

数字逻辑电路实验报告

实验题目： DCE-01 阻塞与非阻塞电路

实验日期： 2025 年 10 月 18 日, 2025 年 10 月 25 日

实验者姓名： 丁毅 2023K8009908031

1 实验目的

- 1、掌握阻塞赋值与非阻塞赋值的概念和区别；
- 2、了解阻塞和非阻塞赋值的不同使用场合；
- 3、掌握测试模块的编写、综合和不同层次的仿真；
- 4、掌握 EGO1 开发板的使用。

2 实验原理

在数字电路设计中，阻塞赋值 (blocking assignment) 与非阻塞赋值 (nonblocking assignment) 在Verilog语言中承担着不同的功能角色，它们各自对应着特定的应用场景。这两种赋值方式的核心差异体现在其综合 (synthesis) 后所形成的电路结构上。阻塞赋值通常与输入信号电平 (signal level) 的变化直接关联，当在设计组合逻辑 (combinational logic) 的always块中采用它时，综合工具会将其实现为组合逻辑电路。而非阻塞赋值的行为则与时钟触发沿 (clock edge) 紧密相关，在描述时序逻辑 (sequential logic) 的 always 块中主要使用此种赋值方式，其综合结果通常为时序逻辑电路结构。

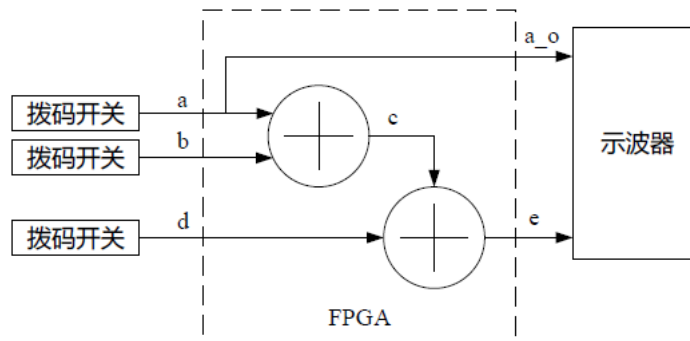


图 1. 组合电路阻塞赋值实验原理

本次实验分别对组合电路阻塞赋值与时序电路非阻塞赋值的输出结果进行观测。图 1 与图 2 分别为组合电路阻塞赋值与时序电路非阻塞赋值的实验原理，实验中的输入信号 a、b 和 d 均为 1 比特（bit）宽度，它们通过三个拨码开关（DIP switch）来提供。经过运算产生两个 2 比特宽度的输出，即 c[1:0]与 e[1:0]，其中 c[1:0]被用作中间结果（intermediate result），而 e[1:0]的最高位 e[1]则被送至示波器进行观测。为了清晰地分析输入与输出信号之间的时序关系（timing relationship），输入信号 a 也被同步输出至示波器。

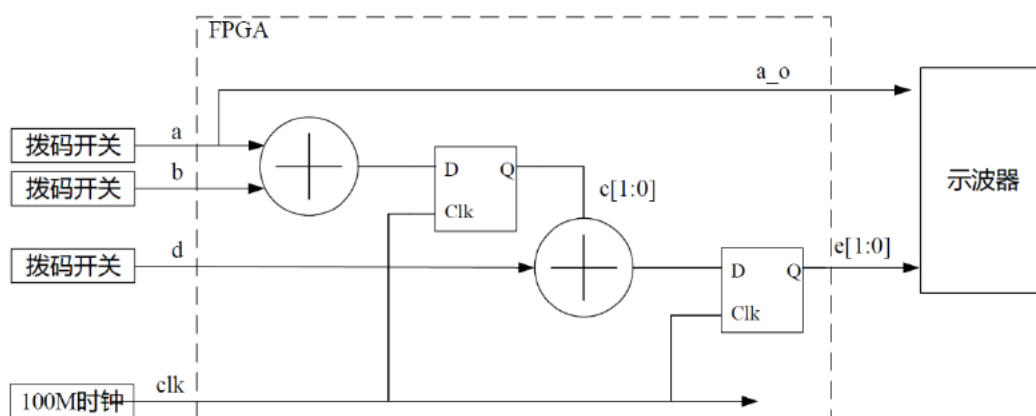


图 2. 时序电路非阻塞赋值实验原理

实验开始时设定初始状态，令 $a=0$, $b=0$, $d=1$ ，此时对应的输出为 $a_o=0$, $e[1]=0$ 。随后，将控制信号 a 的拨码开关拨动至 1，即使得 $a=1$ ，而 $b=0$, $d=1$ 保持不变，此时可以观察到输出变为 $a_o=1$, $e[1]=1$ 。利用示波器所具有的边沿触发（edge trigger）功能，可以分别精确测量在组合逻辑阻塞赋值和时序逻辑非阻塞赋值这两种不同情况下，输出信号 e[1] 的变化相对于参考信号 a_o 变化所产生的延时。

3 实验仪器和设备

EGO1 开发板、RIGOL MSO2202 示波器、导线等。

4 实验方法和步骤

1、使用 vivado 创建工程，并编写三种赋值电路的 Verilog 代码：

(1) combinational + blocking (组合电路阻塞赋值)

-
- (2) sequential + nonblocking (时序电路非阻塞赋值)
 - (3) sequential + blocking (时序电路阻塞赋值)
 - 2、编写 testbench (测试激励文件):
 - (4) testbench for combinational circuit (组合电路)
 - (5) testbench for both sequential modules (两个时序电路同时测试)
 - 3、完成各模块 behavior-level simulation (行为级仿真)
 - 4、将代码下载到 FPGA，使用示波器观察三种电路的输入输出的时序关系

5 实验内容和结果

- 1、使用 vivado 创建工程，并编写三种赋值电路的 Verilog 代码:

- (1) combinational + blocking (组合电路阻塞赋值)

```
// (1) 组合 + 阻塞赋值
// md_combinational_blocking.v

module md_combinational_blocking(
    input IN_A,
    input IN_B,
    input IN_D,
    output OUT_A0,
    output reg [1:0] OUT_E
);

    // 1. Variable Declaration (变量声明)
    reg[1:0] c = 0; // 在 Verilog 中，reg 变量可以在声明时赋初始值（如 reg [1:0] c = 0;），但这主要用于仿真；在 FPGA 综合时，初始值通常被忽略，建议在 always 块中通过复位逻辑设置初始值以确保可综合性。

    // 2. Main Code (主代码)
    always @(IN_A, IN_B, IN_D) begin // 这里逗号和 or 的作用等价
        c = IN_A + IN_B;
        OUT_E = c + IN_D;
    end

    // 3. Output Assignment (输出赋值)
    assign OUT_A0 = IN_A;

endmodule
```

在该组合电路中，`always @(IN_A, IN_B, IN_D)` 表示后续电路值的变化依赖于输入的变化，且立即执行，并不依赖于时钟的变化，同时在 `always` 块内电路采用的是阻塞赋

值，最后的 assign 语句是一个连续赋值语句。这个模块是一个纯粹的组合电路。组合电路的输出仅由当前的输入决定，不涉及任何时序控制。

(2) sequential + nonblocking (时序电路非阻塞赋值)

```
// (2) 时序 + 非阻塞赋值
// md_sequential_nonblocking.v

module md_sequential_nonblocking(
    input clk,
    input IN_A,
    input IN_B,
    input IN_D,
    output OUT_A0,
    output reg [1:0] OUT_E
);

    // 1. Variable Declaration (变量声明)
    reg [1:0] c = 0; // 在 Verilog 中，reg 变量可以在声明时赋初始值（如 reg [1:0] c = 0;），但这主要用于仿真；在 FPGA 综合时，初始值通常被忽略，建议在 always 块中通过复位逻辑设置初始值以确保可综合性。

    // 2. Main Code (主代码)
    always @(posedge clk) begin
        c <= IN_A + IN_B; // 非阻塞赋值：后语句 **不可见** 前语句结果
        OUT_E <= c + IN_D;
    end

    // 3. Output Assignment (输出赋值)
    assign OUT_A0 = IN_A;

endmodule
```

其中，输出变量的改变是在时钟上升沿 `always @(posedge clk)` 才发生的，且块内使用了非阻塞赋值运算符“<=”，是一个时序非阻塞赋值电路。

(3) sequential + blocking (时序阻塞赋值)

```
// (3) 时序 + 阻塞赋值
// md_sequential_blocking.v

module md_sequential_blocking(
    input clk,
    input IN_A,
    input IN_B,
```

```

    input IN_D,
    output OUT_A0,
    output reg [1:0] OUT_E
);

    // 1. Variable Declaration (变量声明)
    reg [1:0] c = 0; // 在 Verilog 中, reg 变量可以在声明时赋初始值 (如 reg [1:0] c
= 0;), 但这主要用于仿真; 在 FPGA 综合时, 初始值通常被忽略, 建议在 always 块中通过复
位逻辑设置初始值以确保可综合性。

    // 2. Main Code (主代码)
    always @(posedge clk) begin
        c = IN_A + IN_B; // 阻塞赋值: 按顺序执行, 后语句可见前语句结果
        OUT_E = c + IN_D;
    end

    // 3. Output Assignment (输出赋值)
    assign OUT_A0 = IN_A;

endmodule

```

与 (2) sequential + nonblocking (时序电路非阻塞赋值) 相比, 此处只是把 always 块中的非阻塞赋值改为了阻塞赋值。在时序电路中, 阻塞赋值意味着赋值操作会按顺序立即执行, 导致运算顺序和结果可能与非阻塞赋值不同 (我们将在后面的 testbench 看到), 在设计中需要谨慎使用。

2、编写 testbench (测试激励文件), 完成 behavior-level simulation (行为级仿真)

(4) testbench for combinational circuit (组合电路)

```

// (4) testbench (组合电路): `tb_combinational.v`

/*
说明 (针对 Verilog `timescale 1ns/1ps`):

- 时间单位 (time unit = 1ns):
    - 源代码中的时间延迟字面量以 1 纳秒 为单位。例如 `#1` 表示 1 纳秒, `#2` 表示 2 纳
秒。
    - 所有以单位为基础的延迟都会按此单位来解释和显示。

- 时间精度 (time precision = 1ps):
    - 仿真的时间分辨率为 1 皮秒, 所有时间值在内部按 1ps 的精度进行表示和四舍五入。
    - 比如 0.5ns = 500ps, 会以 1ps 为最小步长来表示; 小于 1ps 的值将按 1ps 精度舍入。

- 影响与注意事项:
    - 精度应小于或等于时间单位 (通常选择比单位更细的精度以获得准确结果)。

```

- 更细的时间精度能提高仿真精度，但会增加仿真开销（时间和内存）。
- ``timescale`` 指令对当前源文件中的模块有效，通常应在模块声明之前出现；如果缺省，仿真器会使用默认 `timescale`。
- 示例（概念性）：
 - ``#1`` \rightarrow 1 ns
 - ``#0.5`` \rightarrow 0.5 ns = 500 ps（在 1ps 精度下表示为 500 ps）

```
*/  
  
`timescale 1ns/1ps  
  
module tb_combinational;  
  
    // 1. Variable Declaration (变量声明)  
    reg in_a;  
    reg in_b;  
    reg in_d;  
    wire out_a0;  
    wire [1:0] out_e;  
  
    // 2. Instantiate the Unit Under Test (UUT) (被测单元实例化)  
    md_combinational_blocking uut (  
        .IN_A(in_a),  
        .IN_B(in_b),  
        .IN_D(in_d),  
        .OUT_A0(out_a0),  
        .OUT_E(out_e)  
    );  
  
    // 3. Testbench Logic (测试逻辑)  
    initial begin  
        // 1. Initialize Inputs (初始化输入)  
        in_a = 0;  
        in_b = 0;  
        in_d = 0;  
  
        // 2. Apply Test Vectors (施加测试向量)  
        #10 in_a = 1; in_b = 0; in_d = 1;  
        #10 in_a = 0; in_b = 1; in_d = 0;  
        #10 in_a = 1; in_b = 1; in_d = 1;  
  
        // 3. Finish Simulation (结束仿真)  
        #10 $finish;  
  
        // 4. Monitor Outputs (监测输出)  
        $monitor("Time: %0t | IN_A: %b | IN_B: %b | IN_D: %b | OUT_A0: %b | OUT_E: %b",  
$time, in_a, in_b, in_d, out_a0, out_e);
```

```
end
```

```
endmodule
```

(5) testbench for both sequential modules (两个时序电路同时测试)

```
// Testbench for both sequential modules simultaneously (两个时序电路同时测试):  
`tb_sequential_both.v`
```

```
`timescale 1ns/1ps
```

```
module tb_sequential_both;
```

```
    // 1. Variable Declaration (变量声明)
```

```
    reg clk;
```

```
    reg in_a;
```

```
    reg in_b;
```

```
    reg in_d;
```

```
    wire out_a0_blocking;
```

```
    wire [1:0] out_e_blocking;
```

```
    wire out_a0_nonblocking;
```

```
    wire [1:0] out_e_nonblocking;
```

```
    parameter PERIOD = 20;
```

```
    // 2. Instantiate the Units Under Test (UUTs)
```

```
    md_sequential_blocking uut_blocking (
```

```
        .clk(clk),
```

```
        .IN_A(in_a),
```

```
        .IN_B(in_b),
```

```
        .IN_D(in_d),
```

```
        .OUT_A0(out_a0_blocking),
```

```
        .OUT_E(out_e_blocking)
```

```
    );
```

```
    md_sequential_nonblocking uut_nonblocking (
```

```
        .clk(clk),
```

```
        .IN_A(in_a),
```

```
        .IN_B(in_b),
```

```
        .IN_D(in_d),
```

```
        .OUT_A0(out_a0_nonblocking),
```

```
        .OUT_E(out_e_nonblocking)
```

```
    );
```

```

// 3. Testbench Logic (测试逻辑)
initial begin
    // Clock generation (时钟生成)
    clk = 1'b0;
    #(PERIOD/2);
    forever #(PERIOD/2) clk = ~clk;
end

integer delay1_a, delay2_a, k_a;
initial begin
    // Initialize Inputs (初始化输入)
    in_a = 0;
    in_b = 0;
    in_d = 0;
    #10;
    // Generate random signals for in_a (生成 in_a 的随机信号)
    for(k_a = 0; k_a < 200; k_a = k_a + 1) begin
        delay1_a = (PERIOD/3) * ({ $random } % 3);
        delay2_a = (PERIOD/3) * ({ $random } % 3);
        #delay1_a in_a = 1;
        #delay2_a in_a = 0;
    end
end

integer delay1_b, delay2_b, k_b;
initial begin
    // Generate random signals for in_b (生成 in_b 的随机信号)
    #10 in_b = 0;
    for(k_b = 0; k_b < 200; k_b = k_b + 1) begin
        delay1_b = (PERIOD/3) * ({ $random } % 5);
        delay2_b = (PERIOD/3) * ({ $random } % 5);
        #delay1_b in_b = 1;
        #delay2_b in_b = 0;
    end
end

integer delay1_d, delay2_d, k_d;
initial begin
    // Generate random signals for in_d (生成 in_d 的随机信号)
    #10 in_d = 0;
    for(k_d = 0; k_d < 200; k_d = k_d + 1) begin
        delay1_d = (PERIOD/3) * ({ $random } % 6);
        delay2_d = (PERIOD/3) * ({ $random } % 6);
        #delay1_d in_d = 1;
        #delay2_d in_d = 0;
    end
end

```

```

end

// 4. Monitor Outputs (监测输出)
initial begin
    $monitor("Time: %0t | clk: %b | IN_A: %b | IN_B: %b | IN_D: %b | Blocking
OUT_A0: %b | OUT_E: %b | Nonblocking OUT_A0: %b | OUT_E: %b", $time, clk, in_a,
in_b, in_d, out_a0_blocking, out_e_blocking, out_a0_nonblocking,
out_e_nonblocking);
end

endmodule

```

3、完成各模块 behavior-level simulation (行为级仿真)

(1) combinational + blocking (组合电路阻塞赋值) 的仿真结果如下：

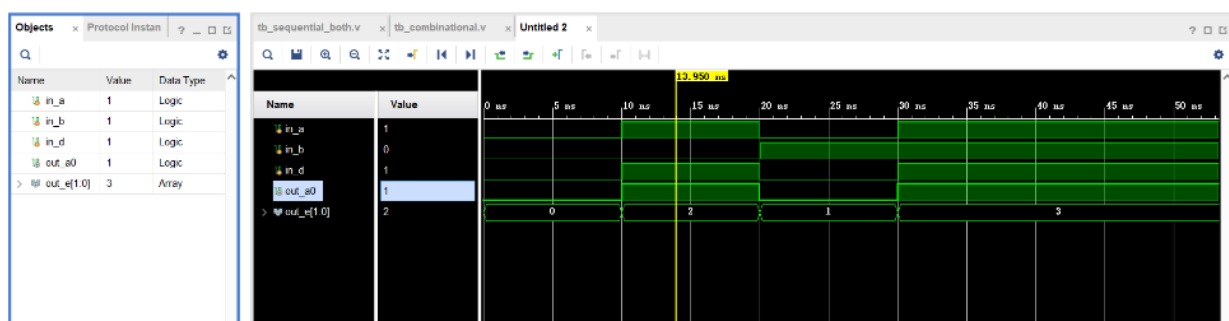


图 3. (1) combinational + blocking (组合电路阻塞赋值) 行为级仿真结果

组合电路不含时序元件，输入变化后，输出会通过一系列逻辑门（构成的组合电路）立即响应。

(2) sequential + nonblocking (时序电路非阻塞赋值) 和 (3) sequential + blocking (时序阻塞赋值) 的仿真结果如下：

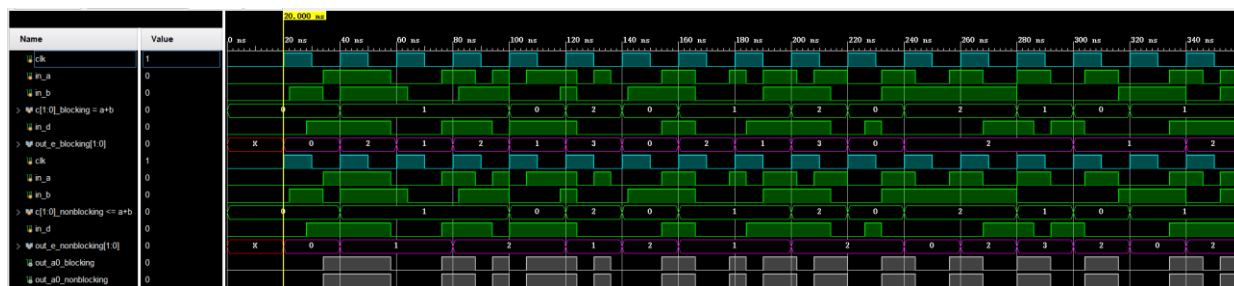


图 4. (2) sequential + nonblocking (时序电路非阻塞赋值) 和 (3) sequential + blocking (时序阻塞赋值) 行为级仿真结果

首先明确：如果在时钟上升沿瞬间模块的输入发生了变化，**模块看的是上升沿之前的值 (D Flipflop 的性质所致)**，而不是之后的值。

在我们的时序代码中，第一句 `c = a + b` or `c <= a + b` 的运算是完全相同的，因为它们都是在时钟上升沿瞬间 "保存输入初始值 (上升沿之前的值)"，然后进行运算 `a + b` 并赋值给 `c`。这里阻塞和非阻塞的区别体现在第二句 `e = c + d` or `e <= c + d` 上：

- 时序 + 非阻塞：等号右端所有用到的值，也即 `a, b, c, d` 都是上升沿时保存下来的初始值 (保存的是上升沿前一瞬间的值)，本次 `c + d` 运算 "看不到" `c <= a + b` 的新结果值；换句话说，本次 `c + d` 运算使用的是上次 `c <= a + b` 计算得到的值。**等价于 $e_{\{n\}} = a_{\{n-1\}} + b_{\{n-1\}} + d_{\{n\}}$ 的效果。**
- 时序 + 阻塞：仅有整个模块的输入量 `a, b, d` 上升沿时保存下来，语句 `e = c + d` 会先等待 `c` 完成更新，被赋值为 `a + b` 的新结果，然后才来计算 `c + d` 并赋值给 `e`，所以本次 `c + d` 运算 "看到了" 本次 `c = a + b` 的新结果值。**等价于 $e = (a + b) + d$ ，也即 $e_{\{n\}} = a_{\{n\}} + b_{\{n\}} + d_{\{n\}}$ 的效果。**

不妨在图中找一个具体的例子：

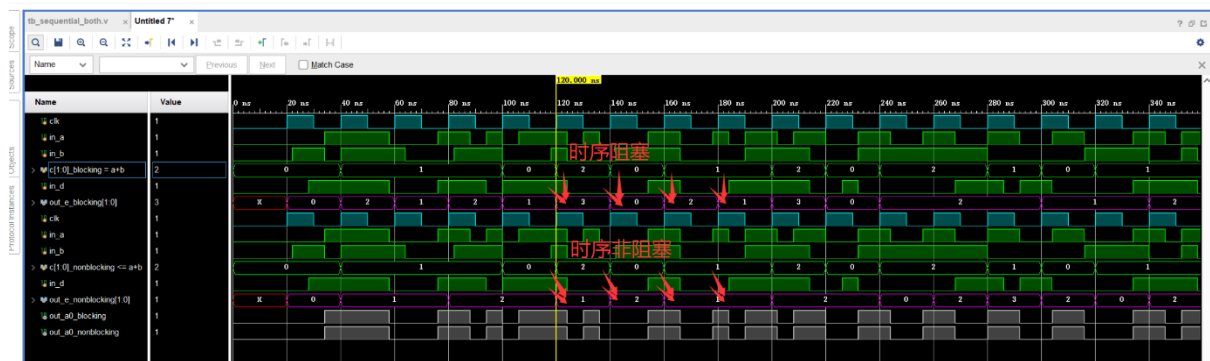


图 5. 时序电路阻塞和非阻塞赋值对比

4、将代码下载到 FPGA，使用示波器观察三种电路的输入输出的时序关系

编写好管脚约束后将程序编译烧写到开发板上，利用示波器单次触发功能，观察输出 `e[1]` 和输入 `a` 的时序关系，结果如下：

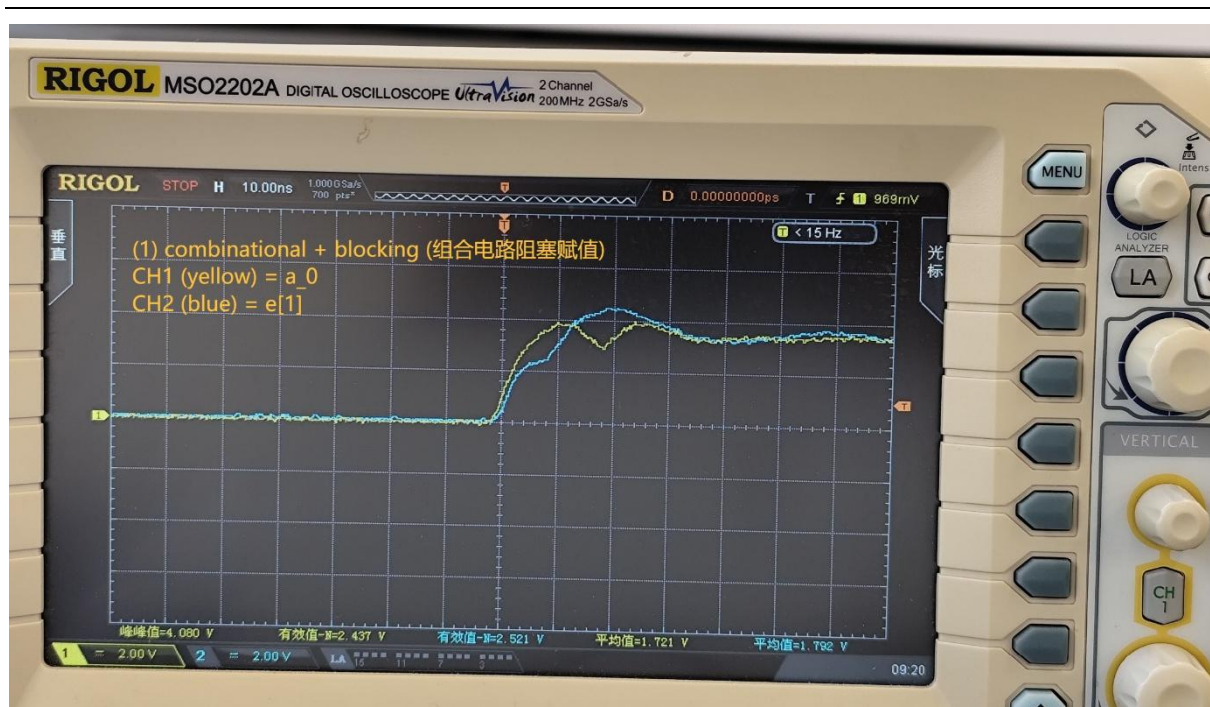


图 6. (1) combinational + blocking (组合电路阻塞赋值) 示波器测量结果

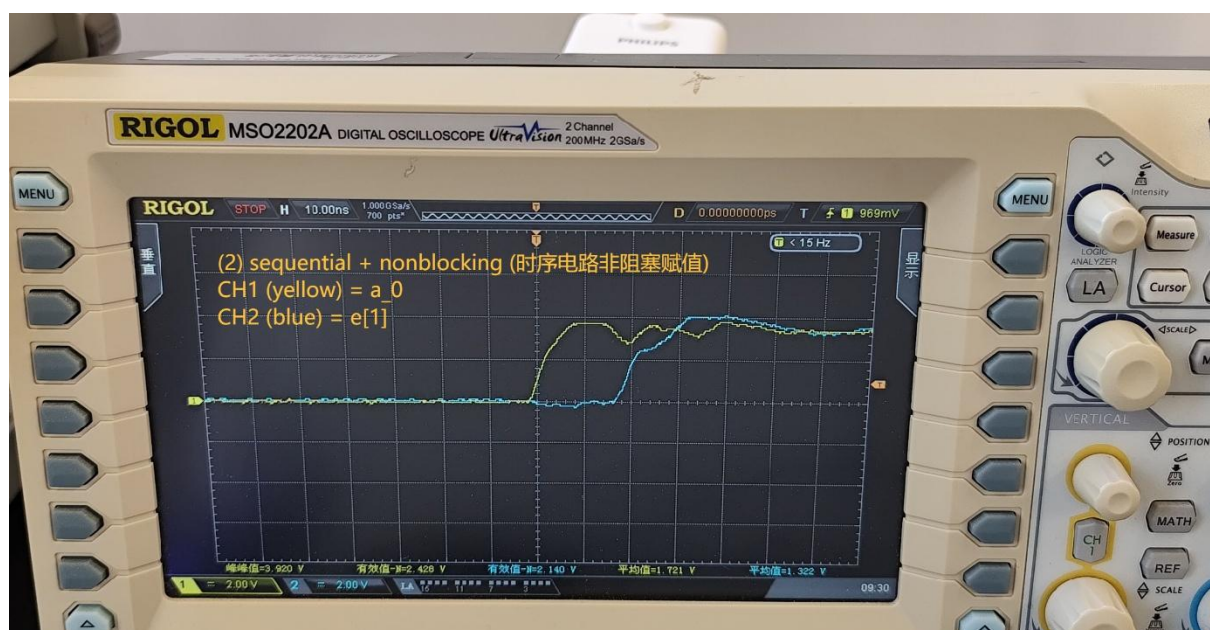


图 7. (2) sequential + nonblocking (时序电路非阻塞赋值) 示波器测量结果

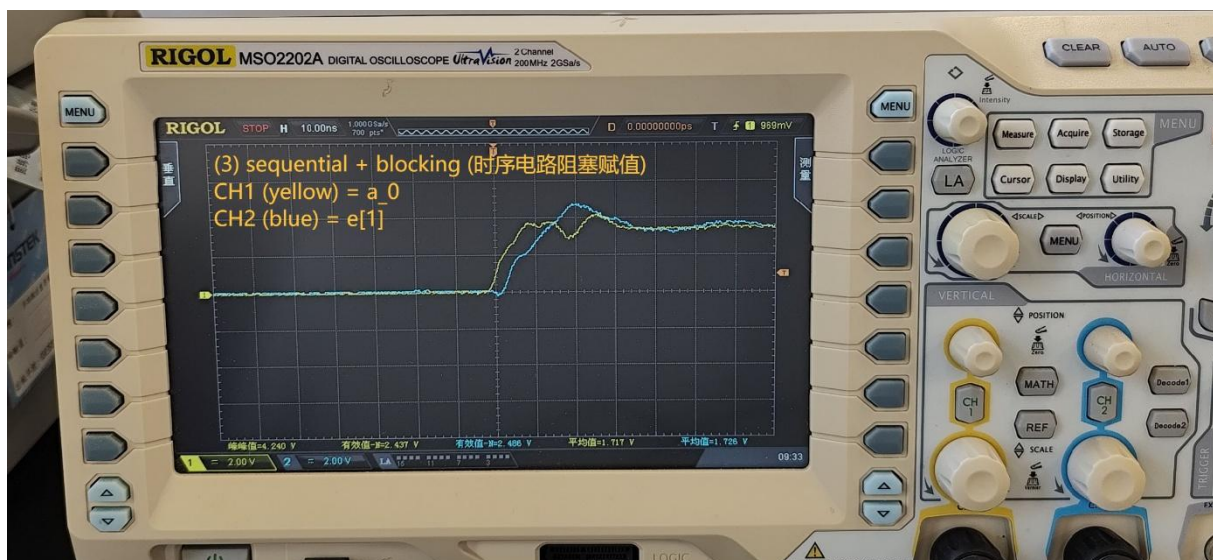


图 8. (3) sequential + blocking (时序阻塞赋值) 示波器测量结果

图中可以看到, (2) sequential + nonblocking (时序电路非阻塞赋值) 的延迟最高, 约为 16.4 ns, 其次是 (3) sequential + blocking (时序阻塞赋值), 约为 3.1 ns, 延迟最小的是 (1) combinational + blocking (组合电路阻塞赋值), 延迟仅 0.7 ns。

6 思考题及结论

(1) 时序电路中, 阻塞与非阻塞赋值的区别是什么?

以此次实验的相关代码为例:

时序非阻塞赋值: 等号右端所有用到的值, 也即 ``a, b, c, d`` 都是时钟上升沿时保存下来的初始值 (保存的是上升沿前一瞬间的值), 本次 ``c + d`` 运算 "看不到" ``c <= a + b`` 的新结果值; 换句话说, 本次 ``c + d`` 运算使用的是上次 ``c <= a + b`` 计算得到的值。最终等价于 ``e_{n} = a_{n-1} + b_{n-1} + d_{n}` 的效果。

时序阻塞赋值: 时钟上升沿到来时, 仅保存整个模块几个输入端口的初始值, 也即 ``a, b, d`` 在上升沿被保存下来。语句 ``e = c + d`` 会先等待 ``c`` 完成更新, 被赋值为 ``a + b`` 的新结果, 然后才来计算 ``c + d`` 并赋值给 ``e``, 所以本次 ``c + d`` 运算 "看到了" 本次 ``c = a + b`` 的新结果值。等价于 ``e = (a + b) + d``, 也即 ``e_{n} = a_{n} + b_{n} + d_{n}` 的效果。

图 5 中的结果验证了我们的分析。

(2) 组合 + 阻塞, 时序 + 非阻塞, 时序 + 阻塞, 三种方法的响应速度如何?

总体而言, 组合逻辑 + 阻塞赋值的响应速度最快, 而时序逻辑 + 非阻塞赋值最慢, 其响应存在一个时钟周期的固有延迟; 时序逻辑与阻塞赋值虽然在仿真中可能表现出与

组合逻辑类似的快速响应，但这通常是一种不符合真实硬件行为的仿真假象，在实际综合电路中会导致无法预料的问题。

需要注意的是，如果在描述时序逻辑的 `always` 块中错误地使用了阻塞赋值，其仿真行为会变得复杂且危险。在同一个 `always` 块内，后续的赋值语句会立即看到前面赋值语句的结果，这可能导致仿真器在一个时钟周期内就计算出最终的寄存器值，从而在仿真中表现出“零延迟”的快速响应。但这种行为与真实的寄存器工作原理不符，综合后的电路可能是一个组合逻辑与触发器的混合体，其实际响应速度取决于关键路径的延迟，并且很可能产生与仿真结果不一致的竞争冒险和功能错误，因此这种用法在正确设计中被严格禁止。