模拟与数字电路

Analog and Digital Circuits

课程主页 扫一扫

第 七 讲： **Verilog 、 FPGA 介绍**

Lecture 7: **Introduction on Verilog & FPGA**

主 讲： 陈迟晓

Instructor : Chixiao Chen

提纲

- 复习
  - Verilog always 块结构
- Verilog语法 续
- Verilog 应用举例
- FPGA简介

基于下列Verilog代码求模块逻辑，并化简

```
module truth_table_verilog_example (F, A, B, C);
        input A, B, C;
        output F;
        reg F;

        always @(A or B or C)
        begin   //begin the procedural statements
                case ({A,B,C})  //variables that affect the procedure

                                        //the 3'b simply means that we're
                        3'b000: F= 'b0;  //considering three bit binary variable
                        3'b001: F= 'b0;
                        3'b010: F= 'b1;
                        3'b011: F= 'b0;
                        3'b100: F= 'b1;
                        3'b101: F= 'b0;
                        3'b110: F= 'b1;
                        3'b111: F= 'b0;
                endcase
        end     //end the begin statement

endmodule
```

3.5.1 常数

- 考虑带有进位的加法器，一种方法是手工扩展输入1位，执行常规加法，截取和的最高位作为进位，代码如例3.12所示

例3.12 固定位宽的加法器描述

```
module adder_carry_hard_lit
      (
        input wire [3:0] a, b,
        output wire [3:0] sum,
        output wire cout    // 进位
      );
        wire [4:0] sum_ext ;
        assign sum_ext = {1'b0, a} + {1'b0, b};
        assign sum = sum_ext [3:0] ;
        assign  cout= sum_ext [4] ;
 endmodule
```

- 代码描述的是4位加法器，固定值如3和4用以表示范围。
- 如果我们想改成8位加法器，这些字都要手工修改，如果代码很复杂而且这些文字出现在很多地方，这将是一个**繁琐且容易出错**的过程。

3.5.2 参数

- Verilog模块可以实例化为组件并成为更大设计的一部分。 Verilog提供一种结构称为parameter ，向模块传递信息，这种机制使得模块多功能化并能重复使用。参数在模块内不能改变，因此功能与常数类似。
- 在Verilog-2001中，参数声明部分可以在开头，在端口声明之前，其语法如下：

```
module [模块名]
      # (
      parameter [参数名] = [默认值] ,
                [参数名] = [默认值] ;
       )
       (
         . . .
       );
```

3.5.2 参数

- 例3.13可以改成采用参数的加法器，如例3.14所示。参数N声明为默认值4，N声明之后，就可以像常数一样在端口声明和模块体中使用。

例3.14 使用参数的加法器描述

```
module adder_carry_ para
   #（parameter N=4）
   (input  [N-1:0] a, b,
   output  [N-1:0] sum,
   output cout   );
        localparam  N1 = N-1;                    //常数声明
        wire  [N:0] sum_ext;
        assign  sum_ext = {1'b0, a} + {1'b0, b};
        assign  sum = sum_ext [N1:0] ;
        assign  cout= sum_ext [N] ;
endmudule
```

3.5.2 参数

例3.15 加法器例化的例子

```
module adder_insta
      (
      input   [3:0] a4, b4,
      output  [3:0] sum4,
      output  c4,
      input  [7:0] a8, b8,
      output  [7:0] sum8,
      output   c8
      );
      // 实例化8位加法器
      Adder_carry_para #(. N(8)) unit1
        (. a(a8), . b(b8), . sum(sum8), . cout (c8)) ;
      //实例化4位加法器
      Adder_carry_para unit2
        (.a(a4), .b(b4),.sum(sum4),.cout(c4));
   endmodule
```

- 可以在组件实例化中给参数赋所需的值并将原来的默认值覆盖
- 如果省略了参数赋值，则采用默认参数
- 参数提供了一种创建可扩展代码机制，可调整电路的位宽以适应特定的需求，这使代码移植性更好，有利于设计重用。

3.6 设计实例

- 本节给出了一些常用的组合电路的设计实例，包括：
  - **3.6.1 多路选择器**
  - **3.6.2 比较器**
  - **3.6.3 译码器**
  - **3.6.4 编码器和编码转换器**

3.6.1 多路选择器

- 多路选择器（Multiplexer）是一个**多输入、单输出**的组合逻辑电路，一个n输入的多路选择器就是一个n路的数字开关，可以根据通道选择控制信号的不同，从n个输入中选取一个输出到公共的输出端。

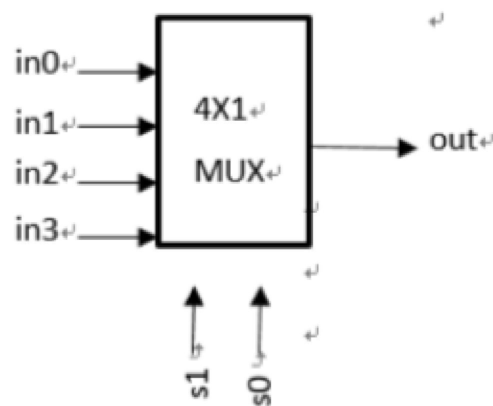


图3.2：4选1多路选择器的电路模型和真值表

s1	s0	out
0	0	in0
0	1	in1
1	0	in2
1	1	in3

- in0、in1、in2和in3是4个输入端口
- s1和s0是通道选择控制信号端口
- out是输出端口

- 当s1和s0取值分别为00、01、10和11时，输出端out将分别输出in0、in1、in2和in3的数据。

3.6.1 多路选择器

- 例3.16和例3.17分别是4选1多路选择器的if-else语句描述和case语句描述。

例3.16：4选1多路选择器的if-else语句描述

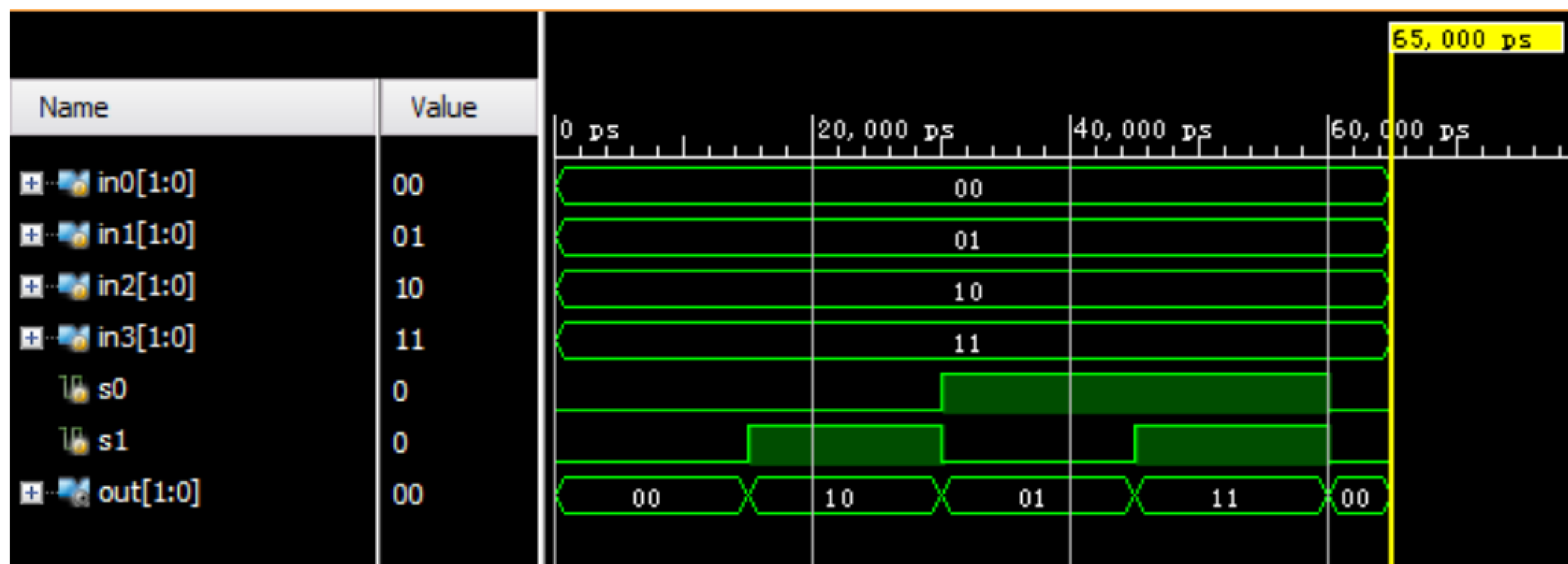
```
module mux41_if
    (    input in0,in1,in2,in3,
         input s0,s1,
         output reg out    //out声明为reg类型
    );
    always @*
        begin
           if ({s1,s0} == 2'b00)
              out = in0;
           else if({s1,s01} == 2'b01)
              out = in1;
           else if({s1,s01} == 2'b10)
              out = in2;
        else
              out = in3;
    end
endmodule
```

例3.17：4选1多路选择器的case语句描述

```
module mux41_case
    (
         input in0,in1,in2,in3,
         input s0,s1,
         output reg out    //out声明为reg类型
    );
    always @*
        begin
          case ({s1,s0})
            2'b00:   out = in0;
            2'b01:   out = in1;
            2'b10:   out = in2;
            default: out = in0;
          endcase
        end
 endmodule
```

3.6.1 多路选择器

□4选1多路选择器的仿真波形如图3.3所示。



3.6.2 比较器

- 比较器是用来完成数值大小比较逻辑的组合电路。
- 一位二进制数比较器是它的基础，其电路真值表如表3.3所示。
- in0和in1是一位输入比较信号
- lt、eq和gt分别是两个输入信号大小的比较结果

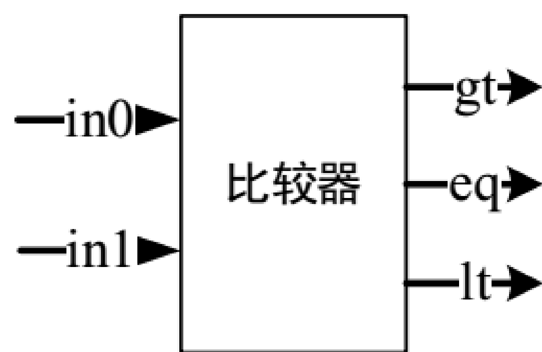


表3.3 一位二进制比较器的真值表

in0	in1	gt(in0>in1)	eq(in0=in1)	lt(in0<in1)
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

3.6.2 比较器

- 例3.18是其Verilog HDL描述。

注意：本设计中，在always块内if语句之前，我们对gt，eq和lt都赋值为0。这样做的重要性是保证每个输出都被分配一个值。如果没有这样做，Verilog会认为你不想让它们的值改变，系统将会自动生成一个锁存器，那么得到的电路就不再是一个组合电路了。

例3.19 N位二进制比较器的描述

```
module comp_N
  #(parameter N = 8)
  (
    input [N-1:0] in0,in1,
    output  reg  gt,eq,lt
  );
```

例3.18 一位二进制比较器的描述

```
module comp_1
  (
  input  in0,in1,
  output  reg  gt,eq,lt
 );
  always @*
    begin
      gt = 0;
      eq = 0;
      lt = 0;
      if(in0>in1)
        gt = 1;
      if(in0==in1)
        eq = 1;
      if(in0<in1)
        lt = 1;
    end
endmodule
```

3.6.3译码器和编码器

3-8译码器

- **译码器电路有n个输入和2^n个输出，每个输出都对应着一个可能的二进制输入。**
- **通常情况下，一次只能有一个输出有效。**

8-3编码器

- **编码器是译码器的反向器件**
- **它有2n个输入和n个输出，输出的n位二进制数代表着输入标号为2^n中的哪一路输入是高电平。**

3.6.3译码器和编码器

例3.20：3.8译码器的描述

```
module decode_3_8
 (
   input [2:0] in,
   output  reg [7:0]  y
 );
 always @*
   begin
     case (in)
       3'b000 :  y = 8'b00000001;
       3'b001 :  y = 8'b00000010;
       3'b010 :  y = 8'b00000100;
       3'b011 :  y = 8'b00001000;
       3'b100 :  y = 8'b00010000;
       3'b101 :  y = 8'b00100000;
       3'b110 :  y = 8'b01000000;
       3'b111 :  y = 8'b10000000;
   endcase
 end
endmodule
```

例3.21：8-3编码器的描述

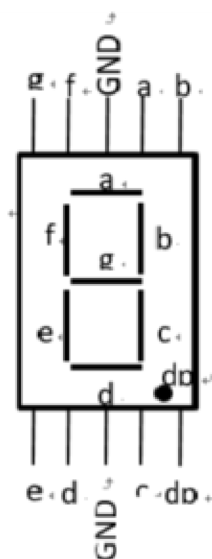
```
module encode_8_3(input [7:0] in,
   output  reg [2:0]  encode_out,
   output  reg  valid );
 always @*
   begin
     valid = 1;
     case (in)
       8'b00000001 :  encode_out = 3'b000;
       8'b00000010 :  encode_out = 3'b001;
       8'b00000100 :  encode_out = 3'b010;
       8'b00001000 :  encode_out = 3'b011;
       8'b00010000 :  encode_out = 3'b100;
       8'b00100000 :  encode_out = 3'b101;
       8'b01000000 :  encode_out = 3'b110;
       8'b10000000 :  encode_out = 3'b111;
       default     :  valid = 0;
     endcase
   end
endmodule
```

3.6.4：十六进制数七段LED显示译码器

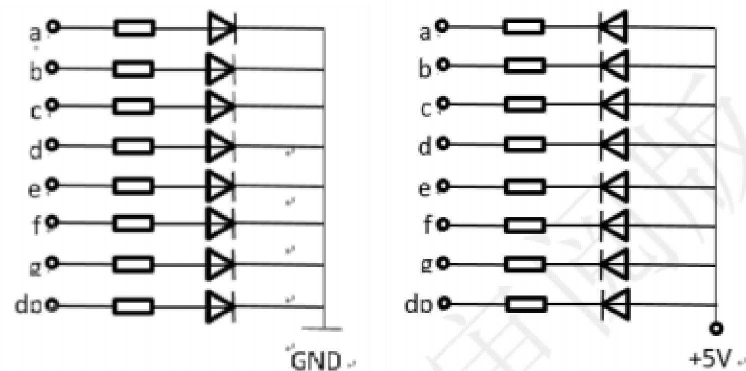
■七段LED显示器也称为七段数码管，其示意图如图3.8(a)所示

- 包括七个LED管和一个圆形LED小数点。
- 按LED单元连接方式可以分为共阳数码管和共阴数码管，共阳数码管是指将所有发光二极管的阳极接到一起形成公共阳极(COM)的数码管，COM接到逻辑高电平，当某一字段发光二极管的阴极为低电平时，相应字段就点亮。
- 共阴数码管是指将所有发光二极管的阴极接到一起形成公共阴极(COM)的数码管，COM接到逻辑低电平，当某一字段发光二极管的阳极为高电平时，相应字段就点亮。

图3.8：七段码LED示意图



(a)引脚



(b)共阴极 (c)共阳极

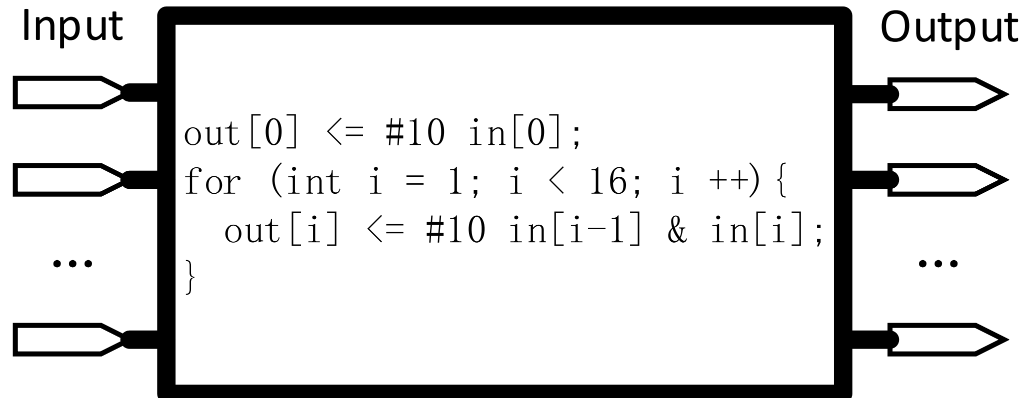
3.6.4 十六进制数七段LED显示译码器

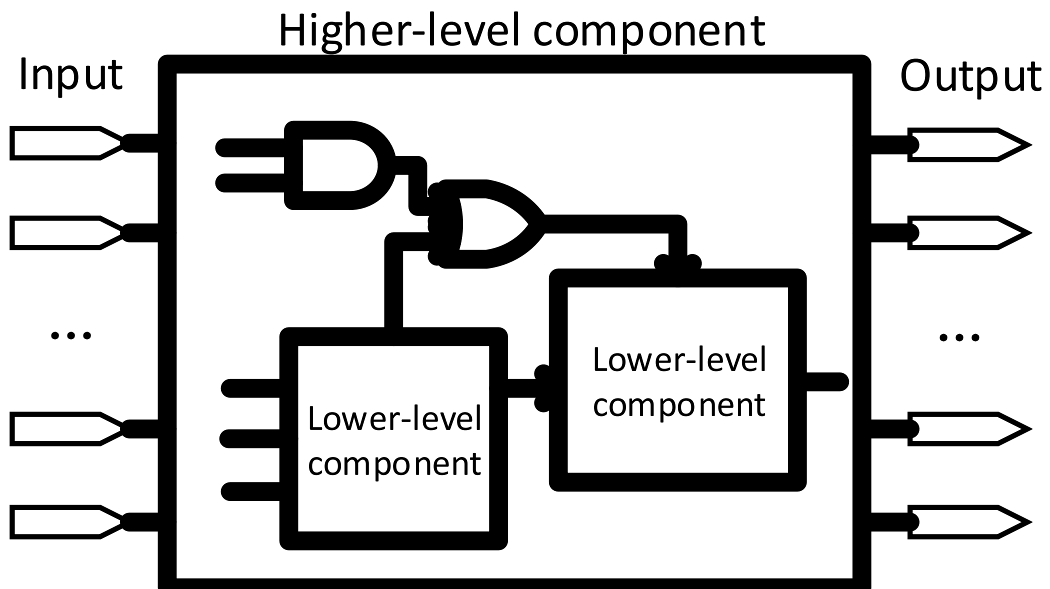
- 本节设计的十六进制数七段LED显示译码器，驱动共阳数码管
- 把一个4位十六进制数输入，转换为驱动7段LED显示管的控制逻辑；为了完整，我们把1位小数点的控制dp也列出
- Verilog HDL描述如例3.23所示

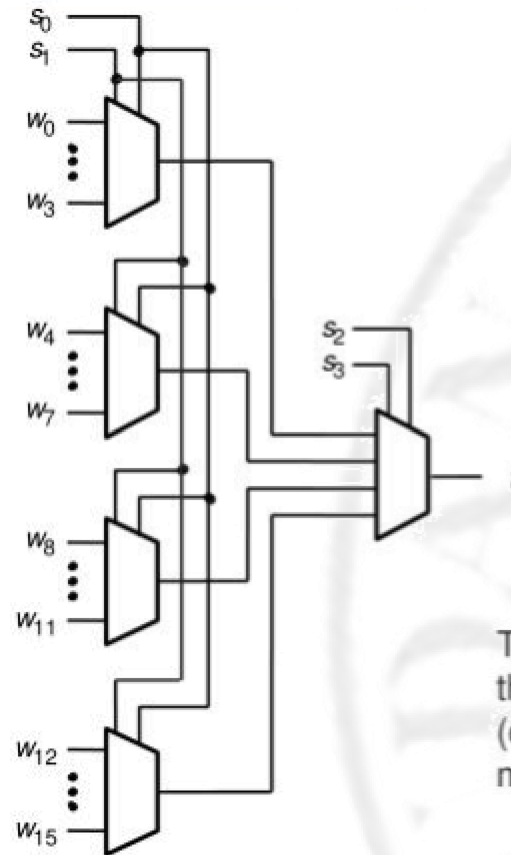
例3.23十六进制数七段LED显示译码器的描述

```
module hex_7seg
   (
     input [3:0] hex,
     input  dp,
     output reg [7:0]  sseg
   );
   always @*
     begin
       case (hex)
         4'h0: sseg [6:0] = 7'b0000001;
         4'h1: sseg [6:0] = 7'b1001111;
         4'h2: sseg [6:0] = 7'b0010010;
         4'h3: sseg [6:0] = 7'b0000110;
         4'h4: sseg [6:0] = 7'b1001100;
         4'h5: sseg [6:0] = 7'b0100100;
         4'h6: sseg [6:0] = 7'b0100000;
         4'h7: sseg [6:0] = 7'b0001111;
         4'h8: sseg [6:0] = 7'b0000000;
         4'h9: sseg [6:0] = 7'b0000100;
         4'ha: sseg [6:0] = 7'b0001000;
         4'hb: sseg [6:0] = 7'b1100000;
         4'hc: sseg [6:0] = 7'b0110001;
         4'hd: sseg [6:0] = 7'b1000010;
         4'he: sseg [6:0] = 7'b0110000;
         4'hf: sseg [6:0] = 7'b0111000;
       endcase
       sseg [7] = dp;
     end
endmodule
```

结构化的Verilog 与模块的级联调用







Structural style Verilog

```
module mux16x1(f,W,S);
    input [0:15] W;
    input [3:0] S;
    output f;
    wire [0:3] M;

    mux4x1 mux1(M[0],W[0:3],S[1:0]);
    mux4x1 mux2(M[1],W[4:7],S[1:0]);
    mux4x1 mux3(M[2],W[8:11],S[1:0]);
    mux4x1 mux4(M[3],W[12:15],S[1:0]);
    mux4x1 mux5(f,M,S[3:2]);
endmodule
```

The Verilog code for mux4x1 must be either in the same file as mux16x1, or in a separate file (called mux4x1.v) in the same directory as mux16x1

级联调用举例

- 已知一个3-8线译码器的Verilog模块，请用使用该module和若干基础逻辑门实现的一个4-16线译码器

例3.20： 3.8译码器的描述

```
module decode_3_8
  (
    input [2:0] in,
    output reg [7:0]  y
  );
  always @*
    begin
      case (in)
        3'b000 :  y = 8'b00000001;
        3'b001 :  y = 8'b00000010;
        3'b010 :  y = 8'b00000100;
        3'b011 :  y = 8'b00001000;
        3'b100 :  y = 8'b00010000;
        3'b101 :  y = 8'b00100000;
        3'b110 :  y = 8'b01000000;
        3'b111 :  y = 8'b10000000;
    endcase
  end
endmodule
```

可编程逻辑门阵列 FPGA 简介

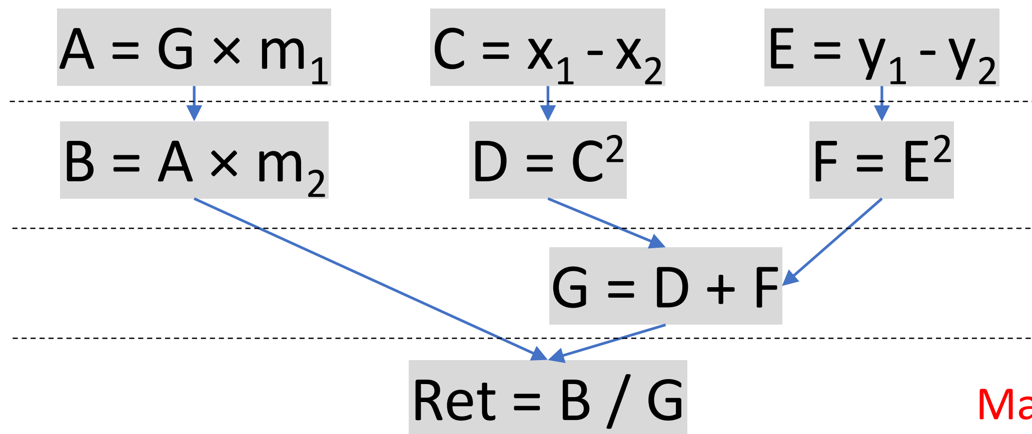
- ***Field-Programmable*** Gate Array
- Can be configured to act like any circuit by HDL
- Can do many things, but we focus on computation acceleration

FPGA vs. CPU/GPUs

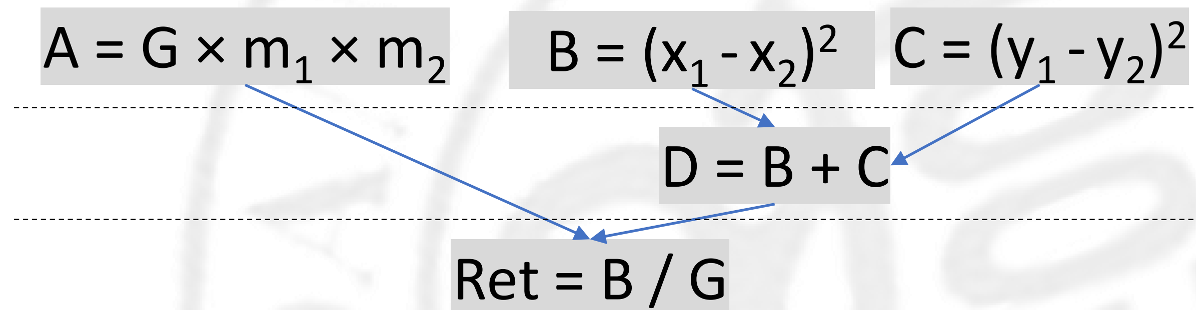
- GPU – The other major accelerator for parallel Computing
- CPU/GPU hardware is fixed
  - “General purpose”
  - we write programs (sequence of instructions) for them
- FPGA hardware is not fixed
  - “Special purpose”
  - Hardware can be whatever we want
  - Will our hardware require/support software? Maybe!
- Optimized hardware is very efficient
  - GPU-level performance**
  - 10x power efficiency (300 W vs 30 W)

Fine-Grained Parallelism of Special-Purpose Circuits

- Example -- Calculating gravitational force: $\frac{G \times m_1 \times m_2}{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
- 8 instructions on a CPU → 8 cycles**
- Much fewer cycles on a special purpose circuit



4 cycles with basic operations



3 cycles with compound operations

May slow down clock

Ret = (G × m_1 × m_2) / ((x_1 - x_2)^2 + (y_1 - y_2)^2)

1 cycle with even further compound operations

How Is It Different From ASICs

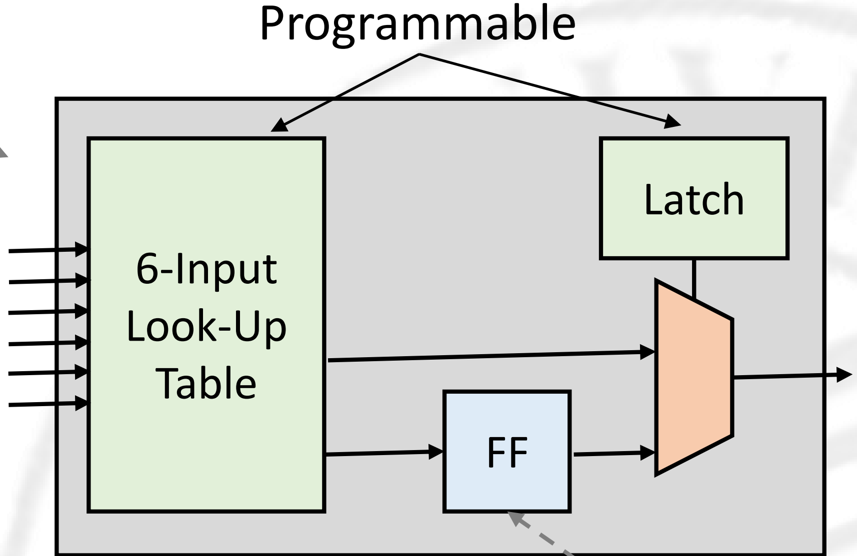
- ASIC (Application-Specific Integrated Circuit)
  - Special chip purpose-built for an application
  - E.g., ASIC bitcoin miner, Intel neural network accelerator
  - Function cannot be changed once expensively built
- \+ FPGAs can be ***field-programmed***
  - Function can be changed completely whenever
  - FPGA fabric ***emulates*** custom circuits
- \- Emulated circuits are not as efficient as bare-metal
  - ~10x performance (larger circuits, faster clock)
  - ~10x power efficiency

Basic FPGA Architecture

"Configurable logic block (CLB)"

I/O block

Programmable interconnect



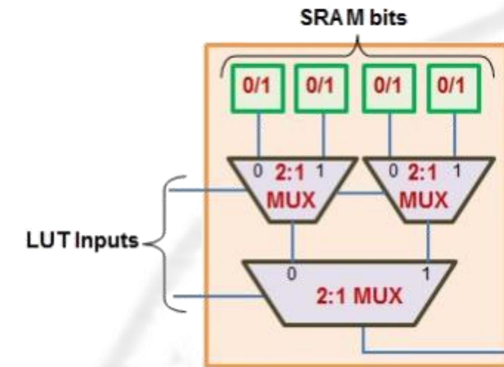
Sequential circuit construction

Ex) 2-LUT for "AND"

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

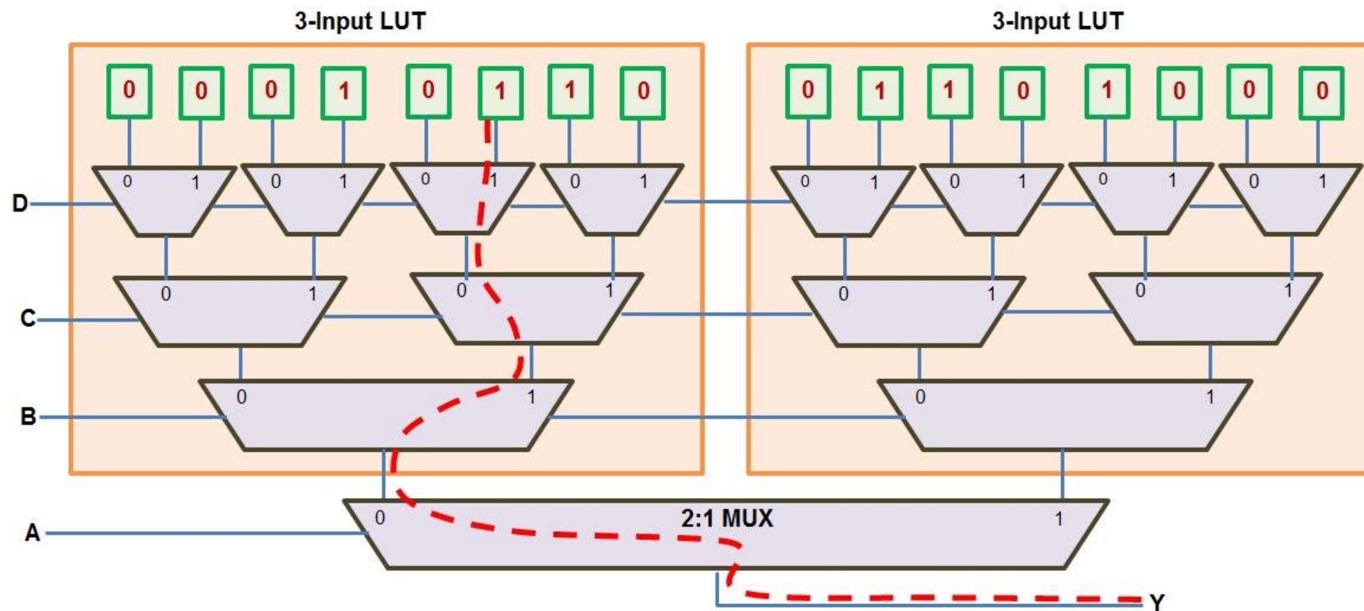
FPGA: Field Programmable Grate Array

- Basic Cell: Look-Up-Table (LUT)
- Basic structure: RAM + Switches/ MUX's
- Mapping of "any" logic – generate bit table



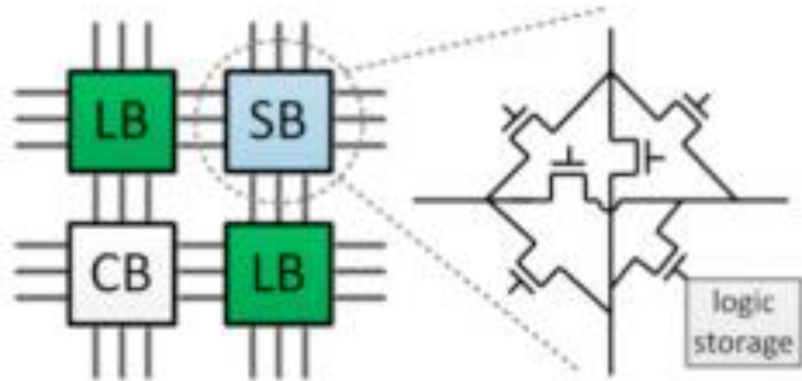
Truth Table

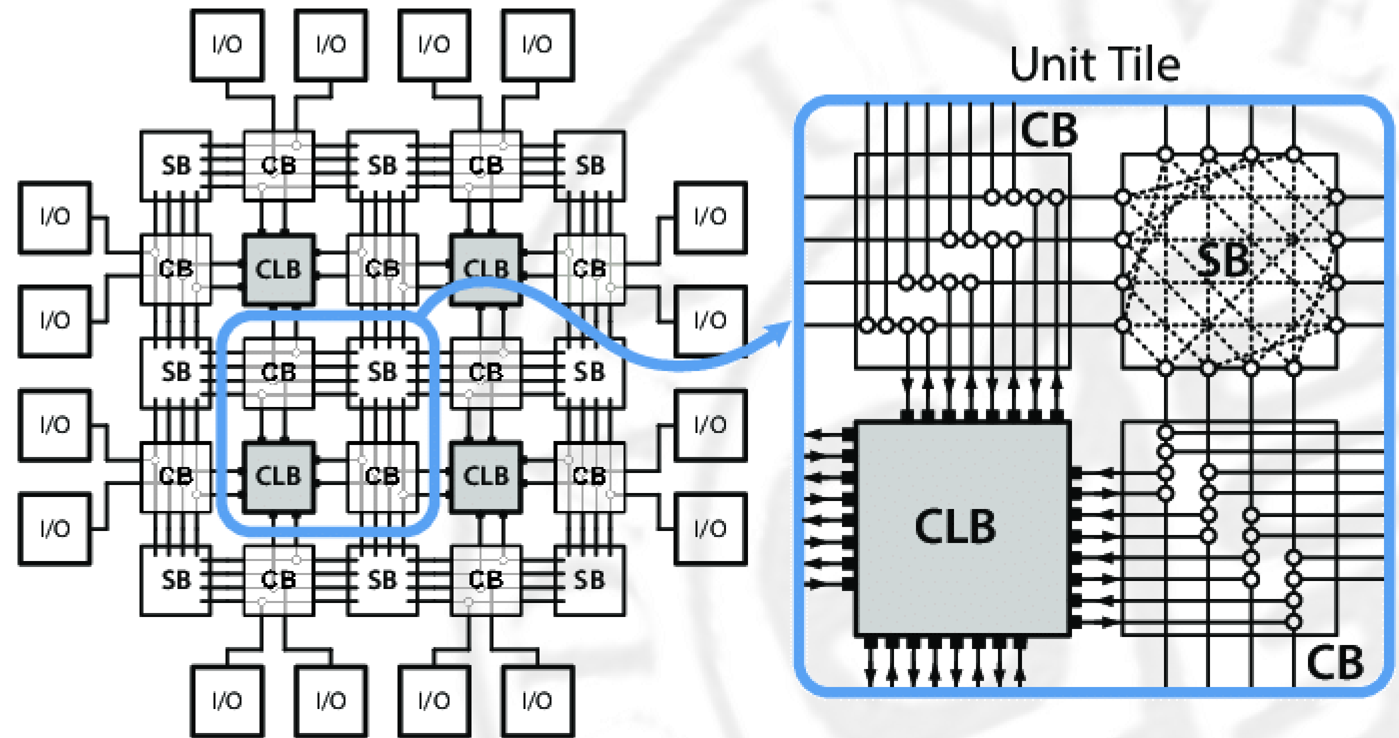
Inputs				Output
A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0



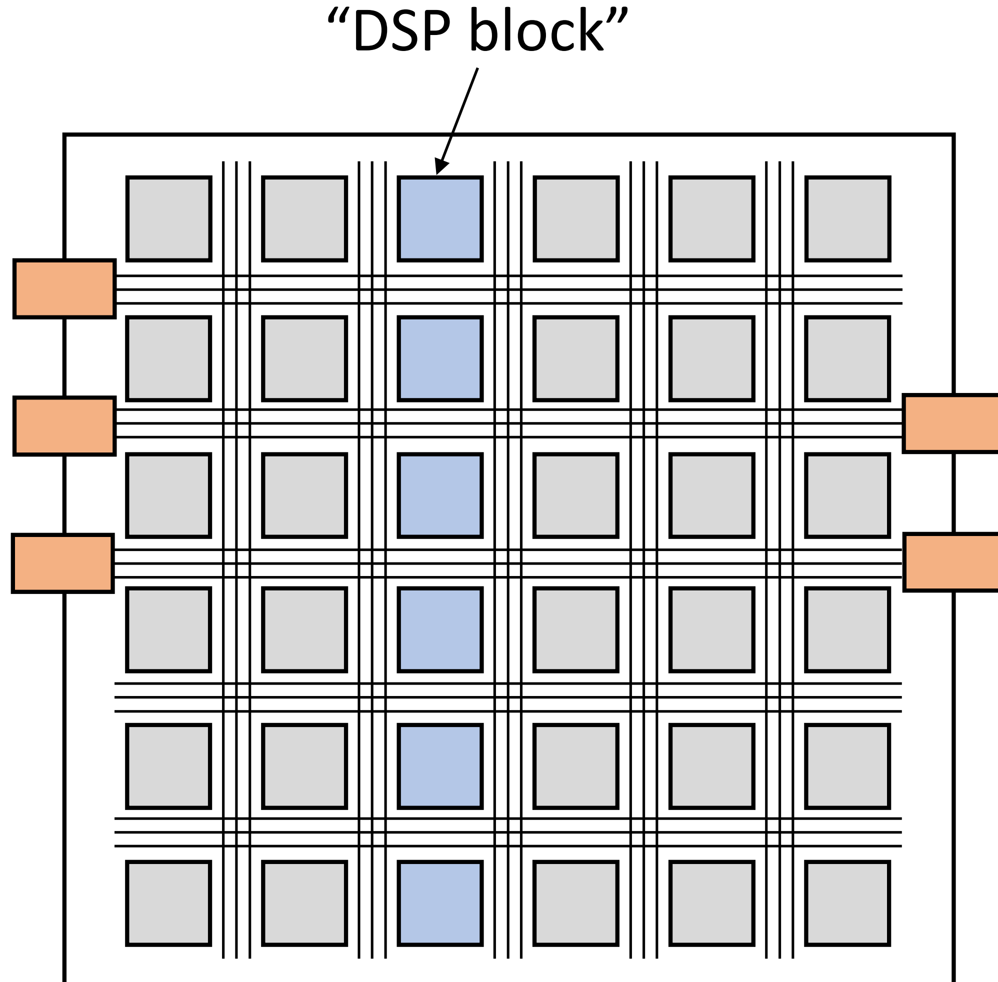
Global Fabric

- Interconnection between CLBs:
  - CB = connection boxes
  - SB = switch boxes

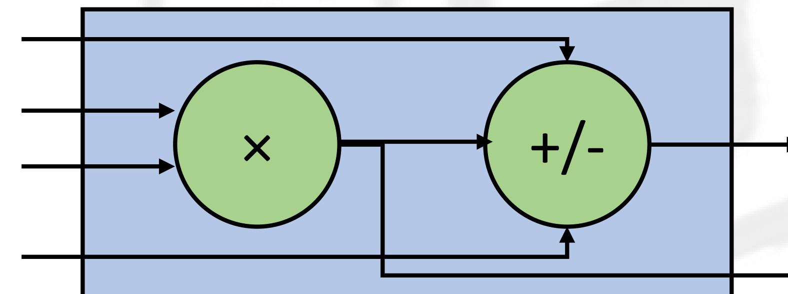




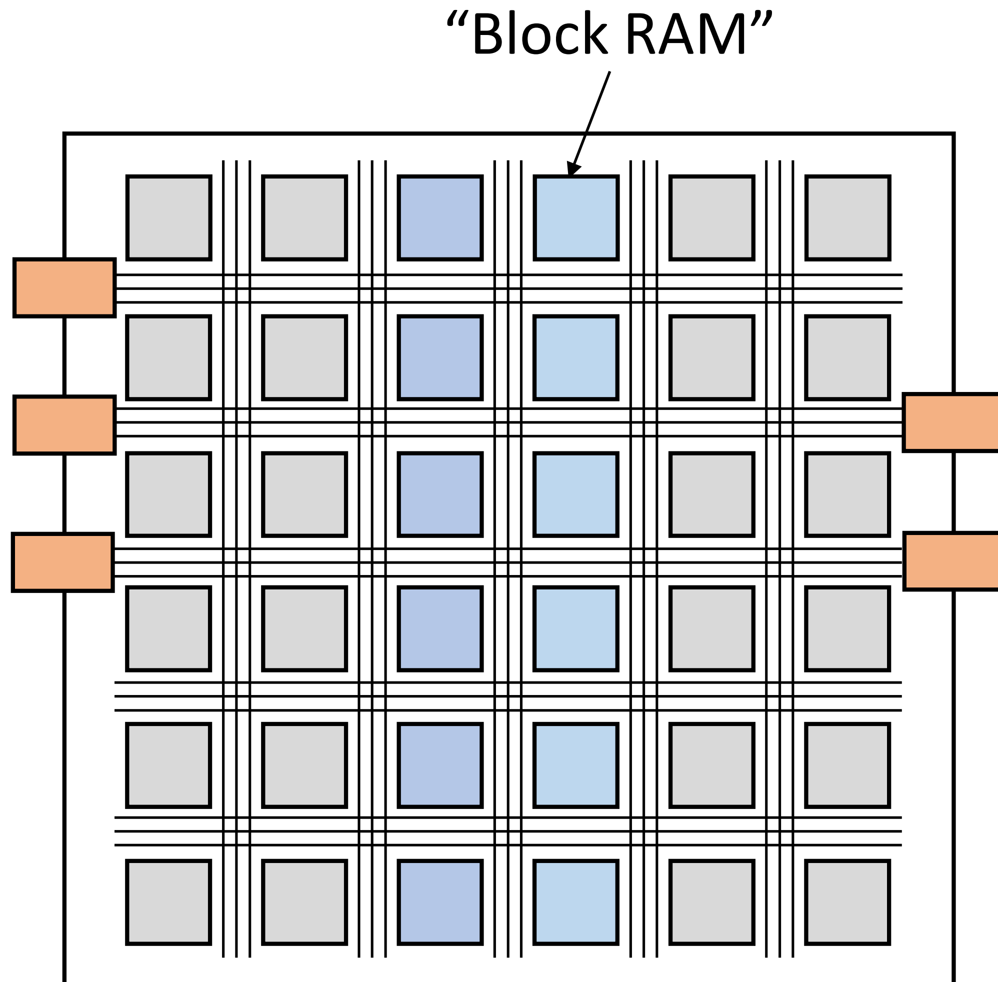
Basic FPGA Architecture – DSP Blocks



- CLBs act as gates – Many needed to implement high-level logic
- Arithmetic operation provided as efficient ALU blocks
  - “Digital Signal Processing (DSP) blocks”
  - Each block provides an adder + multiplier

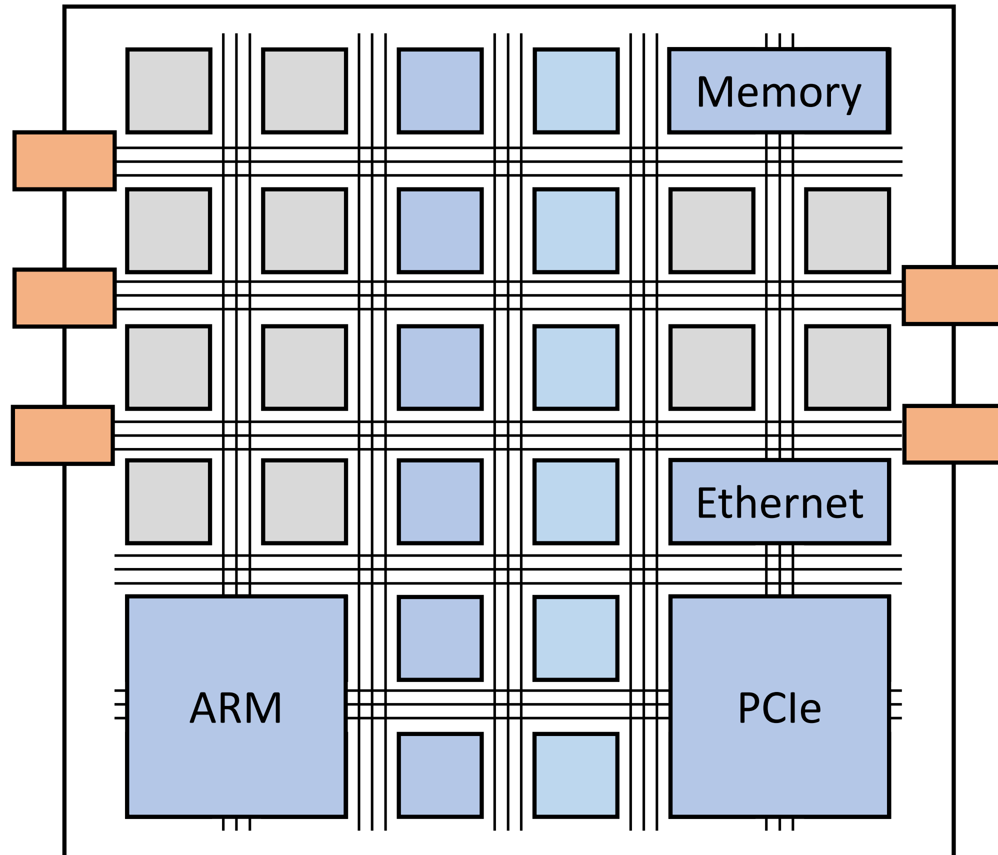


Basic FPGA Architecture – Block RAM



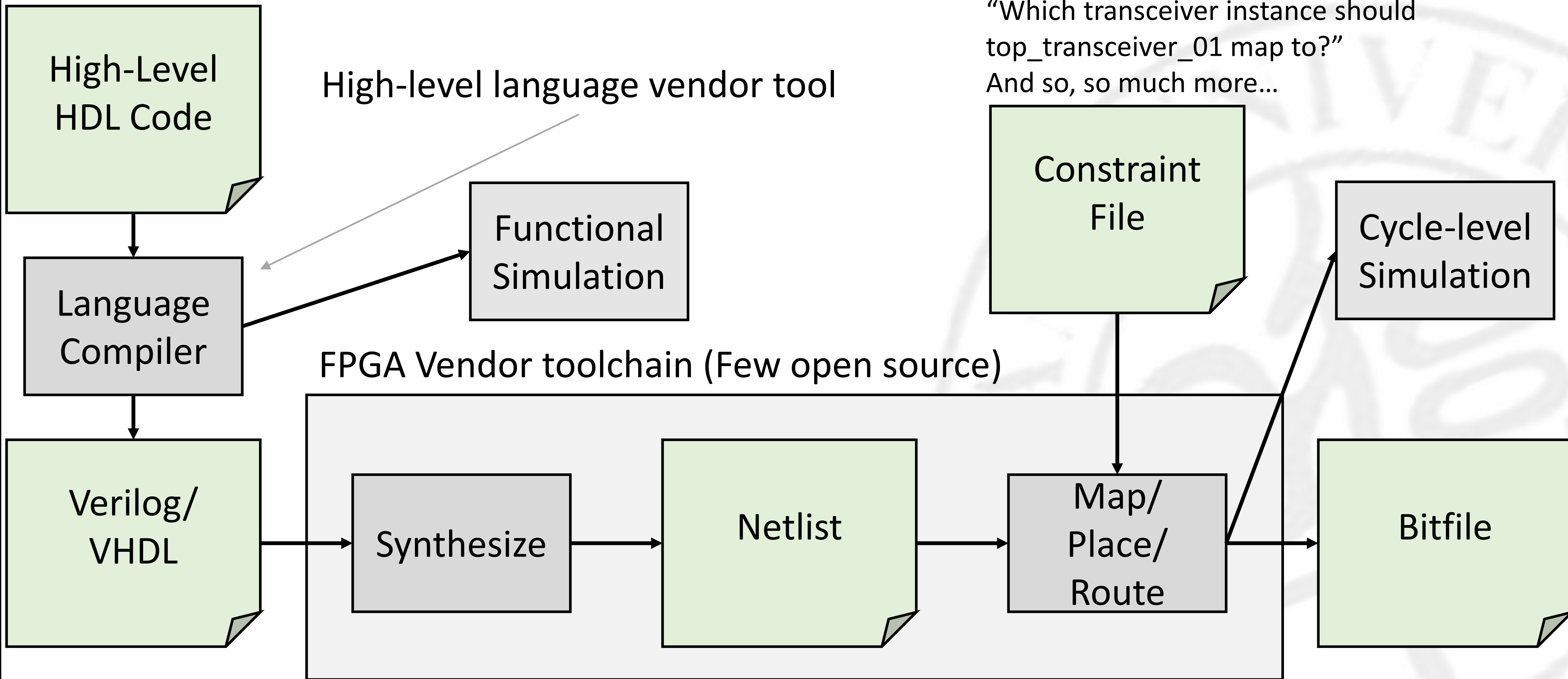
- CLB can act as flip-flops
  - (~1 bit/block) – tiny!
- Some on-chip SRAM provided as blocks
  - ~18/36 Kbit/block, MBs per chip
  - Massively parallel access to data → multi-TB/s bandwidth

Basic FPGA Architecture – Hard Cores



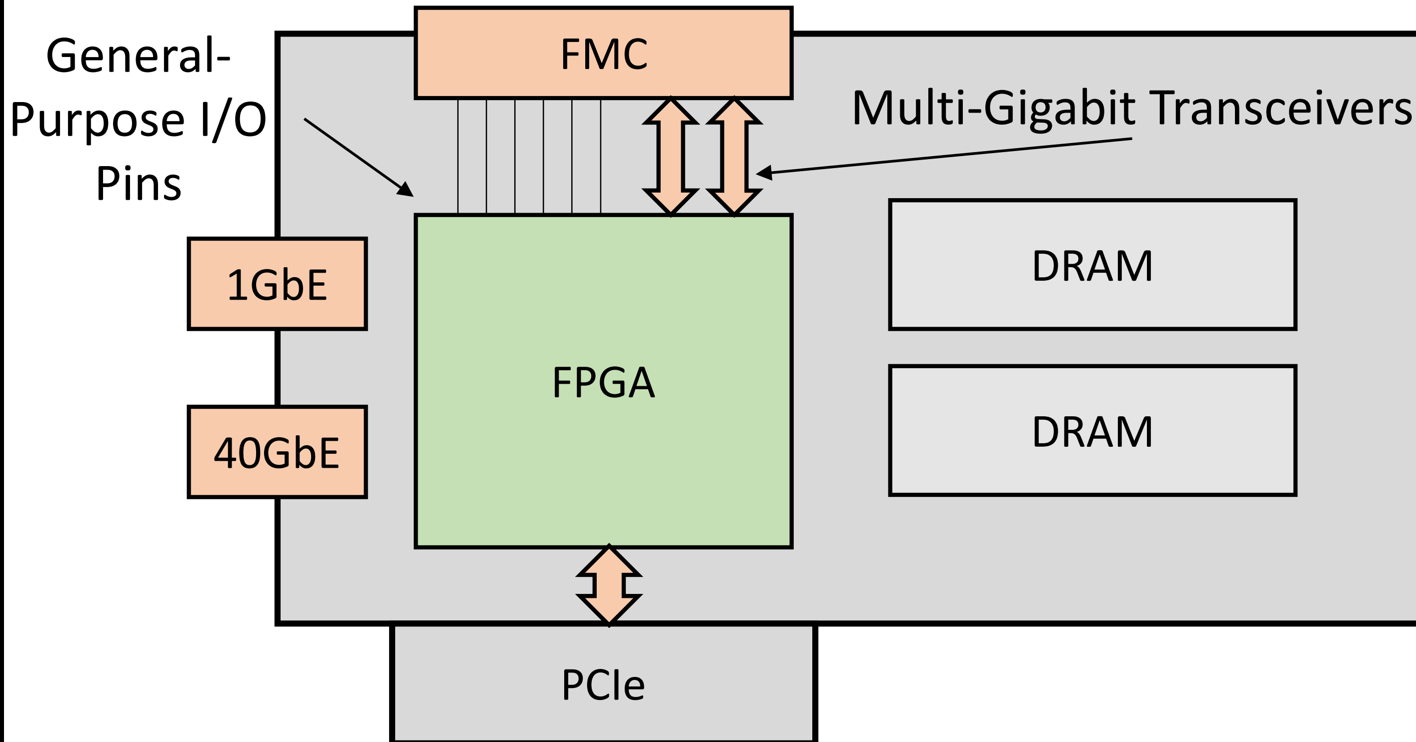
- Some functions are provided as efficient, non-configurable "hard cores"
  - Multi-core ARM cores ("Zynq" series)
  - Multi-Gigabit Transceivers
  - PCIe/Ethernet PHY
  - Memory controllers
  - …

FPGA Compilation Toolchain

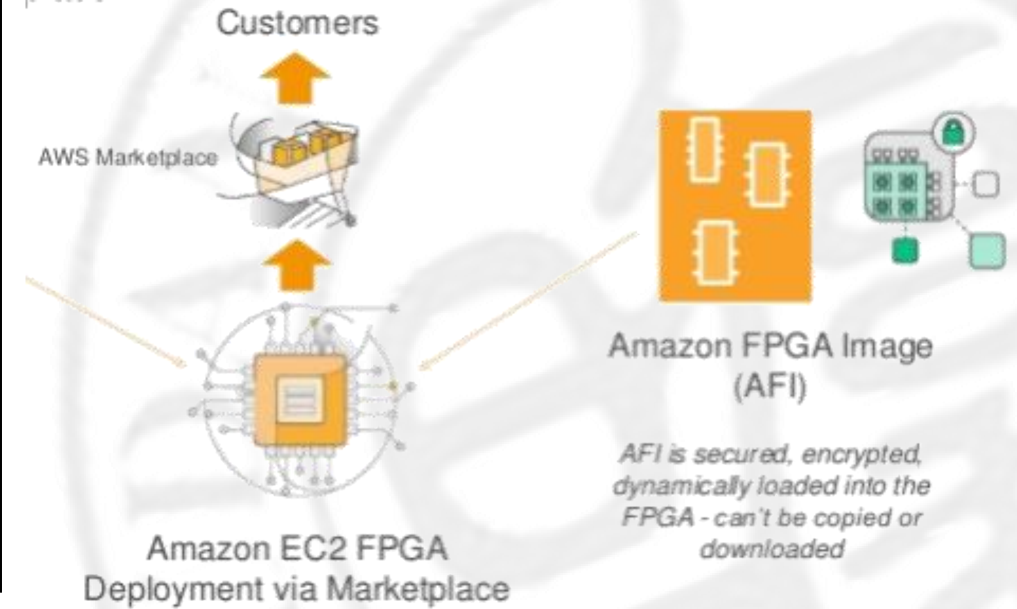


FPGA Accelerator Card Architecture & Cloud

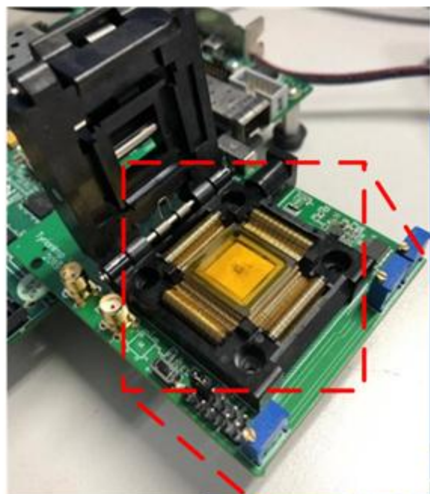
- "FPGA Mezzanine Card" Expansion
  - Network Ports, Memory, Storage, PCIe, …



- Amazon EC2 F1 instance (1 – 4 FPGAs)
- Microsoft Azure, etc…



课程实验计划



Chip

PYNQ-based Host Interface