版本

更新记录	文档名	实验指导书 _lab1
	版本号	0.3
	创建人	计算机组成原理教学组
	创建日期	2017/10/18

更新历史

序号	更新日期	更新人	版本号	更新内容
1	2017/10/8	吕昱峰	0.1	初版,存储器与 ALU 实验
2	2018/10/8	吕昱峰	0.2	去除无用的章节前述,改为实验开始前的准备内容;删除存储器实验冗杂的拨码要求;删除验证性实验,将 0.1 版本中验证性实验改为设计实验。
3	2019/11/2	吕昱峰	0.3	调整存储器部分至实验 2,实验 1 添加简单的流水线设计;修改实验原理图。

致谢:

本文档流水线设计部分采用中国科学院大学实验指导书中流水线基础内容,其设计与 Demo 均取自龙芯《计算机体系结构导教班》提供的资源,特此说明并感谢。

文档错误反馈: 本文档经多版本迭代,但由于作者精力能力有限,其中出现错误请联系:

lvyufeng@cqu.edu.cn

1 实验一　简单流水线与运算器实验

在进行本次实验前，你需要具备以下实验环境及基础能力：

1. 装有Vivado2019.1的电脑一台；
   (a). 本实验不对 Vivado 环境有硬性要求，但涉及 Xilinx 库中 IP 的实验，在不同版本环境下无法兼容，且低版本 Vivado 无法运行高版本生成的项目，为方便实验检查，应当尽量使用实验要求版本。
   (b). 若未曾安装 Vivado，请参考文档“**A03_Vivado 安装说明 _v1.00**”。
2. 熟悉 Vivado 的 IDE 环境，并能够使用其进行仿真、综合；
   (a). 如果对 Vivado 不熟悉，参考文档“**A04_Vivado 使用说明 _v1.00**”。
3. 熟悉 Nexys4 DDR 开发板 (Artix-7)；
   请确保在实验进行前阅读过“**A01_Nexys4_DDR 用户手册**”。

1.1 实验目的

1. 理解流水线 (Pipeline) 设计原理；
2. 了解算术逻辑单元 ALU 的原理；
3. 熟悉并运用 Verilog 语言设计 ALU；
4. 熟悉并运用 Verilog 语言设计流水线全加器；
5. 学习 Verilog 不同形式的编程方式，理解 assign 和 always 的区别；

1.2 实验设备

1. 计算机 1 台 (尽可能达到 8G 及以上内存)；
2. Nexys4 DDR 实验开发板；
3. Xilinx Vivado 开发套件 (2018.1 版本)。

1.3 实验任务

本次实验包含两部分，包括 ALU 设计和简单流水线电路设计。

1.3.1 ALU 设计实验

图 1给出了一个具有 N 位输入和 N 位输出的算数逻辑单元的电路符号。算术逻辑单元接收说明执行哪个功能的控制信号 F，执行对应功能后输出 N 位结果。

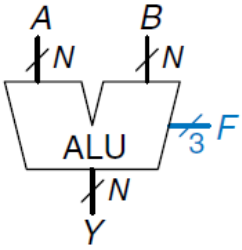

图 1: ALU

实验要求实现以下算术运算功能,其对应的控制码及功能如下:

$F_{2:0}$	功能	$F_{2:0}$	功能
000	A + B(Unsigned)	100	$\overline{A}$
001	A - B	***101***	***SLT***
010	A AND B	110	未使用
011	A OR B	111	未使用

表 1: 算数运算控制码及功能

本次实验将 ALU 输出结果通过板载七段数码管进行显示验证,原理图如图 2所示:

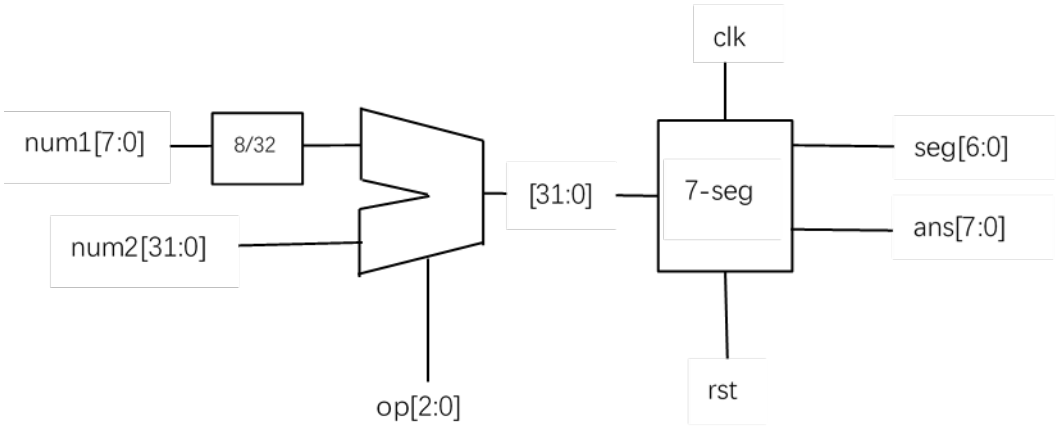

图 2: ALU 实验原理图

实验要求:

1. 根据 ALU 原理图 (图 1), 使用 Verilog 语言定义 ALU 模块, 其中输入输出端口参考实验原理,运算指令码长度为 [2:0]。
2. 自行编写 Testbench 仿真文件, 时钟周期设为 10ns, 每个时钟周期输入 A、B、OP, 捕获输出并打印在控制台。
   注意: 由于疫情原因无硬件设备上板测试, 因此不需要做 8-32bit 扩展。但仍需测试每种运算结果(具体见实验报告表格)。
3. 验证表 1中所有功能。

4. 实现 SLT 功能。
5. 给出 RTL 源程序(.v 文件)

6. 内置一个 32 位 num2(值为 32h'01)作为输入到运算器端口 A;
7. 将 sw0-sw7 输入到 num1, 经过无符号扩展至 32 位后,输入到运算器的端口 B;
8. 运算器支持"加、减、与、或、非"5 种运算,需要 3 位(8 个操作)。将 sw15-sw14 输入到 op 作为运算器的控制信号;
9. 将计算 32 位结果 s 显示到七段数码管 (16 进制)。

灰色部分由于疫情原因暂不实现

1.3.2 流水线实验

本次实验为仿真实验,设计完成后仅需进行**行为仿真**。

实验要求:

1. 实现 4 级流水线 32bit 全加器,每一级进行 8bit 加法运算,需带有流水线暂停和刷新;
2. 模拟流水线暂停,仿真时控制 10 周期后暂停流水线 2 周期(第 2 级),流水线恢复流动;
3. 模拟流水线刷新,仿真时控制 15 周期时流水线刷新(第 3 级)。

1.4 实验环境

以下表格中红色部分需自行实现,黑色部分与实验发布包中提供。

1.4.1 ALU 实验

|–top.v	设计顶层文件,参照图 2将各模块连接。
|—-alu.v	ALU 模块,本次实验重点。
|—-display.v	七段数码管显示模块文件,已提供。
|——seg7.v	七段数码管显示模块组成文件,已提供。
|–constr.xdc	综合实现时,约束文件,已提供。
|–sim.v	仿真文件,控制时钟、信号。

表 2: ALU 实验文件树

1.4.2 流水线实验

|–sim.v	仿真文件,控制时钟、信号。
|—-stallable_pipeline_adder.v	有阻塞 4 级流水线 8bit 全加器实现文件

表 3: 流水线实验文件树

2 流水线

2.1 流水线原理

2.1.1 什么是流水线

流水线设计就是将组合逻辑系统地分割,并在各个部分(分级)之间插入寄存器,并暂存中间数据的方法。目的是将一个大操作分解成若干的小操作,每一步小操作的时间较小,所以能提高频率,各小操作能并行执行,所以能提高数据吞吐率(提高处理速度)。

2.1.2 什么时候用流水线设计

使用流水线一般是时序比较紧张,对电路工作频率较高的时候。典型情况如下:

1. 功能模块之间的流水线,用乒乓 buffer 来交互数据。代价是增加了 memory 的数量,但是和获得的巨大性能提升相比,可以忽略不计。
2. I/O 瓶颈,比如某个运算需要输入 8 个数据,而 memroy 只能同时提供 2 个数据,如果通过适当划分运算步骤,使用流水线反而会减少面积。
3. 片内 sram 的读操作,因为 sram 的读操作本身就是两极流水线,除非下一步操作依赖读结果,否则使用流水线是自然而然的事情。
4. 组合逻辑太长,比如 (a+b)*c,那么在加法和乘法之间插入寄存器是比较稳妥的做法。

2.1.3 使用流水线的优缺点

优点:流水线缩短了在一个时钟周期内给的那个信号必须通过的通路长度,增加了数据吞吐量,从而可以提高时钟频率,但也导致了数据的延时。

缺点:功耗增加,面积增加,硬件复杂度增加,特别对于复杂逻辑如 CPU 的流水线而言,流水越深,发生需要暂停流水线或刷新流水线的情况时,时间损失越大。

所以使用流水线并非有利无害,设计时需权衡考虑。

2.2 流水线基础示例

首先要说明的一点是,流水线电路纯粹就是一个数字电路的概念,不要一谈到流水线就仅仅认为是处理器中的流水线。

我们先来看一个完全不会被阻塞的 3 级流水线电路怎么写。

Listing 1: 无阻塞 3 级流水线实现

```
module non_stall_pipeline #
(
    parameter WIDTH = 100
)
(
    input              clk,
    input  [WIDTH-1:0] datain,
    output [WIDTH-1:0] dataout
);

reg [WIDTH-1:0] pipe1_data;
reg [WIDTH-1:0] pipe2_data;
reg [WIDTH-1:0] pipe3_data;

always @(posedge clk) begin
    pipe1_data <= datain;
end

always @(posedge clk) begin
    pipe2_data <= pipe1_data;
end

always @(posedge clk) begin
    pipe3_data <= pipe2_data;
end

assign dataout = pipe3_data;

endmodule
```

但是实际电路设计中,并不总是上面这种无阻塞的流水线。下面给出一个具有阻塞功能的流水线代码。

Listing 2: 有阻塞 3 级流水线实现

```
module stallable_pipeline #
(
parameter WIDTH = 100
)
(
input              clk,
input              rst,
input              validin,
input  [WIDTH-1:0] datain,
```

```
10 input               out_allow,
11 output              validout,
12 output [WIDTH-1:0] dataout
13 );
14
15 reg             pipe1_valid;
16 reg [WIDTH-1:0] pipe1_data;
17 reg             pipe2_valid;
18 reg [WIDTH-1:0] pipe2_data;
19 reg             pipe3_valid;
20 reg [WIDTH-1:0] pipe3_data;
21
22 // pipeline stage1
23 wire            pipe1_allowin;
24 wire            pipe1_ready_go;
25 wire            pipe1_to_pipe2_valid;
26
27 assign pipe1_ready_go = ……
28 assign pipe1_allowin = !pipe1_valid || pipe1_ready_go &&
       pipe2_allowin;
29 assign pipe1_to_pipe2_valid = pipe1_valid && pipe1_ready_go;
30
31 always @(posedge clk) begin
32     if (rst) begin
33         pipe1_valid <= 1'b0;
34     end
35     else if (pipe1_allowin) begin
36         pipe1_valid <= validin;
37     end
38     if (validin && pipe1_allowin) begin
39         pipe1_data <= datain;
40     end
41 end
42
43 // pipeline stage2
44 wire            pipe2_allowin;
45 wire            pipe2_ready_go;
46 wire            pipe2_to_pipe3_valid;
47
48 assign pipe2_ready_go = ……
49 assign pipe2_allowin = !pipe2_valid || pipe2_ready_go &&
       pipe3_allowin;
50 assign pipe2_to_pipe3_valid = pipe2_valid && pipe2_ready_go;
51
52 always @(posedge clk) begin
53     if (rst) begin
54         pipe2_valid <= 1'b0;
55     end
56     else if (pipe2_allowin) begin
```

```
57          pipe2_valid <= pipe1_to_pipe2_valid;
58      end
59      if (pipe1_to_pipe2_valid && pipe2_allowin) begin
60          pipe2_data <= pipe1_data;
61      end
62  end
63
64  // pipeline stage3
65  wire            pipe3_allowin;
66  wire            pipe3_ready_go;
67
68  assign pipe3_ready_go = ……
69  assign pipe3_allowin = !pipe3_valid || pipe3_ready_go &&
        out_allow;
70
71  always @(posedge clk) begin
72      if (rst) begin
73          pipe3_valid <= 1'b0;
74      end
75      else if (pipe3_allowin) begin
76          pipe3_valid <= pipe2_to_pipe3_valid;
77      end
78      if (pipe2_to_pipe3_valid && pipe3_allowin) begin
79          pipe3_data <= pipe2_data;
80      end
81  end
82
83  assign validout = pipe3_valid && pipe3_ready_go;
84  assign dataout  = pipe3_data;
85
86  endmodule
```

上述代码表述中 pipe**X**_valid 为 1 表示第 X 级流水级上当前存有有效的数据，为 0 表示第 X 级是空的。因为控制位的好处是，清空流水线的时候不用刷各级流水线 data 域的值，可以节约逻辑资源，但是需要注意的是根据流水级的 data 域信息产生控制信号时，有时候不要忘记“与上”这一级的 valid 信号。

pipe**X**_allowin 为 1 表示第 X 级可以接收前一级流水送来的数据。

上面流水线的 RTL 写法对于流水级的关键控制信号采用了链式写法。这样写的好处是，如果发生流水级功能调整时，譬如将一级再切分为两级，或者将某一级的停顿周期数改变了，都只需要修改局部，不会涉及全局性的调整。有的同学可能会觉得我这种写法很麻烦，觉得自己写出来的简单一些。请你们针对具体情况分析，是否对于你所涉及的流水线，上面给出的代码中，某些 pipeX_allowin 和 pipeX_ready_go 可以恒置为 1，将其代入上面的表达式中，并将级联的控制信号完全展开，就形成了你们的那种简单版本。这里给出这样的一个流水线电路的“模板”式写法，主要是因为它易于理解不易出错，当同学们在自己进行流水线设计时如果因为流水级控制信号搞得很糊涂时，可以回过头来看看此处给出的例子。

再额外说一点，上面的写法是把流水线的 control logic 和 data path 写到了一个模块内部。你也可以把两者分在不同的模块中。国外的教科书上在描述流水线 CPU 时，通常会在流水线之外画一个叫做 pipeline control logic 的椭圆形框，从这个框引出控制信号去往各级流水所在触发器的使能端。这就是把 control logic 和 data path 分离的写法。我们在上面给出的例子，pipeX_data 就是 data path 主体。

2.3 8bit 全加器示例

2.3.1 非流水线方式

Listing 3: 8bit 全加器非流水线实现

```
1  module adder_8bits(
2      a,
3      b,
4      c
5  );
6  input  [7:0] a;
7  input  [7:0] b;
8  output [8:0] c;
9
10 assign c[8:0] = {1'd0, a} + {1'd0, b};
11 endmodule
```

2.3.2 2 级流水线

Listing 4: 8bit 全加器 2 级流水线实现

```
1  module adder_4bits_2steps(cin_a, cin_b, cin, clk, cout, sum);
2  input [7:0]     cin_a;
3  input [7:0]     cin_b;
4  input           cin;
5  input           clk;
6  output          cout;
7  output [7:0]    sum;
8
9  reg         cout;
10 reg         cout_temp;
11 reg [7:0]   sum;
12 reg [3:0]   sum_temp;
```

```
13
14 always @(posedge clk) begin
15     {cout_temp,sum_temp} <= cin_a[3:0] + cin_b[3:0] + cin;
16 end
17
18 always @(posedge clk) begin
19     {cout,sum} <= {{1'b0,cin_a[7:4]} + {1'b0,cin_b[7:4]} +
           cout_temp, sum_temp};
20 end
21 endmodule
```

2.3.3 4 级流水线

Listing 5: 8bit 全加器 4 级流水线实现

```
1  module adder_8bits_4steps(cin_a, cin_b, c_in, clk, c_out,
       sum_out);
2  input [7:0]      cin_a;
3  input [7:0]      cin_b;
4  input            c_in;
5  input            clk;
6  output           c_out;
7  output [7:0]     sum_out;
8
9  reg c_out;
10 reg c_out_t1, c_out_t2, c_out_t3;
11
12 reg [7:0] sum_out;
13 reg [1:0] sum_out_t1;
14 reg [3:0] sum_out_t2;
15 reg [5:0] sum_out_t3;
16
17 always @(posedge clk) begin
18     {c_out_t1, sum_out_t1} <= {1'b0, cin_a[1:0]} + {1'b0, cin_b
           [1:0]} + c_in;
19 end
20
21 always @(posedge clk) begin
22     {c_out_t2, sum_out_t2} <= {{1'b0, cin_a[3:2]} + {1'b0,
           cin_b[3:2]} + c_out_t1, sum_out_t1};
23 end
24
25 always @(posedge clk) begin
26     {c_out_t3, sum_out_t3} <= {{1'b0, cin_a[5:4]} + {1'b0,
           cin_b[5:4]} + c_out_t2, sum_out_t2};
```

```
27 end
28
29 always @(posedge clk) begin
30     {c_out, sum_out} <= {{1'b0, cin_a[7:6]} + {1'b0, cin_b
       [7:6]} + c_out_t3, sum_out_t3};
31 end
32
33 endmodule
```

3 Verilog 不同实现方式

验证实验给出两种不同的组合逻辑实现方式：

Listing 6: assign 实现组合逻辑

```
module calculate(
    input wire [7:0] num1,
    input wire [2:0] op,
    output [31:0] result
    );
    wire [31:0] num2;
    wire [31:0] Sign_extend;

    assign num2 = 32'h00000001;
    assign Sign_extend={{24{1'b0}},num1[7:0]};
    assign result = (op == 3'b000)? Sign_extend + num2:
                    (op == 3'b001)? Sign_extend - num2:
                    (op == 3'b010)? Sign_extend & num2:
                    (op == 3'b011)? Sign_extend | num2:
                    (op == 3'b100)? ~Sign_extend: 32'h00000000;
endmodule
```

Listing 7: always 实现组合逻辑

```
module calculate(
    input wire [7:0] num1,
    input wire [2:0] op,
    output reg [31:0] result
    );
    reg [31:0] num2;
    reg[31:0] Sign_extend;
    always @(op) begin
       num2 = 32'h00000001;
       Sign_extend={24'h000000,num1[7:0]};
            case(op)
                3'b000:result = Sign_extend + num2;
                3'b001:result = Sign_extend - num2;
                3'b010:result = Sign_extend &  num2;
                3'b011:result = Sign_extend |  num2;
                3'b100:result = ~Sign_extend  ;
                default:result = 32'h00000000;
            endcase
        end
```

```
20  endmodule
```

二者区别对比如下:

	assign	always
result	wire	reg
num2	wire	reg
sign_extend	wire	reg

表 4: assign & always 信号对比

Always 即可实现组合逻辑,也可实现时序逻辑:

(1) always @(a or b or c)或 always@(*)形式的,即不带时钟边沿的,综合出来还是组合逻辑;
(2) always @(posedge clk)形式的,即带有边沿的,综合出来一般是时序逻辑,会包含触发器(Flip-Flop)

观察 Listing 7中 always 的触发信号 op,可以得知其为组合逻辑。在这里,给出两种方式在 FPGA 资源及编程难度上的对比。

	assign	always
资源占用	使用 wire 变量,综合得到的组合逻辑全部由导线构成,不占用 FPGA 资源	由于内部赋值与输出都需要使用 reg 变量,同为组合逻辑,需占用一定资源
编程及理解难度	面向信号进行状态描述,一般需要考虑单个信号在所有状态下的可能性,需要完全理解后进行编程。 阅读者理解难度大。	面向实验需求描述,always 模块内书写方式与高级语言相仿,可以按照实验要求逐一列举。 阅读者理解较容易。

表 5: assign & always 设计对比

注意:Imagination 及龙芯高校开源计划代码均采取 assign 方式,实验推荐使用 assign 方式实现组合逻辑。设计实验中新增内容使用 assign 方式增加输出信号更为简单。

A Nexys4 DDR 开发板基本信息

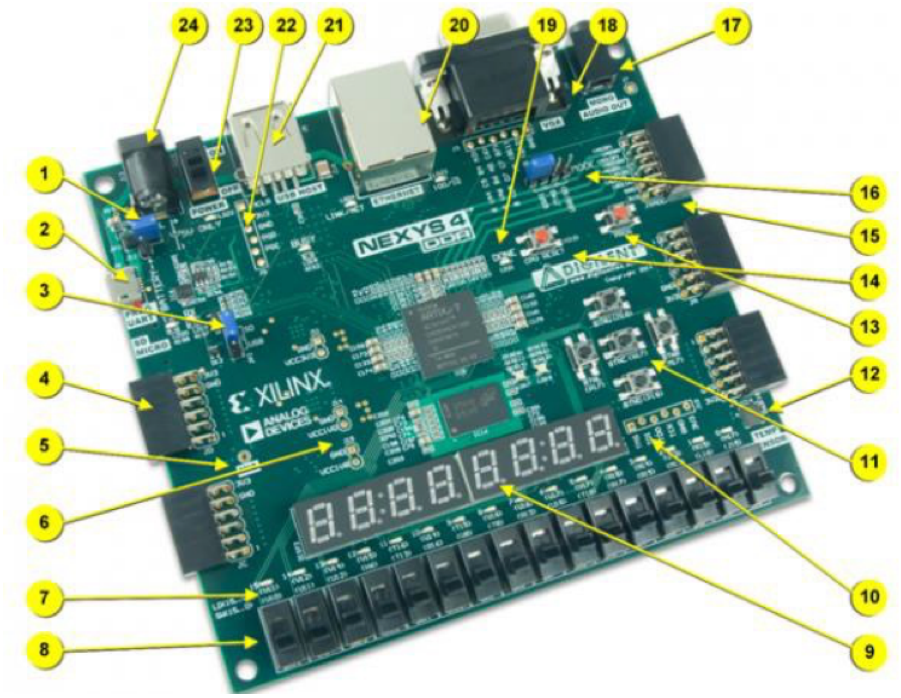

图 3: Nexys4 DDR 示意图

1	选择供电跳线	13	FPGA 配置复位按键
2	UART/ JTAG 共用 USB 接口	14	CPU 复位按键 (用于软核)
3	外部配置跳线柱 (SD / USB)	15	模拟信号 Pmod 端口 (XADC)
4	Pmod 端口	16	编程模式跳线柱
5	扩音器	17	音频连接口
6	电源测试点	18	VGA 连接口
7	16 个 LED	19	FPGA 编程完成 LED
8	16 个拨键开关	20	以太网连接口
9	8 位 7 段数码管	21	USB 连接口
10	可选用于外部接线的 JTAG 端口	22	(工业用) PIC24 编程端口
11	5 个按键开关	23	电源开关
12	板载温度传感器	24	电源接口

表 6: Nexys4 DDR 功能表

B Nexys4 DDR 引脚说明

100MHz 时钟 E3

按键、拨码管、LED、七段数码管、Reset:

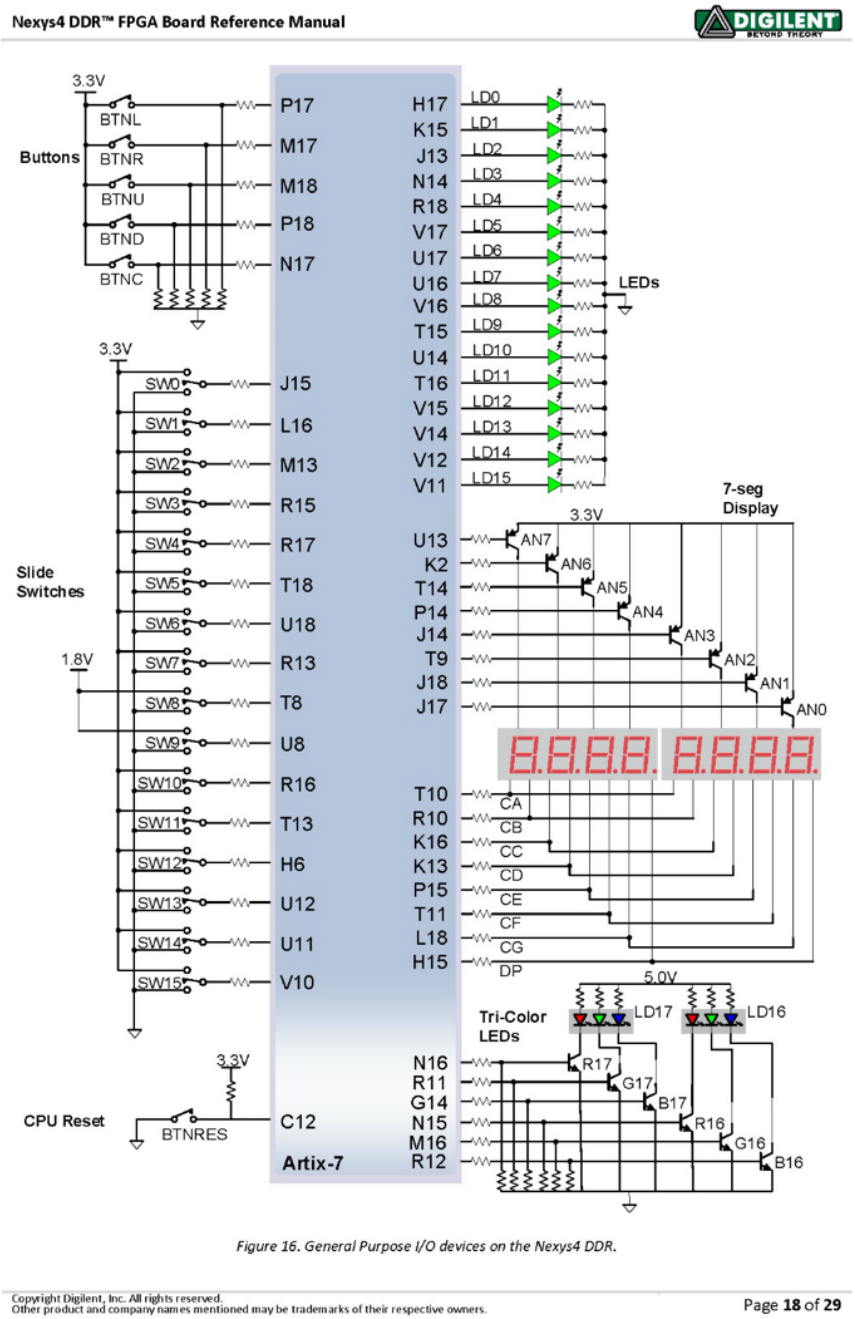

Figure 16. General Purpose I/O devices on the Nexys4 DDR.

图 4: Nexys4 DDR 管脚图

C 七段数码管的使用

Nexys4 DDR 实验板上有两个 4 位带小数点的七段数码管，图 5显示了它们与主芯片的连接方式。其中 A7 A0 是数码管 8 个位的使能信号，而 CA CG/DP 则对应各个位上七个段以及小数点的触发信号。需要注意的是，使能信号和触发信号都是低电平触发的。

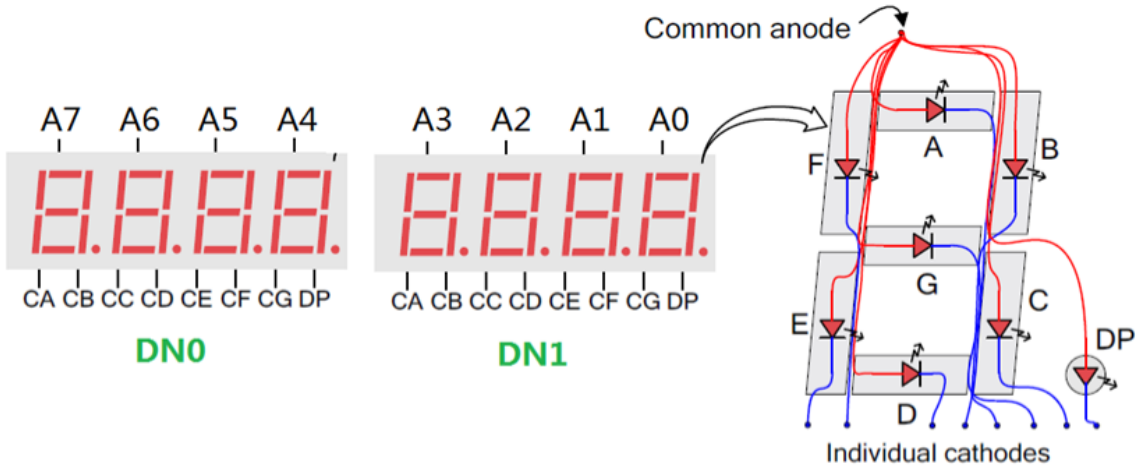

图 5: 七段数码管示意图

图 5以数码管中最右侧的 A0 数码管为例说明了 Nexys4 DDR 板卡上的 7-段数码管的连接方式。8 个位中的各个相应的段及小数点分别连接到一组低电平触发的引脚上，他们被称为 CA、CB、CC、…、CG、DP，其中，CA 接到这 8 个数码管中每一个数码管 A 段的负极，CB 接到这 8 个数码管中每一个数码管 B 段的负极，以此类推。

此外，每一个数码管都有一个使能信号 A[7:0]。A[7:0] 通过一个反相器接到对应数码管的每一个段的正极上。比如说，只有到 A[0] 为 0 的时候，最右侧数码管的显示才会受到 CA…CG 这几个信号的驱动。

图 6中列出了数码管显示 0 到 F 时点亮的段。比如说在显示数字 0 的时候，除了中间的 G 段外其他的段都被点亮了。而数字 1 只点亮了 B 段和 C 段。

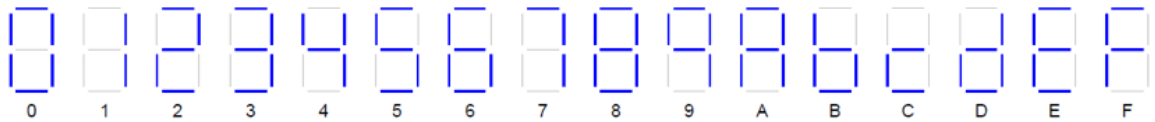

图 6: 0-F 点亮示意图

要想让每个数码管显示不同的数字，使能信号（A[7:0]）和段信号（CA…CG）必须依次地被持续驱动，数码管之间的刷新速度应该足够快这样就看不出来数码管之间在闪烁。举个例子，如果想在数码管 0 上显示数字 3 而数码管 1 上显示数字 9，可以先把 CA…CG 设置为显示数字 3，并拉低 A[1] 信号，然后再把 CA…CG 设置为显示数字 9 并拉高 A[1] 拉低 A[2]。刷新频率可以设置为 2ms 刷新一次，这样人眼就看不出闪烁了。