel. Therefore their **order does not matter**.

Continuous Assignment

```
module mux2to1 (out, outbar,
                a, b, sel);

    output  out, outbar;
    input   a, b, sel;

    assign  out = sel ? a : b;
    assign  outbar = ~out;

endmodule
```



- Keyword assign specifies **continuous assignment** to describe combinational logic
- Right-hand expression continuously evaluated …. responding to input change immediately
- Left-hand is a net driven with evaluated value
- Left side must be a scalar, a net or a concatenation of nets and vector nets. (nets, vectors etc. described later)
- All continuous assignments execute in parallel
- Operators in expressions are low-level:
  - Conditional assignment: (cond)? vTrue: vFalse
  - Boolean: ~, &, |,
  - Arithmetic: +, -, *
- Operators can be nested. For example, here is a 4-to-1 mux:

```
assign   out = s1 ? (s0 ? i3 : i2) : (s0 ? i1 : i0);
```

PYKC 9 Oct 2017 | MSc Lab – Mastering Digital Design | Lecture 1 Slide 19

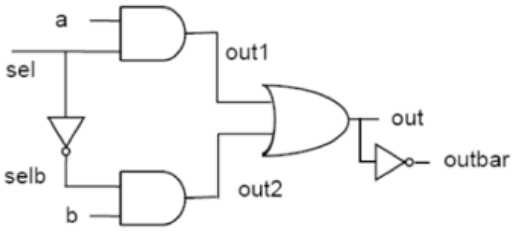
Continuous assignment specifies combinational circuits – output is continuously reflecting the operations applied to the input, just like hardware.
Remember that unlike a programming language, the two continuous assignment statements here ARE specifying hardware in PARALLEL, not in series.
Here we also see the conditional assignment statement that is found in C. This maps perfectly to the function of a 2-to-1 multiplexer in hardware and is widely used in Verilog.
Furthermore, there are many other Boolean and arithmetic operators defined in Verilog (as in C). Here is a quick summary of all the Verilog operators (used in an expression).

{}, {{}}	concatenation
+ - * /	arithmetic
%	modulus
> >= < <=	relational
!	logical negation
&&	logical and
||	logical or
==	logical equality
!=	logical inequality
===	case equality
!==	case inequality
~	bit-wise negation
&	bit-wise and
|	bit-wise inclusive or
^	bit-wise exclusive or
^~ or ~^	bit-wise equivalence

&	reduction and
~&	reduction nand
|	reduction or
~|	reduction nor
^	reduction xor
~^ or ^~	reduction xnor
<<	left shift
>>	right shift
? :	condition
or	event or

Description at Gate Level

```
// Function: 2-to-1 multiplexer as gates

module mux_gate (out, outbar,
                 a, b, sel);

    output  out, outbar;
    input   a, b, sel;
    wire    out1, out2, selb;

    not i1 (selb, sel);
    and a1 (out1, a, sel);
    and a2 (out2, b, selb);
    or o1 (out, out1, out2);
    not i2 (outbar, out);

endmodule
```



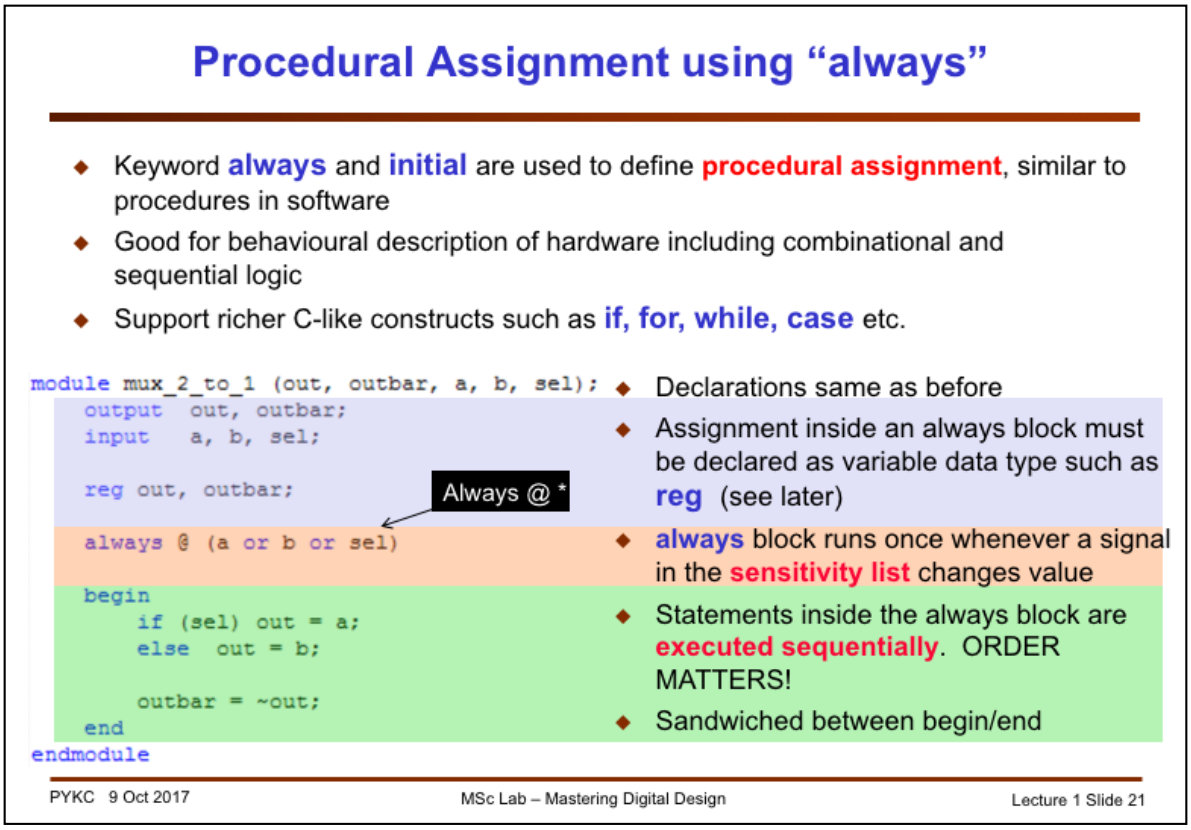
- Built-in logic gate primitives:
  - and, nand, or, nor, xor, xnor, not, buf
- Can use arbitrary number of inputs, e.g. and gate_name (out, in1, in2, in3, …)
- Tri-state buffers: bufif1 and bufif0
- Connect gates with nets using declaration keyword wire
- **and a1 (out1, a, sel);**
- This is called an **instantiation** of an AND gate. **a1** is the name of THIS particular AND gate.

While the previous Verilog code for the 2-to-1 mux only specify “behaviour”, here is one that specify a gate implementation of the same circuit. Three types of gates are used: and, or an not gates. There are internal nets (declared as wire) which take on any names.

Keywords such as **and**, **or** and **xor** are special – they specify actual logic gates. They are also special in that the number of inputs to the and-gate can be 2, 3, 4, ….. Any length!

Note that this module uses TWO AND gates, and they have different names: a1 and a2. There are TWO separate instances of the AND gate. For software functions, “calling” a function simple execute the same piece of programme code. Here the two lines “and a1 (out1, …” and “and a2 (out2 …” produce two separate piece of hardware. We say that each line is “instantiating” an AND gate.

Wiring up the gates is through the use of ports and wires, and depends on the positions of these “nets”. For example, out1 is the output net of the AND gate a1, and it is connect to the input of the OR gate o1 by virtual of its location in the gate port list.



So far we have used Verilog in a very hardware specific way. "**assign**" and using gate specification are special to Verilog.

Here is something that is more like C – and it is called "procedural assignment". Typically we use something called "always" block to specify a "procedure" or collection of sequential statements which are sandwiched between begin-end construct.

The always block needs a sensitive list – a list of signals such that if ANY of these signal changes, the always block will be invoked. You may read this block as:

"always at any changes in nets a, b or sel, do the bits between **begin** and **end**"
Actually, if you are defining a combinational circuit module, an even better way to define the always block is to use:
….. always @ * …. // always at any change with any input signals

Inside the begin-end block, you are allowed to use C-like statements. In this case, we use the if-else statement. All statements inside the begin-end block are **executed sequentially**.

Verilog "register" is NOT what it appears!

- Registers normally represents storage elements in digital logic, they need clock signals to update their output value
- In Verilog **reg** are NOT the same as digital registers, it is used only to declare a variable that holds a value
- Values of variables (declared as **reg**) can be changed anytime in a simulation, and can be used for nets of a combinational circuit
- In other words, in Verilog, **reg** is similar to declarations such as **int, real** etc.

Note that Verilog keyword **reg** does not implies that there is a register created in the hardware. It is much more like declaring a variable that holds a value. It is a rule in Verilog that **assignment INSIDE an always block MUST be declared reg**, and NOT a net (**wire**). This is one of the few peculiarities of Verilog that can be confusing to students.

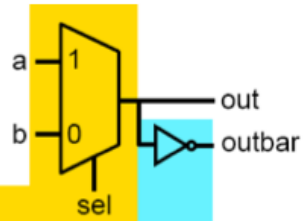
Mixing procedural & continuous assignments

- Procedural and continuous assignments can co-exist within a module
- In procedural assignments, the value of variables declared as **reg** are changed only once when the procedural block is invoked by changes in the sensitivity list
- In continuous assignments, the right-hand expression is constantly evaluated and the left-side net is updated all the time

```
module mux_2_to_1(a, b, out,
                  outbar, sel);
  input a, b, sel;
  output out, outbar;
  reg out;

  always @ (a or b or sel)
  begin
    if (sel) out = a;
    else out = b;
  end

  assign outbar = ~out;

endmodule
```

procedural description

continuous description



This slide shows how the procedural statement is mapped to the basic MUX circuit. The continuous assignment statement corresponds to the NOT gate.

case statement – better alternative to if-else

- **case** statement can often replace **if-else** construct within an **always** block, and provides better abstraction
- Here is an example using the mux_2_to_1 module:

```
always @ (a or b or sel)
begin
    case (sel)
      1'b0: out = b;
      1'b1: out = a;
    endcase
end
```

```
if (sel) out = a;
else  out = b;
```

Notation for numbers:
<size> ‘ <base> <number>

2’b10	2 bit binary, v=2
‘b10	Unsized binary 32-bit, v=2
31	Unsized decimal, v=31
8’hAf	8-bit hex, v=175
-16’d47	16-bit negative decimal, v=-47

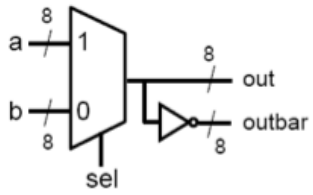
PYKC 9 Oct 2017 MSc Lab – Mastering Digital Design Lecture 1 Slide 24

This is yet another way to specify the MUX circuit. It is still a procedural assignment with the “always” block. However, we replace the if-else statement with a “**case**” statement. The case variable is sel. Since sel is a 1-bit signal (or net), it can only take on 0 or 1.

Note that the various case values can be expressed in different number formats as shown in the slide. For example, consider 2’b10. The 2 is the number of bits in this number. ‘b means it is specified in binary format. The value of this number is 10 in binary.

n-bit signals - buses

- Verilog is powerful in specifying multi-bit signals and buses.
- Here is an example for 8-bit wide 2-to-1 multiplexer:




```
module mux_2_to_1(a, b, out,
                  outbar, sel);
  input[7:0] a, b;
  input sel;
  output[7:0] out, outbar;
  reg[7:0] out;
  always @ (a or b or sel)
  begin
    if (sel) out = a;
    else out = b;
  end
  assign outbar = ~out;
endmodule
```

Concatenate signals using the { } operator

```
assign {b[7:0],b[15:8]} = {a[15:8],a[7:0]};
```

effects a byte swap

PYKC 9 Oct 2017 | MSc Lab – Mastering Digital Design | Lecture 1 Slide 25

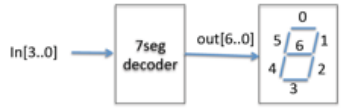
This slide demonstrates why language specification of hardware is so much better than schematic diagrams. By simple declaring the signals as a multi-bit bus (8 bits [7:0]), we change this module to one that specifies 8 separate 2-to-1 multiplexers.

Another useful way to specify a bus is using the concatenation operator: { …. } as shown above.

The concatenation operator is particularly useful in converting digital signals from one word length (i.e. number of bits in a word) to another. For example, to convert an 8-bit unsigned number a[7:0] to a 13-bit unsigned number b[12:0], you can simple do this:

```
assign   b[12:0] = {5'b0, a[7:0]};
```

Putting everything together – 7 seg decoder



In3 : in2

In1 : in0 \ out6	00	01	11	10
00	1	0	1	0
01	1	0	0	0
11	0	1	0	0
10	0	0	0	0

in[3..0]	out[6:0]	Digit
0000	1000000	0
0001	1111001	1
0010	0100100	2
0011	0110000	3
0100	0011001	4
0101	0010010	5
0110	0000010	6
0111	1111000	7

in[3..0]	out[6:0]	Digit
1000	0000000	8
1001	0010000	9
1010	0001000	A
1011	0000011	b
1100	1000110	C
1101	0100001	d
1110	0000110	E
1111	0001110	F

out6 = /in3*/in2*/in1 + in3*in2*/in1*/in0 + /in3*in2*in1*in0

out5 = /in3*/in2*in0 + /in3*/in2*in1 + /in3*in1*in0 + in3*in2*/in1*in0

out4 = /in3*in0 + /in3*in2*/in1 + in3*/in2*/in1*in0

out3 = /in3*in2*/in1*/in0 + /in3*/in2*/in1*in0 + in2*in1*in0 + /in2*in1*/in0

out2 = /in3*/in2*in1*/in0 + in3*in2*/in0 + in3*in2*in1

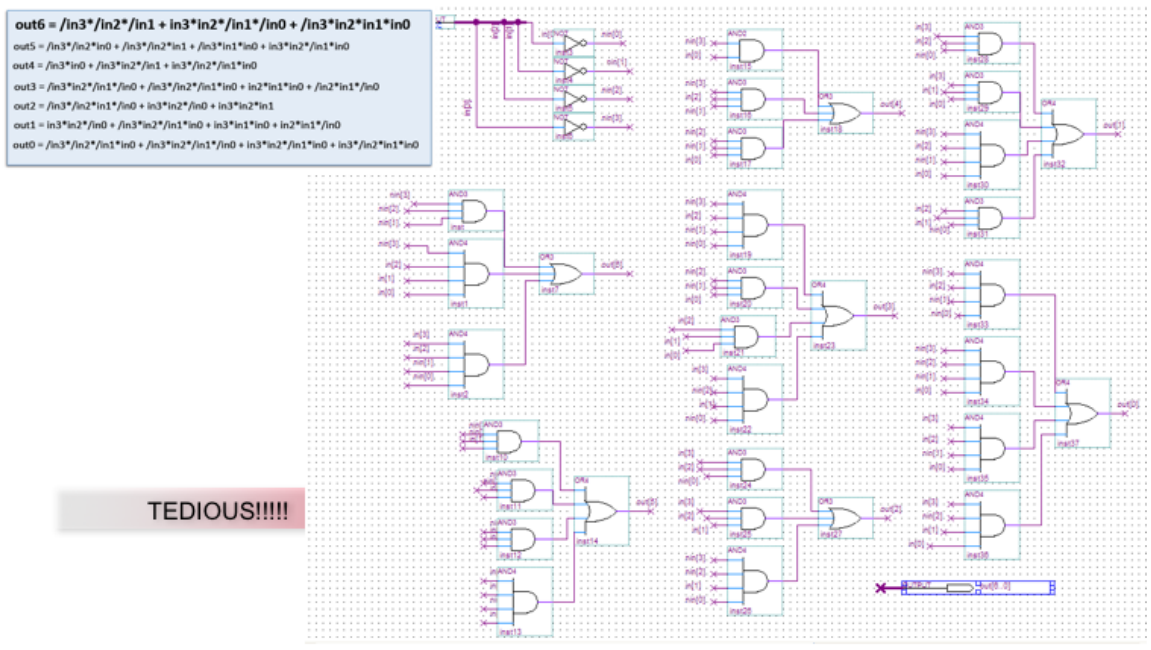
out1 = in3*in2*/in0 + /in3*in2*/in1*in0 + in3*in1*in0 + in2*in1*/in0

out0 = /in3*/in2*/in1*in0 + /in3*in2*/in1*/in0 + in3*in2*/in1*in0 + in3*/in2*in1*in0

Here is a simple example: the design of a 4-bit hex code to 7 segment decoder. You can express the function of this 7-segment decoder in three forms: 1) as a truth table (note that the segments are low active); 2) as 7 separate K-maps (shown here is for out6 segment only); 3) as Boolean equations.

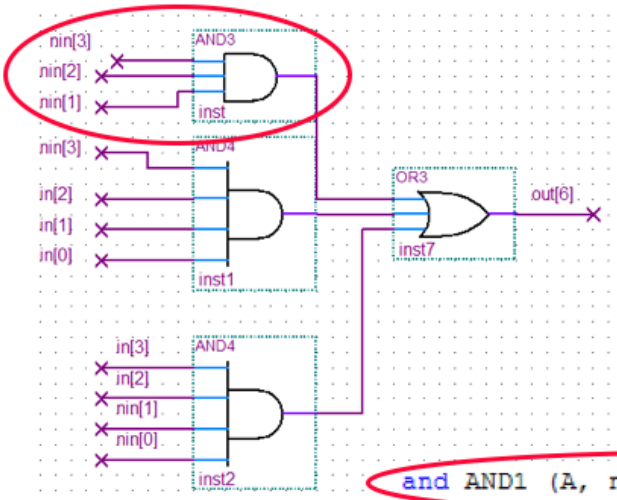
This is probably the last time you see K-maps. In practical digital design, you would rely heavily on CAD tools. In which case, the logic simplifications are done for you automatically – **you never need to use K-maps to do that manually**!

Method 1: Schematic Entry Implementation



Here is a tedious implementation in the form of schematic of interconnected gates. Very hard to do and very prone to errors.

Method 2: Use primitive gates in Verilog



Equally TEDIOUS!!!!!

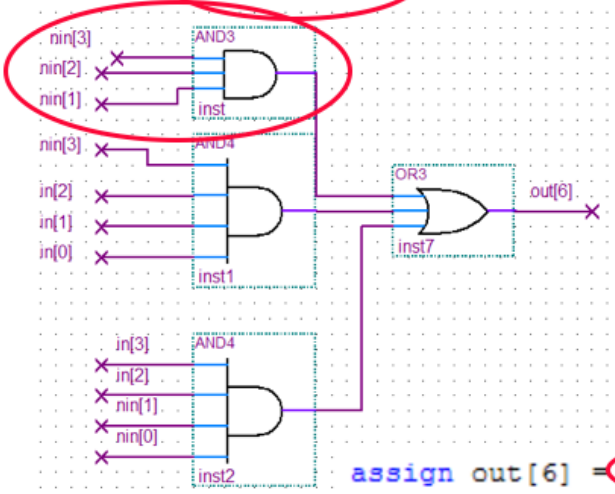
Direct mapping of gates to primitives

```
and AND1 (A, nin[3], nin[2], nin[1]);
and AND2 (B, nin[3], in[2], in[1], in[0]);
and AND3 (C, in[3], in[2], nin[1], nin[0]);
or  OR1  (out[6], A, B, C);
```


One could take a group of gates and specify the gates in Verilog gate primitives such as and, or etc. Still very tedious. Here is the implementation for the out6 output.

Method 3: Use continuous assignment in Verilog

out6 = /in3*/in2*/in1 + in3*in2*/in1*/in0 + /in3*in2*in1*in0

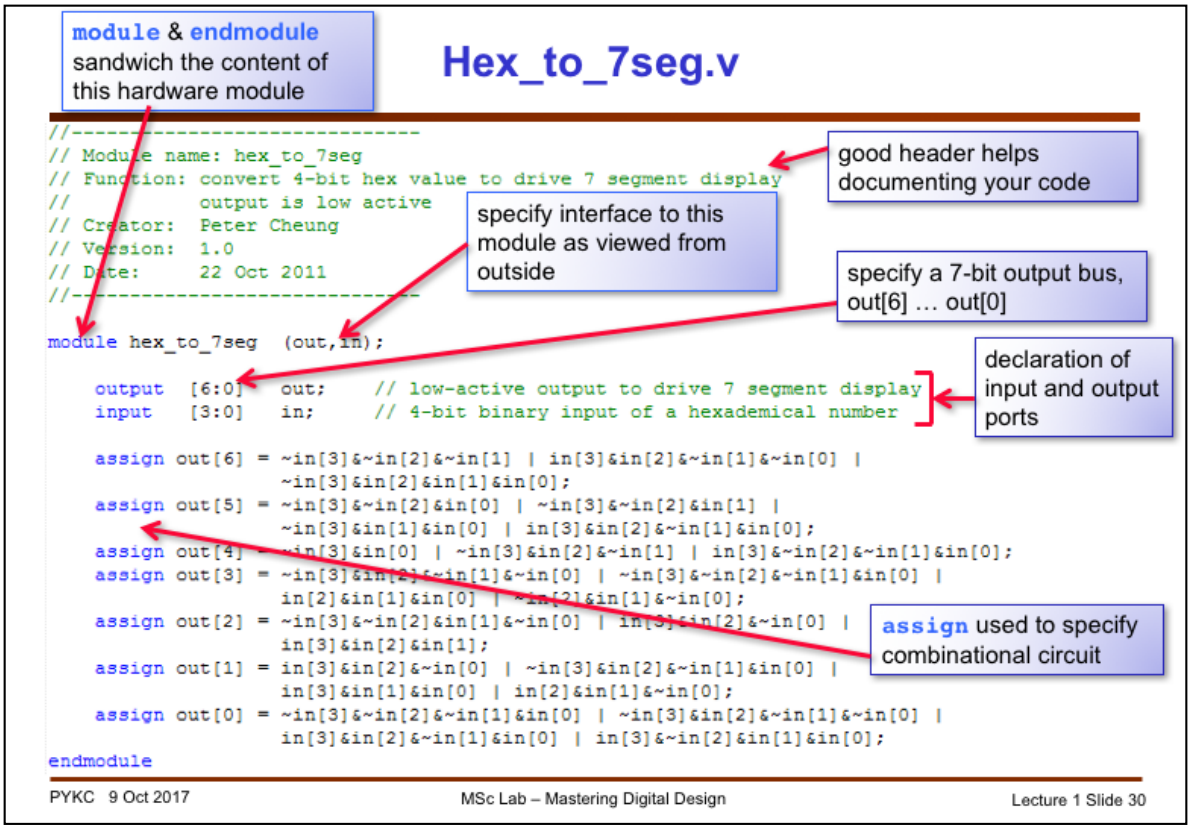


Much Better?

Direct mapping of Boolean equation using continuous assignment..

```
assign out[6] = ~in[3]&~in[2]&~in[1] |
                in[3]&in[2]&~in[1]&~in[0] |
                ~in[3]&in[2]&in[1]&in[0];
```


Instead of specifying each gate separately, here is using continuous assignment statement, mapping the Boolean equation direction to a single Verilog statement. This is better.



Here is the complete specification of the hex_to_7seg module using continuous assignment. It shows how one should write Verilog code with good comments and clear documentation of input and output ports.

Method 4: Power of behavioural abstraction

```
module hex_to_7seg  (out,in);

    output  [6:0]   out;    // low-active output to
    input   [3:0]   in;     // 4-bit binary input o

    reg     [6:0]   out;    // make out a variable

    always @ (in)
      case (in)
        4'h0: out = 7'b1000000;
        4'h1: out = 7'b1111001;     // -- 0 ---
        4'h2: out = 7'b0100100;     // |       |
        4'h3: out = 7'b0110000;     // 5       1
        4'h4: out = 7'b0011001;     // |       |
        4'h5: out = 7'b0010010;     // -- 6 ---
        4'h6: out = 7'b0000010;     // |       |
        4'h7: out = 7'b1111000;     // 4       2
        4'h8: out = 7'b0000000;     // |       |
        4'h9: out = 7'b0011000;     // -- 3 ---
        4'ha: out = 7'b0001000;
        4'hb: out = 7'b0000011;
        4'hc: out = 7'b1000110;
        4'hd: out = 7'b0100001;
        4'he: out = 7'b0000110;
        4'hf: out = 7'b0001110;
      endcase
endmodule
```

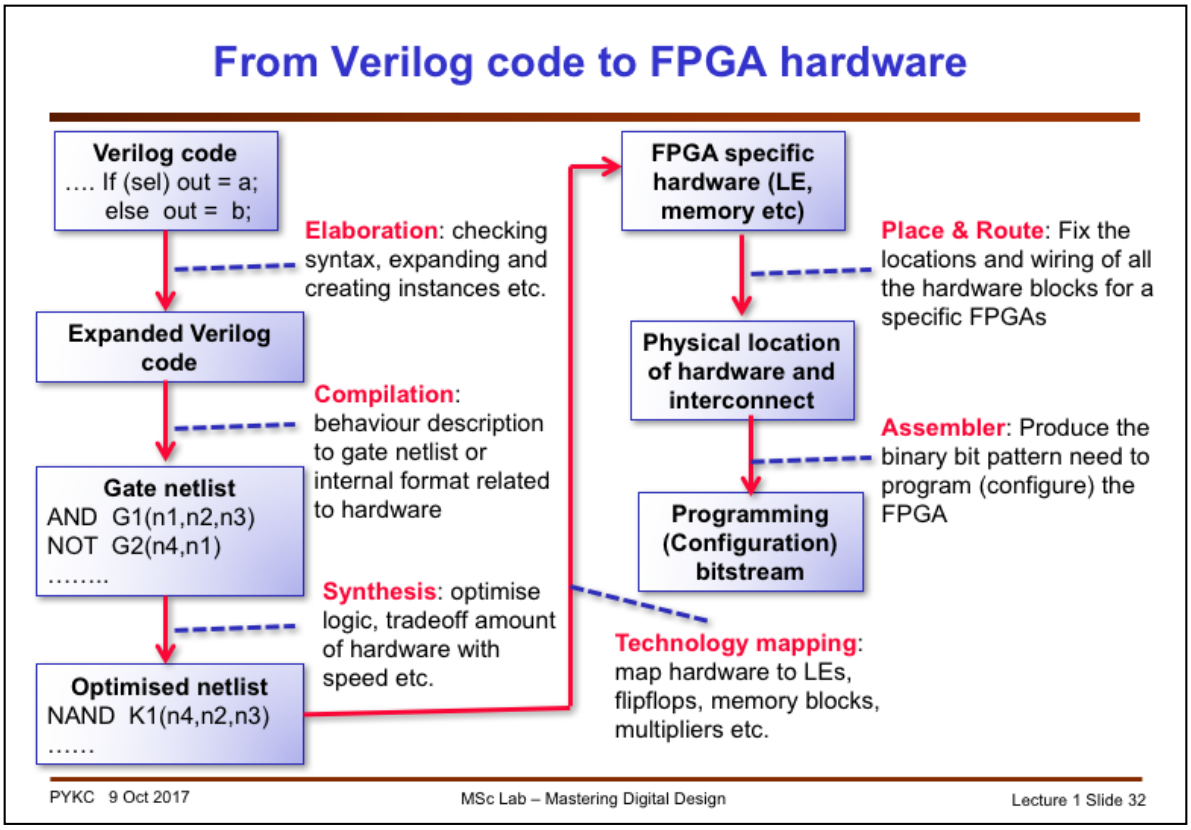
BEAUTIFUL !!!

- Direct mapping of truth table to case statement
- Close to specification, not implementation

in[3..0]	out[6:0]	Digit
0000	1000000	0
0001	1111001	1
0010	0100100	2
0011	0110000	3
0100	0011001	4
0101	0010010	5
0110	0000010	6
0111	1111000	7
1000	0000000	8
1001	0010000	9
1010	0001000	A
1011	0000011	b
1100	1000110	C
1101	0100001	d
1110	0000110	E
1111	0001110	F

Finally the 4th method is the best. We use the case construct to specify the behaviour of the decoder. Here one directly maps the truth table to **the case statement** – easy and elegant.

Instead of using: always @ (in), you could also use always @*



How is a Verilog description of a hardware module turned into FPGA configuration? This flow diagram shows the various steps taken inside the Quartus II CAD system.

For the Lab Experiment, you will be working in pairs. In other to ensure that you get to know each other, I have randomly paired you together with someone else as Lab Partner for this module.

Group	Name
1	Mr A. Dworniczek
1	Mr J. Cheng
2	Miss R. Ng
2	Miss S. Dai
3	Ms V. Rasalingam
3	Mr S. Yu
4	Mr X. Zhang
4	Mr J. Zhang
5	Mr S. Pournias
5	Miss J. Huang

Group	Name
6	Mr P. Tripathi
6	Mr L. Bu
7	Mr M. Simpson
7	Miss J. Chen
8	Mr J. Xie
8	Miss Z. Jiang
9	Miss C. Alvarado
9	Miss L. Yuan
10	Mr Y. Weng
10	Mr Y. Zhu
11	Mr J. Shen
11	Mr J. Huang