

FPGA STORY 《驱动篇 I 》



ALINX 系列教程

追求精品 宁缺莫乱



黑金动力社区
荣誉呈献
www.heijin.org

实验一：流水灯模块

对于发展商而言，动土仪式无疑是最重要的任务。为此，流水灯实验作为低级建模 II 的动土仪式再适合不过了。废话少说，我们还是开始实验吧。



图 1.1 实验一建模图。

如图 1.1 所示，实验一有名为 led_funcmod 的功能模块。如果无视环境信号（时钟信号还有复位信号），该功能模块只有一组输出端，亦即 4 位 LED 信号。接下来让我们来看具体内容：

led_funcmod.v

```
1. module led_funcmod
2. (
3.     input CLOCK, RESET,
4.     output [3:0]LED
5. );
```

以上内容为出入端声明。

```
6.     parameter T1S = 26'd50_000_000; //1Hz
7.     parameter T100MS = 26'd5_000_000; //10Hz
8.     parameter T10MS = 26'd500_000; //100Hz
9.     parameter T1MS = 26'd50_000; //1000Hz
10.
```

以上内容为常量声明。分别是 1 秒至 1 毫秒。

```
11.    reg [3:0]i;
12.    reg [25:0]C1;
13.    reg [3:0]D;
14.    reg [25:0]T;
15.    reg [3:0]isTag;
16.
17.    always @ ( posedge CLOCK or negedge RESET )
```

```

18.         if( !RESET )
19.             begin
20.                 i <= 4'd0;
21.                 C1 <= 26'd0;
22.                 D <= 4'b0001;
23.                 T <= T1S;
24.                 isTag <= 4'b0001;
25.             end
26.         else

```

以上内容是相关的寄存器声明以及复位操作。寄存器 i 用来指向步骤，寄存器 C1 用来计数，寄存器 D 用来暂存结果和驱动输出，寄存器 T 用来暂存计数数量，isTag 则用来暂存延迟标签。

```

27.         case( i )
28.
29.             0:
30.                 if( C1 == T -1) begin C1 <= 26'd0; i <= i + 1'b1; end
31.                 else begin C1 <= C1 + 1'b1; D <= 4'b0001; end
32.
33.             1:
34.                 if( C1 == T -1) begin C1 <= 26'd0; i <= i + 1'b1; end
35.                 else begin C1 <= C1 + 1'b1; D <= 4'b0010; end
36.
37.             2:
38.                 if( C1 == T -1) begin C1 <= 26'd0; i <= i + 1'b1; end
39.                 else begin C1 <= C1 + 1'b1; D <= 4'b0100; end
40.
41.             3:
42.                 if( C1 == T -1) begin C1 <= 26'd0; i <= i + 1'b1; end
43.                 else begin C1 <= C1 + 1'b1; D <= 4'b1000; end
44.
45.             4:
46.                 begin isTag <= { isTag[2:0], isTag[3] }; i <= i + 1'b1; end
47.
48.             5:
49.                 if( isTag[0] ) begin T <= T1S; i <= 4'd0; end
50.                 else if( isTag[1] ) begin T <= T100MS; i <= 4'd0; end
51.                 else if( isTag[2] ) begin T <= T10MS; i <= 4'd0; end
52.                 else if( isTag[3] ) begin T <= T1MS; i <= 4'd0; end
53.
54.             endcase
55.

```

```
56.      assign LED = D;
57.
58. endmodule
```

以上内容为是核心操作以及输出驱动声明。步骤 0~3 用来实现流水灯效果。最初，每个步骤的停留时间是 1 秒，然后步骤 0~3 按顺序执行便会产生流水效果。步骤 4 是用来切换模式，步骤 5 则是根据 isTag 的内容再为 T 寄存器载入不同的延迟内容，如 [0] 延迟 1 秒，[1] 延迟 100 毫秒，[2] 延迟 10 毫秒，[3] 延迟 1 毫秒。默认下为模式 0（第 24 行），既延迟 1 秒（第 23 行）。

这个实验所在乎的内容根本不是实验结果而是低级建模 II 本身。假若比较《建模篇》的流水实验，低级建模 I 与低级建模 II 之间是有明显的差距。首先，低级建模 II 再也不见计数器或者定时器等周边操作。再者，低级建模 II 的整合度很高，例如步骤 0~3：

```
1.      0:
2.      if( C1 == T -1) begin C1 <= 26'd0; i <= i + 1'b1; end
3.      else begin C1 <= C1 + 1'b1; D <= 4'b0001; end
4.      1:
5.      if( C1 == T -1) begin C1 <= 26'd0; i <= i + 1'b1; end
6.      else begin C1 <= C1 + 1'b1; D<= 4'b0010; end
7.      2:
8.      if( C1 ==T -1) begin C1 <= 26'd0; i <= i + 1'b1; end
9.      else begin C1 <= C1 + 1'b1; D <= 4'b0100; end
10.     3:
11.     if( C1 == T -1) begin C1 <= 26'd0; i <= i + 1'b1; end
12.     else begin C1 <= C1 + 1'b1; D <= 4'b1000; end
```

代码 1.1

内容如代码 1.1 所示，步骤 0~3 每个步骤示意一段完整的小操作，例如步骤 0 为 4'b0001 保持一段时间，步骤 1 为 4'b0010 保持一段时间，步骤 2~3 也是如此。其中 -1 也考虑了步骤切换的时间。假设流水间隔要求 1 毫秒，那么每个步骤都会准确无误停留 50000 个时钟。事实上，步骤 0 也可以换成比较方便的写法，如代码 1.2 所示：

```
1.      reg [3:0] D = 4' b0001;
2.      .....
3.      0,1,2,3 :
4.      if( C1 == T -1) begin D <= { D[2:0], D[3] }; C1 <= 26'd0; i <= i + 1'b1; end
5.      else begin C1 <= C1 + 1'b1; end
```

代码 1.2

代码 1.2 表示，只要寄存器 D 准备好初值，例如 4'b0001，那么步骤 0~3 都可以共享同样的操作，如此一来会大大减少行数，节省空间。好奇的同学一定觉得疑惑，既然代码

1.2 的写法那么方便，反之笔者为何要选择代码 1.1 的写法呢？原因很单纯，那是为了清晰模块内容，以致我们容易脑补时序。感觉上，两者虽然都差不多，但是我们只要仔看，我们便会发现 … 代码 1.2 它虽然书写方便，可是细节模糊而且内容也不直观。

在此，笔者需要强调！低级建模 II 虽然有整合技巧让操作变得更加便捷，不过比起整合它更加注重表达能力以及清晰度。这样做的关键是为了发挥主动思想，以便摆脱无谓的仿真。所以说，如果读者想和低级建模 II 作朋友，建模之前应该优先考虑内容的清晰度，而不是内容的精简性。如果内容即精简又直观，这种情况当然是最好的结果。

至于实验一是否真的必要，因为内容足够直白，这种程度足以脑补时序。最后笔者还要说道，实验一虽然没有什么学习的价值，但是实验一要表达的信息也非常清楚，即低级建模 II 是注重清晰，直观的建模技巧。此外，实验一也可以作为学习的热身。

实验二：按键模块① - 消抖

按键消抖实验可谓是经典中的经典，按键消抖实验虽曾在《建模篇》出现过，而且还惹来一堆麻烦。事实上，笔者这是在刁难各位同学，好让对方的惯性思维短路一下，但是惨遭口水攻击 … 面对它，笔者宛如被甩的男人，对它又爱又恨。不管怎么样，如今 I'll be back，笔者再也不会重复一样的悲剧。

按键消抖说傻不傻说难不难。所谓傻，它因为原理不仅简单（就是延迟几下而已），而且顺序语言（C 语言）也有无数不尽的例子。所谓难，那是因为人们很难从单片机的思维跳出来 … 此外，按键消抖也有许多细节未曾被人重视，真是让人伤心。按键消抖一般有 3 段操作：

- 检测电平变化；
- 过滤抖动（延迟）；
- 产生有效按键。

假设 C 语言与单片机的组合想要检测电平变化，它们一般是利用 if 查询或者外部中断。事后，如果这对组合想要过滤抖动，那么可以借用 for 延迟的力量，又或者依赖定时中断产生精明的延迟效果。反观有效案件的产生，这对组合视乎而外钟情“按下有效”似的 … 不管怎么样，C 语言与单片机这对组合在处理按键的时候，它们往往会错过一些黄金。

“黄金？”，读者震撼道。

所谓黄金时间就是电平发生变化那一瞬间，还有消抖（延迟）以后那一瞬间。按键按下期间，按键的输入电平故会发生变化，如果使用 if 查询去检测，结果很容易浪费单片机的处理资源，因为单片机必须一直等待 … 换之，如果反用外部中断，中断寻址也会耽误若干时间。

假设 C 语言与单片机这对组合挨过电平检测这起难关，余下的困难却是消抖动作。如果利用 for 循环实现去消抖，例如 Delay_ms(10) 之类的函数。For 循环不仅计数不紧密，而且还会白白浪费单片机的处理资源。定时中断虽然计数紧密，但是中断触发依然也会产生若干的寻址延迟。补上，所谓寻址延迟是处理器处理中断触发的时候，它要事先保护现场之余，也要寻址中断处理入口，然后执行中断函数，完后回复现场，最后再返回当前的工作。

感觉上，笔者好似在欺负 C 语言以及单片机，死劲说它们的不是。亲爱的读者，千万别误会，笔者只是在陈述事实而已。单片机本来就是比较粗野的爷们，它很难做到紧凑又毫无空隙的操作，反观 FPGA 却异常在行。所以说，单片机的思路很难沿用在 FPGA 身上，否则会出现许多笑话。如今，这是描述语言以及 FPGA 的新时代，所谓后浪推前浪正是新旧时代的替换。

FPGA 不仅没有隐性处理，而且描述语言也是自由自身。我们只要方法得当，手段有效，“黄金”要多少就有多少 ... 哇哈哈！在此，笔者说了那么多废话只是告知读者，千万别用单片机的思维去猜摸 FPGA 如何处理按键抖动，不然问号会没完没了。好了，废话差不多说完了，让我们切回主题吧。

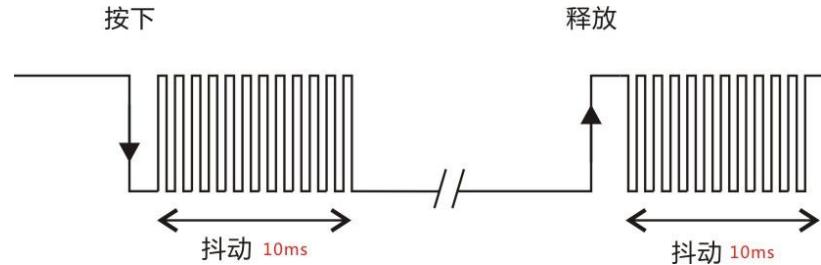


图 2.1 按键活动的时序示意图。

如图 2.1 所示，那是按键活动的时序图。高电为平按键默认状态，按键一旦按下，“按下事件”就发生了，电平随之发生抖动，抖动周期大约为 10ms。事后，如果按键依然按着不放，那么电平便会处于低电平。换之，如果按键此刻被释放，那么“释放事件”发生了，电平随之由低变高，然后发生抖动，抖动周期大约为 10ms。笔者曾在前面说过，按键消抖组一般有 3 个工作要做，亦即检测电平变化，过滤抖动，还有产生有效按键。

检测电平变化：

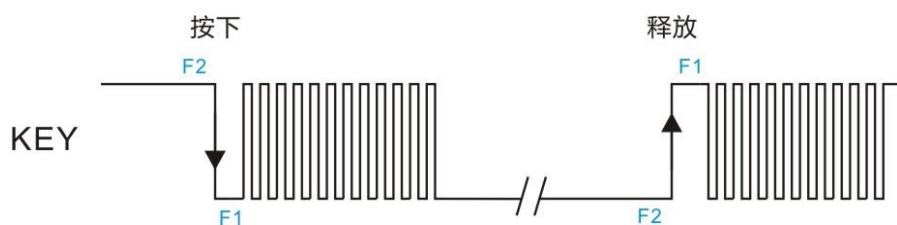


图 2.2 按键电平变化，按下事件与释放事件。

顾名思义，检测电平变化就是用来察觉“按下事件”还有“释放事件”，或者监控电平的状态变化。如图 2.2 所示，笔者建立一组寄存器 F1~F2，F1 暂存当前的电平状态，F2 则暂存上一个时钟的电平状态。Verilog 语言可以这样表示，如代码 2.1 所示：

```
reg F2, F1;
always @ ( posedge CLOCK )
  { F2,F1 } <= { F1,KEY };
```

代码 2.1 所示

根据图 2.2 的显示，按下事件是 F2 为 1 值，F1 为 0 值；释放事件则是 F2 为 0 值，F1 为 1 值。为了不要错过电平变化的黄金时间，“按下事件”还有“释放事件”必须作为“即时”，为此 Verilog 语言可以这样表示：

```

wire isH2L = ( F2 == 1 && F1 == 0 );
wire isL2H = ( F2 == 0 && F1 == 1 );

```

过滤抖动：

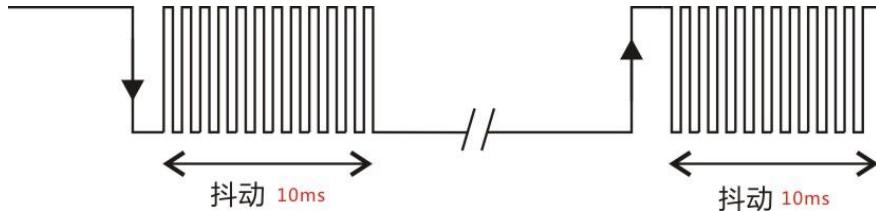


图 2.3 过滤抖动。

过滤抖动就是也可以称为延迟抖动，常规又廉价的机械按键，抖动时间大约是 10ms 之内，如图 2.3 所示。抖动一般都发生在“按下事件”或者“释放事件”以后，过滤抖动就是延迟若干时间即可。Verilog 语言则可以这样表示，如代码 2.2 所示：

```

3:
if( C1 == T10MS -1 ) begin C1 <= 19'd0; i <= i + 1'b1; end
else C1 <= C1 + 1'b1;

```

代码 2.2

产生有效按键：

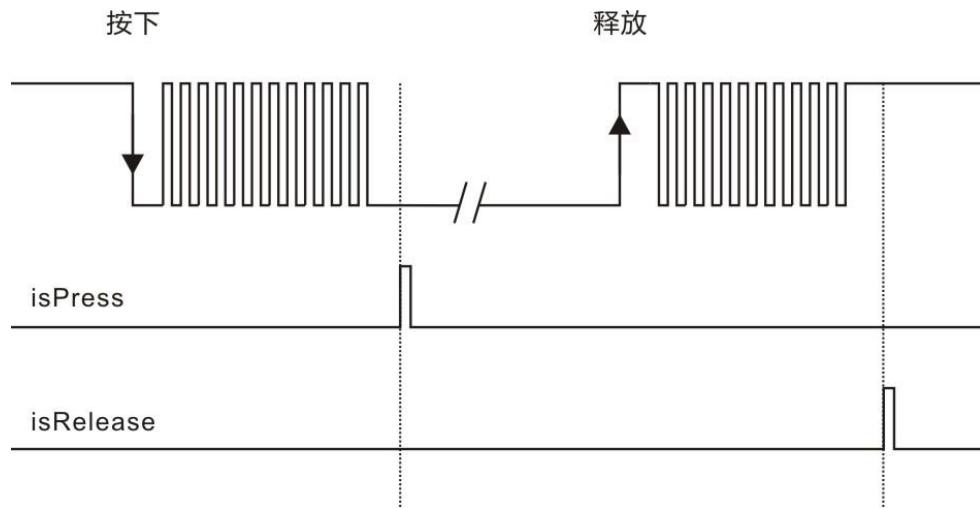


图 2.4 产生有效按键。

产生有效按键亦即立旗有效的按键事件，如图 2.4 所示，按下有效 isPress 信号，还有释放有效 isRelease 信号，两个信号分别都是拉高一个时钟。除了常见的按下有效或者释放有效以外，根据设计要求，有效按键按也有其它，例如：按键按下两下有效（双击），按键按下一段时间有效（长击）。至于 Verilog 语言则可以这样表示，如代码 2.3 所示：

```

1: begin isPress <= 1'b1; i <= i + 1'b1; end
2: begin isPress <= 1'b0; i <= i + 1'b1; end

```

```
...
1: begin isRelease <= 1'b1; i <= i + 1'b1; end
2: begin isRelease <= 1'b0; i <= i + 1'b1; end
```

代码 2.3

热身完毕后，我们就可以进入实验主题了。

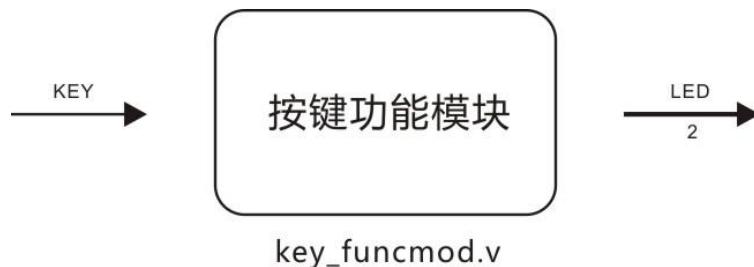


图 2.5 实验二建模图。

如图 2.5 所示，哪里有一块名为 key_funcmod 的功能模块，输入端为 KEY 信号，输出端却为 LED 信号。KEY 信号连接按键资源，LED 信号分别驱动两位 LED 资源。按键功能模块的工作主要是过滤 KEY 信号变化以后所发生的抖动，接着产生“按下有效”还有“释放有效”两个有效按键，然后点亮 LED 资源。

key_funcmod.v

```
1. module key_funcmod
2. (
3.     input CLOCK, RESET,
4.     input KEY,
5.     output [1:0]LED
6. );
```

以上内容为出入端声明。

```
7.     parameter T10MS = 19'd500_000;
8.
9.     /***** //sub
10.
11.    reg F2,F1;
12.
13.    always @ ( posedge CLOCK or negedge RESET )
14.        if( !RESET )
15.            { F2, F1 } <= 2'b11;
16.        else
17.            { F2, F1 } <= { F1, KEY };
```

```
18.  
19.      ****//core  
20.  
21.      wire isH2L = ( F2 == 1 && F1 == 0 );  
22.      wire isL2H = ( F2 == 0 && F1 == 1 );
```

以上内容为常量声明以及电平检测的周边操作。第 7 行则是 10ms 的常量声明 , 第 11~17 则是电平状态检测的周边操作。至于第 21~22 行则是 “按下事件” 还有 “释放事件” 的即时声明。

```
23.      reg [3:0]i;  
24.      reg isPress, isRelease;  
25.      reg [18:0]C1;  
26.  
27.      always @ ( posedge CLOCK or negedge RESET )  
28.          if( !RESET )  
29.              begin  
30.                  i <= 4'd0;  
31.                  { isPress,isRelease } <= 2'b00;  
32.                  C1 <= 19'd0;  
33.              end  
34.          else
```

以上内容为相关的寄存器声明以及复位操作。 i 用来指向步骤 , isPress 与 isRelease 则表示按下有效亦即释放有效 , C1 则用来计数。

```
35.          case(i)  
36.  
37.              0: // H2L check  
38.                  if( isH2L ) i <= i + 1'b1;  
39.  
40.              1: // H2L debounce  
41.                  if( C1 == T10MS -1 ) begin C1 <= 19'd0; i <= i + 1'b1; end  
42.                  else C1 <= C1 + 1'b1;  
43.  
44.              2: // Key trigger prees up  
45.                  begin isPress <= 1'b1; i <= i + 1'b1; end  
46.  
47.              3: // Key trigger prees down  
48.                  begin isPress <= 1'b0; i <= i + 1'b1; end  
49.  
50.              4: // L2H check  
51.                  if( isL2H ) i <= i + 1'b1;
```

```

52.          5: // L2H debounce
53.          if( C1 == T10MS -1 ) begin C1 <= 19'd0; i <= i + 1'b1; end
54.          else C1 <= C1 + 1'b1;
55.
56.          6: // Key trigger prees up
57.          begin isRelease <= 1'b1; i <= i + 1'b1; end
58.
59.          7: // Key trigger prees down
60.          begin isRelease <= 1'b0; i <= 4'd0; end
61.
62.      endcase

```

以上内容为核心操作。具体的核心操作过程如下：

步骤 0 等待电平由高变低；
 步骤 2 用来过滤由高变低所引发的抖动；
 步骤 1~3 用来产生按下有效的高脉冲；
 步骤 4 等待电平由低变高；
 步骤 5 用来过滤由低变高所引发的抖动；
 步骤 6~7 用来产生释放有效的高脉冲，然后返回步骤 0。

```

63.
64.      ****// sub-demo
65.
66.      reg [1:0]D1;
67.
68.      always @ ( posedge CLOCK or negedge RESET )
69.          if( !RESET )
70.              D1 <= 2'b00;
71.          else if( isPress )
72.              D1[1] <= ~D[1];
73.          else if( isRelease )
74.              D1[0] <= ~D[0];
75.
76.      assign LED = D1;
77.
78.  endmodule

```

以上内容为演示用的周边操作，它根据 isPress 还有 isRelease 的高脉冲，分别翻转 D1[1] 还有 D1[0]的内容。至于第 77 行则是输出驱动的声明，D1 驱动 LED 输出端。编译完后便下载程序。

我们会发现第一次按下 <KEY2> 会点亮 LED[1]，释放<KEY2>会点亮 LED[0]。第二

次按下 `<KEY2>` 会消灭 `LED[1]`，释放 `<KEY2>` 则会消灭 `LED[0]`。如此一来，实验二已经成功。实验二未结束之前，让笔者分析一下实验二的若干细节：

细节一：过分消抖

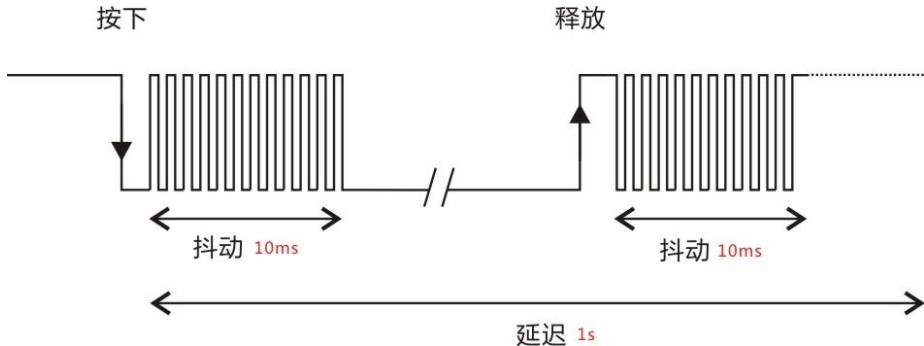


图 2.6 过分消抖 (过分延迟)。

结果如图 2.6 所示，假设笔者手痒将消抖时间拉长至 1s，Verilog 语言则可以这样表示，如代码 2.4 所示：

```
3:  
if( C1 == T1S -1 ) begin C1 <= 19'd0; i <= i + 1'b1; end  
else C1 <= C1 + 1'b1;
```

代码 2.4

一般消抖期间，按键功能模块的核心操作就会停留在消抖（延迟）步骤，如果消抖期间笔者释放按键，那么释放事件会被无视，然后核心操作会被打乱，最后整个功能模块会跑飞。反之，如果笔者等待消抖完毕再释放按键，那么释放事件会照常发生，整个功能模块也会照常运作。

细节二：精密控时

```
1: // H2L debounce  
if( C1 == T10MS -1 ) begin C1 <= 19'd0; i <= i + 1'b1; end  
else C1 <= C1 + 1'b1;  
2: // Key trigger presses up  
begin isPress <= 1'b1; i <= i + 1'b1; end  
3: // Key trigger presses down  
begin isPress <= 1'b0; i <= i + 1'b1; end
```

代码 2.5

根据按键功能模块，核心操作执行消抖之后都会产生有效按键，亦即为立旗寄存器 (`isPress` 或者 `isRelease`) 拉高又拉低一个时钟，如代码 2.5 所示。如果笔者是一位精密控时的狂人，这段拉高又拉低的时钟消耗，笔者也会将其考虑进去消抖时间。为此，

笔者可以这样修改，如代码 2.6 所示：

```
1: // H2L debounce
if( C1 == T10MS - 1 - 2 ) begin C1 <= 19'd0; i <= i + 1'b1; end
else C1 <= C1 + 1'b1;
2: // Key trigger presses up
begin isPress <= 1'b1; i <= i + 1'b1; end
3: // Key trigger presses down
begin isPress <= 1'b0; i <= i + 1'b1; end
```

代码 2.6

代码 2.6 相较代码 2.6，消抖步骤部分的 if 判断内多了 -2，其中 -2 表示产生按键有效所消耗的时钟。

细节三：完整的个体模块



图 2.7 完整的按键功能模块。

图 2.6 是演示用的建模图 然而图 2.7 则是完整的建模图 其中按键功能模块有一个 KEY 输入端，主要连接按键资源。此外，按键功能模块也有一组两位的沟通信号 Trig，亦即按下 Trig[1]产生一个高脉冲，释放 Trig[0]产生一个高脉冲。

key_funcmod.v

```
1. module key_funcmod
2. (
3.     input CLOCK, RESET,
4.     input KEY,
5.     output [1:0]oTrig
6. );
7. parameter T10MS = 19'd500_000;
8.
9.     /***** //sub
10.
11.    reg F2,F1;
12.
13.    always @ ( posedge CLOCK or negedge RESET )
```

```

14.         if( !RESET )
15.             { F2, F1 } <= 2'b11;
16.         else
17.             { F2, F1 } <= { F1, KEY };
18.
19.         //*****
20.
21.         wire isH2L = ( F2 == 1 && F1 == 0 );
22.         wire isL2H = ( F2 == 0 && F1 == 1 );
23.         reg [3:0]i;
24.         reg isPress, isRelease;
25.         reg [18:0]C1;
26.
27.         always @ ( posedge CLOCK or negedge RESET )
28.             if( !RESET )
29.                 begin
30.                     i <= 4'd0;
31.                     { isPress,isRelease } <= 2'b00;
32.                     C1 <= 19'd0;
33.                 end
34.             else
35.                 case(i)
36.
37.                     0:// H2L check
38.                     if( isH2L ) i <= i + 1'b1;
39.
40.                     1:// H2L debounce
41.                     if( C1 == T10MS -1 ) begin C1 <= 19'd0; i <= i + 1'b1; end
42.                     else C1 <= C1 + 1'b1;
43.
44.                     2:// Key trigger presses up
45.                     begin isPress <= 1'b1; i <= i + 1'b1; end
46.
47.                     3:// Key trigger releases down
48.                     begin isPress <= 1'b0; i <= i + 1'b1;     end
49.
50.                     4:// L2H check
51.                     if( isL2H ) i <= i + 1'b1;
52.
53.                     5:// L2H debounce
54.                     if( C1 == T10MS -1 ) begin C1 <= 19'd0; i <= i + 1'b1; end
55.                     else C1 <= C1 + 1'b1;
56.

```

```
57.          6: // Key trigger prees up
58.          begin isRelease <= 1'b1; i <= i + 1'b1; end
59.
60.          7: // Key trigger prees down
61.          begin isRelease <= 1'b0; i <= 4'd0; end
62.
63.      endcase
64.
65.      /***** */
66.
67.      assign oTrig = { isPress, isRelease };
68.
69. endmodule
```

最后别向笔者要仿真了 ,因为按键消抖没有仿真的意义 ,单是代码已经足够脑补时序了。

实验三：按键模块② — 点击与长点击

实验二我们学过按键功能模块的基础内容，其中我们知道按键功能模块有如下操作：

- 电平变化检测；
- 过滤抖动；
- 产生有效按键。

实验三我们也会执行同样的事情，不过却是产生不一样的有效按键：

- 按下有效（点击）；
- 长按下有效（长点击）。

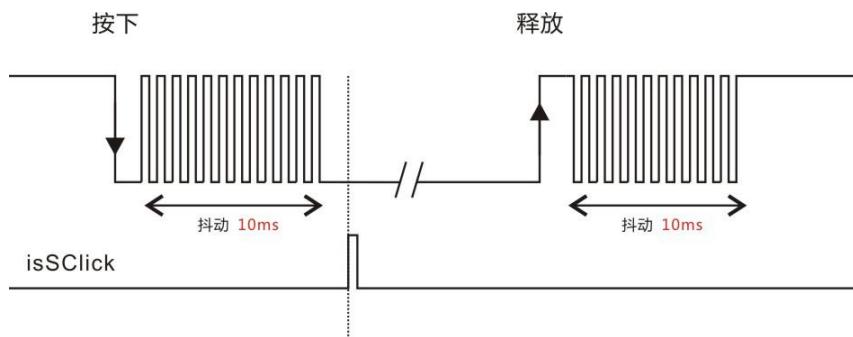


图 3.1 按下有效，时序示意图。

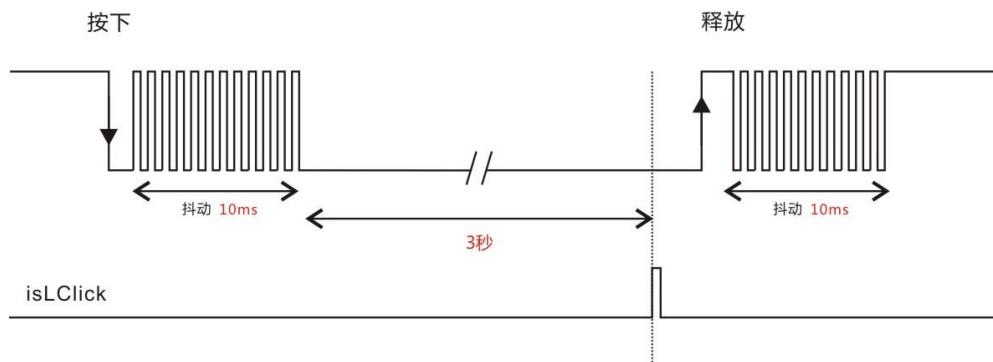


图 3.2 长按下有效，时序示意图。

如图 3.1 所示，按下有效既是“点击”，当按键被按下并且消抖完毕以后，`isSClick` 信号就有被拉高一个时钟（Short Click）。换之，长按下有效也是俗称为的“长点击”，如图 3.2 所示，当按键被按下并且消抖完毕以后，如果按键 3 秒之内都没有被释放，那么 `isLClick` 信号就会拉高一个时钟（Long Click）。



图 3.3 实验三的建模图。

如图 3.3 所示，那是实验三的建模图，同样按键功能模块有一位 KEY 输入，并且连接至按键资源，然后它有两位 LED 输出，并且连接至 2 位 LED 资源。至于多按键功能模块的具体内容，让我们来看代码吧：

key_funcmod.v

```

1. module key_funcmod
2. (
3.     input CLOCK, RESET,
4.     input KEY,
5.     output [1:0]LED
6. );

```

以上内容为相关的出入端声明。

```

7.     parameter T10MS = 26'd500_000; // Debouncing time
8.     parameter T3S = 28'd150_000_000; // Long press time
9.
10.    /***** //sub
11.
12.    reg F2,F1;
13.
14.    always @ ( posedge CLOCK or negedge RESET )
15.        if( !RESET )
16.            { F2, F1 } <= 2'b11;
17.        else
18.            { F2, F1 } <= { F1, KEY };
19.
20.    /***** //core
21.
22.    wire isH2L = ( F2 == 1 && F1 == 0 );
23.    wire isL2H = ( F2 == 0 && F1 == 1 );

```

以上内容为相关的常量声明，周边操作以及即时声明。第 12~18 行是电平状态检测的周

边操作。第 22~23 行是按下事件与释放事件的即时声明。

```
24.      reg [3:0]i;
25.      reg isLClick,isSClick;
26.      reg [1:0]isTag;
27.      reg [27:0]C1;
28.
29.      always @ ( posedge CLOCK or negedge RESET )
30.          if( !RESET )
31.              begin
32.                  i <= 4'd0;
33.                  isLClick <= 1'd0;
34.                  isSClick <= 1'b0;
35.                  isTag <= 2'd0;
36.                  C1 <= 28'd0;
37.              end
38.          else
```

以上内容为相关的寄存器声明以及复位操作。*i* 用作指向步骤，*isLClick* 与 *isSClick* 同是标示寄存器，分别是长按下有效与按下有效。*isTag* 用来判定那种有效按键，*C1* 用来计数。

```
39.          case(i)
40.
41.              0:// Wait H2L
42.              if( isH2L ) i <= i + 1'b1;
43.
44.              1:// H2L debouce
45.              if( C1 == T10MS -1 ) begin C1 <= 28'd0; i <= i + 1'b1; end
46.              else C1 <= C1 + 1'b1;
47.
48.              2:// Key S Check
49.              if( isL2H ) begin isTag <= 2'd1; C1 <= 28'd0; i <= i + 1'b1; end
50.              else if( {F2,F1} == 2'b00 && C1 >= T3S -1 ) begin isTag <= 2'd2; C1 <= 28'd0; i <= i + 1'd1; end
51.              else C1 <= C1 + 1'b1;
52.
53.              3:// S Trigger (pree up)
54.              if( isTag == 2'd1 ) begin isSClick <= 1'b1; i <= i + 1'b1; end
55.              else if( isTag == 2'd2 ) begin isLClick <= 1'b1; i <= i + 1'b1; end
56.
57.              4:// S Trigger (pree down)
58.              begin { isLClick,isSClick } <= 2'b00; i <= i + 1'b1; end
59.
```

```

60.      5: // L2H debounce check
61.      if( isTag == 2'd1 ) begin isTag <= 2'd0; i <= i + 2'd2; end
62.      else if( isTag == 2'd2 ) begin isTag <= 2'd0; i <= i + 1'b1; end
63.
64.      6: // Wait L2H
65.      if( isL2H ) i <= i + 1'b1;
66.
67.      7: // L2H debounce
68.      if( C1 == T10MS -1 ) begin C1 <= 28'd0; i <= 4'd0; end
69.      else C1 <= C1 + 1'b1;
70.
71.      endcase

```

以上内容为核心操作，至于核心的操作过程如下：

步骤 0，等待按下事件；

步骤 1，过滤又高变低所产生的抖动；

步骤 2，检测那种有效按键，如果 3 秒以内发生释放事件就为 isTag 赋值 1；反之，如果持续 3 秒低电平则为 isTag 赋值 2；

步骤 3~4，根据 Mode 的内容产生不同有效按键的高脉冲，isTag 为 1 是“点击” isSClick，isTag 为 2 则是“长点击” isLClick。

步骤 5，用来检测释放事件，如果之前发生“点击”就直接跳向步骤 7。反之，如果之前发生“长点击”就进入步骤 6。

步骤 6，等待释放事件（长点击有效）。

步骤 7，过滤又低变高所产生的抖动，然后返回步骤 0。

```

72.
73.      ****// sub demo
74.
75.      reg [1:0]D1;
76.
77.      always @ ( posedge CLOCK or negedge RESET )
78.          if( !RESET )
79.              D1 <= 2'b00;
80.          else if( isLClick )
81.              D1[1] <= ~D1[1];
82.          else if( isSClick )

```

```

83.          D1[0] <= ~D1[0];
84.
85.      ****
86.
87.      assign LED = D1;
88.
89. endmodule

```

以上内容为演示用的周边操作，它根据那种有效按键就翻转那位 D1 寄存器。第 87 行则是输出驱动声明。编译完成便下载程序。

我们会发现，第一次按下 `<KEY2>` 3 秒不放会点亮 `LED[1]`，换之按下 `<KEY2>` 不到 3 秒便释放则会点亮 `LED[0]`。第二次按下 `<KEY2>` 3 秒不放会消灭 `LED[0]`，按下 `<KEY2>` 不到 3 秒便释放会消灭 `LED[0]`。如此一来，实验三已经成

细节一： 精密控时

```

2:
if( isL2H ) begin S <= 2'd1; C1 <= 28'd0; i <= i + 1'b1; end
else if( {F2,F1} == 2'b00 && C1 >= T3S -1 ) begin S <= 2'd2; C1 <= 28'd0; i <= i + 1'd1; end
else C1 <= C1 + 1'b1;

3:
if( S == 2'd1 ) begin isSClick <= 1'b1; i <= i + 1'b1; end
else if( S == 2'd2 ) begin isLCClick <= 1'b1; i <= i + 1'b1; end

4:
begin { isLCClick,isSClick } <= 2'b00; i <= i + 1'b1; end

5:
if( S == 2'd1 ) begin S <= 2'd0; i <= i + 2'd2; end
else if( S== 2'd2 ) begin S <= 2'd0; i <= i + 1'b1; end

6:
if( isL2H )i <= i + 1'b1;

7:
if( C1 == T10MS -1 ) begin C1 <= 28'd0; i <= 4'd0; end
else C1 <= C1 + 1'b1;

```

代码 3.1

代码 3.1 是 `key_funcmod` 的部分代码，分别是步骤 2~7。如果笔者是精密控时狂人，事实上代码 3.1 还可以进一步细化，然而还有什么可以细化的地方呢？如代码 3.1 所示，距离步骤 7（消抖）之前，共有步骤 2~6 等 5 个时钟所消耗。如果也考虑消抖时间之内，然而步骤 7 修改为 `if(C1 == T10MS -1 -5)` 是行不通，因为两种有效按键都有不通的情况。

如果 `isTag` 为 1，步骤 5 会直接跳向步骤 7，步骤 7 的消抖只要加入步骤 2~5 所消耗的

4个时钟。如果 isTag 为 2，那么步骤 5 会前进步骤 6，然后乖乖等待释放事件，期间没消耗而外的时钟。为此，步骤 7 可以这样修改，结果如代码 3.2 所示：

```
5:  
if( S == 2'd1 ) begin i <= i + 2'd2; end  
else if( S == 2'd2 ) begin i <= i + 1'b1; end  
6:  
if( isL2H )i <= i + 1'b1;  
7:  
if( Mode == 2'd1 && C1 == T10MS -1 -4) begin Mode <= 2'd0; C1 <= 28'd0; i <= 4'd0; end  
else if( Mode == 2'd2 && C1 == T10MS -1 ) begin Mode <= 2'd0; C1 <= 28'd0; i <= 4'd0; end  
else C1 <= C1 + 1'b1;
```

代码 3.2

如代码 3.2 所示，步骤 5 被拿掉 isTag <= 2'd0 操作，然后步骤 7 稍微修改一下消抖过程。如果 isTag 为 1，那么消抖多考虑 4 个而外的时钟消耗。反之，如果 isTag 为 2，那么消抖过程照常。

细节二：完整的按键功能模块



key_funcmod.v

图 3.4 完整的按键功能模块。

如图 3.4 所示，那是完整的按键功能模块，它有一位链接至按键资源的 KEYn 信号，它也有一组两位的沟通信号 Trig。Trig[1]产生“点击”的个高脉冲，Trig[0]产生“长点击”的个高脉冲。

key_funcmod.v

```
1. module key_funcmod  
2. (  
3.     input CLOCK, RESET,  
4.     input KEY,  
5.     output [1:0]oTrig  
6. );  
7.     parameter T10MS = 26'd500_000; // Debouncing time
```

```

8.      parameter T3S = 28'd150_000_000; // Long press time
9.
10.     /***** //sub
11.
12.     reg F2,F1;
13.
14.     always @ ( posedge CLOCK or negedge RESET )
15.         if( !RESET )
16.             { F2, F1 } <= 2'b11;
17.         else
18.             { F2, F1 } <= { F1, KEY };
19.
20.     /***** //core
21.
22.     wire isH2L = ( F2 == 1 && F1 == 0 );
23.     wire isL2H = ( F2 == 0 && F1 == 1 );
24.     reg [3:0]i;
25.     reg isLClick,isSClick;
26.     reg [1:0]isTag;
27.     reg [27:0]C1;
28.
29.     always @ ( posedge CLOCK or negedge RESET )
30.         if( !RESET )
31.             begin
32.                 i <= 4'd0;
33.                 isLClick <= 1'd0;
34.                 isSClick <= 1'b0;
35.                 isTag <= 2'd0;
36.                 C1 <= 28'd0;
37.             end
38.         else
39.             case(i)
40.
41.                 0: // Wait H2L
42.                 if( isH2L ) i <= i + 1'b1;
43.
44.                 1: // H2L debouce
45.                 if( C1 == T10MS -1 ) begin C1 <= 28'd0; i <= i + 1'b1; end
46.                 else C1 <= C1 + 1'b1;
47.
48.                 2: // Key Tag Check
49.                 if( isL2H ) begin isTag <= 2'd1; C1 <= 28'd0; i <= i + 1'b1; end
50.                 else if( {F2,F1} == 2'b00 && C1 >= T3S -1 ) begin isTag <= 2'd2; C1 <= 28'd0; i <= i + 1'd1; end

```

```

51.           else C1 <= C1 + 1'b1;
52.
53.           3: // Tag Trigger (pree up)
54.           if( isTag == 2'd1 ) begin isSClick <= 1'b1; i <= i + 1'b1; end
55.           else if( isTag == 2'd2 ) begin isLClick <= 1'b1; i <= i + 1'b1; end
56.
57.           4: // Tag Trigger (pree down)
58.           begin { isLClick,isSClick } <= 2'b00; i <= i + 1'b1; end
59.
60.           5: // L2H deboce check
61.           if( isTag == 2'd1 ) begin S <= 2'd0; i <= i + 2'd2; end
62.           else if( isTag == 2'd2 ) begin S <= 2'd0; i <= i + 1'b1; end
63.
64.           6: // Wait L2H
65.           if( isL2H )i <= i + 1'b1;
66.
67.           7: // L2H debounce
68.           if( C1 == T10MS -1 ) begin C1 <= 28'd0; i <= 4'd0; end
69.           else C1 <= C1 + 1'b1;
70.
71.       endcase
72.
73.   *****/
74.
75.   assign oTrig = { isSClick,isLClick };
76.
77. endmodule

```

实验四：按键模块③ — 单击与双击

实验三我们创建了“点击”还有“长点击”等有效按键的多功能按键模块。在此，实验四同样也是创建多功能按键模块，不过却有不同的有效按键。实验四的按键功能模块有以下两项有效按键：

- 单击（按下有效）；
- 双击（连续按下两下有效）。

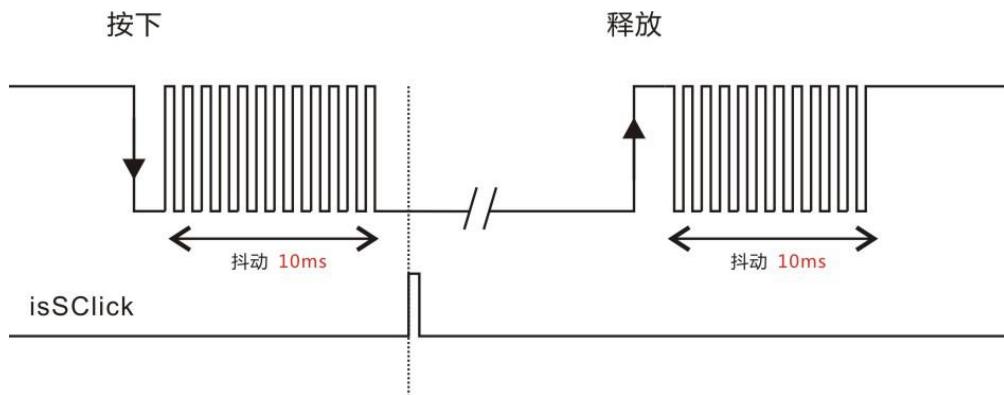
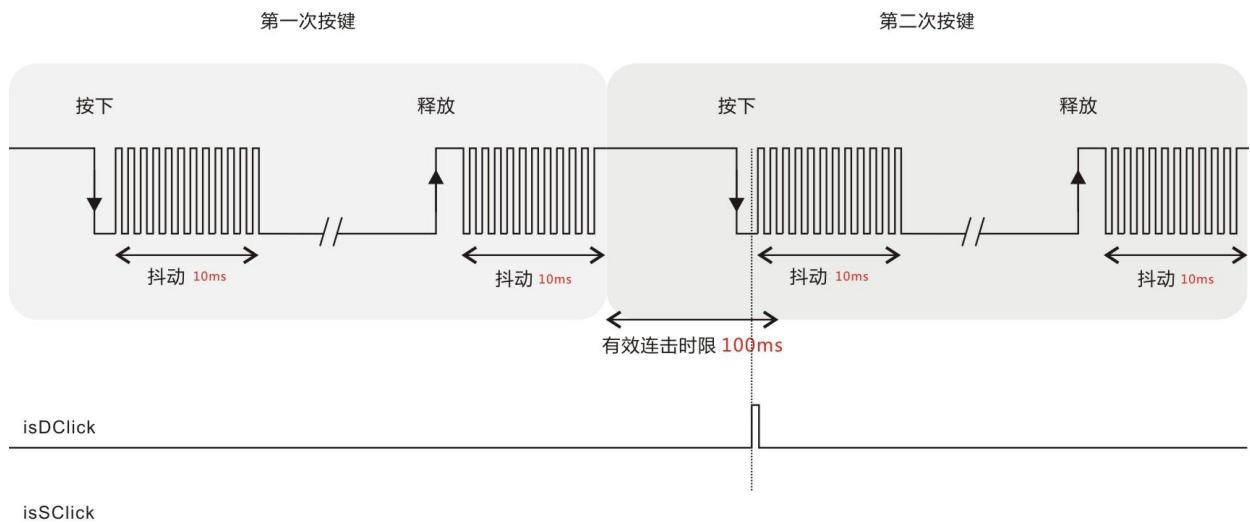


图 4.1 单击有效按键，时序示意图。

实验四的“单击”基本上与实验三的“点击”一模一样，既按键被按下，经过消抖以后 isSClick 信号被拉高一个时钟，结果如图 4.1 所示，过程非常单调。反之，“双击”实现起来，会比较麻烦一些，因为我们还要考虑而外的细节，即人为连打极限。所谓人为连打极限就是两次按下按键之间的最短间隔。

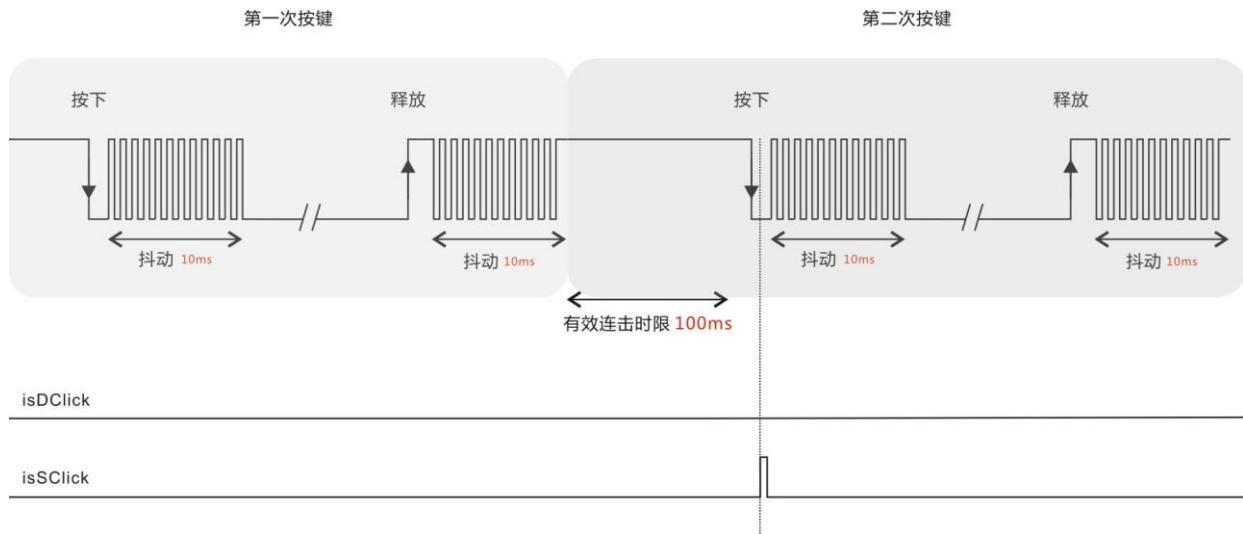


如图 4.2 双击有效按键，时序示意图（双击成功）。

根据笔者的理解，常人的连打极限是 60ms 左右，笔者是 80ms 左右，超人是 20ms 左右。为了兼容常人还有笔者的连打极限，我们必须设置有效的连击时限，为此 100ms 是最好

的选择。如图 4.2 所示，假设那是按键过程，笔者先是缓缓按下然后又缓缓释放按键完成第一次按键行为，结果有如往常般，按下事件发生，抖动发生，释放事件发生，抖动发生，但是 isDClick (Double Click) 信号还有 isSClick (Single Click) 信号都没有产生高脉冲。

第一次按键完成以后就会引来第二次按键的黄金时间，亦即有效连击时限，在此笔者设为 100ms。假设笔者在这 100ms 的黄金时间内按下按键，那么 isDClick 信号会立即产生高脉冲。余下有如往常那样，抖动发生，释放事件发生，抖动发生 … 对此，isSClick 由始至终都没有状况发生。



如图 4.3 双击有效按键，时序示意图（双击失败）。

假设笔者没在有限的 100ms 黄金时间内执行第二次按键按下的动作，那么“双击”就会失败，结果如图 4.3 所示。第一次按键过程与图 4.2 一样，反之第二次按键却不同了，如图 4.3 所示，第二次按键的按下事件是发生在 100ms 以后，为此 isSClick 产生高脉冲，然而 isDClick 信号却没有动静。

明白上述这些简单的原理以后，我们就可以开始建模了。



图 4.4 实验四的建模图。

如图 4.4 所示，那是实验四的建模图，它有一位 KEY 输入端，连接至按键资源。此外，它也有两位 LED 输出端，分别连接两位 LED 资源。

key_funcmod.v

```
1. module key_funcmod
2. (
3.     input CLOCK, RESET,
4.     input KEY,
5.     output [1:0]LED
6. );
```

以上内容为出入端的相关声明。

```
7.     parameter T10MS      = 28'd500_000;
8.     parameter T100MS     = 28'd5_000_000;
9.     parameter T200MS     = 28'd10_000_000;
10.    parameter T300MS     = 28'd15_000_000;
11.    parameter T400MS     = 28'd20_000_000;
12.    parameter T500MS     = 28'd25_000_000;
13.
14.    /***** //sub
15.
16.    reg F2,F1;
17.
18.    always @ ( posedge CLOCK or negedge RESET )
19.        if( !RESET )
20.            { F2, F1 } <= 2'b11;
21.        else
22.            { F2, F1 } <= { F1, KEY };
23.
24.    /***** //core
25.
26.    wire isH2L = ( F2 == 1 && F1 == 0 );
27.    wire isL2H = ( F2 == 0 && F1 == 1 );
```

以上内容为相关常量声明，周边操作以及即时声明。第 7~12 行是各种时间的常量声明，除了 10 毫秒声明消抖时间以外，第 8~12 行的时间常量是用来自定义双击的敏感度。第 18~23 行是检测电平状态的周边操作，第 26~27 行则是按下事件还有释放事件。

```
28.    reg [3:0]i;
29.    reg isDClick,isSClick;
30.    reg [1:0]isTag;
31.    reg [27:0]C1;
32.
33.    always @ ( posedge CLOCK or negedge RESET )
```

```

34.         if( !RESET )
35.             begin
36.                 i <= 4'd0;
37.                 isDClick <= 1'd0;
38.                 isSClick <= 1'b0;
39.                 isTag <= 2'd0;
40.                 C1 <= 28'd0;
41.             end
42.         else

```

以上内容为相关寄存器声明以及复位操作。i 用作指向步骤，isDClick 还有 isSClick 是用作标示“双击”还有“单击”。isTag 是有效按键的标签，C1 则用来计数。至于第 33~41 行则是核心操作的复位活动。

```

43.         case(i)
44.
45.             0: // Wait H2L
46.                 if( isH2L ) begin i <= i + 1'b1; end
47.
48.             1: // H2L debounce
49.                 if( C1 == T10MS -1 ) begin C1 <= 28'd0; i <= i + 1'b1; end
50.                 else C1 <= C1 + 1'b1;
51.
52.             2: // Wait L2H
53.                 if( isL2H ) i <= i + 1'b1;
54.
55.             3: // L2H debounce
56.                 if( C1 == T10MS -1 ) begin C1 <= 28'd0; i <= i + 1'b1; end
57.                 else C1 <= C1 + 1'b1;
58.
59.             4: // Key Tag Check
60.                 if( isH2L && C1 <= T100MS -1 ) begin isTag <= 2'd2; C1 <= 28'd0; i <= i + 1'b1; end
61.                 else if( C1 >= T100MS -1) begin isTag <= 2'd1; C1 <= 28'd0; i <= i + 1'b1; end
62.                 else C1 <= C1 + 1'b1;
63.
64.             5: // Key trigger press up
65.                 if( isTag == 2'd2 ) begin isDClick <= 1'b1; i <= i + 1'b1; end
66.                 else if( isTag == 2'd1 ) begin isSClick <= 1'b1; i <= i + 1'b1; end
67.
68.             6: // Key trigger pree down
69.                 begin { isSClick , isDClick } <= 2'b00; i <= i + 1'b1; end
70.
71.             7: // L2H deounce check

```

```
72.           if( isTag == 2'd1 ) begin isTag <= 2'd0; i <= 4'd0; end
73.           else if( isTag == 2'd2 ) begin isTag <= 2'd0; i <= i + 1'b1; end
74.
75.           8: // Wait L2H
76.           if( isL2H ) begin i <= i + 1'b1; end
77.
78.           9: // L2H debounce
79.           if( C1 == T10MS - 1 ) begin C1 <= 28'd0; i <= 4'd0; end
80.           else C1 <= C1 + 1'b1;
81.
82.       endcase
```

以上内容为核心操作。具体过程如下所示：

步骤 0，等待第一次按下事件；

步骤 1，过滤抖动；

步骤 2，等待第一次释放事件；

步骤 3，过滤抖动；

步骤 4，如果 100ms 发生第二回按下事件，isTag 设置为 2，反之 isTag 设置为 1；

步骤 5~6，根据 isTag 的内容拉高又拉低 isDClick 或者 isSClick；

步骤 7，根据 S 内容，S 为 1 便清除 S 然后返回步骤 0。反之，isTag 为 2 就清除 isTag 然后步骤继续前进；

步骤 8，等待第二次释放事件；

步骤 9：，过滤抖动然后返回步骤 0。

```
83.
84.      ****// sub-demo
85.
86.      reg [1:0]D1;
87.
88.      always @ ( posedge CLOCK or negedge RESET )
89.          if( !RESET )
90.              D1 <= 2'd0;
91.          else if( isDClick )
92.              D1[1] <= ~D1[1];
```

```

93.         else if( isSClick )
94.             D1[0] <= ~D1[0];
95.
96.             *****/
97.
98.         assign LED = D1;
99.
100. endmodule

```

以上内容为演示用的周边操作以及输出驱动声明。它会根据 isDClick 还有 isSClick 的高脉冲翻转 D1[1] 还有 D1[0] 的内容。第 98 行则是输出驱动声明。编译完后便下载程序。

当我们双击 <KEY2> 建 LED[1] 就会发亮，然后再双击 <KEY2> 建 LED[1] 则会消灭，发生双击的前提条件是 ... 第一次按下时间的 100ms 之内必须发生第二次按下时间才能成立。换之，如果我们单击 <KEY2> 建 LED[0] 便会发亮，再单击 <KEY2> 建 LED[0] 则会消灭。

细节一： 双击的敏感度

```

parameter T100MS  = 28'd5_000_000;
parameter T200MS  = 28'd10_000_000;
parameter T300MS  = 28'd15_000_000;
parameter T400MS  = 28'd20_000_000;
parameter T500MS  = 28'd25_000_000;

```

代码 4.1

代码 4.1 是 key_funcmod 的部分内容，亦即 100ms~500ms 的时间声明。所谓双击的敏感度就是按键第二次按下所有有效的时限。事实上，有效时间 100ms 已经足够应付一般“双击”要求，然而“双击”的敏感度除了人为连打极限这个因数以外，还有按键资源本身。开发板常见的按键都是经济型机械按键，手感较为迟钝，所以有效时间推荐在 100ms~500ms 范围之内。

如果读者所使用的按键资源是精致的家伙，想必手感一定很爽，例如鼠标之类的按键，100ms 有效时间可能会影响双击的敏感度。为此，有效时间必须设置在 40ms~100ms 的范围，常见的有效时间是 50ms。最后不管怎么样，手感还有敏感度这种东西非常暧昧，完全因人而异 ... 也有传言说那些骨灰级的游戏鼠标是可以自定义敏感度，事实究竟如何？对于笔者这种游戏冷漠者则是永远的迷。

细节二：完成的个体模块



图 4.5 完整的按键功能模块。

如图 4.5 所示，那是完整的按键功能模块，输入端一边的 KEY 连接至按键资源，Trig[1] 产生“单击”的个高脉冲，Trig[0]产生“双击”的个高脉冲。

key_funcmod.v

```

1. module key_funcmod
2. (
3.     input CLOCK, RESET,
4.     input KEY,
5.     output [1:0]oTrig
6. );
7. parameter T10MS      = 28'd500_000;
8. parameter T100MS     = 28'd5_000_000;
9. parameter T200MS     = 28'd10_000_000;
10. parameter T300MS    = 28'd15_000_000;
11. parameter T400MS    = 28'd20_000_000;
12. parameter T500MS    = 28'd25_000_000;
13.
14. //*****
15.
16. reg F2,F1;
17.
18. always @ ( posedge CLOCK or negedge RESET )
19.     if( !RESET )
20.         { F2, F1 } <= 2'b11;
21.     else
22.         { F2, F1 } <= { F1, KEY };
23.
24. //*****
25.
26. wire isH2L = ( F2 == 1 && F1 == 0 );
27. wire isL2H = ( F2 == 0 && F1 == 1 );
28. reg [3:0]i;
29. reg isDClick,isSClick;
```

```

30.      reg [1:0]isTag;
31.      reg [27:0]C1;
32.
33.      always @ ( posedge CLOCK or negedge RESET )
34.          if( !RESET )
35.              begin
36.                  i <= 4'd0;
37.                  isDClick <= 1'd0;
38.                  isSClick <= 1'b0;
39.                  isTag <= 2'd0;
40.                  C1 <= 28'd0;
41.              end
42.          else
43.              case(i)
44.
45.                  0: // Wait H2L
46.                  if( isH2L ) begin i <= i + 1'b1; end
47.
48.                  1: // H2L debounce
49.                  if( C1 == T10MS -1 ) begin C1 <= 28'd0; i <= i + 1'b1; end
50.                  else C1 <= C1 + 1'b1;
51.
52.                  2: // Wait L2H
53.                  if( isL2H ) i <= i + 1'b1;
54.
55.                  3: // L2H debounce
56.                  if( C1 == T10MS -1 ) begin C1 <= 28'd0; i <= i + 1'b1; end
57.                  else C1 <= C1 + 1'b1;
58.
59.                  4: // Key Tag Check
60.                  if( isH2L&& C1 <= T100MS -1 ) begin isTag <= 2'd2; C1 <= 28'd0; i <= i + 1'b1; end
61.                  else if( C1 >= T100MS -1) begin isTag <= 2'd1; C1 <= 28'd0; i <= i + 1'b1; end
62.                  else C1 <= C1 + 1'b1;
63.
64.                  5: // Key trigger press up
65.                  if( isTag == 2'd2 ) begin isDClick <= 1'b1; i <= i + 1'b1; end
66.                  else if( isTag == 2'd1 ) begin isSClick <= 1'b1; i <= i + 1'b1; end
67.
68.                  6: // Key trigger pree down
69.                  begin { isSClick , isDClick } <= 2'b00; i <= i + 1'b1; end
70.
71.                  7: // L2H deounce check
72.                  if( isTag == 2'd1 ) begin isTag <= 2'd0; i <= 4'd0; end

```

```
73.           else if( isTag == 2'd2 ) begin isTag <= 2'd0; i <= i + 1'b1; end
74.
75.           8: // Wait L2H
76.           if( isL2H ) begin i <= i + 1'b1; end
77.
78.           9: // L2H debounce
79.           if( C1 == T10MS -1 ) begin C1 <= 28'd0; i <= 4'd0; end
80.           else C1 <= C1 + 1'b1;
81.
82.       endcase
83.
84.   /***** */
85.
86.   assign oTrig = { isSClick,isDClick };
87.
88. endmodule
```

实验五：按键模块④ — 点击，长点击，双击

实验二至实验四，我们一共完成如下有效按键：

- 点击（按下有效）
- 点击（释放有效）
- 长击（长按下有效）
- 双击（连续按下有效）

然而，不管哪个实验都是只有两项“功能”的按键模块而已，如今我们要创建三项“功能”的按键模块，亦即点击（按下有效），长击，还有双击。实验继续之前，让我们先来复习一下各种有效按键。

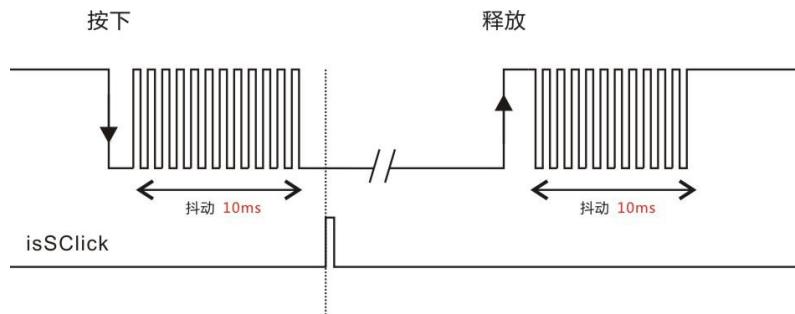


图 5.1 点击（按下有效）。

如图 5.1 所示，所谓点击（按下有效）就是按键按下以后，isSClick 信号 (Single Click) 产生一个高脉冲。

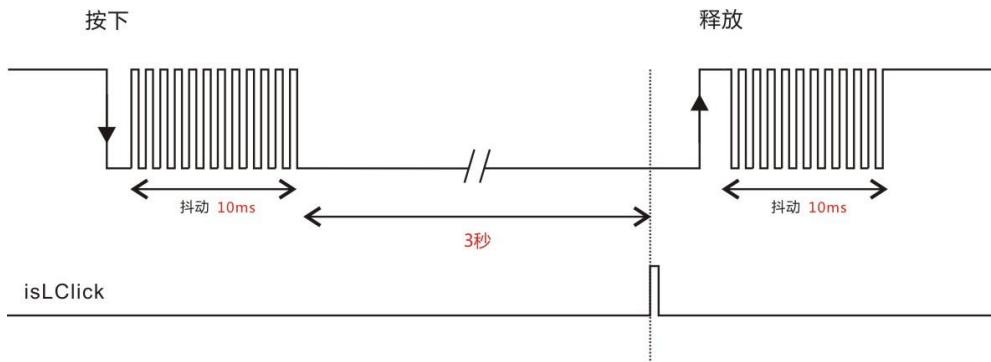


图 5.2 长点击。

如图 5.2 所示，所谓长点击就是按键按下以后，长达 3 秒不放，isLClick 信号(Long Click) 就会产生一个高脉冲。

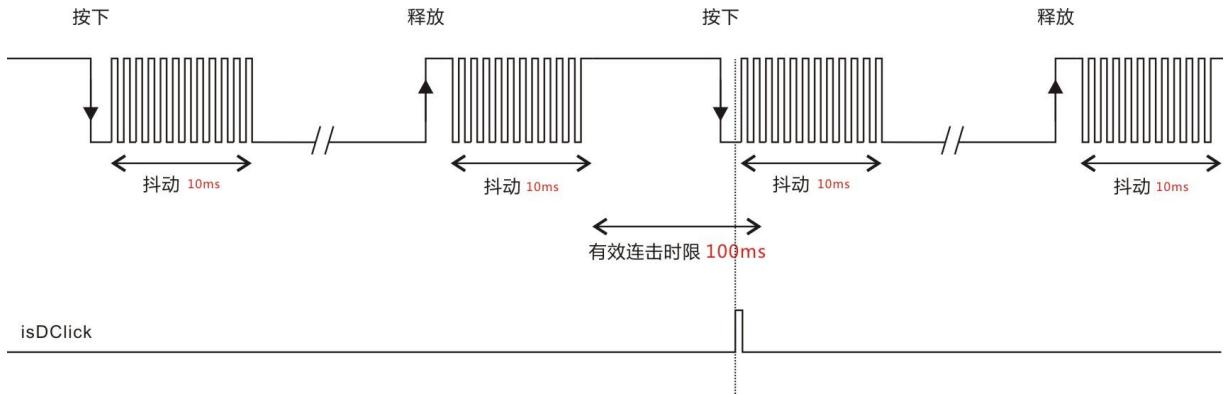


图 5.3 双点击。

如图 5.3 所示，所谓双点击就是距离第一次“点击（按下有效）”，如果在有效的连击时限内容完成第二次“点击（按下有效）”，那么 isDCClick 信号（Double Click）就会产生一个高脉冲。

话语上，实验五虽然要整合 3 项按键功能，然而实验五相较实验二至实验四却有一定程度的难度。那是因为仅有两项功能的按键模块，仅需判断一次性的有效按键而已，反之三项功能的按键模块却要分开二次判断，而且判断也要有序。根据实验五，核心操作会优先判断按键是否“长点击”，然后再来判断“点击”还是“双点击”。期间绝对不能搞错判断的次序。

具体内容，我们还是直接来看代码吧：



图 5.4 实验五的建模图。

如图 5.4 所示，那是实验五的按键功能模块，输入端一边是连接至按键资源的 KEY 信号，输出端另一边则是连接至 3 位 LED 资源的 LED 信号。

key_funcmod.v

```

1. module key_funcmod
2. (
3.     input CLOCK, RESET,
4.     input KEY,
5.     output [2:0]LED

```

```
6. );
```

以上为相关的出入端声明。

```
7.     parameter T10MS      = 28'd500_000;
8.     parameter T100MS     = 28'd5_000_000;
9.     parameter T200MS     = 28'd10_000_000;
10.    parameter T300MS     = 28'd15_000_000;
11.    parameter T400MS     = 28'd20_000_000;
12.    parameter T500MS     = 28'd25_000_000;
13.    parameter T3S       = 28'd150_000_000;
14.
15.   /**************************************************************************//sub
16.
17.   reg F2,F1;
18.
19.   always @ ( posedge CLOCK or negedge RESET )
20.     if( !RESET )
21.       { F2, F1 } <= 2'b11;
22.     else
23.       { F2, F1 } <= { F1, KEY };
24.
25.   /**************************************************************************//core
26.
27.   wire isH2L=( F2 == 1 && F1 == 0 );
28.   wire isL2H=( F2 == 0 && F1 == 1 );
```

以上内容是时间常量声明以及周边操作。第 17~23 行是检测电平状态的周边操作，第 27~28 行则是按下事件还有释放事件的声明。

```
29.   reg [3:0]i;
30.   reg isLClick, isDClick,isSClick;
31.   reg [1:0]isTag;
32.   reg [27:0]C1;
33.
34.   always @ ( posedge CLOCK or negedge RESET )
35.     if( !RESET )
36.       begin
37.         i <= 4'd0;
38.         isLClick <= 1'b0;
39.         isDClick <= 1'b0;
40.         isSClick <= 1'b0;
41.         isTag <= 2'd0;
```

```
42.           C1 <= 28'd0;
43.       end
44.   else
```

以上内容为相关的寄存器声明以及复位操作。i 指向步骤，寄存器 isLClick，寄存器 isDClick，还有寄存器 isSClick 则是相关的“有效按键”标志。isTag 表示“有效按键”的标签，1 为“点击”，2 为“双点击”，3 为“长点击”。C1 用来计数。第 35~43 行则是相关的复位操作。

```
45.   case(i)
46.
47.       0: // Wait H2L
48.       if( isH2L ) begin i <= i + 1'b1; end
49.
50.       1: // H2L debounce
51.       if( C1 == T10MS -1 ) begin C1 <= 28'd0; i <= i + 1'b1; end
52.       else C1 <= C1 + 1'b1;
53.
54.       2: // Key Tag Check 1
55.       if( isL2H ) begin C1 <= 28'd0; i <= i + 1'b1; end
56.       else if( {F2,F1} == 2'b00 && C1 >= T3S -1 ) begin isTag <= 2'd3; C1 <= 28'd0; i <= 4'd5; end
57.       else C1 <= C1 + 1'b1;
58.
59.       3: // L2H debounce
60.       if( C1 == T10MS -1 ) begin C1 <= 28'd0; i <= i + 1'b1; end
61.       else C1 <= C1 + 1'b1;
62.
63.       4: // Key Tag Check 2
64.       if( isH2L && C1 <= T100MS -1 ) begin isTag <= 2'd2; C1 <= 28'd0; i <= i + 1'b1; end
65.       else if( C1 >= T100MS -1) begin isTag <= 2'd1; C1 <= 28'd0; i <= i + 1'b1; end
66.       else C1 <= C1 + 1'b1;
67.
68.       5: // Key trigger press up
69.       if( isTag == 2'd3 ) begin isLClick <= 1'b1; i <= i + 1'b1; end
70.       else if( isTag == 2'd2 ) begin isDClick <= 1'b1; i <= i + 1'b1; end
71.       else if( isTag == 2'd1 ) begin isSClick <= 1'b1; i <= i + 1'b1; end
72.
73.       6: // Key trigger pree down
74.       begin { isLClick, isSClick, isDClick } <= 3'b000; i <= i + 1'b1; end
75.
76.       7: // L2H deounce check
77.       if( isTag == 2'd1 ) begin isTag <= 2'd0; i <= i + 2'd2; end
78.       else if( isTag == 2'd2 ) begin isTag <= 2'd0; i <= i + 1'b1; end
```

```

79.           else if( isTag == 2'd3 ) begin isTag <= 2'd0; i <= i + 1'b1; end
80.
81.           8: // Wait L2H
82.           if( isL2H ) begin i <= i + 1'b1; end
83.
84.           9: // L2H debounce
85.           if( C1 == T1OMS -1 ) begin C1 <= 28'd0; i <= 4'd0; end
86.           else C1 <= C1 + 1'b1;
87.
88.       endcase

```

第 45~88 行是核心操作，具体的操作过程如下：

步骤 0，等待第一次按下事件；

步骤 1，过滤抖动；

步骤 2，检测是不是长点击，如果是 isTag 为 3 然后出发步骤 5，否则等待释放事件；

步骤 3，过滤抖动；

步骤 4，检测是不是双击，如果是 isTag 为 2，否则 S 为 1；

步骤 5~6，根据 isTag 内容产生高脉冲；

步骤 7，根据 isTag 内容检测是否需要过滤抖动，isTag 为 1 直接返回步骤 0，其它需要；

步骤 8，等待释放事件；

步骤 9，过滤抖动然后返回步骤 0.

```

89.
90.      ****// sub-demo
91.
92.      reg [2:0]D1;
93.
94.      always @ ( posedge CLOCK or negedge RESET )
95.          if( !RESET )
96.              D1 <= 2'd0;
97.          else if( isLClick )
98.              D1[2] <= ~D1[2];
99.          else if( isDClick )
100.             D1[1] <= ~D1[1];
101.         else if( isSClick )
102.             D1[0] <= ~D1[0];
103.
104.     ****/
105.
106.     assign LED = D1;
107.
108. endmodule

```

以上内容为演示用的周边操作以及输出驱动。它会根据各种“有效按键”翻转 D1 的内容。第 106 行则是输出驱动的声明。编译完后下载程序。

如果笔者点击一下 `<KEY1>` 建，那么 `LED[0]` 会点亮，如果笔者双击 `<KEY1>` 建，结果 `LED[1]` 会点亮，再如果笔者长按 `<KEY1>` 建 3 秒不放，那么 `LED[2]` 则会点亮。总结说，一个按键资源可以执行 3 种功能，控制 3 位 LED 资源。

细节一：完整的个体模块



图 5.5 完整的按键功能模块。

如图 5.5 所示，那是完整的按键功能模块，它有一位输入端 KEY 连接至按键资源，然后则有 3 位触发信号，`Trig[2]`产生“单击”的个高脉冲，`Trig[1]` 产生“双击”的个高脉冲，`Trig[0]`产生“长击”的个高脉冲

key_funcmod.v

```
1. module key_funcmod
2. (
3.     input CLOCK, RESET,
4.     input KEY,
5.     output [2:0]oTrig
6. );
7. parameter T10MS      = 28'd500_000;
8. parameter T100MS     = 28'd5_000_000;
9. parameter T200MS     = 28'd10_000_000;
10. parameter T300MS    = 28'd15_000_000;
11. parameter T400MS    = 28'd20_000_000;
12. parameter T500MS    = 28'd25_000_000;
13. parameter T3S       = 28'd150_000_000;
14.
15. //*****
16.
17. reg F2,F1;
18.
```

```

19.      always @ ( posedge CLOCK or negedge RESET )
20.          if( !RESET )
21.              { F2, F1 } <= 2'b11;
22.          else
23.              { F2, F1 } <= { F1, KEY };
24.
25.      //*****
26.
27.      wire isH2L = ( F2 == 1 && F1 == 0 );
28.      wire isL2H = ( F2 == 0 && F1 == 1 );
29.      reg [3:0]i;
30.      reg isLClick, isDClick,isSClick;
31.      reg [1:0]isTag;
32.      reg [27:0]C1;
33.
34.      always @ ( posedge CLOCK or negedge RESET )
35.          if( !RESET )
36.              begin
37.                  i <= 4'd0;
38.                  isLClick <= 1'b0;
39.                  isDClick <= 1'b0;
40.                  isSClick <= 1'b0;
41.                  isTag <= 2'd0;
42.                  C1 <= 28'd0;
43.              end
44.          else
45.              case(i)
46.
47.                  0: // Wait H2L
48.                      if( isH2L ) begin i <= i + 1'b1; end
49.
50.                  1: // H2L debounce
51.                      if( C1 == T10MS -1 ) begin C1 <= 28'd0; i <= i + 1'b1; end
52.                      else C1 <= C1 + 1'b1;
53.
54.                  2: // Key Tag Check 1
55.                      if( isL2H ) begin C1 <= 28'd0; i <= i + 1'b1; end
56.                      else if( {F2,F1} == 2'b00 && C1 >= T3S -1 ) begin isTag <= 2'd3; C1 <= 28'd0; i <= 4'd5; end
57.                      else C1 <= C1 + 1'b1;
58.
59.                  3: // L2H debounce
60.                      if( C1 == T10MS -1 ) begin C1 <= 28'd0; i <= i + 1'b1; end
61.                      else C1 <= C1 + 1'b1;

```

```

62.
63.        4: // Key Tag Check 2
64.        if( isH2L && C1 <= T100MS -1 ) begin isTag <= 2'd2; C1 <= 28'd0; i <= i + 1'b1; end
65.        else if( C1 >= T100MS -1) begin isTag <= 2'd1; C1 <= 28'd0; i <= i + 1'b1; end
66.        else C1 <= C1 + 1'b1;
67.
68.        5: // Key trigger press up
69.        if( isTag == 2'd3 ) begin isLClick <= 1'b1; i <= i + 1'b1; end
70.        else if( isTag == 2'd2 ) begin isDClick <= 1'b1; i <= i + 1'b1; end
71.        else if( isTag == 2'd1 ) begin isSClick <= 1'b1; i <= i + 1'b1; end
72.
73.        6: // Key trigger pree down
74.        begin { isLClick, isSClick, isDClick } <= 3'b000; i <= i + 1'b1; end
75.
76.        7: // L2H deounce check
77.        if( isTag == 2'd1 ) begin isTag <= 2'd0; i <= i + 2'd2; end
78.        else if( isTag == 2'd2 ) begin isTag <= 2'd0; i <= i + 1'b1; end
79.        else if( isTag == 2'd3 ) begin isTag <= 2'd0; i <= i + 1'b1; end
80.
81.        8: // Wait L2H
82.        if( isL2H ) begin i <= i + 1'b1; end
83.
84.        9: // L2H debounce
85.        if( C1 == T10MS -1 ) begin C1 <= 28'd0; i <= 4'd0; end
86.        else C1 <= C1 + 1'b1;
87.
88.    endcase
89.
90.    *****/
91.
92.    assign oTrig = { isSClick, isDClick, isLClick };
93.
94. endmodule

```

实验六：数码管模块

有关数码管的驱动，想必读者已经学烂了 ... 不过，作为学习的新仪式，再烂的东西也要温故知新，不然学习就会不健全。黑金开发板上的数码管资源，由始至终都没有改变过，笔者因此由身怀念。为了点亮多位数码管从而显示数字，一般都会采用动态扫描，然而有关动态扫描的信息请怒笔者不再重复。在此，同样也是动态扫描，但我们却用不同的思路去理解。

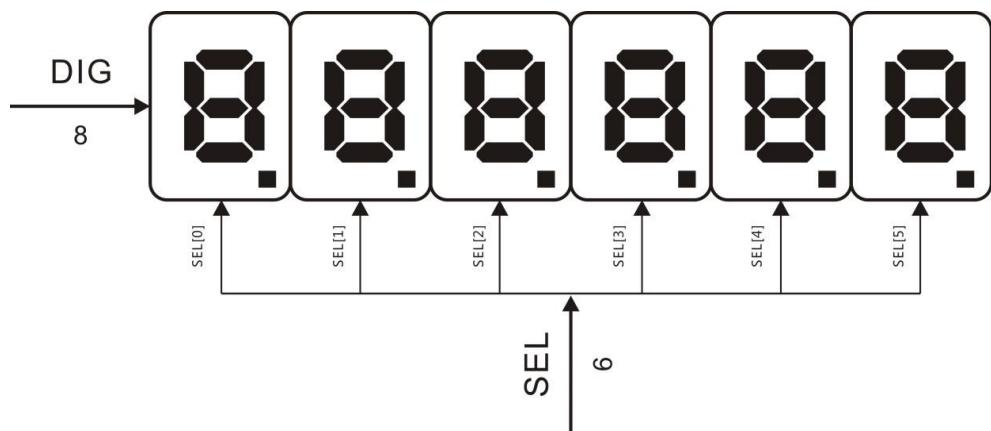


图 6.1 6 位数码管。

如图 6.1 所示，哪里有一排 6 位数码管，其中包好 8 位 DIG 信号还有 6 位 SEL 信号。DIG 为 digit，即俗称的数码管码，如果数码管预要显示 “A”，那么 DIG 必须输入 “A”的数码管码。SEL 为 select，即俗称的位选，从左至右即 SEL[0]~SEL[5]，如果想要使能第一位最左边的数码管，SEL[0]必须设置有效位。不管 DIG 还是 SEL，黑金开发板都是拉低有效，亦即 0 值表示有效位。

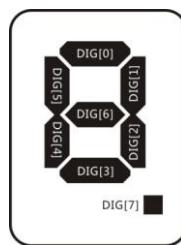


图 6.2 信号 DIG 与数码管码。

DIG 信号位宽为 8，亦即一个数码管资源内藏 8 个 LED，8 位 DIG 信号分别表示各个 LED，结果如图 6.2 所示。除了 DIG[7]较为少用以外，DIG[0]~[6] 一般都用来显示十六进制的数字 0~F。笔者在此强调，黑金开发板所采用的数码管是拉低有效，亦即点亮 LED 笔者设置为 0。为此，十六进制的数字 0~F 可以用 Verilog 这样表示，如代码 6.1 所示：

```
1.      parameter _0 = 8'b1100_0000, _1 = 8'b1111_1001, _2 = 8'b1010_0100,  
2.          _3 = 8'b1011_0000, _4 = 8'b1001_1001, _5 = 8'b1001_0010,
```

```

3.           _6 = 8'b1000_0010, _7 = 8'b1111_1000, _8 = 8'b1000_0000,
4.           _9 = 8'b1001_0000, _A = 8'b1000_1000, _B = 8'b1000_0011,
5.           _C = 8'b1100_0110, _D = 8'b1010_0001, _E = 8'b1000_0110,
6.           _F = 8'b1000_1110;

```

代码 6.1

如代码 6.1 所示，笔者用常量声明 16 个 16 进制的数字。

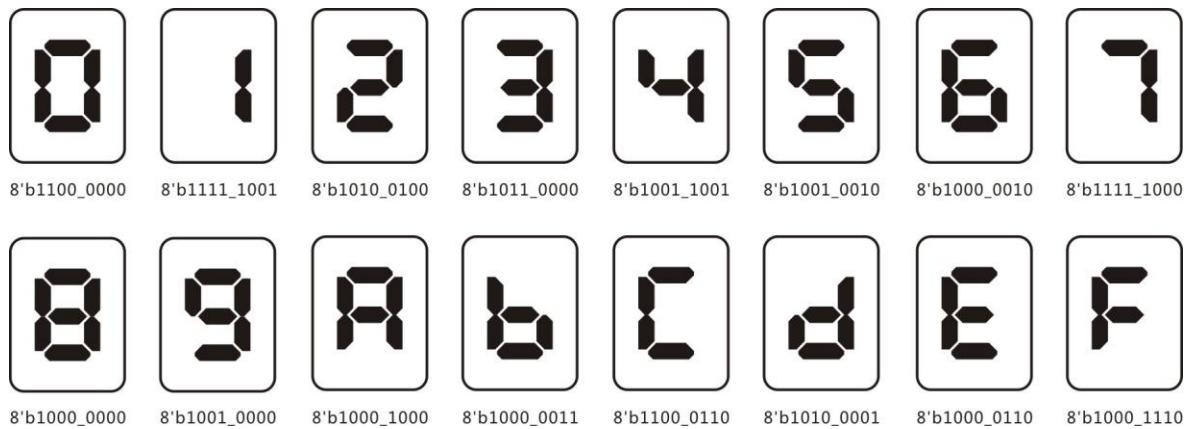


图 6.3 16 进制数字的数码管码（拉低有效）。

为了方便读者，笔者也顺便绘制一张直观的示意图 ... 如图 6.3 所示，哪里有 16 个 16 进制数字以及相关的数码管码。理解 DIG 信号与数码管码的关系以后，接下来笔者会解释 SEL 信号与数码管的关系。

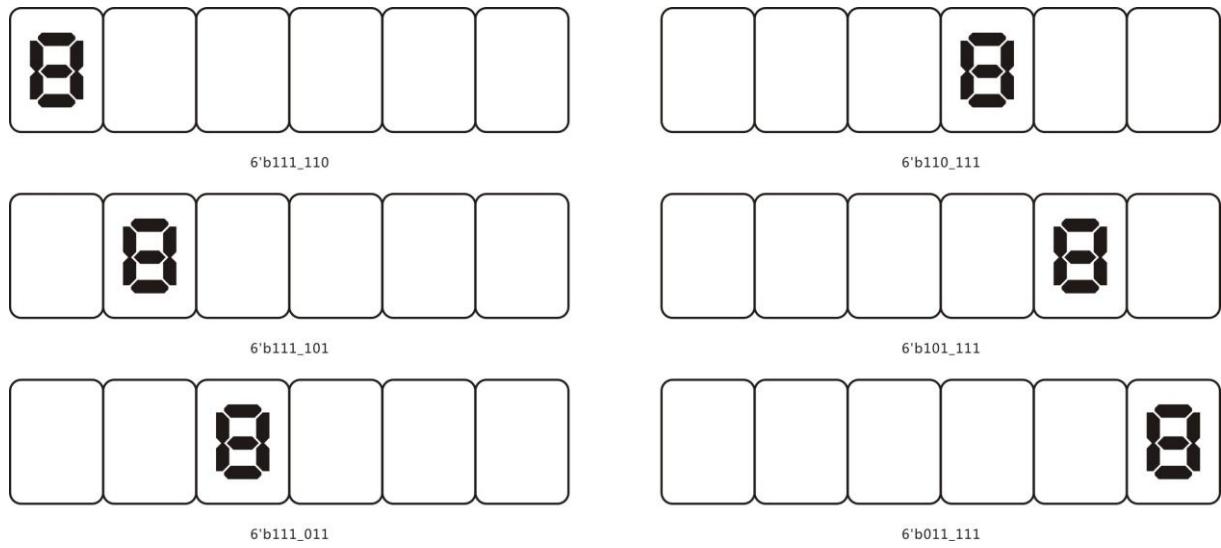


图 6.4 信号 SEL 与数码管码。

如图 6.4 所示，我们可以看见 SEL 信号与数码管的关系，每当 SEL 为值不同，相关的数码管就会显示数字，例如 SEL 为值 6'b111_110，最左边的数码管就会显示数字；SEL 为值 6'b011_111，最右边的数码管就会显示数字。如何实现自左向右轮流显示数字，就是将 6'b111_110 其中的“0 值”按间隔向左位移即可，这也是动态扫描最基本的理论。

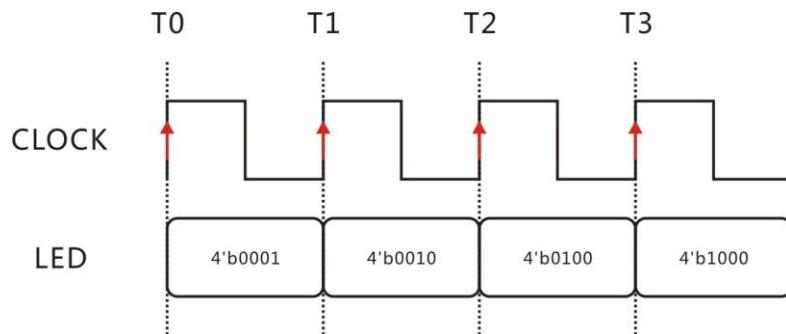


图 6.5 流水灯的理想时序图 (脑补)。

例如实验一的流水灯实验，流水操作负责轮流点亮 4 位 1 组的 LED 资源。假设流水间隔是一个时钟，如图 6.5 所示，LED 信号分别在 T0~T3 之间输出 4'b0001, 4'b0010, 4'b0100, 4'b1000，上述行为重复 N 次以后便产生流水效果。Verilog 则可以这样描述，结果如代码 6.2 所示：

```

1.      case( i )
2.
3.          0 :
4.              begin LED <= 4' b0001 ; i <= i + 1' b1; end
5.
6.          1 :
7.              begin LED <= 4' b0010 ; i <= i + 1' b1; end
8.
9.          2 :
10.             begin LED <= 4' b0100 ; i <= i + 1' b1; end
11.
12.             3 :
13.                 begin LED <= 4' b1000 ; i <= 4' d0; end
14.
15.             endcase

```

代码 6.2

如代码 6.2 所示，相较实验一的内容，步骤 0~3 也是实现流水效果，不过步骤 0~3 却没有考虑每个步骤所保持的时间，亦即流水间隔仅有一个时钟而已。

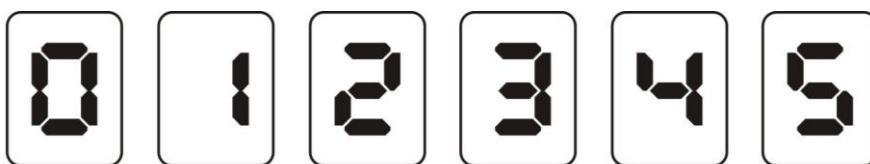


图 6.6 数码管显示数字的例子。

相较实验一的流水灯实验，动态扫描就是功能稍微复杂一点的流水等而已，SEL 信号类似 LED 信号，不过不是点亮 LED 而是负责位选工作，换之 DIG 信号则是显示内容。如图 6.6 所示，假设笔者想要显示上述的结果，即自左向右轮流显示数字 0~5，其中 DIG

信号负责数字 0~5 等信息，至于数字的显示次序则是 SIG 信号负责。

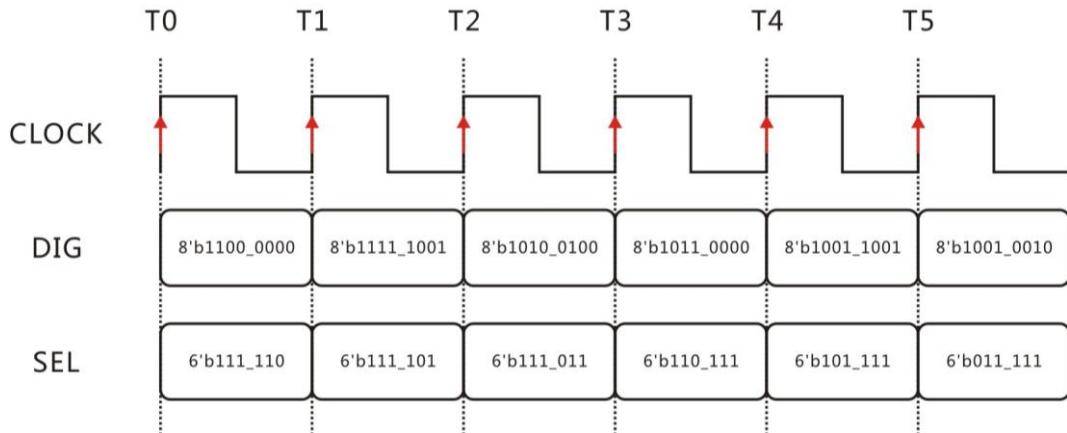


图 6.7 自左向右显示数字 0~5 的理想时序图 (脑补)。

假设流水间隔亦然是 1 个时钟，为了自左向右轮流显示数字“012345”，每个时钟的 SEL (位选) 信息，必须对应有效的 DIG 内容 (数码管码)，结果如图 6.6 所示。时序发生过程如下：

T0 的时候，DIG 发送未来值 8'b1100_0000，SEL 发送未来值 6'b111_110;
 T1 的时候，DIG 发送未来值 8'b1111_1001，SEL 发送未来值 6'b111_101;
 T2 的时候，DIG 发送未来值 8'b1010_0100，SEL 发送未来值 6'b111_011;
 T3 的时候，DIG 发送未来值 8'b1011_0000，SEL 发送未来值 6'b110_111;
 T4 的时候，DIG 发送未来值 8'b1001_1001，SEL 发送未来值 6'b101_111;
 T5 的时候，DIG 发送未来值 8'b1001_0010，SEL 发送未来值 6'b011_111;

Verilog 则可以这样表示，如代码 6.3 所示：

```

1.      case( i )
2.
3.          0 :
4.              begin DIG <= 8' b1100_0000 ; SEL <= 6' b111_110; i <= i + 1' b1; end
5.
6.          1 :
7.              begin DIG <= 8' b1111_1001 ; SEL <= 6' b111_101; i <= i + 1' b1; end
8.
9.          2 :
10.             begin DIG <= 8' b1010_0100 ; SEL <= 6' b111_011; i <= i + 1' b1; end
11.
12.             3 :
13.                 begin DIG <= 8' b1011_0000 ; SEL <= 6' b110_111; i <= i + 1' b1; end
14.
15.             4 :
16.                 begin DIG <= 8' b1001_1001 ; SEL <= 6' b101_111; i <= i + 1' b1; end
17.
18.             5 :
19.                 begin DIG <= 8' b1001_0010 ; SEL <= 6' b011_111; i <= 4' d0; end
20.
21.
```

16. endcase

代码 6.3

如代码 6.3 所示，步骤 0~5 分别对应 6 位数码管的显示次序。如步骤 0 为 DIG 被赋予 8'b1100_0000，即数字 0 的数码管码，期间 SEL 也被赋予 6'b111_110，即点亮第一位数码管（左边第一个）；步骤 1 为 DIG 被赋予 8'b1111_1001，即数字 1 的数码管码，期间 SEL 也被赋予 6'b111_101，即点亮第二位数码管（左边第二个）；至于步骤 2~5 以此类推，完后便返回步骤 0，重复一样的操作。

流水间隔亦即动态扫描频率，常规是 10ms，不过丧心病狂的笔者却设置为 100us。100us 经过 50Mhz 的时钟量化以后是 5000，Verilog 则可以这样表示：

```
parameter T100US = 13'd5000;
```

如果步骤 6.3 的流水间隔不是一个时钟而是 100us，那么代码 6.3 可以这样修改，修改结果如代码 6.4 所示：

```
1. case( i )
2.
3.     0:
4.         if( C1 == T100US -1 ) begin C1 <= 13' d0; i <= i + 1' b1; end
5.         else DIG <= 8' b1100_0000 ; SEL <= 6' b111_110; end
6.         ...
7.     5:
8.         if( C1 == T100US -1 ) begin C1 <= 13' d0; i <= i + 1' b1; end
9.         else DIG <= 8' b1001_0010 ; SEL <= 6' b011_111; end
10.
11. endcase
```

代码 6.4

理解这些内容以后，我们便可以开始建模了 ...

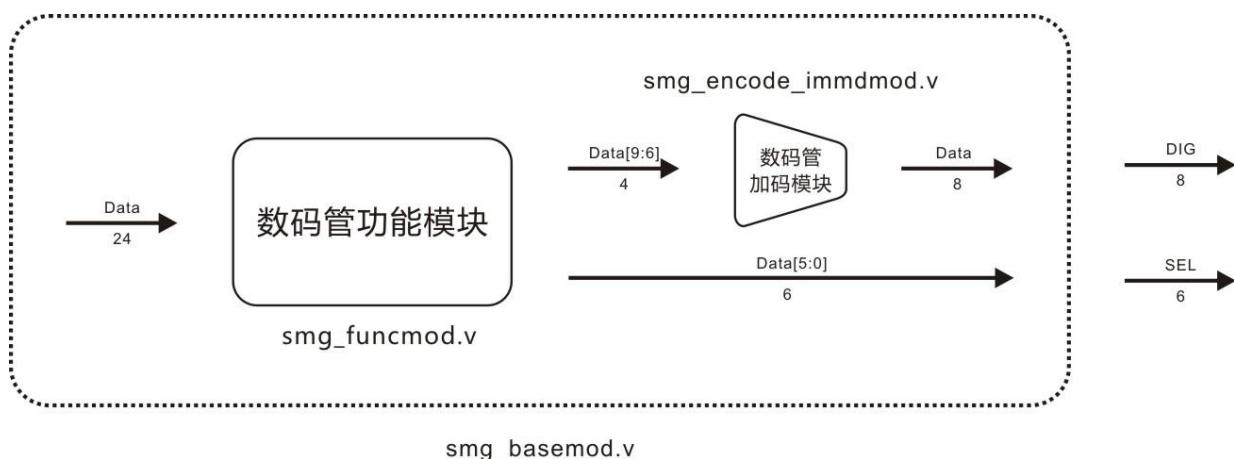


图 6.8 实验六的建模图。

如图 6.8 所示，那是实验六的建模图，其中 smg_basemod 是组合模块，它包含数码管功能模块，还有数码管加码模块。接下来，让我们来分析一下内部情况，数码管功能模块它有一组 24 位的 iData，然后又有 10 位 oData。随后 oData[9:6]会经由加码模块（即时模块）成为 8 位的数码管信息，并且驱动 DIG 顶层信号。反之 oData[5:0]则会直接驱动 SEL 顶层信号。

smg_funcmod.v

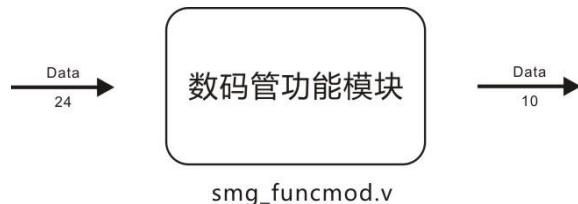


图 6.9 数码管功能模块。

接下来，让我们独自分析个体模块 ... 首先是数码管功能模块，人如其名它是负责所有数码管驱动工作的功能模块，24 位的 iData 分别针对 6 位数码管的显示内容（数字），位分配如表 6.1 所示：

表 6.1 输入数据位分配

位分配	[23..20]	[19..16]	[15..12]	[11..8]	[7..4]	[3..0]
数码管分配	第一位	第二位	第三位	第四位	第五位	第六位

至于 oData 的作用如上所示，oData[9:6]必须经由数码管加码模块，oData[5:0]则直接驱动 SEL 顶层信号。详细内容就让我们直接窥视代码吧：

```
1. module smg_funcmod
2. (
3.     input CLOCK, RESET,
4.     input [23:0]iData,
5.     output [9:0]oData
6. );
7. parameter T100US = 13'd5000;
```

以上内容为相关出入端声明。第 8 行则是 100us 的常量声明（流水间隔/停留时间）。

```
8.
9.     reg [3:0]i;
10.    reg [12:0]C1;
11.    reg [3:0]D1;
12.    reg [5:0]D2;
13.
14.    always @ ( posedge CLOCK or negedge RESET )
```

```

15.      if( !RESET )
16.          begin
17.              i <= 4'd0;
18.              C1 <= 13'd0;
19.              D1 <= 4'd0;
20.              D2 <= 6'b111_110;
21.          end

```

以上内容为相关的寄存器声明以及复位操作。D1 暂存 iData 的部分数据，D2 则暂存位选数据。第 17~21 则是这些寄存器的复位操作。

```

22.      else
23.          case( i )
24.
25.              0:
26.                  if( C1 == T100US -1 ) begin C1 <= 13'd0; i <= i + 1'b1; end
27.                  else begin C1 <= C1 + 1'b1; D1 <= iData[23:20]; D2 <= 6'b111_110; end
28.
29.              1:
30.                  if( C1 == T100US -1 ) begin C1 <= 13'd0; i <= i + 1'b1; end
31.                  else begin C1 <= C1 + 1'b1; D1 <= iData[19:16]; D2 <= 6'b111_101; end
32.
33.              2:
34.                  if( C1 == T100US -1 ) begin C1 <= 13'd0; i <= i + 1'b1; end
35.                  else begin C1 <= C1 + 1'b1; D1 <= iData[15:12]; D2 <= 6'b111_011; end
36.
37.              3:
38.                  if( C1 == T100US -1 ) begin C1 <= 13'd0; i <= i + 1'b1; end
39.                  else begin C1 <= C1 + 1'b1; D1 <= iData[11:8]; D2 <= 6'b110_111; end
40.
41.              4:
42.                  if( C1 == T100US -1 ) begin C1 <= 13'd0; i <= i + 1'b1; end
43.                  else begin C1 <= C1 + 1'b1; D1 <= iData[7:4]; D2 <= 6'b101_111; end
44.
45.              5:
46.                  if( C1 == T100US -1 ) begin C1 <= 13'd0; i <= 4'd0; end
47.                  else begin C1 <= C1 + 1'b1; D1 <= iData[3:0]; D2 <= 6'b011_111; end
48.
49.          endcase
50.
51.      assign oData = { D1,D2 };
52.
53. endmodule

```

以上内容为核心操作以及输出驱动声明，步骤 0~5 用来轮流点亮数码管，大概的思路之前已经解释过 ... 举例来说，步骤 0 为 D1 赋予 iData[23:20] 的内容，D2 赋予 6'b111_110 的内容，简单说就是将 iData[23:20] 的数字显示在第一位数码管，至于步骤 1~5 也是以此类推，完后操作会返回步骤 0。第 51 行则是输出驱动声明。

smg_encode_immdmod.v



smg_encode_immdmod.v

图 6.10 数码管加码（即时）模块。

如图 6.10 所示，数码管加码模块是一只即时模块，它有一组 4 位的 iData 与一组 8 位的 oData。该模块接收 iData 的内容，然后转换为数码管信息，最后再经由 oData 输出。具体内容，还是来浏览代码吧：

```

1. module smg_encode_immdmod
2. (
3.     input [3:0]iData,
4.     output [7:0]oData
5. );
6. parameter _0 = 8'b1100_0000, _1 = 8'b1111_1001, _2 = 8'b1010_0100,
7.           _3 = 8'b1011_0000, _4 = 8'b1001_1001, _5 = 8'b1001_0010,
8.           _6 = 8'b1000_0010, _7 = 8'b1111_1000, _8 = 8'b1000_0000,
9.           _9 = 8'b1001_0000, _A = 8'b1000_1000, _B = 8'b1000_0011,
10.          _C = 8'b1100_0110, _D = 8'b1010_0001, _E = 8'b1000_0110,
11.          _F = 8'b1000_1110;
12.
13. reg [7:0]D = 8'b1111_1111;
14.
15. always @ ( * )
16.     if( iData == 4'd0 ) D = _0;
17.     else if( iData == 4'd1 ) D = _1;
18.     else if( iData == 4'd2 ) D = _2;
19.     else if( iData == 4'd3 ) D = _3;
20.     else if( iData == 4'd4 ) D = _4;
21.     else if( iData == 4'd5 ) D = _5;
22.     else if( iData == 4'd6 ) D = _6;
23.     else if( iData == 4'd7 ) D = _7;

```

```

24.         else if( iData == 4'd8 ) D = _8;
25.         else if( iData == 4'd9 ) D = _9;
26.         else if( iData == 4'hA ) D = _A;
27.         else if( iData == 4'hB ) D = _B;
28.         else if( iData == 4'hC ) D = _C;
29.         else if( iData == 4'hD ) D = _D;
30.         else if( iData == 4'hE ) D = _E;
31.         else if( iData == 4'hF ) D = _F;
32.         else D = 8'dx;
33.
34.     assign oData = D;
35.
36. endmodule

```

第 3~4 行是出入端声明。第 6~11 行是数码管码 0~F 的常量声明。第 13 行是相关的寄存器声明。第 15~32 行则是加码操作。第 34 行是输出驱动声明。

smg_basemod.v

至于组合模块 smg_basemod 笔者就不重复贴图了，读者请自行看回图 6.8。详细的内容让我们来浏览代码吧：

```

1. module smg_basemod
2. (
3.     input CLOCK, RESET,
4.     output [7:0]DIG,
5.     output [5:0]SEL
6. );
7. wire [9:0]DataU1;
8.
9. smg_funcmod U1
10. (
11.     .CLOCK( CLOCK ),
12.     .RESET( RESET ),
13.     .iData( 24'hABCDEF ), // < top
14.     .oData( DataU1 ) // > U2
15. );
16.
17. assign SEL = DataU1[5:0];
18.
19. smg_encode_immdmod U2
20. (
21.     .iData( DataU1[9:6] ), // < U1

```

```

22.           .oData( DIG )      // > top
23.       );
24.
25. endmodule

```

该代码由于演示的作用，并没有将 U1 的 iData 直接引出，而是直接在其输入设置常量 24'hABCDEF（第 13 行）。至于相关的连线部署就复习图 6.8 吧。编译完后下载程序，我们便会发现数字“ABCDEF”分别自左向右显示在 6 位数码管。

细节一：完整的个体模块

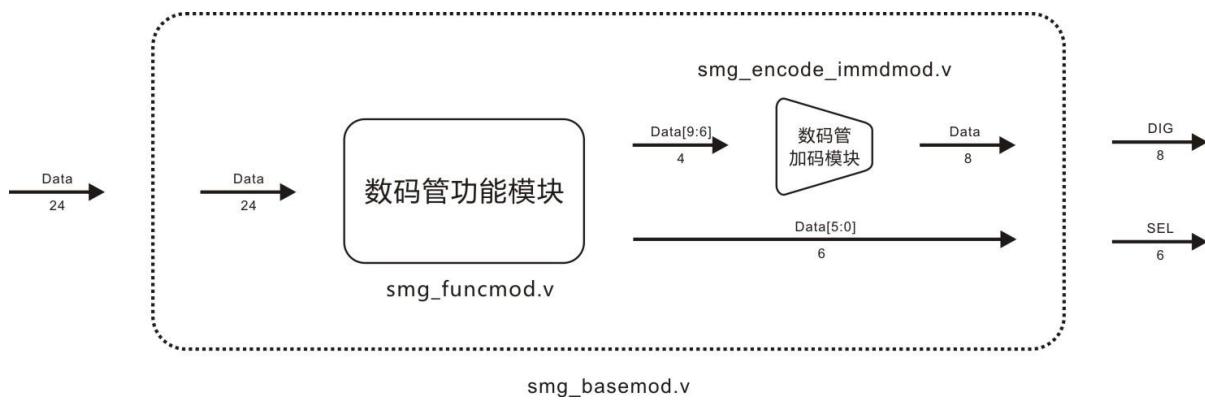


图 6.11 数码管基础模块的建模图。

如图 6.11 所示，那是完整的数码管基础模块，除了将 smg_funcmod 的 iData 向外引出，余下都一样。

smg_basemod.v

```

1. module smg_basemod
2. (
3.     input CLOCK, RESET,
4.     input [23:0]iData,
5.     output [7:0]DIG,
6.     output [5:0]SEL
7. );
8.     wire [3:0]DataU1;
9.
10.    smg_funcmod U1
11.    (
12.        .CLOCK( CLOCK ),
13.        .RESET( RESET ),
14.        .iData( iData ), // < top

```

```
15.           .oData( DataU1 ), // > U2
16. );
17.
18. assign SEL = DataU1[5:0];
19.
20. smg_encode_immdmod U2
21. (
22.     .iData( DataU1[9:6] ), // < U1
23.     .oData( DIG )         // > top
24. );
25.
26. endmodule
```

实验七：PS/2 模块① — 键盘

实验七依然也是熟悉的 PS/2 键盘。相较《建模篇》的 PS/2 键盘实验，实验七除了实现基本的驱动以外，我们还要深入解 PS/2 时序，还有 PS/2 键盘的行为。不过，为了节省珍贵的页数，怒笔者不再重复有关 PS/2 的基础内容，那些不晓得的读者请复习《建模篇》或者自行谷歌一下。

市场上常见的键盘都是应用第二套扫描码，各种扫描码如图 7.2 所示。《建模篇》之际，笔者也只是擦边一下 PS/2 键盘，简单读取单字节通码与断码而已。所谓单字节通码，就是有效的按下内容，例如 <A> 键被按下的时候会输出 1C。所谓单字节断码，就是有效的释放内容，例如 <A> 键被释放的时候会输出 F0 1C。

除了单字节的通码以外，PS/2 键盘也有双字节通码与断码。所谓双字节通码，例如 <R CTRL> 键被按下时候会输出 E0 14；反之，所谓双字节断码，例如 <R CTRL> 键被释放时候会输出 E0 F0 14。不管是单字节还是双字节，断码都包含 F0。

除了上述的要求以外，笔者还要实现双组合键，例如 <Ctrl> + <A>。不仅而已，笔者也要实现三组合键，例如 <Ctrl> + <Alt> + <A>。常识上，这些任性的要求都是软件的工作，然而这种认识也仅局限小气的脑袋而已。换做笔者，笔者就算霸王硬上弓，笔者也要使用 Verilog 实现这些任性的要求。

未进入实验之前，笔者需要强调一下！Verilog 究竟如何驱动 PS/2 设备，然后又如何实现软件的工作，这一切 Verilog 自有方法。不管 C 语言还有单片机这对活宝，驱动 PS/2 设备再怎么神，它们也没有资格在旁指指点点。读者千万也别尝试用借用它们的思路去思考 Verilog，否则后果只有撞墙而已。

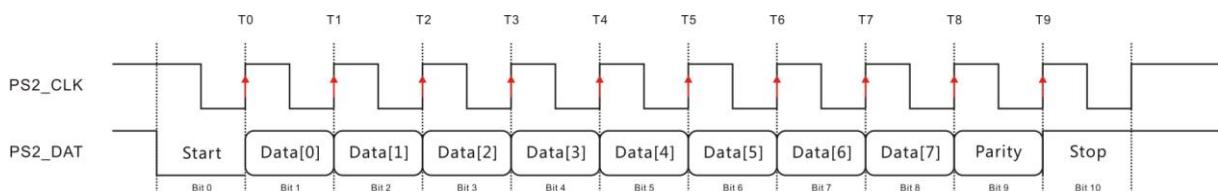


图 7.1 PS/2 键盘发送数据（主机视角）。

PS/2 传输协议与一般的传输协议一样，除了主从之分之余，它也有“读写”两个访问的方向。除非有特殊的需要，不然从机（FPGA）是不会访问 PS/2 键盘的内部。换之，从机只要不停从 PS/2 键盘哪里读取数据即可 … 换句话说，驱动 PS/2 键盘仅有读数据这一环而已，然而 PS/2 键盘是主机，FPGA 是从机。（主机的定义是时钟信号的拥有者）

不管是何种传输协议，只要协议当中存有时钟信号，那么“什么时钟沿，怎样对待数据”这个铁则不会改变的。如图 7.1 所示，那是 PS/2 键盘发送数据的时序图，亦即上升沿设置数据（输出数据）。根据主机视角，除了开始位以外，PS/2 键盘一共利用 10 个上升沿输出 10 位数据。

KEY	MAKE	BREAK	KEY	MAKE	BREAK	KEY	MAKE	BREAK
A	1C	F0,1C	9	46	F0,46	[54	F0,54
B	32	F0,32	'	0E	F0,0E	INSERT	E0,70	E0,F0,70
C	21	F0,21	-	4E	F0,4E	HOME	E0,6C	E0,F0,6C
D	23	F0,23	=	55	F0,55	PG UP	E0,7D	E0,F0,7D
E	24	F0,24	\	5D	F0,5D	DELETE	E0,71	E0,F0,71
F	2B	F0,2B	BKSP	66	F0,66	END	E0,69	E0,F0,69
G	34	F0,34	SPACE	29	F0,29	PG DN	E0,7A	E0,F0,7A
H	33	F0,33	TAB	0D	F0,0D	U ARROW	E0,75	E0,F0,75
I	43	F0,43	CAPS	58	F0,58	L ARROW	E0,6B	E0,F0,6B
J	3B	F0,3B	L SHFT	12	F0,12	D ARROW	E0,72	E0,F0,72
K	42	F0,42	L CTRL	14	F0,14	R ARROW	E0,74	E0,F0,74
L	4B	F0,4B	L GUI	E0,1F	E0,F0,1F	NUM	77	F0,77
M	3A	F0,3A	L ALT	11	F0,11	KP /	E0,4A	E0,F0,4A
N	31	F0,31	R SHFT	59	F0,59	KP *	7C	F0,7C
O	44	F0,44	R CTRL	E0,14	E0,F0,14	KP -	7B	F0,7B
P	4D	F0,4D	R GUI	E0,27	E0,F0,27	KP +	79	F0,79
Q	15	F0,15	R ALT	E0,11	E0,F0,11	KP EN	E0,5A	E0,F0,5A
R	2D	F0,2D	APPS	E0,2F	E0,F0,2F	KP .	71	F0,71
S	1B	F0,1B	ENTER	5A	F0,5A	KP 0	70	F0,70
T	2C	F0,2C	ESC	76	F0,76	KP 1	69	F0,69
U	3C	F0,3C	F1	05	F0,05	KP 2	72	F0,72
V	2A	F0,2A	F2	06	F0,06	KP 3	7A	F0,7A
W	1D	F0,1D	F3	04	F0,04	KP 4	6B	F0,6B
X	22	F0,22	F4	0C	F0,0C	KP 5	73	F0,73
Y	35	F0,35	F5	03	F0,03	KP 6	74	F0,74
Z	1A	F0,1A	F6	0B	F0,0B	KP 7	6C	F0,6C
0	45	F0,45	F7	83	F0,83	KP 8	75	F0,75
1	16	F0,16	F8	0A	F0,0A	KP 9	7D	F0,7D
2	1E	F0,1E	F9	01	F0,01]	5B	F0,5B
3	26	F0,26	F10	09	F0,09	;	4C	F0,4C
4	25	F0,25	F11	78	F0,78	'	52	F0,52
6	36	F0,36	PRNT SCRN	E0,12, E0,7C	E0,F0, 7C,E0, F0,12	.	49	F0,49
7	3D	F0,3D	SCROLL	7E	F0,7E	/	4A	F0,4A
8	3E	F0,3E	PAUSE	E1,I4,77, E1,F0,14, F0,77	-NONE-			

图 7.2 第二套键盘的扫描码。

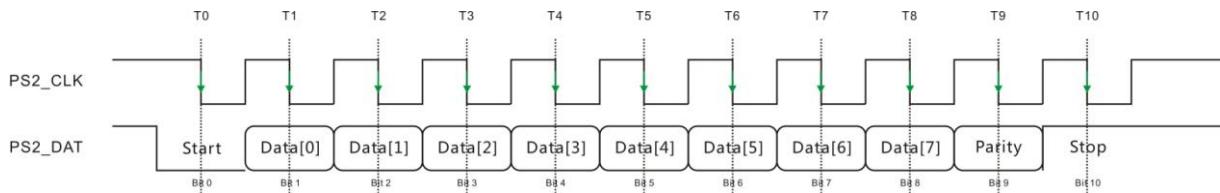


图 7.3 PS/2 键盘发送数据，FPGA 读取数据 (从机视角)。

PS/2 传输数据一般都是一帧一帧互相往来，一帧有 11 位数据。Bit 0 位为拉低的开始位，Bit 1~8 是由低自高的数据位，Bit 9 为校验位，Bit 10 为拉高的结束位。根据从机视角，如图 7.3 所示，PS/2 键盘在发送数据的时候，FPGA 是下降沿锁存数据（读取数据）。PS2_CLK 信号一共产生了 11 个下降沿，FPGA 也根据这 11 次下降沿锁存 11 位数据。

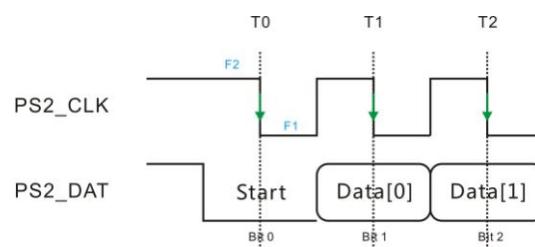


图 7.4 检测 PS2_CLK 的电平变化。

为了察觉下降沿，我们可以借用 F2~F1 的力量，对此 Verilog 可以这样表示：

```
reg F2, F1;
always @ ( posedge CLOCK )
  { F2,F1 } <= { F1,KEY };
```

然后下降沿声明为即时：

```
wire isH2L = ( F2 == 1 && F1 == 0 );
```

那么，从机接收 1 帧 11 位数据的操作可以这样描述，结果如代码 7.1 所示：

```
1.      case( i )
2.
3.          0 :
4.              if ( isH2L ) i <= i + 1' b1;
5.          1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 :
6.              if ( isH2L ) D1[i-1] <= PS2_DAT; i <= i + 1' b1; end
7.          9 :
8.              if ( isH2L ) i <= i + 1' b1;
9.          10 :
10.             if ( isH2L ) i <= 4 'd0 ;
```

```
11.  
12.      endcase
```

代码 7.1

不过，为了方便控制代码 7.1，笔者设法将代码 7.1 设置为伪函数，结果如代码 7.2 所示：

```
1.      parameter RDFUNC = 4' d4;  
2.      .....  
3.      case( i )  
4.          .....  
5.          /*****  
6.          4 :  
7.              if ( isH2L ) i <= i + 1' b1;  
8.              5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 :  
9.                  if ( isH2L ) D1[i-5] <= PS2_DAT; i <= i + 1' b1; end  
10.             13 :  
11.                 if ( isH2L ) i <= i + 1' b1;  
12.                 14 :  
13.                     if ( isH2L ) i <= Go ;  
14.  
15.      endcase
```

代码 7.2

理解这些以后，我们就要开始认识 PS/2 键盘的按键行为了。

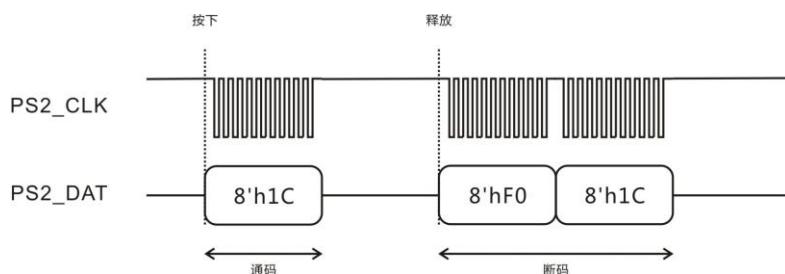


图 7.5 PS/2 键盘，按一下又释放。

假设笔者轻按一下<A>然后又释放，如图 7.5 所示，PS/2 键盘先会发送一帧 8'h1C 的通码，然后又发送两帧 8'hF0 8'h1C 的断码，这是 PS/2 键盘最常见的按键行为。

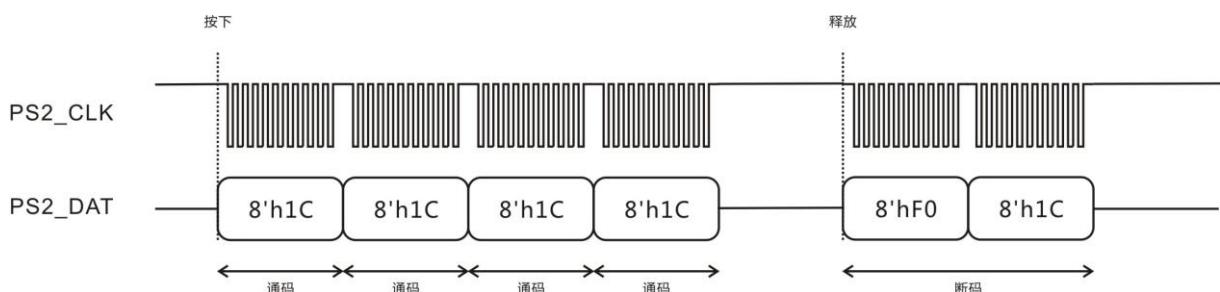


图 7.6 PS/2 键盘，长按又释放。

如果笔者长按 $<\text{A}>$ 键不放，如图 7.6 所示，PS/2 键盘会不停发送通码，直至释放才发送断码。至于长按期间，通码的发送间隔大约是 100ms，亦即 1 秒内发送 10 个通码。

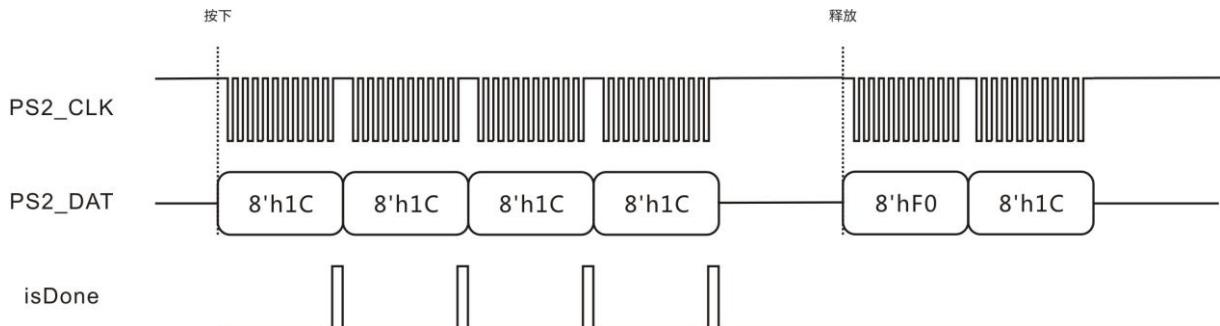


图 7.7 PS/2 键盘，有效通码。

一般而言，我们都会选择通码放弃断码，为了表示一次性，而且也是有效性的通码。每当一帧通码完成接收，`isDone` 就会产生一个高脉冲，以示一次性而且有效的通码已经接收完毕。

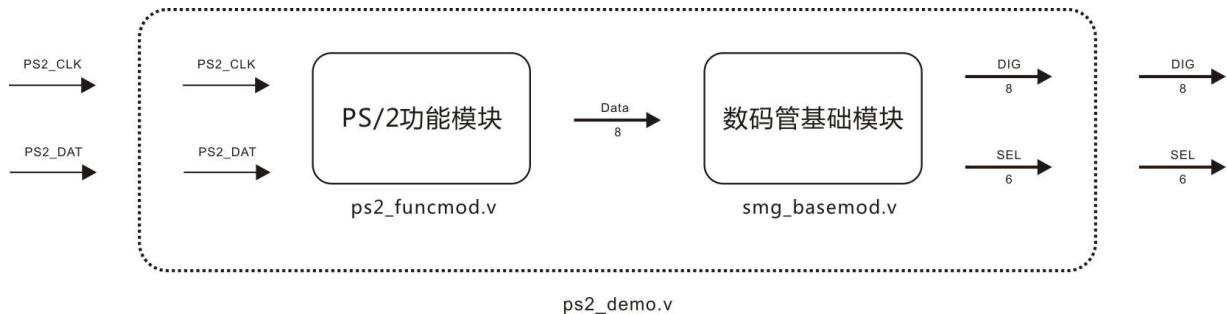


图 7.8 实验七的建模图。

如图 7.8 所示，那是实验七的建模图，其中名为 `ps2_demo` 的组合模块，内容包含实验六的 `smg_basemod` 以外，该组合模块也包含 `ps2_funcmod`。PS/2 功能模块接收来 PS/2 键盘发送过来的数据，然后再经由 `oData` 驱动 `smg_basemod` 的 `iData`，最后并将通码显示在数码管上。

`ps2_funcmod.v`

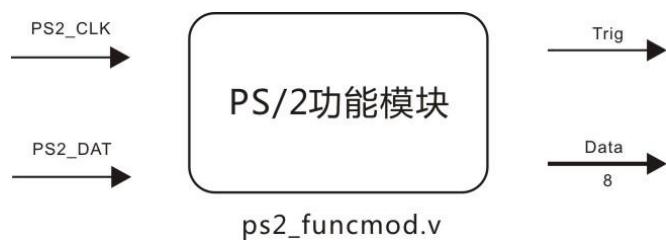


图 7.9 PS/2 功能模块。

图 7.9 是 PS/2 功能模块的建模图，左方是 PS2_CLK 与 PS2_DAT 顶层信号的输入，右方则是 1 位 oTrig 与 8 位 oData。具体内容，让我们来瞧瞧代码：

```
1. module ps2_funcmod
2. (
3.     input CLOCK, RESET,
4.     input PS2_CLK, PS2_DAT,
5.     output oTrig,
6.     output [7:0]oData
7. );
8. parameter BREAK = 8'hF0;
9. parameter FF_Read = 5'd4;
```

以上内容为相关的出入端声明以及常量声明。第 8 行是断码的常量声明(第一帧)，第 9 行则是伪函数的入口。

```
10.
11.    /***** // sub
12.
13.    reg F2,F1;
14.
15.    always @ ( posedge CLOCK or negedge RESET )
16.        if( !RESET )
17.            { F2,F1 } <= 2'b11;
18.        else
19.            { F2, F1 } <= { F1, PS2_CLK };
20.
21.    /***** // core
22.
23.    wire isH2L= ( F2 == 1'b1 && F1 == 1'b0 );
24.    reg [7:0]D1;
25.    reg [4:0]i,Go;
26.    reg isDone;
27.
28.    always @ ( posedge CLOCK or negedge RESET )
29.        if( !RESET )
30.            begin
31.                D1<= 8'd0;
32.                i <= 5'd0;
33.                Go <= 5'd0;
34.                isDone <= 1'b0;
35.            end
```

```
36.           else
```

以上内容是周边操作以及相关寄存器声明，还有它们的复位操作。周边操作主要用来检测 PS2_CLK 的电平变化。第 23 行是下降沿的即时声明。第 24~26 行是相关的寄存器声明，第 30~34 行则是这些寄存器的复位操作。

```
37.           case( i )
38.
39.               0:
40.                   begin i <= FF_Read; Go <= i + 1'b1; end
41.
42.               1:
43.                   if( D1 == BREAK ) begin i <= FF_Read; Go <= 5'd0; end
44.                   else i <= i + 1'b1;
45.
46.               2:
47.                   begin isDone <= 1'b1; i <= i + 1'b1; end
48.
49.               3:
50.                   begin isDone <= 1'b0; i <= 5'd0; end
51.
52.               /***** // PS2 read function
53.
54.               4: // Start bit
55.                   if( isH2L ) i <= i + 1'b1;
56.
57.               5,6,7,8,9,10,11,12: // Data byte
58.                   if( isH2L ) begin D1[ i-5 ] <= PS2_DAT; i <= i + 1'b1; end
59.
60.               13: // Parity bit
61.                   if( isH2L ) i <= i + 1'b1;
62.
63.               14: // Stop bit
64.                   if( isH2L ) i <= Go;
65.
66.           endcase
```

以上内容为核心操作。其中步骤 4~14（第 54~64 行）是读取 1 帧数据的伪函数，入口地址是 4。步骤 0~3 则是主要操作，过程如下：

步骤 0，进入伪函数准备读取第一帧数据。读完第一帧数据以后便返回步骤 1。

步骤 1，判断第一帧数据是否为断码？是，进入伪函数，完整第二帧数据的读取，然后

返回步骤指向为 0。否，继续步骤。

步骤 2~3，产生完成的触发信号，然后返回步骤 0。

```
67.  
68.      ****  
69.  
70.      assign oTrig = isDone;  
71.      assign oData = D1;  
72.  
73.      ****  
74.  
75. endmodule
```

以上内容为输出驱动的声明。

ps2_demo.v

组合模块 ps2_demo 的联系部署请复习图 7.8。

```
1. module ps2_demo  
2.  (  
3.      input CLOCK, RESET,  
4.      input PS2_CLK, PS2_DAT,  
5.      output [7:0]DIG,  
6.      output [5:0]SEL  
7.  );  
8.  
9.      wire [7:0]DataU1;  
10.  
11.     ps2_funcmod U1  
12.     (  
13.         .CLOCK(CLOCK ),  
14.         .RESET(RESET ),  
15.         .PS2_CLK( PS2_CLK ), // < top  
16.         .PS2_DAT( PS2_DAT ), // < top  
17.         .oData( DataU1 ), // > U2  
18.         .oTrig()  
19.     );  
20.  
21.     smg_basemod U2  
22.     (  
23.         .CLOCK(CLOCK ),
```

```

24.      .RESET( RESET ),
25.      .DIG( DIG ), // > top
26.      .SEL( SEL ), // > top
27.      .iData( { 16'h0000, DataU1 } ) // < U1
28.    );
29.
30. endmodule

```

上述代码没有什么特别，除了第 18 行，无视触发信号的输出以外，还有第 27 行其 16'h0000 则表示数码管的前四位皆为 0，后两位则是通码。编译完后便下载程序。

如果笔者按下 `<A>` 键，数码管便会显示 1C；如果笔者释放 `<A>` 键，数码管也是显示 1C，期间也会发生一丝的闪耀。由于 ps2_funcmod 的暂存空间 D 直接驱动 oData，所以数码管事实反映 ps2_funcmod 的读取状况。从演示上来看的确如此，不过在时序身上，唯有通码读取成功以后，才会产生触发信号。

细节一：主操作与伪函数的距离

```

1. case( i )
2.
3.   0:
4.     begin i <= FF_Read; Go <= i + 1'b1; end
5.   1:
6.     if( D1 == BREAK ) begin i <= FF_Read; Go <= 5'd0; end
7.     else i <= i + 1'b1;
8.   2:
9.     begin isDone <= 1'b1; i <= i + 1'b1; end
10.  3:
11.    begin isDone <= 1'b0; i <= 5'd0; end
12.    /***** // PS2 read function
13.    4: // Start bit
14.      if( isH2L ) i <= i + 1'b1;
15.      5,6,7,8,9,10,11,12: // Data byte
16.      if( isH2L ) begin D1[ i-5 ] <= PS2_DAT; i <= i + 1'b1; end
17.      13: // Parity bit
18.      if( isH2L ) i <= i + 1'b1;
19.      14: // Stop bit
20.      if( isH2L ) i <= Go;
21.
22. endcase

```

代码 7.3

如代码 7.3 所示，步骤 0~3 是主操作，步骤 4~14 则是伪函数，期间主操作的下任步骤

直接连接伪函数的入口。一般而言，如果模块的核心操作是小功能的话，这样做倒没有什么问题。反之，如果遇上复杂功能的核心操作，主操作与伪函数之间必须隔空一段距离。根据笔者的习惯，默认下都会设为 16 或者 32，不过也有例外的情况。

```
23. case( i )
24.
25.     0:
26.         begin i <= FF_Read; Go <= i + 1'b1; end
27.     1:
28.         if( D1 == BREAK ) begin i <= FF_Read; Go <= 5'd0; end
29.         else i <= i + 1'b1;
30.     2:
31.         begin isDone <= 1'b1; i <= i + 1'b1; end
32.     3:
33.         begin isDone <= 1'b0; i <= 5'd0; end
34.         //***** // PS2 read function
35.     16: // Start bit
36.         if( isH2L ) i <= i + 1'b1;
37.     17,18,19,20,21,22,23,24: // Data byte
38.         if( isH2L ) begin D1[ i-5 ] <= PS2_DAT; i <= i + 1'b1; end
39.     25: // Parity bit
40.         if( isH2L ) i <= i + 1'b1;
41.     26: // Stop bit
42.         if( isH2L ) i <= Go;
43.
44. endcase
```

代码 7.4

如代码 7.4 所示，伪函数的入口地址已经设为 16，为此主操作与伪函数之间有 16 个步骤的距离。如此一来，主操作拥有更多的步骤空间。

细节二：完整的个体模块



图 7.10 PS/2 键盘功能模块。

图 7.10 是 PS/2 键盘功能模块，内容基本上与 PS/2 功能模块一模一样，至于区别就是穿上其它马甲而已，所以怒笔者不再重复粘贴了。

实验八：PS/2 模块② — 键盘与组合键

实验七之际，我们学习如何读取 PS/2 键盘发送过来的通码与断码，不过实验内容也是一键按下然后释放，简单按键行为而已。然而，实验八的实验内容却是学习组合键的按键行为。

不知读者是否有类似的经历？当我们使用键盘的时候，如果 5~6 按键同时按下，电脑随之便会发出“哔哔”的警报声，键盘立即失效。这是键盘限制设计，不同产品也有不同限制的按键数量。默认下，最大按键数量是 5~7 个。所谓组合键就是两个以上的按键所产生的有效按键。举例而言，按下按键 `<A>` 输出“字符 a”，按下 `<Shift> + <A>` 便输出“字符 A”。不过要实现组合键，我们必须深入了解键盘的按键行为不可。

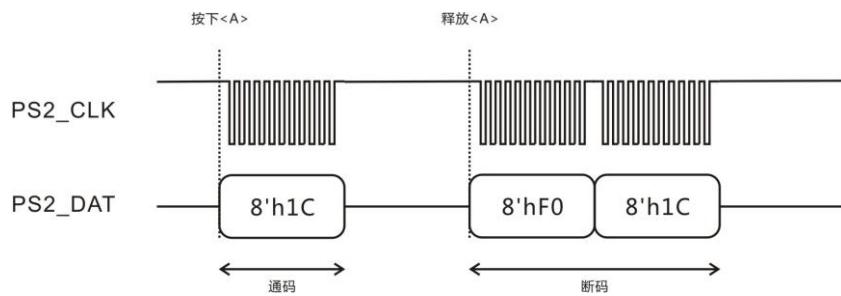


图 8.1 按下又立即释放。

PS/2 键盘最常见的按键行为是按下以后又立即释放，假设笔者按下 `<A>` 键又立即释放 `<A>` 键，那么 PS/2 键盘便会产生类似图 8.1 的时序。如图 8.1 所示，当笔者按下 `<A>` 的时候，PS/2 键盘便会发送 `8'h1C` 的通码；反之，如果 `<A>` 被释放，PS2 键盘也会立即发送 `8'hF0 8'h1C` 的断码。

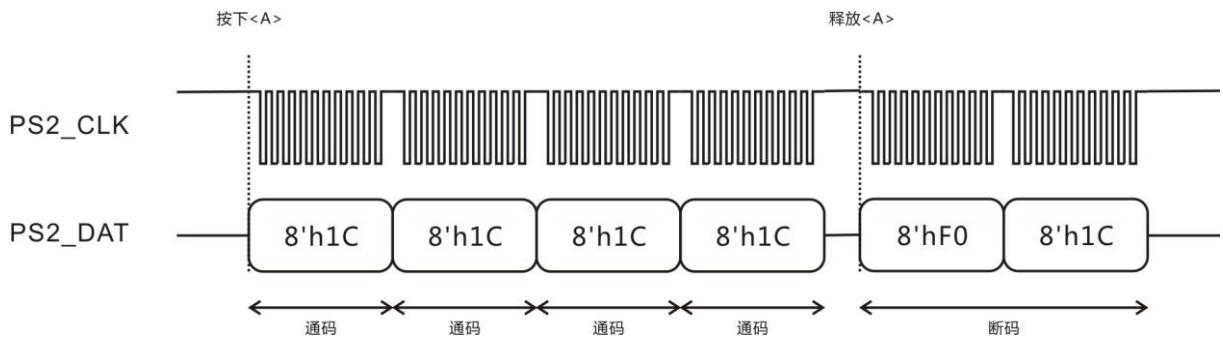


图 8.2 长按又立即释放。

如果笔者手痒长按 `<A>` 不放，那么 PS/2 键盘便会按照 100ms 的间隔时间，不断发送通码 `8'h1C`。期间，如果笔者释放 `<A>`，那么 PS/2 键盘便会发送 `8'hF0 8'h1C` 的断码，时序结果如图 8.2 所示。不管是图 8.1 还是图 8.2 的情况，都是 PS/2 键盘最常见的按键行为，亦即单键行为。话虽如此，单键行为既是最基础的按键行为，多键行为也必须基于它。

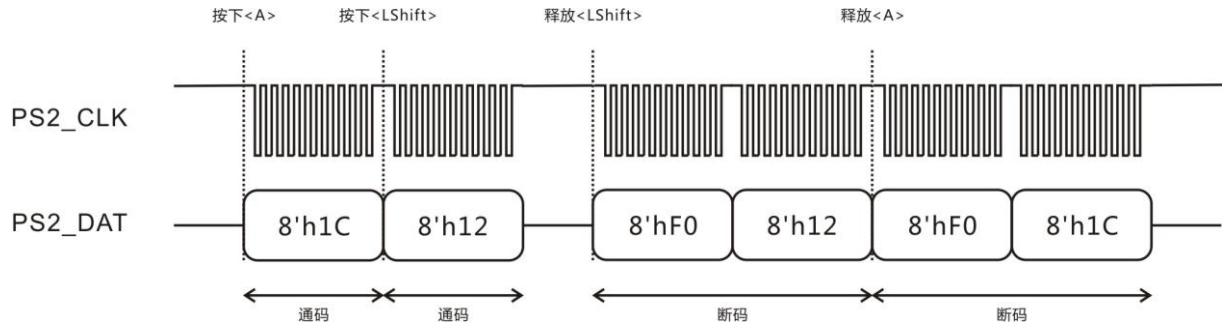


图 8.3 多键行为，先按后放①。

多键行为不同于单键行为，因为多键行为同时存在两个以上的按键被按下，因此多键行为便有先按后放，先按先放等次序。假设笔者先按下 `<A>`，然后又按下 `<LShift>`，随之 PS/2 键盘便会接续发送通码 `8'h1C` 与 `8'h12`。如果笔者想要撒手，`<LShift>` 必须事先释放，再者是 `<A>`，结果 PS/2 键盘便会连续发送 `8'hF0` `8'h12` 与 `8'hF0` `8'h1C` 的断码。

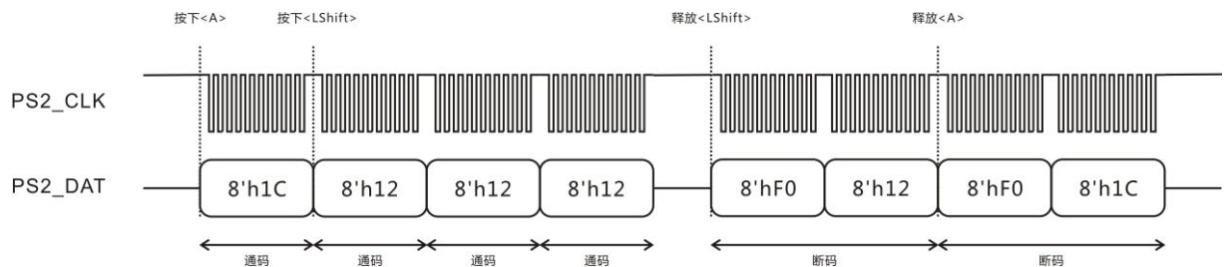


图 8.3 多键行为，先按后放②。

再假设笔者先按下 `<A>` 后按下 `<LShift>` 以后并没有立即释放任何按键，作为最后按下的按键，它可以得到执行权。如图 8.3 所示，笔者先是按下 `<A>` 然后又按下 `<Shift>`，那么 PS/2 键盘便会接续发送 `8'h1C` 与 `8'h12` 等通码。假设笔者手指麻痹没有立即释放任何按键，那么 `<LShift>` 就会得到执行权，结果保持长按状态。此刻，PS/2 键盘便会不停发送 `<LShift>` 的通码。

一旦手指回复知觉，然后按照先按后放的次序，先行释放 `<LShift>` 然后释放 `<A>`，结果 PS/2 键盘便会接续发送 `8'hF0` `8'h12` 与 `8'hF0` `8'h1C` 等断码。

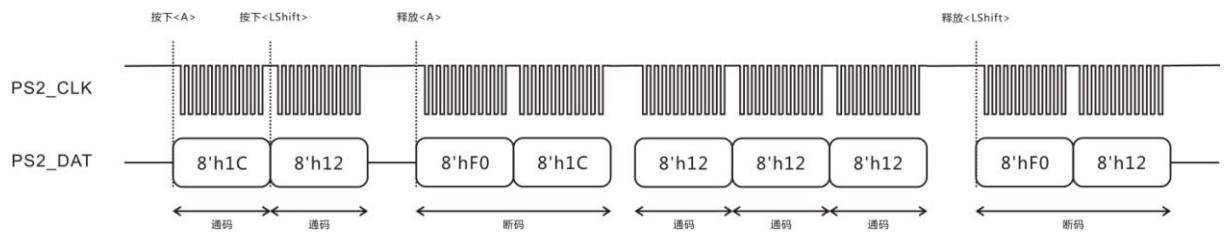


图 8.5 多键行为，先按先放。

如果读者不是按照先按后放，而是先按先放的次序，先按下 `<A>`，后按下 `<LShift>` 的话 ... 如图 8.5 所示，假设笔者先按下 `<A>`，然后又按下 `<LShift>`，此刻 PS/2 键盘便会接续发送 `8'h1C` 与 `8'h12` 等通码。期间，笔者忽然手痒，觉得先按先放比较好玩，

于是笔者故意松开 `<A>`，此刻 PS/2 键盘便会发送 `8'hF0 8'h1C` 的断码。

同一时刻，`<LShift>` 亦然保持按下的姿势，PS/2 键盘发送完毕 `<A>` 的断码以后，PS/2 键盘也会不停发送 `<LShift>` 的通码 … 直至笔者释放 `<LShift>`，PS/2 键盘发送 `8'hF0 8'h12` 的断码为止。

多键行为的终点就在于“先按后放”还是“先按先放”。不管是哪一种次序，下一刻按键都会抢夺上一刻按键的执行权与长按状态。不过根据习惯，先按后放固然已经成为主流，唯有意外或者那个神经不协调的傻子才会选择先按先放的次序。当我们理解 PS/2 键盘的多键行为以后，我们便可以开始实现组合键。

根据笔者的认识，PS/2 键盘也有按键分类，如：`<Shift>`，`<Ctrl>` 还有 `<Alt>` 等按键，它们都是常见的组合（补助）按键。除此之外，笔记本或者一些特殊键盘也有不同的组合键，如：`<FN>` 与 `<WIN>` 按键。一般而言，我们都认为组合键是软件的工作，虽然这是不择不扣的事实，不过我们只要换个思路，Verilog 也可以实现组合键。对此，我们只要将一只组合键视为一个立旗状态，所有难题都能迎刃而解。

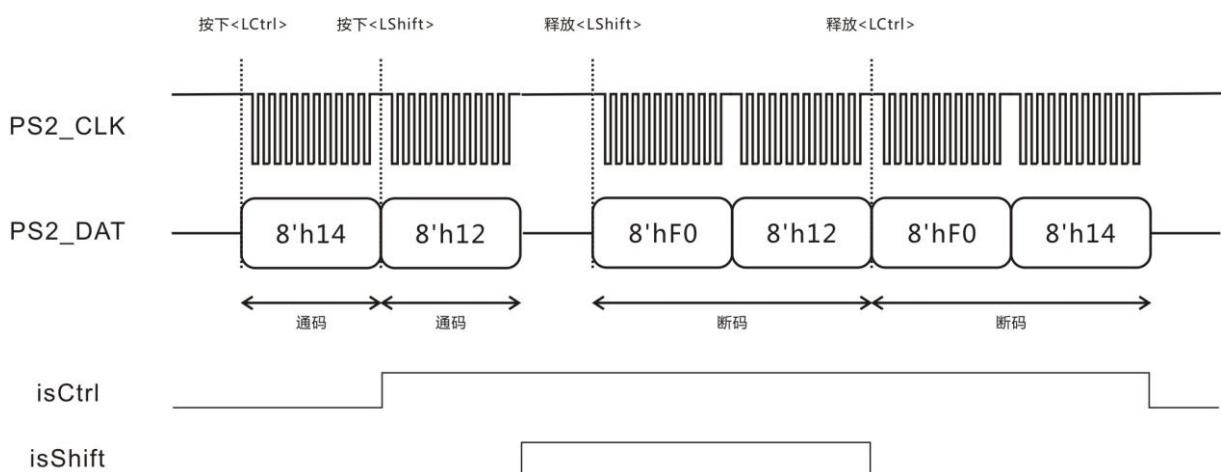


图 8.6 组合键与立旗状态。

假设笔者先按下 `<LCtrl>` 又按下 `<LShift>`，PS/2 键盘发送完毕 `<LCtrl>` 的通码以后，`isCtrl` 便会立旗。紧接着 PS/2 键盘又会发送 `<LShift>` 的通码，随后 `isShift` 也会立旗。事后，笔者先释放 `<LShift>` 再释放 `<LCtrl>`，那么 PS/2 键盘便会接续发送 `<LShift>` 与 `<LCtrl>` 的断码。`<LShift>` 断码发送完毕以后，`isShift` 便会消除立旗。同样 `<LCtrl>` 断码发送完毕以后 `isCtrl` 也会消除立旗。

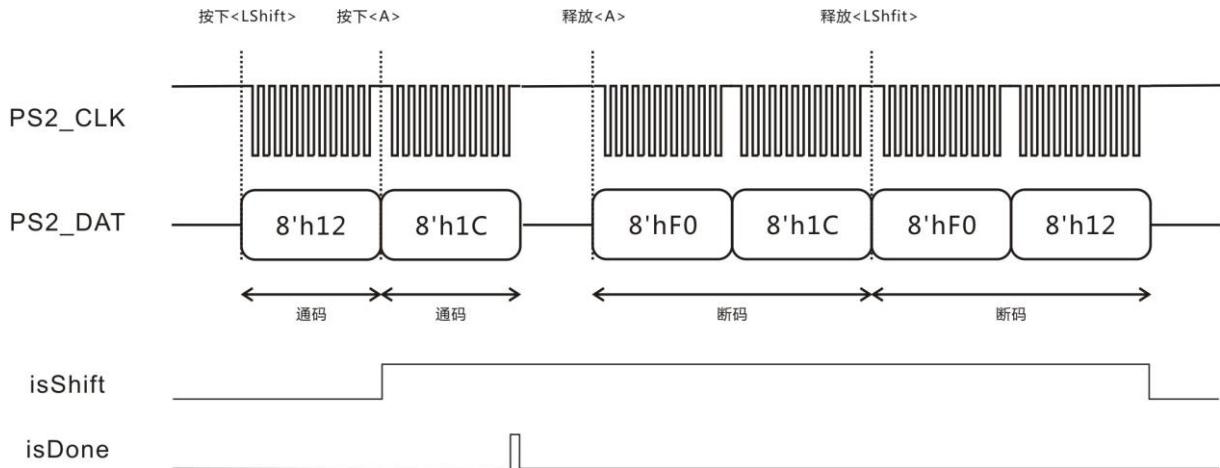


图 8.7 有效的组合键①。

为了表示有效的组合键，我们依然需要 `isDone` 这个高脉冲，我们虽然知道 `isDone` 产生高脉冲都是一般通码输出以后。不过在此，组合键不被认为是一般通码。如图 8.7 所示，假设笔者先按下 `<LShift>` 又按下 `<A>`，`<LShift>` 通码发送完毕以后便立旗 `isShift`；`<A>` 通码发送完毕以后便拉高一会儿 `isDone`。如果此刻 `isShift` 为拉高状态，而且通码 `<A>` 又有效，那么有效的组合键 `<Shift> + <A>` 便产生。

完后，笔者先释放 `<A>` 在释放 `<LShift>`，PS/2 键盘便会接续发送 `<A>` 与 `<LShift>` 的断码。`<A>` 的断码没有产生任何效果，反之 `<LShift>` 的断码则消除 `isShift` 的立旗状态。

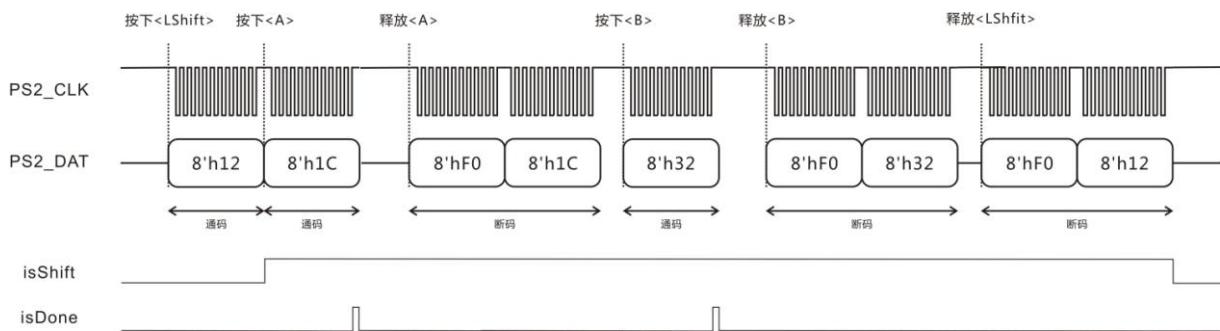


图 8.8 有效的组合键②。

为了产生各种各样的有效组合键，我们不可能不断按下又释放组合键 ... 换言之，不断切换的家伙只有非组合键而已，组合键则一直保持有效的状态，直至发送断码为止。如图 8.8 所示，假设笔者先按下 `<LShift>` 又按下 `<A>`，`<LShift>` 通码使 `isShift` 立旗，`<A>` 通码使 `isDone` 产生高脉冲，对此组成键 `<Shift> + <A>` 完成。

随后，笔者释放 `<A>`，PS/2 键盘便发送 `<A>` 断码。不一会，笔者又按下 ``，`` 通码使 `isDone` 产生高脉冲，结果完成组合键 `<Shift> + `。事后，笔者释放 `` 又释放 `<LShift>`，PS/2 键盘便会接续发送断码 `` 与 `<LShift>`，`` 断码没有异样，`<LShift>` 断码则消除 `isShift` 的立旗状态。

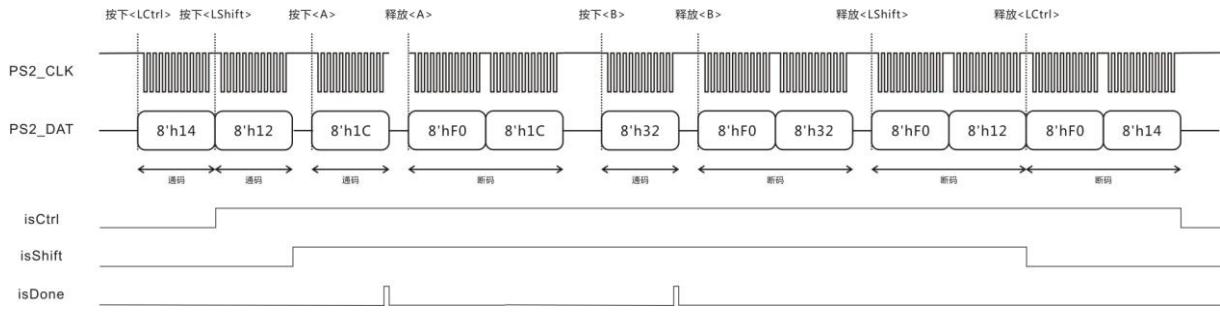


图 8.9 多状态有效组合键。

除了当个组合键（一个立即状态）以外，同样的道理也能实现多个组合键（多个立旗状态）。如图 8.9 所示，笔者先是按下 `<LCtrl>` 又按下 `<LShift>`，`<LCtrl>` 通码立旗 `isCtrl` 状态，`<LShift>` 通码则立旗 `isShift` 状态。紧接着笔者又按下 `<A>`，`<A>` 通码导致 `isDone` 产生一个高脉冲，此刻组合键 `<Ctrl> + <Shift> + <A>` 已经完成。然后笔者释放 `<A>` 使其产生 `<A>` 断码。

不一会，笔者又按下 ``，结果 `` 通码驱使 `isDone` 又产生另一个高脉冲，此刻组合键 `<Ctrl> + <Shift> + ` 已经完成。心满意足的笔者接续释放 ``，`<LShift>` 还有 `<LCtrl>`。`` 断码没有任何异样，`<LShift>` 断码消除 `isShift` 立旗状态，`<LCtrl>` 断码则消除 `isCtrl` 立旗状态。

一般而言，组合键最多可以达到 3 级，亦即 `<Ctrl> + <Shift> + <Alt> + ?`。话虽如此，除非对方的手指比猴子更灵活，不然要同时按照次序按下 4 个按键是一件容易伤害手指的蠢事。换之，一级与两级的组合键已经足够应用。理论上，Verilog 要实现多少级组合键也没有问题，但是过多的功能只是浪费而已。

好了，上述这些内容理解完毕以后，我们便可以开始建模了！

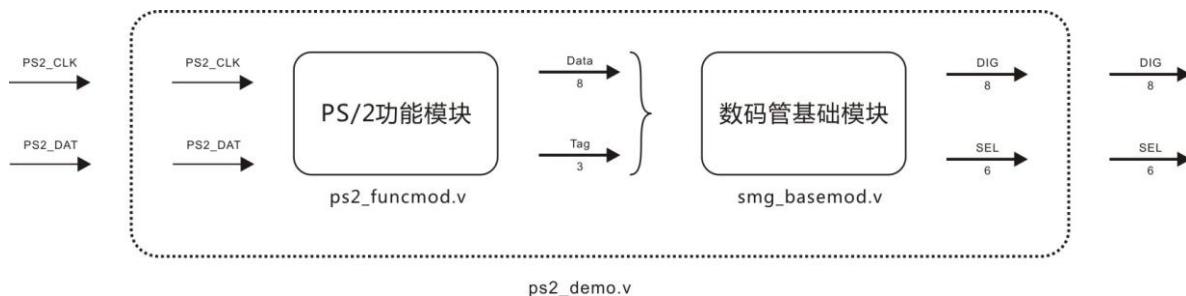


图 8.10 实验八建模图。

图 8.10 是实验八的建模图，一个名为 `ps2_demo` 的组合模块，内含 PS/2 功能模块，还有数码管基础模块。PS/2 功能模块的左方是 `PS2_CLK` 与 `PS2_DAT` 等顶层信号的输入，右方则是 `oData` 与 `oTag` 联合驱动数码管基础模块。对此，数码管除了输出通码以外，数码管也会表示组合键的有效状态。

ps2_funcmod.v

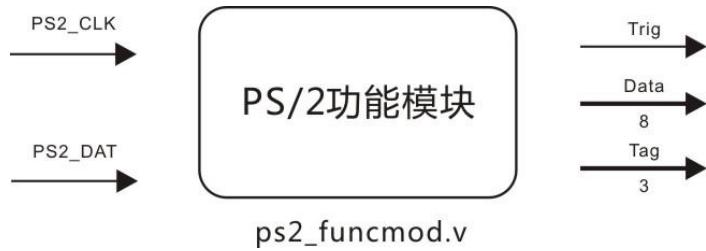


图 8.11 PS/2 功能模块的建模图。

相较图 8.10 与图 8.11 , 图 8.11 的 PS/2 功能模块还有 oTrig , 用来发送 isDone 的高脉冲。至于具体内容如何 , 让我们来瞧瞧代码吧 :

```
1. module ps2_funcmod
2. (
3.     input CLOCK, RESET,
4.     input PS2_CLK, PS2_DAT,
5.     output oTrig,
6.     output [7:0]oData,
7.     output [2:0]oTag
8. );
```

以上内容为出入端声明。

```
9.
10.    parameter LSHIFT = 8'h12, LCTRL = 8'h14, LALT = 8'h11, BREAK = 8'hF0;
11.    parameter FF_Read= 5'd5;
12.
13.    /***** // sub1
14.
15.    reg F2,F1;
16.
17.    always @ ( posedge CLOCK or negedge RESET )
18.        if( !RESET )
19.            { F2,F1 } <= 2'b11;
20.        else
21.            { F2, F1 } <= { F1, PS2_CLK };
22.
23.    /***** // core
24.
25.    wire isH2L = ( F2 == 1'b1 && F1 == 1'b0 );
```

以上内容为常量声明 , 周边操作以及即时声明。第 10 行是 LSHIFT , LCTRL 还有 LALT 等通码的常量声明。此外也有 BREAK 断码第一帧数据 , 还有伪函数的入口(第 11 行)。

第 15~21 行是用来检测电平变化的周边操作，第 25 行则是下降沿的即时声明。

```
26.      reg [7:0]D1;
27.      reg [2:0]isTag; // [2] isShift, [1] isCtrl, [0] isAlt
28.      reg [4:0]i,Go;
29.      reg isDone;
30.
31.      always @ ( posedge CLOCK or negedge RESET )
32.          if( !RESET )
33.              begin
34.                  D1 <= 8'd0;
35.                  isTag <= 3'd0;
36.                  i <= 5'd0;
37.                  Go <= 5'd0;
38.                  isDone <= 1'b0;
39.              end
40.          else
```

以上内容是相关的寄存器声明以及复位操作。期间 isTag 是状态寄存器，isTag[2] 标示 isShift，isTag[1] 标示 isCtrl，isTag[0] 标示 isAlt。第 33~38 行则是这番寄存器的复位操作。

```
65.      **** // PS2 Read Function
66.
67.      5: // Start bit
68.      if( isH2L ) i <= i + 1'b1;
69.
70.      6,7,8,9,10,11,12,13: // Data byte
71.      if( isH2L ) begin i <= i + 1'b1; D1[ i-6 ] <= PS2_DAT; end
72.
73.      14: // Parity bit
74.      if( isH2L ) i <= i + 1'b1;
75.
76.      15: // Stop bit
77.      if( isH2L ) i <= Go;
78.
79.      endcase
```

以上内容为部分核心操作的伪函数。该伪函数读取 PS/2 的 1 帧数据。

```
41.      case( i )
42.
43.          0: // Read Make
```

```

44.          begin i <= FF_Read; Go <= i + 1'b1; end
45.
46.          1: // Set Flag
47.          if( D1 == LSHIFT ) begin isTag[2] <= 1'b1; D1 <= 8'd0; i <= 5'd0;end
48.          else if( D1 == LCTRL ) begin isTag[1] <= 1'b1; D1 <= 8'd0; i <= 5'd0; end
49.          else if( D1 == LALT ) begin isTag[0] <= 1'b1; D1 <= 8'd0; i <= 5'd0; end
50.          else if( D1 == BREAK ) begin i <= FF_Read; Go <= i + 5'd3; end
51.          else begin i <= i + 1'b1; end
52.
53.          2:
54.          begin isDone <= 1'b1; i <= i + 1'b1; end
55.
56.          3:
57.          begin isDone <= 1'b0; i <= 5'd0; end
58.
59.          4: // Clear Flag
60.          if( D1 == LSHIFT ) begin isTag[2] <= 1'b0; D1 <= 8'd0; i <= 5'd0; end
61.          else if( D1 == LCTRL ) begin isTag[1] <= 1'b0; D1 <= 8'd0; i <= 5'd0; end
62.          else if( D1 == LALT ) begin isTag[0] <= 1'b0; D1 <= 8'd0; i <= 5'd0; end
63.          else begin D1 <= 8'd0; i <= 5'd0; end
64.

```

以上内容是核心操作，操作的过程如下：

步骤 0，进入伪函数等待读取通码，并且 Go 指向下一个步骤。

步骤 1，检测组合键与断码，如果是 LShift 那么 isTag[2]立旗，然后返回步骤 0；如果是 LCTRL 那么 isTag[1] 立旗，然后返回步骤 0；如果是 LALT 那么 isTag[0] 立旗，然后返回步骤 0。如果是 BREAK 便进入伪函数，然后 Go 指向步骤 4。如果什么都不不是便进入步骤 2~3。

步骤 2~3，产生完成信号，然后返回步骤 0。

步骤 4，用来消除立旗状态。步骤 1 为 BREAK 便会进入这里，如果断码为 LSHIFT 便会消除 isTag[2]，LCTRL 消除 isTag[1]，LALT 消除 isTag[0]，无视其它断码。最后返回步骤 0。

```

80.
81.      assign oTrig = isDone;
82.      assign oData = D1;
83.      assign oTag = isTag;
84.
85.  endmodule

```

第 81~83 行是输出驱动声明。

ps2_demo.v

笔者在此就不再重复粘贴建模图了，请自行复习图 8.10。

```
1. module ps2_demo
2. (
3.     input CLOCK, RESET,
4.     input PS2_CLK, PS2_DAT,
5.     output [7:0]DIG,
6.     output [5:0]SEL
7. );
8.     wire [7:0]DataU1;
9.     wire [2:0]TagU1;
10.
11.    ps2_funcmod U1
12.    (
13.        .CLOCK( CLOCK ),
14.        .RESET( RESET ),
15.        .PS2_CLK( PS2_CLK ), // < top
16.        .PS2_DAT( PS2_DAT ), // < top
17.        .oTrig(),
18.        .oData( DataU1 ), // > U2
19.        .oTag( TagU1 ) // > U2
20.    );
21.
22.    smg_basemod U2
23.    (
24.        .CLOCK( CLOCK ),
25.        .RESET( RESET ),
26.        .DIG( DIG ), // > top
27.        .SEL( SEL ), // > top
28.        .iData( { 12'h000 , 1'b0, TagU1, DataU1 } ) // < U1
29.    );
30.
31. endmodule
```

基本上，*ps2_demo* 的内容并没有什么难度，所有连线部署都按照图 8.10。至于第 28 行，*DataU1* 还有 *TagU1* 联合驱动数码管基础模块的 *iData*。换句话说，无视数码管的 1~3 位，第 4 位数码管显示组合键状态，第 5~6 位数码管则显示通码。

编译完后便下载程序。如果同时按下 $<\text{LShift}> + <\text{LCtrl}> + <\text{LAlt}>$ ，第 4 位数码管便会显示 $4'h7$ ，亦即 $4'b0111$ ，或者说 $\text{isTag}[2..0]$ 皆为立旗状态。如果按下其它按键，如 $<\text{A}>$ ，那么第 5~6 位的数码管便会显示 $8'h1C$ 。假设释放 $<\text{LShift}>$ ，第 4 位数码管便会显示 $4'h3$ ，亦即 $4'b0011$ ，或者说 $\text{isTag}[1..0]$ 皆为立旗状态。释放 $<\text{A}>$ ，第 5~6 位数码管则会显示 $8'h00$ 。

细节一：完整的个体模块

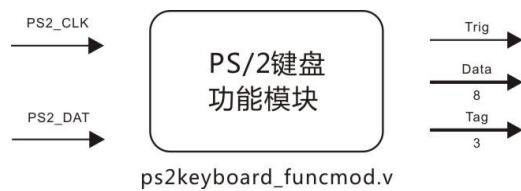


图 8.12 PS/2 键盘功能模块。

图 8.12 是 PS/2 键盘功能模块，内容基本上与 PS/2 功能模块一模一样，至于区别就是穿上其它马甲而已，所以怒笔者不再重复粘贴了。

实验九：PS/2 模块③ — 键盘与多组合键

笔者曾经说过，通码除了单字节以外，也有双字节通码，而且双字节通码都是 8'hE0 开头，别名又是 E0 按键。常见的的 E0 按键有， $\langle\uparrow\rangle$, $\langle\downarrow\rangle$, $\langle\leftarrow\rangle$, $\langle\rightarrow\rangle$, $\langle\text{HOME}\rangle$, $\langle\text{PRTSC}\rangle$ 等编辑键。除此之外，一些组合键也是 E0 按键，例如 $\langle\text{RCtrl}\rangle$ 或者 $\langle\text{RAlt}\rangle$ 。所以说，当我们设计组合键的时候，除了考虑“左边”的组合键以外，我们也要考虑“右边”的组合键。 $\langle\text{Ctrl}\rangle$ 为例：

```
<LCtrl> 通码是 8'h14 ;  
<RCtrl> 通码则是 8'hE0 8'h14。
```

E0 按键除了通码携带 8'hE0 字节以外，E0 按键的断码同样也会携带 8'hE0 字节。 $\langle\text{Ctrl}\rangle$ 继续为例：

```
<LCtrl> 断码是 8'hF0 8'h14 ;  
<RCtrl> 断码是 8'hE0 8'hF0 8'h14。
```

至于时序方面呢 ...

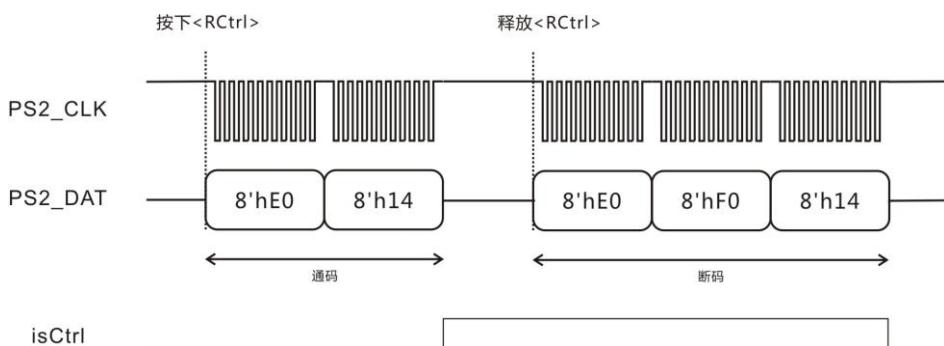


图 9.1 含有 E0 的通码与断码。

如图 9.1 所示，当笔者按下 $\langle\text{RCtrl}\rangle$ ，紧接着 PS/2 键盘会发送 8'hE0 8'h14 的通码，完后 isCtrl 立旗。假设笔者立即释放 $\langle\text{RCtrl}\rangle$ ，那么 PS/2 键盘会发送 8'hE0 8'hF0 8'h14 的断码，事后 isCtrl 就会消除立旗状态。

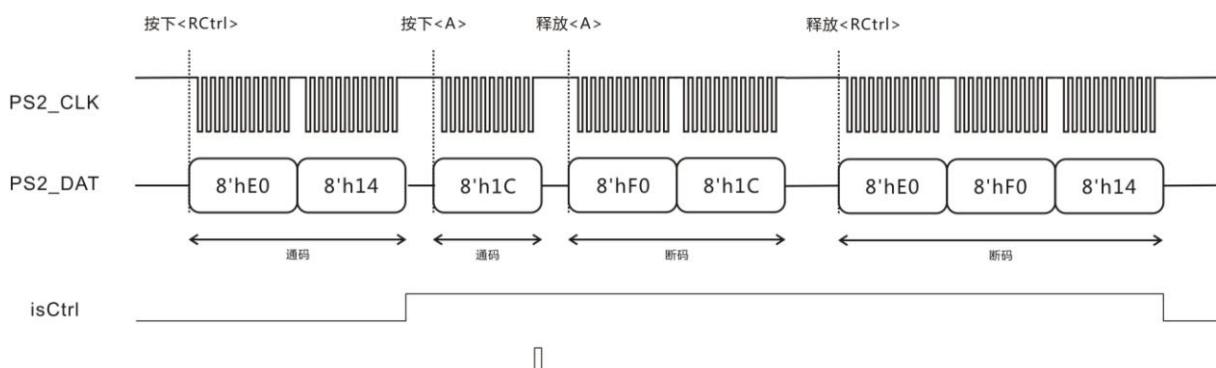


图 9.2 E0 按键与组合键①。

假设笔者按下 $\langle RCtrl \rangle$ 又按下 $\langle A \rangle$ ，那么 $\langle RCtrl \rangle$ 通码会导致 $isCtrl$ 立旗， $\langle A \rangle$ 通码则会导致 $isDone$ 产生高脉冲，此刻组合键 $\langle Ctrl \rangle + \langle A \rangle$ 完成。假设笔者手痒，先释放 $\langle A \rangle$ 再释放 $\langle RCtrl \rangle$ ， $\langle A \rangle$ 断码没有异常，反之 $\langle RCtrl \rangle$ 断码则会消除 $isCtrl$ 的立旗状态。

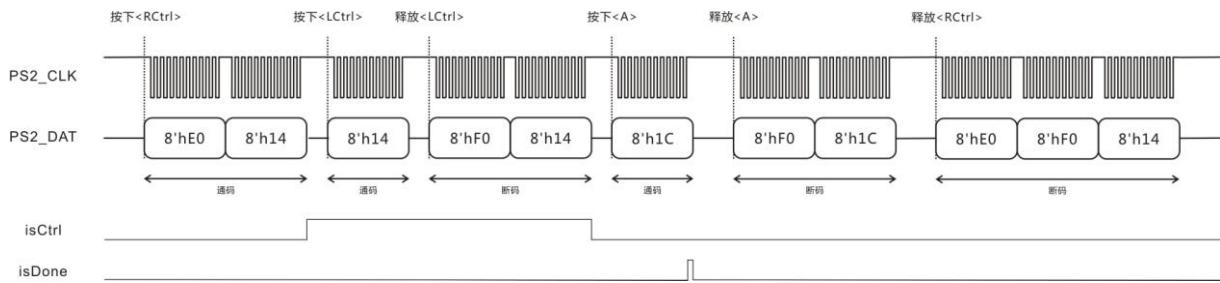


图 9.3 E0 按键与组合键②。

假设顽皮的笔者先按下 $\langle RCtrl \rangle$ 又按下 $\langle LCtrl \rangle$ 然后释放 $\langle LCtrl \rangle$ 。首先 $\langle RCtrl \rangle$ 通码会导致 $isCtrl$ 立旗，不过 $\langle LCtrl \rangle$ 通码会驱使 $isCtrl$ 重复立旗，但是 $\langle LCtrl \rangle$ 断码则会消除 $isCtrl$ 的立旗状态。如果此刻笔者按下 $\langle A \rangle$ ，虽然 $\langle A \rangle$ 通码使产生 $isDone$ 的高脉冲，但是组合键 $\langle Ctrl \rangle + \langle A \rangle$ 则没有成立。心灰意冷的笔者，于是便释放 $\langle A \rangle$ 又释放 $\langle RCtrl \rangle$ ，期间 $\langle A \rangle$ 断码与 $\langle RCtrl \rangle$ 断码都没有异样。

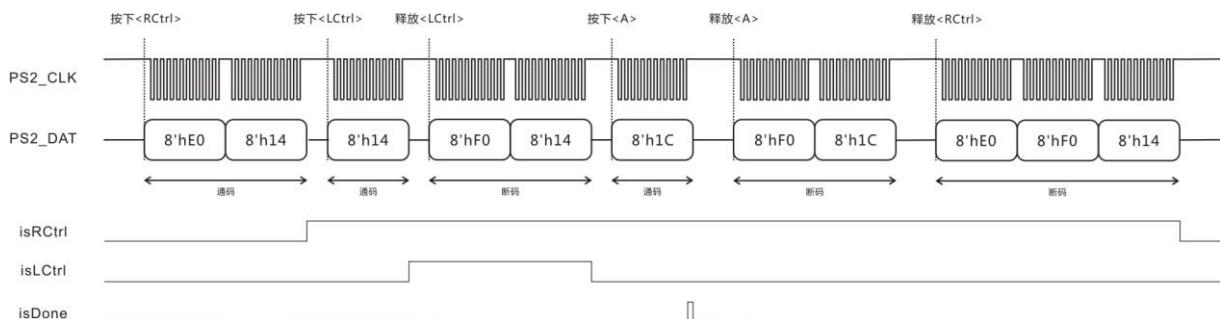


图 9.4 E0 按键与组合键③。

为了解决这个问题，我们必须把 $isCtrl$ 旗标区分为 $isLCtrl$ 与 $isRCtrl$ 为两种旗标。如图 9.4 所示，同样的按键过程，不过却有不同的按键结果。期间， $\langle RCtrl \rangle$ 通码立旗 $isRCtrl$ ，换之 $\langle LCtrl \rangle$ 通码立旗 $isLCtrl$ 。虽然 $\langle LCtrl \rangle$ 断码消除 $isLCtrl$ 的立旗状态，但是 $\langle A \rangle$ 通码还有 $isRCtrl$ 立旗因为合作无间，结果造就组合键 $\langle Ctrl \rangle + \langle A \rangle$ 完成。事后 $\langle RCtrl \rangle$ 断码再消除 $isRCtrl$ 的立旗状态。

为此，我们 $isLCtrl$ 与 $isRCtrl$ 之间的关系可以这样表示：

```
wire isCtrl = isLCtrl | isRCtrl;
```

除此之外， $isLShift$, $isRShift$, $isLAlt$ 与 $isRAlt$ 也是同样的道理。

我们虽然已经解决 E0 按键还有组合键之间的问题，但是还有根本性的问题在等待我们。实验七~八之际，解读一帧数据，数据要么就是通码，数据要么就是断码 … 换句话说，检测数据的时候，我们只要检测 1×2 等两种可能性而已，即 $8'hF0$ 或者非 $8'hF0$ 。如果数据是 $8'hF0$ ，那么数据就是断码，否则就是通码。

一旦 E0 按键乱入搅局，检测的可能性也从原来的 1×2 等两种可能性，变成 $1 \times 2 \times 3$ 等 8 种可能性，这个事实无疑会加剧 Verilog 的描述难度。简言之就是实验七，还有实验八的思路却不适合实验九，为此我们需要更换一下思路。

假设实验九所针对的组合键有：

```
<LShift> 与 <RShift>
<LCtrl> 与 <RCtrl>
<LAlt> 与 <RAlt>
```

然后，我们必须事先考虑所有可能性，包括这些组合键的通码与断码，然后用常量表达出来，结果如代码 9.1 所示：

```
parameter MLSHIFT = 24'h00_00_12, MLCTRL = 24'h00_00_14, MLALT = 24'h00_00_11;
parameter BLSHIFT = 24'h00_F0_12, BLCTRL = 24'h00_F0_14, BLALT = 24'h00_F0_11;
parameter MRSIFHT = 24'h00_00_59, MRCTRL = 24'hE0_00_14, MRALT = 24'hE0_00_11;
parameter BRSHIFT = 24'h00_F0_59, BRCTRL = 24'hE0_F0_14, BRAILT = 24'hE0_F0_11;
```

代码 9.1

如代码 9.1 所示，M \times x 表示通码，B \times x 表示断码 … 如果算计字节 $8'hF0$ 与 $8'hE0$ ，所有组合键的通码与断码都可以使用 3 字节来表达。期间<RCtrl> 与 <RAlt> 的通码与断码都有 $8'hE0$ 的字眼。此外，我们知道低级建模 II 是追求表达能力的建模技巧，凡事力求直观 … 为此，我们必须建立 3 个步骤，而且每个步骤处理单一情况，结果如代码 9.2 所示：

```
1.    1: // E0_xx_xx & E0_F0_xx Check
2.    if( T == 8'hE0 ) begin D1[23:16] <= T; i <= FF_Read; Go <= i; end
3.    else if( D1[23:16] == 8'hE0 && T == 8'hF0 ) begin D1[15:8] <= T; i <= FF_Read; Go <= i; end
4.    else if( D1[23:8] == 16'hE0_F0 ) begin D1[7:0] <= T; i <= CLEAR; end
5.    else if( D1[23:16] == 8'hE0 && T != 8'hF0 ) begin D1[15:0] <= {8'd0, T}; i <= SET; end
6.    else i <= i + 1'b1;
7.
8.    2: // 00_F0_xx Check
9.    if( T == BREAK ) begin D1[23:8] <= {8'd0,T}; i <= FF_Read; Go <= i; end
10.   else if( D1[23:8] == 16'h00_F0 ) begin D1[7:0] <= T; i <= CLEAR; end
11.   else i <= i + 1'b1;
12.
13.   3: // 00_00_xx Check
```

```
14.      begin D1 <= {16'd0,T}; i <= SET; end
```

代码 9.2

如代码 9.2 所示：

步骤 1 处理 E0_××_×× 或者 E0_F0_××，亦即针对 E0 通码与 E0 断码。

步骤 2 处理 00_F0_××，亦即针对一般断码。

步骤 3 处理 00_00_××，亦即针对一般通码。

接下来，让让我们详细理解一下各个步骤的内容：

步骤 1：

第 2 行 if(T == 8'hE0) 表示，如果第一字节是 8'hE0 便将 8'hE0 暂存在 D1[23:16]，即 E0 按键，然后步骤指向伪函数读取第二字节，并且 Go 返回当前步骤。

第 3 行 if(D1[23:16] == 8'hE0 && T == 8'hF0) 表示，如果 D1[23:16] 的内容是 8'hE0 并且第二字节是 8'hF0，即 E0 断码。为此，F0 暂存在 D1[15:8]，然后步骤指向伪函数读取第三字节，Go 则返回当前步骤。

第 4 行 if(D1[23:8] == 16'hE0_F0) 表示，如果 D1[23 :8] 为 16'hE0_F0，那么第三字节也是断码。为此，D1[7:0] 暂存第三字节，然后步骤指向 Clear (消除步骤)。

第 5 行 if(D1[23:16] == 8'hE0 && T != 8'hF0) 表示，D1[23:16] 为 8'hE0，但是第二字节不是 8'hF0，即 E0 通码。为此，D1[16:8] 赋值 8'h00，D1[7:0] 则暂存第二字节，然后步骤指向 SET (设置步骤)。

第 6 行，当什么都不是则表示对象不是 E0 按键，i 递增以示下一个步骤。

步骤 2：

第 9 行，if(T == BREAK) 表示，第一字节为 8'hF0，即是一般断码。为此，D1[23:18] 赋值 8'h00，D1[16:8] 则暂存 8'hF0，然后 i 指向伪函数读取第二字节，Go 返回当前步骤。

第 10 行，if(D1[23:8] == 16'h00_F0) 表示，第一字节 8'hF0 已经读取完毕，现正准备断码的后续字节。为此，D1[7:0] 暂存第二字节，然后 i 指向 Clear (消除步骤)。

第 11 行，当什么都是则表示对象只是一般通码而已。

步骤 3：

第 14 行，D1[23:8] 赋值 16'h00_00 然后 D1[7:0] 暂存第一字节，然后 i 指向 SET (设置步骤)。

```

1.    4: // Set state
2.    if( D1 == MRSIFHT ) begin isTag[5] <= 1'b1; D1 <= 24'd0; i <= 5'd0; end
3.    else if( D1 == MRCTRL ) begin isTag[4] <= 1'b1; D1 <= 24'd0; i <= 5'd0; end
4.    else if( D1 == MRALT ) begin isTag[3] <= 1'b1; D1 <= 24'd0; i <= 5'd0; end
5.    else if( D1 == MLSIFHT ) begin isTag[2] <= 1'b1; D1 <= 24'd0; i <= 5'd0; end
6.    else if( D1 == MLCTRL ) begin isTag[1] <= 1'b1; D1 <= 24'd0; i <= 5'd0; end
7.    else if( D1 == MLALT ) begin isTag[0] <= 1'b1; D1 <= 24'd0; i <= 5'd0; end
8.    else i <= DONE;
9.
10.   5: // Clear state
11.    if( D1 == BRSHIFT ) begin isTag[5] <= 1'b0; D1 <= 24'd0; i <= 5'd0; end
12.    else if( D1 == BRCCTRL ) begin isTag[4] <= 1'b0; D1 <= 24'd0; i <= 5'd0; end
13.    else if( D1 == BRAALT ) begin isTag[3] <= 1'b0; D1 <= 24'd0; i <= 5'd0; end
14.    else if( D1 == BLSHIFT ) begin isTag[2] <= 1'b0; D1 <= 24'd0; i <= 5'd0; end
15.    else if( D1 == BLCTRL ) begin isTag[1] <= 1'b0; D1 <= 24'd0; i <= 5'd0; end
16.    else if( D1 == BLALT ) begin isTag[0] <= 1'b0; D1 <= 24'd0; i <= 5'd0; end
17.    else begin D1 <= 24'd0; i <= 5'd0; end

```

代码 9.3

当第一至第三字节经由步骤 1~3 分析并且整理完毕以后，就会路由步骤 4 (SET) 或者步骤 5 (CLEAR)。

步骤 4 ,第 2~7 行是用来立旗 ,如果 D1 的内容是组合键 ,那么相关的标志位 isTag[n] 就会立旗 ,D1 清空 ,然后 i 返回步骤 0 ;否则 ,对象只是一般字符按键的通码而已 ,结果 i 指向 DONE 并且产生完成信号 (第 8 行)。

步骤 5 第 11~16 行是用来消除立旗 如果 D 的内容是组合键 那么相关的标志位 isTag[n] 就会被消除 ,D1 清空 ,i 返回步骤 0 。否则 ,对象只是一般的字符按键的断码而已 ,结果 D1 清零 ,i 则返回步骤 0 。

理解这些内容以后 ,我们就可以开始建模了。

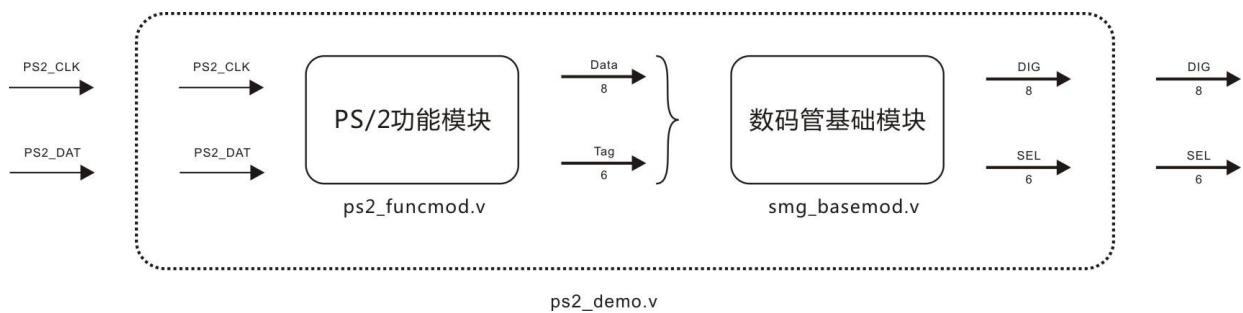


图 9.5 实验九的建模图。

图 9.5 是实验九的建模图，相较实验八，PS/2 功能模块的 oTag 则多了 3 个状态，余下都一样。

ps2_funcmod.v

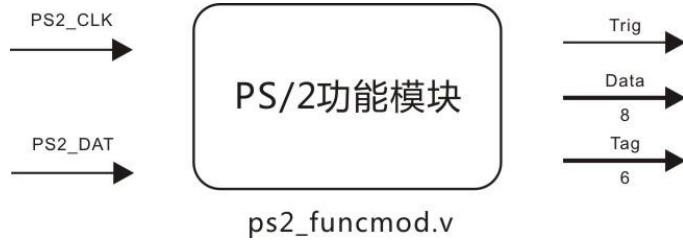


图 9.6 PS/2 功能模块的建模图。

同样，PS/2 功能模块相较实验八，oTag 增多了 3 个位宽，此外内容也发生不少改变。

```
1. module ps2_funcmod
2. (
3.     input CLOCK, RESET,
4.     input PS2_CLK, PS2_DAT,
5.     output oTrig,
6.     output [7:0]oData,
7.     output [5:0]oTag
8. );
```

以上内容为相关的出入端声明。

```
9.     parameter MLSHIFT = 24'h00_00_12, MLCTRL = 24'h00_00_14, MLALT = 24'h00_00_11;
10.    parameter BLSHIFT = 24'h00_F0_12, BLCTRL = 24'h00_F0_14, BLALT = 24'h00_F0_11;
11.    parameter MRS SHIFT = 24'h00_00_59, MRCTRL = 24'hE0_00_14, MRALT = 24'hE0_00_11;
12.    parameter BRSHIFT = 24'h00_F0_59, BRCTRL = 24'hE0_F0_14, BRAILT = 24'hE0_F0_11;
13.    parameter BREAK = 8'hF0;
14.    parameter FF_Read = 5'd8, DONE = 5'd6, SET = 5'd4, CLEAR = 5'd5;
15.
```

以上内容为组合键的常量声明（三字节）。第 13 行是 BREAK 的常量声明。第 14 行是伪函数，SET 步骤与 CLEAR 步骤等入口地址声明。

```
16.     **** // sub1
17.
18.     reg F2,F1;
19.
```

```

20.      always @ ( posedge CLOCK or negedge RESET )
21.          if( !RESET )
22.              { F2,F1 } <= 2'b11;
23.          else
24.              { F2, F1 } <= { F1, PS2_CLK };
25.
26.          //*****
27.
28.      wire isH2L = ( F2 == 1'b1 && F1 == 1'b0 );

```

以上内容是检测电平的周边操作，第 28 行则是下降沿的即时声明。

```

29.      reg [7:0]T;
30.      reg [23:0]D1;
31.      reg [5:0]isTag; // [5]isRShift, [4]isRCtrl, [3]isRAlt, [2]isLShift, [1]isLCtrl, [0]isLAlt;
32.      reg [4:0]i,Go;
33.      reg isDone;
34.
35.      always @ ( posedge CLOCK or negedge RESET )
36.          if( !RESET )
37.              begin
38.                  T <= 8'd0;
39.                  D1 <= 24'd0;
40.                  isTag <= 6'd0;
41.                  i <= 5'd0;
42.                  Go <= 5'd0;
43.                  isDone <= 1'b0;
44.              end
45.          else

```

以上内容是相关寄存器的声明以及复位操作。T 用于伪函数的暂存空间，D1 用来暂存按键数据，isTag 用来标示各个组合按键的状态，i 指向步骤，Go 返回步骤，isDone 则标示有效按键。

```

46.          case( i )
47.
48.              0: // Read Make
49.                  begin i <= FF_Read; Go <= i + 1'b1; end
50.
51.              1: // E0_xx_xx & E0_F0_xx Check
52.                  if( T == 8'hE0 ) begin D1[23:16] <= T; i <= FF_Read; Go <= i; end
53.                  else if( D1[23:16] == 8'hE0 && T == 8'hF0 ) begin D1[15:8] <= T; i <= FF_Read; Go <= i; end

```

```

54.           else if( D1[23:8] == 16'hE0_F0 ) begin D1[7:0] <= T; i <= CLEAR; end
55.           else if( D1[23:16] == 8'hE0 && T != 8'hF0 ) begin D1[15:0] <= {8'd0, T}; i <= SET; end
56.           else i <= i + 1'b1;
57.
58.           2: // 00_F0_xx Check
59.           if( T == BREAK ) begin D1[23:8] <= {8'd0,T}; i <= FF_Read; Go <= i; end
60.           else if( D1[23:8] == 16'h00_F0 ) begin D1[7:0] <= T; i <= CLEAR; end
61.           else i <= i + 1'b1;
62.
63.           3: // 00_00_xx Check
64.           begin D1 <= {16'd0,T}; i <= SET; end
65.

```

以上内容为部分核心操作，过程如下：

步骤 0，进入伪函数以读取第一字节数据。
步骤 1 处理 E0_××_×× 或者 E0_F0_××，亦即针对 E0 通码与 E0 断码。
步骤 2 处理 00_F0_××，亦即针对一般断码。
步骤 3 处理 00_00_××，亦即针对一般通码。

```

66.           4: // Set state
67.           if( D1 == MRSIFFT ) begin isTag[5] <= 1'b1; D1 <= 24'd0; i <= 5'd0; end
68.           else if( D1 == MRCTRL ) begin isTag[4] <= 1'b1; D1 <= 24'd0; i <= 5'd0; end
69.           else if( D1 == MRALT ) begin isTag[3] <= 1'b1; D1 <= 24'd0; i <= 5'd0; end
70.           else if( D1 == MLSIFFT ) begin isTag[2] <= 1'b1; D1 <= 24'd0; i <= 5'd0; end
71.           else if( D1 == MLCTRL ) begin isTag[1] <= 1'b1; D1 <= 24'd0; i <= 5'd0; end
72.           else if( D1 == MLALT ) begin isTag[0] <= 1'b1; D1 <= 24'd0; i <= 5'd0; end
73.           else i <= DONE;
74.
75.           5: // Clear state
76.           if( D1 == BRSHIFT ) begin isTag[5] <= 1'b0; D1 <= 24'd0; i <= 5'd0; end
77.           else if( D1 == BRCCTRL ) begin isTag[4] <= 1'b0; D1 <= 24'd0; i <= 5'd0; end
78.           else if( D1 == BRALT ) begin isTag[3] <= 1'b0; D1 <= 24'd0; i <= 5'd0; end
79.           else if( D1 == BLSHIFT ) begin isTag[2] <= 1'b0; D1 <= 24'd0; i <= 5'd0; end
80.           else if( D1 == BLCTRL ) begin isTag[1] <= 1'b0; D1 <= 24'd0; i <= 5'd0; end
81.           else if( D1 == BLALT ) begin isTag[0] <= 1'b0; D1 <= 24'd0; i <= 5'd0; end
82.           else begin D1 <= 24'd0; i <= 5'd0; end
83.

```

以上内容为部分核心操作，过程如下：

步骤 4，用来立旗组合键。

步骤 5，则用来消除组合键。

```
84.          6: // DONE
85.          begin isDone <= 1'b1; i <= i + 1'b1; end
86.
87.          7:
88.          begin isDone <= 1'b0; i <= 5'd0; end
89.
```

以上内容为部分核心操作，步骤 6~7 用来产生完成信号。

```
90.          /***** // PS2 Read Function
91.
92.          8: // Start bit
93.          if( isH2L ) i <= i + 1'b1;
94.
95.          9,10,11,12,13,14,15,16: // Data byte
96.          if( isH2L ) begin i <= i + 1'b1; T[ i-9 ] <= PS2_DAT; end
97.
98.          17: // Parity bit
99.          if( isH2L ) i <= i + 1'b1;
100.
101.         18: // Stop bit
102.         if( isH2L ) i <= Go;
103.
104.         endcase
105.
```

以上内容为部分核心操作。步骤 8~18 是读取一帧数据的伪函数。

```
106.     assign oTrig = isDone;
107.     assign oData = D1[7:0];
108.     assign oTag = isTag;
109.
110. endmodule
```

以上内容是输出驱动声明。

ps2_demo.v

组合模块 ps2_demo 的连线部署请浏览图 9.5。

```
1. module ps2_demo
2. (
3.     input CLOCK, RESET,
4.     input PS2_CLK, PS2_DAT,
5.     output [7:0]DIG,
6.     output [5:0]SEL
7. );
8.     wire [7:0]DataU1;
9.     wire [5:0]TagU1;
10.
11.    ps2_funcmod U1
12.    (
13.        .CLOCK( CLOCK ),
14.        .RESET( RESET ),
15.        .PS2_CLK( PS2_CLK ), // < top
16.        .PS2_DAT( PS2_DAT ), // < top
17.        .oTrig(),
18.        .oData( DataU1 ), // > U2
19.        .oTag( TagU1 ) // > U2
20.    );
21.
22.    smg_basemod U2
23.    (
24.        .CLOCK( CLOCK ),
25.        .RESET( RESET ),
26.        .DIG( DIG ), // > top
27.        .SEL( SEL ), // > top
28.        .iData( { 8'h00, 1'b0, TagU1[5:3], 1'b0, TagU1[2:0], DataU1 } ) // < U1
29.    );
30.
31. endmodule
```

上述代码基本上没有什么难点，除了第 28 行的联合驱动。8'h00 表示无视第 1~2 位的数码管。1'b0 + TagU1[5:3] 表示第 3 位数码管显示 <RCtrl> <RShift> 还有 <RAlt> 等立旗状态。1'b0 + TagU1[2:0] 表示第 4 位数码管显示 <LCtrl> <LShift> 还有 <LAAlt> 等立旗状态。DataU1 则表示第 5~6 位数码管显示通码。

编译完毕便下载程序。如果读者同时按下 $<\text{RCtrl}>$ $<\text{RShift}>$ 还有 $<\text{RAlt}>$ ，第 3 位数码管就会显示 $4'h7$ ，即 $4'b0111$ 。如果同时按下 $<\text{LCtrl}>$ 还有 $<\text{LShift}>$ ，第 4 位数码管就会显示 $4'h6$ ，即 $4'b0110$ 。如果按下 $<\text{A}>$ ，第 5~6 位数码管则会显示 $8'h1C$ 。注意，千万不要太贪心，同时按下 6 个以上的按键，PS/2 键盘会因此而罢工的 ...

本实验结束之前，让我们来聊聊一些八卦 ... 实验九的 PS/2 功能模块虽然支持 E0 按键，但是仅限 E0 的组合键而已。至于那些 E0 编辑键，如 $<\uparrow>$ 或者 $<\downarrow>$ ，则需要进一步扩展，不过该要求已经超出本书的讨论范围。不管对象是 E0 组合键，还是 E0 编辑键，设计思路也是一样的，不过后者比较偏向软件。对此，读者只要基于实验九，再简单扩展一下即可。

细节一：完整的个体模块



图 9.7 PS/2 键盘功能模块。

图 9.7 是 PS/2 键盘功能模块，内容基本上与 PS/2 功能模块一模一样，至于区别就是穿上其它马甲而已，所以怒笔者不再重复粘贴了。

实验十：PS/2 模块④ — 普通鼠标

学习 PS/2 键盘以后，接下来就要学习 PS/2 鼠标。PS/2 鼠标相较 PS/2 键盘，驱动难度稍微高了一点点，因为 FPGA（从机）不仅仅是从 PS/2 鼠标哪里读取数据，FPGA 还要往鼠标里写数据 ... 反之，FPGA 只要对 PS/2 键盘读取数据即可。然而，最伤脑筋的地方就在于 PS/2 传输协议有奇怪的写时序。

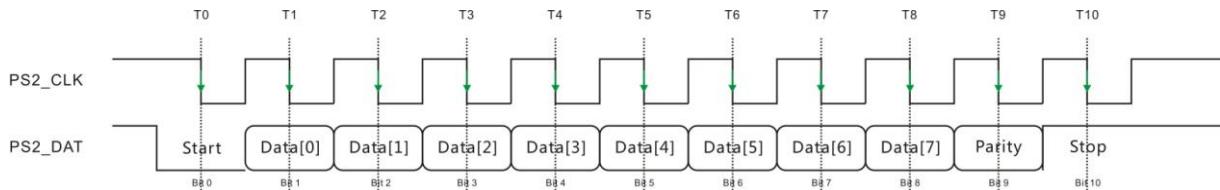


图 10.1 从机视角，从机读数据。

为了方便理解，余下我们经由从机的视角去观察 PS/2 的读写时序。图 10.1 是从机视角的读时序，从机都是皆由 PS2_CLK 的下降沿读取 1 帧为 11 位的数据 ... 期间，有 11 个下降沿，不过为了方便调用，笔者将其整理为伪函数，结果如代码 10.1 所示：

```
1. 32: // Start bit
2. if( isH2L ) i <= i + 1'b1;
3.
4. 33,34,35,36,37,38,39,40: // Data byte
5. if( isH2L ) begin T[i-33] <= PS2_DAT; i <= i + 1'b1; end
6.
7. 41: // Parity bit
8. if( isH2L ) i <= i + 1'b1;
9.
10. 42: // Stop bit
11. if( isH2L ) i <= Go;
```

代码 10.1

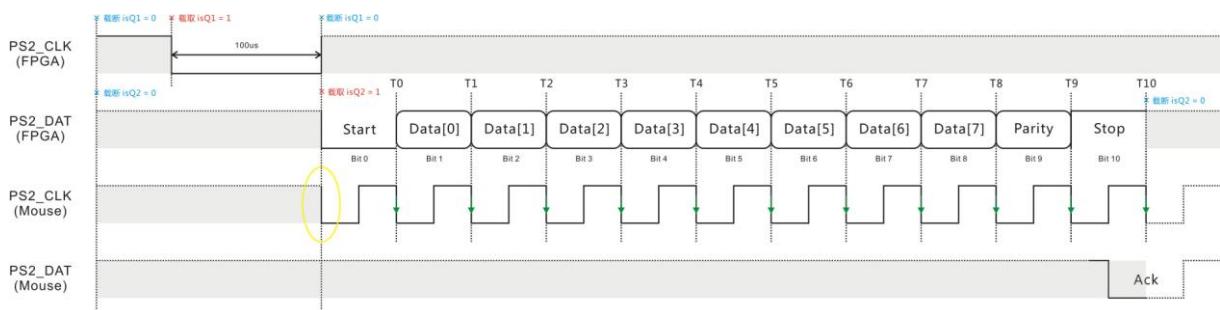


图 10.2 从机视角，从机写数据，第一帧。

首先我们必须明白，PS2_CLK 与 PS2_DAT 信号是双向的 IO 口，由于 FPGA 是从机的关系，所以 FPGA 一开始都处于输入状态，而 PS/2 鼠标一开始则处于输出状态。假设

isQ1 是 FPGA 针对 PS2_CLK 的输出控制 , isQ2 是 FPGA 针对 PS2_DAT 的输出控制 , 然而复位状态 (初始化) 都为拉低状态。

FPGA 为了获取数据的发送权 , 首先它必须摘取 PS2_CLK , 亦即拉高 isQ1 然后又拉低 PS2_CLK 100us。

事后 , FPGA 必须释放 PS2_CLK , 亦即关闭 IO 或者拉低 isQ1 , 期间顺手拉高 isQ2 , 让 PS2_DAT 成为输出状态。那么重点来了 , 读者是否看见黄色的圈圈 ? 当 FPGA 释放 PS2_CLK 瞬间 , PS/2 鼠标早已拉低 PS2_CLK , FPGA 也因此错过 PS2_CLK 第一次珍贵的下降沿。在此 , 读者需要好好注意 , 因为许多资料都忽略这点 , 笔者也不小心扑街好几次。

读者需要知道 , 根据 PS/2 传输协议 , 从机 (FPGA) 不管怎样挣扎也没有 PS2_CLK 的拥有权。换句话说 , 无论从机 (FPGA) 是写数据还是读数据 , 从机 (FPGA) 都必须借助主机 (PS/2 鼠标) 发送过来的 PS2_CLK 下降沿。如果我们不小心忽略第一个下降沿 , 余下几个下降沿 , 我们也会搞错次序 , 最终造成数据读取错位的悲剧。

FPGA 无论是写数据还是读数据 , 从机都必须借用 PS2_CLK 的下降沿。FPGA 释放 PS2_CLK 之后 , isQ2 立即拉高 , PS2_DAT 也随之拉低以示一帧数据的起始位 ... 直至下一个下降沿到来为此。

FPGA 一共用了 10 个下降沿 , 即 T0~T9 发送 8 位数据位 , 1 位校验位 , 还有 1 位停止位。 T9 过去不久 , PS2_CLK 就会引来上升沿 , 此刻 PS/2 鼠标则会反馈 1 位应答位 , 不过此刻也无暇招呼它。 T10 之际 , 那是最后一个下降沿 , FPGA 拉低 isQ2 让 PS2_DAT 成为输入状态 , 并且读取应答位 , 然而懒惰的笔者决定无视它。就这样从机写一帧数据就结束了。

在这里 , 读者是否觉得从机写一帧数据比从机读一帧数据还要麻烦呢 ? 没错 , 笔者也这样觉得。可是 , 麻烦归麻烦 , 余下我们还要完成 Verilog 的描述工作 , 结果如代码 10.2 所示 :

```
1.          **** // PS2 Write Function
2.
3.          32: // Press low CLK 100us
4.          if( C1 == T100US -1 ) begin C1 <= 13'd0; i <= i + 1'b1; end
5.          else begin isQ1 = 1'b1; rCLK <= 1'b0; C1 <= C1 + 1'b1; end
6.
7.          33: // Release PS2_CLK and set in, PS2_DAT set out
8.          begin isQ1 <= 1'b0; rCLK <= 1'b1; isQ2 <= 1'b1; i <= i + 1'b1; end
9.
10.         34: // start bit
11.         begin rDAT <= 1'b0; i <= i + 1'b1; end
12.
```

```

13.           35,36,37,38,39,40,41,42,43: // Data byte
14.           if( isH2L ) begin rDAT <= T[ i-35 ]; i <= i + 1'b1; end
15.
16.           44: // Stop bit
17.           if( isH2L ) begin rDAT <= 1'b1; i <= i + 1'b1; end
18.
19.           45: // Ack bit
20.           if( isH2L ) begin i <= i + 1'b1; end
21.
22.           46: // PS2_DAT set in
23.           begin isQ2 <= 1'b0; i <= i + 1'b1; end
24.           .....
25.
26.   assign PS2_CLK = isQ1 ? rCLK : 1'bz;
27.   assign PS2_DAT = isQ2 ? rDAT : 1'bz;

```

代码 10.2

如代码 10.2 所示：

步骤 32，来拉高 isQ1 又拉低 rCLK 持续 100us。

步骤 33，拉低 isQ1 释放 PS2_CLK，然后又拉高 isQ2 准备发送数据。

步骤 34，重点！由于错过第一个下降沿，我们只能拉低 rDAT 产生起始位。

步骤 35~43，发送 8 位数据位，还有 1 位校验位。

步骤 44，发送结束位。

步骤 45，无视应答位。

步骤 46，拉低 isQ2，让 PS2_CLK 为输入状态。

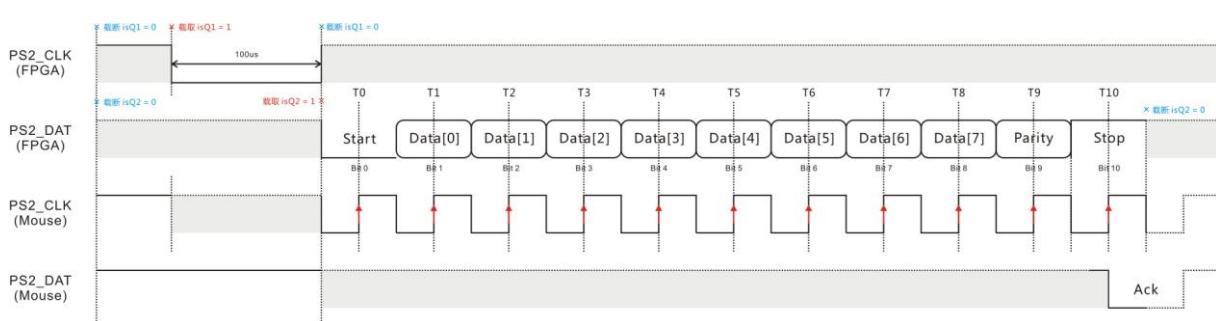


图 10.3 主机视角，从机写数据，第一帧。

接下来，让我们经由主机的视角去观察，主机如何读取从机发送过来的数据。初始状态，主机亦即 PS/2 鼠标掌握 PS2_CLK 与 PS2_DAT。一旦从机即 FPGA 载取 PS2_CLK，并且拉低输出 100us，立刻 PS/2 鼠标已经理解从机准备发送数据。100us 过后，FPGA 释放 PS2_CLK 并且载取 PS2_DAT，PS/2 鼠标便开始产生时钟。如图 10.2 所示，PS/2 鼠标都是借用上升沿读取 FPGA 发送过来的数据。大约 11 个上升沿过后，PS/2 鼠标就结束读取动作，并且反馈应答位。

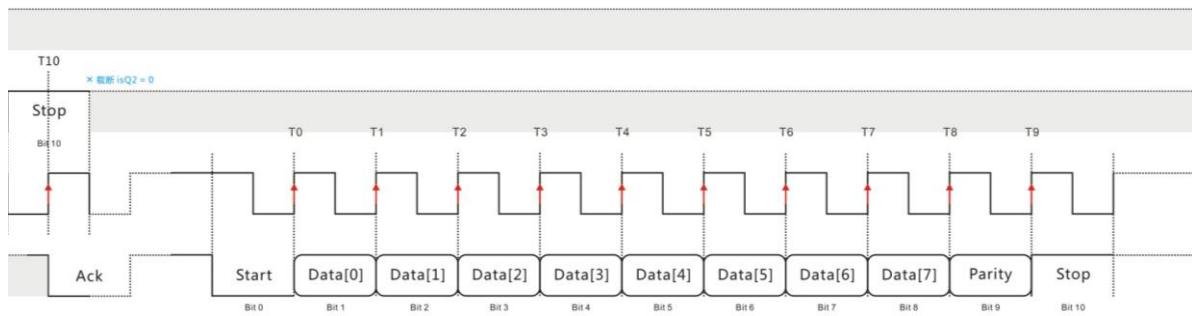


图 10.4 主机视角，从机写数据，第二帧。

每当主机接收完毕一帧数据，就会反馈一帧数据。如图 10.2 所示，那是图 10.1 的下半部分，依然是从主机视角观察从机写数据。图中显示，每当 PS/2 鼠标（主机）接收完毕一帧数据以后，PS/2 鼠标便会反馈一帧数据，亦即 PS/2 鼠标会再度借用 10 个上升沿发送一帧数据。期间，FPGA（从机）的 PS2_CLK 与 PS2_DAT 都都处于输入状态。

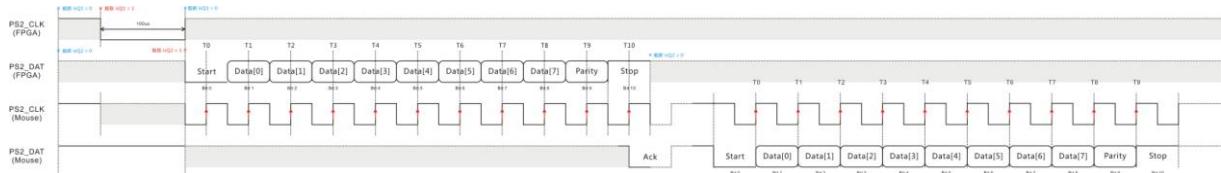


图 10.5 主机视角，从机写数据，完整时序。

图 10.5 是图 10.3 与图 10.4 的完整时序（主机视角）。

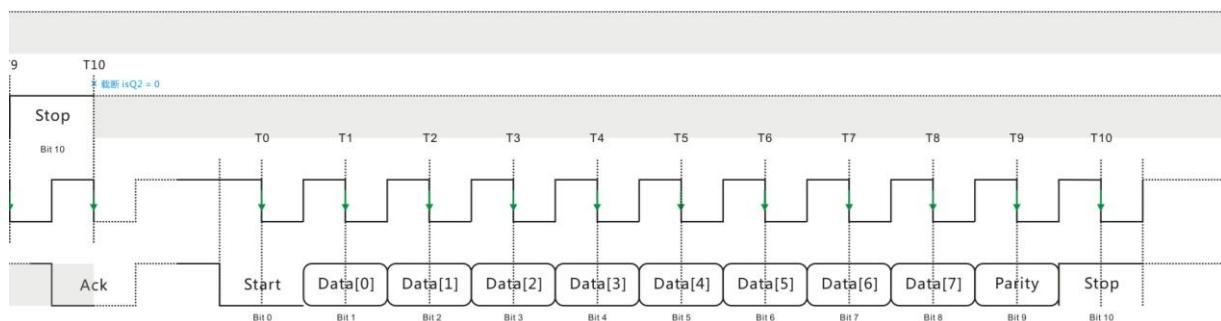


图 10.6 从机视角，从机写数据，第二帧。

既然主机反馈一帧数据，那么从机也不能无视，如图 10.6 所示，那是经由从机视角观察从机如何读取下一帧数据。期间，从机借用 11 个下降沿读取 1 一帧数据。

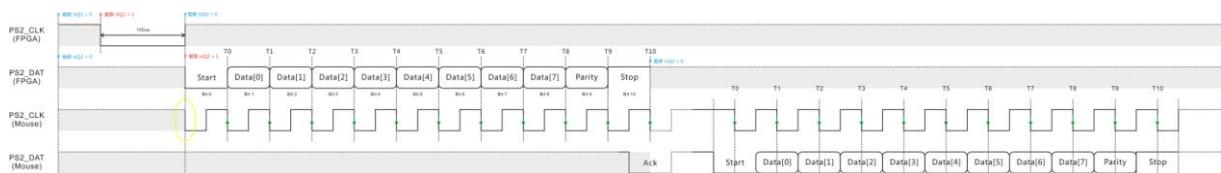


图 10.7 从机视角，从机写数据，完整时序。

图 10.7 是图 10.2 与图 10.5 的完整时序（从机视角）。

为此，代码 10.2 可以继续扩张，结果如代码 10.3 所示：

```
1.      ****// PS2 Write Function
2.
3.      32: // Press low CLK 100us
4.      if( C1 == T100US -1 ) begin C1 <= 13'd0; i <= i + 1'b1; end
5.      else begin isQ1 = 1'b1; rCLK <= 1'b0; C1 <= C1 + 1'b1; end
6.
7.      33: // Release PS2_CLK and set in, PS2_DAT set out
8.      begin isQ1 <= 1'b0; rCLK <= 1'b1; isQ2 <= 1'b1; i <= i + 1'b1; end
9.
10.     34: // start bit
11.     begin rDAT <= 1'b0; i <= i + 1'b1; end
12.
13.     35,36,37,38,39,40,41,42,43: // Data byte
14.     if( isH2L ) begin rDAT <= T[ i-35 ]; i <= i + 1'b1; end
15.
16.     44: // Stop bit
17.     if( isH2L ) begin rDAT <= 1'b1; i <= i + 1'b1; end
18.
19.     45: // Ack bit
20.     if( isH2L ) begin i <= i + 1'b1; end
21.
22.     46: // PS2_DAT set in
23.     begin isQ2 <= 1'b0; i <= i + 1'b1; end
24.
25.     47,48,49,50,51,52,53,54,55,56,57: // 1 Frame
26.     if( isH2L ) i <= i + 1'b1;
27.
28.     58: // Return
29.     i <= Go;
30.
31.     .....
32.
33.     assign PS2_CLK = isQ1 ? rCLK : 1'bz;
34.     assign PS2_DAT = isQ2 ? rDAT : 1'bz;
```

代码 10.3

步骤 32~46 曾在前面说过，即从机发送一帧数据。余下步骤 47~57（共有 11 个步骤），主要用来过滤主机反馈过来的下一帧数据，步骤 58 则是步骤返回。不过为什么步骤 47~57 不是读取数据，而是过滤数据？别着急，答案很快就会揭晓，暂时忍耐一下。

呼！PS/2 传输协议的写数据（从机视角）解释起来真有够呛。最后让我们来总结一下，主机无论是输出数据，还是读取数据都是借用 PS2_CLK 的上升沿。反之，从机无论是输出数据，还是读取数据都是借用 PS2_CLK 的下降沿。PS/2 传输协议是可谓是爷爷级别的传输协议吧，因为近代的传输协议不管对象是从机还是主机，或者是写还是读，一般都是上升沿设置数据下降沿锁存数据。很少情况是主机使用一个时间沿，从机使用另一个时间沿，例如 SPI 传输协议就是最好的例子。

说完 PS/2 传输协议，接下来我们要进入 PS/2 鼠标的主题了。

鼠标也有普通鼠标与滚扩展鼠标之分 … 所谓普通鼠标就是包含左键，中键还有右键；所谓扩展鼠标则包含左键，中键，右键还有滚轮，扩展鼠标也称为滚轮鼠标，不过一般的扩展鼠标只有一个滚轮而已。实验十的实验目的就是驱动普通鼠标。PS/2 鼠标不像 PS/2 键盘，PS/2 鼠标即使一上电也不会立即工作，期间从机必须将它使能才行。

普通鼠标一旦上电就便会立即复位，然后得到默认化参数，并且进入 Steam 模式。所谓 Steam 模式，即位置状况或者按键状况一有变化就会鼠标便会立即发送报告。话虽然那么说，实际上 Stream 模式还要依赖采集频率，默认的采集频率是 100 次/秒，即采集间隔为 10ms，也就是说鼠标在一秒内会检测 100 次位置状况还有按键状况。

举例而言，假设笔者按着左键不放，那么鼠标在一秒内会发送 100 次“左键好疼！左键好疼！”，直至笔者释放左键为止。再假设鼠标不小心被笔者退了一下，然后鼠标在 10ms 内向左移动 10mm，当鼠标察觉位置状况发生变化以后，鼠标便会发送“哎呀！被人推向左边 10mm 了！”。

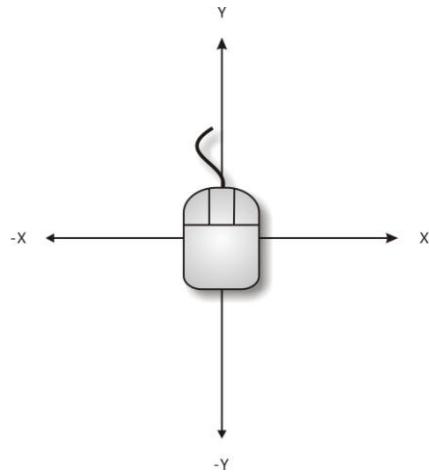


图 10.7.1 鼠标的位置标示。

鼠标为了标示位置，内建 2 组 9 位的寄存器 X 与 Y，结果如图 10.7.1 所示。默认下，鼠标的分辨率为 4 计数/mm。此外，鼠标也有能力辨识 4 处的移动方向，例如左移 10mm 寄存器 X 便计数 -40，右移 10mm 寄存器 X 便计数 +40，上移 10mm 寄存器 Y 便计数

+40，下移 10mm 寄存器 Y 便计数 -40。鼠标每隔 10ms（默认采集频率）便会清零一次寄存器 X 与 Y 的内容。

PS/2 鼠标不像 PS/2 键盘一上电便立即工作，我们必须事先发送命令 8'hF4 即“使能鼠标发送数据”，开启数据的水龙头。每当鼠标接收一帧数据，鼠标便会反馈一帧数据，为此 ... 从机每次向鼠标写入一帧数据，就必须接收一帧反馈数据。反馈数据为 8'hFA 表示“数据接收成功”，反馈数据为 8'hFE 表示“第一帧数据接收失败”，反馈数据为 8'hFC 则表示“第二帧数据接收成功”（有些命令是由 2 帧或者以上组成）。

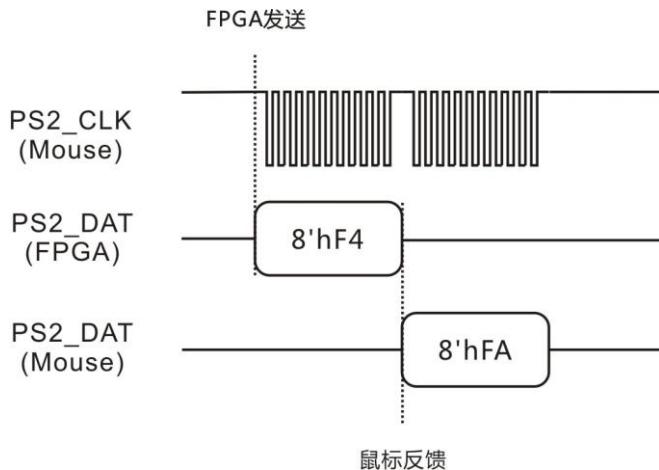


图 10.8 从机发送“使能报告”命令，鼠标反馈接收成功。

为了驱使鼠标工作，PS/2 鼠标上电以后，从机必须发送命令 8'hF4，并且接收反馈 8'hFA。如果一切顺利，那么鼠标就会开始工作，结果如图 10.8 所示。鼠标“使能”以后，鼠标便处于就绪状态，采集便开始 ... 此刻，如果鼠标的位置状况或者按键状况发生变化，鼠标就会发送 3 帧，亦即 3 字节的报告。

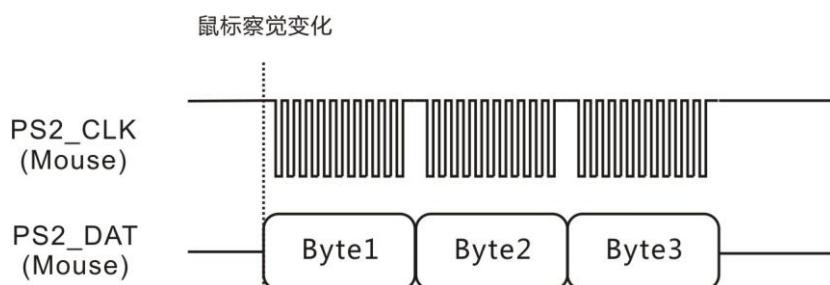


图 10.9 鼠标发送报告。

如图 10.9 所示，那是一份 3 个字节的报告，Verilog 可以这样描述：

```

1.          0: // Read 1st byte
2.          begin i <= FF_Read; Go <= i + 1'b1; end
3.
4.          1: // Store 1st byte
5.          begin D1[7:0] <= T; i <= i + 1'b1; end

```

```

6.
7.          2: // Read 2nd byte
8.          begin i <= FF_Read; Go <= i + 1'b1; end
9.
10.         3: // Store 2nd byte
11.         begin D1[15:8] <= T; i <= i + 1'b1; end
12.
13.         4: // Read 3rd byte
14.         begin i <= FF_Read; Go <= i + 1'b1; end
15.
16.         5: // Store 3rd byte
17.         begin D1[23:16] <= T; i <= i + 1'b1; end
18.
19.         .....
20.
21.         32: // Start bit
22.         if( isH2L ) i <= i + 1'b1;
23.
24.         33,34,35,36,37,38,39,40: // Data byte
25.         if( isH2L ) begin T[i-33] <= PS2_DAT; i <= i + 1'b1; end
26.
27.         41: // Parity bit
28.         if( isH2L ) i <= i + 1'b1;
29.
30.         42: // Stop bit
31.         if( isH2L ) i <= Go;

```

代码 10.4

步骤 32~42 是读一帧数据的伪函数，步骤 0~1 读取第一字节并且暂存道 D[7:0]，步骤 2~3 读取第二字节并且暂存到 D[15:8]，步骤 4~5 读取第三字节并且暂存到 D[23:16]。

至于报告的内容如表 10.1 所示：

表 10.1 普通鼠标的报告。

字节/位	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
字节一	Y 溢出位	X 溢出位	Y[8]符号位	X[8]符号位	保留	中键	右键	左键
字节二	X[7:0]							
字节三	Y[7:0]							

如表 10.1 所示，字节一的第四位表示按键状况以外，字节一的高四位也与内部寄存器 X 与 Y 有关。字节二为寄存器 X 的内容，字节三位寄存器 Y 的内容。

字节一，[0]标示左键，1 表示左键按下；[1]标示右键，1 表示右键按下；[2]标示中键，1 表示中键按下；[4]为字寄存器的最高位也是符号位；[5]为寄存器最高位也是符号位；

节二是寄存器 X 的低八位，字节三是寄存器 Y 的低八位，因此寄存器 X 与 Y 的位宽为 9。这样作的目的是为了使用补码表示鼠标的移动状况。至于补码是什么？失忆的朋友请复习《时序篇》。

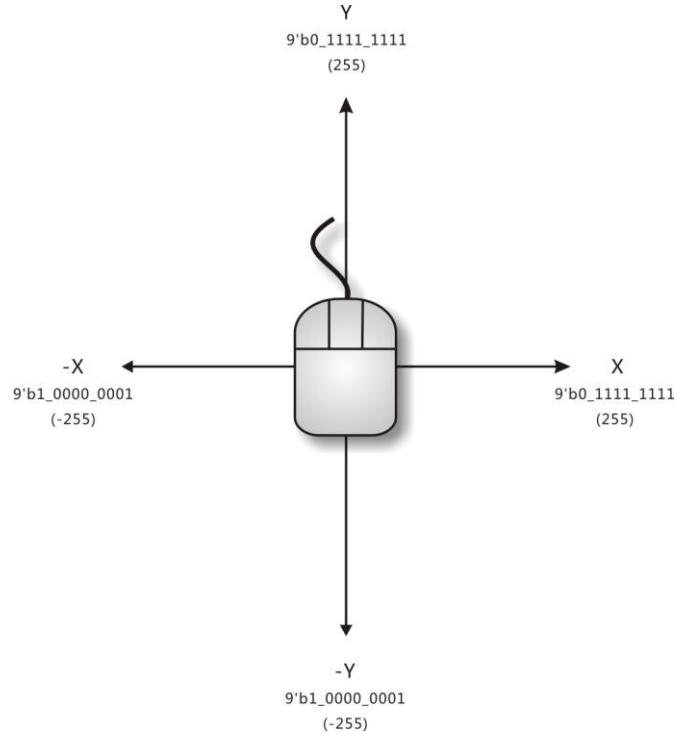


图 10.9.1 鼠标的有效位置。

假设寄存器 X 的内容为 $9'b1_1111_1100$ ，[8]为 1'b1 表示鼠标正在左移，[7:0]为 8'b1111_1100 也是 8 个计数，亦即移动 2mm 的距离。因此 $9'b1_1111_1100$ 表示鼠标左移 2mm 的举例。再假设寄存器 Y 的内容为 $9'b0_0001_0000$ ，[8]为 0 表示鼠标正在上移，[7:0] 为 8'b0001_0000 也是 16 个计数，亦即移动 4mm。因此 $9'b0_0001_0000$ 表示鼠标上移 4mm。结果如图 10.9.1 所示。

上述内容理解完毕以后，我们便可以开始建模了。

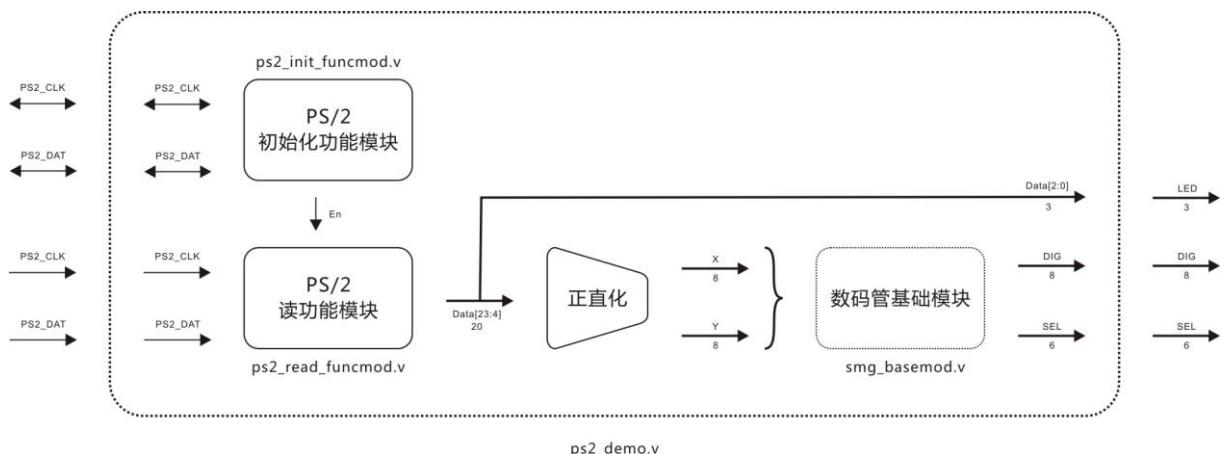


图 10.10 实验十的建模图。

图 10.10 是实验十的建模图，组合模块 ps2_demo 内部包含，PS/2 初始化功能模块，PS/2 读功能模块，数码管基础模块，然后中间还要正直化的即时操作。顾名思义，PS/2 初始化功能模块主要负责初始化的工作，简言之就是发送命令 8'hF4，完后便拉高 oEn 使能 PS/2 读功能模块。PS/2 读功能模块接收 iEn 拉高便会开始读取 3 字节的报告，并且经由 oData 将其输出。

稍微注意一下 PS2_CLK 还有 PS2_DAT 顶层信号，由于 PS/2 初始化功能模块需要双向访问 PS/2 鼠标，为此该顶层信号皆是出入状态（IO）。反之，PS/2 读功能模块只有接收数据而已，因此该顶层信号只是出入状态。PS/2 读功能模块的 oData，其中[2:0]是 3 只按键的状况，并且直接驱动三位 LED 资源。

至于[23:4]则是寄存器 X 与寄存器 Y 的内容，它们经由即时操作正直化以后便联合驱动数码管基础模块，然后再显示内容。

ps2_init_funcmod.v

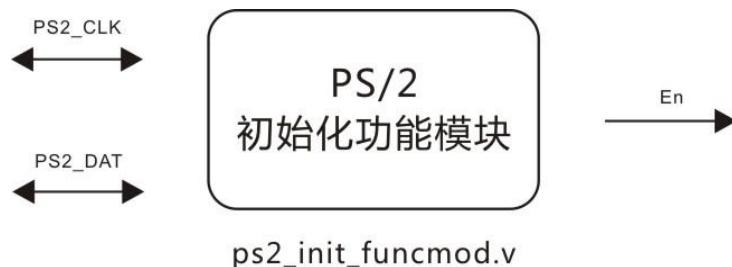


图 10.11 PS/2 初始化功能模块的建模图。

由于该模块需要来问读写 PS/2 鼠标，因此顶层信号 PS2_CLK 与 PS2_DAT 都是双向，亦即 IO 口。此外，一旦该模块完成初始化的工作，oEn 就会一直拉高。

```

1. module ps2_init_funcmod
2. (
3.     input CLOCK, RESET,
4.     inout PS2_CLK,
5.     inout PS2_DAT,
6.     output oEn
7. );
8. parameter T100US = 13'd5000;
9. parameter FF_Write = 7'd32;
10.

```

以上内容是出入端声明。第 8 行是 100us 的常量声明，第 9 行则是伪函数的入口。

```

11.      ****// sub1
12.
13.      reg F2,F1;
14.
15.      always @ ( posedge CLOCK or negedge RESET )
16.          if( !RESET )
17.              { F2,F1 } <= 2'b11;
18.          else
19.              { F2, F1 } <= { F1, PS2_CLK };
20.
21.      ****// Core
22.
23.      wire isH2L = ( F2 == 1'b1 && F1 == 1'b0 );

```

以上是检测电平变化的周边操作。第 23 行则是下降沿的即时声明。

```

24.      reg [8:0]T;
25.      reg [6:0]i,Go;
26.      reg [12:0]C1;
27.      reg rCLK,rDAT;
28.      reg isQ1,isQ2,isEn;
29.
30.      always @ ( posedge CLOCK or negedge RESET )
31.          if( !RESET )
32.              begin
33.                  T <= 9'd0;
34.                  C1 <= 13'd0;
35.                  { i,Go } <= { 7'd0,7'd0 };
36.                  { rCLK,rDAT } <= 2'b11;
37.                  { isQ1,isQ2,isEn } <= 3'b000;
38.              end
39.          else

```

以上内容为相关的寄存器声明以及复位操作。注意，PS2_CLK 与 PS2_DAT 默认下都是高电平，所示 rCLK 与 rDAT 被赋予复位值 2'b11。

```

40.          case( i )
41.
42.              ****// INIT Normal Mouse
43.
44.                  0: // Send F4 1111_0100
45.                  begin T <= { 1'b0, 8'hF4 }; i <= FF_Write; Go <= i + 1'b1; end
46.

```

```
47.           1:  
48.           isEn <= 1'b1;  
49.
```

以上内容是部分核心操作。步骤 0~1 是主操作，主要发送命令 8'hF4，然后拉高 isEn。第 45 行{ 1'b0, 8'hF4 }，其中 1'b0 是校验位，PS/2 的校验位是“奇校验”，如果“1”的数量为单数，那么校验位便是 0。如第 45 所示，8'hF4 有 5 个“1”所示，校验位为 0。

```
50.           **** // PS2 Write Function  
51.  
52.           32: // Press low CLK 100us  
53.           if( C1 == T100US -1 ) begin C1 <= 13'd0; i <= i + 1'b1; end  
54.           else begin isQ1 = 1'b1; rCLK <= 1'b0; C1 <= C1 + 1'b1; end  
55.  
56.           33: // Release PS2_CLK and set in, PS2_DAT set out  
57.           begin isQ1 <= 1'b0; rCLK <= 1'b1; isQ2 <= 1'b1; i <= i + 1'b1; end  
58.  
59.           34: // start bit  
60.           begin rDAT <= 1'b0; i <= i + 1'b1; end  
61.  
62.           35,36,37,38,39,40,41,42,43: // Data byte  
63.           if( isH2L ) begin rDAT <= T[ i-35 ]; i <= i + 1'b1; end  
64.  
65.           44: // Stop bit  
66.           if( isH2L ) begin rDAT <= 1'b1; i <= i + 1'b1; end  
67.  
68.           45: // Ack bit  
69.           if( isH2L ) begin i <= i + 1'b1; end  
70.  
71.           46: // PS2_DAT set in  
72.           begin isQ2 <= 1'b0; i <= i + 1'b1; end  
73.  
74.           47,48,49,50,51,52,53,54,55,56,57: // 1 Frame  
75.           if( isH2L ) i <= i + 1'b1;  
76.  
77.           58: // Return  
78.           i <= Go;  
79.  
80.           endcase
```

以上内容是部分核心操作。第 32~58 行是从机写一帧数据，读一帧反馈数据的伪函数。

```
81.
```

```

82.      assign PS2_CLK = isQ1 ? rCLK : 1'bz;
83.      assign PS2_DAT = isQ2 ? rDAT : 1'bz;
84.      assign oEn = isEn;
85.
86. endmodule

```

以上内容是输出驱动声明。

ps2_read_funcmod.v



图 10.12 PS/2 读化功能模块的建模图。

PS/2 读功能模块，如果 iEn 不拉高就不工作。此外，该模块也只是读入 3 字节的报告而已，完后便经由 oTrig 产生完成信号，报告内容则经由 oData。

```

1. module ps2_read_funcmod
2. (
3.     input CLOCK, RESET,
4.     input PS2_CLK,PS2_DAT,
5.     input iEn,
6.     output oTrig,
7.     output [23:0]oData
8. );
9. parameter FF_Read = 7'd32;
10.

```

以上内容是出入端声明。第 9 行则是伪函数的入口地址。

```

11.     ****// sub1
12.
13.     reg F2,F1;
14.
15.     always @ ( posedge CLOCK or negedge RESET )
16.         if( !RESET )
17.             { F2,F1 } <= 2'b11;
18.         else

```

```

19.          { F2, F1 } <= { F1, PS2_CLK };
20.
21.      ****// core
22.
23.      wire isH2L = ( F2 == 1'b1 && F1 == 1'b0 );

```

以上内容是检测电平变化的周边操作，第 23 行则是下降沿的即时声明。

```

24.      reg [23:0]D1;
25.      reg [7:0]T;
26.      reg [6:0]i,Go;
27.      reg isDone;
28.
29.      always @ ( posedge CLOCK or negedge RESET )
30.          if( !RESET )
31.              begin
32.                  D1 <= 24'd0;
33.                  T <= 8'd0;
34.                  { i,Go } <= { 7'd0,7'd0 };
35.                  isDone <= 1'b0;
36.              end

```

以上内容是相关的寄存器声明以及复位操作。

```

37.      else if( iEn )
38.          case( i )
39.
40.              ****// Normal mouse
41.
42.                  0: // Read 1st byte
43.                      begin i <= FF_Read; Go <= i + 1'b1; end
44.
45.                  1: // Store 1st byte
46.                      begin D1[7:0] <= T; i <= i + 1'b1; end
47.
48.                  2: // Read 2nd byte
49.                      begin i <= FF_Read; Go <= i + 1'b1; end
50.
51.                  3: // Store 2nd byte
52.                      begin D1[15:8] <= T; i <= i + 1'b1; end
53.
54.                  4: // Read 3rd byte
55.                      begin i <= FF_Read; Go <= i + 1'b1; end

```

```
56.  
57.          5: // Store 3rd byte  
58.          begin D1[23:16] <= T; i <= i + 1'b1; end  
59.  
60.          6:  
61.          begin isDone <= 1'b1; i <= i + 1'b1; end  
62.  
63.          7:  
64.          begin isDone <= 1'b0; i <= 7'd0; end  
65.
```

以上内容为部分核心操作。第 37 行 if(iEn) 表示，iEn 不拉高核心操作就不运行。步骤 0~1 是读取第一字节，步骤 2~3 是读取第二字节，步骤 4~5 是读取第三字节，步骤 6~7 则是反馈完成信号，以示一次性的报告读取已经完成。完后，i 便指向步骤 0。

```
66.          /***** // PS2 Write Function  
67.  
68.          32: // Start bit  
69.          if( isH2L ) i <= i + 1'b1;  
70.  
71.          33,34,35,36,37,38,39,40: // Data byte  
72.          if( isH2L ) begin T[i-33] <= PS2_DAT; i <= i + 1'b1; end  
73.  
74.          41: // Parity bit  
75.          if( isH2L ) i <= i + 1'b1;  
76.  
77.          42: // Stop bit  
78.          if( isH2L ) i <= Go;  
79.  
80.      endcase
```

以上内容为部分核心操作。步骤 32~42 则是伪函数，主要是负责读取一帧数据。

```
81.  
82.      assign oTrig = isDone;  
83.      assign oData = D1;  
84.  
85.  endmodule
```

以上内容是输出驱动声明。

ps2_demo.v

组合模块 ps2_demo.v 的建模图就不再重复粘贴了。

```
1. module ps2_demo
2. (
3.     input CLOCK, RESET,
4.     inout PS2_CLK, PS2_DAT,
5.     output [7:0]DIG,
6.     output [5:0]SEL,
7.     output [2:0]LED
8. );
9.     wire EnU1;
10.
11.    ps2_init_funcmod U1
12.    (
13.        .CLOCK(CLOCK),
14.        .RESET(RESET),
15.        .PS2_CLK(PS2_CLK), // < top
16.        .PS2_DAT(PS2_DAT), // < top
17.        .oEn(EnU1) // > U2
18.    );
19.
20.    wire [23:0]DataU2;
21.
22.    ps2_read_funcmod U2
23.    (
24.        .CLOCK(CLOCK),
25.        .RESET(RESET),
26.        .PS2_CLK(PS2_CLK), // < top
27.        .PS2_DAT(PS2_DAT), // < top
28.        .iEn(EnU1), // < U1
29.        .oTrig(),
30.        .oData(DataU2) // > U2
31.    );
32.
33.    // immediate proses
34.    wire[7:0] X = DataU2[4] ? (~DataU2[15:8] + 1'b1) : DataU2[15:8];
35.    wire[7:0] Y = DataU2[5] ? (~DataU2[23:16] + 1'b1) : DataU2[23:16];
36.
37.    smg_basemod U3
38.    (
39.        .CLOCK(CLOCK),
```

```

40.      .RESET( RESET ),
41.      .DIG( DIG ), // > top
42.      .SEL( SEL ), // > top
43.      .iData( { 3'd0,DataU2[5],Y,3'd0,DataU2[4],X } ) // < U2
44.    );
45.
46.    assign LED = {DataU2[1], DataU2[2], DataU2[0]};
47.
48. endmodule

```

该代码非常简单，第 30 行表示 U2 的 oTrig 无用武之地。第 34~35 行是正直化的即时声明。第 43 行是 U3 的联合驱动，其中 3'd0, DataU2[5] 表示第 1 位数码管显示 Y 的符号位，Y 表示第 2~3 位的数码管显示 Y 的正直结果，3'd0, DataU2[4] 表示第 4 位数码管显示 X 的符号位，X 表示第 5~6 位数码管显示 X 的正直结果。第 46 行则是各个按键情况直接驱动 LED 资源。

编译完成并且下载程序。假设笔者按下左键，那么 LED[0]便会点亮，释放则消灭。再假设笔者向左移动鼠标，那么鼠标第 4 位数码管会显示 1，第 5~6 数码管则会显示 X 的内容，亦即鼠标移动的举例。

细节一：精简与直观

```

1.    0: // Read 1st byte
2.    begin i <= FF_Read; Go <= i + 1'b1; end
3.    1: // Store 1st byte
4.    begin D1[7:0] <= T; i <= i + 1'b1; end
5.    2: // Read 2nd byte
6.    begin i <= FF_Read; Go <= i + 1'b1; end
7.    3: // Store 2nd byte
8.    begin D1[15:8] <= T; i <= i + 1'b1; end
9.    4: // Read 3rd byte
10.   begin i <= FF_Read; Go <= i + 1'b1; end
11.   5: // Store 3rd byte
12.   begin D1[23:16] <= T; i <= i + 1'b1; end

```

代码 10.5

代码 10.5 是 PS/2 读功能模块的部分内容，期间步骤 0~5 表示 3 字节读取且暂存的过程。事实上，代码 10.5 可以进一步精简，结果如代码 10.6 所示：

```

13.   0: // Read 1st byte
14.   begin i <= FF_Read; Go <= i + 1'b1; end
15.   1: // Store 1st byte
16.   begin D1[7:0] <= T; i <= FF_Read; Go <= i + 1'b1; end

```

```

17.      2: // Read 2nd byte
18.      begin D1[15:8] <= T; i <= FF_Read; Go <= i + 1'b1; end
19.      3: // Store 2nd byte
20.      begin D1[23:16] <= T; i <= i + 1'b1; end
21.      .....

```

代码 10.6

代码 10.6 相较代码 10.5，它虽然有很高程度的精简度，不过直观程度却不如代码 10.6。到底来到底是直观好，还是精简好，唯有见仁见智了。

细节二：完整的个体模块

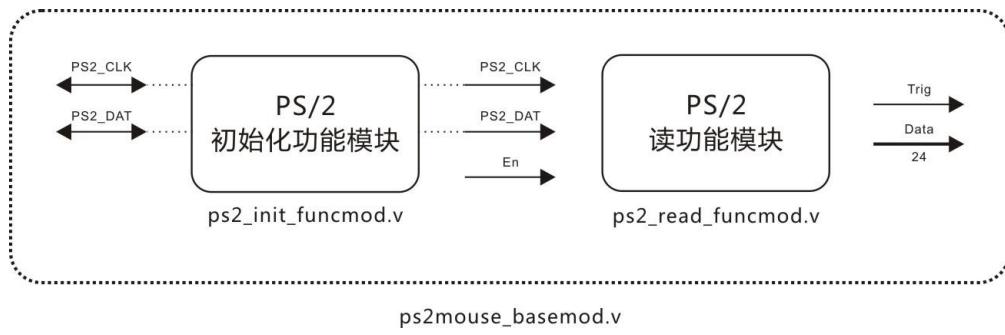


图 10.13 实验十的完整个体模块。

图 10.13 是 PS/2 鼠标基础模块，里边包含 PS/2 初始化功能模块，还有 PS/2 读功能模块。该模块的最左边是顶层信号 PS2_CLK 与 PS2_DAT 的输入，鼠标完成初始化以后，PS/2 初始化功能模块便会拉高 oEn 使能 PS/2 读功能模块。PS/2 读功能模块的左边除了顶层信号以外还有 iEn，iEn 不拉高该模块就不工作。PS/2 读功能模块每完成 3 字节报告的读取，就会经由 oTrig 产生完成信号。

ps2mouse_basemod.v

```

1. module ps2mouse_basemod
2. (
3.     input CLOCK, RESET,
4.     inout PS2_CLK, PS2_DAT,
5.     output oTrig,
6.     output [31:0]oData
7. );
8.     wire EnU1;
9.
10.    ps2_init_funcmod U1
11.    (
12.        .CLOCK(CLOCK),
13.        .RESET(RESET),

```

```
14.          .PS2_CLK( PS2_CLK ), // < top
15.          .PS2_DAT( PS2_DAT ), // < top
16.          .oEn( EnU1 ) // > U2
17.      );
18.
19.      ps2_read_funcmod U2
20.      (
21.          .CLOCK( CLOCK ),
22.          .RESET( RESET ),
23.          .PS2_CLK( PS2_CLK ), // < top
24.          .PS2_DAT( PS2_DAT ), // < top
25.          .iEn( EnU1 ),      // < U1
26.          .oTrig( oTrig ),   // > Top
27.          .oData( oData )    // > Top
28.      );
29.
30. endmodule
```

实验十一：PS/2 模块⑤ — 扩展鼠标

当普通鼠标即三键鼠标再也无法满足需求的时候，扩展鼠标即滚轮鼠标就诞生了，然而实验十一的实验目的就是实现滚轮鼠标的驱动。不过，进入整体之前，先让我们来了解一下鼠标的常用命令。

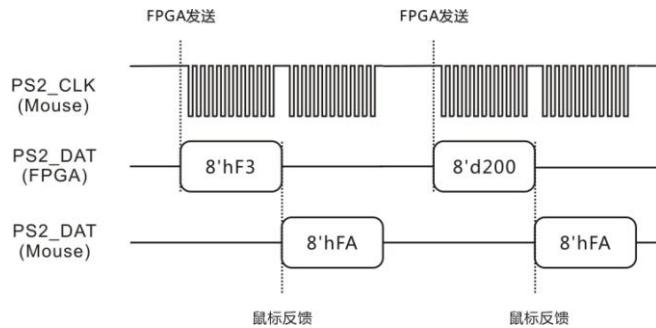


图 11.1 命令 F3，设置采样频率。

命令 F3 也是 Set Sample Rate，主要是用来设置采集频率。笔者曾经说过，采集频率就是鼠标采集按键状况还有位置状况的间隔时间，默认下是 100 次/秒。如图 11.1 所示，FPGA 先发送命令数据 `8'hF3`，事后鼠标会反馈 `8'hFA` 以示接收成功，余下 FPGA 再发送参数数据 `8'd200`，鼠标接收成功后也会反馈 `8'hFA`。如此一来，鼠标的采集频率从原本的 100 次/秒，变成 200 次/秒。

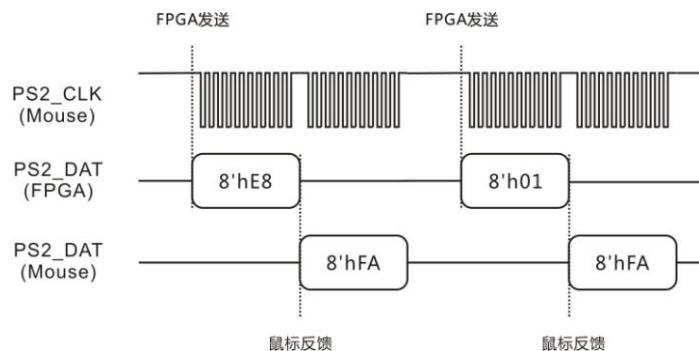


图 11.2 命令 E8，设置分辨率。

命令 E8 也是 Set Resolution，主要是用来设置分辨率。所谓分辨率就是位置对应寄存器计数的单位，默认下是 4 计数/mm，亦即 1mm 的距离，鼠标计数 4 下。如图 11.2 所示，FPGA 先发送命令数据 `8'hE8`，鼠标接收以后便反馈 `8'hFA`，FPGA 随之也会发送参数数据 `8'h01`，鼠标接收以后也会反馈数据 `8'hFA`。完后，鼠标的分辨从原本的 4 计数/mm 变成 2 计数/mm。

参数数据所对应的分辨率如表 11.1 所示：

表 11.1 参数数据所对应的分辨率。

参数数据	分辨率
------	-----

8' h00	1 计数/mm
8' h01	2 计数/mm
8' h02	4 计数/mm
8' h03	8 计数/mm

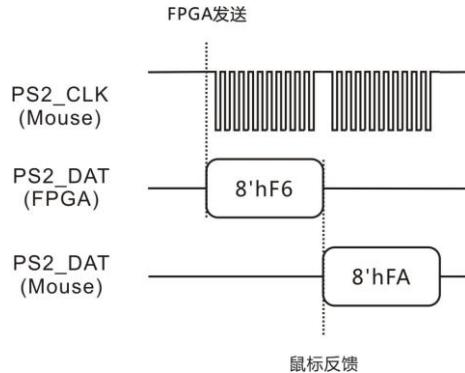


图 11.3 命令 F6，使用默认参数。

假设笔者手痒，不小心打乱鼠标内部的参数数据，此刻笔者可以发送命令 F6，即 Set Defaults 将参数数据回复成原来的缺省值。如图 11.3 所示，FPGA 先发送命令数据 8'hF6，鼠标完成接收以后便会反馈 8'hFA。

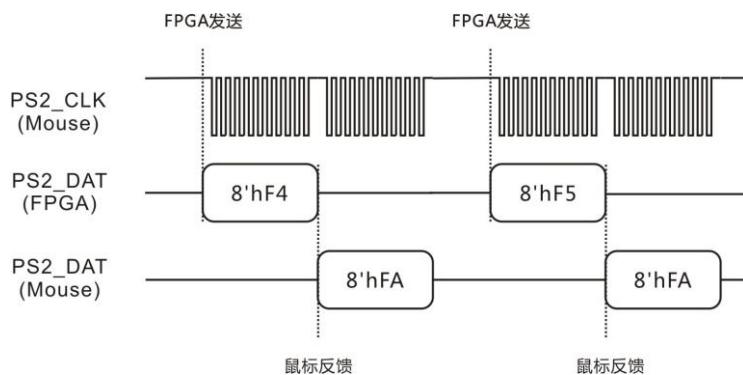


图 11.4 命令 F4 使能报告，命令 F5 关闭报告。

PS/2 鼠标不像 PS/2 键盘，上电并且完成初始化以后它便会陷入发呆状态，如果不发送命令数据 8'hF4（即 Enable Data Report）手动开启鼠标的水龙头，鼠标是不会发送报告（即夹杂按键状况与位置状况的数据）。如图 11.4 所示，FPGA 先发送命令数据 8'hF4，鼠标接收以后便会反馈 8'hFA，事后鼠标立即处于就绪状态，一旦按键状况或者位置状况发生改变，鼠标就会发送报告。

假设读者觉得鼠标太唠叨，什么大事小事都报告，笔者可以发送命令数据 8'hF5（即 Disable Data Report）为了使其闭嘴。如图 11.4 所示，FPGA 先发送命令数据 8'hF4，鼠标接收完毕以后便会反馈 8'hFA，事后鼠标就成为闭嘴状态，大事小事再也不会烦人。如果读者觉得寂寞，读者可以再度发送命令数据 8'hF4，让鼠标再度唱歌。

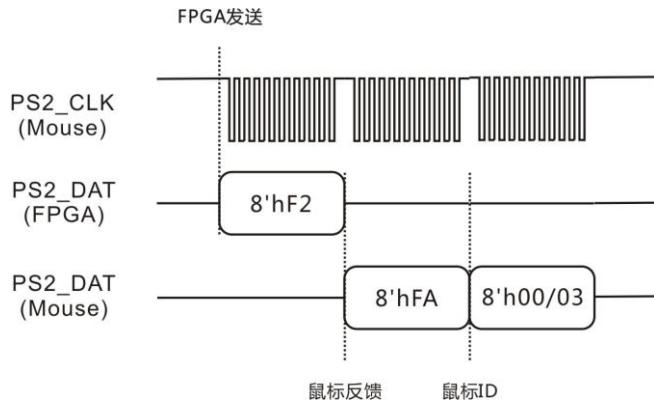


图 11.5 命令 F2 , 读取鼠标 ID。

为了区分鼠标是普通鼠标还是扩展鼠标，期间我们必须使用命令 $8'hF2$ ，即 Get Device ID。如图 11.5 所示，FPGA 发送命令数据 $8'hF2$ ，鼠标接收以后先反馈 $8'hFA$ ，再来便发送鼠标 ID。如果内容是 $8'h00$ ，则表示该鼠标只是普通鼠标 … 反之，如果内容是 $8'h03$ ，那么该鼠标就是扩展鼠标。因为如此，我们需要更改一下伪函数，结果如代码 11.1 所示：

```

1.      32: // Press low PS2_CLK 100us
2.      if( C1 == T100US -1 ) begin C1 <= 13'd0; i <= i + 1'b1; end
3.      else begin isQ1 = 1'b1; rCLK <= 1'b0; C1 <= C1 + 1'b1; end
4.
5.      33: // release PS2_CLK and set in ,PS2_DAT set out
6.      begin isQ1 <= 1'b0; rCLK <= 1'b1; isQ2 <= 1'b1; i <= i + 1'b1; end
7.
8.      34: // start bit 1
9.      begin rDAT <= 1'b0; i <= i + 1'b1; end
10.
11.     35,36,37,38,39,40,41,42,43: // data bit 9
12.     if( isH2L ) begin rDAT <= T[ i-35 ]; i <= i + 1'b1; end
13.
14.     44: // stop bit 1
15.     if( isH2L ) begin rDAT <= 1'b1; i <= i + 1'b1; end
16.
17.     45: // Ack bit
18.     if( isH2L ) begin i <= i + 1'b1; end
19.
20.     46: // PS2_DAT set in
21.     begin isQ2 <= 1'b0; i <= i + 1'b1; end
22.
23.     /***** // Receive 1st Frame
24.
25.     47,48,49,50,51,52,53,54,55,56,57: // Ingnores

```

```

26.         if( isH2L ) i <= i + 1'b1;
27.
28.         58: // Check comd F2
29.             if( T[7:0] == 8'hF2 ) i <= i + 1'b1;
30.             else i <= Go;
31.
32.         /***** // Receive 2nd Frame
33.
34.         59: // Start bit 1
35.             if( isH2L ) i <= i + 1'b1;
36.
37.             60,61,62,63,64,65,66,67,68: // Data bit 9
38.                 if( isH2L ) begin T[i-60] <= PS2_DAT; i <= i + 1'b1; end
39.
40.             69: // Stop bit 1
41.                 if( isH2L ) i <= Go;

```

代码 11.1

如代码 11.1 所示，步骤 32~57 则是发送一帧数据又忽略一帧反馈，基本上与实验十一模一样。至于第 58 行则是用来判断，FPGA 所发送的命令是否是 8'hF2 即 Get Device ID？如果是，步骤则继续读取操作，因为命令 8'hF2 令鼠标反馈 8'hFA 之余，还会导致鼠标会发送一帧 ID 数据。否则的话，即表示其他命令，步骤返回。步骤 59~69 是用来读取下一帧 ID 数据，期间步骤 60~68 用来读取 8 位数据位，还有 1 位校验位。完后，步骤便返回。

小时候的笔者很爱假扮刺客，笔者与近邻的小孩就总是瞎着玩，其它小朋友则扮演秘密商人。刺客为了与秘密商人进行交易，两者之间必须经过暗语核对，例如：

“阳光的男孩赤裸裸 … ”，对方问道。
“对面的女来看过来 … ”，笔者答道。

滚轮鼠标也是扩展鼠标，上电以后也不会立即变成扩展鼠标，如果扩展鼠标不经过核对暗语，扩展鼠标也是一只普通的 3 键鼠标而已 … 反之，如果完成暗语核对，扩展鼠标才会发挥滚轮功能。

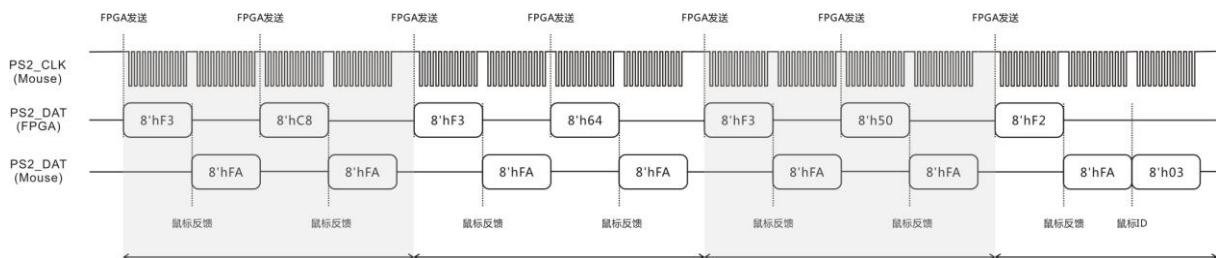


图 11.6 设置扩展鼠标的暗语。

如图 11.6 所示，那是设置扩展鼠标的暗语：

发送命令数据 8'hF3，接收反馈 8'hFA，再发送参数数据 8'hC8，在接收反馈 8'hFA；
发送命令数据 8'hF3，接收反馈 8'hFA，再发送参数数据 8'h64，在接收反馈 8'hFA；
发送命令数据 8'hF3，接收反馈 8'hFA，再发送参数数据 8'h50，在接收反馈 8'hFA；
发送命令数据 8'hF2，接收反馈 8'hFA，再接收鼠标 ID8'h03。

完后，鼠标便成为扩展鼠标，内部也自动初始化，然后进入默认模式。

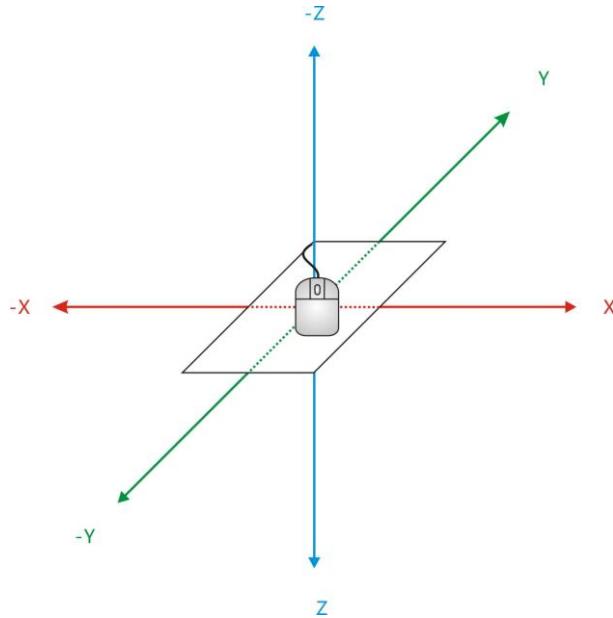


图 11.7 扩展鼠标标示的位置。

普通鼠标相较扩展鼠标，它多了滚轮功能，即鼠标除了标示左键，中键，右键，X 以外，扩展还会标示 Z。如图 11.7 所示，X 与 Y 可以看成面积，至于 Z 则可以看成上下。当鼠标向西移动，X 呈现正直，反之负值；当鼠标向北移动，Y 呈现正直，反之负值；当滚动向下活动，Z 呈现正直，反之负值。

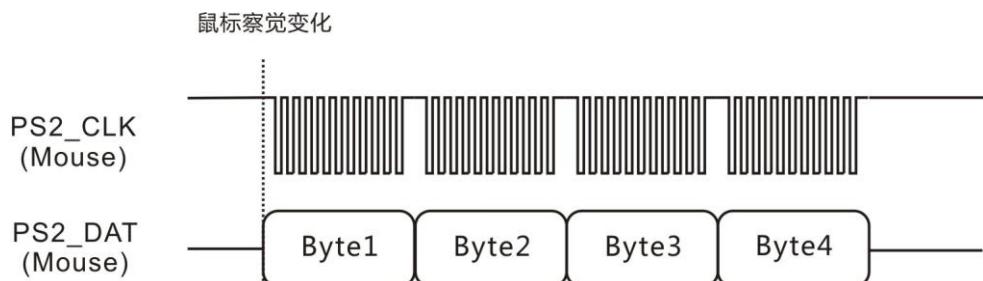


图 11.8 扩展鼠标的报告长度。

为此，扩展鼠标相较普通鼠标，报告长度则多了一个字节。如图 11.8 所示，当鼠标察觉变化以后，鼠标便会发送 4 个字节长度的报告，然而字节之间的位分配如表 11.1 所示：

表 11.1 Device ID 为 8'h03 的报告内容。

字节/位	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
字节一	Y 溢出位	X 溢出位	Y[8]符号位	X[8]符号位	保留	中键	右键	左键
字节二	X[7:0]							
字节三	Y[7:0]							
字节四	保留	保留	保留	保留	Z[3]符号位	Z[2]	Z[1]	Z[0]

笔者需要补充一下 ... 由于早期 Intel 称王，所以扩展鼠标标准都是 Intel 说话算话，Device ID 为 8'h03 就是其中一种扩展标准。如表 11.1 所示，字节一至字节三基本上变化不大，反之字节四则稍微不同。字节四的[2..0]位是 Z[2:0]，字节四的[3]是 Z[3]，也是 Z 的符号位。换句话说，寄存器 Z 有 4 位，内容用补码表示。

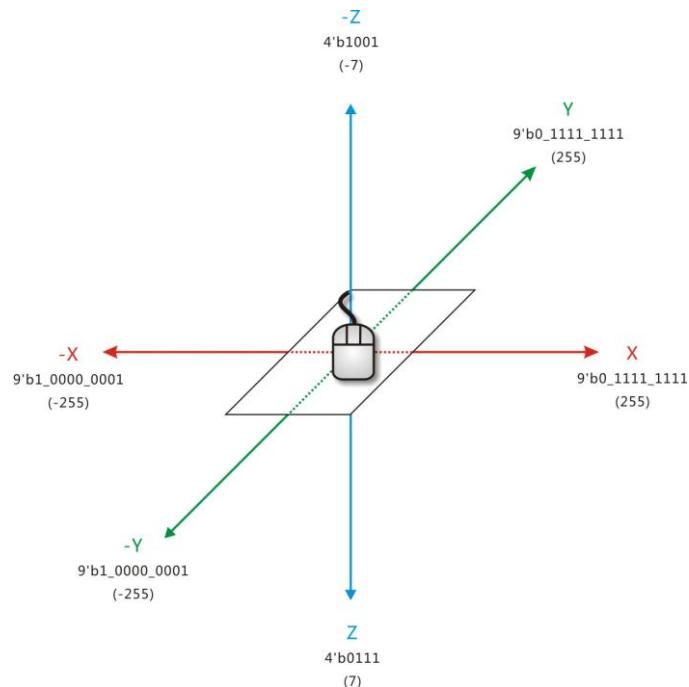


图 11.9 扩展鼠标的位置范围。

图 11.9 表示扩展鼠标的位置范围，X 与 Y 与普通鼠标一样，Z 比较畸形一点，因为 Z 向上不是正直而是负值，反之亦然。Z 的有效范围是 4'b1001~4'b0111 或者 -7~7，也就是说滚轮向下活动，寄存器 Z 就递增，向上滚动，寄存器 Z 就递减。

上述内容理解完毕以后，我们便可以开始建模了：

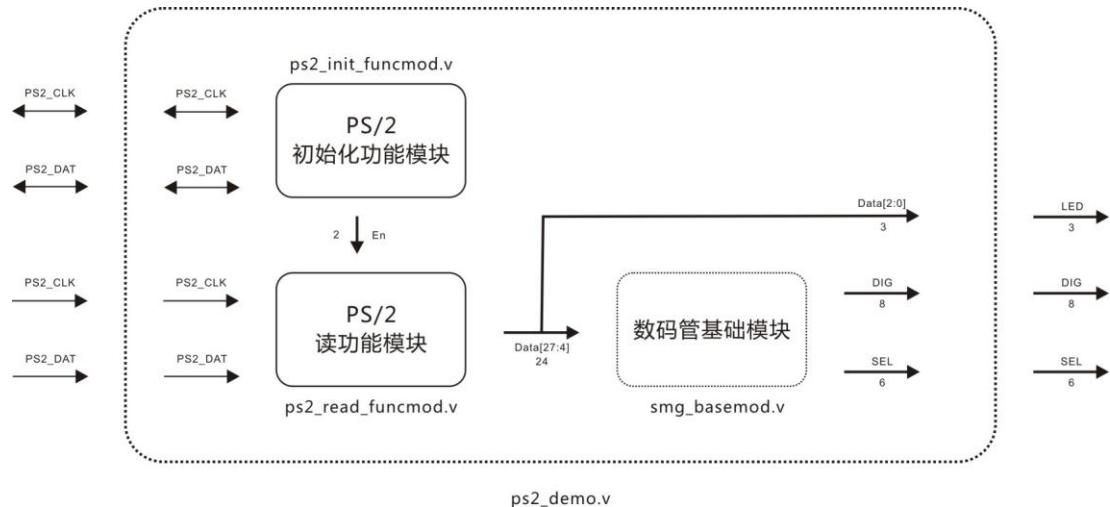


图 11.10 实验十一的建模图。

如图 11.10 所示，组合模块 ps2_demo 包含的内容与实验十相差不了多少，不过却少了正直化的即时操作。期间，PS/2 初始化功能模块的 oEn 有两位，oEn[1] 拉高表示鼠标为扩展鼠标，oEn[0] 拉高表示鼠标为普通鼠标。PS/2 读取功能模块的 oData[2:0] 直接驱动 LED 资源， oData[27:4] 则驱动数码管基础模块的 iData。

ps2_init_funcmod.v

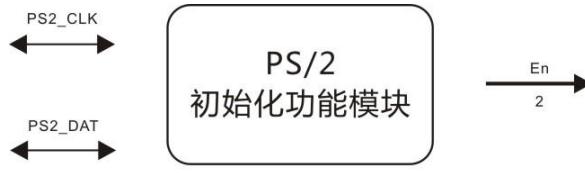


图 11.11 PS/2 初始化功能模块的建模图。

如图 11.11 所示，PS/2 初始化功能模块有两位 oEn，[1]拉高表示鼠标为扩展鼠标，[0]拉高则表示鼠标为普通鼠标。

```

1. module ps2_init_funcmod
2. (
3.     input CLOCK, RESET,
4.     inout PS2_CLK,
5.     inout PS2_DAT,
6.     output [1:0]oEn
7. );
8. parameter T100US = 13'd5000;
9. parameter FF_Write = 7'd32;
10.

```

以上内容为相关的出入端声明。第 8 行是 100us 的常量声明，第 9 行则是伪函数的入口地址。

```
11.      ****// sub1
12.
13.      reg F2,F1;
14.
15.      always @ ( posedge CLOCK or negedge RESET )
16.          if( !RESET )
17.              { F2,F1 } <= 2'b11;
18.          else
19.              { F2, F1 } <= { F1, PS2_CLK };
20.
21.      ****// core
22.
23.      wire isH2L = ( F2 == 1'b1 && F1 == 1'b0 );
```

以上内容是用来检测电平变化的周边操作，第 23 行则是下降沿的即时声明。

```
24.      reg [8:0]T;
25.      reg [6:0]i,Go;
26.      reg [12:0]C1;
27.      reg rCLK,rDAT;
28.      reg isQ1,isQ2,isEx;
29.      reg [1:0]isEn;
30.
31.      always @ ( posedge CLOCK or negedge RESET )
32.          if( !RESET )
33.              begin
34.                  T <= 9'd0;
35.                  C1 <= 13'd0;
36.                  { i,Go } <= { 7'd0,7'd0 };
37.                  { rCLK,rDAT } <= 2'b11;
38.                  { isQ1,isQ2,isEx } <= 3'b000;
39.                  isEn <= 2'b00;
40.              end
41.          else
```

以上内容是相关的寄存器声明，第 33~39 行则是这群寄存器的复位操作。其中 isEn 有两位，isEx 为扩展鼠标的立旗。

```
42.          case( i )
43.
```

```

44.          **** // INIT Mouse
45.
46.          0: // Send F3 1111_0011
47.          begin T <= { 1'b1, 8'hF3 }; i <= FF_Write; Go <= i + 1'b1; end
48.
49.          1: // Send C8 1100_1000
50.          begin T <= { 1'b0, 8'hC8 }; i <= FF_Write; Go <= i + 1'b1; end
51.
52.          2: // Send F3 1111_0011
53.          begin T <= { 1'b1, 8'hF3 }; i <= FF_Write; Go <= i + 1'b1; end
54.
55.          3: // Send 64 0110_1000
56.          begin T <= { 1'b0, 8'h64 }; i <= FF_Write; Go <= i + 1'b1; end
57.
58.          4: // Send F3 1111_0011
59.          begin T <= { 1'b1, 8'hF3 }; i <= FF_Write; Go <= i + 1'b1; end
60.
61.          5: // Send 50 0101_0000
62.          begin T <= { 1'b1, 8'h50 }; i <= FF_Write; Go <= i + 1'b1; end
63.
64.          6: // Send F2 1111_0010
65.          begin T <= { 1'b0, 8'hF2 }; i <= FF_Write; Go <= i + 1'b1; end
66.
67.          7: // Check Mouse ID 00(normal), 03(extend)
68.          if( T[7:0] == 8'h03 ) begin isEx <= 1'b1; i <= i + 1'b1; end
69.          else if( T[7:0] == 8'h00 ) begin isEx <= 1'b0; i <= i + 1'b1; end
70.
71.          8: // Send F4 1111_0100
72.          begin T <= { 1'b0, 8'hF4 }; i <= FF_Write; Go <= i + 1'b1; end
73.
74.          9:
75.          if( isEx ) isEn[1] <= 1'b1;
76.          else if( !isEx ) isEn[0] <= 1'b1;
77.

```

以上内容是核心操作。步骤 0~9 是主操作，步骤 0~6 则是发送用来开启扩展鼠标的暗语，步骤 7 用来判断鼠标返回的 Device ID 是否为 8'h03，如果是 isEx 立旗，否则 isEx 消除立旗。步骤 8 用来使能鼠标。步骤 9 根据 isEx 的状态再来决定 isEn 的结果，如果 isEx 为 1 isEn[1] 便拉高，否则 isEx 拉高，完后步骤停留。

```

78.          **** // PS2 Write Function
79.
80.          32: // Press low PS2_CLK 100us

```

```

81.           if( C1 == T100US -1 ) begin C1 <= 13'd0; i <= i + 1'b1; end
82.           else begin isQ1 = 1'b1; rCLK <= 1'b0; C1 <= C1 + 1'b1; end
83.
84.           33: // release PS2_CLK and set in ,PS2_DAT set out
85.           begin isQ1 <= 1'b0; rCLK <= 1'b1; isQ2 <= 1'b1; i <= i + 1'b1; end
86.
87.           34: // start bit 1
88.           begin rDAT <= 1'b0; i <= i + 1'b1; end
89.
90.           35,36,37,38,39,40,41,42,43: // data bit 9
91.           if( isH2L ) begin rDAT <= T[ i-35 ]; i <= i + 1'b1; end
92.
93.           44: // stop bit 1
94.           if( isH2L ) begin rDAT <= 1'b1; i <= i + 1'b1; end
95.
96.           45: // Ack bit
97.           if( isH2L ) begin i <= i + 1'b1; end
98.
99.           46: // PS2_DAT set in
100.          begin isQ2 <= 1'b0; i <= i + 1'b1; end
101.
102.          /***** // Receive 1st Frame
103.
104.          47,48,49,50,51,52,53,54,55,56,57: // Ingnores
105.          if( isH2L ) i <= i + 1'b1;
106.
107.          58: // Check comd F2
108.          if( T[7:0] == 8'hF2 ) i <= i + 1'b1;
109.          else i <= Go;
110.

```

以上内容是部分核心操作。步骤 32~58 是部分伪函数，内容则是发送一帧数据，再读取一帧反馈，完后便进入步骤 58 判断，发送的命令是否为 8'hF2，如果是便继续步骤，否则便返回步骤。

```

111.          /***** // Receive 2nd Frame
112.
113.          59: // Start bit 1
114.          if( isH2L ) i <= i + 1'b1;
115.
116.          60,61,62,63,64,65,66,67,68: // Data bit 9
117.          if( isH2L ) begin T[i-60] <= PS2_DAT; i <= i + 1'b1; end
118.

```

```

119.           69: // Stop bit 1
120.           if( isH2L ) i <= Go;
121.
122.       endcase
123.

```

以上内容是部分核心操作。步骤 59~69 也是部分伪函数，主要用来读取下一帧数据的字节内容，在此是针对命令 8'hF2，也就是 Device ID。读完一帧数据以后便返回步骤。

```

124.     assign PS2_CLK = isQ1 ? rCLK : 1'bz;
125.     assign PS2_DAT = isQ2 ? rDAT : 1'bz;
126.     assign oEn = isEn;
127.
128. endmodule

```

以上内容为驱动输出声明。

ps2_read_funcmod.v



图 11.12 PS/2 读功能模块的建模图。

实验十一的 PS/2 读功能模块与实验十相比，左边的 iEn 出入多出一位以外，右边的 oData 也多出一个字节。

```

1. module ps2_read_funcmod
2. (
3.     input CLOCK, RESET,
4.     input PS2_CLK,PS2_DAT,
5.     input [1:0]iEn,
6.     output oTrig,
7.     output [31:0]oData
8. );
9.     parameter FF_Read = 7'd32;

```

以上内容是相关的出入端声明。第 9 行是伪函数的入口。

```

10.

```

```

11.      ****// sub1
12.
13.      reg F2,F1;
14.
15.      always @ ( posedge CLOCK or negedge RESET )
16.          if( !RESET )
17.              { F2,F1 } <= 2'b11;
18.          else
19.              { F2, F1 } <= { F1, PS2_CLK };
20.
21.      ****// core
22.
23.      wire isH2L = ( F2 == 1'b1 && F1 == 1'b0 );

```

以上内容是检测电平变化的周边操作，第 23 行则是下降沿的即时声明。

```

24.      reg [31:0]D1;
25.      reg [7:0]T;
26.      reg [6:0]i,Go;
27.      reg isDone;
28.
29.      always @ ( posedge CLOCK or negedge RESET )
30.          if( !RESET )
31.              begin
32.                  D1 <= 32'd0;
33.                  T <= 8'd0;
34.                  { i,Go } <= { 7'd0,7'd0 };
35.                  isDone <= 1'b0;
36.              end

```

以上内容为相关的寄存器声明以及复位操作。

```

37.          else if( iEn[1] )
38.              case( i )
39.
40.                  ****// Extend Mouse Read Data
41.
42.                      0: // Read Data 1st byte
43.                      begin i <= FF_Read; Go <= i + 1'b1; end
44.
45.                      1: // Store Data 1st byte
46.                      begin D1[7:0] <= T; i <= i + 1'b1; end
47.

```

```

48.          2: // Read Data 2nd byte
49.          begin i <= FF_Read; Go <= i + 1'b1; end
50.
51.          3: // Store Data 2nd byte
52.          begin D1[15:8] <= T; i <= i + 1'b1; end
53.
54.          4: // Read Data 3rd byte
55.          begin i <= FF_Read; Go <= i + 1'b1; end
56.
57.          5: // Store Data 3rd byte
58.          begin D1[23:16] <= T; i <= i + 1'b1; end
59.
60.          6: // Read Data 4rd byte
61.          begin i <= FF_Read; Go <= i + 1'b1; end
62.
63.          7: // Store Data 4rd byte
64.          begin D1[31:24] <= T; i <= i + 1'b1; end
65.
66.          8:
67.          begin isDone <= 1'b1; i <= i + 1'b1; end
68.
69.          9:
70.          begin isDone <= 1'b0; i <= 7'd0; end

```

以上内容为部分核心操作。第 37 行的 if(iEn[1]) 表示下面所有内容都是扩展鼠标的核
心操作。步骤 0~7 则是读取 4 个字节的数据，步骤 8~9 用来产生完成信号以示一次性的
报告已经接收完毕。

```

71.
72.          ****// PS2 Write Function
73.
74.          32: // Start bit
75.          if( isH2L ) i <= i + 1'b1;
76.
77.          33,34,35,36,37,38,39,40: // Data byte
78.          if( isH2L ) begin T[i-33] <= PS2_DAT; i <= i + 1'b1; end
79.
80.          41: // Parity bit
81.          if( isH2L ) i <= i + 1'b1;
82.
83.          42: // Stop bit
84.          if( isH2L ) i <= Go;
85.

```

```
86.          endcase
```

以上内容为部分核心操作。步骤 32~42 是读取一帧数据的伪函数。

```
87.          else if( iEn[0] )
88.              case( i )
89.
90.                  /***** // Normal Mouse Read Data
91.
92.                      0: // Read Data 1st byte
93.                      begin i <= FF_Read; Go <= i + 1'b1; end
94.
95.                      1: // Store Data 1st byte
96.                      begin D1[7:0] <= T; i <= i + 1'b1; end
97.
98.                      2: // Read Data 2nd byte
99.                      begin i <= FF_Read; Go <= i + 1'b1; end
100.
101.                     3: // Store Data 2nd byte
102.                     begin D1[15:8] <= T; i <= i + 1'b1; end
103.
104.                     4: // Read Data 3rd byte
105.                     begin i <= FF_Read; Go <= i + 1'b1; end
106.
107.                     5: // Store Data 3rd byte
108.                     begin D1[23:16] <= T; i <= i + 1'b1; end
109.
110.                     6:
111.                     begin isDone <= 1'b1; i <= i + 1'b1; end
112.
113.                     7:
114.                     begin isDone <= 1'b0; i <= 7'd0; end
115.
```

以上内容为部分核心操作。第 87 行的 `if(iEn[0])` 表示下面的内容均为普通鼠标的核心操作。步骤 0~5 用来读取 3 个字节的内容，步骤 6~7 则用来产生完成信号以示一次性的报告已经读取完毕。

```
116.          /***** // PS2 Write Function
117.
118.          32: // Start bit
119.          if( isH2L ) i <= i + 1'b1;
120.
```

```
121.           33,34,35,36,37,38,39,40: // Data byte
122.           if( isH2L ) begin T[i-33] <= PS2_DAT; i <= i + 1'b1; end
123.
124.           41:// Parity bit
125.           if( isH2L ) i <= i + 1'b1;
126.
127.           42:// Stop bit
128.           if( isH2L ) i <= Go;
129.
130.       endcase
131.
```

以上内容为部分核心操作。步骤 32~42 是读取一帧数据的伪函数。

```
132.     assign oTrig = isDone;
133.     assign oData = D1;
134.
135. endmodule
```

以上内容是输出驱动声明。

ps2_demo.v

笔者就不重复粘贴实验十一的建模图了，具体内容我们还是来看代码吧。

```
1. module ps2_demo
2. (
3.     input CLOCK, RESET,
4.     inout PS2_CLK, PS2_DAT,
5.     output [7:0]DIG,
6.     output [5:0]SEL,
7.     output [2:0]LED
8. );
9.     wire [1:0]EnU1;
10.
11.     ps2_init_funcmod U1
12.     (
13.         .CLOCK( CLOCK ),
14.         .RESET( RESET ),
15.         .PS2_CLK( PS2_CLK ), // < top
16.         .PS2_DAT( PS2_DAT ), // < top
17.         .oEn( EnU1 ) // > U2
18.     );
```

```

19.
20.      wire [31:0]DataU2;
21.
22.      ps2_read_funcmod U2
23.      (
24.          .CLOCK( CLOCK ),
25.          .RESET( RESET ),
26.          .PS2_CLK( PS2_CLK ), // < top
27.          .PS2_DAT( PS2_DAT ), // < top
28.          .iEn( EnU1 ),       // < U1
29.          .oTrig(),
30.          .oData( DataU2 )  // > U3
31.      );
32.
33.      smg_basemod U3
34.      (
35.          .CLOCK( CLOCK ),
36.          .RESET( RESET ),
37.          .DIG( DIG ), // > top
38.          .SEL( SEL ), // > top
39.          .iData( { 2'd0,DataU2[5],DataU2[4],DataU2[27:24],DataU2[23:16],DataU2[15:8] } ) // < U2
40.      );
41.
42.      assign LED = {DataU2[1], DataU2[2], DataU2[0]};
43.
44. endmodule

```

上诉内容的连线部署基本上与图 11.10 差不了多少，期间第 39 行的 `2'd0` , `DataU2[5]` , `DataU2[4]` 表示数码管的第一位显示 X 与 Y 的符号位；`DataU2[27:24]` 表示数码管的第二位显示 Z 的内容；`DataU2[23:16]` 表示数码管的第三至第四位显示 Y 的内容；`DataU2[15:8]` 表示数码管的第五至第六位显示 X 的内容。第 42 行则表示 LED[2]显示右键，LED[1]显示中键，LED[0]显示左键。

编译完毕并且下载程序。当鼠标向西南方移动的时候，第一位数码管便会显示 `4'h3`，即 `4'b0011`，也就是说 X 与 Y 的符号位都是拉高状态（负值）。当滚轮向上滚动的时候，第二位数码管便会显示 `4'hF`，即 `4'b1111`，也就是 Z 为负值 -1（只要滚动速度够快，负值还能更小）。至于数码管第 3~4 显示 Y 的内容（补码形式），数码管 5~6 则显示 X 的内容（补码形式）。

细节一：两个人，两把汤匙

```

1.      else if( iEn[1] )
2.          case( i )

```

```
3.          扩展鼠标的核心操作 ;
4.          伪函数;
5.          endcase
6.      else if ( isEn[0] )
7.          case(i)
8.          普通鼠标的核心操作 ;
9.          伪函数 ;
10.         endcase
```

代码 11.2

PS/2 读取功能模块有一个有趣的现象，即资源多义性的问题。如代码 11.2 所示，PS/2 读取功能模块用 `if(iEn[1])` 与 `if(iEn[0])` 表示该模块针对两种鼠标的读取操作。这种感觉好比一对兄弟在吃饭 ... 正常情况下，当然是一个人一把汤匙才对，这种比喻完全对应代码 11.2 的内容。

PS/2 读取功能模块负责两种鼠标的读取操作之际，里边好比有一对兄弟，一个人负责扩展鼠标的读取操作，另一个人则针对普通鼠标的读取操作。期间，伪函数就是某种操作资源，也可以看成是汤匙。为了不让两位兄弟争用一把汤匙而吵架，身为设计者的我们，应该为每个人分配一把汤匙。

对此，我们必须多花一些钱买另一把汤匙，这样做我们可能多消耗一些逻辑资源。不过，家和为贵，为使模块可以和谐共处以致提高表达能力，要笔者多消耗一些逻辑资源，笔者也觉得值得。

细节二：完整的个体模块

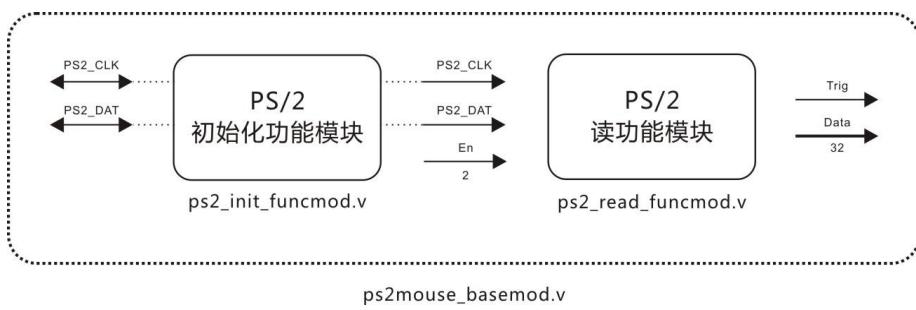


图 11.13 PS/2 鼠标基础模块的建模图。

图 11.13 是 PS/2 鼠标基础模块的建模图。

ps2mouse_basemod.v

```
1. module ps2mouse_basemod
2. (
3.     input CLOCK, RESET,
4.     inout PS2_CLK, PS2_DAT,
```

```
5.      output oTrig,
6.      output [31:0]oData
7.  );
8.      wire [1:0]EnU1;
9.
10.     ps2_init_funcmod U1
11.     (
12.         .CLOCK( CLOCK ),
13.         .RESET( RESET ),
14.         .PS2_CLK( PS2_CLK ), // < top
15.         .PS2_DAT( PS2_DAT ), // < top
16.         .oEn( EnU1 ) // > U2
17.     );
18.
19.     ps2_read_funcmod U2
20.     (
21.         .CLOCK( CLOCK ),
22.         .RESET( RESET ),
23.         .PS2_CLK( PS2_CLK ), // < top
24.         .PS2_DAT( PS2_DAT ), // < top
25.         .iEn( EnU1 ),        // < U1
26.         .oTrig( oTrig ),   // > top
27.         .oData( oData )    // > top
28.     );
29.
30. endmodule
```

实验十二：串口模块① — 发送

串口固然是典型的实验，想必许多同学已经作烂，不过笔者还要循例介绍一下。我们知道串口有发送与接收之分，实验十二的实验目的就是实现串口发送，然而不同的是 ... 笔者会用另一种思路去实现串口发送。

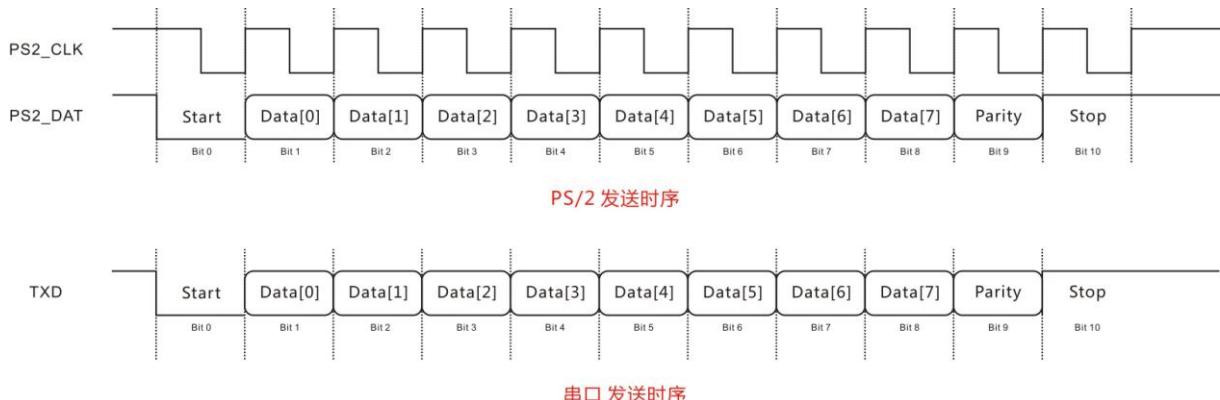


图 12.1 PS/2 发送时序与串口发送时序。

如图 12.1 所示，串口发送时序相较 PS/2 发送时序，串口发送时序就像断了翅膀的小鸟般，没有时钟信号控制整个传输协议。除此之外，串口发送时序与 PS/2 发送时序近似的地方也非常惊人 ... 默认下，一帧 PS/2 数据有 11 位，对此一帧串口数据也有 11 位，其中位分配如表 12.1 所示：

表 12.1 一帧串口数据的位分配。

位分配	位定义
[0]	起始位
[8:1]	数据位
[9]	校验位
[10]	停止位

如表 12.1 所示，[0]为拉低的起始位，[8:1]为任意填充的数据位，[9]为任意填充的校验位，[10]为拉高的停止位。

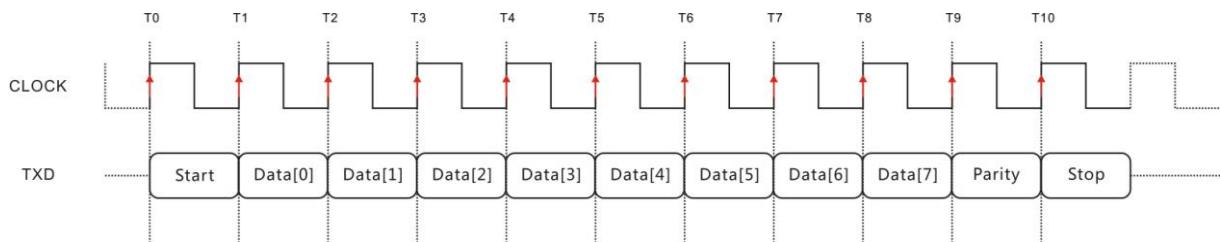


图 12.2 FPGA 发送一帧串口数据 (无视波特率)。

假设我们无视波特率，并且利用 FPGA 发送一帧串口数据的话 ... 如图 12.2 所示，一帧有 11 位的串口数据，一共使用了 11 个上升沿发送出去，为此 Verilog 则可以这样表示，结果如代码 12.1 所示：

```

reg [10:0]D1;
always @ ( posedge CLOCK )
  case ( i )
    0,1,2,3,4,5,6,7,8,9,10:
      begin TXD <= D1[i]; i <= i + 1'b1; end
    .....
  endcase

```

代码 12.1

如代码 12.1 所示，寄存器 D 为 11 位宽的寄存器，并且驱动 TXD 输出口，期间步骤 0~11 公按照 i 的位寻址，将 D1 的内容逐个发送出去。

虽然串口传输协议极为类似 PS/2 传输协议，但是串口传输协议也有区别的地方。其一串口传输协议有波特率的概念，而且串口协议有各种各样的波特率，常用的波特率有 9600 bps 或者 115200 bps，波特率最低为 110 bps，最高为 256000 bps（目前暂定）。所谓波特率就是一秒内，串口可以发送多少位数据，此外波特率也是一位数据的周期，或者说是一位数据的保持时间，就拿 115200 bps 为例：

$$1/115200 = 8.68E-6$$

115200 波特率的一位数据周期为 8.68us，如果用 50Mhz 的时钟频率去量化的话：

$$(1/115200) / (1/50E+6) = 8.68E-6 / 20E-9 \\ = 434$$

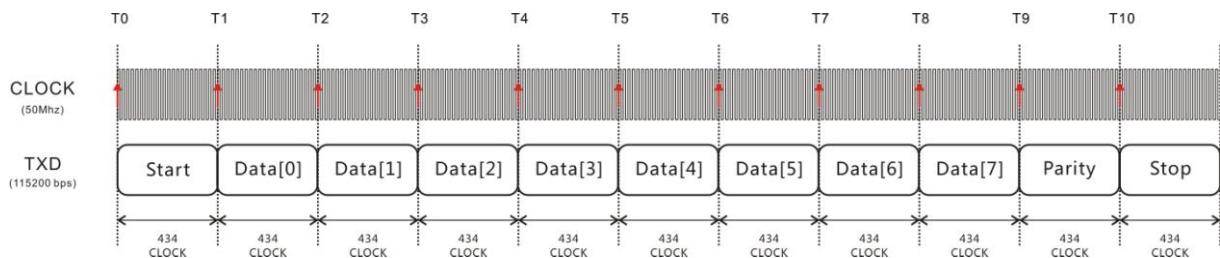


图 12.3 FPGA 发送一帧串口数据（考虑波特率）。

如果图 12.3 考虑 115200 的波特率，结果如图 12.3 所示，每一位数据都保持 434 个时钟，为此 Verilog 可以这样表示，如代码 12.2 所示：

```

reg [10:0]D1;
reg [8:0]C1;
always @ ( posedge CLOCK )
  case ( i )
    0,1,2,3,4,5,6,7,8,9,10:
      if( C1 == 9'd434 - 1 ) begin C1 <= 9'd0; i <= i + 1'b1; end

```

```

else begin TXD <= D1[i]; C1 <= C1 + 1'b1; end
.....
endcase

```

代码 12.1

如代码 12.1 所示，步骤 0~10 不再保持一个时钟，换之每个步骤都保持 434 个时钟，因此每位 TXD 的发送数据也保持 8.68us。

除此此外，串口传输协议不仅可以自定义波特率，串口传输协议也可以自定义一帧数据的位宽，自定义内容如表 12.2 所示：

表 12.2 自定义一帧数据。

自定义数据位	自定义内容
数据位	5~9
校验位	有/无
停止位	1~2

如表 12.2 所示，可以自定义的数据其中便包含数据位，默认下为 1 字节，自定义内容则是 5~9 位，校验位也可以设置为有或者无（默认下是有），停止位也可以增至 2 位（默认下是 1 位）。不管怎么样，表 12.2 是比较官方的自定义内容 ... 只要读者欢喜，任何畸形的自定义内容也有可能实现。

理解完毕以后，我们便可以开始建模了。

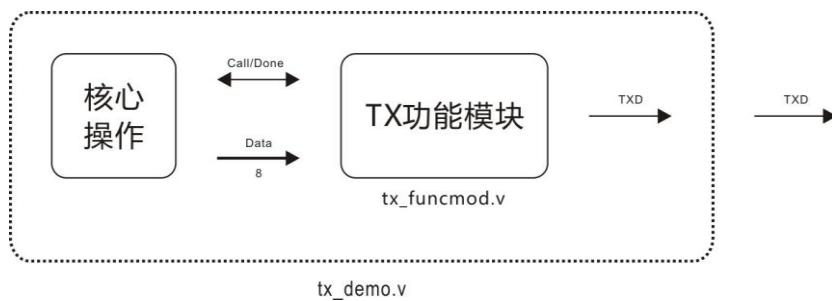


图 12.4 实验十二的建模图。

图 12.4 是实验十二的建模图，不过内容较为寒酸 ... 组合模块 tx_demo 内容包括一段核心操作，还有一只 TX 功能模块。核心操作负责 TX 功能模块的调用，亦即控制沟通信号还有 Data 的输入。TX 功能模块被使能以后，便将 iData 经由 TXD 输出端发送出去，完后便反馈完成信号以示一次性的操作已经完毕。

tx_funcmod.v



图 12.5 TX 功能模块的建模图。

如图 12.5 所示，该模块的左方有问答信号，还有 8 位的 iData，至于右方则是 TXD 顶层信号。此外，一帧数据的波特率为 115200 bps。

```
1. module tx_funcmod
2. (
3.     input CLOCK, RESET,
4.     output TXD,
5.     input iCall,
6.     output oDone,
7.     input [7:0]iData
8. );
9. parameter B115K2 = 9'd434; // formula : ( 1/115200 )/( 1/50E+6 )
10.
```

以上内容为相关的出入端声明，第 9 行则是波特率为 115200 的常量声明。

```
11. reg [3:0]i;
12. reg [8:0]C1;
13. reg [10:0]D1;
14. reg rTXD;
15. reg isDone;
16.
17. always @( posedge CLOCK or negedge RESET )
18.     if( !RESET )
19.         begin
20.             i <= 4'd0;
21.             C1 <= 9'd0;
22.             D1 <= 11'd0;
23.             rTXD <= 1'b1;
24.             isDone <= 1'b0;
25.         end
```

以上内容为相关的寄存器声明以及复位操作。

```
26.     else if( iCall )
```

```
27.         case( i )
28.
29.             0:
30.                 begin D1 <= { 2'b11 , iData , 1'b0 }; i <= i + 1'b1; end
31.
32.                 1,2,3,4,5,6,7,8,9,10,11:
33.                     if( C1 == B115K2 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
34.                     else begin rTXD <= D1[i - 1]; C1 <= C1 + 1'b1; end
35.
36.             12:
37.                 begin isDone <= 1'b1; i <= i + 1'b1; end
38.
39.             13:
40.                 begin isDone <= 1'b0; i <= 4'd0; end
41.
42.         endcase
43.
```

以上内容为部分核心操作。第 26 行的 if(iCall) 表示该模块不使能就不工作。步骤 0 用来准备发送数据，其中 2'b11 是停止位与校验位(随便填)，1'b0 则是起始位。步骤 1~11 用来发送一帧数据。步骤 12~13 用来反馈完成信号并返回步骤。

```
44.     assign TXD = rTXD;
45.     assign oDone = isDone;
46.
47. endmodule
```

以上内容为驱动输出的声明。

tx_demo.v

连线部署直接看代码比较具体一点。

```
1. module tx_demo
2. (
3.     input CLOCK, RESET,
4.     output TXD
5. );
6. wire DoneU1;
7.
```

以上内容是相关的出入端声明。

```
8.      tx_funcmod U1
9.      (
10.         .CLOCK( CLOCK ),
11.         .RESET( RESET ),
12.         .TXD( TXD ),
13.         .iCall( isTX ),
14.         .oDone( DoneU1 ),
15.         .iData( D1 )
16.     );
17.
```

以上内容是 TX 功能模块的实例化，其中 isCall 由 isTX 驱动，iData 由 D 驱动。

```
18.      reg [3:0]i;
19.      reg [7:0]D1;
20.      reg isTX;
21.
22.      always @ ( posedge CLOCK or negedge RESET )
23.          if( !RESET )
24.              begin
25.                  i <= 4'd0;
26.                  D1 <= 8'd0;
27.                  isTX <= 1'b0;
28.              end
```

以上内容是相关的寄存器声明，第 23~28 行则是这些寄存器的复位操作。

```
29.      else
30.          case( i )
31.
32.              0:
33.                  if( DoneU1 ) begin isTX <= 1'b0; i <= i + 1'b1; end
34.                  else begin isTX <= 1'b1; D1 <= 8'hA1; end
35.
36.              1:
37.                  if( DoneU1 ) begin isTX <= 1'b0; i <= i + 1'b1; end
38.                  else begin isTX <= 1'b1; D1 <= 8'hA2; end
39.
40.              2:
41.                  if( DoneU1 ) begin isTX <= 1'b0; i <= i + 1'b1; end
42.                  else begin isTX <= 1'b1; D1 <= 8'hA3; end
43.
```

```
44.           3: // Stop
45.           i <= i;
46.
47.       endcase
48.
49.   endmodule
```

以上内容是核心操作的内容，步骤 0 发送数据 8'hA1，步骤 1 发送数据 8'hA2，步骤 2 发送数据 8'hA3。

编译完毕便下载程序，并且将串口线连接至电脑与开发板。打开串口调试助手，波特率设为 115200，数据位为 8，校验位随便，停止位为 1 位 ... 事后，显示方式设置为 HEX（十六进制）。当程序下载完毕以后，串口调试助手便会出现 A1，A2 与 A3。

细节一：完整的个体模块

实验十二的 TX 功能模块已经是完整的个体，可以直接拿来调用。

实验十三：串口模块② — 接收

我们在实验十二实现了串口发送，然而这章实验则要实现串口接收 ... 在此，笔者也会使用其它思路实现串口接收。

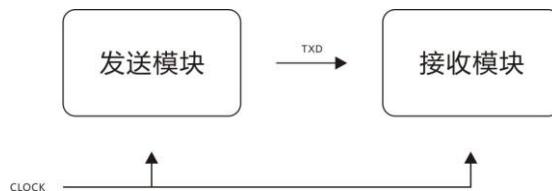


图 13.1 模块之间的数据传输。

假设我们不考虑波特率，而且一帧数据之间的传输也只是发生在 FPGA 之间，即两只模块之间互连，并且两块模块都使用相同的时钟频率，结果如图 13.1 所示。只要成立上述的假设成立，串口传输不过是简单的数据传输活动而已，图中的发送模块经由 TXD 将一帧 11 位的数据发送至接收模块。

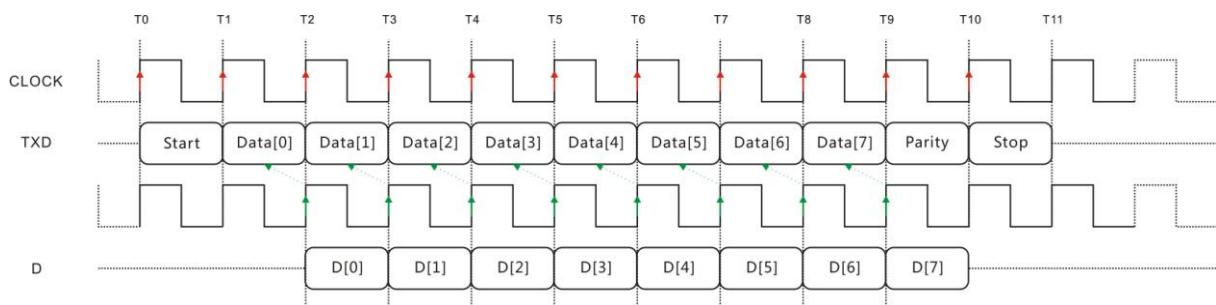


图 13.2 发送与接收一帧数据。

至于两者之间的时序过程，则如图 13.2 所示 ... 发送方经由 TXD，从 T0~T10 总共发送一帧为 11 位的数据，反之接收方则从 T2~T9 读取其中的 8 位数据而已（D 为寄存器的暂存内容）。从图 13.2 当中，我们可以看见发送方，即 TXD 都是经由上升沿发送未来值，接收方 D 则是经由上升沿读取过去值。对此，Verilog 可以这样描述，结果如代码 13.1 所示：

```
//发送方
reg [10:0]rTXD;
always @ ( posedge CLOCK )
  case ( i )
    0,1,2,3,4,5,6,7,8,9,10:
      begin TXD <= rTXD[i]; i <= i + 1'b1; end
      .....
  endcase

//接收方
```

```

reg [7:0]D1;
always @ ( posedge CLOCK )
  case ( i )
    2,3,4,5,6,7,8,9:
      begin D1[i] <= TxD; i <= i + 1'b1; end
    .....
  endcase

```

代码 13.1

如代码 13.1 所示，发送方在步骤 0~10 一共发送一帧为 11 位的数据 ... 反之接收方，则在步骤 2~9 读取其中的数据[7:0]。心机重的朋友的一定会疑惑道，为什么笔者要换个角度去思考串口怎样接收呢？原因其实很简单，目的就是为了简化理解，脑补时序，实现精密控制。

对此，FPGA 与其它设备互转数据，其实可以反映成两只模块正在互转数据，然而理想时序就是关键。因为 Verilog 无法描述理想以外的时序，对此所有时序活动都必须看成理想时序。

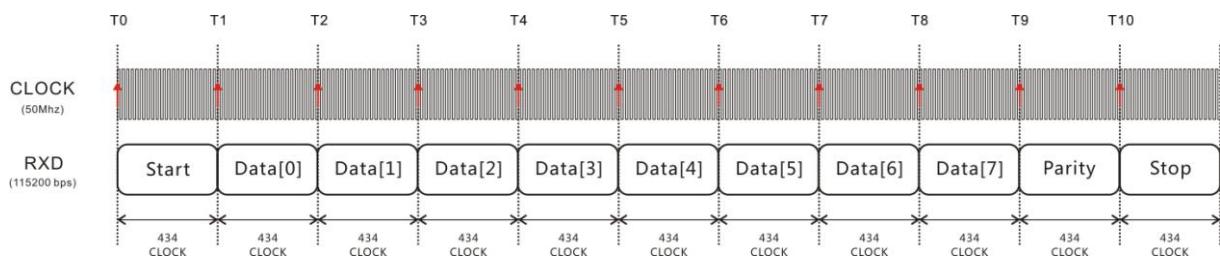


图 13.3 FPGA 接收一帧波特率为 115200 的数据。

当FPGA接收一帧数据为波特率 115200 之际，情况差不多如图 13.3 所示。50Mhz 是 FPGA 的时钟源，也是一帧数据的采集时钟，RXD 则是一帧数据的输入端。波特率为 115200 的一位数据经过 50Mhz 的时钟量化以后，每一位数据大约保持 8.68us，即 434 个时钟。串口传输没有自己的时钟信号，所以我们必须利用 FPGA 的时钟源“跟踪”每一位数据。对此，FPGA 只能借用计数器“同步跟踪”而已，至于 Verilog 则可以这样描述，结果如代码 13.2 所示：

```

0,1,2,3,4,5,6,7,8,9,10: //同步跟踪中 ...
if( C1 == 434 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
else C1 <= C1 + 1'b1;

```

代码 13.2

如代码 13.2 所示，所谓同步跟踪，就是利用计数器估计每一位数据 ... 期间，步骤 0~10 表示每一位数据，至于 C1 计数 434 个时钟则是同步跟踪中。其中 -1 考虑了步骤之间的跳转所耗掉的时钟。

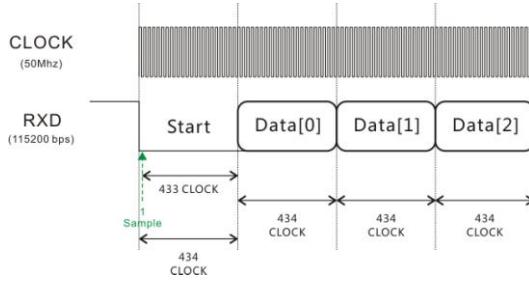


图 13.4 读取起始位。

我们都知道串口的一帧数据都是从拉低的起始位开始，然而为了完美尾行，亦即实现精密控时，起始位的读取往往都是关键。如图 13.4 所示，当我们在第一个时钟读取（采集）起始位的时候，由于 Verilog 的读取只能经过读取过去值而已，余下起始位还有 433 个时钟需要我们跟踪，为此 Verilog 可以这样描述，结果如代码 13.3 所示：

```

0:
if( RXD == 1'b0 ) begin i <= i + 1'b1; C1 <= C1 + 4'd1; end

1: // stalk start bit
if( C1 == BPS115K2 - 1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
else C1 <= C1 + 1'b1;

```

代码 13.3

如代码 13.3 所示，步骤 0 用来检测起始位，如果 RXD 的电平为拉低状态，C1 立即递增以示同步跟踪已经用掉一个时钟，同样也可以看成 i 进入下一个步骤用掉一个时钟。然而步骤 1 是用来跟踪余下的 433 个时钟，但是计数器 C1 不是从 0 开始计数，而是从 1 开始计算，因为 C1 在步骤已经递增的缘故。

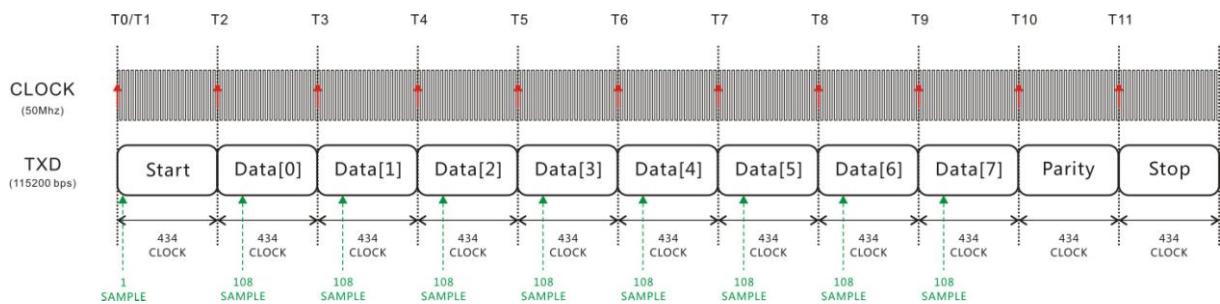


图 13.5 读取一帧数据当中的数据位。

一帧数据的跟踪结果与读取结果如图 13.5 所示 ... 除了起始位，我们使用了两个步骤采集并跟踪之余，接下来便用 8 个步骤数据一边跟踪一边采集所有数据位，然而采集的时候则是 1/4 周期，即每位数据的第 108 个时钟。最后的校验位及结束位则是跟踪而已。对此，Verilog 可以这样表示，结果如代码 13.4 所示：

```
0:
```

```

if( RXD == 1'b0 ) begin i <= i + 1'b1; C1 <= C1 + 4'd1; end

1: // start bit
if( C1 == 434 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
else C1 <= C1 + 1'b1;

2,3,4,5,6,7,8,9:
begin
    if( C1 == 108 ) D1[i-2] <= RXD;
    if( C1 == 434 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
    else C1 <= C1 + 1'b1;
end

10,11: // parity bit & stop bit
if( C1 == 434 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
else C1 <= C1 + 1'b1;

```

代码 13.4

如代码 13.4 所示，步骤 0~1 用来采集与跟踪起始位，步骤 2~9 则用来跟踪数据位，并且采集为 1/4 周期。步骤 10~11 则用来跟踪校验位于结束位。理解完毕以后，我们便可以开始建模了。

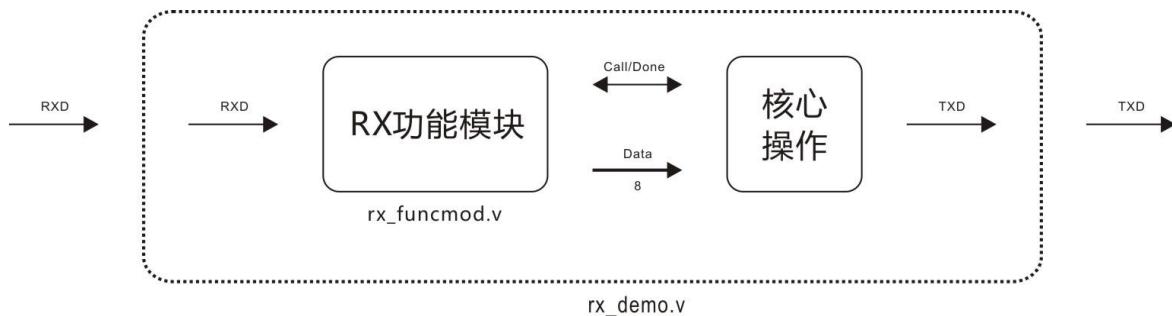


图 13.6 实验十三的建模图。

图 13.6 是实验十三的建模图，rx_demo 组合模块包含 RX 功能模块与核心操作。RX 功能模块的左方链接至 RXD 顶层信号，它主要是负责一帧数据的接收，然后反馈给核心操作。核心操作则负责 RX 功能模块的使能工作，当它领取完成信号以后，变桨收回回来的数据再经由 TXD 顶层信号发送出去。

rx_funcmod.v



图 13.7 RX 功能模块的建模图。

图 13.7 是 RX 功能模块的建模图，左方链接至顶层信号 RXD，右方则是问答信号还有 8 位的 oData。

```
1. module rx_funcmod
2. (
3.     input CLOCK, RESET,
4.     input RXD,
5.     input iCall,
6.     output oDone,
7.     output [7:0]oData
8. );
9. parameter BPS115K2 = 9'd434, SAMPLE = 9'd108;
10.
```

以上内容是相关的出入端声明，第 9 行则是波特率为 115200 的常量声明，其外还有采集的周期。

```
11. reg [3:0]i;
12. reg [8:0]C1;
13. reg [7:0]D1;
14. reg isDone;
15.
16. always @ ( posedge CLOCK or negedge RESET )
17.     if( !RESET )
18.         begin
19.             i <= 4'd0;
20.             C1 <= 9'd0;
21.             D1 <= 8'd0;
22.             isDone <= 1'b0;
23.         end
```

以上内容行是相关的寄存器声明，第 17~22 行则是这些寄存器的复位操作。

```
24.     else if( iCall )
```

```

25.           case( i )
26.
27.               0:
28.                   if( RXD == 1'b0 ) begin i <= i + 1'b1; C1 <= C1 + 4'd1; end
29.
30.               1:// start bit
31.                   if( C1 == BPS115K2 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
32.                   else C1 <= C1 + 1'b1;
33.
34.               2,3,4,5,6,7,8,9://stalk and count 1~8 data's bit , sample data at 1/2 for bps
35.                   begin
36.                       if( C1 == SAMPLE ) D1[i-2] <= RXD;
37.                       if( C1 == BPS115K2 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
38.                       else C1 <= C1 + 1'b1;
39.                   end
40.
41.               10,11:// parity bit & stop bit
42.                   if( C1 == BPS115K2 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
43.                   else C1 <= C1 + 1'b1;
44.
45.               12:
46.                   begin isDone <= 1'b1; i <= i + 1'b1; end
47.
48.               13:
49.                   begin isDone <= 1'b0; i <= 4'd0; end
50.
51.           endcase
52.

```

以上内容是核心操作。第 24 行的 if(iCall) 表示该模块不使能便不工作。步骤 0~1 用来判断与跟踪起始位；步骤 2~9 用来跟踪并且读取当中的数据位；步骤 10 至 11 则是用来跟踪校验位与停止位而已。步骤 12~13 则用来反馈完成信号，以示一次性的接收工作已经完成。

```

53.     assign oDone = isDone;
54.     assign oData = D1;
55.
56. endmodule

```

以上内容是输出驱动声明。

rx_demo.v

rx_demo 的连线布局请浏览回图 13.6 , 至于核心操作的内容请浏览代码。

```
1. module rx_demo
2. (
3.     input CLOCK, RESET,
4.     input RXD,
5.     output TXD
6. );
```

以上内容是相关的出入端声明。

```
7.     wire DoneU1;
8.     wire [7:0]DataU1;
9.
10.    rx_funcmod U1
11.    (
12.        .CLOCK( CLOCK ),
13.        .RESET( RESET ),
14.        .RXD( RXD ),      // < top
15.        .iCall( isRX ),   // < core
16.        .oDone( DoneU1 ), // > core
17.        .oData( DataU1 ) // > core
18.    );
19.
```

以上内容为是 RX 功能模块的实例化 , 第 7~8 是连线声明。

```
20. parameter B115K2 = 9'd434, TXFUNC = 5'd16;
21.
22. reg [4:0]i,Go;
23. reg [8:0]C1;
24. reg [10:0]D1;
25. reg rTXD;
26. reg isRX;
27.
28. always @ ( posedge CLOCK or negedge RESET )
29.     if( !RESET )
30.         begin
31.             i <= 5'd0;
32.             C1 <= 9'd0;
33.             D1 <= 11'd0;
```

```
34.          rTXD <= 1'b1;
35.          isRX <= 1'b0;
36.      end
```

以上内容为相关的寄存器声明以及复位操作。第 20 行是波特率为 115200 常量声明之余还有伪函数的入口地址。第 22~26 行是相关的寄存器声明，第 29~33 行则是这些寄存器的复位操作。

```
37.      else
38.          case( i )
39.
40.              0:
41.                  if( DoneU1 ) begin isRX <= 1'b0; D1 <= { 2'b11,DataU1,1'b0 }; i <= TXFUNC; Go <= 5'd0; end
42.                  else isRX <= 1'b1;
43.
44.                  *****/
45.
46.              16,17,18,19,20,21,22,23,24,25,26:
47.                  if( C1 == B115K2 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
48.                  else begin rTXD <= D1[i - 16]; C1 <= C1 + 1'b1; end
49.
50.              27:
51.                  i <= Go;
52.
53.          endcase
54.
```

以上内容为核心操作。步骤 16~27 是发送一帧数据的伪函数，笔者直接将 TX 功能整合进来。步骤 0 则是用来接收完成反馈，并且准备好发送输数，然后 i 指向伪函数。

```
55.      assign TXD = rTXD;
56.
57.  endmodule
```

以上内容是相关的输出驱动声明。编译完毕便下载程序，串口调试助手设置为 115200 波特率，8 位数据位，奇偶校验位随便，停止位 1。事后，每当串口调试助手想 FPGA 发送什么数据，FPGA 也会回馈串口调试助手，不过仅限于一帧又有间隔的数据而已。目前是实验十三还不能支持数据流的接收，因为实验十三没有空间缓冲数据流 ... 此外，核心操作没发送一帧数据也有一定的时间耽误。

细节一：完整的个体模块

实验十三的 RX 功能模块已经是完整的个体模块，可以直接拿来调用。

实验十四：储存模块

实验十四比起动手笔者更加注重原理，因为实验十四要讨论的东西，不是其它而是低级建模 II 之一的模块类，即储存模块。接触顺序语言之际，“储存”不禁让人联想到变量或者数组，结果它们好比数据的暂存空间。

```
1. int main()
2. {
3.     int VarA;
4.     char VarB;
5.     VarA = 20;
6.     VarB = 5;
7. }
```

代码 14.1

如代码 14.1 所示，主函数内一共声明两个变量 VarA 与 VarB（第 3~4 行）。VarA 是两个字节的整型变量，VarB 是一个字节的字符变量，然后 VarA 赋值 20（第 5 行），VarB 则赋值 5（第 6 行），其中 int 与 char 等字眼用来表示字节，即暂存空间的位宽，然后储存的内容仅局限于二进制，非 0 即 1。

```
1. int main()
2. {
3.     int VarC[20];
4.     VarC[0] = 30;
5.     for( int i = 0; i < 20; i++ ) VarC[i] = i;
6.     VarC[0] = VarC[1];
7. }
```

代码 14.2

除了变量以外，顺序语言也有数组这个玩意，亦即一连串的变量。如代码 14.2 所示，主函数内声明数组 VarC，数组的成员位宽是两个字节的 int，数组的成员长度则是 20（第 3 行）。然而数组常见的赋值方法除了成员直接赋值以外（第 4 行），也有使用 for 循环为逐个成员赋值的方法（第 5 行）。此外，还有某个数组成员为某个数组成员直接赋值的方法（第 6 行）。目前为止，顺序语言还有储存之间的故事就圆满结束。

人是一种自虐的生物，事情越是顺利，越是容易萌起疑心 … 然后暗道：“储存是不是太容易理解呢？容易到让人觉得恶心！”。没错，事实的确如此。“储存”一旦投入描述语言之中，话题便会严肃起来。顺序语言是一件懒人多得的语言，它有许多底层工作都交由编译器处理，相较之下描述语言是一件多劳多得的语言，许多底层工作都必须交由我们自己声明与定义。

```
1. reg [3:0]D1 = 4' d1;
```

```

2. reg [3:0]D2 ;
3. reg [3:0]D3;
4.
5. initial begin D2 = 4' d2; end
6.
7. always @ ( posedge CLOCK or negedge RESET )
8.     if( !RESET )
9.         D3 <= 4' d3;
10.    .....

```

代码 14.3

首先让我们来理解一下初始化与复位化之间的区别。我们知道顺序语言的变量只有初始化，没有复位化这一回事 ... 反之，描述语言却不同。如代码 14.3 所示，笔者在第 1~3 行声明 D1~D3 三个寄存器，其中 D1 声明不久便立即赋予初值 4'd1。换之，D2 则在第 5 行赋予初值 4'd2，最后 D3 则在第 8~9 行赋予复位值 4'd3。

所谓初值就是初始化所给予的起始内容，反之复位值就是复位触发所给予的内容。初始化一般都是编译器的赋值活动，第 1 行的 D1 还有第 5 行的 D2 都是经由编译器的活动给予初值。反观之下，复位化不是编译器活动而是硬件活动，也是俗称的 RESET，即电平变化所引起的复位触发。

如代码行第 7~9 所示，敏感区种含有 negedge RESET 的字眼表示，如果 RESET 的电平由高变低并且产生下降沿触发，结果就执行一下 always 的内容。其中的内容便是复位操作，最终 D3 赋予复位值 4'd3。

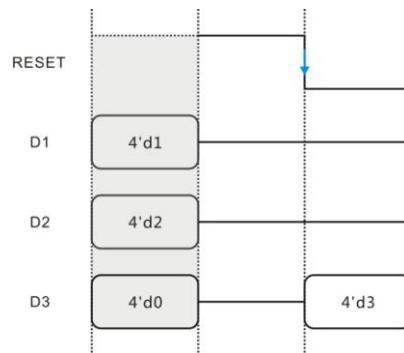


图 14.1 初始化与复位化的时序图。

如果用时序来表示的话 ... 如图 14.1 所示，灰色区域表示初始化状态又或者未上电状态，当中 D1 与 D2 都赋予初值 4'd1 与 4'd2，同样 D3 也给予初值 4'd0。虽然 D3 在代码 14.3 之间并没有任何初始化的痕迹，不过默认下编译器都会一视同仁，将所有暂存声明都给予初值 0，除非有特别声明，例如第 1 行的 D1 与第 5 行的 D2。上电以后，RESET 电平又高变低便产生下降沿，结果复位发生了，然后 D3 被赋予复位值 4'd3。

我们知道容器有大有小，所以储存空间也有大小之分，然而决定空间大小就是位宽这种

东西。位宽一般是指数据的长度，顺序语言会用 int 或者 char 等关键字表示位宽，反之描述语言会直接声明位宽的大小，如 reg[3:0]D1。在此，顺序语言的位宽区别都是一个字节一个字节比较，反之描述语言比较随意。

```
1. reg [3:0]D1; // Verilog  
2. reg var [3:0]D2; // System Verilog  
3. logic var [3:0]D3; // System Verilog
```

代码 14.4

除了位宽以外，我们还要理解什么是储存内容。描述语言相较顺序语言，储存内容的花样较多一些 ... 顺序语言由于比较偏向软件，所以储存内容仅有两态，即 1 与 0 而已。反之描述语言是介于硬件与软件之间，所以储存内容除了 0 与 1 两态之外，也有高阻态 z 与切开 x。

如代码 14.4 所示，当我们声明 D1 的时候，除了需要强调位宽以外，我们还要强调储存内容 ... 以 Verilog 为例的话，关键字 reg 的用意并非强调储存资源是基于寄存器，而是表示储存内容有 0 有 1，还有 z 与 x 等四个状态。相反的，SystemVerilog 在这方面却做得很好，如代码行 2~3 所示，var 关键字表示对象是储存空间，reg 关键字表示对象的储存内容有 4 态，logic 关键字则表示对象的储存内容有 2 态。

```
1. char VarA; // 变量(内存)  
2. char VarB[4] // 数组(内存)  
3. reg [7:0] D1; // 寄存器  
4. reg [31:0] D2; // 寄存器
```

代码 14.5

我们知道顺序语言有所谓的变量与数组，储存资源一般都是基于内存，例如第 1 行的 VarA 与第 2 行的 VarB。反之，描述语言不仅可以用寄存器资源建立变量，寄存器资源也能建立数组，例如与第 3 行的 D1 与第 4 行的 D2。[（虽然顺序语言偶尔也会用到寄存器型的储存资源，不过该储存资源对处理器来说太珍贵了，如果不是特殊条件，一般都不会随意使用）](#)

```
1. reg [3:0] RAM [15:0]; // 片上内存
```

代码 14.6

此外，描述语言还有另外一种叫做片上内存的储存资源，声明方法如代码 14.6 所示。FPGA 的片上内存与单片机的内存虽然都是内存，但是两者之间却是不同性质的内存。简单而言，单片机的内存是经过烧烤的熟肉，随时可以享用 ... 反之，FPGA 的片上内存则是未经过烧烤的生肉，享用之前必须做好事先准备。为此，FPGA 的片上内存无法像级单片机内存那样，随便赋值，随意调用。

```
1. int main()  
2. {  
3.     char VarC[4];
```

```
4.     for( int i = 0; i < 3; i++ ) VarC[i] = VarC[i+1]
5. }
```

代码 14.7

如代码 14.7 所示，笔者先建立一个 char 类型的数组 VarC 长度并且为 4，紧接着利用 for 循环为数组的整组成员赋值，其中 VarC[i] 的赋予内容是 VarC[i+1] 的结果。代码 14.7 算是顺序语言常见的例子，期间初始化也好，还是利用 for 循环为数组赋值也好，许多底层的工作都交由编译器去作，我们只要翘脚把代码照顾好就行。

```
1. reg [7:0] RAM [3:0]
2. reg [3:0]i;
3.
4. always @ ( posedge CLOCK ) // 错误
5.     for( i = 0; i < 3; i = i + 1 ) RAM[i] = RAM[i+1];
```

代码 14.8

换做描述语言，如代码 14.8 所示 ... 笔者在第 1~2 行当中先声明位宽为 8 长度为 4 的 RAM，随之又声明 i。假设 RAM 要实现代码 14.7 同样的赋值操作，首先最常见的错误就是第 4~5 行的例子 ... 许多人会直接将关键字 for 套用在 always 块里，这种赋值操作有两种问题：

其一，编译器并不清楚我们到底要利用[空间实现 for](#)，还是利用[时钟实现 for](#)。默认下，编译器会选择前者，后果就是吃光逻辑资源。

其二，RAM[i] = RAM[i+1] 这种赋值操作会搞砸综合，结果片上内存的布线状况会变得非常复杂，从而导致综合失败。

代码 14.8 算是新手最容易犯下的问题之一，代码 14.8 虽然没有语法上的错误，而且仿真也会通过，但是综合却万万不可。为此，代码 14.8 需要更动一下。

```
1. reg [7:0] RAM [3:0]
2. reg [3:0]i;
3.
4. always @ ( posedge CLOCK ) // 错误
5.     case( i )
6.         0,1,2:
7.             begin RAM[i] <= RAM[i+1]; i <= i + 1' b1; end
8.     endcase
```

代码 14.9

如代码 14.9 所示，笔者舍弃关键字 for，取而代之却利用仿顺序操作充当循环，这是一种利用时钟实现 for 的方法（伪循环）。不过代码 14.9 依然不被综合器接受，结果报错 ... 因为片上内存并不支持类似 RAM[i] <= RAM[i+1] 的赋值方式，因为综合期间会导致布

线复杂化，并且进一步搞砸综合。为此，代码 14.9 需要继续更动。

```
1. reg [7:0] RAM [3:0]
2. reg [3:0]i;
3. reg [7:0]D1;
4.
5. always @ ( posedge CLOCK ) // 正确
6.   case( i )
7.     0,2,4:
8.       begin D1 <= RAM[i<<1]; i <= i + 1' b1; end
9.     1,3,5:
10.      begin RAM[ (i<<1)+1 ] <= D1; i <= i + 1' b1; end
11.    endcase
```

代码 14.10

如代码 14.10 所示，笔者多建立一个作为暂存作用的寄存器 D1，然后利用两组步骤移动 RAM 之间的数据。步骤 0，2 与 4 将 RAM[i] 的内容暂存至 D1，步骤 1，3 与 5 则将 D1 的内容赋予 RAM[i+1]。如此一来，片上内存成员与成员之间的数据移动便大功告成。事实上代码 14.7 也干同样的事情，不过事实却被编译器隐藏了 ... 如果读者打开代码 14.7 的编译结果，读者会看见类似的汇编语言，结果如代码 14.11 所示：

```
T0 Load RAM[1] => R0;
T1 Load R0 => RAM[0];
T2 Load RAM[2] => R0;
T3 Load R0 => RAM[1];
T4 Load RAM[3] => R0;
T5 Load R0 => RAM[2];
```

代码 14.11

如代码 14.11 所示，汇编内容会重复使用 Load 指令将某个 RAM 的内容先暂存至通用寄存器 R0，然后又从 R0 移至另某个 RAM 当中。至于代码 14.11 的正确性，笔者不能确保什么，毕竟距离上一次接触汇编语言已经是 N 年前的事情。不过感觉上差不多就是那样 ... 这就是被编译器所隐藏的底层工作之一，代码 14.10 不过是将其模仿而已。

讲到这里，我们开始接触重点了。上述的例子告诉我们，编译器不会帮忙描述语言处理底层操作。所以，变量与数组之间的储存操作不及顺序语言那么便捷，而且模仿起来也非常麻烦 ... 不过，我们也不用那么灰心，良驹有良驹的跑法，歪驹有歪驹的走法，我们只要换个角度去面对情况，不视问题，问题自然迎刃而解。

根据笔者的妄想，储存有“储存资源”还有“储存方式”之分。描述语言可用的储存资源有寄存器还有片上内存，然而变量与数组也是最简单也是最基础的“储存方式”。基于这些 ... 事实上，描述语言可以描述各种各样的“储存方式”。

```

1. module rom( input [1:0]iAddr, output [7:0]oData );
2.     reg [7:0]D1;
3.     always @ (*)
4.         if( iAddr == 2' b00 ) D1 = 8' hA;
5.         else if( iAddr == 2' b01 ) D1 = 8' hB;
6.         else if( iAddr == 2' b10 ) D1 = 8' hC;
7.         else if( iAddr == 2' b11 ) D1 = 8' hD;
8.         else D1 = 8' dx;
9.
10.    assign oData = D1;
11.
12. endmodule

```

代码 14.12

例如一个简单的静态 ROM 模块，它可以基于寄存器或者片上内存，结果如代码 14.12 与 14.13 所示。代码 14.12 是基于寄存器的静态 ROM，它有 2 位 iAddr 与 8 位的 oData，其中第 3~8 行是 ROM 的内容定义，第 10 行则是输出驱动，为此 oData 会根据 iAddr 的输入产生不同的输出。

```

1. module rom( input [1:0]iAddr, output [7:0]oData );
2.     reg [7:0] RAM [3:0];
3.     initial begin
4.         RAM[0] = 8' hA;
5.         RAM[1] = 8' hB;
6.         RAM[2] = 8' hC;
7.         RAM[3] = 8' hD;
8.     end
9.
10.    assign oData = RAM[ iAddr ];
11.
12. endmodule

```

代码 14.13

反之，代码 14.13 是基于片上内存的静态 ROM，它也有 2 位 iAddr 与 8 位 oData，第 3~7 行是内容的定义也是初始化片上内存，第 10 行则是输出驱动，oData 会根据 iAddr 的输出产生不同的输出。

代码 14.12 与代码 14.13 虽然都是静态 ROM，不过却有根本性的不同，因为两者源于不同的储存资源，其中最大的证据就是第 10 行的输出驱动，前者由寄存器驱动，后者则由片上内存驱动。不同的储存资源也有不同的性质，例如寄存器操作简单，而且布线有余，不过不支持大容量的储存行为。换之，片上内存虽然操作麻烦，布线也紧凑，可是却支持大容量的储存行为。

储存方式相较储存资源理解起来稍微抽象一点，而且想象范围也非常广大 ... 如果储存资源是“容器的种类”，那么储存方式就是“容器的用法”。举例而言，一个简单静态 ROM，根据需要它还可以演变成为其它亚种，例如常见的单口 ROM 或者双口 ROM 或等。

```
1. module rom( input CLOCK , input [1:0]iAddr, output [7:0]oData );
2.     reg [7:0] RAM [3:0];
3.     initial begin
4.         RAM[0] = 8' hA;
5.         RAM[1] = 8' hB;
6.         RAM[2] = 8' hC;
7.         RAM[3] = 8' hD;
8.     end
9.
10.    reg [1:0] D1;
11.    always @ ( posedge CLOCK )
12.        D1 <= iAddr;
13.
14.    assign oData = RAM[ D1 ];
15.
16. endmodule
```

代码 14.14

如代码 14.14 所示，那是单口 ROM 的典型例子，然而单口 ROM 与静态 ROM 之间的差别就在于前者有时钟信号，后者没有时钟信号。期间，代码 14.14 用 D1 暂存 iAddr，然后再由 D1 充当 RAM 的寻址工具。

```
1. module rom( input CLOCK , input [1:0]iAddr1, iAddr2 , output [7:0]oData1 , oData2 );
2.     reg [7:0] RAM [3:0];
3.     initial begin
4.         RAM[0] = 8' hA;
5.         RAM[1] = 8' hB;
6.         RAM[2] = 8' hC;
7.         RAM[3] = 8' hD;
8.     end
9.
10.    reg [1:0] D1;
11.    always @ ( posedge CLOCK )
12.        D1 <= iAddr1;
13.
14.    assign oData1 = RAM[ D1 ];
15.
16.    reg [1:0] D2;
```

```

17.      always @ ( posedge CLOCK )
18.          D2 <= iAddr2;
19.
20.      assign oData2 = RAM[ D2 ];
21.
22. endmodule

```

代码 14.15

如代码 14.15 所示，那是双口 ROM 的典型例子，如果将其比较单口 ROM，它则多了一组 iAddr 与 oData 而已，即 iAddr1 与 oData1，iAddr2 与 oData2。第 10~14 行是第一组（第一口），第 16~20 行则是第二组（第二口），不过两组 iAddr 与 oData 都从同样的 RAM 资源哪里读取结果。

事实上，ROM 还会根据更多不同要求产生更多亚种，而且**亚种的种类也绝非局限在于专业规范**，因为亚种的储存模块会依照设计者的欲望——有多畸形就多畸形，死守传统只会固步自封而已。无论模块对象是静态 ROM，单口 ROM 还是双口 ROM 等 ... 笔者眼中，它们都是任意的“储存方式”而已。

根据笔者的妄想，储存方式的覆盖范围非常之广。简单而言，凡是模块涉及数据的储存操作，低级建模 II 都视为储存类。举例而言，ROM 模块储存自读不写的数据；RAM 模块储存又读又写的数据；FIFO 模块储存先写先读的数据。

为此，我们可以这样命名它们：

```

rom_savemod.v // rom 储存模块
ram_savemod.v // ram 储存模块
fifo_savemod.v // fifo 储存模块

```

好奇的朋友一定会觉得疑惑，笔者究竟是为了定义储存类呢？事情说来话长，笔者也是经过多番考虑以后才狠下心肠去决定的。首先，让我们继续从顺序语言的角度去理解吧：

```

1.  unsigned char Variable;
2.  void FunctionA( unsinged char A ) { Variable = A; }
3.  unsinged char FunctionB( void ) { return Variable; }
4.  int main()
5.  {
6.      unsigned char D1;
7.      FunctionA( 0x0A );
8.      D1 =FunctionB();
9.      .....
10. }

```

代码 14.16

假设有 N 个函数想共享数据，一般而言我们都会建立全局变量（数组）。如代码 14.16 所示，笔者先建立全局变量 Variable，然后又声明函数 A 为 Variable 赋值，反之函数 B 则返回 Variable 的内容。完后，再编辑主函数的操作 ... 期间，主函数先声明变量 D，然后调用函数 A，并且传递参数 0x0A，完后便调用函数 B，并且将返回的内容赋予 D。函数之间之所以可以共享数据，那是因为编译器在后面大力帮忙，并且处理底层操作才得以实现。换之，描述语言虽然没有类似的好处，但是描述语言可以模仿。

```
1. reg [7:0]Variable;
2. reg [7:0]T,D1;
3. reg [3:0]i,Go;
4. always @ ( posedge CLOCK ) // 核心操作
5.     case(i)
6.         0: // 主操作
7.             begin T <= 8' h0A; i <= 4' d8; Go <= i + 1' b1; end
8.         1:
9.             begin i <= 4' d9; Go <= i + 1' b1; end
10.        2:
11.            begin D1 <= T; i <= i + 1' b1; end
12.        .....
13.        8:// Fake Function A 伪函数 A
14.        begin Variable = T; i <= Go; end
15.        9:// Fake Function B 伪函数 B
16.        begin T = Variable; i <= Go; end
17.    endcase
```

代码 14.17

如代码 14.17 所示，笔者先建立 Variable，然后又建立 T 与 D，还有 i 与 Go。Variable 模仿全局变量，T 则是伪函数的暂存空间（数据传递），i 指向步骤，Go 则是指向返回步骤。步骤 0~2，我们可以视为主函数，步骤 8~9 则是伪函数 A 与伪函数 B。

步骤 0，i 将指向伪函数 A 的入口，T 赋予 8'h0A，Go 则指向下一个步骤。

步骤 8，Variable 赋予 T 的内容，然后返回步骤。

步骤 1，i 将指向伪函数 B 的入口，Go 则指向下一个步骤。

步骤 9，T 赋予 Varibale 的内容，然后返回步骤。

步骤 2，D1 赋予 Varibale 的内容，然后操作结束。

如果我们将代码 14.16 与代码 14.17 互相比较的话，它们存在几处区别甚微的地方。

其一，代码 14.17 的代码量比代码 14.16 还要多；

其二，代码 14.16 的 Variable 是真正意义上的全局变量，反之代码 14.17 则是山寨。

除此之外，代码 14.17 还是一只核心操作组成，或者代码 14.17 是有一只函数而已。

如果主函数，函数 A 还有函数 B 之间只有简单操作，而且数据的传递量也不多的话，那么仅有一只核心操作也没有什么问题。相反的，如果函数之间不仅有复杂的操作，而且数据的传递量也很多的话，独秀的核心操作就要举白旗投降了。为此，我们必须借助多模块的力量来解决复杂的操作，但是多模块之间又如何共享数据呢？首先，让我们换个思路思考问题。

```

1. unsigned char Variable;      // 储存类
2. void FunctionA( unsinged char A ) { Variable = A; }    // 功能类
3. unsinged char FunctionB( void ) { return Variable; } // 功能类
4. int main() { ..... } // 控制类

```

代码 14.18

如代码 14.18 所示，全局变量视为储存类，函数 A 与函数 B 视为功能类，至于主函数视为控制类。

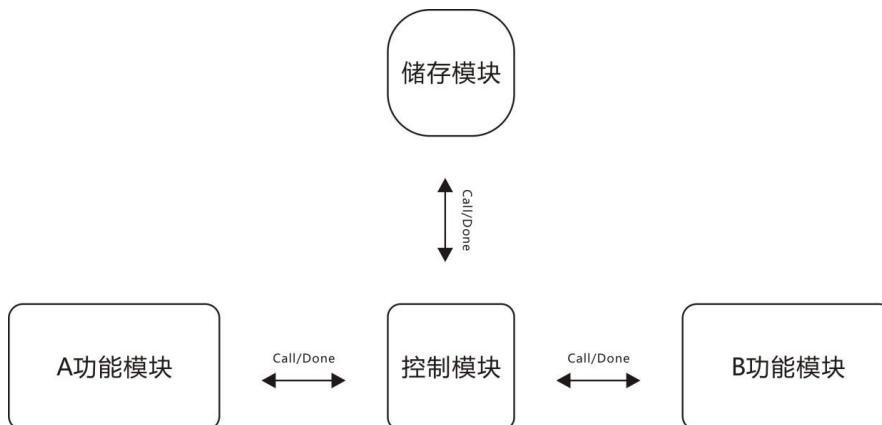


图 14.2 代码 14.18 的建模图。

代码 14.18 经过分类以后，大致的建模布局如图 14.2 所示。一只名为 main 的控制模块充当中介，次序调度，协调者等角色。其中，A 功能模块与 B 功能模块负责最基本的操作，variable 储存模块则负责储存操作。余下，所有模块都经由问答信号联系起来，至于 Verilog 则可以这样表示：

```

1. module ( ... );
2.
3.     wire [2:0]CallU1 ;
4.     main_ctrlmod U1
5.     (
6.         .oCall( CallU1 ),
7.         .iDone( { DoneU1, DoneU2, DoneU3 } ),
8.         ...
9.     );

```

```

10.
11.     wire DoneU2;
12.     a_funcmod U2
13.     (
14.         .iCall( CallU1[0] ),
15.         .oDone( DoneU2 ),
16.         ...
17.     );
18.
19.     wire DoneU3;
20.     b_funcmod U3
21.     (
22.         .iCall( CallU1[1] ),
23.         .oDone( DoneU3 ),
24.         ...
25.     );
26.
27.     wire DoneU4;
28.     varibale_savemod U1
29.     (
30.         .iCall( CallU1[2] ),
31.         .oDone( DoneU4 ),
32.         ...
33.     );
34.
35. endmodule

```

代码 14.18

如代码 14.18 所示，组合模块的内容包含，main 控制模块为实例 U1，a 功能模块与 b 功能模块为实例 U2~U3，variable 储存模块为实例 U4。最后，各个模块经由问答信号 Call/Done 联系起来。

前面的例子告诉我们，描述语言在变量上的运用，远远不及顺序语言那么便捷，毕竟描述语言没有底层补助，而且模仿它人也超麻烦。话虽如此，这是描述语言的缺点也是优点 ... 优点？笔者有没有搞错？那么麻烦还称为优点，笔者是不是脑子进水了？这位同学别猴急，笔者会慢慢解释的。

```

1. unsigned char LUT[4] = { 10, 20, 30, 40 };
2. int main()
3. {
4.     int D1;
5.     D1 = LUT[1] + LUT[2];
6.     ...

```

如代码 14.19 所示，第 1 行声明位宽为 8，长度为 4 的 LUT 查表，第 2~7 行则是查表的运用。表面上，顺序语言虽有惊人的便捷性，不过底子里却是一片死残，尤其是时钟的利用率更是惨不忍睹。**那些写过算法的同学一定知道，查表常常用来优化算法的运算速度 ...** 简单来说，查表就是顺序语言“空间换速度”的优化手段。

查表既是 ROM 也是一种储存方式。如果把话说难听一点，所谓查表也不过是顺序语言在利用数组模仿 ROM 而已，它除了便捷性好以外，无论是资源的消耗，还是时钟的消耗等效率都远远不及描述语言的 ROM。**顺序语言偶尔虽然也有山寨的 FIFO, Shift 等储存方式，不过性能却是差强人意。**

顺序语言之所以那么逊色，那是因为被钢铁一般坚固的顺序结构绑得死死。述语言是自由的语言，结构也是自由。虽然自由结构为人们带来许多麻烦，但是“储存方式”可以描述的范畴，绝对超乎人们的估量。归根究底，究竟是顺序语言好，还是描述语言模比较厉害呢？除了见仁见智以外，答案也只有天知晓。

随着时代不断变迁，“储存方式”的需求也逐渐成长，例如 50 年代需要 rom，60 年代需要 ram，70 年代需要 fifo。二十一世纪的今天，保守的规范再也无法压抑“储存方式”的放肆衍生，例如 rom 衍生出来静态 rom，单口 rom，双口 rom 等许多亚种；此外，fifo 也衍生出同步 fifo 或者异步 fifo 等亚种。至于 ram 的亚种，比前两者更加恐怖！不管怎么样，大伙都是笔者的好孩子，亦即 `xx_savemod`。

虽然伟大的官方早已准备数之不尽的储存模块，但是笔者还是强调**手动建模**比较好，因为官方的东西有太多限制了。此刻，可能有人跳出来反驳道：“为什么不用官方插件模块，它们既完整又便捷，那个白痴才不吃天上掉下来的馅饼！笔者是呆子！蠢货！”。话说这位同学也别那么激动，如果读者一路索取它人的东西，学习只会本末倒置而已。

除此之外，官方插件模块是商业的产物，不仅自定义有限内容也是隐性，而且还是不择不扣的快餐。快餐即美味也方便，偶尔吃下还不错，但是长期食用就会危害健康，危害学习。

“fifo 插件的数据位宽能不能设为 11 位？”，某人求救道。

“ram 插件怎样调用？怎样仿真？”，某人求救道。

类似问题每月至少出现数十次，而且还是快餐爱好者提问的。笔者也有类似的经验，所以非常明白这种心境。年轻的笔者就是爱好快餐，凡事拿来主义，伸手比吃饭更多。渐渐地，笔者愈来愈懒，能不增反降，最终变成只会求救的肥仔而已。后悔以后，笔者才脚踏实地自力建模，慢慢减肥。

在此，笔者滔滔不绝只想告知读者 ... 自由结构虽然麻烦，不过这是将想象力具体化的

关键因素，储存模块的潜能远超保守的规范。规范有时候就像一粒绊脚石，让人不经意跌倒一次又一次，阻碍人们前进，限制人们想象，最后让人成为不动手即不动脑的懒人。最后，让我们建立一只不规格又畸形的储存模块作为本实验的句号。

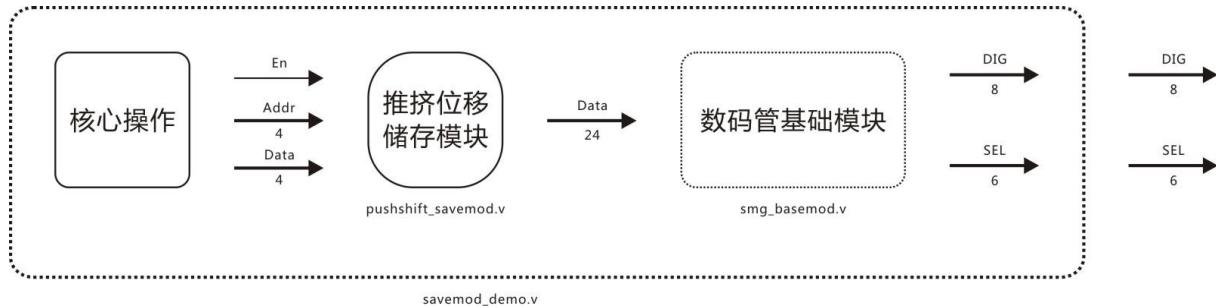


图 14.3 实验十四的建模图。

图 14.3 是实验十四的建模图，组合模块 savemod_demo 的内容包括一支核心操作，一只数码管基础模块，还有一只名字帅到掉渣的储存模块。核心操作会拉高 oEn，并且将相关的 Addr 与 Data 写入储存模块，紧接着该储存模块会经由 oData 驱动数码管基础模块。事不宜迟，让我们先来瞧瞧推挤位移储存模块这位帅哥。

pushshift_savemod.v



图 14.4 推挤位移储存模块的建模图。

顾名思义，该模块是推挤功能再加上位移功能的储存模块，左边是储存模块常见的 iEn，iAddr 与 iData，右边则是超乎常规的 oData。

```

1. module pushshift_savemod
2. (
3.     input CLOCK,RESET,
4.     input iEn,
5.     input [3:0]iAddr,
6.     input [3:0]iData,
7.     output [23:0]oData
8. );

```

第 3~7 行是相关的出入端声明。

```

9.      reg [3:0] RAM [15:0];
10.     reg [23:0] D1;
11.
12.     always @ ( posedge CLOCK or negedge RESET )
13.         if( !RESET )
14.             begin
15.                 D1 <= 24'd0;
16.             end

```

第 9 行是片上内存 RAM 的声明 , 第 10 行则是寄存器 D1 的声明。第 15 行则是 D1 的复位操作。

```

17.           else if( iEn )
18.               begin
19.                   RAM[ iAddr ] <= iData;
20.                   D1[3:0] <= RAM[ iAddr ];
21.                   D1[7:4] <= D1[3:0];
22.                   D1[11:8] <= D1[7:4];
23.                   D1[15:12] <= D1[11:8];
24.                   D1[19:16] <= D1[15:12];
25.                   D1[23:20] <= D1[19:16];
26.               end
27.
28.           assign oData = D1;
29.
30. endmodule

```

第 17 行表示 iEn 不拉高该模块就不工作。第 18~26 行是该模块的核心操作 , 第 19 行表示 RAM 将 iData 储存至 iAddr 指定的位置 ; 第 20 行表示 , RAM 将 iAddr 指定的内容赋予 D1[3:0]。如此一来 , 第 19 行与第 20 行的结合就成为推挤功能。至于第 21~25 行则是 6 个深度的位移功能 (即 4 位宽为一个深度) , iEn 每拉高一个时钟 , D1 的内容就向左移动一个深度。

savemod_demo.v

该组合模块的连线部署根据图 14.3 , 具体内容我们还是来看代码吧。

```

1. module savemod_demo
2. (
3.     input CLOCK,RESET,
4.     output [7:0]DIG,
5.     output [5:0]SEL

```

```
6. );
```

以上内容是相关的出入端声明。

```
7.     reg [3:0]i;
8.     reg [3:0]D1,D2; // D1 for Address, D2 for Data
9.     reg isEn;
10.
11.    always @ ( posedge CLOCK or negedge RESET )// Core
12.        if( !RESET )
13.            begin
14.                i <= 4'd0;
15.                { D1,D2 } <= 8'd0;
16.                isEn <= 1'b0;
17.            end
18.        else
```

以上内容是相关的寄存器声明以及复位操作。其中 D1 用来暂存地址数据，D2 用来暂存读写数据。第 12~17 行是这些寄存器的复位操作。

```
19.         case( i )
20.
21.             0:
22.                 begin isEn <= 1'b1; D1 <= 4'd0; D2 <= 4'hA; i <= i + 1'b1; end
23.
24.             1:
25.                 begin isEn <= 1'b1; D1 <= 4'd0; D2 <= 4'hB; i <= i + 1'b1; end
26.
27.             2:
28.                 begin isEn <= 1'b1; D1 <= 4'd0; D2 <= 4'hC; i <= i + 1'b1; end
29.
30.             3:
31.                 begin isEn <= 1'b1; D1 <= 4'd0; D2 <= 4'hD; i <= i + 1'b1; end
32.
33.             4:
34.                 begin isEn <= 1'b1; D1 <= 4'd0; D2 <= 4'hE; i <= i + 1'b1; end
35.
36.             5:
37.                 begin isEn <= 1'b1; D1 <= 4'd0; D2 <= 4'hF; i <= i + 1'b1; end
38.
39.             6:
40.                 begin isEn <= 1'b1; D1 <= 4'd0; D2 <= 4'h0; i <= i + 1'b1; end
41.
```

```

42.          7:
43.          begin isEn <= 1'b0; i <= i; end
44.
45.          endcase
46.

```

以上内容为核心操作，操作过程如下：

步骤 0 为地址 0 写入数据 4'hA；，将原本的数据挤出来，并且发生位移。

步骤 1 为地址 0 写入数据 4'hB；，将 4'hA 挤出来，并且发生位移。

步骤 2 为地址 0 写入数据 4'hC；，将 4'hB 挤出来，并且发生位移。

步骤 3 为地址 0 写入数据 4'hD；，将 4'hC 挤出来，并且发生位移。

步骤 4 为地址 0 写入数据 4'hE；，将 4'hD 挤出来，并且发生位移。

步骤 5 为地址 0 写入数据 4'hF，将 4'hE 挤出来，并且发生位移。

步骤 6 为地址 0 写入数据 4'd0，将 4'hF 挤出来，并且发生位移。

步骤 7 结束操作。

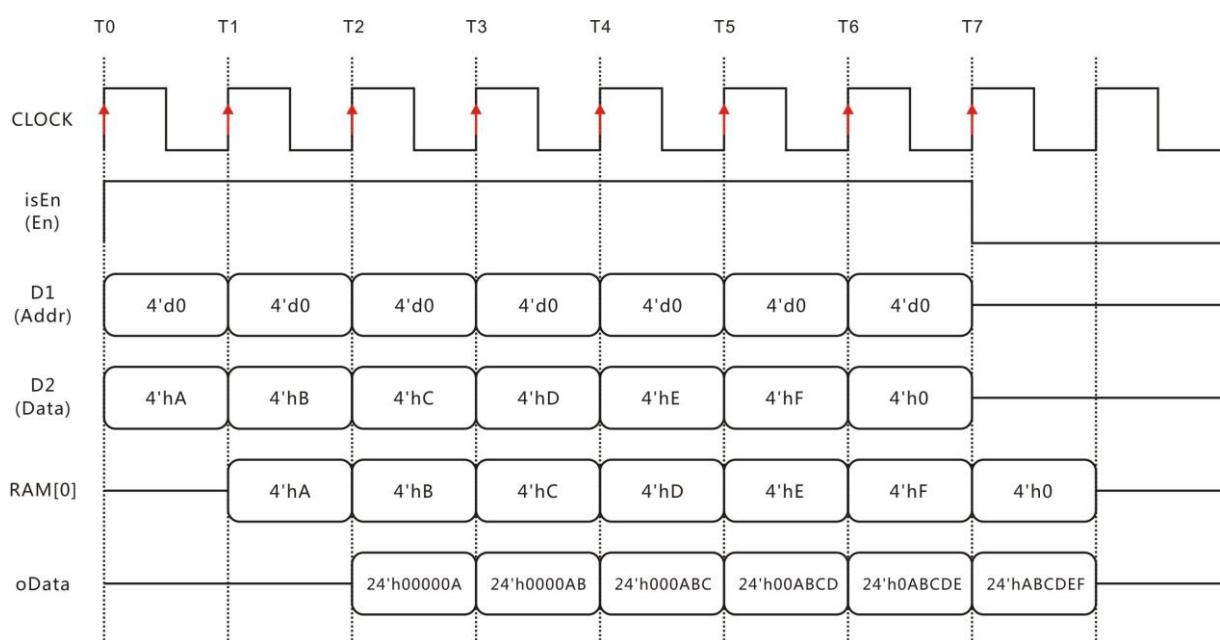


图 14.5 savemod_demo 部分时序图。

图 14.5 是 savemod_demo 部分重要的理想时序图，其中 isEn，D1 与 D2 是核心操作所发送的数据，至于 RAM[0]与 oData 是推挤位移储存模块的内部状况与输出结果。时序过程如下：

T0，核心操作拉高 isEn，发送 4'd0 地址数据与 4'hA 读写数据。

T1，核心操作拉高 isEn，发送 4'd0 地址数据与 4'hB 读写数据。储存模块将 4'hA 载入地址 0。

T2，核心操作拉高 isEn，发送 4'd0 地址数据与 4'hC 读写数据。储存模块将 4'hB 载入地址 0，并且将数据 4'hA 挤出，oData 的结果为 24'h00000A。

T3，核心操作拉高 isEn，发送 4'd0 地址数据与 4'hD 读写数据。储存模块将 4'hC 载入地址 0，并且将数据 4'hB 挤出，同时发生位移，oData 的结果为 24'h0000AB。

T4，核心操作拉高 isEn，发送 4'd0 地址数据与 4'hE 读写数据。储存模块将 4'hD 载入地址 0，并且将数据 4'hC 挤出，同时发生位移，oData 的结果为 24'h000ABC。

T5，核心操作拉高 isEn，发送 4'd0 地址数据与 4'hF 读写数据。储存模块将 4'hE 载入地址 0，并且将数据 4'hD 挤出，同时发生位移，oData 的结果为 24'h00ABCD。

T6，核心操作拉高 isEn，发送 4'd0 地址数据与 4'd0 读写数据。储存模块将 4'hF 载入地址 0，并且将数据 4'hE 挤出，同时发生位移，oData 的结果为 24'h0ABCDE。

T7，储存模块将 4'd0 载入地址 0，并且将数据 4'hF 挤出，同时发生位移，oData 的结果为 24'hABCDEF。

```
47.      wire [23:0]DataU1;
48.
49.      pushshift_savemod U1
50.      (
51.          .CLOCK( CLOCK ),
52.          .RESET( RESET ),
53.          .iEn( isEn ), // < Core
54.          .iAddr( D1 ), // < Core
55.          .iData( D2 ), // < Core
56.          .oData( DataU1 )// > U2
57.      );
58.
```

第 47~58 行是该储存模块的实例化。

```
59.      smg_basemod U2
60.      (
61.          .CLOCK( CLOCK ),
62.          .RESET( RESET ),
63.          .DIG( DIG ),    // top
64.          .SEL( SEL ),    // top
65.          .iData( DataU1 ) // < U1
66.      );
67.
68. endmodule
```

第 59~66 行是数码管基础模块的实例化。编译完毕便下载程序，如果数码管从左至右显示“ABCDEF”，那么表示实验成功。最后还是要强调一下，推挤位移目前是没有意义的储存模块，可是实验十四的目的也非常清楚，就是解释储存模块，演示畸形的储存模块。

实验十五：FIFO 储存模块（同步）

笔者虽然在实验十四曾解释储存模块，而且也演示奇怪的家伙，但是实验十四只是一场游戏而已。至于实验十五，笔者会稍微严肃一点，手动建立有规格的储存模块，即同步 FIFO。那些看过《时序篇》的同学一定对同步 FIFO 不会觉得陌生吧？因为笔者曾在《时序篇》建立基于移位寄存器的同步 FIFO。不过那种同步 FIFO 只是用来学习的玩具而已。因此，这回笔者可要认真了！

事实告诉笔者，同步 FIFO 的利用率远胜其它储存模块，几乎所有接口模块都会出现它的身影。早期的时候，笔者都会利用官方准备的同步 FIFO（官方插件模块），大伙都知道官方插件模块都非常傲娇，心意（内容）不仅不容易看透，而且信号也不容易捉摸，最重要是无法随心所欲摆布它们。与其跪下向它求救，笔者还不如创建自己专属的同步 FIFO。

故名思议，“同步”表示相同频率的时钟源，“FIFO”表示先进先出的意思。FIFO 的用意一般都是缓冲数据，另模块独立，让模块回避调用的束缚。同步 FIFO 是 RAM 的亚种，它基于 RAM，再加上先进先出的机制，学习同步 FIFO 就是学习如何建立先进先出的机制。



图 15.1 同步 FIFO 建模图 (常规)。

常规上，同步 FIFO 的建模图如图 15.1 所示，左边有写入请求 ReqW，写入数据 DataW，还有写满标示 Full。换之，右边则有读出请求 ReqR，读出数据 DataR，还有读空标示 Empty。写入方面，ReqW 必须拉高 DataW 才能写入，一旦 FIFO 写满，那么 Full 就会拉高。至于读出方面，ReqR 必须拉高，数据才能经由 DataR 读出，一旦 FIFO 读空，Empty 就会拉高。不过图 15.1 可以稍微更动一下，另它更加接近低级建模 II 的形象。

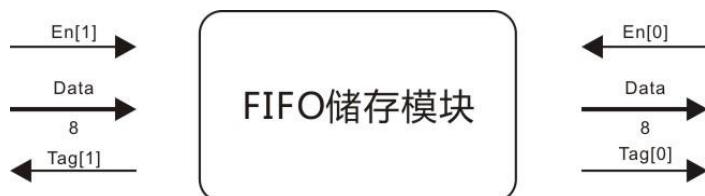


图 15.2 同步 FIFO 建模图 (低级建模 II)。

如图 15.2 所示，Req× 改为沟通信号 En，其中 En[1] 表示写入使能，En[0] 表示读出使能。Data× 改为数据信号 Data，iData 为写入数据，oData 为读出数据。Full 与 Empty 则改为状态信号 Tag[1] 与 Tag[0]。

```

1. module fifo_savemod
2. (
3.     input CLOCK, RESET,
4.     input [1:0]iEn,
5.     input [3:0]iData,
6.     output [3:0]oData
7.     output [1:0]oTag
8. );
9. .....
10. assign oTag[1] = ...; // Full
11. assign oTag[0] = ...; // Empty
12.
13. endmodule

```

代码 15.1

同步 FIFO 大致的外皮如代码 15.1 所示，第 3~7 行是相关的出入端声明，第 10~11 行则是相关的输出驱动声明。理解这些以后，接下来我们要学习先进先出这个机制。

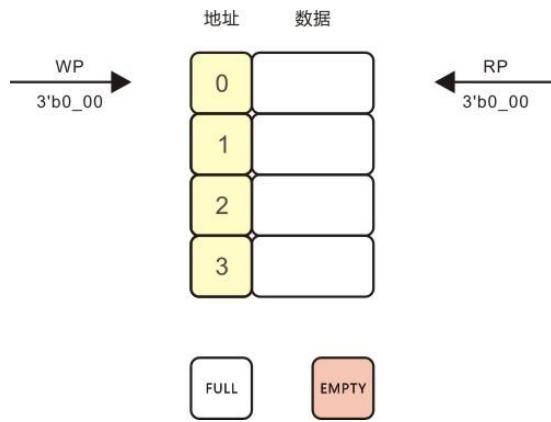


图 15.3 读空状态。

假设笔者建立位宽为 4，深度为 4 的 ram，然后又建立位宽为 3 的写指针 WP 与读指正 RP。同学一定会好奇道，既然 ram 只有 4 个深度，那么指针只要 2 位宽 ($2^2 = 4$) 即不是可以访问所有深度呢？话虽如此，为了利用指正表示写满与读空状态，指针必须多出一位 … 因此，指针的最高位常常也被称为方向位。

如图 15.3 所示，一开始的时候，写指针与读指针同样指向地址 0，而且 ram 里边也是空空如也，为此读空状态“叮咚”亮着红灯。为此，我们可以暂时这样表示读空的逻辑关系：

```
Empty = (WP == RP);
```

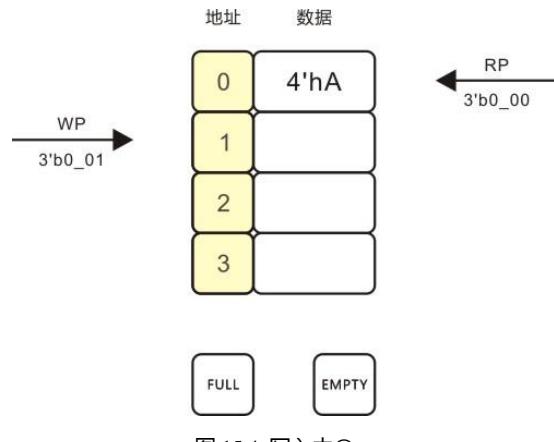


图 15.4 写入中①。

当火车开动以后，首先数据 $4'hA$ 写入地址 0，然后写指针从原来的 $3'b0_00$ 递增为 $3'b0_01$ ，并且指向地址 1。此刻，ram 再也不是空空如也，所示读空状态消除红灯，结果如图 15.4 所示。

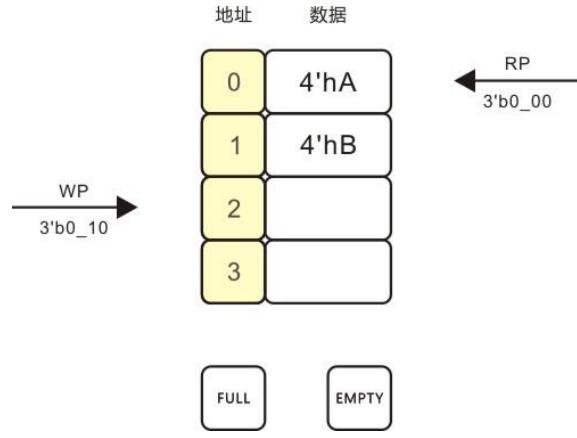


图 15.5 写入中②。

紧接着，数据 $4'hB$ 写入地址 1，然后写指针从原来的 $3'b0_01$ 递增为 $3'b0_10$ ，并且指向地址 2，结果如图 15.5 所示。

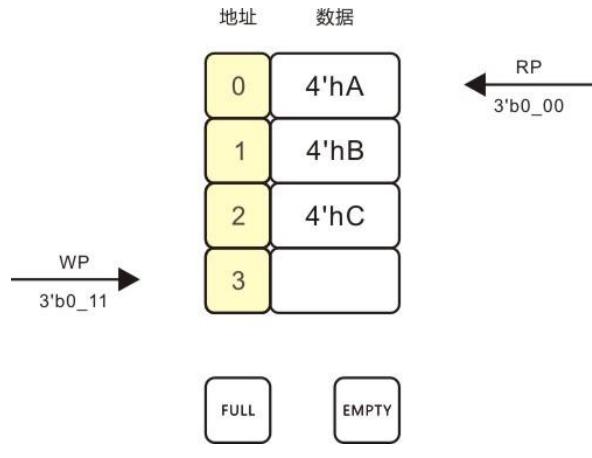


图 15.6 写入中③

然后，数据 $4'hC$ 写入地址 2，然后写指针从原来的 $3'b0_{_}10$ 递增为 $3'b0_{_}11$ ，并且指向地址 3，结果如图 15.6 所示。

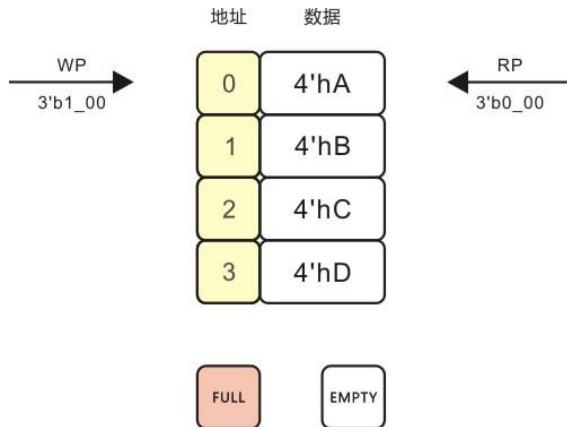


图 15.7 写满状态。

接着，数据 $4'hD$ 写入地址 3，然后写指针从原来的 $3'b0_{_}11$ 递增为 $3'b1_00$ ，并且重新指向地址 0。此刻写指针的最高位为 1，这表示写指针已经绕弯 ram 一圈又回来原点，反之读指针从刚才开始一动也不动，结果最高为 0... 所以我们可以说写指针与读指针目前处于不同的方向。在此 ram 已经写满，所以写满状态便“叮咚”亮红灯，结果如图 15.6 所示。写满状态的逻辑关系则可以这样表示：

```
FULL = ( WP[2] ^ RP[2] && WP[1:0] == RP[1:0] );
```

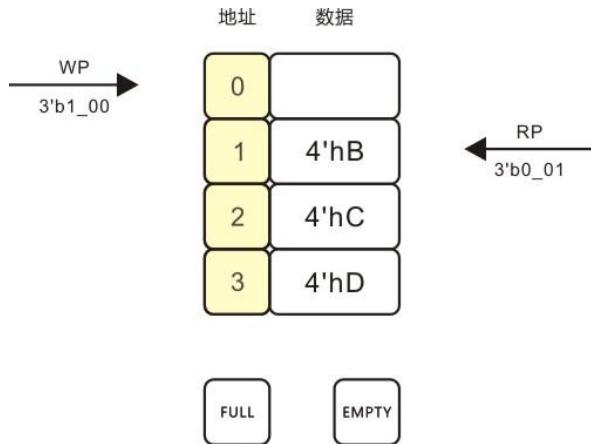


图 15.8 读出中①。

从现在开始，另一头火车才开始走动... 首先数据 $4'hA$ 从地址 0 读出来，读指针也从原本的 $3'b0_00$ 递增为 $3'b0_01$ ，并且指向地址 1。此刻 ram 再也不是吃饱饱的状态，所以写满状态被消除红灯，结果如图 15.8 所示。

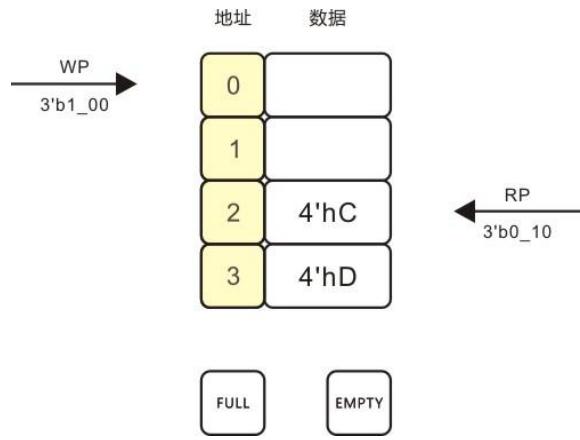


图 15.9 读出中②。

接下来，数据 $4'hB$ 从地址 1 哪里读出，读指针也从原本的 $3'b0_01$ 递增为 $3'b0_{10}$ ，并且指向地址 2，结果如图 15.9 所示。

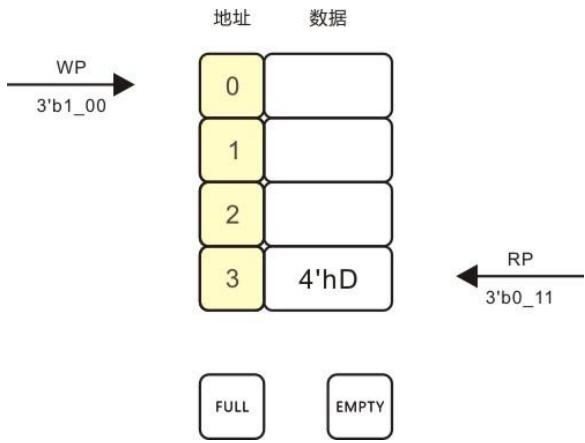


图 15.10 读出中③。

随之，数据 $4'hC$ 从地址 2 哪里读出，读指针也从原本的 $3'b0_{10}$ 递增为 $3'b0_{11}$ ，并且指向地址 3，结果如图 15.10 所示。

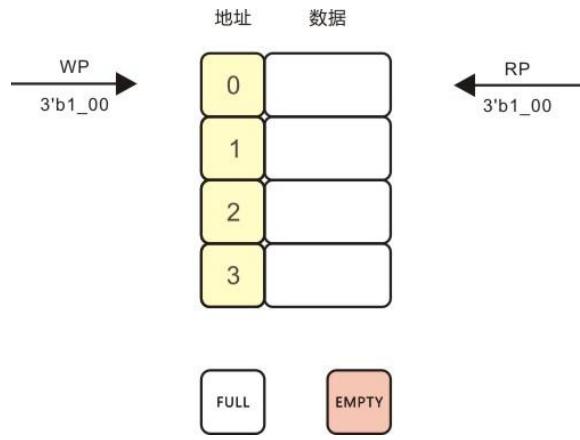


图 15.11 读空状态。

最后，数据 4'hD 从地址 3 哪里读出，读指针也从原本的 3'b0_11 递增为 3'b1_00，并且重新指向地址 0。当读指针绕弯一圈又回到原点的时候，读者的最高位也成为值 1，换句话说 … 此刻的读指针与写指针也处于同样的位置。同一个时候，ram 也是空空如也，所以读空状态便“叮咚”亮起红灯，结果如图 15.11 所示。为此，读空状态的逻辑关系可以这样表示：

```
Empty = (WP == RP);
```

总结而言，当我们设置 N 位位宽的时候，读写指针的位宽便是 N + 1。此外，读空状态为写指针等价读指针。反之，写满状态是两个指针方向一致（异或状态），然后地址一致。理解先进先出的机制以后，接下来我们便可以填充一下 FIFO 储存模块的内容。

```
1. module fifo_savemod
2. (
3.     input CLOCK, RESET,
4.     input [1:0]iEn,
5.     input [3:0]iData,
6.     output [3:0]oData,
7.     output [1:0]oTag
8. );
9.     reg [3:0] RAM [3:0];
10.    reg [3:0]D1;
11.    reg [2:0]C1,C2; // N+1
12.
13.    always @ ( posedge CLOCK or negedge RESET )
14.        if( !RESET )
15.            begin
16.                C1 <= 3'd0;
17.            end
18.        else if( iEn[1] )
19.            begin
20.                RAM[ C1[1:0] ] <= iData;
21.                C1 <= C1 + 1'b1;
22.            end
23.
24.    always @ ( posedge CLOCK or negedge RESET )
25.        if( !RESET )
26.            begin
27.                D1 <= 4'd0;
28.                C2 <= 3'd0;
29.            end
30.        else if( iEn[0] )
```

```

31.           begin
32.               D1 <= RAM[ C2[1:0] ];
33.               C2 <= C2 + 1'b1;
34.           end
35.
36.       assign oData = D1;
37.       assign oTag[1] = ( C1[2]^C2[2] & C1[1:0] == C2[1:0] ); // Full
38.       assign oTag[0] = ( C1 == C2 ); // Empty
39.
40.   endmodule

```

代码 15.2

笔者在第 9~11 行创建相关的寄存器，C1 取代 WP，C2 取代 RP。第 13~22 行是写操作，内容非常单纯，即 iEn[1] 拉高便将 iData 写入 C1[1:0] 指定的地方，然后 C1 递增。第 24~34 行是读操作，内容也是一样单纯，iEn[0] 拉高便将 C2[1:0] 指定的数据暂存至 D1，随后 C2 递增，最后由 D 驱动 oData。第 37~38 行是写满状态与读空状态的逻辑关系。

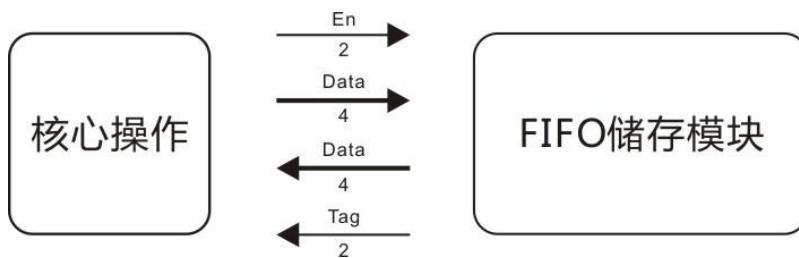


图 15.12 调用 FIFO 储存模块。

创建同步 FIFO 基本上没有什么难度，但是调用 FIFO 倒是一件难题。如图 15.12 所示，笔者建立一支核心操作尝试调用 FIFO 储存模块，至于核心操作的内容如代码 15.3 所示：

```

1. case( i ) // Core
2.   0:
3.     if( iTag[1]! ) begin oEn[1] <= 1' b1; oData <= 4' hA; i <= i + 1' b1; end
4.   1:
5.     if( iTag[1]! ) begin oEn[1] <= 1' b1; oData <= 4' hB; i <= i + 1' b1; end
6.   2:
7.     if( iTag[1]! ) begin oEn[1] <= 1' b1; oData <= 4' hC; i <= i + 1' b1; end
8.   3:
9.     if( iTag[1]! ) begin oEn[1] <= 1' b1; oData <= 4' hD; i <= i + 1' b1; end
10.  4:
11.    begin oEn[1] <= 1' b0; i <= i + 1' b1; end
12.  5:
13.    if( iTag[0]! ) begin oEn[0] <= 1' b1; i <= i + 1' b1; end
14.  6:

```

```

15.     if( iTag[0]! ) begin oEn[0] <= 1' b1; i <= i + 1' b1; end
16.     7:
17.     if( iTag[0]! ) begin oEn[0] <= 1' b1; i <= i + 1' b1; end
18.     8:
19.     if( iTag[0]! ) begin oEn[0] <= 1' b1; i <= i + 1' b1; end
20.     9:
21.     begin oEn[0] <= 1' b0; i <= i + 1' b1; end
22. endcase

```

代码 15.3

如代码 15.3 所示，步骤 0~3 用来一边检测 Tag[1] 是否为高，一边向储存模块写入数据 4'hA~4'hD，步骤 4 则用来拉低 oEn[1] 并且歇息一下。步骤 5~8 用来一边检测 Tag[0] 是否为高，一边从储存模块哪里读出数据，步骤 9 则用来拉低 oEn[0] 并且偷懒一下。

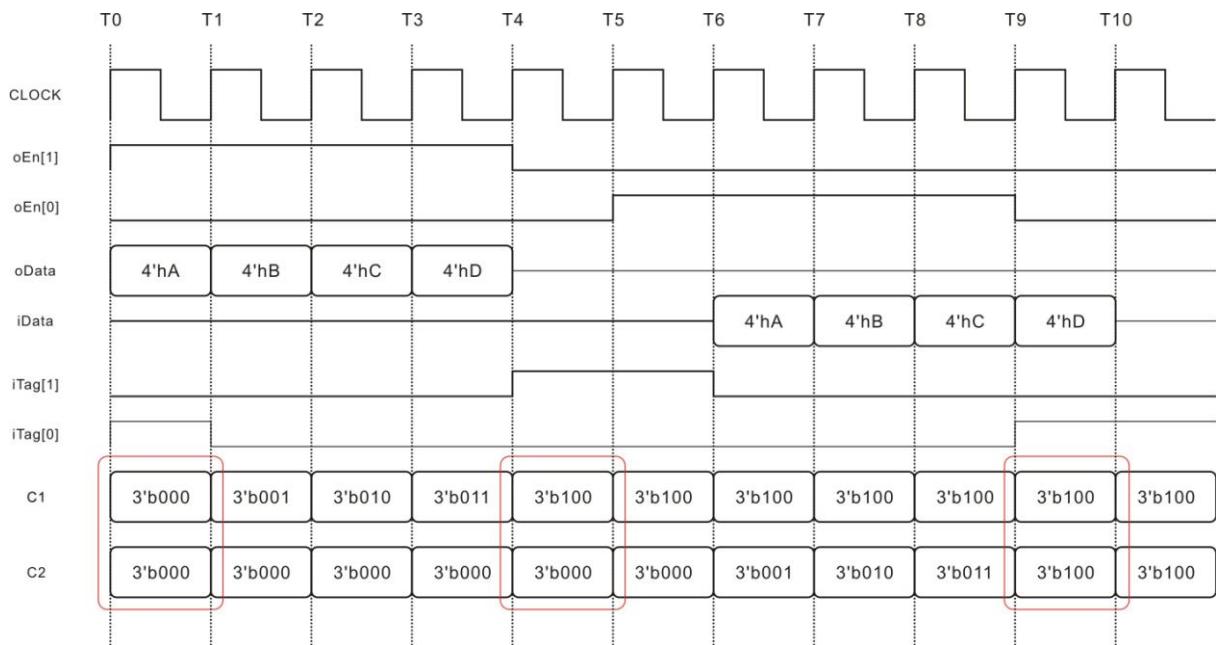


图 15.13 读写 FIFO 储存模块的理想时序图。

图 15.13 是代码 15.3 所生产的理想时序图，同时也是核心操作作为视角的时序，至于 C1~C2 是 FIFO 储存模块作为视角的时序。各个视角的时序过程如下：

核心操作视角：

- T0 , isTag[1] 为低 (即时值) , 拉高 oEn[1] (未来值) , 并且发送数据 4'hA (未来值)。
- T1 , isTag[1] 为低 (即时值) , 拉高 oEn[1] (未来值) , 并且发送数据 4'hB (未来值)。
- T2 , isTag[1] 为低 (即时值) , 拉高 oEn[1] (未来值) , 并且发送数据 4'hC (未来值)。
- T3 , isTag[1] 为低 (即时值) , 拉高 oEn[1] (未来值) , 并且发送数据 4'hD (未来值)。
- T4 , isTag[1] 为高 (即时值) , 拉低 oEn[1] (未来值)。
- T5 , isTag[0] 为低 (即时值) , 拉高 oEn[1] (未来值) , 数据 4'hA 读出 (过去值)。
- T6 , isTag[0] 为低 (即时值) , 拉高 oEn[1] (未来值) , 数据 4'hB 读出 (过去值)。
- T7 , isTag[0] 为低 (即时值) , 拉高 oEn[1] (未来值) , 数据 4'hC 读出 (过去值)。
- T8 , isTag[0] 为低 (即时值) , 拉高 oEn[1] (未来值) , 数据 4'hD 读出 (过去值)。
- T9 , isTag[0] 为高 (即时值) , 拉低 oEn[1] (未来值)。
- T10 , isTag[0] 为高 (即时值) , 拉低 oEn[1] (未来值)。

- T7 , isTag[0]为低 (即时值), 拉高 oEn[1] (未来值), 数据 4'hB 读出 (过去值)。
- T8 , isTag[0]为低 (即时值), 拉高 oEn[1] (未来值), 数据 4'hC 读出 (过去值)。
- T9 , isTag[0]为高 (即时值), 拉低 oEn[1] (未来值), 数据 4'hD 读出 (过去值)。
- T10 , isTag[0]为高 (即时值)。

FIFO 储存模块视角 :

- T0 , oEn[1]为低 (过去值)。 C1 等价 C2 为读空状态 , iTag[0]拉高 (即时值)。
- T1 , oEn[1]为高(过去值),读取数据 4'hA(过去值),递增 C1。 C1 不等价 C2 ,iTtag[0]拉低 (即时值)。
- T2 , oEn[1]为高 (过去值), 读取数据 4'hB (过去值), 递增 C1。
- T3 , oEn[1]为高 (过去值), 读取数据 4'hC (过去值), 递增 C1。
- T4 , oEn[1]为高 (过去值), 读取数据 4'hA (过去值), 递增 C1。 C1 等价 C2 为写满状态 , iTtag[1]拉高 (即时值)。
- T5 , oEn[1]为低 (过去值)。
- T6 ,oEn[0]为高(过去值),读出数据 4'hA(未来值),递增 C2。 C1 不等价 C2 ,isTag[1]拉低 (即时值)。
- T7 , oEn[0]为高 (过去值), 读出数据 4'hB (未来值), 递增 C2。
- T8 , oEn[0]为高 (过去值), 读出数据 4'hC (未来值), 递增 C2。
- T9 , oEn[0]为高 (过去值), 读出数据 4'hD (未来值), 递增 C2。 C1 等价 C2 为读空状态 , isTag[0]拉高 (即时值)。
- T10 , oEn[0]为低 (过去值)。

读者是不是一边浏览一边捏蛋蛋呢 ? 什么过去值 , 又什么未来值 , 又什么即时值的 ... 没错 , 同步 FIFO 的设计原理虽然简单 , 但是时序解读却让人泪流满面。因为同步 FIFO 夹杂两种时序表现——时间点事件还有即时事件。如图 15.13 所示 , 除了 iTag 信号是触发即时事件以外 , 所有信号都是触发时间点事件。读过《时序篇》或者《工具篇 II》的朋友一定知晓 , 即时值不仅比过去值优先 , 而且即时值也会无视时钟。

好奇的同学可能困惑道 :“为什么 iTag 不能设计成为时间点事件呢 ? ”。笔者曾在《时序篇》建立基于移位寄存器的 FIFO , 其中 iTag 就是设计成为时间点事件 , 结果 FIFO 的写满状态或者读空状态都来不及反馈 , 因此发生调用上的混乱。

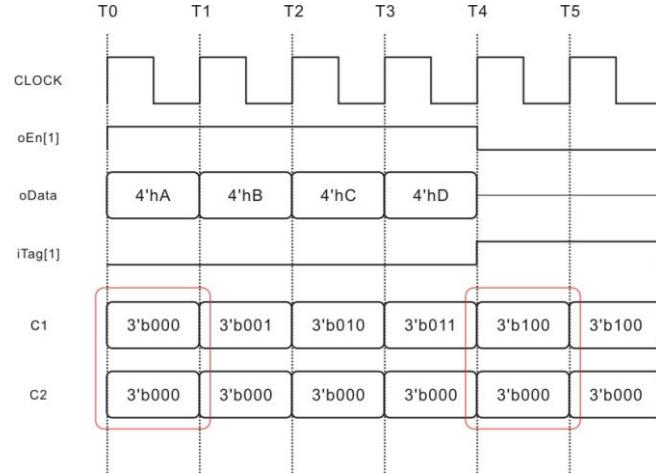


图 15.14 读写 FIFO 储存模块的即时事件。

为了理解重点，首先让我们来焦距写数据的部分。如图 15.14 所示，关键的地方就是发生在 T4——这只时钟沿。T4 之际，FIFO 储存模块读取 oEn[1]的过去值，C1 也因此递增，即时事件就在这个瞬间发生了。写满状态成立，iTTag[1]也随之拉高即时值。从时序上来看，C1 的更新（C1 为 4'b100）是发生在 T4 之后，不过那也无关紧要，因为即时值是更新在小小的时间沿之间，也是即时层 … 然而，即时层是无法显示在时序之上。

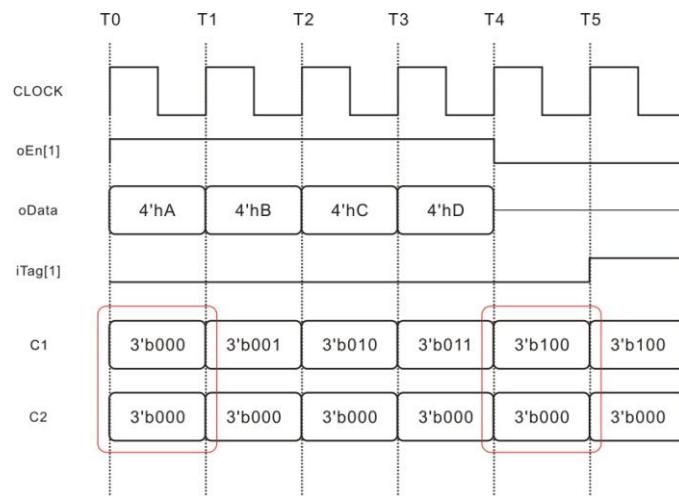


图 15.15 迟到的写满状态。

假设，iTTag[1]不是经由即时事件触发而是事件点事件，那么 iTTag 就会反馈迟到的写满状态。如图 15.15 所示，T4 之际 oEn[1] 为高，C1 也因此递增为 3'b100。T5 之际，C1 与 C2 的过去值均为 3'b100 与 3'b000，然后拉高 iTTag[1]。由于时间点事件的关系，所以 iTTag[1]迟一拍被拉高 … 读者千万别小看这样慢来一拍，它是搞乱调用的罪魁祸首。

如果核心操作在 T5 继续写操作的话，此刻 iTTag[1]的过去值为 0，它会认为 FIFO 未满，然后不管三七二十一执行写操作，结果 FIFO 发生错乱随之机能崩溃。从某种程度来看，即时事件的偷时钟能力，是建立同步 FIFO 的关键。

fifo_savemod.v



图 15.16 fifo 储存模块的建模图。

图 15.16 基本上与图 15.2 没什么两样，不过 FIFO 储存模块的位宽还有深度发生改变而已。此外，图 15.16 的信号布局虽然有点违规低，不过这点小细节读者就不要太计较了。建模技巧毕竟不是暴力规范，用不着死守，反之随机应变才是本意。

```
1. module fifo_savemod
2. (
3.     input CLOCK, RESET,
4.     input [1:0]iEn,
5.     input [7:0]iData,
6.     output [7:0]oData,
7.     output [1:0]oTag
8. );
```

以上内容是相关的出入端声明。

```
9.     reg [7:0] RAM [15:0];
10.    reg [7:0]D1;
11.    reg [4:0]C1,C2; // N+1
12.
```

以上内容是相关的内存与寄存器声明。第 9 行，RAM 声明为 8 位宽还有 $2^4=16$ 个深度。为此，第 11 行的写指针 C1 与读指针 C2 声明为 5 个位宽。

```
13.    always @ ( posedge CLOCK or negedge RESET )
14.        if( !RESET )
15.            begin
16.                C1 <= 5'd0;
17.            end
18.        else if( iEn[1] )
19.            begin
20.                RAM[ C1[3:0] ] <= iData;
21.                C1 <= C1 + 1'b1;
22.            end
```

23.

以上内容是 fifo 的写操作 , 第 18 行的 iEn[1] 每拉高一个时钟 , 第 20 行的 iData 便写入 C1[3:0] 指定的位置 , 随后第 21 行写指针也递增。

```
24.      always @ ( posedge CLOCK or negedge RESET )
25.          if( !RESET )
26.              begin
27.                  D1 <= 8'd0;
28.                  C2 <= 5'd0;
29.              end
30.          else if( iEn[0] )
31.              begin
32.                  D1 <= RAM[ C2[3:0] ];
33.                  C2 <= C2 + 1'b1;
34.              end
35.
```

以上内容是 fifo 的读操作 , 第 30 行的 iEn[0] 每拉高一个时钟 , 第 32 行的 D1 便赋予 C2[3:0] 指定的数据 , 随后第 33 行的读指针也递增。

```
36.      assign oData = D1;
37.      assign oTag[1] = ( C1[4]^C2[4] & C1[3:0] == C2[3:0] ); // Full
38.      assign oTag[0] = ( C1 == C2 ); // Empty
39.
40.  endmodule
```

以上内容是相关输出驱动声明 , 其中第 37 行是写满状态 , 第 38 行是读空状态。在此 , 读者需要注意一下 ... 第 36 行相较第 37~38 行 , 前者由寄存器 D1 驱动 , 即 oData 信号为时间点事件。反之 , 后者由组合逻辑驱动 , 即 oTag[1:0] 信号为即时事件。为此 , 该储存模块的内部状态是以即时的方式反馈出去。

tx_rx_demo.v

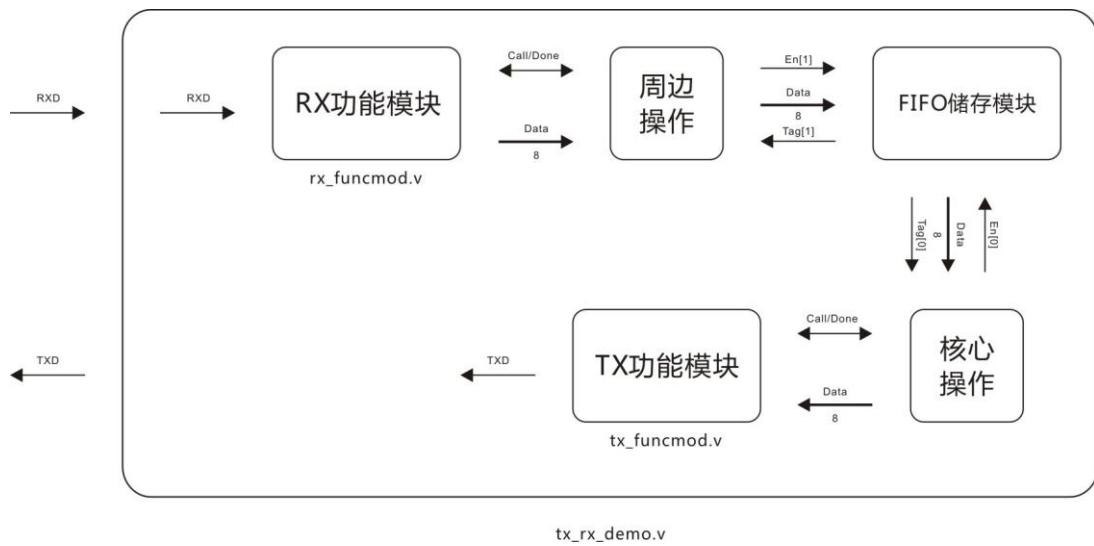


图 15.17 实验十五的建模图。

实验十五是实验十三的延续 ... 实验十三之际，RX 功能模块接收并且失败发送一连串的数据，因为发送方不仅来不及，而且接收成功的数据也没有地方缓冲。如今实验十五多了一只 FIFO 储存模块作为缓冲空间。注意，图 15.17 虽然是实验十五的建模图，可是却与实际的连线部署有一点出入，不过大意上都是差不多的。

RX 功能模块接收一连串的数据，然后经由周边操作协调，事后再将数据缓冲至 FIFO 储存模块。至于核心操作会不停从 FIFO 储存模块哪里读取数据，然后再调用 TX 功能模块将数据发送出去。

```

1. module tx_rx_demo
2. (
3.     input CLOCK, RESET,
4.     input RXD,
5.     output TXD
6. );

```

以上内容是相关的出入端声明。

```

7.     wire DoneU1;
8.     wire [7:0]DataU1;
9.
10.    rx_funcmod U1
11.    (
12.        .CLOCK(CLOCK),
13.        .RESET(RESET),
14.        .RXD(RXD),           // < top
15.        .iCall(isRX),       // < sub

```

```
16.           .oDone( DoneU1 ), // > U2
17.           .oData( DataU1 ) // > U2
18.       );
19.
```

以上内容是 RX 功能模块的实例化。第 15 行表示 isRX 充当使能。

```
20.     reg isRX;
21.
22.     always @ ( posedge CLOCK or negedge RESET ) // sub
23.         if( !RESET ) isRX <= 1'b0;
24.         else if( DoneU1 ) isRX <= 1'b0;
25.         else isRX <= 1'b1;
26.
```

以上内容是周边操作，它主要重复调用 RX 功能模块。

```
27.     wire [1:0]TagU2;
28.     wire [7:0]DataU2;
29.
30.     fifo_savemod U2
31.     (
32.         .CLOCK( CLOCK ),
33.         .RESET( RESET ),
34.         .iEn ( { DoneU1 , isRead } ), // < U1 & Core
35.         .iData ( DataU1 ), // < U1
36.         .oData ( DataU2 ), // > U3
37.         .oTag ( TagU2 ) // > core
38.     );
39.
```

以上内容是 FIFO 储存模块的实例化。第 34 行表示，DoneU1 充当写入使能，isRead 充当读出使能。

```
40.     wire DoneU3;
41.
42.     tx_funcmod U3
43.     (
44.         .CLOCK( CLOCK ),
45.         .RESET( RESET ),
46.         .TXD( TXD ), // > top
47.         .iCall( isTX ), // < core
48.         .oDone( DoneU3 ), // > core
```

```
49.           .iData( DataU2 ) // < U2
50.       );
51.
```

以上内容是 TX 功能模块的实例化。第 47 行表示 isTX 充当使能。第 49 行表示，该模块的 iData 直接经由 DataU2 驱动。

```
52.     reg [3:0]i;
53.     reg isRead;
54.     reg isTX;
55.
56.     always @ ( posedge CLOCK or negedge RESET ) // core
57.         if( !RESET )
58.             begin
59.                 i <= 4'd0;
60.                 isRead <= 1'b0;
61.                 isTX<= 1'b0;
62.             end
```

以上内容是核心操作的相关寄存器声明与复位操作。

```
63.         else
64.             case( i )
65.
66.                 0:
67.                     if( !TagU2[0] ) begin isRead <= 1'b1; i <= i + 1'b1; end
68.
69.                 1:
70.                     begin isRead <= 1'b0; i <= i + 1'b1; end
71.
72.                 2:
73.                     if( DoneU3 ) begin isTX <= 1'b0; i <= 4'd0; end
74.                     else isTX <= 1'b1;
75.
76.             endcase
77.
78. endmodule
```

以上内容是操作操作。步骤 0 用来判断 FIFO 是否读空，否则就拉高 isRead。步骤 1 则拉低 isRead，然后 FIFO 就会读出数据。步骤 2 则使能 TX 功能模块，并且将方才读出的数据发送出去。

编译完毕并下载程序。此刻，串口便可以支持一连串的数据发送与接收，为了避免部分

数据凭空消失的怪事，数据流的容量必须配合 FIFO 的缓冲容量（深度）。此外，实验十五还有许多优化的空间，然而这些都是交由读者的功课。（注意，某些串口调试助手必须把[检验位设置为标志位](#)才能显示字符）

细节一：完整的个体模块

该实验的 `fifo_savemod.v` 已经是完整的个体。

实验十六：IIC 储存模块

IIC 储存器是笔者用来练习精密控时的经典例子。《整合篇》之际，IIC 储存器的解释，笔者也自认变态。如今笔者回头望去，笔者也不知道自己当初到底发什么神经，既然将 IIC 的时序都解释一番。由于开发上板也嵌着 IIC 储存器（24LC04），笔者还得循例地介绍一下。

IIC 储存器是应用 IIC 总线的储存器，时序本身并不是很复杂不过缺有一大堆时序参数，而且官方提供的时序也不利于描述，所以许多时序都必须自行绘制，真是麻烦死人。麻烦归麻烦，笔者终究还要吃饭，为了肚子，再麻烦的事情也要硬着头皮捱过去 … 这也是白骆驼的恶作剧！

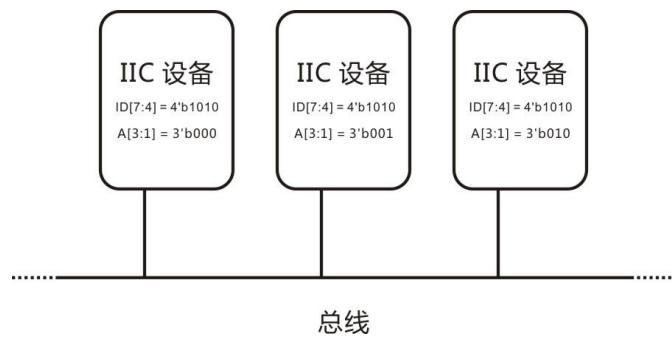


图 16.1 IIC 总线与 IIC 设备。

图 16.1 是 IIC 总线与 IIC 设备常见的示意图。理想上，一条 IIC 总线允许千万 IIC 设备占据在上 … 物理下，一条 IIC 总监究竟允许可多少 IIC 设备占据其中必须根据设备地址的长度。默认下，设备地址为八位宽，因此设备地址也称为设备字节。设备地址的高四位，即 [7..4] 记录硬件 ID，接续三位即 [3..1] 则记录硬件地址，最后一位则是设备的访问方向。结果如表 16.1 所示：

表 16.1 设备地址的位分配。

[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
硬件 ID				硬件地址			访问方向

所谓硬件 ID 就是 IIC 设备的辨识 ID，硬件 ID 会随着厂商还有设备的种类而有所改变。开发板上的 IIC 设备是某厂商的 IIC 储存器，即 24LC04，硬件 ID 为 4'b1010。至于硬件地址就是 IIC 设备在总线上辨识地址，默认下为 3 位，即同类的 IIC 设备在同一条 IIC 总线上仅允许占据 8 个而已。然而，开发板上的 24LC04 为 3'b000。最后的访问方向位则是主机用来通知从机，此刻的访问目的是读还是写。

总结来说，设备地址除了访问方向以外，前七位一般都是固定的，例如开发板的 IIC 储存器 24LC04，设备地址就是 8'b1010_000_×。

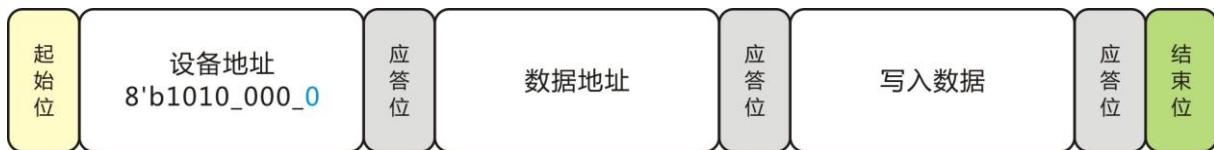


图 16.2 24LC04 的写操作 (主机视角)。

IIC 总线的时序，感觉上一组完成的操作宛如是一堆拼图。如图 16.2 所示，那是 24LC04 的写操作，时序先填上为起始位，再来是设备地址，余下是应答位，随之是数据地址，然后又是应答位，接着是写入数据，再一次应答位，最后挂上结束位以示一次性的写操作已经完成。那么，写操作的经过如下所示：

- (一) 主机发送起始位；
- (二) 主机发送设备地址 (写)；
- (三) 等待从机应答；
- (四) 主机发送数据地址；
- (五) 等待从机应答；
- (六) 主机发送数据；
- (七) 等待从机应答；
- (八) 主机发送结束位。

读者稍微注意一下设备地址的最低位，笔者稍微用蓝色将其高亮。由于此刻是写操作，所以设备地址的访问方向是“写”，所以访问方向位设置为 0。



图 16.3 24LC04 的读操作 (主机视角)。

图 16.3 是 24LC04 的读时序，同样它也是由一堆“拼图”组合而成。相较写操作，读操作不仅多了许多“拼图”，而且途中也改变访问方向。那么，读操作的经过如下所示：

- (一) 主机发送起始位；
- (二) 主机发送设备地址 (写)；
- (三) 等待从机应答；
- (四) 主机发送数据地址；
- (五) 主机发送起始位；
- (六) 主机发送设备地址 (读)；
- (七) 等待从机应答；
- (八) 主机读取数据；
- (九) 从机没有应答 (主机无视应答)；
- (十) 主机发送结束位。

未进入正题之前，请允许笔者加入一些小插曲。IIC 总线是一种低速的总线，不过 IIC 总线有 100Khz 还有 400Khz 两种速率提供我们选择，要么 100Khz，要么 400Khz，要么两者兼施，不管哪一种《整合篇》都曾实验过。在此，实验十六会以 400Khz 的速率作为标准。

笔者曾在前面说过，IIC 总线之所以麻烦，因为 IIC 总线有大小不同的时序参数（时间参数）。一般而言，时间参数都都被顺序语言一笑而过，那是因为顺序语言无法实现精密控时。虽然描述语言也可以一笑而过，但是语言的本质却不允我们这么作，如果我们选择无视时序参数 … 那么，打从一开始我们还是不学为好。

此外，描述 IIC 的总线时序有各种各样的方法，但是笔者会选择表达能力更高，控制能力更细的描述手段。我们知道 IIC 的总线时序是由一块又一块的拼图拼凑而成，当我们在建模的时候，我们会针对各个拼图作出局部性的描述。期间，我们也必须考虑各种时序参数，如表 16.2 所示：

表 16.2 各种时序参数 (50Mhz 量化)

相关参数	标示	最小时间	最小时钟	最大时间	最大时钟
Clock Frequency	FCLK	---	---	400Khz	125
Clock High Time	THIGH	600ns	30	---	---
Clock Low Time	TLOW	1300ns	65	---	---
Rise Time	TR	---	---	300ns	15
Fall Time	TF	---	---	300ns	15
Start Hold Time	THD_STA	600ns	30	---	---
Start Setup Time	TSU_STA	600ns	30	---	---
Data Input Hold Time	THD_DAT	0ns	0	---	---
Data Input Setup Time	TSU_DAT	100ns	5	---	---
Stop Setup Time	TSU_STO	600ns	30	---	---
Output Valid From Clock	TAA	---	---	900ns	45
Bus Free Time	TBUF	1300ns	65	---	---

相比许多同学遇见表 16.2 便会立即憋着蛋蛋，因为它会吓坏一群小朋友。话虽如此，表 16.2 只有外表可怕的纸老虎而已，任何有时序基础的同学，随便擦擦两下就搞定。笔者虽然也想一笑打过，不过笔者还要循例介绍一下：

- Clock Frequency，既是频率也是速率，在此是 400Khz。
- Clock High Time，既 SCL 信号保持高电平所需的最小时间。
- Clock Low Time，既 SCL 信号保持低电平所需的最小时间。
- Rise Time，既信号由底变高所需最大的时间。
- Fall Time，既信号由高变低所需最小的时间。
- Start Hold Time，既起始位所需最小的保持时间。
- Start Setup Time，既起始位所需最小的建立时间。
- Data Input Hold Time，既数据位所需最小的保持时间。
- Data Input Setup Time，既数据位所需最小的建立时间。

- Stop Setup Time , 既结束位所需的最小保持时间。
- Output Valid From Clock , 既数据位经时钟沿触发以后的有效时间。
- Bus Free Time , 既释放总线的最长时间。

IIC 总线是一种串行传输协议 ,既有时钟信号 SCL ,还有数据信号 SDA。Clock Frequency 表示 SCL 信号的频率 ,Clock High Time 表示 SCL 信号保持高电平所需的最长时间 ,Clock Low Time 则表示 SCL 信号保持低电平所需的时间。

至于 Rise Time 与 Fall Time 表示 ,SCL 信号还有 SDA 信号由高变低或者由低变高时所需的最长时间 ,即上山与下山时间。Hold Time 与 Setup Time 是用来评估数据是否成功打入寄存器的时序参数 ,算是典型中的典型。Setup Time 表示建立时间 ,即数据写入寄存器之前所需的稳定时间 ;反之 ,Hold Time 则是保持时间 ,即数据打入寄存器之后所需的稳定时间。只要两者得到满足 ,那么数据的寄存活动就得到确保。

Start 是 IIC 总线的起始位 ,Stop 是 IIC 总线的结束位 ,Data 是 IIC 总线的数据位 ,为了确保三者成功写入从机 ,Setup Time 与 Hold Time 必须得到满足。Output Valid From Clock 是关系数据位的时序参数 ,还有 Bus Free Time 是关系结束位的时序参数 ,在此先丢胃口一下。此外 ,为了简化时序 ,笔者将各种参数的实际时间转换为 50Mhz 量化以后的结果。对此 ,Verilog 可以这样表示 ,结果如代码 16.1 所示 :

```

1. parameter FCLK = 10'd125, FHALF = 10'd62, FQUARTER = 10'd31;
2. parameter THIGH = 10'd30, TLOW = 10'd65, TR = 10'd15, TF = 10'd15;
3. parameter THD_STA = 10'd30, TSU_STA = 10'd30, TSU_STO = 10'd30;

```

代码 16.1

如代码 16.1 所示 ,FCLK 表示 400Khz 的周期 ,FHALF 表示 1/2 周期 ,FQUARTER 表示 1/4 周期。至于为什么代码 16.1 不见 ,Data Input Hold Time 与 Bus Free Time 的时序参数 ,请读者暂时忍耐 ,往后会解释。

(话题继续之前 ,请读者确保自己对 “整合时序” 有一定的理解 ,不然的话 ... 接下来的内容 ,读者一定会看到泪流满面。)

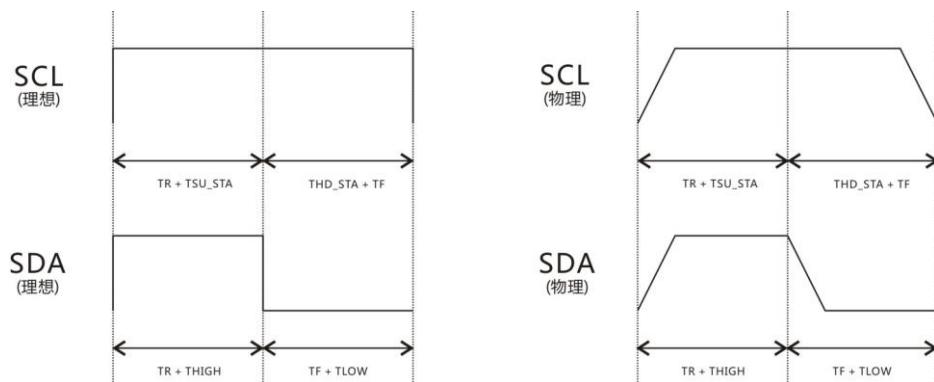


图 16.4 起始位。

首先让我们先瞧瞧起始位这枚拼图。如图 16.4 所示 ,左图是起始位的理想时序 ,右图是

起始位的物理时序。IIC 总线的起始位也就类似串口或者 PS/2 等传输协议的起始位，然而不同的是，IIC 总线的起始位是 SCL 拉高 $TR + TSU_STA + THD_STA + TF$ 之久，换之 SDA 则是拉高 $TR + THIGH$ 然后拉低 $TF + TLOW$ 。起始位总和所用掉的时间，恰恰好有一个速率的周期。对此，Verilog 则可以这样描述，结果如代码 16.2 所示：

```

1. begin
2.     isQ = 1;
3.     rSCL <= 1'b1;
4.     if( C1 == 0 ) rSDA <= 1'b1;
5.     else if( C1 == (TR + THIGH) ) rSDA <= 1'b0;
6.     if( C1 == (FCLK) -1) begin C1 <= 10'd0; i <= i + 1'b1; end
7.     else C1 <= C1 + 1'b1;
8. end

```

代码 16.2

如代码 16.2 所示，第 2 行的 $isQ = 1$ 表示设置 SDA 为输出状态（即时结果），第 3 行则表示 SCL 一直持续拉高状态，第 4~5 行表示 $C1$ 为 0 的时候 SDA 拉高，直到 $C1$ 为 $TR+THIGH$ 才拉低 SDA。第 6~7 行表示一个步骤所逗留的时间。

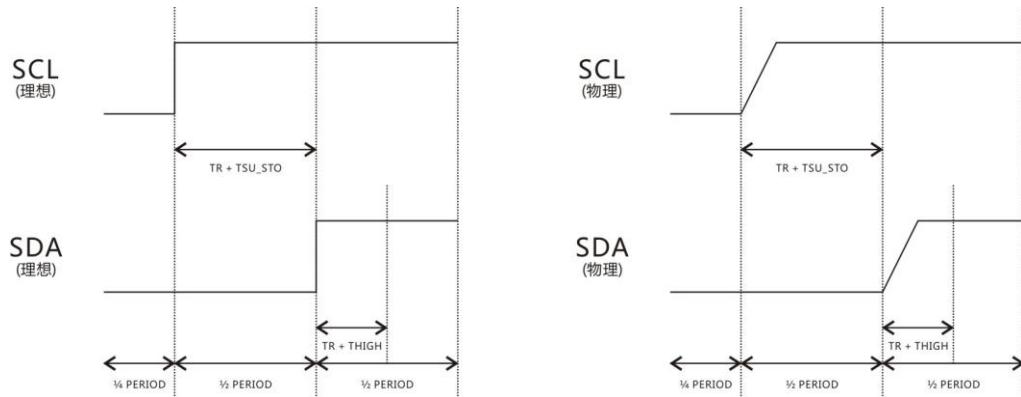


图 16.5 结束位。

图 16.5 是结束位的时序图，IIC 设备的操作好坏一般都取决结束位。保险起见，SCL 与 SDA 都事先拉低 $1/4$ 周期，紧接着 SCL 会拉高 $TR+TSU_STO$ （或者 $1/2$ 周期），最后又保持高电平 $1/2$ 周期。反之，SDA 会拉低 $1/2$ 周期，随之拉高 $TR+THIGH$ （或者 $1/2$ 周期）。对此，Verilog 可以这样表示，结果如代码 16.3 所示：

```

1. begin
2.     isQ = 1'b1;
3.     if( C1 == 0 ) rSCL <= 1'b0;
4.     else if( C1 == FQUARTER ) rSCL <= 1'b1;
5.     if( C1 == 0 ) rSDA <= 1'b0;
6.     else if( C1 == (FQUARTER + TR + TSU_STO) ) rSDA <= 1'b1;
7.     if( C1 == ( FQUARTER + FCLK ) -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end

```

```

8.           else C1 <= C1 + 1'b1;
9.       end

```

代码 16.3

如代码 16.3 所示，第 2 行表示 SDA 为输出状态（即时），第 3~4 行表示 C1 为 0 拉高 SCL，C1 为 1/4 周期就拉高。第 5~6 行表示，C1 为 0 拉低 SDA，C1 为 1/4 周期 + TR + TSU_STO 就拉高 SDA。第 7~8 行表示该步骤所逗留的时间。

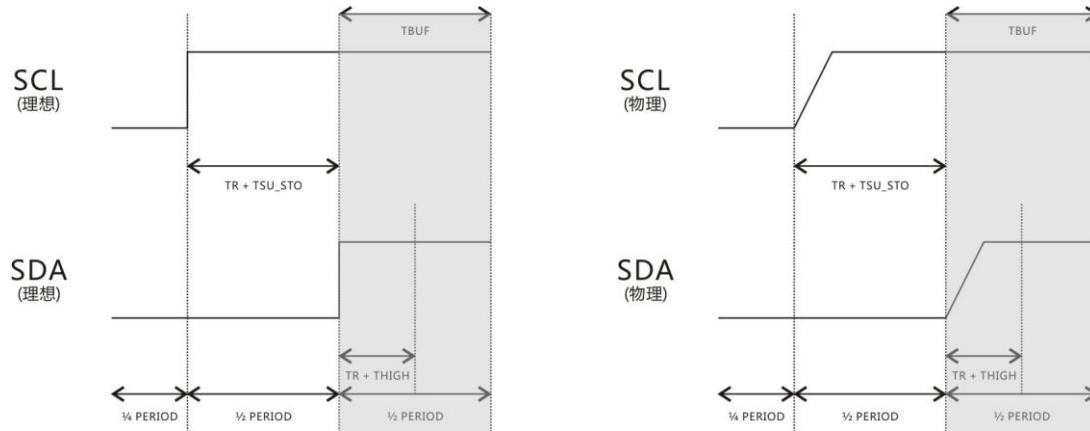


图 16.6 释放总线。

此外，结束位还有 Bus Free Time 这个时序参数，IIC 总线在闲置的状态下 SCL 与 SDA 等信号都持续高电平。主机发送结束位以示结束操作，然而主机持续拉高 SCL 信号与 SDA 信号 TBUF 以示总线释放。TBUF 的有效时间从 SCL 信号与 SDA 信号拉高那一刻开始算起

根据表 16.2 所示，TBUF 是 65 个时钟，结果如图 16.6 所示，SDA 信号拉高之后，SCL 与 SDA 信号只要持续保持 1/2 周期（即 62 个时），基本上就能满足 TBUF。如果笔者是一位紧密控时狂人，可能无法接受这样的结果，因为满足 TBUF 少了 3 个时钟，为此代码 16.3 需要更动一下：

```

7.           if( C1 == ( FQUARTER + FCLK + 3) -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
8.           else C1 <= C1 + 1'b1;
9.       end

```

代码 16.4

如代码 16.4 所示，笔者为第 7 行写下 +3 表示该步骤多逗留 3 个时钟，以致满足 TBUF。

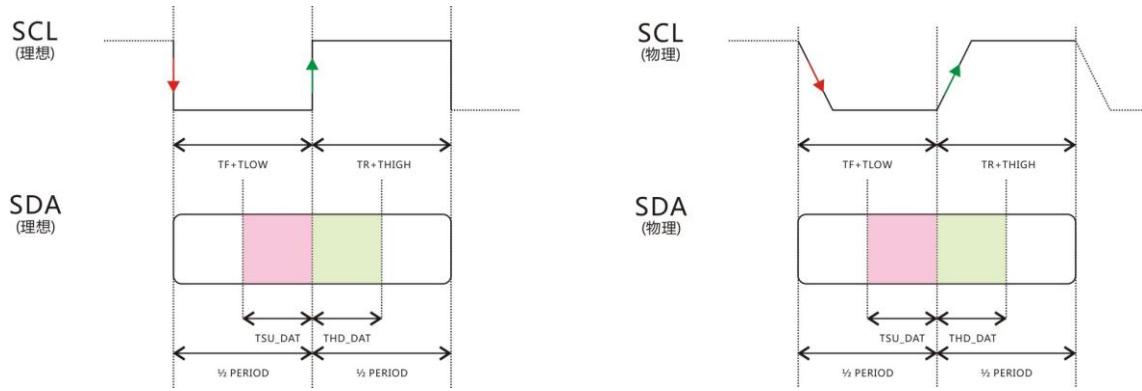


图 16.7 数据位。

不管对象是设备地址，数据地址，写入数据，读出数据，还是应答位，大伙都视为数据位。IIC 总线类似其他传输协议，它有时钟信号也有上升沿与下降沿。如图 16.7 所示，SCL 信号的下降沿导致设备设置（更新）数据，上升沿则是锁存（读取）数据。期间，TF+TLOW 表示时钟信号的前半周期，TR+THIGH 则表示后半周期。此外，为了确保数据成功打入寄存器，数据被上升沿锁存哪一刻起，TSU_DAT 还有 THD_DAT 必须得到满足。

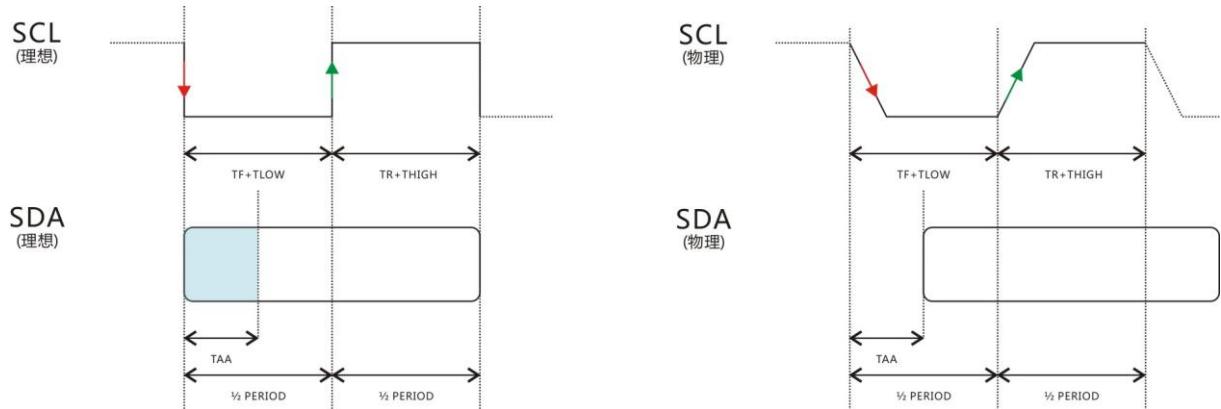


图 16.8 数据位更新有效。

除此之外，为了确保数据有效被更新，我们也必须确保 TAA 得到满足，结果如图 16.8 所示。理解完毕以后，我们就可以开始学习，写一字节数据与读一字节数据，还有应答位。

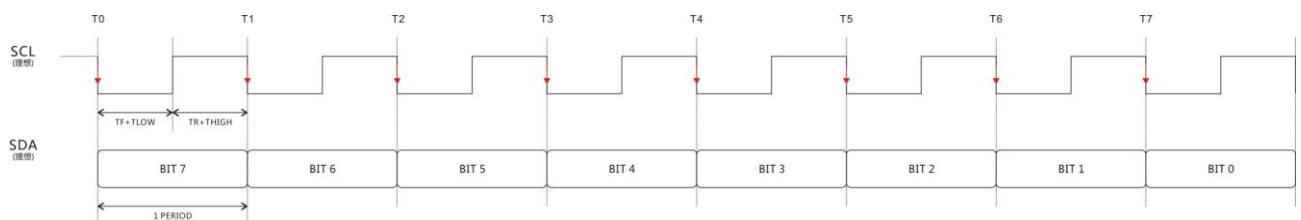


图 16.9 写一字节。

IIC 总线一般都是一个字节一个字节读写数据，如图 16.9 所示，那是写一字节的理想时

序图，一字节数据是从最高位开始写起。对此，Verilog 可以这样描述，结果如代码 16.5 所示：

```

1. 0,1,2,3,4,5,6,7;
2. begin
3.     isQ = 1'b1;
4.     rSDA <= D1[7-i];
5.     if( C1 == 0 ) rSCL <= 1'b0;
6.     else if( C1 == (TF + TLOW) ) rSCL <= 1'b1;
7.     if( C1 == FCLK -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
8.     else C1 <= C1 + 1'b1;
9. end

```

代码 16.5

如代码 16.5 所示，第 1 行有 8 个步骤，表示写一个字节。第 3 行 isQ 为 1 表示 SDA 为输出状态。第 4 行表示从最高位开始更新 SDA 的数据位。第 5~6 行表示，C1 为 0 拉低 SCL，C1 为 TF+TLOW 则拉高 SCL。第 7~8 行表示该步骤逗留一个周期的时间。

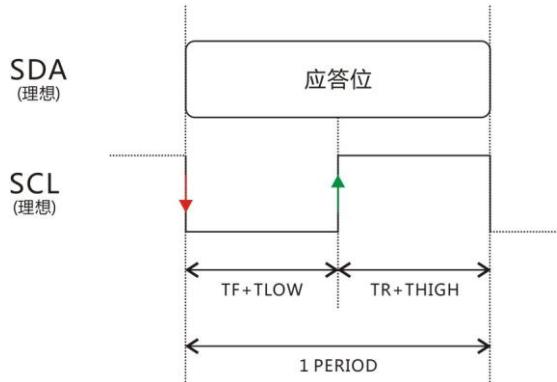


图 16.10 应答位。

应答位是从机给予主机的回答，0 为是 1 为否。然而，从旁观看，读取应答位也是读取一位数据位。当主机完成写入一个字节或者读取一个字节数据的时候，从机都会产生应答位。主机拉低 SCL 那刻，从机便会发送应答位，然后主机会借由上升沿读取应答位。如图 16.10 所示，上升沿会产生在 $TF + TLOW$ 之后，也是 $1/2$ 周期。对此，Verilog 可以这样表示，结果如代码 16.6 所示：

```

1. begin
2.     isQ = 1'b0;
3.     if( C1 == FHALF ) isAck <= SDA;
4.     if( C1 == 0 ) rSCL <= 1'b0;
5.     else if( C1 == FHALF ) rSCL <= 1'b1;
6.     if( C1 == FCLK -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
7.     else C1 <= C1 + 1'b1;

```

```
8. end
```

代码 16.6

如代码 16.6 所示，第 2 行表示 SDA 为输入状态。第 4~5 行表示，C1 为 0 拉低 SCL，C1 为 1/2 周期则拉高 SCL。第 3 行表示，C1 为 1/2 周期的时候读取应答位。第 6~7 行表示该步骤逗留 1 个周期的时间。

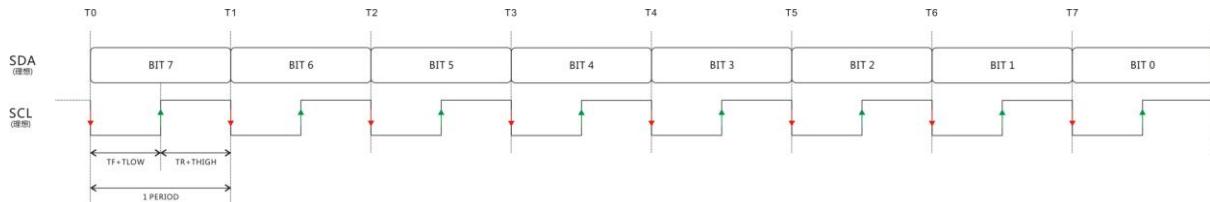


图 16.11 读一字节。

所谓读一字节数据就是重复读取 8 次应答位。如图 16.11 所示，SCL 的下降沿导致从机更新数据，然后主机在 SCL 的上升沿读取数据。此外，从机也会由高至低更新数据位。至于 Verilog 则可以这样表示，结果如代码 16.7 所示：

```
1. 0,1,2,3,4,5,6,7 :  
2. begin  
3.     isQ = 1'b0;  
4.     if( C1 == FHALF ) D1[7-i] <= SDA;  
5.     if( C1 == 0 ) rSCL <= 1'b0;  
6.     else if( C1 == FHALF ) rSCL <= 1'b1;  
7.     if( C1 == FCLK - 1 ) begin C1 <= 10'd0; i <= i + 1'b1; end  
8.     else C1 <= C1 + 1'b1;  
9. end
```

代码 16.7

如代码 16.7 所示，第 1 行表示读取一字节。第 3 行表示 SDA 为输入状态，第 5~6 行表示，C1 为 0 拉低 SCL，C1 为 1/2 周期则拉高 SCL。第 4 行表示，C1 为 1/2 周期的时候读取数据，而且数据位由高至低存入 D1。第 7~8 行表示该步骤逗留一个周期的时间。

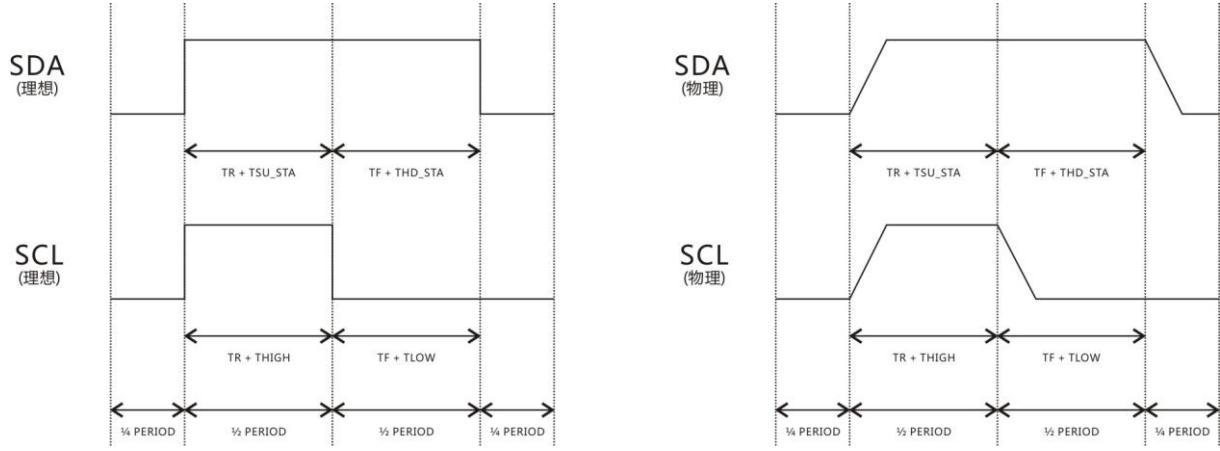


图 16.12 第二次起始位。

我们知道主机向从机读取数据的时候，它必须改变设备地址的方向，因此读操作又第二次起始位。如图 16.12 所示，感觉上第二次起始位也是第一次起始位，不过为了促使改变方向成功，第二次起始位相较第一次起始位的前后都拉低 $1/4$ 周期。对此，Verilog 可以这样表示，结果如代码 16.8 所示：

```

1. begin
2.     isQ = 1'b1;
3.     if( C1 == 0 ) rSCL <= 1'b0;
4.     else if( C1 == FQUARTER ) rSCL <= 1'b1;
5.     else if( C1 == (FQUARTER + TR + TSU_STA + THD_STA + TF) ) rSCL <= 1'b0;
6.
7.     if( C1 == 0 ) rSDA <= 1'b0;
8.     else if( C1 == FQUARTER ) rSDA <= 1'b1;
9.     else if( C1 == ( FQUARTER + TR + THIGH) ) rSDA <= 1'b0;
10.
11.    if( C1 == (FQUARTER + FCLK + FQUARTER) -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
12.    else C1 <= C1 + 1'b1;
13. end

```

代码 16.8

如代码 16.8 所示，第 2 行表示 SDA 为输出状态。第 3~5 行表示，C1 为 0 拉低 SCL，C1 为 $1/4$ 周期拉高 SCL，C1 为 $1/4$ 周期 + TR + TSU_STA + THD_STA + TF 便拉低 SCL。第 7~9 行表示，C1 为 0 拉低 SDA，C1 为 $1/4$ 周期拉高 SDA，C1 为 $1/4$ 周期 + TR + THIGH 便拉低 SDA。第 11~12 行表示该步骤停留一个周期的时间。

理解完毕以后，我们便可以开始建模了。

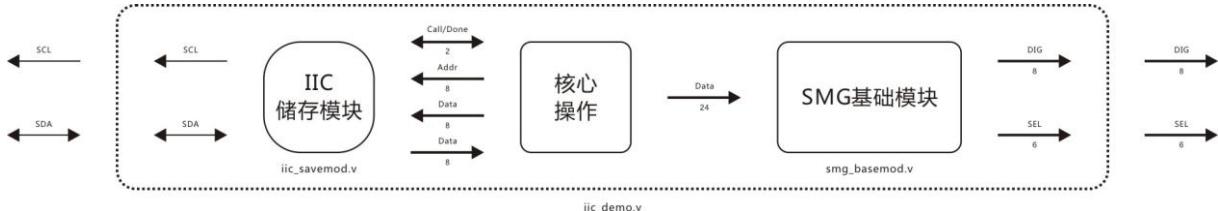


图 16.13 实验十六的建模图。

图 16.13 是实验十六的建模图，组合模块 `iic_demo` 内容包含 IIC 储存模块，核心操作还有 SMG 基础模块。首先核心操作会将数据写入 IIC 储存模块，然后又从中读取，完后再将读出的数据驱动 SMG 基础模块。

iic_savemod.v



图 16.14 IIC 储存模块的建模图。

图 16.14 是 IIC 储存模块的建模图，左边是顶层信号，右边则是沟通用的问答信号，写入地址 `iAddr`，写入数据 `iData`，还有读出数据 `oData`。Call/Done 有两位，即表示该模块有读功能还有些功能。具体内容，我们还是来看代码吧：

```

1. module iic_savemod
2. (
3.     input CLOCK, RESET,
4.     output SCL,
5.     inout SDA,
6.     input [1:0]iCall,
7.     output oDone,
8.     input [7:0]iAddr,
9.     input [7:0]iData,
10.    output [7:0]oData
11. );

```

以上内容为相关的出入端声明。

```

12. parameter FCLK = 10'd125, FHALF = 10'd62, FQUARTER = 10'd31; // (1/400E+3)/(1/50E+6)
13. parameter THIGH = 10'd30, TLOW = 10'd65, TR = 10'd15, TF = 10'd15;
14. parameter THD_STA = 10'd30, TSU_STA = 10'd30, TSU_STO = 10'd30;
15. parameter FF_Write1 = 5'd7;

```

```
16.      parameter FF_Write2 = 5'd9, RDFUNC = 5'd19;
17.
```

以上内容为相关的速率还有时序参数声明。第 15~16 行则是相关的伪函数声明。

```
18.      reg [4:0]i;
19.      reg [4:0]Go;
20.      reg [9:0]C1;
21.      reg [7:0]D1;
22.      reg rSCL,rSDA;
23.      reg isAck, isDone, isQ;
24.
25.      always @ ( posedge CLOCK or negedge RESET )
26.          if( !RESET )
27.              begin
28.                  { i,Go } <= { 5'd0,5'd0 };
29.                  C1 <= 10'd0;
30.                  D1 <= 8'd0;
31.                  { rSCL,rSDA,isAck,isDone,isQ } <= 5'b11101;
32.              end
```

以上内容为相关的寄存器声明以及复位操作。

```
33.      else if( iCall[1] )
34.          case( i )
35.
36.              0: // Call
37.              begin
38.                  isQ = 1;
39.                  rSCL <= 1'b1;
40.
41.                  if( C1 == 0 ) rSDA <= 1'b1;
42.                  else if( C1 == (TR + THIGH) ) rSDA <= 1'b0;
43.
44.                  if( C1 == (FCLK) -1) begin C1 <= 10'd0; i <= i + 1'b1; end
45.                  else C1 <= C1 + 1'b1;
46.              end
47.
```

以上内容为部分核心操作。第 33 行的 iCall[1] 为使能写操作。步骤 0 用来产生起始位。

```
48.          1: // Write Device Addr
49.          begin D1 <= {4'b1010, 3'b000, 1'b0}; i <= 5'd7; Go <= i + 1'b1; end
```

```
50.  
51.          2: // Wirte Word Addr  
52.          begin D1 <= iAddr; i <= FF_Write1; Go <= i + 1'b1; end  
53.  
54.          3: // Write Data  
55.          begin D1 <= iData; i <= FF_Write1; Go <= i + 1'b1; end  
56.  
57.          ****  
58.
```

以上内容为部分核心操作。步骤 1 用来写入设备地址，并且调用伪函数。步骤 2 用来写入数据地址，并且调用伪函数。步骤 3 用来写入数据，并且调用伪函数。

```
59.          4: // Stop  
60.          begin  
61.              isQ = 1'b1;  
62.  
63.              if( C1 == 0 ) rSCL <= 1'b0;  
64.              else if( C1 == FQUARTER ) rSCL <= 1'b1;  
65.  
66.              if( C1 == 0 ) rSDA <= 1'b0;  
67.              else if( C1 == (FQUARTER + TR + TSU_STO) ) rSDA <= 1'b1;  
68.  
69.              if( C1 == (FQUARTER + FCLK) - 1 ) begin C1 <= 10'd0; i <= i + 1'b1; end  
70.              else C1 <= C1 + 1'b1;  
71.          end  
72.
```

以上内容为部分核心操作。步骤 4 用来产生结束位。

```
73.          5:  
74.          begin isDone <= 1'b1; i <= i + 1'b1; end  
75.  
76.          6:  
77.          begin isDone <= 1'b0; i <= 5'd0; end  
78.
```

以上内容为部分核心操作。步骤 5~6 用来产生完成信号。

```
79.          **** //function  
80.  
81.          7,8,9,10,11,12,13,14:  
82.          begin
```

```

83.           isQ = 1'b1;
84.           rSDA <= D1[14-i];
85.
86.           if( C1 == 0 ) rSCL <= 1'b0;
87.           else if( C1 == (TF + TLOW) ) rSCL <= 1'b1;
88.
89.           if( C1 == FCLK -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
90.           else C1 <= C1 + 1'b1;
91.       end
92.

```

以上内容为部分核心操作。步骤 7~14 是写一个字节的伪函数。

```

93.           15: // waiting for acknowledge
94.           begin
95.               isQ = 1'b0;
96.               if( C1 == FHALF ) isAck <= SDA;
97.
98.               if( C1 == 0 ) rSCL <= 1'b0;
99.               else if( C1 == FHALF ) rSCL <= 1'b1;
100.
101.              if( C1 == FCLK -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
102.              else C1 <= C1 + 1'b1;
103.          end
104.
105.          16:
106.          if( isAck != 0 ) i <= 5'd0;
107.          else i <= Go;
108.
109.          /***** // end function
110.
111.      endcase
112.

```

以上内容为部分核心操作。步骤 15 则用来读取应答位，步骤 16 则用来判断应答位，应答成功返回步骤，失败则重新来过。

```

113.      else if( iCall[0] )
114.          case( i )
115.
116.              0: // Start
117.              begin
118.                  isQ = 1;

```

```

119.          rSCL <= 1'b1;
120.
121.          if( C1 == 0 ) rSDA <= 1'b1;
122.          else if( C1 == (TR + THIGH) ) rSDA <= 1'b0;
123.
124.          if( C1 == FCLK -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
125.          else C1 <= C1 + 1'b1;
126.
127.      end

```

以上内容为部分核心操作。第 113 行表示 iCall[0] 使能读操作。步骤 0 用来产生起始位。

```

128.      1: // Write Device Addr
129.      begin D1 <= {4'b1010, 3'b000, 1'b0}; i <= 5'd9; Go <= i + 1'b1; end
130.
131.      2: // Wirte Word Addr
132.      begin D1 <= iAddr; i <= FF_Write2; Go <= i + 1'b1; end
133.
134.      3: // Start again
135.      begin
136.          isQ = 1'b1;
137.
138.          if( C1 == 0 ) rSCL <= 1'b0;
139.          else if( C1 == FQUARTER ) rSCL <= 1'b1;
140.          else if( C1 == (FQUARTER + TR + TSU_STA + THD_STA + TF) ) rSCL <= 1'b0;
141.
142.          if( C1 == 0 ) rSDA <= 1'b0;
143.          else if( C1 == FQUARTER ) rSDA <= 1'b1;
144.          else if( C1 == ( FQUARTER + TR + THIGH ) ) rSDA <= 1'b0;
145.
146.          if( C1 == (FQUARTER + FCLK + FQUARTER) -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
147.          else C1 <= C1 + 1'b1;
148.
149.      end

```

以上内容为部分核心操作。步骤 1 用来写入设备地址，并且调用伪函数。步骤 2 用来写入数据地址，并且调用伪函数。步骤 3 用来产生第二次起始位。

```

150.      4: // Write Device Addr ( Read )
151.      begin D1 <= {4'b1010, 3'b000, 1'b1}; i <= 5'd9; Go <= i + 1'b1; end
152.
153.      5: // Read Data
154.      begin D1 <= 8'd0; i <= RDFUNC; Go <= i + 1'b1; end

```

```

155.
156.           6: // Stop
157.           begin
158.             isQ = 1'b1;
159.
160.             if( C1 == 0 ) rSCL <= 1'b0;
161.             else if( C1 == FQUARTER ) rSCL <= 1'b1;
162.
163.             if( C1 == 0 ) rSDA <= 1'b0;
164.             else if( C1 == (FQUARTER + TR + TSU_STO) ) rSDA <= 1'b1;
165.
166.             if( C1 == (FCLK + FQUARTER) -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
167.             else C1 <= C1 + 1'b1;
168.           end
169.
170.           7:
171.           begin isDone <= 1'b1; i <= i + 1'b1; end
172.
173.           8:
174.           begin isDone <= 1'b0; i <= 5'd0; end
175.

```

以上内容为部分核心操作。步骤 4 用来写入设备地址（读），并且调用伪函数。步骤 5 用来读取一个字节，并且调用伪函数。步骤 6 用来产生结束位。步骤 7~8 则用来产生完成信号。

```

176.           ****//function
177.
178.           9,10,11,12,13,14,15,16:
179.           begin
180.             isQ = 1'b1;
181.
182.             rSDA <= D1[16-i];
183.
184.             if( C1 == 0 ) rSCL <= 1'b0;
185.             else if( C1 == (TF + TLOW) ) rSCL <= 1'b1;
186.
187.             if( C1 == FCLK -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
188.             else C1 <= C1 + 1'b1;
189.           end
190.

```

以上内容为部分核心操作。步骤 9~16 是用来写一字节的伪函数。

```

191.      17: // waiting for acknowledge
192.      begin
193.          isQ = 1'b0;
194.
195.          if( C1 == FHALF ) isAck <= SDA;
196.
197.          if( C1 == 0 ) rSCL <= 1'b0;
198.          else if( C1 == FHALF ) rSCL <= 1'b1;
199.
200.          if( C1 == FCLK -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
201.          else C1 <= C1 + 1'b1;
202.      end
203.
204.      18:
205.      if( isAck != 0 ) i <= 5'd0;
206.      else i <= Go;
207.

```

以上内容为部分核心操作。步骤 17 用来读取应答位，步骤 18 则用来判断应答位。

```

208.      *****/
209.
210.      19,20,21,22,23,24,25,26: // Read
211.      begin
212.          isQ = 1'b0;
213.          if( C1 == FHALF ) D1[26-i] <= SDA;
214.
215.          if( C1 == 0 ) rSCL <= 1'b0;
216.          else if( C1 == FHALF ) rSCL <= 1'b1;
217.
218.          if( C1 == FCLK -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
219.          else C1 <= C1 + 1'b1;
220.      end
221.

```

以上内容为部分核心操作。步骤 19~26 是读取一字节的伪函数。

```

222.      27: // no acknowledge
223.      begin
224.          isQ = 1'b1;
225.          //if( C1 == 100 ) isAck <= SDA;
226.

```

```

227.           if( C1 == 0 ) rSCL <= 1'b0;
228.           else if( C1 == FHALF ) rSCL <= 1'b1;
229.
230.           if( C1 == FCLK -1 ) begin C1 <= 10'd0; i <= Go; end
231.           else C1 <= C1 + 1'b1;
232.       end
233.
234.   /**************************************************************************// end fucntion
235.
236.   endcase
237.
```

以上内容为部分核心操作。步骤 27 用来无视应答位。

```

238.   /*************************************************************************/
239.
240.   assign SCL = rSCL;
241.   assign SDA = isQ ? rSDA : 1'bz;
242.   assign oDone = isDone;
243.   assign oData = D1;
244.
245.   /*************************************************************************/
246.
247. endmodule
```

以上内容为相关的驱动声明。

iic_demo.v

该组合模块的连线部署请参考图 16.13，具体内容让我们来看代码吧。

```

1. module iic_demo
2. (
3.     input CLOCK, RESET,
4.     output SCL,
5.     inout SDA,
6.     output [7:0]DIG,
7.     output [5:0]SEL
8. );
```

以上内容为相关的出入端声明。

```
9.     wire [7:0]DataU1;
```

```

10.      wire DoneU1;
11.
12.      iic_savemod U1
13.      (
14.          .CLOCK( CLOCK ),
15.          .RESET( RESET ),
16.          .SCL( SCL ),           // > top
17.          .SDA( SDA ),           // <> top
18.          .iCall( isCall ),    // < core
19.          .oDone( DoneU1 ),    // > core
20.          .iAddr( D1 ),        // < core
21.          .iData( D2 ),        // < core
22.          .oData( DataU1 )     // > core
23.      );
24.

```

以上内容为 IIC 储存模块的实例化。

```

25.      smg_basemod U2
26.      (
27.          .CLOCK( CLOCK ),
28.          .RESET( RESET ),
29.          .DIG( DIG ),           // > top
30.          .SEL( SEL ),           // > top
31.          .iData( D3 )           // < core
32.      );
33.

```

以上内容为数码管基础模块的实例化。

```

34.      *****/
35.
36.      reg [3:0]i;
37.      reg [7:0]D1,D2;
38.      reg [23:0]D3;
39.      reg [1:0]isCall;
40.
41.      always @ ( posedge CLOCK or negedge RESET ) // core
42.          if( !RESET )
43.              begin
44.                  i <= 4'd0;
45.                  { D1,D2 } <= { 8'd0,8'd0 };
46.                  D3 <= 24'd0;

```

```
47.           isCall <= 2'b00;
48.       end
49.   else
```

以上内容为相关的寄存器声明以及复位操作。

```
50.           case( i )
51.
52.               0:
53.                   if( DoneU1 ) begin isCall <= 2'b00; i <= i + 1'b1; end
54.                   else begin isCall <= 2'b10; D1 <= 8'd0; D2 <= 8'hAB; end
55.
56.               1:
57.                   if( DoneU1 ) begin isCall <= 2'b00; i <= i + 1'b1; end
58.                   else begin isCall <= 2'b10; D1 <= 8'd1; D2 <= 8'hCD; end
59.
60.               2:
61.                   if( DoneU1 ) begin isCall <= 2'b00; i <= i + 1'b1; end
62.                   else begin isCall <= 2'b10; D1 <= 8'd2; D2 <= 8'hEF; end
63.
64.               3:
65.                   if( DoneU1 ) begin D3[23:16] <= DataU1; isCall <= 2'b00; i <= i + 1'b1; end
66.                   else begin isCall <= 2'b01; D1 <= 8'd0; end
67.
68.               4:
69.                   if( DoneU1 ) begin D3[15:8] <= DataU1; isCall <= 2'b00; i <= i + 1'b1; end
70.                   else begin isCall <= 2'b01; D1 <= 8'd1; end
71.
72.               5:
73.                   if( DoneU1 ) begin D3[7:0] <= DataU1; isCall <= 2'b00; i <= i + 1'b1; end
74.                   else begin isCall <= 2'b01; D1 <= 8'd2; end
75.
76.               6:
77.                   i <= i;
78.
79.           endcase
80.
81. endmodule
```

以上内容为核心操作。步骤 0~2 将数据 8'hAB 写入地址 0，8'hCD 写入地址 1，8'hEF 写入地址 2。步骤 3~5 则是从地址 0 读出数据 8'hAB 并且暂存至 D3[23:16]，从地址 1 读出数据 8'hCD 并且暂存至 D3[15:8]，从地址 2 读出数据 8'hEF 并且暂存至 D3[7:0]。编辑完毕便下载程序，如果数码管从左至右显示“ABCDEF”，那么表示实验

成功。

细节一： IIC 储存模块，还是 IIC 功能模块？

有关 IIC 储存器的实验曾在《整合篇》出现过，不过是作为功能类来对待。换之，本实验的 IIC 储存器则作为储存类来看待，然而它究竟是功能类还是储存类呢？其实这是见仁见智的问题。如果读者认为功能类有助理解，那么它就是功能类 … 相反的，笔者认为储存类有助理解，所以承认它就是储存类。

细节二： 100Khz 与 400Khz 速率

IIC 储存器——24LC04 有两种速率供我们选择，100Khz 是比较规格的速率，因为 SCL 有 50% 的占空比，反之 400Khz 则是比较不规格的速率，因为 SCL 的前半周期为 36%，后半周期为 64%。审美而言，100Khz 比 400Khz 美丽 … 速度而言，400Khz 比 100Khz 快 4 倍。100Khz 的时序参数还有 50Mhz 量化结果如表 16.3 所示：

表 16.3 相关的时序参数 (50Mhz 量化)

相关参数	标示	最小时间	最小时钟	最大时间	最大时钟
Clock Frequency	FCLK	---	---	100Khz	500
Clock High Time	THIGH	4000ns	200	---	---
Clock Low Time	TLOW	4700ns	235	---	---
Rise Time	TR	---	---	1000ns	50
Fall Time	TF	---	---	300ns	15
Start Hold Time	THD_STA	4000ns	200	---	---
Start Setup Time	TSU_STA	4700ns	235	---	---
Data Input Hold Time	THD_DAT	0ns	0	---	---
Data Input Setup Time	TSU_DAT	250ns	12	---	---
Stop Setup Time	TSU_STO	4000ns	200	---	---
Output Valid From Clock	TAA	---	---	3500ns	175
Bus Free Time	TBUF	4700ns	235	---	---

Verilog 的常量声明如代码 16.9 所示：

```
1. parameter FCLK = 10'd500, FHALF = 10'd250, FQUARTER = 10'd125;
2. parameter THIGH = 10'd200, TLOW = 10'd235, TR = 10'd50, TF = 10'd15;
3. parameter THD_STA = 10'd200, TSU_STA = 10'd235, TSU_STO = 10'd200;
```

代码 16.9

细节三：完整的个体模块

实验十六的 IIC 储存模块已经是完整的个体模块，随之可以调用。

实验十七：IIC 储存模块 - FIFO 读写

```
1. int main()
2. {
3.     int A;
4.     A = 16;
5. }
```

代码 17.1

话题为进入之前，首先让我们来聊聊一些题外话。那些学过软核 NIOS 的朋友可曾记得，软核 NIOS 可利用片上内存作为储存资源，而且它也能利用 SDRAM 作为储存资源，然而问题是在这里 … 如代码 17.1 所示，笔者先建立变量 A，然后变量 A 赋值 16。如果站在高级语言上的角度去思考，无论是建立变量 A 还是为变量 A 赋值，我们没有必要去理解变量 A 利用什么储存资源，然后赋值变量 A 又是利用怎样的储存功能去实现。

我们只要负责轻松的表层工作，然而那些辛苦的底层工作统统交由编译器处理。读者也许会认为那是高级语言的温柔，不过这股温柔却不适合描述语言，还不如说那是一种抹杀性的伤害。这种感觉好比父母亲过度溺爱自己的孩子，娇生惯养的孩子最终只会失去可能性而已。

笔者曾在前面说过，储存模块基本上可以分为“储存资源”还有“储存方式”。默认下，描述语言可用的储存资源有寄存器还有片上内存，然而两者都是内在资源。换之，实验十六却利用外在的储存资源，IIC 储存器。

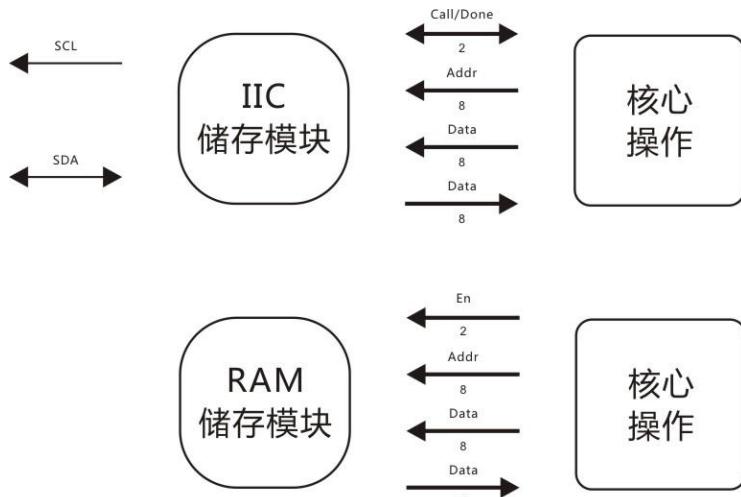


图 17.1 IIC 储存模块与 RAM 储存模块。

如图 17.1 所示，那是实验十六的 IIC 储存模块，然而图 17.1 也表示 IIC 储存模块的调用方法也不仅近似 RAM 储存模块，而且只有一方调用它而已。这种感觉好比顺序语言的主函数调用某个储存函数一样，结果如代码 17.2 所示：

```
1. int ram_func( int Addr, int WrData ) { ... }
```

```

2.
3. int main()
4. {
5.     ram_func( 0 , 20 );
6.     ...
7. }

```

代码 17.2

如代码 17.2 所示，第 1 行声明函数 `ram_func`，然后主函数在第 5 行将其调用，并且传递地址参数 0，数据参数 20。高级语言是一位顺序又寂寞的家伙，函数永远只能被一方调用而已 ... 换之，描述语言是一位并行又多愁的家伙，模块有可能同时被两方以上调用，情况宛如两位男生同时追求一位少女，对此少女会烦恼选择谁。哎~这是奢侈的少女忧愁。



图 17.2 双口 RAM 储存模块。

如图 17.2 所示，RAM 储存模块同时被两方调用，周边操作为它写入数据，核心操作则为它读出数据。图 17.2 也是俗称的双口 RAM。对此，我们也可以双口 RAM 储存模块是 RAM 储存模块的亚种。



图 17.3 基于双口 RAM 储存模块的 FIFO 储存模块。

此外，双口 RAM 储存模块只要稍微更换一下马甲，然后加入一些先进先出的机制，随之基于双口 RAM 储存模块的 FIFO 储存模块便完成，结果如图 17.3 所示。对此，我们可以说 FIFO 储存模块是双口 RAM 储存模块的亚种。

那么问题来了：

“请问，实验十六的 IIC 储存模块是否也能成为双口 IIC 储存模块？”，笔者问道。
“再请问，它还可以成为有 FIFO 机制的储存模块呢？”，笔者再问道。

没错，上述问题就是实验十七的主要目的。如果这些要求可以成真，我们便可以断定描述语言不仅不逊色与顺序语言，描述语言也充满许多可能性，而且储存类作为一个模块

类有着举足轻重的地位。废话少说，我们还是开始实验吧，因为笔者已经压抑不了蛋蛋的冲动！

首先我们要明白，片上内存是效率又优秀的储存资源，基本上只要 1 个时钟就可完成读写操作，而且读写也可以同时进行，两方也可以同时调用，不过就是不能随意扩充。反之，IIC 储存器虽然可以随意扩充，但是又笨又麻烦的它，读写操作不仅用时很长，而且不能也同时进行，对此造就两方不能同时调用的问题。为此，我们必须先解决这个问题。



图 17.4 写操作与读操作。

实验十六告诉我们，IIC 储存模块有两位 Call 信号，其中 Call[1] 表示写操作，Call[0] 表示读操作。不过不管是写操作还是读操作，IIC 储存模块都必须调用 IIC 储存器，而且读写操作一次也只能进行其中一项。如图 17.4，假设左边的周边操作负责写操作，右边的核心操作负责读操作 … 如果两者同时拉高 Call 信号就会发生多义性的问题，对此笔者该如何协调呢？



图 17.5 轮流协调。

为了公平起见，笔者采取轮流的方式来协调多义性的问题。图 17.5 所示是轮流协调的概念图，一般 Call 兼职“提问与使能”，即 Call 一拉高操作便执行。如今 Call 信号作为第一层提问，isDo 则作为第二层使能 … 换句话说，不管 Call 拉不拉高，只要 isDo 不拉高，操作也不会执行。如图 17.5 所示，isDo 位宽有 3 表示模块有 3 种操作，或者说 3 个操作共享一个模块资源。至于右边是称为使能指针的箭头，它的作用是给予使能权。



图 17.6 轮流协调例子①。

如图 17.6 所示，假设 Call[2] 拉高以示提问，但是指针并没有指向它，所以它没有使能权也不能执行操作。这种情况好比举手的学生没被老师点名，这位学生就不能随意开口。当然，使能指针也不是静止不动，只要遇见有人举手提问，它便会按照顺序检测各个对

象。

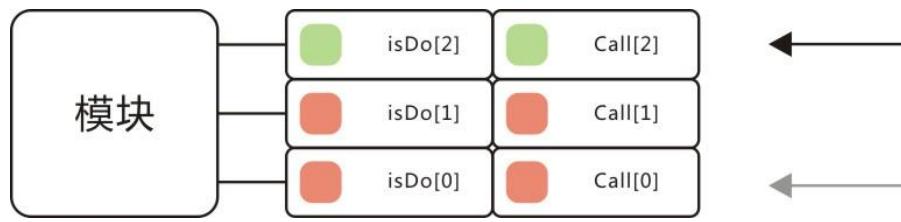


图 17.7 轮流协调例子②。

如图 17.7 所示，当指针来到 Call[2] 的面前并且给予使能权，isDo[2] 立即拉高使能操作，直至操作完成之前，该操作都享有模块的使用权。（灰色指针为过去，黑色指针为现在）

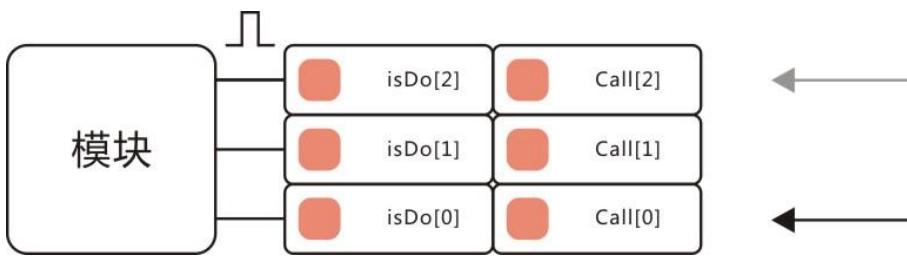


图 17.8 轮流协调例子③。

如图 17.8 所示，操作执行完毕之际，模块便会反馈完成信号以示结束操作，isDo[2] 还有 Call[2] 都会经由完成信号拉低内容。此外，指针也会立即指向下一个对象。



图 17.9 轮流协调例子④。

如图 17.9 所示，假设 Call[2] 还有 Call[1] 同时提问，由于指针没有指向它们，所以 Call[2] 与 Call[1] 都没有使能权。此刻，指针开始一动。

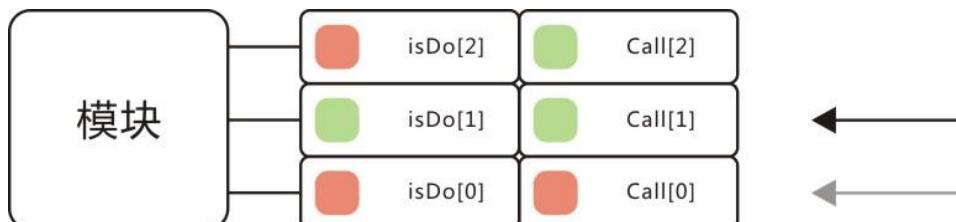


图 17.10 轮流协调例子⑤。

首先，Call[1] 会得到使能权，isDo[1] 因此拉高并且开始执行操作，直至操作结束之前，

isDo[1]都独占模块，结果如图 17.10 所示。

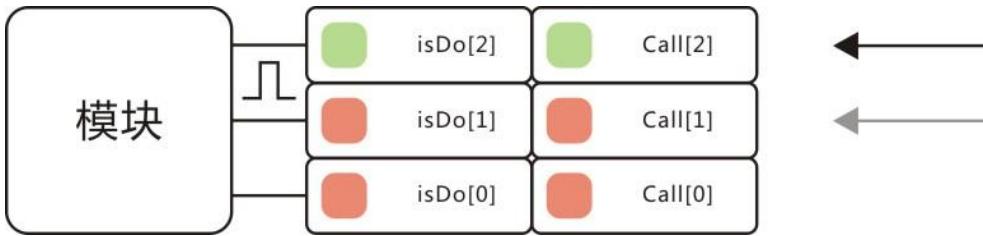


图 17.11 轮流协调例子⑥。

如图 17.11 所示，当操作执行完毕，模块便会反馈完成信号，随之 isDo[1] 还有 Call[1] 都会拉低内容，而且指针也会指向下一个对象。对此，isDo[2] 得到使能权，并且开始执行操作 ... 直至操作结束之前，它都独占模块。

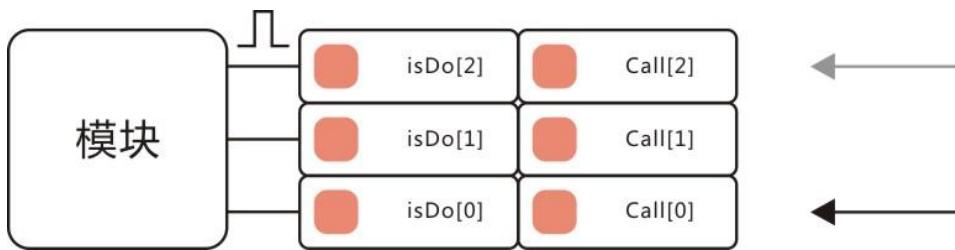


图 17.12 轮流协调例子⑦。

操作结束之际，模块便会反馈完成信号，isDo[2] 还有 Call[2] 随之也会拉低内容，然后指针指向另一个对象，结果如图 17.12 所示。轮流协调的概念基本上就是这样而已，即单纯也非常逻辑。接下来，让我们来看看 Verilog 如何描述轮流协调，结果如代码 17.3 所示：

```
1. module iic_savemod
2. (
3.     input [1:0]iCall,
4.     output [1:0]oDone,
5. );
6. reg [1:0]C7;
7. reg [1:0]isDo;
8.
9. always @ ( posedge CLOCK or negedge RESET )
10.    if( !RESET )
11.        begin
12.            C7 <= 2' b10;
13.            isDo <= 2' b00;
14.        end
15.    else
16.        begin
17.            if( iCall[1] & C7[1] ) isDo[1] <= 1' b1;
```

```

18.           else if( iCall[0] & C7[0] ) isDo[0] <= 1' b1;
19.
20.           if( isDo[1] & isDone[1] ) isDo[1] <= 1' b0;
21.           else if( isDo[0] & isDone[0] ) isDo[0] <= 1' b0;
22.
23.           if( isDone ) C7 <= { isDo[0], isDo[1] };
24.           else if( iCall ) C7 <= { C7[0], C7[1] };
25.       end
26.

```

代码 17.3

第 3~4 行是相关的出入端声明，其中 Call 还有 Done 均为两位。第 6~7 行是轮流协调作用的寄存器 isDo 与 C7，C7 为使能指针。第 10~14 行则是这些寄存器的初始化，注意 C7 默认下指向 Call[1]。第 16~25 行则是轮流协调的主要操作，第 17~18 行是提问与使能，其中 iCall[N] & C7[N] 表示提问并且被指针指向，isDo[N] 表示给予使能权。

第 20~21 行是消除提问和使能，其中 isDo[N] & isDone[N] 表示相关的完成信号对应相关操作，然后 isDo[N] 表示消除使能。第 24 行的表示有提问，指针就立即移动。第 23 行表示结束操作，指针便指向下一个对象。

```

27.   reg [4:0]i;
28.   reg [1:0]isDone;
29.
30.   always @ ( posedge CLOCK or negedge RESET )
31.     if( !RESET )
32.       begin
33.         ...
34.         i <= 5' d0;
35.         isDone <= 2' b00;
36.       end
37.     else if( isDo[1] )
38.       case( i )
39.         ...
40.           5: begin isDone[1] <= 1' b1; i <= i + 1' b1; end
41.           6: begin isDone[1] <= 1' b0; i <= 5' d0; end
42.       endcase
43.     else if( isDo[0] )
44.       case( i )
45.         ...
46.           7: begin isDone[0] <= 1' b1; i <= i + 1' b1; end
47.           8: begin isDone[0] <= 1' b0; i <= 5' d0; end
48.       endcase
49.

```

```
50. endmodule
```

代码 17.3

第 27~28 行只核心操作相关的寄存器，第 31~36 行则是这些寄存器的复位操作。第 37 行表示 `isDo[1]` 拉高才执行操作 1。第 40~41 行表示操作 1 反馈完成信号。第 43 行表示 `isDo[0]` 拉高才指向操作 0。第 46~47 行表示操作 0 反馈完成信号。如此一来，问答信号便有轮流协调，接下来就是为 IIC 储存模块加入 FIFO 机制。



图 17.13 有 FIFO 机制的 IIC 储存模块。

如图 17.13 所示，那是拥有 FIFO 机制的 IIC 储存模块，它那畸形的储存功能，可谓是 IIC 储存模块的亚种。其中 `Call/Done[1]` 表示写入调用，`Tag[1]` 表示写满状态，反之既然。由于目前的 IIC 储存模块是 FIFO 的关系，所以写入地址还有读出地址都是在里边建立。为此，Verilog 可以这样描述，结果如代码 17.4 所示：

```
1. module iic_savemod
2. (
3.     input [1:0]iCall,
4.     output [1:0]oDone,
5.     input [7:0]iData,
6.     output [7:0]oData,
7.     output [1:0]oTag
8. );
9.     always @ ( posedge CLOCK or negedge RESET ) // 轮流协调的周边操作
10.    ...
11.
12.    reg [8:0]C2,C3; // C2 Write Pointer, C3 Read Pointer;
13.
14.    always @ ( posedge CLOCK or negedge RESET )
15.        if( !RESET )
16.            begin
17.                C2 <= 9' d0;
18.                C3 <= 9' d0;
19.            end
20.        else if( isDo[1] )
21.            case( i )
22.                ...
```

```

23.          2: // Write Word Addr
24.          begin D1 <= C2[7:0]; i <= FF_Write1; Go <= i + 1'b1; end
25.          ...
26.          5:
27.          begin C2 <= C2 + 1'b1; isDone[1] <= 1'b1; i <= i + 1'b1; end
28.          ...
29.      endcase
30.  else if( isDo[0] )
31.      case( i )
32.          ...
33.          2: // Write Word Addr
34.          begin D1 <= C3[7:0]; i <= FF_Write2; Go <= i + 1'b1; end
35.          ...
36.          7:
37.          begin C3 <= C3 + 1'b1; isDone[0] <= 1'b1; i <= i + 1'b1; end
38.          ...
39.      endcase
40.
41.  ...
42.  assign oTag[1] = ( (C2[8]^C3[8]) && (C2[7:0] == C3[7:0]) );
43.  assign oTag[0] = ( C2 == C3 );
44.
45. endmodule

```

代码 17.4

如代码 17.4 所示，第 3~7 行是相关的出入端声明。第 12 行建立相关的寄存器，C2 为写指针，C3 为读指针，位宽为 $N + 1$ 。第 23~24 行表示 C2[7:0] 为写数据地址。第 26~27 行表示 C2 递增。第 33~34 行表示 C3[7:0] 为读数据地址。第 36~37 行表示 C3 递增。第 42 行表示写满状态，第 43 行则表示读空状态。完后，我们便可以开始建模了。

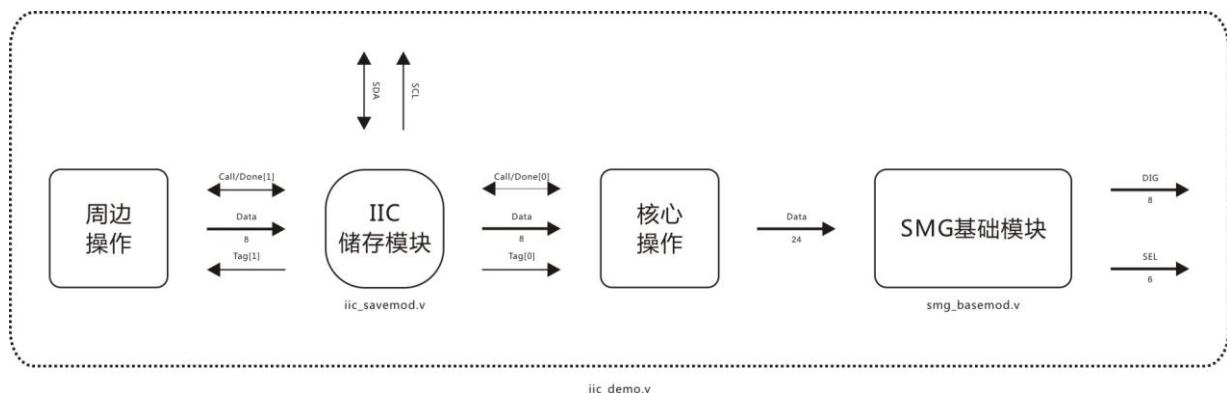


图 17.14 实验十七的建模图。

图 17.14 是实验十七的建模图，周边操作为 IIC 储存模块写入数据，核心操作则从哪里

读取数据，并且将读出的数据驱动数码管基础模块。

iic_savemod.v

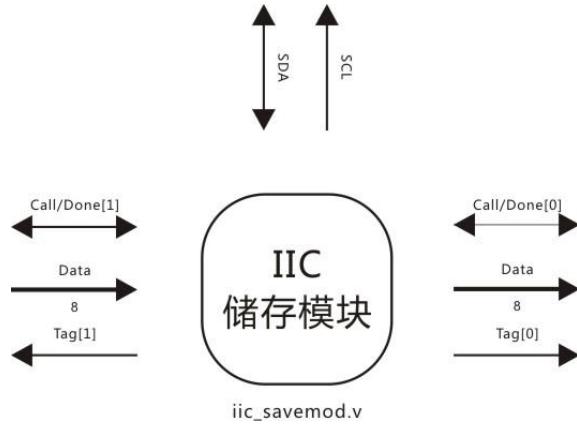


图 17.15 IIC 储存模块。

图 17.15 是 IIC 储存模块的建模图，左方是写入操作，右边是读出操作，上方则是链接至顶层信号 SCL 与 SDA。

```
1. module iic_savemod
2. (
3.     input CLOCK, RESET,
4.     output SCL,
5.     inout SDA,
6.     input [1:0]iCall,
7.     output [1:0]oDone,
8.     input [7:0]iData,
9.     output [7:0]oData,
10.    output [1:0]oTag
11. );
12. parameter FCLK = 10'd125, FHALF = 10'd62, FQUARTER = 10'd31; // (1/400E+3)/(1/50E+6)
13. parameter THIGH = 10'd30, TLOW = 10'd65, TR = 10'd15, TF = 10'd15;
14. parameter THD_STA = 10'd30, TSU_STA = 10'd30, TSU_STO = 10'd30;
15. parameter FF_Write1 = 5'd7;
16. parameter FF_Write2 = 5'd9, FF_Read = 5'd19;
17.
18. /*****
19.
20. reg [1:0]C7;
21. reg [1:0]isDo;
22.
23. always @ ( posedge CLOCK or negedge RESET )
```

```

24.         if( !RESET )
25.             begin
26.                 C7 <= 2'b10;
27.                 isDo <= 2'b00;
28.             end
29.         else
30.             begin
31.
32.                 if( iCall[1] & C7[1] ) isDo[1] <= 1'b1;
33.                 else if( iCall[0] & C7[0] ) isDo[0] <= 1'b1;
34.
35.                 if( isDo[1] & isDone[1] ) isDo[1] <= 1'b0;
36.                 else if( isDo[0] & isDone[0] ) isDo[0] <= 1'b0;
37.
38.                 if( isDone ) C7 <= {isDo[0],isDo[1]};
39.                 else if( iCall ) C7 <= { C7[0], C7[1] };
40.
41.             end
42.
43.
44.     /******/
45.
46.     reg [4:0]i;
47.     reg [4:0]Go;
48.     reg [9:0]C1;
49.     reg [7:0]D1;
50.     reg [1:0]isDone;
51.     reg [8:0]C2,C3; // C2 Write Pointer, C3 Read Pointer
52.     reg rSCL,rSDA;
53.     reg isAck,isQ;
54.
55.     always @ ( posedge CLOCK or negedge RESET )
56.         if( !RESET )
57.             begin
58.                 { i,Go } <= { 5'd0,5'd0 };
59.                 C1 <= 10'd0;
60.                 D1 <= 8'd0;
61.                 isDone <= 2'd0;
62.                 { C2, C3 } <= 18'd0;
63.                 { rSCL,rSDA,isAck,isQ } <= 4'b1111;
64.             end
65.         else if( isDo[1] )
66.             case( i )

```

```

67.
68.          0: // Call
69.          begin
70.              isQ = 1;
71.              rSCL <= 1'b1;
72.
73.              if( C1 == 0 ) rSDA <= 1'b1;
74.              else if( C1 == (TR + THIGH) ) rSDA <= 1'b0;
75.
76.              if( C1 == (FCLK) -1) begin C1 <= 10'd0; i <= i + 1'b1; end
77.              else C1 <= C1 + 1'b1;
78.          end
79.
80.          1: // Write Device Addr
81.          begin D1 <= {4'b1010, 3'b000, 1'b0}; i <= 5'd7; Go <= i + 1'b1; end
82.
83.          2: // Wirte Word Addr
84.          begin D1 <= C2[7:0]; i <= FF_Write1; Go <= i + 1'b1; end
85.
86.          3: // Write Data
87.          begin D1 <= iData; i <= FF_Write1; Go <= i + 1'b1; end
88.
89.          /***** */
90.
91.          4: // Stop
92.          begin
93.              isQ = 1'b1;
94.
95.              if( C1 == 0 ) rSCL <= 1'b0;
96.              else if( C1 == FQUARTER ) rSCL <= 1'b1;
97.
98.              if( C1 == 0 ) rSDA <= 1'b0;
99.              else if( C1 == (FQUARTER + TR + TSU_STO ) ) rSDA <= 1'b1;
100.
101.             if( C1 == (FQUARTER + FCLK) -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
102.             else C1 <= C1 + 1'b1;
103.         end
104.
105.         5:
106.         begin C2 <= C2 + 1'b1; isDone[1] <= 1'b1; i <= i + 1'b1; end
107.
108.         6:
109.         begin isDone[1] <= 1'b0; i <= 5'd0; end

```

```

110.
111.      ****//function
112.
113.      7,8,9,10,11,12,13,14:
114.      begin
115.          isQ = 1'b1;
116.          rSDA <= D1[14-i];
117.
118.          if( C1 == 0 ) rSCL <= 1'b0;
119.          else if( C1 == (TF + TLOW) ) rSCL <= 1'b1;
120.
121.          if( C1 == FCLK -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
122.          else C1 <= C1 + 1'b1;
123.      end
124.
125.      15:// waiting for acknowledge
126.      begin
127.          isQ = 1'b0;
128.          if( C1 == FHALF ) isAck <= SDA;
129.
130.          if( C1 == 0 ) rSCL <= 1'b0;
131.          else if( C1 == FHALF ) rSCL <= 1'b1;
132.
133.          if( C1 == FCLK -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
134.          else C1 <= C1 + 1'b1;
135.      end
136.
137.      16:
138.      if( isAck != 0 ) i <= 5'd0;
139.      else i <= Go;
140.
141.      ****// end function
142.
143.      endcase
144.
145.      else if( isDo[0] )
146.          case( i )
147.
148.              0:// Call
149.              begin
150.                  isQ = 1;
151.                  rSCL <= 1'b1;
152.
```

```

153.           if( C1 == 0 ) rSDA <= 1'b1;
154.           else if( C1 == (TR + THIGH) ) rSDA <= 1'b0;
155.
156.           if( C1 == FCLK -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
157.           else C1 <= C1 + 1'b1;
158.       end
159.
160.       1: // Write Device Addr
161.       begin D1 <= {4'b1010, 3'b000, 1'b0}; i <= 5'd9; Go <= i + 1'b1; end
162.
163.       2: // Write Word Addr
164.       begin D1 <= C3[7:0]; i <= FF_Write2; Go <= i + 1'b1; end
165.
166.       3: // Start again
167.       begin
168.           isQ = 1'b1;
169.
170.           if( C1 == 0 ) rSCL <= 1'b0;
171.           else if( C1 == FQUARTER ) rSCL <= 1'b1;
172.           else if( C1 == (FQUARTER + TR + TSU_STA + THD_STA + TF) ) rSCL <= 1'b0;
173.
174.           if( C1 == 0 ) rSDA <= 1'b0;
175.           else if( C1 == FQUARTER ) rSDA <= 1'b1;
176.           else if( C1 == (FQUARTER + TR + THIGH) ) rSDA <= 1'b0;
177.
178.           if( C1 == (FQUARTER + FCLK + FQUARTER) -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
179.           else C1 <= C1 + 1'b1;
180.       end
181.
182.       4: // Write Device Addr ( Read )
183.       begin D1 <= {4'b1010, 3'b000, 1'b1}; i <= 5'd9; Go <= i + 1'b1; end
184.
185.       5: // Read Data
186.       begin D1 <= 8'd0; i <= FF_Read; Go <= i + 1'b1; end
187.
188.       6: // Stop
189.       begin
190.           isQ = 1'b1;
191.
192.           if( C1 == 0 ) rSCL <= 1'b0;
193.           else if( C1 == FQUARTER ) rSCL <= 1'b1;
194.
195.           if( C1 == 0 ) rSDA <= 1'b0;

```

```

196.           else if( C1 == (FQUARTER + TR + TSU_STO) ) rSDA <= 1'b1;
197.
198.           if( C1 == (FCLK + FQUARTER) -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
199.           else C1 <= C1 + 1'b1;
200.       end
201.
202.       7:
203.       begin C3 <= C3 + 1'b1; isDone[0] <= 1'b1; i <= i + 1'b1; end
204.
205.       8:
206.       begin isDone[0] <= 1'b0; i <= 5'd0; end
207.
208.      /**************************************************************************//function
209.
210.      9,10,11,12,13,14,15,16:
211.      begin
212.          isQ = 1'b1;
213.
214.          rSDA <= D1[16-i];
215.
216.          if( C1 == 0 ) rSCL <= 1'b0;
217.          else if( C1 == (TF + TLOW) ) rSCL <= 1'b1;
218.
219.          if( C1 == FCLK -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
220.          else C1 <= C1 + 1'b1;
221.      end
222.
223.      17:// waiting for acknowledge
224.      begin
225.          isQ = 1'b0;
226.
227.          if( C1 == FHALF ) isAck <= SDA;
228.
229.          if( C1 == 0 ) rSCL <= 1'b0;
230.          else if( C1 == FHALF ) rSCL <= 1'b1;
231.
232.          if( C1 == FCLK -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
233.          else C1 <= C1 + 1'b1;
234.      end
235.
236.      18:
237.      if( isAck != 0 ) i <= 5'd0;
238.      else i <= Go;

```

```

239.
240.      *****/
241.
242.      19,20,21,22,23,24,25,26: // Read
243.      begin
244.          isQ = 1'b0;
245.          if( C1 == FHALF ) D1[26-i] <= SDA;
246.
247.          if( C1 == 0 ) rSCL <= 1'b0;
248.          else if( C1 == FHALF ) rSCL <= 1'b1;
249.
250.          if( C1 == FCLK -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
251.          else C1 <= C1 + 1'b1;
252.      end
253.
254.      27: // no acknowledge
255.      begin
256.          isQ = 1'b1;
257.          //if( C1 == 100 ) isAck <= SDA;
258.
259.          if( C1 == 0 ) rSCL <= 1'b0;
260.          else if( C1 == FHALF ) rSCL <= 1'b1;
261.
262.          if( C1 == FCLK -1 ) begin C1 <= 10'd0; i <= Go; end
263.          else C1 <= C1 + 1'b1;
264.      end
265.
266.      *****/ // end function
267.
268.      endcase
269.
270.      *****/
271.
272.      assign SCL = rSCL;
273.      assign SDA = isQ ? rSDA : 1'bz;
274.      assign oDone = isDone;
275.      assign oData = D1;
276.      assign oTag[1] = ( (C2[8]^C3[8]) && (C2[7:0] == C3[7:0]) );
277.      assign oTag[0] = ( C2 == C3 );
278.
279.      *****/
280.
281.  endmodule

```

具体内容笔者也懒得解释了，读者自己看着办吧。

iic_demo.v

连线部署请参考图 17.14。

```
1. module iic_demo
2. (
3.     input CLOCK, RESET,
4.     output SCL,
5.     inout SDA,
6.     output [7:0]DIG,
7.     output [5:0]SEL
8. );
```

以上内容为相关的出入端声明。

```
9.     reg [3:0]j;
10.    reg [7:0]D1;
11.    reg isWR;
12.
13.    always @ ( posedge CLOCK or negedge RESET )
14.        if( !RESET )
15.            begin
16.                j <= 4'd0;
17.                D1 <= 8'd0;
18.                isWR <= 1'b0;
19.            end
20.        else
21.            case(j)
22.
23.                0:
24.                    if( !TagU1[1] ) j <= j + 1'b1;
25.
26.                1:
27.                    if( DoneU1[1] ) begin isWR <= 1'b0; j <= j + 1'b1; end
28.                    else begin isWR <= 1'b1; D1 <= 8'hAB; end
29.
30.                2:
31.                    if( !TagU1[1] ) j <= j + 1'b1;
32.
33.                3:
```

```

34.           if( DoneU1[1] ) begin isWR <= 1'b0; j <= j + 1'b1; end
35.           else begin isWR <= 1'b1; D1 <= 8'hCD; end
36.
37.           4:
38.           if( !TagU1[1] ) j <= j + 1'b1;
39.
40.           5:
41.           if( DoneU1[1] ) begin isWR <= 1'b0; j <= j + 1'b1; end
42.           else begin isWR <= 1'b1; D1 <= 8'hEF; end
43.
44.           6:
45.           i <= i;
46.
47.       endcase
48.

```

以上内容为写入作用的周边操作，操作过程如下：

步骤 0，判断是否写满状态。
 步骤 1，写入数据 8'hAB；
 步骤 2，判断是否写满状态。
 步骤 3，写入数据 8'hCD；
 步骤 4，判断是否写满状态。
 步骤 5，写入数据 8'hEF；
 步骤 6，发呆。

```

49.   wire [7:0]DataU1;
50.   wire [1:0]DoneU1;
51.   wire [1:0]TagU1;
52.
53.   iic_savemod U1
54.   (
55.     .CLOCK( CLOCK ),
56.     .RESET( RESET ),
57.     .SCL( SCL ),           // > top
58.     .SDA( SDA ),           // <> top
59.     .iCall( { isWR, isRD } ), // < sub & core
60.     .oDone( DoneU1 ),      // > core
61.     .iData( D1 ),          // < core
62.     .oData( DataU1 ),      // > core
63.     .oTag( TagU1 )
64.   );
65.

```

以上内容为 IIC 储存模块的实例化。第 59 行表示 isWR 为 Call[1]，isRD 为 Call[0]。第 61 行表示 D1 驱动该输入。

```
66.      reg [3:0]i;
67.      reg [23:0]D2;
68.      reg isRD;
69.
70.      always @ ( posedge CLOCK or negedge RESET ) // core
71.          if( !RESET )
72.              begin
73.                  i <= 4'd0;
74.                  D2 <= 24'd0;
75.                  isRD <= 1'b0;
76.              end
77.          else
78.              case( i )
79.
80.                  0:
81.                      if( !TagU1[0] ) i <= i + 1'b1;
82.
83.                  1:
84.                      if( DoneU1[0] ) begin D2[23:16] <= DataU1; isRD <= 1'b0; i <= i + 1'b1; end
85.                      else isRD <= 1'b1;
86.
87.                  2:
88.                      if( !TagU1[0] ) i <= i + 1'b1;
89.
90.                  3:
91.                      if( DoneU1[0] ) begin D2[15:8] <= DataU1; isRD <= 1'b0; i <= i + 1'b1; end
92.                      else isRD <= 1'b1;
93.
94.                  4:
95.                      if( !TagU1[0] ) i <= i + 1'b1;
96.
97.                  5:
98.                      if( DoneU1[0] ) begin D2[7:0] <= DataU1; isRD <= 1'b0; i <= i + 1'b1; end
99.                      else isRD <= 1'b1;
100.
101.                 6:
102.                     i <= i;
103.
104.             endcase
```

以上内容为读出数据并且驱动数码管基础模块的核心操作，操作过程如下：

步骤 0，判断是否为读空状态。

步骤 1，读出数据 8'hAB，并且暂存至 D2[23:16]。

步骤 2，判断是否为读空状态。

步骤 3，读出数据 8'hCD，并且暂存至 D2[15:8]。

步骤 4，判断是否为读空状态。

步骤 5，读出数据 8'hEF，并且暂存至 D2[7:0]。

步骤 6，发呆。

```

106.      smg_basemod U2
107.      (
108.          .CLOCK( CLOCK ),
109.          .RESET( RESET ),
110.          .DIG( DIG ),           // > top
111.          .SEL( SEL ),           // > top
112.          .iData( D2 )          // < core
113.      );
114.
115. endmodule

```

第 106~113 行是数码管基础模块的实例化，第 112 行表示 D2 驱动该输入。编译完毕并且下载程序，如果数码管自左向右显示“ABCDEF”表示实验成功。

细节一：完整的个体模块

实验十七的 IIC 储存模块随时可以使用。

实验十八：SDRAM 模块① — 单字读写

笔者与 SDRAM 有段不短的孽缘，它作为冤魂日夜不断纠缠笔者。笔者尝试过许多方法将其退散，不过屡试屡败的笔者，最终心情像橘子一样橙。《整合篇》之际，笔者曾经大战几回儿，不过内容都是点到即止。最近它破蛊而出，日夜不停：“好~痛苦！好~痛苦！”地呻吟着，吓得笔者不敢半夜如厕。疯狂之下，誓要歪它不可 … 可恶的东西，笔者要它血债血还！

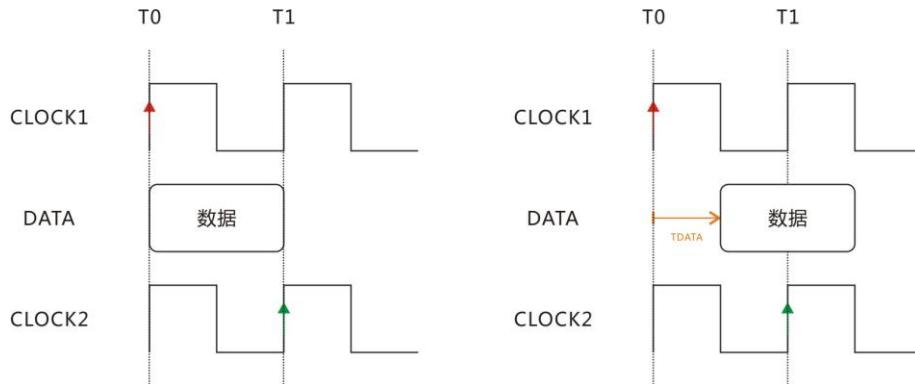


图 18.1 数据读取（理想时序左，物理时序右）。

首先，让我们来了解一下，什么才是数据读取的最佳状态？如图 18.1 所示，红色箭头是上升沿，绿色箭头是锁存沿。左图是理想时序读取数据的最佳状态，即 T0 发送数据，T1 锁存数据。右图则是物理时序读取数据的最佳状态，即 T0 发送数据，然后数据经由 TDATA 延迟，然后 T1 锁存数据。理想状态下，读取数据不用考虑任何物理因数，凡是过去值都会读取成功。

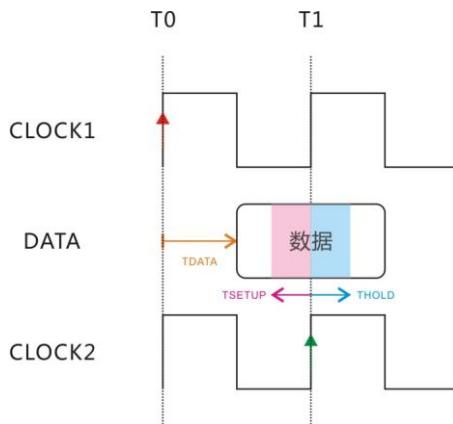


图 18.2 读取数据（物理时序）。

然而物理状态下，读取数据则必须考虑物理因数，但是物理时序也有所谓的理想状态，即数据被 TDATA 推挤，然后恰好停留在锁存沿的正中间。该状态之所以称为理想，那是因为建立时间 TSETUP 与保持时间 THOLD 都被满足。

如图 18.2 所示，TSETUP 从数据中间向左边覆盖，THOLD 从数据中间向右边覆盖，如

果两者不完全覆盖数据，那么数据的有效性就能得到保证。简言之，数据是否读取成功，建立时间还有保持时间都必须得到满足。但是我们也知道，Verilog 不能描述理想以外的东西，即 Verilog 无力描述 TDATA。话虽如此，我们可以改变时钟位移来达到同样的效果。

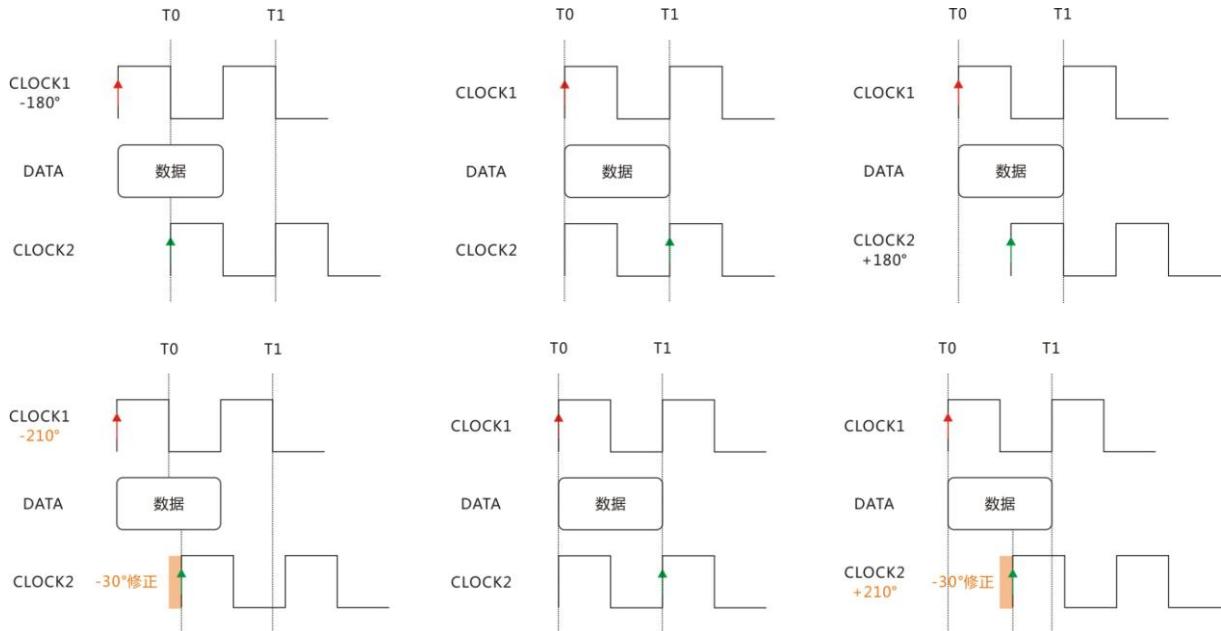


图 18.3 CLOCK1 位移 -180° (左图), 没有位移 (中图), CLOCK2 位移 $+180^\circ$ (右图), 以及修正结果。

常见的理想时序，最多适用在 FPGA 的内部而已。当描述功活动涉及 FPGA 的外部，那么理想时序必须考虑对外的情况。如图 18.3 所示，中间的理想时序图可以经由 CLOCK1 位移 -180° ，又或者 CLOCK2 位移 $+180^\circ$ 来得到同样的效果。虽说 180° 的位移是理想效果，但是我们还要考虑物理路径所带来的影响。根据 Alinx 301 这只开发板，我们必须追加 -30° 位移才能达到修正的效果。（**注意：追加 -30° 的修正时序仅仅为适用 Alinx 301 这只板子而已**）。理解完毕以后，我们便可进入正题。

驱动 SDRAM 而言，简单可以分为以下四项操作：

- (一) 初始化
- (二) 刷新操作
- (三) 读操作
- (四) 写操作

初始化令 SDRAM 就绪，刷新操作就是不失掉内容（数据），读操作就是从 SDRAM 哪里读取数据，写操作就是向 SDRAM 写数据。其中，读写操作又有单字读写，多字读写还有页读写。

首先，让我们来分析一下 Alinx 开发板上 HY57V2562GTR 这只 SDRAM。根据手册，这只 SDRAM 有 256Mb 的容量，4 个 BANK（即一个 BANK 为 64Mb），频率极限为 200Mhz，数据保留周期为 8192 / 64ms。至于引脚定义如表 18.1 所示：

表 18.1 SDRAM 的引脚定义

分类	标示	信号	说明
时钟信号	CLK	S_CLK	时钟源
地址信号	BA0~1	S_BA[1:0]	BANK 地址
	A0~A12	S_A[12:0]	读写地址，行列共用，A0~A12 为行地址，CA0~CA8 为列地址
命令信号	CKE	S_CKE,	时钟选，拉高有效
	CS	S_NCS,	片选，拉低有效
	RAS	S_NRAS,	命令选，拉低有效
	CAS	S_NCAS,	命令选，拉低有效
	WE	S_NWE	命令选，拉低有效
数据信号	DQ0~DQ15	S_DQ[15:0]	读写数据的 IO
	LDQM , UDMQ	S_DQM[1:0]	遮盖数据，一般拉低无视

如表 18.1 所示，CLK 为 SDRAM 的时钟源。CKE，CS，RAS，CAS 还有 WE 皆为命令信号，五者相互组合形成以下几个常用命令，结果如表 18.2 所示：

表 18.2 常用命令。

命令	CKE	CS	RAS	CAS	WE	说明
NOP	1	0	1	1	1	空命令
ACT	1	0	0	1	1	激活命令，选择 Bank 地址与行地址
WR	1	0	1	0	0	写命令，开始写数据
RD	1	0	1	0	1	读命令，开始读数据
BSTP	1	0	1	1	0	停止命令，停止读写
PR	1	0	0	1	0	预充命令，释放选择
AR	1	0	0	0	1	刷新命令，刷新内容
LMR	1	0	0	0	0	设置命令，设置 SDRAM

- NOP 为 No Operation，即空命令，除了给空时间以外没有任何意义。
- ACT 为 Active，即激活命令，用来选择某 Bank 某行。
- WR 为 Write，即写命令，通知设备开始写数据。
- RD 为 Read，即读命令，通知设备开始读数据。
- BSTP 为 Burst Stop，即停止命令，禁止设备继续读写。
- PR 为 Precharge，即预充命令，用来释放某 Bank 与某行的选择。
- AR 为 Auto Refresh，即刷新命令，用来刷新或者更新数据内容。
- LMR 为 Load Mode Register，即设置命令，用来配置设备参数。

Verilog 则可以这样描述这些命令，结果如代码 18.1 所示：

```
parameter _INIT = 5'b01111, _NOP = 5'b10111, _ACT = 5'b10011, _RD = 5'b10101, _WR = 5'b10100,
_BSTP = 5'b10110, _PR = 5'b10010, _AR = 5'b10001, _LMR = 5'b10000;
```

代码 18.1

DQ0~DQ15 为数据信号。BA0~1 与 A0~A12 皆为地址信号，其中 A0~A12 行列共用，然而地址信号可以指向的范围，如下计算：

$$\begin{aligned}
 2(2 \text{ Bank} + 13 \text{ Row} + 9 \text{ Column}) \times 16 \text{ bit} &= 2^{24} \times 16 \text{ bit} \\
 &= 1.6777216e7 \times 16 \text{ bit} // 16M \times 16 \text{ bit} \\
 &= 2.68435456e8 \text{ bit} \\
 &= 262144 \text{ kbit} \\
 &= 256 \text{ Mbits}
 \end{aligned}$$

初始化：

初始化除了就绪 SDRAM 以外，我们还要设置 SDRAM 内部的 Mode Register，设置内容如表 18.3 所示：

表 18.3 Mode Register 的内容。

Mode Register												
A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	0	OP Code	0	0	CAS Latency	BT	Burst Length					
1	1											
A3	Burst Type				Burst Length							
0	Sequential				A2	A1	A0	A3 = 0	A3 = 1			
1	Interleave				0	0	0	1	1			
					0	0	1	2	2			
					0	1	0	4	4			
					0	1	1	8	8			
					1	1	1	Full Page	Reserved			
A9	Write Mode				A6	A5	A4	CAS Latency				
0	Burst Read and Burst Write				0	1	0	2				
1	Burst Read and Single Write				0	1	1	3				

如表 18.3 所示，设置内容必须经由地址信号 A12~A0。其中 A2~A0 表示字读写的长度，实验十八为单字读写，所以 A2~A0 设置为 3'b000。A3 表示读写次序，1'b0 表示顺序读写。A6~A4 表示 CAS 延迟（也可以视为读出延迟），设为 3'b011 是为读出更稳定。A9 表示读写模式，一般都是设置为 1'b0。

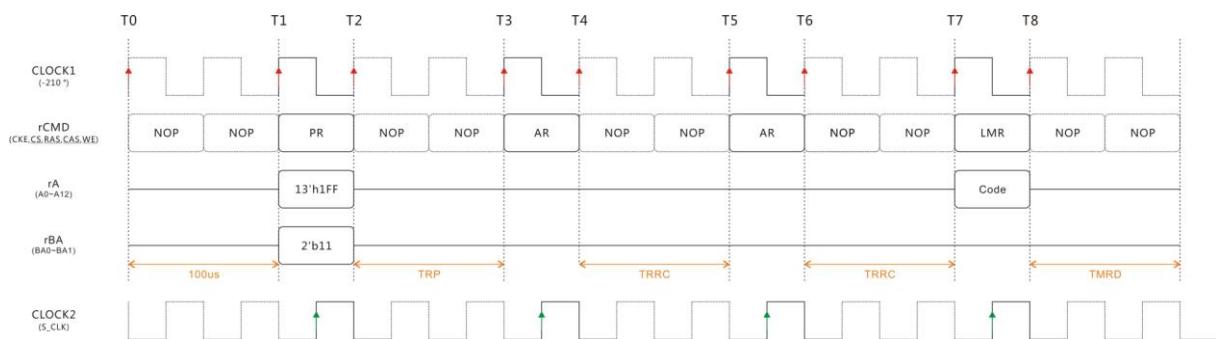


图 18.4 初始化的理想时序图。

图 18.4 是初始化的理想时序图，其中 CLOCK1 为 -210° 的系统时钟，CLOCK2 为 SDRAM 的时钟。rCMD 为 CKE, CS, RAS, CAS 还有 WE 等命令。rA 为 A0~A12, rBA 为 BA0~BA1 等地址信号。初始化过程如下所示：

- T0, 满足 100us；
- T1, 发送 PR 命令，拉高所有 rA 与 rBA。
- T1 半周期，SDRAM 读取。
- T2, 满足 TRP；
- T3, 发送 AR 命令。
- T3 半周期，SDRAM 读取。
- T4, 满足 TRRC，
- T5, 发送 AR 命令。
- T5 半周期，SDRAM 读取。
- T6, 满足 TRRC，
- T7, 发送 LMR 命令与相关 Code (设置内容)。
- T7 半周期，SDRAM 读取。
- T8, 满足 TMRD。

怎么样？读者是不是觉得很单纯呢？事后，Verilog 则可以这样描述，结果如代码 18.2 所示：

```
1. case( i )
2.
3.     0: // delay 100us
4.         if( C1 == T100US -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
5.         else begin C1 <= C1 + 1'b1; end
6.
7.         1: // Send Precharge Command
8.             begin rCMD <= _PR; { rBA, rA } <= 15'h3fff; i <= i + 1'b1; end
9.
10.        2: // wait TRP 20ns
11.            if( C1 == TRP -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
12.            else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
13.
14.        3: // Send Auto Refresh Command
15.            begin rCMD <= _AR; i <= i + 1'b1; end
16.
17.        4: // wait TRRC 63ns
18.            if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
19.            else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
20.
21.        5: // Send Auto Refresh Command
```

```

22.      begin rCMD <= _AR; i <= i + 1'b1; end
23.
24.      6: // wait TRRC 63ns
25.      if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
26.      else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
27.
28.      7: // Send LMR Cmd. Burst Read & Write, 3'b011 mean CAS latency = 3, Sequential, 1 burst length
29.      begin rCMD <= _LMR; rBA <= 2'b11; rA <= {3'd0,1'b0,2'd0,3'b011,1'b0, 3'b000}; i <= i + 1'b1; end
30.
31.      8: // Send 2 nop CLK for tMRD
32.      if( C1 == TMRD -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
33.      else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
34.
35.      9: // Generate done signal
36.      begin isDone <= 1'b1; i <= i + 1'b1; end
37.
38.      10:
39.      begin isDone <= 1'b0; i <= 4'd0; end
40.
41. endcase

```

代码 18.2

代码 18.2 完全按照图 18.4 去驱动，读者只要将 i 看为 T 就万事大吉，其中步骤 7 发送 LMR 命令还有设置 Code 内容。至于步骤 8~9 则用来产生完成信号。

刷新操作：

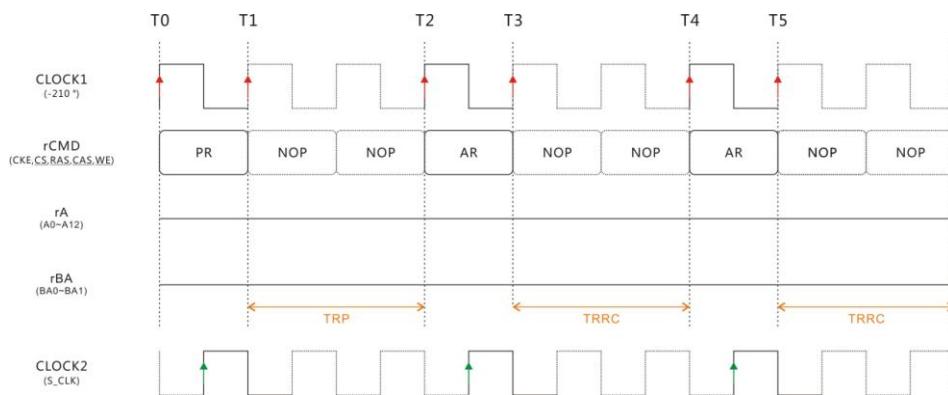


图 18.5 刷新操作的理想时序图。

所谓定期刷新就是被宫掉的初始化，如图 18.5 所示，时序过程如下：

- T0，发送 PR 命令（拉高所有 rA 与 rBA 视喜好而定）；
- T0 半周期，SDRAM 读取。

- T1 , 满足 TRP ;
- T2 , 发送 AR 命令。
- T2 半周期 , SDRAM 读取。
- T3 , 满足 TRRC ,
- T4 , 发送 AR 命令。
- T4 半周期 , SDRAM 读取。
- T5 , 满足 TRRC ,

Verilog 则可以这样表示 , 结果如表 18.3 所示 :

```

1. case( i )
2.
3.     0: // Send Precharge Command
4.         begin rCMD <= _PR; i <= i + 1'b1; end
5.
6.     1: // wait TRP 20ns
7.         if( C1 == TRP -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
8.         else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
9.
10.    2: // Send Auto Refresh Command
11.        begin rCMD <= _AR; i <= i + 1'b1; end
12.
13.    3: // wait TRRC 63ns
14.        if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
15.        else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
16.
17.    4: // Send Auto Refresh Command
18.        begin rCMD <= _AR; i <= i + 1'b1; end
19.
20.    5: // wait TRRC 63ns
21.        if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
22.        else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
23.
24.    6: // Generate done signal
25.        begin isDone <= 1'b1; i <= i + 1'b1; end
26.
27.    7:
28.        begin isDone <= 1'b0; i <= 4'd0; end
29.
30. endcase

```

代码 18.3

除了步骤 6~7 用来产生完成信号以外 , 代码 18.3 都是据图 18.5 描述。SDRAM 储存的

内容是非常脆弱的，如果我们不定期刷新内容，该内容有可能会蒸发掉。根据 HY57V2562GTR 这只 SDRAM，它的内容储存周期为 $8192 / 64\text{ms}$ ，然而定期刷新的计算如下：

$$64\text{ms} / 8192 = 7.8125\mu\text{s}$$

换言之，每隔 7.8125 微妙就要刷新一次所有内容。

写操作：

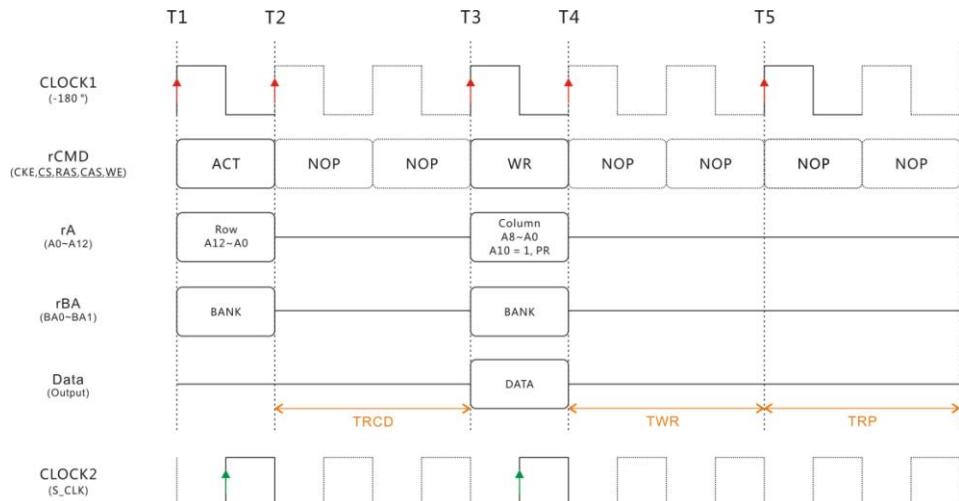


图 18.6 写操作的理想时序图。

图 18.6 是写操作的理想时序图，过程如下：

- T1，发送 ACT 命令，BANK 地址与行地址；
- T1 半周期，SDRAM 读取；
- T2，满足 TRCD；
- T3，发送 WR 命令，BANK 地址与列地址，还有写数据；
- T3 半周期，SDRAM 读取
- T4，满足 TWR；
- T5，满足 TRP。

正如前面说过，ACT 命令式用来选择 BANK 地址与行地址，然而关键就在 T3。T3 除了发送 WR 命令，列地址，还有些数据以外，A10 拉高是为了执行预充电。所谓预充电就是释放 BANK 地址，行地址与列地址等的选择。因此，满足 TWR 以后，我们还要满足 TRP 的释放时间，好让 SDRAM 有足够的时间自行释放选择。

Verilog 则可以这样描述，结果如代码 18.4 所示：

```
1. case( i )
2.
3.     0: // Set IO to output State
```

```

4.      begin isOut <= 1'b1; i <= i + 1'b1; end
5.
6.      1: // Send Active Command with Bank and Row address
7.      begin rCMD <= _ACT; rBA <= iAddr[23:22]; rA <= iAddr[21:9]; i <= i + 1'b1; end
8.
9.      2: // wait TRCD 20ns
10.     if( C1 == TRCD -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
11.     else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
12.
13.     3: // Send Write cmd with row address, pull up A10 1 clk to PR
14.     begin rCMD <= _WR; rBA <= iAddr[23:22]; rA <= { 4'b0010, iAddr[8:0] }; i <= i + 1'b1; end
15.
16.     4: // wait TWR 2 clock
17.     if( C1 == TWR -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
18.     else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
19.
20.     5: // wait TRP 20ns
21.     if( C1 == TRP -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
22.     else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
23.
24.     6: // Generate done signal
25.     begin isDone <= 1'b1; i <= i + 1'b1; end
26.
27.     7:
28.     begin isDone <= 1'b0; i <= 4'd0; end
29.
30.   endcase

```

代码 18.4

根据前面的计算，BA1~BA0 再加上 RA12~A0 与 CA8~A0 以后，一共有 24 位宽，详细的位分配如表 18.4 所示：

表 18.4 Addr 的位分配。

位分配	地址内容
Addr[23:22]	BANK 地址
Addr[21:9]	行地址
Addr[8:0]	列地址

如代码 18.4 所示，步骤用来设置 IO 口为输出。步骤 1 为 rA 赋值行地址，步骤 3 则为 rA 赋值列地址，并且拉高 A10 以示自行预充电。步骤 6~7 用来产生完成信号。

读操作：

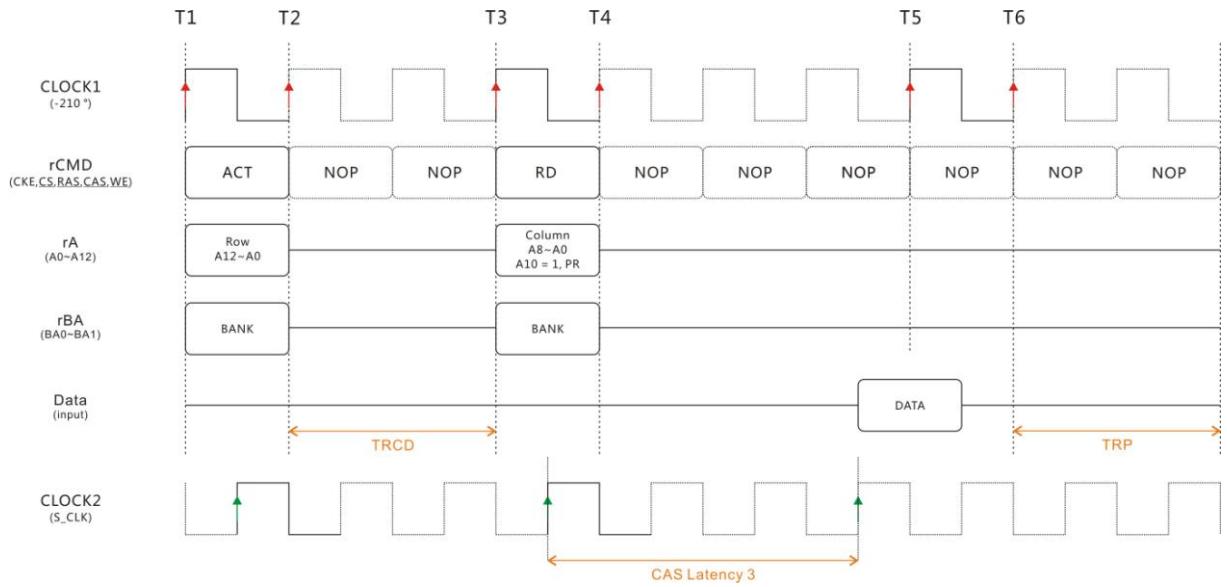


图 18.7 读操作的理想时序。

图 18.7 为读操作的理想时序，大致过程如下：

- T1，发送 ACT 命令，BANK 地址与行地址；
- T1 半周期，SDRAM 读取；
- T2，满足 TRCD；
- T3，发送 RD 命令，BANK 地址与列地址；
- T3 半周期，SDRAM 读取命令。
- T4，满足 CAS Latency。
- T5，读取数据。
- T6，满足 TRP。

读操作与写操作的过程大同小异，除了 WR 命令变成 RD 命令以外，A10 为 1 同样表示自行预充电，余下就是满足 CAS Latency。好奇的同学一定会觉得疑惑，为何 CL 为 3 呢？其实没什么，只是直感上觉得 3 这个数字比较顺眼一点。注意 CL 的计算方式是读取 RD 命令以后开始计算。

Verilog 可以这样描述，结果如代码 18.5 所示：

```

1. case( i )
2.
3.     0:
4.         begin isOut <= 1'b0; D1 <= 16'd0; i <= i + 1'b1; end
5.
6.     1: // Send Active command with Bank and Row address
7.         begin rCMD <= _ACT; rBA <= iAddr[23:22]; rA <= iAddr[21:9]; i <= i + 1'b1; end
8.
9.     2: // wait TRCD 20ns

```

```

10.          if( C1 == TRCD -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
11.          else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
12.
13.          3: // Send Read command and column address, pull up A10 to PR.
14.          begin rCMD <= _RD; rBA <= iAddr[23:22]; rA <= { 4'b0010, iAddr[8:0]}; i <= i + 1'b1; end
15.
16.          4: // wait CL 3 clock
17.          if( C1 == CL -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
18.          else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
19.
20.          5: // Read Data
21.          begin D1 <= S_DQ; i <= i + 1'b1; end
22.
23.          6: // wait TRP 20ns
24.          if( C1 == TRP -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
25.          else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
26.
27.          7: // Generate done signal
28.          begin isDone <= 1'b1; i <= i + 1'b1; end
29.
30.          8:
31.          begin isDone <= 1'b0; i <= 4'd0; end
32.
33.      endcase

```

代码 18.5

代码 18.5 完全根据图 18.7 描述，除了步骤 7~8 用于产生完成信号以外。SDRAM 的基本操作大致上就是这样而已，完后我们便可以开始建模了。

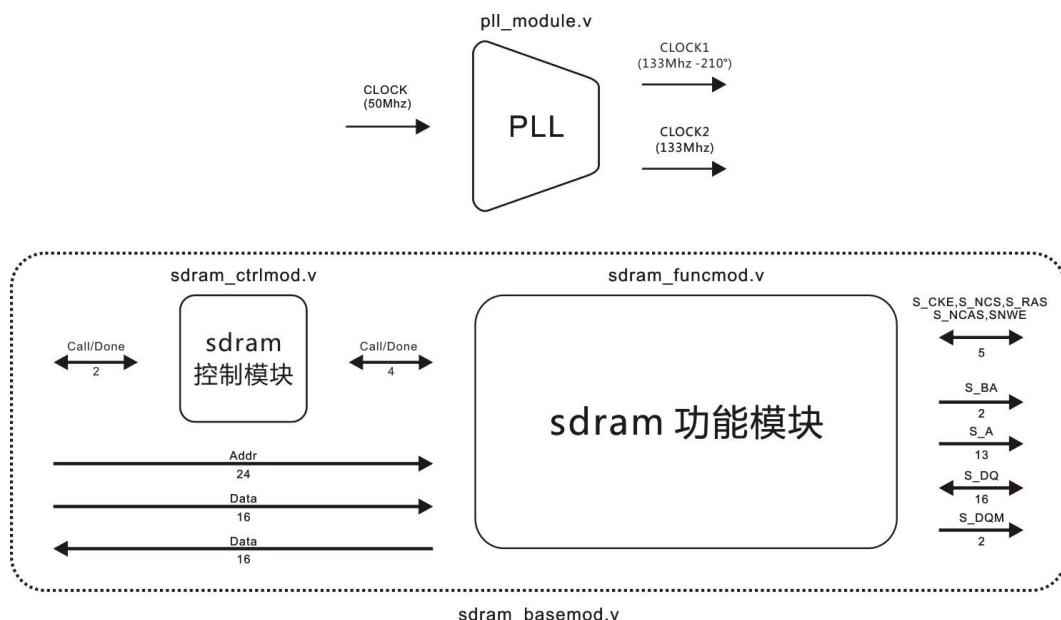


图 18.8 SDRAM 基础模块的建模图。

图 18.8 是 SDRAM 基础模块的建模图，SDRAM 基础模块的内容包括 SDRAM 控制模块，还有 SDRAM 功能模块。外围的 PLL 模块应用频率为 133Mhz 向左位移 210° 的 CLOCK1，还有 133Mhz 的 CLOCK2。CLOCK1 用作系统时钟，CLOCK 用作 SDRAM 时钟。如果 PLL 模块硬要分类的话，它应该属于特殊性质的即时类吧！？

SDRAM 控制模块主要负责一些操作的调度，左边 2 位 Call/Done 由外部调用，其中 [1] 为写操作 [0] 为读操作；右边 4 位 Call/Done 为调用 SDRAM 功能模块，其中 [3] 为写操作 [2] 为读操作 [1] 为刷新 [0] 为初始化。SDRAM 功能模块的右边是驱动 SDRAM 硬件资源的顶层信号，左边的问答信号被控制模块调用以外，地址信号还有数据信号都直接连接外部。

sdram_funcmod.v



图 18.9 SDRAM 功能模块的建模图。

该说的东西笔者都已经说了，具体内容我们还是来看代码吧。

```
1. module sdram_funcmod
2. (
3.     input CLOCK,
4.     input RESET,
5.
6.     output S_CKE, S_NCS, S_NRAS, S_NCAS, S_NWE,
7.     output [1:0]S_BA,
8.     output [12:0]S_A,
9.     output [1:0]S_DQM,
10.    inout [15:0]S_DQ,
11.
12.    input [3:0]iCall,
13.    output oDone,
14.    input [23:0]iAddr, // [23:22]BA,[21:9]Row,[8:0]Column
15.    input [15:0]iData,
16.    output [15:0]oData
17. );
```

以上内容为相关的出入端声明。

```
18.     parameter T100US = 14'd13300;
19.     // tRP 20ns, tRRC 63ns, tRCD 20ns, tMRD 2CLK, tWR/tDPL 2CLK, CAS Latency 3CLK
20.     parameter  TRP = 14'd3, TRRC = 14'd9, TMRD = 14'd2, TRCD = 14'd3, TWR = 14'd2, CL = 14'd3;
21.     parameter  _INIT = 5'b01111, _NOP = 5'b10111, _ACT = 5'b10011, _RD = 5'b10101, _WR = 5'b10100,
22.                 _BSTP = 5'b10110, _PR = 5'b10010, _AR = 5'b10001, _LMR = 5'b10000;
23.
```

以上内容为相关的常量声明，其中第 18~20 行的是将常量都是经由 133Mhz 量化。

```
24.     reg [4:0]i;
25.     reg [13:0]C1;
26.     reg [15:0]D1;
27.     reg [4:0]rCMD;
28.     reg [1:0]rBA;
29.     reg [12:0]rA;
30.     reg [1:0]rDQM;
31.     reg isOut;
32.     reg isDone;
33.
34.     always @ ( posedge CLOCK or negedge RESET )
35.         if( !RESET )
36.             begin
37.                 i <= 4'd0;
38.                 C1 <= 14'd0;
39.                 D1 <= 16'd0;
40.                 rCMD <= _NOP;
41.                 rBA <= 2'b11;
42.                 rA <= 13'h1fff;
43.                 rDQM <= 2'b00;
44.                 isOut <= 1'b1;
45.                 isDone <= 1'b0;
46.             end
```

以上内容为相关的寄存器声明以及复位操作。

```
47.             else if( iCall[3] )
48.                 case( i )
49.
50.                     0: // Set IO to output State
51.                     begin isOut <= 1'b1; i <= i + 1'b1; end
```

```

52.
53.      1: // Send Active Command with Bank and Row address
54.      begin rCMD <= _ACT; rBA <= iAddr[23:22]; rA <= iAddr[21:9]; i <= i + 1'b1; end
55.
56.      2: // wait TRCD 20ns
57.      if( C1 == TRCD -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
58.      else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
59.
60.      /***** */
61.
62.      3: // Send Write command with row address, pull up A10 1 clk to PR
63.      begin rCMD <= _WR; rBA <= iAddr[23:22]; rA <= { 4'b0010, iAddr[8:0] }; i <= i + 1'b1; end
64.
65.      4: // wait TWR 2 clock
66.      if( C1 == TWR -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
67.      else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
68.
69.      5: // wait TRP 20ns
70.      if( C1 == TRP -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
71.      else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
72.
73.      /***** */
74.
75.      6: // Generate done signal
76.      begin isDone <= 1'b1; i <= i + 1'b1; end
77.
78.      7:
79.      begin isDone <= 1'b0; i <= 4'd0; end
80.
81.      endcase

```

以上内容为部分核心操作。第 47 行的 if(iCall[3]) 表示余下内容为写操作。

```

82.      else if( iCall[2] )
83.          case( i )
84.
85.              0:
86.                  begin isOut <= 1'b0; D1 <= 16'd0; i <= i + 1'b1; end
87.
88.              1: // Send Active command with Bank and Row address
89.                  begin rCMD <= _ACT; rBA <= iAddr[23:22]; rA <= iAddr[21:9]; i <= i + 1'b1; end
90.
91.              2: // wait TRCD 20ns

```

```

92.         if( C1 == TRCD -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
93.         else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
94.
95.         /*****
96.
97.         3: // Send Read command and column address, pull up A10 to PR
98.         begin rCMD <= _RD; rBA <= iAddr[23:22]; rA <= { 4'b0010, iAddr[8:0] }; i <= i + 1'b1; end
99.
100.        4: // wait CL 3 clock
101.        if( C1 == CL -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
102.        else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
103.
104.        *****/
105.
106.        5: // Read Data
107.        begin D1 <= S_DQ; i <= i + 1'b1; end
108.
109.        *****/
110.
111.        6: // wait TRP 20ns
112.        if( C1 == TRP -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
113.        else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
114.
115.        *****/
116.
117.        7: // Generate done signal
118.        begin isDone <= 1'b1; i <= i + 1'b1; end
119.
120.        8:
121.        begin isDone <= 1'b0; i <= 4'd0; end
122.
123.    endcase

```

以上内容为部分核心操作。第 82 行的 if(iCall[2]) 表示余下内容为读操作。

```

124.    else if( iCall[1] )
125.        case( i )
126.
127.            0: // Send Precharge Command
128.            begin rCMD <= _PR; i <= i + 1'b1; end
129.
130.            1: // wait TRP 20ns
131.            if( C1 == TRP -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end

```

```

132.         else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
133.
134.         2: // Send Auto Refresh Command
135.         begin rCMD <= _AR; i <= i + 1'b1; end
136.
137.         3: // wait TRRC 63ns
138.         if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
139.         else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
140.
141.         4: // Send Auto Refresh Command
142.         begin rCMD <= _AR; i <= i + 1'b1; end
143.
144.         5: // wait TRRC 63ns
145.         if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
146.         else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
147.
148.         *****/
149.
150.         6: // Generate done signal
151.         begin isDone <= 1'b1; i <= i + 1'b1; end
152.
153.         7:
154.         begin isDone <= 1'b0; i <= 4'd0; end
155.
156.     endcase

```

以上内容为部分核心操作。第 124 行的 if(iCall[1]) 表示余下内容为刷新操作。

```

157.     else if( iCall[0] )
158.     case( i )
159.
160.         0: // delay 100us
161.         if( C1 == T100US -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
162.         else begin C1 <= C1 + 1'b1; end
163.
164.         *****/
165.
166.         1: // Send Precharge Command
167.         begin rCMD <= _PR; { rBA, rA } <= 15'h3fff; i <= i + 1'b1; end
168.
169.         2: // wait TRP 20ns
170.         if( C1 == TRP -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
171.         else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end

```

```

172.
173.      3: // Send Auto Refresh Command
174.      begin rCMD <= _AR; i <= i + 1'b1; end
175.
176.      4: // wait TRRC 63ns
177.      if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
178.      else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
179.
180.      5: // Send Auto Refresh Command
181.      begin rCMD <= _AR; i <= i + 1'b1; end
182.
183.      6: // wait TRRC 63ns
184.      if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
185.      else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
186.
187.      *****/
188.
189.      7: // Send LMR Cmd. Burst Read & Write,  3'b010 mean CAS latency = 3, Sequential, 1 burst length
190.      begin rCMD <= _LMR; rBA <= 2'b11; rA <= { 3'd0, 1'b0, 2'd0, 3'b011, 1'b0, 3'b000 }; i <= i + 1'b1; end
191.
192.      8: // Send 2 nop CLK for tMRD
193.      if( C1 == TMRD -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
194.      else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
195.
196.      *****/
197.
198.      9: // Generate done signal
199.      begin isDone <= 1'b1; i <= i + 1'b1; end
200.
201.      10:
202.      begin isDone <= 1'b0; i <= 4'd0; end
203.
204.      endcase
205.

```

以上内容为部分核心操作。第 157 行的 if(iCall[0]) 表示余下内容为初始化。

```

206.      assign { S_CKE, S_NCS, S_NRAS, S_NCAS, S_NWE } = rCMD;
207.      assign { S_BA, S_A } = { rBA, rA };
208.      assign S_DQM = rDQM;
209.      assign S_DQ = isOut ? iData : 16'hzzzz;
210.      assign oDone = isDone;
211.      assign oData = D1;

```

```
212.  
213. endmodule
```

以上内容为相关的输出驱动声明，注意 iData 直接驱动 S_DQ。

sdram_ctrlmod.v



图 18.10 SDRAM 控制模块的建模图。

前面说过该模块负责一些功能调用，此外该模块也负责定时刷新的计算，具体内容我们还是来看代码吧。

```
1. module sdram_ctrlmod
2. (
3.     input CLOCK,
4.     input RESET,
5.     input [1:0]iCall, // [1]Write, [0]Read
6.     output [1:0]oDone,
7.     output [3:0]oCall,
8.     input iDone
9. );
10.    parameter WRITE = 4'd1, READ = 4'd4, REFRESH = 4'd7, INITIAL = 4'd8;
11.    parameter TREF = 11'd1040;
12.
```

以上内容为相关的出入端声明。第 10 行是各个入口地址的常量声明，第 11 行则是定时刷新的周期——7.8125us。

```
13.    reg [3:0]i;
14.    reg [10:0]C1;
15.    reg [3:0]isCall; // [3]Write [2]Read [1]A.Refresh [0]Initial
16.    reg [1:0]isDone;
17.
18.    always @ ( posedge CLOCK or negedge RESET )
19.        if( !RESET )
20.            begin
21.                i <= INITIAL;           // Initial SDRam at first
```

```
22.          C1 <= 11'd0;
23.          isCall <= 4'b0000;
24.          isDone <= 2'b00;
25.      end
```

以上内容为相关的寄存器声明以及复位操作。第 21 行表示 i 首先会指向初始化。

```
26.      else
27.          case( i )
28.
29.              0: // IDLE
30.                  if( C1 >= TREF ) begin C1 <= 11'd0;  i <= REFRESH; end
31.                  else if( iCall[1] ) begin C1 <= C1 + 1'b1; i <= WRITE; end
32.                  else if( iCall[0] ) begin C1 <= C1 + 1'b1; i <= READ; end
33.                  else begin C1 <= C1 + 1'b1; end
34.
35.          *****/
36.
```

以上内容为部分核心操作。步骤 0 为待机状态，期间第 33 行的 C1 会一直递增，如果期间没有任何读写操作，而且 C1 的计数内容也超过 TREF，那么 C1 会清零，i 指向 REFRESH（第 30 行）。反之，如果读写操作被使能，i 指向相关的步骤入口，期间 C1 也会递增以示步骤翻转所用掉的时钟。

```
37.      1: //Write
38.          if( iDone ) begin isCall[3] <= 1'b0; C1 <= C1 + 1'b1; i <= i + 1'b1; end
39.          else begin isCall[3] <= 1'b1; C1 <= C1 + 1'b1; end
40.
41.      2:
42.          begin isDone[1] <= 1'b1; C1 <= C1 + 1'b1; i <= i + 1'b1; end
43.
44.      3:
45.          begin isDone[1] <= 1'b0; C1 <= C1 + 1'b1; i <= 4'd0; end
46.
47.      *****/
48.
```

以上内容为部分核心操作。步骤 1~3 是写操作。步骤 1 表示，功能模块反馈完成信号之前，C1 会不停递增。当完成信号接收到手，isCall[3]拉低，C1 递增，i 也递增。步骤 2~3 则是用来反馈写操作的完成信号，期间 C1 也会递增。

```
49.      4: // Read
50.          if( iDone ) begin isCall[2] <= 1'b0; C1 <= C1 + 1'b1; i <= i + 1'b1; end
```

```
51.           else begin isCall[2] <= 1'b1; C1 <= C1 + 1'b1; end
52.
53.           5:
54.           begin isDone[0] <= 1'b1; C1 <= C1 + 1'b1; i <= i + 1'b1; end
55.
56.           6:
57.           begin isDone[0] <= 1'b0; C1 <= C1 + 1'b1; i <= 4'd0; end
58.
59.           *****/
60.
```

以上内容为部分核心操作。步骤 4~6 是读操作。步骤 4 表示接收完成信号之前，isCall[2]会不停拉高，C1 也会不停递增 ... 直至接收完成信号，isCall[2]才会拉低，然而 C1 也会递增。步骤 5~6 用反馈读操作的完成信号。

```
61.           7: // Auto Refresh
62.           if( iDone ) begin isCall[1] <= 1'b0; i <= 4'd0; end
63.           else begin isCall[1] <= 1'b1; end
64.
65.           *****/
66.
```

以上内容为部分核心操作。步骤 7 是刷新操作，接收完成信号之前 isCall[1] 会不停拉高，直至接收完成信号为止，isCall[1]才会拉低，然后 i 指向步骤 0。

```
67.           8: // Initial
68.           if( iDone ) begin isCall[0] <= 1'b0; i <= 4'd0; end
69.           else begin isCall[0] <= 1'b1; end
70.
71.           endcase
72.
73.           assign oDone = isDone;
74.           assign oCall = isCall;
75.
76.   endmodule
```

以上内容为部分核心操作。步骤 8 用来执行初始化，接收完成信号之前，isCall[0]会不停拉高，直至接收完成信号为止，isCall[0]才会拉低，然后 i 指向步骤 0。第 73~74 行则是相关的输出驱动。整体而言，除了读写操作必须反馈完成信号给上层以外，其余的定期刷新还有初始化都是该内部操作，所以不用反馈完成信号。

sram_basemod.v

内容的连线部署完全依照图 18.8。

```
1. module sram_basemod
2. (
3.     input CLOCK,
4.     input RESET,
5.
6.     output S_CKE, S_NCS, S_NRAS, S_NCAS, S_NWE,
7.     output [1:0]S_BA,
8.     output [12:0]S_A,
9.     output [1:0]S_DQM,
10.    inout [15:0]S_DQ,
11.
12.    input [1:0]iCall,
13.    output [1:0]oDone,
14.    input [23:0]iAddr,
15.    input [15:0]iData,
16.    output [15:0]oData
17. );
```

以上内容为相关的出入端声明，第 5~10 行是顶层信号，第 12~16 行是模块左右两边的信号。

```
18.     wire [3:0]CallU1; // [3]Refresh, [2]Read, [1]Write, [0]Initial
19.
20.     sram_ctrlmod U1
21.     (
22.         .CLOCK( CLOCK ),
23.         .RESET( RESET ),
24.         .iCall( iCall ),      // < top ,[1]Write [0]Read
25.         .oDone( oDone ),      // > top ,[1]Write [0]Read
26.         .oCall( CallU1 ),     // > U2
27.         .iDone( DoneU2 )     // < U2
28.     );
29.
```

以上内容为控制模块的实例化。

```
30.     wire DoneU2;
31.
32.     sram_funcmod U2
```

```

33.      (
34.          .CLOCK( CLOCK ),
35.          .RESET( RESET ),
36.          .S_CKE( S_CKE ),      // > top
37.          .S_NCS( S_NCS ),      // > top
38.          .S_NRAS( S_NRAS ),    // > top
39.          .S_NCAS( S_NCAS ),    // > top
40.          .S_NWE( S_NWE ),     // > top
41.          .S_BA( S_BA ),       // > top
42.          .S_A( S_A ),         // > top
43.          .S_DQM( S_DQM ),     // > top
44.          .S_DQ( S_DQ ),       // <> top
45.          .iCall( CallU1 ),    // < U1
46.          .oDone( DoneU2 ),    // > U1
47.          .iAddr( iAddr ),     // < top
48.          .iData( iData ),      // < top
49.          .oData( oData )       // > top
50.      );
51.
52. endmodule

```

以上内容为功能模块的实例化。

sdram_demo.v

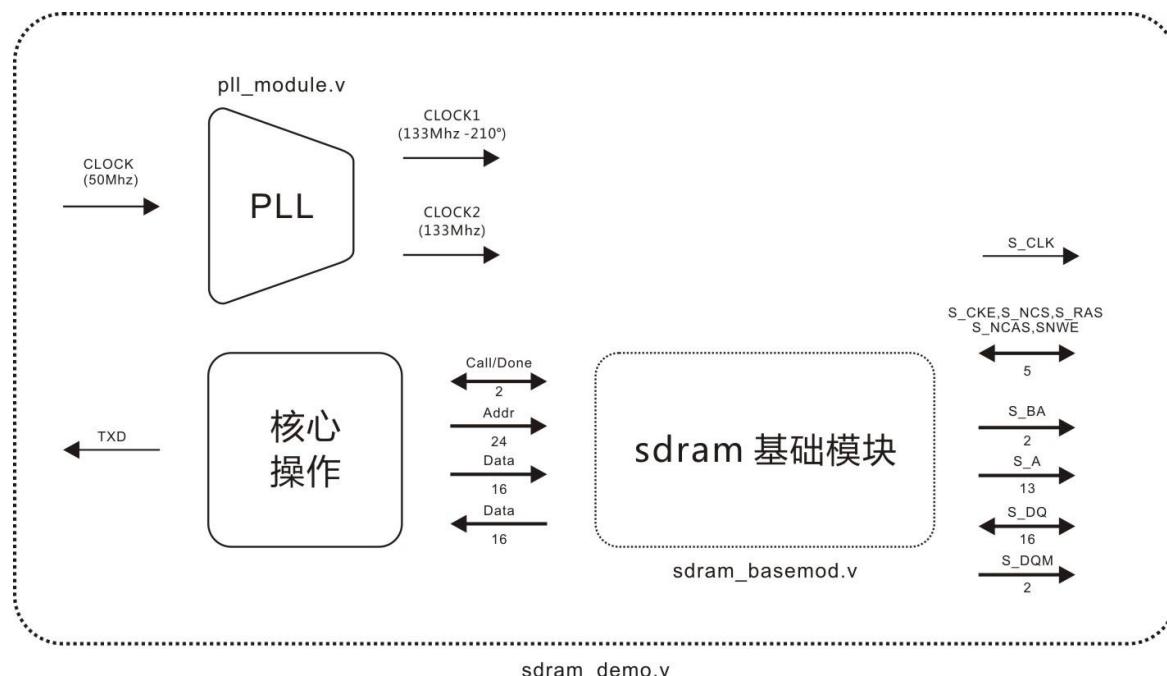


图 18.11 实验十八的建模图。

图 18.11 是实验十八的建模图 ,其中 sram_demo 包含 PLL 模块 ,核心操作还有 SDRAM 基础模块。PLL 模块将 50Mhz 的时钟倍频为 133Mhz 而且左移 210° 的 CLOCK1 ,还有 133Mhz 的 CLOCK2 ,它直接驱动 S_CLK 顶层信号。核心操作负责调用 SDRAM 基础模块 ,并且将读写内容经由 TXD 发送出去。SDRAM 基础模块左边的问答信号只有两位 ,其中[1]为写 [0]为读 ,具体内容我们还是来看代码吧。

```
1. module sram_demo
2. (
3.     input CLOCK,
4.     input RESET,
5.     output S_CLK,
6.     output S_CKE, S_NCS, S_NRAS, S_NCAS, S_NWE,
7.     output [12:0]S_A,
8.     output [1:0]S_BA,
9.     output [1:0]S_DQM,
10.    inout [15:0]S_DQ,
11.    output TXD
12. );
```

以上内容为相关的出入端声明。

```
13.     wire CLOCK1,CLOCK2;
14.
15.     pll_module U1
16.     (
17.         .inclk0 ( CLOCK ), // 50Mhz
18.         .c0 ( CLOCK1 ), // 133Mhz -210 degree phase
19.         .c1 ( CLOCK2 ) // 133Mhz
20.     );
21.
```

以上内容为 PLL 模块的实例化 , CLOCK1 为 133Mhz 频率并且左移 210° , CLOCK2 为 133Mhz 频率 , 并且直接驱动 S_CLK 。

```
22.     wire [1:0]DoneU2;
23.     wire [15:0]DataU2;
24.
25.     sram_basemod U2
26.     (
27.         .CLOCK( CLOCK1 ),
28.         .RESET( RESET ),
29.         .S_CKE( S_CKE ),
```

```

30.      .S_NCS( S_NCS ),
31.      .S_NRAS( S_NRAS ),
32.      .S_NCAS( S_NCAS ),
33.      .S_NWE( S_NWE ),
34.      .S_A( S_A ),
35.      .S_BA( S_BA ),
36.      .S_DQM( S_DQM ),
37.      .S_DQ( S_DQ ),
38.      .iCall( isCall ),
39.      .oDone( DoneU2 ),
40.      .iAddr( D1 ),
41.      .iData( D2 ),
42.      .oData( DataU2 )
43. );
44.

```

以上内容为 SDRAM 基础模块的实例化，第 40~41 行表示 iAddr 为 D1 驱动，iData 为 D2 驱动。

```

45. parameter B115K2 = 11'd1157, TXFUNC = 6'd16;
46.
47. reg [5:0]i,Go;
48. reg [10:0]C1;
49. reg [23:0]D1;
50. reg [15:0]D2,D3;
51. reg [10:0]T;
52. reg [1:0]isCall;
53. reg rTXD;
54.
55. always @ ( posedge CLOCK1 or negedge RESET )
56.     if( !RESET )
57.         begin
58.             i <= 6'd0;
59.             Go <= 6'd0;
60.             C1 <= 11'd0;
61.             D1 <= 24'd0;
62.             D2 <= 16'd0;
63.             D3 <= 16'd0;
64.             T <= 11'd0;
65.             isCall <= 2'b00;
66.             rTXD <= 1'b1;
67.         end

```

以上内容为相关的寄存器以及复位操作。第 45 行是波特率为 115200 还有伪函数入口的常量声明。

```
68.         else
69.             case( i )
70.
71.             0:
72.                 if( DoneU2[1] ) begin isCall[1] <= 1'b0; i <= i + 1'b1; end
73.                 else begin isCall[1] <= 1'b1; D1 <= 24'd0; D2 <= 16'hABCD; end
74.
75.             1:
76.                 if( DoneU2[0] ) begin D3 <= DataU2; isCall[0] <= 1'b0; i <= i + 1'b1; end
77.                 else begin D1 <= 24'd0; isCall[0] <= 1'b1; end
78.
79.             2:
80.                 begin T <= { 2'b11, D3[15:8], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
81.
82.             3:
83.                 begin T <= { 2'b11, D3[7:0], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
84.
85.             4:
86.                 i <= i;
87.
88.             //*****
89.
```

以上内容为部分核心操作。步骤 0 将数据 16'hABCD 写入地址 0。步骤 1 从地址 0 读出数据 16'hABCD，并且暂存至 D3。步骤 2 先发送 D3 的高 8 位，步骤 3 则发送 D3 的低 8 位。步骤 4 发呆。

```
90.             16,17,18,19,20,21,22,23,24,25,26:
91.             if( C1 == B115K2 -1 ) begin C1 <= 11'd0; i <= i + 1'b1; end
92.             else begin rTXD <= T[i - 16]; C1 <= C1 + 1'b1; end
93.
94.             27:
95.                 i <= Go;
96.
97.             endcase
98.
99.             assign S_CLK = CLOCK2;
100.            assign TXD = rTXD;
101.
102. endmodule
```

以上内容为部分核心操作。步骤 16~27 是发送一帧数据的伪函数。第 99~100 行则是相关的输出驱动。综合完毕并且下载程序，如果串口调试软件出现 ABCD 等两字节数据，结果表示实验成功。

细节一：完整的个体模块

SDRAM 基础模块已经就绪完毕。

细节二：其它时序参数

驱动 SDRAM 最大的收获莫过于学习各种稀奇古怪的时序参数，虽然实验十六的 IIC，也有时序参数，但是前者好比一粒面包屑，后者则是一片面包，两种时序参数有“体积”上的明确差距。笔者曾经说过，时序参数即时间要求有第一层与第二层之分，第一层时间要求正如 IIC 的时序参数，打得像面包一样 … 反之，第二层时间要求宛如 SDRAM 的时序参数，小得似面包屑一般。

SDRAM 的时序参数除了 tRP , TRRC , TMRD , CAS Latency 等这些东西以外，它还有更为极为，而且不能控制的时序参数。更确切来说，这些时序参数都属于物理因数的范围 … 难得有机会学习 SDRAM，笔者就稍微聊聊它们吧。

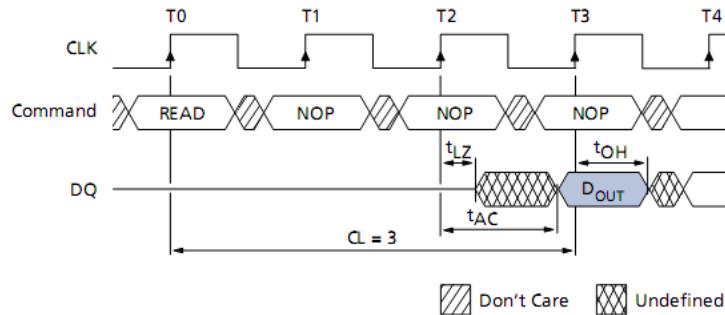


图 18.12 时序参数①。

图 18.12 是读操作的部分时序，当 CL 得到满足以后，数据就会被吐出来，其中：

TLZ (TLOZ) 为 clock to data output in low-Z time。简单来说，就是数据被出发沿吐出之前，必须经过的延迟时间。根据手册，133Mhz 为 1ns。

TAC 为 access time from clock。简单来说就是有效时间。根据手册，133Mhz 为 5.4ns

TOH 为 data out hold time。简单来说就是常见的 THOLD。根据手册，133Mhz 为 2.5ns

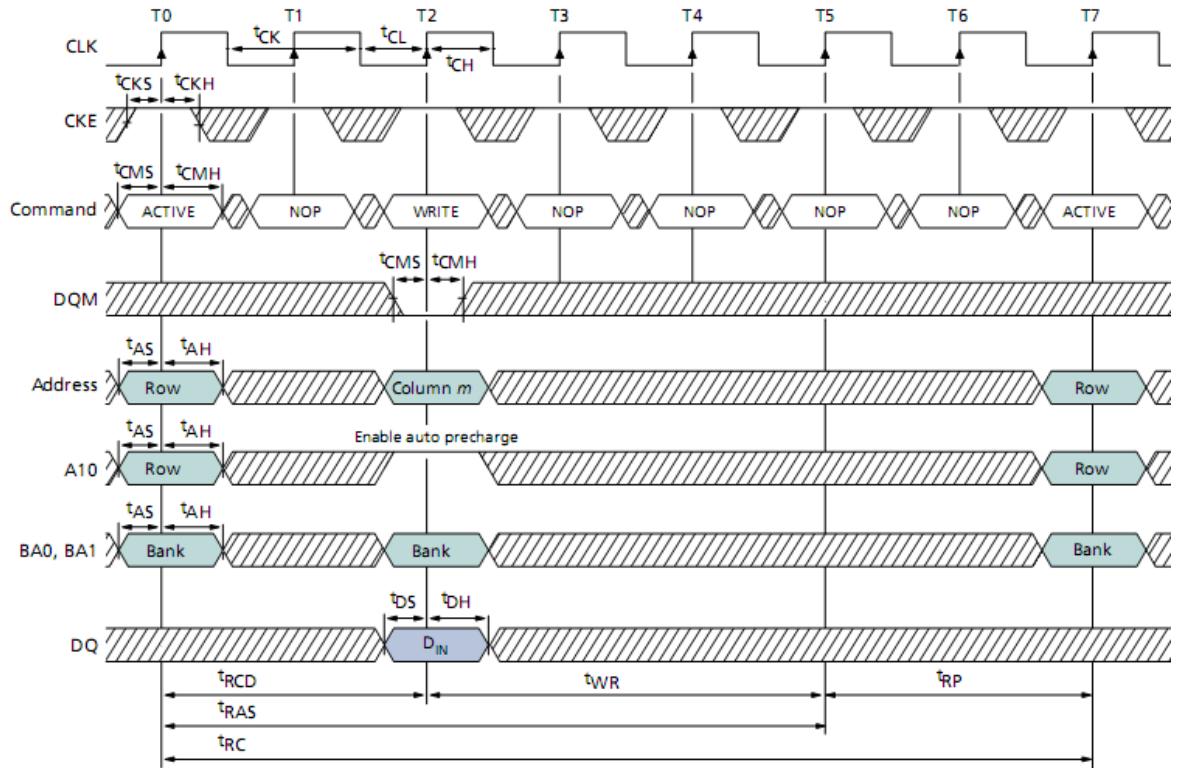


图 18.13 时序参数②。

图 18.13 是写操作的部分时序图，然而重点家伙就是当中 $T \times \times S$ 或者 $T \times \times H$ 。一般 $\times \times$ 是指数据的属性或者类别，不过 S 与 H 都有相同的意义，就是典型的 TSETUP 还有 THOLD。笔者习惯称呼它们为寄存器特性，因为只要任何一方得不到满足，数据读入寄存器就得不到保证。寄存器特性好比哥布林一样，数量常常多到令人喷饭，如果一一分析会耗死爷爷不偿命。

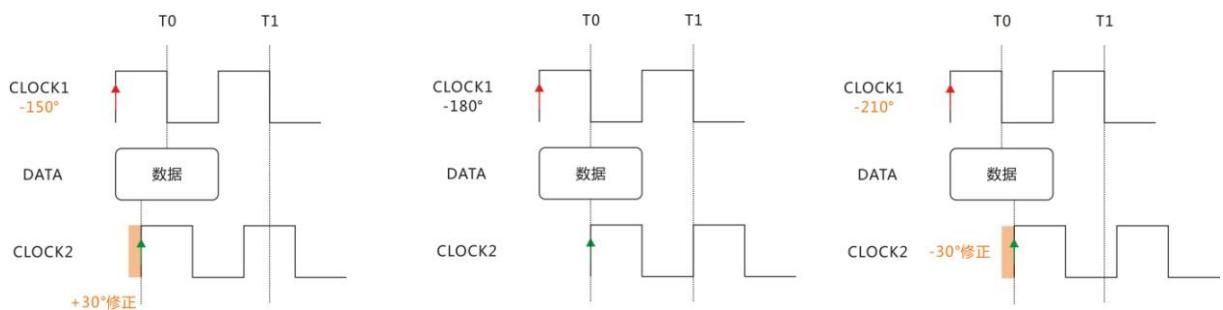


图 18.14 对外的理想时序。

为了用足一支竹竿扫尽一切，笔者才故意向将 CLOCK1 左移 180° 测试手气，看看 SDRAM 能不能读出正确的结果，如果不是再追加位移或者减少位移以致修正，结果如图 18.14 所示。一般而言， $T \times \times S$ 或者 $T \times \times H$ 这些家伙都会得到满足，然后乖乖就范。话虽如此，同学们还须注意，Verilog 充其量只能满足第二层的时间要求，却不能涉及（解决）其中，我们往往只能依赖运气与直觉。当然，我们可以借助静态时序分析的力量去搞定一切，有兴趣的朋友请看《工具篇 I》。

实验十九：SDRAM 模块② — 多字读写

表示 19.1 Mode Register 的内容。

Mode Register												
A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	0	OP Code		0	0	CAS Latency			BT	Burst Length		

A3	Burst Type			Burst Length		
0	Sequential			A3 = 0		
1	Interleave			A3 = 1		
0	0			1		
0	0			2		
0	1			4		
0	1			8		
1	1			Full Page	Reserved	

A9	Write Mode			CAS Latency		
0	Burst Read and Burst Write			2		
1	Burst Read and Single Write			3		

实验十八我们实现单字读写，实验十九则要实现多字读写。表 19.1 告诉我们，A2~A0 控制 Burst Length 的长度，为了实现长度为 4 的字读写，A2~A0 的内容设置为 3'b010。

```
7: // Send LMR Cmd. Burst Read & Write, 3'b011 mean CAS latency = 3, Sequential 4 burst length
begin rCMD <= _LMR; rBA <= 2'b11; rA <= { 3'd0, 1'b0, 2'd0, 3'b011, 1'b0, 3'b010 }; i <= i + 1'b1; end
```

代码 19.2

如代码基本上，初始化的大致过程与实验十八没有什么两样，仅有更改 Mode Register 的内容，结果如代码 19.2 所示。对此，写操作还有读操作因为读写字节改变，所以时序也稍微改变了一下。

多字写操作：

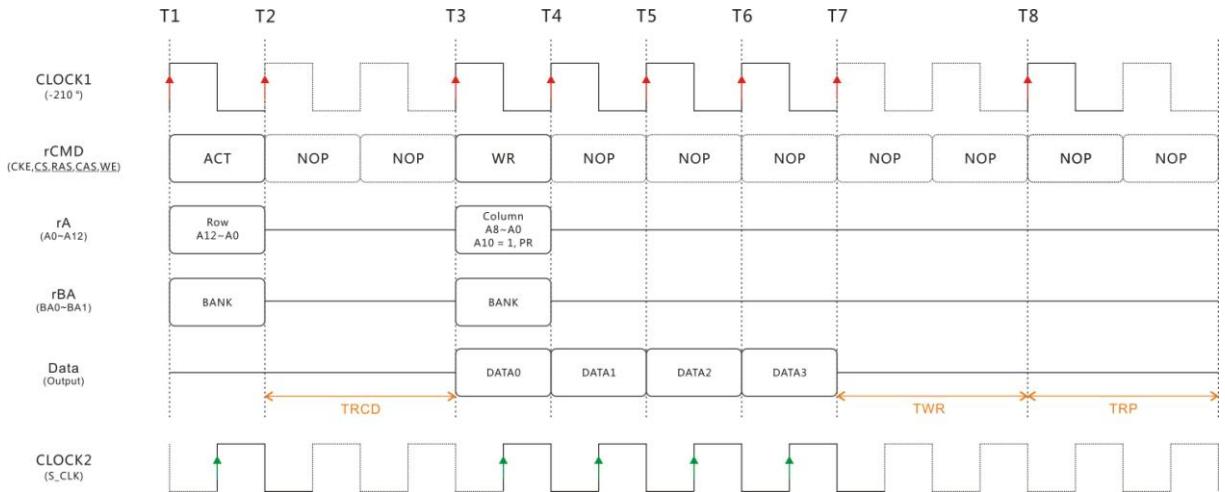


图 19.1 多字写操作的理想时序图。

图 19.1 是多字写操作的理想时序图。T1~T3 基本上没有什么改变，反之接续的 T4~T6 会依据 Burst Length 设置的长度而有所改变。由于 Burst Length 设置为 4，结果 T3 写第一字数据，T4 写第二字数据，T5 写第三字数据，T6 则写第四字，然而大致的过程如下所示：

- T1，发送 ACT 命令，BANK 地址与行地址；
- T1 半周期，SDRAM 读取；
- T2，满足 TRCD；
- T3，发送 WR 命令，BANK 地址与列地址，还有写第一字数据；
- T3 半周期，SDRAM 读取；
- T4，写第二字数据；
- T4 半周期，SDRAM 读取；
- T5，写第三字数据；
- T5 半周期，SDRAM 读取；
- T6，写第四字数据；
- T6 半周期，SDRAM 读取；
- T7，满足 TWR；
- T8，满足 TRP。

Verilog 则可以这样描述，结果如代码 19.2 所示：

```

1.      1: // Send Active Command with Bank and Row address
2.      begin rCMD <= _ACT; rBA <= iAddr[23:22]; rA <= iAddr[21:9]; i <= i + 1'b1; end
3.
4.      2: // wait TRCD 20ns
5.      if( C1 == TRCD -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
6.      else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
7.
8.      3: // Send Write command with row address, pull up A10 to PR
9.      begin rCMD <= _WR; rBA <= iAddr[23:22]; rA <= {4'b0010,iAddr[8:0]}; D1 <= iData[63:48]; i <= i + 1'b1;

```

```

    end
10.
11.    4:
12.        begin rCMD <= _NOP; D1 <= iData[47:32]; i <= i + 1'b1; end
13.
14.    5:
15.        begin rCMD <= _NOP; D1 <= iData[31:16]; i <= i + 1'b1; end
16.
17.    6:
18.        begin rCMD <= _NOP; D1 <= iData[15:0]; i <= i + 1'b1; end
19.
20.    7: // wait TWR 2 clock
21.        if( C1 == TWR -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
22.        else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
23.
24.    8: // wait TRP 20ns
25.        if( C1 == TRP -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
26.        else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end

```

代码 19.2

假设多字读写由高至低，那么步骤 3 写入 iData[63:48]，步骤 4 写入 iData[47:32]，步骤 5 写入 iData[31:16]，步骤 6 写 iData[15:0]。

多字读操作：

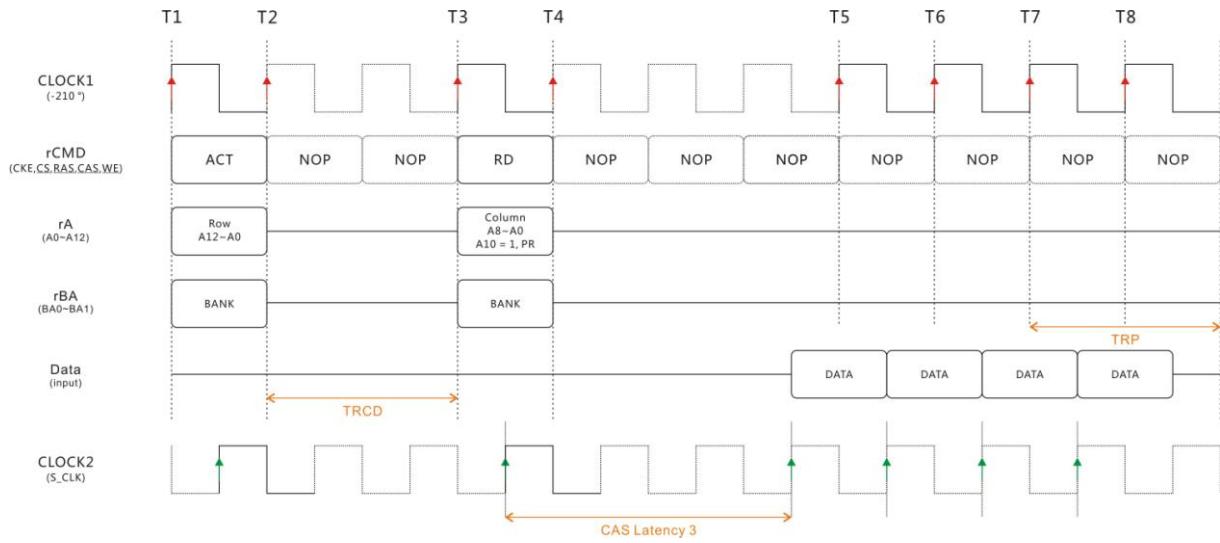


图 19.2 多字读操作的理想时序图。

图 19.2 是多字读操作的理想时序图，大致过程如下：

- T1，发送 ACT 命令，BANK 地址与行地址；
- T1 半周期，SDRAM 读取；

- T2 , 满足 TRCD ;
- T3 , 发送 RD 命令 , BANK 地址与列地址 ;
- T3 半周期 , SDRAM 读取命令 ;
- T4 , 满足 CAS Latency ;
- T5 , 读取第一字数据 ;
- T6 , 读取第二字数据 ;
- T7 , 读取第三字数据 ;
- T8 , 读取第四字数据。

多字读操作相较单字读操作稍微有一些不同 , 不同的地方除了读取数据的字变长以外 , 还有 TRP 满足在最后两字之中。至于 Verilog 则可以这样描述 , 结果如代码 19.3 所示 :

```

1.      1: // Send Active command with Bank and Row address
2.      begin rCMD <= _ACT; rBA <= iAddr[23:22]; rA <= iAddr[21:9]; i <= i + 1'b1; end
3.
4.      2: // wait TRCD 20ns
5.      if( C1 == TRCD -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
6.      else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
7.
8.      3: // Send Read command and column address, pull up A10 to PR
9.      begin rCMD <= _RD; rBA <= iAddr[23:22]; rA <= { 4'b0010, iAddr[8:0]}; i <= i + 1'b1; end
10.
11.     4: // wait CL 3 clock
12.     if( C1 == CL -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
13.     else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
14.
15.     5: // Read Data
16.     begin T[63:48] <= S_DQ; i <= i + 1'b1; end
17.
18.     6: // Read Data
19.     begin T[47:32] <= S_DQ; i <= i + 1'b1; end
20.
21.     7: // Read Data
22.     begin T[31:16] <= S_DQ; i <= i + 1'b1; end
23.
24.     8: // Read Data
25.     begin T[15:0] <= S_DQ; i <= i + 1'b1; end

```

代码 19.3

代码 19.3 也没有什么好解释的 , 基本上完全根据图 19.2 描述。理解完毕以后 , 我们就可以开始建模了。

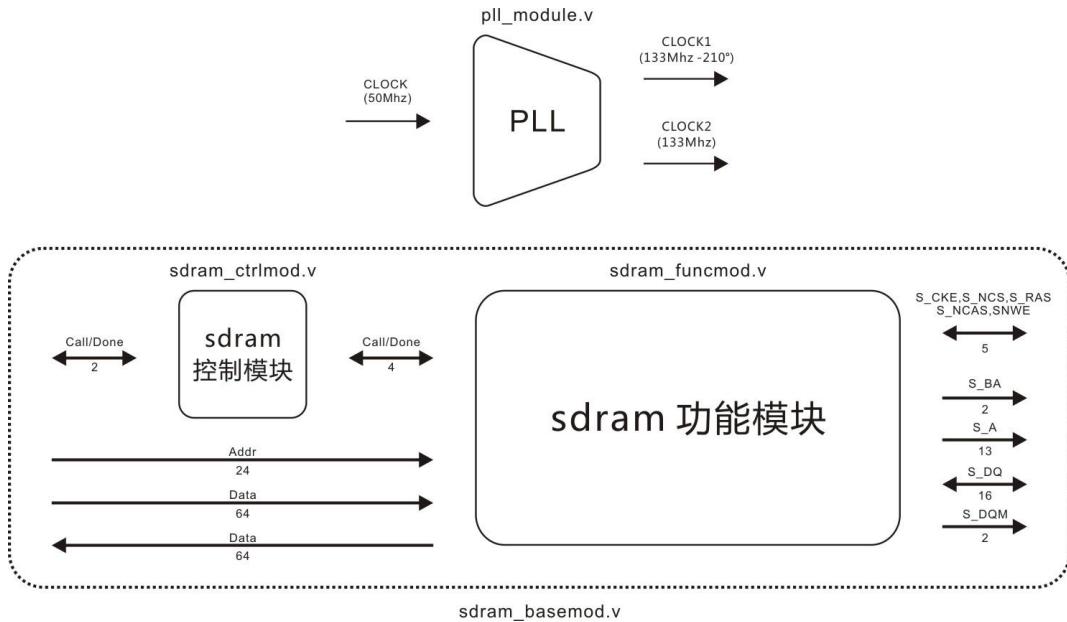


图 19.3 SDRAM 基础模块的建模图。

图 19.3 是 SDRAM 基础模块的建模图，这家伙比较实验十八，最大的区别就是 iData 与 oData 的位宽增大而已。

sdram_funcmod.v



图 19.4 SDRAM 功能模块的建模图。

图 19.4 是 SDRAM 功能模块的建模图，具体内容我们还是来看代码吧。

```

1. module sdram_funcmod
2. (
3.     input CLOCK,
4.     input RESET,
5.
6.     output S_CKE, S_NCS, S_NRAS, S_NCAS, S_NWE,
7.     output [1:0]S_BA, //2
8.     output [12:0]S_A, //12, CA0~CA8, RA0~RA12, BA0~BA1, 9+13+2 = 24;
9.     output [1:0]S_DQM,
```

```

10.      inout [15:0]S_DQ,
11.
12.      input [3:0]iCall,
13.      output oDone,
14.      input [23:0]iAddr, // [23:22]BA,[21:9]Row,[8:0]Column
15.      input [63:0]iData,
16.      output [63:0]oData
17.  );

```

以上内容为相关的出入端声明，注意第 15~16 行的位宽增大至 64 位。

```

18.      parameter T100US = 14'd13300;
19.      // tRP 20ns, tRRC 63ns, tRCD 20ns, tMRD 2CLK, tWR/tDPL 2CLK, CAS Latency 3CLK
20.      parameter TRP = 14'd3, TRRC = 14'd9, TMRD = 14'd2, TRCD = 14'd3, TWR = 14'd2, CL = 14'd3;
21.      parameter _INIT = 5'b01111, _NOP = 5'b10111, _ACT = 5'b10011, _RD = 5'b10101, _WR = 5'b10100,
22.          _BSTP = 5'b10110, _PR = 5'b10010, _AR = 5'b10001, _LMR = 5'b10000;
23.

```

以上内容为相关的常量声明。

```

24.      reg [4:0]i;
25.      reg [13:0]C1;
26.      reg [15:0]D1;
27.      reg [63:0]T;
28.      reg [4:0]rCMD;
29.      reg [1:0]rBA;
30.      reg [12:0]rA;
31.      reg [1:0]rDQM;
32.      reg isOut;
33.      reg isDone;
34.
35.      always @ ( posedge CLOCK or negedge RESET )
36.          if( !RESET )
37.              begin
38.                  i <= 4'd0;
39.                  C1 <= 14'd0;
40.                  D1 <= 16'd0;
41.                  T <= 64'd0;
42.                  rCMD <= _NOP;
43.                  rBA <= 2'b11;
44.                  rA <= 13'h1fff;
45.                  rDQM <= 2b00;
46.                  isOut <= 1'b1;

```

```
47.           isDone <= 1'b0;
48.       end
```

以上内容为相关的寄存器声明与复位操作，注意寄存器 T 是用来暂存读取数据。

```
49.           else if( iCall[3] )
50.             case( i )
51.
52.               0: // Set IO to output State
53.               begin isOut <= 1'b1; i <= i + 1'b1; end
54.
55.               1: // Send Active Command with Bank and Row address
56.               begin rCMD <= _ACT; rBA <= iAddr[23:22]; rA <= iAddr[21:9]; i <= i + 1'b1; end
57.
58.               2: // wait TRCD 20ns
59.               if( C1 == TRCD -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
60.               else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
61.
62.               /*****
63.
64.               3: // Send Write command with row address, pull up A10 to PR
65.               begin rCMD <= _WR; rBA <= iAddr[23:22]; rA <= { 4'b0010, iAddr[8:0] }; D1 <= iData[63:48]; i <= i + 1'b1; end
66.
67.               4:
68.               begin rCMD <= _NOP; D1 <= iData[47:32]; i <= i + 1'b1; end
69.
70.               5:
71.               begin rCMD <= _NOP; D1 <= iData[31:16]; i <= i + 1'b1; end
72.
73.               6:
74.               begin rCMD <= _NOP; D1 <= iData[15:0]; i <= i + 1'b1; end
75.
76.               *****/
77.
78.               7: // wait TWR 2 clock
79.               if( C1 == TWR -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
80.               else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
81.
82.               8: // wait TRP 20ns
83.               if( C1 == TRP -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
84.               else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
85.
86.               *****/
```

```

87.
88.         9: // Generate done signal
89.         begin isDone <= 1'b1; i <= i + 1'b1; end
90.
91.         10:
92.         begin isDone <= 1'b0; i <= 4'd0; end
93.
94.     endcase

```

以上内容为部分核心操作。注意步骤 3~6，D1 用来驱动 SD_Q，而 iData 赋值 D1，次序由高至低。

```

95.     else if( iCall[2] )
96.         case( i )
97.
98.             0:
99.             begin isOut <= 1'b0; D1 <= 16'd0; i <= i + 1'b1; end
100.
101.            1: // Send Active command with Bank and Row address
102.            begin rCMD <= _ACT; rBA <= iAddr[23:22]; rA <= iAddr[21:9]; i <= i + 1'b1; end
103.
104.            2: // wait TRCD 20ns
105.            if( C1 == TRCD -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
106.            else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
107.
108.            *****/
109.
110.            3: // Send Read command and column address, pull up A10 to PR
111.            begin rCMD <= _RD; rBA <= iAddr[23:22]; rA <= { 4'b0010, iAddr[8:0] }; i <= i + 1'b1; end
112.
113.            4: // wait CL 3 clock
114.            if( C1 == CL -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
115.            else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
116.
117.            *****/
118.
119.            5: // Read Data
120.            begin T[63:48] <= S_DQ; i <= i + 1'b1; end
121.
122.            6: // Read Data
123.            begin T[47:32] <= S_DQ; i <= i + 1'b1; end
124.
125.            7: // Read Data

```

```

126.      begin T[31:16] <= S_DQ; i <= i + 1'b1; end
127.
128.      8: // Read Data
129.      begin T[15:0] <= S_DQ; i <= i + 1'b1; end
130.
131.      *****/
132.
133.      9: // Generate done signal
134.      begin rCMD <= _NOP; isDone <= 1'b1; i <= i + 1'b1; end
135.
136.      10:
137.      begin isDone <= 1'b0; i <= 4'd0; end
138.
139.      endcase

```

以上内容为部分核心操作。注意步骤 5~8，寄存器 T 用来暂存读数据，次序由高至低。

```

140.      else if( iCall[1] )
141.      case( i )
142.
143.          0: // Send Precharge Command
144.          begin rCMD <= _PR; i <= i + 1'b1; end
145.
146.          1: // wait TRP 20ns
147.          if( C1 == TRP -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
148.          else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
149.
150.          2: // Send Auto Refresh Command
151.          begin rCMD <= _AR; i <= i + 1'b1; end
152.
153.          3: // wait TRRC 63ns
154.          if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
155.          else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
156.
157.          4: // Send Auto Refresh Command
158.          begin rCMD <= _AR; i <= i + 1'b1; end
159.
160.          5: // wait TRRC 63ns
161.          if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
162.          else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
163.
164.          *****/
165.

```

```

166.           6: // Generate done signal
167.           begin isDone <= 1'b1; i <= i + 1'b1; end
168.
169.           7:
170.           begin isDone <= 1'b0; i <= 4'd0; end
171.
172.       endcase

```

以上内容为部分核心操作。刷新操作，基本上没有什么改变。

```

173.           else if( iCall[0] )
174.             case( i )
175.
176.               0: // delay 100us
177.               if( C1 == T100US -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
178.               else begin C1 <= C1 + 1'b1; end
179.
180.               /*****
181.
182.               1: // Send Precharge Command
183.               begin rCMD <= _PR; { rBA, rA } <= 15'h3fff; i <= i + 1'b1; end
184.
185.               2: // wait TRP 20ns
186.               if( C1 == TRP -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
187.               else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
188.
189.               3: // Send Auto Refresh Command
190.               begin rCMD <= _AR; i <= i + 1'b1; end
191.
192.               4: // wait TRRC 63ns
193.               if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
194.               else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
195.
196.               5: // Send Auto Refresh Command
197.               begin rCMD <= _AR; i <= i + 1'b1; end
198.
199.               6: // wait TRRC 63ns
200.               if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
201.               else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
202.
203.               /*****
204.
205.               7: // Send LMR Cmd. Burst Read & Write, 3'b011 mean CAS latency = 3, Sequentia 4 burst length

```

```

206. begin rCMD <= _LMR; rBA <= 2'b11; rA <= { 3'd0, 1'b0, 2'd0, 3'b011, 1'b0, 3'b010 }; i <= i + 1'b1; end
207.
208.     8: // Send 2 nop CLK for tMRD
209.     if( C1 == TMRD -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
210.     else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
211.
212.     /***** */
213.
214.     9: // Generate done signal
215.     begin isDone <= 1'b1; i <= i + 1'b1; end
216.
217.     10:
218.     begin isDone <= 1'b0; i <= 4'd0; end
219.
220.     endcase
221.

```

以上内容为部分核心操作。初始化操作，注意步骤 7，Burst Length 设置为 3'b010。

```

222. assign { S_CKE, S_NCS, S_NRAS, S_NCAS, S_NWE } = rCMD;
223. assign { S_BA, S_A } = { rBA, rA };
224. assign S_DQM = rDQM;
225. assign S_DQ = isOut ? D1 : 16'hzzzz;
226. assign oDone = isDone;
227. assign oData = T;
228.
229. endmodule

```

以上内容为相关的输出驱动，D1 驱动 S_DQ，T 驱动 oData。

sdram_ctrlmod.v

该控制模块的内容基本上与实验十八一模一样，笔者就不重复粘贴了。

sdran_demo.v

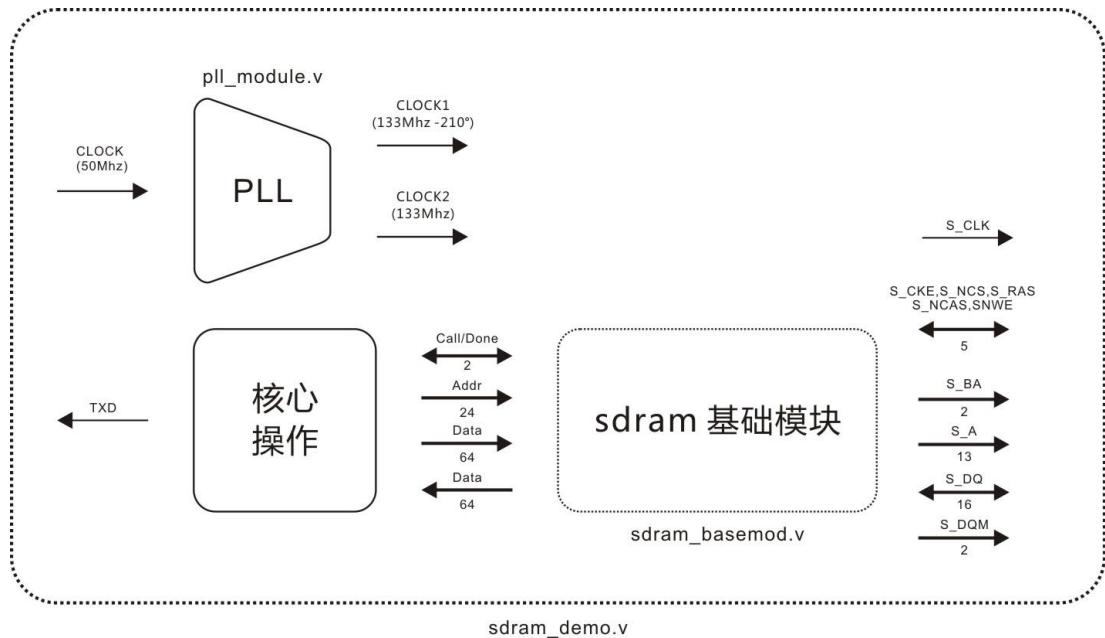


图 19.5 实验十九的建模图。

图 19.5 是实验十九的建模图，虽然外观上改变不大，最多只是 Data 的位宽改为 64 位而已 ... 话虽如此，核心操作则有点不同，具体的内容让我们来看代码吧。

```

1. module sdran_demo
2. (
3.     input CLOCK,
4.     input RESET,
5.     output S_CLK,
6.     output S_CKE, S_NCS, S_NRAS, S_NCAS, S_NWE,
7.     output [12:0]S_A,
8.     output [1:0]S_BA,
9.     output [1:0]S_DQM,
10.    inout [15:0]S_DQ,
11.    output TXD
12. );

```

以上内容为相关的出入端声明。

```

13.     wire CLOCK1,CLOCK2;
14.
15.     pll_module U1
16.     (
17.         .inclk0 ( CLOCK ), // 50Mhz
18.         .c0 ( CLOCK1 ), // 133Mhz -210 degree phase

```

```
19.           .c1 ( CLOCK2 ) // 133Mhz
20.       );
21.
```

以上内容为 PLL 模块的实例化。

```
22.      wire [1:0]DoneU2;
23.      wire [63:0]DataU2;
24.
25.      sdram_basemod U2
26.      (
27.          .CLOCK( CLOCK1 ),
28.          .RESET( RESET ),
29.          .S_CKE( S_CKE ),
30.          .S_NCS( S_NCS ),
31.          .S_NRAS( S_NRAS ),
32.          .S_NCAS( S_NCAS ),
33.          .S_NWE( S_NWE ),
34.          .S_A( S_A ),
35.          .S_BA( S_BA ),
36.          .S_DQM( S_DQM ),
37.          .S_DQ( S_DQ ),
38.          .iCall( isCall ),
39.          .oDone( DoneU2 ),
40.          .iAddr( {D1,2'b00} ),
41.          .iData( D2 ),
42.          .oData( DataU2 )
43.      );
44.
```

以上内容为 sdram 基础模块的实例化。

```
45.      parameter B115K2 = 11'd1157, TXFUNC = 6'd16;
46.
47.      reg [5:0]i,Go;
48.      reg [10:0]C1;
49.      reg [21:0]D1;
50.      reg [63:0]D2,D3;
51.      reg [10:0]T;
52.      reg [1:0]isCall;
53.      reg rTXD;
54.
55.      always @ ( posedge CLOCK1 or negedge RESET )
```

```

56.      if( !RESET )
57.          begin
58.              i <= 6'd0;
59.              Go <= 6'd0;
60.              C1 <= 11'd0;
61.              D1 <= 22'd0;
62.              D2 <= 64'd0;
63.              D3 <= 64'd0;
64.              T <= 11'd0;
65.              isCall <= 2'b00;
66.              rTXD <= 1'b1;
67.          end

```

以上内容为相关的寄存器声明还有复位操作。第 45 行是波特率为 115200 的常量声明还有伪函数入口。

```

68.      else
69.          case( i )
70.
71.              0:
72.                  if( DoneU2[1] ) begin isCall[1] <= 1'b0; i <= i + 1'b1; end
73.                  else begin isCall[1] <= 1'b1; D1 <= 22'd0; D2 <= 64'hAABCCDDEEFF8899; end
74.
75.              1:
76.                  if( DoneU2[0] ) begin D3 <= DataU2; isCall[0] <= 1'b0; i <= i + 1'b1; end
77.                  else begin isCall[0] <= 1'b1; D1 <= 22'd0; end
78.
79.              2:
80.                  begin T <= { 2'b11, D3[63:56], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
81.
82.              3:
83.                  begin T <= { 2'b11, D3[55:48], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
84.
85.              4:
86.                  begin T <= { 2'b11, D3[47:40], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
87.
88.              5:
89.                  begin T <= { 2'b11, D3[39:32], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
90.
91.              6:
92.                  begin T <= { 2'b11, D3[31:24], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
93.
94.              7:

```

```

95.          begin T <= { 2'b11, D3[23:16], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
96.
97.          8:
98.          begin T <= { 2'b11, D3[15:8], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
99.
100.         9:
101.         begin T <= { 2'b11, D3[7:0], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
102.
103.         10:
104.         i <= i;
105.
106.         *****/
107.

```

以上内容为部分核心操作。步骤 0 将 64 位的数据写入地址 0，步骤 1 则将数据从地址 0 读取。步骤 2~9 则是轮番将数据送出去。

```

108.          16,17,18,19,20,21,22,23,24,25,26:
109.          if( C1 == B115K2 -1 )begin C1 <= 11'd0; i <= i + 1'b1; end
110.          else begin rTXD <= T[i - 16]; C1 <= C1 + 1'b1; end
111.
112.          27:
113.          i <= Go;
114.
115.          endcase
116.
117.          assign S_CLK = CLOCK2;
118.          assign TXD = rTXD;
119.
120. endmodule

```

以上内容为部分核心操作。步骤 16~27 是发送一帧数据的伪函数。第 117~118 行则是相关的输出驱动。综合完毕并且下载程序，如果串口调试程序出现数据 AABBCCDDEEFF8899，结果表示实验成功。

细节一：完整的个体模块

本实验的 SDRAM 基础模块已经准备就绪。

细节二：读写地址的驱动方式

```

1. sram_basemod
2. (

```

```

3.      ...
4.      iAddr( {D1 , 2' b00} ),
5.      ...
6.  );
7.  reg [21:0]D1;
8.  ...
9.  always @ ( posedge CLOCK1 )
10.    ...
11.    case( i )
12.      0:
13.        if( ... ) ...
14.        else D1 <= 22'd0; ...

```

代码 19.4

代码 19.4 是 sdram_demo 的部分内容，其中 iAddr 由 22 位宽的 D1 与 2'b00 联合驱动，好奇的朋友一定会觉得疑惑“为什么”？其实这是经过深思以后的写法。实验十九是多字读写操作，其中长度为 4，或者说地址的偏移量为 4，所以 iAddr[1:0] 所指定的范围基本作废。为了正式这点问题，代码 19.4 需要这样表达。

```

1.  sdram_basemod
2.  (
3.    ...
4.    iAddr( D1 ),
5.    ...
6.  );
7.  reg [23:0]D1;
8.  ...
9.  always @ ( posedge CLOCK1 )
10.    ...
11.    case( i )
12.      0:
13.        if( ... ) ...
14.        else D1 <= 24'd1; ...

```

代码 19.5

如代码 19.5 所示，假设我们无视这个问题，直接使用 24 位宽的 D1 驱动 iAddr，然后将数据写入地址 24'd1。根据 SDRAM 的内部操作，数据会依序写入地址为 1234，而不是 0123 或者 4567 之类 ... 如此一来，我们会不小心破坏地址的偏移量。

实验二十：SDRAM 模块③ — 页读写 α

完成单字读写与多字读写以后，接下来我们要实验页读写。丑话当前，实验二十的页读写只是实验性质的东西，其中不存在任何实用价值，笔者希望读者可以把它当成页读写的热身运动。

表示 20.1 Mode Register 的内容。

Mode Register																				
A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0								
0	0	OP Code		0	0	CAS Latency				BT	Burst Length									
A3		Burst Type																		
0		Sequential																		
1		Interleave																		
A9		Write Mode																		
0		Burst Read and Burst Write																		
1		Burst Read and Single Write																		
A6		A5	A4	CAS Latency																
0	1	0		2																
0	1	1		3																

所谓页读写就是全列读写，而且表 20.1 告诉我们，页读写必须将 A2~A0 设置为 3'b111。然而，Verilog 的描述结果如代码 20.1 所示：

```
7: // Send LMR Cmd. Burst Read & Write, 3'b010 mean CAS latency = 3, Sequential,Full Page
begin rCMD <= _LMR; rBA <= 2'b11; rA <= { 3'd0, 1'b0, 2'd0, 3'b011, 1'b0, 3'b111 }; i <= i + 1'b1; end
```

代码 20.1

如果我们一页一页的叫，基本上“一页”的定义是非常暧昧的，因为“一页”所指定的范围会随着该存储器的容量而有所改变。举例 HY57V2562GTR 这只 SDRAM，地址的指定范围有 BA1~BA0，R12~R0，C8~C0，其中“一页”是全列，亦即 C8~C0。根据计算，C8~C0 等价 $2^9 = 512$ ，或者说页读写有 512 的地址偏移量。

页写操作：

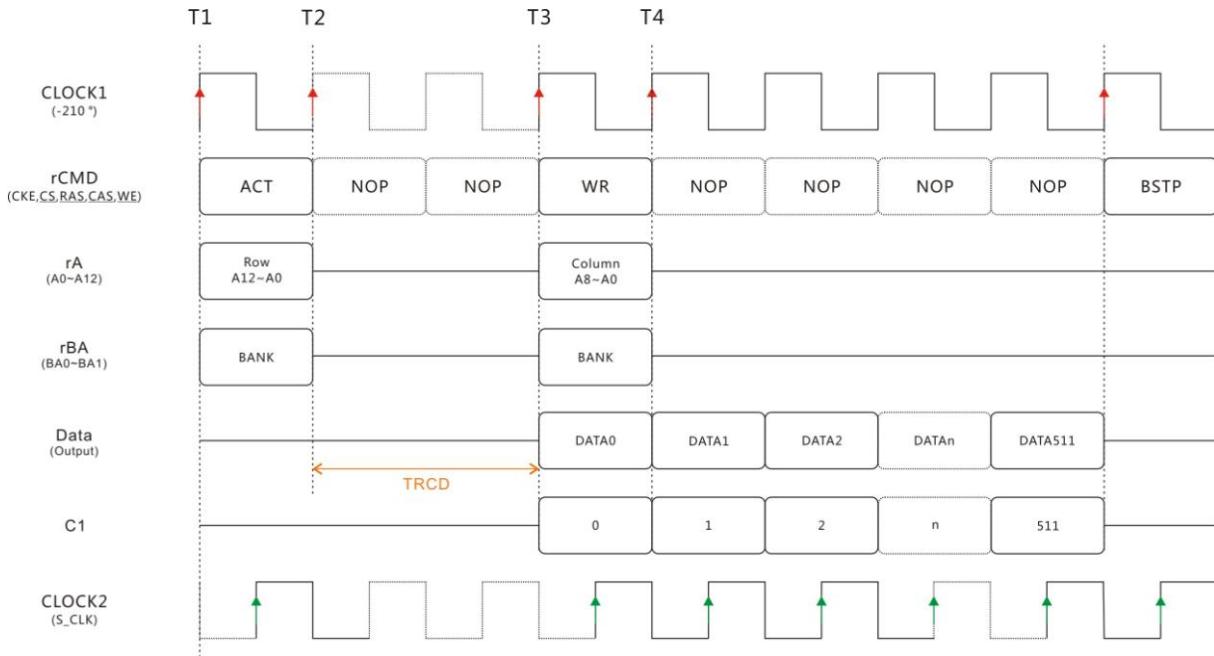


图 20.1 页写操作的理想时序图。

图 20.1 是笔者自定义的页写操作的理想时序图，其中 C1 是为了控制读写的次数。页读写相较字读写，前者好比一只不会停下冲锋的山猪。一旦读写开始，SDRAM 内部的计数器就会从 0 开始计数，计数结果为 511 又会从 0 重新计数。因为如此，页读写需要利用 BSTP 命令禁止山猪继续冲锋。

此外，自动预充对页读写来说是无效的东西，因此 A10 拉不拉高都没有关系，而且页写操作也不需要满足 TWR/TDPL 与 TPR。图 20.1 大致的时序过程如下：

- T1，发送 ACT 命令，BANK 地址与行地址；
- T1 半周期，SDRAM 读取；
- T2，满足 TRCD；
- T3，发送 WR 命令，BANK 地址与列地址，还有第 0 数据；
- T3 半周期，SDRAM 读取
- T4，发送第 1~511 数据，然后发送 BSTP 命令结束页写。

Verilog 则可以这样描述，结果如代码 20.2 所示：

```

1. 1: // Send Active Command with Bank and Row address
2. begin rCMD <= _ACT; rBA <= iAddr[23:22]; rA <= iAddr[21:9]; i <= i + 1'b1; end
3.
4. 2: // wait TRCD 20ns
5. if( C1 == TRCD - 1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
6. else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
7.
8. 3: // Send Write command with row address
9. begin rCMD <= _WR; rBA <= iAddr[23:22]; rA <= { 4'b0010, iAddr[8:0] }; D1 <= iData; i <= i + 1'b1; end

```

```

10.
11. 4: // continue write until end and send BSTP
12. if( C1 == 512 - 1 ) begin rCMD <= _BSTP; C1 <= 14'd0; i <= i + 1'b1 ;end
13. else begin rCMD <= _NOP; C1 <= C1 + 1'b1; D1 <= D1 + 1'b1; end

```

代码 20.2

如代码 20.2 所示，步骤 3 写第 0 数据，步骤 4 则写入第 1~511 数据并且发送 BSTP 命令。

页读操作：

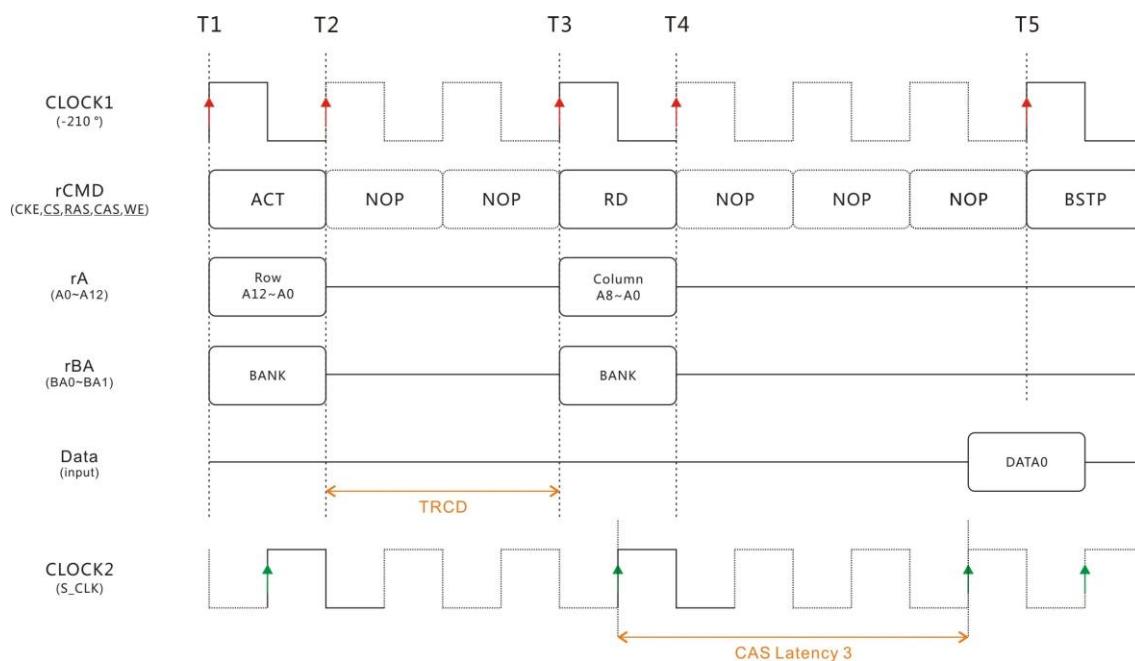


图 20.2 页读操作的理想时序图。

图 20.2 也是笔者自定义的理想时序图。同样，页读操也是一只不断冲锋的山猪，因此它需要 BSTP 这支停下的告示牌。除此之外，页读也没有自行预充电的必要，而且 TPR 也不用满足。实验二十要实验的页读比较单纯，我们读取第 0 数据以后立即发送 BSTP 命令来结束操作。图 20.2 大致的时序过程如下：

- T1，发送 ACT 命令，BANK 地址与行地址；
- T1 半周期，SDRAM 读取；
- T2，满足 TRCD；
- T3，发送 RD 命令，BANK 地址与列地址；
- T3 半周期，SDRAM 读取命令。
- T4，满足 CAS Latency。
- T5，读取第 0 数据，然后发送 BSTP 命令。

Verilog 则可以这样描述，结果如代码 20.3 所示：

```
1. 1: // Send Active command with Bank and Row address
2. begin rCMD <= _ACT; rBA <= iAddr[23:22]; rA <= iAddr[21:9]; i <= i + 1'b1; end
3.
4. 2: // wait TRCD 20ns
5. if( C1 == TRCD - 1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
6. else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
7.
8. 3: // Send Read command and column address
9. begin rCMD <= _RD; rBA <= iAddr[23:22]; rA <= { 4'b0010, iAddr[8:0]}; i <= i + 1'b1; end
10.
11. 4: // wait CL 3 clock
12. if( C1 == CL - 1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
13. else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
14.
15. 5: // Read Data
16. begin D1 <= S_DQ; rCMD <= _BSTOP; i <= i + 1'b1; end
```

代码 20.3

如代码 20.3 所示，步骤 5 读取数据以后立即发送 BSTP 命令以示结束页读操作。理解完毕以后我们便可以开始建模了。

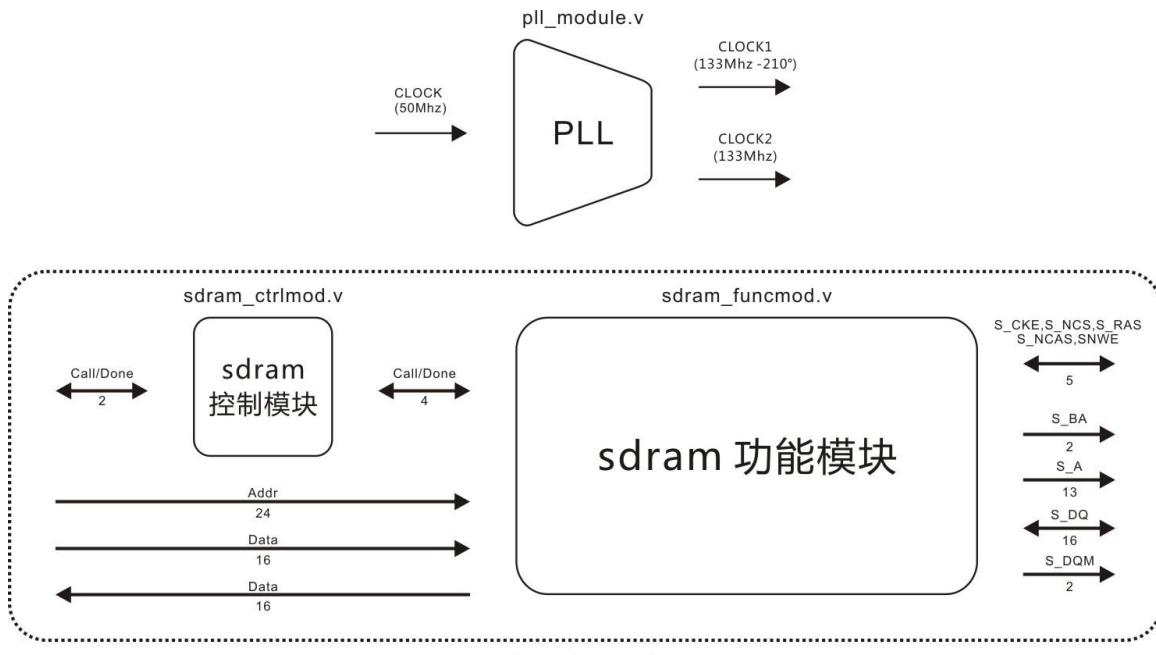


图 20.3 SDRAM 基础模块的建模图。

图 20.3 是 SDRAM 基础模块的建模图，外表上和实验十八差不多，不过 SDRAM 功能模块的内容却有一些改变。

sdram_funcmod.v

```
1. module sdram_funcmod
2. (
3.     input CLOCK,
4.     input RESET,
5.
6.     output S_CKE, S_NCS, S_NRAS, S_NCAS, S_NWE,
7.     output [1:0]S_BA,
8.     output [12:0]S_A,
9.     output [1:0]S_DQM,
10.    inout [15:0]S_DQ,
11.
12.    input [3:0]iCall,
13.    output oDone,
14.    input [23:0]iAddr, // [23:22]BA,[21:9]Row,[8:0]Column
15.    input [15:0]iData,
16.    output [15:0]oData
17. );
```

第 3~16 行是相关的输入端声明。

```
18. parameter T100US = 14'd13300;
19. // tRP 20ns, tRRC 63ns, tRCD 20ns, tMRD 2CLK, tWR/tDPL 2CLK, CAS Latency 3CLK
20. parameter TRP = 14'd3, TRRC = 14'd9, TMRD = 14'd2, TRCD = 14'd3, TWR = 14'd2, CL = 14'd3;
21. parameter _INIT = 5'b01111, _NOP = 5'b10111, _ACT = 5'b10011, _RD = 5'b10101, _WR = 5'b10100,
22.           _BSTP = 5'b10110, _PR = 5'b10010, _AR = 5'b10001, _LMR = 5'b10000;
23.
```

第 18~22 行是相关的常量声明。

```
24. reg [4:0]i;
25. reg [13:0]C1;
26. reg [15:0]D1;
27. reg [4:0]rCMD;
28. reg [1:0]rBA;
29. reg [12:0]rA;
30. reg [1:0]rDQM;
31. reg isOut;
32. reg isDone;
33.
```

```

34.      always @ ( posedge CLOCK or negedge RESET )
35.          if( !RESET )
36.              begin
37.                  i <= 4'd0;
38.                  C1 <= 14'd0;
39.                  D1 <= 16'd0;
40.                  rCMD <= _NOP;
41.                  rBA <= 2'b11;
42.                  rA <= 13'h1fff;
43.                  rDQM <= 2'b00;
44.                  isOut <= 1'b1;
45.                  isDone <= 1'b0;
46.              end

```

第 24~46 行是相关的寄存器声明与复位操作。

```

47.      else if( iCall[3] )
48.          case( i )
49.
50.              0: // Set IO to output State
51.              begin isOut <= 1'b1; i <= i + 1'b1; end
52.
53.              1: // Send Active Command with Bank and Row address
54.              begin rCMD <= _ACT; rBA <= iAddr[23:22]; rA <= iAddr[21:9]; i <= i + 1'b1; end
55.
56.              2: // wait TRCD 20ns
57.              if( C1 == TRCD - 1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
58.              else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
59.
60.              /*****
61.
62.              3: // Send Write command with row address
63.              begin rCMD <= _WR; rBA <= iAddr[23:22]; rA <= { 4'b0010, iAddr[8:0] }; D1 <= iData; i <= i + 1'b1; end
64.
65.              4: // continue write until end and send BSTP
66.              if( C1 == 512 - 1 ) begin rCMD <= _BSTP; C1 <= 14'd0; i <= i + 1'b1 ;end
67.              else begin rCMD <= _NOP; C1 <= C1 + 1'b1; D1 <= D1 + 1'b1; end
68.
69.              *****/
70.
71.              5: // Generate done signal
72.              begin rCMD <= _NOP; isDone <= 1'b1; i <= i + 1'b1; end
73.

```

```

74.           6:
75.             begin isDone <= 1'b0; i <= 4'd0; end
76.
77.           endcase

```

以上内容为页写操作，注意步骤 3 写入第 0 数据，步骤 4 则写入第 1~511 数据并且发送 BSTP 命令。

```

78.       else if( iCall[2] )
79.         case( i )
80.
81.           0:
82.             begin isOut <= 1'b0; D1 <= 16'd0; i <= i + 1'b1; end
83.
84.           1: // Send Active command with Bank and Row address
85.             begin rCMD <= _ACT; rBA <= iAddr[23:22]; rA <= iAddr[21:9]; i <= i + 1'b1; end
86.
87.           2: // wait TRCD 20ns
88.             if( C1 == TRCD -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
89.             else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
90.
91.             *****/
92.
93.           3: // Send Read command and column address
94.             begin rCMD <= _RD; rBA <= iAddr[23:22]; rA <= { 4'b0010, iAddr[8:0] }; i <= i + 1'b1; end
95.
96.           4: // wait CL 3 clock
97.             if( C1 == CL -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
98.             else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
99.
100.            *****/
101.
102.           5: // Read Data
103.             begin D1 <= S_DQ; rCMD <= _BSTOP; i <= i + 1'b1; end
104.
105.            *****/
106.
107.           6: // Generate done signal
108.             begin rCMD <= _NOP; isDone <= 1'b1; i <= i + 1'b1; end
109.
110.           7:
111.             begin isDone <= 1'b0; i <= 4'd0; end
112.

```

```
113.         endcase
```

以上内容为页读操作，注意步骤 5 是读取第 0 数据并且发送 BSTP 命令。

```
114.     else if( iCall[1] )
115.         case( i )
116.
117.             0: // Send Precharge Command
118.             begin rCMD <= _PR; i <= i + 1'b1; end
119.
120.             1: // wait TRP 20ns
121.             if( C1 == TRP -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
122.             else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
123.
124.             2: // Send Auto Refresh Command
125.             begin rCMD <= _AR; i <= i + 1'b1; end
126.
127.             3: // wait TRRC 63ns
128.             if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
129.             else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
130.
131.             4: // Send Auto Refresh Command
132.             begin rCMD <= _AR; i <= i + 1'b1; end
133.
134.             5: // wait TRRC 63ns
135.             if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
136.             else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
137.
138.             *****/
139.
140.             6: // Generate done signal
141.             begin isDone <= 1'b1; i <= i + 1'b1; end
142.
143.             7:
144.             begin isDone <= 1'b0; i <= 4'd0; end
145.
146.         endcase
```

以上内容是刷新操作。

```
147.     else if( iCall[0] )
148.         case( i )
149.
```

```

150.          0: // delay 100us
151.          if( C1 == T100US -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
152.          else begin C1 <= C1 + 1'b1; end
153.
154.          *****/
155.
156.          1:// Send Precharge Command
157.          begin rCMD <= _PR; { rBA, rA } <= 15'h3fff; i <= i + 1'b1; end
158.
159.          2:// wait TRP 20ns
160.          if( C1 == TRP -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
161.          else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
162.
163.          3:// Send Auto Refresh Command
164.          begin rCMD <= _AR; i <= i + 1'b1; end
165.
166.          4:// wait TRRC 63ns
167.          if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
168.          else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
169.
170.          5:// Send Auto Refresh Command
171.          begin rCMD <= _AR; i <= i + 1'b1; end
172.
173.          6:// wait TRRC 63ns
174.          if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
175.          else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
176.
177.          *****/
178.
179.          7:// Send LMR Cmd. Burst Read & Write,  3'b010 mean CAS latecy = 3, Sequential,Full Page
180.          begin rCMD <= _LMR; rBA <= 2'b11; rA <= { 3'd0, 1'b0, 2'd0, 3'b011, 1'b0, 3'b111 }; i <= i + 1'b1; end
181.
182.          8:// Send 2 nop CLK for tMRD
183.          if( C1 == TMRD -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
184.          else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
185.
186.          *****/
187.
188.          9:// Generate done signal
189.          begin isDone <= 1'b1; i <= i + 1'b1; end
190.
191.          10:
192.          begin isDone <= 1'b0; i <= 4'd0; end

```

```
193.  
194.      endcase  
195.
```

以上内容是初始化，注意步骤 7 的 Mode Register 内容，Busrt Length 为 3'b111。

```
196.      assign { S_CKE, S_NCS, S_NRAS, S_NCAS, S_NWE } = rCMD;  
197.      assign { S_BA, S_A } = { rBA, rA };  
198.      assign S_DQM = rDQM;  
199.      assign S_DQ = isOut ? D1 : 16'hzzzz;  
200.      assign oDone = isDone;  
201.      assign oData = D1;  
202.  
203. endmodule
```

第 196~201 行是相关的输出驱动。

sdram_ctrlmod.v

该控制模块的内容与实验十八一致。

sdram_basemod.v

该组合模块的内容也与实验十八一致

sdram_demo.v

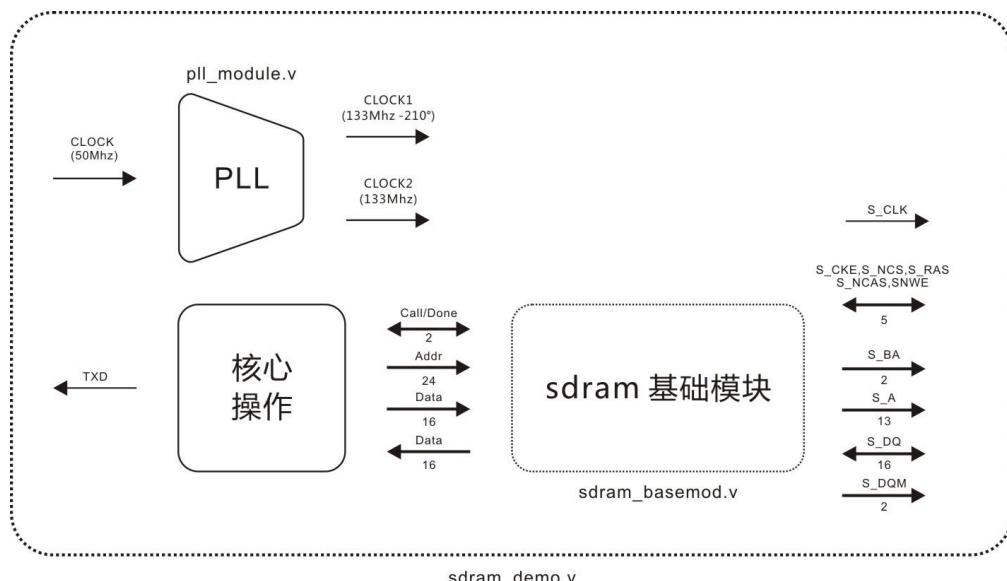


图 20.4 实验二十的建模图。

图 20.4 是实验二十的建模图 , 外观上与实验十八一样 , 不过核心操作的内容却有所不同 , 具体内容我们还是来看代码吧。

```
1. module sdram_demo
2. (
3.     input CLOCK,
4.     input RESET,
5.     output S_CLK,
6.     output S_CKE, S_NCS, S_NRAS, S_NCAS, S_NWE,
7.     output [12:0]S_A,
8.     output [1:0]S_BA,
9.     output [1:0]S_DQM,
10.    inout [15:0]S_DQ,
11.    output TXD
12. );

```

以上内容为相关的出入端声明。

```
13.     wire CLOCK1,CLOCK2;
14.
15.     pll_module U1
16.     (
17.         .inclk0( CLOCK ), // 50Mhz
18.         .c0( CLOCK1 ), // 133Mhz -210 degree phase
19.         .c1( CLOCK2 ) // 133Mhz
20.     );
21.
```

以上内容为 PLL 模块的实例化。

```
22.     wire [1:0]DoneU2;
23.     wire [15:0]DataU2;
24.
25.     sdram_basemod U2
26.     (
27.         .CLOCK( CLOCK1 ),
28.         .RESET( RESET ),
29.         .S_CKE( S_CKE ),
30.         .S_NCS( S_NCS ),
31.         .S_NRAS( S_NRAS ),
32.         .S_NCAS( S_NCAS ),
```

```

33.      .S_NWE( S_NWE ),
34.      .S_A( S_A ),
35.      .S_BA( S_BA ),
36.      .S_DQM( S_DQM ),
37.      .S_DQ( S_DQ ),
38.      .iCall( isCall ),
39.      .oDone( DoneU2 ),
40.      .iAddr( D1 ),
41.      .iData( D2 ),
42.      .oData( DataU2 )
43. );
44.

```

以上内容为 SDRAM 基础模块的实例化。

```

45. parameter B115K2 = 11'd1157, TXFUNC = 6'd16;
46.
47. reg [5:0]i,Go;
48. reg [10:0]C1;
49. reg [23:0]D1;
50. reg [15:0]D2,D3;
51. reg [10:0]T;
52. reg [1:0]isCall;
53. reg rTXD;
54.
55. always @ ( posedge CLOCK1 or negedge RESET )
56.     if( !RESET )
57.         begin
58.             i <= 6'd0;
59.             Go <= 6'd0;
60.             C1 <= 11'd0;
61.             D1 <= 24'd0;
62.             D2 <= 16'd0;
63.             D3 <= 16'd0;
64.             T <= 11'd0;
65.             isCall <= 2'b00;
66.             rTXD <= 1'b1;
67.         end
68.     else

```

以上内容为相关的寄存器声明还有复位操作。第 45 行是波特率还有伪函数入口的常量声明。

```

69.           case( i )
70.
71.             0:
72.               if( DoneU2[1] ) begin isCall[1] <= 1'b0; i <= i + 1'b1; end
73.               else begin isCall[1] <= 1'b1; D1 <= 24'd0; D2 <= 16'hA000; end
74.
75.             1:
76.               if( DoneU2[0] ) begin D3 <= DataU2; isCall[0] <= 1'b0; i <= i + 1'b1; end
77.               else begin isCall[0] <= 1'b1; end
78.
79.             2:
80.               begin T <= { 2'b11, D3[15:8], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
81.
82.             3:
83.               begin T <= { 2'b11, D3[7:0], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
84.
85.             4:
86.               if( D1 == 24'd511 ) i <= i + 1'b1;
87.               else begin D1 <= D1 + 1'b1; i <= 6'd1; end
88.
89.             5:
90.               i <= i;
91.
92.             *****/
93.

```

以上内容为部分核心操作。步骤 0 将数据 16'hA×××从地址 0 写至地址 511，其中×××会经由页写而自行递增。换句话说，数据 16'hA000~16'hA1FF 从地址 0 写至地址 511。步骤 1 则用来读取数据，步骤 2~3 将读出的数据——发送出去。步骤 4 用来递增地址，从 0~511，然后返回步骤 1，直至人为页读结束。

```

94.           16,17,18,19,20,21,22,23,24,25,26:
95.           if( C1 == B115K2 -1 ) begin C1 <= 11'd0; i <= i + 1'b1; end
96.           else begin rTXD <= T[i - 16]; C1 <= C1 + 1'b1; end
97.
98.           27:
99.           i <= Go;
100.
101.         endcase
102.
103.         assign S_CLK = CLOCK2;
104.         assign TXD = rTXD;
105.

```

```
106. endmodule
```

以上内容为部分核心操作。步骤 16~27 是发送一帧数据的伪函数。第 103~104 行则是相关的输出驱动。综合完毕并且下载程序，如果串口调试软件出现数据 A000~A1FF 表示实验成功。

实验二十一：SDRAM 模块④ — 页读写 β

未进入主题之前，让我们先来谈谈一些重要的体外话。《整合篇》之际，笔者曾经比拟 Verilog 如何模仿 for 循环，我们知道 for 循环是顺序语言的产物，如果 Verilog 要实现属于自己的 for 循环，那么它要考虑的东西除了步骤以外，还有非常关键的时钟。

```
for( i=0; i<4; i++ ) 操作 A ;  
  
i = 0 ;  
while ( i<4 ) { 操作 A ; i++ ; }  
  
i = 0 ;  
do { 操作 A ; i++ ; } while ( i = 3 )
```

代码 21.1

代码 2.11 有三段经典的循环操作，即 for 循环，while 循环，还有 do ... while 循环。如果三个循环互相交流的话，谁又是谁的朋友呢？为了解决这个问题，我们必须先建立交友标准，即先[执行后判断，还是先判断后执行？](#)如此一来，我们可以百分百确信 for 循环与 while 循环才是好朋友，因为两者都是先判断后执行的类型。反之 do ... while 循环则是先执行后判断的类型。

为了验证上述的接着，我们试着解剖一下种循环的执行过程 ...

for 循环：

清零 i

i 为 0，判断 i 是否小于 4，如是执行操作 A 第 0 次，递增 i 为 1；
i 为 1，判断 i 是否小于 4，如是执行操作 A 第 1 次，递增 i 为 2；
i 为 2，判断 i 是否小于 4，如是执行操作 A 第 2 次，递增 i 为 3；
i 为 3，判断 i 是否小于 4，如是执行操作 A 第 3 次，递增 i 为 4；
i 为 4，判断 i 是否小于 4，不是结束循环。

while 循环：

清零 i；

i 为 0，判断 i 是否小于 4，如是执行操作 A 第 0 次，递增 i 为 1；
i 为 1，判断 i 是否小于 4，如是执行操作 A 第 1 次，递增 i 为 2；
i 为 2，判断 i 是否小于 4，如是执行操作 A 第 2 次，递增 i 为 3；
i 为 3，判断 i 是否小于 4，如是执行操作 A 第 3 次，递增 i 为 4；
i 为 4，判断 i 是否小于 4，不是结束循环。

do ... while 循环：

清零 i；

i 为 0，执行操作 A 第 0 次，i 递增为 1，判断 i 是否小于 4，如是继续；

i 为 1，执行操作 A 第 1 次，i 递增为 2，判断 i 是否小于 4，如是继续；

i 为 2，执行操作 A 第 2 次，i 递增为 3，判断 i 是否小于 4，如是继续；

i 为 3，执行操作 A 第 3 次，i 递增为 4，判断 i 是否小于 4，不是则结束循环；

读者可能会很奇怪，执行与判断的次序究竟有什么好困惑？作为一只小气鬼，笔者会非常执著小细节。Verilog 是并行性质的语言，执行与判断有可能同时执行，也有可能并非同时执行 … 根据直觉，笔者相信 Verilog 的循环操作更加适合[先执行后判断](#)，而不是先判断后执行。

```
1.    0:  
2.    begin  
3.      if( C1 == 0 ) isEn <= 1' b1;  
4.      else if( C1 == 4-2 ) isEn <= 1' b0;  
5.  
6.      if( C1 == 4 -1 ) begin C1 <= 4' d0; i <= i + 1' b1; end  
7.      else C1 <= C1 + 1' b1;  
8.    end
```

代码 21.1

如代码 21.1 所示，第 6~7 行表示步骤 0 保持 4 个时钟，其中 C1 为 0 拉高 isEn，C1 为 4-2 的拉低 isEn；代码 21.1 告诉我们，执行与判断是同时进行，然而从解读代码的顺序来看，笔者更加倾向“先执行后判断”这种结构性

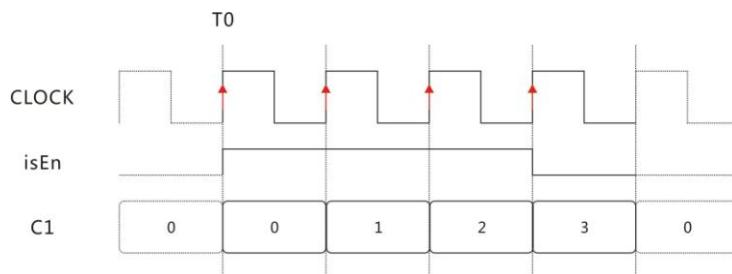


图 21.1 代码 21.1 的时序图。

图 21.1 是代码 21.1 所描述的时序图，其中 isEn 拉高是否完全根据 C1 的计数。T0 之前也是复位的状态，C1 为 0。T0 之际，C1 为 0（过去值）拉高 isEn，C1 为 2（过去值）拉低 isEn。

```
1.    0:  
2.    begin isEn <= 1' b1; i <= i + 1' b1; end  
3.    1:  
4.    begin isEn <= 1' b0; i <= i + 1' b1; end
```

```

5. 2 :
6. if( C1 == 4 -1 ) begin C1 <= 4' d0; i <= i + 1' b1; end
7. else begin C1 <= C1 + 1' b1; i <= 4' d0; end

```

代码 21.2

同样的结构性甚至可以衍生至不同的操作，如代码 21.2 所示。步骤 0 拉高 isEn，步骤 1 拉低 isEn，步骤 2 判断。如果步骤 0~2 来回重复，isEn 一共拉高 4 次，或者说 isEn 产生四个高脉冲。

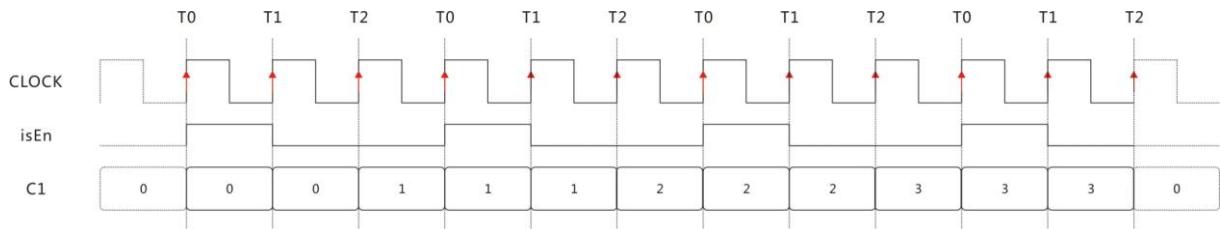


图 21.2 代码 21.2 的理想时序图。

图 21.2 是代码 21.2 所产生的时序图。只要稍微比较图 21.1 与图 21.2，我们会发现两者之间都有相似的“结构性”，因为两者都是倾向“先执行后判断”。

```

1. 0:
2. if( Done ) begin isCall[0] <= 1' b0; i <= i + 1' b1; end
3. else begin isCall[0] <= 1' b1; end
4. 1:
5. if( Done ) begin isCall[1] <= 1' b0; i <= i + 1' b1; end
6. else begin isCall[1] <= 1' b1; end
7. 2:
8. if( C1 == 4 -1 ) begin C1 <= 4' d0; i <= i + 1' b1; end
9. else begin C1 <= C1 + 1' b1; i <= 4' d0; end

```

代码 21.3

再举例而言，如代码 21.3 所示，步骤 0 执行功能 0，步骤 1 执行功能 1，步骤 2 用来判断循环次数。步骤 0~2 之间会来回重复，直至功能 0 与 1 都执行四次，这种感觉好比代码 21.4。

```

1. int main()
2. {
3.     for( int i = 0; i < 4; i ++ )
4.     {
5.         Function0();
6.         Function1();
7.     }
8.     return -1;

```

不过代码 21.1~21.3 都有相似的结构性，即都是先执行后判断。笔者作为热爱结构的男人，笔者会尝尽一切挖掘结构的可能性。此外，这种先执行后判断的模仿对象，笔者称为伪循环（Fake Iteration）。说完题外话，接下来让我们进入本实验的主题。

页读写之所以分为 α 与 β ，那是因为过多的数据吞吐量导致读写变成非常麻烦。天真的朋友可能会认为，如果 Burst Length 为 4，那么数据位宽就是 $4 * 16\text{bit}$ 。再如果 Burst Length 为 512，那么数据位宽就是 $512 * 16\text{ bit}$ 。理论上，这样说是没错，不过那样做会撑爆 FPGA 的嘴巴。为此，读写入口必须加入缓冲机制才行。此刻，实验十五所学过的知识（同步 FIFO）就派上用场了。

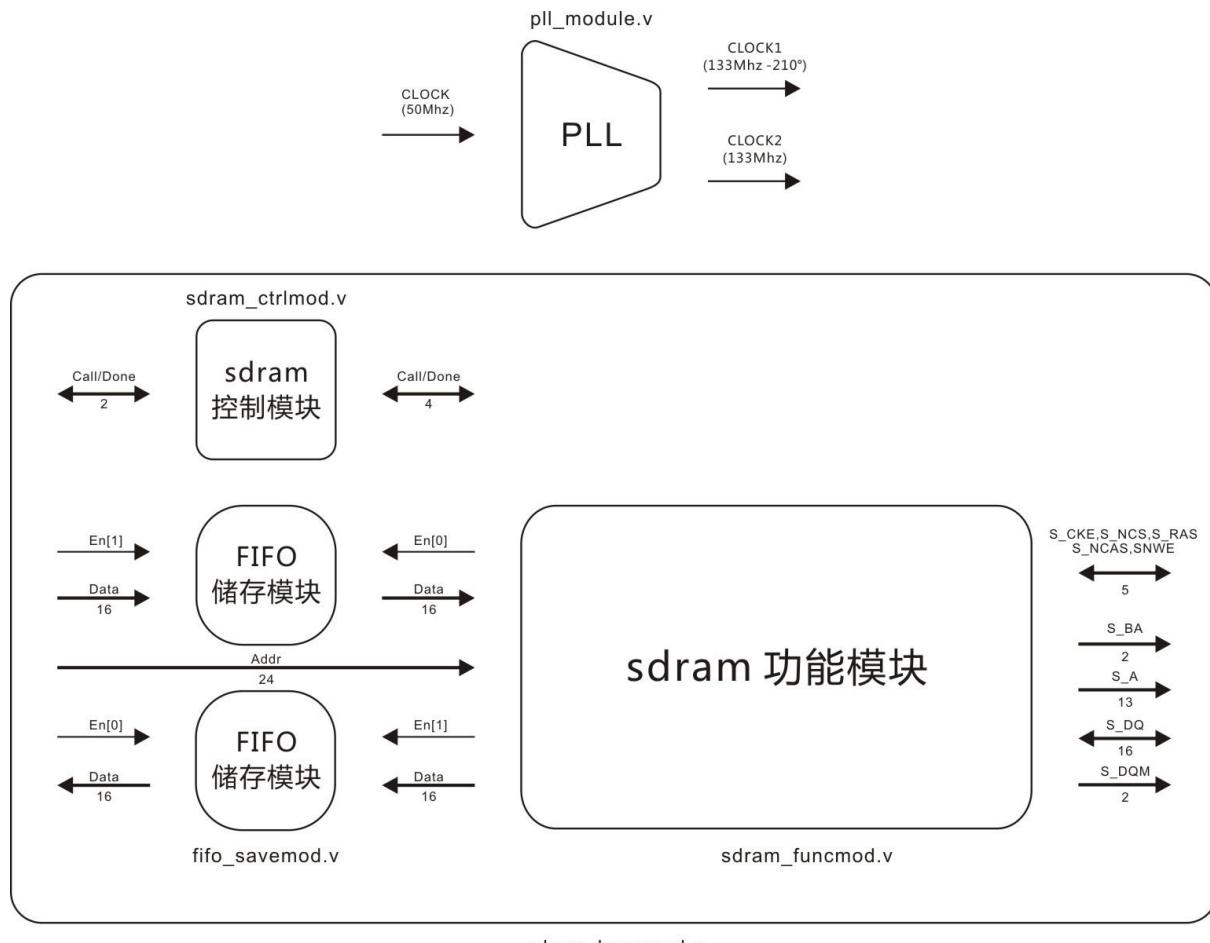


图 21.3 SDRAM 基础模块的建模图。

加入 FIFO 储存模块的 SDRAM 基础模块，大致上如图 21.3 所示。为了简化连线，笔者稍微加大 FIFO 的深度，对此 FIFO 有没有反馈状态都没有问题。还没有进入建模之前，让笔者先来解释一下，页读写与 FIFO 之间，究竟有什么细节需要注意。

页写操作 (请求 FIFO) :

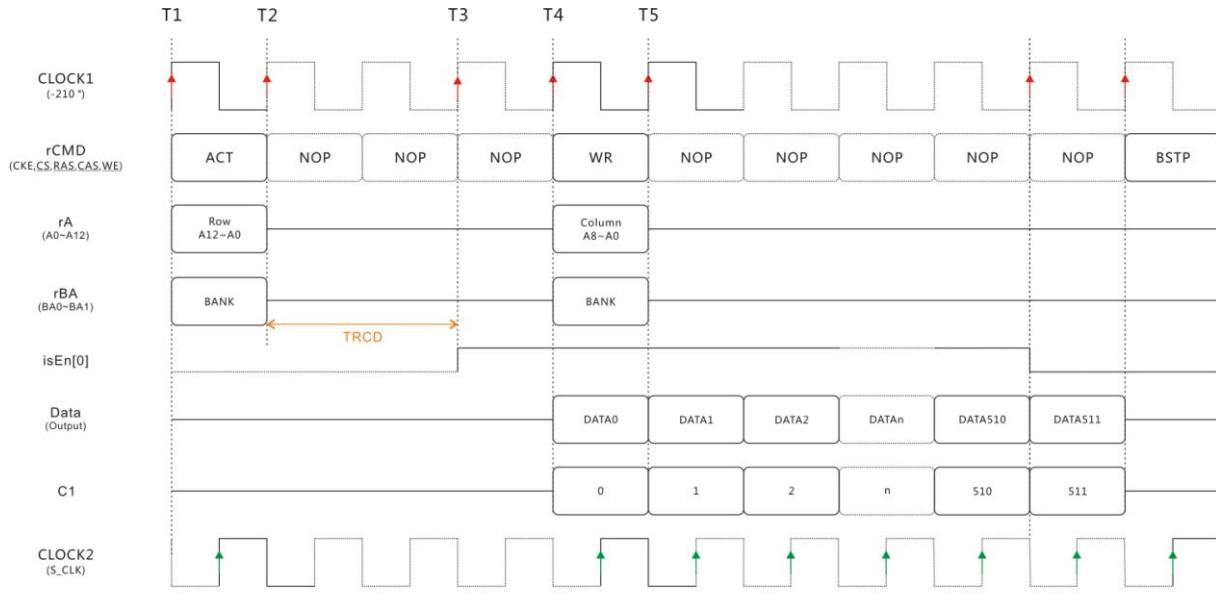


图 21.4 页写操作的理想时序图。

图 21.4 是页写操作的理想时序图，相较实验二十的页写时序，实验二十一的页写时序夹杂了 FIFO 储存模块，其中 isEn[0]是 SDRAM 功能模块向 FIFO 的读请求。时序的大致过程如下：

- T1，发送 ACT 命令，BANK 地址与行地址；
- T1 半周期，SDRAM 读取；
- T2，满足 TRCD；
- T3，拉高 isEn[0]；
- T4，发送 WR 命令，BANK 地址与列地址，FIFO 发送第 0 数据；
- T4 半周期，SDRAM 读取
- T5，FIFO 发送第 1~511 数据，C1 为 510 拉低 isEn[0]，C1 为 511 发送 BSTP 命令。

图 22.4 基本上已经表达非常清楚，为使 FIFO 可以同步发送数据，因为 SDRAM 功能模块必须提前一个时钟拉高 isEn[0]，即 T3 拉高 isEn[0]。此外，为了不使 FIFO 吐出过多的数据，C1 为 510 的时候便拉低 isEn[1]。C1 为 511 的时候则发送 BSTP 命令。对此，Verilog 则可以这样描述，结果如代码 21.5 所示：

```

1. 1: // Send Active Command with Bank and Row address
2. begin rCMD <= _ACT; rBA <= iAddr[23:22]; rA <= iAddr[21:9]; i <= i + 1'b1; end
3.
4. 2: // wait TRCD 20ns
5. if( C1 == TRCD - 1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
6. else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end

```

```

7.
8. 3:
9. begin isEn[0] <= 1'b1; i <= i + 1'b1; end
10.
11. 4: // Send Write command with row address,
12. begin rCMD <= _WR; rBA <= iAddr[23:22]; rA <= { 4'b0000, iAddr[8:0] }; i <= i + 1'b1; end
13.
14. 5: // continue write until end and send BSTP
15. begin
16.     if( C1 == 512 -2 ) begin isEn[0] <= 1'b0; end
17.     if( C1 == 512 -1 ) begin rCMD <= _BSTP; C1 <= 14'd0; i <= i + 1'b1; end
18.     else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
19. end

```

代码 21.5

如代码 21.5 所示，步骤 1 发送 Active 命令，步骤 2 满足 TRCD，步骤 3 提前拉高读请求，即 isEn[0]。步骤 4 发送 Write 命令，还有写入第 0 数据。步骤 5 写入第 1~511 数据，C1 为 510 的时候拉低 isEn[0]，C1 为 511 的时候发送 Burst Stop 命令。

页读操作（请求 FIFO）：

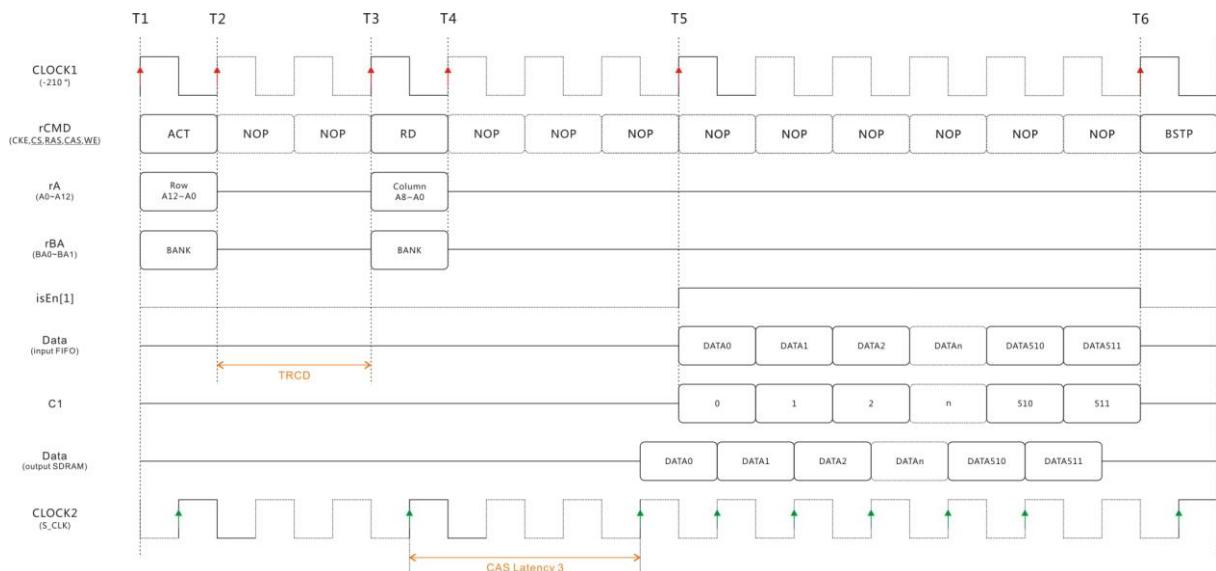


图 21.5 页读操作的理想时序图。

图 21.5 还是笔者自定义的页写的时序图，相较实验二十，其中多了 FIFO 的写请求 isEn[1]，FIFO 的 iData，还有 C1 计数。一旦 CAS Latency 得到满足，SDRAM 便会开始读出数据。半个周期之后(T5)FPGA 读取，并且拉高 isEn[1]，然后将数据转交 FIFO。T6 之际，512 个数据读写完毕，拉低 isEn[1]，然后发送 BSTP 命令。

时序的大致过程如下：

- T1 , 发送 ACT 命令 , BANK 地址与行地址 ;
- T1 半周期 , SDRAM 读取 ;
- T2 , 满足 TRCD ;
- T3 , 发送 RD 命令 , BANK 地址与列地址 ;
- T3 半周期 , SDRAM 读取命令。
- T4 , 满足 CAS Latency。
- T5 , 拉高 isEn[1] , 读取第 0~511 数据 , 并且向 FIFO 的 iData 写入。
- T6 , 拉低 isEn[1] , 发送 BSTP 命令。
- T6 半周期 , SDRAM 读取。

对此 , Verilog 可以这样描述 , 结果如代码 21.6 所示:

```

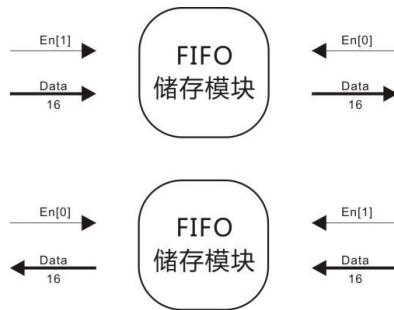
1. 1: // Send Active command with Bank and Row address
2. begin rCMD <= _ACT; rBA <= iAddr[23:22]; rA <= iAddr[21:9]; i <= i + 1'b1; end
3.
4. 2: // wait TRCD 20ns
5. if( C1 == TRCD -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
6. else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
7.
8. 3: // Send Read command and column address
9. begin rCMD <= _RD; rBA <= iAddr[23:22]; rA <= { 4'b0000, iAddr[8:0]}; i <= i + 1'b1; end
10.
11. 4: // wait CL 3 clock
12. if( C1 == CL -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
13. else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
14.
15. 5: // Read Data
16. begin
17.     D1 <= S_DQ; isEn[1] <= 1'b1;
18.     if( C1 == 512 -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
19.     else begin C1 <= C1 + 1'b1; end
20. end
21.
22. 6:
23. begin isEn[1] <= 1'b0; rCMD <= _BSTP; i <= i + 1'b1; end

```

代码 21.6

如代码 21.6 所示 , 步骤 4 满足 CL 以后 , 步骤 5 便拉高 isEn[1] , 并且读取 512 个数据。步骤 6 将 isEn[1] 拉低 , 并且发送命令 BSTP。理解完毕以后 , 我们可以开始建模了。

fifo_savemod.v



fifo_savemod.v

图 21.6 FIFO 储存模块的建模图。

图 21.6 是 FIFO 储存模块的建模图，为了简化设计，笔者不小心吃掉 FIFO 的输出标签，取而代之就是增加深度。虽然实验只有两只 FIFO 储存模块，方向也不一样，不过母体都是一样的东西。具体内容我们还是来看代码吧：

```
1. module fifo_savemod
2. (
3.     input CLOCK, RESET,
4.     input [1:0]iEn,
5.     input [15:0]iData,
6.     output [15:0]oData,
7.     output [1:0]oTag
8. );
```

以上内容为相关的出入端声明。

```
9.     initial begin
10.         for( C1 = 0; C1 < 1024; C1 = C1 + 1'b1 )
11.             begin  RAM[ C1 ] <= 16'd0; end
12.         end
13.
14.     reg [15:0] RAM [1023:0];
```

以上内容为声明位宽为 16，深度为 1024 的 RAM，然后将其初始化。

```
15.     reg [10:0] C1 = 11'd0,C2 = 11'd0; // N+1
16.
17.     always @ ( posedge CLOCK or negedge RESET )
18.         if( !RESET )
19.             begin
20.                 C1 <= 11'd0;
21.             end
```

```

22.      else if( iEn[1] )
23.          begin
24.              RAM[ C1[9:0] ] <= iData;
25.              C1 <= C1 + 1'b1;
26.          end
27.
28.      always @ ( posedge CLOCK or negedge RESET )
29.          if( !RESET )
30.              begin
31.                  C2 <= 11'd0;
32.              end
33.          else if( iEn[0] )
34.              begin
35.                  //D1 <= RAM[ C2[9:0] ];
36.                  C2 <= C2 + 1'b1;
37.              end
38.

```

以上内容为核心内容。第 15 行声明写指针 C1，还有读指针 C2，位宽为 RAM 的深度 +1。第 17~26 行是 FIFO 的写操作，第 28~37 行是 FIFO 的读操作，笔者将第 35 行注释掉。

```

39.      assign oData = RAM[ C2[9:0]];
40.      assign oTag[1] = ( C1[10]^C2[10] & C1[9:0] == C2[9:0] ); // Full Left
41.      assign oTag[0] = ( C1 == C2 ); // Empty Right
42.
43.  endmodule

```

取而代之，笔者将 RAM 直接驱动 oData。好奇的朋友一定觉得疑惑？其实笔者是为了偷时钟，如果用 D1 驱动 oData，FIFO 的读数据（未来值）就会慢了半拍。所以，FIFO 与 SDRAM 功能模块之间会同步失败。反之，RAM 直接驱动输出，好比组合逻辑直接驱动输出，读取数据都是即时值。第 40~41 行是写满状态还有读空状态的输出驱动声明。

sdram_funcmod.v

```

1.  module sdram_funcmod
2.  (
3.      input CLOCK,
4.      input RESET,
5.
6.      output S_CKE, S_NCS, S_NRAS, S_NCAS, S_NWE,
7.      output [1:0]S_BA,
8.      output [12:0]S_A,

```

```
9.      output [1:0]S_DQM,
10.     inout [15:0]S_DQ,
11.
12.     output [1:0]oEn, // [1]Write [0]Read
13.     input [23:0]iAddr, // [23:22]BA,[21:9]Row,[8:0]Column
14.     input [15:0]iData,
15.     output [15:0]oData,
16.
17.     input [3:0]iCall,
18.     output oDone
19. );
```

以上内容为相关的出入端声明。

```
20.    parameter T100US = 14'd13300;
21.    // tRP 20ns, tRRC 63ns, tRCD 20ns, tMRD 2CLK, tWR/tDPL 2CLK, CAS Latency 3CLK
22.    parameter TRP = 14'd3, TRRC = 14'd9, TMRD = 14'd2, TRCD = 14'd3, TWR = 14'd2, CL = 14'd3;
23.    parameter _INIT = 5'b01111, _NOP = 5'b10111, _ACT = 5'b10011, _RD = 5'b10101, _WR = 5'b10100,
24.          _BSTP = 5'b10110, _PR = 5'b10010, _AR = 5'b10001, _LMR = 5'b10000;
25.
```

以上内容为相关的常量声明。

```
26.    reg [4:0]i;
27.    reg [13:0]C1;
28.    reg [15:0]D1;
29.    reg [4:0]rCMD;
30.    reg [1:0]rBA;
31.    reg [12:0]rA;
32.    reg [1:0]rDQM;
33.    reg [1:0]isEn;
34.    reg isOut;
35.    reg isDone;
36.
37.    always @ ( posedge CLOCK or negedge RESET )
38.        if( !RESET )
39.            begin
40.                i <= 4'd0;
41.                C1 <= 14'd0;
42.                D1 <= 16'd0;
43.                rCMD <= _NOP;
44.                rBA <= 2'b11;
45.                rA <= 13'h1fff;
```

```

46.          rDQM <= 2'b00;
47.          isEn <= 2'b00;
48.          isOut <= 1'b1;
49.          isDone <= 1'b0;
50.      end

```

以上内容为相关的寄存器声明与复位操作。

```

51.      else if( iCall[3] )
52.          case( i )
53.
54.              0: // Set IO to output Tag
55.              begin isOut <= 1'b1; i <= i + 1'b1; end
56.
57.              1: // Send Active Command with Bank and Row address
58.              begin rCMD <= _ACT; rBA <= iAddr[23:22]; rA <= iAddr[21:9]; i <= i + 1'b1; end
59.
60.              2: // wait TRCD 20ns
61.              if( C1 == TRCD -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
62.              else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
63.
64.              /*****
65.
66.              3:
67.              begin isEn[0] <= 1'b1; i <= i + 1'b1; end
68.
69.              4: // Send Write command with row address
70.              begin rCMD <= _WR; rBA <= iAddr[23:22]; rA <= { 4'b0000, iAddr[8:0] }; i <= i + 1'b1; end
71.
72.              5: // continue write until end and send BSTP
73.              begin
74.                  if( C1 == 512 -2 ) begin isEn[0] <= 1'b0; end
75.                  if( C1 == 512 -1 ) begin rCMD <= _BSTP; C1 <= 14'd0; i <= i + 1'b1; end
76.                  else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
77.              end
78.
79.              *****/
80.
81.              6: // Generate done signal
82.              begin rCMD <= _NOP; isDone <= 1'b1; i <= i + 1'b1; end
83.
84.              7:
85.              begin isDone <= 1'b0; i <= 4'd0; end

```

```
86.  
87.          endcase
```

以上内容为部分核心操作。以上内容是写操作，步骤 3 提前请求 FIFO，步骤 4 写入第 0 数据，步骤 5 写入第 1~511 数据，然后发送 BSTP 命令。步骤 6 发送 NOP 命令之余，也用来产生完成信号。

```
88.      else if( iCall[2] )  
89.          case( i )  
90.  
91.              0:  
92.                  begin isOut <= 1'b0; D <= 16'd0; i <= i + 1'b1; end  
93.  
94.              1: // Send Active command with Bank and Row address  
95.                  begin rCMD <= _ACT; rBA <= iAddr[23:22]; rA <= iAddr[21:9]; i <= i + 1'b1; end  
96.  
97.              2: // wait TRCD 20ns  
98.                  if( C1 == TRCD -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end  
99.                  else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end  
100.  
101.             /*****  
102.  
103.             3: // Send Read command and column address  
104.             begin rCMD <= _RD; rBA <= iAddr[23:22]; rA <= { 4'b0000, iAddr[8:0] }; i <= i + 1'b1; end  
105.  
106.             4: // wait CL 3 clock  
107.                 if( C1 == CL -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end  
108.                 else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end  
109.  
110.            /*****  
111.  
112.             5: // Read Data  
113.             begin  
114.                 D1 <= S_DQ; isEn[1] <= 1'b1;  
115.                 if( C1 == 512 -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end  
116.                 else begin C1 <= C1 + 1'b1; end  
117.             end  
118.  
119.            /*****  
120.  
121.             6:  
122.             begin isEn[1] <= 1'b0; rCMD <= _BSTP; i <= i + 1'b1; end  
123.
```

```

124.      *****/
125.
126.      7: // Generate done signal
127.      begin rCMD <= _NOP; isDone <= 1'b1; i <= i + 1'b1; end
128.
129.      8:
130.      begin isDone <= 1'b0; i <= 4'd0; end
131.
132.      endcase

```

以上内容为部分核心操作。以上内容是读操作。步骤 4 满足 CL 以后，步骤 5 读取并且发送 512 个数据给 FIFO。步骤 6，拉低 isEn[1]然后发送 BSTP 命令。步骤 7 发送 NOP 命令，然后产生完成信号。

```

133.      else if( iCall[1] )
134.          case( i )
135.
136.          0: // Send Precharge Command
137.          begin rCMD <= _PR; i <= i + 1'b1; end
138.
139.          1: // wait TRP 20ns
140.          if( C1 == TRP -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
141.          else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
142.
143.          2: // Send Auto Refresh Command
144.          begin rCMD <= _AR; i <= i + 1'b1; end
145.
146.          3: // wait TRRC 63ns
147.          if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
148.          else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
149.
150.          4: // Send Auto Refresh Command
151.          begin rCMD <= _AR; i <= i + 1'b1; end
152.
153.          5: // wait TRRC 63ns
154.          if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
155.          else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
156.
157.          *****/
158.
159.          6: // Generate done signal
160.          begin isDone <= 1'b1; i <= i + 1'b1; end
161.

```

```

162.          7:
163.          begin isDone <= 1'b0; i <= 4'd0; end
164.
165.      endcase

```

以上内容为部分核心操作。以上内容是刷新操作。

```

166.      else if( iCall[0] )
167.          case( i )
168.
169.              0: // delay 100us
170.              if( C1 == T100US -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
171.              else begin C1 <= C1 + 1'b1; end
172.
173.          /*****
174.
175.          1: // Send Precharge Command
176.          begin rCMD <= _PR; { rBA, rA } <= 15'h3fff; i <= i + 1'b1; end
177.
178.          2: // wait TRP 20ns
179.          if( C1 == TRP -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
180.          else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
181.
182.          3: // Send Auto Refresh Command
183.          begin rCMD <= _AR; i <= i + 1'b1; end
184.
185.          4: // wait TRRC 63ns
186.          if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
187.          else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
188.
189.          5: // Send Auto Refresh Command
190.          begin rCMD <= _AR; i <= i + 1'b1; end
191.
192.          6: // wait TRRC 63ns
193.          if( C1 == TRRC -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
194.          else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
195.
196.          *****/
197.
198.          7: // Send LMR Cmd. Burst Read & Write, 3'b010 mean CAS latency = 3, Sequential, 1 burst length
199.          begin rCMD <= _LMR; rBA <= 2'b11; rA <= { 3'd0, 1'b0, 2'd0, 3'b011, 1'b0, 3'b111 }; i <= i + 1'b1; end
200.
201.          8: // Send 2 nop CLK for tMRD

```

```

202.         if( C1 == TMRD -1 ) begin C1 <= 14'd0; i <= i + 1'b1; end
203.         else begin rCMD <= _NOP; C1 <= C1 + 1'b1; end
204.
205.         /***** */
206.
207.         9: // Generate done signal
208.         begin isDone <= 1'b1; i <= i + 1'b1; end
209.
210.         10:
211.         begin isDone <= 1'b0; i <= 4'd0; end
212.
213.     endcase
214.

```

以上内容为部分核心操作。以上内容是初始化 ,注意步骤 7 的设置内容 ,Burst Length 设置为 3'b111。

```

215.     assign { S_CKE, S_NCS, S_NRAS, S_NCAS, S_NWE } = rCMD;
216.     assign { S_BA, S_A } = { rBA, rA };
217.     assign S_DQM = rDQM;
218.     assign S_DQ = isOut ? iData : 16'hzzzz;
219.     assign oEn = isEn;
220.     assign oDone = isDone;
221.     assign oData = D1;
222.
223. endmodule

```

以上内容为相关的输出驱动。注意 , 第 218 行表示 FIFO 直接驱动 S_DQ 的输出。

sdram_ctrlmod.v

该控制模块不曾修改 , 所以笔者就不用重复粘贴了。

sdram_basemod.v

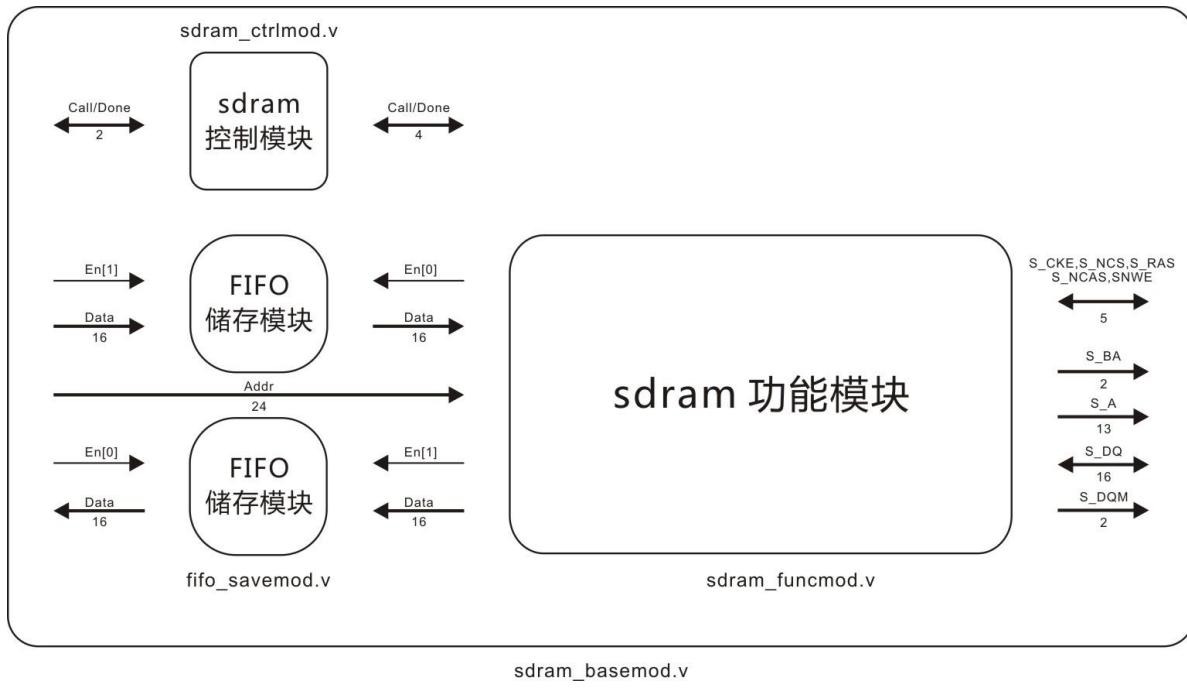


图 21.7 SDRAM 基础模块的建模图。

图 21.7 是 SDRAM 基础模块的建模图，其中控制模块只是负责上级调用，还有 SDRAM 功能模块的调用而已。SDRAM 功能模块的读出操作经由 FIFO 储存模块缓冲，然后再由上级读写数据。Addr 地址信号也是上级调用。注意，由于 SDRAM 基础模块的连线部署稍微复杂一点，为此图 21.7 稍微不遵守格式。

```

1. module sram_basemod
2. (
3.     input CLOCK,
4.     input RESET,
5.
6.     output S_CKE, S_NCS, S_NRAS, S_NCAS, S_NWE,
7.     output [1:0]S_BA,
8.     output [12:0]S_A,
9.     output [1:0]S_DQM,
10.    inout [15:0]S_DQ,
11.
12.    input [1:0]iEn,
13.    input [23:0]iAddr,
14.    input [15:0]iData,
15.    output [15:0]oData,
16.    output [1:0]oTag,
17.
18.    input [1:0]iCall,
19.    output [1:0]oDone
20. );

```

```

21.      wire [3:0]CallU1; // [3]Write, [2]Read, [1]A.Ref, [0]Initial
22.
23.      sdram_ctrlmod U1
24.      (
25.          .CLOCK( CLOCK ),
26.          .RESET( RESET ),
27.          .iCall( iCall ),           // < top , [1]Write [0]Read
28.          .oDone( oDone ),         // > top , [1]Write [0]Read
29.          .oCall( CallU1 ),        // > U4
30.          .iDone( DoneU4 )        // < U4
31.
32.      );
33.
34.      wire [15:0]DataU2;
35.
36.      fifo_savemod U2
37.      (
38.          .CLOCK( CLOCK ),
39.          .RESET( RESET ),
40.          .iEn( {iEn[1],EnU4[0]} ),    // < top
41.          .iData( iData ),           // < top
42.          .oData( DataU2 ),          // > top
43.          .oTag() //
44.      );
45.
46.      fifo_savemod U3
47.      (
48.          .CLOCK( CLOCK ),
49.          .RESET( RESET ),
50.          .iEn( {EnU4[1],iEn[0]} ),    // < U4 & top
51.          .iData( DataU4 ),           // < U4
52.          .oData( oData ),          // > top
53.          .oTag() //
54.      );
55.
56.      wire DoneU4;
57.      wire [1:0]EnU4;
58.      wire [15:0]DataU4;
59.
60.      sdram_funcmod U4
61.      (
62.          .CLOCK( CLOCK ),
63.          .RESET( RESET ),

```

```

64.      .S_CKE( S_CKE ),           // > top
65.      .S_NCS( S_NCS ),          // > top
66.      .S_NRAS( S_NRAS ),        // > top
67.      .S_NCAS( S_NCAS ),        // > top
68.      .S_NWE( S_NWE ),          // > top
69.      .S_BA( S_BA ),            // > top
70.      .S_A( S_A ),              // > top
71.      .S_DQM( S_DQM ),          // > top
72.      .S_DQ( S_DQ ),            // <> top
73.      .oEn( EnU4 ),             // > U2 && U3
74.      .iAddr( iAddr ),          // < top
75.      .iData( DataU2 ),          // < U2
76.      .oData( DataU4 ),          // > top
77.      .iCall( CallU1 ),          // < U1
78.      .oDone( DoneU4 )           // > U1
79.    );
80.
81. endmodule

```

该组合模块的连线部署完全遵照图 21.7。读者自己看着办吧。

sdram_demo.v

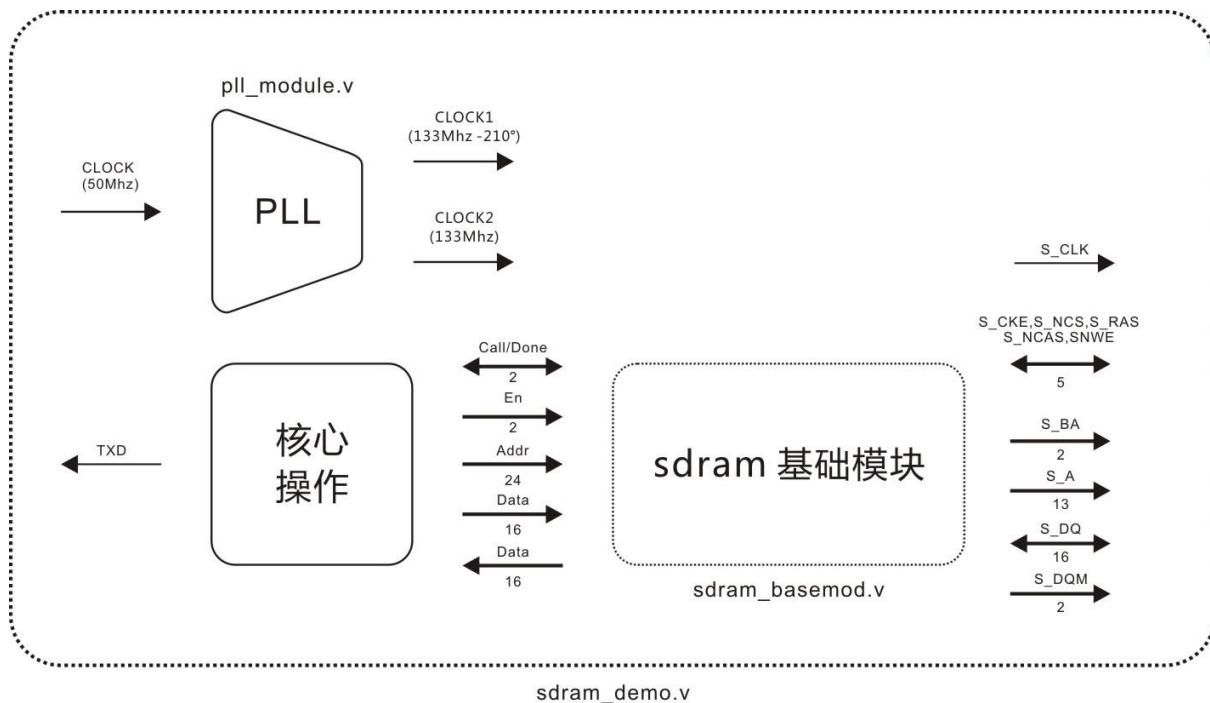


图 21.8 实验二十一的建模图。

图 21.8 是实验二十一的建模图，内容上的改变也只有 En 多出来而已，不过核心操作

的内容却有很大的改变。具体内容让我们来看代码吧。

```
1. module sdram_demo
2. (
3.     input CLOCK,
4.     input RESET,
5.     output S_CLK,
6.     output S_CKE, S_NCS, S_NRAS, S_NCAS, S_NWE,
7.     output [12:0]S_A,
8.     output [1:0]S_BA,
9.     output [1:0]S_DQM,
10.    inout [15:0]S_DQ,
11.    output TXD
12. );
```

以上内容为相关的出入端声明。

```
13.     wire CLOCK1,CLOCK2;
14.
15.     pll_module U1
16.     (
17.         .inclk0 ( CLOCK ),// 50Mhz
18.         .c0 ( CLOCK1 ), // 133Mhz -210 degree phase
19.         .c1 ( CLOCK2 ) // 133Mhz
20.     );
21.
```

以上内容为 PLL 模块的实例化。

```
22.     wire [1:0]DoneU2;
23.     wire [15:0]DataU2;
24.     wire [1:0]TagU2;
25.
26.     sdram_basemod U2
27.     (
28.         .CLOCK( CLOCK1 ),
29.         .RESET( RESET ),
30.         .S_CKE( S_CKE ),
31.         .S_NCS( S_NCS ),
32.         .S_NRAS( S_NRAS ),
33.         .S_NCAS( S_NCAS ),
34.         .S_NWE( S_NWE ),
35.         .S_A( S_A ),
```

```
36.          .S_BA( S_BA ),  
37.          .S_DQM( S_DQM ),  
38.          .S_DQ( S_DQ ),  
39.          .iEn( isEn ),  
40.          .iAddr( {D1, 9'd0} ),  
41.          .iData( D2 ),  
42.          .oData( DataU2 ),  
43.          .oTag( TagU2 ),  
44.          .iCall( isCall ),  
45.          .oDone( DoneU2 )  
46.      );  
47.
```

以上内容为 SDRAM 基础模块的实例化。

```
48. parameter B115K2 = 11'd1157, TXFUNC = 6'd16;  
49.  
50. reg [5:0]i,Go;  
51. reg [10:0]C1,C2;  
52. reg [14:0]D1;  
53. reg [15:0]D2,D3;  
54. reg [10:0]T;  
55. reg [1:0]isCall,isEn;  
56. reg rTXD;  
57.  
58. always @ ( posedge CLOCK1 or negedge RESET )  
59.     if( !RESET )  
60.         begin  
61.             i <= 6'd0;  
62.             Go <= 6'd0;  
63.             C1 <= 11'd0;  
64.             C2 <= 11'd0;  
65.             D1 <= 15'd0;  
66.             D2 <= 16'hA000;  
67.             D3 <= 16'd0;  
68.             T <= 11'd0;  
69.             isCall <= 2'b00;  
70.             isEn <= 2'b00;  
71.             rTXD <= 1'b1;  
72.         end  
73.     else
```

以上内容为相关的寄存器声明与复位操作。第 48 行是波特率为 115200 与伪函数入口的

实例化。注意，由于本实验是页读写，所以 `iAddr[8:0]` 基本作废，所以 D1 也只有 15 位宽而已。然后 `{ D1, 9'd0 }` 联合驱动 `iAddr`。

```

74.         case( i )
75.
76.             0:
77.                 begin isEn[1] <= 1'b1; i <= i + 1'b1; end
78.
79.             1:
80.                 begin isEn[1] <= 1'b0; i <= i + 1'b1; end
81.
82.             2:
83.                 if( C2 == 511 ) begin C2 <= 11'd0; i <= i + 1'b1; end
84.                 else begin D2[11:0] <= (D2[11:0] + 1'b1); C2 <= C2 + 1'b1; i <= 6'd0; end
85.
86.             3:
87.                 if( DoneU2[1] ) begin isCall[1] <= 1'b0; i <= i + 1'b1; end
88.                 else begin isCall[1] <= 1'b1; D1 <= 15'd0; end
89.
90.             4:
91.                 if( DoneU2[0] ) begin   isCall[0] <= 1'b0; i <= i + 1'b1; end
92.                 else begin isCall[0] <= 1'b1; D1 <= 15'd0; end
93.
94.             5:
95.                 begin isEn[0] <= 1'b1; i <= i + 1'b1; end
96.
97.             6:
98.                 begin D3 <= DataU2; isEn[0] <= 1'b0; i <= i + 1'b1; end
99.
100.            7:
101.                begin T <= { 2'b11, D3[15:8], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
102.
103.            8:
104.                begin T <= { 2'b11, D3[7:0], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
105.
106.            9:
107.                if( C2 == 24'd511 ) begin C2 <= 11'd0; i <= i + 1'b1; end
108.                else begin C2 <= C2 + 1'b1; i <= 6'd5; end
109.
110.            10:
111.                i <= i;
112.
113.                *****/

```

114.

以上内容为部分核心操作。步骤 0~2 是用来写满 FIFO。步骤 3 将 FIFO 的内容写入 SDRAM。步骤 4 又将内容读至 FIFO。步骤 5~9 从 FIFO 读出内容，并且发送出去，直至 512 个数据读完为止。步骤 10 发呆。

```
115.          16,17,18,19,20,21,22,23,24,25,26:  
116.          if( C1 == B115K2 -1 ) begin C1 <= 11'd0; i <= i + 1'b1; end  
117.          else begin rTXD <= T[i - 16]; C1 <= C1 + 1'b1; end  
118.  
119.          27:  
120.          i <= Go;  
121.  
122.          endcase  
123.  
124.          assign S_CLK = CLOCK2;  
125.          assign TXD = rTXD;  
126.  
127. endmodule
```

以上内容为部分核心操作。步骤 16~27 是发送一帧数据的伪函数。第 124~125 行是相关输出驱动声明。综合完毕并且下载程序，如果串口调试软件出现数据 A000~A1FF 表示实验成功。

细节一：完整的个体模块

本实验的 SDRAM 基础模块已经准备就绪。

实验二十二：SDRAM 模块⑤ — FIFO 读写

经过漫长的战斗以后，我们终于来到最后。对于普通人而言，页读写就是一名战士的墓碑（最终战役）… 然而，怕死的笔者想透过这个实验告诉读者，旅程的终点就是旅程的起点。一直以来，笔者都在烦恼“SDRAM 是否应该成为储存类？” SDRAM 作为一介储存资源（储存器），它的好处就是大容量空间，坏处则就是麻烦的控制规则，还有中规中矩的沟通速率。

相比之下，片上内存无论是控制的难度，还是沟通的速率，它都远远领先 SDRAM。俗语常说，愈是强力的资源愈是珍贵… 对此，片上内容的容量可谓稀罕的程度。实验二十二的要求非常单纯：

“请问如何建立基于 SDRAM 储存资源的 FIFO 存储模块呢？”，笔者问道。

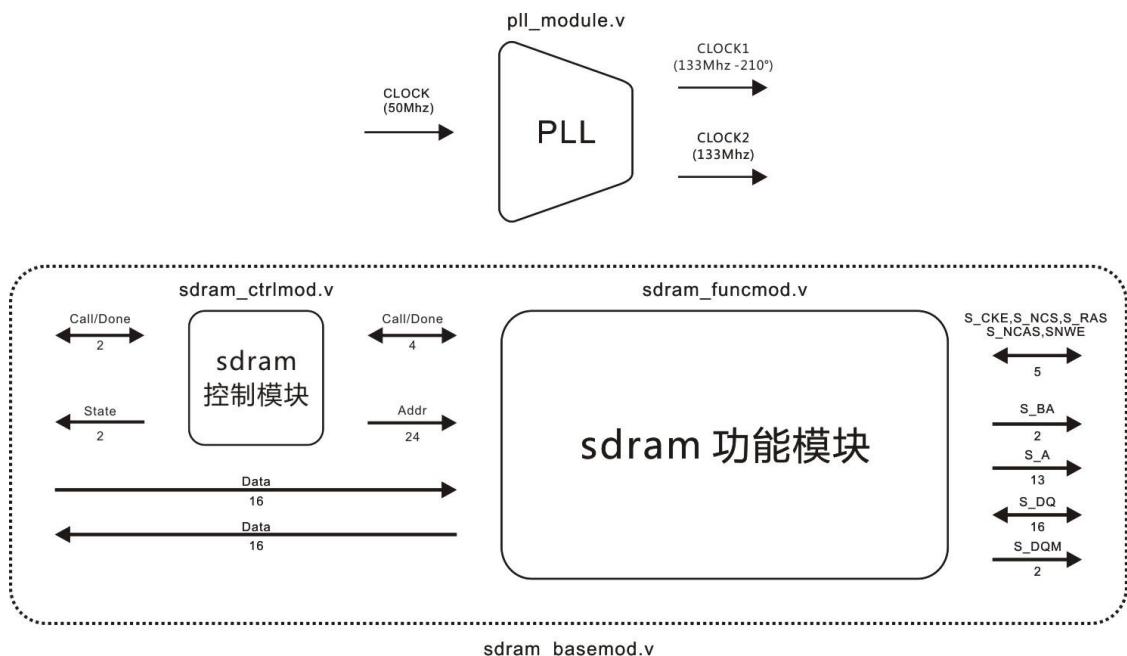


图 22.1 SDRAM 基础模块。

图 22.1 是基于实验十八修改而成的 SDRAM 基础模块，修改对象除了 SDRAM 控制模块以外，SDRAM 功能模块保持实验十八的状态，即单字读写。SDRAM 控制模块，除了多出 Tag 以外，Addr 的驱动也由该模块负责。具体的内容，让我们来看代码吧：

`sram_ctrlmod.v`

```
1. module sram_ctrlmod
2. (
3.     input CLOCK,
4.     input RESET,
```

```

5.      input [1:0]iCall, // [1]Write, [0]Read
6.      output [1:0]oDone,
7.      output [3:0]oCall,
8.      input iDone,
9.      output [23:0]oAddr,
10.     output [1:0]oTag
11.    );
12.    parameter WRITE = 4'd1, READ = 4'd4, REFRESH = 4'd7, INITIAL = 4'd8;
13.    parameter TREF = 11'd1040;
14.

```

以上内容为相关的出入端声明以及常量。其中多了 24 位宽的 oAddr 与 2 位宽的 oTag。

```

15.    reg [1:0]C7;
16.    reg [1:0]isDo;
17.
18.    always @ ( posedge CLOCK or negedge RESET ) // sub
19.      if( !RESET )
20.        begin
21.          C7 <= 2'b10;
22.          isDo <= 2'b00;
23.        end
24.      else
25.        begin
26.
27.          if( iCall[1] & C7[1] ) isDo[1] <= 1'b1;
28.          else if( iCall[0] & C7[0] ) isDo[0] <= 1'b1;
29.
30.          if( isDo[1] & isDone[1] ) isDo[1] <= 1'b0;
31.          else if( isDo[0] & isDone[0] ) isDo[0] <= 1'b0;
32.
33.          if( isDone ) C7 <= { isDo[0],isDo[1] };
34.          else if( iCall ) C7 <= { C7[0], C7[1] };
35.
36.        end
37.

```

以上内容为轮流协调的周边操作。具体内容与实验十七一样。

```

38.    reg [3:0]i;
39.    reg [10:0]C1;
40.    reg [3:0]isCall; // [3]Write [2]Read [1]A.Refresh [0]Initial
41.    reg [1:0]isDone;

```

```

42.      reg [23:0]D1;
43.      reg [24:0]C2,C3; // N + 1
44.
45.      always @ ( posedge CLOCK or negedge RESET )// core
46.          if( !RESET )
47.              begin
48.                  i <= INITIAL;           // Initial SDRAM at first
49.                  C1 <= 11'd0;
50.                  isCall <= 4'b0000;
51.                  isDone <= 2'b00;
52.                  D1 <= 24'd0;
53.                  C2 <= 25'd0;
54.                  C3 <= 25'd0;
55.              end

```

以上内容为相关的寄存器声明与复位操作。其中 C2 是写指针，C3 是读指针，位宽为 oAddr + 1。D 用来驱动 oAddr。

```

56.      else
57.          case( i )
58.
59.              0: // IDLE
60.                  if( C1 >= TREF ) begin C1 <= 11'd0; i <= REFRESH; end
61.                  else if( isDo[1] ) begin C1 <= C1 + 1'b1; i <= WRITE; end
62.                  else if( isDo[0] ) begin C1 <= C1 + 1'b1; i <= READ; end
63.                  else begin C1 <= C1 + 1'b1; end
64.
65.                  *****/
66.

```

以上内容为部分核心内容。步骤 0 是待机状态，其中 61~62 行改为 isDo[1] 与 isDo[2]。

```

67.          1: //Write
68.          if( iDone ) begin isCall[3] <= 1'b0; C1 <= C1 + 1'b1; i <= i + 1'b1; end
69.          else begin isCall[3] <= 1'b1; D1 <= C2[23:0]; C1 <= C1 + 1'b1; end
70.
71.          2:
72.          begin C2 <= C2 + 1'b1; isDone[1] <= 1'b1; C1 <= C1 + 1'b1; i <= i + 1'b1; end
73.
74.          3:
75.          begin isDone[1] <= 1'b0; C1 <= C1 + 1'b1; i <= 4'd0; end
76.
77.          *****/

```

78.

以上内容为部分核心内容。步骤 1~3 是写操作，步骤 1 将 C2[23:0] 的内容赋值 D。步骤 2~3 产生完成信号之余也递增 C2。

```
79.          4: // Read
80.          if( iDone ) begin isCall[2] <= 1'b0; C1 <= C1 + 1'b1; i <= i + 1'b1; end
81.          else begin isCall[2] <= 1'b1; D1 <= C3[23:0]; C1 <= C1 + 1'b1; end
82.
83.          5:
84.          begin C3 <= C3 + 1'b1; isDone[0] <= 1'b1; C1 <= C1 + 1'b1; i <= i + 1'b1; end
85.
86.          6:
87.          begin isDone[0] <= 1'b0; C1 <= C1 + 1'b1; i <= 4'd0; end
88.
89.          *****/
90.
```

以上内容为部分核心内容。步骤 4~6 是写操作，步骤 4 将 C3[23:0] 的内容赋值 D。步骤 5~7 产生完成信号之余也递增 C3。

```
91.          7: // Auto Refresh
92.          if( iDone ) begin isCall[1] <= 1'b0; i <= 4'd0; end
93.          else begin isCall[1] <= 1'b1; end
94.
95.          *****/
96.
97.          8: // Initial
98.          if( iDone ) begin isCall[0] <= 1'b0; i <= 4'd0; end
99.          else begin isCall[0] <= 1'b1; end
100.
101.         endcase
102.
```

以上内容为部分核心内容。步骤 7~8 保持不变。

```
103.     assign oDone = isDone;
104.     assign oCall = isCall;
105.     assign oAddr = D1;
106.     assign oTag[1] = ( C2[24]^C3[24] & C2[23:0] == C3[23:0] );
107.     assign oTag[0] = ( C2 == C3 );
108.
109. endmodule
```

以上内容为相关的输出驱动。D1 驱动 oAddr，第 106~107 行是写满状态与读空状态的声明。

sdram_funcmod.v

该功能模块与实验十八的内容一模一样。

sdram_demo.v

该组合模块的连线部署完全参照图 22.1。

```
1. module sdram_basemod
2. (
3.     input CLOCK,
4.     input RESET,
5.
6.     output S_CKE, S_NCS, S_NRAS, S_NCAS, S_NWE,
7.     output [1:0]S_BA,
8.     output [12:0]S_A,
9.     output [1:0]S_DQM,
10.    inout [15:0]S_DQ,
11.
12.    input [1:0]iCall,
13.    output [1:0]oDone,
14.    output [1:0]oTag,
15.    input [15:0]iData,
16.    output [15:0]oData
17. );
18. wire [3:0]CallU1; // [3]Refresh, [2]Read, [1]Write, [0]Initial
19. wire [23:0]AddrU2;
20.
21. sdram_ctrlmod U1
22. (
23.     .CLOCK( CLOCK ),
24.     .RESET( RESET ),
25.     .iCall( iCall ),      // < top ,[1]Write [0]Read
26.     .oDone( oDone ),      // > top ,[1]Write [0]Read
27.     .oAddr( AddrU2 ),      // > U2
28.     .oTag( oTag ),      // > top
29.     .oCall( CallU1 ),      // > U2
30.     .iDone( DoneU2 )      // < U2
```

```

31.      );
32.
33.      wire DoneU2;
34.
35.      sdram_funcmod U2
36.      (
37.          .CLOCK( CLOCK ),
38.          .RESET( RESET ),
39.          .S_CKE( S_CKE ),           // > top
40.          .S_NCS( S_NCS ),           // > top
41.          .S_NRAS( S_NRAS ),        // > top
42.          .S_NCAS( S_NCAS ),        // > top
43.          .S_NWE( S_NWE ),           // > top
44.          .S_BA( S_BA ),            // > top
45.          .S_A( S_A ),              // > top
46.          .S_DQM( S_DQM ),           // > top
47.          .S_DQ( S_DQ ),             // < top
48.          .iCall( CallU1 ),          // < U1
49.          .oDone( DoneU2 ),          // > U1
50.          .iAddr( AddrU2 ),          // < U1
51.          .iData( iData ),           // < top
52.          .oData( oData )            // > top
53.      );
54.
55. endmodule

```

连线内容请自己看着办吧。

sdram_demo.v

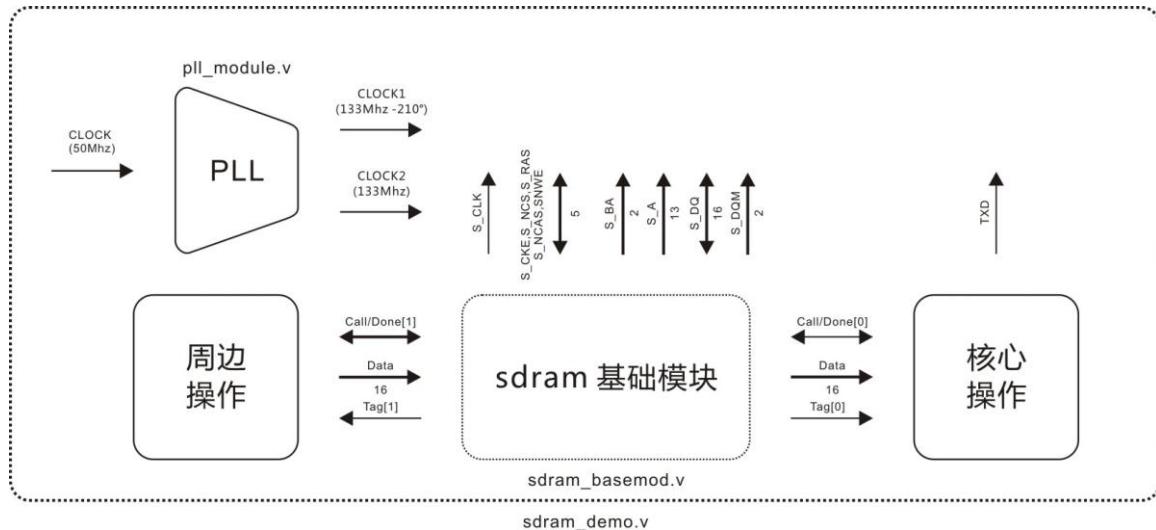


图 22.3 实验二十二的建模图。

图 22.3 是实验二十二的建模图，左边的周边操作负责写入数据，右边的核心操作负责读取数据并且经由 TXD 发送出去。具体内容我们还是来看代码吧。

```
1. module sdram_demo
2. (
3.     input CLOCK,
4.     input RESET,
5.     output S_CLK,
6.     output S_CKE, S_NCS, S_NRAS, S_NCAS, S_NWE,
7.     output [12:0]S_A,
8.     output [1:0]S_BA,
9.     output [1:0]S_DQM,
10.    inout [15:0]S_DQ,
11.    output TXD
12. );
```

以上内容为相关的出入端声明。

```
13.     wire CLOCK1,CLOCK2;
14.
15.     pll_module U1
16.     (
17.         .inclk0( CLOCK ), // 50Mhz
18.         .c0( CLOCK1 ), // 133Mhz -210 degree phase
19.         .c1( CLOCK2 ) // 133Mhz
20.     );
21.
```

以上内容为 PLL 模块的实例化。

```
22.     wire [1:0]DoneU2;
23.     wire [15:0]DataU2;
24.     wire [1:0]TagU2;
25.
26.     sdram_basemod U2
27.     (
28.         .CLOCK( CLOCK1 ),
29.         .RESET( RESET ),
30.         .S_CKE( S_CKE ),
31.         .S_NCS( S_NCS ),
32.         .S_NRAS( S_NRAS ),
```

```

33.      .S_NCAS( S_NCAS ),
34.      .S_NWE( S_NWE ),
35.      .S_A( S_A ),
36.      .S_BA( S_BA ),
37.      .S_DQM( S_DQM ),
38.      .S_DQ( S_DQ ),
39.      .iCall( {isWR,isRD} ),
40.      .oDone( DoneU2 ),
41.      .iData( D2 ),
42.      .oData( DataU2 ),
43.      .oTag( TagU2 )
44. );
45.

```

以上内容为 SDRAM 基础模块的实例化，注意第 39 行的 iCall 是由 isWR 与 isRD 联合驱动。此外，第 43 行也多了 oTag。

```

46.      reg [5:0]i;
47.      reg [15:0]D2;
48.      reg isWR;
49.
50.      always @ ( posedge CLOCK1 or negedge RESET )
51.          if( !RESET )
52.              begin
53.                  i <= 6'd0;
54.                  D2 <= 16'hA000;
55.                  isWR <= 1'b0;
56.              end
57.          else
58.              case( i )
59.
60.                  0:
61.                      if( !TagU2[1] ) i <= i + 1'b1;
62.
63.                  1:
64.                      if( DoneU2[1] ) begin isWR <= 1'b0; i <= i + 1'b1; end
65.                      else begin isWR <= 1'b1; end
66.
67.                  2:
68.                      if( D2 == 16'hA1FF ) i <= i + 1'b1;
69.                      else begin D2[11:0] <= D2[11:0] + 1'b1; i <= 6'd0; end
70.
71.                  3:

```

```
72.          i <= i;  
73.  
74.      endcase  
75.
```

以上内容为写作用的周边操作。步骤 0 检测是否写满，步骤 1 写入数据，步骤 2 判断是否写满 512 次，不是的话就递增内容。步骤 3 是写完发呆。

```
76.      reg [5:0]j,Go;  
77.      reg [10:0]C1;  
78.      reg [15:0]D3;  
79.      reg [10:0]T;  
80.      reg isRD;  
81.      reg rTXD;  
82.  
83.      parameter B115K2 = 11'd1157, TXFUNC = 6'd16;  
84.  
85.      always @ ( posedge CLOCK1 or negedge RESET )  
86.          if( !RESET )  
87.              begin  
88.                  j <= 6'd0;  
89.                  Go <= 6'd0;  
90.                  C1 <= 11'd0;  
91.                  D3 <= 16'd0;  
92.                  T <= 11'd0;  
93.                  isRD <= 1'b0;  
94.                  rTXD <= 1'b1;  
95.              end  
96.          else  
97.              case(j )  
98.  
99.                  0:  
100.                     if( !TagU2[0] ) j <= j + 1'b1;  
101.  
102.                  1:  
103.                     if( DoneU2[0] ) begin D3 <= DataU2; isRD <= 1'b0; j <= j + 1'b1; end  
104.                     else begin isRD <= 1'b1; end  
105.  
106.                  2:  
107.                     begin T <= { 2'b11, D3[15:8], 1'b0 }; j <= TXFUNC; Go <= j + 1'b1; end  
108.  
109.                  3:  
110.                     begin T <= { 2'b11, D3[7:0], 1'b0 }; j <= TXFUNC; Go <= j + 1'b1; end
```

```

111.
112.        4:
113.        if( D3 == 16'hA1FF ) j <= j + 1'b1;
114.        else j <= 6'd0;
115.
116.        5:
117.        j <= j;
118.
119.        ****
120.

```

以上内容为部分核心操作。步骤 0 检测是否读空，步骤 1 读出数据，步骤 2~3 将数据发送数据，步骤 4 判断是否执行 512 次？如果不是的话就返回步骤 0，是的话就递增 i 进入发呆的步骤 5。

```

121.        16,17,18,19,20,21,22,23,24,25,26:
122.        if( C1 == B115K2 -1 ) begin C1 <= 11'd0; j <= j + 1'b1; end
123.        else begin rTXD <= T[j - 16]; C1 <= C1 + 1'b1; end
124.
125.        27:
126.        j <= Go;
127.
128.        endcase
129.
130.        assign S_CLK = CLOCK2;
131.        assign TXD = rTXD;
132.
133. endmodule

```

以上内容为核心操作以及输出驱动。综合完毕便下载程序，如果串口调试软件出现 A000~A1FF，表示实验成功。

细节一：完整的个体模块

本实验的 SDRAM 基础模块已经准备就绪。

细节二：小谈储存类

FIFO 机制的 SDRAM 很可能实用性不强。相对低级建模 II 而言，SDRAM 已经脱离死板的印象，任何畸形的储存方式，都有可能实现在任何储存资源之上。举例而言，片上内存储存资源可以实现 FIFO 功能，IIC 储存资源也可以实现 FIFO 功能，SDRAM 储存资源也可以实现 FIFO 功能。

如果按照这条思路逆向思考的话，如果 SDRAM 储存资源可以实现功能 C，那么 IIC 储存资源还有片上内存储存资源同样也可以实现功能 C。如此一来，建模的自由度会大大增高，喜爱建模的孩子也会开心至极。如果读者是变态，那么一块小小的 SDRAM 储存资源实现多功能读写如：单字读写，多字读写，页读写，甚至 FIFO 读写，集于一身是有可能的。不管怎么样，实验二十二所要表达的信息已经非常清晰，即储存类的伸缩性与可塑性。

实验二十三：DS1302 模块

DS1302 这只硬件虽然曾在《建模篇》介绍过，所以重复的内容请怒笔者懒得唠叨了，笔者尽可以一笑带过，废话少说让我们进入正题吧。DS1302 是执行事实时钟(Real Time Clock)的硬件，采用 SPI 传输。

表示 23.1 访问 (地址) 字节。

[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
1	A5	A4	A3	A2	A1	A0	R/W

DS1302 作为从机任由主机蹂躏 ... 啊，是任由主机访问才对。对此，访问便有方向之分。如表 23.1 所示，访问字节 (地址字节) [0] 为访问方向 1 读 0 写。[6..1] 为地址。[7] 为常量 1。除了访问字节以外，DS1302 也有数据字节。

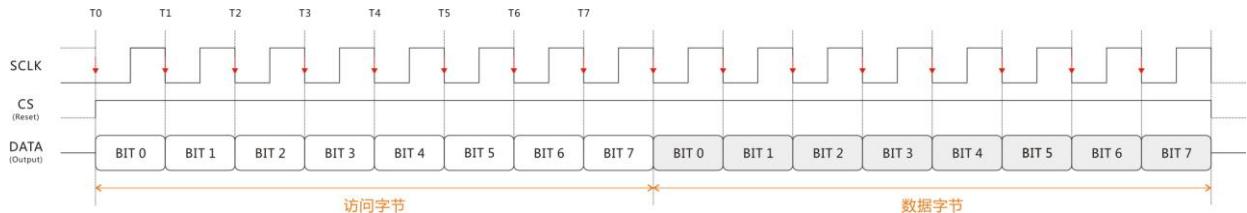


图 23.1 写操作的理想时序 (主机视角)。

图 23.1 是写操作的理想时序图，SCLK 为串行时钟，CS 为拉高有效的片选 (又名为 RESET)，DATA 是数据进出的 I/O。忘了说，DS1302 由于使用 SPI 传输的关系，所以下降沿设置数据，上升沿锁存数据。如图 23.1 所示，左边为访问字节，右边则是数据字节，CS 拉高为写操作有效。对此，主机先将访问字节写入，再将数据字节写入指定的位置。闲置状态，SCLK 信号还有 CS 信号都是拉低发呆，而且读写数据都是由低至高。

至于 Verilog 则可以这样描述，结果如代码 23.1 所示：

```
1. 0:
2. begin { rRST,rSCLK } <= 2'b10; T <= iAddr; i <= FF_Write; Go <= i + 1'b1; end
3. 1:
4. begin T <= iData; i <= FF_Write; Go <= i + 1'b1; end
5. 2:
6. begin { rRST,rSCLK } <= 2'b00; i <= i + 1'b1; end
7. ...
8. 16,17,18,19,20,21,22,23:
9. begin
10.    isQ = 1'b1;
11.    rSIO <= T[i-16];
12.    if( C1 == 0 ) rSCLK <= 1'b0;
13.    else if( C1 == FHALF ) rSCLK <= 1'b1;
```

```

14.      if( C1 == FCLK -1) begin C1 <= 6'd0; i <= i + 1'b1; end
15.      else C1 <= C1 + 1'b1;
16.  end
17.  24:
18.  i <= Go;

```

代码 23.1

步骤 0 拉高片选，拉低时钟，准备访问字节，然后进入伪函数。步骤 1 准备数据字节，然后进入伪函数。步骤 2 拉低使能，拉低时钟。

步骤 16~23 为写入一个字节的伪函数，isQ 为 IO 的输出控制，rSIO 为 DATA 的输出驱动，rSCLK 为 SCLK 的输出驱动，FCLK 为一个时钟周期，FHALF 为半周期。写操作只要任由 T 全程驱动 rSIO 即可，期间 C1 为 0 拉低时钟，C1 为半个周期便拉高时钟。步骤 24 则返回步骤。

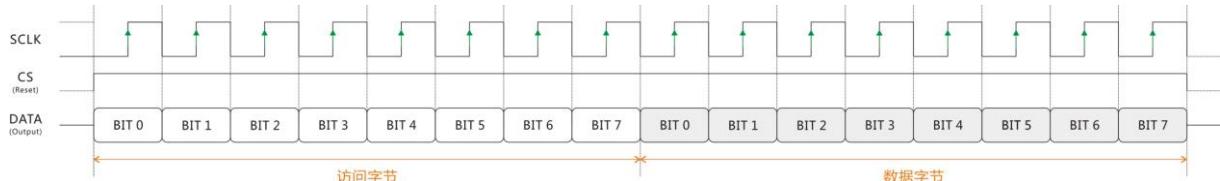


图 23.2 写操作的理想时序 (从机视角)。

图 23.2 则是从机视角的写操作时序，从机任何时候都是利用上升沿读取数据。

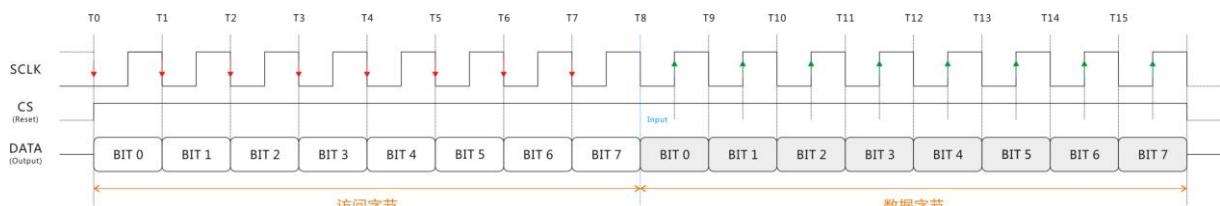


图 23.3 读操作的理想时序 (主机视角)。

图 23.3 为读操作的理想时序。T0~T7，主机写入访问字节并且指定读出地址。T8~T15，从机读出数据字节，期间 DATA 为输入状态，从机根据下降沿设置 (更新) 数据，主机为上升沿读取。至于 Verilog 则可以这样描述，结果如代码 23.2 所示：

```

1. 0 :
2. begin { rRST,rSCLK } <= 2'b10; T <= iAddr; i <= FF_Write; Go <= i + 1'b1; end

3. 1:
4. begin i <= FF_Read; Go <= i + 1'b1; end
5. 2:
6. begin { rRST,rSCLK } <= 2'b00; D1 <= T; i <= i + 1'b1; end
7. ...

```

```

8. 16,17,18,19,20,21,22,23:
9. begin
10.    isQ = 1'b1;
11.    rSIO <= T[i-16];
12.    if( C1 == 0 ) rSCLK <= 1'b0;
13.    else if( C1 == FHALF ) rSCLK <= 1'b1;
14.    if( C1 == FCLK -1) begin C1 <= 6'd0; i <= i + 1'b1; end
15.    else C1 <= C1 + 1'b1;
16. end
17. 24:
18. i <= Go;
19. ...
20. 32,33,34,35,36,37,38,39:
21. begin
22.    isQ = 1'b0;
23.    if( C1 == 0 ) rSCLK <= 1'b0;
24.    else if( C1 == FHALF ) begin rSCLK <= 1'b1; T[i-32] <= RTC_DATA; end
25.    if( C1 == FCLK -1) begin C1 <= 6'd0; i <= i + 1'b1; end
26.    else C1 <= C1 + 1'b1;
27. end
28. 40:
29. i <= Go;

```

代码 23.2

步骤 0 拉低使能，拉低时钟，准备访问字节，进入伪函数写。步骤 1 准备读数据，进入为函数读。步骤 2 为读取数据，拉低使能，拉低时钟。步骤 16~24 为协议一个字节的伪函数，步骤 32~40 为读取一个字节的伪函数。拉低 isQ 让 IO 为输入状态，C1 为 0 拉低 rSCLK，C1 为半个周期拉高 rSCLK 并且读取数据，上述操作重复 8 次便搞定。

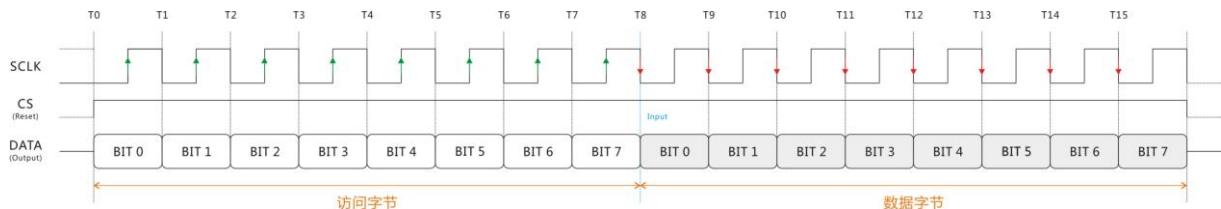


图 23.4 读操作的理想时序 (从机视角)。

图 23.4 为从机视角的读操作，从机在 T0~T7 利用上升沿读取数据，然后在 T8~T15 利用下降沿输出数据。

表 23.2 访问内容/寄存器内容。

访问地址		寄存器内容									
读	写	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	范围	
81H	80H	CH	秒十位				秒个位				59~00

83H	82H		分十位			分个位			59~00
85H	84H	12/24		时十位		时个位			12/24~00
87H	86H			日十位		日个位			31~1
89H	88H				月十位	月个位			12~1
8BH	8AH					天			7~1
8DH	8CH	年十位			年个位				99~00
8FH	8EH	读保护							
C1H	C0H								FFH~00H
...
FDH	FCH								FFH~00H

表 23.2 为 DS1302 的访问内容 , 或者说为寄存器地址与寄存器内容 , 其中秒寄存器的[7]为 0 开始计时 , 为 1 则停止计时。调用过程大致如下 :

初始化 :

- 发送 8EH 访问字节 , 关闭写保护 ;
- 发送 84H 访问字节 , 初始化 “时种” ;
- 发送 82H 访问字节 , 初始化 “分钟” ;
- 发送 80H 访问字节 , 初始化 “秒钟” , 开始计时。

调用 :

- 发送 81H 访问字节 , 读取 “秒钟” 内容 ;
- 发送 83H 访问字节 , 读取 “分钟” 内容 ;
- 发送 85H 访问字节 , 读取 “时种” 内容 ;
- 重复上述内容。

至于详细过程还有具体的寄存器内容 , 笔者已在《建模篇》解释过 , 所以读者自己看着办吧。接下来 , 让我们进入本实验的重点内容吧。

表 32.3 DS1302 的时序参数①。

时序参数	标示	最小		最大	
		时间	时钟 (50Mhz)	时间	时钟 (50Mhz)
Clock Frequency	FCLK			2Mhz	25
Clock High Time	TCH	250ns	12.5		
Clock Low Time	TCL	250ns	12.5		
Clock Rise and Fall	TR,TF	0ns	0	500ns	25

首先让我们先来瞧瞧相关的时序参数。表 23.3 为速率为 2Mhz 的时序参数。DS1302 最高速率为 2Mhz 并且无下限 , 50Mhz 的量化结果为 25。时钟信号拉高 TCH 或拉低 TCL 至少需要保持 250ns , 量化结果为 12.5。至于时钟信号上山 TR 或者下山 TF 最大时间为 500ns , 极端说是最长时间是 0ns。

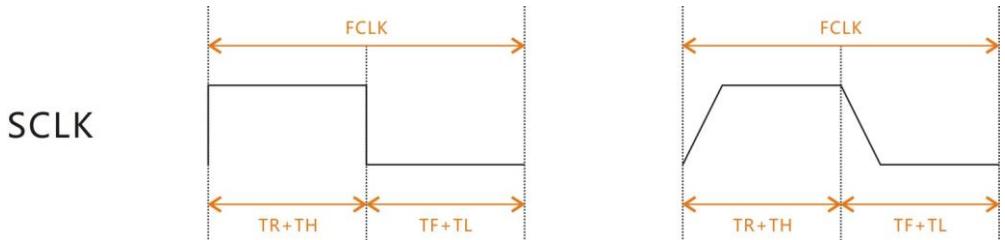


图 23.5 时序参数①。

如图 23.5 所示 ,那是时钟信号还有相关的时序参数 ,左图为理想时序 ,右图为物理时序。TR+TH 造就前半时钟周期 ,TF+TL 造就后半时钟周期 ,然后 TR+TH+TF+TL 为一个时钟周期。

表 23.4 DS1302 的时序参数②。

时序参数	标示	最小		最大	
		时间	时钟 (50Mhz)	时间	时钟 (50Mhz)
CE to Clock Setup	TCC	1us	50		
Data to Clock Setup	TDC	50ns	2.5		

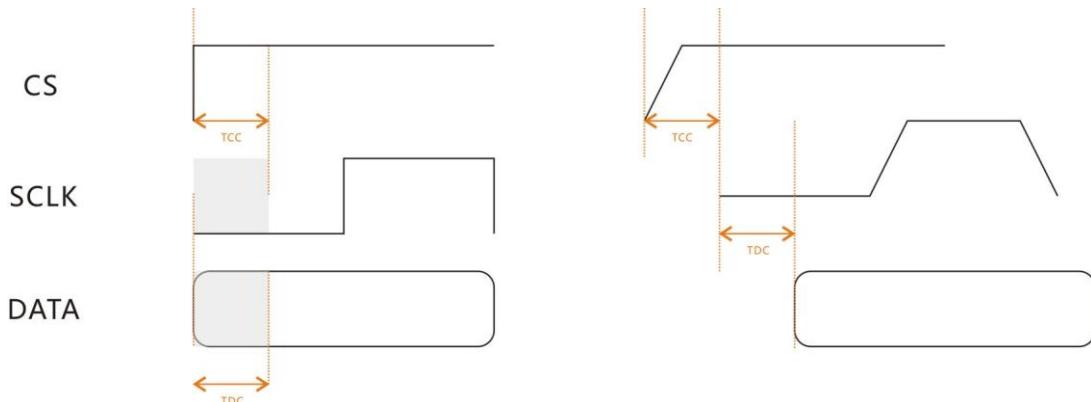


图 23.6 时序参数②。

如表 23.4 是 TCC 还有 TDC 的时序参数 ,虽然两者都有 Setup 字眼 ,实际上它们都是推挤作用的时序参数。如图 23.6 的右图所示 ,那是物理时序图 ,TCC 向右推挤 SCLK 信号 ,TDC 向右推挤 TDC 信号。换做左图的理想时序图 ,TCC 使 SCLK 被覆盖 ,TDC 使 DATA 被覆盖。有些同学可能会被 TCC 的要求吓到 ,既然是 1us。原理上 ,它是时钟周期的两倍 ,不过那是推挤作用的时序参数 ,只要操作结束之前一直拉高 CS 即可无视。

表 23.4 DS1302 的时序参数③。

时序参数	标示	最小		最大	
		时间	时钟 (50Mhz)	时间	时钟 (50Mhz)
CLOCK to Data Delay	TCDD	200ns	10		
Clock to Data Hold	TCDH	70ns	3.5		

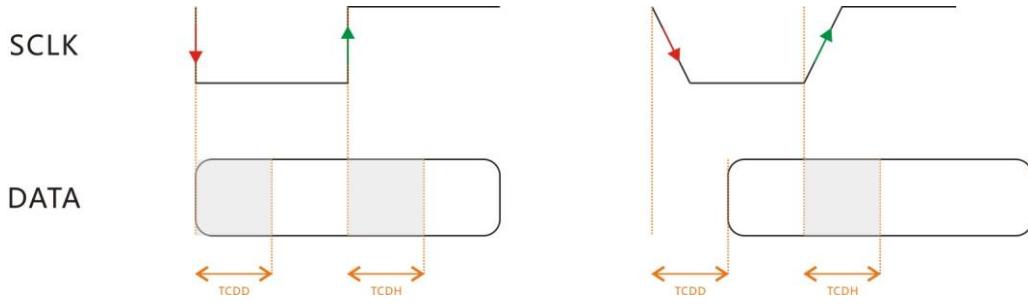


图 23.7 时序参数③。

表 23.4 显示 TCDD 还有 TCDH 两只时序参数，如 23.7 的右图所示，那是物理时序。我们知道 DS1302 是下降沿设置（输出）数据，上升沿锁存数据，期间 TCDD 将数据向右推挤，TCDH 则就是典型的 THold，即锁存数据以后的确保时间。如 23.7 的左图所示，那是理想时序，由于这两只都是小家伙，一般都无法完全覆盖数据。

表 23.5 DS1302 的时序参数④

时序参数	标示	最小		最大	
		时间	时钟 (50Mhz)	时间	时钟 (50Mhz)
CE Inactive Time	TCWH	1us	50		
CE to I/O High Impedance	TCDZ	70ns	3.5		
SCLK to I/O High Impedance	TCCZ	70ns	3.5		

表 23.5 有 3 个比较奇怪的时序参数，TCWH 为片选信号进入静态所需的时间，也认为是释放片选所需的最长时间。至于 TCDZ 与 TCCZ 都是与高阻态有关的时序参数，而高阻态也与 I/O 息息相关。感觉上，高阻态好比一只切断输出的大刀，而且这只大刀必须由人操纵，其中 TCDZ 就是 CE 切断输出状态所需的最长时间，TCCZ 就是 SCLK 切断输出状态所需的最长时间。

具体而言，如代码代码 23.3 所示：

```
module (...)

...
input Call,
 inout SIO,
 ...
assign SIO = !Call ? D1 : 1' bz;
endmodule
```

代码 23.3

代码 23.3 告诉我们，SIO 由 D1 驱动输出，而且 Call 拉高便将 SIO 的输出切断。假设 TSSZ (Call to I/O High Impedance) 是 Call 切断输出所需的时间 ... 细心观察，Call 一旦拉低，SIO 并不会立即输出高阻态，则是必须经历一段时间。至于高阻态的大小姨妈，一般都是硬件内部（从机）的纠纷，鲜少与驱动方（主机）扯上关系，所以我们在此聊聊就好。

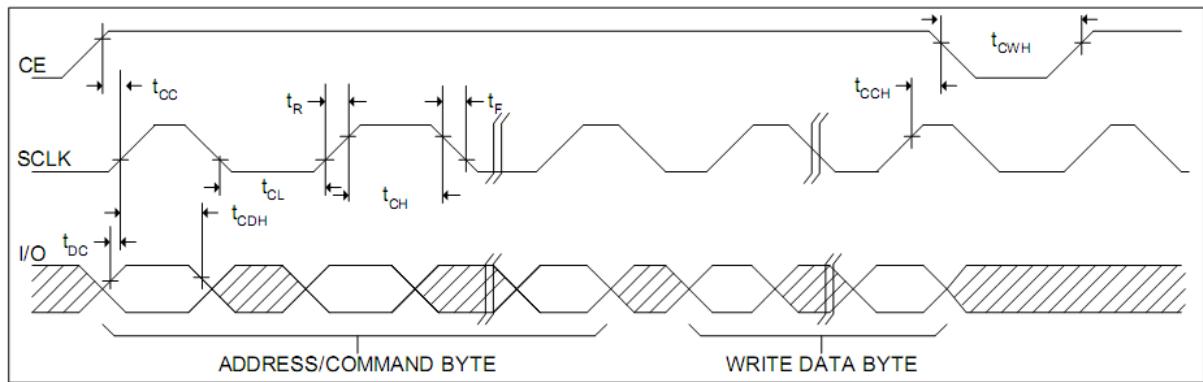


图 23.8 写操作的物理时序 (时序参数)。

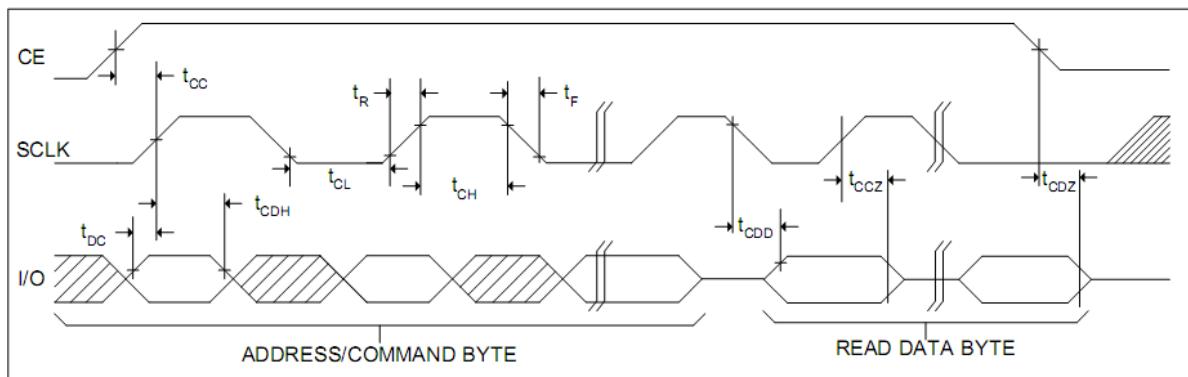


图 23.9 读操作的物理时序 (时序参数)。

最后附上两张官方的原图 23.8 还有图 23.9 作为本说明的谢幕，读者就自己慢慢看着办吧。理解完毕，我们便可以开始建模了。

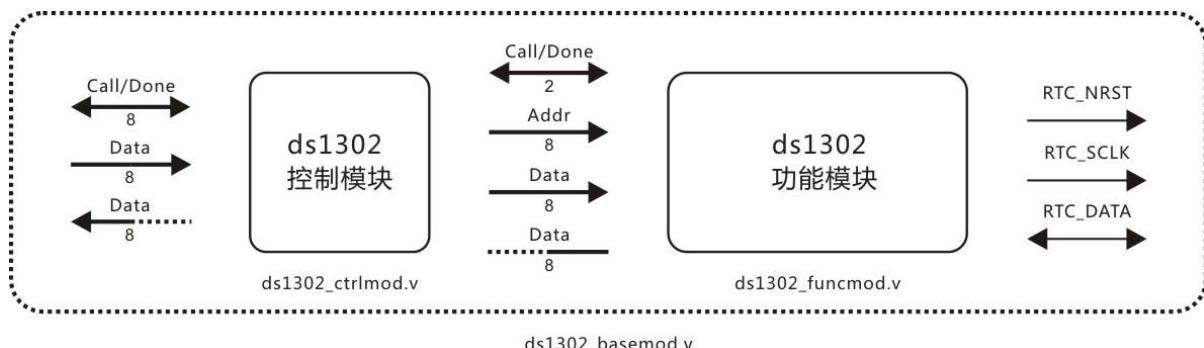


图 23.10 DS1302 基础模块的建模图。

图 23.10 是 DS1302 基础模块的建模图，内容包含控制模块与功能模块，功能模块负责最基础的读写操作，控制模块则负责功能调度，准备访问字节等任务。换之，功能模块的右边是驱动硬件资源的顶层信号。其中，功能模块的 oData 穿过控制模块直接驱动外围。

ds1302_funcmod.v



图 23.11 DS1302 功能模块的建模图。

图 23.11 是 DS1302 功能模块的建模图，Call/Done 位宽为 2，其中 [1] 为写操作，[0] 为读操作。具体内容让我们来看代码吧：

```
1. module ds1302_funcmod
2. (
3.     input CLOCK, RESET,
4.     output RTC_NRST,RTC_SCLK,
5.     inout RTC_DATA,
6.     input [1:0]iCall,
7.     output oDone,
8.     input [7:0]iAddr,iData,
9.     output [7:0]oData
10. );
```

以上内容为相关的出入端声明。

```
11. parameter FCLK = 6'd25, FHALF = 6'd12; // 2Mhz,(1/2Mhz)/(1/50Mhz)
12. parameter FF_Write = 6'd16, FF_Read = 6'd32;
13.
```

以上内容为伪函数入口地址以及常量声明。FCLK 为一个周期，FHALF 为半周期。

```
14. reg [5:0]C1;
15. reg [5:0]i,Go;
16. reg [7:0]D1,T;
17. reg rRST, rSCLK, rSIO;
18. reg isQ,isDone;
19.
20. always @ ( posedge CLOCK or negedge RESET )
21.     if( !RESET )
```

```

22.           begin
23.             C1 <= 6'd0;
24.             { i,Go } <= { 6'd0,6'd0 };
25.             { D1,T } <= { 8'd0,8'd0 };
26.             { rRST, rSCLK, rSIO } <= 3'b000;
27.             { isQ, isDone } <= 2'b00;
28.           end

```

以上内容为相关的寄存器还有复位操作。D1 为暂存读取结果，T 为伪函数的操作空间，isQ 为 IO 的控制输出。

```

29.       else if( iCall[1] )
30.         case( i )
31.
32.           0:
33.             begin { rRST,rSCLK } <= 2'b10; T <= iAddr; i <= FF_Write; Go <= i + 1'b1; end
34.
35.           1:
36.             begin T <= iData; i <= FF_Write; Go <= i + 1'b1; end
37.
38.           2:
39.             begin { rRST,rSCLK } <= 2'b00; i <= i + 1'b1; end
40.
41.           3:
42.             begin isDone <= 1'b1; i <= i + 1'b1; end
43.
44.           4:
45.             begin isDone <= 1'b0; i <= 6'd0; end
46.
47.           *****/
48.
49.           16,17,18,19,20,21,22,23:
50.             begin
51.               isQ = 1'b1;
52.               rSIO <= T[i-16];
53.
54.               if( C1 == 0 ) rSCLK <= 1'b0;
55.               else if( C1 == FHALF ) rSCLK <= 1'b1;
56.
57.               if( C1 == FCLK -1) begin C1 <= 6'd0; i <= i + 1'b1; end
58.               else C1 <= C1 + 1'b1;
59.
60.             end

```

```
61.          24:  
62.          i <= Go;  
63.  
64.      endcase
```

以上内容为部分核心操作。以上内容是写操作，步骤 16~24 是写一个字节的伪函数。步骤 0 拉高片选，准备访问字节，并且进入伪函数。步骤 1 准备写入数据并且进入伪函数。步骤 2 拉低片选，步骤 3~4 则是用来产生完成信号。

```
65.      else if( iCall[0] )  
66.          case( i )  
67.  
68.              0 :  
69.                  begin { rRST,rSCLK } <= 2'b10; T <= iAddr; i <= FF_Write; Go <= i + 1'b1; end  
70.  
71.              1:  
72.                  begin i <= FF_Read; Go <= i + 1'b1; end  
73.  
74.              2:  
75.                  begin { rRST,rSCLK } <= 2'b00; D1 <= T; i <= i + 1'b1; end  
76.  
77.              3:  
78.                  begin isDone <= 1'b1; i <= i + 1'b1; end  
79.  
80.              4:  
81.                  begin isDone <= 1'b0; i <= 6'd0; end  
82.  
83.          /*****/  
84.  
85.          16,17,18,19,20,21,22,23:  
86.          begin  
87.              isQ = 1'b1;  
88.              rSIO <= T[i-16];  
89.  
90.              if( C1 == 0 ) rSCLK <= 1'b0;  
91.              else if( C1 == FHALF ) rSCLK <= 1'b1;  
92.  
93.              if( C1 == FCLK -1) begin C1 <= 6'd0; i <= i + 1'b1; end  
94.              else C1 <= C1 + 1'b1;  
95.          end  
96.  
97.          24:  
98.              i <= Go;
```

```

99.
100.      ****
101.
102.      32,33,34,35,36,37,38,39:
103.      begin
104.          isQ = 1'b0;
105.
106.          if( C1 == 0 ) rSCLK <= 1'b0;
107.          else if( C1 == FHALF ) begin rSCLK <= 1'b1; T[i-32] <= RTC_DATA; end
108.
109.          if( C1 == FCLK -1) begin C1 <= 6'd0; i <= i + 1'b1; end
110.          else C1 <= C1 + 1'b1;
111.      end
112.
113.      40:
114.          i <= Go;
115.
116.      endcase
117.

```

以上内容为部分核心操作。以上内容是读操作，步骤 16~24 是写一个字节的伪函数，步骤 32~40 则是读一个字节的伪函数。步骤 0 拉高使能，准备访问字节并且进入写函数。步骤 1 进入读函数。步骤 2 拉低使能之余，也将读取结果暂存至 D。步骤 3~4 用来产生完成信号。

```

118.      assign { RTC_NRST,RTC_SCLK } = { rRST,rSCLK };
119.      assign RTC_DATA = isQ ? rSIO : 1'bz;
120.      assign oDone = isDone;
121.      assign oData = D1;
122.
123. endmodule

```

以上内容为相关输出驱动声明，其中 rSIO 驱动 RTC_DATA，D 驱动 oData。

ds1302_ctrlmod.v



图 23.11 DS1302 控制模块的建模图。

图 23.11 是该控制模块的建模图，右边信号用来调用功能模块，左边信号则被调用，其中 Call/Done 为 8 位宽，位宽内容如表 23.6 所示：

表 23.6 Call/Done 的位宽内容。

位	内容
[7]	关闭写保护
[6]	写入时钟
[5]	写入分钟
[4]	写入秒钟
[3]	开启写保护
[2]	读取时钟
[1]	读取分钟
[0]	读取秒钟

```
1. module ds1302_ctrlmod
2. (
3.     input CLOCK, RESET,
4.     input [7:0]iCall,
5.     output oDone,
6.     input [7:0]iData,
7.     output [1:0]oCall,
8.     input iDone,
9.     output [7:0]oAddr, oData
10. );
```

以上内容为相关的出入端声明。

```
11.     reg [7:0]D1,D2;
12.
13.     always @ ( posedge CLOCK or negedge RESET )
14.         if( !RESET )
15.             begin
16.                 D1 <= 8'd0;
17.                 D2 <= 8'd0;
18.             end
19.         else
20.             case( iCall[7:0] )
21.
22.                 8'b1000_0000 : // Write unprotect
23.                 begin D1 = 8'h8E; D2 = 8'b0000_0000; end
24.
25.                 8'b0100_0000 : // Write hour
```

```

26.          begin D1 = 8'h84; D2 = iData; end
27.
28.          8'b0010_0000 : // Write minit
29.          begin D1 = 8'h82; D2 = iData; end
30.
31.          8'b0001_0000 : // Write second
32.          begin D1 = 8'h80; D2 = iData; end
33.
34.          8'b0000_1000 : // Write protect
35.          begin D1 = 8'h8E; D2 = 8'b1000_0000; end
36.
37.          8'b0000_0100 : // Read hour
38.          begin D1 = 8'h85; end
39.
40.          8'b0000_0010 : // Read minit
41.          begin D1 = 8'h83; end
42.
43.          8'b0000_0001 : // Read second
44.          begin D1 = 8'h81; end
45.
46.      endcase
47.

```

以上内容为准备访问字节还有数据节还的周边操作，它会根据 iCall 的内容准备 D1 与 D2。

```

48.      reg [1:0]i;
49.      reg [1:0]isCall;
50.      reg isDone;
51.
52.      always @ ( posedge CLOCK or negedge RESET )
53.          if( !RESET )
54.              begin
55.                  i <= 2'd0;
56.                  isCall <= 2'b00;
57.                  isDone <= 1'b0;
58.              end

```

以上内容为相关寄存器声明还有复位操作。

```

59.          else if( iCall[7:3] ) // Write action
60.              case( i )
61.

```

```
62.          0 :
63.          if( iDone ) begin isCall[1] <= 1'b0; i <= i + 1'b1; end
64.          else begin isCall[1] <= 1'b1; end
65.
66.          1 :
67.          begin isDone <= 1'b1; i <= i + 1'b1; end
68.
69.          2 :
70.          begin isDone <= 1'b0; i <= 2'd0; end
71.
72.      endcase
```

以上内容为调用写操作。

```
73.      else if( iCall[2:0] ) // Read action
74.          case( i )
75.
76.          0 :
77.          if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
78.          else begin isCall[0] <= 1'b1; end
79.
80.          1 :
81.          begin isDone <= 1'b1; i <= i + 1'b1; end
82.
83.          2 :
84.          begin isDone <= 1'b0; i <= 2'd0; end
85.
86.      endcase
87.
```

以上内容为调用读操作。

```
88.      assign oDone = isDone;
89.      assign oCall = isCall;
90.      assign oAddr = D1;
91.      assign oData = D2;
92.
93.  endmodule
```

以上内容为相关的输出驱动。

ds1302_basemod.v

该模块的连线部署请参考图 23.10。

```
1. module ds1302_basemod
2. (
3.     input CLOCK, RESET,
4.     output RTC_NRST, RTC_SCLK,
5.     inout RTC_DATA,
6.     input [7:0]iCall,
7.     output oDone,
8.     input [7:0]iData,
9.     output [7:0]oData
10. );
11.    wire [7:0]AddrU1;
12.    wire [7:0]DataU1;
13.    wire [1:0]CallU1;
14.
15.    ds1302_ctrlmod U1
16.    (
17.        .CLOCK(CLOCK),
18.        .RESET(RESET),
19.        .iCall(iCall),           // < top
20.        .oDone(oDone),          // > top
21.        .iData(iData),          // > top
22.        .oCall(CallU1),         // > U2
23.        .iDone(DoneU2),         // < U2
24.        .oAddr(AddrU1),         // > U2
25.        .oData(DataU1)          // > U2
26.    );
27.
28.    wire DoneU2;
29.
30.    ds1302_funcmod U2
31.    (
32.        .CLOCK(CLOCK),
33.        .RESET(RESET),
34.        .RTC_NRST(RTC_NRST),    // > top
35.        .RTC_SCLK(RTC_SCLK),    // > top
36.        .RTC_DATA(RTC_DATA),    // <> top
37.        .iCall(CallU1),          // < U1
38.        .oDone(DoneU2),          // > U1
39.        .iAddr(AddrU1),          // > U1
```

```

40.           .iData( DataU1 ),          // > U1
41.           .oData( oData )          // > top
42.       );
43.
44. endmodule

```

ds1302_demo.v

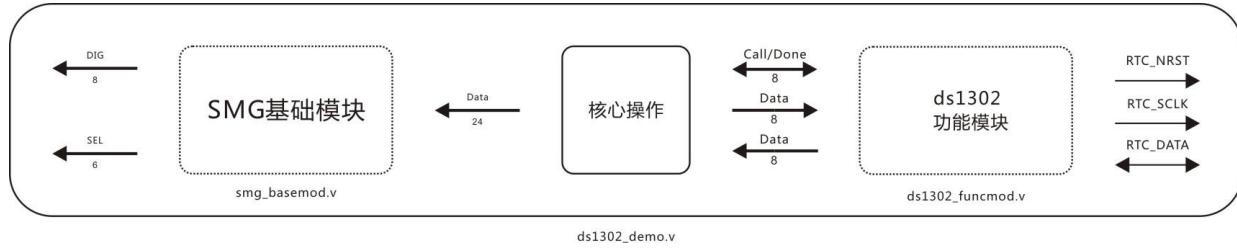


图 23.12 实验二十三的建模图。

图 23.12 是实验二十三的基础模块。核心操作先初始化 DS1302 基础模块，然后无间断从哪里读取时钟，分钟还有秒钟，最后驱动至 SMG 基础模块。具体内容让我们来看代码吧。

```

1. module ds1302_demo
2. (
3.     input CLOCK, RESET,
4.     output RTC_NRST, RTC_SCLK,
5.     inout RTC_DATA,
6.     output [7:0]DIG,
7.     output [5:0]SEL
8. );

```

以上内容为相关出入端声明。

```

9.     wire DoneU1;
10.    wire [7:0]DataU1;
11.
12.    ds1302_basemod U1
13.    (
14.        .CLOCK(CLOCK),
15.        .RESET(RESET),
16.        .RTC_NRST(RTC_NRST),      // > top
17.        .RTC_SCLK(RTC_SCLK),      // > top
18.        .RTC_DATA(RTC_DATA),      // > top
19.        .iCall(isCall),          // < core
20.        .oDone(DoneU1),          // > core

```

```
21.         .iData( D1 ),           // < core
22.         .oData( DataU1 )       // > core
23.     );
24.
```

以上内容为 DS1302 基础模块的实例化。

```
25.     smg_basemod U2
26.     (
27.         .CLOCK( CLOCK ),
28.         .RESET( RESET ),
29.         .DIG( DIG ),           // > top
30.         .SEL( SEL ),           // > top
31.         .iData( D2 )           // < core
32.     );
33.
```

以上内容为 SMG 基础模块的实例化。

```
34.     reg [3:0]i;
35.     reg [7:0]isCall;
36.     reg [7:0]D1;
37.     reg [23:0]D2;
38.
39.     always @ ( posedge CLOCK or negedge RESET )
40.         if( !RESET )
41.             begin
42.                 i <= 4'd0;
43.                 isCall <= 8'd0;
44.                 D1 <= 8'd0;
45.                 D2 <= 24'd0;
46.             end
47.         else
```

以上内容为相关寄存器声明还有复位操作。

```
48.         case( i )
49.
50.             0:
51.                 if( DoneU1 ) begin isCall[7] <= 1'b0; i <= i + 1'b1; end
52.                 else begin isCall[7] <= 1'b1; D1 <= 8'h00; end
53.
54.             1:
```

```

55.           if( DoneU1 ) begin isCall[6] <= 1'b0; i <= i + 1'b1; end
56.           else begin isCall[6] <= 1'b1; D1 <= { 4'd2, 4'd1 }; end
57.
58.           2:
59.           if( DoneU1 ) begin isCall[5] <= 1'b0; i <= i + 1'b1; end
60.           else begin isCall[5] <= 1'b1; D1 <= { 4'd5, 4'd9 }; end
61.
62.           3:
63.           if( DoneU1 ) begin isCall[4] <= 1'b0; i <= i + 1'b1; end
64.           else begin isCall[4] <= 1'b1; D1 <= { 4'd5, 4'd0 }; end
65.
66.           *****/
67.

```

以上内容为核心操作的部分内容，步骤 0 关闭写保护，步骤 1 初始化时钟，步骤 2 初始化分钟，步骤 3 初始化秒钟并且开启计时。

```

68.           4:
69.           if( DoneU1 ) begin D2[7:0] <= DataU1; isCall[0] <= 1'b0; i <= i + 1'b1; end
70.           else begin isCall[0] <= 1'b1; end
71.
72.           5:
73.           if( DoneU1 ) begin D2[15:8] <= DataU1; isCall[1] <= 1'b0; i <= i + 1'b1; end
74.           else begin isCall[1] <= 1'b1; end
75.
76.           6:
77.           if( DoneU1 ) begin D2[23:16] <= DataU1; isCall[2] <= 1'b0; i <= 4'd4; end
78.           else begin isCall[2] <= 1'b1; end
79.
80.           endcase
81.
82. endmodule

```

步骤 4 读取秒钟然后暂存至 D2[7:0]，步骤 5 读取分钟然后暂存至 D2[15:8]，步骤 6 读取时钟然后暂存至 D2[23:16]。综合完毕情切下载程序，如果数码管从 21-59-50 开始起跑，表示实验成功。

细节一：完整的个体模块

本实验的 DS1302 基础模块虽然是就绪的完整模块，不过依然无法满足那些欲无止境的读者 ... 例如，读年或者读天，还是利用 DS1302 的 RAM。对此，读者请自己探索然后尝试扩展该基础模块吧。

实验二十四：SD 模块

驱动 SD 卡是件容易让人抓狂的事情，驱动 SD 卡好比 SDRAM 执行页读写，SD 卡虽然不及 SDRAM 的麻烦要求（时序参数），但是驱动过程却有猥琐操作。除此此外，描述语言只要稍微比较一下 C 语言，描述语言一定会泪流满面，因为嵌套循环，嵌套判断，或者嵌套函数等都是它的痛。.

史莱姆模块是多模块建模的通病，意指结构能力非常脆弱的模块，暴力的嵌套行为往往会让模块的美丽身躯，好让脆弱结构更加脆弱还有惨不忍睹，最终搞垮模块的表达能力。描述语言预想驾驭 SD 卡，关键的地方就是如何提升模块的结构能力。简单而言，描述语言如何不失自身的美丽，又用自身的方法，去实现嵌套循环或者嵌套函数等近似的内容呢？

低级建模 I 之际，论结构能力它确实有点勉强，所以 SD 卡的实验才姗姗来迟。如今病猫已经进化为老虎，而且进化之初的新生儿都会饥饿如心焚，理智也不健全。因为如此，低级建模 II 才会不停舔着嘴唇，然后渴望新生的第一祭品。遇见 SD 卡，它仿佛遇见美味的猎物，口水都下流到一塌糊涂。

诸位少年少女们，让我们一起欢呼活祭仪式的开始吧！

二十一世纪的今天，SD 卡演化的速度简直迅雷不及掩耳。如今 SD 卡已经逐渐突破 64GB 大关。对此，SD 卡也存在 N 多版本，如版本 SDV1.x，版本 SDV2，或者 SDHCV2 等，当然未来还会继续演化下去。所谓版本是指建造工艺还有协议，粗略而言，版本 SDV1.x 是指容量为 2GB 以下的 SD 卡，版本 SDV2 则指容量为 2GB~4GB 之间的 SD 卡，版本 SDHCV2 则是容量为 4GB 以上的 SD 卡。

话虽如此，不过实际情况还要根据各个厂商的心情而定，有些厂商的 SD 卡虽为 4GB，但是版本却是 SDV1.x，还有厂商的 SD 卡的虽为 2GB，不过版本却是 SDV2，情况尽是让人哭笑不得。此外，版本不会印刷在硬件的表面上，而且不同版本也有不同驱动方法。俗语有云，擒贼先擒卒——凡事从娃娃抓起，所以笔者遵循伟大的智慧，从版本 SDV1.x 开始动手。

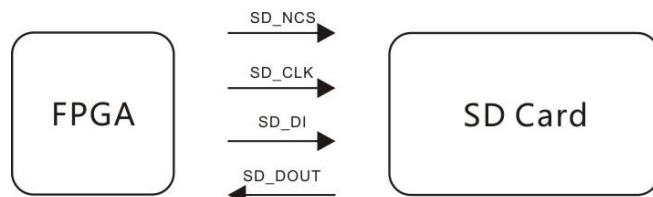


图 24.1 SPI 模式。

SD 卡有 SDIO 还有 SPI 两种模式，后者简单又省事，所以 SPI 模式都是众多懒惰鬼的喜爱。SPI 模式一般只用 4 只引脚，而且主机（FPGA）与从机（SD 卡）之间的链接如图 24.1 所示，至于引脚的繁荣如表 24.1 所示：

表 24.1 SD 卡 SPI 模式的引脚说明。

引脚	说明
SD_CLK	串行时钟，闲置为高
SD_NCS	片选，闲置为高，拉低有效
SD_DI	数据输入，也是主机输出
SD_DOUT	数据输出，也是主机输入

虽然 DS1302 也有 SPI，但是数据线是双向 IO，反之 SD 卡则是一对出入的数据线。话虽如此，它们两者都有乖乖遵守 SPI 的传输协议，即下降沿设置数据，上升沿锁存数据。

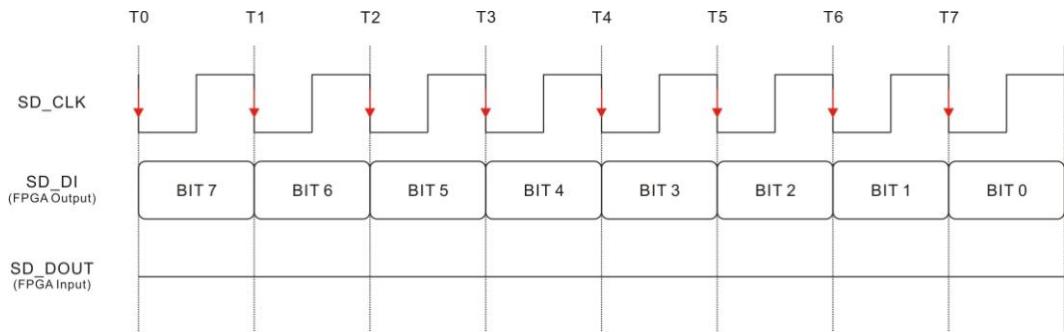


图 24.2 写一个字节 (主机视角)。

图 24.2 是主机视角写一个字节的理想时序。主机会利用时钟的下降沿，由高至低发送一个字节的数据。

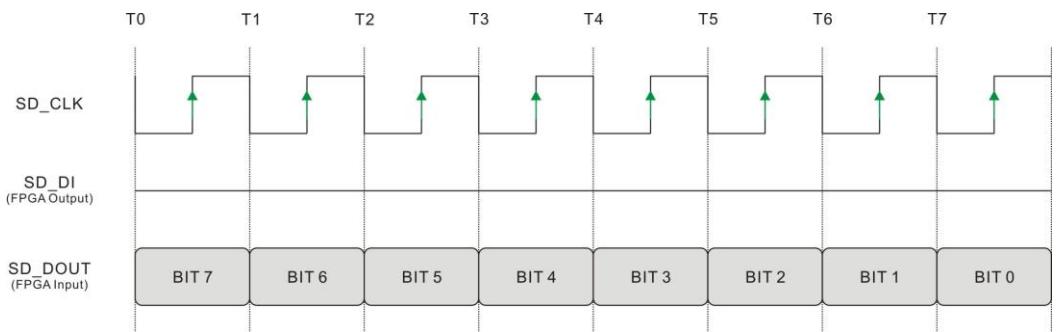


图 24.3 读一个字节 (主机视角)。

图 24.2 是主机视角读一个字节的理想时序。从机会利用时钟的下降沿，由高至低发送一个字节的数据，主机则会利用时钟信号的上升沿，由高至低读取一个字节的数据。

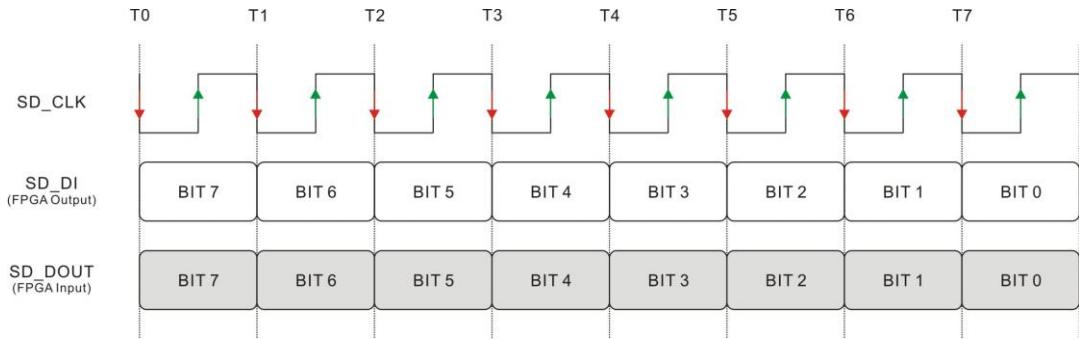


图 24.4 同时读写一个字节 (主机视角)。

我们知道 SD 卡有一对读写的数据线 ,为了节省时间 ,数据读写是同时发生的。如图 24.4 所示 ,那是主机在同时读写的理想时序 ,读者可以看成是图 24.2 还有图 24.3 的结合体。对此 ,Verilog 可以这样描述 ,结果如代码 24.1 所示 :

```

1. case(i)
2.
3.     0,1,2,3,4,5,6,7:
4.     begin
5.         rDI <= iData[ 7-i ];
6.
7.         if( C1 == 0 ) rSCLK <= 1'b0;
8.         else if( C1 == isHalf ) rSCLK <= 1'b1;
9.
10.        if( C1 == isQuarter ) D1[ 7-i ] <= SD_DOUT;
11.
12.        if( C1 == isFull -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
13.        else begin C1 <= C1 + 1'b1; end
14.    end

```

代码 24.1

如代码 24.1 所示 ,第 12~13 行表示步骤逗留的时间 ,其中 isFull 表示一个时钟周期。第 7~8 行表示 ,C1 为 0 拉低时钟 ,C1 为半个周期则拉高时钟。第 5 行表示 ,任何时候都更新数据 ,也可以看成 C1 为 0 输出数据。第 10 行表示 ,C1 为四分之一周期锁存数据。第 3 行表示 ,步骤 0~7 造就一个字节的读写。还有第 5~10 行的 D1[7-i] 表示 ,读写数据由高至低。

好奇的朋友一定会疑惑道 ,为何第 10 行的锁存行为不是时钟的半周期 (上升沿),而是四分之一呢 ?原因很单纯 ,因为数据在这个时候最为有效。

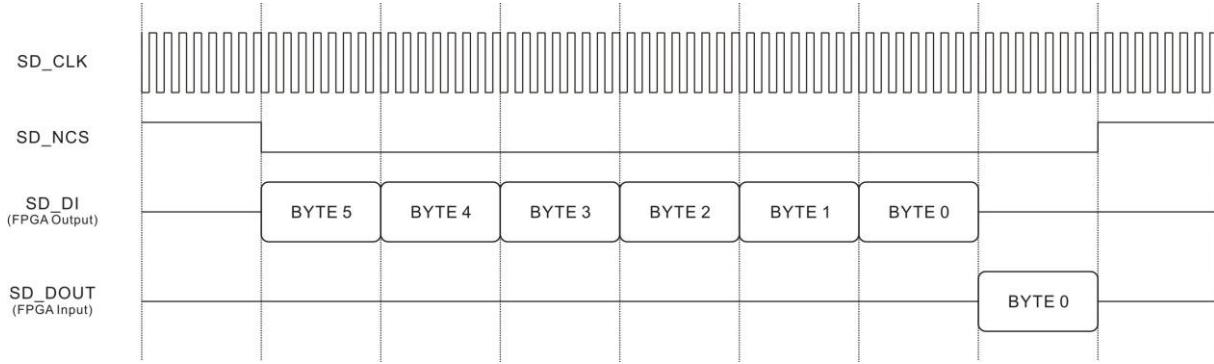


图 24.5 写命令 (主机视角)。

当然，SD 卡不是给足两只骨头就会满足的哈士奇 ... 为此，除了单纯的读写数据意外，SD 卡还有所谓的写命令，而写命令则是读写字节的复合体。如图 24.5 所示，那是主机写命令的理想时序，主机先由高至低发送 6 个字节的命令。SD 卡接受完毕以后，便会反馈一个字节的数据。期间，片选信号必须处于拉低状态。对此，Verilog 可以这样表示，结果如代码 24.2 所示：

```

1.   case( i )
2.
3.     0:
4.       begin rCMD <= iAddr; i <= i + 1'b1; end
5.
6.     1,2,3,4,5,6:
7.       begin T <= rCMD[47:40]; rCMD <= rCMD << 8; i <= FF_Write; Go <= i + 1'b1; end
8.
9.     7:
10.    begin i <= FF_Read; Go <= i + 1'b1; end
11.
12.    8:
13.      if( C2 == 100 ) begin C2 <= 10'd0; i <= i + 1'b1; end
14.      else if( D1 != 8'hff ) begin C2 <= 10'd0; i <= i + 1'b1; end
15.      else begin C2 <= C2 + 1'b1; i <= FF_Read; Go <= i; end
16.
17.    ...
18.
19.    12,13,14,15,16,17,18,19:
20.    begin
21.      rDI <= T[ 19-i ];
22.      if( C1 == 0 ) rSCLK <= 1'b0;
23.      else if( C1 == isHalf ) rSCLK <= 1'b1;
24.
25.      if( C1 == isQuarter ) D1[ 19-i ] <= SD_DOUT;
26.

```

```

27.           if( C1 == isFull -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
28.           else begin C1 <= C1 + 1'b1; end
29.       end
30.
31.   20:
32.   begin i <= Go; end

```

代码 24.2

步骤 12~20 是读写一个字节的伪函数，步骤 0 准备 6 个字节的命令，步骤 1~6 由高至低发送命令，并且进入伪函数。步骤 7 进入伪函数，并且读取一个字节的反馈数据（注意 FF_Write 与 FF_Read 都指向步骤 12）。反馈数据一般都是 8'hff 以外的结果，如果不是则重复读取反馈数据 100 次，如果 SD 卡反应正常，都会在这 100 次以内反馈 8'hff 以外的结果。

简单而言，如何驱动 SD 卡就是如何使用相关的命令。版本 SDV1.x 的 SD 卡只需 4 个命令而已，亦即：

- (一) CMD0，复位命令；
- (二) CMD1，初始化命令；
- (三) CMD24，写命令；
- (四) CMD17，读命令。

CMD0 用来复位 SD 卡，好让 SD 卡处于 (IDLE) 待机状态。CMD1 用来初始化 SD 卡，好让 SD 卡处于 (Transfer) 传输状态。CMD24 将 512 字节数据写入指定的地址，CMD17 则将 512 字节数据从指定的地址读出来。

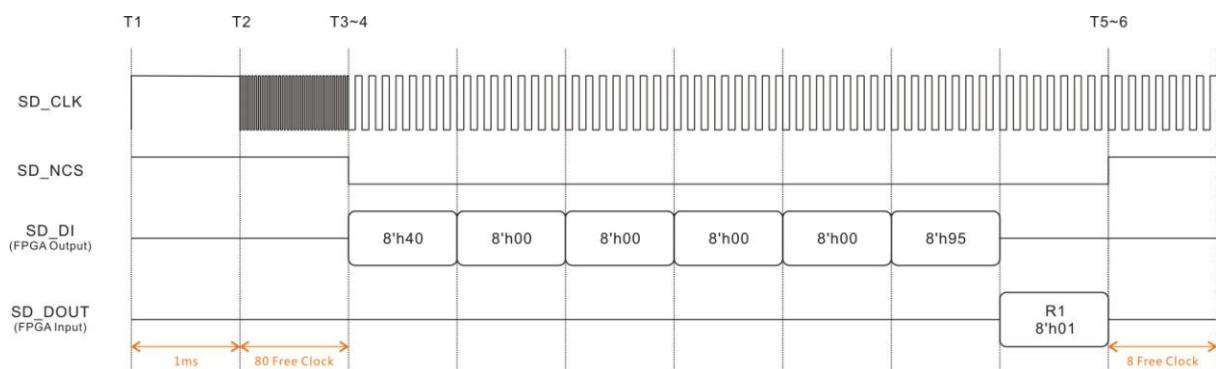


图 24.6 CMD0 的理想时序图。

图 24.6 是 CMD0 的理想时序图，首先在 T1 延迟 1ms 给予 SD 卡热身时间，然后再在 T2 给予 80 个准备的时钟。T3 之际拉低 CS，T4 之际则发送命令 CMD0 { 8'h40, 32'd0, 8'h95 }，然后等待 SD 卡反馈数据 R1。如果 SD 卡成功接收命令 CMD0，内容则是 8'h01。T5 之际拉高 CS，T6 之际再 8 个结束时钟。对此，Verilog 可以这样描述，结果如代码 24.3 所示：

```

1.   case( i )
2.
3.     0: // Disable cs, prepare Cmd0
4.       begin rCS <= 1'b1; D4 <= {8'h40, 32'd0, 8'h95}; i <= i + 1'b1; end
5.
6.     1: // Wait 1MS for warm up;
7.       if( C1 == T1MS -1) begin C1 <= 16'd0; i <= i + 1'b1; end
8.       else begin C1 <= C1 + 1'b1; end
9.
10.    2: // Send 80 free clock
11.      if( C1 == 10'd10 ) begin C1 <= 16'd0; i <= i + 1'b1; end
12.      else if( iDone ) begin isCall[0] <= 1'b0; C1 <= C1 + 1'b1; end
13.      else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
14.
15.    3: // Enable cs
16.      begin rCS <= 1'b0; i <= i + 1'b1; end
17.
18.    4: // Try 200 time, ready error code.
19.      if( C1 == 10'd200 ) begin D2 <= CMD0ERR; C1 <= 16'd0; i <= 4'd8; end
20.      else if( iDone && iData != 8'h01) begin isCall[1] <= 1'b0; C1 <= C1 + 1'b1; end
21.      else if( iDone && iData == 8'h01 ) begin isCall[1] <= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
22.      else isCall[1] <= 1'b1;
23.
24.    5: // Disable cs
25.      begin rCS <= 1'b1 ; i <= i + 1'b1; end
26.
27.    6: // Send free clock
28.      if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
29.      else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
30.
31.    7: // Disable cs, ready OK code
32.      begin D2 <= CMD0OK; i <= i + 1'b1; end //;
33.
34.    8: // Disbale cs, generate done signal
35.      begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
36.
37.    9:
38.      begin isDone <= 1'b0; i <= 4'd0; end

```

代码 24.3

我们先假设 isCall[1]执行写命令，isCall[0]则是执行读写字节。如代码 24.3 所示，步骤 0 用来准备 CMD0 命令。步骤 1 延迟 1ms。步骤 2 执行 10 次无意义的读写，以示给予 80 个准备时钟。在此读者稍微注意一下第 12 行，每当完成一次读写 C1 便会递增一下，

C1 递增 10 次便表示读写执行 10 次。

步骤 3 拉低 CS，并且步骤 4 发送命令。步骤 4 可能会吓坏一群小朋友，不过只要耐心解读，其它它并不可怕。首先执行第 22 行的写命令，如果反馈数据不为 8'h01（第 20 行），消除 isDo 便递增 C1，然后再返回第 22 行。如果反馈数据为 8'h01（第 21 行），消除 isDo 与 C1 然后继续步骤。如果重复执行 100 次都失败，D2 赋值 CMD0 的失败信息，消除 C1 并且 i 直接步骤 8。

步骤 5 拉低 CS，步骤 6 则给予 8 个结束时钟。步骤 7 为 D2 赋值 CMD0 的成功信息，步骤 8~9 拉高 CS 并且产生完成信号。

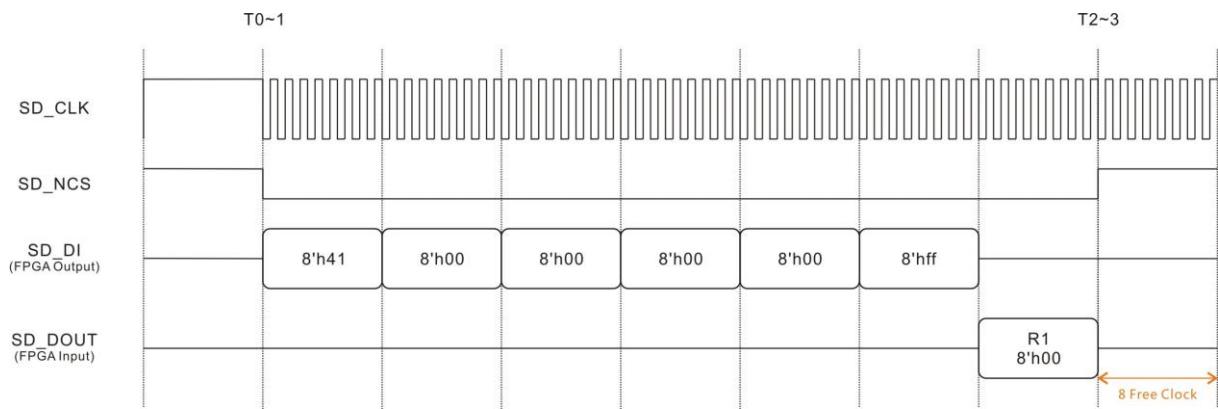


图 24.7 CMD1 的理想时序图。

图 24.7 是 CMD1 的理想时序图，T0&T1 之际拉低 CS 并且发送六个字节的命令 CMD1 {8'h41,32'd0,8'hff}。SD 卡接受命令以后便反馈数据 R1——8'h00。T2&T3 之际拉高 CS 并且给予 8 个结束时钟。Verilog 的描述结果如代码 24.4 所示：

```

1.   case( i )
2.
3.     0: // Enable cs, prepare Cmd1
4.       begin rCS <= 1'b0; D4 <= { 8'h41,32'd0,8'hff }; i <= i + 1'b1; end
5.
6.     1: // Try 100 times, ready error code.
7.       if( C1 == 10'd100 ) begin D2 <= CMD1ERR; C1 <= 16'd0; i <= 4'd5; end
8.       else if( iDone && iData != 8'h00) begin isCall[1]<= 1'b0; C1 <= C1 + 1'b1; end
9.       else if( iDone && iData == 8'h00 ) begin isCall[1] <= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
10.      else isCall[1] <= 1'b1;
11.
12.     2: // Disable cs
13.       begin rCS <= 1'b1; i <= i + 1'b1; end
14.
15.     3: // Send free clock
16.       if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
17.       else begin isCall[0] <= 1'b1; D1 <= 8'hff; end

```

```

18.
19.    4: // Disable cs, ready OK code.
20.    begin D2 <= CMD1OK; i <= i + 1'b1; end
21.
22.    5: // Disable cs, generate done signal
23.    begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
24.
25.    6:
26.    begin isDone <= 1'b0; i <= 4'd0; end

```

代码 24.4

如代码 24.4 所示，步骤 0 准备命令 CMD1。步骤 1 重复发送 CMD1 命令 100 次，直至反馈数据 R1 为 8'h00 为止，否则反馈错误信息。步骤 2 拉高 CS，步骤 3 则给予结束时钟。步骤 4 反馈成功信息，步骤 5~6 拉高 CS 之余也产生完成信号。

好奇的同学一定会觉得疑惑，命令 CMD0 与命令 CMD1 同样反馈数据 R1，为何前者是 8'h01，后者则是 8'h00 呢？事实上，R1 的内容也反应 SD 卡的当前状态，SD 卡有待机状态（IDLE）还有传输状态（Transfer）等两个常见状态。

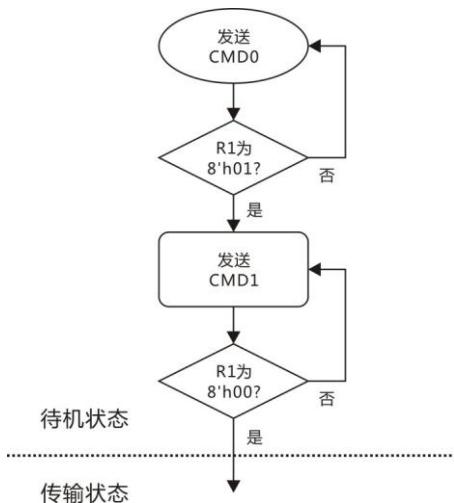


图 24.8 版本 V1.x 的初始化流程图。

如图 24.8 所示，那是版本 V1.x 的初始化流程图。主机先发送 CMD0，SD 卡接收以后如果反馈 R1 为 8'h01 便继续流程，否则重复发送 CMD0。主机接着发送 CMD1，如果 SD 卡接收并且反馈 R1 为 8'h00，该结果表示 SD 卡以从待机状态进入传输状态，余下 CMD24 还有 CMD17 才有效。

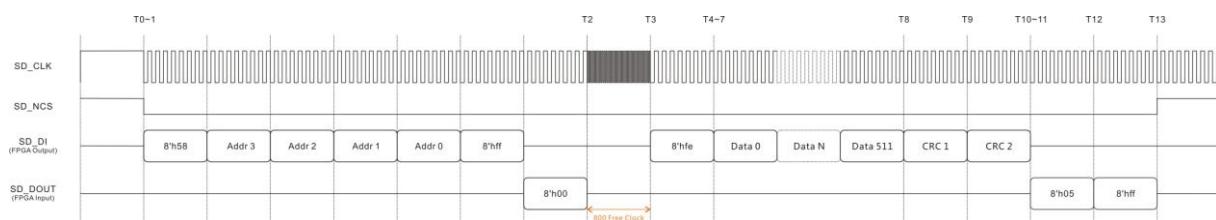


图 24.9 CMD24 的理想时序图。

图 24.9 是 CMD24 的理想时序图。T0~1 之际，主机拉低 CS 之余，主机也向 SD 卡发送写命令 CMD24，其中 Addr 3~Addr 0 是写入地址。SD 卡接收以后便以反馈数据 8'h00 表示接收成功。保险起见，主机在 T2 给足 800 个准备时钟，如果读者嫌准备时钟给太多，读者可以自行缩小至 80。T3 之际，主机发送 8'hfe 以示写 512 字节开始，T4~T7 则是写 512 字节的过程。T8~T9 分别写入两个 CRC 字节（CRC 校验）。

完后，SD 卡便会反馈 8'h05 以示写 512 字节成功，此刻（T10~11）主机读取并且检测。事后直至 SD 卡发送 8'hff 为止，SD 卡都处于忙状态。换言之，如果主机在 T12 成功读取 8'hff，结果表示 SD 卡已经忙完了。T13 之际，主机再拉高 CS。对此，Verilog 可以这样描述，结果如代码 24.5 所示：

```
1. case(i)
2.
3.     0: // Enable cs, prepare cmd24
4.         begin rCS <= 1'b0; D4 = { 8'h58, iAddr, 9'd0, 8'hFF }; i <= i + 1'b1; end
5.
6.     1: // Try 100 times, ready error code.
7.         if( C1 == 100 ) begin D2 <= CMD24ERR; C1 <= 16'd0; i <= 4'd14; end
8.         else if( iDone && iData != 8'h00) begin isCall[1] <= 1'b0; C1 <= C1 + 1'b1; end
9.         else if( iDone && iData == 8'h00 ) begin isCall[1] <= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
10.        else isCall[1] <= 1'b1;
11.
12.    2: // Send 800 free clock
13.        if( C1 == 100 ) begin C1 <= 16'd0; i <= i + 1'b1; end
14.        else if( iDone ) begin isCall[0] <= 1'b0; C1 <= C1 + 1'b1; end
15.        else begin isCall[0] <= 1'b1; D1 <= 8'hFF; end
16.
17.    3: // Send Call byte 0xfe
18.        if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
19.        else begin isCall[0] <= 1'b1; D1 <= 8'hFE; end
20.
21.    4: // Pull up read req.
22.        begin isEn[0] <= 1'b1; i <= i + 1'b1; end
23.
24.    5: // Pull down read req.
25.        begin isEn[0] <= 1'b0; i <= i + 1'b1; end
26.
27.    6: // Write byte from fifo
28.        if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
29.        else begin isCall[0] <= 1'b1; D1 <= iDataFF; end
30.
```

```

31.      7: // Repeat 512 times
32.      if( C1 == 10'd511 ) begin C1 <= 16'd0; i <= i + 1'b1; end
33.      else begin C1 <= C1 + 1'b1; i <= 4'd4; end
34.
35.      8: // Write 1st CRC
36.      if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
37.      else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
38.
39.      9: // Write 2nd CRC
40.      if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
41.      else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
42.
43.      10: // Read Respond
44.      if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
45.      else begin isCall[0] <= 1'b1; end
46.
47.      11: // if not 8'h05, faild and ready error code
48.      if( (iData & 8'h1F) != 8'h05 ) begin D2 <= CMD24ERR; i <= 4'd14; end
49.      else i <= i + 1'b1;
50.
51.      12: // Wait until sdcard free
52.      if( iDone && iData == 8'hff ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
53.      else if( iDone ) begin isCall[0] <= 1'b0; end
54.      else begin isCall[0] <= 1'b1; end
55.
56.      13: // Disable cs, ready OK code;
57.      begin D2 <= CMD24OK; i <= i + 1'b1; end
58.
59.      14: // Disable cs, generate done signal
60.      begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
61.
62.      15:
63.      begin isDone <= 1'b0; i <= 4'd0; end

```

代码 24.5

步骤 0 拉低 CS 之余，它也准备写命令 CMD24，其中 { iAddr, 9'd0 } 表示地址有 512 偏移量，内容等价 $iAddr \ll 9$ 。步骤 1 尝试写命令 100 次，直至反馈内容为 8'h00，否则便准备错误信息。步骤 2 发送 800 个准备时钟，如果嫌多可以自行缩小。步骤 3 写入开始字节 8'hfe。步骤 4~5 主要是从 FIFO 取得数据，步骤 6 则是将数据写入 SD 卡，步骤 7 用来控制循环的次数。步骤 8~9 分别写入两个 CRC 字节，内容随意。

步骤 10 读取反馈数据，步骤 11 则检测反馈数据是否为 8'h05，是则继续，不是则准备错误信息，并且跳转步骤 14（结束操作）。步骤 12 不断读取数据，直至读取内容为 8'hff

位置。步骤 13 准备成功信息。步骤 14~15 拉高 CS 之余也产生完成信号。在此，读者要稍微注意一下，步骤 4~7 组合起来，类似先执行后判断的 do ... while 循环。

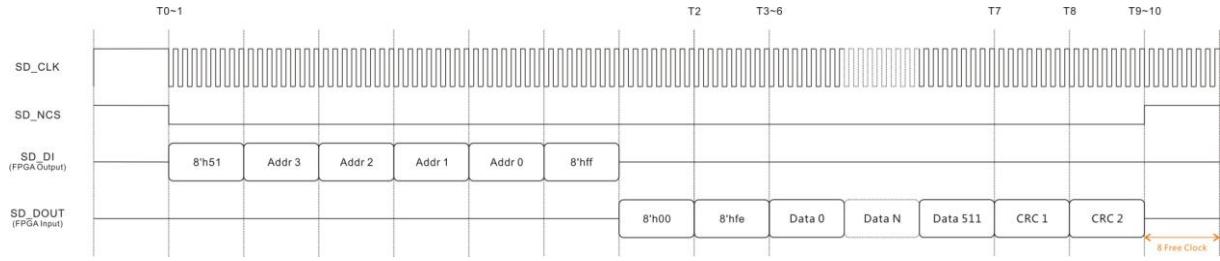


图 24.10 CMD17 的理想时序图。

图 24.10 是 CMD17 的理想时序图。T0~T1 之际，主机先拉低 CS 再发送命令 CMD17，其中 Addr3~Addr0 是指定的读地址，事后 SD 卡便会反馈数据 8'h00 以示接收成功。T2 之际，主机会不断读取数据，如果读取内容是 8'hfe，结果表示 SD 卡已经准备发送 512 字节数据。T3~T6 之际，主机不断从 SD 卡那里读取 512 个字节的数据。T7~T8 之际，主机读取两个字节的 CRC，然后在 T9~10 拉高 CS 之余也给足 8 个结束时钟。

换之，Verilog 的描述结果如代码 24.6 所示：

```

1.    case( i )
2.
3.      0: // Enable cs, prepare cmd17
4.      begin rCS <= 1'b0; D4 <= { 8'h51, iAddr, 9'd0, 8'hFF }; i <= i + 1'b1; end
5.
6.      1: // Try 100 times, ready error code
7.      if( C1 == 100 ) begin D2 <= CMD17ERR; C1 <= 16'd0; i <= 4'd12; end
8.      else if( iDone && iData != 8'h00 ) begin isCall[1] <= 1'b0; C1 <= C1 + 1'b1; end
9.      else if( iDone && iData == 8'h00 ) begin isCall[1] <= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
10.     else isCall[1] <= 1'b1;
11.
12.      2: // Wait read ready
13.      if( iDone && iData == 8'hfe ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
14.      else if( iDone && iData != 8'hfe ) begin isCall[0] <= 1'b0; end
15.      else isCall[0] <= 1'b1;
16.
17.      3: // Read byte
18.      if( iDone ) begin D3 <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
19.      else begin isCall[0] <= 1'b1; end
20.
21.      4: // Pull up write req.
22.      begin isEn[1] <= 1'b1; i <= i + 1'b1; end
23.
24.      5: // Pull down write req.

```

```

25.      begin isEn[1] <= 1'b0; i <= i + 1'b1; end
26.
27.      6: // Repeat 512 times
28.          if( C1 == 10'd511 ) begin C1 <= 16'd0; i <= i + 1'b1; end
29.          else begin C1 <= C1 + 1'b1; i <= 4'd3; end
30.
31.      7,8: // Read 1st and 2nd byte CRC
32.          if( iDone ) begin D3 <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
33.          else isCall[0] <= 1'b1;
34.
35.      9: // Disable cs
36.          begin rCS <= 1'b1; i <= i + 1'b1; end
37.
38.      10: // Send free clock
39.          if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
40.          else begin isCall[0] <= 1'b1; D1 <= 8'hFF; end
41.
42.      11: // Ready OK code
43.          begin D2 <= CMD17OK; i <= i + 1'b1; end
44.
45.      12: // Disable cs, generate done signal
46.          begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
47.
48.      13:
49.          begin isDone <= 1'b0; i <= 4'd0; end

```

代码 24.6

步骤 0 拉低 CS 之余也准备命令 CMD17，其中 {iAddr,9'd0} 为 512 的偏移量。步骤 1 重复 100 次写命令，直至 SD 卡反馈 8'h00，否则准备错误信息，然后跳转步骤 12（结束操作）。步骤 2 不断读取数据，直至读取内容为 8'hfe 为止。步骤 3 读取数据，步骤 4~5 则将数据写入 FIFO，步骤 6 用来控制循环。步骤 7~8 读取两个字节 CRC。步骤 9 拉高 CS，步骤 10 则给足 8 个结束时钟。步骤 11 准备成功信息。步骤 12 拉高 CS 之余，也产生完成信号。

上述内容理解完毕以后，我们便可以开始建模了。

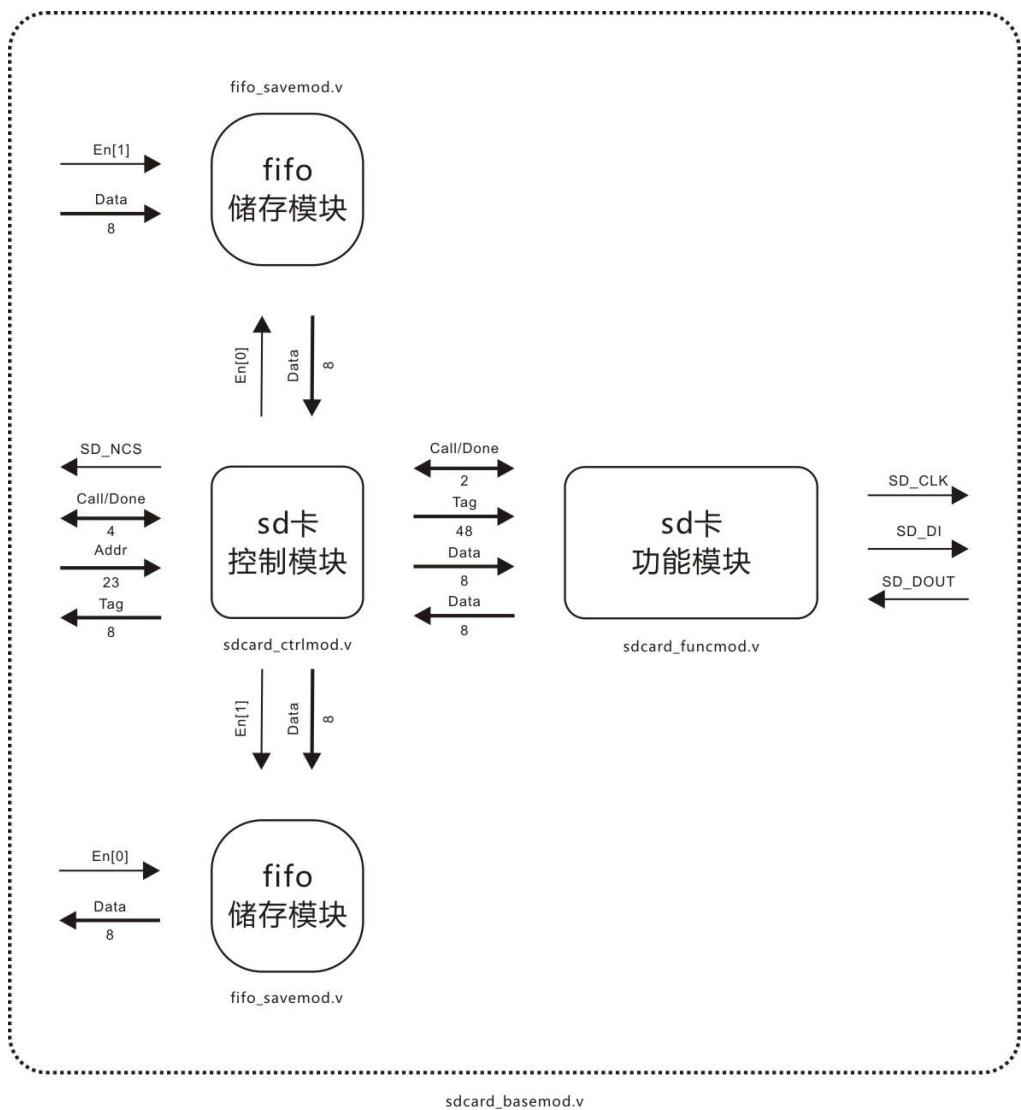


图 24.11 SD 卡基础模块的建模图。

图 24.11 是 SD 卡基础模块的建模图，其中内容包括 SD 卡控制模块，SD 卡功能模块，还有两个 fifo 储存模块。SD 卡功能模块的沟通信号 Call/Done 有两位位宽，其中[1]为写命令，[0]为字节读写。SD 卡控制模块的 Call/Done 位宽有四，表示它支持 4 个命令，其中[3]为 CMD24，[2]为 CMD17，[1]为 CMD1，[0]为 CMD0。两只 FIFO 储存模块充当写缓存(上)还有读缓存(下)，它们被外界调用以外，它们也被 SD 卡控制模块调用。

sdcard_funcmod.v



图 24.12 SD 卡功能模块的建模图。

图 24.12 是 SD 卡功能模块的建模图，右边是驱动 SD 卡的顶层信号，左边则是沟通用，还有命令，iData 与 oData 等数据信号。Call/Done 位宽有两，其中[1]为写命令，[0]为读写数据。

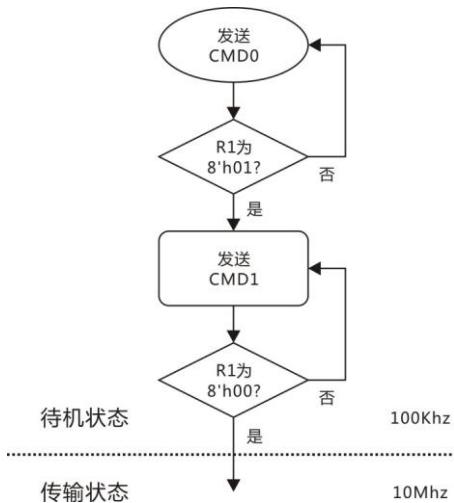


图 24.13 不同状态之间的传输速率。

话题继续之前，请允许笔者作足一些小补充。如图 24.13 所示，待机状态 SD 卡为低速状态，速率推荐为 100Khz~500Khz 之间。保险起见，笔者取为 100Khz。反之，传输状态 SD 卡处于高速状态，速率推荐为 2Mhz 或者以上。笔者衡量各种因数以后，笔者决定选择 10Mhz。丧心病狂的读者当然可以选择 10Mhz 以上的速率，如果硬件允许的话 ... 据说，100Mhz 也没有问题。

```
1. module sdcard_funcmod
2. (
3.     input CLOCK, RESET,
4.     input SD_DOUT,
5.     output SD_CLK,
6.     output SD_DI,
7.
8.     input [1:0]iCall,
9.     output oDone,
10.    input [47:0]iAddr,
11.    input [7:0]iData,
12.    output [7:0]oData
13. );
```

以上内容为出入端声明。

```
14. parameter FLCLK = 10'd500, FLHALF = 10'd250, FLQUARTER = 10'd125; //T20US = 100Khz
15. parameter FHCLK = 10'd5, FHHALF = 10'd1, FHQUARTER = 10'd2; // T1us = 10Mhz
```

```

16.
17.     reg [9:0]isFull,isHalf,isQuarter;
18.
19.     always @ ( posedge CLOCK or negedge RESET )
20.         if( !RESET )
21.             begin
22.                 isFull <= FLCLK;
23.                 isHalf <= FLHALF;
24.                 isQuarter <= FLQUARTER;
25.             end
26.         else if( iAddr[47:40] == 8'h41 && isDone )
27.             begin
28.                 isFull <= FHCLK;
29.                 isHalf <= FHHALF;
30.                 isQuarter <= FHQUARTER;
31.             end
32.

```

第 14~15 行是 100Khz 还有 10Mhz 的常量声明 , F \times CLK 为一个周期 , F \times HALF 为半个周期 , F \times QUARTER 为四分之一周期 , 其中 \times 为 L 表示低速 , \times 为 H 表示高速。第 17~31 行是设置速率的周边操作 , 起始下为低速 (第 22~24)。不过 , 当命令 CMD1 执行成功以后 , 速率转为为高速 (第 26~31 行)。

```

33.     parameter FF_Write = 5'd12, FF_Read = 5'd12;
34.
35.     reg [5:0]i,Go;
36.     reg [9:0]C1,C2;
37.     reg [7:0]T,D1;
38.     reg [47:0]rCMD;
39.     reg rSCLK,rDI;
40.     reg isDone;
41.
42.     always @ ( posedge CLOCK or negedge RESET )
43.         if( !RESET )
44.             begin
45.                 { i,Go } <= { 6'd0,6'd0 };
46.                 { C1,C2 } <= { 10'd0, 10'd0 };
47.                 { T,D1 } <= { 8'd0,8'd0 };
48.                 rCMD <= 48'd0;
49.                 { rSCLK,rDI } <= 2'b11;
50.                 isDone <= 1'b0;
51.             end

```

以上内容为相关的寄存器声明还有复位操作。其中第 33 行是伪函数的入口地址。

```
52.         else if( iCall[1] )
53.             case( i )
54.
55.                 0:
56.                     begin rCMD <= iAddr; i <= i + 1'b1; end
57.
58.                 1,2,3,4,5,6:
59.                     begin T <= rCMD[47:40]; rCMD <= rCMD << 8; i <= FF_Write; Go <= i + 1'b1; end
60.
61.                 7:
62.                     begin i <= FF_Read; Go <= i + 1'b1; end
63.
64.                 8:
65.                     if( C2 == 100 ) begin C2 <= 10'd0; i <= i + 1'b1; end
66.                     else if( D1 != 8'hff ) begin C2 <= 10'd0; i <= i + 1'b1; end
67.                     else begin C2 <= C2 + 1'b1; i <= FF_Read; Go <= i; end
68.
69.                 9:
70.                     begin isDone <= 1'b1; i <= i + 1'b1; end
71.
72.                 10:
73.                     begin isDone <= 1'b0; i <= 6'd0; end
74.
75.                     *****/
76.
77.                 12,13,14,15,16,17,18,19:
78.                     begin
79.                         rDI <= T[ 19-i ];
80.
81.                         if( C1 == 0 ) rSCLK <= 1'b0;
82.                         else if( C1 == isHalf ) rSCLK <= 1'b1;
83.
84.                         if( C1 == isQuarter ) D1[ 19-i ] <= SD_DOUT;
85.
86.                         if( C1 == isFull -1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
87.                         else begin C1 <= C1 + 1'b1; end
88.                     end
89.
90.                 20:
91.                     begin i <= Go; end
92.
```

```
93.           endcase
```

以上内容为写命令。

```
94.           else if( iCall[0] )
95.               case( i )
96.
97.                   0,1,2,3,4,5,6,7:
98.                   begin
99.                       rDI <= iData[ 7-i ];
100.
101.                      if( C1 == 0 ) rSCLK <= 1'b0;
102.                      else if( C1 == isHalf ) rSCLK <= 1'b1;
103.
104.                      if( C1 == isQuarter ) D1[ 7-i ] <= SD_DOUT;
105.
106.                      if( C1 == isFull - 1 ) begin C1 <= 10'd0; i <= i + 1'b1; end
107.                      else begin C1 <= C1 + 1'b1; end
108.                  end
109.
110.                  8:
111.                  begin isDone <= 1'b1; i <= i + 1'b1; end
112.
113.                  9:
114.                  begin isDone <= 1'b0; i <= 6'd0; end
115.
116.              endcase
117.
```

以上内容为读写一个字节。

```
118.      assign SD_CLK = rSCLK;
119.      assign SD_DI = rDI;
120.      assign oDone = isDone;
121.      assign oData = D1;
122.
123. endmodule
```

以上内容为相关的输出驱动。

fifo_savemod.v



图 24.14 FIFO 储存模块的建模图。

图 24.14 是大伙看烂的 FIFO 储存模块，具体内容让我们来看代码吧。

```
1. module fifo_savemod
2. (
3.     input CLOCK, RESET,
4.     input [1:0]iEn,
5.     input [7:0]iData,
6.     output [7:0]oData,
7.     output [1:0]oTag
8. );
9. initial begin
10.    for( C1 = 0; C1 < 1024; C1 = C1 + 1'b1 )
11.        begin  RAM[ C1 ] <= 8'd0; end
12.    end
13.
14.    reg [7:0] RAM [1023:0];
15.    reg [10:0] C1 = 11'd0,C2 = 11'd0; // N+1
16.    reg [7:0]D1;
17.
18.    always @ ( posedge CLOCK or negedge RESET )
19.        if( !RESET )
20.            begin
21.                C1 <= 11'd0;
22.            end
23.        else if( iEn[1] )
24.            begin
25.                RAM[ C1[9:0] ] <= iData;
26.                C1 <= C1 + 1'b1;
27.            end
28.
29.    always @ ( posedge CLOCK or negedge RESET )
30.        if( !RESET )
31.            begin
32.                C2 <= 11'd0;
```

```

33.           D1 <= 8'd0;
34.       end
35.   else if( iEn[0] )
36.   begin
37.       D1 <= RAM[ C2[9:0] ];
38.       C2 <= C2 + 1'b1;
39.   end
40.
41.   assign oData = D1;
42.   assign oTag[1] = ( C1[10]^C2[10] & C1[9:0] == C2[9:0] ); // Full Left
43.   assign oTag[0] = ( C1 == C2 ); // Empty Right
44.
45. endmodule

```

由于数据缓冲对象不是 SDRAM，所以第 41 行的 oData 由 D1 驱动而不是 RAM 直接驱动。余下内容，读者自己看着办吧。

sdcard_ctrlmod.v

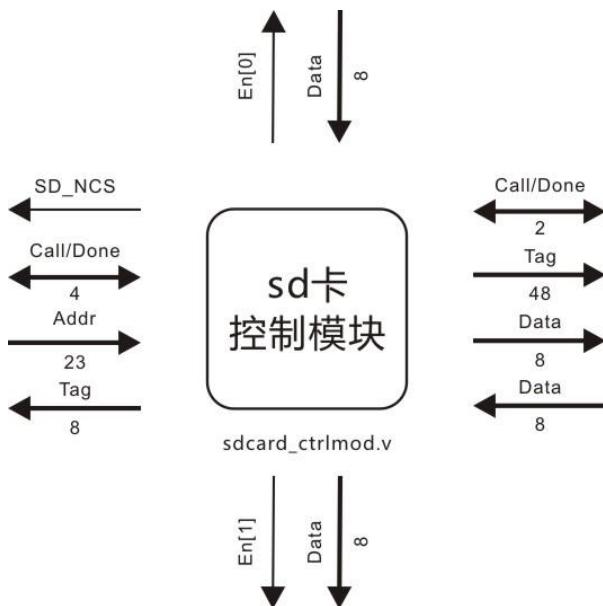


图 24.15 SD 卡控制模块的建模图。

图 24.15 是 SD 卡控制模块的建模图，它好比一只刺猬，全身上下都长满箭头，让人看见也怕怕。右边是调用功能模块的信号群，上下则是调用储存模块的信号群。左边则是被外界调用的信号群，其中顶层信号 SD_NCS 是 SD 卡的片选信号。此外，Call/Done 位宽有 4，表示该模块支持 4 个命令，[3]为 CMD24，[2]为 CMD17，[1]为 CMD1，[0]为 CMD0。至于 oTag 则是用来反馈命令的执行状态。

```

1. module sdcard_ctrlmod
2. (
3.     input CLOCK, RESET,
4.     output SD_NCS,
5.
6.     input [3:0]iCall,
7.     output oDone,
8.     input [22:0]iAddr,
9.     output [7:0]oTag,
10.
11.    output [1:0]oEn, // [1] Write [0] Read
12.    input [7:0]iDataFF,
13.    output [7:0]oDataFF,
14.
15.    output [1:0]oCall,
16.    input iDone,
17.    output [47:0]oAddr,
18.    input [7:0]iData,
19.    output [7:0]oData
20. );

```

以上内容为相关的出入端声明，第 6~9 行是外界调用的信号，第 11~13 行是调用 FIFO 的信号，第 15~19 则是调用功能模块的信号。

```

21. parameter CMD0ERR = 8'hA1, CMD0OK = 8'hA2, CMD1ERR = 8'hA3, CMD1OK = 8'hA4;
22. parameter CMD24ERR = 8'hA5, CMD24OK = 8'hA6, CMD17ERR = 8'hA7, CMD17OK = 8'hA8;
23. parameter T1MS = 16'd10;
24.

```

以上内容为各个命令的成功信息还有失败信息之间的常量声明。

```

25.     reg [3:0]i;
26.     reg [15:0]C1;
27.     reg [7:0]D1,D2,D3; // D1 WrData, D2 FbData, D3 RdData
28.     reg [47:0]D4;      // D4 Cmd
29.     reg [1:0]isCall,isEn;
30.     reg rCS;
31.     reg isDone;
32.
33.     always @ ( posedge CLOCK or negedge RESET )
34.         if( !RESET )
35.             begin
36.                 i <= 4'd0;

```

```

37.          C1 <= 16'd0;
38.          { D1,D2,D3 } <= { 8'd0, 8'd0, 8'd0 };
39.          D4 <= 48'd0;
40.          { isCall, isEn } <= { 2'd0,2'd0 };
41.          rCS <= 1'b1;
42.      end

```

以上内容为相关的寄存器声明还有复位操作，其中 D1 暂存写数据，D2 暂存反馈信息，D3 暂存读数据，D4 暂存命令，isDo 控制写命令还有字节读写，isEn 控制 FIFO 的读写。所有寄存器的复位值为 0，rCS 除外。

```

43.      else if( iCall[3] ) // cmd24
44.          case( i )
45.
46.              0: // Enable cs, prepare cmd24
47.              begin rCS <= 1'b0; D4 = { 8'h58, iAddr, 9'd0, 8'hFF }; i <= i + 1'b1; end
48.
49.              1: // Try 100 times, ready error code.
50.              if( C1 == 100 ) begin D2 <= CMD24ERR; C1 <= 16'd0; i <= 4'd14; end
51.              else if( iDone && iData != 8'h00) begin isCall[1] <= 1'b0; C1 <= C1 + 1'b1; end
52.              else if( iDone && iData == 8'h00 ) begin isCall[1] <= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
53.              else isCall[1] <= 1'b1;
54.
55.              2: // Send 800 free clock
56.              if( C1 == 100 ) begin C1 <= 16'd0; i <= i + 1'b1; end
57.              else if( iDone ) begin isCall[0] <= 1'b0; C1 <= C1 + 1'b1; end
58.              else begin isCall[0] <= 1'b1; D1 <= 8'hFF; end
59.
60.              3: // Send Call byte 0xfe
61.              if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
62.              else begin isCall[0] <= 1'b1; D1 <= 8'hFE; end
63.
64.              /*****
65.
66.              4: // Pull up read req.
67.              begin isEn[0] <= 1'b1; i <= i + 1'b1; end
68.
69.              5: // Pull down read req.
70.              begin isEn[0] <= 1'b0; i <= i + 1'b1; end
71.
72.              6: // Write byte from fifo
73.              if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
74.              else begin isCall[0] <= 1'b1; D1 <= iDataFF; end

```

```

75.
76.          7: // Repeat 512 times
77.          if( C1 == 10'd511 ) begin C1 <= 16'd0; i <= i + 1'b1; end
78.          else begin C1 <= C1 + 1'b1; i <= 4'd4; end
79.
80.          *****/
81.
82.          8: // Write 1st CRC
83.          if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
84.          else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
85.
86.          9: // Write 2nd CRC
87.          if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
88.          else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
89.
90.          10: // Read Respond
91.          if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
92.          else begin isCall[0] <= 1'b1; end
93.
94.          11: // if not 8'h05, faild and ready error code
95.          if( (iData & 8'h1F) != 8'h05 ) begin D2 <= CMD24ERR; i <= 4'd14; end
96.          else i <= i + 1'b1;
97.
98.          12: // Wait until sdcard free
99.          if( iDone && iData == 8'hff ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
100.         else if( iDone ) begin isCall[0] <= 1'b0; end
101.         else begin isCall[0] <= 1'b1; end
102.
103.         *****/
104.
105.         13: // Disable cs, ready OK code;
106.         begin D2 <= CMD24OK; i <= i + 1'b1; end
107.
108.         14: // Disable cs, generate done signal
109.         begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
110.
111.         15:
112.         begin isDone <= 1'b0; i <= 4'd0; end
113.
114.         endcase

```

以上内容为命令 CMD24。

```

115.    else if( iCall[2] ) // cmd17
116.        case( i )
117.
118.            0: // Enable cs, prepare cmd17
119.                begin rCS <= 1'b0; D4 <= { 8'h51, iAddr, 9'd0, 8'hFF }; i <= i + 1'b1; end
120.
121.            1: // Try 100 times, ready error code
122.                if( C1 == 100 ) begin D2 <= CMD17ERR; C1 <= 16'd0; i <= 4'd12; end
123.                else if( iDone && iData != 8'h00 ) begin isCall[1] <= 1'b0; C1 <= C1 + 1'b1; end
124.                else if( iDone && iData == 8'h00 ) begin isCall[1] <= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
125.                else isCall[1] <= 1'b1;
126.
127.            2: // Wait read ready
128.                if( iDone && iData == 8'hfe ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
129.                else if( iDone && iData != 8'hfe ) begin isCall[0] <= 1'b0; end
130.                else isCall[0] <= 1'b1;
131.
132.                *****/
133.
134.            3: // Read byte
135.                if( iDone ) begin D3 <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
136.                else begin isCall[0] <= 1'b1; end
137.
138.            4: // Pull up write req.
139.                begin isEn[1] <= 1'b1; i <= i + 1'b1; end
140.
141.            5: // Pull down write req.
142.                begin isEn[1] <= 1'b0; i <= i + 1'b1; end
143.
144.            6: // Repeat 512 times
145.                if( C1 == 10'd511 ) begin C1 <= 16'd0; i <= i + 1'b1; end
146.                else begin C1 <= C1 + 1'b1; i <= 4'd3; end
147.
148.                *****/
149.
150.            7,8: // Read 1st and 2nd byte CRC
151.                if( iDone ) begin D3 <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
152.                else isCall[0] <= 1'b1;
153.
154.            9: // Disable cs
155.                begin rCS <= 1'b1; i <= i + 1'b1; end
156.
157.            10: // Send free clock

```

```

158.         if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
159.         else begin isCall[0] <= 1'b1; D1 <= 8'hFF; end
160.
161.         11:// Ready OK code
162.         begin D2 <= CMD17OK; i <= i + 1'b1; end
163.
164.         12:// Disable cs, generate done signal
165.         begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
166.
167.         13:
168.         begin isDone <= 1'b0; i <= 4'd0; end
169.
170.     endcase

```

以上内容为命令 CMD17。

```

171.     else if( iCall[1] )// cmd1
172.         case( i )
173.
174.             0:// Enable cs, prepare Cmd1
175.             begin rCS <= 1'b0; D4 <= { 8'h41,32'd0,8'hff }; i <= i + 1'b1; end
176.
177.             1:// Try 100 times, ready error code.
178.             if( C1 == 10'd100 ) begin D2 <= CMD1ERR; C1 <= 16'd0; i <= 4'd5; end
179.             else if( iDone && iData != 8'h00) begin isCall[1]<= 1'b0; C1 <= C1 + 1'b1; end
180.             else if( iDone && iData == 8'h00 ) begin isCall[1] <= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
181.             else isCall[1] <= 1'b1;
182.
183.             2:// Disable cs
184.             begin rCS <= 1'b1; i <= i + 1'b1; end
185.
186.             3:// Send free clock
187.             if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
188.             else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
189.
190.             *****/
191.
192.             4:// Disable cs, ready OK code.
193.             begin D2 <= CMD1OK; i <= i + 1'b1; end
194.
195.             5:// Disable cs, generate done signal
196.             begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
197.

```

```

198.           6:
199.             begin isDone <= 1'b0; i <= 4'd0; end
200.
201.           endcase

```

以上内容为命令 CMD1。

```

202.       else if( iCall[0] ) // cmd0
203.         case( i )
204.
205.           0: // Disable cs, prepare Cmd0
206.             begin rCS <= 1'b1; D4 <= { 8'h40, 32'd0, 8'h95 }; i <= i + 1'b1; end
207.
208.           1: // Wait 1MS for warm up;
209.             if( C1 == T1MS -1) begin C1 <= 16'd0; i <= i + 1'b1; end
210.             else begin C1 <= C1 + 1'b1; end
211.
212.           2: // Send 80 free clock
213.             if( C1 == 10'd10 ) begin C1 <= 16'd0; i <= i + 1'b1; end
214.             else if( iDone ) begin isCall[0] <= 1'b0; C1 <= C1 + 1'b1; end
215.             else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
216.
217.           3: // Enable cs
218.             begin rCS <= 1'b0; i <= i + 1'b1; end
219.
220.           4: // Try 200 time, ready error code.
221.             if( C1 == 10'd200 ) begin D2 <= CMD0ERR; C1 <= 16'd0; i <= 4'd8; end
222.             else if( iDone && iData != 8'h01) begin isCall[1] <= 1'b0; C1 <= C1 + 1'b1; end
223.             else if( iDone && iData == 8'h01 ) begin isCall[1] <= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
224.             else isCall[1] <= 1'b1;
225.
226.           5: // Disable cs
227.             begin rCS <= 1'b1 ; i <= i + 1'b1; end
228.
229.           6: // Send free clock
230.             if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
231.             else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
232.
233.           7: // Disable cs, ready OK code
234.             begin D2 <= CMD0OK; i <= i + 1'b1; end
235.
236.           8: // Disbale cs, generate done signal
237.             begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end

```

```
238.  
239.          9:  
240.          begin isDone <= 1'b0; i <= 4'd0; end  
241.  
242.          endcase  
243.
```

以上内容为命令 CMD0。

```
244.      assign SD_NCS = rCS;  
245.      assign oDone = isDone;  
246.      assign oTag = D2;  
247.      assign oEn = isEn;  
248.      assign oDataFF = D3;  
249.      assign oCall = isCall;  
250.      assign oAddr = D4;  
251.      assign oData = D1;  
252.  
253. endmodule
```

以上内容为相关的输出驱动声明。

sdcards_basemod.v

该模块为 SD 卡基础模块，连线部署请参考图 24.11。

```
1. module sdcards_basemod  
2. (  
3.     input CLOCK, RESET,  
4.     input SD_DOUT,  
5.     output SD_CLK,  
6.     output SD_DI,  
7.     output SD_NCS,  
8.  
9.     input [3:0]iCall,  
10.    output oDone,  
11.    input [22:0]iAddr,  
12.    output [7:0]oTag,  
13.  
14.    input [1:0]iEn,  
15.    input [7:0]iData,  
16.    output [7:0]oData
```

```

17. );
18.     wire [1:0]EnU1;
19.     wire [7:0]DataFFU1;
20.     wire [1:0]CallU1;
21.     wire [47:0]AddrU1;
22.     wire [7:0]DataU1;
23.
24.     sdcard_ctrlmod U1
25.     (
26.         .CLOCK( CLOCK ),
27.         .RESET( RESET ),
28.         .SD_NCS( SD_NCS ),      // > top
29.         .iCall( iCall ),        // < top
30.         .oDone( oDone ),        // < top
31.         .iAddr( iAddr ),        // < top
32.         .oTag( oTag ),          // > top
33.         .oEn( EnU1 ),           // > U2 & U3
34.         .iDataFF( DataFFU2 ),    // < U2
35.         .oDataFF( DataFFU1 ),    // > U3
36.         .oCall( CallU1 ),        // > U4
37.         .iDone( DoneU4 ),        // < U4
38.         .oAddr( AddrU1 ),        // > U4
39.         .iData( DataU4 ),        // < U4
40.         .oData( DataU1 )         // > U4
41.     );
42.
43.     wire [7:0]DataFFU2;
44.
45.     fifo_savemod U2
46.     (
47.         .CLOCK ( CLOCK ),
48.         .RESET( RESET ),
49.         .iEn ( { iEn[1],EnU1[0] } ),   // < top & U1
50.         .iData ( iData ),            // < top
51.         .oData ( DataFFU2 ),        // > U1
52.         .oTag ()
53.     );
54.
55.     fifo_savemod U3
56.     (
57.         .CLOCK ( CLOCK ),
58.         .RESET( RESET ),
59.         .iEn ( { EnU1[1],iEn[0] } ), // < top & U1

```

```

60.      .iData ( DataFFU1 ),           // < U1
61.      .oData ( oData ),           // > top
62.      .oTag ()
63. );
64.
65.      wire DoneU4;
66.      wire [7:0]DataU4;
67.
68.      sdcards_funcmod U4
69. (
70.          .CLOCK( CLOCK ),
71.          .RESET( RESET ),
72.          .SD_CLK( SD_CLK ),           // > top
73.          .SD_DOUT( SD_DOUT ),       // < top
74.          .SD_DI( SD_DI ),           // > top
75.          .iCall( CallU1 ),           // < U1
76.          .oDone( DoneU4 ),           // > U1
77.          .iAddr( AddrU1 ),           // < U1
78.          .iData( DataU1 ),           // < U1
79.          .oData( DataU4 )           // > U1
80. );
81.
82. endmodule

```

以上内容，读者自己看着办吧，笔者犯懒了。

sdcards_demo.v

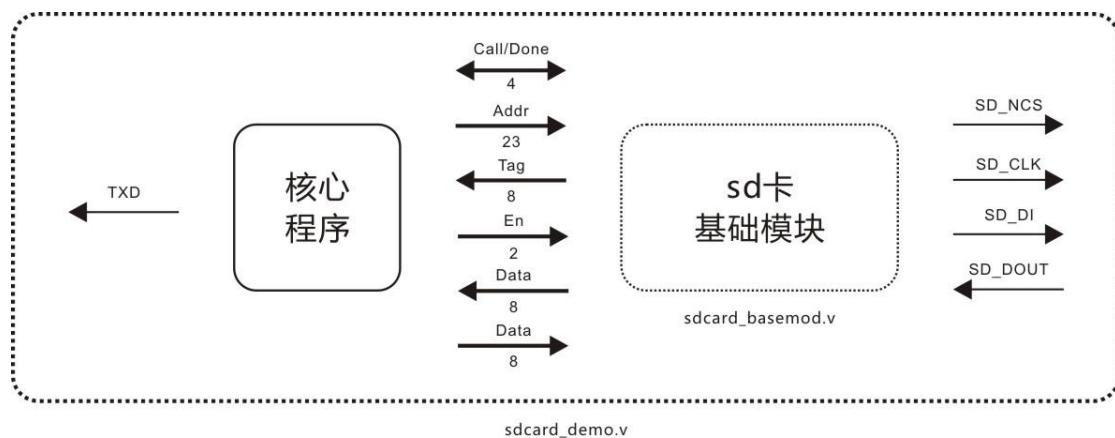


图 24.16 实验二十四的建模图。

图 24.16 是实验二十四的建模图，右边是 SD 卡基础模块，右边则是调用该模块的核心程序。核心程序先初始化 SD 卡，期间也将反馈信息经由 TXD 发送出去。再者，它将

512 个字节写入 SD 卡 , 又从中读出 , 然后经由 TXD 发送出去。具体内容让我们来看代码吧 :

```
1. module sdcard_demo
2. (
3.     input CLOCK,RESET,
4.     output SD_NCS,
5.     output SD_CLK,
6.     input SD_DOUT,
7.     output SD_DI,
8.     output TXD
9. );
```

以上内容为相关的出入端声明。

```
10.    wire DoneU1;
11.    wire [7:0]TagU1;
12.    wire [7:0]DataU1;
13.
14.    sdcard_basemod U1
15.    (
16.        .CLOCK( CLOCK ),
17.        .RESET( RESET ),
18.        .SD_DOUT( SD_DOUT ),
19.        .SD_CLK( SD_CLK ),
20.        .SD_DI( SD_DI ),
21.        .SD_NCS( SD_NCS ),
22.        .iCall( isCall ),
23.        .oDone( DoneU1 ),
24.        .iAddr( D1 ),
25.        .oTag( TagU1 ),
26.        /*****
27.        .iEn( isEn ),
28.        .iData( D2 ),
29.        .oData( DataU1 )
30.    );
31.
```

以上内容为 SD 卡基础模块实例化 , 其中 isCall 驱动 iCall , D1 驱动 iAddr , isEn 驱动 iEn , D2 驱动 iData。

```
32.    parameter B115K2 = 11'd434, TXFUNC = 6'd16;
33.
```

```

34.      reg [5:0]i,Go;
35.      reg [10:0]C1,C2;
36.      reg [22:0]D1;
37.      reg [7:0]D2;
38.      reg [10:0]T;
39.      reg [3:0]isCall;
40.      reg [1:0]isEn;
41.      reg rTXD;
42.
43.      always @ ( posedge CLOCK or negedge RESET )
44.          if( !RESET )
45.              begin
46.                  { i,Go } <= { 6'd0,6'd0 };
47.                  { C1,C2 } <= { 11'd0,11'd0 };
48.                  { D1,D2,T } <= { 23'd0,8'd0,11'd0 };
49.                  { isCall,isEn } <= { 4'd0,2'd0 };
50.                  rTXD <= 1'b1;
51.              end
52.          else

```

以上内容为相关的寄存器声明还有复位操作，当然也包括波特率的常量声明还有伪函数入口。

```

53.          case( i )
54.
55.              0: // cmd0
56.                  if( DoneU1 ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
57.                  else begin isCall[0] <= 1'b1; end
58.
59.              1:
60.                  begin T <= { 2'b11, TagU1, 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
61.
62.                  *****/
63.

```

步骤 0 执行 CMD0，然后步骤 1 反馈执行结果。

```

64.          2: // cmd1
65.              if( DoneU1 ) begin isCall[1] <= 1'b0; i <= i + 1'b1; end
66.              else begin isCall[1] <= 1'b1; end
67.
68.          3:
69.              begin T <= { 2'b11, TagU1, 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end

```

```
70.  
71.          *****/  
72.
```

步骤 2 执行 CMD1 , 然后步骤 3 反馈执行结果。

```
73.          4: // write data to fifo  
74.          begin isEn[1] <= 1'b1; i <= i + 1'b1; end  
75.  
76.          5:  
77.          begin isEn[1] <= 1'b0; i <= i + 1'b1; end  
78.  
79.          6:  
80.          if( C2 == 511 ) begin C2 <= 11'd0; i <= i + 1'b1; end  
81.          else begin D2 <= D2 + 1'b1; C2 <= C2 + 1'b1; i <= 6'd4; end  
82.  
83.          *****/  
84.
```

步骤 4~6 将数据 00~FF 写入 FIFO 两遍。

```
85.          7: // cmd24  
86.          if( DoneU1 ) begin isCall[3] <= 1'b0; i <= i + 1'b1; end  
87.          else begin isCall[3] <= 1'b1; D1 <= 23'd0; end  
88.  
89.          8:  
90.          begin T <= { 2'b11, TagU1, 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end  
91.  
92.          *****/  
93.
```

步骤 7 执行 CMD24 , 写入地址为 23'd0。 步骤 8 反馈执行结果。

```
94.          9: // cmd17  
95.          if( DoneU1 ) begin isCall[2] <= 1'b0; i <= i + 1'b1; end  
96.          else begin isCall[2] <= 1'b1; D1 <= 23'd0; end  
97.  
98.          10:  
99.          begin T <= { 2'b11, TagU1, 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end  
100.  
101.         *****/  
102.
```

步骤 9 执行 CMD17 , 步骤 10 则反馈执行结果。

```
103.           11: // Read data from fifo
104.           begin isEn[0] <= 1'b1; i <= i + 1'b1; end
105.
106.           12:
107.           begin isEn[0] <= 1'b0; i <= i + 1'b1; end
108.
109.           13:
110.           begin T <= { 2'b11, DataU1, 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
111.
112.           14:
113.           if( C2 == 511 ) begin C2 <= 11'd0; i <= i + 1'b1; end
114.           else begin C2 <= C2 + 1'b1; i <= 6'd11; end
115.
116.           15:
117.           i <= i;
118.
119.           *****/
120.
```

步骤 11~14 从 FIFO 哪里读出数据 512 次 , 然后再经由 TXD 发送出去。

```
121.           16,17,18,19,20,21,22,23,24,25,26:
122.           if( C1 == B115K2 -1 ) begin C1 <= 11'd0; i <= i + 1'b1; end
123.           else begin rTXD <= T[i - 16]; C1 <= C1 + 1'b1; end
124.
125.           27:
126.           i <= Go;
127.
128.           endcase
129.
130.           assign TXD = rTXD;
131.
132. endmodule
```

步骤 16~27 是发送一帧数据的伪函数。综合完毕 , 插入版本 V1.x 的 SD 卡 , 例如笔者手上 IProc 制 , 容量为 256MB 的 SD 卡 , 然后下载程序。演示过程如下 :

```
A2 // CMD0 执行成功
A4 // CMD1 执行成功
A6 // CMD24 执行成功
A8 // CMD17 执行成功
```

```

00~FF // 读出数据 0~255
00~FF // 读出数据 256~511

```

		0001 0203 0405 0607 0809 0A0B 0C0D 0EOF 0123456789ABCDEF
0x00:	0x00	0001 0203 0405 0607 0809 0A0B 0C0D 0EOF
0x00:	0x10	1011 1213 1415 1617 1819 1A1B 1C1D 1E1F
0x00:	0x20	2021 2223 2425 2627 2829 2A2B 2C2D 2E2F !"#\$%&'()*+,-./
0x00:	0x30	3031 3233 3435 3637 3839 3A3B 3C3D 3E3F 0123456789:;=>?
0x00:	0x40	4041 4243 4445 4647 4849 4A4B 4C4D 4E4F @ABCDEFGHIJKLMNO
0x00:	0x50	5051 5253 5455 5657 5859 5A5B 5C5D 5E5F PQRSTUVWXYZ[\]^_
0x00:	0x60	6061 6263 6465 6667 6869 6A6B 6C6D 6E6F `abcdefghijklmn
0x00:	0x70	7071 7273 7475 7677 7879 7A7B 7C7D 7E7F pqrstuvwxyz{ }~
0x00:	0x80	8081 8283 8485 8687 8889 8A8B 8C8D 8E8F
0x00:	0x90	9091 9293 9495 9697 9899 9A9B 9C9D 9E9F
0x00:	0xA0	A0A1 A2A3 A4A5 A6A7 A8A9 AAAB ACAD AEAF
0x00:	0xB0	B0B1 B2B3 B4B5 B6B7 B8B9 BABB BCBD BEBF
0x00:	0xC0	C0C1 C2C3 C4C5 C6C7 C8C9 CACB CCCD CECF
0x00:	0xD0	D0D1 D2D3 D4D5 D6D7 D8D9 DADB DCDD DEDF
0x00:	0xE0	E0E1 E2E3 E4E5 E6E7 E8E9 EAEB ECED EEEF
0x00:	0xF0	F0F1 F2F3 F4F5 F6F7 F8F9 FAFB FCFD FEFF
0x00:	0x0100	0001 0203 0405 0607 0809 0A0B 0C0D 0EOF
0x00:	0x0110	1011 1213 1415 1617 1819 1A1B 1C1D 1E1F
0x00:	0x0120	2021 2223 2425 2627 2829 2A2B 2C2D 2E2F !"#\$%&'()*+,-./
0x00:	0x0130	3031 3233 3435 3637 3839 3A3B 3C3D 3E3F 0123456789:;=>?
0x00:	0x0140	4041 4243 4445 4647 4849 4A4B 4C4D 4E4F @ABCDEFGHIJKLMNO
0x00:	0x0150	5051 5253 5455 5657 5859 5A5B 5C5D 5E5F PQRSTUVWXYZ[\]^_
0x00:	0x0160	6061 6263 6465 6667 6869 6A6B 6C6D 6E6F `abcdefghijklmn
0x00:	0x0170	7071 7273 7475 7677 7879 7A7B 7C7D 7E7F pqrstuvwxyz{ }~
0x00:	0x0180	8081 8283 8485 8687 8889 8A8B 8C8D 8E8F
0x00:	0x0190	9091 9293 9495 9697 9899 9A9B 9C9D 9E9F
0x00:	0x01A0	A0A1 A2A3 A4A5 A6A7 A8A9 AAAB ACAD AEAF
0x00:	0x01B0	B0B1 B2B3 B4B5 B6B7 B8B9 BABB BCBD BEBF
0x00:	0x01C0	C0C1 C2C3 C4C5 C6C7 C8C9 CACB CCCD CECF
0x00:	0x01D0	D0D1 D2D3 D4D5 D6D7 D8D9 DADB DCDD DEDF
0x00:	0x01E0	E0E1 E2E3 E4E5 E6E7 E8E9 EAEB ECED EEEF
0x00:	0x01F0	F0F1 F2F3 F4F5 F6F7 F8F9 FAFB FCFD FEFF

图 24.17 SD 卡的内容。

为了验证 SD 卡是否成功写入 00~FF 两遍，笔者稍微瞧瞧 SD 卡的内容 ... 如图 24.17 所示，地址 0x00~0xF0 (0~255) 的内容是 00~FF，地址 0x0100~0x01F0 (256~511) 的内容也是 00~FF。

细节一：完整的个体模块

虽然本实验的 SD 卡基础模块已经就绪，不过 SD 卡的前提条件必须是版本 SDV1.x，还有健康的硬件。嘛，SD 卡基础模块傻是傻了一点，不过它还可以继续扩展。

实验二十五：SDHC 模块

笔者曾经说过，SD 卡发展至今已经衍生许多版本，实验二十四就是针对版本 SDV1.x 的 SD 卡。实验二十四也说过，CMD24 还有 CMD17 会故意偏移地址 2^9 ，让原本范围指向从原本的 2^{32} 变成 2^{23} ，原因是 SD 卡读写一次都有 512 个字节。为此我们可以这样计算：

```
SDV1.x = 223 * 512 * Bytes // 512 个字节读写
        = 4.294967296e9 Bytes
        = 4.194304e6 kBytes
        = 4096 MBytes
        = 4 GBytes
```

```
SDV1.x = 232 * Bytes          // 单字节读写
        = 4.294967296e9 Bytes
        = 4.194304e6 kBytes
        = 4096 MBytes
        = 4 GBytes
```

不管是 2^{23} 乘以 512 字节读写，还是 2^{32} 直接进行单字节读写，该内容显示版本 SDV1.x 可以支持的最大容量只有 4GB 而已。不过，容量 4GB 再也无法满足现今的饿狼，为此 SD 卡被迫提升版本，即版本 SDV2 还有版本 SDHCV2。所谓版本 SDV2 就是 SD Card Version 2，所谓 SDHCV2 就是 SD Card High Capacity Version 2。

好奇的朋友一定会觉得疑惑，为何进化以后的 SD 卡会有版本 SDV2 还有版本 SDHCV2 之分呢？版本 SDV2 可谓是版本 SDHCV2 的脚垫，为何笔者会怎么说呢？其实事情的背后有隐藏这样一起凄惨的故事，无疑听着会伤心，闻者会流泪，就连坚强的比卡丘也不再放电。故事是这样的 ...

西元两千年之际，SD 卡因为其方便性还有大容量等特征，结果一炮而红，名声也随之传遍各个人类居住的大陆。根据杰克斯的理论，备受需求的东西都会急剧发展。为此，好评如潮的 SD 卡也在短短的几年内，从原本小小的 64MB 膨胀至 4GB 容量，但是需求必定超过结构的负荷。

SD 卡为了满足日益剧增的大容量需求，结果它不得不放弃版本 SDV1.x，取而代之就是版本 SDV2，还有版本 SDHCV2。版本 SDV1.x 与版本 SDV2 之间的差别是前者从单字节开始读写，而后者则是从 512 字节可是读写。为此，我们可以这样计算：

```
SDV2 = 232 * 512 * Bytes
      = 2.199023255552e12 Bytes
      = 2.147483648e9 kBytes
      = 2.097152e6 MBytes
```

$$\begin{aligned}
 &= 2028 \text{ GBytes} \\
 &= 2 \text{ TBytes}
 \end{aligned}$$

简单来说，版本 2DV2 单个地址是指向 512 字节，因此 2^{32} 可以指向 2TB 的范围。理论而言，版本 SDV2 的确可以支持 2TB 的大容量，但是闹肚子的厂商们，根本来不及实验便纷纷将老版本的 SD 卡改为版本 SDV2。云之间，市场便充斥许多版本 SDV1.x 与版本 SDV2 的 SD 卡，同样是 4GB 的 SD 卡，不过同时兼有版本 SDV1.x 与版本 SDV2，可谓是 SD 卡的浑沌时代。

随着科技发展，SD 卡的容量也直线上升，8GB, 16GB, 32GB, 64GB 等各种大容量 SD 卡也随之面世，如今 128GB 的 SD 卡也是近在眼前。此刻，问题发生了 … 我们知道版本 SDV2 是倡促之下衍生的产物，理论上它虽然可以支持 2TB 的容量，但是它并不适合支持大容量的 SD 卡，因为有许多小细节都顾及不到，结果版本 SDHCV2 诞生了。

现实总是太残酷，版本 SDHCV2 有如剧毒般，慢慢侵蚀版本 SDV2，版本 SDV2 也随之失去为期不长的光亮，最终沦落为脚垫 … 版本 SDV2 实在太可怜，让我们为它默哀一下吧。听完故事以后，笔者希望读者始理解，与其驱动版本 SDV2 还不如驱动版本 SDHCV2，只要理解后者前者自然不学而会。那么，我们开始实验吧。

版本 SDHCV2 需要用到以下几个命令：

- (一) CMD0，复位命令，令 SD 卡处于待机 (IDLE) 状态；
- (二) CMD8，配置命令，配置一些物理参数；
- (三) CMD58，状态命令，令 SD 卡反馈状态；
- (四) CMD55，扩充命令，告诉 SD 卡下一个命令为扩充命令；
- (五) ACMD41，扩充命令，配置高容量标示 (HCS)；
- (六) CMD16，配置命令，配置读写字节的长度，默认为 512；
- (七) CMD24，写命令；
- (八) CMD24，读命令。

看着看着，双腿也会开始发软，因为遇见那么多不认识的命令，驱动大容量 SD 卡确实吓人，不过笔者会陪伴左右，所以不要太担心。首先是 CMD0，这个命令曾在实验二十三解释过，请恕笔者不重复了，CMD0 的作用主要是令 SD 卡处于待机状态。

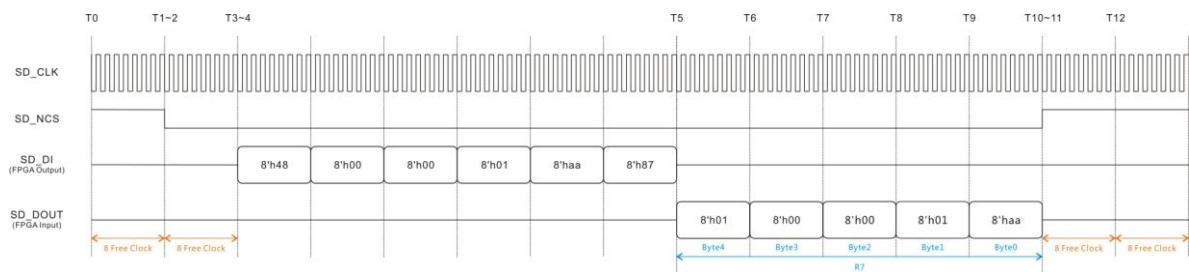


图 25.1 CMD8 的理想时序图。

图 25.1 是 CMD8 的理想时序图。T0 之际，主机拉高 CS 并且给足 8 个时钟。T1~2 之际，主机拉低 CS 并且给足 8 个时钟。T3~4 之际，主机发送命令 { 8'h48, 16'd0, 8'h01, 8'haa, 8'h87 }，其中 { 8'h01, 8'haa } 是一些默认物理配置，好奇的朋友可以翻查手册。T5 之际，SD 卡接收命令以后便开始反馈数据 R7，主机在 T5~T9 期间接收反馈数据 R7。

R7 是由 5 个字节组成的反馈数据，第 4 个字节类似 R1，内容 8'h01 表示 SD 卡处于待机状态。接续四个字节的内容是命令 CMD8 的倒影——{ 16'd0, 8'h01, 8'haa }。T10~11 之际，主机拉高 CS 并且给足 8 个时钟。T12 之际，主机再给足 8 个结束时钟。对此，Verilog 可以这样描述，结果如代码 25.1 所示：

```
1. case( i )
2.
3.     0: // Send free clock
4.         if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
5.         else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
6.
7.     1: // Enable cs
8.         begin rCS <= 1'b0; i <= i + 1'b1; end
9.
10.    2: // Send free clock
11.        if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
12.        else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
13.
14.    3: // Prepare Cmd8 // 8'h01 = 2.7~3.6v, 8'hAA default check pattern
15.        begin D4 <= { 8'h48,16'd0,8'h01,8'hAA,8'h87 }; i <= i + 1'b1; end
16.
17.    4: // Try 100 times, ready error code.
18.        if( C1 == 10'd100 ) begin D2[7:0] <= CMD8ERR; C1 <= 16'd0; i <= 4'd13; end
19.        else if( (iDone && iData != 8'h01) ) begin isCall[1]<= 1'b0; C1 <= C1 + 1'b1; end
20.        else if( (iDone && iData == 8'h01) ) begin isCall[1] <= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
21.        else isCall[1] <= 1'b1;
22.
23.    5: // Store R7
24.        begin D2[39:32] <= iData; i <= i + 1'b1; end
25.
26.    6: // Read and store R7
27.        if( iDone ) begin D2[31:24] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
28.        else begin isCall[0] <= 1'b1; end
29.
30.    7: // Read and store R7
31.        if( iDone ) begin D2[23:16] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
32.        else begin isCall[0] <= 1'b1; end
33.
```

```

34.      8: // Read and store R7
35.      if( iDone ) begin D2[15:8] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
36.      else begin isCall[0] <= 1'b1; end
37.
38.      9: // Read and store R7
39.      if( iDone ) begin D2[7:0] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
40.      else begin isCall[0] <= 1'b1; end
41.
42.      10: // Disable cs
43.      begin rCS <= 1'b1; i <= i + 1'b1; end
44.
45.      11,12: // Send free clock
46.      if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
47.      else begin isCall[0] <= 1'b1; D1 <= 8'hFF; end
48.
49.      13: // Disable cs, generate done signal
50.      begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
51.
52.      14:
53.      begin isDone <= 1'b0; i <= 4'd0; end

```

代码 25.1

如代码 25.1 所示，步骤 0 给足 8 个时钟，步骤 1 拉低 CS，步骤 2 再给足 8 个时钟。步骤 3 准备命令 CMD8，步骤 4 则重复 100 次写命令，直至第 4 字节的反馈数据为 8'h01 为止，否则准备错误信息，然后跳转步骤 13。步骤 5 暂存第一字节的反馈数据，步骤 6~9 则是执行读写并且暂存接续 4 个字节的反馈数据。步骤 10 拉高 CS，步骤 11~12 则给足 8 个时钟。步骤 13 拉高 CS 之余也产生完成信号。

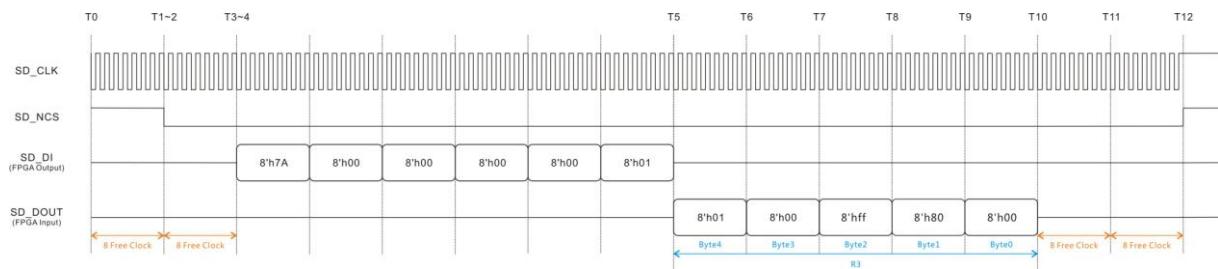


图 25.2 CMD58 的理想时序图 (待机状态)。

图 25.2 是 CMD58 的理想时序图。T0 之际，主机拉高 CS 并且给足 8 个时钟。T1~2 之际，主机拉低 CS 并且给足 8 个时钟。T3~4 之际，主机发送命令 CMD58—{ 8'h7A, 32'd0, 8'h01 }。T5 之际，SD 卡完成接收并且开始反馈数据 R3，主机也在 T6~9 期间读取它们。反馈数据 R3 与 R7 一样，它们都是由 5 个字节组成，其中第 4 字节也类似 R1，内容为 8'h01 表示 SD 卡还处于待机状态。

除此之外，R3 的第 3 字节是有意义的，而接续的 3 个字节只是哈拉哈拉而已。R3 第 3 字节的位意义如表 25.1 所示：

表 25.1 R3 第 3 字节的位意义。

R3 的第 3 字节							
[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
<u>Busy</u>	CCS	无视	无视	无视	无视	无视	无视

如表 25.1 所示，第 7 位为 0 表示 SD 卡处于“忙状态”或者“待机状态”，反之就是“传输状态”。第 6 位 CCS 为 1 表示 SD 卡支持高容量，反之亦然。余下内容笔者就无视了，爱八卦的朋友请自行查阅手册。

T10~T11 之际，主机给足 8 个时钟，然后主机在 T12 拉高 CS。对此，Verilog 可以这样描述，结果如代码 25.2 所示：

```

1. case( i )
2.
3.   0: // Send free clock
4.     if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
5.   else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
6.
7.   1: // Enable cs
8.     begin rCS <= 1'b0; i <= i + 1'b1; end
9.
10.  2: // Send free clock
11.    if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
12.    else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
13.
14.
15.  3: // prepare cmd 58
16.    begin D4 <= { 8'h7A,32'd0,8'h01 }; i <= i + 1'b1; end
17.
18.  4: // Try 100 time, ready error code.
19.    if( C1 == 10'd100 ) begin D2[7:0] <= CMD58ERR; C1 <= 16'd0; i <= 4'd12; end
20.    else if( (iDone && iData != 8'h01) ) begin isCall[1] <= 1'b0; C1 <= C1 + 1'b1; end
21.    else if( (iDone && iData == 8'h01) ) begin isCall[1] <= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
22.    else isCall[1] <= 1'b1;
23.
24.  5: // Store R3
25.    begin D2[39:32] <= iData; i <= i + 1'b1; end
26.
27.  6: // Read and store R3
28.    if( iDone ) begin D2[31:24] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end

```

```

29.      else begin isCall[0] <= 1'b1; end
30.
31.      7: // Read and store R3
32.      if( iDone ) begin D2[23:16] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
33.      else begin isCall[0] <= 1'b1; end
34.
35.      8: // Read and store R3
36.      if( iDone ) begin D2[15:8] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
37.      else begin isCall[0] <= 1'b1; end
38.
39.      9: // Read and store R3
40.      if( iDone ) begin D2[7:0] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
41.      else begin isCall[0] <= 1'b1; end
42.
43.      10,11: // Send free clock
44.      if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
45.      else begin isCall[0] <= 1'b1; D1 <= 8'hFF; end
46.
47.      12: // Disable cs, generate done signal
48.      begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
49.
50.      13:
51.      begin isDone <= 1'b0; i <= 4'd0; end

```

代码 25.2

如代码 25.2 所示，步骤 0 给足 8 个时钟，步骤 1 拉低 CS，步骤 2 则再给足 8 个时钟。步骤 3 准备命令 CMD58，步骤 4 则重复 100 次写命令，直至第 4 字节的反馈数据为 8'h01 为止，否则准备错误信息，并且跳转步骤 12。步骤 5 暂存第 4 字节的内容，步骤 6~9 则是读取并且暂存接续的内容。步骤 10~11 分别给足 8 个时钟，然后步骤 12~13 拉高 CS 之余也产生完成信号。

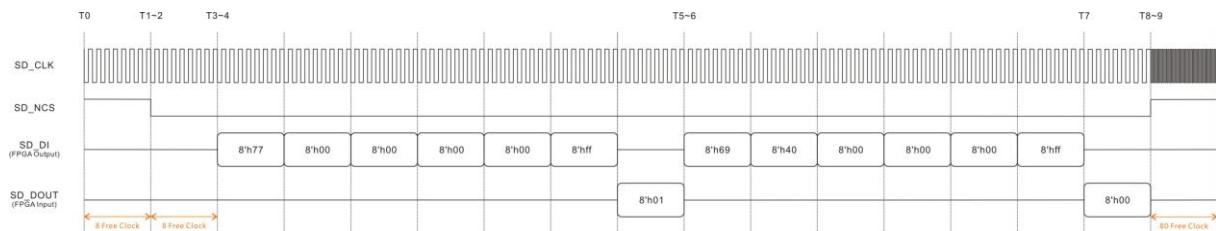


图 25.3 CMD55+ACMD41 的理想时序图。

图 25.3 是 CMD55+ACMD41 的理想时序图。ACMD $\times\times$ 是版本 SDV2 才有的扩展命令，凡是发送扩展命令之前，主机必须事先发送命令 CMD55 示意 SD 卡下一个命令为扩展命令。T0 之际，主机拉高 CS 并且给足 8 个时钟。T1~2 之际，主机拉低 CS 并且给足 8 个时钟。T3~4 之际主机发送命令 CMD55—{ 8'h77,32'd0,8'hff }，然后 SD 卡会返回数

据 8'h01 以示接收成功。

T5~6 之际，主机发送命令 ACMD41—{ 8'h69, 8'h40, 24'd0,8'hff }，然后 SD 卡在 T7 接收并且反馈数据 8'h00 以示接收成功，同时也告诉主机 SD 卡的当前状态已经切至“传输状态”。T8~9 之际，主机拉高 CS 并且给足 80 个结束时钟。在此笔者需要补充一下，命令 ACMD41 的 8'h40，亦即 8'b0100_0000，恰好针对 R3 第三字节的标志位 CCS。简言之，命令 ACMD41 的作用是手动为 SD 卡设置 CCS 标志位。

对此，Verilog 的描述结果如代码 25.3 所示：

```
1.  case( i )
2.
3.    0: // Send free clock
4.      if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
5.      else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
6.
7.    1: // Enable cs
8.      begin rCS <= 1'b0; i <= i + 1'b1; end
9.
10.   2: // Send free clock
11.     if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
12.     else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
13.
14.   3: // Prepare cmd55
15.     begin D4 <= { 8'h77,32'd0,8'hff }; i <= i + 1'b1; end
16.
17.   4: // Send and store R1
18.     if( iDone ) begin D2[39:32] <= iData; isCall[1] <= 1'b0; i <= i + 1'b1; end
19.     else isCall[1] <= 1'b1;
20.
21.   5: // Prepare acmd41
22.     begin D4 <= { 8'h69,8'h40,24'd0,8'hff }; i <= i + 1'b1; end
23.
24.   6: // Send and store R1
25.     if( iDone ) begin D2[31:24] <= iData; isCall[1] <= 1'b0; i <= i + 1'b1; end
26.     else isCall[1] <= 1'b1;
27.
28.   7: // Try 1000 times, ready error code.
29.     if( C1 == 16'd1000 ) begin D2[7:0] <= CMD41ERR; C1 <= 16'd0; i <= 4'd10; end
30.     else if( iData != 8'h00 ) begin C1 <= C1 + 1'b1; i <= 4'd3; end
31.     else if( iData == 8'h00 ) begin C1 <= 16'd0; i <= i + 1'b1; end
32.
33.   8: // Disable cs
```

```

34.      begin rCS <= 1'b1; i <= i + 1'b1; end
35.
36.      9: // Send free clock
37.      if( C1 == 10 ) begin C1 <= 16'd0; i <= i + 1'b1; end
38.      else if( iDone ) begin isCall[0] <= 1'b0; C1 <= C1 + 1'b1; end
39.      else begin isCall[0] <= 1'b1; D1 <= 8'hFF; end
40.
41.      10: // Disable cs, generate done signal
42.      begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
43.
44.      11:
45.      begin isDone <= 1'b0; i <= 4'd0; end

```

代码 25.3

如代码 25.3 所示，步骤 0 拉高 CS 并且给足 8 个时钟，步骤 1 拉低 CS，步骤 2 则给足 8 个时钟。步骤 3 准备命令 CMD55，然后再步骤 4 将其写入，反馈数据则暂存至 D2[39:32]，步骤 5 准备命令 ACMD41，并且手动设置 CCS 标示位—8'h40，然后在步骤 6 将其写入，事后才将反馈数据暂存 D2[31:24]当中。

步骤 7 检测 ACMD41 的反馈数据，如果内容不为 8'h00 便跳转步骤 3，并且重复 1000 次同样的操作，直至反馈数据为 8'h00 为止，否则准备错误信息，然后跳转步骤 10。简单来说，步骤 3~7 组成简单的 do ... while 循环，其中步骤 7 用来控制循环。步骤 8 拉高 CS，然后步骤 9 给足 80 个结束时钟。步骤 10~11 拉高 CS 之余也产生完成信号。

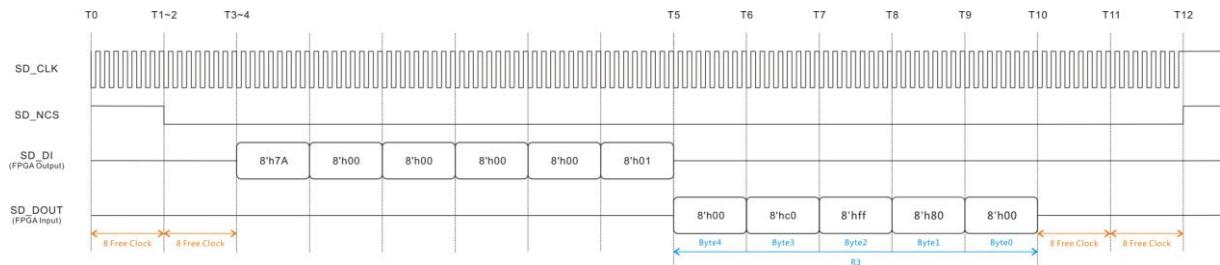


图 25.4 CMD58 的理想时序图 (传输状态) 。

当主机成功写入命令 CMD55+ACMD41 的时候，CMD58 的反馈数据 R3 也会跟着发生变化。如图 25.4 所示，T0 之际主机拉高 CS 之余也给足 8 个时钟。T1~T2 之际，主机拉低 CS 也给足 8 个时钟。T3~T4 之际，主机发送命令 CMD58。T5 之际，SD 卡接收以后便会反馈 5 个字节的 R3，其中字节 4 与字节 3 的内容已经发生变化。

字节 4 为 8'h00，表示 SD 卡已经处于传输状态。字节 3 为 8'hC0（也是 8'b1100_0000），表示 SD 卡结束忙碌之余，它也成功认识自己是大容量的储存器。对此，Verilog 可以这样描述，结果如代码 25.4 所示：

```

1. .....

```

```

2.   4: // Try 100 times, ready error code
3.   if( C1 == 10'd100 ) begin D2[7:0] <= CMD58ERR; C1 <= 16'd0; i <= 4'd12; end
4.   else if( (iDone && iData != 8'h00) ) begin isCall[1] <= 1'b0; C1 <= C1 + 1'b1; end
5.   else if( (iDone && iData == 8'h00) ) begin isCall[1] <= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
6.   else isCall[1] <= 1'b1;
7.   .....

```

代码 25.4

如代码 25.4 所示，代码 25.4 与代码 25.2 的内容大同小异，除了第 4~5 行 8'h00 以外。

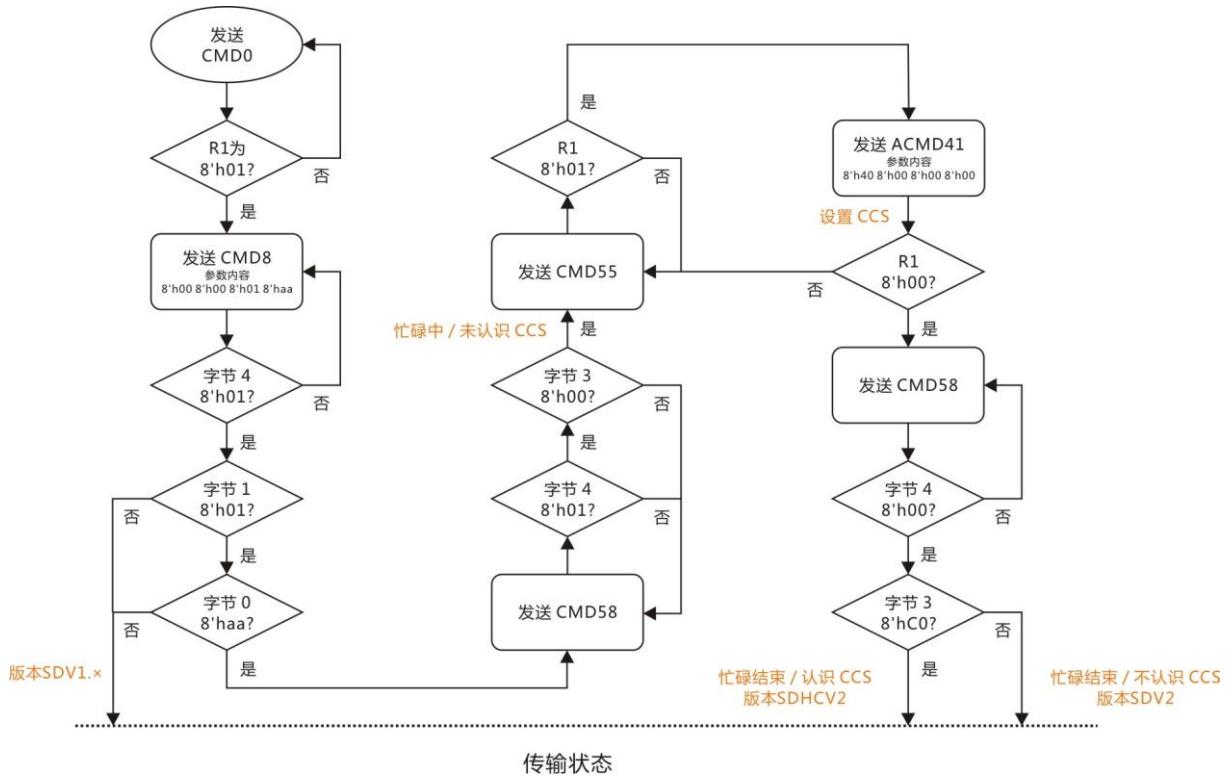


图 25.5 版本 SDV2 与 SDHCV2 初始化的流程图。

图 25.5 是 SD 卡版本 SDV2 与版本 SDHCV2 的初始化流程图，而目前提条件是 SD 卡是健康又不损坏的硬件资源。主机首先发送 CMD0，如果 SD 卡反馈 8'h01，SD 卡便进入待机模式。再者，主机发送 CMD8 并且跟随一些参数内容—{ 8'h00 , 8'h00 , 8'h01 , 8'haa }，如果反馈内容为 8'h01 流程便继续。主机紧接着检测字节 1 和字节 0 的内容是否为 8'h01 与 8'haa，如果任一不是便可可知那是版本 SDV1.x 的 SD 卡。换之，如果字节 1 和字节 0 的内容是 8'h01 与 8'haa。那么继续流程。

主机随后发送 CMD58 要求 SD 卡反馈内部状态。如果字节 4 为 8'h01 便继续读取字节 3，如果字节 3 为 8'h00，则表示 SD 卡不仅忙碌中，而且还未认识 CCS 标示。再来主机发送 CMD55+ACMD41，并且伴随参数 { 8'h40 , 8'h00 , 8'h00 , 8'h00 }。如果反馈内容为 8'h00 就表示 CCS 的设置动作不仅成功，而且 SD 卡也正在步入传输模式。

最后，主机发送 CMD58 再次要求 SD 卡反馈内容状态。如果反馈内容字节 4 为 8'h00

便继续读取字节 3，如果字节 3 的内容为 8'hC0 便可确认那是版本 SDHCV2 的 SD 卡。反之，字节 3 为 8'h80 便可确认那是版本 SDV2 的 SD 卡。完后，SD 卡便全面进入传输模式，初始化过程也因此结束，真是可喜可贺！

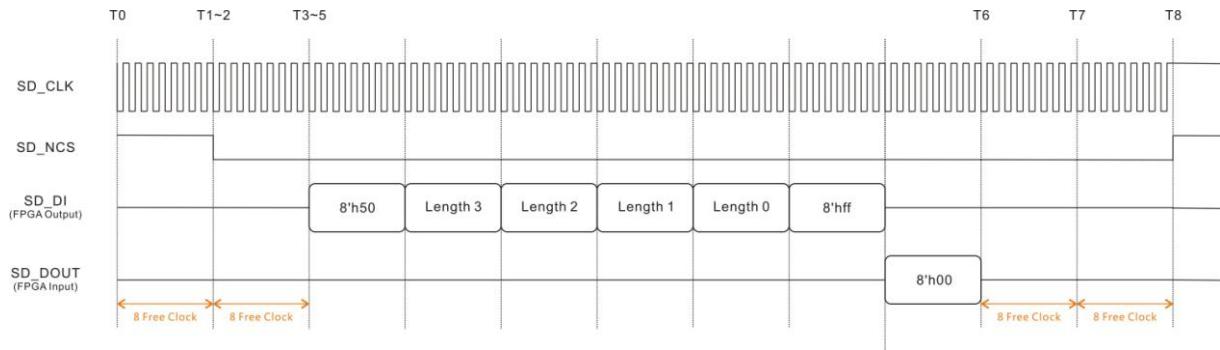


图 25.6 CMD16 的理想时序图。

CMD16 的作用主要是设置多字节读写的长度，默认下一次性多字节读写设置为 512。SD 卡进化为版本 SDV2 或者 SDHCV2 以后，我们也知道它们不用偏移地址，因此多字节读写的长度才可以更改。当然，更改内容也不是任由我们随心所欲 … 根据手册，更改内容要么是 512（默认），要么是 1024，要么就是 2048。

图 25.6 是 CMD16 的理想时序图，而 CMD16 也是可有可无的可怜虫，不过笔者还是大发慈悲介绍它吧。T0 之际，主机拉高 CS 并且给足 8 个时钟。T1~2 之际，主机拉低 CS 至于它也给足 8 个时钟。T3~5 之际，主机发送命令 CMD16—{ 8'h50,32'd512,8'cff }，其中 32'd512 示意多字节读写的长度为 512，结果 SD 卡返回 8'h00 以示了解。T6~7 之际，主机分别给足 8 个结束时钟，然后再 T8 拉高 CS。

对此，Verilog 的描述结果如代码 25.5 所示：

```

1. case( i )
2.
3.     0: // Send free clock
4.         if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
5.         else begin isCall[0] <= 1'b1; D1 <= 8'cff; end
6.
7.     1: // Enable CS
8.         begin rCS <= 1'b0; i <= i + 1'b1; end
9.
10.    2: // Send free clock
11.        if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
12.        else begin isCall[0] <= 1'b1; D1 <= 8'cff; end
13.
14.    3: // Prepare cmd 16, 512 block length
15.        begin D4 <= { 8'h50,32'd512,8'cff }; i <= i + 1'b1; end

```

```

16.
17.    4: // Try 100 times, ready error code.
18.    if( C1 == 10'd100 ) begin D2[7:0] <= CMD16ERR; C1 <= 16'd0; i <= 4'd8; end
19.    else if( (iDone && iData != 8'h00) ) begin isCall[1] <= 1'b0; C1 <= C1 + 1'b1; end
20.    else if( (iDone && iData == 8'h00) ) begin isCall[1] <= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
21.    else isCall[1] <= 1'b1;
22.
23.    5: // Ready OK code
24.    begin D2[7:0] <= CMD16OK; i <= i + 1'b1; end
25.
26.    6,7: // Send free clock
27.    if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
28.    else begin isCall[0] <= 1'b1; D1 <= 8'hFF; end
29.
30.    8: // Disable cs , generate done signal
31.    begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
32.
33.    9:
34.    begin isDone <= 1'b0; i <= 4'd0; end

```

代码 25.5

如代码 25.5 所示，步骤 0 拉高 CS 之余也给足 8 个时钟，步骤 1 拉低 CS，步骤 2 再给足 8 个时钟。步骤 3 准备 CMD16 并且伴随参数 32'd512，内容意指多字节读写的长度为 512。随后，步骤 4 将其写入 100 次，直至反馈数据为 8'h00 为止，否则准备错误信息，然后跳转步骤 8。步骤 5 准备成功信息，步骤 6~7 分别给足 8 个时钟，然后步骤 8~9 拉高 CS 之余也产生完成信号。

虽然版本 SDV2 与版本 SDHCV2 多少也有影响 CMD24 与 CMD17，不过不打紧，事后笔者会继续解释的 … 所以暂请读者憋着蛋蛋一下。上述内容理解完毕以后，我们便可以开始建模了。

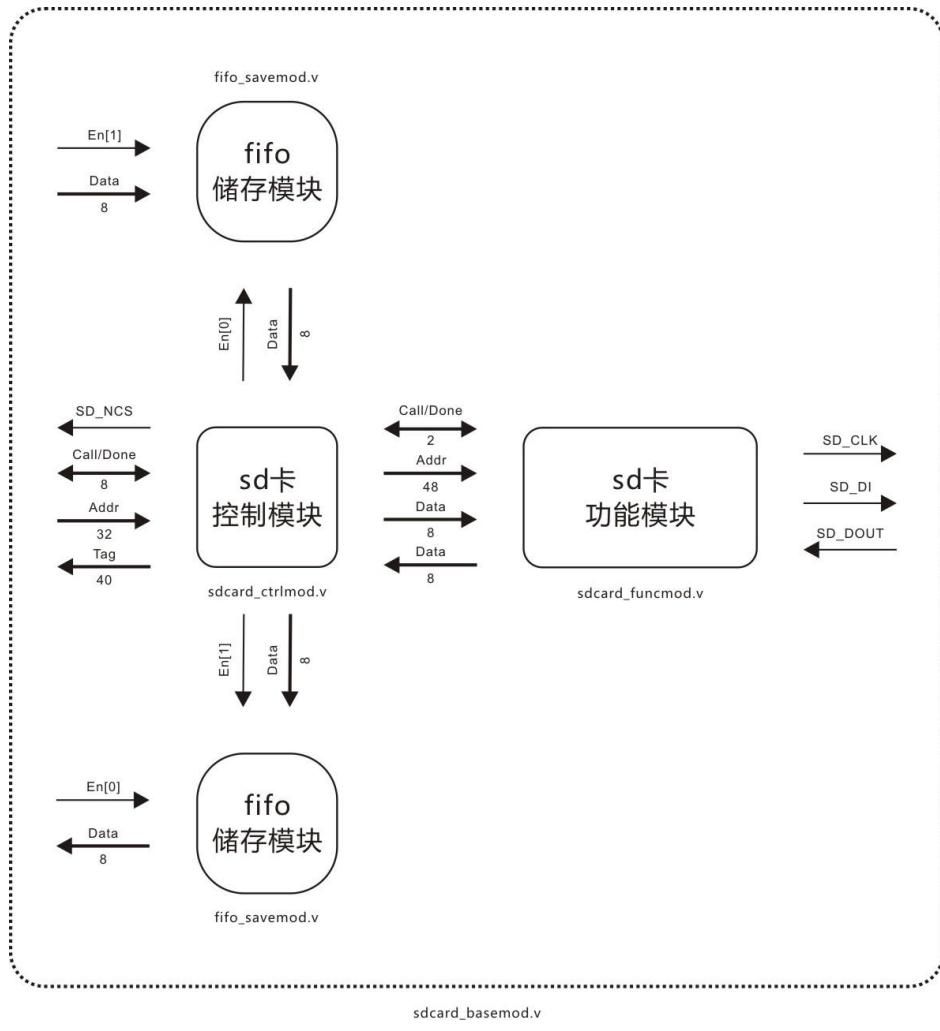


图 25.7 SD 卡基础模块的建模图。

图 25.7 是 SD 卡基础模块的建模图 ... 目视下，更改的内容却不多，如 SD 卡控制模块的 oTag 有 40 位宽，iAddr 有 32 位宽，Call/Done 则有 8 位宽，位分配如表 25.2 所示：

表 25.2 Call/Done 的位宽内容。

位分配	说明
Call[7]	调用 CMD24，写命令
Call[6]	调用 CMD17，读命令
Call[5]	调用 CMD16，设置读写长度
Call[4]	调用 CMD58，读取 SD 卡状态（传输状态）
Call[3]	调用 CMD55+ACMD41，设置 CCS 标示位
Call[2]	调用 CMD58，读取 SD 卡状态（待机状态）
Call[1]	调用 CMD8，配置物理参数
Call[0]	调用 CMD0，复位 SD 卡

sdcards_funcmod.v



图 25.8 SD 卡功能模块的建模图。

图 25.8 是 SD 卡功能模块的建模图，这家伙相较实验二十四的兄弟差别并不大，不过具体内容还是直接窥视代码吧。

```
19.    always @ ( posedge CLOCK or negedge RESET )
20.        if( !RESET )
21.            begin
22.                isFull <= FLCLK;
23.                isHalf <= FLHALF;
24.                isQuarter <= FLQUARTER;
25.            end
26.        else if( iCmd[47:40] == 8'h50 && isDone )
27.            begin
28.                isFull <= FHCLK;
29.                isHalf <= FHHALF;
30.                isQuarter <= FHQUARTER;
31.            end
```

以上内容为周边操作，改变非内容是 26 行的 8'h50，该行表示 CMD16 调用完毕以后便更动速率。

sdcards_ctrlmod.v

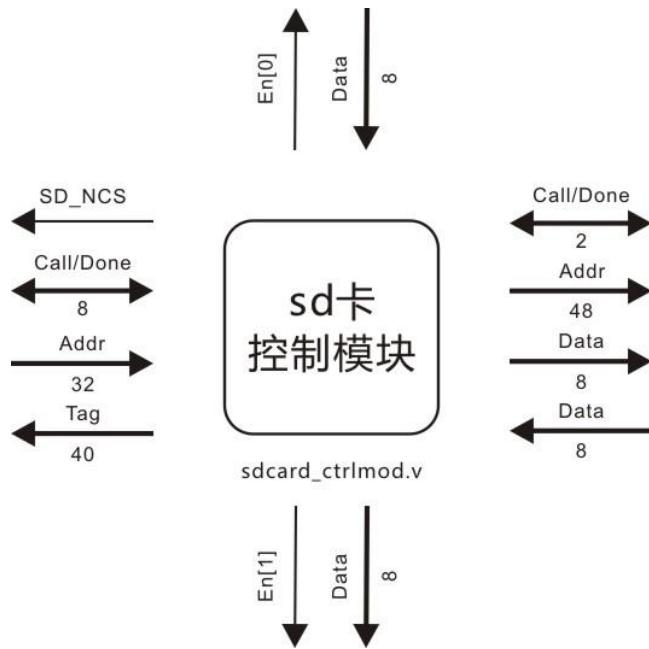


图 25.9 SD 卡控制模块的建模图。

如图 25.9 所示，该控制模块依然还是一只长满箭头的刺猬，不过改变内容也只有左边的 Call/Done 信号，Addr 信号，还有 Tag 信号而已，具体内容我们还是来看代码吧。

```

1. module sdcard_ctrlmod
2. (
3.     input CLOCK, RESET,
4.     output SD_NCS,
5.
6.     input [7:0]iCall,
7.     output oDone,
8.     input [31:0]iAddr,
9.     output [39:0]oTag,
10.
11.    output [1:0]oEn, // [1] Write [0] Read
12.    input [7:0]iDataFF,
13.    output [7:0]oDataFF,
14.
15.    output [1:0]oCall,
16.    input iDone,
17.    output [47:0]oAddr,
18.    input [7:0]iData,
19.    output [7:0]oData
20. );

```

以上内容为相关的出入端声明。

```

21. parameter CMD0ERR = 8'hA1, CMD0OK = 8'hA2, CMD1ERR = 8'hA3, CMD1OK = 8'hA4;
22. parameter CMD24ERR = 8'hA5, CMD24OK = 8'hA6,  CMD17ERR = 8'hA9, CMD17OK = 8'hAA;
23. parameter CMD16ERR = 8'hA7,CMD16OK = 8'hA8;
24. parameter CMD8ERR = 8'hB1, CMD41ERR = 8'hC0, CMD58ERR = 8'hC1;
25. parameter SDV2 = 8'hD1, SDV2HC = 8'hD2;
26. parameter T1MS = 16'd10;
27.

```

以上内容为失败信息与成功信息的常量声明，此外也有版本信息还有延迟常量。

```

28.     reg [3:0]i;
29.     reg [15:0]C1;
30.     reg [7:0]D1,D3; // D1 WrData, D2 FbData, D3 RdData
31.     reg [39:0]D2;
32.     reg [47:0]D4;      // D4 Cmd
33.     reg [1:0]isCall,isEn;
34.     reg rCS;
35.     reg isDone;
36.
37.     always @ ( posedge CLOCK or negedge RESET )
38.         if( !RESET )
39.             begin
40.                 i <= 4'd0;
41.                 C1 <= 16'd0;
42.                 { D1,D3 } <= { 8'd0,8'd0 };
43.                 D2 <= 40'd0;
44.                 D4 <= 48'd0;
45.                 { isCall, isEn } <= { 2'd0,2'd0 };
46.                 rCS <= 1'b1;
47.             end

```

以上内容为相关的寄存器声明还有复位操作，注意那只暂存反馈信息的 D2 已经扩展至 40 位宽。

```

48.         else if( iCall[7] ) // write block
49.             case( i )
50.
51.                 0: // Enable cs and prepare cmd 24
52.                 begin rCS <= 1'b0; D4 = { 8'h58, iAddr, 8'hFF }; i <= i + 1'b1; end
53.
54.                 1: // Try 100 times , 8'h03 for error code.
55.                 if( C1 == 100 ) begin D2[7:0] <= CMD24ERR; C1 <= 16'd0; i <= 4'd14; end
56.                 else if( iDone && iData != 8'h00) begin isCall[1] <= 1'b0; C1 <= C1 + 1'b1; end

```

```

57.           else if( iDone && iData == 8'h00 ) begin isCall[1] <= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
58.           else isCall[1] <= 1'b1;
59.
60.           2: // Send 800 free clock
61.           if( C1 == 100 ) begin C1 <= 16'd0; i <= i + 1'b1; end
62.           else if( iDone ) begin isCall[0] <= 1'b0; C1 <= C1 + 1'b1; end
63.           else isCall[0] <= 1'b1;
64.
65.           3: // Send Call Byte 0xfe
66.           if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
67.           else begin isCall[0] <= 1'b1; D1 <= 8'hFE; end
68.
69.           *****/
70.
71.           4: // Pull up read req.
72.           begin isEn[0] <= 1'b1; i <= i + 1'b1; end
73.
74.           5: // Pull down read req.
75.           begin isEn[0] <= 1'b0; i <= i + 1'b1; end
76.
77.           6: // Write byte read from fifo
78.           if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
79.           else begin isCall[0] <= 1'b1; D1 <= iDataFF; end
80.
81.           7: // Repeat 512 times
82.           if( C1 == 10'd511 ) begin C1 <= 16'd0; i <= i + 1'b1; end
83.           else begin C1 <= C1 + 1'b1; i <= 4'd4; end
84.
85.           *****/
86.
87.           8: // Write 1st CRC
88.           if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
89.           else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
90.
91.           9: // Write 2nd CRC
92.           if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
93.           else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
94.
95.           10: // Read respond
96.           if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
97.           else begin isCall[0] <= 1'b1; end
98.
99.           11: // if not 8'h05, write block faild, 8'h03 for error code.

```

```

100.         if( (iData & 8'h1F) != 8'h05 ) begin D2[7:0] <= CMD24ERR; i <= 4'd14; end
101.         else i <= i + 1'b1;
102.
103.         12: // Wait until sdcard free
104.         if( iDone && iData == 8'hff ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
105.         else if( iDone ) begin isCall[0] <= 1'b0; end
106.         else begin isCall[0] <= 1'b1; end
107.
108.         *****/
109.
110.         13: // Read OK code;
111.         begin D2[7:0] <= CMD24OK; i <= i + 1'b1; end
112.
113.         14: // Disable cs and generate done signal
114.         begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
115.
116.         15:
117.         begin isDone <= 1'b0; i <= 4'd0; end
118.
119.     endcase

```

以上内容为命令 CMD24 , 注意第 52 行的写地址 , 地址没有左移 9 位。

```

120.     else if( iCall[6] ) // read block
121.         case( i )
122.
123.             0: // Enable cs and prepare cmd 17;
124.             begin rCS <= 1'b0; D4 <= { 8'h51, iAddr, 8'hff }; i <= i + 1'b1; end
125.
126.             1: // Try 100 times, ready error code;
127.             if( C1 == 100 ) begin D2[7:0] <= CMD17ERR; C1 <= 16'd0; i <= 4'd11; end
128.             else if( iDone && iData != 8'h00 ) begin isCall[1] <= 1'b0; C1 <= C1 + 1'b1; end
129.             else if( iDone && iData == 8'h00 ) begin isCall[1] <= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
130.             else isCall[1] <= 1'b1;
131.
132.             2: // Waiting read ready 8'hfe
133.             if( iDone && iData == 8'hfe ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
134.             else if( iDone && iData != 8'hfe ) begin isCall[0] <= 1'b0; end
135.             else isCall[0] <= 1'b1;
136.
137.             *****/
138.
139.             3: // Read 1 byte form sdcard

```

```

140.         if( iDone ) begin D3 <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
141.         else begin isCall[0] <= 1'b1; end
142.
143.         4: // Pull up write req.
144.         begin isEn[1] <= 1'b1; i <= i + 1'b1; end
145.
146.         5: // Pull down write req.
147.         begin isEn[1] <= 1'b0; i <= i + 1'b1; end
148.
149.         6: // Repeat 512 times
150.         if( C1 == 10'd511 ) begin C1 <= 16'd0; i <= i + 1'b1; end
151.         else begin C1 <= C1 + 1'b1; i <= 4'd3; end
152.
153.         *****/
154.
155.         7,8: // Read 1st and 2nd byte CRC
156.         if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
157.         else isCall[0] <= 1'b1;
158.
159.         9: // Disable CS, ready OK code.
160.         begin D2[7:0] <= CMD17OK; rCS <= 1'b1; i <= i + 1'b1; end
161.
162.         *****/
163.
164.         10: // Send 8 free clock
165.         if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
166.         else begin isCall[0] <= 1'b1; D1 <= 8'hFF; end
167.
168.         11: // Disable cs, generate done signal
169.         begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
170.
171.         12:
172.         begin isDone <= 1'b0; i <= 4'd0; end
173.
174.     endcase

```

以上内容为命令 CMD17 , 注意第 124 行的读地址 , 地址没有左移 9 位。

```

175.     else if( iCall[5] ) // cmd16
176.         case( i )
177.
178.             0: // Send free clock
179.             if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end

```

```

180.           else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
181.
182.           1: // Enable CS
183.           begin rCS <= 1'b0; i <= i + 1'b1; end
184.
185.           2: // Send free clock
186.           if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
187.           else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
188.
189.           *****/
190.
191.           3: // Prepare cmd 16, 512 block length
192.           begin D4 <= { 8'h50,32'd512,8'hff }; i <= i + 1'b1; end
193.
194.           4: // Try 100 times, ready error code.
195.           if( C1 == 10'd100 ) begin D2[7:0] <= CMD16ERR; C1 <= 16'd0; i <= 4'd8; end
196.           else if( (iDone && iData != 8'h00) ) begin isCall[1] <= 1'b0; C1 <= C1 + 1'b1; end
197.           else if( (iDone && iData == 8'h00) ) begin isCall[1] <= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
198.           else isCall[1] <= 1'b1;
199.
200.           5: // Ready OK code
201.           begin D2[7:0] <= CMD16OK; i <= i + 1'b1; end
202.
203.           *****/
204.
205.           6,7: // Send free clock
206.           if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
207.           else begin isCall[0] <= 1'b1; D1 <= 8'hFF; end
208.
209.           8: // Disable cs , generate done signal
210.           begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
211.
212.           9:
213.           begin isDone <= 1'b0; i <= 4'd0; end
214.
215.       endcase

```

以上内容为命令 CMD16。

```

216.       else if( iCall[4] ) // cmd58 transfer mode
217.           case( i )
218.
219.               0: // Send free clock

```

```

220.           if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
221.           else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
222.
223.           1:// Enable cs
224.           begin rCS <= 1'b0; i <= i + 1'b1; end
225.
226.           2:// Send free clock
227.           if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
228.           else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
229.
230.           *****/
231.
232.           3:// Prepare cmd 58
233.           begin D4 <= { 8'h7A,32'd0,8'h01 }; i <= i + 1'b1; end
234.
235.           4:// Try 100 times, ready error code
236.           if( C1 == 10'd100 ) begin D2[7:0] <= CMD58ERR; C1 <= 16'd0; i <= 4'd12; end
237.           else if( (iDone && iData != 8'h00) ) begin isCall[1] <= 1'b0; C1 <= C1 + 1'b1; end
238.           else if( (iDone && iData == 8'h00) ) begin isCall[1] <= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
239.           else isCall[1] <= 1'b1;
240.
241.           5:// Store R3
242.           begin D2[39:32] <= iData; i <= i + 1'b1; end
243.
244.           6:// Read and store R3
245.           if( iDone ) begin D2[31:24] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
246.           else begin isCall[0] <= 1'b1; end
247.
248.           7:// Read and store R3
249.           if( iDone ) begin D2[23:16] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
250.           else begin isCall[0] <= 1'b1; end
251.
252.           8:// Read and store R3
253.           if( iDone ) begin D2[15:8] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
254.           else begin isCall[0] <= 1'b1; end
255.
256.           9:// Read and store R3
257.           if( iDone ) begin D2[7:0] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
258.           else begin isCall[0] <= 1'b1; end
259.
260.           *****/
261.
262.           10,11:// Send free clock

```

```

263.         if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
264.         else begin isCall[0] <= 1'b1; D1 <= 8'hFF; end
265.
266.         12: // Disable cs, generate done signal
267.         begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
268.
269.         13:
270.         begin isDone <= 1'b0; i <= 4'd0; end
271.
272.     endcase

```

以上内容为命令 CMD58 (传输状态)。

```

273.     else if( iCall[3] ) // cmd55 + acmd41
274.         case( i )
275.
276.             0: // Send free clock
277.             if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
278.             else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
279.
280.             1: // Enable cs
281.             begin rCS <= 1'b0; i <= i + 1'b1; end
282.
283.             2: // Send free clock
284.             if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
285.             else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
286.
287.             *****/
288.
289.             3: // Prepare cmd55
290.             begin D4 <= { 8'h77,32'd0,8'hff }; i <= i + 1'b1; end
291.
292.             4: // Send and store R1
293.             if( iDone ) begin D2[39:32] <= iData; isCall[1] <= 1'b0; i <= i + 1'b1; end
294.             else isCall[1] <= 1'b1;
295.
296.             5: // Prepare acmd41
297.             begin D4 <= { 8'h69,8'h40,24'd0,8'hff }; i <= i + 1'b1; end
298.
299.             6: // Send and store R1
300.             if( iDone ) begin D2[31:24] <= iData; isCall[1] <= 1'b0; i <= i + 1'b1; end
301.             else isCall[1] <= 1'b1;
302.

```

```

303.    7: // Try 1000 times, ready error code.
304.    if( C1 == 16'd1000 ) begin D2[7:0] <= CMD41ERR; C1 <= 16'd0; i <= 4'd10; end
305.    else if( iData != 8'h00 ) begin C1 <= C1 + 1'b1; i <= 4'd3; end
306.    else if( iData == 8'h00 ) begin C1 <= 16'd0; i <= i + 1'b1; end
307.
308.    *****/
309.
310.    8: // Disable cs
311.    begin rCS <= 1'b1; i <= i + 1'b1; end
312.
313.    9: // Send free clock
314.    if( C1 == 10 ) begin C1 <= 16'd0; i <= i + 1'b1; end
315.    else if( iDone ) begin isCall[0] <= 1'b0; C1 <= C1 + 1'b1; end
316.    else begin isCall[0] <= 1'b1; D1 <= 8'hFF; end
317.
318.    10: // Disable cs, generate done signal
319.    begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
320.
321.    11:
322.    begin isDone <= 1'b0; i <= 4'd0; end
323.
324.  endcase

```

以上内容为命令 CMD55+ACMD41。

```

325.    else if( iCall[2] ) // cmd58 idle mode
326.        case( i )
327.
328.            0: // Send free clock
329.            if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
330.            else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
331.
332.            1: // Enable cs
333.            begin rCS <= 1'b0; i <= i + 1'b1; end
334.
335.            2: // Send free clock
336.            if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
337.            else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
338.
339.        *****/
340.
341.    3: // prepare cmd 58
342.    begin D4 <= { 8'h7A,32'd0,8'h01 }; i <= i + 1'b1; end

```

```

343.
344.        4: // Try 100 time, ready error code.
345.        if( C1 == 10'd100 ) begin D2[7:0] <= CMD58ERR; C1 <= 16'd0; i <= 4'd12; end
346.        else if( (iDone && iData != 8'h01)  ) begin isCall[1] <= 1'b0; C1 <= C1 + 1'b1; end
347.        else if( (iDone && iData == 8'h01)  ) begin isCall[1] <= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
348.        else isCall[1] <= 1'b1;
349.
350.        5: // Store R3
351.        begin D2[39:32] <= iData; i <= i + 1'b1; end
352.
353.        6: // Read and store R3
354.        if( iDone ) begin D2[31:24] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
355.        else begin isCall[0] <= 1'b1; end
356.
357.        7: // Read and store R3
358.        if( iDone ) begin D2[23:16] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
359.        else begin isCall[0] <= 1'b1; end
360.
361.        8: // Read and store R3
362.        if( iDone ) begin D2[15:8] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
363.        else begin isCall[0] <= 1'b1; end
364.
365.        9: // Read and store R3
366.        if( iDone ) begin D2[7:0] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
367.        else begin isCall[0] <= 1'b1; end
368.
369.        *****/
370.
371.        10,11: // Send free clock
372.        if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
373.        else begin isCall[0] <= 1'b1; D1 <= 8'hFF; end
374.
375.        12: // Disable cs, genarate done signal
376.        begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
377.
378.        13:
379.        begin isDone <= 1'b0; i <= 4'd0; end
380.
381.        endcase

```

以上内容为命令 CMD58 (待机状态)。

```

382.        else if( iCall[1] )// Cmd8

```

```

383.         case( i )
384.
385.             0:// Send free clock
386.             if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
387.             else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
388.
389.             1:// Enable cs
390.             begin rCS <= 1'b0; i <= i + 1'b1; end
391.
392.             2:// Send free clock
393.             if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
394.             else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
395.
396.             *****/
397.
398.             3:// Prepare Cmd8 // 8'h01 = 2.7~3.6v, 8'hA0A0 default check pattern
399.             begin D4 <= { 8'h48,16'd0,8'h01,8'hAA,8'h87 }; i <= i + 1'b1; end
400.
401.             4:// Try 100 times, ready error code.
402.             if( C1 == 10'd100 ) begin D2[7:0] <= CMD8ERR; C1 <= 16'd0; i <= 4'd13; end
403.             else if( (iDone && iData != 8'h01) ) begin isCall[1]<= 1'b0; C1 <= C1 + 1'b1; end
404.             else if( (iDone && iData == 8'h01) ) begin isCall[1]<= 1'b0; C1 <= 16'd0; i <= i + 1'b1; end
405.             else isCall[1]<= 1'b1;
406.
407.             5:// Store R7
408.             begin D2[39:32] <= iData; i <= i + 1'b1; end
409.
410.             6:// Read and store R7
411.             if( iDone ) begin D2[31:24] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
412.             else begin isCall[0] <= 1'b1; end
413.
414.             7:// Read and store R7
415.             if( iDone ) begin D2[23:16] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
416.             else begin isCall[0] <= 1'b1; end
417.
418.             8:// Read and store R7
419.             if( iDone ) begin D2[15:8] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
420.             else begin isCall[0] <= 1'b1; end
421.
422.             9:// Read and store R7
423.             if( iDone ) begin D2[7:0] <= iData; isCall[0] <= 1'b0; i <= i + 1'b1; end
424.             else begin isCall[0] <= 1'b1; end
425.
```

```

426.           10: // Disable cs
427.           begin rCS <= 1'b1; i <= i + 1'b1; end
428.
429.           *****/
430.
431.           11,12: // Send free clock
432.           if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
433.           else begin isCall[0] <= 1'b1; D1 <= 8'hFF; end
434.
435.           13: // Disable cs, generate done signal
436.           begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
437.
438.           14:
439.           begin isDone <= 1'b0; i <= 4'd0; end
440.
441.       endcase

```

以上内容为命令 CMD8。

```

442.       else if( iCall[0] ) // cmd0
443.           case( i )
444.
445.               0: // Prepare Cmd0
446.               begin D4 <= 48'h40_00_00_00_00_95; i <= i + 1'b1; end
447.
448.               1: // Wait 1MS for warm up;
449.               if( C1 == T1MS -1) begin C1 <= 16'd0; i <= i + 1'b1; end
450.               else begin C1 <= C1 + 1'b1; end
451.
452.               2: // Send free clock
453.               if( C1 == 10'd10 ) begin C1 <= 16'd0; i <= i + 1'b1; end
454.               else if( iDone ) begin isCall[0] <= 1'b0; C1 <= C1 + 1'b1; end
455.               else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
456.
457.               3: // Disable cs
458.               begin rCS <= 1'b0; i <= i + 1'b1; end
459.
460.               4: // Try 200 time, ready error code.
461.               if( C1 == 10'd200 ) begin D2[7:0] <= CMD0ERR; C1 <= 16'd0; i <= 4'd8; end
462.               else if( iDone && iData != 8'h01) begin isCall[1] <= 1'b0; C1 <= C1 + 1'b1; end
463.               else if( iDone && iData == 8'h01 ) begin isCall[1] <= 1'b0; D2<= iData; C1 <= 16'd0; i <= i + 1'b1; end
464.               else isCall[1] <= 1'b1;
465.

```

```

466.      5: // Disable cs
467.          begin rCS <= 1'b1 ; i <= i + 1'b1; end
468.
469.          6: // Send free clock
470.          if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
471.          else begin isCall[0] <= 1'b1; D1 <= 8'hff; end
472.
473.          7: // Ready OK code.
474.          begin D2[7:0] <= CMD0OK; i <= i + 1'b1; end
475.
476.          8: // Disbale cs, generate done signal
477.          begin rCS <= 1'b1; isDone <= 1'b1; i <= i + 1'b1; end
478.
479.          9:
480.          begin isDone <= 1'b0; i <= 4'd0; end
481.
482.      endcase
483.

```

以上内容为命令 CMD0。

```

484.      assign SD_NCS = rCS;
485.      assign oDone = isDone;
486.      assign oTag = D2;
487.      assign oEn = isEn;
488.      assign oDataFF = D3;
489.      assign oCall = isCall;
490.      assign oAddr = D4;
491.      assign oData = D1;
492.
493. endmodule

```

以上内容为相关的驱动声明。

fifo_savemod.v

该模块与实验二十四一样。

sdcardsavemod.v

连线部署的内容请参考图 25.7。此外，相较实验二十四，该模块的修改内容只有部分位宽而已。

```
1. module sdcard_basemod
2. (
3.     input CLOCK, RESET,
4.     input SD_DOUT,
5.     output SD_CLK,
6.     output SD_DI,
7.     output SD_NCS,
8.
9.     input [7:0]iCall,
10.    output oDone,
11.    input [31:0]iAddr,
12.    output [39:0]oTag,
13.
14.    input [1:0]iEn,
15.    input [7:0]iData,
16.    output [7:0]oData
17. );
18. .....
```

修改的内容有第 9 行的 iCall，第 11 行的 iAddr，还有第 12 行的 oTag。

sdcard_demo.v

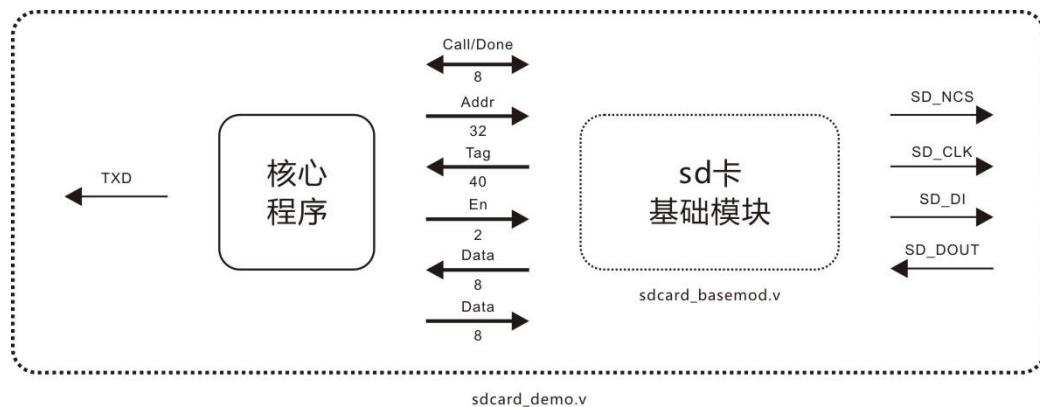


图 25.10 实验二十五的建模图。

图 25.10 是实验二十五的建模图，目视之下的修改内容也是 Call/Done 等信号的位宽而已。不过，核心程序的内容相较实验二十四却有天壤之别，具体内容让我们来看代码吧。

```
1. module sdcard_demo
2. (
3.     input CLOCK,RESET,
4.     output SD_NCS,
5.     output SD_CLK,
6.     input SD_DOUT,
7.     output SD_DI,
8.     output TXD
9. );
```

以上内容为相关的出入端声明。

```
10.    wire DoneU1;
11.    wire [39:0]TagU1;
12.    wire [7:0]DataU1;
13.
14.    sdcard_basemod U1
15.    (
16.        .CLOCK( CLOCK ),
17.        .RESET( RESET ),
18.        .SD_DOUT( SD_DOUT ),
19.        .SD_CLK( SD_CLK ),
20.        .SD_DI( SD_DI ),
21.        .SD_NCS( SD_NCS ),
22.        .iCall( isCall ),
23.        .oDone( DoneU1 ),
24.        .iAddr( D1 ),
25.        .oTag( TagU1 ),
26.        /*****
27.        .iEn( isEn ),
28.        .iData( D2 ),
29.        .oData( DataU1 )
30.    );
31.
```

以上内容为 SD 卡基础模块的实例化。

```
32.    parameter B115K2 = 11'd434, TXFUNC = 6'd48;
33.
34.    reg [5:0]i,Go;
35.    reg [10:0]C1,C2;
36.    reg [31:0]D1;
```

```

37.      reg [7:0]D2;
38.      reg [10:0]T;
39.      reg [7:0]isCall;
40.      reg [1:0]isEn;
41.      reg rTXD;
42.
43.      always @ ( posedge CLOCK or negedge RESET )
44.          if( !RESET )
45.              begin
46.                  { i,Go } <= { 6'd0,6'd0 };
47.                  { C1,C2 } <= { 11'd0,11'd0 };
48.                  { D1,D2,T } <= { 32'd0,8'd0,11'd0 };
49.                  { isCall,isEn } <= { 8'd0,2'd0 };
50.                  rTXD <= 1'b1;
51.              end
52.          else

```

以上内容为相关寄存器声明，复位操作，还有波特率与入口地址的常量声明。

```

53.      case( i )
54.
55.          0: // cmd0
56.          if( DoneU1 ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
57.          else begin isCall[0] <= 1'b1; end
58.
59.          1:
60.          begin T <= { 2'b11, TagU1[7:0], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
61.
62.          ****
63.

```

步骤 0 执行 CMD0，步骤 1 输出反馈结果。

```

64.          2: // cmd8
65.          if( DoneU1 ) begin isCall[1] <= 1'b0; i <= i + 1'b1; end
66.          else begin isCall[1] <= 1'b1; end
67.
68.          3:
69.          begin T <= { 2'b11, TagU1[39:32], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
70.
71.          4:
72.          begin T <= { 2'b11, TagU1[31:24], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
73.

```

```

74.      5:
75.      begin T <= { 2'b11, TagU1[23:16], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
76.
77.      6:
78.      begin T <= { 2'b11, TagU1[15:8], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
79.
80.      7:
81.      begin T <= { 2'b11, TagU1[7:0], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
82.
83.      *****/
84.

```

步骤 2 执行 CMD8 , 步骤 3~7 输出反馈结果。

```

85.      8: // cmd58
86.      if( DoneU1 ) begin isCall[2] <= 1'b0; i <= i + 1'b1; end
87.      else begin isCall[2] <= 1'b1; end
88.
89.      9:
90.      begin T <= { 2'b11, TagU1[39:32], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
91.
92.      10:
93.      begin T <= { 2'b11, TagU1[31:24], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
94.
95.      11:
96.      begin T <= { 2'b11, TagU1[23:16], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
97.
98.      12:
99.      begin T <= { 2'b11, TagU1[15:8], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
100.
101.     13:
102.     begin T <= { 2'b11, TagU1[7:0], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
103.
104.     *****/
105.

```

步骤 8 执行 CMD58 , 步骤 9~13 输出反馈结果。

```

106.     14: // cmd55 + acmd41
107.     if( DoneU1 ) begin isCall[3] <= 1'b0; i <= i + 1'b1; end
108.     else begin isCall[3] <= 1'b1; end
109.
110.    15:

```

```

111. begin T <= { 2'b11, TagU1[39:32], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
112.
113. 16:
114. begin T <= { 2'b11, TagU1[31:24], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
115.
116. 17:
117. begin T <= { 2'b11, TagU1[23:16], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
118.
119. 18:
120. begin T <= { 2'b11, TagU1[15:8], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
121.
122. 19:
123. begin T <= { 2'b11, TagU1[7:0], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
124.
125. *****/
126.

```

步骤 14 执行 CMD55+ACMD41 , 步骤 15~19 输出反馈结果 , 其中步骤 17~19 的内容纯属花瓶而已。

```

127. 20: // cmd58
128. if( DoneU1 ) begin isCall[4] <= 1'b0; i <= i + 1'b1; end
129. else begin isCall[4] <= 1'b1; end
130.
131. 21:
132. begin T <= { 2'b11, TagU1[39:32], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
133.
134. 22:
135. begin T <= { 2'b11, TagU1[31:24], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
136.
137. 23:
138. begin T <= { 2'b11, TagU1[23:16], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
139.
140. 24:
141. begin T <= { 2'b11, TagU1[15:8], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
142.
143. 25:
144. begin T <= { 2'b11, TagU1[7:0], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
145.
146. *****/
147.

```

步骤 20 执行 CMD58 , 步骤 21~15 输出反馈结果。

```

148.          26: // cmd16
149.          if( DoneU1 ) begin isCall[5] <= 1'b0; i <= i + 1'b1; end
150.          else begin isCall[5] <= 1'b1; end
151.
152.          27:
153.          begin T <= { 2'b11, TagU1[7:0], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
154.
155.          *****/
156.

```

步骤 26 执行 CMD16 , 步骤 27 输出反馈结果。

```

157.          28: // Write Data 00~FF
158.          begin isEn[1] <= 1'b1; i <= i + 1'b1; end
159.
160.          29:
161.          begin isEn[1] <= 1'b0; i <= i + 1'b1; end
162.
163.          30:
164.          if( C2 == 511 ) begin C2 <= 11'd0; i <= i + 1'b1; end
165.          else begin D2 <= D2 + 1'b1; C2 <= C2 + 1'b1; i <= 6'd28; end
166.
167.          *****/
168.
169.          31: // cmd24
170.          if( DoneU1 ) begin isCall[7] <= 1'b0; i <= i + 1'b1; end
171.          else begin isCall[7] <= 1'b1; D1 <= 32'd0; end
172.
173.          32:
174.          begin T <= { 2'b11, TagU1[7:0], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
175.
176.          *****/
177.

```

步骤 28~30 写入两遍 8'h00~8'hFF 至 FIFO 里边 , 然后步骤 31 执行 CMD24 将其写入 SD 卡里边 , 步骤 32 随之输出反馈结果。

```

178.          33: // cmd17
179.          if( DoneU1 ) begin isCall[6] <= 1'b0; i <= i + 1'b1; end
180.          else begin isCall[6] <= 1'b1; D1 <= 32'd0; end
181.

```

```

182.          34:
183.          begin T <= { 2'b11, TagU1[7:0], 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
184.
185.          *****/
186.
187.          35: // Read Data 00~FF
188.          begin isEn[0] <= 1'b1; i <= i + 1'b1; end
189.
190.          36:
191.          begin isEn[0] <= 1'b0; i <= i + 1'b1; end
192.
193.          37:
194.          begin T <= { 2'b11, DataU1, 1'b0 }; i <= TXFUNC; Go <= i + 1'b1; end
195.
196.          38:
197.          if( C2 == 511 ) begin C2 <= 11'd0; i <= i + 1'b1; end
198.          else begin C2 <= C2 + 1'b1; i <= 6'd35; end
199.

```

步骤 33 执行 CMD17 , 步骤 34 则输出反馈结果。步骤 35~38 分别从 FIFO 哪里读出 512 个字节 , 并且经由 TXD 输出。

```

200.          39:
201.          i <= i;
202.
203.          *****/
204.
205.          48,49,50,51,52,53,54,55,56,57,58:
206.          if( C1 == B115K2 -1 ) begin C1 <= 11'd0; i <= i + 1'b1; end
207.          else begin rTXD <= T[i - 48]; C1 <= C1 + 1'b1; end
208.
209.          59:
210.          i <= Go;
211.
212.          endcase
213.
214.      assign TXD = rTXD;
215.
216. endmodule

```

步骤 39 为发呆。步骤 48~59 则是发送一帧数据的伪函数。总结完毕 , 便插入健康的大容量 SD 卡 , 如笔者手上 Kingston 制 16GB 的 SD 卡 , 然后再下载程序。操作过程如下所示 :

```

A2          // CMD0 执行成功
01 00 00 01 AA // CMD8 执行成功，字节 4 为 0x01，字节 1 为 0x01，字节 0 为 0xaa。
01 00 FF 80 00 // CMD58 执行成功，字节 4 为 0x01，字节 3 为 0x00
01 00 (FF 80 00) // 0x01 表示 CMD55 执行成功，0x00 表示 ACMD41 执行成功。后边 3 个字节作废。
00 C0 FF 80 00 // CMD58 执行成功，字节 4 为 0x00，字节 3 为 0xC0
A8          // CMD16 执行成功
A6          // CMD24 执行成功
AA          // CMD17 执行成功
00~FF      // 地址 0~255 的读取数据
00~FF      // 地址 256~511 的读取数据

```

读者稍微注意一下第二次执行 CMD58 的反馈结果 ... 其中 8'hC0 表示 SD 卡已经结束忙碌，而且也认识 CCS 的标示位。

		0001 0203 0405 0607 0809 0A0B 0C0D 0EOF	0123456789ABCDEF
0x00:	0x00	0001 0203 0405 0607 0809 0A0B 0C0D 0EOF
0x00:	0x10	1011 1213 1415 1617 1819 1A1B 1C1D 1E1F
0x00:	0x20	2021 2223 2425 2627 2829 2A2B 2C2D 2E2F	!"#\$%&'()*+,--/
0x00:	0x30	3031 3233 3435 3637 3839 3A3B 3C3D 3E3F	0123456789:;=>?
0x00:	0x40	4041 4243 4445 4647 4849 4A4B 4C4D 4E4F	@ABCDEFGHIJKLMNO
0x00:	0x50	5051 5253 5455 5657 5859 5A5B 5C5D 5E5F	PQRSTUVWXYZ[\]^_
0x00:	0x60	6061 6263 6465 6667 6869 6A6B 6C6D 6E6F	'abcdefghijklmn
0x00:	0x70	7071 7273 7475 7677 7879 7A7B 7C7D 7E7F	pqrstuvwxyz{ }~
0x00:	0x80	8081 8283 8485 8687 8889 8A8B 8C8D 8E8F
0x00:	0x90	9091 9293 9495 9697 9899 9A9B 9C9D 9E9F
0x00:	0xA0	A0A1 A2A3 A4A5 A6A7 A8A9 AAAB ACAD AEAF
0x00:	0xB0	B0B1 B2B3 B4B5 B6B7 B8B9 BABB BCBD BEBF
0x00:	0xC0	C0C1 C2C3 C4C5 C6C7 C8C9 CACB CCCD CECF
0x00:	0xD0	D0D1 D2D3 D4D5 D6D7 D8D9 DADB DCDD DEDF
0x00:	0xE0	E0E1 E2E3 E4E5 E6E7 E8E9 EAEB ECED EEEF
0x00:	0xF0	F0F1 F2F3 F4F5 F6F7 F8F9 FAFB FCFD FEFF
0x00:	0x0100	0001 0203 0405 0607 0809 0A0B 0C0D 0EOF
0x00:	0x0110	1011 1213 1415 1617 1819 1A1B 1C1D 1E1F
0x00:	0x0120	2021 2223 2425 2627 2829 2A2B 2C2D 2E2F	!"#\$%&'()*+,--/
0x00:	0x0130	3031 3233 3435 3637 3839 3A3B 3C3D 3E3F	0123456789:;=>?
0x00:	0x0140	4041 4243 4445 4647 4849 4A4B 4C4D 4E4F	@ABCDEFGHIJKLMNO
0x00:	0x0150	5051 5253 5455 5657 5859 5A5B 5C5D 5E5F	PQRSTUVWXYZ[\]^_
0x00:	0x0160	6061 6263 6465 6667 6869 6A6B 6C6D 6E6F	'abcdefghijklmn
0x00:	0x0170	7071 7273 7475 7677 7879 7A7B 7C7D 7E7F	pqrstuvwxyz{ }~
0x00:	0x0180	8081 8283 8485 8687 8889 8A8B 8C8D 8E8F
0x00:	0x0190	9091 9293 9495 9697 9899 9A9B 9C9D 9E9F
0x00:	0x01A0	A0A1 A2A3 A4A5 A6A7 A8A9 AAAB ACAD AEAF
0x00:	0x01B0	B0B1 B2B3 B4B5 B6B7 B8B9 BABB BCBD BEBF
0x00:	0x01C0	C0C1 C2C3 C4C5 C6C7 C8C9 CACB CCCD CECF
0x00:	0x01D0	D0D1 D2D3 D4D5 D6D7 D8D9 DADB DCDD DEDF
0x00:	0x01E0	E0E1 E2E3 E4E5 E6E7 E8E9 EAEB ECED EEEF
0x00:	0x01F0	F0F1 F2F3 F4F5 F6F7 F8F9 FAFB FCFD FEFF

图 25.11 SDHC 卡，地址 0~511 的内容。

图 25.11 是 SDHC 卡的五脏六腑，地址 0x00~0xff 的数据为 0x00~0xff，地址

0x0100~0x01f0 的数据也是 0x00~0xff。对此，表示实验已经成功。

细节一：完整的个体模块

实验二十五的 SD 卡基础模块虽然已经准备就绪，不过它不聪明也不支持版本 SDV1.x 的 SD 卡。此外，SD 卡也必须健康无患，不然该基础模块会运行失败。

实验二十六：VGA 模块

VGA 这家伙也算孽缘之一，从《建模篇》那时候开始便一路缠着笔者。《建模篇》之际，学习主要针对像素，帧，颜色等 VGA 的简单概念。《时序篇》之际，笔者便开始摸索 VGA 的时序。《整合篇》之际，笔者尝试控制 VGA 的时序。如今《驱动篇 I》的内容返回 VGA 的本题，也就是图像方面的故事。

此刻，澎湃之情不容怠慢，请怒笔者不再回忆往事，失忆者请复习《Verilog HDL 那些事儿》，笔者虽然也想直奔主题 … 可是在此之前，笔者必须补足那些不易注意的细节。俗语有云，细节是关键的藏身之地，那些不起眼的小细节，往往都是左右大局的关键，学习也是如此。不过，实验二十六究竟存在多少影响成败的小细节呢？请竖起耳朵，让笔者慢慢告诉读者 …

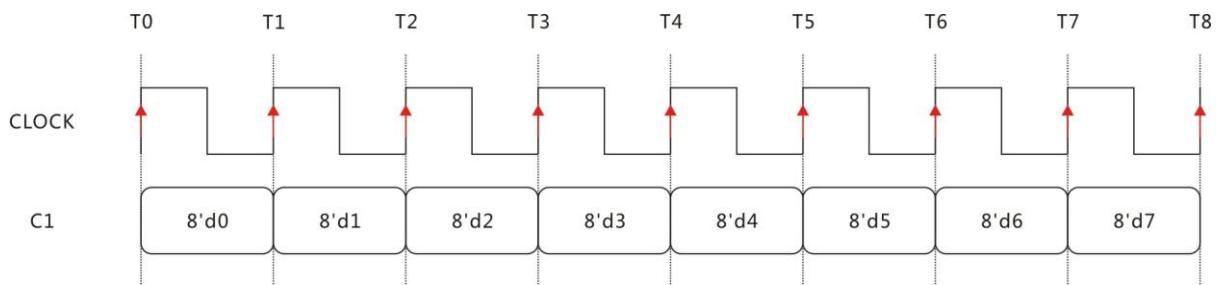


图 26.1 C1 计数的理想时序。

图 26.1 是一段计数时序，C1 扮演计数器，而且时序理想。对此，这段时序一定按照“时间点”在表现。假设笔者想要建立判断，那么判断基准会基于 C1 的过去值，例如：

```
if( C1 == 0 );
```

那么 if(C1 == 0) 必须在 T1 才能成立。再假设笔者想要拉长判断的长度，例如 T0~T8 之间，那么笔者可以这样写：

```
if( C1 == 0 || C1 == 1 || C1 == 2 || C1 == 3 || C1 == 4 || C1 == 5 || C1 == 6 || C1 == 7 );
```

如果长度像妈妈一样长气，例如 T2~T99。当然，上述方法一定行不通，因为后果不仅累坏自己，而且模块的内容也会沉长臃肿。为此，笔者可以或者这样写：

```
if ( C1 >= 0 && C1 <= 7 );
```

其中，if(C1 >= 0 && C1 <= 7) 表示 9 个时钟沿的长度。

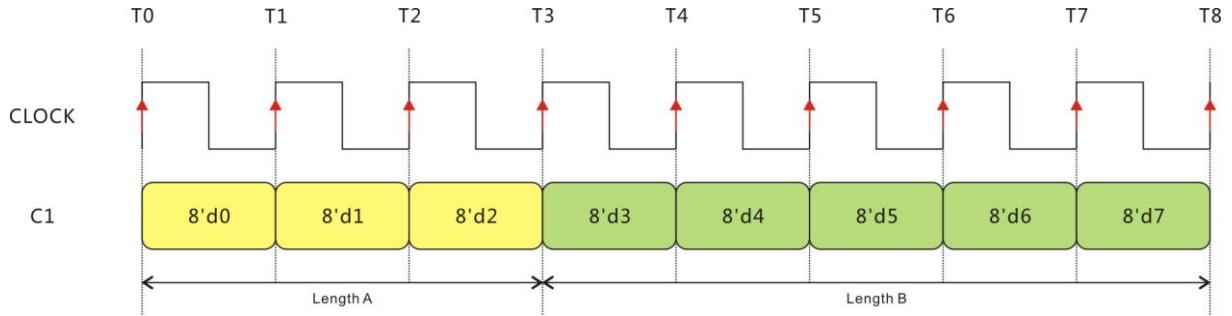


图 26.2 长度有 AB 之别的计数时序。

假设某某长度是固定的家伙，例如图 26.2，T0~T3 组成 4 个时钟沿的长度 A，而 T3~T8 组成 5 个时钟沿的长度 B。为了方便，笔者可以建立常量：

```
parameter A = 3, B = 5;
```

如果笔者想把长度 A 作为有效的判断基准，那么笔者可以这样写：

```
if( C1 >= 0 && C1 <= A -1 );
```

代码的用意非常明显， $C1 \geq 0$ 表示长度的起始，然而 $C1 \leq A - 1$ 则是长度的结束。如果长度 B 也作为判断的基准，同样的写法也适用：

```
if( C1 >= 2 && C1 <= A + B -1 );
```

可是，不管怎么看 ... 笔者觉得 $C1 \geq 2$ 非常别扭，对此笔者可以这样修改：

```
if( C1 >= A -1 && C1 < A + B -1 );
```

怎么样，读者是不是稍微顺眼一点？

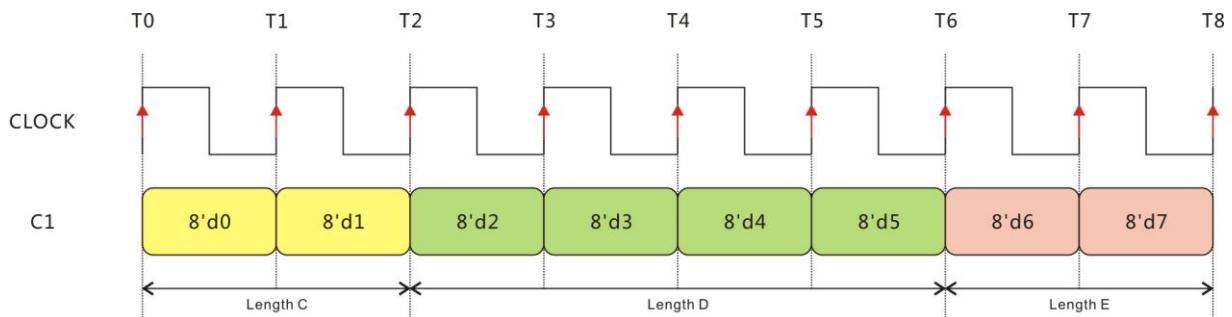


图 26.3 长度有 CDE 之别的计数时序。

假设顽皮的长度作出手势，然后分身成为 3 个，结果如图 26.3 所示。T0~T1 组成 3 个时钟沿的长度 C，T2~T6 组成 5 个时钟沿的长度 D，T6~T8 组成 3 个时钟沿的长度 E。为了方便，笔者先做常量声明：

```
parameter C = 2, D = 4, E = 2.
```

如果笔者想把长度 C 作为判断的基准，那么笔者可以这样写：

```
if( C1 >= 0 && C1 <= C -1 );
```

如果笔者想把长度 D 作为判断的基准，那么笔者可以这样写：

```
if( C1 >= C -1 && C1 <= C + D -1 );
```

如果笔者想把长度 E 作为判断的基准，那么笔者可以这样写：

```
if( C1 >= C + D -1 && C1 <= C + D + E -1 );
```

理解完毕以后，笔者可以这样总结 ... 由于 C1 从 0 开始计数，除非长度的起始地方是从 0 开始，否则长度的起始地方必须添加 -1 的处理。反之，不管结束地方怎么歇斯底里抵抗，长度的结束地方都必须加上 -1 处理。

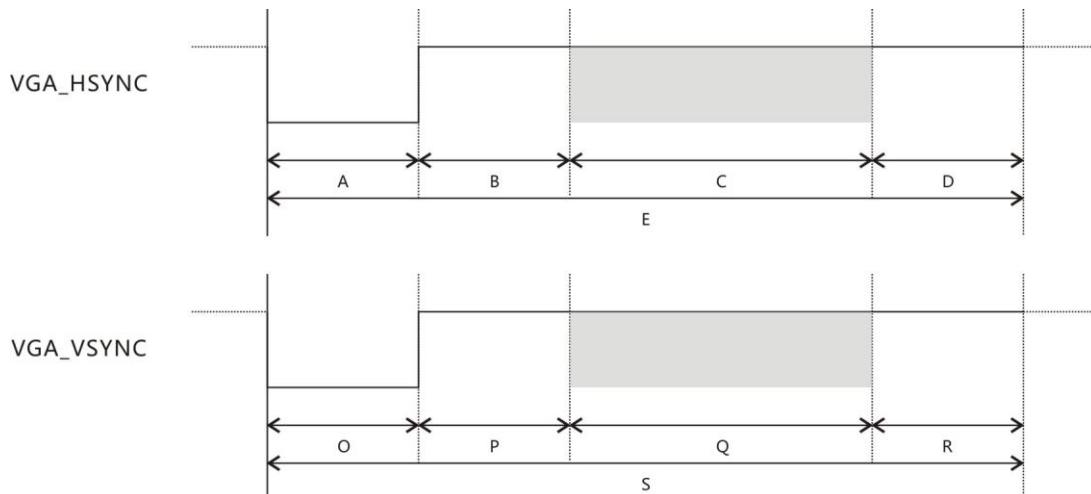


图 26.4 VGA 时序。

图 26.4 是典型的 VGA 时序，VGA 有 5 条信号，其中 HSYNC 与 VSYNC 控制显示分辨率还有显示帧。本实验的显示标准选为 $1024 \times 768 @ 60Hz - 65Mhz$ ，而表 26.1 就是该显示标准的内容：

表 26.1 显示标准 $1024 \times 768 @ 60Hz - 65Mhz$ 。

信号	A	B	C	D	E
VGA_HSYNC	136	160	1024	24	1344
信号	0	P	Q	R	S
VGA_VSYNC	6	29	768	3	806

如表 26.1 所示，A 段与 O 段都是拉低的起始段，然后 B 段与 P 段是准备段，C 段与 Q

段是数据段，D 段与 R 段都是结束段，至于 E 段是 ABCD 段的总和，S 段则是 OPRQ 段的总和。HSYNC 有时候也称为列像素或者 X，VSYNC 则称为行像素或者 Y。列像素最小的周期时间（或称像素时钟 Pixel Clock）是 $1/65\text{Mhz}$ ，行像素最小的周期时间则是 E 段 $\times 1/65\text{Mhz}$ 。话题继续之前，笔者再稍微补充一下细节内容。

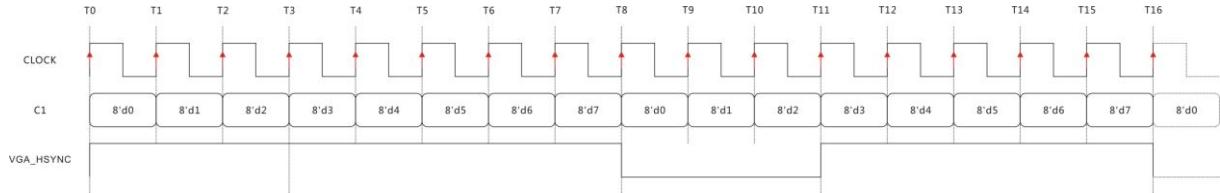


图 26.5 计数例子。

我们先假设 HSYNC 长度有 8，前 3 个为拉低的起始段，后 8 个为拉高的数据段。如图 26.5 所示，C1 总共计数两次，分别是 T0~T8 还有 T8~T16。期间，第一次是无效的计数，所以 HSYNC 并没有拉低起始段。换之，第二次开始才是有效的计数，因为 HSYNC 拉低起始段。为此，Verilog 可以这样描述：

```
if( C1 == 7 ) HSYNC <= 1' b0;
else if( C1 == 2 ) HSYNC <= 1' b1;
```

假设起始段声明为 A，而结束段声明为 D，E 则是 A 与 D 的总和，那么内容可以继续修改：

```
parameter A = 3, D = 5, E = 8;

if( C1 == E -1 ) HSYNC <= 1' b0;
else if( C1 == A -1 ) HSYNC <= 1' b1;
```

明白以后，我们便可以用 Verilog 描述表 26.1 的内容，结果如代码 26.1 所示：

```
1.      parameter SA = 11'd136, SE = 11'd1344;
2.      parameter SO = 11'd6, SS = 11'd806;
3.
4.      reg [10:0]C1;
5.      reg [9:0]C2;
6.      reg rH,rV;
7.
8.      always @ ( posedge CLOCK or negedge RESET )
9.          if( !RESET )
10.              begin
11.                  C1 <= 11'd0;
12.                  C2 <= 10'd0;
13.                  rH <= 1'b1;
```

```

14.           rV <= 1'b1;
15.       end
16.   else
17.     begin
18.       if( C1 == SE -1 ) rH <= 1'b0;
19.       else if( C1 == SA -1 ) rH <= 1'b1;
20.
21.       if( C2 == SS -1 ) rV <= 1'b0;
22.       else if( C2 == SO -1 ) rV <= 1'b1;
23.
24.       if( C2 == SS -1 ) C2 <= 10'd0;
25.       else if( C1 == SE -1 ) C2 <= C2 + 1'b1;
26.
27.       if( C1 == SE -1 ) C1 <= 11'd0;
28.       else C1 <= C1 + 1'b1;
29.     end

```

代码 26.1

第 1~2 行是 A 段与 E 段，O 段还有 S 段的常量声明。第 4~6 是相关的寄存器声明，第 10~14 行则是这些寄存器的复位操作。请注意一下第 8 行的主时钟是 65Mhz。第 27~28 行是针对列像素的计数器 C1，计数范围为 0~1343。第 24~25 行是针对行像素的计数器 C2，计数范围为 0~805。第 18~19 行用来控制 HSYNC，第 21~22 行则是用来控制 VSYNC。

理解完毕 HSYNC 与 VSYNC 信号以后，接下来我们便要学习 RGB 信号。事实上，HSYNC 与 VSYNC 的数据段，其实也是 RGB 有效的数据段。表 26.1 基于 $1024 \times 768 @ 60Hz$ ，所以 HSYNC 数据段的长度有 1024，而 VSYNC 的数据段的长度则有 768。如果 VGA 拥有显示标准，那么 RGB 信号也有所谓的颜色标准。根据 Photoshop，常见的颜色如表 26.2 所示：

表 26.2 常见的颜色。

颜色	位宽
黑白	1bit
灰度级	4bit , 8bit
RGB	8bit , 16bit , 24bit , 32bit

黑白可谓是最典型的颜色，不禁让笔者想起怀念的小绿人。灰度级正如字面上的意思，意指失去颜色的图像，4bit 有 16 灰度级，8bit 则有 256 灰度级。RGB 是电脑专用的颜色标准，从过去发展至今，RGB 的颜色分辨率也从 4bit 发展到 32bit。虽说 8 位 RGB 是历史悠久的前辈，不过至今它还未退休吧，例如街边的 LED 招牌。8 位 RGB 也称为索引色。

16 位 RGB 可是人眼比较接近的颜色标准，也是本实验的主题，其中 $RGB[15:11]$ 是 5 位红色， $RGB[10:5]$ 是 6 位绿色， $RGB[4:0]$ 则是 5 位蓝色，16 位 RGB 称为高彩。24 位

RGB 也是目前流行的颜色标准，其中字节 0 为蓝色，字节 1 为绿色，字节 2 为红色，24 位称为真彩。32 位 RGB 相较 24 位 RGB 则多了一个字节 3 的通道字节，通道也指透明效果，它称为全彩。

为消除读者对 RGB 的恐惧，笔者就解剖一下 16 位 RGB：



图 26.6 4×1 与 16 位 RGB 图像。

图 26.6 是一幅宽为 4 高为 1 的 16 位 RGB 图像，如果从左至右开始数起，列 0 为饱和的红色，列 1 为饱和的绿色，列 2 为饱和的蓝色，列 3 为纯黑。如果将其卸甲并且一字排开高位在前的内容，结果如表 26.2 所示：

表示 26.3 4×1 与 16 位 RGB 图像内容(高位在前)。

列 0	列 1	列 2	列 3
饱和红	饱和绿	饱和蓝	黑
16' hF800	16' h07E0	16' h001F	16' h0000
16' b1111_1000_0000_0000	16' b0000_0111_1110_0000	16' b0000_0000_0001_1111	16' b0000_0000_0000_0000

如表 26.3 所示，我们可以看见列 0 的红色占据 RGB[15:11]的位置，亦即红色有 $2^5=32$ 的分辨率。列 1 的绿色则是占据 RGB[10:5]的位置，亦即 $2^6=64$ 的分辨率。至于列 2 的蓝色占据 RGB[4:0]的位置，结果它有 $2^5=32$ 的分辨率。



图 26.7 4×1 与 16 位 RGB 图像 (红色渐变)。

图 26.7 也是一幅宽有 4 高有 1 的 16 位 RGB 图像，不过是红色渐变的图像。如果从左至右开始数起，列 0 为 4/4 饱和的红色，列 1 为 3/4 饱和的红色，列 2 为 2/4 饱和的红色，列 3 为 1/4 饱和的红色。笔者随之伸出魔爪将其卸甲，然后一字排开高位在前的内容，结果如表 26.4 所示：

表示 26.4 4×1 与 16 位 RGB 图像内容 (红色渐变与高位在前)。

列 0	列 1	列 2	列 3
4/4 饱和红	3/4 饱和红	2/4 饱和红	1/4 饱和红
16' hF800	16' hC000	16' h8000	16' h4000
16' b1111_1000_0000_0000	16' b1100_0000_0000_0000	16' b1000_0000_0000_0000	16' b0100_0000_0000_0000

如表 26.4 所示，我们可以看见列 0 的红色充斥整座 RGB[15:11]，列 1 的红色则是占据其中的 14 份内容而已。列 2 更惨，它占据的份儿只有 8 份而已，而最惨的列 3 则是只有麻雀眼泪般的 4 份内容而已。



图 26.8 实验用的小可爱。

图 26.6 是实验二十六所用的图像资源，内容是一群可爱的比卡丘在玩耍。图像大小为 $128 \times 96 \times 16$ bit，结果容量为：

$$128 \times 96 \times 16 \text{ bit} = 196608 \text{ bit}$$

如果储存器的位宽有 16 位，那么储存器的地址位宽则是：

$$128 \times 96 = 12208$$

因此，必须建立位宽为 2^{14} 的地址信号：

$$2^{14} = 16384$$

简单而言，图 26.8 的 X 为 128 还有 Y 为 96，而 Y 也充当偏移量的角色。为此，正确的地址公式如下所示：

$$\begin{aligned} \text{Address} &= X + (Y \times 128) \\ &= X + (Y \ll 7) \end{aligned}$$

理解完毕这些准备知识以后，我们便可以开始建模了。

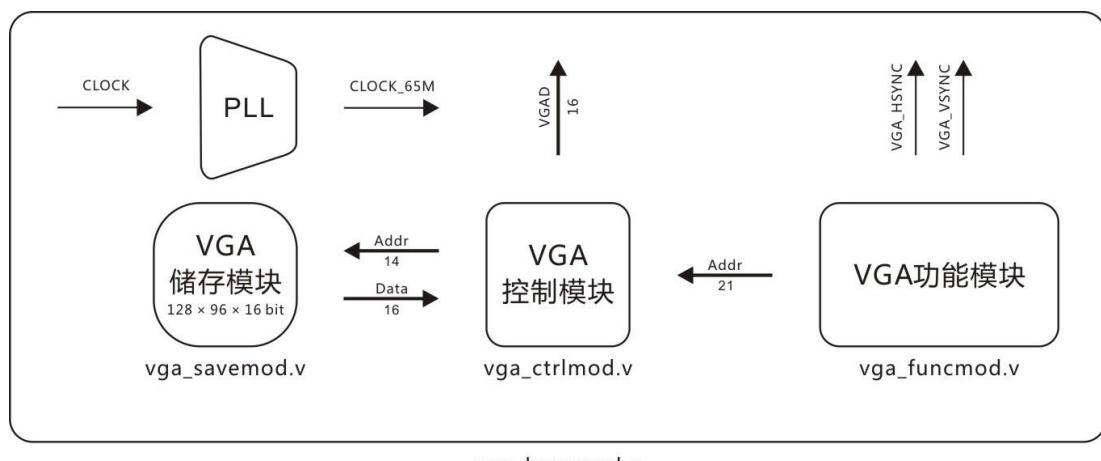


图 26.9 实验二十六 (VGA 基础模块) 的建模图。

图 26.9 是实验二十六的建模图，其中 PLL 将 50Mhz 的 CLOCK 增至 65Mhz。VGA 储存模块有 $128 \times 96 \times 16$ bit 的容量，里边暂存皮卡丘的图像信息。VGA 功能模块则是负责 $1024 \times 768 @ 60Hz$ 的显示标准，还有将内部的计数信息反馈给 VGA 控制模块。VGA 控制模块位与中间，它借助 iAddr 然后转换成为图像读取的地址，最后再将反馈回来的图像信息发至 VGAD 顶层信号。

vga_savemod.v

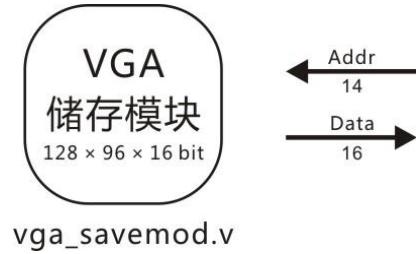


图 26.10 VGA 储存模块的建模图。

图 26.10 虽是 VGA 储存模块的建模图，不过内容却是简单 ROM 模块，具体内容让我们来看代码吧。

```
1. module vga_savemod
2. (
3.     input CLOCK, RESET,
4.     input [13:0]iAddr,
5.     output [15:0]oData
6. );
7. (* ramstyle = "no_rw_check , m9k" , ram_init_file = "pikachu_128x96_16_msb.mif" *)
8. reg [15:0] RAM[12287:0];
9. reg [15:0]D1;
10.
11. always @ ( posedge CLOCK or negedge RESET )
12.     if( !RESET )
13.         D1 <= 16'd0;
14.     else
15.         D1 <= RAM[ iAddr ];
16.
17. assign oData = D1;
18.
19. endmodule
```

以上内容虽然简单，不过还是注意一下第 7 行的建立声明。其中 no_rw_check 是用来告诉综合器无视 read during write 的检测，m9k 则声明片上内存用 m9k，至于 ram init file 则是 RAM 的初始化内容，亦即图 26.8。

vga_funcmod.v



图 26.11 VGA 功能模块的建模图。

图 26.11 是 VGA 功能模块的建模图，左边是反馈给朋友的 oAddr，其中[20:10]是 X 地址，[9:0]则是 Y 地址。右边是驱动顶层的 VGA_HSYNC 与 VGA_VSYNC。

```
1. module vga_funcmod
2. (
3.     input CLOCK, RESET,
4.     output VGA_HSYNC, VGA_VSYNC,
5.     output [20:0]oAddr
6. );
7. parameter SA = 11'd136, SE = 11'd1344;
8. parameter SO = 11'd6, SS = 11'd806;
9.
```

以上内容为相关的出入端声明，还有 A 段，E 段，O 段还有 S 段的常量声明。

```
10.    reg [10:0]C1;
11.    reg [9:0]C2;
12.    reg rH,rV;
13.
14.    always @ ( posedge CLOCK or negedge RESET )
15.        if( !RESET )
16.            begin
17.                C1 <= 11'd0;
18.                C2 <= 10'd0;
19.                rH <= 1'b1;
20.                rV <= 1'b1;
21.            end
22.        else
```

以上内容为相关的寄存器声明，其中 C1 为 HSYNC 计数，C2 则为 VSYNC 计数。

```
23.        begin
```

```

24.
25.          if( C1 == SE -1 ) rH <= 1'b0;
26.          else if( C1 == SA -1 ) rH <= 1'b1;
27.
28.          if( C2 == SS -1 ) rV <= 1'b0;
29.          else if( C2 == SO -1 ) rV <= 1'b1;
30.
31.          if( C2 == SS -1 ) C2 <= 10'd0;
32.          else if( C1 == SE -1 ) C2 <= C2 + 1'b1;
33.
34.          if( C1 == SE -1 ) C1 <= 11'd0;
35.          else C1 <= C1 + 1'b1;
36.
37.      end
38.

```

以上内容为 HSYNC 与 VSYNC 的驱动过程。第 35~36 行是计数 HSYNC，第 32~33 行则是计数 VSYNC，一个 VSYNC 为 1344 个 HSYNC。第 26~27 行是 rH 的控制，第 29~30 行则是 rV 的控制。

```

39.      reg [1:0]B1,B2,B3;
40.
41.      always @ ( posedge CLOCK or negedge RESET )
42.          if( !RESET )
43.              { B3, B2, B1 } <= 6'b11_11_11;
44.          else
45.              begin
46.                  B1 <= { rH,rV };
47.                  B2 <= B1;
48.                  B3 <= B2;
49.              end
50.

```

第 40~51 则是用来延迟 rH 与 rV 的周边操作，期间总共延迟 3 个时钟。详细内容往后再说。

```

51.      assign { VGA_HSYNC, VGA_VSYNC } = B3;
52.      assign oAddr = {C1,C2}
53.
54.  endmodule

```

以上内容为相关的输出驱动。

vga_ctrlmod.v

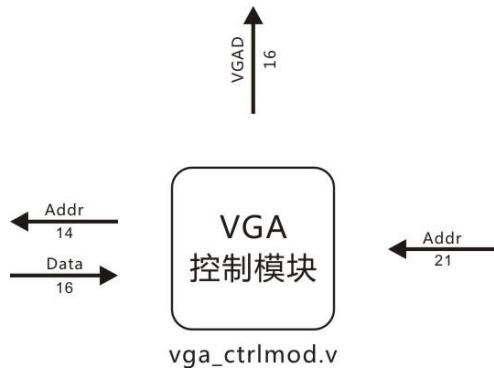


图 26.12 VGA 控制模块的建模图。

图 26.12 是 VGA 控制模块的建模图，而设计思路来源《整合篇》。左边信号连接储存模块，右边信号连接功能模块，北边信号则驱动顶层。具体内容还是让我们来看代码吧：

```
1. module vga_ctrlmod
2. (
3.     input CLOCK, RESET,
4.     output [15:0]VGAD,
5.     output [13:0]oAddr,
6.     input [15:0]iData,
7.     input [20:0]iAddr
8. );
```

以上内容为相关的出入端声明。

```
9.     parameter SA = 11'd136, SB = 11'd160, SC = 11'd1024, SD = 11'd24, SE = 11'd1344;
10.    parameter SO = 11'd6, SP = 11'd29, SQ = 11'd768, SR = 11'd3, SS = 11'd806;
11.
12.    // Height , width, x-offset and y-offset
13.    parameter XSIZE = 8'd128, YSIZE = 8'd96, XOFF = 10'd0, YOFF = 10'd0;
14.
15.    wire isX = ( iAddr[20:10] >= SA + SB + XOFF - 1 ) && ( iAddr[20:10] <= SA + SB + XOFF + XSIZE - 1 );
16.    wire isY = ( iAddr[9:0] >= SO + SP + YOFF - 1 ) && ( iAddr[9:0] <= SO + SP + YOFF + YSIZE - 1 );
17.    wire isReady = isX & isY;
18.
19.    wire [31:0] x = iAddr[20:10] - XOFF - SA - SB - 1;
20.    wire [31:0] y = iAddr[9:0] - YOFF - SO - SP - 1;
21.
```

以上内容为相关的常量声明。第 9~10 行是针对 $1024 \times 768 @ 60Hz$ 显示标准的常量声明。第 13 行声明图像信息，如大小还有位置。第 15 行声明有效的列像素，第 16 行则声明有效的 Y 像素，至于第 17 行则是声明图像的有效区域，注意它们都是即时事件。

第 19~20 行用来是计算有效的 X 与 Y , 主要用来取得图像的读取地址 , 注意它们也是即时事件。

```
22.      reg [31:0]D1;
23.      reg [15:0]rVGAD;
24.
25.      always @ ( posedge CLOCK or negedge RESET )
26.          if( !RESET )
27.              begin
28.                  D1 <= 18'd0;
29.                  rVGAD <= 16'd0;
30.              end
31.          else
```

以上内容为相关的寄存器声明还有复位操作。

```
32.          begin
33.
34.              // step 1 : compute data address and index-n
35.              if( isReady )
36.                  D1 <= (y << 7) + x;
37.              else
38.                  D1 <= 14'd0;
39.
40.              // step 2 : reading data from rom
41.              // but do-nothing
42.
43.              // step 3 : assign RGB_Sig
44.              rVGAD <= isReady ? iData : 16'd0;
45.
46.          end
47.
```

以上内容为 VGA 控制模块的核心操作。该控制模块采用流水操作 , 一边不断发送读取地址 , 一边不断等待图像信息读出 , 一边不断输出图像信息。

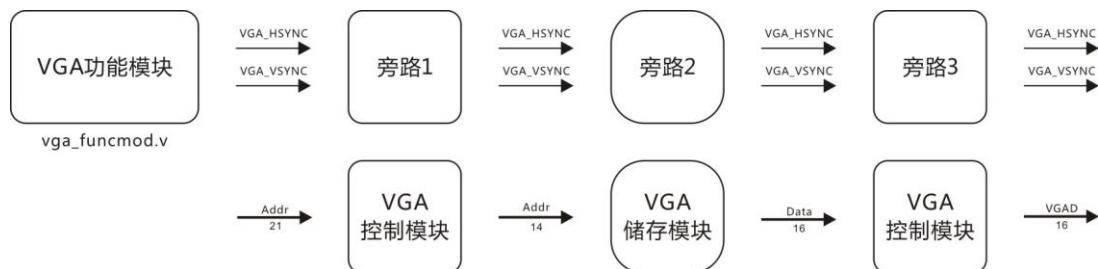


图 26.13 VGA 功能模块的流水操作。

从某种程度来说，实验二十六的 VGA 基础模块可以看成如图 26.13 所示。VGA 功能模块输出 HSYNC 与 VSYNC 的瞬间，它也发送 Addr 给 VGA 控制模块。VGA 控制模块计算读取地址以后再发送给 VGA 储存模块，VGA 储存模块随意也将图像信息返回发送给 VGA 控制模块，最后 VGA 控制模块再将图像信息驱动至 VGAD。

简单来说，图像信息一共慢 HSYNC 和 VSYNC 信号 3 拍，因此 VGA 功能模块多了旁路的周边操作，目的是为了同步 HSYNC, VSYNC 还有 VGAD 之间的延迟。返回话题，有效的读取地址只有 isReady 拉高的时候，如代码行第 36 所示。同样，有效的图像信息也是 isReady 拉高的时候，如代码行第 45 所示。

```
48.     assign VGAD = rVGAD;
49.     assign oAddr = D1[13:0];
50.
51. endmodule
```

以上内容为相关的输出驱动。

vga_basemod.v

该模块是 VGA 基础模块，至于内部的连线部署请参考图 26.9。

```
1. module vga_basemod
2. (
3.     input CLOCK, RESET,
4.     output VGA_HSYNC, VGA_VSYNC,
5.     output [15:0]VGAD
6. );
7.     wire CLOCK_65M;
8.
9.     pll_module U1
10.    (
11.        .inclk0( CLOCK ),
12.        .c0( CLOCK_65M ) // CLOCK 65Mhz
13.    );
14.
15.    wire [20:0]AddrU2;
16.
17.    vga_funcmod U2      // 1024 * 758 @ 60Hz
18.    (
19.        .CLOCK(CLOCK_65M ),
```

```
20.      .RESET( RESET ),
21.      .VGA_HSYNC( VGA_HSYNC ),
22.      .VGA_VSYNC( VGA_VSYNC ),
23.      .oAddr( AddrU2 ),
24. );
25.
26.      wire [15:0]DataU3;
27.
28.      vga_savemod U3
29.      (
30.          .CLOCK( CLOCK_65M ),
31.          .RESET( RESET ),
32.          .iAddr( AddrU4 ),
33.          .oData ( DataU3 )
34.      );
35.
36.      wire [13:0]AddrU4;
37.
38.      vga_ctrlmod U4 // 128 * 96 * 16bit, X0,Y0
39.      (
40.          .CLOCK( CLOCK_65M ),
41.          .RESET( RESET ),
42.          .VGAD( VGAD ),
43.          .iData( DataU3 ),
44.          .oAddr( AddrU4 ),
45.          .iAddr( AddrU2 ),
46.      );
47.
48. endmodule
```

详细的内容请读者自己看着办吧。



图 26.14 实验二十六的结果。

综合完毕便下载程序，如果实验成功，分辨率为 $1024 * 768$ 的显示器左上角便会出现一幅 $128 \times 96 \times 16$ bit 的图像显示在 X0 与 Y0 的位置，结果如图 26.14 所示。

细节一：完整的个体模块

实验二十六的 VGA 基础模块虽为就绪状态，不过它是一只能力有限的家伙。原因很简单，因为储存 $128 \times 96 \times 16$ bit 的图像已经是 EP4CE6F17C8 这块 FPGA 的极限，这家伙的度量很小，所以片上内存也不怎么大。

细节二：片上内存

笔者曾经说过，片上内存是资源很棒的储存资源，它不仅访问时间短，自定义强，而且操作也简单，但是容量是它永远的痛。EP4CE6F17C8 虽然有 476280 bit 的片上内存，不过实际可用的范围只有其中的 90%而已。

细节三：缓冲图像的储存模块

如果片上内存不行，那么 SDRAM 当然是最好的替代。同样，笔者也曾经说过，SDRAM 的优点除了容量大以外，无论是访问速度还有操作程度也不及片上内存。一旦 SDRAM 谋朝散位成功，VGA 控制模块，VGA 储存模块还有 VGA 功能模块之间就会失去同步性。此外，SDRAM 也不能经过综合器初始化内容。不管怎么样，实验二十六已经完成任务，以后的问题以后再说吧。

实验二十七：TFT 模块 - 显示

所谓 TFT (Thin Film Transistor) 就是众多 LCD 当中，其中一种支持颜色的 LCD，相较古老的点阵 LCD (12864 点)，它可谓高级了。黑金的 TFT LCD 除了 320×240 大小以外，内置 SSD1289 控制器，同时也是独立模块。事实上，无论是驱动点阵 LCD 还是 TFT LCD，结果都是配置里边的控制器，差别就在于控制器的复杂程度而已。不管怎么样，如果只是单纯地显示内容，SSD1289 控制器也不会难到那里去。

未进入主题之前，请容许笔者补足一下循环操作的内容。首先笔者必须强调，循环操作与循环操作模式本身是不同性质的东西，操作模式是为了优化操作才存在这个世界上，例如空间换时钟，或者时钟换空间等优化倾向。反之，循环操作类似关键字 for，while，do ... while 等代码意义上的重复运行。

笔者曾在实验二十一说过，顺序语言 for，while，do ... while 等的键字，它们的作用主要是简化代码的重复性。不过很遗憾的是，这些关键字并不怎么喜欢描述语言，所以循环操作一直是描述语言的痛。对此，笔者才怂恿描述语言，先模仿顺序操作，再模仿循环操作。这种操纵它人的背德感，笔者无比满足与喜悦 … 嘻哈嘻哈（兴奋声）。

相较先判断后执行，笔者比较倾向先执行后判断，结果 do ... while 是描述语言最佳的模仿对象。举例而言，如代码 27.1 所示：

```
// C 语言 do ... while 循环
do { FuncA(); i++; } while( i < 4 )

i 为 0，执行函数 A，i 递增为 1，i 小于 4，如是继续；
i 为 1，执行函数 A，i 递增为 2，i 小于 4，如是继续；
i 为 2，执行函数 A，i 递增为 3，i 小于 4，如是继续；
i 为 3，执行函数 A，i 递增为 4，i 小于 4，不是则结束操作。
```

代码 27.1

如代码 27.1 所示，C 语言使用 do ... while 重复执行函数 A，其中 i 控制循环， $i < 4$ 为执行条件。i 为 0~3 期间，总共执行 4 次函数 A。至于 Verilog 可以这样模仿代码 27.1，结果如代码 27.2 所示：

```
// Verilog 语言
case( i )

  0:
    if (DoneA) begin isStart <= 1' b0; i <= i + 1' b1; end
    else begin isStart <= 1' b1; end

  1:
```

```
if( C1 == 4 -1) begin C1 <= 8' d0; i <= i + 1' b1; end  
else begin C1 <= C1 + 1' b1; i <= 4' d0; end
```

步骤0, i 为 0, 执行功能 A; 步骤1, i 为 0, 条件不成立, i 递增为 1, 返回步骤;
步骤0, i 为 1, 执行功能 A; 步骤1, i 为 1, 条件不成立, i 递增为 2, 返回步骤;
步骤0, i 为 2, 执行功能 A; 步骤1, i 为 2, 条件不成立, i 递增为 3, 返回步骤;
步骤0, i 为 3, 执行功能 A; 步骤1, i 为 3, 条件成立, i 清零, 继续步骤;

代码 27.2

如代码 27.2 所示, 那是 Verilog 模仿 do .. while 循环的结果。代码 27.1 与代码 27.2 之间最大的差别就是后者 (描述语言) 必须自行建立顺序结构, 执行建立循环操作。虽说, 描述语言什么都要自己动手, 非常劳动。不过, 只要模仿起来似模似样, 描述语言也有循环利器。为了磨尖这把利器, 笔者需要改动一下代码 27.2, 结果如代码 27.3 所示:

```
// Verilog 语言  
case( i )  
  
    0:  
        if ( DoneA ) begin isStart <= 1' b0; Go <= i; i <= i + 1' b1; end  
        else begin isStart <= 1' b1; end  
  
    1:  
        if( C1 == 4 -1) begin C1 <= 8' d0; i <= i + 1' b1; end  
        else begin C1 <= C1 + 1' b1; i <= Go; end
```

代码 27.3

如代码 27.3 所示, 笔者为循环加入返回作用的 Go。步骤 0 之际, 功能 A 执行完毕, Go 便会暂存当前步骤, 然后 i 继续步骤。步骤 1 之际, 如果 if 条件不成立, C1 递增, i 返回 Go 指向的地方。反之, 如果 if 条件成立, C1 会清零, 然后 i 继续步骤。好奇的朋友一定会觉得疑惑, 这样作究竟有什么好处?

嘻嘻! 好处可多了 ... 笔者这样做除了让代码变漂亮一些以外, 我们还能实现梦寐以求的嵌套循环, 并且不失代码的表达能力, 举例而言, 如代码 27.4:

```
// C 语言, 2 层嵌套 for  
for( C2 = 0 ; C2 < 8 ; C2++ )  
    for( C1 = 0 ; C1 < 4 ; C1 ++ )  
        FuncA();
```

代码 27.4

如代码 27.4 所示, 哪里有一组嵌套 for, C1 控制函数 A 执行 4 次, C2 则控制 C1 执行 8 次, 结果函数 A 一共执行 32 次。一般而言, 描述语言是很难实现这种嵌套 for, 即使侥幸成功, 我们也得付出巨大的代价。如今仿顺序操作在后面撑腰, 惨痛已经成为过去式

的悲剧 ... 废话不多说，让我们瞧瞧低级建模 II 的力量吧！少年少女们！

```
// Verilog 语言
case( i )

    0:
        if( DoneA ) begin isStart <= 1' b0; Go <= i; i <= i + 1' b1; end
        else begin isStart <= 1' b1; end

    1:
        if( C1 == 4 -1) begin C1 <= 8' d0; i <= i + 1' b1; end
        else begin C1 <= C1 + 1' b1; i <= Go; end

    2:
        if( C2 == 8 -1) begin C2 <= 8' d0; i <= i + 1' b1; end
        else begin C2 <= C2 + 1' b1; i <= Go; end
```

代码 27.5

如代码 27.5 所示，我们只要在步骤 1 的下面再添加一段代码即可，其中 C1 控制步骤 0 执行 4 次，步骤 2 则控制步骤 1 执行 8 次。操作期间，如果 C1 不满足便会返回步骤 0，反之继续步骤；如果 C2 不满足也会返回步骤 0，反之继续。步骤之间不停来回跳转，如此一来，功能 A 总共执行 32 次。

即使对象是 3 层 for 嵌套，我们照搬无误，如代码 27.6 所示：

```
// C 语言，3 层嵌套 for
for( C3 = 0 ; C3 < 8 ; C3++ )
    for( C2 = 0 ; C2 < 8 ; C2++ )
        for( C1 = 0 ; C1 < 4 ; C1 ++ )
            FuncA();

// Verilog 语言
case( i )

    0:
        if( DoneA ) begin isStart <= 1' b0; Go <= i; i <= i + 1' b1; end
        else begin isStart <= 1' b1; end

    1:
        if( C1 == 4 -1) begin C1 <= 8' d0; i <= i + 1' b1; end
        else begin C1 <= C1 + 1' b1; i <= Go; end

    2:
```

```

if( C2 == 8 -1) begin C2 <= 8' d0; i <= i + 1' b1; end
else begin C2 <= C2 + 1' b1; i <= Go; end

3:
if( C3 == 10 -1) begin C3 <= 8' d0; i <= i + 1' b1; end
else begin C3 <= C3 + 1' b1; i <= Go; end

```

代码 27.6

如代码 27.6 所示，C 语言一共拥有 3 层 for 嵌套，反之 Verilog 只要再添加步骤 3 即可。期间，C3 控制 C2 执行 10 次，C2 控制 C1 执行 8 次，C1 则控制功能 A 执行 4 次 … 如此一来，功能 A 一共执行 320 次。如果一鼓作气下去的话，不管循环 for 有多少层也势如破竹。读者是不是觉得很神奇呢？然而，最神奇的地方是，步骤依然从上之下解读。

```

// C 语言， 3 层嵌套 for
for( C3 = 0 ; C3 < 8 ; C3++ )
    for( C2 = 0 ; C2 < 8 ; C2++ )
    {
        for( C1 = 0; C1 < 4; C1 ++ ) FuncA();
        FuncB();
    }
}

```

代码 27.7

假设笔者稍微顽皮一点，让 C2 控制 C1 以外，也让它控制函数 B。对此，Verilog 又该怎样描述呢？

```

// Verilog 语言
case( i )

0:
if( DoneA ) begin isStart[1] <= 1' b0; Go <= i; i <= i + 1' b1; end
else begin isStart[1] <= 1' b1; end

1:
if( C1 == 4 -1) begin C1 <= 8' d0; i <= i + 1' b1; end
else begin C1 <= C1 + 1' b1; i <= Go; end

2 :
if( DoneB ) begin isStart[0] <= 1' b0; i <= i + 1' b1; end
else begin isStart[0] <= 1' b1; end

3:
if( C2 == 8 -1) begin C2 <= 8' d0; i <= i + 1' b1; end
else begin C2 <= C2 + 1' b1; i <= Go; end

```

4:

```
if( C3 == 10 -1) begin C3 <= 8' d0; i <= i + 1' b1; end  
else begin C3 <= C3 + 1' b1; i <= Go; end
```

代码 27.8

如代码 27.8 所示，我们只要在 C2~C1 之间再插入一段步骤即可。期间，步骤 2 先执行功能 B，然后再进入步骤 3。如此一来，功能 A 一共执行 320 次，然而功能 B 一共执行 80 次。好奇的同学一定会觉得奇怪，怎么说说 TFT LCD 忽然插入循环操作的话题呢？原因很单纯，因为绘图功能必须借用循环操作才行。因此，笔者事先为读者洗白白了 ... 好了，理解完毕以后，我们便可以进入主题了？

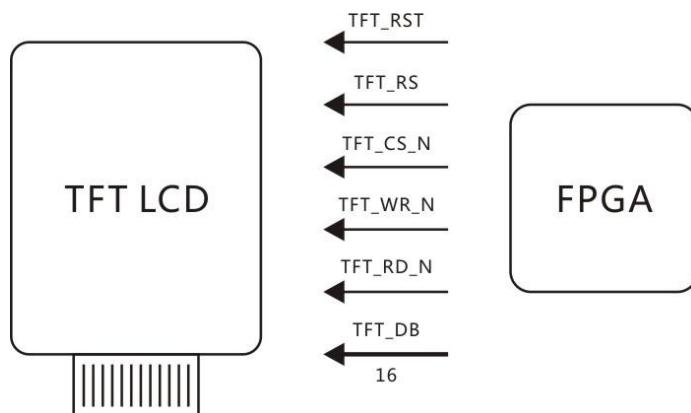


图 27.1 TFT 连线 FPGA。

一般而言，如果 TFT LCD 只是单纯地显示图像，然后 FPGA 也是单纯地写入地址或者写入图像信息 ... 对此，它们之间所需的连线并不多。如图 27.1 所示，RST 信号用来复位 TFT LCD，RS 信号用来分辨写入数据是命令还是图像信息，CS 是使能信号，WR 是写入有效信号，RD 是读出有效信号，DB 则是数据信号。此外，为了简化设计，FPGA 只需负责写数据而已，所以连线箭头往左一面倒。

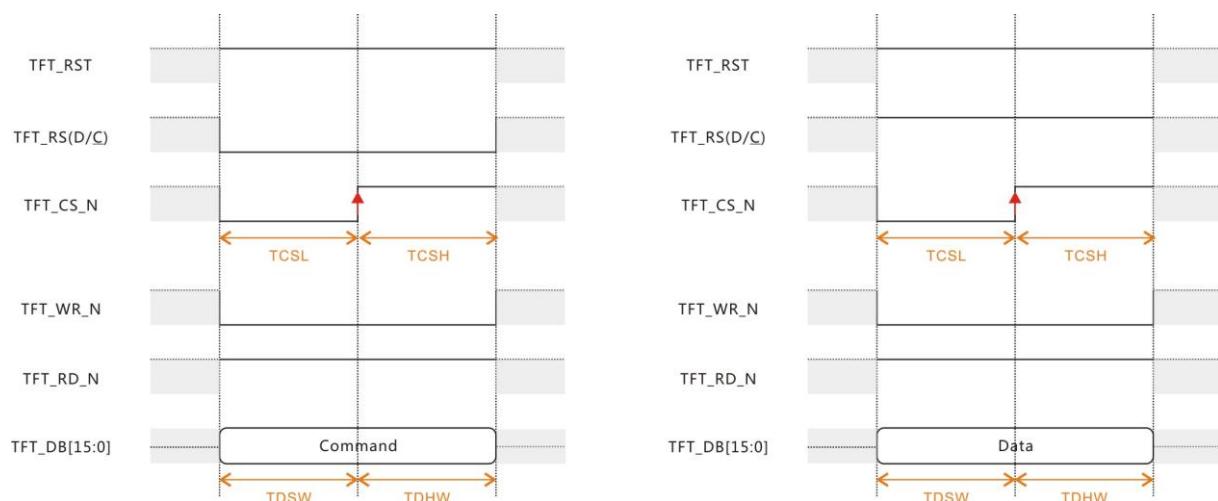


图 27.2 控制器 SSD1289 的写时序。

图 27.1 是控制器 SSD1289 的写命令还有写数据的时序图，左图是写命令，右图则是写数据。写命令与写数据的区别就在于 RS 信号拉低还是拉高而已，写命令拉低 RS，写数据则拉高 RS。如果我们不打算复位控制器，RST 信号可以常年拉高。此外，笔者也说过 FPGA 只写不读，所以 WR 信号常年拉低，RD 信号则是常年拉高。余下只有 CS 信号还有 DB 信号而已。

在这里，CS 信号充当时钟信号，因为 CS 信号由低变高所产生的上升沿会使 DB 信号的内容打入控制器里面。为使上升沿稳如泰山，我们必须满足 TCSL 与 TCSH 这两只时间要求。根据手册，TCSL 最少为 50ns，TCSH 最少则是 500ns，因此 CS 最小周期需要 550ns，或者说该控制器拥有速率 1.818181 Mhz。此外，TDSW 与 TDHW 都是常见的 TSETUP 与 THOLD，手册显示只有 5ns 而已，所以我们无须理会。

```
parameter TCSL = 3, TCSH = 25; // tCSDL = 50ns, tCSH = 500ns;
```

代码 27.9

如代码 27.9 所示，TCSL 与 TCSH 经过 50Mhz 量化以后，取得结果是 3 与 25。

```
1. case( i )
2.
3.   0:
4.     if( C1 == TCSL -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
5.     else begin { rRS,rCS } <= 2'b00; D1 <= { 8'd0, iAddr }; C1 <= C1 + 1'b1; end
6.
7.   1:
8.     if( C1 == TCSH -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
9.     else begin { rRS,rCS } <= 2'b01; C1 <= C1 + 1'b1; end
```

代码 27.10

如代码 27.10 所示，那是写命令操作。CS 在闲置状态下都处于拉高状态，步骤 0 拉低 RS 与 CS，还有赋值命令（地址），然后等待 TCSL。步骤 2 拉高 CS 之余，它也等待 TCSH。

```
1. case( i )
2.
3.   0:
4.     if( C1 == TCSL -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
5.     else begin { rRS,rCS } <= 2'b10; D1 <= iData; C1 <= C1 + 1'b1; end
6.
7.   1:
8.     if( C1 == TCSH -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
9.     else begin { rRS,rCS } <= 2'b11; C1 <= C1 + 1'b1; end
```

代码 27.11

如代码 27.11 所示，那是写数据。默认下，CS 处于高电平。步骤 0，拉高 RS 之余也拉低 CS，准备写命令之后，便等待 TCSL。步骤 1，拉高 RS 之余也拉高 CS，然后等待 TCSH。基本上，控制器 SSD1289 的写时序就是那么简单而已，接下来就是该控制器的配置信息。

控制器 SSD1289 内置超过 50 个配置寄存器，如果逐个解释，笔者一定会捏爆自己的蛋蛋 ... 对此，笔者仅对看懂又重要的寄存器解释而已。

Oscillator (R00h) (POR = 0000h)

R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
W	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	OSCEN
POR	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	

图 27.3 Oscillator 配置寄存器。

如图 27.3 所示，那是 Oscillator 配置寄存器。它可谓是最简单的寄存器，IB0 为 1，内部的晶振就开始鼓动，反之亦然。

Driver Output Control (R01h) (POR = 433Fh)

R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
W	1	0	RL	REV	CAD	BGR	SM	TB	MUX8	MUX7	MUX6	MUX5	MUX4	MUX3	MUX2	MUX1	MUX0
POR	-	1	0	0	0	0	1	1	0	0	1	1	1	1	1	1	

图 27.4 Driver Output Control 配置寄存器。

如图 27.4 所示，那是 Driver Output Control 配置寄存器。MUX0~8 用来设置 TFT 的显示高度（Vertical），最大为 319（从 0 开始计算）或者 0x13F。BGR 用来设置颜色的排序，BGR 为 1，颜色排序为 <R><G>，为 0 则是 <G><R>。

Sleep mode (R10h) (POR = 0001h)

R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
W	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SLP
POR	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	

图 27.5 Sleep Mode 配置寄存器。

如图 27.5 所示，那是 Sleep Mode 配置寄存器，其中 IB0 为 1 表示控制器在睡觉。我们只要将其设置为 0，该控制器就会起床。

Entry Mode (R11h) (POR = 6830h)

R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
W	1	VMode	DFM1	DFM0	TRANS	OEDef	WMode	DMode1	DMode0	TY1	TY0	ID1	ID0	AM	LG2	LG1	LG0
POR	0	1	1	0	1	0	0	0	0	0	0	1	1	0	0	0	

图 27.6 Entry Mode 配置寄存器

图 27.6 所示是 Entry Mode 配置寄存器，它可谓重量级的配置寄存器之一。DFM 表示色彩的分辨率，DFM 为 2'b11 就是 16 位 RGB（65k 颜色），DFM 为 2'b10 就是 18

位 RGB (262k)。如果选择 16 位 RGB , 我们可以无视 TY。DMode 为 2'b00 , 表示显示 ram 里边的图像信息。

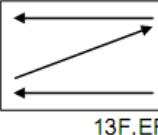
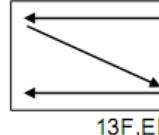
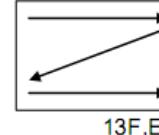
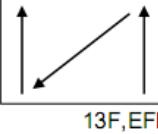
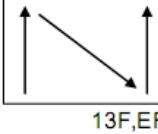
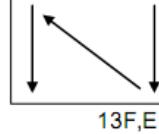
	ID[1:0] = "00" Horizontal: decrement Vertical: decrement	ID[1:0] = "01" Horizontal: increment Vertical: decrement	ID[1:0] = "10" Horizontal: decrement Vertical: increment	ID[1:0] = "11" Horizontal: increment Vertical: increment
AM = "0" Horizontal	00,00h 	00,00h 	00,00h 	00,00h 
AM = "1" Vertical	00,00h 	00,00h 	00,00h 	00,00h 

图 27.7 扫描次序。

再者就是 ID 与 AM 了 , 根据配置内容不同 , 控制器也会产生不同的扫描次序 , 结果如图 27.7 所示。笔者选择先左至右 , 后上至下的扫描次序 , 目的是为了迎合 VGA 的扫描次序 , 所以 AM 设置为 0 , ID 则是 2'b11。

Horizontal Porch (R16h) (POR = EF1Ch)

R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
W	1	XL7	XL6	XL5	XL4	XL3	XL2	XL1	XL0	HBP7	HBP6	HBP5	HBP4	HBP3	HBP2	HBP1	HBP0

图 27.8 Horizontal Porch 配置寄存器

图 27.8 显示 Horizontal Porch 配置寄存器的内容 , 其中 XL 是 HSYNC 的数据段长度 , HBP 则是起始段还有准备段的长度。读者是否还记得 VGA 时序 ? 我们利用 HSYNC 信号还有 VSYNC 信号控制 VGA 的显示标准。同样 , TFT LCD 内部也使用了 VGA 时序 , 不过驱动对象却是控制器 SSD 1289。

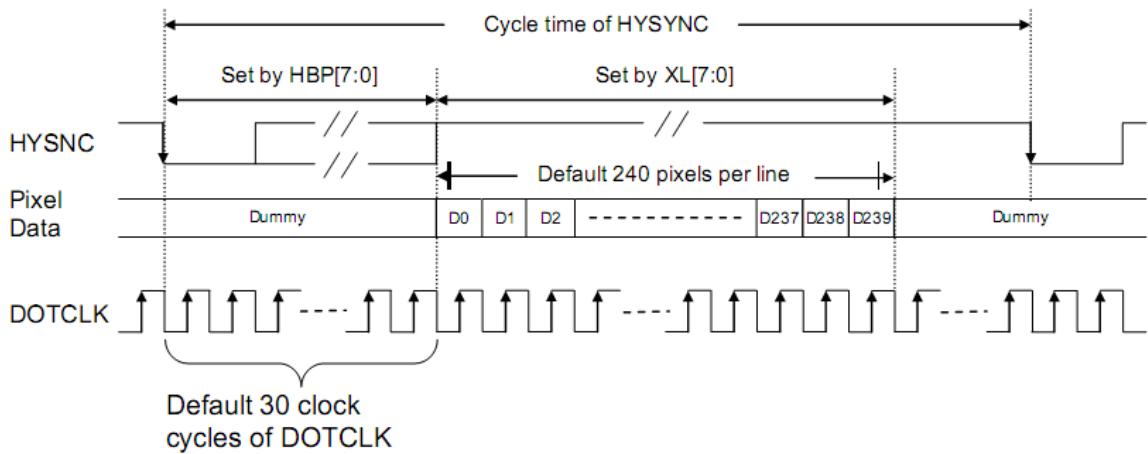


图 27.9 TFT LCD 内部的 HSYNC 时序。

如图 27.9 所示，那是 TFT LCD 内部的 HSYNC 时序。其中 HBP 配置起始段还有准备段的长度，HBP 默认下为 30，配置信息则是 8'b1C。换之，XL 用来控制数据段的长度，而且 240 即是默认值也是最大值，配置信息则是 8'hEF。

Vertical Porch (R17h) (POR = 0003h)

R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
POR	-	0	VFP6	VFP5	VFP4	VFP3	VFP2	VFP1	VFP0	VBP7	VBP6	VBP5	VBP4	VBP3	VBP2	VBP1	VBP0

图 27.9 Vertical Porch 配置寄存器。

图 27.9 显示 Vertical Porch 配置寄存器的内容，亦即控制内部的 VSYNC 信号。VFP 用来配置结束段的长度，VBP 则是配置起始段还有准备段的长度。

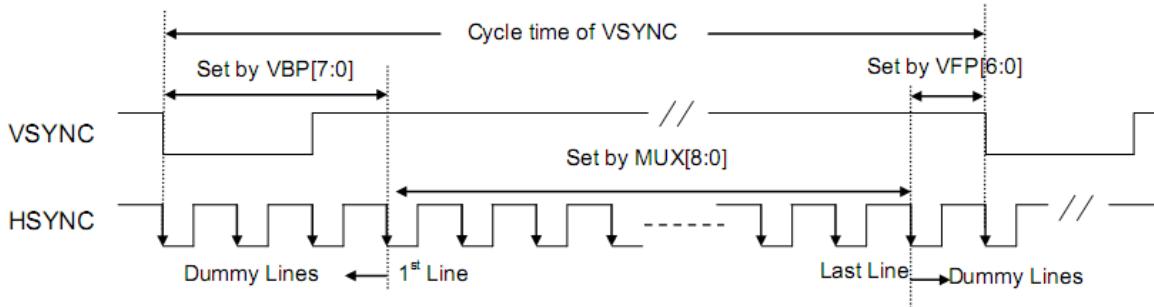


图 27.10 TFT LCD 内部的 CSYNC 时序。

如图 27.10 所示，VBP 的默认长度为 4 个 HSYNC 的下降沿(起始段)，结果默认值为 4，配置信息则是 8'h03。换之，VFP 的默认长度为 1 个 HSYNC 周期，所以默认值为 1，配置信息则是 8'h00。至于 VSYNC 的数据段则在 Driver Output Control 哪里配置。

Display Control (R07h) (POR = 0000h)

R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
W	1	0	0	0	PT1	PT0	VLE2	VLE1	SPT	0	0	GON	DTE	CM	0	D1	D0
POR	-	-	-	0	0	0	0	0	0	-	-	0	0	0	-	0	0

图 27.11 Display Control 配置寄存器。

图 27.11 是 Display Control 配置寄存器，而重点内容就在 D1 与 D0。D1 控制屏幕开关，值 1 显示，值 0 关闭。D0 控制控制器的活动状态，值 1 干活，值 0 挂机。为此，屏幕正常活动的时候 D1 与 D0 必须设为 2'b11。

Gate Scan Position (R0Fh) (POR = 0000h)

R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
W	1	0	0	0	0	0	0	0	SCN8	SCN7	SCN6	SCN5	SCN4	SCN3	SCN2	SCN1	SCN0
POR	-	-	-	-	-	-	-	0	0	0	0	0	0	0	0	0	

图 27.12 Gate Scan Position 配置寄存器。

图 27.12 是 Gate Scan Position 配置寄存器，其中 SCN 表示扫描的起始位置。

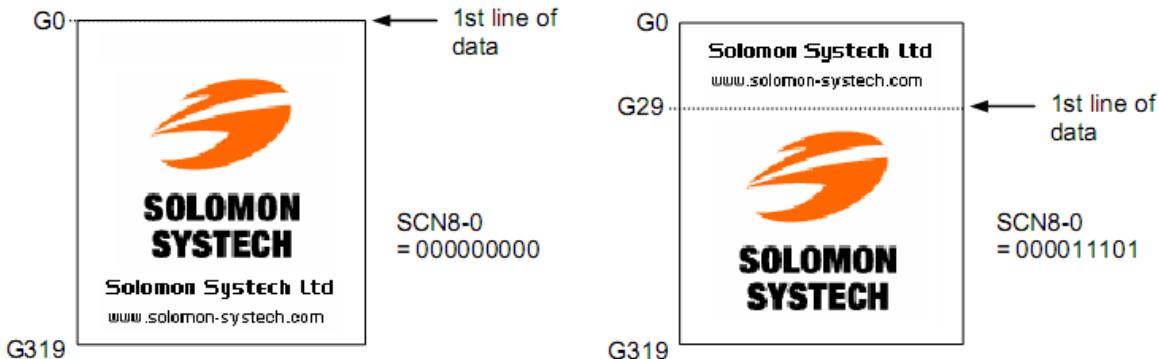


图 27.13 默认扫描起始位置 (左)，配置过后的扫描起始位置 (右)。

如图 27.13 所示，左图是默认下的起始位置，右图则是从 29 行开始扫描，结果文字信息与图标信息调转位置了。所以，SCN 一般都设为 0 值。

1st Screen driving position (R48h-R49h)

Reg#	R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
R48h	W	1	0	0	0	0	0	0	SS18	SS17	SS16	SS15	SS14	SS13	SS12	SS11	SS10	
	POR	-	-	-	-	-	-	-	0	0	0	0	0	0	0	0	0	
R49h	W	1	0	0	0	0	0	0	SE18	SE17	SE16	SE15	SE14	SE13	SE12	SE11	SE10	
	POR	-	-	-	-	-	-	-	1	0	0	1	1	1	1	1	1	

图 27.14 1st Screen Driving Position 配置寄存器。

图 27.14 是 1st Screen Driving Position 配置寄存器。黑金所用的 TFT LCD，主要分为主屏幕 (First Screen) 还有次屏幕 (Second Screen)。主屏幕一般作为主要显示源，而且扫描也是一行一行执行，期间 SS1 表示扫描的开始位置，SE1 则表示扫描的结束位置。默认下，SS1 为 0，配置信息则是 9'd0，SE1 为 319，配置信息则是 9'h13F。

Horizontal RAM address position (R44h) (POR = EF00h)																	
R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
W	1	HEA7	HEA6	HEA5	HEA4	HEA3	HEA2	HEA1	HEA0	HSA7	HSA6	HSA5	HSA4	HSA3	HSA2	HSA1	HSA0
POR	1	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0

图 27.15 Horizontal RAM Address Position 配置寄存器。

图 27.15 是 Horizontal RAM Address Position 配置寄存器。HSA 表示有效的起始列，默认下为 0，配置信息则是 8'h00。换之，HEA 表示有效的结束列，默认下为 239，配置信息则是 8'hEF。

Vertical RAM address position (R45h-R46h)																		
Reg#	R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
R45h	W	1	0	0	0	0	0	0	0	VSA8	VSA7	VSA6	VSA5	VSA4	VSA3	VSA2	VSA1	VSA0
	POR	-	-	-	-	-	-	-	0	0	0	0	0	0	0	0	0	
R46h	W	1	0	0	0	0	0	0	0	VEA8	VEA7	VEA6	VEA5	VEA4	VEA3	VEA2	VEA1	VEA0
	POR	-	-	-	-	-	-	-	1	0	0	1	1	1	1	1	1	

图 27.16 Vertical RAM Address Position 配置寄存器

图 27.16 是 Vertical RAM Address Position 配置寄存器。VSA 表示有效的起始行，默认下为 0，配置信息则是 9'h000。VEA 表示有效的结束行，默认下为 319，配置信息则是 9'h13F。

感觉上，Horizontal RAM Address Position 好比 vga_ctrlmod 的 isX，Vertical RAM Address Position 好比 isY，两者求与后便构成 isReady。例如实验二十六，显示空间虽然有 1024×768 的范围，不过笔者只用左上一角显示 128×96 的图像而已。其中，列 0 至列 127 之间为有效 X，行 0 至行 95 之间为有效 Y，结果两者构成有效的显示区域。

RAM write data mask (R23h – R24h) (POR = 0000h)																		
Reg#	R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
R23h	W	1	WMR	WMR	WMR	WMR	WMR	WMR	0	0	WMG	WMG	WMG	WMG	WMG	WMG	0	0
	POR	0	0	0	0	0	0	0	-	-	0	0	0	0	0	-	-	
R24h	W	1	0	0	0	0	0	0	0	0	WMB	WMB	WMB	WMB	WMB	WMB	0	0
	POR	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0	-	-

图 27.17 RAM Write Data Mask 配置寄存器。

图 27.17 是 RAM Write Data Mask 配置寄存器，WMR 表示红色源的屏蔽信息，WMG 表示绿色源的屏蔽信息，WMB 则是蓝色源的屏蔽信息。值 1 表示屏蔽有效，值 0 表示屏蔽无效。屏蔽一旦启动，相关的颜色位便会写入失效。其实这些家伙并没有多大用处，笔者也是循例介绍而已。

RAM address set (R4Eh-R4Fh)																		
Reg#	R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
R4Eh	W	1	0	0	0	0	0	0	0	0	XAD7	XAD6	XAD5	XAD4	XAD3	XAD2	XAD1	XAD0
	POR	-	-	-	-	-	-	-	-	0	0	0	0	0	0	0	0	
R4Fh	W	1	0	0	0	0	0	0	0	YAD8	YAD7	YAD6	YAD5	YAD4	YAD3	YAD2	YAD1	YAD0
	POR	-	-	-	-	-	-	-	0	0	0	0	0	0	0	0	0	

图 27.18 RAM Address set 配置寄存器

图 27.18 是 RAM Address set 配置寄存器，XAD 表示列计数器的内容，YAD 则表示行计数器的内容。写数据期间，CS 每次上升沿都使 XAD 递增，直至 239 为止便会清零（默认下），然后递增 YAD。默认下，YAD 递增到 319 也会清零。每当写数据之前，我们都会顺手将它们设为 0 值。

Write Data to GRAM (R22h)																	
R/W	DC	D[17:0]															
W	1	WD[17:0] mapping depends on the interface setting															

图 27.19 Write Data to CGRAM 寄存器。

图 27.19 是 Write Data to CGRAM 寄存器。根据手册，写图像信息之前，我们必须事先设置写数据的目标地址 ... 然而，那个目标地址就是 Write Data to GRAM 寄存器。随后，写入信息再由控制器写入内部 CGRAM。基本上，有用又看懂的寄存器就是这些而已，接下来让我们来瞧瞧如何初始化控制器 SSD 1289。

因为犯懒的关系，笔者尝试将写命令和写数据捆绑在一起，结果如代码 27.12 所示：

```

1.   case( i )
2.
3.       0:
4.           if( C1 == TCSL -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
5.           else begin { rRS,rCS } <= 2'b00; D1 <= { 8'd0, iAddr }; C1 <= C1 + 1'b1; end
6.
7.       1:
8.           if( C1 == TCSH -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
9.           else begin { rRS,rCS } <= 2'b01; C1 <= C1 + 1'b1; end
10.
11.      2:
12.          if( C1 == TCSL -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
13.          else begin { rRS,rCS } <= 2'b10; D1 <= iData; C1 <= C1 + 1'b1; end
14.
15.      3:
16.          if( C1 == TCSH -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
17.          else begin { rRS,rCS } <= 2'b11; C1 <= C1 + 1'b1; end

```

代码 27.12

如代码 27.12 所示，步骤 0~1 是写命令（地址），步骤 2~3 则是写数据。如此一来，笔者只要调用一次该功能便能同时执行写命令还有写数据。

根据官方源码，控制器 SSD1289 的初始化过程是很长很臭的，对此让笔者 N 行，N 行慢慢解释吧。

```
1. case( i )
2.
3.     0: // Oscillator, On
4.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
5.     else begin isDo[2] <= 1'b1; D1 <= 8'h00; D2 <= 16'h0001; end
```

代码 27.13

假设拉高 isDo[2] 调用代码 27.12，步骤 0 开启使能晶振。

```
1.     1: // Power control 1
2.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
3.     else begin isDo[2] <= 1'b1; D1 <= 8'h03; D2 <= 16'h6664; end
4.
5.     2: // Power control 2
6.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
7.     else begin isDo[2] <= 1'b1; D1 <= 8'h0C; D2 <= 16'h0000; end
8.
9.     3: // Power control 3
10.    if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
11.    else begin isDo[2] <= 1'b1; D1 <= 8'h0D; D2 <= 16'h080C; end
12.
13.    4: // Power control 4
14.    if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
15.    else begin isDo[2] <= 1'b1; D1 <= 8'h0E; D2 <= 16'h2B00; end
16.
17.    5: // Power control 5
18.    if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
19.    else begin isDo[2] <= 1'b1; D1 <= 8'h1E; D2 <= 16'h00B0; end
```

代码 27.14

步骤 1~5 是相关的电源配置信息，别问笔者为什么，笔者也是照搬而已。

```
1.     6: // Driver output control, MUX = 319 , <R><G><B>
2.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
3.     else begin isDo[2] <= 1'b1; D1 <= 8'h01; D2 <= 16'h2B3F; end
4.
5.     7: // LCD driving waveform control
```

```

6.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
7.      else begin isDo[2] <= 1'b1; D1 <= 8'h02; D2 <= 16'h0600; end
8.
9.      8: // Sleep mode, weak-up
10.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
11.     else begin isDo[2] <= 1'b1; D1 <= 8'h10; D2 <= 16'h0000; end
12.
13.     9: // Entry mode, 65k color, DM = 2' b00, AM = 0, ID = 2' b11
14.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
15.     else begin isDo[2] <= 1'b1; D1 <= 8'h11; D2 <= 16'h6070; end

```

代码 27.15

步骤 6 用来配置 VSYNC 的数据段长度之余，也设置 RGB 的排序。步骤 7 不清楚，步骤 8 唤醒控制器。步骤 9 用来配置 16 位 RGB，还有扫描次序。

```

1.      10: // Compare register
2.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
3.      else begin isDo[2] <= 1'b1; D1 <= 8'h05; D2 <= 16'h0000; end
4.
5.      11: // Compare register
6.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
7.      else begin isDo[2] <= 1'b1; D1 <= 8'h06; D2 <= 16'h0000; end
8.
9.      12: // Horizontal porch, HBP = 30, XL = 239
10.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
11.     else begin isDo[2] <= 1'b1; D1 <= 8'h16; D2 <= 16'hEF1C; end
12.
13.      13: // Vertical porch, VBP = 1, VBP = 4
14.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
15.      else begin isDo[2] <= 1'b1; D1 <= 8'h17; D2 <= 16'h0003; end
16.
17.      14: // Display control, 8 color mode, display on, operation on
18.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
19.      else begin isDo[2] <= 1'b1; D1 <= 8'h07; D2 <= 16'h0233; end

```

代码 27.16

步骤 10~11 不清楚，步骤 12 用来配置 HSYNC，步骤 13 则用来配置 VSYNC。步骤 14 用来配置显示器开启，控制器进入活动。

```

1.      15: // Frame cycle control
2.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
3.      else begin isDo[2] <= 1'b1; D1 <= 8'h0B; D2 <= 16'h0000; end
4.

```

```

5.      16: // Gate scan position, SCN = 0
6.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
7.      else begin isDo[2] <= 1'b1; D1 <= 8'h0F; D2 <= 16'h0000; end
8.
9.      17: // Vertical scroll control
10.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
11.     else begin isDo[2] <= 1'b1; D1 <= 8'h41; D2 <= 16'h0000; end
12.
13.      18: // Vertical scroll control
14.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
15.      else begin isDo[2] <= 1'b1; D1 <= 8'h42; D2 <= 16'h0000; end

```

代码 27.16

步骤 15 不清楚，步骤 16 配置扫描的起始位置。步骤 17~18 好像与拖动效果有关，具体内容并不清楚，也不使用。

```

1.      19: // 1st screen driving position, G0~
2.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
3.      else begin isDo[2] <= 1'b1; D1 <= 8'h48; D2 <= 16'h0000; end
4.
5.      20: // 1st screen driving position, ~G319
6.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
7.      else begin isDo[2] <= 1'b1; D1 <= 8'h49; D2 <= 16'h013F; end
8.
9.      21: // 2nd screen driving position
10.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
11.     else begin isDo[2] <= 1'b1; D1 <= 8'h4A; D2 <= 16'h0000; end
12.
13.      22: // 2nd screen driving position
14.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
15.      else begin isDo[2] <= 1'b1; D1 <= 8'h4B; D2 <= 16'h0000; end

```

代码 27.17

步骤 19~20 用来配置主屏幕的扫描范围，步骤 21~22 则是用来配置次屏幕的扫描范围，不过只有主屏幕被使用而已。

```

1.      23: // Horizontal RAM address position, HSA = 0 HEA = 239
2.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
3.      else begin isDo[2] <= 1'b1; D1 <= 8'h44; D2 <= 16'hEF00; end
4.
5.      24: // Vertical RAM address position, VSA = 0
6.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
7.      else begin isDo[2] <= 1'b1; D1 <= 8'h45; D2 <= 16'h0000; end

```

```

8.
9.      25: // Vertical RAM address position, VEA = 319
10.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
11.     else begin isDo[2] <= 1'b1; D1 <= 8'h46; D2 <= 16'h013F; end

```

代码 27.18

步骤 23 用来配置有效列，步骤 24~25 则是用来配置有效行。

```

1.      26: // Gamma control, PKP0~1
2.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
3.      else begin isDo[2] <= 1'b1; D1 <= 8'h30; D2 <= 16'h0707; end
4.
5.      27: // Gamma control, PKP2~3
6.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
7.      else begin isDo[2] <= 1'b1; D1 <= 8'h31; D2 <= 16'h0204; end
8.
9.      28: // Gamma control, PKP4~5
10.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
11.     else begin isDo[2] <= 1'b1; D1 <= 8'h32; D2 <= 16'h0204; end
12.
13.     29: // Gamma control, PRP0~1
14.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
15.     else begin isDo[2] <= 1'b1; D1 <= 8'h33; D2 <= 16'h0502; end
16.
17.     30: // Gamma control, PKN0~1
18.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
19.     else begin isDo[2] <= 1'b1; D1 <= 8'h34; D2 <= 16'h0507; end
20.
21.     31: // Gamma control, PKN2~3
22.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
23.     else begin isDo[2] <= 1'b1; D1 <= 8'h35; D2 <= 16'h0204; end
24.
25.     32: // Gamma control, PKN4~5
26.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
27.     else begin isDo[2] <= 1'b1; D1 <= 8'h36; D2 <= 16'h0204; end
28.
29.     33: // Gamma control, PRN0~1
30.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
31.     else begin isDo[2] <= 1'b1; D1 <= 8'h37; D2 <= 16'h0502; end
32.
33.     34: // Gamma control, VRP0~1
34.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
35.     else begin isDo[2] <= 1'b1; D1 <= 8'h3A; D2 <= 16'h0302; end

```

```

36.
37.      35: // Gamma control, VRN0~1
38.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
39.      else begin isDo[2] <= 1'b1; D1 <= 8'h3B; D2 <= 16'h0302; end

```

代码 27.19

步骤 26~35 好像是用来配置屏幕的亮度或者对比度，虽然手册有详细的介绍与公式，不过真心看不懂，所以照搬而已。

```

1.      36: // RAM write data mask
2.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
3.      else begin isDo[2] <= 1'b1; D1 <= 8'h23; D2 <= 16'h0000; end
4.
5.      37: // RAM write data mask
6.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
7.      else begin isDo[2] <= 1'b1; D1 <= 8'h24; D2 <= 16'h0000; end
8.
9.      38: // Unknown
10.     if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
11.     else begin isDo[2] <= 1'b1; D1 <= 8'h25; D2 <= 16'h8000; end
12.
13.      39: // RAM address set, horizontal(X)
14.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
15.      else begin isDo[2] <= 1'b1; D1 <= 8'h4E; D2 <= 16'h0000; end
16.
17.      40: // RAM address set, vertical(Y)
18.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
19.      else begin isDo[2] <= 1'b1; D1 <= 8'h4F; D2 <= 16'h0000; end

```

代码 27.20

步骤 36~37 用来配置颜色源的屏蔽信息，步骤 38 不明因为手册没有解释，步骤 39~40 则是用来配置 X 计数器与 Y 计数器的初值。基本上，控制器 SSD1289 的初始化就这样而已。准备知识理解完毕以后，我们便可以开始建模了。

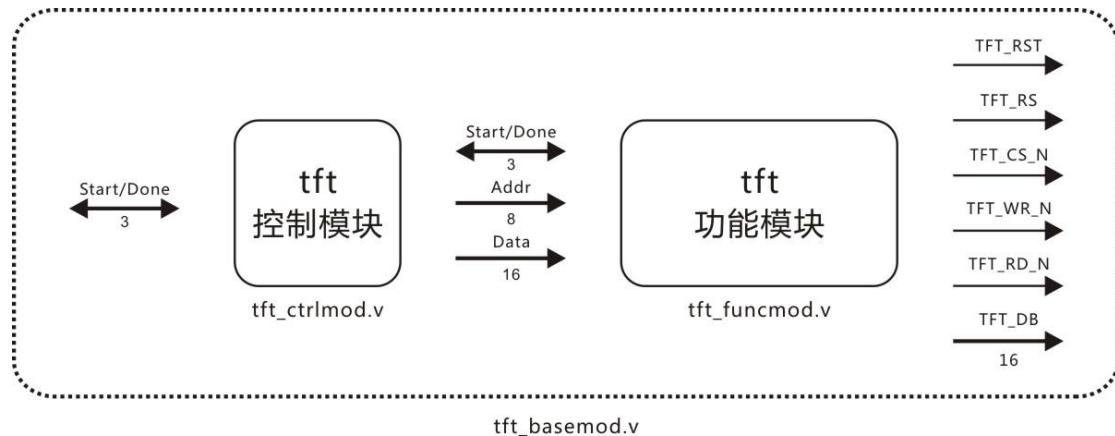


图 27.20 TFT 基础模块的建模图。

图 27.20 是 TFT 基础模块的建模图，TFT 功能模块作有三位宽的沟通信号，结果表示它有 3 项功能，[2]为写命令与写数据，[1]为写命令，[0]为写数据。换之，右边则是驱动 TFT LCD 的顶层信号。至于 TFT 控制模块除了调用功能模块以外，左边也有 3 位宽的沟通信号，其中[0]为初始化，[1]为清屏，[2]为画图。

tft_funcmod.v

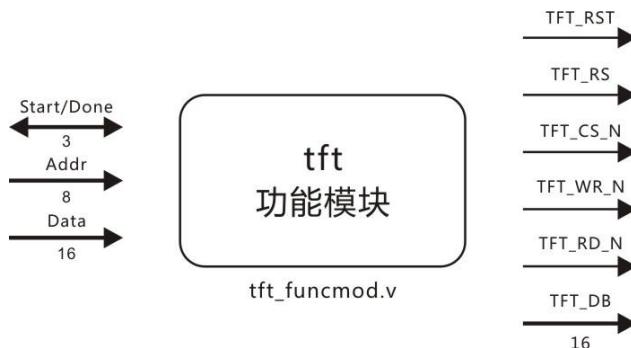


图 27.21 TFT 功能模块的建模图。

图 27.21 是 TFT 功能模块的建模图，它虽然浑身布满箭头，不过它还是简单易懂的好家伙。左边的 Start/Done 有 3 位宽，恰好表示它有 3 个功能，其中[2]为写命令再写数据，[1]为写命令，[0]则是写数据。iAddr 是写入命令所需的内容，iData 则是写数据所需的内容。

```

1. module tft_funcmod
2. (
3.     input CLOCK, RESET,
4.     output TFT_RS,      // 1 = Data, 0 = Command(Register)
5.     output TFT_CS_N,
6.     output TFT_WR_N,
7.     output TFT_RD_N,

```

```

8.      output [15:0]TFT_DB,
9.      input [2:0]iStart,
10.     output oDone,
11.     input [7:0]iAddr,
12.     input [15:0]iData
13.   );
14.   parameter TCSL = 3, TCSH = 25; // tCSL = 50ns, tCSH = 500ns;
15.

```

以上内容为相关的出入端声明还有相关的常量声明。

```

16.   reg [3:0]i;
17.   reg [4:0]C1;
18.   reg [15:0]D1;
19.   reg rRS,rCS,rWR,rRD;
20.   reg isDone;
21.
22.   always @ ( posedge CLOCK or negedge RESET )
23.     if( !RESET )
24.       begin
25.         i <= 4'd0;
26.         C1 <= 5'd0;
27.         D1 <= 16'd0;
28.         { rRS, rCS, rWR, rRD } <= 4'b1101;
29.         isDone <= 1'b0;
30.       end

```

以上内容为相关的寄存器声明还有复位操作，其中第 28 行表示 WR 常年拉低，RD 则是常年拉高。

```

31.   else if( iStart[2] ) // Write command and data
32.     case( i )
33.
34.       0:
35.         if( C1 == TCSL -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
36.         else begin { rRS,rCS } <= 2'b00; D1 <= { 8'd0, iAddr }; C1 <= C1 + 1'b1; end
37.
38.       1:
39.         if( C1 == TCSH -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
40.         else begin { rRS,rCS } <= 2'b01; C1 <= C1 + 1'b1; end
41.
42.       2:
43.         if( C1 == TCSL -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end

```

```

44.           else begin { rRS,rCS } <= 2'b10; D1 <= iData; C1 <= C1 + 1'b1; end
45.
46.           3:
47.             if( C1 == TCSH -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
48.             else begin { rRS,rCS } <= 2'b11; C1 <= C1 + 1'b1; end
49.
50.           4:
51.             begin isDone <= 1'b1; i <= i + 1'b1; end
52.
53.           5:
54.             begin isDone <= 1'b0; i <= 4'd0; end
55.
56.         endcase

```

以上内容为写命令再写数据。

```

57.       else if( iStart[1] ) // Write command
58.         case( i )
59.
60.           0:
61.             if( C1 == TCSL -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
62.             else begin { rRS,rCS } <= 2'b00; D1 <= { 8'd0, iAddr }; C1 <= C1 + 1'b1; end
63.
64.           1:
65.             if( C1 == TCSH -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
66.             else begin { rRS,rCS } <= 2'b01; C1 <= C1 + 1'b1; end
67.
68.           2:
69.             begin isDone <= 1'b1; i <= i + 1'b1; end
70.
71.           3:
72.             begin isDone <= 1'b0; i <= 4'd0; end
73.
74.         endcase

```

以上内容为写命令。

```

75.       else if( iStart[0] ) // Write data
76.         case( i )
77.
78.           0:
79.             if( C1 == TCSL -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
80.             else begin { rRS,rCS } <= 2'b10; D1 <= iData; C1 <= C1 + 1'b1; end

```

```

81.
82.           1:
83.           if( C1 == TCSH -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
84.           else begin { rRS,rCS } <= 2'b11; C1 <= C1 + 1'b1; end
85.
86.           2:
87.           begin isDone <= 1'b1; i <= i + 1'b1; end
88.
89.           3:
90.           begin isDone <= 1'b0; i <= 4'd0; end
91.
92.       endcase
93.

```

以上内容为写数据。

```

94.     assign TFT_DB = D1;
95.     assign TFT_RS = rRS;
96.     assign TFT_CS_N = rCS;
97.     assign TFT_WR_N = rWR;
98.     assign TFT_RD_N = rRD;
99.     assign oDone = isDone;
100.
101. endmodule

```

以上内容为相关的输出驱动声明。

tft_ctrlmod.v



图 27.22 TFT 控制模块的建模图。

图 27.22 是 TFT 控制模块的建模图，它外表虽然简单，内容却很啰嗦 ... 具体情况让我们来看代码吧。

```

1. module tft_ctrlmod
2. (

```

```
3.      input CLOCK, RESET,
4.      input [2:0]iStart,
5.      output oDone,
6.      output [2:0]oStart,
7.      input iDone,
8.      output [7:0]oAddr,
9.      output [15:0]oData
10. );
```

以上内容为相关的出入端声明。

```
11.     reg [5:0]i,Go;
12.     reg [7:0]D1;    // Command(Register)
13.     reg [15:0]D2;   // Data
14.     reg [16:0]C1;
15.     reg [7:0]C2,C3;
16.     reg [2:0]isDo;
17.     reg isDone;
18.
19.     always @ ( posedge CLOCK or negedge RESET )
20.         if( !RESET )
21.             begin
22.                 { i,Go } <= { 6'd0,6'd0 };
23.                 D1 <= 8'd0;
24.                 D2 <= 16'd0;
25.                 C1 <= 17'd0;
26.                 { C2,C3 } <= { 8'd0,8'd0 };
27.                 isDo <= 3'd0;
28.                 isDone <= 1'b0;
29.             end
```

以上内容为相关的寄存器声明还有复位操作。其中第 16 行的 isDo 作用类似 iStart , D1 暂存地址 (命令) , D2 暂存读写数据 , C1~C3 则是用来控制循环。

```
30.     else if( iStart[2] )  // White blank page
31.         case( i )
32.
33.             0: // X0
34.                 if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
35.                 else begin isDo[2] <= 1'b1; D1 <= 8'h4E; D2 <= 16'h0000; end
36.
37.             1: // Y0
38.                 if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
```

```

39.           else begin isDo[2] <= 1'b1; D1 <= 8'h4F; D2 <= 16'h0000; end
40.
41.           2: // Write data to ram 0x22
42.           if( iDone ) begin isDo[1] <= 1'b0; i <= i + 1'b1; end
43.           else begin isDo[1] <= 1'b1; D1 <= 8'h22; end
44.
45.           *****/
46.
47.           3: // Write color
48.           if( iDone ) begin isDo[0] <= 1'b0; i <= i + 1'b1; Go <= i; end
49.           else begin isDo[0] <= 1'b1; D2 <= { C3[4:0],6'd0,5'd0 } ; end
50.
51.           4: // Loop 1, rectangle width
52.           if( C1 == 240 -1 ) begin C1 <= 17'd0; i <= i + 1'b1; end
53.           else begin C1 <= C1 + 1'b1; i <= Go; end
54.
55.           5: // Loop 2, rectangle high
56.           if( C2 == 10 -1 ) begin C2 <= 8'd0; i <= i + 1'b1; end
57.           else begin C2 <= C2 + 1'b1; i <= Go; end
58.
59.           6: // Loop 3, color and times
60.           if( C3 == 32 -1 ) begin C3 <= 8'd0; i <= i + 1'b1; end
61.           else begin C3 <= C3 + 1'b1; i <= Go; end
62.
63.           *****/
64.
65.           7:
66.           begin isDone <= 1'b1; i <= i + 1'b1; end
67.
68.           8:
69.           begin isDone <= 1'b0; i <= 6'd0; end
70.
71.       endcase

```

以上内容为绘图功能。步骤 0 配置写入地址 X 为 0，步骤 1 配置写入地址 Y 为 0，步骤 2 配置写数据的目标地址。步骤 3 则是写数据，其中 { C3[4:0], 6'd0, 5'd0 }，红色信息取自 C3，绿色为 0，蓝色为 0。步骤 4，C1 控制循环 240 次，主要是写完 1 行 240 列。步骤 5，C2 用来控制 C1，主要是控制矩形的高度为 10。步骤 6，C3 用来控制 C2 之余，它也用来控制矩形的数量还有颜色渐变。

```

for( C3 = 0; C3 < 32; C3++ )
    for( C2 = 0; C2 < 10; C2++ )
        for( C1 = 0; C1 < 240; C1++ )

```

```
Write_Data( C3,0,0 ); // <R><G><B>
```

代码 27.20

简单来说，操作过程如代码 27.20 所示。

```
72.     else if( iStart[1] ) // White blank page
73.         case( i )
74.
75.             0: // X0
76.                 if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
77.                 else begin isDo[2] <= 1'b1; D1 <= 8'h4E; D2 <= 16'h0000; end
78.
79.             1: // Y0
80.                 if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
81.                 else begin isDo[2] <= 1'b1; D1 <= 8'h4F; D2 <= 16'h0000; end
82.
83.             2: // Write data to ram 0x22
84.                 if( iDone ) begin isDo[1] <= 1'b0; i <= i + 1'b1; end
85.                 else begin isDo[1] <= 1'b1; D1 <= 8'h22; end
86.
87.             /*****
88.
89.             3: // Write white color 0~768000
90.                 if( iDone ) begin isDo[0] <= 1'b0; i <= i + 1'b1; Go <= i; end
91.                 else begin isDo[0] <= 1'b1; D2 <= 16'hFFFF ; end
92.
93.             4:
94.                 if( C1 == 76800 -1 ) begin C1 <= 17'd0; i <= i + 1'b1; end
95.                 else begin C1 <= C1 + 1'b1; i <= Go; end
96.
97.             5:
98.                 begin isDone <= 1'b1; i <= i + 1'b1; end
99.
100.            6:
101.                begin isDone <= 1'b0; i <= 6'd0; end
102.
103.        endcase
```

以上内容为清屏功能。步骤 0 配置写入地址 X，步骤 1 则是配置写入地址 Y，步骤 2 配置写数据的目的地址。步骤 3 将白色的图像信息写入 CGRAM，步骤 4 则是控制步骤 3 执行 76800 次 (0~76799)，因为 320×240 等价 76800，完后便能达成清屏功能。

```
104.     else if( iStart[0] ) // Initial TFT
```

```

105.    case( i )
106.
107.        0: // Oscillator, On
108.            if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
109.            else begin isDo[2] <= 1'b1; D1 <= 8'h00; D2 <= 16'h0001; end
110.
111.        1: // Power control 1
112.            if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
113.            else begin isDo[2] <= 1'b1; D1 <= 8'h03; D2 <= 16'h6664; end
114.
115.        2: // Power control 2
116.            if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
117.            else begin isDo[2] <= 1'b1; D1 <= 8'h0C; D2 <= 16'h0000; end
118.
119.        3: // Power control 3
120.            if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
121.            else begin isDo[2] <= 1'b1; D1 <= 8'h0D; D2 <= 16'h080C; end
122.
123.        4: // Power control 4
124.            if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
125.            else begin isDo[2] <= 1'b1; D1 <= 8'h0E; D2 <= 16'h2B00; end
126.
127.        5: // Power control 5
128.            if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
129.            else begin isDo[2] <= 1'b1; D1 <= 8'h1E; D2 <= 16'h00B0; end
130.
131.        6: // Driver output control, MUX = 319, <R><G><B>
132.            if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
133.            else begin isDo[2] <= 1'b1; D1 <= 8'h01; D2 <= 16'h2B3F; end
134.
135.        7: // LCD driving waveform control
136.            if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
137.            else begin isDo[2] <= 1'b1; D1 <= 8'h02; D2 <= 16'h0600; end
138.
139.        8: // Sleep mode, weak-up
140.            if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
141.            else begin isDo[2] <= 1'b1; D1 <= 8'h10; D2 <= 16'h0000; end
142.
143.        9: // Entry mode, 65k color, DM = 2'b00, AM = 0, ID = 2'b11
144.            if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
145.            else begin isDo[2] <= 1'b1; D1 <= 8'h11; D2 <= 16'h6070; end
146.
147.        10: // Compare register

```

```

148.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
149.      else begin isDo[2] <= 1'b1; D1 <= 8'h05; D2 <= 16'h0000; end
150.
151.      11: // Compare register
152.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
153.      else begin isDo[2] <= 1'b1; D1 <= 8'h06; D2 <= 16'h0000; end
154.
155.      12: // Horizontal porch, HBP = 30, XL = 240
156.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
157.      else begin isDo[2] <= 1'b1; D1 <= 8'h16; D2 <= 16'hEF1C; end
158.
159.      13: // Vertical porch, VBP = 1, VBP = 4
160.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
161.      else begin isDo[2] <= 1'b1; D1 <= 8'h17; D2 <= 16'h0003; end
162.
163.      14: // Display control, 8 color mode, display on, operation on
164.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
165.      else begin isDo[2] <= 1'b1; D1 <= 8'h07; D2 <= 16'h0233; end
166.
167.      15: // Frame cycle control
168.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
169.      else begin isDo[2] <= 1'b1; D1 <= 8'h0B; D2 <= 16'h0000; end
170.
171.      16: // Gate scan position, SCN = 0
172.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
173.      else begin isDo[2] <= 1'b1; D1 <= 8'h0F; D2 <= 16'h0000; end
174.
175.      17: // Vertical scroll control
176.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
177.      else begin isDo[2] <= 1'b1; D1 <= 8'h41; D2 <= 16'h0000; end
178.
179.      18: // Vertical scroll control
180.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
181.      else begin isDo[2] <= 1'b1; D1 <= 8'h42; D2 <= 16'h0000; end
182.
183.      19: // 1st screen driving position, G0~
184.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
185.      else begin isDo[2] <= 1'b1; D1 <= 8'h48; D2 <= 16'h0000; end
186.
187.      20: // 1st screen driving position, ~G319
188.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
189.      else begin isDo[2] <= 1'b1; D1 <= 8'h49; D2 <= 16'h013F; end
190.

```

```

191.      21: // 2nd screen driving position
192.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
193.      else begin isDo[2] <= 1'b1; D1 <= 8'h4A; D2 <= 16'h0000; end
194.
195.      22: // 2nd screen driving position
196.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
197.      else begin isDo[2] <= 1'b1; D1 <= 8'h4B; D2 <= 16'h0000; end
198.
199.      23: // Horizontal RAM address position, HSA = 0 HEA = 239
200.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
201.      else begin isDo[2] <= 1'b1; D1 <= 8'h44; D2 <= 16'hEF00; end
202.
203.      24: // Vertical RAM address position, VSA = 0
204.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
205.      else begin isDo[2] <= 1'b1; D1 <= 8'h45; D2 <= 16'h0000; end
206.
207.      25: // Vertical RAM address position, VEA = 319
208.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
209.      else begin isDo[2] <= 1'b1; D1 <= 8'h46; D2 <= 16'h013F; end
210.
211.      26: // Gamma control, PKP0~1
212.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
213.      else begin isDo[2] <= 1'b1; D1 <= 8'h30; D2 <= 16'h0707; end
214.
215.      27: // Gamma control, PKP2~3
216.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
217.      else begin isDo[2] <= 1'b1; D1 <= 8'h31; D2 <= 16'h0204; end
218.
219.      28: // Gamma control, PKP4~5
220.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
221.      else begin isDo[2] <= 1'b1; D1 <= 8'h32; D2 <= 16'h0204; end
222.
223.      29: // Gamma control, PRP0~1
224.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
225.      else begin isDo[2] <= 1'b1; D1 <= 8'h33; D2 <= 16'h0502; end
226.
227.      30: // Gamma control, PKN0~1
228.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
229.      else begin isDo[2] <= 1'b1; D1 <= 8'h34; D2 <= 16'h0507; end
230.
231.      31: // Gamma control, PKN2~3
232.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
233.      else begin isDo[2] <= 1'b1; D1 <= 8'h35; D2 <= 16'h0204; end

```

```

234.
235.      32: // Gamma control, PKN4~5
236.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
237.      else begin isDo[2] <= 1'b1; D1 <= 8'h36; D2 <= 16'h0204; end
238.
239.      33: // Gamma control, PRN0~1
240.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
241.      else begin isDo[2] <= 1'b1; D1 <= 8'h37; D2 <= 16'h0502; end
242.
243.      34: // Gamma control, VRP0~1
244.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
245.      else begin isDo[2] <= 1'b1; D1 <= 8'h3A; D2 <= 16'h0302; end
246.
247.      35: // Gamma control, VRN0~1
248.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
249.      else begin isDo[2] <= 1'b1; D1 <= 8'h3B; D2 <= 16'h0302; end
250.
251.      36: // RAM write data mask
252.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
253.      else begin isDo[2] <= 1'b1; D1 <= 8'h23; D2 <= 16'h0000; end
254.
255.      37: // RAM write data mask
256.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
257.      else begin isDo[2] <= 1'b1; D1 <= 8'h24; D2 <= 16'h0000; end
258.
259.      38: // Unknown
260.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
261.      else begin isDo[2] <= 1'b1; D1 <= 8'h25; D2 <= 16'h8000; end
262.
263.      39: // RAM address set, horizontal(X)
264.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
265.      else begin isDo[2] <= 1'b1; D1 <= 8'h4E; D2 <= 16'h0000; end
266.
267.      40: // RAM address set, vertical(Y)
268.      if( iDone ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end
269.      else begin isDo[2] <= 1'b1; D1 <= 8'h4F; D2 <= 16'h0000; end
270.
271.      41:
272.      begin isDone <= 1'b1; i <= i + 1'b1; end
273.
274.      42:
275.      begin isDone <= 1'b0; i <= 6'd0; end
276.

```

```
277.          endcase  
278.
```

以上内容为初始化过程。该控制模块之所以变烦，原因都是这家伙在搞鬼。

```
279.      assign oStart = isDo;  
280.      assign oDone = isDone;  
281.      assign oAddr = D1;  
282.      assign oData = D2;  
283.  
284.endmodule
```

以上内容为相关的输出驱动声明。

tft_basemode.v

该组合模块的连线部署请参考图 27.20。

```
1. module tft_basemod  
2. (  
3.     input CLOCK,RESET,  
4.     output TFT_RST,  
5.     output TFT_RS,      // 1 = Data, 0 = Command(Register)  
6.     output TFT_CS_N,  
7.     output TFT_WR_N,  
8.     output TFT_RD_N,  
9.     output [15:0]TFT_DB,  
10.    input [2:0]iStart,  
11.    output oDone  
12. );  
13.    wire [2:0]StartU1;  
14.    wire [7:0]AddrU1;  
15.    wire [15:0]DataU1;  
16.  
17.    tft_ctrlmod U1  
18.    (  
19.        .CLOCK( CLOCK ),  
20.        .RESET( RESET ),  
21.        .iStart( iStart ),      // < top  
22.        .oDone( oDone ),      // > top  
23.        .oStart( StartU1 ),    // > U2  
24.        .iDone( DoneU2 ),      // < U2  
25.        .oAddr( AddrU1 ),      // > U2
```

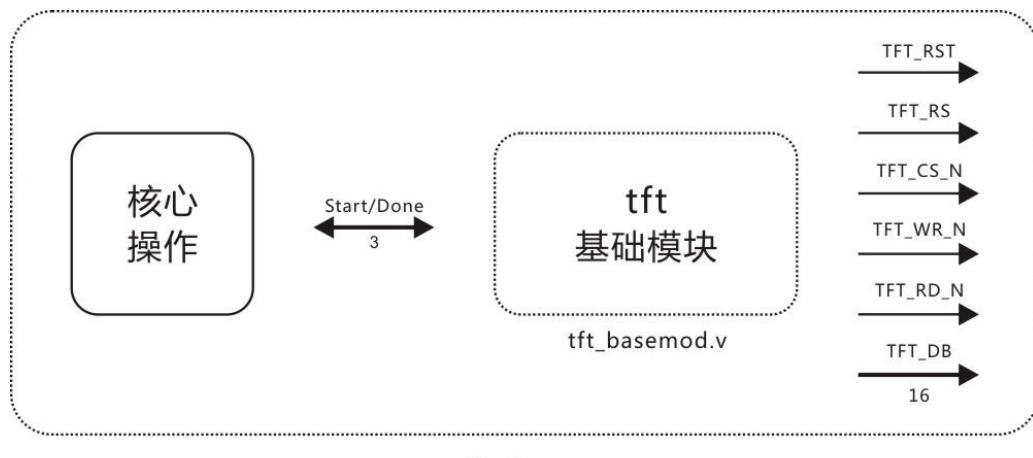
```

26.           .oData( DataU1 )      // > U2
27.       );
28.
29.       wire DoneU2;
30.
31.       tft_funcmod U2
32.       (
33.           .CLOCK( CLOCK ),
34.           .RESET( RESET ),
35.           .TFT_RS( TFT_RS ),      // > top
36.           .TFT_CS_N( TFT_CS_N ), // > top
37.           .TFT_WR_N( TFT_WR_N ), // > top
38.           .TFT_RD_N( TFT_RD_N ), // > top
39.           .TFT_DB( TFT_DB ),     // > top
40.           .iStart( StartU1 ),    // < U1
41.           .oDone( DoneU2 ),      // > U1
42.           .iAddr( AddrU1 ),      // < U1
43.           .iData( DataU1 )       // < U1
44.       );
45.
46.       assign TFT_RST = 1'b1;
47.
48.   endmodule

```

内容自己看着办吧，不过还是注意一下第 46 行的输出声明，其中 TFT_RST 赋值为 1。

tft_demo.v



tft_demo.v

图 27.23 实验二十七的建模图。

图 27.23 是实验二十七的建模图，其中 TFT 基础模块被核心操作调用，具体内容让我们来看代码吧。

```
1. module tft_demo
2. (
3.     input CLOCK, RESET,
4.     output TFT_RST,
5.     output TFT_RS,
6.     output TFT_CS_N,
7.     output TFT_WR_N,
8.     output TFT_RD_N,
9.     output [15:0]TFT_DB
10. );
```

以上内容为相关的出入端声明。

```
11.     wire DoneU1;
12.
13.     tft_basemod U1
14.     (
15.         .CLOCK( CLOCK ),
16.         .RESET( RESET ),
17.         .TFT_RST( TFT_RST ),
18.         .TFT_RS( TFT_RS ),
19.         .TFT_CS_N( TFT_CS_N ),
20.         .TFT_WR_N( TFT_WR_N ),
21.         .TFT_RD_N( TFT_RD_N ),
22.         .TFT_DB( TFT_DB ),
23.         .iStart( isDo ),
24.         .oDone( DoneU1 )
25.     );
26.
```

以上内容为 TFT 基础模块的实例化。

```
27.     reg [3:0]i;
28.     reg [2:0]isDo;
29.
30.     always @ ( posedge CLOCK or negedge RESET )
31.         if( !RESET )
32.             begin
33.                 i <= 4'd0;
34.                 isDo <= 3'd0;
```

```
35.           end  
36.       else
```

以上内容为相关的寄存器声明还有复位操作。

```
37.           case( i )  
38.  
39.               0: // Inital TFT  
40.               if( DoneU1 ) begin isDo[0] <= 1'b0; i <= i + 1'b1; end  
41.               else begin isDo[0] <= 1'b1; end  
42.  
43.               1: // Clear Screen  
44.               if( DoneU1 ) begin isDo[1] <= 1'b0; i <= i + 1'b1; end  
45.               else begin isDo[1] <= 1'b1; end  
46.  
47.               2: // Draw Function  
48.               if( DoneU1 ) begin isDo[2] <= 1'b0; i <= i + 1'b1; end  
49.               else begin isDo[2] <= 1'b1; end  
50.  
51.               3:  
52.               i <= i;  
53.  
54.           endcase  
55.  
56. endmodule
```

以上内容为核心操作。步骤 0 调用 isDo[0]执行初始化，步骤 1 调用 isDo[1]执行清屏，步骤 2 调用 isDo[2]执行绘图。



图 27.24 演示结果。

综合完毕便下载程序，如果 TFT LCD 户县由上至下的红色渐变，结果如图 27.24 所示，表示实验成功。事实上，图 27.24 是由 32 个，高为 10 宽为 240 的矩形组成，第 0 个矩

形接近黑色，第 31 个矩形则接近红色。如此一来，便产生红色渐变的效果。

细节一：完整的个体模块

事实上，本实验的 TFT 基础模块还不能称为完整的个体，因为实验二十七并没有明确的设计目的，所以 TFT 基础模块也没有具体封装要求，这种情况好比 VGA 基础模块。TFT 基础模块除了初始化功能还有清屏功能以外，绘图功能则是为了演示才故意加上去而已。往后如果有需要，读者再执行扩充吧。

细节二：提高速率

tft 功能模块曾经这样声明过，TCSL 为 50ns，量化结果为 3，还有 TCSH 为 500ns，量化结果则是 25。根据手册，写周期 TCYCLE 最小可以去到 100ns，亦即 TCSL 还有 TCSH 皆为 50ns。因此，速率也从原本的 1.818181 Mhz 飞升为 10Mhz。经过测试，笔者也没发现什么问题。不过，胆小的笔者认为，如果没有必要，TCSH 还是设为 500ns 比较保险，因为意外从总是突如其来。

实验二十八：TFT 模块 - 触屏

读者在上一个实验所玩弄过的 TFT LCD 模块，除了显示大小为 320×240 ，颜色为 16 位 RGB 的图像信息以外，它还支持触屏。所谓触屏就是鼠标还有键盘以外的输入手段，例如现在流行平板还有智能手机，触屏输入对我们来说，已经成为日常的一部分。描述语言一门偏向硬件的语言，面对触屏，它顶多只能做些驱动的工作，其余如滤波，还有像素转换等计算，它必须交由高级语言去负责。

面向无能为力的描述语言，笔者不禁联想过去的自己 … 好痛苦，好难受。话虽如此，天生我才必有用，描述语言虽然完成不了触屏的伟业，不过只要做些驱动，平稳过着日子，它便心满意足了。

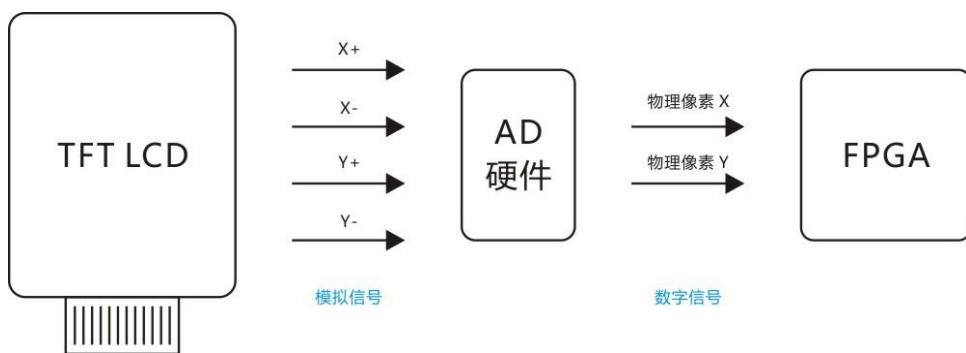


图 28.1 触屏的硬件部署。

如图 28.1 所示，那是触屏的硬件部署。根据理论，TFT 显示屏上面布满一层敏感的电磁层（触屏层）作为输入或者传感，而且传感的强弱会随着位置不同而不同。如果笔者不小心触碰任何一点，控制器 SSD 1289 会经由差分的 X+/- 信号还有 Y+/- 信号，将传感结果发送出去。在此，TFT 显示屏的工作已经结束。

差分是什么？估计失忆的同学会非常好奇。假设笔者不小心触碰位置 X20，然后像素 X20 所对应的模拟信号，例如 1.2v，其中正电压 +1.2v 会经由 X+ 信号发送出去，至于负电压 -1.2v 则会经由 X- 信号后发送出去。差分的目的通常是为了使电压更稳定更准确，读者只要这样理解即可。

事后，差分的传感结果便会传入 A/D 硬件的耳中，如果输入源是差分，A/D 硬件理应长出一对接收差分的耳朵，对此也称为差分 A/D 硬件。如果读者不知道 A/D 是什么，读者必须抓去枪毙了 … A/D 全名是 Analogue to Digital Convert，即模拟转为数字。这只命运般的 A/D 硬件就是鼎鼎大名的 XPT2046 硬件。

硬件 XPT2046 就是差分的 A/D 硬件，同时也是本实验的主角。如图 28.1 所示，它将 TFT 显示屏哪里发来的传感结果转换为数字信号，然后再经由 FPGA 读取。好奇的同学可能会继续好奇什么是物理像素？这个问题说来话长 … 假设屏幕是一个容器，藏在容器

里边的像素就称为屏幕像素，然而围绕容器外边的像素就称为物理像素。

比喻来讲，同是人形生物，住在地球的人形生物就称为地球人，住在地球以外的人形生物就称为外星人。结果而言，地球人并不等价外星人，虽然两者都有相似的外表。所以说，物理像素还有屏幕像素，它们听起来都是像素，不过实际上是不等价的东西。对此，物理像素为了成为屏幕像素，它们必须经过“转换算法”才能成为如假包换的屏幕像素。不过，“转换算法”对描述语言来说，这项工作实在太巨大了。

好了好了，笔者也差不多要进入主题了，废话再说下去，口水就会把电脑淹没了。

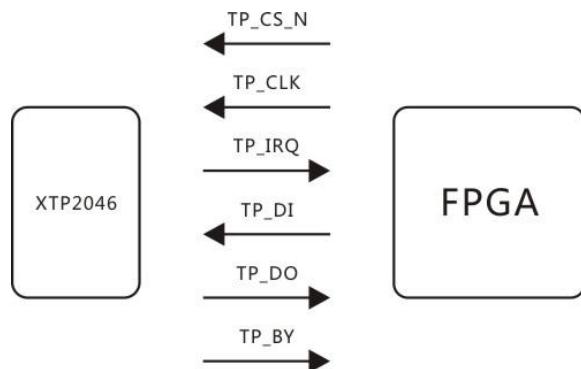


图 28.2 硬件 XTP2046 链接 FPGA。

图 28.2 显示硬件 XTP2046 链接 FPGA 所需要的连线，由于硬件 XTP2046 是 SPI 传输协议的信奉者，所以这些信号多少也有 SPI 的影子。CS 信号也是拉低有效的使能信号，CLK 信号是串行时钟，IRQ 是拉低有效的沟通信号，DI 与 DO 是读写作用的串行数据信号，BY 是 BUSY 的缩写，亦即表达状态的状态信号。如图 28.2 所示，因为 FPGA 控制 CLK 信号还有 CS 信号，所以 FPGA 是主机，硬件 XTP2046 则是从机。

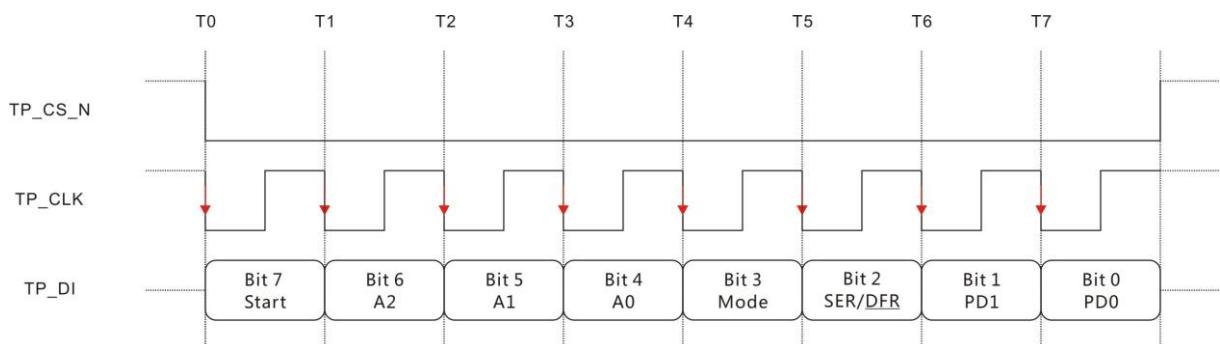


图 28.3 写时序/写命令(主机视角)。

首先让我们来瞧瞧 XTP2046 的写时序，如图 28.3 所示，写时序也称为写命令，因为主机访问从机只有写命令而没有写数据。我们知道 SPI 都是下降沿设置数据，上升沿锁存数据 … 对此，写一个字节命令需要用到 8 下降沿，期间 CS 必须一直拉低。如图 28.3 所示，一字节命令当中却有各个不同的位作用，结果如表 28.1 所示：

表 28.1 命令位说明。

位宽	名称	说明
Bit 7	Call	起始位，值 1 为起始。
Bit 6~4	A2~A0	通道地址。X 通道 = 3' b001, Y 通道 3' b101。
Bit 3	Mode	12 位/8 位分辨率选择。值 0 为 12 位分辨率，反之亦然。
Bit 2	SER/DFR	输入源选择。值 0 为差分输入，值 1 为单端输入。
Bit 1~0	PD1~PD0	功率选择。2' b00 为低功率，2' b11 为平常。

如表 28.1 所示，那是命令位的说明。Call 为起始，值 1 表示开始命令，A2~A0 为通道地址，常用有 X 通道与 Y 通道。Mode 为分辨率选择，即转换结果是 8 位还是 12 位，常见为 12 位分辨率。SER/DFR 为输入源选择，虽然 XTP2046 是差分 A/D，不过它可以选择输入源。PD1~PD0 为功率选择，默认为低功率。

表 28.1 虽然令人眼花缭乱，不过常见的命令只有以下两个：

读取 X 通道转化结果，12 位分辨率，差分输入，低功率，8'b1_001_0000 或者 8'h90。
读取 Y 通道转换结果，12 位分辨率，差分输入，低功率，8'b1_101_0000 或者 8'hD0。

此外，Verilog 可以这样描述图 28.3，结果如代码 28.1 所示：

```

1.      0,1,2,3,4,5,6,7:
2.      begin
3.          rDI <= T1[7-i];
4.
5.          if( C1 == 0 ) rCLK <= 1'b0;
6.          else if( C1 == FHALF ) rCLK <= 1'b1;
7.
8.          if( C1 == FCLK -1) begin C1 <= 8'd0; i <= i + 1'b1; end
9.          else begin C1 <= C1 + 1'b1; end
10.     end

```

代码 28.1

如代码 28.1 所示，第 1 行表示相同的操作有 8 次，第 8~9 行表示一个步骤所停留的时间，其中的 FCLK 表示一个时钟周期。第 5~6 行表示，C1 为 0 拉低 CLK，C1 为半个周期（FHALF）则拉高时钟。第 3 行则是由高至低逐位赋值。

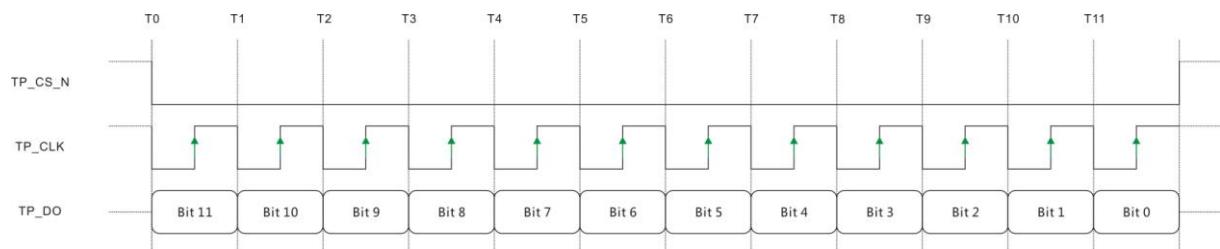


图 28.4 读时序/读数据(主机视角)。

如图 28.4 所示，那是 XTP2046 的读时序或者说为读数据。我们知道 SPI 利用时钟信号的上升沿锁存数据，如果数据的分辨率设为 12 位（Mode 为 0 值），那么一次性的读数据需要 12 个上升沿，期间 CS 必须拉低。根据手册，虽然一次性的读数据有 12 位，不过仅有当中的高八位有利用价值，亦即 Bit 11~Bit 4，结果低四位被无视。对此，Verilog 可以这样表示，结果如代码 28.2 所示：

```

1. 0,1,2,3,4,5,6,7,8,9,10,11:
2. begin
3.   if( C1 == FHALF ) T2[11-i] <= TP_DO;
4.
5.   if( C1 == 0 ) rCLK <= 1'b0;
6.   else if( C1 == FHALF ) rCLK <= 1'b1;
7.
8.   if( C1 == FCLK -1) begin C1 <= 8'd0; i <= i + 1'b1; end
9.   else begin C1 <= C1 + 1'b1; end
10. end

```

代码 28.2

如代码 28.2 所示，第 1 行表示该操作重复 12 次，第 8~9 行表示该步骤停留 FCLK 周期。第 5~6 行表示，C1 为 0 拉低 CLK，C1 为 FHALF 则拉高 CLK。第 3 行表示，C1 为 HALF 就从 TP_DO 哪里暂存结果。

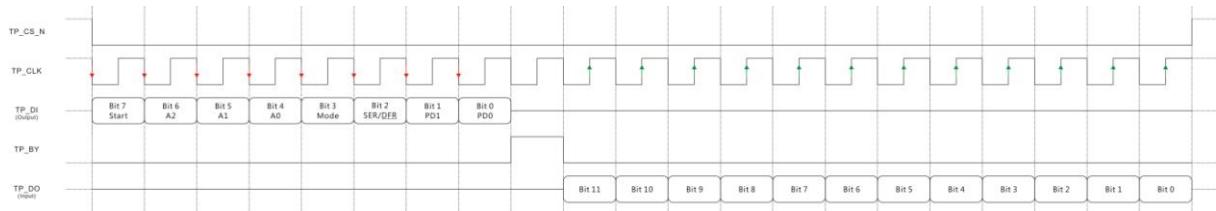


图 28.5 一次性写命令与读数据的时序图（主机视角）。

图 28.5 是一次性写命令与读数据的时序图。操作期间，CS 必须持续拉低，首先主机利用下降沿向从机发送一字节的写命令，事后从机会拉高 BY 一个时钟以示忙状态，期间主机必须给足一个空时钟。从机忙完以后便会拉低 BY 信号，紧接着便会发送 12 位宽的转换结果，期间主机必须利用上升沿读取数据。对此，Verilog 可以这样描述，结果如代码 28.3 所示：

```

1. 1: // Write byte
2. begin rCS <= 1'b0; T1 <= Command; i <= FF_Write; Go <= i + 1'b1; end
3.
4. 2: // Wait Busy
5. begin
6.   if( C1 == 0 ) rCLK <= 1'b0;
7.   else if( C1 == FHALF ) rCLK <= 1'b1;

```

```

8.
9.      if( C1 == FCLK -1) begin C1 <= 8'd0; i <= i + 1'b1; end
10.     else begin C1 <= C1 + 1'b1; end
11.   end
12.
13.   3: // Read 12 bit
14.   begin i <= FF_Read; Go <= i + 1'b1; end
15.
16.   4:
17.   begin D1 <= T2[11:4]; rCS <= 1'b1; i <= i + 1'b1; end

```

代码 28.3

如代码 28.3 所示，步骤 0 拉低 CS 之余，它也赋值写命令，然后 i 指向写命令的伪函数（代码 28.1）。写命令完成以后，主机必须给足一个时钟，对此步骤 1 发呆一个时钟周期。步骤 2 将 i 指向读数据的伪函数（代码 28.2），随后在步骤 4 保存高 8 位的结果，最后也顺便拉高一下 CS 信号。



图 28.6 IRQ 信号。

初期阶段（放手阶段）IRQ 呈现拉高状态，如果我们不小心触碰屏幕，XTP2046 便会察觉输入源发生变化，然后便会拉低 IRQ 信号以示触屏事件发生了，结果如图 28.6 所示。对此，Verilog 可以这样描述，结果如代码 28.4 所示：

```

1. 0:
2. if( !TP_IRQ ) begin ...; i <= i + 1' b1; end
3.
4. 1:
5. .....

```

代码 28.4

准备知识理解完毕以后，我们便可以开始建模了。

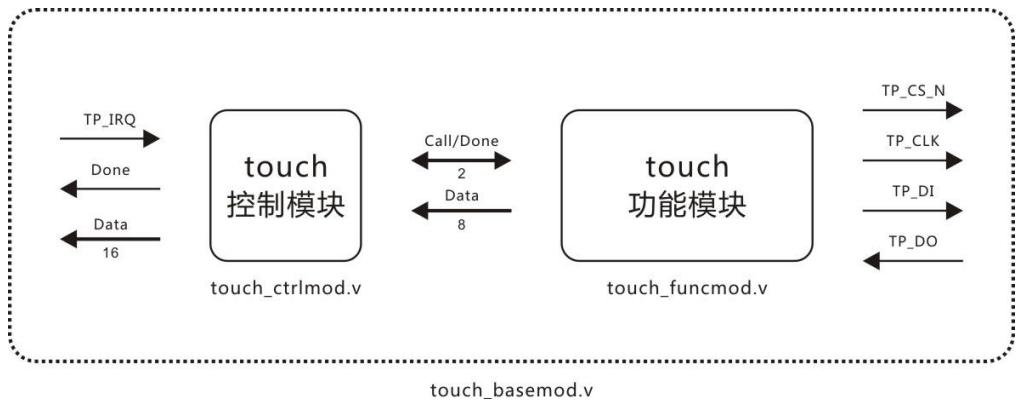


图 28.7 Touch 基础模块的建模图。

图 28.7 是 Touch 基础模块，内容包含控制模块还有功能模块。功能模块左边有两位宽的 Call/Done，其中[1]为读取物理像素 X，[0]为读取物理像素 Y。至于右边则是一些顶层信号。控制模块的左边，除了顶层信号 IRQ 是输入以外，其余信号都呈现输出状态。16 位宽的 Data 对应物理像素 X 还有物理像素 Y，而 Done 则是沟通作用的触发信号。整体来说，一旦触碰触屏，IRQ 拉低，控制模块利用功能模块读取物理像素 X 与 Y，然后再产生完成信号以示一次性的触屏结果。

touch_funcmod.v



图 28.8 Touch 功能模块的建模图。

图 28.8 是 Touch 功能模块的建模图，具体内容我们还是来看代码吧。

```

1. module touch_funcmod
2. (
3.     input CLOCK,RESET,
4.     output TP_CS_N,
5.     output TP_CLK,
6.     output TP_DI,
7.     input TP_DO,
8.
9.     input [1:0]iCall,
10.    output oDone,
11.    output [7:0]oData
12. );

```

以上内容为相关的出入端声明。

```
13.     parameter FCLK = 8'd20, FHALF = 8'd10; // 2.5Mhz
14.     parameter FF_Write = 6'd16, FF_Read = 6'd32;
15.
```

以上内容为常量声明，内容包括 2.5Mhz 周期，半周期，写命令还有读数据入口地址。

```
16.     reg [5:0]i,Go;
17.     reg [11:0]D1;
18.     reg [7:0]C1;
19.     reg rCS,rCLK,rDI;
20.     reg isDone;
21.
22.     always @ ( posedge CLOCK or negedge RESET )
23.         if( !RESET )
24.             begin
25.                 { i,Go } <= { 6'd0, 6'd0 };
26.                 D1 <= 12'd0;
27.                 C1 <= 8'd0;
28.                 { rCS,rCLK,rDI } <= 3'b111;
29.                 isDone <= 1'b0;
30.             end
```

以上内容为相关的寄存器声明还有复位操作。

```
31.     else if( iCall )
32.         case( i )
33.
34.             0: // [1] Read 12bit X, [0],Read 12 bit Y
35.                 if( iCall[1] ) begin D1[7:0] <= 8'h90; i <= i + 1'b1; end
36.                 else if( iCall[0] ) begin D1[7:0] <= 8'hd0; i <= i + 1'b1; end
37.
38.             1: // Write byte
39.                 begin rCS <= 1'b0; i <= FF_Write; Go <= i + 1'b1; end
40.
41.             2: // Wait Busy
42.                 begin
43.                     if( C1 == 0 ) rCLK <= 1'b0;
44.                     else if( C1 == FHALF ) rCLK <= 1'b1;
45.
46.                     if( C1 == FCLK -1) begin C1 <= 8'd0; i <= i + 1'b1; end
```

```

47.           else begin C1 <= C1 + 1'b1; end
48.       end
49.
50.           3: // Read 12 bit
51.       begin i <= FF_Read; Go <= i + 1'b1; end
52.
53.           4:
54.       begin rCS <= 1'b1; i <= i + 1'b1; end
55.
56.           5:
57.       begin isDone <= 1'b1; i <= i + 1'b1; end
58.
59.           6:
60.       begin isDone <= 1'b0; i <= 6'd0; end
61.

```

以上内容为该功能模块的核心操作。步骤 0 会根据 iCall 为 D1 赋值 8'h90(读 X), 还是 8'hD0 (读 Y)。步骤 1 拉低 CS 之余也写命令 , 步骤 2 则是给足一个空时间。步骤 3 为读数据 , 步骤 4 拉低 CS , 步骤 5~6 则产生完成信号。

```

62.           *****/
63.
64.           16,17,18,19,20,21,22,23:
65.       begin
66.           rDI <= D1[23-i];
67.
68.           if( C1 == 0 ) rCLK <= 1'b0;
69.           else if( C1 == FHALF ) rCLK <= 1'b1;
70.
71.           if( C1 == FCLK -1) begin C1 <= 8'd0; i <= i + 1'b1; end
72.           else begin C1 <= C1 + 1'b1; end
73.       end
74.
75.           24:
76.           i <= Go;
77.

```

步骤 16~23 为写一个字节。

```

78.           *****/
79.
80.           32,33,34,35,36,37,38,39,40,41,42,43:
81.       begin

```

```

82.           if( C1 == FHALF ) D1[43-i] <= TP_DO;
83.
84.           if( C1 == 0 ) rCLK <= 1'b0;
85.           else if( C1 == FHALF ) rCLK <= 1'b1;
86.
87.           if( C1 == FCLK -1) begin C1 <= 8'd0; i <= i + 1'b1; end
88.           else begin C1 <= C1 + 1'b1; end
89.       end
90.
91.       44:
92.       i <= Go;
93.
94.   endcase
95.

```

步骤 32~92 为读 12 位数据。

```

96.           assign TP_CS_N = rCS;
97.           assign TP_CLK = rCLK;
98.           assign TP_DI = rDI;
99.           assign oDone = isDone;
100.          assign oData = D1[11:4];
101.
102. endmodule

```

以上内容为输出驱动声明。注意第 100 行，oData 赋值 D1 的高八位。

touch_ctrlmod.v



图 28.9 Touch 控制模块的建模图。

图 28.9 是 Touch 控制模块的建模图，基本上也没有什么好说的，具体内容让我们来看代码吧。

```

1. module touch_ctrlmod
2. (
3.     input CLOCK,RESET,

```

```
4.      input TP_IRQ,
5.      output oDone,
6.      output [15:0]oData,
7.
8.      output [1:0]oCall,
9.      input iDone,
10.     input [7:0]iData
11. );
```

以上内容为相关的出入端声明。

```
12.     reg [5:0]i;
13.     reg [7:0]D1,D2;
14.     reg [1:0]isCall;
15.     reg isDone;
16.
17.     always @ ( posedge CLOCK or negedge RESET )
18.         if( !RESET )
19.             begin
20.                 i <= 6'd0;
21.                 { D1,D2 } <= { 8'd0,8'd0 };
22.                 isCall <= 2'd0;
23.                 isDone <= 1'b0;
24.             end
```

以上内容为相关的寄存器声明还有复位操作。

```
25.     else
26.         case( i )
27.
28.             0:
29.                 if( !TP_IRQ ) i <= i + 1'b1;
30.
31.             1:// Read X
32.                 if( iDone ) begin isCall[1] <= 1'b0; D1 <= iData; i <= i + 1'b1; end
33.                 else begin  isCall[1] <= 1'b1; end
34.
35.             2:// Read Y
36.                 if( iDone ) begin isCall[0] <= 1'b0; D2 <= iData; i <= i + 1'b1; end
37.                 else begin  isCall[0] <= 1'b1; end
38.
39.             3:
40.                 begin isDone <= 1'b1; i <= i + 1'b1; end
```

```
41.  
42.          4:  
43.          begin isDone <= 1'b0; i <= 6'd0; end  
44.  
45.      endcase  
46.
```

以上内容为该控制模块的核心内容。步骤 0 用来检测 IRQ 信号拉低，步骤 1 读取物理像素 X，步骤 2 读取物理像素 Y。步骤 3~4 则是用来产生完成信号。

```
47.      assign oDone = isDone;  
48.      assign oData = {D1,D2};  
49.      assign oCall = isCall;  
50.  
51. endmodule
```

以上内容为相关的输出驱动声明。

touch_basemod.v

有关这个模块的连线部署请参考图 28.7。

```
1. module touch_basemod  
2. (  
3.     input CLOCK,RESET,  
4.     output TP_CS_N,  
5.     output TP_CLK,  
6.     input TP_IRQ,  
7.     output TP_DI,  
8.     input TP_DO,  
9.  
10.    output oDone,  
11.    output [15:0]oData  
12. );  
13.    wire [1:0]CallU1;  
14.  
15.    touch_ctrlmod U1  
16.    (  
17.        .CLOCK( CLOCK ),  
18.        .RESET( RESET ),  
19.        .TP_IRQ( TP_IRQ ), // < top  
20.        .oDone( oDone ), // > top  
21.        .oData( oData ), // > top
```

```

22.           .oCall( CallU1 ),      // > U2
23.           .iDone( DoneU2 ),    // < U1
24.           .iData( DataU2 )     // < U1
25.       );
26.
27.       wire DoneU2;
28.       wire [7:0]DataU2;
29.
30.       touch_funcmod U2
31.       (
32.           .CLOCK( CLOCK ),
33.           .RESET( RESET ),
34.           .TP_CS_N( TP_CS_N ),   // > top
35.           .TP_CLK( TP_CLK ),    // > top
36.           .TP_DI( TP_DI ),      // > top
37.           .TP_DO( TP_DO ),      // < top
38.           .iCall( CallU1 ),      // < U1
39.           .oDone( DoneU2 ),     // > U1
40.           .oData( DataU2 )      // > U1
41.       );
42.
43.   endmodule

```

以上内容为 Touch 基础模块的连线部署，读者自己看着办吧。

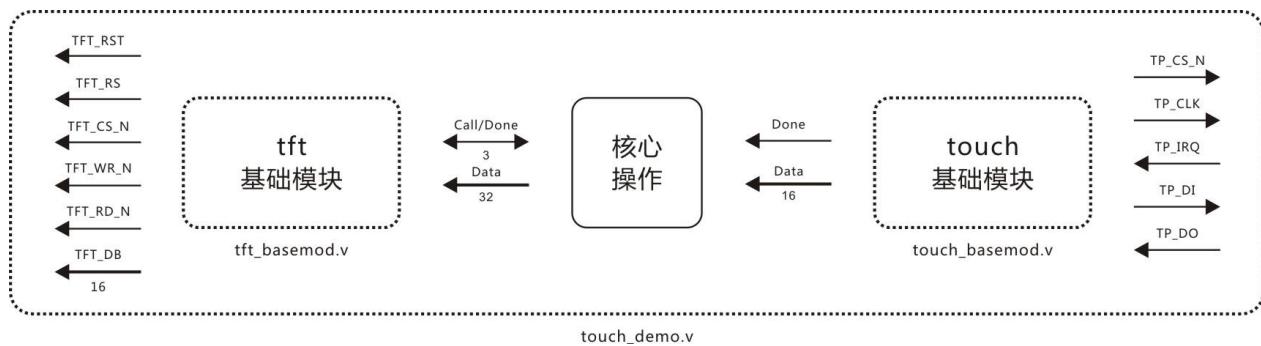


图 28.10 实验二十八的建模图。

图 28.10 是实验二十八的建模图，首先核心操作会初始化还有清屏 TFT 基础模块，然后触屏信息会经由 Touch 基础模块传至核心操作，随后核心操作再将触屏信息作为屏幕像素写入 TFT 屏当中。此外，笔者也改动基础模块的绘图功能，它会将 32 位宽的像素 X，像素 Y，以及颜色信息写入其中。废话少说，让我们看看该绘图功能究竟发生什么改变呢？

tft_ctrlmod.v

```
6.      input [7:0]iData,
```

首先是 tft 控制模块多了两个出入端的声明。

```
29. else if( iCall[2] )
30.         case( i )
31.
32.             0:// X0
33.             if( iDone ) begin isCall[2] <= 1'b0; i <= i + 1'b1; end
34.             else begin isCall[2] <= 1'b1; D1 <= 8'h4E; D2 <= { 8'd0, iData[31:24] }; end
35.
36.             1:// Y0
37.             if( iDone ) begin isCall[2] <= 1'b0; i <= i + 1'b1; end
38.             else begin isCall[2] <= 1'b1; D1 <= 8'h4F; D2 <= { 8'd0, iData[23:16] }; end
39.
40.             2:// Write data to ram 0x22
41.             if( iDone ) begin isCall[1] <= 1'b0; i <= i + 1'b1; end
42.             else begin isCall[1] <= 1'b1; D1 <= 8'h22; end
43.
44.             *****/
45.
46.             3:// Write color
47.             if( iDone ) begin isCall[0] <= 1'b0; i <= i + 1'b1; end
48.             else begin isCall[0] <= 1'b1; D2 <= [15:0]; end
49.
50.             *****/
51.
52.             4:
53.             begin isDone <= 1'b1; i <= i + 1'b1; end
54.
55.             5:
56.             begin isDone <= 1'b0; i <= 6'd0; end
57.
58.         endcase
```

紧接着是受更动的绘图功能。步骤 0 根据 Data[31:24] 设置 X 像素，步骤 1 根据 Data[23:16]设置 Y 像素，然后步骤 2 锁定数据该写入的地址，最后步骤 3 再将 Data[15:0] 的图像信息写进去。步骤 4~5 则是用来产生完成信号。读者是不是觉得很简单呢？

touch_demo.v

```
1. module touch_demo
2. (
3.     input CLOCK, RESET,
4.     output TFT_RST,
5.     output TFT_RS,
6.     output TFT_CS_N,
7.     output TFT_WR_N,
8.     output TFT_RD_N,
9.     output [15:0]TFT_DB,
10.
11.    output TP_CS_N,
12.    output TP_CLK,
13.    input TP_IRQ,
14.    output TP_DI,
15.    input TP_DO
16. );
```

以上内容为相关的出入端声明。

```
17.     wire DoneU1;
18.     wire [15:0]DataU1;
19.
20.     touch_basemod U1
21.     (
22.         .CLOCK( CLOCK ),
23.         .RESET( RESET ),
24.         .TP_CS_N( TP_CS_N ),
25.         .TP_CLK( TP_CLK ),
26.         .TP_IRQ( TP_IRQ ),
27.         .TP_DI( TP_DI ),
28.         .TP_DO( TP_DO ),
29.         .oDone( DoneU1 ),
30.         .oData( DataU1 ),
31.     );
32.
```

以上内容为 Touch 基础模块的实例化。

```
33.     wire DoneU2;
34.
35.     tft_basemod U2
```

```

36.      (
37.          .CLOCK( CLOCK ),
38.          .RESET( RESET ),
39.          .TFT_RST( TFT_RST ),
40.          .TFT_RS( TFT_RS ),
41.          .TFT_CS_N( TFT_CS_N ),
42.          .TFT_WR_N( TFT_WR_N ),
43.          .TFT_RD_N( TFT_RD_N ),
44.          .TFT_DB( TFT_DB ),
45.          .iCall( isCall ),
46.          .oDone( DoneU2 ),
47.          .iData( {D1,D2,D3} ),
48.      );
49.

```

以上内容为 TFT 基础模块的实例化。

```

50.      reg [5:0]i,Go;
51.      reg [2:0]isCall;
52.      reg [7:0]D1;
53.      reg [8:0]D2;
54.      reg [15:0]D3;
55.
56.      always @ ( posedge CLOCK or negedge RESET )
57.          if( !RESET )
58.              begin
59.                  { i,Go } <= { 6'd0,6'd0 };
60.                  isCall <= 3'd0;
61.                  { D1,D2,D3 } <= { 8'd0,9d0,16'd0 };
62.              end
63.          else

```

以上内容为相关的寄存器声明还有复位操作。

```

73.
74.          2:
75.          if( DoneU1 ) begin i <= i + 1'b1; end
76.
77.          3:
78.          if( DoneU2 ) begin isCall[2] <= 1'b0; i <= i + 1'b1;   end
79.          else begin isCall[2] <= 1'b1; D1 <= DataU1[15:8]; D2 <= { 1'b0, DataU1[7:0] }; D3 <= 16'd0; end
80.
81.          4:
82.          i <= 2;
83.
84.      endcase
85.
86.  endmodule

```

以上内容为核心操作。步骤 0 初始化 TFT 显示屏，步骤 1 则是清屏。步骤 2 等待 Touch 基础模块反馈完成，然后继续步骤。步骤 3 将反馈过来的 X 与 Y 以及黑色写入其中。完后，步骤步骤 2~3 不停来回重复。

综合完毕便下载程序，然后我们便可以用手指在屏幕上涂鸦了，虽然方向还有涂鸦大小有点暴走，那是因为物理像素对屏幕像素是不同性质的东西，结果还是视为实验成功。

细节一：完整的个体模块

本实验的 Touch 基础模块充其量还是半身熟的鸡蛋，因为当中缺省重要的环节，然而这个环节却是描述语言难以负担的重任。



图 28.11 物理像素转换为屏幕像素的概念图。

如图 28.11 所示，原始的物理像素一般都包含噪声，对此物理像素必须预先经过一层滤波。滤波以后的物理像素才会开始进入转换阶段，转换阶段必须借用转换算法的力量，论道算法，想必有些同学会不经意缩紧眉心，因为我们必须向数学打交道不可。那怕对象是简单乘法或者除法，读过《时序篇》的朋友一定会晓得算法都很麻烦。

物理像素成功转换为屏幕像素以后，它还不能立即纳入使用，因为转换算法有可能存在细节上的缺失。对此，屏幕像素必须经过校正 ... 校正期间不仅有出现算法，我们还要和屏幕发生互动，实在烦死人了。现阶段而言，如果描述语言如果不及顺序语言那么便捷，什么物理像素转换屏幕像素，什么校正 ... 我们最好想也不要，不然结果只有自

讨苦吃。

虽然笔者有可能被指责为胆小鬼，不负责任之类的渣渣，对此笔者不否认，理性而言，那些危及生命的事情，笔者另可背负屈辱也要逃跑，因为没有什么东西比生命更可贵。此刻，只要承认懦弱，我们才会安全成长，未来的事情就让未来的自己去打算吧！少年少女们！

细节二：时序参数

老实说，笔者也觉得懦弱的自己太没出息了 … 不过，作为补偿，让我们来瞧瞧硬件 XTP2046 的时序参数吧。

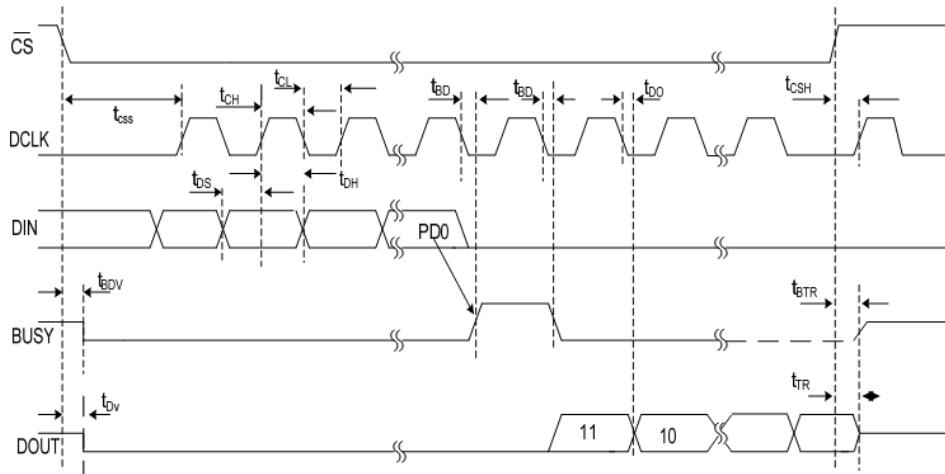


图 28.11 XTP2046 的物理时序图。

图 28.11 是从官方手册哪里拷贝过来的物理时序图，这张图虽然有可能是吓跑小朋友的虎姑婆 … 不过，我们只要习惯以后，它也可能是弱小的小虎猫。换之，表 28.2 是各个时序参数的详细信息。

表 28.2 时序参数的详细信息。

参数	说明	最大	最小	单位
t _{DS}	DIN 在 DCLK 上升沿前生效	100		ns
t _{DH}	DIN 保持在 DCLK 高电平后	50		ns
t _{DO}	DCLK 下降沿到 DOUT 生效		200	ns
t _{DV}	CS 下降沿到 DOUT 使能		200	ns
t _{TR}	CS 上升沿到 DOUT 禁止		200	ns
t _{CS}	CS 下降沿到第一个 DCLK 上升沿	100		ns
t _{CSH}	CS 上升沿到 DCLK 被忽略	10		ns
t _{CH}	DCLK 高电平	200		ns
t _{CL}	DCLK 低电平	200		ns
t _{BD}	DCLK 下降沿到 BUSY 上升/下降		200	ns

tBDV	CS 下降沿到 BUSY 使能		200	ns
tBTR	CS 上升沿到 BUSY 禁止		200	ns

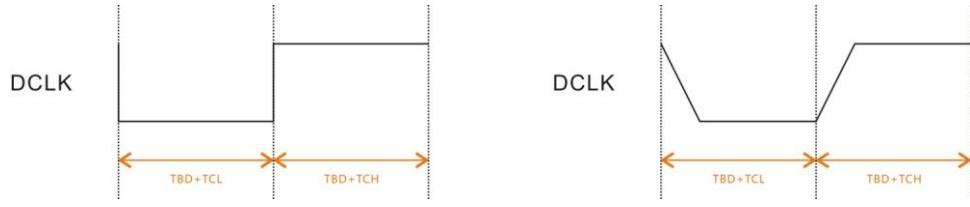


图 28.11 DCLK 相关的时序参数。

首先是时序第一员的 TCH 还有 TCL，如图 28.11 所示，TBL+TCL+ TBL+TCH 造就一个时钟周期。根据表 28.2 所示，由于 TBL 最小为 0ns，所以可以无视（TBL 可以视为上山信号还有下上信号）。对此，造就一个时钟周期的成分只有 TCH+TCL，而且两者最小皆是 200ns，所以最高速率是：

$$1/(200\text{ns} + 200\text{ns}) = 2.5\text{Mhz}$$

话虽如此，这仅仅是手册给出的安全速率，如果读者是一名头文字 D 的死粉，读者随时都可以驾驶藤原豆腐车超频。

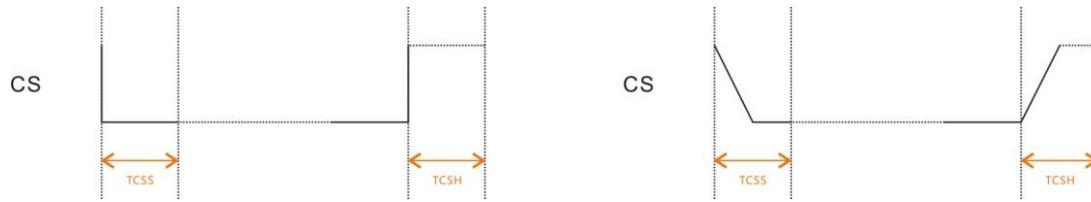


图 28.12 CS 相关的时序参数。

接下来让我们看看 TCSS 还有 TCSH，前者可以视为 CS 拉低所需的最长时间，后者则是 CS 拉高所需的最长时间。前者需要 100ns，后者则是 10ns，如果无法满足它们，CS 的有效性就会存在风险。话虽如此，因为手册比较怕死，所以参数略显保险（夸张），我们只要随便应付就好。

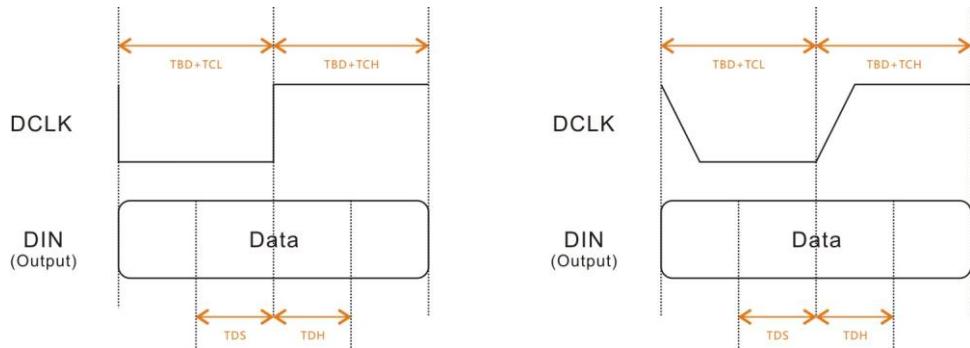


图 28.13 数据相关的时序参数（主机视角）。

再者就是 TDS 还有 TDH，前者是典型的 setup 要求，后者则是 hold 要求，只要两者得

到满足，数据打入寄存器才能得到确保。首先，读者必须注意一下图 28.13 是主机视角的写时序（从机读数据），所以 Data 是主机发给从机的食物。饿昏的从机会借用上升沿锁存时序，此刻只要 $TBD+TCL$ 大于 TDS ，又或者 $TBD+TCH$ 大于 TDH ，数据就会成功被锁存。

根据表 28.2 所示， TDS 是 100ns， TDH 则是 50ns，换之 $TBD+TCL$ 是 200ns， $TBD+TCH$ 则是 200ns。简单来说， TDS 还有 TDH 都无法完全覆盖数据，结果数据的有效性是得不到人头担保的。

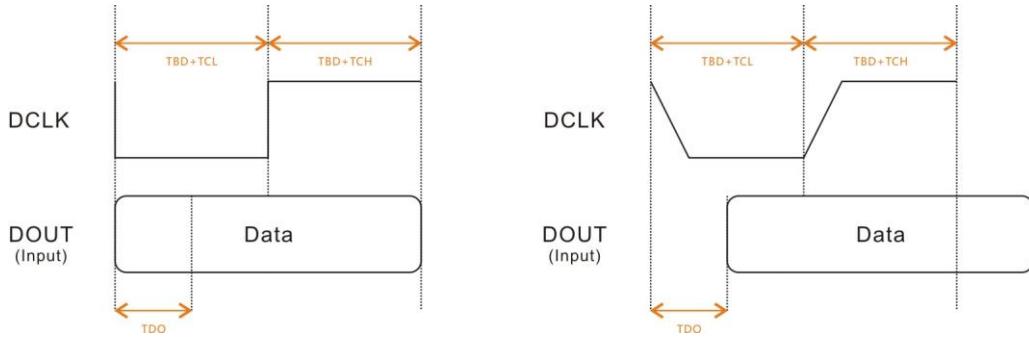


图 28.14 数据相关的时序参数（主机视角）。

图 28.14 还是数据相关的时序参数，不过方向是从机给主机发送数据，当从机借由下降沿设置数据的时候，必须被 TDO 拖后腿若干时间。根据理想时序（左图），TDO 只是单纯覆盖 Data 而已 … 反之，右图的物理时序则会推挤 Data。根据表 28.2 所示，TDO 最大有可能拖后腿半个周期，反过来就是不拖任何后腿。活着就要乐观一点，凡事都往好的方向去想，所以我们可以无视 TDO。

最后还有一些仅是与 CS 信号扯上关系的小啰嗦，如 TDV，TTR 等参数。我们仅要读写数据之前死拉高 CS 不放，读写数据之间死拉低 CS 不放，然后读写数据之后又死拉高 CS 不放，我们就能成功打发它们。

实验二十九：LCD 模块

据说 Alinx 301 支持 7" TFT，好奇的朋友一定疑惑道，它们 3.2" TFT 以及 7" TFT 等两者之间究竟有何区别呢？答案很简单，前者自带控制器也有图像内存。换之，后者好似缩小版台式的液晶，它除了接口以外什么也没有。



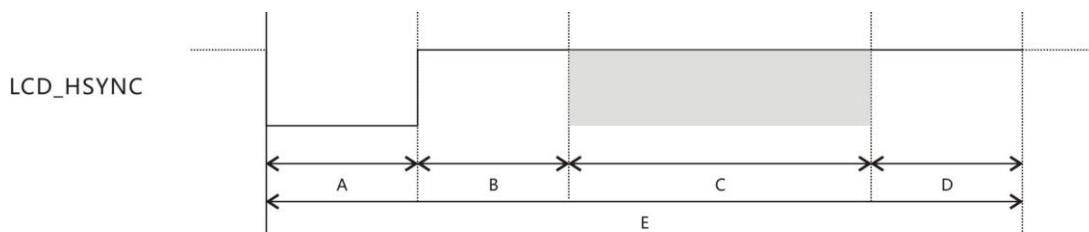
图 29.1 7" TFT 的引脚。

如图 29.1 所示，这只 7" TFT 拥有 840×480 的分辨率，应用 VGA 接口，所以左边才有熟悉的 HSYNC 以及 VSYNC 信号。LCD_CLOCK 是像素时钟，最大为 50Mhz，并且没有下限。LCD_RED/GREEN/BLUE 为 18 位 RGB，颜色支持范围是 $2^{18} = 262K$ 。右边的 DE 为 Data Enable 拉高表示数据输入有效，LR/UD 为扫描次序，例如自左向右，由高至下就是 2'b10，结果如表 29.1 所示：

表 29.1 TFT 的扫描次序。

LR	UD	扫描次序
0	0	自右向左，由上至下
0	1	自右向左，由下至上
1	0	自左向右，由上至下
1	1	自左向右，由下至上

根据手册，它支持两种模式，MODE 拉低表示传统的 VGA 模式，MODE 拉高则是 DE 模式。老实说，什么是 DE 模式，笔者真有点搞不懂，手册也没有详细注明，所以 MODE 信号必须拉低。这只 7" TFT 自带背光，我们可以经由 PWM 信号调节背光的亮度，具体内容请浏览手册，我们一般都是常年拉高。



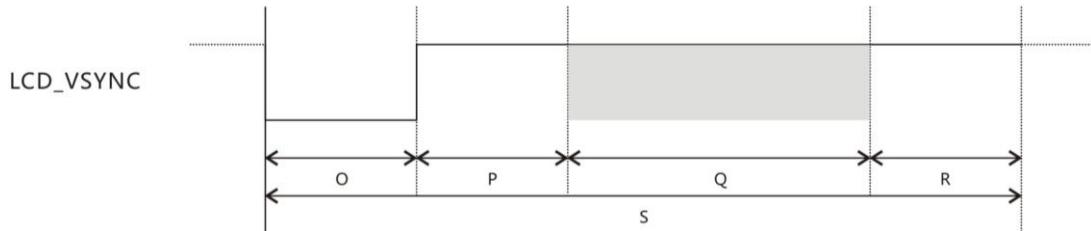


图 26.4 VGA 时序。

如图 29.2 所示，HSYNC 以及 VSYNC 均为五段，具体长度如表 29.2 所示：

表 29.2 显示标准 800 × 480。

信号	A	B	C	D	E
VGA_HSYNC	48	40	800	40	928
信号	O	P	Q	R	S
VGA_VSYNC	3	29	480	13	525

笔者曾前面说过，折尺 7” TFT 应用 VGA 接口，驱动方法与实验二十六差不多。所以说，懒惰的笔者就直接沿用实验二十六的资源。



图 29.3 128×96 大大小小可爱。

图 29.3 是我们要显示的小可爱 ... 啊，不管怎么看，比卡丘最可爱了！完后，我们让我们建模去吧。

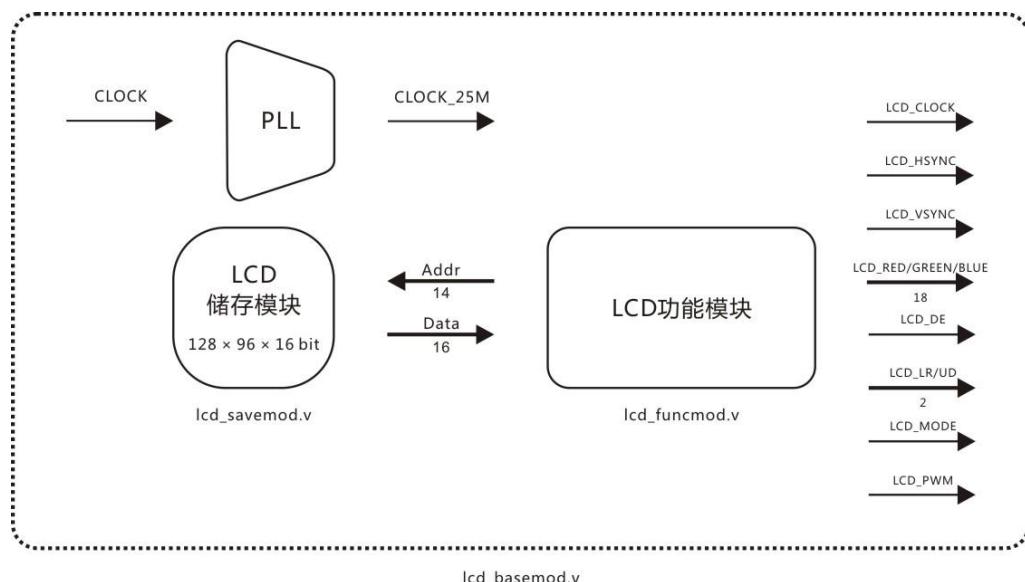


图 29.4 LCD 基础模块的建模图。

图 29.4 是 LCD 基础模块的建模图，内容包括储存模块以及的功能模块。相对 PLL 模块将时间分频为 25Mhz，因为像素时钟是任意的。

lcd_funcmod.v

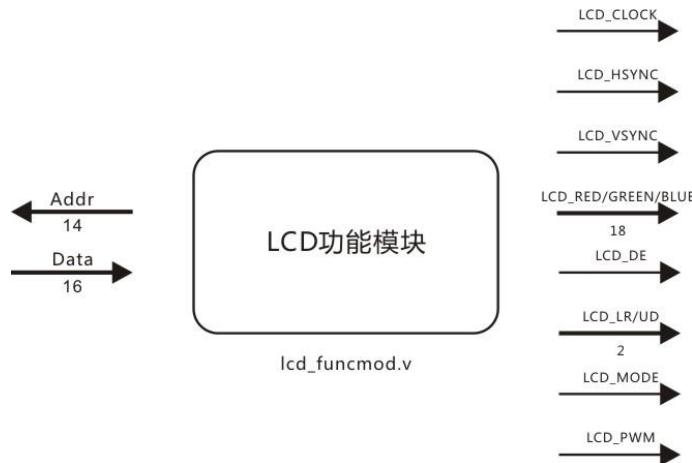


图 29.5 LCD 功能模块。

图 29.5 是 LCD 功能模块的建模图，读者可能很好奇控制模块去哪儿了？非呀，控制模块已经被整合进去了。

```
1. module lcd_funcmod
2. (
3.     input CLOCK, RESET,
4.     output LCD_CLOCK,
5.     output LCD_HSYNC, LCD_VSYNC,
6.     output [5:0]LCD_RED,LCD_GREEN,LCD_BLUE,
7.     output LCD_DE,
8.     output LCD_UD, LCD_LR,
9.     output LCD_MODE,
10.    output LCD_PWM,
11.    output [13:0]oAddr,
12.    input [15:0]iData
13. );
14. parameter SA = 11'd48, SB = 11'd40, SC = 11'd800, SD = 11'd40, SE = 11'd928;
15. parameter SO = 11'd3, SP = 11'd29, SQ = 11'd480, SR = 11'd13, SS = 11'd525;
16.
```

以上内容为相关的出入端声明以及常量声明。

```
17.     reg [10:0]CH;
```

```

18.      always @ ( posedge CLOCK or negedge RESET )
19.          if( !RESET )
20.              CH <= 11'd0;
21.          else if( CH == SE -1 )
22.              CH <= 11'd0;
23.          else
24.              CH <= CH + 1'b1;
25.
26.      reg [9:0]CV;
27.      always @ ( posedge CLOCK or negedge RESET )
28.          if( !RESET )
29.              CV <= 10'd0;
30.          else if( CV == SS -1 )
31.              CV <= 10'd0;
32.          else if( CH == SE -1 )
33.              CV <= CV + 1'b1;
34.

```

以上内容为列计数与行计数的周边操作。

```

35.      reg H;
36.      always @ ( posedge CLOCK or negedge RESET )
37.          if( !RESET )
38.              H <= 1'b1;
39.          else if( CH == SE -1 )
40.              H <= 1'B0;
41.          else if( CH == SA -1 )
42.              H <= 1'b1;
43.
44.      reg V;
45.      always @ ( posedge CLOCK or negedge RESET )
46.          if( !RESET )
47.              V <= 1'b1;
48.          else if( CV == SS -1 )
49.              V <= 1'b0;
50.          else if( CV == SO -1 )
51.              V <= 1'b1;
52.

```

以上内容为列控制以及行控制的周边操作。

```

53.      parameter XSIZE = 8'd128, YSIZE = 8'd96, XOFF = 10'd0, YOFF = 10'd0;
54.

```

```

55.     wire isX = ( (CH >= SA + SB + XOFF -1 ) && ( CH <= SA + SB + XOFF + XSIZEx -1) );
56.     wire isY = ( (CV >= SO + SP + YOFF -1 ) && ( CV <= SO + SP + YOFF + YSIZE -1) );
57.     wire isReady = isX & isY;
58.
59.     wire [31:0] x = CH - XOFF - SA - SB -1;
60.     wire [31:0] y = CV - YOFF - SO - SP -1;
61.

```

以上内容为图像信息的常量声明，有效行列，以及地址转换等即时声明。

```

62.         reg [31:0]D1;
63.         reg [15:0]D2;
64.
65.         always @ ( posedge CLOCK or negedge RESET )
66.             if( !RESET )
67.                 begin
68.                     D1 <= 18'd0;
69.                     D2 <= 16'd0;
70.                 end

```

以上内容为相关的寄存器声明以及复位操作。D1 暂存图像的地址信息，D2 暂存图像信息。

```

71.         else
72.             begin
73.
74.                 // step 1 : compute data address and index-n
75.                 if( isReady )
76.                     D1 <= (y << 7) + x;
77.                 else
78.                     D1 <= 14'd0;
79.
80.                 // step 2 : reading data from rom
81.                 // but do-nothing
82.
83.                 // step 3 : assign RGB_Sig
84.                 D2 <= isReady ? iData : 16'd0;
85.
86.             end
87.

```

以上内容为核心操作。它是流水操作，步骤 1 转换图像信息地址至 D1，步骤 2 等待图像信息反馈，步骤 3 暂存图像信息至 D2。

```

88.      reg [1:0]B1,B2,B3;
89.
90.      always @ ( posedge CLOCK or negedge RESET )
91.          if( !RESET )
92.              { B3, B2, B1 } <= 6'b11_11_11;
93.          else
94.              begin
95.                  B1 <= { H,V };
96.                  B2 <= B1;
97.                  B3 <= B2;
98.              end
99.

```

以上内容为对此行列延迟的周边操作。

```

100.     assign LCD_CLOCK = CLOCK;
101.     assign { LCD_HSYNC, LCD_VSYNC } = B3;
102.     assign LCD_RED = { D2[15:11],1'b0};
103.     assign LCD_GREEN = D2[10:5];
104.     assign LCD_BLUE = { D2[4:0],1'b0};
105.     assign LCD_DE = 1'b1;
106.     assign {LCD_LR, LCD_UD} = 2'b10;
107.     assign LCD_MODE = 1'b0;
108.     assign LCD_PWM = 1'b1;
109.     assign oAddr = D1[13:0];
110.
111. endmodule

```

以上内容为相关的输出驱动声。注意 LCD_RED/BLUE 都是舍弃最低位，LCD_LR/UD 为 2'b10，LCD_MODE 拉低，LCD_DE 常年拉高。

lcd_savemod.v

内容基本上与实验二十六一样。

lcd_basemod.v

连线部署请参考图 29.5。

```

1. module lcd_basemod
2. (

```

```
3.      input CLOCK, RESET,
4.
5.      output LCD_CLOCK,
6.      output LCD_HSYNC, LCD_VSYNC,
7.      output [5:0]LCD_RED,LCD_GREEN,LCD_BLUE,
8.      output LCD_DE,
9.      output LCD_UD, LCD_LR,
10.     output LCD_MODE,
11.     output LCD_PWM
12. );
13.
```

以上内容为相关出入端声明。

```
13.      wire CLOCK_25M;
14.
15.      pll_module U1
16.      (
17.          .inclk0 ( CLOCK ),
18.          .c0 ( CLOCK_25M )
19.      );
20.
```

以上内容为 PLL 的实例化。注意，7" TFT 除了最大像素时钟是 50Mhz 以外，余下可以任意设置。

```
21.      wire [13:0]AddrU2;
22.
23.      lcd_funcmod U2
24.      (
25.          .CLOCK( CLOCK_25M ),
26.          .RESET( RESET ),
27.          .LCD_CLOCK( LCD_CLOCK ),
28.          .LCD_HSYNC( LCD_HSYNC ),
29.          .LCD_VSYNC( LCD_VSYNC ),
30.          .LCD_RED( LCD_RED ),
31.          .LCD_GREEN( LCD_GREEN ),
32.          .LCD_BLUE( LCD_BLUE ),
33.          .LCD_DE( LCD_DE ),
34.          .LCD_LR( LCD_LR ),
35.          .LCD_UD( LCD_UD ),
36.          .LCD_MODE( LCD_MODE ),
37.          .LCD_PWM( LCD_PWM ),
38.          .oAddr( AddrU2 ),
```

```
39.           .iData( DataU3 )  
40.       );  
41.
```

以上内容为功能模块的实例化。

```
42.     wire [15:0]DataU3;  
43.  
44.     lcd_savemod U3  
45.     (  
46.         .CLOCK( CLOCK_25M ),  
47.         .RESET( RESET ),  
48.         .iAddr( AddrU2 ),  
49.         .oData ( DataU3 )  
50.     );  
51.  
52. endmodule
```

以上内容为储存模块的实例化。

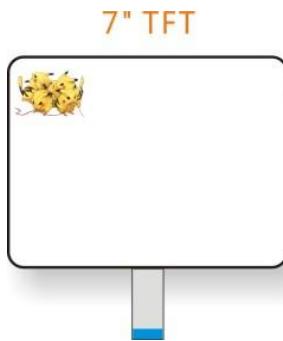


图 29.6 显示效果。

完后，综合程序并且下载进去，如果 7" TFT 的左上角出现一群小可爱，结果如图 29.6，那么表示实验成功。

细节一：完整的个体模块

本实验的 LCD 基础模块只是演示 7" TFT 如何驱动而已。