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

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.

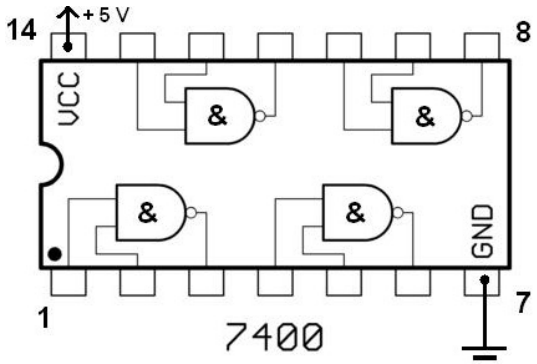
Format:

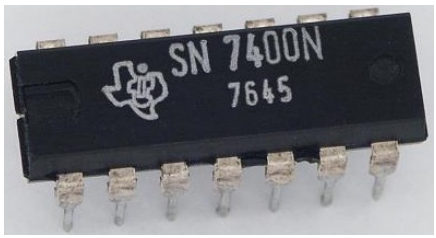
- All lectures by me – to teaching you something or to go over materials in the previous lab session
- Lab Experiment in four parts by Dr Aaron Zhao and GTAs - Each experiment should take 1 week, and each has very clearly defined Learning Outcomes.

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).

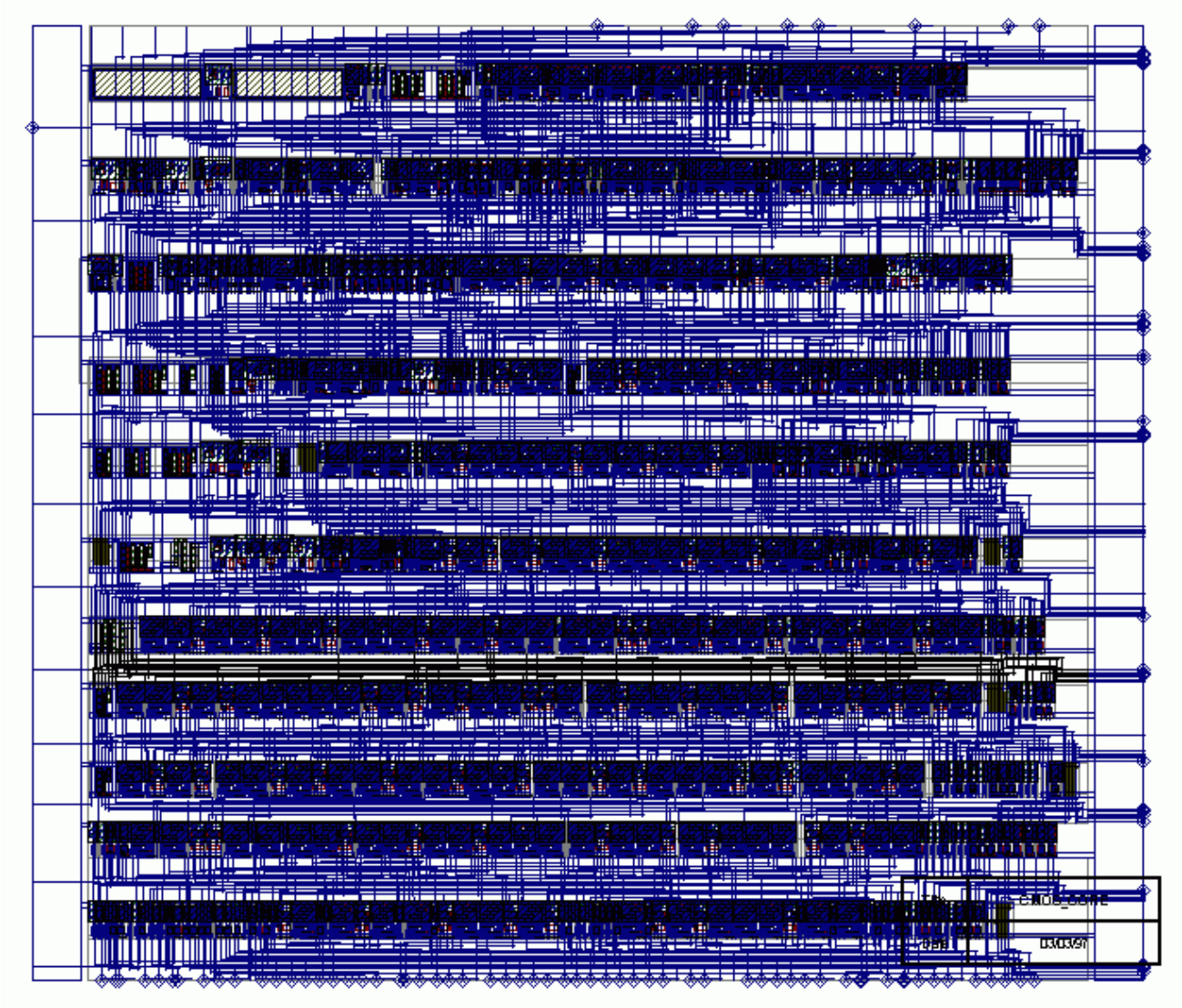
Old ways of implementing digital circuits





- 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

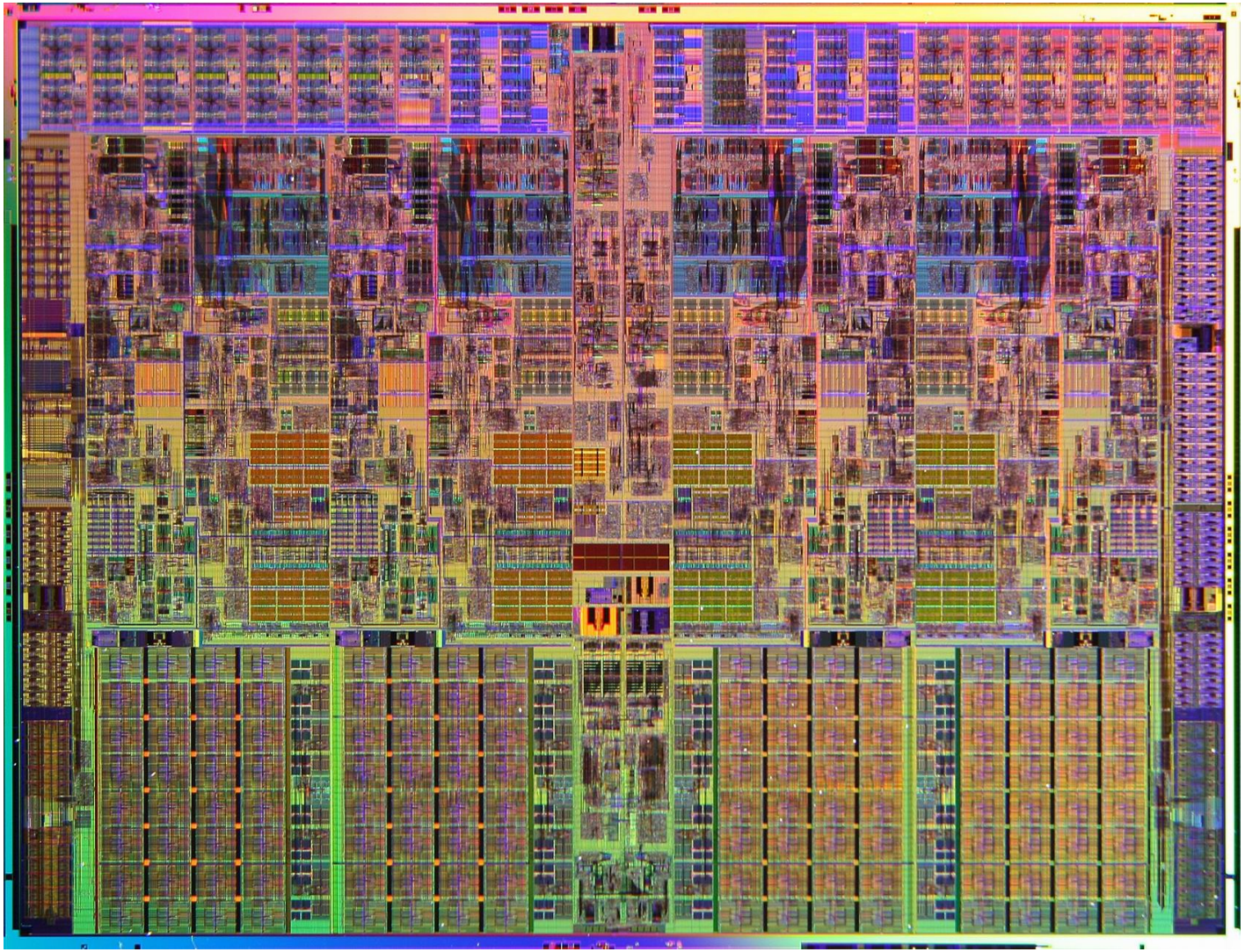
Early integrated circuits based on gate arrays



- 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

Modern digital design – full custom IC

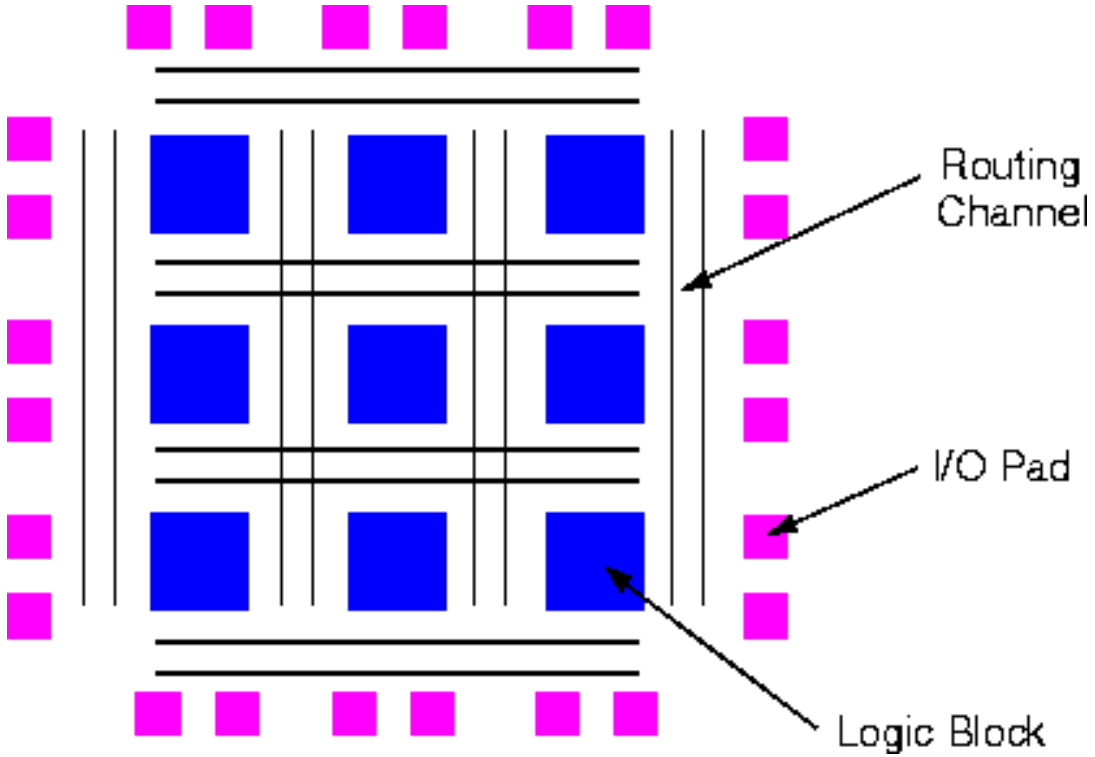
- Intel Core i7
- > ¾ billion trans.
- Very expensive to design
- Very expensive to manufacture
- Not viable unless the market is very large



Field Programmable Gate Arrays (FPGAs)

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

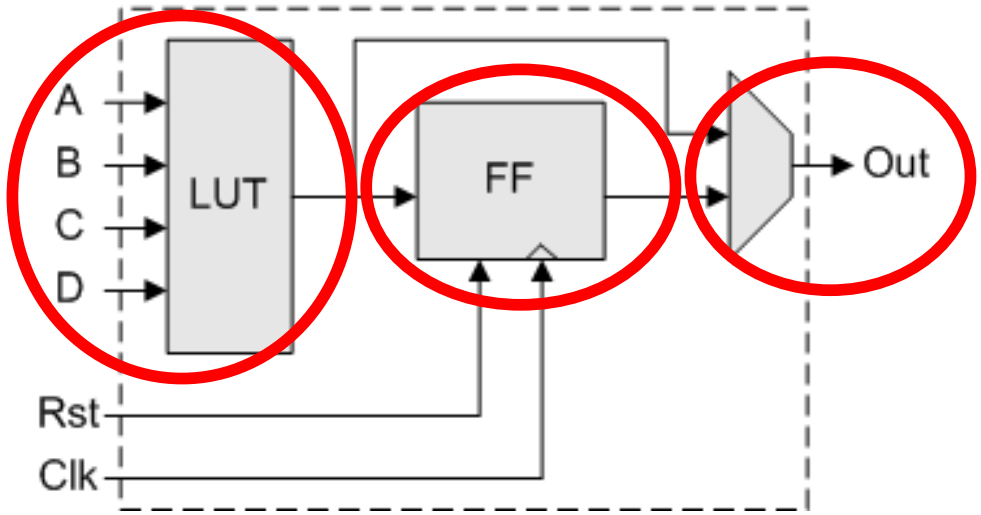
- Arrays of logic blocks (to implement logic functions)
- Lots of programmable wiring in routing channels
- Very flexible I/O interfacing logic core to outside world
- Two dominant FPGA makers:
  - Xilinx and Altera
- Other specialist makers e.g. Actel and Lattice Logic



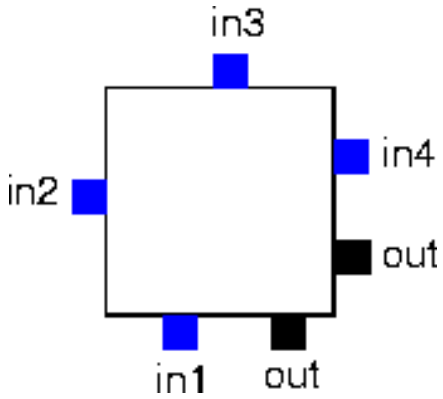
R1.1 p1 - p16

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)



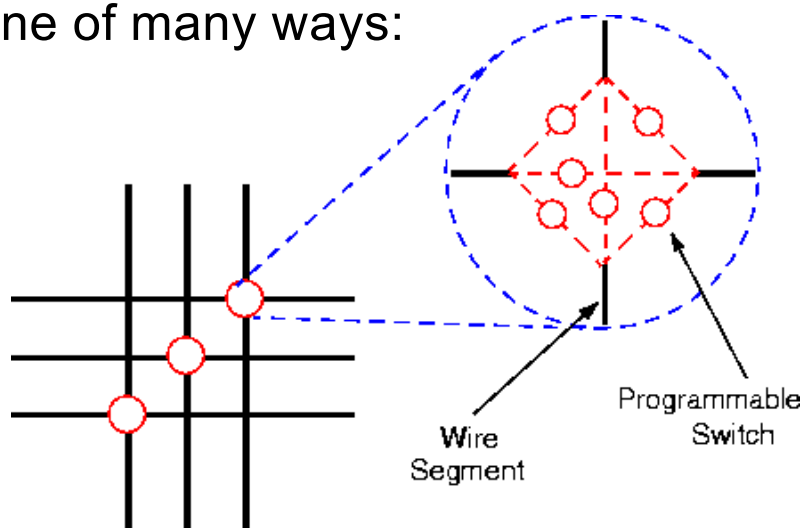
Logic Element (LE)

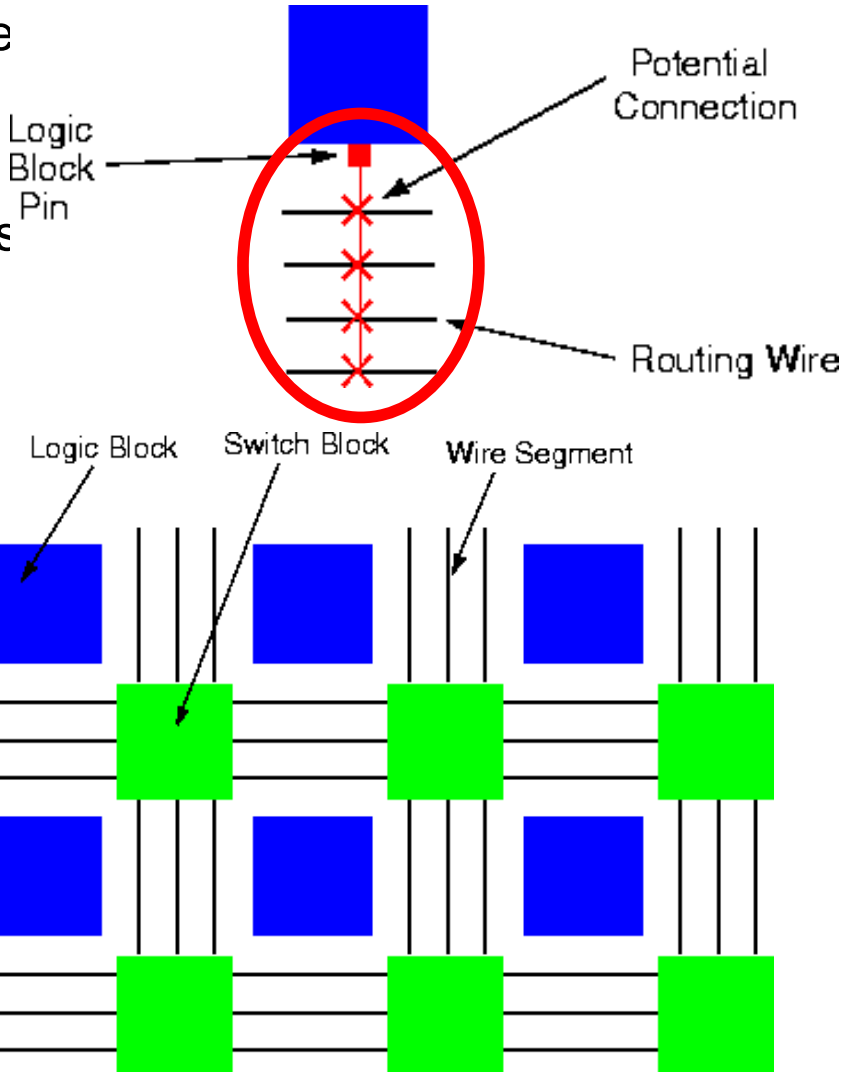


- Each logic block has pins located for easy access:

Programmable Routing

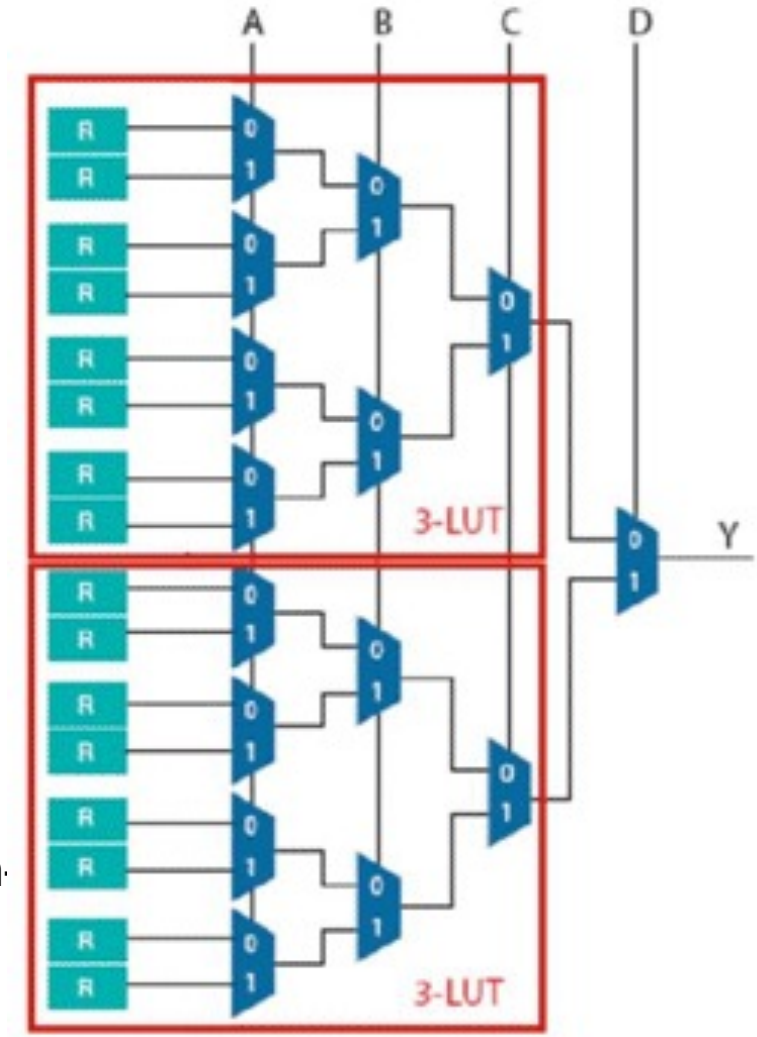
- Between rows and columns of logic blocks are wiring channels
- These are programmable – a logic block pin can be connected to one of many wiring tracks through a programmable switch
- Xilinx FPGAs have dedicated switch block circuits for routing (more flexible)
- Each wire segment can be connected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





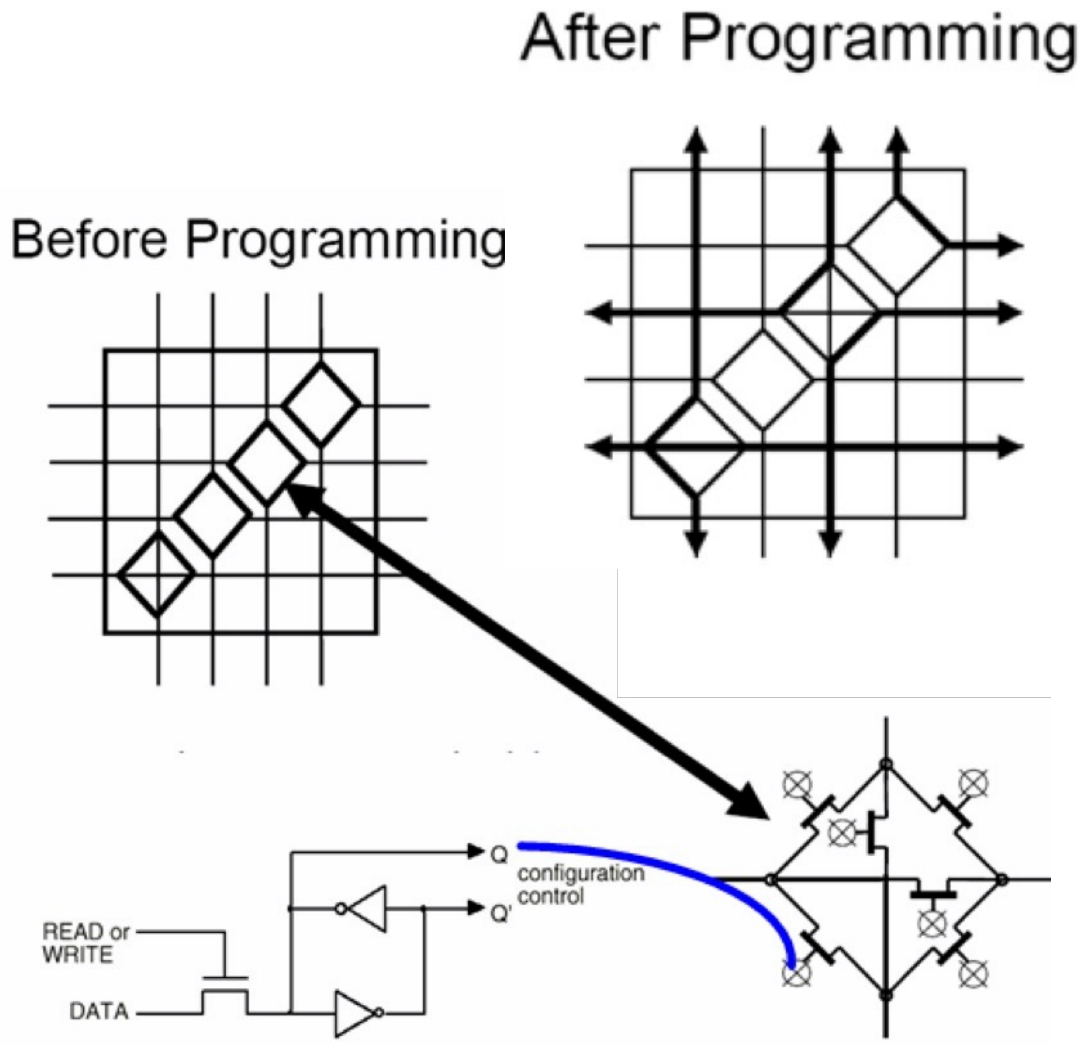
The Idea of Configuring the FPGA

- 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



Configuring the routing in an FPGA

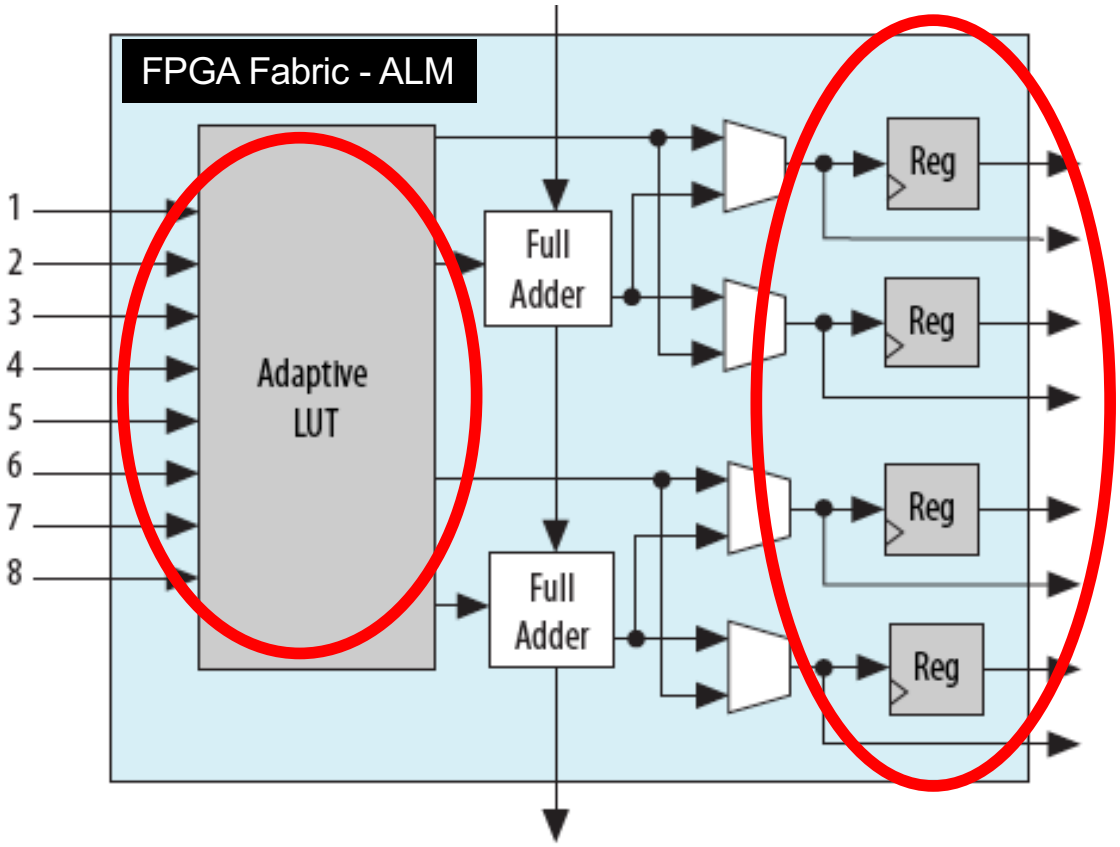
- At each interconnect site, there is a transistor switch which is default OFF (not conducting)
- Each switch is control by the output of a 1-bit configuration register
- Configuring the routing is simply to put a ‘1’ or ‘0’ in this register to control the routing switches
- Bitstream is either stored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



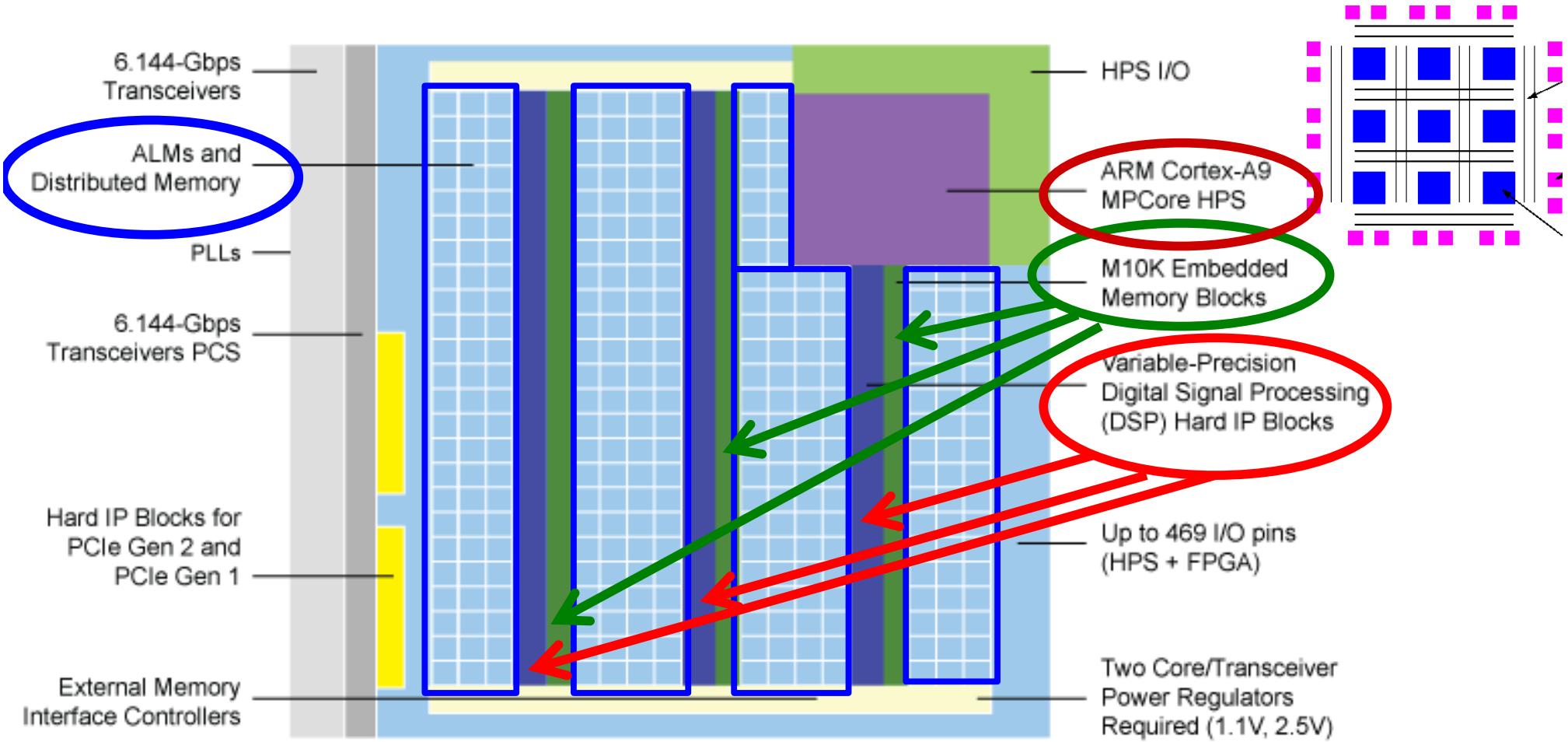
Cyclone V's Adaptive Logic Module (ALM)

- 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

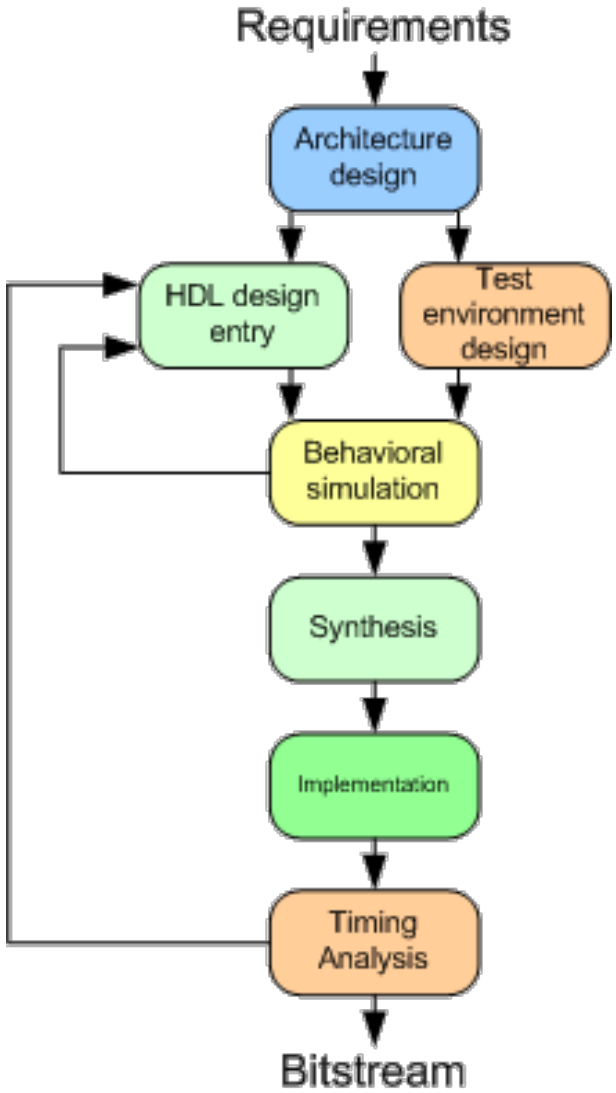
- The logic element is more advanced than the original 4-LUT architecture
- The ALM can implement much larger logic functions, or can be broken into a number of smaller units



Cyclone V Chip-level Structure

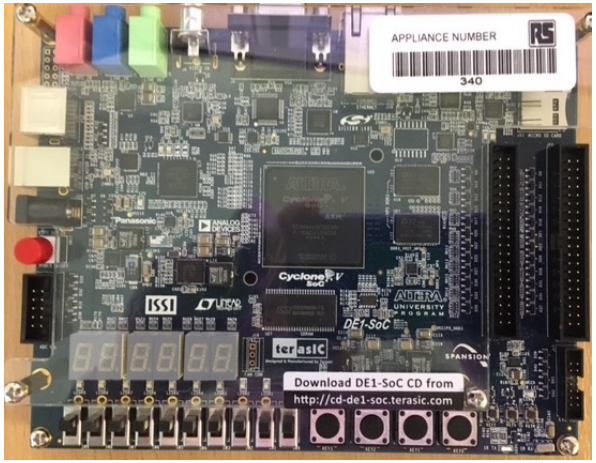


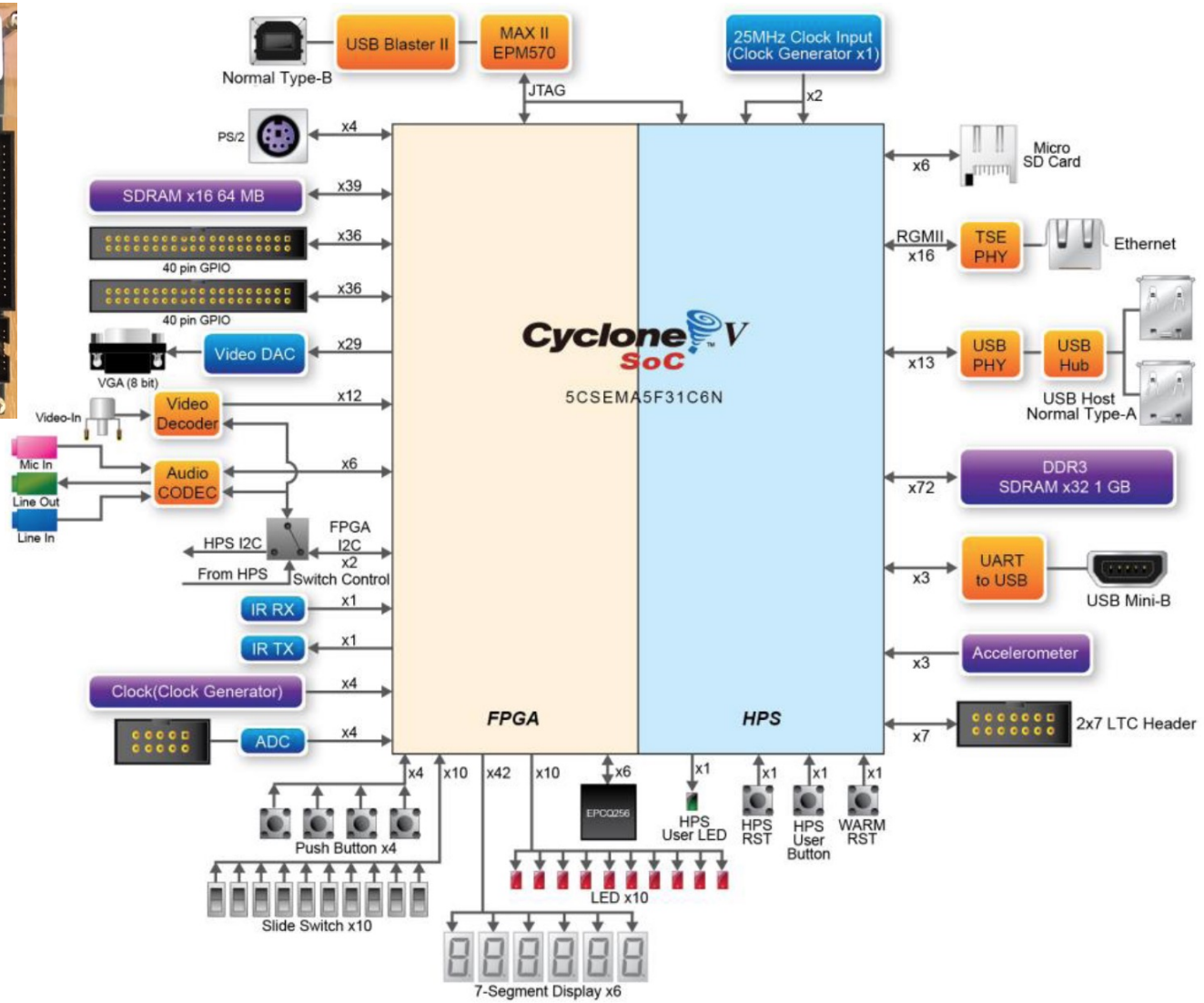
Design Tools – Altera Quartus II



- 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

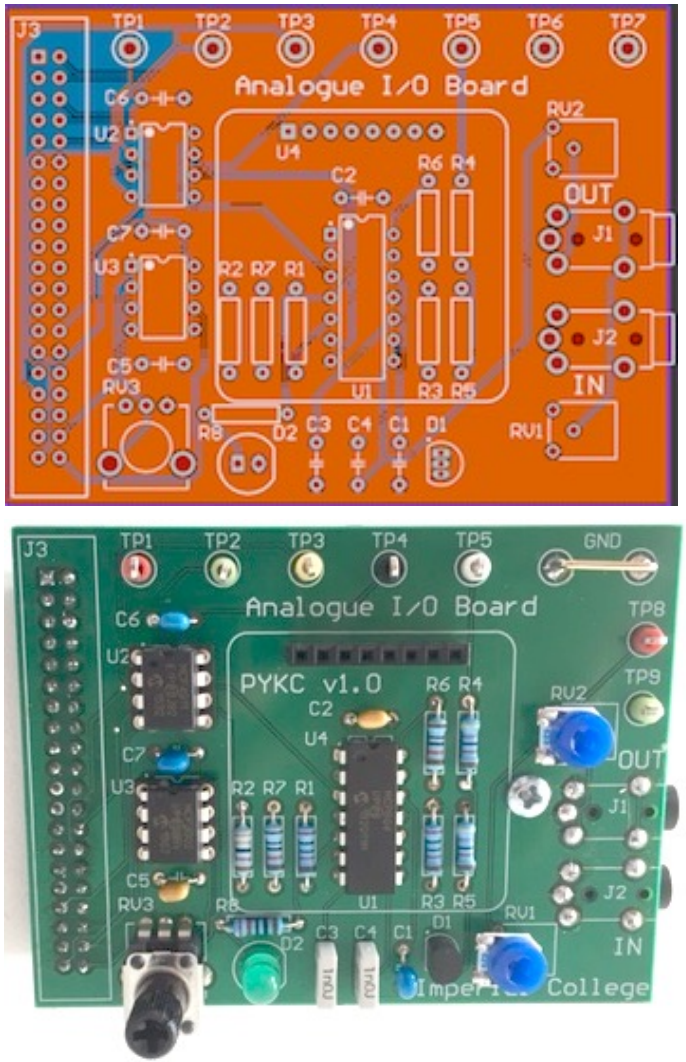
DE1-SOC Board





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



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)

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.

System Verilog: Structural Description

Behavioural

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

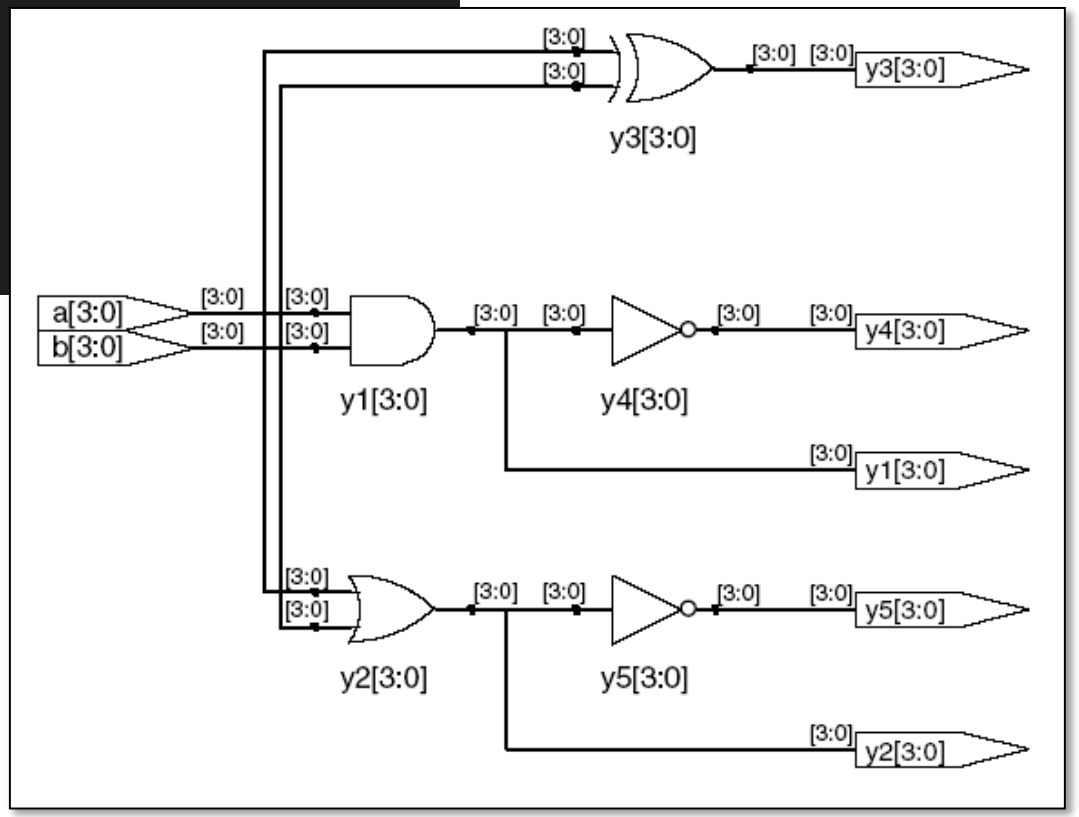
```
module inv(input  logic a,
           output logic y);
  assign y = ~a;
endmodule
```

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
```

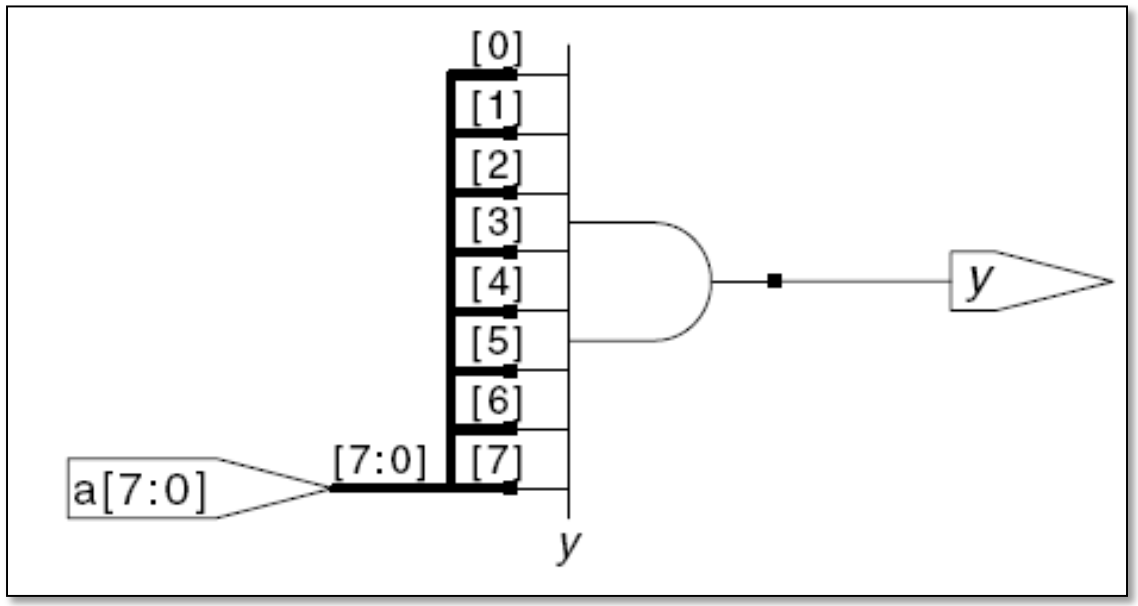
System Verilog: Bitwise Operators

```
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
```



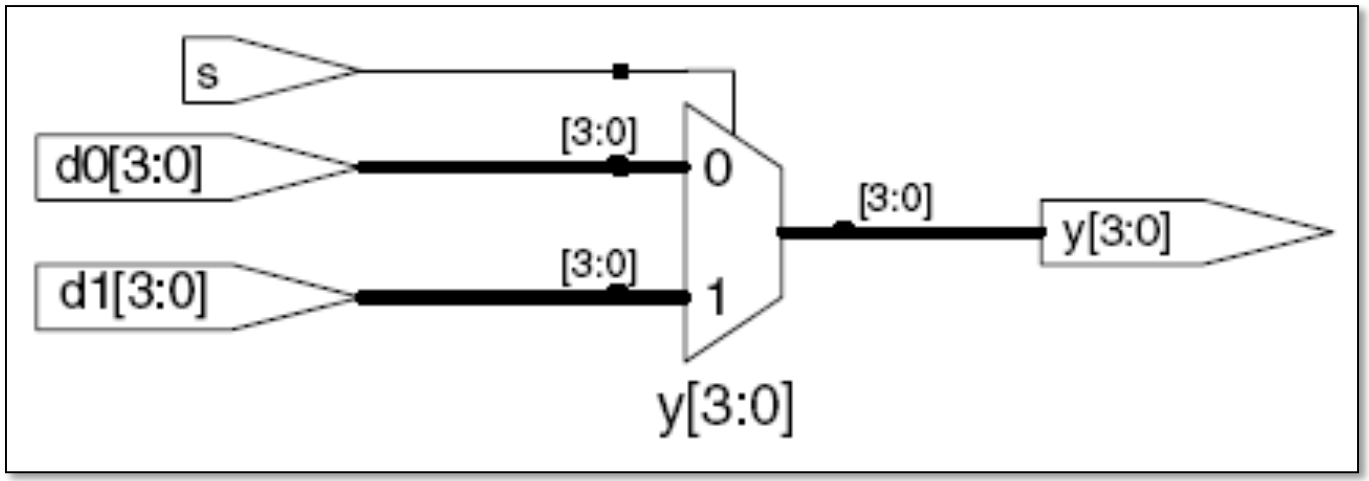
SysytemVerilog: Reduction Operators

```
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
```



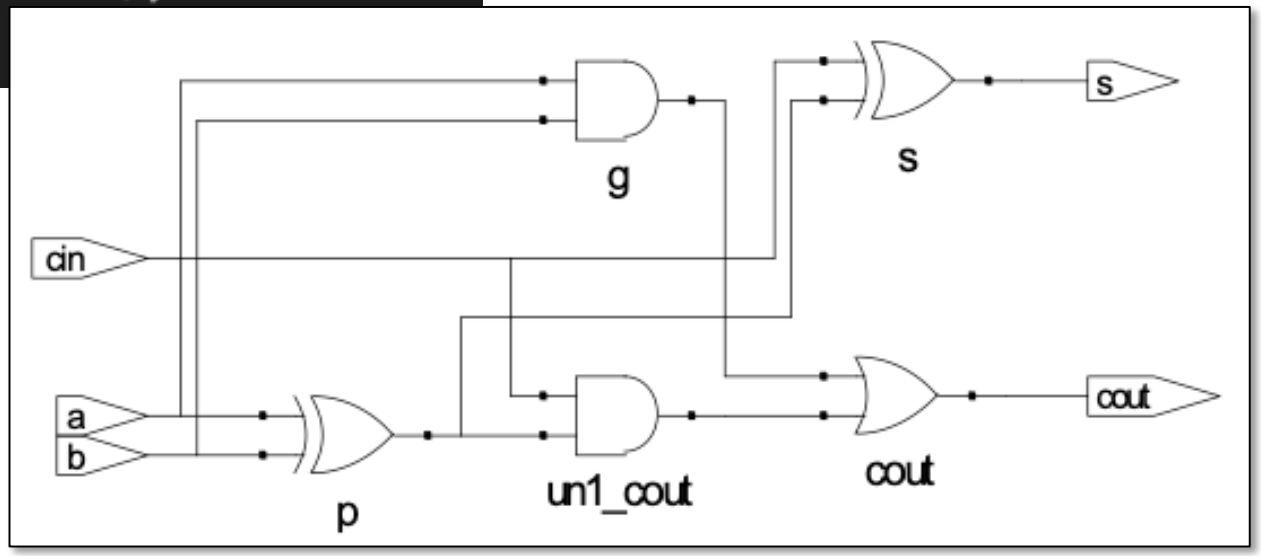
System Verilog: Conditional Assignment

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



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
```



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

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

System Verilog: Bit Manipulations (1)

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

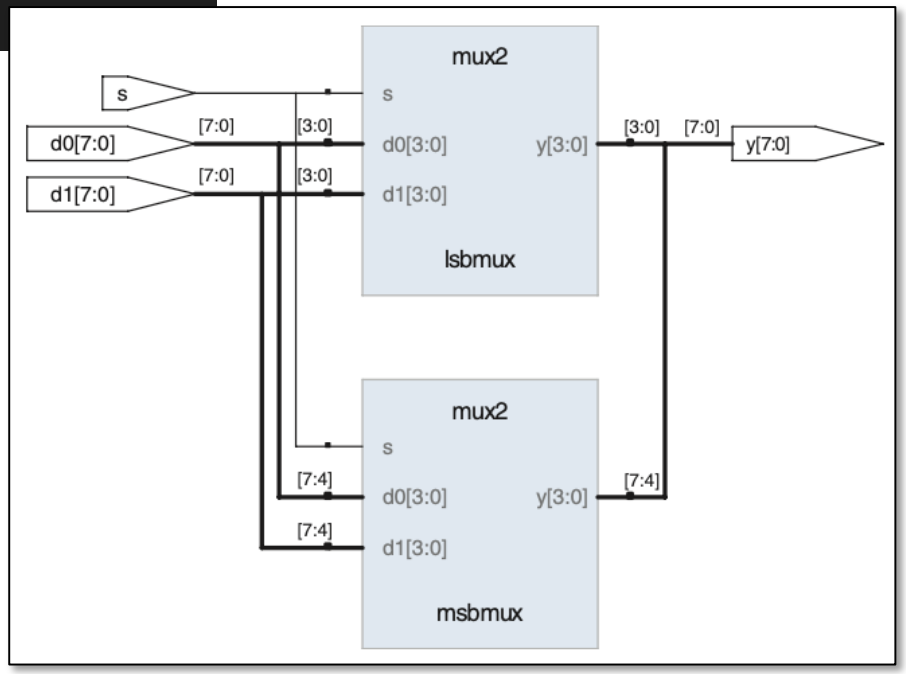
- If y is a 12-bit signal, the above statement produces:

```
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**.

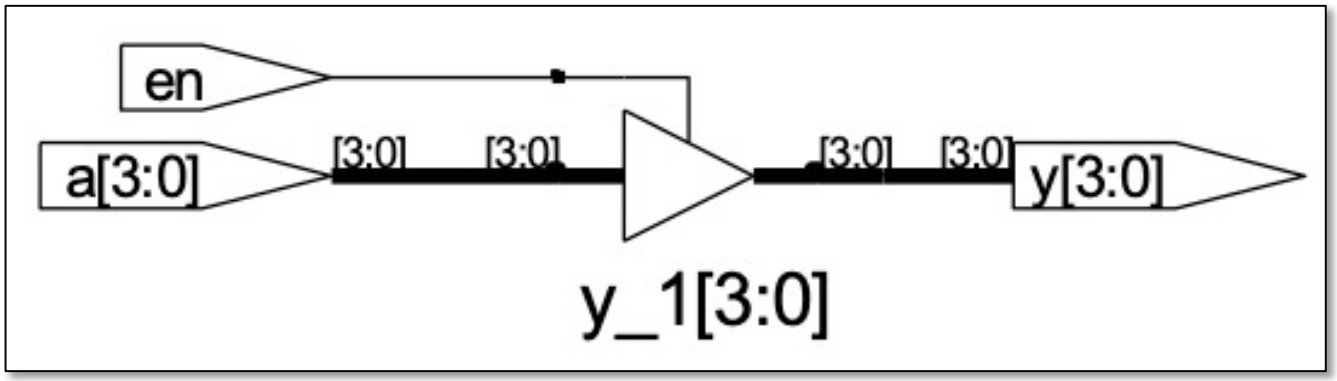
System Verilog: Bit Manipulations (2)

```
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
```



System Verilog: Floating Output Z

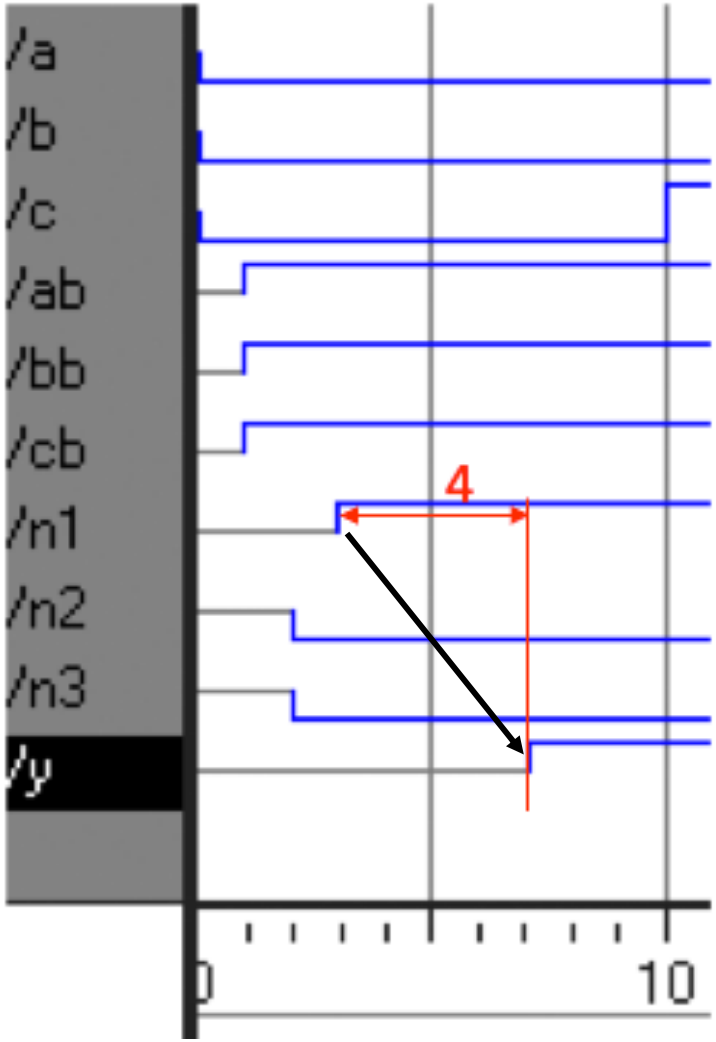
```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);
   assign y = en ? a : 4'bz;
endmodule
```



- Note that Verilator does not handle floating output Z

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.

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

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
```

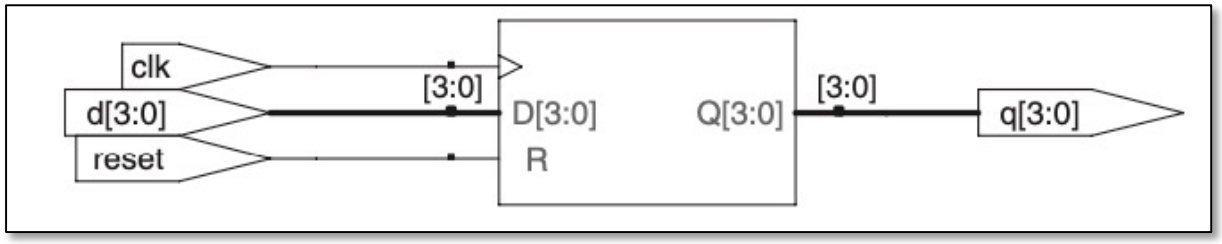
System Verilog: Resettable D Flip-Flop

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
```

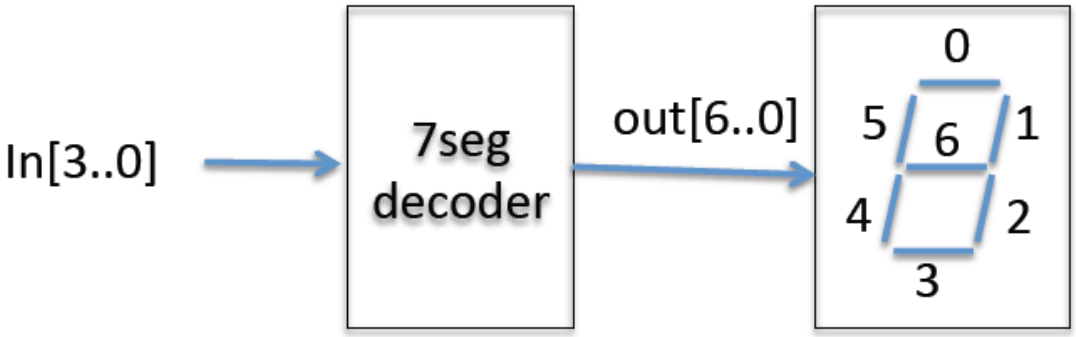


Combinational Logic using always

```
// 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.

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

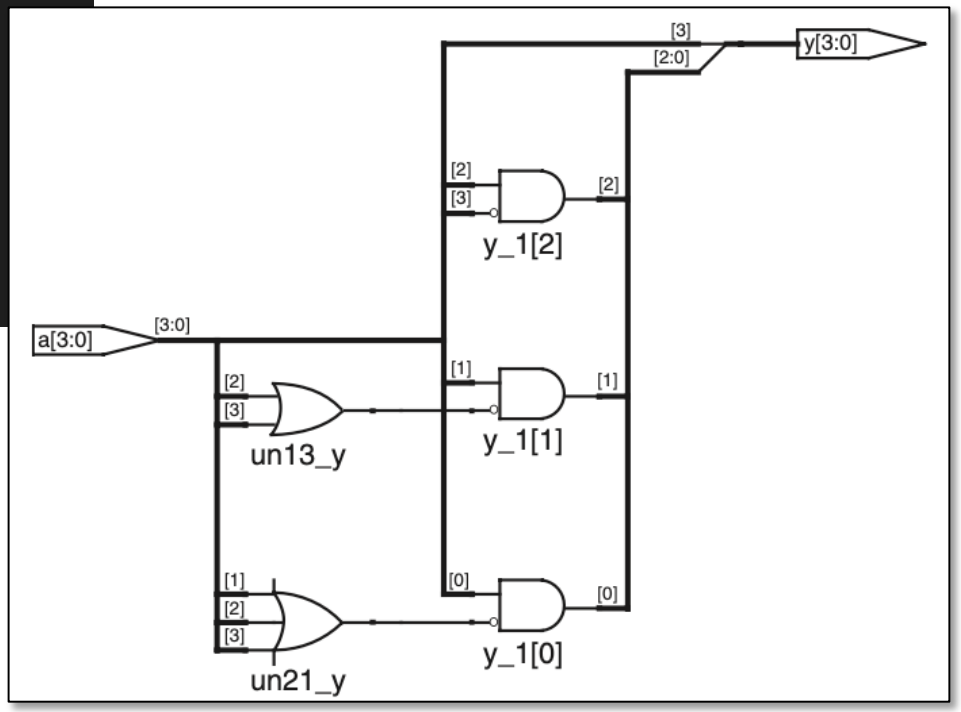
in[3..0]	out[6:0]	Digit
1000	0000000	8
1001	0010000	9
1010	0001000	A
1011	0000011	b
1100	1000110	C
1101	0100001	d
1110	0000110	E
1111	0001110	F

```
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
```

Combinational Logic using if-else

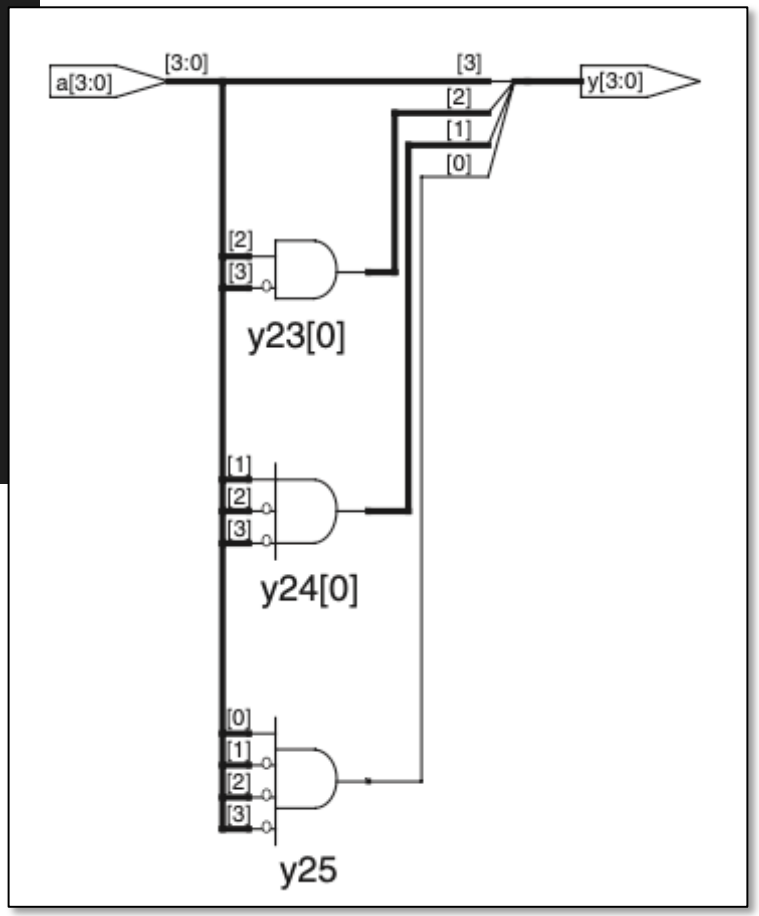
- Prioity encoder circuit

```
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
```



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
```

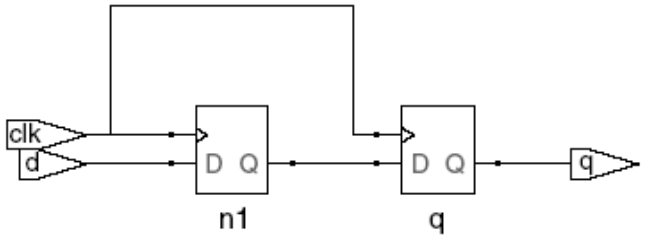


- ? = don't-care
- Beware: MUST have default statement in case not all cases are covered!

Blocking vs. Nonblocking Assignment

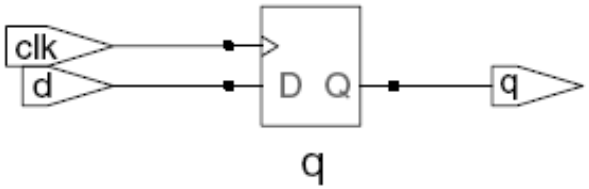
- <= is nonblocking assignment
  - Occurs simultaneously with others

```
// 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
```



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.**

From SystemVerilog code to FPGA hardware