

# Mastering Digital Design

with Verilog on FPGAs

John Wickerson

Lecture 1

# Module aims

- To ensure that all MSc students reach a **common competence level** in RTL design using FPGAs in a hardware description language.
- To act as a **revision exercise** for those already competent in Verilog and FPGAs.

# Format

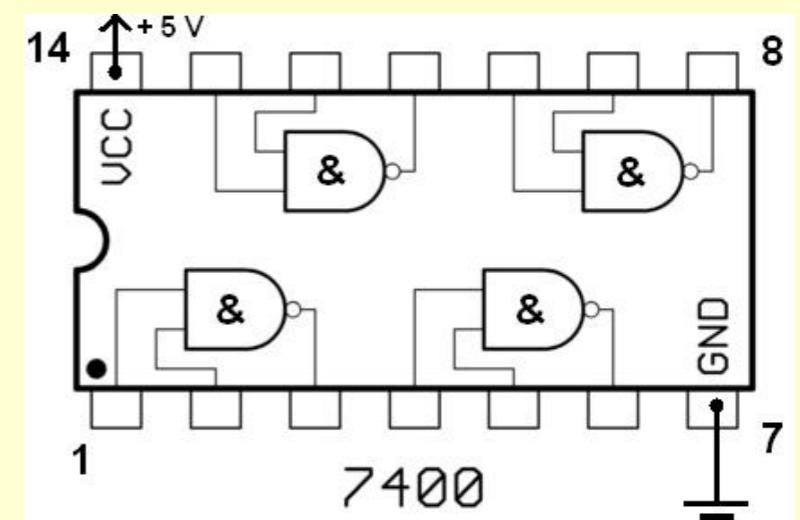
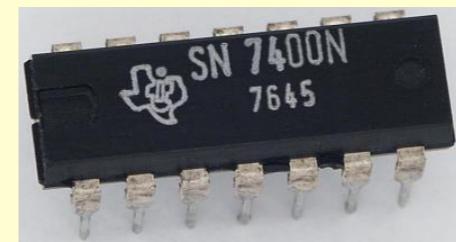
- The lab experiment is divided into four parts.
- It is hoped that you will finish one part each week.
- Each part has clearly defined Learning Outcomes.
- These lectures will supplement the lab with some background information.

# Assessment

- This lab is part of the Coursework component of the MSc course. You must pass this component, but it does not count towards your final MSc grade.
- One-to-one oral interview in December.

# The old days

- **Discrete logic**, built from individual gates or packages containing small digital building blocks (e.g. a 1-bit adder).

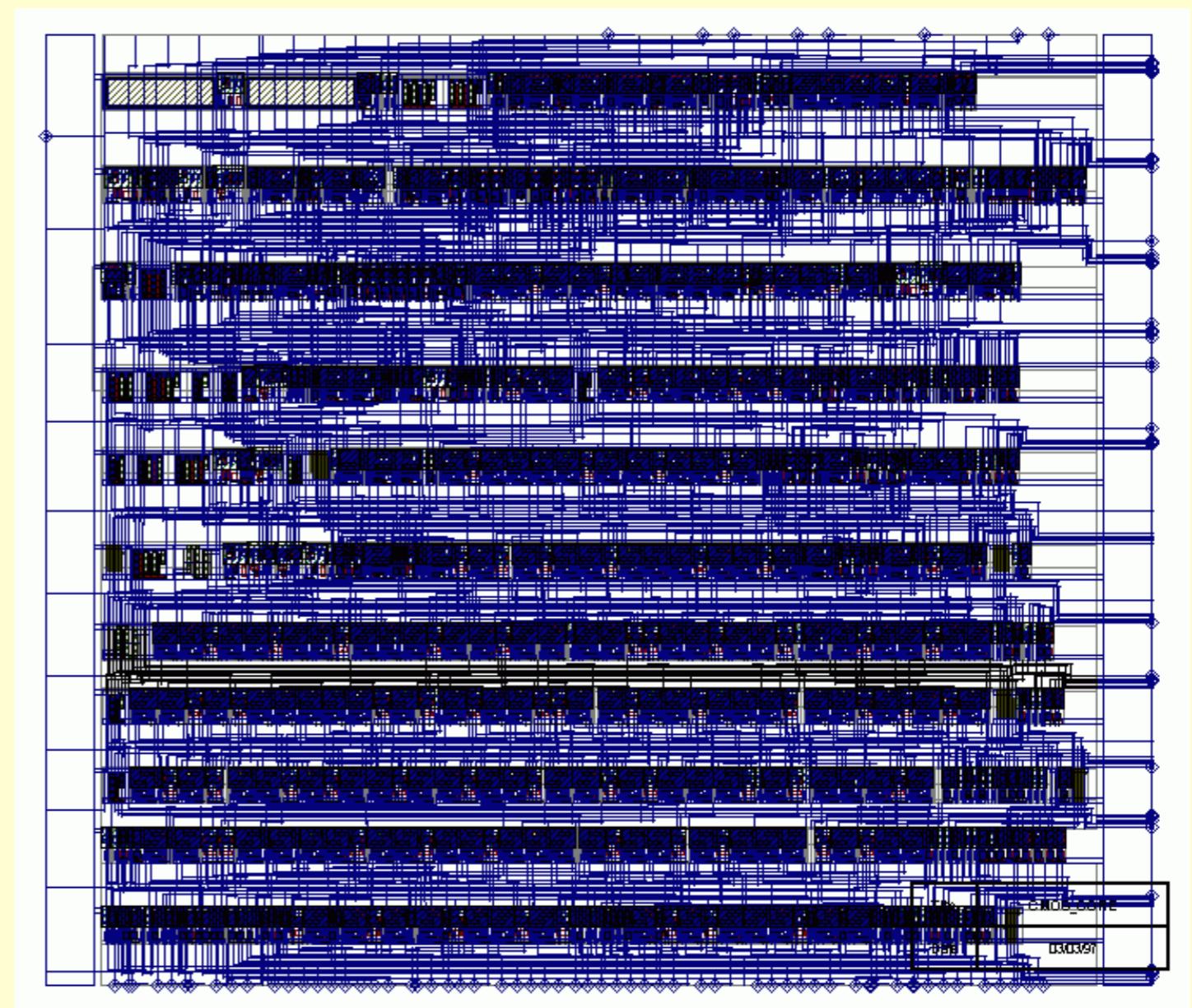


- Thanks to De Morgan, we can build *anything* using 2-input NAND gates!
- But this is tedious, expensive, and prone to wiring errors.



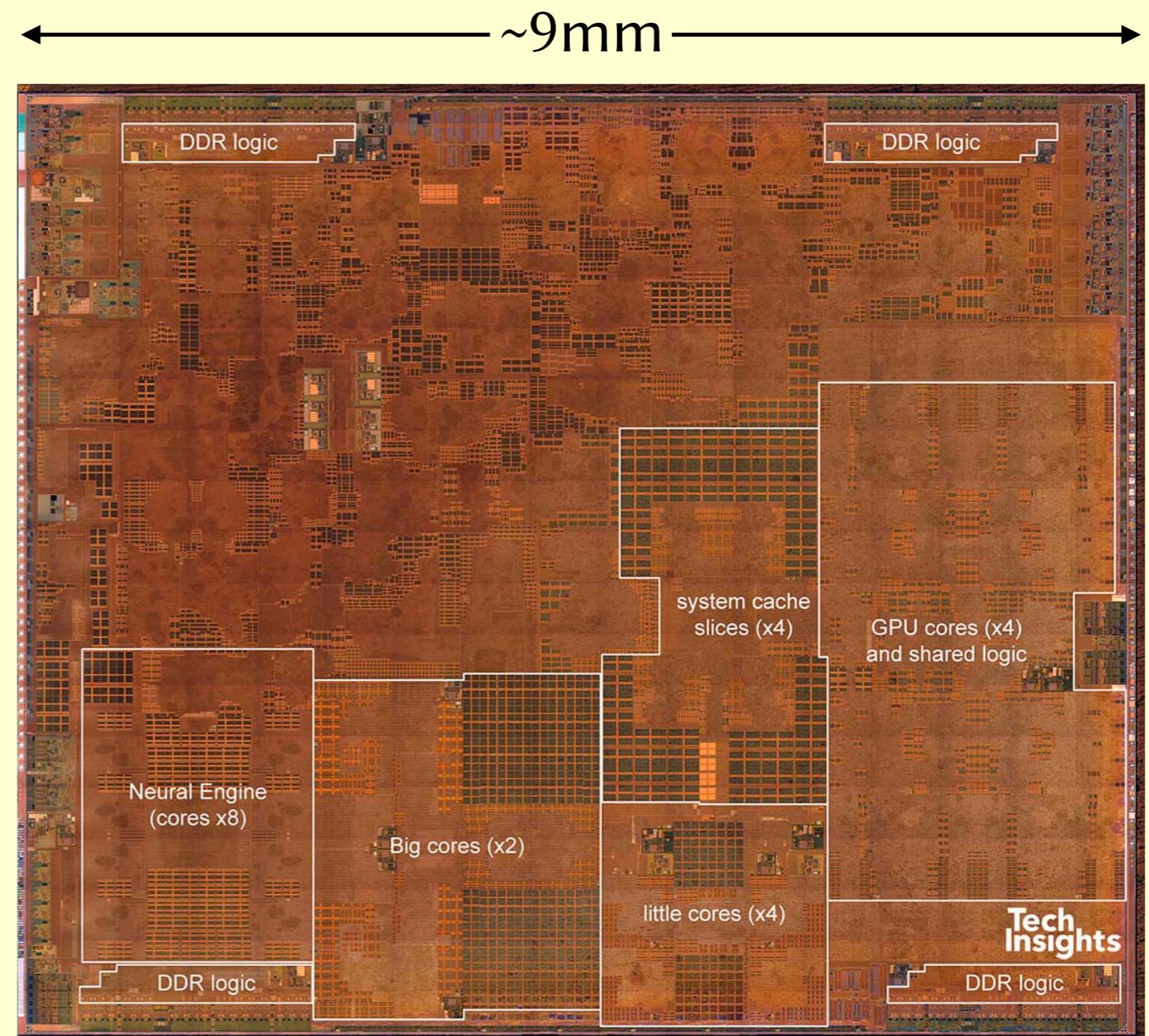
# Early integrated circuits

- Rows of gates, connected to form **application-specific integrated circuits** (ASICs).
- Can be **full-custom** (fabricated from scratch), or **semi-custom** (can only adjust the wiring between rows).
- Functionality cannot be later changed.



# Full-custom ICs

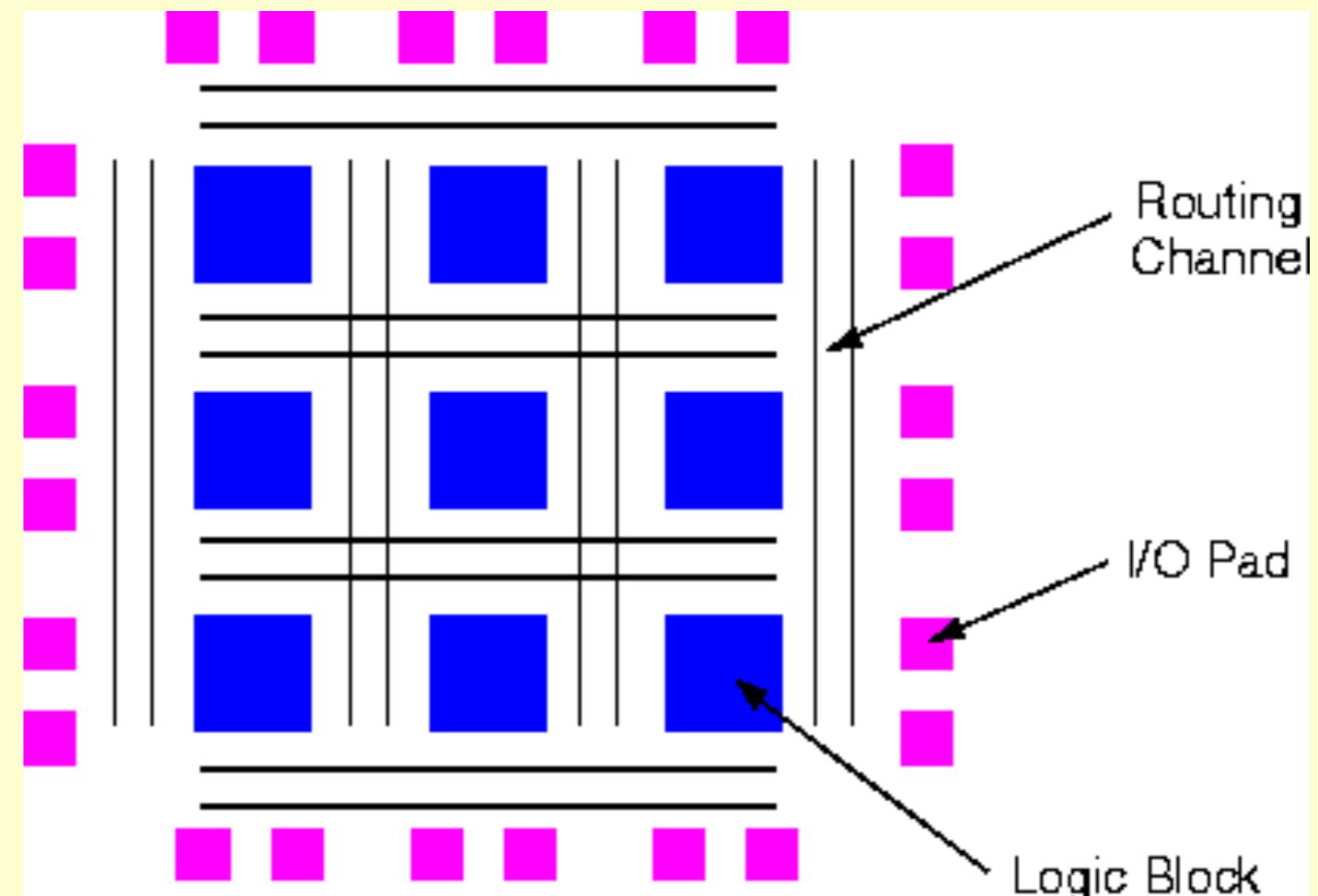
- Here is an Apple A12 processor from the iPhone XS. It has about 6.9 billion transistors.
- Very expensive to design and manufacture, so not viable unless the market is very large.



<https://www.techinsights.com/blog/apple-iphone-xs-max-teardown>

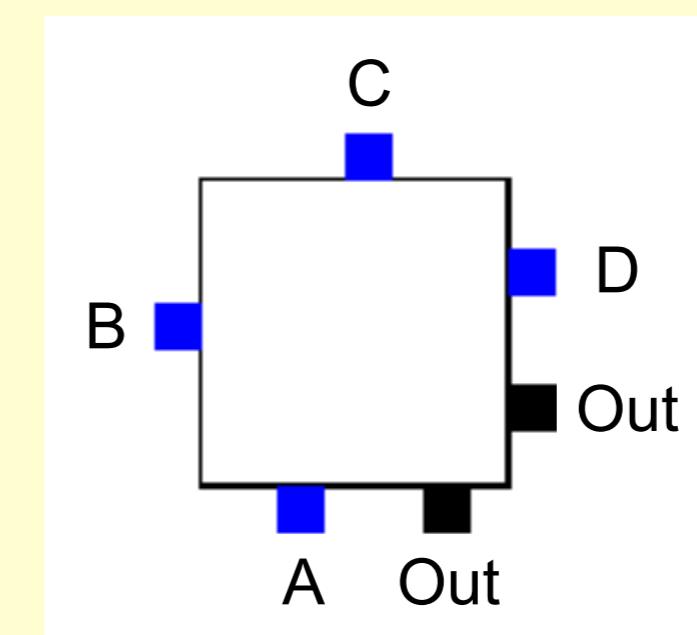
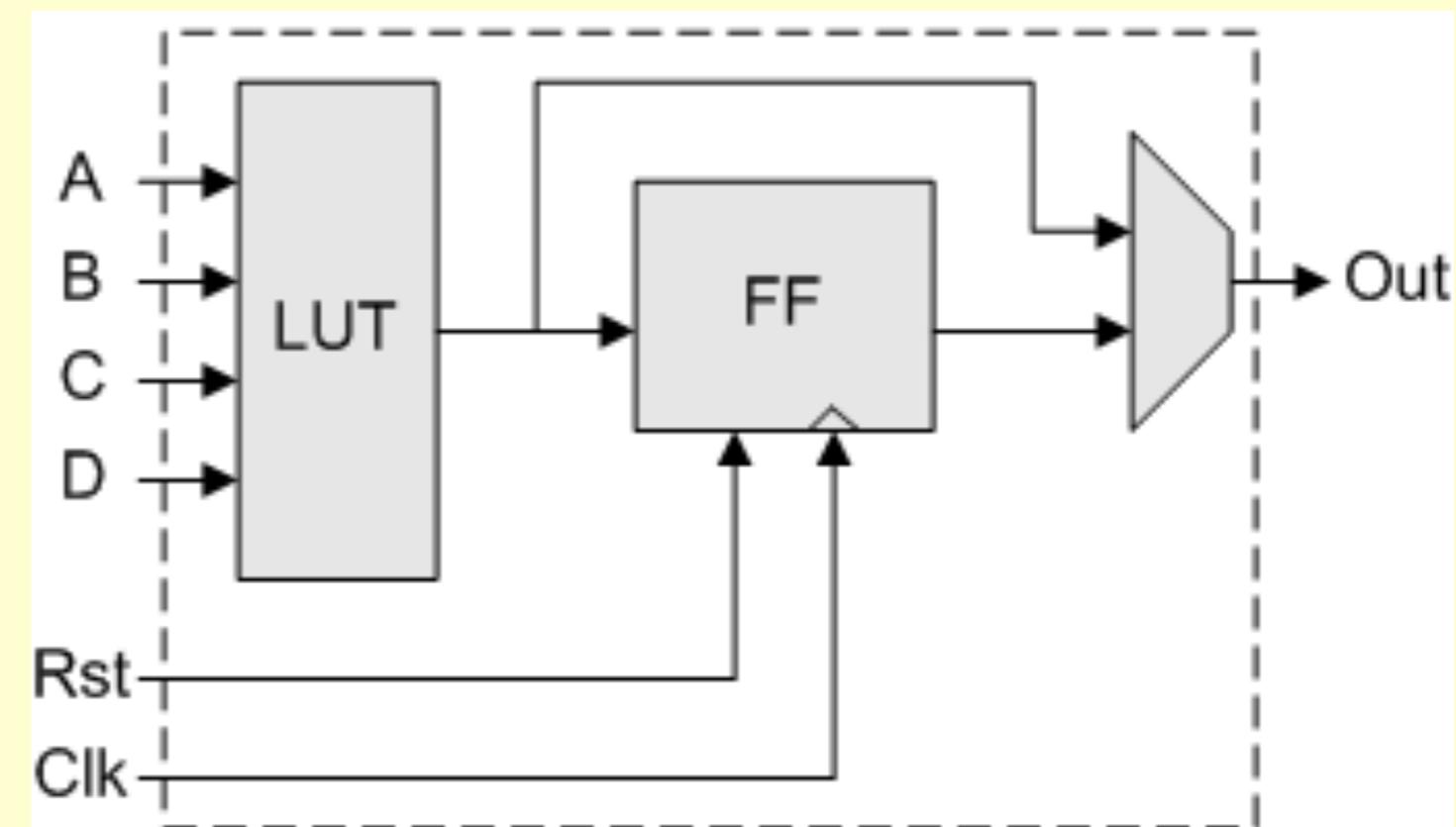
# Field-programmable gate arrays (FPGAs)

- Introduced by Xilinx in 1985. Other manufacturers include Altera (now part of Intel), Microsemi (now part of Microchip), and Lattice.
- Arrays of programmable logic blocks, programmable wiring in between, and flexible I/O.



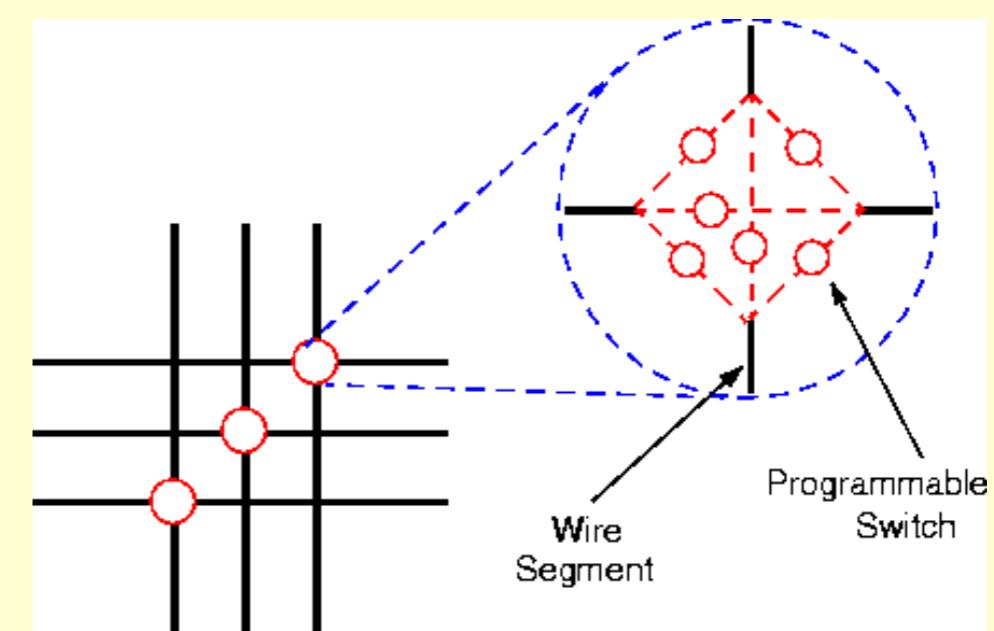
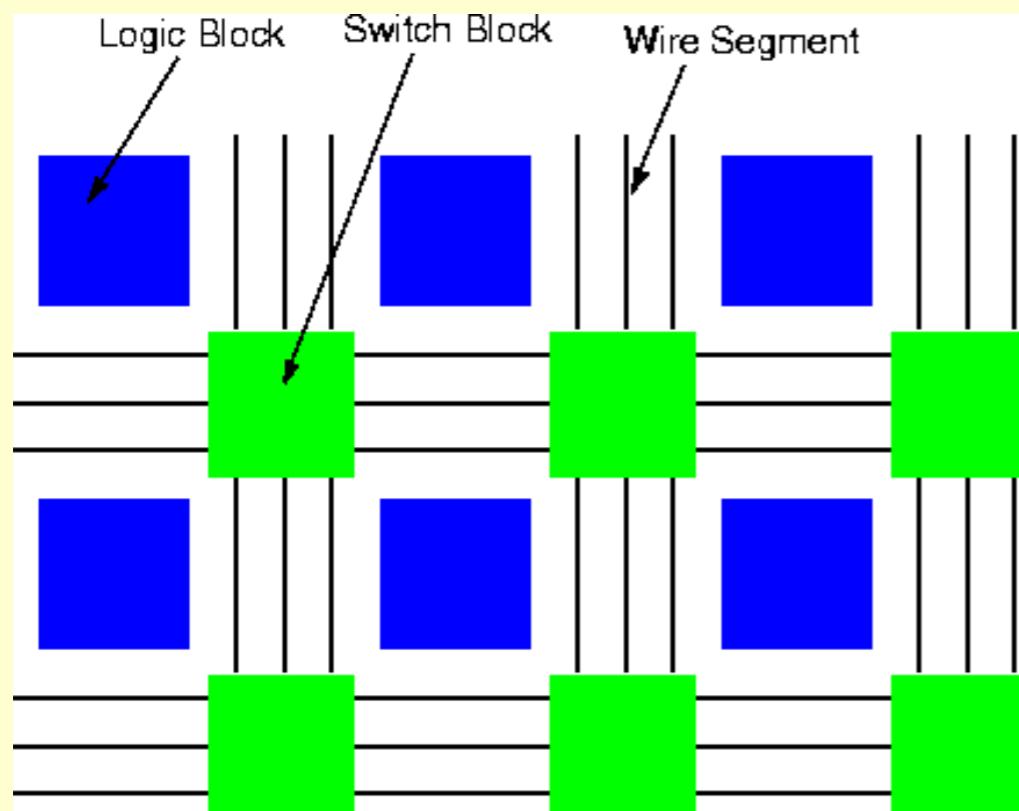
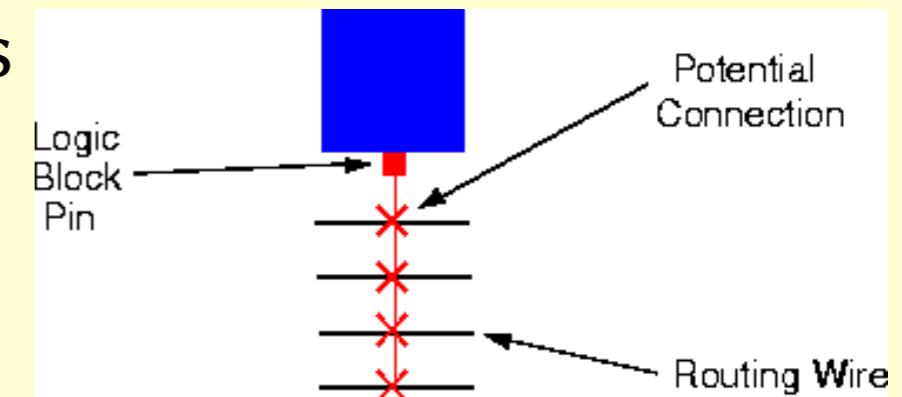
# Logic blocks

- Also called Logic Elements.
- Based around **look-up tables** (LUTs), usually with 4 inputs.
- These can implement *any* 4-input Boolean function (truth table).
- Optional flipflop (FF) at the output.



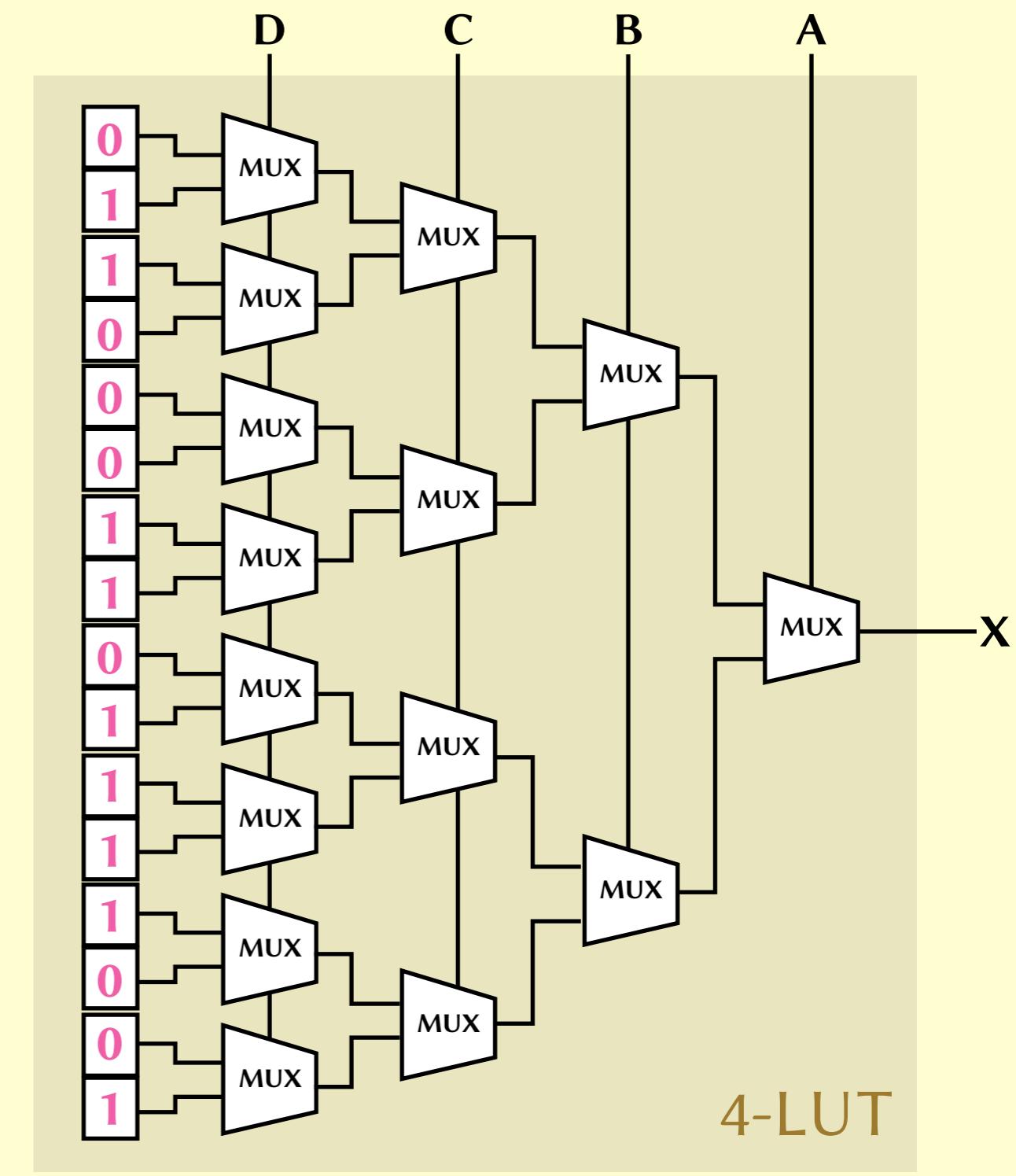
# Programmable routing

- Between rows and columns of logic blocks are programmable **wiring channels**.
- These are also **switch blocks** where the channels intersect.

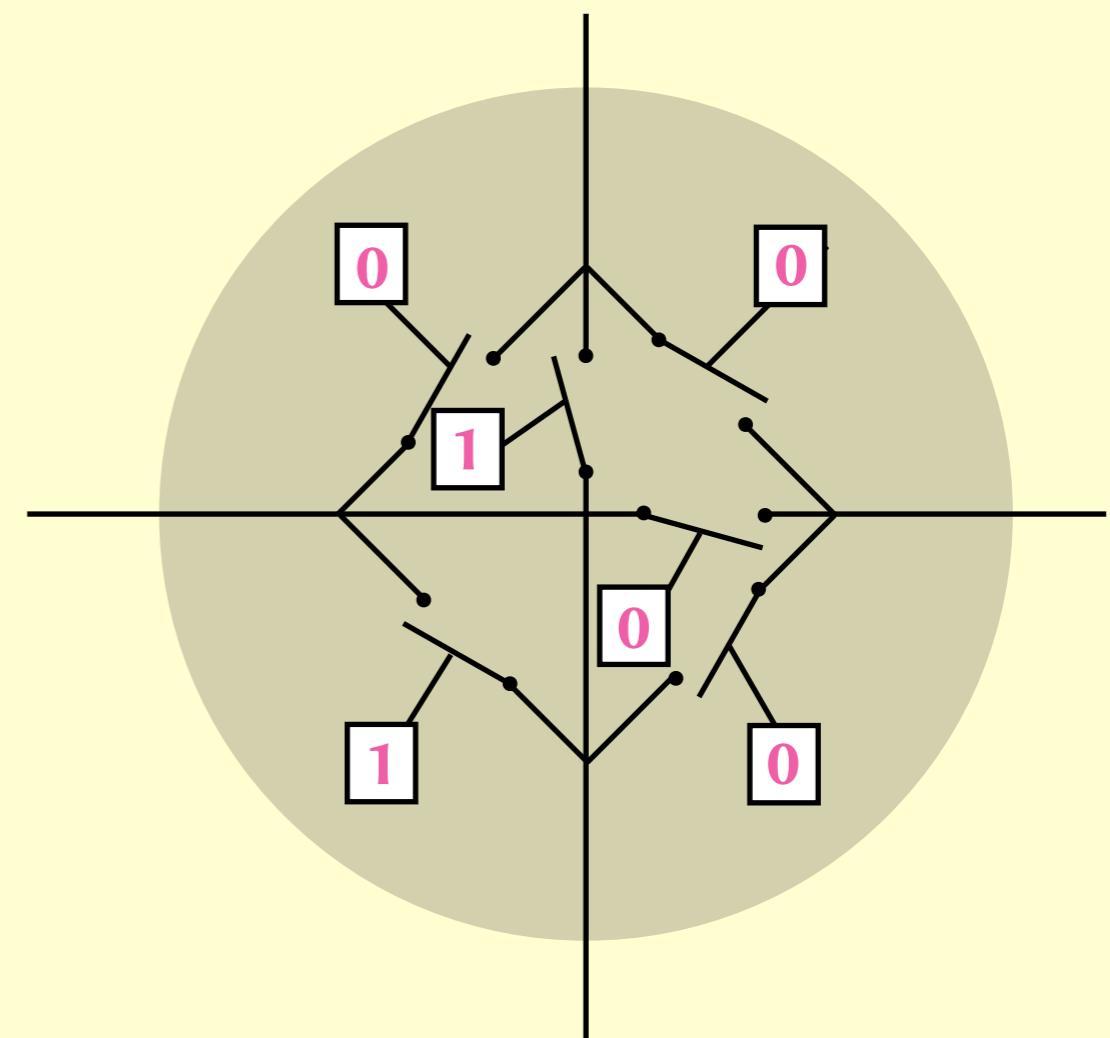
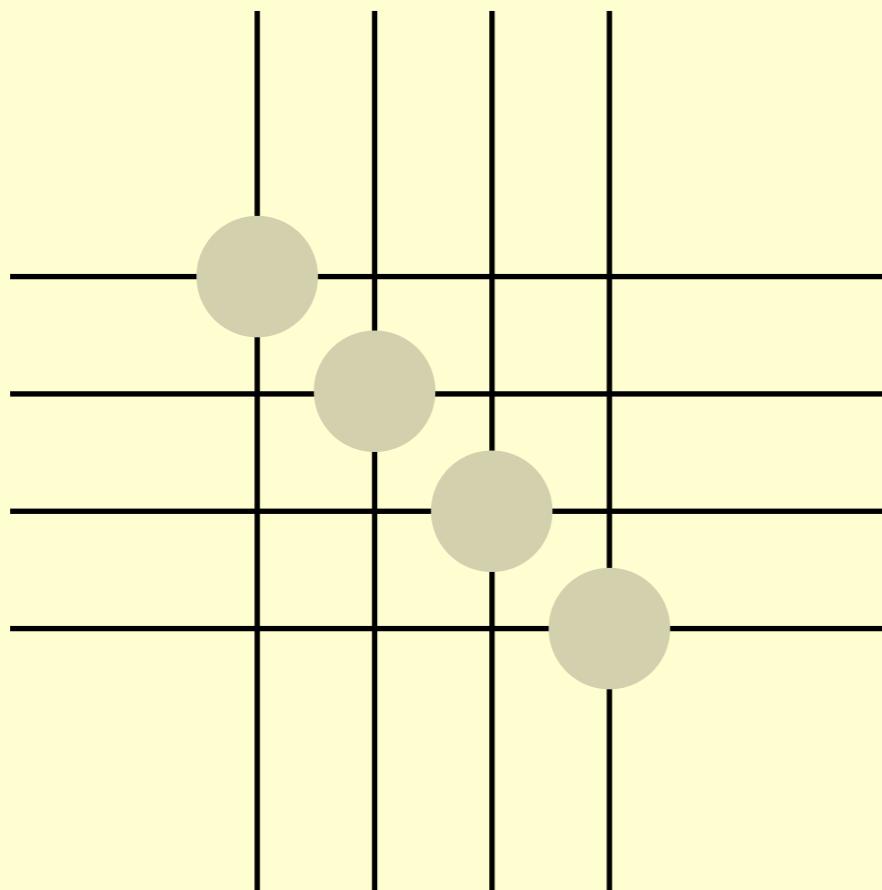


# Configuring an FPGA

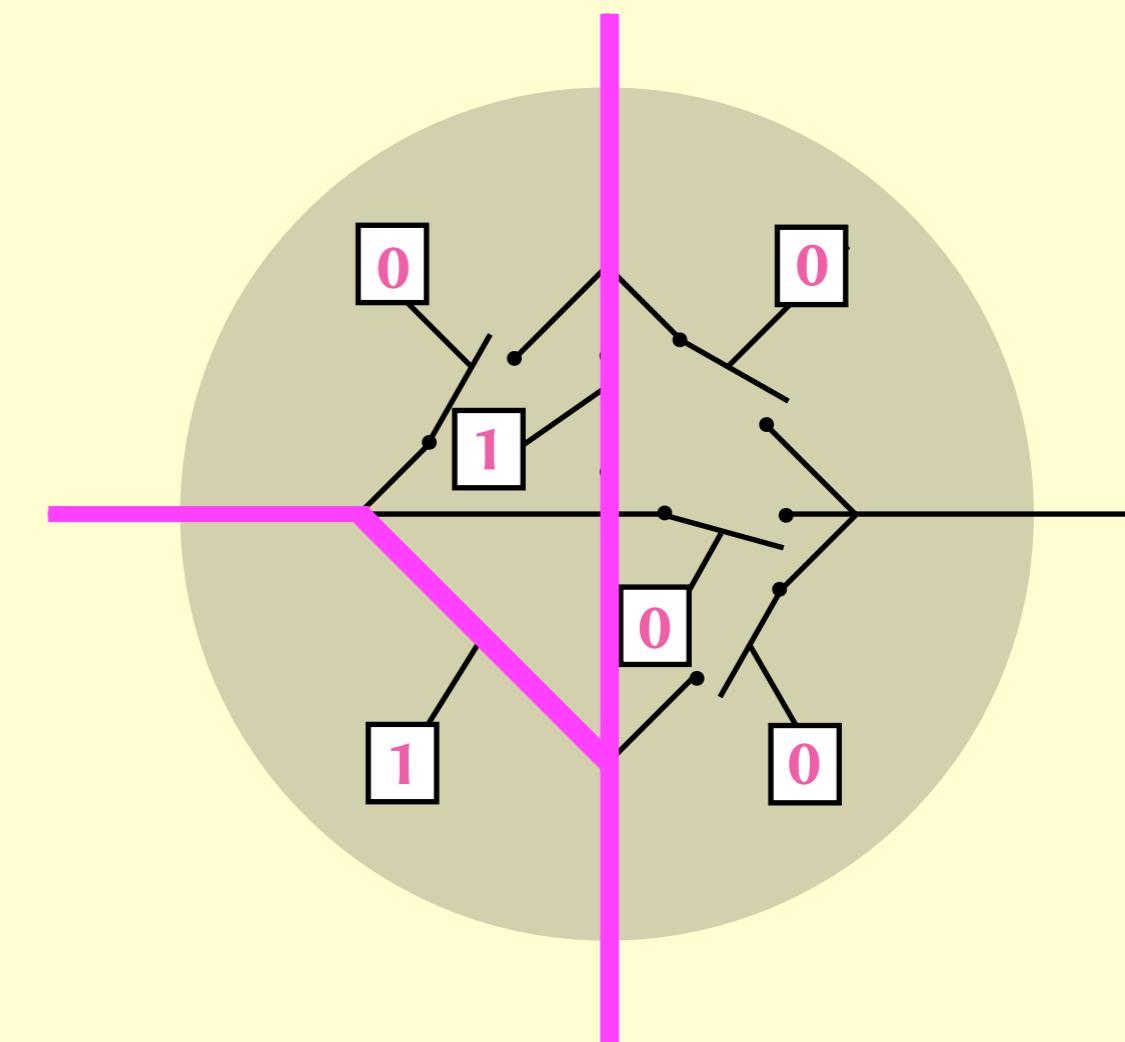
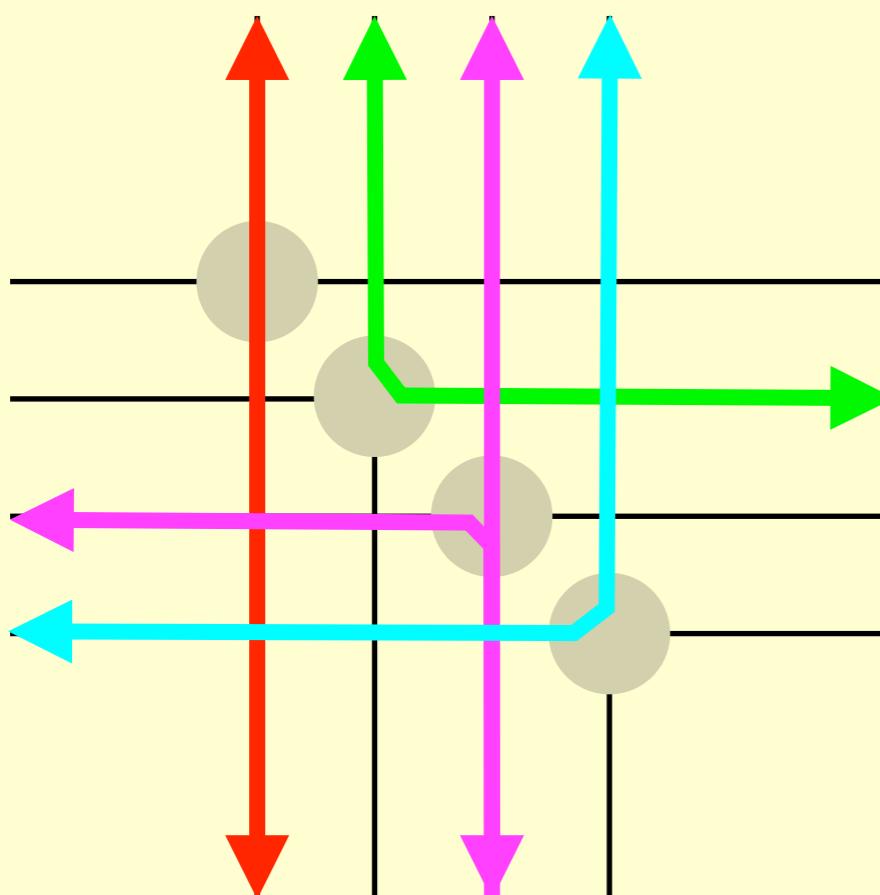
A	B	C	D	X
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1



# Configuring an FPGA



# Configuring an FPGA

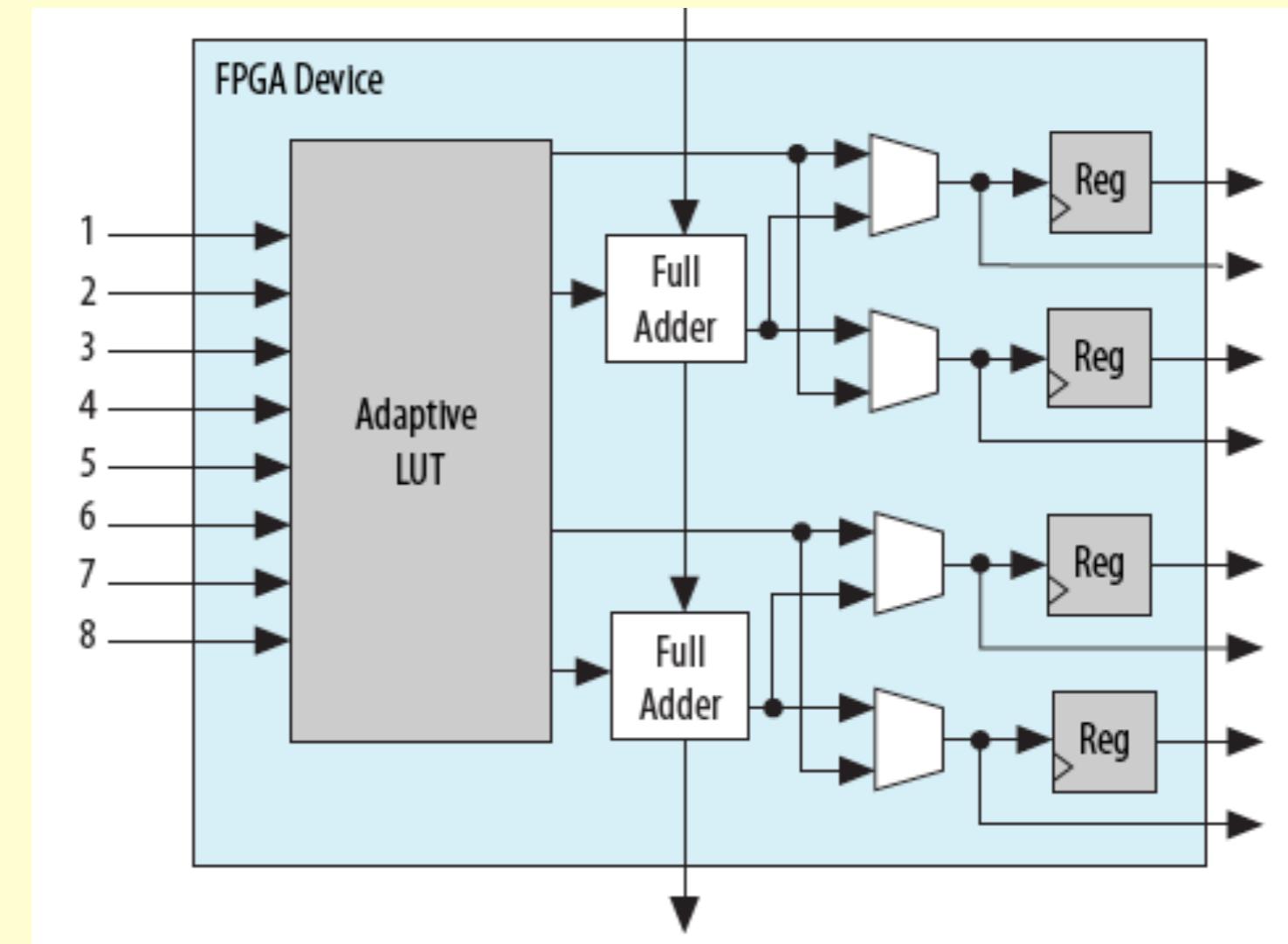


# Configuring an FPGA

- All these configuration registers are set by passing the FPGA a **configuration bitstream**.
- Powering up the DE1-SoC board puts the Cyclone V FPGA chip into a "waiting" mode (LEDs flashing). You then send the board the bitstream of your design via the USB port.

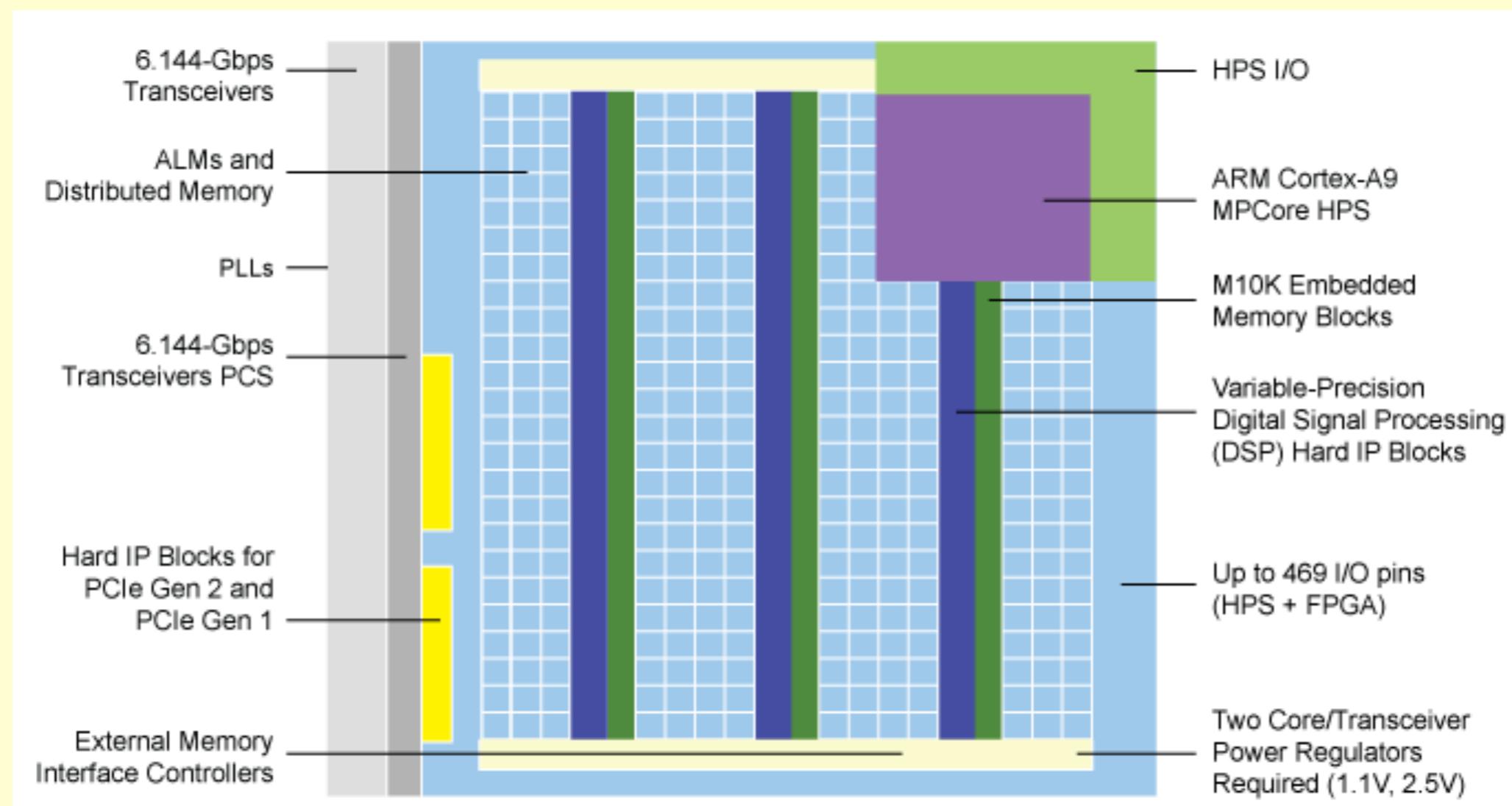
# Cyclone V FPGA

- The Altera Cyclone V FPGA uses a more complex FPGA logic blocks called Adaptive Logic Modules (ALMs).
- The device we use (5CSEMA5F31C6N) has 32000 ALMs. That's roughly equivalent to 85000 4-LUTs, or about 2000 32-bit binary adders.
- Not that this particularly matters for FPGA *users*.



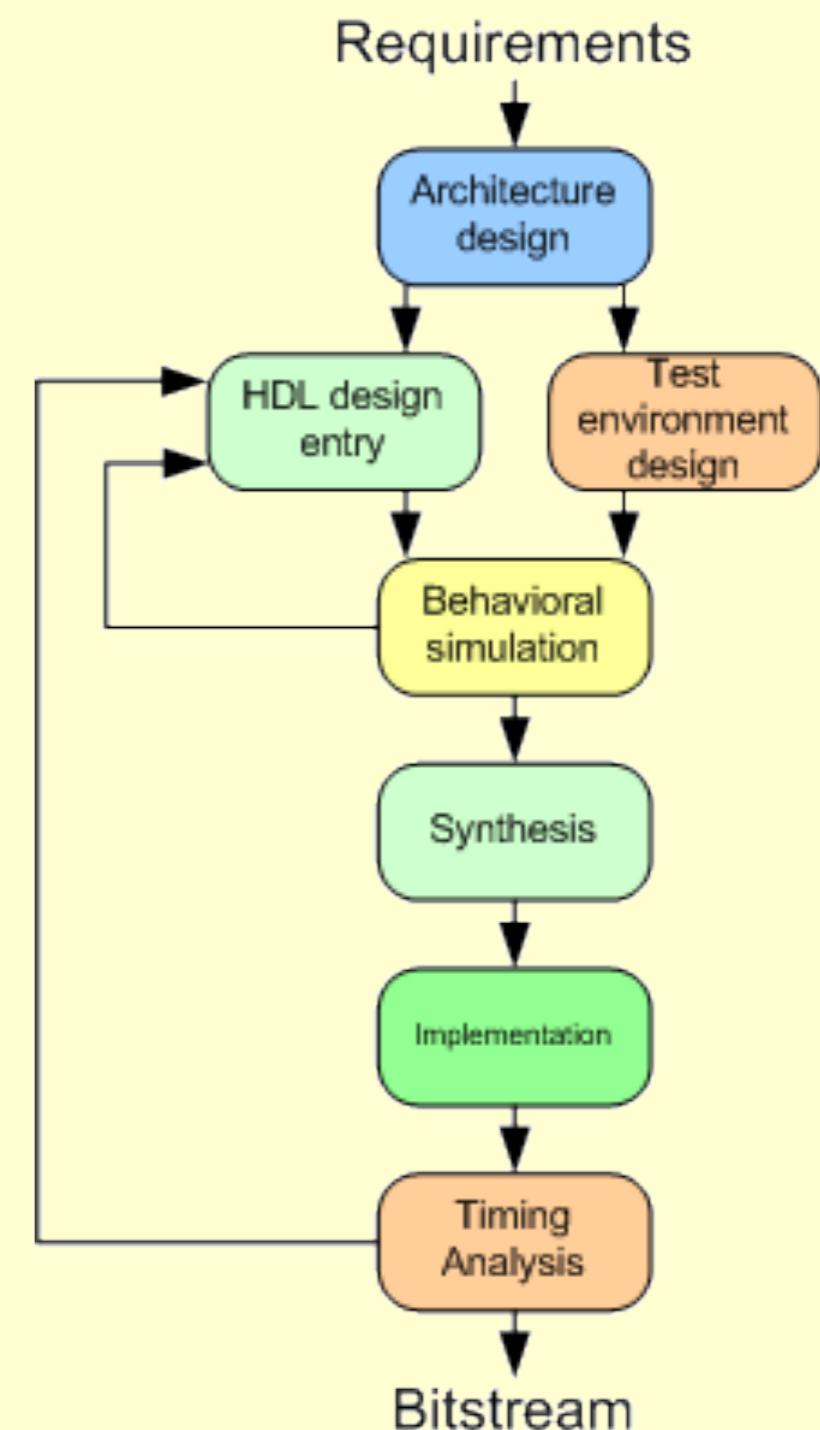
# Cyclone V FPGA

- The Cyclone V also has 4MB of embedded memory, 87 DSP blocks (for multiply-accumulate), a 2-core ARM processor, and hard logic for interfacing with PCIe and external memory.



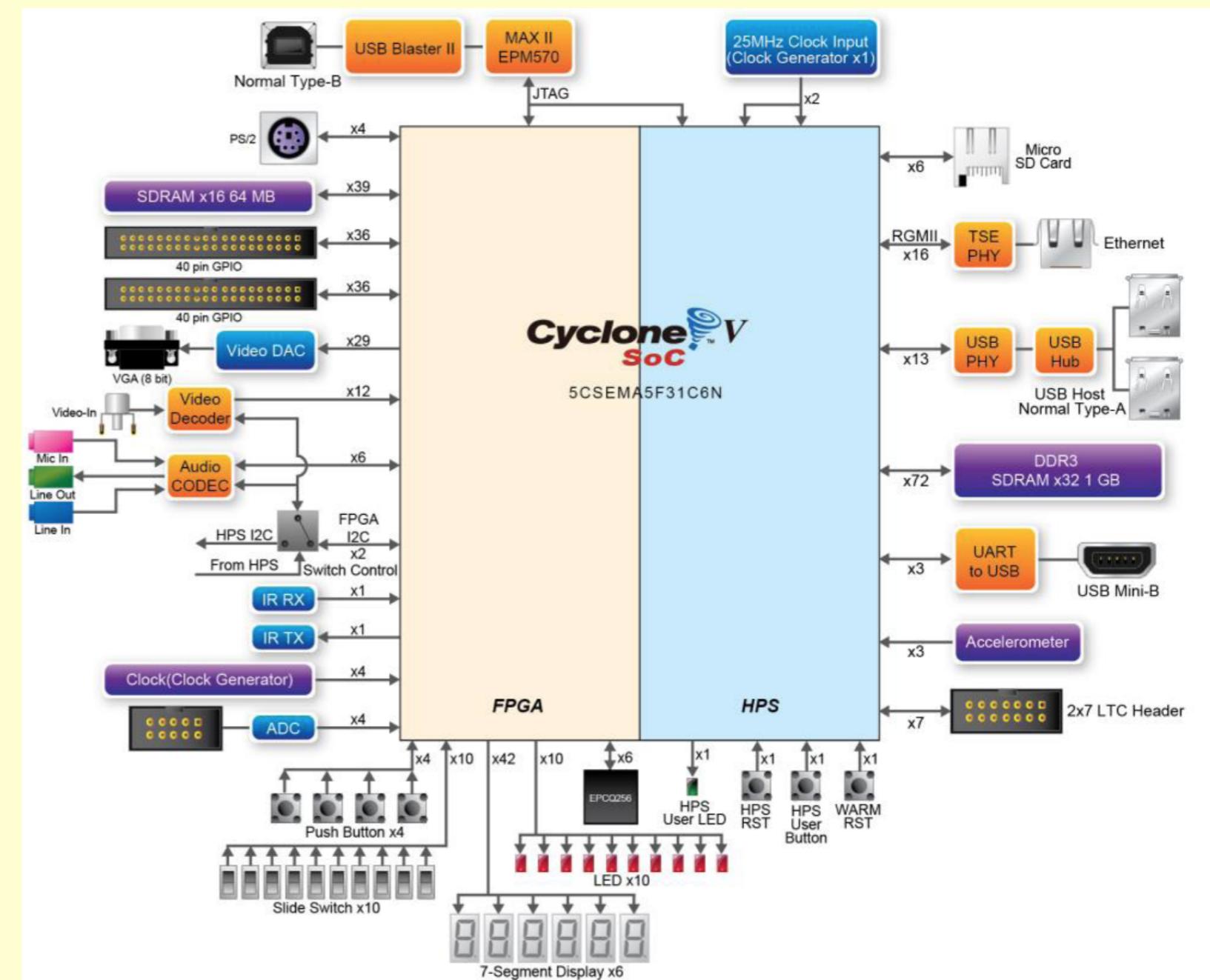
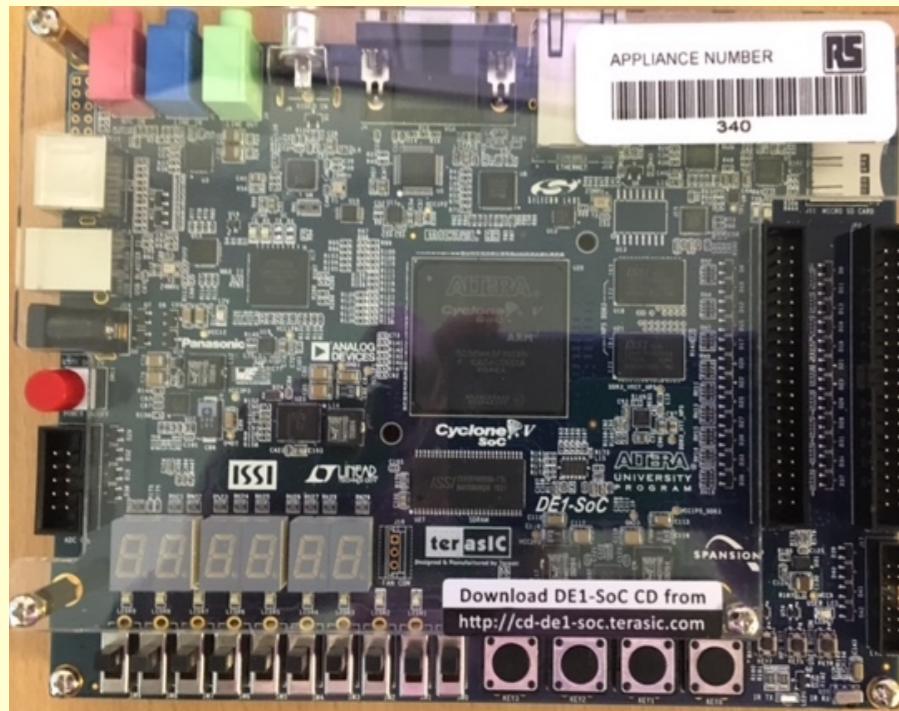
# Design Tools

- We use Altera Quartus II to design for FPGA.
- Features include: design entry, compilation from HDL, synthesis, simulation, timing analysis, power analysis, project management.
- The software is installed on all the PCs in the department. You might manage to run it on your own machine, but beware that you'll want about 8 GB disk space.



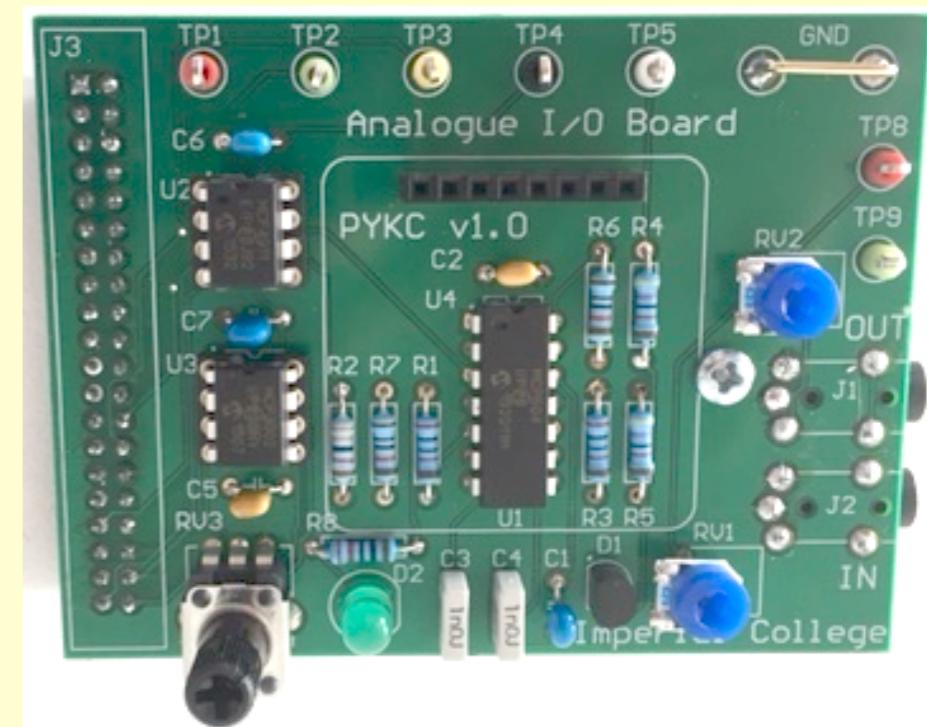
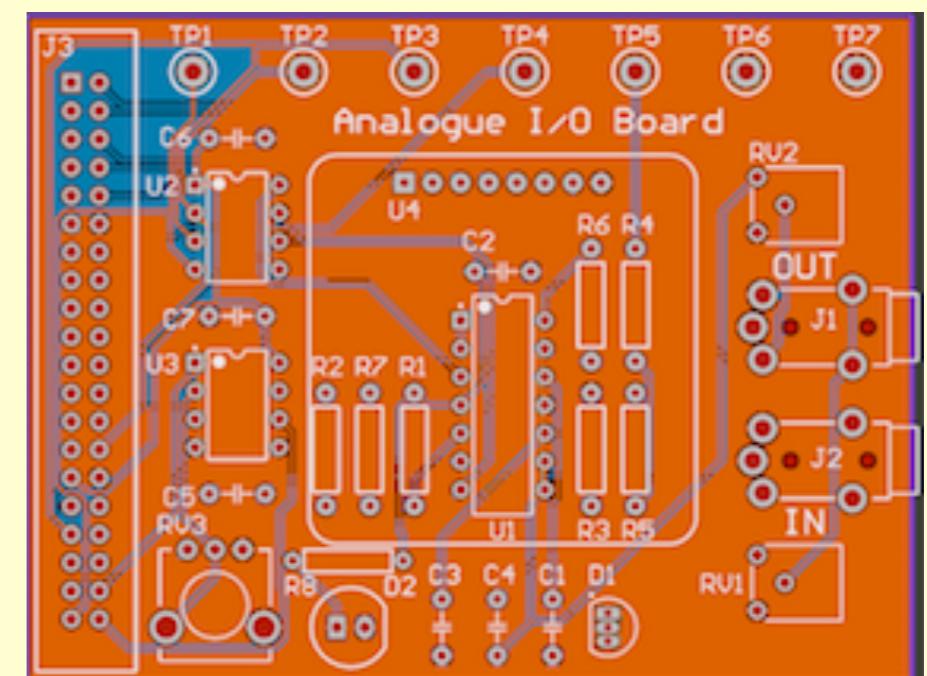
# The DE1-SoC board

- Everything you need to test basic designs involving switches, 7-segment displays, and even a VGA output.



# Add-on board

- Provides analogue inputs (from microphone) and outputs (to headphone jack).
- Will feature in the third and fourth parts of the lab – you can ignore it for now.



# Verilog

- A **hardware description language** based on C syntax.
- Description can be at **behavioural level, register-transfer level (RTL), or gate level**.
- May be easier to learn than its competitor, VHDL. Other HDLs include Chisel and Bluespec.

# Schematic vs HDL

## Schematics:

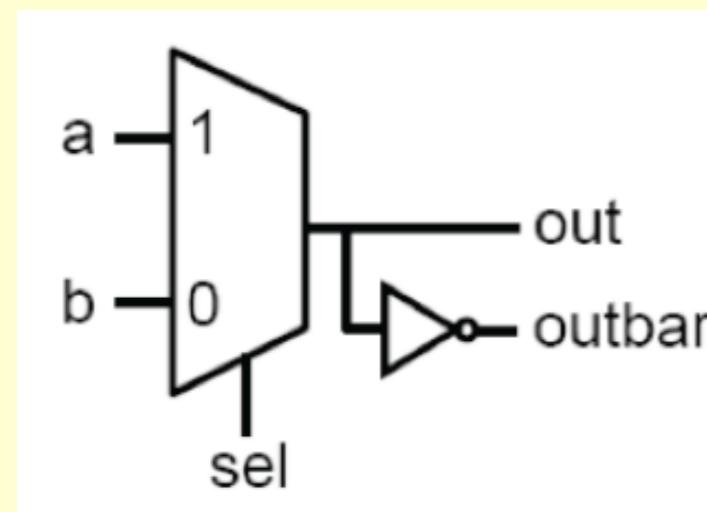
- Provide overview
- Direct connection to hardware
- No programming needed

## HDL:

- Flexible and parameterisable
- Simple textual format
- Connection to high-level languages

# Verilog modules

- A Verilog design consists of many interconnected **modules**.
- A module contains low-level gates, statements, and other modules.



a	b	sel	out	outbar
0	0	0	0	1
0	0	1	0	1
0	1	0	1	0
0	1	1	0	1
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0
0	0	0	0	1
0	0	1	0	1
0	1	0	1	0
0	1	1	0	1
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

```
// 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
```

# Verilog syntax

Specify port as **input**/  
**output/inout**

Express module's behaviour.  
Each statement executes in  
parallel. Order doesn't matter.

This is continuous assignment.  
Output responds to changes in  
input **immediately**.

Declare module,  
list its ports,  
terminate with ;

Comments can also  
use **/\* ... \*/**

```
// 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
```

Other operators include: **&**, **|**, **+**, **-**, **\***

# Gate-level description

- Built-in logic gates include: **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not**, **buf**.

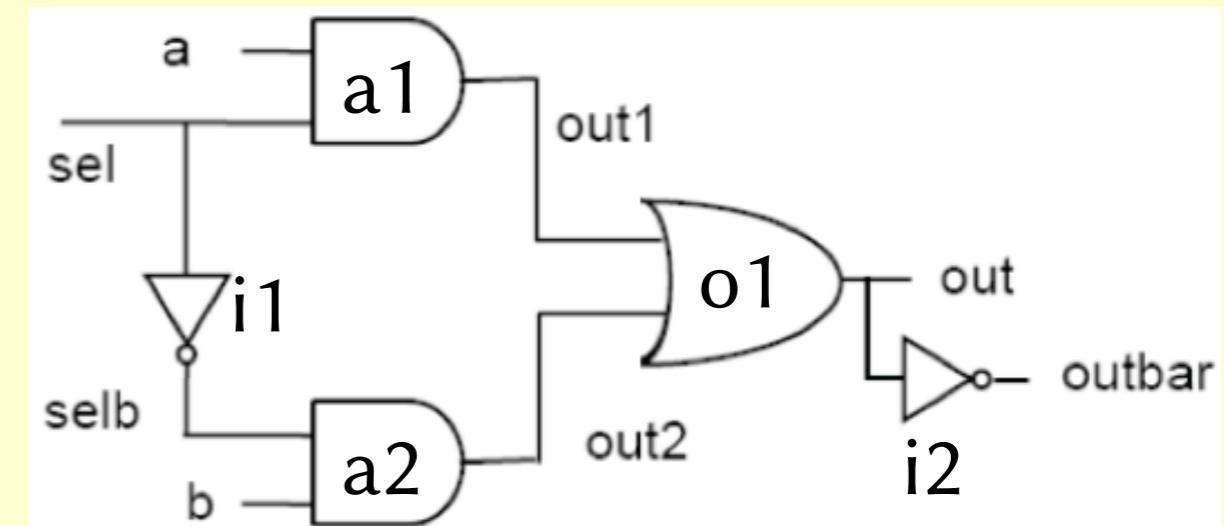
```
// Function: 2-to-1 multiplexer as gates

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
```



- In practice, "gate-level description" depends on what primitives are provided by your target chip.

# Always blocks

- Verilog provides **always** blocks for defining **sequential** logic.

Variables (not registers)

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

    reg out, outbar;

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

            outbar = ~out;
        end
endmodule
```

Sensitivity list

Can also contain  
**case, for, while**

Executed in sequence  
(order matters!)

# Always blocks

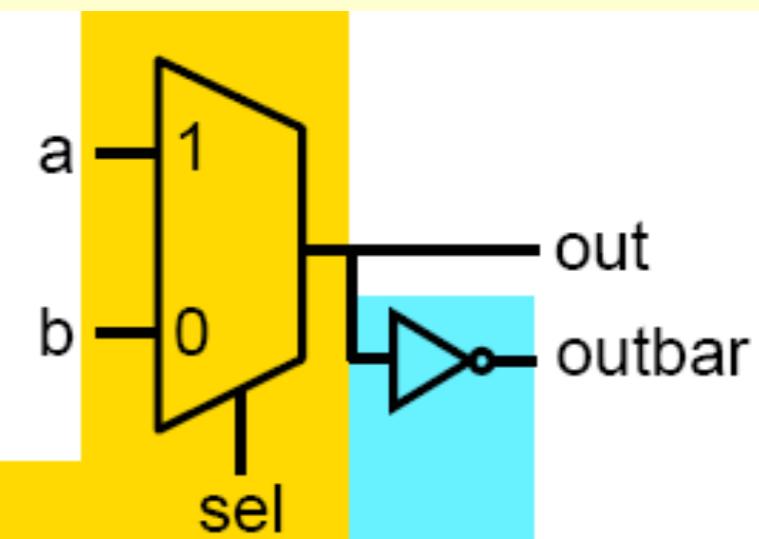
- Can mix procedural and continuous assignments.

```
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*

# Numerical constants

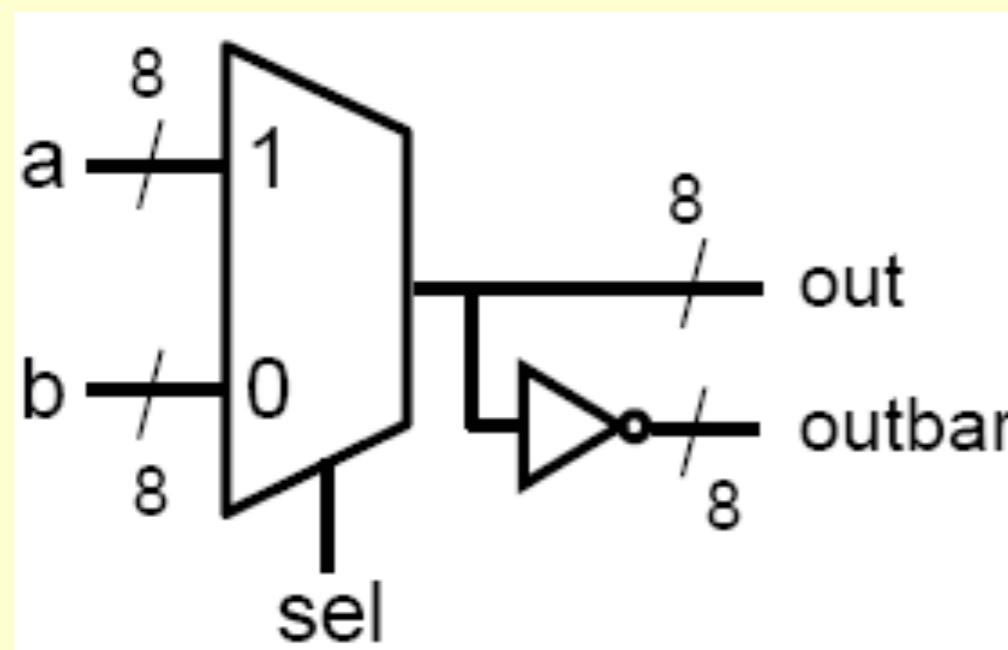
- Format is  $<\text{size}>^{\prime} <\text{signedness}><\text{base}><\text{number}>$
- $<\text{size}>$  defaults to **32**.
- $<\text{base}>$  is either **b** (binary=2), **o** (octal=8), **d** (decimal=10), or **h** (hexadecimal=16). Decimal is the default.
- $<\text{signedness}>$  is either blank (unsigned, default) or **s** (signed).
- Examples: **2'b10**, **'b10**, **31**, **8'hAf**, **2'sb10**

# Buses

- Concatenate wires into a **bus** using `{...}` syntax. E.g.:

```
input [15:0] x;  
output [15:0] y;  
assign y = { x[7:0], x[15:8] };
```

- An 8-bit multiplexer:

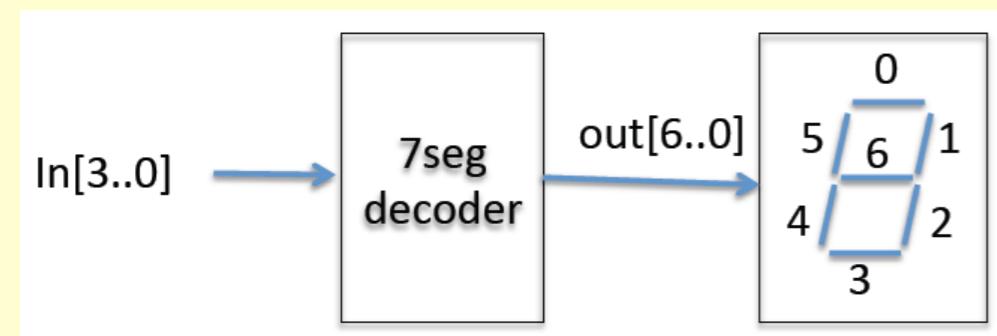


```
module mux_2_to_1(a, b, out,  
                    outbar, sel);  
    input[7:0] a, b;  
    input sel;  
    output[7:0] out, outbar;  
    reg[7:0] out;  
    always @ (a or b or sel)  
    begin  
        if (sel) out = a;  
        else out = b;  
    end  
    assign outbar = ~out;  
endmodule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

# Case statements

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

# 7-seg decoder



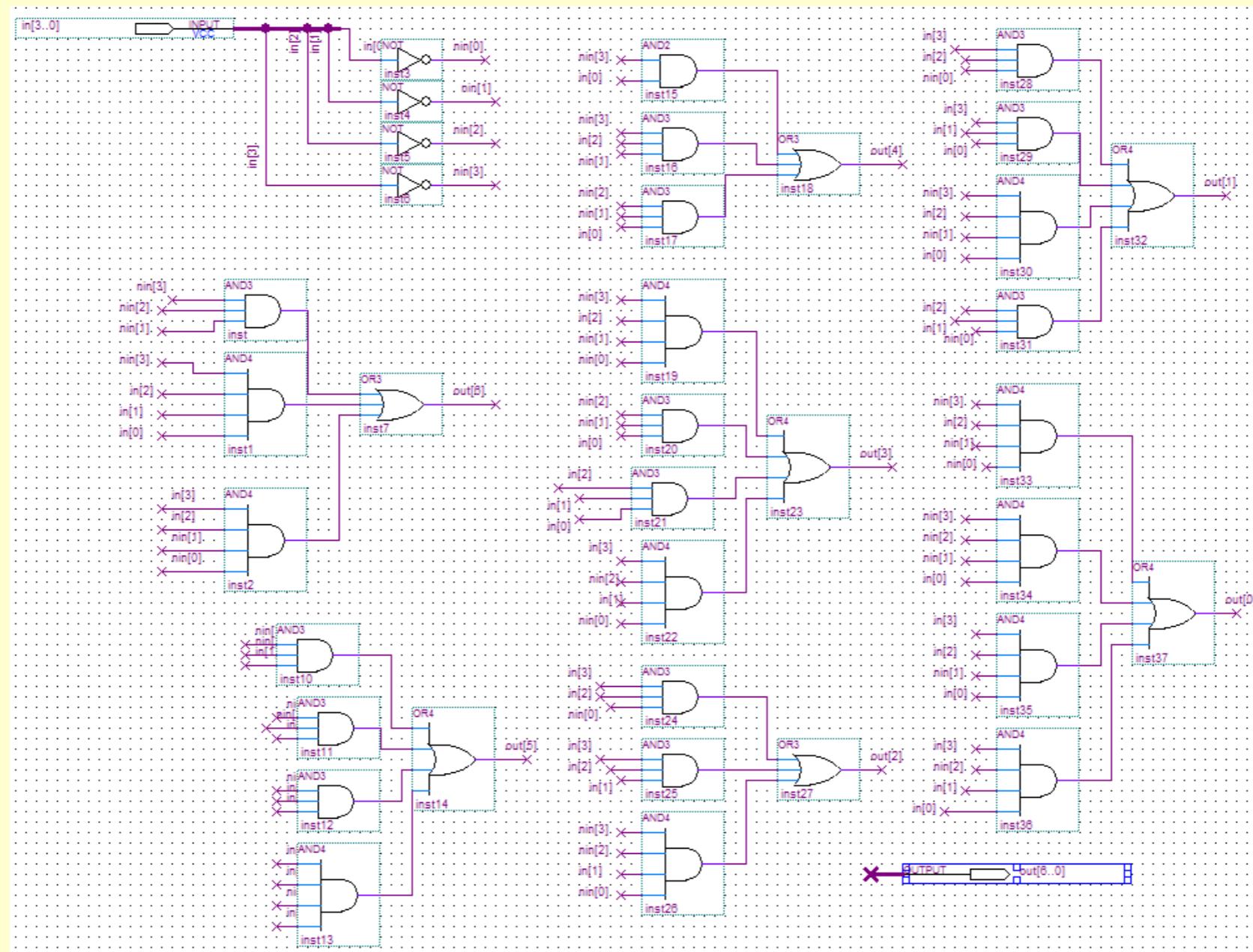
<code>in[3..0]</code>	<code>out[6:0]</code>	Digit	<code>in[3..0]</code>	<code>out[6:0]</code>	Digit
0000	1000000	0	1000	0000000	B
0001	1111001	/	1001	0010000	9
0010	0100100	2	1010	0001000	A
0011	0110000	3	1011	0000011	b
0100	0011001	4	1100	1000110	C
0101	0010010	5	1101	0100001	d
0110	0000010	6	1110	0000110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

<code>In3 : in2</code>				
<code>out6</code>	00	01	11	10
00	1	0	1	0
01	1	0	0	0
11	0	1	0	0
10	0	0	0	0

$$\text{out6} = (\overline{\text{in3}} \wedge \overline{\text{in2}} \wedge \overline{\text{in1}}) \vee (\text{in3} \wedge \text{in2} \wedge \overline{\text{in1}} \wedge \overline{\text{in0}}) \vee (\overline{\text{in3}} \wedge \text{in2} \wedge \text{in1} \wedge \text{in0})$$

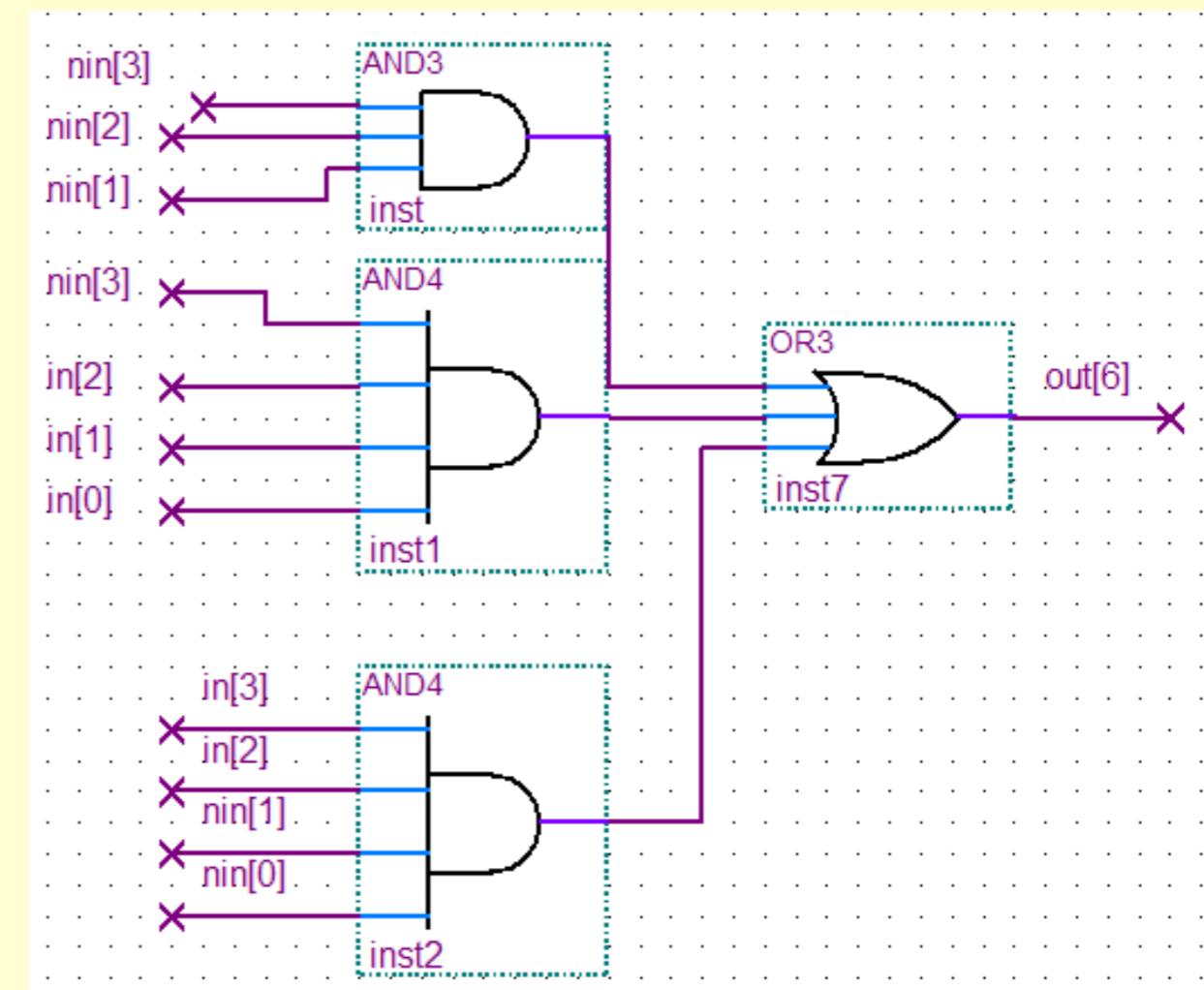
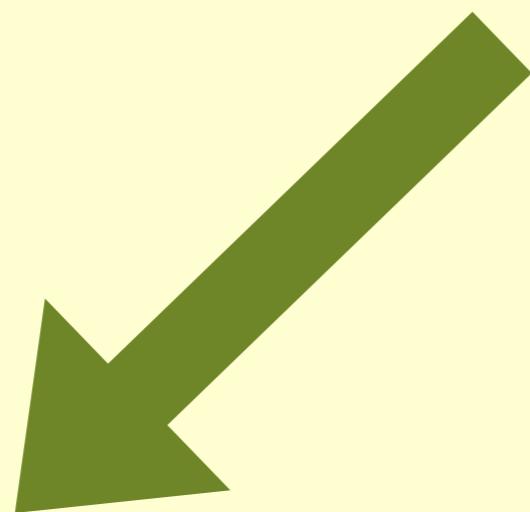
# First attempt

- Draw a schematic:



# Second attempt

- Gate-level description:



```
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);
```

# Third attempt

- Use continuous assignment:

```
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);
```



```
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];
```

```
//-----
// Module name: hex_to_7seg
// Function: convert 4-bit hex value to drive 7 segment display
//           output is low active
// Creator: Peter Cheung
// Version: 1.0
// Date:    22 Oct 2011
//-----

module hex_to_7seg  (out,in);

    output [6:0]    out;      // low-active output to drive 7 segment display
    input  [3:0]    in;       // 4-bit binary input of a hexadematical number

    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];
    assign out[5] = ~in[3]&~in[2]&in[0] | ~in[3]&~in[2]&in[1] |
                  ~in[3]&in[1]&in[0] | in[3]&in[2]&~in[1]&in[0];
    assign out[4] = ~in[3]&in[0] | ~in[3]&in[2]&~in[1] | in[3]&~in[2]&~in[1]&in[0];
    assign out[3] = ~in[3]&in[2]&~in[1]&~in[0] | ~in[3]&~in[2]&~in[1]&in[0] |
                  in[2]&in[1]&in[0] | ~in[2]&in[1]&~in[0];
    assign out[2] = ~in[3]&~in[2]&in[1]&~in[0] | in[3]&in[2]&~in[0] |
                  in[3]&in[2]&in[1];
    assign out[1] = in[3]&in[2]&~in[0] | ~in[3]&in[2]&~in[1]&in[0] |
                  in[3]&in[1]&in[0] | in[2]&in[1]&~in[0];
    assign out[0] = ~in[3]&~in[2]&~in[1]&in[0] | ~in[3]&in[2]&~in[1]&~in[0] |
                  in[3]&in[2]&~in[1]&in[0] | in[3]&~in[2]&in[1]&in[0];

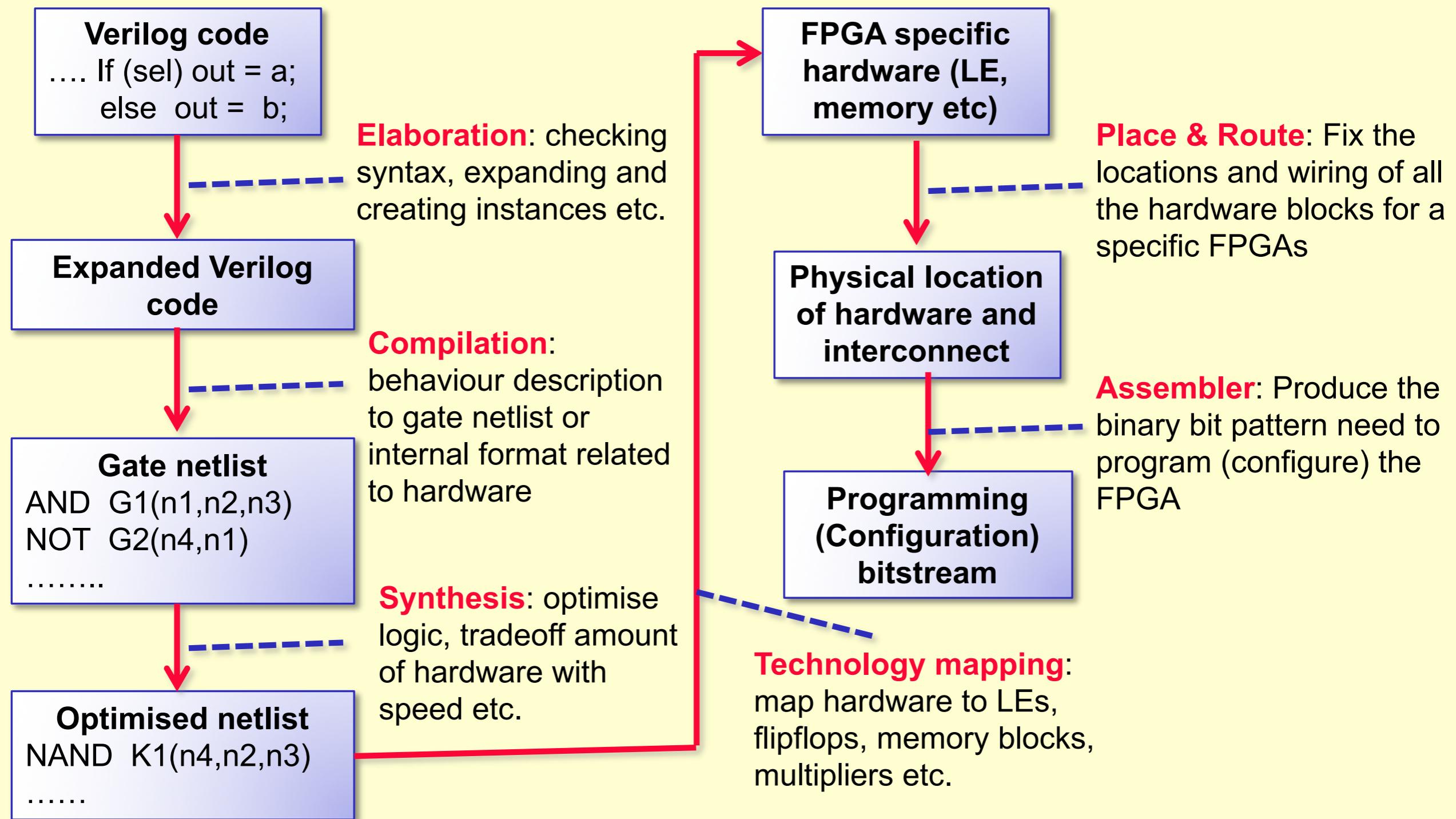
endmodule
```

# Fourth attempt

```
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
```

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

# From Verilog to Hardware

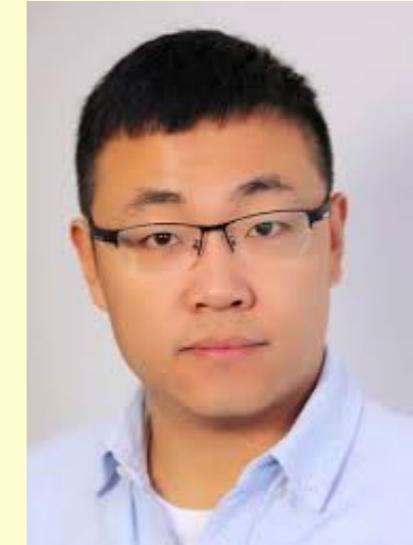
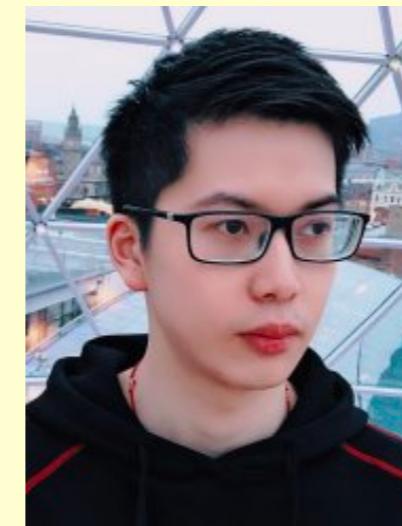


# Lab groups

1. Andreas Michaelides + Jinzhao Han
2. Jiaxiang Li + Jiacheng Zhang
3. Liqi Sun + Lijie Xie
4. Lizhong Yan + Mingxuan Du
5. Qishen Guan + Qirui Hua
6. Qingchun Lyu + Qifan Zhu
7. Shihong Chen + Siyuan Feng
8. Shumin Wang + Shuang Wu
9. Wei Tu + Xueteng Li
10. Yiyang Chen + Yuxin Huang
11. Yusong Liu + Yiqun Xue
12. Yaxing Xu + Yifan Zhang
13. Zewei Du + Zijian Gao
14. Zhengyu Wang + Zixiang Zhu

# Admin

- Lectures: Fridays at 0900–1100, for three more weeks.
- Labs: Fridays at 1300-1600, today and the next three weeks.
- Teaching assistants: Mr Jianyi Cheng and Mr He Li.



- Course webpage:  
[https://github.com/johnwickerson/mastering\\_digital\\_design](https://github.com/johnwickerson/mastering_digital_design)