Imperial College London

MSc in Analogue & Digital IC Design

Mastering Digital Design …
… in SystemVerilog with FPGAs

Peter Cheung
Department of Electrical & Electronic Engineering
Imperial College London

Course webpage: https://github.com/Mastering-Digital-Design/Lab-Module/
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 following Github page:

https://github.com/Mastering-Digital-Design/Lab-Module/

You should bookmark this URL for easy access.

Aims, Objectives and Assessment

1. To ensure all students on the MSc course **reaches a common competence** level in RTL design using FPGAs in a hardware description language;
2. To act as **revision exercise** for those who are already competent in Verilog and FPGA.

Format:

- All lectures by me – to teaching you something or to go over materials in the previous lab session
- Lab Experiment in four parts by Dr Aaron Zhao and GTAs - Each experiment should take 1 week, and each has very clearly defined Learning Outcomes.

Assessment:

1. One-to-one interview at the end of the Autumn Term.
2. Part of the Coursework component of the MSc course (which you MUST pass, but is not counted towards the grade of the final MSc degree).

PYKC 18 Oct 2024 MSc Lab – Mastering Digital Design Lecture 1 Slide 2

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, SystemVerilog and FPGAs as expected with our MSc graduates.

If you are already experienced with Verilog or SystemVerilog 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" exercise – making 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:

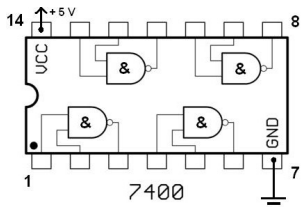
Part 1: Basic competence in using Intel/Altera's Quartus design systems for Cyclone-V FPGA; appreciate the superiority of hardware description language over schematic capture for digital design; use of case statement to specify combinatorial circuit; use higher level constructs in Verilog to specify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

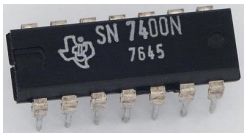
Part 2: Use Verilog to specify sequential circuits; design of basic building blocks including: counters, linear-feedback shift-registers to generate pseudo-random numbers, basic state machines; using enable signals to implement globally synchronisation.

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



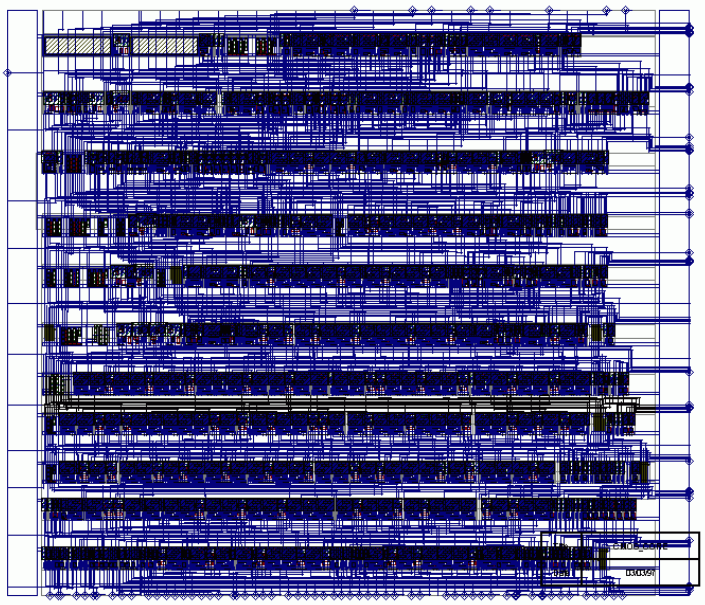


- Discrete logic – based on gates or small packages containing small digital building blocks (at most a 1-bit adder)
- De Morgan's theorem – theoretically we only need 2-input NAND or NOR gates to build anything
- Tedious, expensive, slow, prone to wiring errors

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

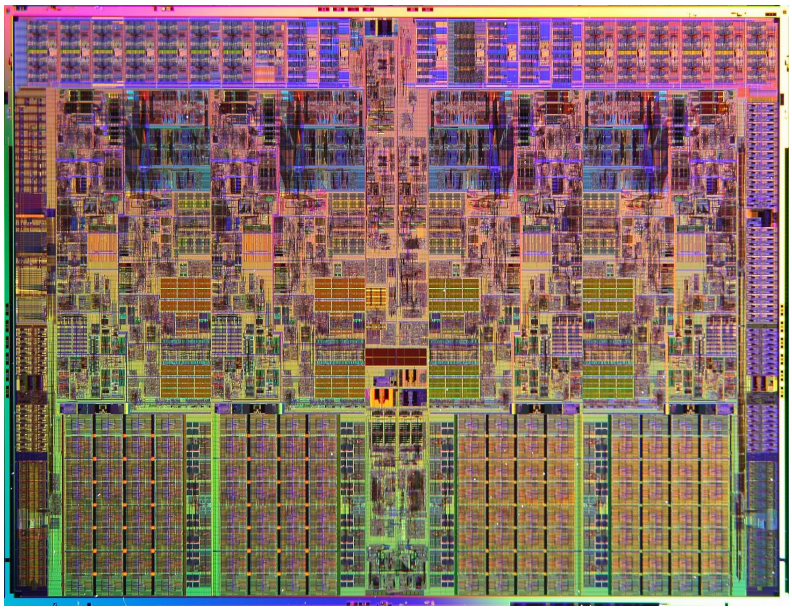


- Rows of gates – often identical in structure
- Connected to form customer specific circuits
- Can be full-custom (i.e. completely fabricated from scratch for a given design)
- Can be semi-custom (i.e. customisation on the metal layers only)
- Once made, design is fixed

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



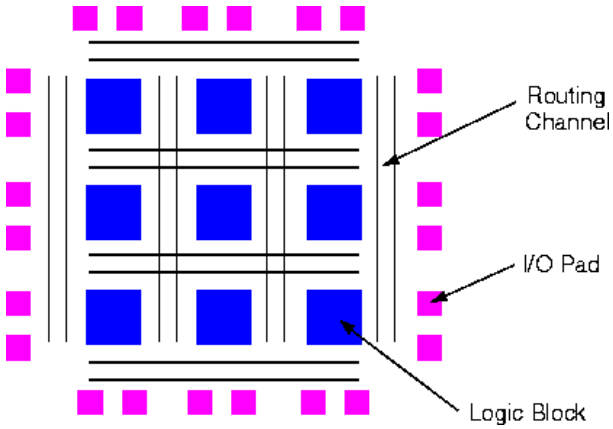
Of course you can also customise everything – each transistor and each wiring connect in a full-custom manner. Here is the layout of Intel i7 microprocessor (with 4 cores). Designing such a circuit is very expensive, highly risky, and once designed, it cannot be changed.

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

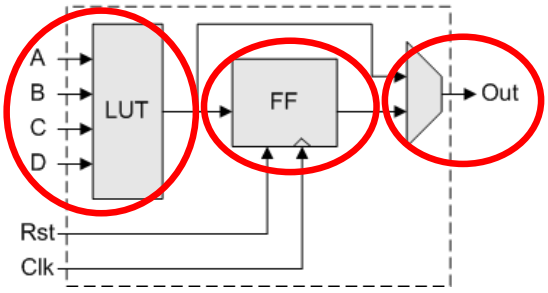
So what is an FPGA? You came across the idea of Programmable Logic Device in the first year, where the user can program what the logic gate does (be it a NAND or NOR or some form of SUM-of-PRODUCT implementation) or an adder, you as a user, can "program" the chip to perform that logic function. Now we 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).

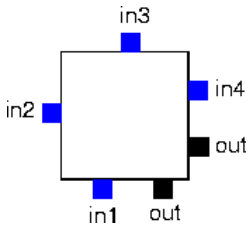
Nowadays, there are two major players in the FPGA domain: Xilinx and Altera (now part of Intel). These two domains 90% of the FPGA market with roughly equal share.

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)

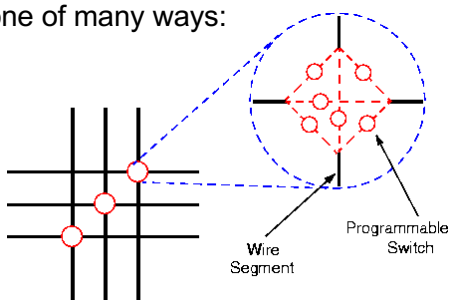


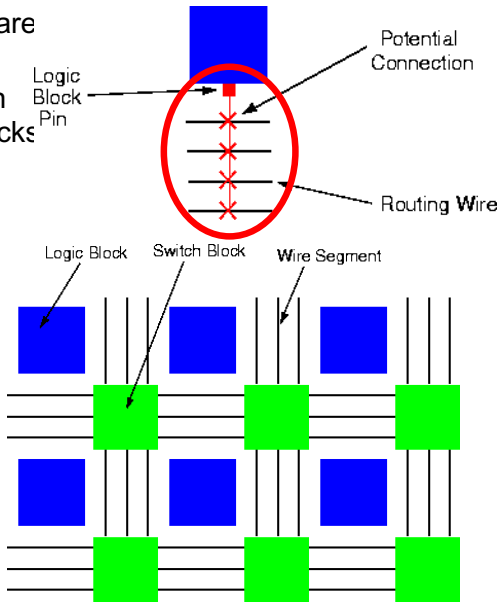
- Each logic block has pins located for easy access:

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 18 Oct 2024 MSc Lab – Mastering Digital Design Lecture 1 Slide 8

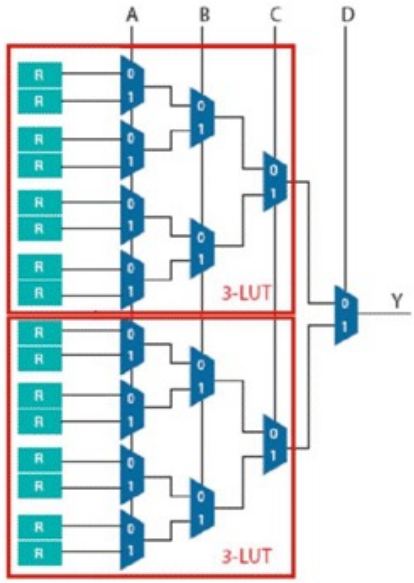
The Logic Elements are surrounded by lots of routing wires and interconnection switches. Typically a signal wire to the Logic Block or Logic Element can be connected to any of these wiring channels through a programmable connection (essentially a digital switch). Xilinx FPGAs also have dedicated switch 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 18 Oct 2024 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.

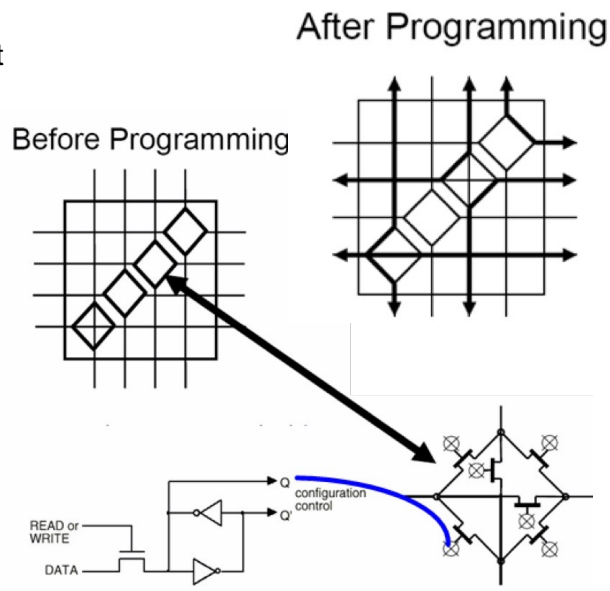
In FPGAs, you only need to **configure** the chip ONCE on power-up. You download to the chip a **BITSTREAM** (also bits in '1's and '0's), which determines the logic functions performed by the Logic Elements, and the interconnecting switches in order to connect the different LEs together to make up your 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 18 Oct 2024 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.

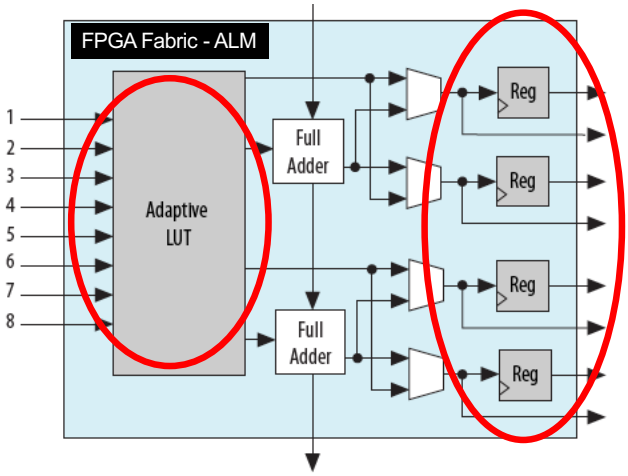
Bitsteam information used for configuration purpose are usually stored on a flash memory chip, which is download to the FPGA during power-up – similar to "booting up a computer". Once this is done, the FPGA is progammed to perform a specific user function (e.g. your design in the VERI experiment).

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

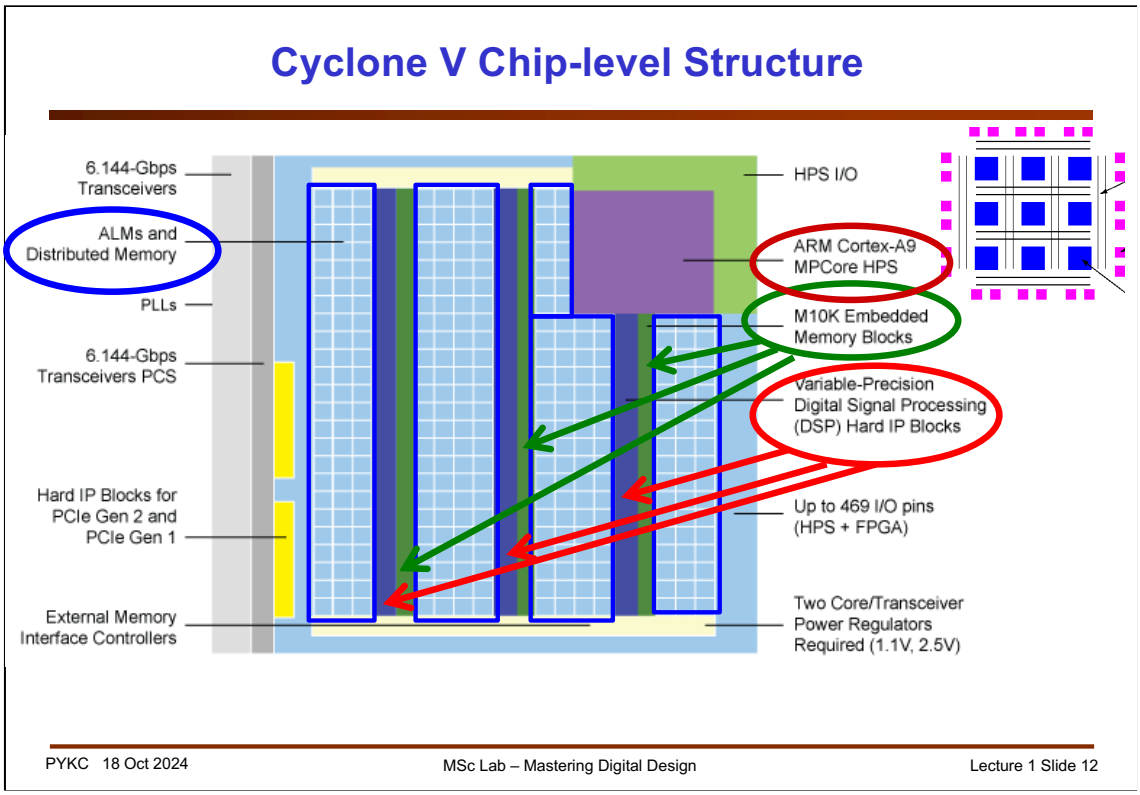


Let us now look at the FPGA that you will use for this course. The Altera Cyclone V FPGA has a more advanced programmable logic element than the simple 4-input LUT that we have considered up to now. The call this a Adaptive Logic Module or ALM.

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.

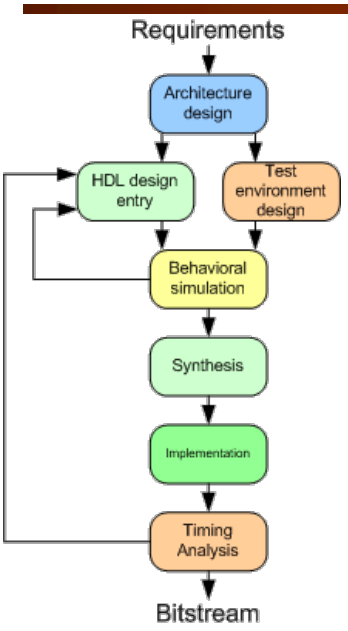
The Cyclone V on the DE1-SOC board has 32,000 ALMs, which could be estimated to be equivalent to 85K+ the old style LEs. Putting this in context, you could put onto this one chip 2,000 32-bit binary adder circuits!



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

PYKC 18 Oct 2024 MSc Lab – Mastering Digital Design Lecture 1 Slide 13

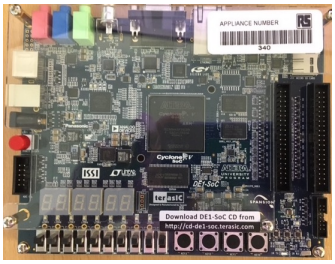
For this lab-based module, you will be designing circuits using the free version of the design suite known as **Quartus Prime Lite** from Intel/Altera. You can download your own copy onto your notebook machine, or you can use the versions that are installed in any PCs located anywhere in the department.

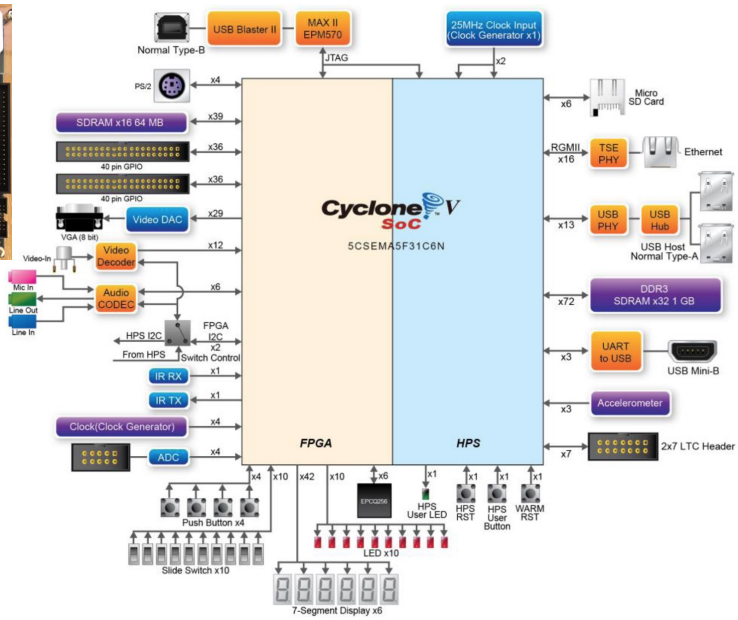
This very powerful design tool contains everything you need to design a complex digital system ON YOUR OWN COMPUTER! However, the software only runs on either a MS Windows or a Linux operating system. If you are using a Mac (Intel processor), you would need to run a Virtual Machine applications (such as Virtual Box) and install Windows or Linux before installing Quartus Prime Lite software. If you are using Mac with Apple Silicon (M1 or above), you are out of luck. The only option would be to use the PC in the Lab!

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

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

DE1-SOC Board

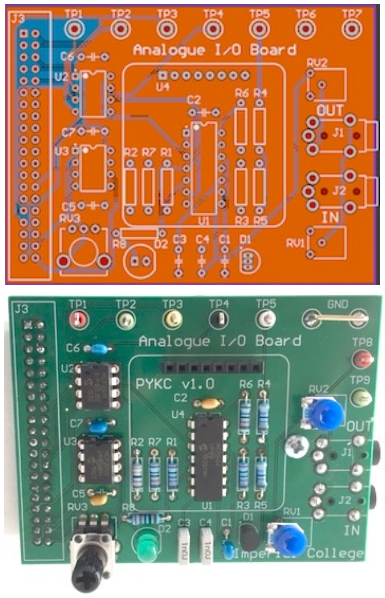




This slide shows you the functional blocks of the DE1-SoC board. This has everything you need test basic designs involving switches, 7-segment displays and even a VGA output.

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



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.
The full circuit schematic is available for this add-on board for those who want to find out more.

Hardware description Languages

- **Hardware description language (HDL):**
  - Specifies logic function only
  - Computer-aided design (CAD) tool produces or synthesizes the optimized gates
- **Most commercial designs use HDLs**
- **Two leading HDLs:**
  - **System Verilog**
    - Developed in 1984 by Gateway Design Automation (Verilog)
    - IEEE standard (1364) in 1995
    - Extended in 2005 (IEEE STD 1800-2009)
  - **VHDL 2008**
    - Developed in 1981 by the Department of Defense
    - IEEE standard (1076) in 1987
    - Updated in 2008 (IEEE STD 1076-2008)

These are the pros and cons of using a HDL instead of schematic to specify digital hardware:

✓ Flexible & parameterisable

✓ Excellent input to optimisation & synthesis

✓ Direct mapping to algorithms

✓ Excellent for datapaths

✓ Easy to handle electronically (only needing a text editor)

× Serial representation

× May not show overall picture

× Need good programming skills

× Divorce from physical hardware

No modern digital integrated circuits or FPGA based designs that are not specified in some sort of HDL from which the final design is synthesized.

For this module, you will learn a particular level of **abstraction** of the processor hardware known as **Register Transfer Level** (RTL). In RTL specifications, all combinational logic are sandwiched between registers controlled by one or more clock signals.

HDL to Gates

- **Simulation**
  - Inputs applied to circuit
  - Outputs checked for correctness
  - Millions of dollars saved by debugging in simulation instead of hardware
- **Synthesis**
  - Transforms HDL code into a netlist describing the hardware (i.e., a list of gates and the wires connecting them)
- **Physical design**
  - Placement, routing, chip layout, …… – not considered in this module

IMPORTANT:

When using an HDL, think of the **hardware** the HDL should produce, then write the appropriate idiom that implies that hardware.

Beware of treating HDL like software and coding without thinking of the hardware.

After specifying your hardware in System Verilog HDL, you need to make sure that your design works according to specification. Simulation tools such as circuit simulators, Matlab, Mathematica etc. allow users to predict circuits and systems behaviour WITHOUT having to implement the actual electronic system. This saves both time and money. Furthermore, it is very hard to find a bug in a million or billion transistor circuit on a physical chip because there is no easy way to access internal signals. (This statement is not entirely true. There is a technique used called "**scan chain**" or **JTAG,** which allows such internal access, but it is not easy to use.)

After simulation, the design is "**translated**" to low level building blocks (such as gates and flops) through a special type of **hardware compiler** to perform **synthesis**. This is the stage at which circuits can be optimized. For example, redundant gates (such as a 2-input NAND gate with one input always 0) are eliminated. Synthesis produces a network of interconnected building blocks, known as the **netlist**. At this stage, the design is still not necessary linked to any technology for implementation.

The netlist is then further processed to produce the final physical design. This final stage involves many steps such as **technology mapping**, **placement**, **routing**, **timing analysis, test vector generation**, **test coverage analysis** etc. We will NOT be considering any part of this stage of design in this module.

System Verilog: Structural Description

Behavioural

```
module and3(input  logic a, b, c,
            output logic y);
  assign y = a & b & c;
endmodule
```

```
module inv(input  logic a,
           output logic y);
  assign y = ~a;
endmodule
```

Structural

```
module nand3(input  logic a, b, c
             output logic y);
  logic n1;                  // internal signal

  and3 andgate(a, b, c, n1); // instance of and3
  inv  inverter(n1, y);      // instance of inv
endmodule
```

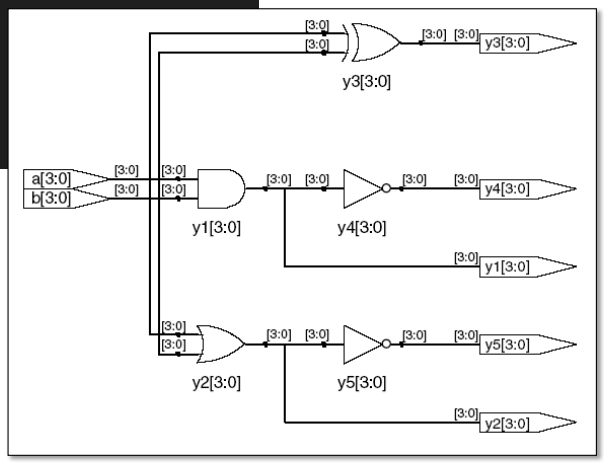

Combinational circuit is easiest to specify using behavioural specification with Boolean operators. You can also choose to provide structural description with interconnected gates as shown on the right.

It is NOT advisable to describe low-level modules in a structural way. It is both tedious, prone to error and not easy to read.

We normally only use structural description when we connect large modules together at a higher level of the design hierarchy.

System Verilog: Bitwise Operators

```
module gates(input  logic [3:0]  a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
   /* Five different two-input logic
      gates acting on 4 bit busses */
   assign y1 = a & b;     // AND
   assign y2 = a | b;     // OR
   assign y3 = a ^ b;     // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR
endmodule
```



Here is an example where signals are bundled into multi-bit bus. In this case, they are 4-bit wide as [3:0]. SV does not restrict you to name the bus from bit 3 to bit 0. You could declare the signals as, say, [4:1] instead. However, we adapt the notation that LSB is bit 0, and MSB is WIDTH-1, in this case 3.

Now the continuous assignment keyword “assign” results in bit-wise operation. For example:

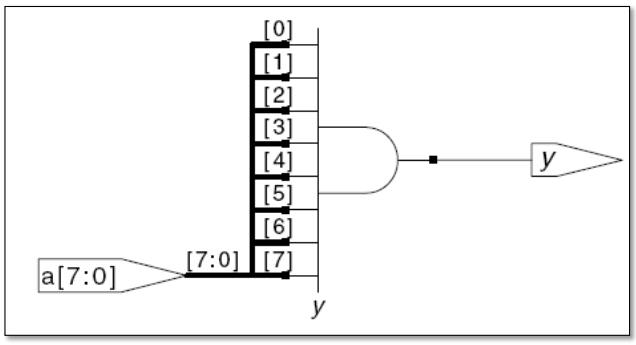
assign y1 = a & b;

Means:

y1[3] = a[3] & b[3], y1[2] = a[2] & b[2]. ……

SysytemVerilog: Reduction Operators

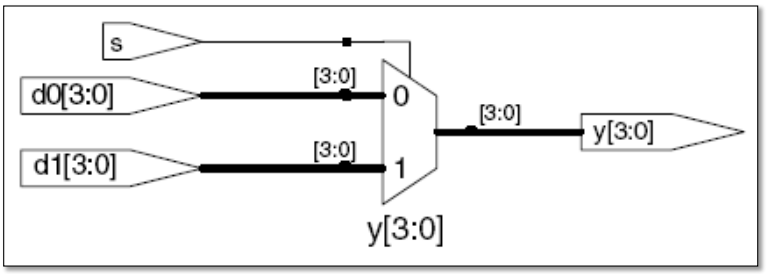
```
module and8(input  logic [7:0] a,
            output logic       y);
  assign y = &a;
  // &a is much easier to write than
  // assign y = a[7] & a[6] & a[5] & a[4] &
  //            a[3] & a[2] & a[1] & a[0];
endmodule
```



The '&' operator can also be used with a single operand as shown here. This is called a "**reduction**" operator. It reduces multiple bits of a[7:0] to a single bit y. It basically ANDs all bits of a[7:0] together as shown in the slide.

System Verilog: Conditional Assignment

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);
  assign y = s ? d1 : d0;
endmodule
```



The conditional assignment operator (as found in C or C++) is:

cond ? True_value : False_value

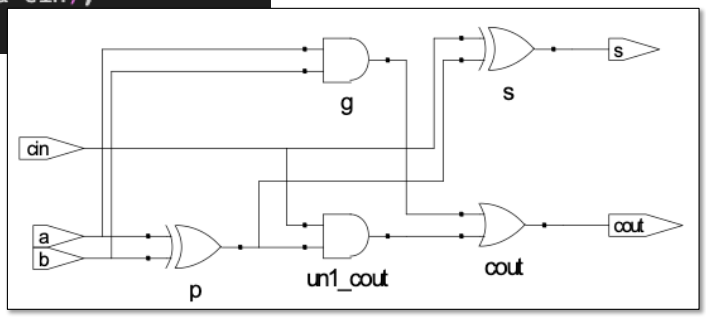
Therefore, assign y = s ? d1 : d0;

Is the same as: If s is true, y = d1, else y = d0.

This effectively produces a **multiplexer** as shown here.

System Verilog: Internal Signals

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;   // internal nodes

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```



For most modules, there are internal signals which are neither inputs nor outputs. The module here is a single bit full adder. There are two internal signals p, g.

These signals are not “visible” outside the module and are declared as local signals (similar to local variables in C++ functions).

System Verilog: Precedence of operators

Highest

~	NOT
*, /, %	mult, div, mod
+, -	add, sub
<<, >>	shift
<<<, >>>	arithmetic shift
<, <=, >, >=	comparison
==, !=	equal, not equal
&, ~&	AND, NAND
^, ~^	XOR, XNOR
|, ~|	OR, NOR
?:	ternary operator

Lowest

Here are all the operators that SystemVerilog understands. They are listed here with their precedence.

System Verilog: Number Format

Format: N'Bvalue

N = number of bits, **B** = base

N'B is optional but recommended (default is decimal)

Number	# Bits	Base	Decimal Equivalent	Stored
3'b101	3	binary	5	101
'b11	unsized	binary	3	00…0011
8'b11	8	binary	3	00000011
8'b1010_1011	8	binary	171	10101011
3'd6	3	decimal	6	110
6'o42	6	octal	34	100010
8'hAB	8	hexadecimal	171	10101011
42	Unsized	decimal	42	00…0101010

When using SystemVerilog to describe hardware, always remember that you are NOT writing a program. All "variables" are in fact signals. So, when specifying number, beware that you are using physical wire.

Therefore numbers are specified with number of bits explicitly stated. The general format is N'Bxxxx.

N is the number of bits. **B** is the base: b = binary, d = decimal, h = hexadecimal.

See above. If you don't provide bit and base specification, the number is assumed to be 32 bits and in decimal by default. Not specifying the size (i.e. number of bits) of a signal in a design is not recommended.

System Verilog: Bit Manipulations (1)

```
assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};
```

- If y is a 12-bit signal, the above statement produces:

```
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0
```

- Underscores (_) are used for formatting only to make it easier to read. **System Verilog ignores them**.

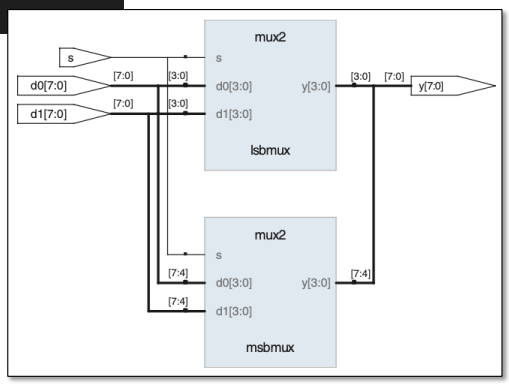
The syntax shown here is very unlike C or C++, and is particularly important to specification of hardware.
{ . } is called a **concatenation** operation. { 1, 0, 1, 1} forms a 4-bit number 4'b1011.

In the example above, a[2:1] is a 2-bit number a[2] and a[1].

{ 3 {b[0]} } forms a 3-bit number with b[0] repeated 3 times.

System Verilog: Bit Manipulations (2)

```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

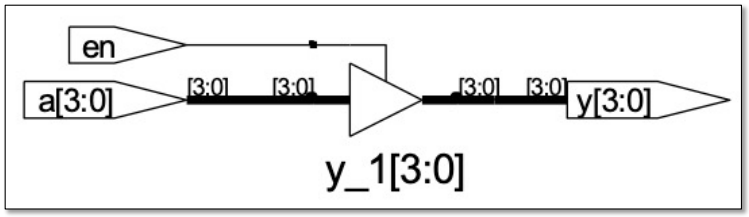


This is an example of slicing and merging different bits of signals d0 and d1 to form an 8-bit output y.

If d0 = 8'b10110101, and d1 = 8'h5A, work out what is y for s = 0, and s = 1?

System Verilog: Floating Output Z

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);
   assign y = en ? a : 4'bz;
endmodule
```



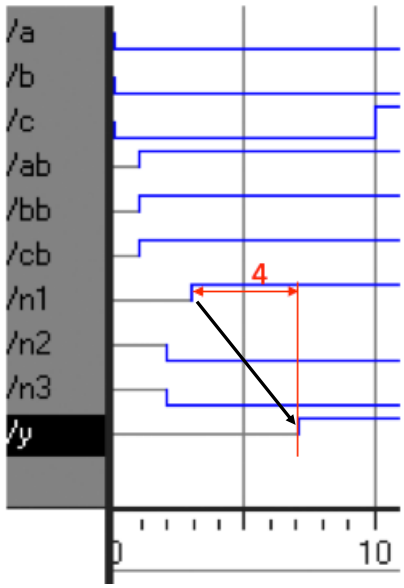
- Note that Verilator does not handle floating output Z

We normally use "**logic**" to specify a signal to be a signal which has values of 0 or 1. However, there is a signal type **tri** which can take on three values: 0, 1, or z, where z is high impedance. This allows System Verilog to **describe tri-state outputs**.

In this module, and if en=1, then y = a. If en=0, the output y is tri-state and is therefore not driven by this module.

System Verilog: Delays

```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



- Delays are for simulation only! They do not determine the delay of your hardware.
- **Verilator similator ignores delays** – it is cycle accurate without timing.

Digital circuits have **delays**. System Verilog provides constructs to specify such delays (default in ns). However, Verilator ignores all such specifications: Verilator assumes that all combinational logic output changes immediately with inputs. As such, Verilator is NOT suitable to verify physical digital circuits – it can only be used for functional verification.

System Verilog: Sequential Logic

- System Verilog uses **idioms** (or special keywords or groups of words) to describe latches, flip-flops and FSMs
- Other coding styles may simulate correctly but produce incorrect hardware
- GENERAL STRUCTURE:

```
always @(sensitivity list)
        statement;
```

- Whenever the event in `sensitivity list` occurs, `statement` is executed

Sequential logics are specified using the pattern:

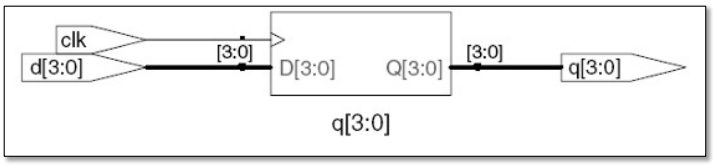
```
always @(sensitivity list)
        statement;
```

The "**always**" followed by **@(sensitivity list)** means that when any signal in the sensitivity list is asserted, "statement" is executed.

All sequential circuits are described in this form.

System Verilog: D Flip-Flop

```
module flop(input  logic       clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;                 // pronounced "q gets d"

endmodule
```



System Verilog has a specific syntax for D flip-flops.

aways_ff @(posedge clk)

will synthesize one or more registers that are triggered on **positive edge** of the signal **clk**.

Note that you can call your clock signal anything, e.g. **fred** would do equally well. There is NO SIGNIFICANCE in the name itself. However, it is of course advisable to use a signal name that is meaningful.

Note also that the statement to execute in this case is:

q <= d;

This is called **non-blocking assignment** (but don't worry about what it is called for now). The effect of this module is: on rising edge of clk, the 4-bit value of d is transferred to q.

This will synthesize to 4-bit D flip-flop.

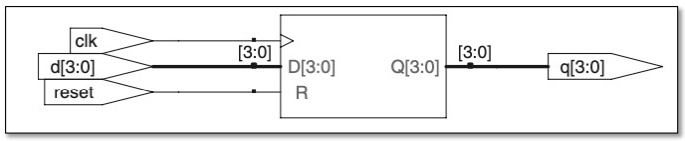
System Verilog: Resettable D Flip-Flop

Asynchronous reset

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

// asynchronous reset
always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else q <= d;
endmodule
```

Synchronous reset

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // synchronous reset
  always_ff @(posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```



PYKC 18 Oct 2024 MSc Lab – Mastering Digital Design Lecture 1 Slide 31

You should **ALWAYS** add a **reset** control to your flops. Otherwise, your digital system may power up in a random state.

Reset can be implemented as **synchronous** or **asynchronous**. Synchronous reset means that reset happens only on the active edge of the clock signal. Asynchronous reset can happen anytime whenevert the reset signal is asserted and is independent of the clock.

The slide shows the two forms of reset description. For asychonrous case, it also shows how the sensitivity list can **contain multiple conditions**.

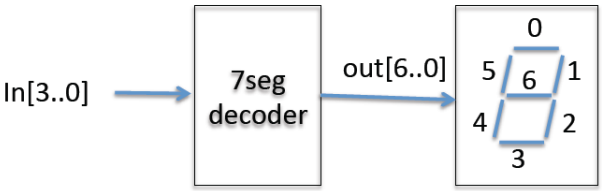
Combinational Logic using always

```
// combinational logic using an always statement
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
  always_comb        // need begin/end because there is
    begin            // more than one statement in always
      y1 = a & b;    // AND
      y2 = a | b;    // OR
      y3 = a ^ b;    // XOR
      y4 = ~(a & b); // NAND
      y5 = ~(a | b); // NOR
    end
endmodule
```

This hardware could be described with **`assign` statements using fewer lines** of code, so it's better to use **`assign`** statements in this case.

There is a form of **always block** which allows the specification of **combinational circuits**. However, there is no advantage in this form of specification as compare to multiple assign statements.

Combinational Logic using always-case



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

```
module hex_to_7seg (
    output  logic [6:0]   out,
    input   logic [3:0]   in);

    always_comb
        case (in)
        4'h0:   out = 7'b1000000;
        4'h1:   out = 7'b1111001;
        4'h2:   out = 7'b0100100;
        4'h3:   out = 7'b0110000;
        4'h4:   out = 7'b0011001;
        4'h5:   out = 7'b0010010;
        4'h6:   out = 7'b0000010;
        4'h7:   out = 7'b1111000;
        4'h8:   out = 7'b0000000;
        4'h9:   out = 7'b0011000;
        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
```

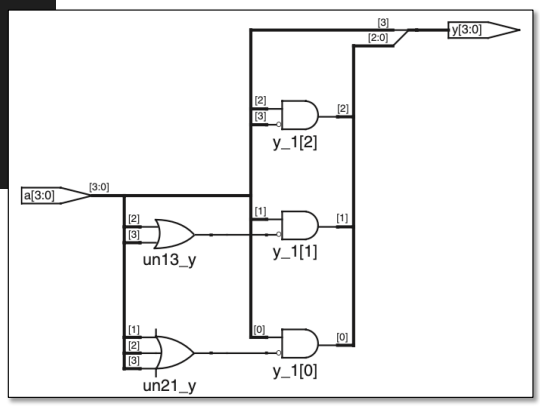

A more common use of the **always_comb** statement is when it is used with the **case** statement.
Here is a 7-segment decoder specification. A 4-bit binary input in[3:0] is decoded to provide 7 output signals to drive a 7-segment display. The outputs are assumed to be **low-active**, i.e. the segments turns ON when the output signals [6:0] are driven LOW.
The function of the decoder can be specified in the truth table shown.
The **case statement** here shows a direct way to specify such a decoder.

In general, any truth tables or ROMs can be specified in this way.

Combinational Logic using if-else

❖ Prioity encoder circuit

```
module priorityckt( input logic  [3:0] a,
                    output logic [3:0] y);
always_comb
    if (a[3])       y = 4'b1000;
    else if (a[2])  y = 4'b0100;
    else if (a[1])  y = 4'b0010;
    else if (a[0])  y = 4'b0001;
    else            y = 4'b0000;
endmodule
```

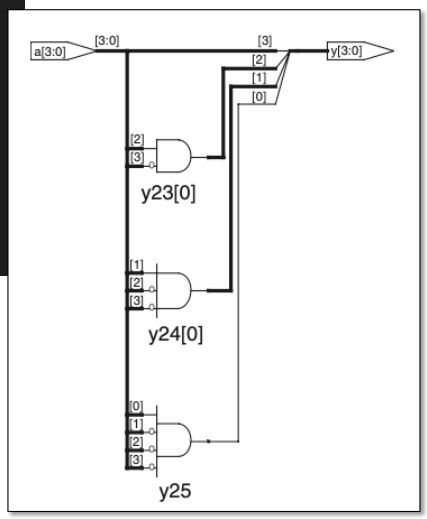


Here is another example called **priority encoder**. **always_comb** statement uses **if-else** constructs. The output y[3:0] reports the position of the first '1' in the input from MSB to LSB. So if a[3] is '1', then y[3] is '1' etc.

This is call a **priority encoder** because it detects the highest priority signal being set. The if-else statement is perfect for such description because it fully describes the behaviour of the circuit explicitly.

Combinational Logic using casez

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);
  always_comb
    casez(a)
      4'b1???: y = 4'b1000; // ? = don't care
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
    endcase
endmodule
```



- ? = don't-care
- Beware: MUST have default statement in case not all cases are covered!

Here is an alternatively method to do the same thing using the casez statement.

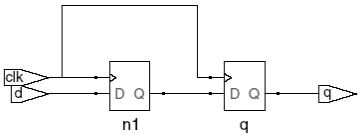
Here the conditions are specified with '?' meaning "**don't care**". Note that with 4-bits input, there are 16 possibilities. '?' allows these bit values to be either '0' or '1' (i.e. don't care).

However, beware that not all cases may be covered by such specifidation. You MUST always specify the default case (i.e. when the input a value is not included in the case list).

Blocking vs. Nonblocking Assignment

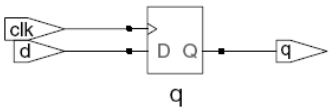
- <= is nonblocking assignment
  - Occurs simultaneously with others

```
// Good synchronizer using
// nonblocking assignments
module syncgood(input  logic clk,
                input  logic d,
                output logic q);
  logic n1;
  always_ff @(posedge clk)
    begin
      n1 <= d;  // nonblocking
      q  <= n1; // nonblocking
    end
endmodule
```



- = is blocking assignment
  - Occurs in order it appears in file

```
// Bad synchronizer using
// blocking assignments
module syncbad(input logic  clk,
               input  logic d,
               output logic q);
  logic n1;
  always_ff @(posedge clk)
    begin
      n1 = d;  // blocking
      q  = n1; // blocking
    end
endmodule
```



Inside any always block, you should use the **non-blocking assignement** "<=" instead of the **block assignment** "=".

With <=, all assignment statements take effectly ONLY at the end of the always block simultaneously.

With = assignment, assignment occurs sequentially. The synthesized results is a single flip-flop instead of a shift register.

ALWAYS USE "<=" IN YOUR SEQUENTIAL CIRCUIT SPECIFICATION.

Rules for Signal Assignment

- **Synchronous sequential logic, use:**

  `always_ff` and nonblocking assignments (<=)

```
always_ff @(posedge clk)
    q <= d; // nonblocking
```

- **Simple combinational logic, use continuous assignments (assign…)**

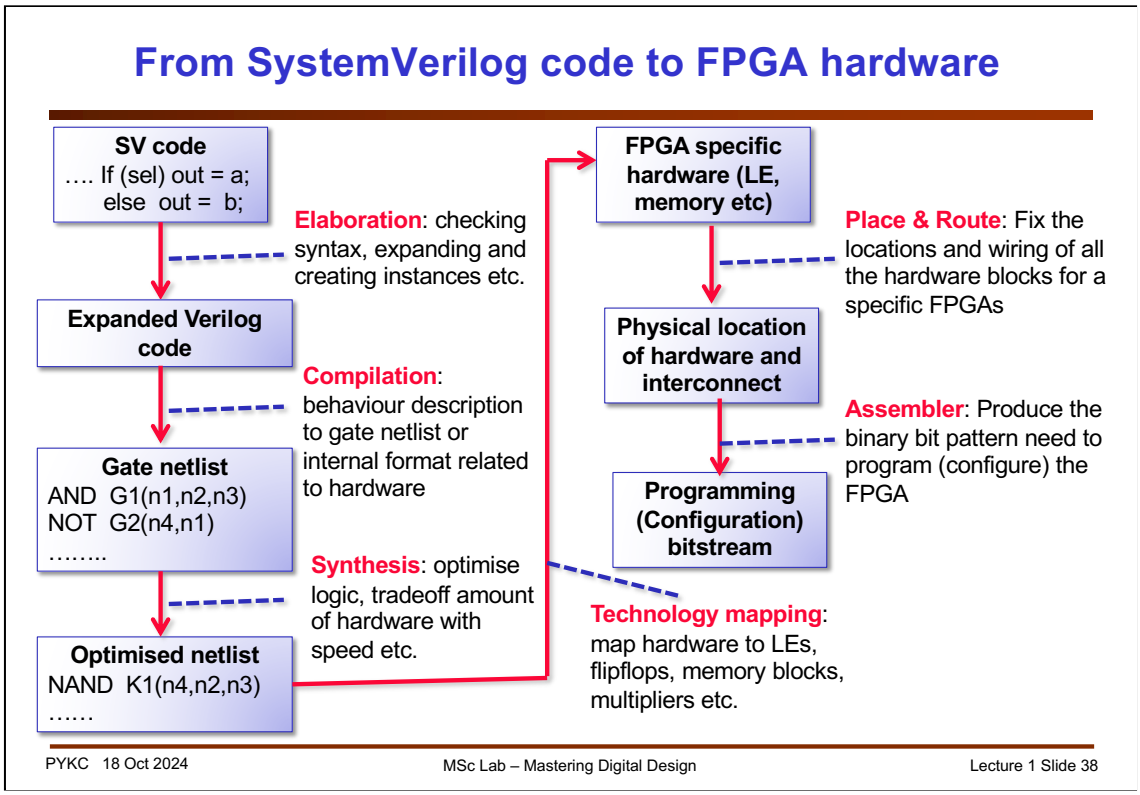
```
assign y = a & b;
```

- **More complicated combinational logic, use:**

  `always_comb and blocking assignments (=)`

- **Assign a signal in ONLY ONE `always` statement or continuous `assignment` statement.**

Here are some general rules about assingments.



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