Verilog HDL 那些事儿整合篇

书语

续建模篇和时序篇以后，两者之间如何整合是一门非常重要的功课。在这本《Verilog 那些事儿-整合篇》一书中，笔者详细的谈论许多在整合里会出现的微妙思路，如：如何把计数器/定时器整合在某个步骤里，从何提升模块解读性和扩展性。此外，在整合篇还有一个重要的讨论，那就是 for，while 和 do ... while 等循环。这些都是一些顺序语言的佼佼者，可是在 Verilog HDL 语言里它们就黯然失色。

要在 Verilog 要实现 for 和 while 等循环是一个矛盾的作业，这话何说呢？首先我们可以用 Verilog 来模仿 for 和 while 等循环，这也是第一章的重点。可是随着我们深入了解 Verilog 各种不同的运行模式，模仿就会失去意义 ... 因为只要有一个指向步骤的 i 再加上一些整合的技巧，怎么样的循环我们都可以实现，这也是第五章的重点。

当然整合篇所讨论的内容不单是循环而已，整合篇的第二个重点是理想时序和物理时序的整合。说实话，笔者自身也认为要结合“两个时序”是一件苦差事，理想时序是 Verilog 的行为，物理时序则是硬件的行为。不过在它们两者之间又有微妙的 “黏糊点”，只要稍微利用一下这个“黏糊点”我们就可以非常轻松的写出符合“两个时序”的模块，但是前提条件是充足了解“理想时序”。

整合篇里还有一个重点，那就是“精密控时”。实现“精密控时”最笨的方法是被动式的设计方法，亦即一边仿真，一边估算时钟的控制精度。这显然是非常“传统”而且“古老”的方法，虽然有效但往往就是最费精神和时间的。相反的，主动式是一种讲求在代码上和想象上实现“精密控时”的设计方法。主动式的设计方法是基于“理想时序”“建模技巧”和“仿顺序操作”作为后盾的整合技巧。不说笔者吹牛，如果采用主动式的设计方法驱动 IIC 和 SDRAM 硬件，任何一段代码都是如此合情合理。

虽然在整合篇里还出现不相关的浮点数。事实上，笔者偶尔在寻找资料的时候才发现浮点数和 Verilog 有关的参考资料实在少得可怜。激动下，笔者写出浮点数加法器，减法器，乘法器和减法器的现实原理。这样做，一是为了补充这方面资料的不足，二是为了最后一章（优化和平衡）有足够的写作资源。

整合篇最后的重点就是模块的优化和平衡。笔者在其中解释到：在优化中常常会出现优化偏向的误点。一般从单片机过度而来的同学，都会把优化偏向 “越快越好”，但是在实际里， Verilog 的优化是“平衡”的：要嘛就是牺牲资源提升操作速度；要嘛就是牺牲操作速度节省资源；要嘛就是牺牲输出质量来优化前两者。以此类想，最终我们会遇上 Verilog 的各种操作模式。

在最后笔者还稍微解释有关组合逻辑设计和转换的知识与技巧。可能是基于建模技巧的优势，笔者设计的模块可以在各种操作模式之间相互转换。总而言之整合篇要讨论的内容都是在：循环，组合逻辑，操作模式，理想时序和物理时序之间盘旋的整合技巧。

前言

“呵呵呵”，先大笑三声庆祝这本笔记写完了，整个写作的时间有超过一年又几个月之久。整合篇可以说是笔者某个程度的至高作品，内容结合了前两篇的认识以外，有进一步扩展在 Verilog 世界里的见识。话来笔者也非常回味那些天，早晚奋力写作的日子。一字又一字成行，一行又一行成页，一页又一页成章，多少章的累计最终成书。

笔者之所以会动笔写整合篇的原因，除了意在完善建模篇和时序篇的内容以外，目的是为《Verilog 那些事儿》-系列，写下完美的句号。严谨来说，整合篇的内容基本上已经超出一般认识的范围，读者能不能接受笔者却不晓得。但是对于笔者来说，忽视整合篇里出现的重点，往往会使自己掉入学习 Verilog 的瓶颈。这番话不是胡说，而是事实。

就好比 for 循环，有多少人纠结与它呢？实际上，Verilog 有没有它都无关紧要，但是网上却有一大堆人不停在问：“for 怎样使用？”。很多时候笔者都苦笑不得 ... 啊,抱歉！笔者没有取笑的意思，笔者非常明白他们的心情，因为笔者也有类似的经历。只要换个角度，不停向内思索，答案往往都是那么简单，for 只是顺序操作，控制循环的关键字而已。

此外还有 IIC 和 SDRAM 等问题，也是 Verilog 的头目怪物之一。一般认为 IIC 和 SDRAM 的实现只要符合预期的效果即可，设计的内容不论什么细节和结构。笔者则认为许多参考资料都有意跳过重点巧妙包装细节，设计思路一字乱以外，根本无法学习其中的知识，往往会出现学习的误解。在此，笔者没有自认自己的设计思路是绝对正确，不过至少笔者还留下一颗学习的良心，可以解释的地方则解释，不可解释的地方则提出问题。人们常说“要死也要死得明白”，学习也是一样。

呼~整合篇的内容可不是一般的多，笔记也分成上下篇，故读者一定要沉住耐性，万里长城不是用几年的时间建成，而是用几代的时间建成。整合篇是一个持久的战役，要胜利先要征服自己，锻炼风吹不到，水冲不掉的耐性。笔者用人头保证，读者读完以后心理一定会萌出一份又一份满足感和成就感。

笔者最后奉上：整合篇所讨论的重点都是学习 Verilog 的瓶颈，一般上许多人都会用马死落地走的方法来学习，然而只要建模篇和时序篇的知识作为后盾，读者一定可以乘风破浪，一方向前。因为 Verilog 就是如此奇怪的家伙，毫无关系的细节却有微妙的联系，只有技巧有效手段正当，学习起来就是得心应手

整合篇的内容有不足之处请多多见谅 ... 还有,这些内容都是笔者的“一厢情愿”的想法，无法接受的朋友就笑笑而过吧，不要太认真。

akuei2 31-08-2012 上

目录

目录 04

第 1 章　整合的概念 07

1.01　源码上的整合 07
　　实验一：字面上的整合 07
1.02　时钟和步骤的定义 12
1.03　有效的使用资源（资源上的整合） 14
　　实验二：资源上的整合 14
1.04　循环保留和循环操作 25
1.05　把 C 语言的好处（循环）整合进来 26
1.06　把 C 语言的好处（嵌入循环）整合进来 30
1.07　把 C 语言的好处（动作的关系）整合进来 34
1.08　步骤的循环和时序的循环之间的区别 40
1.09　实验三：串口发送模块 46
1.10　实验四：串口发送模块 2 51
1.11　模块之间调用的延迟 66
　　实验五：被延迟的数据 68
1.12　同步数据的几个想法 76
　　实验六：加上过路寄存器 77
　　实验七：资源上的整合 82
总结 87

第 2 章　时钟化和信号的长度 88

2.01　一个时钟一块数据的概念 88
2.02　信号时钟化 93
2.03　深入了解模块的 9 通 95
2.04　电平检测模块的整合（即时事件在时序上的不和谐） 100
　　实验八：电平检测模块 101
2.05　电平检测模块的整合（即时事件在模块沟通上的不和谐） 112
2.06　实验九：串口接收模块 114
2.07　不同执行模式的“定时器和计数器” 129
2.08　计算 N 个长度的工具 134
2.09　实验十：重新设计 VGA 模块 137
2.10　主动设计的概念和模块的语言 161
　　实验十一：两个小屏幕 162
2.11　实验十二：比卡丘的 HP 血条 170
2.12　实验十三：我的小小字库 186

2.13 实验十四：字库的屏幕扩展（自定义字符串） 210
总结 227

第 3 章 理想时序的整合 228

3.01 理想时序和实际时钟的关系 228
3.02 速率和步骤的整合 231
实验十五：IIC 实验（芯片 24LC04 驱动） 232
3.03 IIC 总线被忽略的基础知识和速率的关系 254
3.04 按时序表驱动 IIC 设备 258
实验十六：真正意义上的速率 100kHz 267
3.05 非正规的速率 286
实验十七：非正规速率的高速（400Khz）iic 总线 286
3.06 不同世界的模块.....298
实验十八（一）：SDRAM 基本功能介绍 298
3.07 时间要求和理想时序之间的平衡点 318
实验十八（二）：SDRAM 基本功能驱动 319
3.08 高速率的风险 341
实验十九：SDRAM—100Mhz 基本驱动 345
总结 367

第 4 章 浮点数那些事儿 368

4.01 闻风色变的浮点数 368
4.02 浮点数和单精度格式 372
4.03 浮点数加减法 376
4.04 实定和假定小数点的困惑.....388
4.05 浮点数加减法实例 390
4.06 实验二十：浮点数加法器.....400
4.07 实验二十一：浮点数减法器 412
4.08 浮点乘法和陷阱 422
4.09 实验二十二：浮点数乘法器 435
4.10 浮点除法和精度流失 453
4.11 实验二十三：浮点数除法器 464
总结 476

第 5 章 优化和平衡 477

5.01 Verilog HDL 优化的思考 477
5.02 偏向节省逻辑资源的优化 481
实验二十四：轻量版-浮点数加法模块 481

实验二十五：超轻量版-浮点数加法模块 485
5.03 无意义的 for 循环 491
实验二十六：轻量版-浮点数加法模块・改 493
5.04 软件操作和循环操作的疑惑.....496
5.05 实验二十七：轻量版-浮点数乘法模块 498
5.06 实验二十八：超轻量版-浮点数乘法模块 507
5.07 实验二十九：轻量版-浮点数除法模块 515
5.08 再谈步骤和时钟的整合.....523
5.09 组合逻辑和即时操作 526
5.10 即时操作和建模 528
实验三十：即时式-Booth 乘法器 531
5.11 当笨蛋的组合逻辑遇上笨蛋的编译器 539
5.12 实验三十一：即时式-浮点数加法模块 542
5.13 组合逻辑的压力 559
实验三十二：即时式-Booth 算法乘法器（24 位） 561
实验三十三：即时式-浮点数乘法乘模块(Verilog 自带的 *操作符... 567
实验三十四：即时式-浮点数乘法模块（整合实验三十二） 579
5.14 延迟压力和时序上的微妙关系 1 582
5.15 延迟压力和时序上的微妙关系 2 587
5.16 与即时操作拼耐性.....595
实验三十五：即时式-Streamlined 除法器 595
实验三十六：即时式-浮点数除法模块（整合实验三十五） 600
5.17 浅谈嵌套 if.....608
总结 613

结束语 614

第一章 整合的概念

如果以笔者的脑袋去思考 Verilog HDL 语言的“整合概念”就是“减少资源的使用”（最近全世界都搞环保）。在这里“资源”的意思再也不是单单 FPGA 的逻辑资源，而是“时钟”和“步骤”。此外“整合”还有字面的意思，就是将代码整合起来，如果以抽象角度来看，意思就是提高内容的解读能力。除此以外 “整合”还有把其他语言的好处，以 Verilog HDL 语言的特质“整合”起来 .....

1.1 源码上的整合

“源码上的整合”正如字面上的意思，把模块的行数较少使观看更集中以外，还把源码编辑得更加密集起来。这个概念和“C 语言上的精简代码”有很大的不同，读者是不是摸不着头脑呢？呵呵，笔者以一个简单的实验来说明：

实验一：字面上的整合

c1a_module.v

```
1.    module c1a_module
2.    (
3.            input CLK,
4.            input RSTn,
5.
6.            output Q,
7.
8.            /******************/
9.
10.           output [4:0]SQ_C1,
11.           output [1:0]SQ_i
12.
13.   );
14.
15.         /*************************/
16.
17.           reg [4:0]C1;
18.
19.           always @ ( posedge CLK or negedge RSTn )
20.                 if( !RSTn )
21.                       C1 <= 5'd0;
22.                   else if( C1 == 10 - 1 )
```

```
                C1 <= 5'd0;
            else
                C1 <= C1 + 1'b1;

    /************************/

    reg [1:0]i;
    reg rQ;

    always @ ( posedge CLK or negedge RSTn )
        if( !RSTn )
            begin
                i <= 2'd0;
                rQ <= 1'b0;
            end
        else
            case( i )

                0:
                if( C1 == 10 -1 ) i <= i + 1'b1;
                else rQ <= 1'b1;

                1:
                if( C1 == 10 -1 ) i <= i + 1'b1;
                else rQ <= 1'b0;

                2:
                if( C1 == 10 -1) i <= i + 1'b1;
                else rQ <= 1'b1;

                3:
                if( C1 == 10 -1) i <= 2'd0;
                else rQ <= 1'b0;

            endcase

    /**********************************/

    assign Q = rQ;

    /****************/

    assign SQ_i = i;
    assign SQ_C1 = C1;
```

```
67.
68.         /**********************************/
69.
70.
71.   endmodule
```

上面是还没有进行整合的模块名为 c1a_module.v。该模块的作用很简单，就是每隔 10 个时钟翻转一下 Q 输出。在 17~25 行是 10 个时钟的定时器；在 29~57 行是操作步骤。在步骤 0（41~43 行）和步骤 2（49~51 行）是拉高输出，反之步骤 1（45~47 行）和步骤 3（53~55 行）是拉低输出。每个步骤 1 之间的间隔都是 10 个时钟，步骤 0~3 如此循环，直到永远（其中 C1 == 10-1 代表什么？笔者先保留先）。

当 Quartus II 编译完成后，给出逻辑资源的消耗是 “11 / 8,256 (< 1 %)”。

clb_module.v

```
1.    module c1b_module
2.    (
3.            input CLK,
4.            input RSTn,
5.
6.            output Q,
7.
8.            /******************/
9.
10.           output [4:0]SQ_C1,
11.           output [1:0]SQ_i
12.
13.   );
14.
15.         /*************************/
16.
17.           reg [1:0]i;
18.           reg [4:0]C1;
19.           reg rQ;
20.
21.           always @ ( posedge CLK or negedge RSTn )
22.                 if( !RSTn )
23.                         begin
24.                                 i <= 2'd0;
25.                                 C1 <= 5'd0;
26.                                 rQ <= 1'b0;
27.                             end
```

```
28.                 else
29.                     case( i )
30.
31.                             0:
32.                             if( C1 == 10 - 1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
33.                             else begin rQ <= 1'b1; C1 <= C1 + 1'b1; end
34.
35.                             1:
36.                             if( C1 == 10 - 1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
37.                             else begin rQ <= 1'b0; C1 <= C1 + 1'b1; end
38.
39.                             2:
40.                             if( C1 == 10 - 1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
41.                             else begin rQ <= 1'b1; C1 <= C1 + 1'b1; end
42.
43.                             3:
44.                             if( C1 == 10 -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
45.                             else begin rQ <= 1'b0; C1 <= C1 + 1'b1; end
46.
47.                         endcase
48.
49.         /***********************************/
50.
51.         assign Q = rQ;
52.
53.         /****************/
54.
55.         assign SQ_i = i;
56.         assign SQ_C1 = C1;
57.
58.         /***********************************/
59.
60.
61.   endmodule
```

上面是经过整合的模块名为 c1b_module.v。它的功能和 c1a_module.v 没有什么不同，只是它把定时器整合在 17~45 行的操作步骤里。我们再来看看 Quartus II 编译后，逻辑资源使用的结果同样也是“11 / 8,256 (< 1 %)”。

可能读者要问：“把定时器整合在操作步骤中到底有什么好处？”好处可多了。在 c1b_module.v 里定时器已经不复存在了，取而代之的是 C1 这个寄存器。C1 寄存器在操作步骤里是没有角色，或者说 C1 寄存器是一个通用的寄存器。读者要它成为计数器它就成为计数器，笔者要它成为暂存器它就成为暂存器。

当然,，既然这个寄存器已经命名为 C1（Counter1）笔者声明它的目的就是要它为我计数。把它整合在操作步骤里的目的是为了使操作更加方便（范围更广）。打个比方：

```
case( i )   //在这里先无视 -1 的作用，后面会详谈

    0：
    if( C1 == 8 -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
    else C1 <= C1 + 1'b1;

    1:
    if( C1 == 20 -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
    else C1 <= C1 + 1'b1;

    ....
```

从上面一段源码中我们可以看到：在步骤 0 的时候笔者要它计数 8 个时钟；在步骤 1 的时候笔者要它计数 20 个时钟。感觉上，操作的范围是不是更广了？而且从理解上更直接？

在步骤 0 中，C1 在每一个时钟都计数，当计数到的第 8 个时钟就清零 C1 然后递增步骤 i 以示下一个步骤。在步骤 1 中，C1 同样为计数器，但是计数的范围是 20 个时钟。同样的道理也应用在 c1b_module.v 中，更多的内容笔者就不过多解释了（后面会解释到）。

在这里，笔者提出了整合概念中之一的“源码上的整合”。事实上我们不止可以整合定时器在操作步骤里面而已，我们可以根据不同的要求而实现不同的整合方案，只不过定时器的整合确实是用得最多。“源码上的整合”表示了模块的表达能力上升（模块的源码更美观而且解读能力更好）。此外“源码上的整合”还使得可操作的范围变得更广了。

c1a_module.v 和 clb_module.v 同样为一样的逻辑资源消耗，但是 clb_module.v 模块的内容比起 cla_module.v 来得更多好处。换做读者，读者会选择 c1a_module.v 还是 c1b_module.v 的方法？呵呵，答案是不是很明显？如果读者认为优良的传统很重要的话，那么 c1a_module.v 是你的选择，反之一些喜欢挑战传统的家伙（笔者就是其中一个），那么 c1b_module.v 就是你的选择了。

在这一章节中，笔者只是提出其中一个有关整合的概念而已，至于令读者窝心的“-1”是什么？笔者会在下一章节中很好解释的，放心放心！ (*^__^*) 嘻嘻。

1.2 时钟和步骤的定义

关于时钟和步骤，笔者在时序篇上确实没有过多的定义它们。这也导致了时钟和步骤之间的定义产生的“暧昧”的关系。这也没有办法的事情，在早期中过意定义它们反而会本末倒置。但是在这一章节中，时钟和步骤的定义再也不可“暧昧下去”了。

说到步骤，读者会立即反应笔者情有独钟的“i”。“步骤”在 Verilog HDL 语言的角度上可以看成是简易的状态机，但是在笔者的眼中它是一个操作步骤。 i 除了指示步骤以外它还有显示指向时钟的能力，但是不是所有时候 i 都可以显示指向时钟。唯有“一个步骤只消耗一个时钟”的条件下，才可以称呼 i 为“显示指向时钟”。

那么笔者列出简单的例子：

```
case( i )

    0,1,2,3:
    begin rData <= rData + 1'b1; i <= i + 1'b1; end

    4:
    begin rData <= 4'd0; i <= i + 1'b1; end

    5:
    if( rData == 5 -1) rData <= 4'd0; i <= i + 1'b1; end
    else rData <= 1'b1;

endcase
```

在这里我们先假设 rData 和 i 寄存器的位宽均为 4 而且初值都是 0。在步骤 i 等于 0~3 的时候，具体的操作步骤是在每一个时钟都为 rData 递增。这时候的步骤 0~3 的操作都是每一个步骤都消耗一个时钟。同样的事情也发生在步骤 i 等于 4 的时候，步骤 4 的操作是清零 rData，它也是以一个时钟为消耗单位。由于步骤 0~4 的每一个操作都以一个时钟为消耗单位，所以在时序的角度上，我们可以结论下“步骤 i 有显示指向时钟的功能”。

在步骤 5 里，它的消时钟消耗单位为 5 个（为什么是 5 个？先别蛋急，听笔者把话说完）。步骤 5 的操作是在每一个时钟内都为 rData 递增，直到 rData 等于 5 -1 的时候（见鬼！怎么"-1"又出现了？.... 不怕不怕，再过不久笔者就会把它介绍了），亦即 rData 的结果等于 4 的时候，清零 rData 然后递增 i 以示下一个步骤。在这里我们再也不能说步骤 i 有显示指向时钟的能力，因为步骤 5 所需要的时钟消耗为 5 个。

图 1.2a 是上面一段代码中所产生的“理想”时序图。（在 Verilog HDL 里的世界我们需要一直把它看成是理想的状态。）

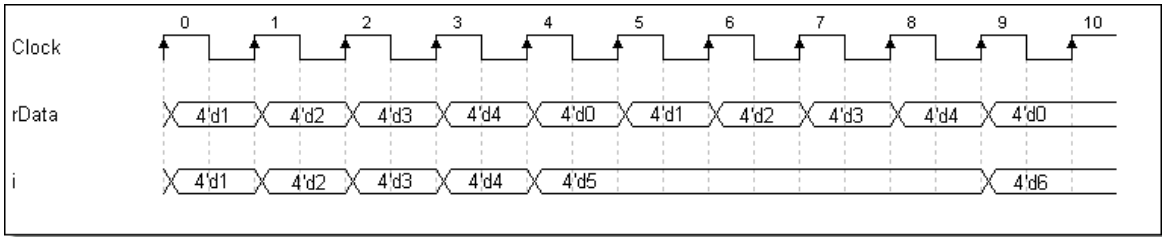


图 1.21a

(上述一段代码中所产生的理想时序图)

在 T0~T3 的时候，也是步骤 i 等于 0~3 的时候。步骤 0~3 的操作都是递增 rData，所以在 T0~T3 每一个时钟的未来值都是 rData 递增的结果，在 T3 的 rData 的未来值是 4'd4。在 T4 的时候，也是步骤 4 的时候，该步骤的操作是清零 rData，所以 rData 的未来值是 4'd0。在 T5 的时候，也是步骤 5 的时候，该步骤的操作是每一个时钟都为 rData 递增。在 T9 的时候，同样也是步骤 5，if 判断到 rData 的过去值是 4'd4，条件成立所以在 T9 的未来，rData 被清零，而且步骤也会递增。（在这里包含了很多时序篇里的基础知识，在这里笔者就不再重复了，如果读者有不懂的地方，请复习之前的笔记。）

到目前为止笔者已经清楚的划分了时钟和步骤的定义，希望读者不要取笑它们的暧昧关系（有关 -1 是什么笔者真的会很好的解释的，请再多忍耐一会）。

1.3 有效的使用资源（资源上的整合）

对于资源上的整合，在这里“资源”的对象不单单再也是逻辑资源了，而是更抽象的东西，既是“时钟和步骤”。“时钟和步骤”与“逻辑资源”的关系是正比的，只要“时钟和步骤”更能有效的利用，相比之下逻辑资源的消耗也会减少。如果可以更有效的使用“时钟和步骤”这个资源，从另一个角度上 Verilog HDL 语言在时序上的表现会更加无误。

实验二：资源上的整合

c2a_module.v

```
module c2a_module
(
        input CLK,
        input RSTn,

        output Q,

        /******************/

        output [4:0]SQ_C2,
        output [1:0]SQ_i

);

      /*************************/

        reg [1:0]i;
        reg [4:0]C2;
        reg rQ;

        always @ ( posedge CLK or negedge RSTn )
            if( !RSTn )
                begin
                        i <= 2'd0;
                        C2 <= 5'd0;
                        rQ <= 1'b0;
                    end
              else
                case( i )

```

```
31.                         0:
32.                         if( C2 == 10 ) begin C2 <= 5'd0; i <= i + 1'b1; end
33.                         else begin rQ <= 1'b1; C2 <= C2 + 1'b1; end
34.
35.                         1:
36.                         if( C2 == 10 ) begin C2 <= 5'd0; i <= i + 1'b1; end
37.                         else begin rQ <= 1'b0; C2 <= C2 + 1'b1; end
38.
39.                         2:
40.                         if( C2 == 10 ) begin C2 <= 5'd0; i <= i + 1'b1; end
41.                         else begin rQ <= 1'b1; C2 <= C2 + 1'b1; end
42.
43.                         3:
44.                         if( C2 == 10 ) begin C2 <= 5'd0; i <= i + 1'b1; end
45.                         else begin rQ <= 1'b0; C2 <= C2 + 1'b1; end
46.
47.                     endcase
48.
49.         /**********************************/
50.
51.         assign Q = rQ;
52.
53.         /****************/
54.
55.         assign SQ_i = i;
56.         assign SQ_C2 = C2;
57.
58.         /**********************************/
59.
60.
61.     endmodule
```

上面的 c2a_module.v 和实验一的 c1b_module.v 内容上是大同小异，不同的只是在 32 行，36 行，40 行和 44 行上面的“-1”。如果没有波形图的帮助，基本上我们根本看不出 c2a_module.v 到底有什么问题？

c2a_module.vt

```
1.    `timescale 1 ps/ 1 ps
2.    module c2a_module_simulation();
3.
4.        reg CLK;
5.        reg RSTn;
```

```
6.
7.          wire Q;
8.          wire [4:0]SQ_C2;
9.          wire [1:0]SQ_i;
10.
11.         /***********************/
12.
13.         c2a_module U1
14.         (
15.                 .CLK(CLK),
16.                 .RSTn(RSTn),
17.                 .Q(Q),
18.                 .SQ_C2(SQ_C2),
19.                 .SQ_i(SQ_i)
20.         );
21.
22.         /***********************/
23.
24.         initial
25.         begin
26.                 RSTn = 0; #1000 RSTn = 1;
27.                  CLK = 1; forever #5 CLK = ~CLK;
28.         end
29.
30.       /***********************/
31.
32.   endmodule
```

上面是 c2a_module.v 的激励文件，它没有什么特别。在第 1 行中声明了仿真用的时间刻度（时间单位）为 ps。在 4~9 行声明了输入和输出，在 13~20 行实例化了仿真对象 c2a_module.v 名为 U1。在 24~28 行是复位信号和时钟的刺激，前者为 1000ps 拉低电平然后拉高，后者是 10ps 周期的时钟（两者都可以随便填）。

仿真结果：

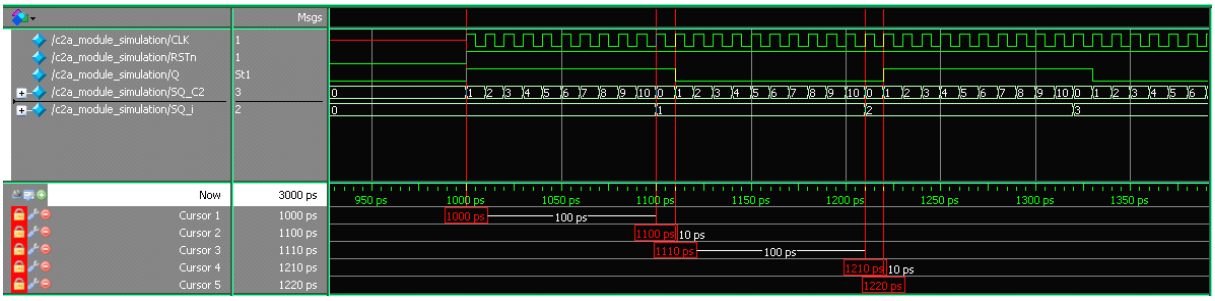


图 1.3a (c2a_module.v 仿真后的波形图)

图 1.3a 是 c2a_module.v 的仿真结果，在这里笔者不再自行的标上 T0~Tx 之类的标号了（会累死人的）。取而代之笔者会尝试以光标之间的距离计算出第几个时钟。假设从光标 1（Cursor1）开始时 0ps，在光标 1 和光标 2 之间含有 100ps，这也表示：

100ps / (时钟周期) = 100ps / 10ps = 10

亦即光标 1 和 2 之间有 10 个时钟，假设从光标 1 开始的上升沿作为 T0，那么光标 2 指向的地方是 T9。

```
31.        0:
32.        if( C2 == 10 ) begin C2 <= 5'd0; i <= i + 1'b1; end
33.        else begin rQ <= 1'b1; C2 <= C2 + 1'b1; end
```

在 c2a_module.v 的 31~33 行亦即步骤 0，在默认的情况下 rQ 会被拉高，然后 C2 在每一个时钟里都会递增，当 32 行的 if 条件成立的话（C2 == 10）的话，那么 C2 会被清零，然后 i 递增以示下一个步骤。综合 1.1 和 1.2 的知识，笔者要求计数器 C2 在步骤 0，为笔者计算 10 个时钟。换句话说，笔者要求步骤 0 所消耗的时钟是 10 个而已。

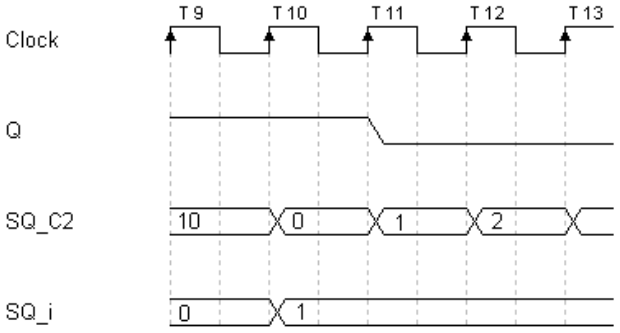


图 1.3b (光标 2 和光标 3 之间的波形图)

但是从仿真结果上我们可以看到（图 1.3b），在光标 1 和光标 2 之间的距离已经是 100ps ，亦即 10 时钟了。在光标 2 指向的时钟亦即 T10，在 T10 中 if 条件判断 C2 的过去值是 10，if 条件成立，C2 被清零和 i 被递增。所以在 T10 的未来，C2 的结果是零，i 的结果是 1。

那么问题来了，笔者所要求的步骤 0 是拉高 Q 保持 10 个时钟。但是在仿真结果中 Q 拉高，并且保持 11 个时钟（光标 1~3 之间的距离），这明显不是笔者要的结果。但是问题到底出在哪里？问题是出现在 T10（注意图 1.3b），在 T10~T11 之间 if 条件的成立使得 c2a_module.v 用了一个时钟的时间去更新 C2 和 i 寄存器的值。

换言之，步骤 0 使用的是 11 个时钟，而不是预期的 10 个时钟，随之的步骤 1~3 也犯同样的错误。如果读者的设计对时序上的要求不挑剔，那么多一个时钟也影响不大。打个例子：如流水灯实验，假设设计使用的频率是 20Mhz，流水灯效果的间隔是 10ms 的话，那么多一个时钟，流水灯效果的间隔也不过是 10ms + 50ns 而已，对于结果影响不大。

但是对于时序要求非常挑剔的设计还是完美主义者，这样的结果果断是一个很大的打击。
为此，笔者需要一种新概念，亦即不过度改变代码，而且还要提高内容的表达能力，此外还能很好的控制时序，故这就是“有效使用资源（资源上的整合）”个概念。

c2b_module.v

废话还是少说，还是直接看看经过修改的 c2a_module.v ，名为 c2b_module.v。

```
module c2b_module
(
        input CLK,
        input RSTn,

        output Q,

        /*****************/

        output [4:0]SQ_C2,
        output [1:0]SQ_i

);

      /************************/

        reg [1:0]i;
        reg [4:0]C2;
        reg rQ;

        always @ ( posedge CLK or negedge RSTn )
            if( !RSTn )
                begin
                        i <= 2'd0;
                        C2 <= 5'd0;
                        rQ <= 1'b0;
                    end
              else
                case( i )

                        0:
                        if( C2 == 10 -1 ) begin C2 <= 5'd0; i <= i + 1'b1; end
                        else begin rQ <= 1'b1; C2 <= C2 + 1'b1; end

                        1:
                        if( C2 == 10 -1 ) begin C2 <= 5'd0; i <= i + 1'b1; end
                        else begin rQ <= 1'b0; C2 <= C2 + 1'b1; end

                        2:
                        if( C2 == 10 -1 ) begin C2 <= 5'd0; i <= i + 1'b1; end
                        else begin rQ <= 1'b1; C2 <= C2 + 1'b1; end

```

```
43.                                 3:
44.                                 if( C2 == 10 -1 ) begin C2 <= 5'd0; i <= i + 1'b1; end
45.                                 else begin rQ <= 1'b0; C2 <= C2 + 1'b1; end
46.
47.                         endcase
48.
49.         /**********************************/
50.
51.         assign Q = rQ;
52.
53.         /****************/
54.
55.         assign SQ_i = i;
56.         assign SQ_C2 = C2;
57.
58.         /**********************************/
59.
60.
61.   endmodule
```

上面的源码是 c2b_module.v 的内容。它比起 c2a_module.v ，在 32,36,40,44 行的 if 条件中加入了“-1”。好了~就拿步骤 0 的 31~33 行来详解。不要小看这 3 行代码，它包含了笔者几天的苦思。在 31~33 行代码中，它所表达内容是：

“步骤 0 需要消耗 10 时钟（if C2 == 10），为此 C2 作为计算时钟的计数器。然而 if(C2 == 10 -1) 表达了，在步骤 0 的第 9 个时钟清零 C2 和递增 i（亦即结束步骤）”

在 c2a_module.v 的仿真结果中（图 1.3a），我们知道 c2a_module.v 为了清零 C2 和更新 i 消耗了 1 个时钟，使得步骤 0 从要求上的消耗 10 个时钟变成消耗 11 个时钟。不同的是，在 c2b_module.v 中 C2 被清零和 i 递增的动作是在步骤 0 的第 9 个时钟执行。换句话说笔者把清零 C2 和递增 i 这一个操作都考虑在步骤 0 的 10 个时钟之内。

读者是不是读到要翻桌子了？是不是捉拿不到笔者的意思？呵呵呵~还是直接看仿真结果吧。

c2b_module.vt

```
1.    `timescale 1 ps/ 1 ps
2.    module c2b_module_simulation();
3.
4.          reg CLK;
5.          reg RSTn;
6.
7.          wire Q;
8.          wire [4:0]SQ_C2;
9.          wire [1:0]SQ_i;
```

```
10.
11.         /**********************/
12.
13.         c2b_module U1
14.         (
15.                 .CLK(CLK),
16.                 .RSTn(RSTn),
17.                 .Q(Q),
18.                 .SQ_C2(SQ_C2),
19.                 .SQ_i(SQ_i)
20.         );
21.
22.         /**********************/
23.
24.         initial
25.         begin
26.                 RSTn = 0; #1000 RSTn = 1;
27.                 CLK = 1; forever #5 CLK = ~CLK;
28.         end
29.
30.       /***********************/
31.
32.   endmodule
```

c2b_module.v 的激励文件和 c2a_module.v 的激励文件没有什么不同，复位信号和信号是 1000ps 和 10ps 的周期。

仿真结果：

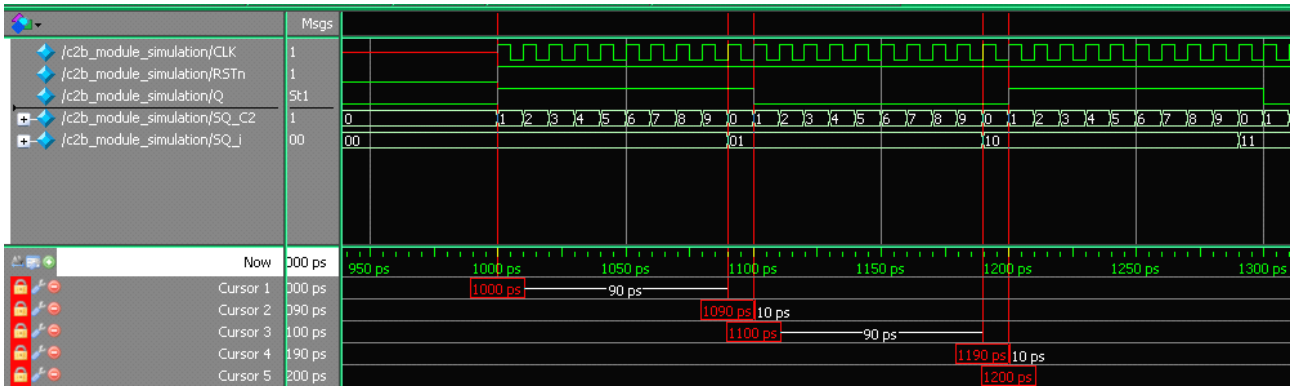


图 1.3c (c2b_module.v 的仿真结果)

上图 1.3c 是模块 c2b_module.v 的仿真结果。在这里，请注意光标 1~2 之间的距离，从上图中距离有 90ps，换做时钟来计算有 9 个时钟。在光标 2 指向之处，亦即 T9 的时候，正好也是在步骤 0。

```
31.                     0:
32.                     if( C2 == 10 -1 ) begin C2 <= 5'd0; i <= i + 1'b1; end
33.                     else begin rQ <= 1'b1; C2 <= C2 + 1'b1; end
34.
35.                     1:
36.                     if( C2 == 10 -1 ) begin C2 <= 5'd0; i <= i + 1'b1; end
37.                     else begin rQ <= 1'b0; C2 <= C2 + 1'b1; end
```

步骤 0 的 31~33 行之中尤其是 32 行，if 条件成立所以在 T9 的未来 C2 被清零和 i 被递增。在 T10 的时候，c2b_module.v 判断到 i 的过去值是 1，所以步骤 1 中 37 行被执行。在 T10 的未来 Q 的值更新为 0。

读者请继续注意图 1.3c，在光标 1~3 之间有 90ps + 10ps 的距离，如果换做时钟来计算的话既是 10 个时钟。话句话说，在步骤 0 中 Q 被拉高 10 个时钟之中，其中 T0~T8 是无聊的计数而已。反之步骤 0 在 T9 的时候，步骤 0 用于更新 C2 和 i 的值，C2 和 i 的值更新总共消耗了 1 个时钟。直到 T10，Q 被拉低之前，Q 在步骤 0 一共被拉高了 10 个时钟。

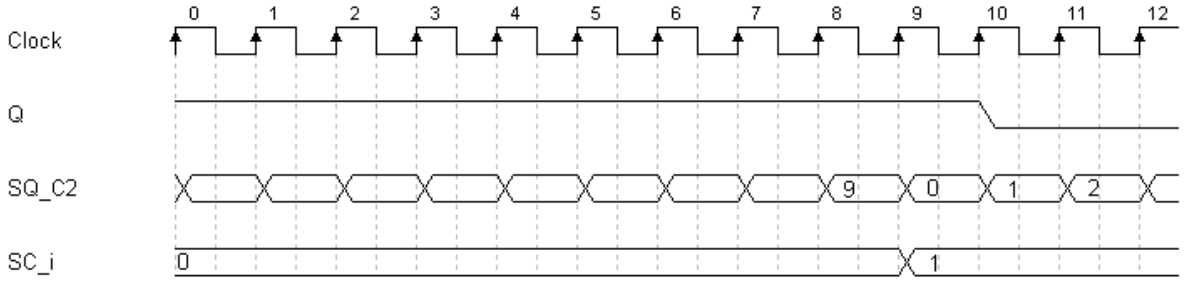


图 1.3d (c2b_module.v 的仿真结果)

还是上一副图来比较好些，图 1.3d 是 c2b_module.v 在 13 时钟之内的时序图。在 T9 之前 Q 已经被拉高 9 个时钟（T0~T8），然后再加上 C2 和 i 的更新所消耗的一个时钟（T9~T10 之间）Q 也是被拉高，所以 Q 一个被拉高了 10 个时钟（T0~T9）。在 T10 的时候，c2b_module.v 已经是转入步骤 1，然后步骤 1 会拉低 Q，所以 Q 在 T10 的未来值是 0。.

==========

在这里读者可能会犯思路上的困乱，因为笔者提出的新概念是“把 C2 和 i 更新所消耗的时钟”计算入步骤所需要消耗的时钟之内。用简单的表达式来表达得话是：

步骤 0 所消耗的时钟 = 9 个时钟计数 + 1 个时钟更新
= 10 个时钟

此外在“时序和 Verilog HDL 语言之间的理解上”笔者都把它整合起来。

举个简单的例子：

```
case( i )

    0,1,2,3,4,5,6,7,8:
    begin Q <= 1'b1; C2 <= C2 + 1'b1; i <= i + 1'b1; end

    9:
    begin Q <= 1'b1; C2 <= 5'd0; i <= i + 1'b1; end

    10,11,12,13,14,15,16,17,18:
    begin Q <= 1'b0; C2 <= C2 + 1'b1; i <= i + 1'b1; end

    19:
    begin Q <= 1'b0; C2 <= 5'd0; i <= i + 1'b1; end

    .......

endcase
```

如果把 c2b_module.v 的步骤 0 和步骤 1 非整合的话，结果会如上面一段代码，笔者稍微的来分析一下。在步骤 0~8，Q 被拉高，i 逐个步骤递增，C2 只是无聊的计数而已，然后在步骤 9 的时候 Q 还是被拉高，i 还是逐个步骤递增，C2 也只是无聊的清零而已。当步骤在 10~19 之间，同样的事情也会发生，不同的只是 Q 被拉低而已。

在这里 C2 好像废物那样-完全没有用处，我们之所以可以知道“在什么时钟发生什么操作”这是因为有 i 的显示指示时钟的功能而已。但是为了更好的减少内容的行数，提高模块的表达能力和使内容美观，笔者把 C2 当做计数器，将“既分散既有共同操作性质的步骤”全部整合起来。那么我们就不要那么多步骤了。

==========

可能读者又要发问：“以 c2b_module.v 的步骤 0 为例。既然要步骤 0 在第 9 个时钟的时候更新值，为什么 if 的判断形式不是 if(C2 == 9) 而是 if(C2 == 10 -1)?”这是一个很好的问题。在笔者的眼中，模块的表达能力是一切的一切，如何有效的把自己设计的东西很清晰的告诉它人，就是一个关键的所在。

再举个例子：

```
case( i )

    0:
    if( C2 == 9 ) begin C2 <= 5'd0; i <= i + 1'b1; end
    else begin Q <= 1'b1; C2 <= C2 + 1'b1; end
```

如果以一个 Verilog HDL 新手去了解上面的代码的话，他会得出这样的答案：

“在步骤 0，Q 被拉高，C2 每一个时钟递增，在 C2 递增到第 9 个时钟的时候，C2 清零，i

递增 ....”

这就是上述一段代码给它人死板板的理解反应，此外也没有“显示指示在哪一个时钟发生了什么操作”。就是类似的原因，许多新手会在理解 Verilog HDL 内容的时候心理会产生不安，所以才会频频依赖仿真。如果这个新手会把波形图的结果和模块的内容联系起来的话，结果还好，反之如果这个新手只会看波形图的话 .... 就像猴子看着罐头食物傻笑而已。为了更活化 Verilog HDL 内容给人的反应 ...“显示指示那个一个时钟发生什么事情”是很重要的。

再把上面的一段代码修改一下：

```
    case( i )

        0:
        if( C2 == 10 -1 ) begin C2 <= 5'd0; i <= i + 1'b1; end
        else begin Q <= 1'b1; C2 <= C2 + 1'b1; end
```

代码给人的答案又会不一样了：

“步骤 0 需要消耗 10 个时钟，步骤 0 在第 9 个时钟会清零 C2 和递增 i。此外在这个 10 个时钟里，步骤 0 的操作是拉低 Q”

在这里 if(C2 == 10) 表示了它需要消耗 10 个时钟；“-1”表示了步骤 0 在第 9 个时钟需要结束步骤（清零计数器和更新 i）。然而 else 这一行就表示了步骤 0 的“默认操作”了。

如果再从另一个角度去活用这样的想法的话，笔者给出一个简单的例子：

```
1.  case( i )
2.      0：
3.      if( C2 == 10 -1 ) begin rNumber <= 4'd9; C2 <= 5'd0; i <= i + 1'b1; end   // T9 结束步骤
4.      else if( C2 == 10 -2 ) begin rNumber <= 4'd8; C2 <= C2 + 1'b1; end        // T8
5.      else if( C2 == 10 -3 ) begin rNumber <= 4'd7; C2 <= C2 + 1'b1; end        // T7
6.      else if( C2 == 10 -4 ) begin rNumber <= 4'd6; C2 <= C2 + 1'b1; end        // T6
7.      else if( C2 == 10 -5 ) begin rNumber <= 4'd5; C2 <= C2 + 1'b1; end        // T5
8.      else begin rNumber <= 4'd0; C2 <= C2 + 1'b1; end                          // T0~T4
```

在上述代码中第 2 行告诉了我们，步骤 0 需要消耗 10 个时钟，在第 9 个时钟结束步骤(T9)。

在默认的情况下（T0~T4）rNumber 都是输出 4'd0；
在 T5 的时候 rNumber 输出 4'd5；
在 T6 的时候 rNumber 输出 4'd6；
在 T7 的时候 rNumber 输出 4'd7；
在 T8 的时候 rNumber 输出 4'd8；
在 T9 的时候 rNumber 输出 4'd9，并且结束步骤。

产生的时序会如同图 1.4e 那样。

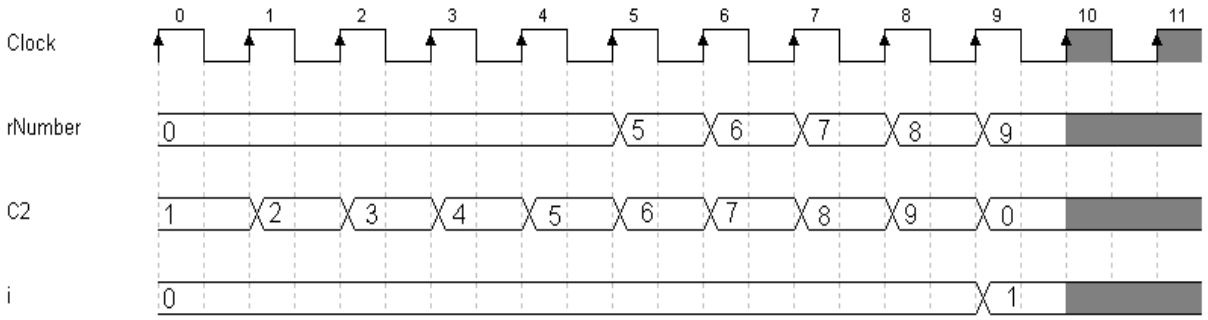


图 1.3e（T0~T9 为步骤 0 一共有 10 个时钟，T10 开始就是一个新步骤）

这个简单的例子告诉了我们一个事实，只要可以准确的计算某个步骤的时钟消耗，那么在这个步骤之内，我们可以任意决定“什么时钟产生什么动作”。至于写法，如果觉得太过猥琐的话也可以这样修改：

```
case( i )
    0：
    begin

        if( C2 == 10 -1 ) begin rNumber <= 4'd9; C2 <= 5'd0; i <= i + 1'b1; end
        else if( C2 == 10 -2 ) rNumber <= 4'd8;
        else if( C2 == 10 -3 ) rNumber <= 4'd7;
        else if( C2 == 10 -4 ) rNumber <= 4'd6;
        else if( C2 == 10 -5 ) rNumber <= 4'd5;
        else rNumber <= 4'd0;

        C2 <= C2 + 1'b1;
    end
```

==========

在这一章节结束之前，笔者可以这样总结。资源上的整合不只是有效的利用资源（把一个步骤向一个步骤之前所消耗的时钟都考虑在该步骤里），此外还大大的美观模块的内容和提高模块的表达能力。最重要的一点就是不在过度依赖步骤 i 来显示指示“哪一个时钟发生什么操作”。

1.4 循环保留和循环操作

“循环操作”估计有学过 C 语言的同学都不陌生这词，反而“循环保留”这词却没有出现在 C 语言的参考书上。这也难怪的“循环保留”只有在 Verilog HDL 会出现而已，举个简单的例子：

```
for( int x = 0; i < 8; i ++ )
{
    SCLK = 1; Delay( 100 );
    SCLK = 0; Delay( 100 );
}
```

假设笔者要有间隔的拉高又拉低高串行时钟信号 SCLK，笔者可以这样写。在传统里我们都习惯在 SCLK 的值更新后，都添加上延迟函数。

```
for( int x = 0; i < 8; i ++ )
{
      for( int y = 0; y <100; y++ )SCLK = 1;
      for( int y = 0; y <100; y++ )SCLK = 0;
}
```

但是在 C 语言里绝对不会出现如上的写法，就是利用 for 循环来逗留 SCLK 的停留时间。反之 Verilog HDL 语言却很乐意这样作。

```
case( i )

      0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15:
      if( C1 == 100 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
      else begin SCLK <= ~CLK; C1 <= C1 + 1'b1; end
```

上面一段代码表示了 SCLK 每一个半周期的更新都停留 100 个时钟，以致产生 16 个停留 100 个时钟的半周期 SCLK 信号，又或者 8 个停留 200 个时钟的周期 SCLK 信号（50%占空比）。以上这就是“循环保留”的意思。

为什么“循环保留”是 Verilog HDL 语言的权力，而不是 C 语言的权力呢？呵呵呵，这个问题的答案很简单，C 语言所消耗的时钟数是不可见的，然而 Verilog HDL 语言的时钟消耗数是可见又可自定义。这就是 Verilog HDL 其中之一的强大之处。

凡是有关“循环操作”对于“顺序语言”来说都是理所当然的事情。当我们用 C 语言实现 for 循环又或者嵌套 for 循环，值得我们在意的只有循环的操作次数和操作次序，根本就没有人会理会在 for 循环当中到底消耗了多少个时钟？如果我们要使用 Verilog HDL 语言来实现循环操作的话，C 语言的一套思想就不行了（除非无视时钟的消耗）。

1.5 把 C 语言的好处(循环)整合进来

说到 C 语言的好处莫过与它的语法和结构，目前我们不谈结构而是先谈语法。如果说到语法又会令人想起循环语法。C 语言有几个常用的循环语法 loop , while , do .. while 还有比较有人气的 for 循环。在早期，当笔者掌握了仿顺序操作的想法以后，基本上这些循环语法都可以模仿出来，但是无论笔者怎么模仿，感觉都是不怎么顺眼（不好控制）。其中笔者到底犯了什么问题呢？

首先我们先来看看 for 循环：

```
for( int i = 0; i <10; i ++ ) Act++ ;
```

关于 for 循环，它始终需要一个变量用来控制循环的次数。以上述的例子为例，i 变量就是用来控制循环的次数，然而右边的“Act++”就是“循环操作”。

如果要用 Verilog HDL 语言去模仿 for 循环的话，首先我们需要求出“循环个数”，亦即 n 个循环，从另一个角度来想的话也就是“多少个时钟数?”。假设以 for(int i = 0; i < 10; i++) 作为例子，循环个数是在 i 的 0 ~ 9 之间，亦即有 10 个数，也就是说在这个循环中使用了“10 个时钟数”。

实际上，在 C 语言的世界中，正如笔者在前 1.4 章节所说的那样。使用者仅注意“循环操作的次数”，而不是注意“这个循环操作使用了多少个时钟？”。一些保守的估算，10 次的循环操作，至少需要消耗超过 10 个以上的时钟。Verilog HDL 语言是富有并行性质的语言，如果不好好的利用这个好处，基本上 Verilog HDL 语言就没有任何模仿循环操作的目的。

好了，现在笔者用比较傻瓜的方法去模仿 for(int i = 0; i <10; i++){动作} :

```
case( i )

    0,1,2,3,4,5,6,7,8,9:
    begin Act <= Act + 1'b1; i <= i + 1'b1; end
```

这个估计是最简单了吧？啊哈哈哈~如果循环个数是 100 个的话，那么笔者既不要去见鬼了?此外 i 在“仿顺序操作”的写法中，是一个很重要的“步骤资源”，仅可以的话不要过度滥用它（之前的笔记就是太过滥用它了） ... 为此我们需要“额外”的计数器来计算循环的次数以外，我们还需要控制变量。

exp1_5a :

```
1.  case( i )
2.
3.  0：
4.  begin
5.
```

```
6.          if( x == C1 ) begin x <= x + 1'b1; Act <= Act + 1'b1; end
7.
8.          if( C1 == 10 -1 ) begin x <= 8'd0; C1 <= 8'd0; i <= i + 1'b1; end
9.          else C1 <= C1 + 1'b1;
10.  end
```

上面的这段代码，笔者把 for(int = 0; i <10; i++)集中在一个步骤里面。笔者不再滥用 i 了，取而代之笔者使用计数器 C1 去记录循环次数，和使用 x 这个控制变量。是不是觉得这个写法有点时曾相似？这种写法的基础思想基本上和“循环保留”的基础思想是没有什么两样，C1 表示了计算时钟的计数器，x 表示了控制循环的控制变量，if(C1 == 10 -1) 表示了循环结束（这一个步骤结束）。嗯 ~ 请读者不要摆出不明白的表情，我们还是在时序图中了解这些发生的经过：

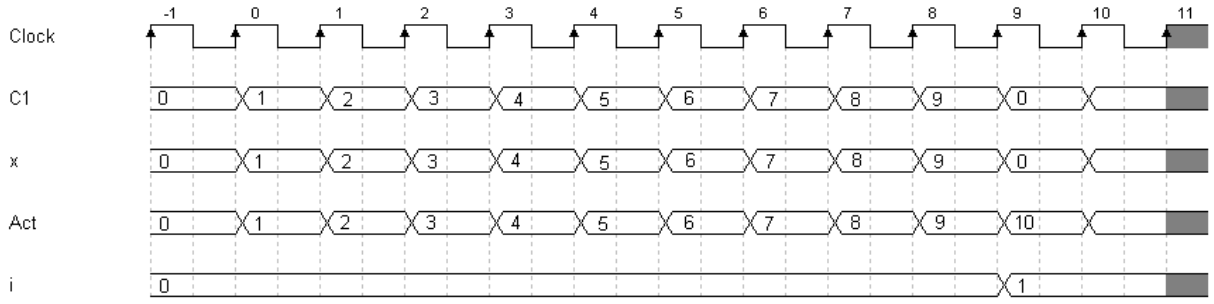


图 1.5a (Verilog HDL 语言模范 for()循环产生的时序)

图 1.5a 表示了上诉代码执行的经过。在 T-1 的时候是复位状态，所有的寄存器都清零。在 T0~T8 的时候（前 9 个时钟），C1，x，Act 寄存器按照步骤 0 的动作一直在递增（第 6 和 9 行）。直到 T9 的时候，第 8 行的 if 条件判断到 C1 的过去值是 10-1（9）并且条件成立，所以在 T9 的未来 C1 和 x 都会被清零，然后 i 会递增以示下一个步骤。在同一个瞬间第 6 行的 if 条件同样判断到 C1 和 x 的过去值都是一样，并且条件成立 ，所以 Act 递增，Act 的未来值是 10。

从 T0~T9 步骤 0 一共停留了 10 个时钟，而且 Act 也被执行了十次，所以 T9 之后 Act 的值是 10。此外在 T10 之前（亦即 T9 的未来值）C1 和 x 被清零，i 也递增成功。在这里“循环操作”和“循环保持”也是使用同样的一个思想，就是在进入下一个步骤之前，把 C1 ，x 和 i 等寄存器更新所消耗的一个时钟保留在同一个步骤中，或者说 C1 ，x 和 i 等寄存器更新所消耗的一个时钟都考虑在 10 次得循环操作之中。

唯一一点会使读者产生疑问的地方就是：

“在 T9 的时候，第 6 行和第 9 行的 if 条件同时都会成立，然而为什么第 6 行的 x <= x + 1'b1 没有被“决定”（执行），反而第 9 行的 x <= 8'd0 成功被“决定”（执行）？从而在根本上，步骤 0 的写法在理解上也存在问题？怎么编译器会通过这样充满矛盾的写法？这到底是怎么一回事？”

呵呵，事实上这样的写法没有存在问题，有问题的是人类在习惯上的理解方式。在步骤 0，

当时钟迎来 T9 的时候，从并行的解读上第 6 行，和第 9 行的 x <= x + 1'b1 和 x <= 8'd0;都会被执行，使而产生“两态性”的问题（就是两个资源同事间争与使用同一个输出），最后编译器会产生问题 ... 但是，事实上却不是如此。不知道读者们也没有读过建模篇-仿顺序操作 哪一篇呢？其中仿顺序操作的建模同样也遇见“两态性”或者“多态性”的问题，在文中笔者是使用输出选择器来协调操作。

同样的道理也发生在步骤 0，在第 6 行和第 9 行的 x <= x + 1'b1 与 x <= 8'd0 之间，编译器“偷偷”的建立输出选择器，大致上是这样写的：

```
always @ ( * )
      if( C1 == 10 -1 ) x <= 8'd0;
      else if( x == C1 ) x <= x + 1'b1;
```

在这里，我们可以看到 (C1 == 10 -1)的优先级高于 (x == C1)，所以 x <= 8'd0 被执行是理所当然的。但是问题是“要如何定义这个莫有（隐藏）输出选择器的优先级呢？”，笔者真的要说声抱歉了，这是取决于编译器的编译行为，一般上越是下面的语句优先级就越高。正如步骤 0 所示，第 9 行的 x <= 8'd0 低于第 6 行的 x <= x + 1'b1，所以第 9 行的 x <= 8'd0 的优先级更高。

exp1_5a :

```
case( i )

     0：
     begin
          if( x == C1 ) begin x <= x + 1'b1; Act <= Act + 1'b1; end

          if( C1 == 10 -1 ) begin x <= 8'd0; C1 <= 8'd0; i <= i + 1'b1; end
          else C1 <= C1 + 1'b1;
     end

     1：
     begin C1 <= 8'd10, x <= 8'd20, Act <= 8'd30; i <= 4'd1; end
```

笔者再添加一个步骤试一试，在 T10 时候会产生怎样的时序：

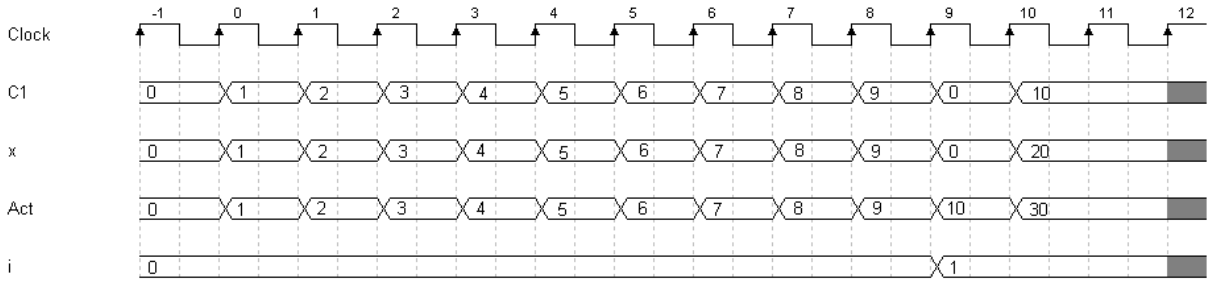


图 1.5b (经过添加步骤 1 的时序图)

图 1.5b 表示了，在 T10 之后 C1，x 和 Act 等寄存器的未来值如步骤 1 的操作所示，C1 的未来值是 10，x 的未来值是 20，Act 的未来值 30。至于 i 故意被赋予 1，即表示一直停留在步骤 1 中。

事实上，无论是“循环操作”还是“循环保留”它们都是使用“同样的想法”，不过是行为上的定义不同而已，所以写法也不同，但是“循环操作”的写法确实是怪了点 ...

在本章节结束之前，笔者要简单的强调一下：

C 语言在传统上是顺序操作语言，就是一个步骤一个动作，反之 Verilog HDL 语言则不是。在 C 语言的世界里，是没有“时序”这一概念，然而 Verilog HDL 语言在时序上既有“时间点事件”和“即时事件”。所以我们不能一股脑把 C 语言见到的都往 Verilog HDL 语言里搬，我们既要照顾好时序，然后又要好好控制循环，对于语法上的编辑需要特别小心。

在这里，正如 for(int i = 0; i < 10; i++) Act++; 所愿 Act 在 10 个时钟之内成功递增 10 次。笔者为了有效利用时钟（时钟资源）笔者把 C1 ，x 和 i 等寄存器更新所消耗的一个时钟，都考虑在循环操作的动作中。也就是说，每一个时钟都要榨干它！（充分利用它）。

1.6 把 C 语言的好处(嵌入循环)整合进来

```
for( int x = 0; x < 10; x++ )
    for( int y = 0; y < 10; y++ )
        Act++;
```

上面是一个 for 嵌入循环，在 Verilog HDL 语言上我们无法再使用 c 语言那样的“循环思想”来模仿它，取而代之我们需要把嵌入 for 循环以“时序”的形式表现出来。如果按照计算循环个数（时钟）的话，它一共循环了 100 次（使用了 100 个时钟）。在 Verilog HDL 语言上，我们可以这样写，假设 x 寄存器的复位值（初值）是 0 的话：

exp1_6a：

```
case( i )

    0:
    begin
        if( x == C1 ) begin x <= x + 1'b1; Act <= Act + 1'b1; end

        if( C1 == 100 -1 ) begin x <= 8'd0; C1 <= 8'd0; i <= i + 1'b1; end
        else C1 <= C1 + 1'b1;
    end
```

==========

但是对于嵌入 for 循环来说，实际上是不会单单出现一个动作而已，很多时候会有多种动作出现在嵌入 for 循环。

```
for( int x = 0; x < 10; x++ )
    for( int y = 0; y < 10; y++ ) { Act1++; Act2++; }
```

首先，我们先把 Act1 和 Act2 看成是“独立关系”的（互不相关），所以它们可以在同一个时钟之中一起被执行。在这个嵌入 for 循环中，x 控制 y，y 控制 Act1 和 Act2，x 循环操作 10 次，y 循环操作 10 次，所以 Act1 和 Act2 一共循环操作了 100 次。

exp1_6b :

```
case( i )

    0:
    begin
        if( x == C1 ) begin x <= x + 1'b1; Act1 <= Act1 + 1'b1; Act2 <= Act2 + 1'b1; end

        if( C1 == 100 -1 ) begin x <= 8'd0; C1 <= 8'd0; i <= i + 1'b1; end
        else C1 <= C1 + 1'b1;
```

```
    end
```

在这里（假设 x 寄存器的复位值都是 0 的话）我们不需要用两个控制变量 x 和 y，反而我们可以仅用 x 变量来控制循环操作。此外，由于 Act1 和 Act2 是“独立关系”所以它们可以在同一个时钟内操作。由此而知，在使用 Verilog HDL 语言实现上述的嵌入 for 循环的时候，我们可以仅单单使用一个控制变量 x 来实现 100 次得循环操作。在 if(x == C1) 条件成立的时候 Act1 和 Act2 同时发生操作，亦即同时递增（双方没有关系）。直到循环到 100 次的时候（T0~T99），x 和 C1 等寄存器会被清零，i 递增以示下一个步骤。

==========

```
for( int x = 0; x < 10; x++ )
{
    Act1++;
    for( int y = 0; y < 10; y++ )
        Act2++;
}
```

这一回在 c 语言的嵌入 for 循环中，从理解上，如果 Act1 和 Act2 还是“独立关系”的话，x 控制 y 和 Act1，y 控制 Act2，x 循环操作 10 次，y 循环操作 10 次，那么 Act1 是循环操作 10 次，至于 Act2 会循环操作 100 次。

如果要以 Verilog HDL 语言实现的话，第一我们要考虑“最大的循环个数”，亦即 100 个循环个数，所以 C1 需要计算 100 个的时钟数。然后使用 x 寄存器控制 Act1，使用 y 寄存器控制 Act2。

exp1_6c:

```
case( i )

    0:
    begin
        if( x == C1 ) begin x <= x + 8'd10; Act1 <= Act1 + 1'b1; end
        if( y == C1 ) begin y <= y + 1'b1; Act2 <= Act2 + 1'b1; end
        if( C1 == 100 -1 ) begin x <= 8'd0; y <= 8'd0; C1 <= 8'd0; i <= i + 1'b1; end
        else C1 <= C1 + 1'b1;
    end
```

请注意一下，由于 x 控制 Act1 并且循环操作次数是 10 次，从 100 次的总循环操作中，x 每 10 个时钟（每次循环操作）就更新一次。总循环操作次数 / x 操作次数，亦即 100 / 10 所以 x 每一次的更新值都是 10。同样，每一次 x 寄存器被更新，Act1 也会被执行。最终在 100 次得循环操作中，x 总共被更新 10 次，Act1 被执行 10 次。

接下来 Act2 是由 y 控制，根据 c 语言嵌入 for 循环的操作方式，每一次 x 更新一次 y 就被更新 10 次，Act2 也被执行 10 次。在这里 x 一共更新 10 次，y 也被更新 100 次，Act2 也被执行 100 次。根据 总循环操作次数 / y 操作次数 ，亦即 100/100 所以 y 每一次更新值都是 1。

(*^__^*) 嘻嘻……笔者就知道，单单用文字表达好似念经那样，看了都要学猴子抓头。

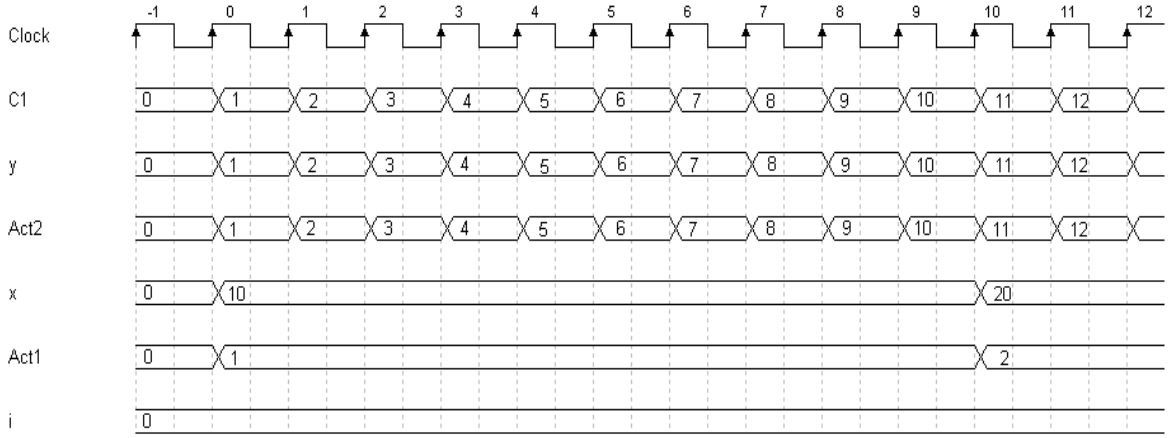


图 1.6a (前 12 个时钟的经过 T0~T11)

图 1.6a 表示了前 12 个时钟的经过，T -1 表示了复位状态，所有相关的寄存器都被清零。在 T0 的时候，x 和 y 与 C1 的过去值都为 0，if(x == C1) 和 if(y == C1) 的条件成立，所以在 T0 的未来，y 和 Act2 的未来值都是 1（Act2 成功被执行 1 次所以 y 控制变量也更新一次），至于 Act1 的未来值也成功递增为 1。但是别忘了，Act1 是被 x 控制着的，然而 Act1 在 100 次得循环操作中，只被允许操作 10 次而已，为止每一次 Act1 成功被执行 x 都会递增 10。

同样的事情也发生在 T10 的时候（我们先无视 y 和 Act2），在步骤 0 里 if(x == C1)判断到 x 和 C1 的过去值均为 10，if 条件成立。所以在 T10 的未来，Act1 再一次被执行并且递增 1，未来值为 2，反之 x 被更新并且递增 10，未来值为 20。

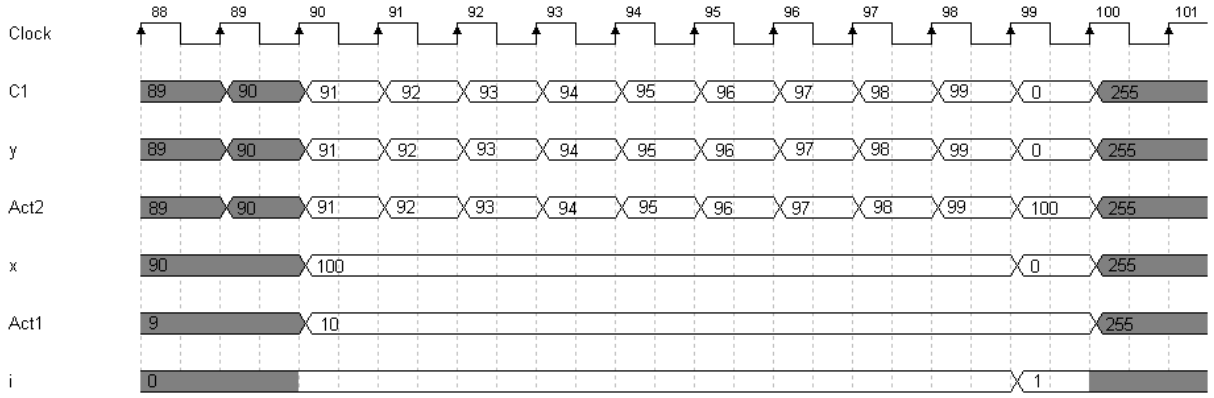


图 1.6b (后 12 个时钟 T88~T99 的发生经过)

图 1.6b 是后 12 个时钟的发生经过，亦即 T88~T99。在 T90 的时候，亦即是 Act1 的最后一次操作，if(x == C1) 判断 x 和 C1 的过去值，结果 if 条件成立。在 T90 的未来，x 和 Act1 的未来值均为从原本的 90 至 100 ~ 9 至 10。经过 T90 之后，基本上 x 和 Act1 的工作都完成了。x 更新了 10 次，Act1 也操作了 10 次。在 T90~T99 之间是 Act2 的最后 10 次循环操作。

在最关键的 T99 到临之际，笔者为了节省寄存器们清零所花的一个时钟，所以也把这一个时钟也考虑在这 100 次得循环操作当中。在 T99 的时候，if(C1 == 100 -1)条件成立 x，y 和 C1

等寄存器陆续被清零，然而 i 递增以示下一个步骤。所以在 T99 的未来，x，y 和 C1 的未来值均为 0，i 的未来值为 1 以示已经准备好进入下一个步骤。所以在 T100 会是进入步骤 1。

同样另一个关键时刻（台湾某个节目~呵呵）也发生在 T99 这个时候。if(y == C1) 条件成立，Act2 决定递增，然而 y <= y + 1'b1; 却因为隐藏输出选择器的关系被无视掉。此外 if(x == C1) 条件不成立，所以它作罢。所以在 T99 的未来，Act2 的未来值是从原前的 99 递增到 100，反之 Act1 没有变化。

最终在这个 100 次得循环操作中，Act2 循环操作了 100 次，至于 Act1 循环操作了 10 次。

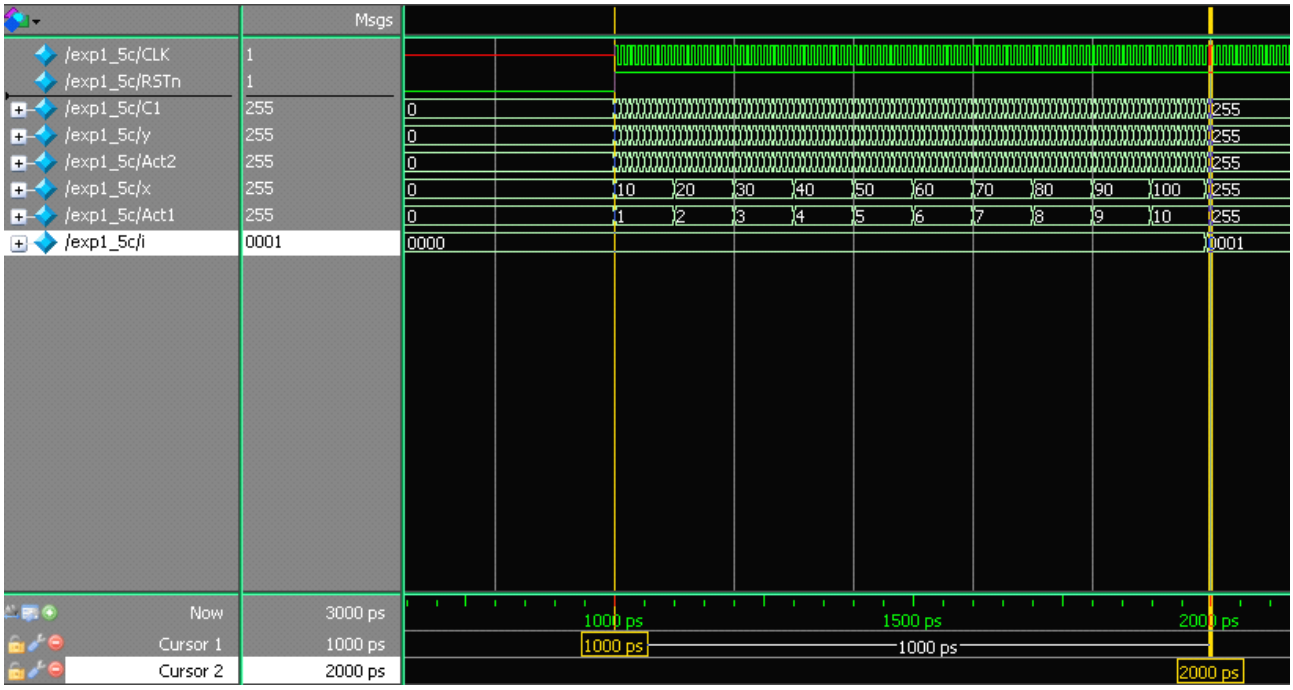


图 1.6c (仿真结果)

图 1.6c 是利用 Modelsim 的仿真结果，在光标 1~2 之间是步骤 0，亦即循环操作发生的经过。关于光标 1~2 之间的时间距离是 1000ps (仿真所使用的时钟周期为 10ps)，将 1000ps / 10ps 结果是 100，亦即在步骤 0 之中消耗了 100 个时钟数。在光标 2 之后的是步骤 1 如愿以偿的操作结果。

1.7 把 C 语言的好处(动作的关系)整合进来

所谓的动作“独立关系”就是动作之间是没有关系的。就好比：

```
for( int x = 0; x < 10; x++ )
{
    Act1++;
    for( int y = 0; y < 10; y++ )
        Act2++;
}
```

和 ：

```
for( int x = 0; x < 10; x++ )
{
    for( int y = 0; y < 10; y++ )
        Act2++;
    Act1++;
}
```

无论 Act1 是放在 y 控制的 for 循环的前面还是后面，只要 Act1 和 Act2 是“独立关系”的话，在 Verilog HDL 语言的理解上，在这 100 次循环操作中，Act2 都会循环操作 100 次，Act1 都会循环操作 10 次。相反的，如果在 C 语言的理解上 Act1 就会有步骤上的关系，但是在前面笔者就说过了，C 语言所用来理解“循环操作”的思想，是不适合用在理解 Verilog HDL 语言的身上。

```
for( int x = 0; x < 10; x++ )
{
    Act1++;
    for( int y = 0; y < 10; y++ )
        Act2 = Act1;
}
```

上面一段代码，Act1 和 Act2 再也不是“独立关系”了，取而代之的是 Act1 和 Act2 有“依赖关系”，其外 Act1 和 Act2 也有“次序关系”。“依赖关系”的意思如字面上所述的那样，Act1 赋值与 Act2，反而“次序关系”是表示 Act1 先操作，然后才轮到 Act2 操作。那么我们又要如何使用 Verilog HDL 语言来实现呢？

exp1_7a:

```
case( 0 )

    0:
    begin
        if( x == C1 ) begin x <= x + 8'd10; Act1 = Act1 + 1'b1; end
```

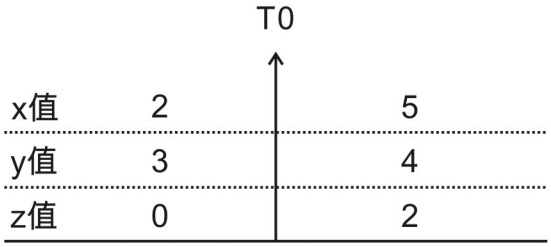
```
        if( y == C1 ) begin y <= y + 1'b1; Act2 <= Act1; end

        if( C1 == 100 -1 ) begin x <= 8'd0; y <= 8'd0; C1 <= 8'd0; i <= i + 1'b1; end
        else C1 <= C1 + 1'b1;
    end
```

exp1_7a 的源码基本上和之前的例子没有多大的改变。x 用于控制 Act1，y 用于控制 Act2。其中 if(C1 == 100 -1)代表了这个循环操作使用了 100 个时钟。x 每次的更新值是 10，反之 y 每次的更新值是 1，也就是说在 100 个时钟里 Act1 每个 10 个时钟执行一次操作 Act1<=Act1 + 1'b1，然而 Act2 每个时钟都执行一次 Act2 <= Act1。请问读者，在这里你们有发现什么奇怪的地方吗？

对 exp1_7a 使用了“即时事件”，亦即 Act1 使用了阻塞式赋值操作符“=”。没有读过“时序篇”的朋友估计要犯傻了，是吗？笔者简单的介绍一遍 ：

图 1.7a (时间点的概念)

对于 Verilog HDL 语言的时序概念来说，最关键的就是“时间点事件"。在时间点的直线中，左边的判为“过去值”右边判为“未来值”。笔者就举个简单的例子：

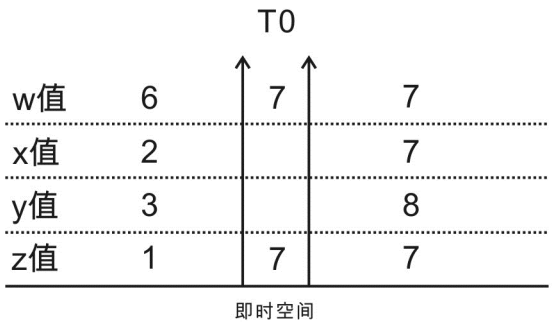



```
case(0)
   0:
   begin
      if( x == 2 ) y <= 4;
      x <= 5; z <= x;
   ......
```

图 1.7b（时间点事件例子）

图 1.7b 是时间点事件的典型例子。在 T0 之前假设 x 的初值是 2，y 的初值是 3, z 的初值是 0。在进入时间点 T0 的时候，if 条件先判断，在 T0 点 x 的过去值是否为 2，结果条件成立，y“决定”赋值与 4。其外在时间点 T0，x“决定”赋予 5，然后 z“决定”赋予 x 的过去值，亦即 2。结果在 T0 的未来，x 的未来值是 5，y 的未来值是 4，z 的未来值是 2。

在这里请读者稍微注意一下，在上述一段内容中，笔者所使用的“决定”代表了“在该时间点相关的值虽然已经更新但是还没有生效”，只有在 T0 过后这些值才生效并且输出。“即时事件”的行为悄悄好相反。我们再来看一个简单的例子：




```
case(0)
  0:
  begin
    w = 7;
    if( w == 7 ) x <= 7;
    y <= 8;
    z = w;
  ......
```

图 1.7c（即时事件例子）

图 1.7c 是“即时事件”的典型例子。假设在 T0 之前 w 的初值是 6，x 的初值是 2，y 的初值是 3，z 的初值是 1。在 T0 来临之际，w“直接”赋予 7，然后 if 条件判断 w==7，在这里，“请问 w 是以该时间点的过去值 6 作为基础，还是在即时空间中的 7 值作为基础？”，答案是以“即时空间中的值 7”作为判断基础。if 条件成立 x“决定”赋值与 7。

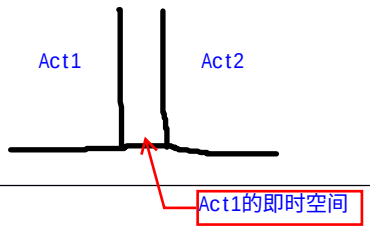
其他的如 y“决定”赋值与 8，然而 z“直接”赋予 w，而不是在该时间 w 的过去值，亦即是 7 不是 6。同样的事情也发生了，由于 z 也是基于“即时事件”所以在即时空间中 z 值更新后立即生效。结果在 T0 的未来，w 的未来值是 7 ， x 的未来值是 7，y 的未来值是 8，z 的未来值是 7。在这一切的过程当中，只有 x 和 y 的赋值是“时间点事件”，但是成立 x 的 if 判断却是“即时事件”。

唉~写到这里笔者又感叹起来 ... 时间点事件和即时事件的区分理解对于明白“时序的概念”，是一件非常重要的基础。可惜呀，很多人选择无视，很多人则不把 Verilog HDL 语言看成是理想的语言 .... 所以也没有这些概念，要把波形图看好需要付出更大的代价 ... 基本上“事件点事件”和“即时事件”的区别仅此而已。

继续被中断的主题，阻塞赋值或者即时事件从另一个角度来说“即时事件的优先级高于时间点事件”，当然事实上并非如此，但是这两个事件之中却有如此的微妙关系。

```
case( 0 )

    0:
    begin
        if( x == C1 ) begin x <= x + 8'd10; Act1 = Act1 + 1'b1; end
        if( y == C1 ) begin y <= y + 1'b1; Act2 <= Act1; end

        if( C1 == 100 -1 ) begin x <= 8'd0; y <= 8'd0; C1 <= 8'd0; i <= i + 1'b1; end
        else C1 <= C1 + 1'b1;
    end
```

有关 exp1.7a 其中的 Act2 <= Act1 是以 Act1 的“即时值”更作为赋值基础，而不是以该时间点的 Act1 过去值作为基础。所以说，Act1 和 Act2 是同步（并行）更新，但是又有优先级这股微妙的关系。Act2依赖Act1



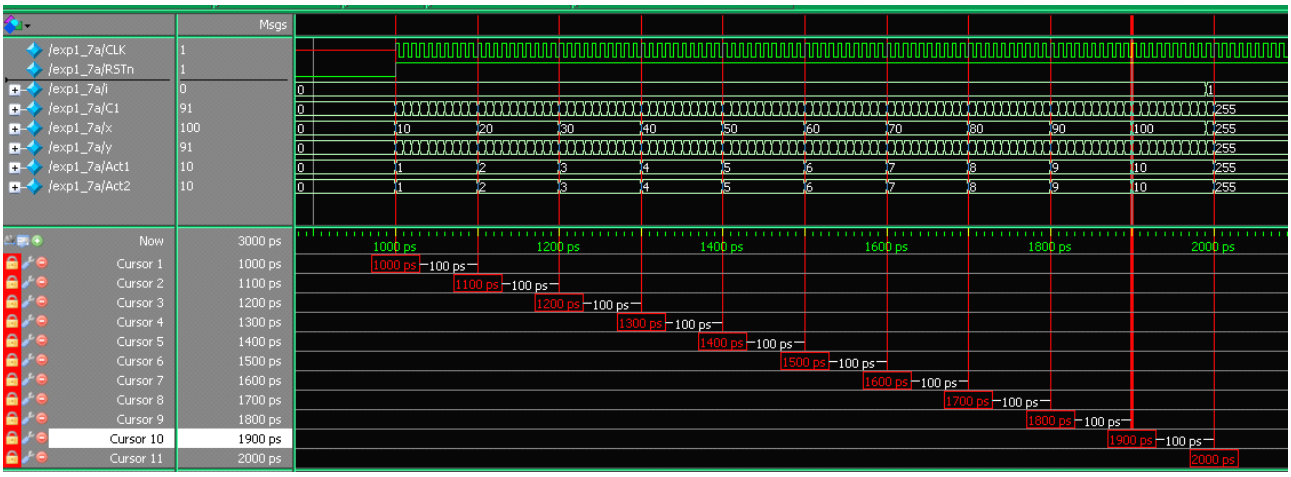


图 1.7d（exp1_7a 仿真结果）

图 1.7d 是仿真结果，其中的 Act1 和 Act2 完全是同步（并行）更新，然而在 100 个时钟之内每一次更新间隔是 10 个时钟。

```
for( int x = 0; x < 10; x++ )
{
    Act1 = Act2;
    for( int y = 0; y < 10; y++ )
        Act2++;
}
```

笔者又稍微换一换嵌入 for 循环看看，这个时候的 Act1 = Act2 的操作次序先于 Act2++，换做 Verilog HDL 语言我们又如何是好！?

exp1_7b:

```
case( i )

    0:
    begin
        if( x == C1 ) begin x <= x + 8'd10; Act1 <= Act2; end
        if( y == C1 ) begin y <= y + 1'b1; Act2 <= Act2 + 1'b1; end

        if( C1 == 100 -1 ) begin x <= 8'd0; y <= 8'd0; C1 <= 8'd0; i <= i + 1'b1; end
        else C1 <= C1 + 1'b1;
    end
```

还是：

```
case( i )

    0:
```

```
begin
        if( x == C1 ) begin x <= x + 8'd10; Act1 = Act2; end
        if( y == C1 ) begin y <= y + 1'b1; Act2 <= Act2 + 1'b1; end

        if( C1 == 100 -1 ) begin x <= 8'd0; y <= 8'd0; C1 <= 8'd0; i <= i + 1'b1; end
        else C1 <= C1 + 1'b1;
end
```

效果上基本都是半斤八两 ，看了仿真结果就知道了。

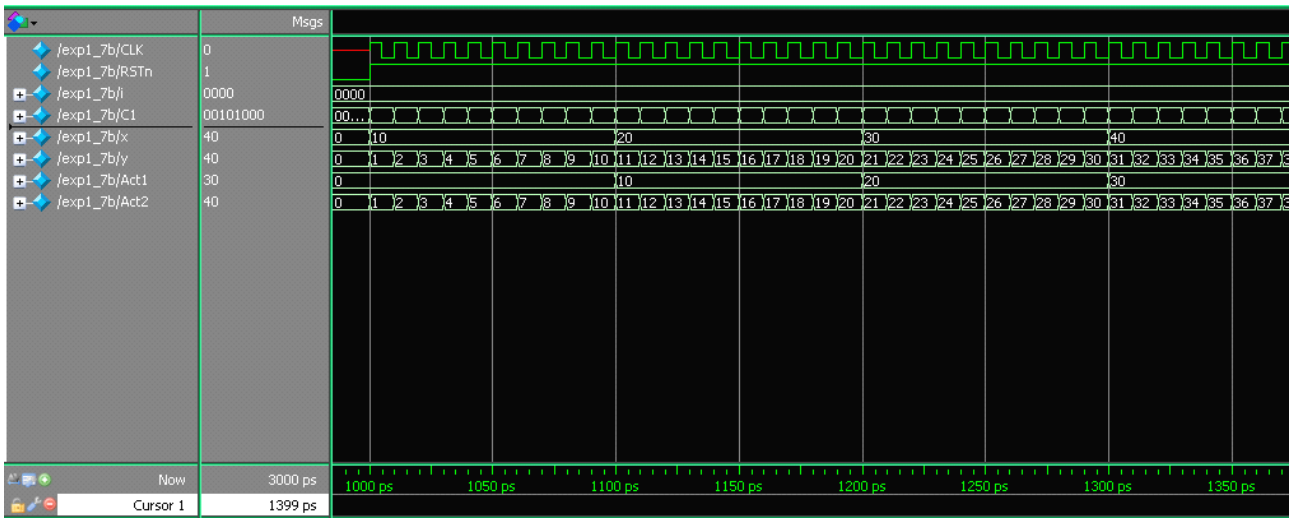


图 1.7e （exp1_7b 仿真结果）

图 1.7e 暴露了，基本上两段代码的效果都是一样的 ... 为什么呢（⊙_⊙）？ 答案很简单，因为 Act2 是每一个时钟更新，反之 Act1 每隔 10 个时钟才更新一次。用其他的话来讲，在默认下 Act2 的执行优先级高于 Act1 的执行优先级。事实上 Act2 和 Act1 是不存在什么优先级之分，由于 Act2 的时序关系 .... 还是用图形来说话：

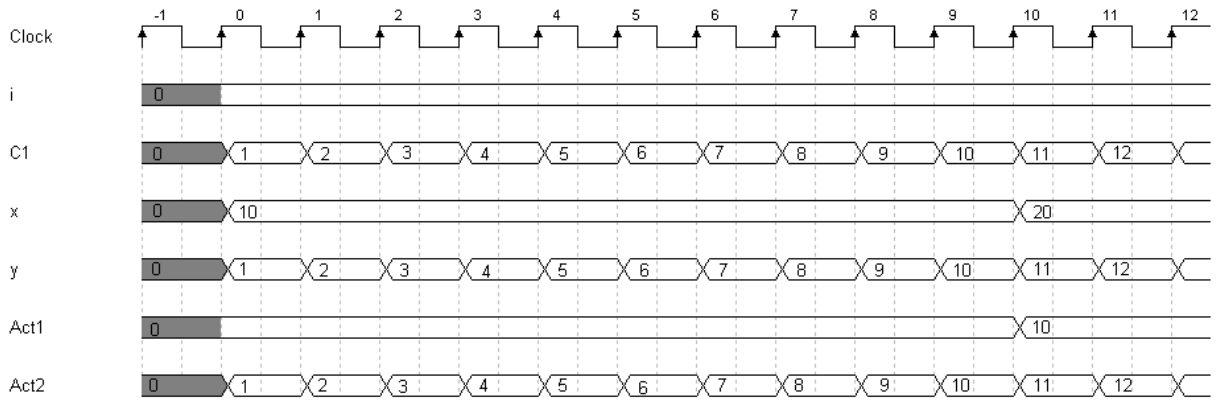


图 1.7f (exp1_7b 前 12 个时钟的经过)

T -1 表示了初始化的状态，T0~T9 是 Act2 的递增经过，直到 T10 的时候 Act1 的 if(x == C1) 的条件成立，然后执行 Act1 <= Act2。在这里 Act1 是执行时间点事件，所以赋予 Act2 是在 T10 时 Act2 的过去值，亦即 10。同样的，如果在 T10 的时候 Act1 触发“即时事件的话”，亦

即 Act1 = Act2，在这里读者需要知道，因为 Act2 在这个“即时空间”中是不存在的，所以 Act1 唯有乖乖的把 Act2 的过去值当着赋值基础。

1.8 步骤的循环和时序的循环之间的区别

当读者看到这里，估计心理一定会憋着许多问题：

假设：

```
for( int x = 0; x < 10; x++ )
{
    for( int y = 0; y < 10; y++ )
        Act1++;
    Act2++;
}
```

那么同样可以用“步骤”的思路在 Verilog HDL 语言中实现它：

```
case( 0 )

    0:
    if( x == 10 ) begin x <= 8'd0; y <= 8'd0; i <= i + 1'b1; end
    else if( y == 10 ) begin x <= x + 1'b1; y <= 8'd0; Act2 <= Act2 + 1'b1; end
    else begin y <= y + 1'b1; Act1 <= Act1 + 1'b1; end
```

再者：

```
for( int x = 0; x < 10; x++ )
{
    Act2++;
    for( int y = 0; y < 10; y++ )
        Act1++;
}
```

同样也可以取“步骤”的概念，用 Verilog HDL 语言来实现：

```
case( 0 )

    0:
    if( x = 10 ) begin x <= 8'd0; i <= i + 4'd2; end
    else begin Act2 <= i + 1'b1; x <= x + 1'b1; i <= i + 1'b1; end

    1:
    if( y == 10 ) begin y <= 8'd0; i <= i - 1'b1; end
    else begin Act1 <= Act1 + 1'b1; y <= y + 1'b1; end;

    2: ......
```

即使是三重的嵌入 for 循环：

```
for( int x = 0; x < 10; x++ )
{
    Act3++;
    for( int y = 0; y < 10; y++ )
    {
        Act2++;
        for( int z = 0; z < 10; z++ )
            Act1++;
    }
}
```

也难不倒 Verilog HDL 语言 ：

```
case( 0 )

    0:
    if( x == 10 ) begin x <= 8'd0; y <= 8'd0; i <= i + 1'b1; end
    else if( y == 10 ) begin x <= x + 1'b1; y <= 8'd0; Act3 <= Act3 + 1'b1; end
    else if( z == 10 ) begin y <= y + 1'b1; z <= 8'd0; Act2 <= Act2+ 1'b1; end
    else begin Act1 <= Act1 + 1'b1; z <= z + 1'b1; end
```

读者终于要爆发了："既然步骤可以实现任何形式的 for 循环，为什么还要研究以时序来实现 for 循环呢？"如果读者还停留在"建模篇"脚步的话，笔者不会责怪你们的，原因很简单~"建模"用其他的话来说就是地基或者结构，它是粗糙，这里的"粗糙"不是物理上的很有"摩擦力"，反之"粗糙"的意思是指"它不在时序上关心"。

换句话说，用步骤实现的 for 循环，在时钟上的消耗，在时序上表现是被隐藏的。如果读者要建立一个效率很高的模块，读者就需要乖乖的把心放在时序上。除此之外，用时序实现的 for 循环，不仅仅是有"重复操作"的功能而已，还有"循环保持"等其他各种各样的功能。但是，最重要的还是"显性指示时钟"。

现在的读者可能听不懂笔者的话中含义，不要紧我们继续话题：

```
for( int x = 0; x < 10; x++ )
{
    Act2++;
    for( int y = 0; y < 10; y ++ )
        Act1 = Act2;
}
```

假设我们继续要使用 Verilog HDL 语言来实现上述的 for 循环，但是不是使用"步骤"也不使用"即时事件"，而且遵守并且照顾好时序的话，那么我们要怎样写呢？在此之前，我们先了解一些简单的原理。

步骤见前页
即时事件见前小节
时序角度见下面分析

```
Act2++;
Act1++;
```

C 语言

```
case( 0 )

    0:
    begin Act2 <= Act2 + 1'b1; i <= i + 1'b1; end

    1:
    begin Act1 <= Act1 + 1'b1; i <= i + 1'b1; end
```

Verilog HDL 语言

如果我们使用 C 语言先执行 Act2++，然后再执行 Act1++，根据 C 语言的性质，我们会以“步骤”来理解，亦即步骤 1 执行 Act2++，然后步骤 2 执行 Act1++。

如果我们使用 Verilog HDL 语言，亦即在时钟 0 执行 Act2 <= Act2 + 1'b1；然后在时钟 1 执行 Act1 <= Act1 + 1'b1；在前面笔者已经说过，如果单个步骤仅使用 1 个时钟的话，那么我们可以说“步骤 i 有指向时钟的功能”，我们先放开“概念”不说，那么余下自然而然只有“时钟”而已。只要我们搞清楚这个简单的原理，那么 Verilog HDL 语言可以这样写：

exp1_8a :

```
case( i )

    0:
    begin
        if( x == C1 ) begin x <= x + 8'd10; Act2 <= ( x != 101 -1 ) ? Act2 + 1'b1 : Act2; end
        if( y +1 == C1 ) begin y <= y + 1'b1; Act1 <= Act2; end

        if( C1 == 101 -1 ) begin x <= 8'd0; y <= 8'd0; i <= i + 1'b1; end
        else C1 <= C1 + 1'b1;
    end
```

在 exp1_7c 实例中， if(C1 == 101 -1) 代表了这一个步骤有 101 个时钟， 然后 -1 表示在 x ，y 和 i 等更新是发生在最后一个时钟。x 控制 Act2 ，y 控制 Act1，x 和 y 均为每一个时钟更新一次。其中的 (x != 101 -1) 表示在前 100 个时钟执行 Act2 + 1'b1，反之就不操作。其中的 (y + 1 == C1) 表示 Act1 的操作延迟一个时钟。

为什么不是 C1 == 100 -1 而是 C1 == 101 -1，亦即步骤 0 有 101 个时钟？原因很简单，在这里 Act1 比起 Act2 慢一个时钟操作，如果只是单单 100 个时钟基本上是不足的。所以必须加上一个延迟所消耗的时钟。乍看下 Act2 的执行优先级比起 Act1 的优先级更高，其实是 Act2 比起 Act1 早一个时钟执行罢了 ...

在这里可能又有同学郁闷 .... x <= x + 8'd10 和 x <= (x != 101 -1) ? Act2 + 1'b1 : Act2;
嗯~(⊙_⊙)还是来仿真图里看看会比较容易介绍。

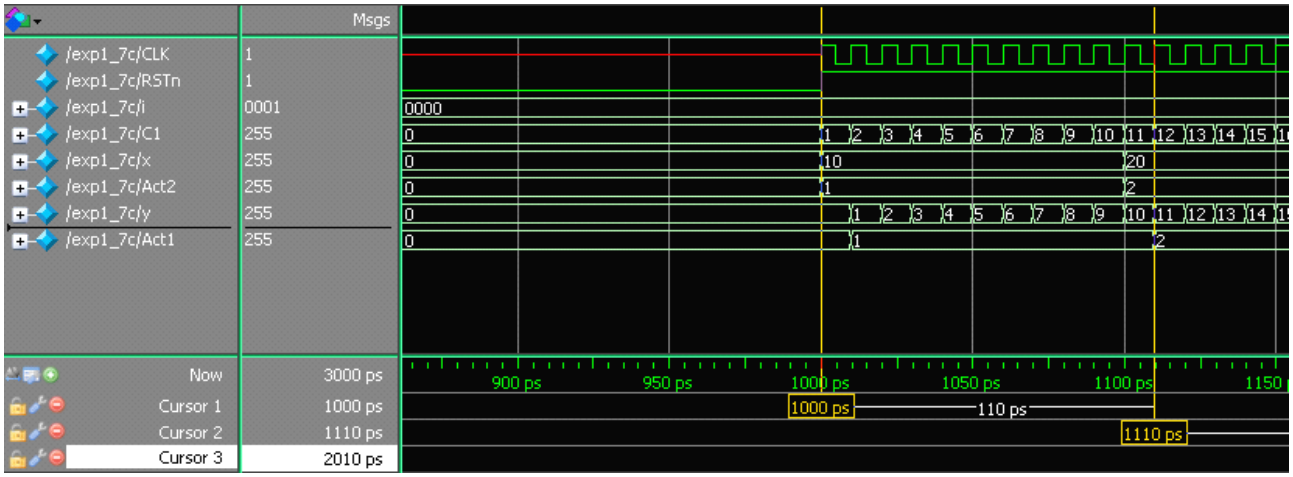


图 1.8a (exp1_8a 仿真结果)

图 1.8a 是仿真结果，其中光标 1~2 之间的距离是时钟 T0~T11。在 T10 的时候 Act2 的条件成立，然后值递增为 2。在另一方面，Act1 在一开始的时候由于被延迟 1 个时钟，所以在 T1 才开始执行。在 T1 的时候 Act1 赋予 Act2 的过去值，亦即 1。

在这里，请读者稍微注意一下：

```
for( int x = 0; x < 10; x++ )
{
    Act2++;
    for( int y = 0; y < 10; y ++ )
        Act1 = Act2;
}
```

上述一段嵌入 for 循环，根据 C 语言的操作会是 ：

1. 递增 Act2。
2. Act1 赋值 10 次 Act2。
3. 重复步骤 1~2 知道 x 大于等于 10。

如果在 Verilog HDL 语言用时序来理解的话 ：

在 T0 的时候 Act2 递增，然后 T1~T10 里 Act1 重复赋值 Act2 的过去值 10 次。在 T10 同一个时候 Act2 再次递增，则 Act1 在 T11~T20 里重复赋值 Act2 的过去值 10 次。以上的执行直到 101 个时钟经过。

嗯，读者不要用困惑的表情面着笔者呀，笔者也知道这样很难理解 ~ 没法子呀，谁叫你们不习惯时序的概念。顶多这样吧！笔者再绘制一段时序图给你们消消气：

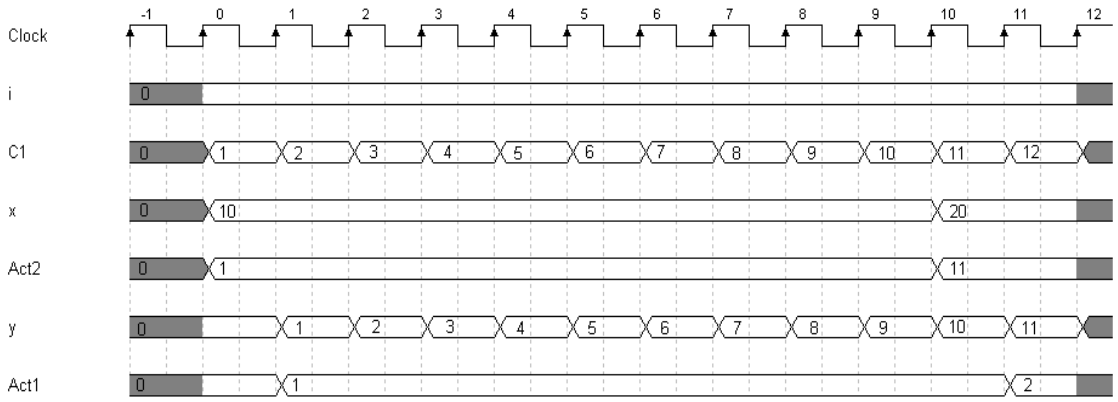


图 1.8b (部分时序图)

笔者把图 1.8a 的 T0~T11 之间的图放大了，自己好好笑消化吧。

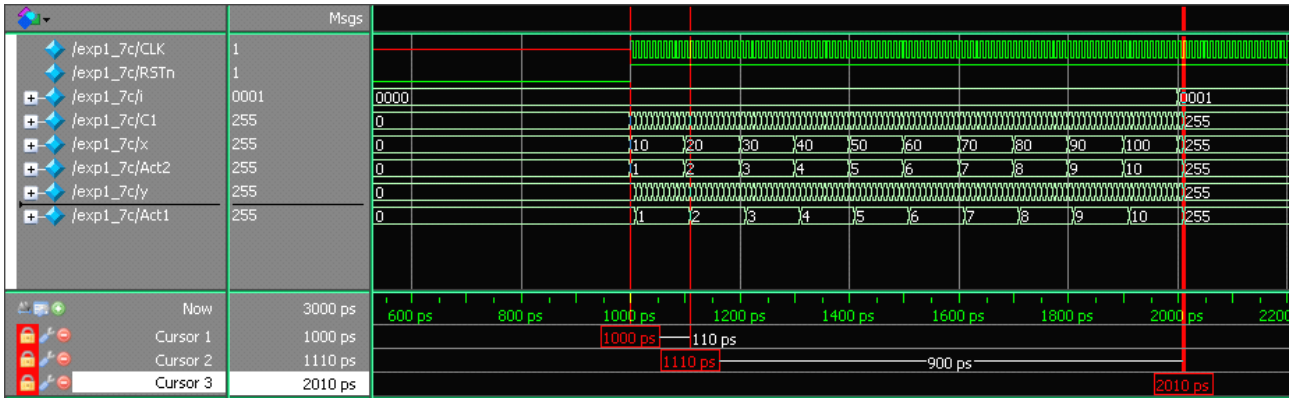


图 1.8c (exp1_8a 仿真结果 101 个时钟的经过，亦即 T0~T100)

图 1.8c 是 101 个时钟的经过（光标 1~3 之间的距离），再次 Act2 一共递增了 10 次，反之 Act1 也更新了 10 次（实际上 Act1 是更新 100 次，但是每隔 10 个时钟 Act1 仅“重复赋值”Act2 的值而已）。在这里，读者是不是明白“为什么步骤 0 需要 101 个时钟”的原因了吧？

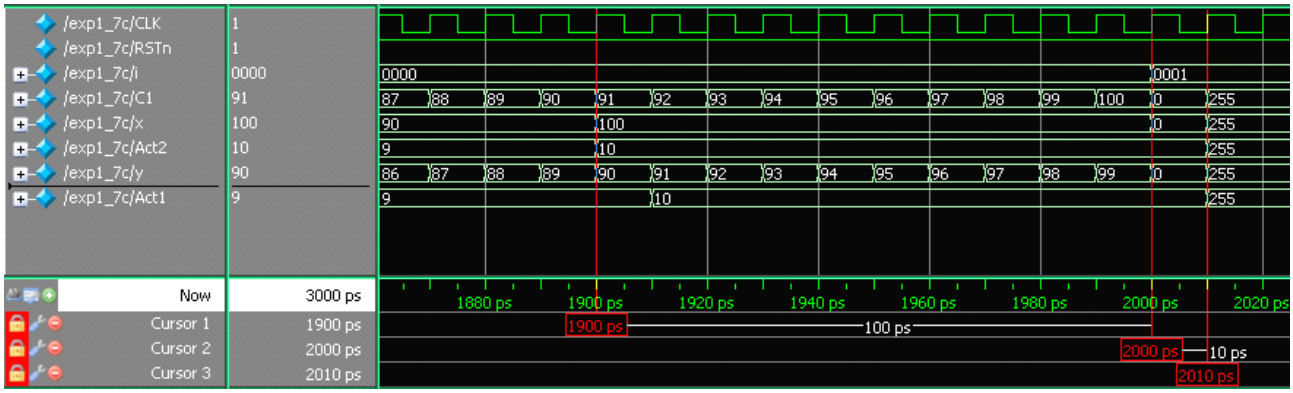


图 1.8d (最后 11 个时钟的经过，亦即 T90~T100)

图 1.8d 是仿真中最后 11 个时钟的经过，已经 T90~T100 之间。还记得 Act2 中有这样的一句表达式吗？ Act2 <= (x != 100 -1) ? Act2 + 1'b1 : Act2; 这是一种扩展目的的表达式，目的是为了阻止 Act2 在 T100 的时候再一次递增，否则就破坏了嵌入 for 循环的仿效了。此外，这

也告诉我们另一个信息，就是我们可以控制 Act2“在什么时候递增？在什么时候停止操作？”这其中对于的控制“动作”起到很好的“扩展作用”。

==========

好吧，烦恼的朋友继续烦恼，忧郁的朋友继续忧郁，郁闷的朋友继续郁闷，但是路子还是要继续。当读者看到这里，是否几乎要爆炸的问：“笔者你从前几章开始就一直有完没完的用 Verilog HDL 语言去实现 for 循环，不是用[步骤]而是用[时序]。以你的性格我们知道笔者一定又在卖关子 .... 行了！算我膜拜大人你，告诉我们真相吧 ... ”

1.9 实验三 ：串口发送模块

(⊙ o ⊙)啊！串口发送模块是建模篇一个经典实例 ... 笔者是不是搞错了？？呵呵，没有没有笔者没有搞错。在这一章节我们需要建立一个经过整合的串口发送模块，至于串口发送的原理笔者就不详谈了，失忆的同学请复习建模篇：

图 1.9a（串口发送模块）

串口发送模块的“图形”还是老样子，这真的激起笔者许多的回忆呀 .... 废话少说，还是切入正题：这个串口发送模块发送一帧 11 位数据，1 个起始位，8 个数据位，1 个校验位，1 个停止位，波特率为 115.2k。以黑金开发板的频率为例，亦即 20Mhz。20Mhz 的频率为 115.2k 计数的话，结果是：

```
(1/115.2k) / (1/20Mhz)
= ( 1/ 115200) / ( 1 / 20E+6 )
=173.61xx
=174
```

当拉高 TX_En_Sig 的话，串口发送模块就开始发送数据，直到完成就反馈完成信号。

tx_module.v

```
module tx_module
(
        input CLK,
        input RSTn,

        input TX_En_Sig,
        output TX_Done_Sig,

        input [7:0]TX_Data,
        output TX_Pin_Out

);

      /******************************/
```

```
15.
16.         // formula : ( 1/115200 )/( 1/20E+6 )
17.
18.         parameter B115K2 = 8'd174;
19.
20.         /******************************/
21.
22.         reg [3:0]i;
23.         reg [7:0]C1;
24.         reg [10:0]rData;
25.         reg rPin;
26.         reg isDone;
27.
28.         always @( posedge CLK or negedge RSTn )
29.               if( !RSTn )
30.                     begin
31.                             i <= 4'd0;
32.                             C1 <= 8'd0;
33.                             rData <= 11'd0;
34.                             rPin <= 1'b1;
35.                             isDone <= 1'b0;
36.                     end
37.               else if( TX_En_Sig )
38.                     case( i )
39.
40.                             0:
41.                             begin rData <= { 2'b11 , TX_Data , 1'b0 }; i <= i + 1'b1; end
42.
43.                             1,2,3,4,5,6,7,8,9,10,11:
44.                             if( C1 == B115K2 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
45.                             else begin rPin <= rData[i - 1]; C1 <= C1 + 1'b1; end
46.
47.                             12:
48.                             begin isDone <= 1'b1; i <= i + 1'b1; end
49.
50.                             13:
51.                             begin isDone <= 1'b0; i <= 4'd0; end
52.
53.                     endcase
54.
55.         /*********************************/
56.
57.         assign TX_Pin_Out = rPin;
58.         assign TX_Done_Sig = isDone;
59.
60.         /*********************************/
```

```
61.
62.
63.   endmodule
```

上面是经过整合以后 tx_module.v 的源码。步骤 0(40~41 行)是初始化的操作,亦即初始 rData 的值，这里 rData <= {2'b11 , TX_Data , 1'b0} 表示 [0]起始位；[1:8]数据位；[9]校验位；[10]停止位。步骤 1~11（43~45 行），是串口发送一帧数据的过程，每一个步骤都使用 174 个时钟。步骤 12~13（47~51 行）是产生完成信号的步骤。

根据计算，串口发送模块发送一帧数据需要消耗 n 个时钟：

1 个时钟初始化
11 * 174 个时钟发送一帧数据
2 个时钟完成信号

亦即，1 + (11 * 174) + 2 = 1917 个时钟。如果以 20Mhz 的时钟频率，一个时钟的周期是 50ns, 结果一个串口发送模块每一次启动需要消耗 1917 * 50ns = 95.85us。

tx_module.vt

```
1.    `timescale 1 ns/ 1 ns
2.    module tx_module_simulaion();
3.
4.          reg CLK;
5.          reg RSTn;
6.          reg TX_En_Sig;
7.          reg [7:0]TX_Data;
8.
9.          wire TX_Done_Sig;
10.         wire TX_Pin_Out;
11.
12.         /***************/
13.
14.         /**********************************/
15.
16.         tx_module U1
17.         (
18.                 .CLK(CLK),
19.                 .RSTn(RSTn),
20.                 .TX_En_Sig(TX_En_Sig),
21.                 .TX_Data(TX_Data),
22.                 .TX_Done_Sig(TX_Done_Sig),
23.                 .TX_Pin_Out(TX_Pin_Out)
24.         );
25.
```

```
26.         /**********************************/
27.
28.         initial
29.         begin
30.                 RSTn = 0; #1000 RSTn = 1;
31.                 CLK = 1; forever #25 CLK = ~CLK;
32.         end
33.
34.        /***********************************/
35.
36.          reg [3:0]i;
37.
38.          always @ ( posedge CLK or negedge RSTn )
39.              if( !RSTn )
40.                  begin
41.                          TX_En_Sig <= 1'b0;
42.                          TX_Data <= 8'd0;
43.                          i <= 4'd0;
44.                  end
45.              else
46.                  case( i )
47.
48.                          0:
49.                          if( TX_Done_Sig ) begin TX_En_Sig <= 1'b0; i <= i + 1'b1; end
50.                          else begin TX_Data <= 8'b01010101; TX_En_Sig <= 1'b1; end
51.
52.                          1:
53.                          i <= i;
54.
55.                  endcase
56.
57.  endmodule
```

这个是激励文本，第 1 行是仿真的时钟单位，亦即时间刻度，在这里就填上 1ns。第 16~24 行是实例化仿真对象。第 28~32 行是复位信号和时钟信号的产生和刺激，笔者近似的模仿黑金开发板的环境。也就是说复位时间是 1us 时钟周期是 50ns。步骤 0（48~50 行）是启动串口发送模块发送一帧为 8'b01010101 的数据。步骤 1（52~53 行）是停留 ....（有点像单片机中的 while(1);）

仿真结果：

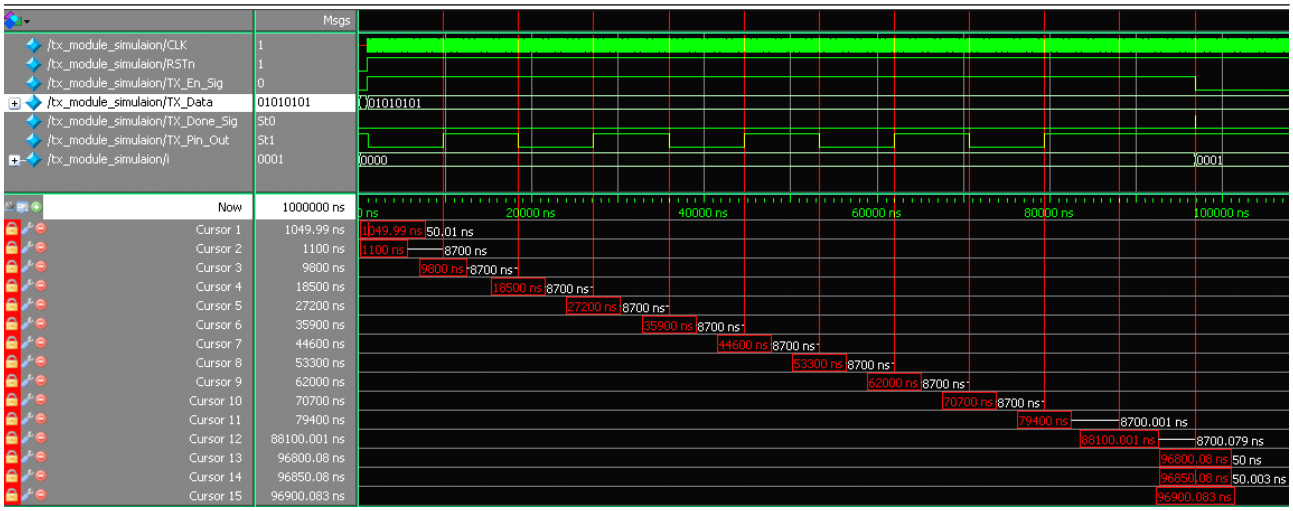


图 1.9a (串口发送模块的仿真结果)

图 1.9a 是串口发送模块的仿真结果，光标 1~15 之间的距离是串口所消耗的时间，亦即：

96900.083ns - 1049.99ns = 96900 - 1050 = 95850ns , 95.85us。

光标 1~15 之间的距离是 95.85us ，这个和之前评估的结果一模一样 ... ⊙﹏⊙b 汗！我们再把时间转换为时钟，那么 95850ns / 50ns = 1917 个时钟。读者是不是觉得，很惊讶！？这就是整合化得功能 .... 可以非常清晰的指向时钟。

1.10 实验四 ：串口发送模块 2

在试验三中我们用了“循环保留”的方法使串口发送模块来产生的一帧信息，在实验四里笔者就是用“循环操作”的方法是串口发送模块运作。

图 1.8a (实验四的串口发送模块)

图 1.8a 是实验四要建立的串口发送模块。看看这个模块和试验三的相比，到底有什么不同？对就是缺少了 Done_Sig 的控制信号。可能有些读者看到这里就急起来了，问道：“如果没有控制信号，那么咱要如何控制这个模块呢？”在前面笔者已经说过了，就是因为模块的消耗时钟无法被评估的原因，所以我们才需要使用 Done_Sig 控制信号来反馈模块的完成。

反之，如果某个模块的时钟消耗个数可以被评估的话“模块的反馈 Done_Sig 就不是可以不要呢？”呵呵呵~这就是这章节的试验目的。丑话当前，这一章节的实验需要大量有关“时序概念”的基础知识，不懂的朋友需要最好先去复习先, 以免看不懂笔者在说什么 ...

tx_module2.v

```
1.    module tx_module2
2.    (
3.            input CLK,
4.            input RSTn,
5.
6.            input TX_En_Sig,
7.            input [7:0]TX_Data,
8.
9.            output TX_Pin_Out,
10.
11.           /**********************/
12.
13.           output [10:0]SQ_C1
14.   );
15.
16.         /**************************/
17.
18.           reg [3:0]i;
19.           reg [10:0]rData;
20.           reg [10:0]C1;
```

```
21.         reg [10:0]x;
22.         reg rPin;
23.
24.         always @ ( posedge CLK or negedge RSTn )
25.             if( !RSTn )
26.                 begin
27.                         i <= 4'd0;
28.                         rData <= 11'd0;
29.                         C1 <= 11'd0;
30.                         x <= 11'd0;
31.                         rPin <= 1'b1;
32.                 end
33.             else if( TX_En_Sig )
34.                 begin
35.
36.                         rData = { 2'b11 , TX_Data, 1'b0 };
37.                         if( x == C1 ) begin x <= x + 11'd174; rPin <= rData[i]; i <= i + 1'b1; end
38.                         if( C1 == 1914 -1 ) begin x <= 11'd0; i <= 4'd0; C1 <= 11'd0; end
39.                         else C1 <= C1 + 1'b1;
40.
41.                     end
42.
43.         /****************************/
44.
45.         assign TX_Pin_Out = rPin;
46.
47.         /****************************/
48.
49.         assign SQ_C1 = C1;
50.
51.  endmodule
```

呵呵呵，估计有些朋友又要学马骝抓头皮了 .... 这一回的串口发送模块，笔者使用“循环操作”的方式来书写。第 3~13 行是该模块的输入输出，至于第 13 行的 SQ_C1 是仿真用的。第 18~22 行是串口发送模块所使用的相关寄存器，第 20 行的 C1 是用来计数时钟，第 21 行的 x 是用来控制循环操作的次数。

在这里需要稍微注意一下，有关第 18 行的 i，由于笔者蛋疼了所以这回就不使用步骤，在此 i 只是简单的计数器，作用是用来寻址下标。第 24~41 行是该模块的主要功能。第 25~32 行复位了相关的寄存器。

在 33 行，当 if 判断到 TX_En_Sig 拉高的时候，这个模块就开始工作。在 38 行的 if(C1 == 1914 -1) 表示了这个模块需要消耗 1914 个时钟，“-1”的意思是 begin x <= 11'd0; i <= 4'd0; C1 <= 11'd0; 等操作是发生在 1914 个时钟内的最后一个时钟。第 37 行的 if(x == C1) 表示了 x 寄存器用来控制循环操作，则 x <= x + 174 表示每一个循环操作的间隔是 174 个时钟。

第 36 行的 rData = { 2'b11 , TX_Data, 1'b0 }; 是用来更新 rData 寄存器，此外 rData 也表示了这个串口发送模块的帧信息的配置，亦即 [0]起始位 [8:1]数据位 [9]校验位 [10]停止位，也就是说一帧有 11 个位（校验位可以随便填）。嗯~估计有同学又要郁闷起来了，因为 rData 是使用阻塞赋值操作符 "="，原因很简单-就是为了取得即时结果，用其他的话来说是为了节省 rData 更新时所消耗的一个时钟。第 37 行的 i 是用来计数，也作为 rData 的下标寻址。

笔者先简单的用文字来描述这个模块执行的经过：

当 TX_En_Sig 拉高的时候，就以 T0 开始吧，在这个瞬间 rData 以举得即时结果，假设 TX_Data 为 8'b0101_0101，那么 rData 的内容是 { 2'b11, 8'b0101_0101, 1'b0 }。在 T0 的时候，C1 开始计数，然后 if(x == C1)的条件成立， 所以 x <= x + 174 操作成立，然后 rPin 也成功赋予 rData[0] 亦即 [0]停止位。此外 i 也成功递增。所以在 T0 的未来，x 是 174，rPin 是 0，i 是 1。

在 T174 的时候，f(x == C1)的条件成立， 所以 x <= x + 174 操作成立，然后 rPin 也成功赋予 rData[1] 亦即 [1]数据位的最低位。此外 i 也成功递增。 所以在 T174 的未来，x 是 348，rPin 是 1，i 是 2。

在 T1384 的时候，f(x == C1)的条件成立， 所以 x <= x + 174 操作成立，然后 rPin 也成功赋予 rData[2] 亦即 [2]数据位的第 1 位。此外 i 也成功递增。 所以在 T384 的未来，x 是 522，rPin 是 0，i 是 3。

在 T522 的时候，f(x == C1)的条件成立， 所以 x <= x + 174 操作成立，然后 rPin 也成功赋予 rData[3] 亦即 [3]数据位的第 2 位。此外 i 也成功递增。 所以在 T522 的未来，x 是 696，rPin 是 1，i 是 4。

在 T696 的时候，f(x == C1)的条件成立， 所以 x <= x + 174 操作成立，然后 rPin 也成功赋予 rData[4] 亦即 [4]数据位的第 3 位。此外 i 也成功递增。 所以在 T696 的未来，x 是 870，rPin 是 0，i 是 5。

在 T870 的时候，f(x == C1)的条件成立， 所以 x <= x + 174 操作成立，然后 rPin 也成功赋予 rData[5] 亦即 [5]数据位的第 4 位。此外 i 也成功递增。 所以在 T870 的未来，x 是 1044，rPin 是 1，i 是 6。

在 T1044 的时候，f(x == C1)的条件成立， 所以 x <= x + 174 操作成立，然后 rPin 也成功赋予 rData[6] 亦即 [6]数据位的第 5 位。此外 i 也成功递增。 所以在 T1044 的未来，x 是 1218，rPin 是 0，i 是 7。

在 T1218 的时候，f(x == C1)的条件成立， 所以 x <= x + 174 操作成立，然后 rPin 也成功赋予 rData[7] 亦即 [7]数据位的第 6 位。此外 i 也成功递增。 所以在 T1218 的未来，x 是 1392，rPin 是 1，i 是 8。

在 T1392 的时候，f(x == C1)的条件成立， 所以 x <= x + 174 操作成立，然后 rPin 也成功赋予 rData[8] 亦即 [8]数据位的第 7 位。此外 i 也成功递增。 所以在 T1392 的未来，x 是 1566，rPin 是 0，i 是 9。

在 T1566 的时候，f(x == C1)的条件成立， 所以 x <= x + 174 操作成立，然后 rPin 也成功赋予 rData[9] 亦即 [9]校验位。此外 i 也成功递增。 所以在 T1566 的未来，x 是 1740，rPin 是 1，i 是 10。

在 T1740 的时候，f(x == C1)的条件成立， 所以 x <= x + 174 操作成立，然后 rPin 也成功赋予 rData[10] 亦即 [10]停止位。此外 i 也成功递增。 所以在 T1740 的未来，x 是 1914，rPin 是 1，i 是 11。注意呀，从 T1740 之后 x 和 i 就没有作为了。

rPin 会拉高直到时钟的 T1914，在期间 T1913，C1 , x 和 i 寄存器都会清零，那么一帧数据已经发送完毕。

tx_module2.vt

```
1.  `timescale 1 ns/ 1 ns
2.  module tx_module2_simulation();
3.
4.      reg CLK;
5.      reg RSTn;
6.
7.      reg TX_En_Sig;
8.      reg [7:0]TX_Data;
9.
10.     wire TX_Pin_Out;
11.
12.     /***********************/
13.
14.     wire [10:0]SQ_C1;
15.
16.     /***********************/
17.
18.     tx_module2 U1
19.     (
20.         .CLK(CLK),
21.         .RSTn(RSTn),
22.         .TX_En_Sig(TX_En_Sig),
23.         .TX_Data(TX_Data),
24.         .TX_Pin_Out(TX_Pin_Out),
25.         .SQ_C1( SQ_C1 )
26.     );
27.
28.     /***********************/
29.
30.     initial
31.     begin
32.         RSTn = 0; #1000 RSTn = 1;
```

```
33.                 CLK = 1; forever #25 CLK = ~CLK;
34.         end
35.
36.       /**********************/
37.
38.         reg [3:0]i;
39.         reg [10:0]C1;
40.
41.       always @ ( posedge CLK or negedge RSTn )
42.             if( !RSTn )
43.                     begin
44.                             i <= 4'd0;
45.                             TX_En_Sig <= 1'b0;
46.                             TX_Data <= 8'd0;
47.                             C1 <= 11'd0;
48.                     end
49.                 else
50.                     case( i )
51.
52.                             0:
53.                             if( C1 == 1914 -1 ) begin C1 <= 11'd0; i <= i + 1'b1; end
54.                             else begin C1 <= C1 + 1'b1; TX_En_Sig <= 1'b1; TX_Data <= 8'b10101010; end
55.
56.                             1:
57.                             begin TX_En_Sig <= 1'b0; i <= i + 1'b1; end
58.
59.                             2:
60.                             if( C1 == 1914 -1 ) begin C1 <= 11'd0; i <= i + 1'b1; end
61.                             else begin C1 <= C1 + 1'b1; TX_En_Sig <= 1'b1; TX_Data <= 8'b01010101; end
62.
63.                             3:
64.                             if( C1 == 1914 -1 ) begin C1 <= 11'd0; i <= i + 1'b1; end
65.                             else begin C1 <= C1 + 1'b1; TX_En_Sig <= 1'b1; TX_Data <= 8'b01111110; end
66.
67.                             4:
68.                             begin TX_En_Sig <= 1'b0; i <= i + 1'b1; end
69.
70.                             5:
71.                             i <= i;
72.
73.                     endcase
74.
75.         /**********************/
76.
77.   endmodule
```

第 50~71 行是 .vt 文件对于 .v 文件的激励过程 .... 读者是不是觉得类似的写法很眼熟呢？读者的多虑是正确的，类似的写法在 FIFO 的调用过程已经无数次得出现过。

```
0: begin Write_Req_Sig <= 1'b1; FIFO_Write_Data <= 8'hff; i <= i + 1'b1; end
1: begin Write_Req_Sig <= 1'b0; i <= i + 1'b1; end
```

上述一段代码算是经典了，代码的作为就是往 FIFO 写入一个深度的数据。当我们想 FIFO 写入数据，那么就会产生“模块的沟通”。FIFO 每次存入一个数据都需要消耗一个始终，那么从步骤 0 到步骤 1 之间的经过恰恰好就是给足了 FIFO 一个时钟可以消耗的时钟。

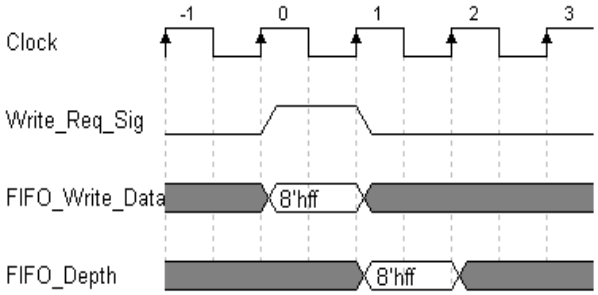


图 1.10a (向 FIFO 写入一个深度数据)

图 1.10a 是向 FIFO 写入一个深度数据的经过。在 T0 的时候我们拉高 Write_Req_Sig 和向 FIFO_Write_Data 写入 8'hff，在 T1 的时候我们拉低 Write_Req_Sig。在 T1 的时候 FIFO 就会判断 Write_Req_Sig 的过去值，亦即拉高，然后 FIFO 会读入 FIFO_Write_Data 的过去值，亦即 8'hff。所以在 T1 的未来 8'hff 会被 FIFO 存入。

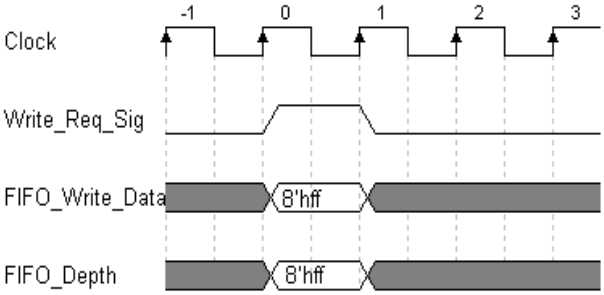


图 1.10b（模块之间的沟通没有延迟的话 ... ）

根据“模块沟通”的原理，模块之间的沟通至少需要一个时钟的延迟 ... 如果在“模块之间没有任何延迟的话 .... ”结果就会像图 1.10b 那样 .... 我们在 T0 拉高 Write_Req_Sig 和写入数据 8'hff，那么 FIFO 就会在瞬间存入数据 .... 事实上这是不可能会发生的！！！

```
0: begin Write_Req_Sig <= 1'b1; FIFO_Write_Data <= 8'hff; i <= i + 1'b1; end
1: begin Write_Req_Sig <= 1'b0; i <= i + 1'b1; end
2: begin Write_Req_Sig <= 1'b1; FIFO_Write_Data <= 8'hee; i <= i + 1'b1; end
3: begin Write_Req_Sig <= 1'b0; i <= i + 1'b1; end
```

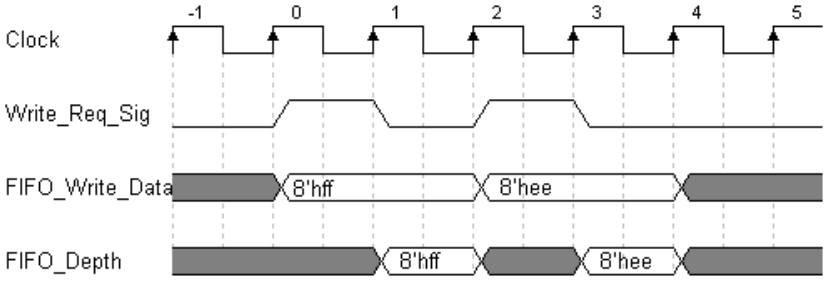


图 1.10c (一个深度一个深度的写入数据)

假设我们一个深度一个深度的向 FIFO 写入数据的话，结果会是如图 1.10c

```
0: begin Write_Req_Sig <= 1'b1; FIFO_Write_Data <= 8'hff; i <= i + 1'b1; end
1: begin Write_Req_Sig <= 1'b1;FIFO_Write_Data <= 8'hee; i <= i + 1'b1; end
2: begin Write_Req_Sig <= 1'b0; i <= i + 1'b1; end
```

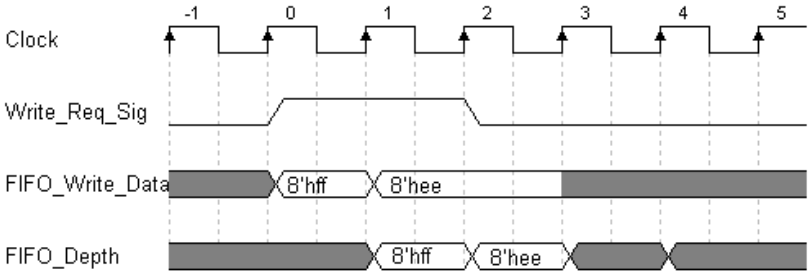


图 1.10d（连续写入数据）

如果向 FIFO 连续写入数据的话，结果会是如同图 1.10d 的经过。嗯~初学的同学笔者还是比较推荐使用一个深度一个深度的写入数据 .... 至此笔者依然还是喜爱一个深度一个深度的方法~解读能力高吗~ (*^__^*)。

如果向 FIFO 写入一个深度的数据需要消耗一个时钟的话，既然我们都已经知道 tx_module2.v 启动一次需要消耗消耗 1914 个时钟的话 ... 同样的道理，我们可以使用循环保持的方法，为 TX_En_Sig 和 TX_Data 保留 1914 个时钟不就可以了吗？每当 1914 个时钟经过，再来拉低 TX_En_Sig, 那么一次性的启动和控制串口发送模块就完成了。所以第 52~57 行（步骤 0~1）就是这么一回事。第 61~70（步骤）行是连续启动串口发送模块，至于理解的方式和连续向 FIFO 写入数据一样。

在代码的理解上就是这么容易，但是要在时序图中看明白，还真需要几分细心 ... 对于笔者来说更需要几分的耐心，把所有重点都表达出来 ...

仿真结果：

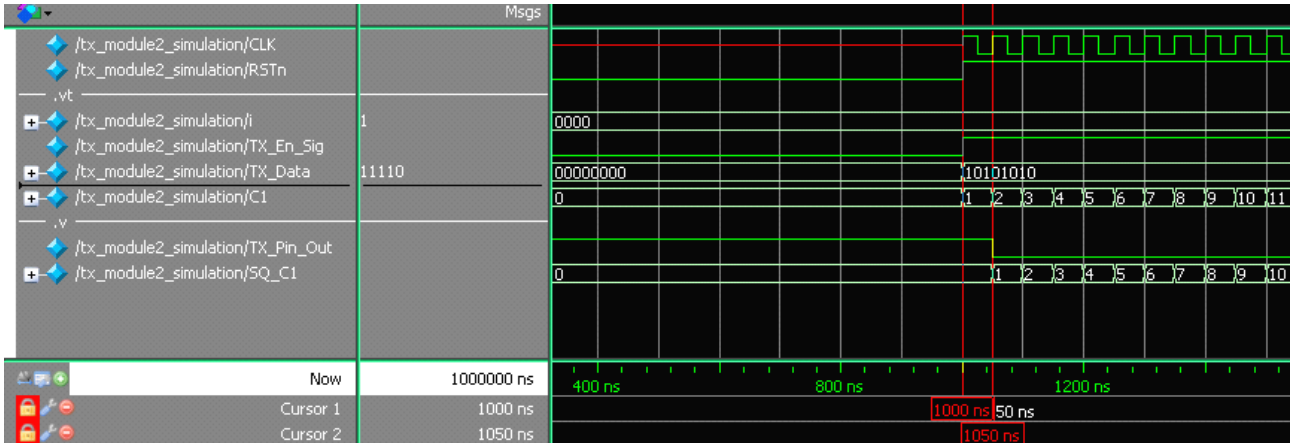


图 1.10e（仿真的开始）

图 1.10e 是 tx_module.v 的仿真初头，根据串口的协议串口发送端在闲置的状态下输出始终拉高(请注意 TX_Pin_Out)。光标 1 是指着 T0，在 T0 的时候 .vt 文件开始拉高 TX_En_Sig，然后在 T0~T1 之间（光标 1~2 之间），就是 .vt 和 .v 模块之间沟通的延迟。

在 T1 的时候（光标 2）.v 文件检查到 TX_En_Sig 的过去值是 1，所以“决定”输出 rData 的第 0 位，亦即起始位~1'b0，所以在 T1 的未来 TX_Pin_Out 开始拉低输出。

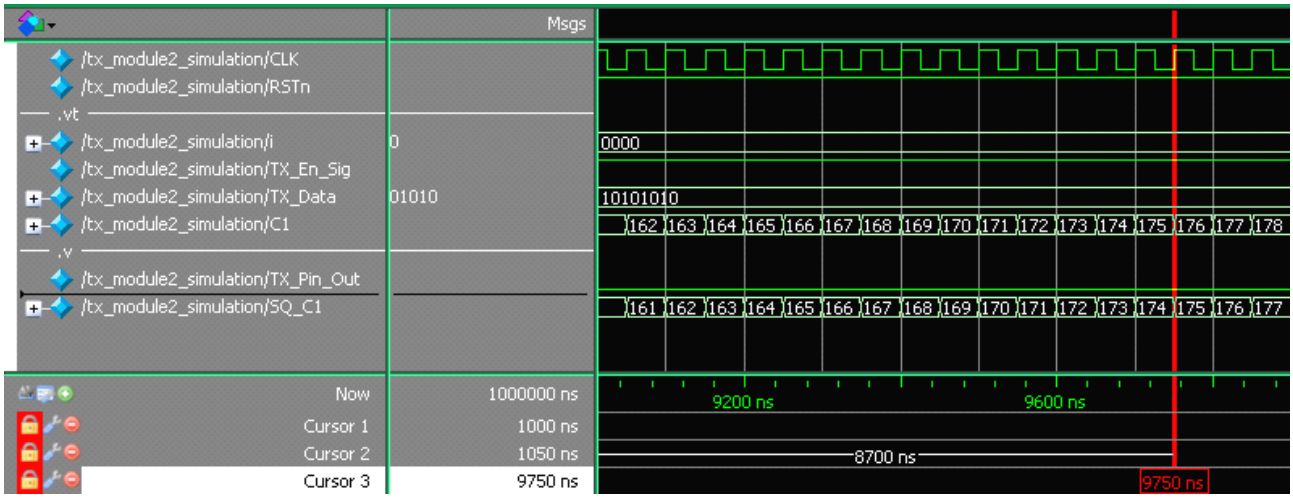


图 1.10f (第一帧第 0 位的输出)

图 1.10f 是第一帧第 0 位保持的时间（始终），在这里请注意一下 C1 和 SQ_C1 之间的比较。C1 表示了 .vt 的始终计数，SQ_C1 表示了 .v 文件的计数。根据 20Mhz 对 115.2k 波特率的计数，一帧的一位数据保留大约 174 个时钟。当 SQ_C1 计数到 174，亦即第一帧的第 0 位数据保持时间已经结束。C2 和 C3 之间的距离是第一帧第 0 位数据所保留的时间，8700ns 如果换做时钟数的话，结果是：

8700ns / 20Mhz 一个时钟的周期：
8700ns / 50ns = 174

由于“模块沟通”所产生一个时钟的延迟，所以 C1 计数到 175，反之 SQ_C1 计数到 174。

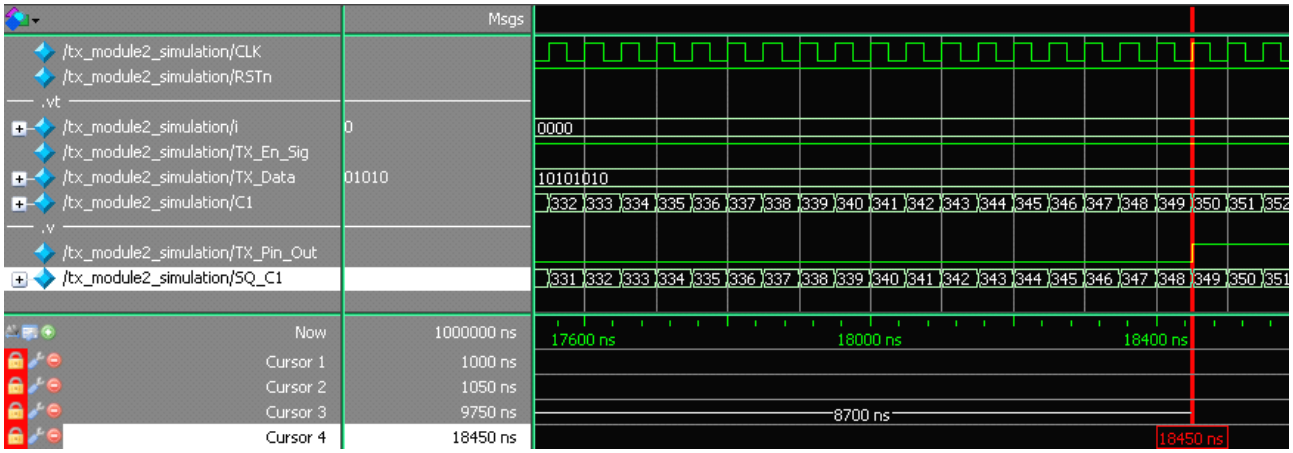


图 1.10g（第一帧第 1 位的输出）

光标 3~4 之间是第一帧第 1 位的输出，亦即数据位第 0 位。注意 C1 和 SQ_C1 之间的比较，依然是延迟了一个时钟。

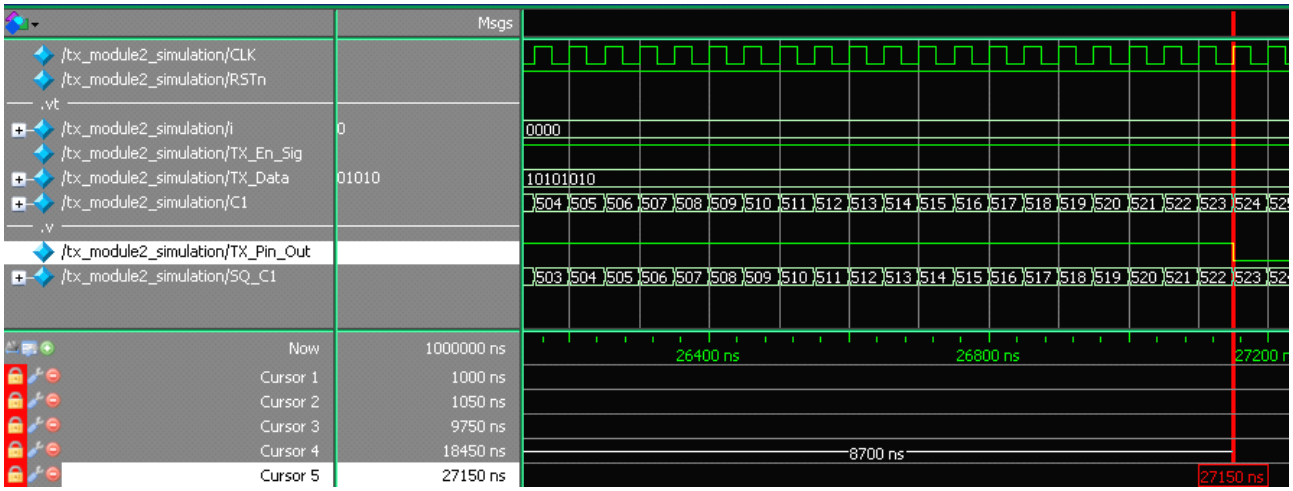


图 1.10h（第一帧第 2 位的输出）

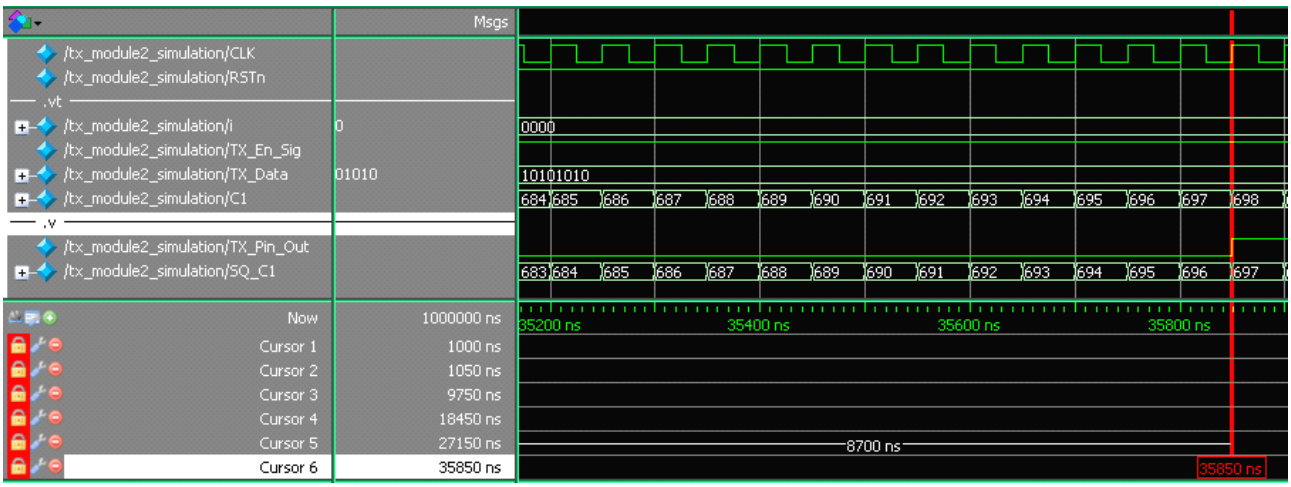


图 1.10i（第一帧第 3 位的输出）

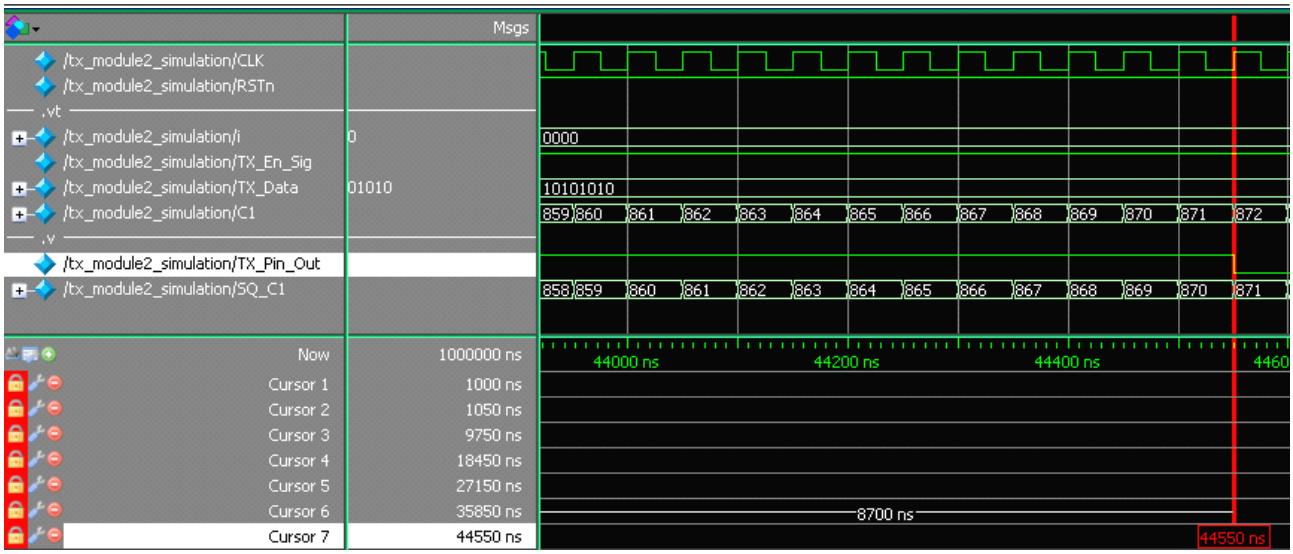


图 1.10j（第一帧第 4 位的输出）

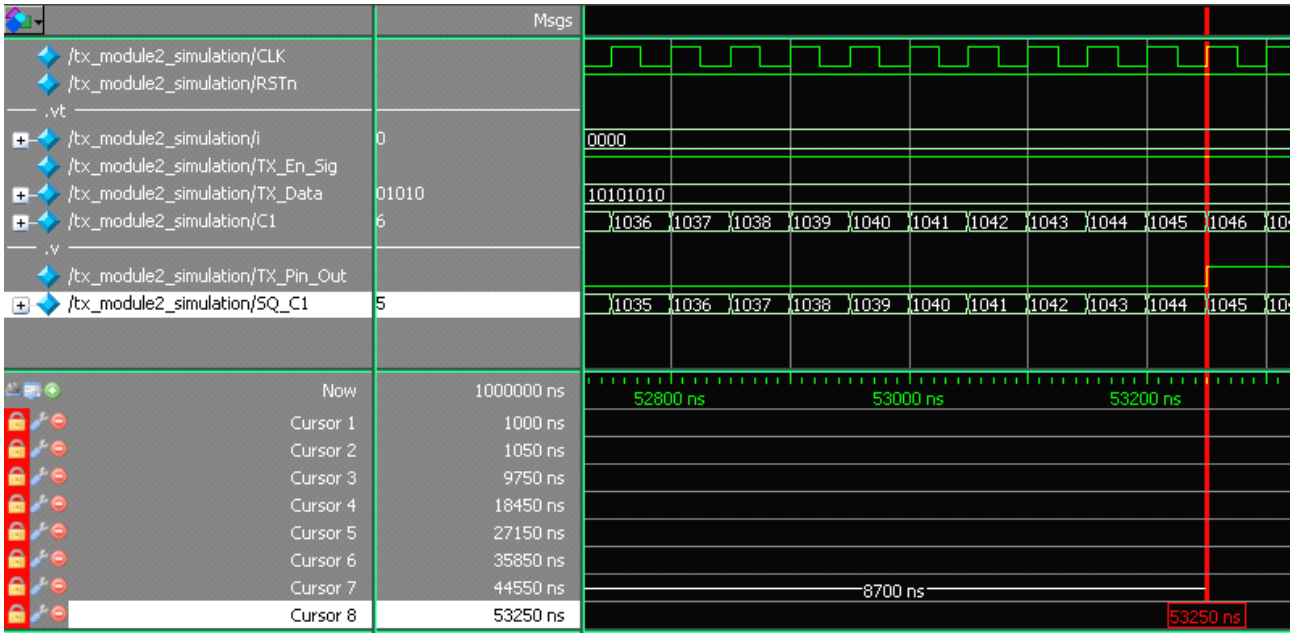


图 1.10k（第一帧第 5 位的输出）

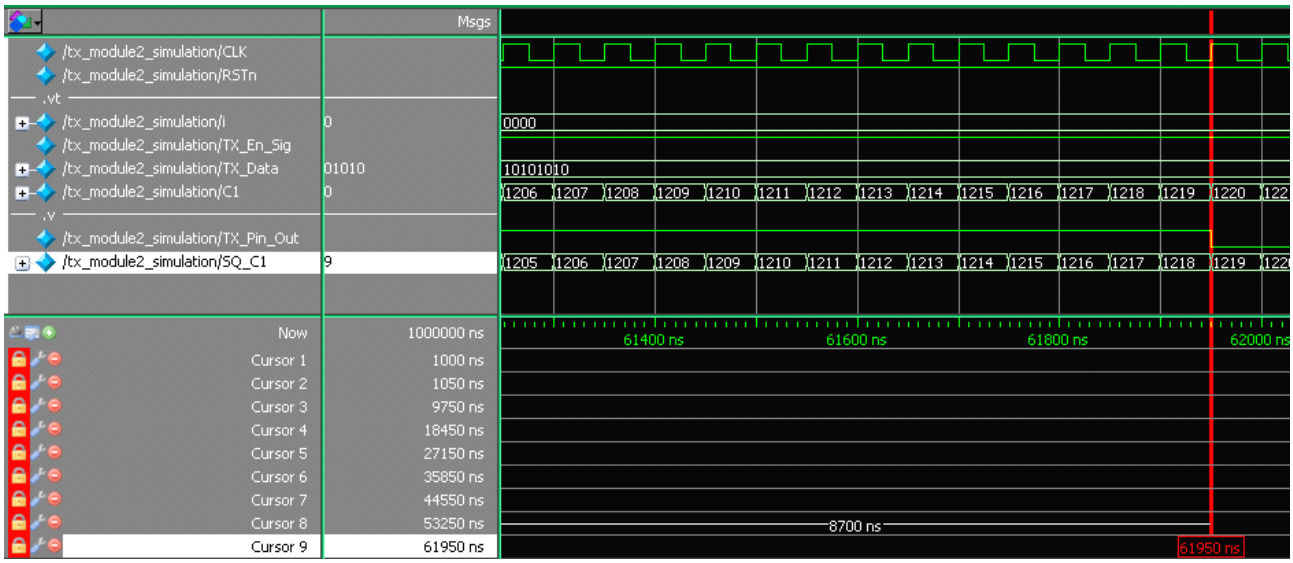


图 1.10l（第一帧第 6 位的输出）

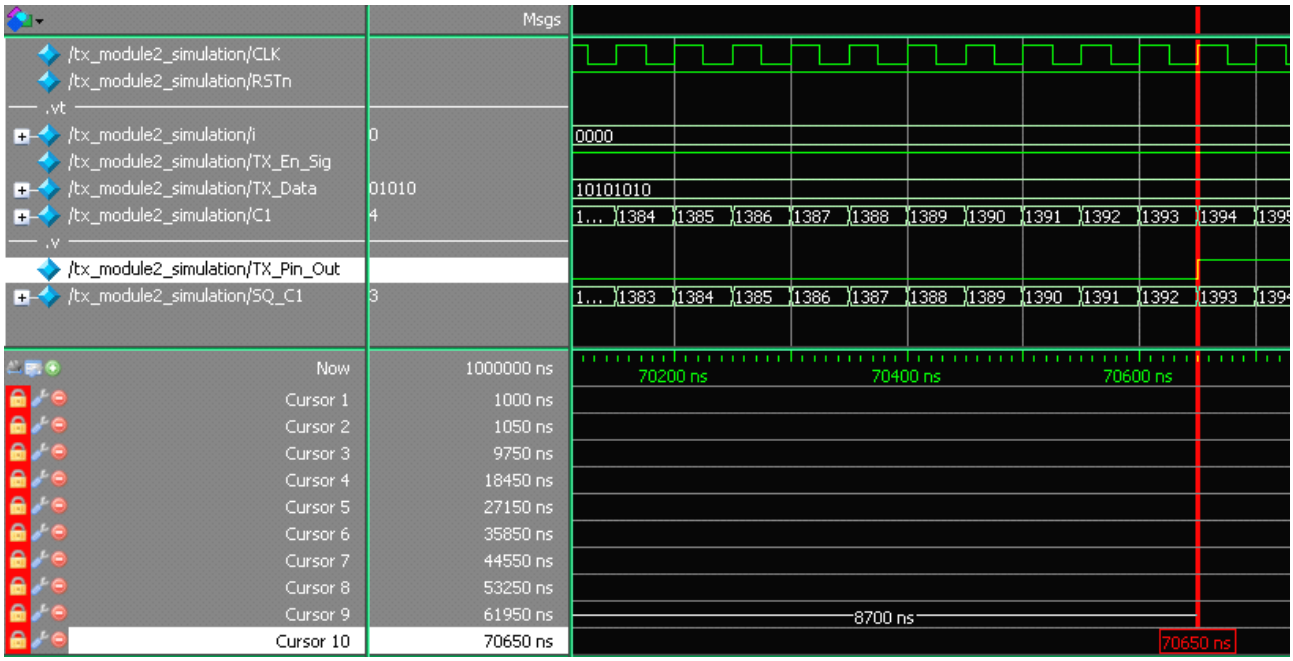


图 1.10m（第一帧第 7 位的输出）

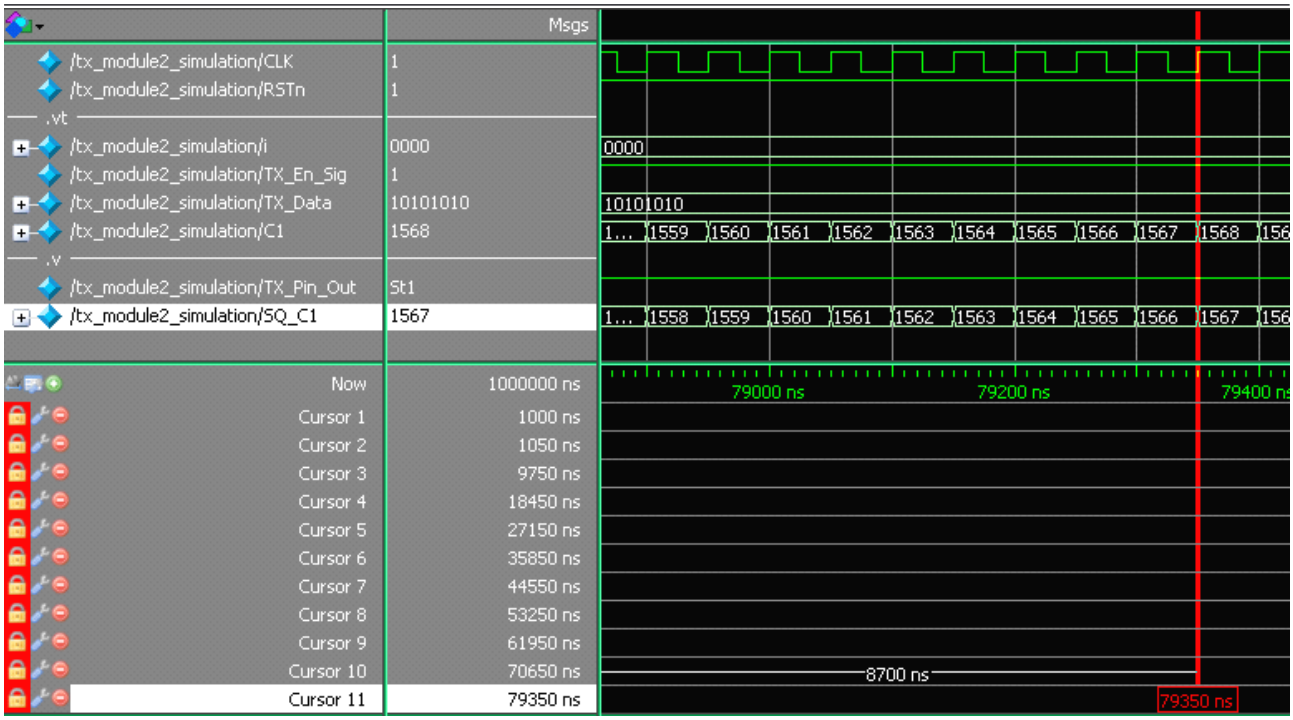


图 1.10n（第一帧第 8 位的输出）

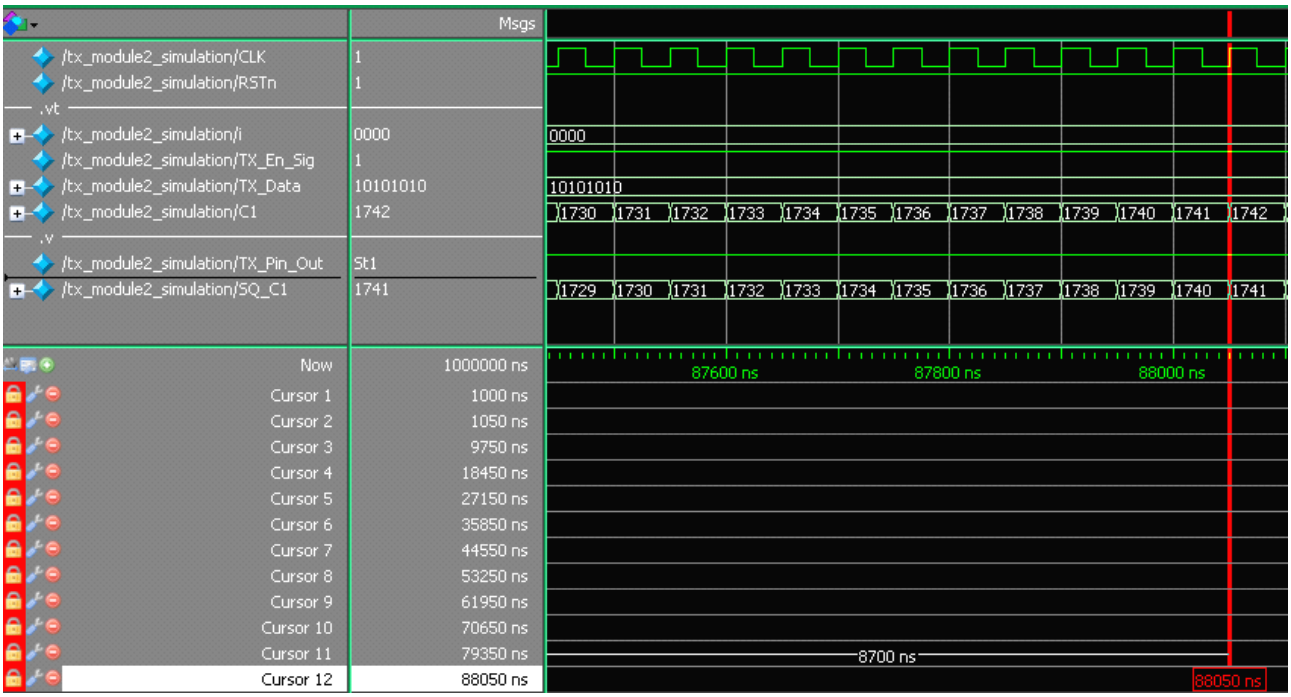


图 1.10o（第一帧第 9 位的输出）

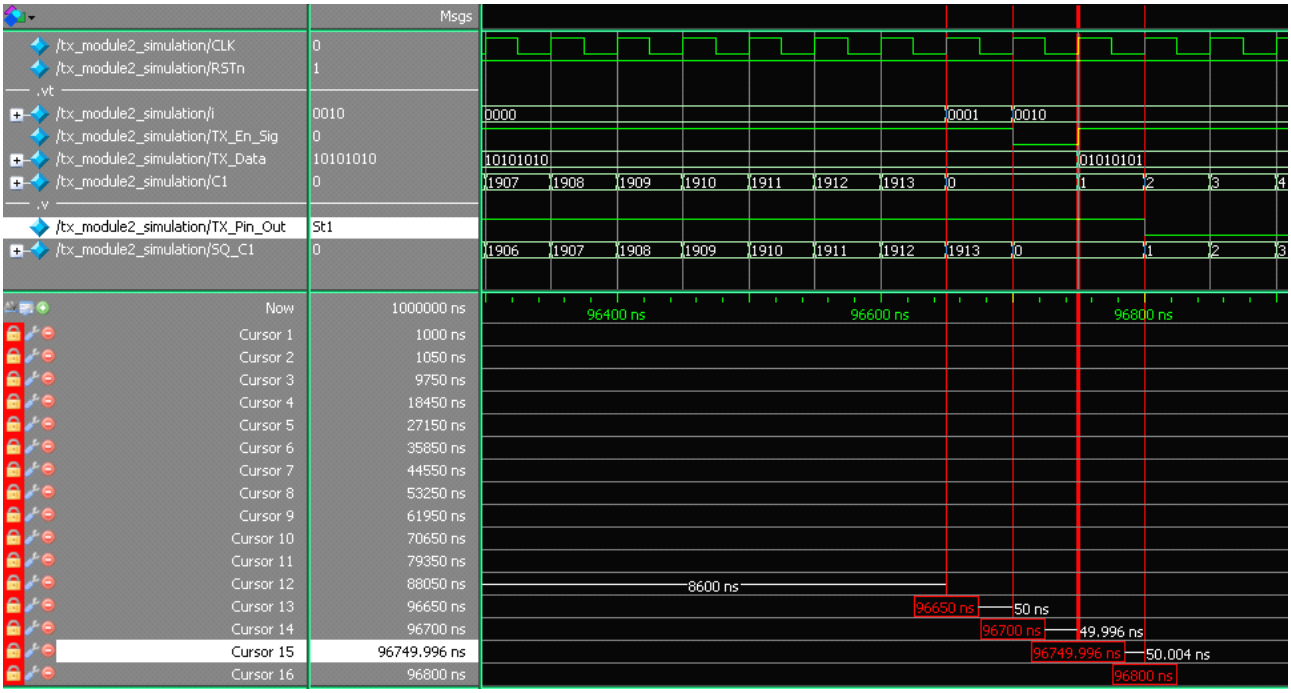


图 1.10p（第一帧第 10 位的输出）

图 1.10p 是第一帧数据最后一位的输出，其中光标 12~15 是最后一位的保留时间。嗯~是不是觉得有点怪呢？咕~~(ᐟ ___ ᐠ)b ... 请读者不要忘了，无论是循环保留还是循环操作，笔者都习惯在最后一个时钟之前，将相关的寄存器都清理（C1 ==1914 -1)。这时候，我们需要把 .vt 和 .v 分开来看。

光标 12~13 是 .vt 中步骤 0 的最后一个时钟，是用来清理计数器 C1，然后递增步骤 i。光标 13~14 是 .v 中第一帧发送的最后一个时钟，是用来清理计数器 SQ_C1，控制寄存器 x 和 i 下标寄存器。读者是不是看到其中的不同呢？就是 .v 文件时钟比 .vt 慢一个时钟。
在这里我们先看 .vt 文件。光标 13 是指向 .vt 文件步骤 0 的第 1913 个时钟，亦即清楚和更新步骤的动作。光标 14 指向 .vt 文件进入步骤 1 ，步骤 1 是拉低 TX_En_Sig，并且更新步骤 i。所以在该时钟的未来 TX_En_Sig 输出低电平，步骤 i 递增为 2，亦即进入步骤 2。光标 15 是指向步骤 2，这个待会儿再谈。

在这里我们看回 .v 文件。光标 14 是指向 .v 文件在 1913 个时钟，亦即清楚相关的寄存器。光标 15 是指向 .v 文件停在闲置状态（注意在光标 15 指向的同时，TX_En_Sig 的过去值是低电平）。光标 15~16 之间是 .v 文件闲置的一个时钟。光标 16 是指向第二帧数据发送的开始。

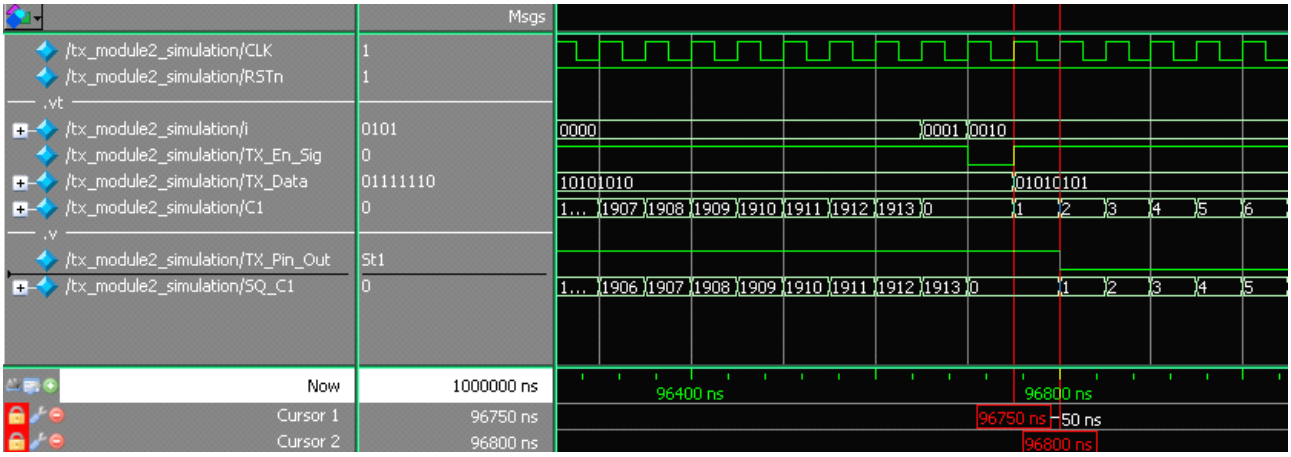


图 1.10q（第二帧数据开始发送）

图 1.10q 是第二帧数据的开始发送，光标 1 时候指向 .vt 的步骤 2 的第 0 个时钟。光标 1~2 之间表示了 .vt 和 .v 因为模块沟通的关系所以延迟了一个时钟，这一点和图 1.10e 很相似不同的是，图 1.10e 是第一帧发送数据发送的开始，而图 1.10q 是第二帧发送数据的开始。

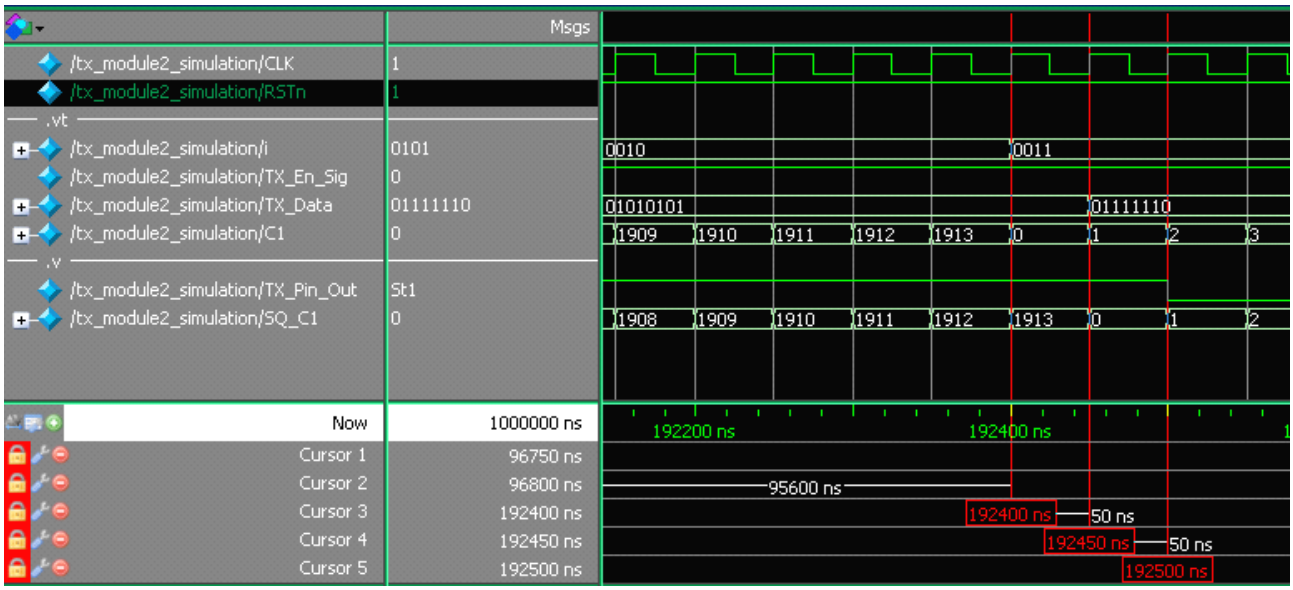


图 1.10r（第二帧数据的结束第三帧数据的开始）

光标 3~4 是指向 .vt 步骤~的最后一个时钟，该动作是清零 C1 然后更新步骤 i。在这里我们知道 .vt 里的步骤 1 和步骤 2 是连续启动串口发送模块，步骤 1 和 2 之间是没有停止的间隔。
光标 4~5 是指向 .vt 步骤 2 第三帧数据发送的启动开始。
光标 4~5 是指向 .v 发送第二帧数据的最后一个时钟，该动作就是清零 SQ_C1，x 和 i 等寄存器。光标 5 是指向 .v 第三帧数据发送的开始。

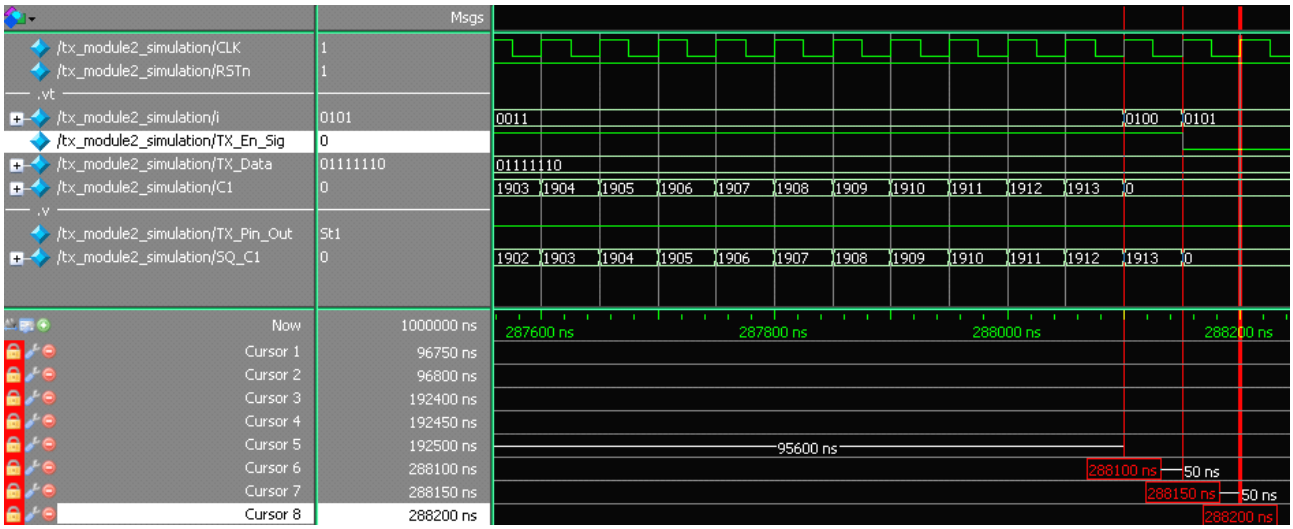


图 1.10s（第三帧数据发送的结束）

图 1.10s 是第三帧数据发送的结束 ... 具体的经过怒笔者懒得解释了。总之光标 6 之前是 .vt 步骤 3 的结束，光标 7 指向步骤 4 亦即决定拉低 TX_En_Sig，并且结束第二帧数据和第三帧数据连续启动发送的结束。光标 7 和之后就是无限停留了。光标 7~8 之间是 .v 中在发送第三帧数据中的最后一个时钟，就是清除相关的寄存器。之后就是无限闲置中 .....

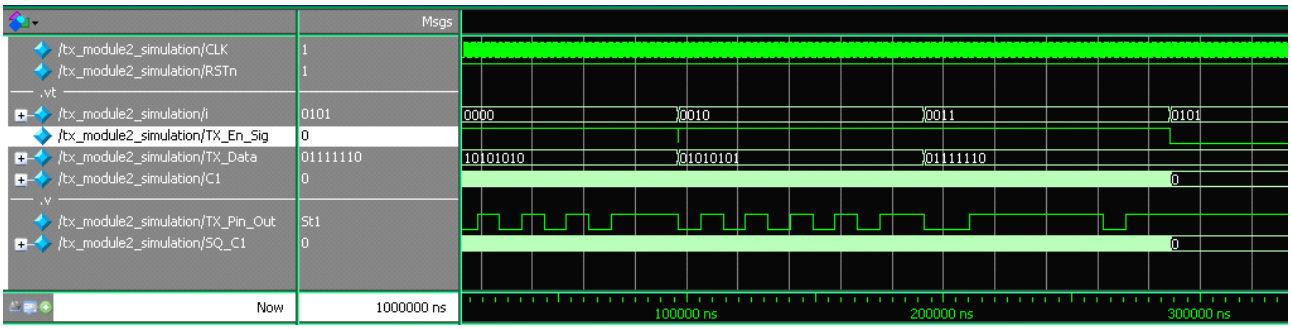


图 1.10t（仿真结果的全家福）

最后就献上一张全家福的仿真结果吧。在结束本章节之前，允许笔者说几句话：

经过实验三和实验四的洗礼，读者是不是发现到可以有效的评估模块所消耗的时钟数，是不是真的很重要呢？尤其是实验四，在 tx_module2.v 的仿真过程中，几乎一个时钟也没有被浪费 ... 但是有一点比较有郁闷的感觉是 tx_module.vt 刺激和控制 tx_module2.v 的过程中，不像使用 Start_Sig 和 Done_Sig 那么傻瓜。反之，用计算时钟数刺激和控制 tx_module2.v 就显得很需要理解“时序的概念”。

嗯，说实话笔者也想不到一套用在解释 FIFO 的方法既然也适合用于解释本章的实验，但是说来说去笔者还是比较倾向 Start_Sig 和 Done_Sig，毕竟用久了多少都有感情。还是那一句郁闷的朋友继续郁闷吧，如果看不明白笔者在说什么就加紧复习了。

1.11 模块之间调用的延迟

当我们在一个设计中实现过多的建模时，就不得不注意模块和模块之间的沟通延迟。
当然如果模块之前有控制信号，如 Start_Sig + Done_Sig 还是如同 FIFO 模块的 Write_Req_Sig | Read_Req_Sig + Full_Sig | Empty_Sig。也就是说控制信号是双向的，不然的话要有效的控制模块是很困难。

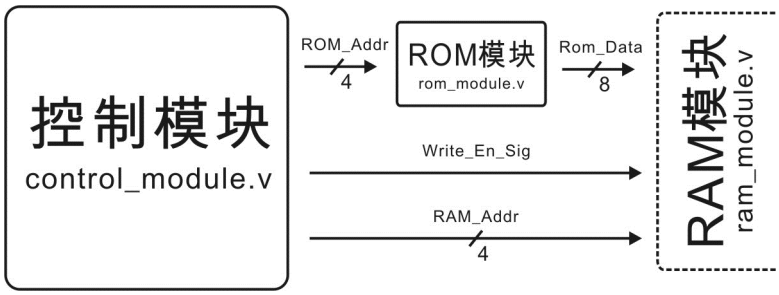


图 1.11a（典型的 LUT 实例）

LUT- Look Up Table 就是查表的意思，早期在建模篇里任何有关图像信息的储存，笔者都会建立 rom 模块来保存。我们先举个简单的例子：

假设 rom 模块里保存着“图像信息”，记得是“图像信息”哦！然后笔者需要将 rom 模块里的信息 16 个信息（Rom_Addr 位宽为四位）按次序写入 ram 模块里。

ROM_Addr 4'd0 空间的信息对应 RAM_Addr 4'd0 的空间。
ROM_Addr 4'd1 空间的信息对应 RAM_Addr 4'd1 的空间。
ROM_Addr 4'd2 空间的信息对应 RAM_Addr 4'd2 的空间。
ROM_Addr 4'd3 空间的信息对应 RAM_Addr 4'd3 的空间。
ROM_Addr 4'd4 空间的信息对应 RAM_Addr 4'd4 的空间。
ROM_Addr 4'd5 空间的信息对应 RAM_Addr 4'd5 的空间。
ROM_Addr 4'd6 空间的信息对应 RAM_Addr 4'd6 的空间。
ROM_Addr 4'd7 空间的信息对应 RAM_Addr 4'd7 的空间。
ROM_Addr 4'd8 空间的信息对应 RAM_Addr 4'd8 的空间。
ROM_Addr 4'd9 空间的信息对应 RAM_Addr 4'd9 的空间。
ROM_Addr 4'd10 空间的信息对应 RAM_Addr 4'd10 的空间。
ROM_Addr 4'd11 空间的信息对应 RAM_Addr 4'd11 的空间。
ROM_Addr 4'd12 空间的信息对应 RAM_Addr 4'd12 的空间。
ROM_Addr 4'd13 空间的信息对应 RAM_Addr 4'd13 的空间。
ROM_Addr 4'd14 空间的信息对应 RAM_Addr 4'd14 的空间。
ROM_Addr 4'd15 空间的信息对应 RAM_Addr 4'd15 的空间。

如果 Verilog HDL 语言是这样写的话：

```
case( i )

    0:
    if( x == 16 ) begin x<= 5'd0; i <= i + 1'b1; end
    else begin rROM <= x; rRAM <= x; isWrite <= 1'b1; x <= x + 1'b1; end

    ......
```

这段代码，无论是思路上还是理解上是完全真确的，但是在时序上却有一点小问题。

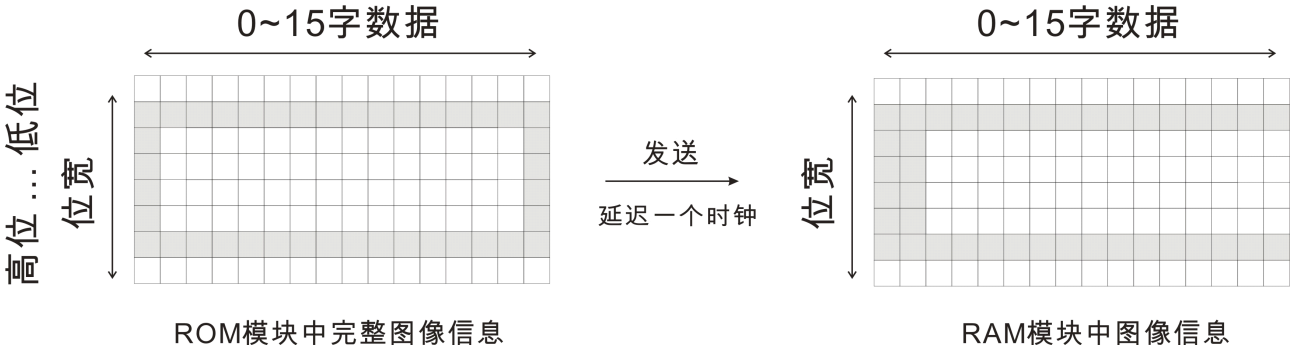


图 1.11b（受延迟的图像信息）

由于写入 ram 模块中的数据是从 rom 模块中查表的，根据"模块沟通的原理" rom 模块的至少需要消耗一个时钟，所以这也导致延迟发生了。从 rom 模块读取信息以后写入 ram 模块最终的结果会如图 1.11b，图像发生了右移现象。

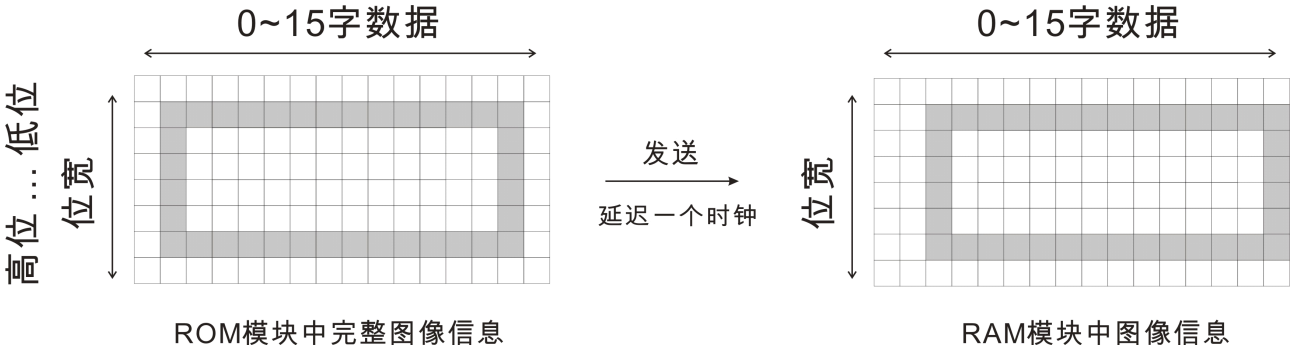
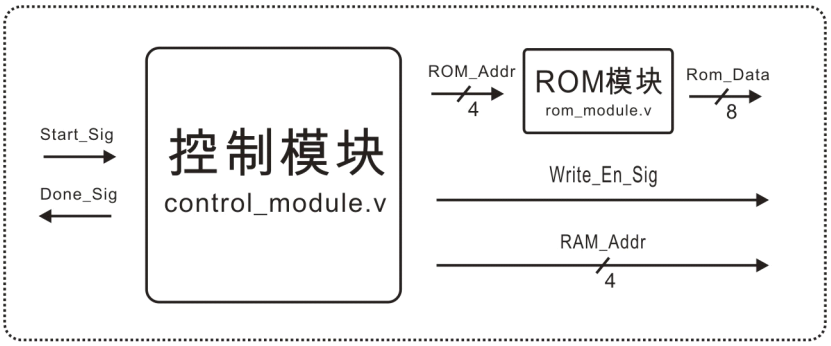


图 1.11c（处理图像信息右移的小手脚）

这种图像信息的现象在建模篇是普遍的小问题，在此笔者都对小问题们动了一些手脚。就是把一副图像信息的，左右上下都留下空白信息，那么在实际的显示效果中，图像信息右移一位人眼是看不出来的，效果如图 1.11c。但是，这始终都是骗小孩的把戏而已~呵呵！笔者在此，再一次的强调这个骗小孩的把戏只适合用于处理图像信息而已 ...

要完全解决在时序上被延迟的问题，我们还是得乖乖在时序上了解问题的真相。



exp5_env.v

1.11d （实验五的仿真环境）

在这里笔者建立用于仿真实验五的环境，exp5_env.v 是一个组合模块，它包含了控制模块和 ROM 模块，其中控制模块控制了所有的操作。我们先看看 ROM 模块的内容：

rom_module.v

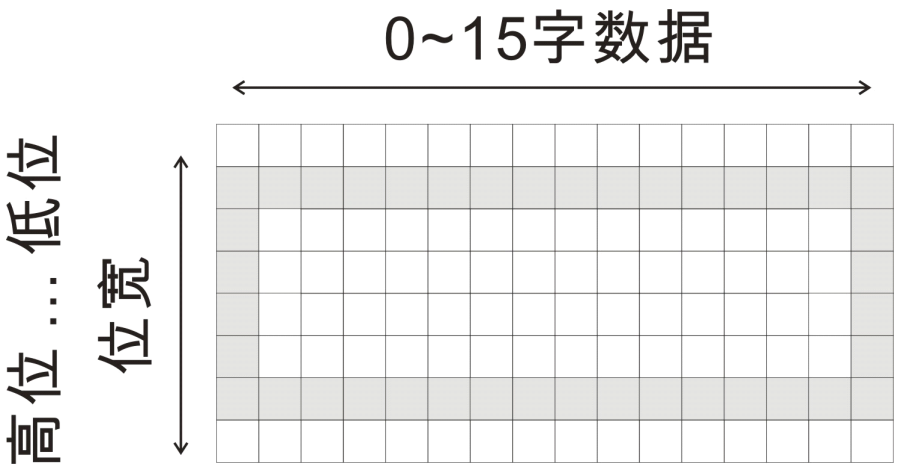


图 1.11e （rom 模块中包含的图像信息）

```
module rom_module
(
        input CLK,
        input RSTn,

        input [3:0]ROM_Addr,
        output [7:0]ROM_Data
);

      /**************************/

        reg [7:0]rData;
```

```
13.
14.         always @ ( posedge CLK or negedge RSTn )
15.             if( !RSTn )
16.                 begin
17.                     rData <= 8'd0;
18.                 end
19.             else
20.                 case( ROM_Addr )
21.
22.                     0: rData <= 8'b0111_1110;
23.                     1: rData <= 8'b0100_0010;
24.                     2: rData <= 8'b0100_0010;
25.                     3: rData <= 8'b0100_0010;
26.                     4: rData <= 8'b0100_0010;
27.                     5: rData <= 8'b0100_0010;
28.                     6: rData <= 8'b0100_0010;
29.                     7: rData <= 8'b0100_0010;
30.                     8: rData <= 8'b0100_0010;
31.                     9: rData <= 8'b0100_0010;
32.                     10: rData <= 8'b0100_0010;
33.                     11: rData <= 8'b0100_0010;
34.                     12: rData <= 8'b0100_0010;
35.                     13: rData <= 8'b0100_0010;
36.                     14: rData <= 8'b0100_0010;
37.                     15: rData <= 8'b0111_1110;
38.
39.                 endcase
40.
41.         /***************************/
42.
43.         assign ROM_Data = rData;
44.
45.         /***************************/
46.
47.     endmodule
```

rom 模块的图像信息基本上如图 1.1e。有关手动建立 rom 的方法笔者就不罗嗦的重复再重复了 ...

control_module.v

```
1.     module control_module
2.     (
3.         input CLK,
4.         input RSTn,
```

```
5.
6.          input Start_Sig,
7.          output Done_Sig,
8.
9.          output [3:0]ROM_Addr,
10.         output Write_En_Sig,
11.         output [3:0]RAM_Addr
12.   );
13.
14.       /****************************/
15.
16.         reg [1:0]i;
17.         reg [4:0]x;
18.         reg [3:0]rROM, rRAM;
19.         reg isWrite;
20.         reg isDone;
21.
22.         always @ ( posedge CLK or negedge RSTn )
23.               if( !RSTn )
24.                     begin
25.                             i <= 2'd0;
26.                             x <= 5'd0;
27.                             rROM <= 4'd0;
28.                             rRAM <= 4'd0;
29.                             isWrite <= 1'b0;
30.                             isDone <= 1'b0;
31.                         end
32.                 else if( Start_Sig )
33.                     case( i )
34.
35.                             0:
36.                             if( x == 16 ) begin x <= 5'd0; isWrite <= 1'b0; i <= i + 1'b1; end
37.                             else begin rROM <= x[3:0]; rRAM <= x[3:0]; isWrite <= 1'b1; x <= x + 1'b1; end
38.
39.                             1:
40.                             begin isDone <= 1'b1; i <= i + 1'b1; end
41.
42.                             2:
43.                             begin isDone <= 1'b0; i <= 2'd0; end
44.
45.                         endcase
46.
47.       /****************************/
48.
49.         assign ROM_Addr = rROM;
50.         assign RAM_Addr = rRAM;
```

```
51.         assign Write_En_Sig = isWrite;
52.         assign Done_Sig = isDone;
53.
54.         /****************************/
55.
56.
57.  endmodule
```

control_module.v 的内容实际上也挺简单的，步骤 0（35~37 行）的动作是同时间在 rom 模块读取图相信，然后又往 ram 模块写入“写入使能信号”和“写入地址”。步骤 1~2（39~43 行）是产生完成信号。

exp5_env.v

```
1.   module exp5_env
2.   (
3.          input CLK,
4.          input RSTn,
5.
6.          input Start_Sig,
7.          output Done_Sig,
8.
9.          output [7:0]ROM_Data,
10.         output [3:0]RAM_Addr,
11.         output Write_En_Sig,
12.
13.         /***********************/
14.
15.         output [3:0]SQ_ROM_Addr
16.
17.         /**********************/
18.  );
19.
20.        /****************************/
21.
22.         wire [3:0]ROM_Addr;
23.
24.         control_module U1
25.         (
26.                .CLK( CLK ),
27.                .RSTn( RSTn ),
28.                .Start_Sig( Start_Sig ),
29.                .Done_Sig( Done_Sig ),
30.                .ROM_Addr( ROM_Addr ),
31.                .Write_En_Sig( Write_En_Sig ),
```

```
                .RAM_Addr( RAM_Addr )
        );

        /*******************************/

        rom_module U2
        (
                .CLK( CLK ),
                .RSTn( RSTn ),
                .ROM_Addr( ROM_Addr ),
                .ROM_Data( ROM_Data )
        );

        /*******************************/

        assign SQ_ROM_Addr = ROM_Addr;

        /*******************************/

endmodule
```

以上是 exp5_env.v 组合模块的代码 ... 很简单，读者自己看着办吧。

exp5_env.vt

```
`timescale 1 ps/ 1 ps
module exp5_env_simulation();

        reg CLK;
        reg RSTn;

        reg Start_Sig;
        wire Done_Sig;

        wire [3:0]SQ_ROM_Addr;
        wire [7:0]ROM_Data;

        wire Write_En_Sig;
        wire [3:0]RAM_Addr;

        /*******************************/

        exp5_env U1
        (
                .CLK(CLK),
```

```
21.             .RSTn(RSTn),
22.             .Start_Sig(Start_Sig),
23.             .Done_Sig(Done_Sig),
24.             .RAM_Addr(RAM_Addr),
25.             .ROM_Data(ROM_Data),
26.             .Write_En_Sig(Write_En_Sig),
27.             .SQ_ROM_Addr(SQ_ROM_Addr)
28.         );
29.
30.         /******************************/
31.
32.         initial
33.         begin
34.             RSTn = 0; #1000 RSTn = 1;
35.             CLK = 0; forever #25 CLK = ~CLK;
36.         end
37.
38.         /******************************/
39.
40.         reg [3:0]i;
41.
42.         always @ ( posedge CLK or negedge RSTn )
43.             if( !RSTn )
44.                 begin
45.                     i <= 4'd0;
46.                     Start_Sig <= 1'b0;
47.                 end
48.             else
49.                 case( i )
50.
51.                     0:
52.                     if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
53.                     else Start_Sig <= 1'b1;
54.
55.                     1:
56.                     i <= i;
57.
58.                 endcase
59.
60.         /******************************/
61.
62.     endmodule
```

这个激励文本也是超简单的不行了，笔者就不多说了，读者还是自己看着办吧。

仿真结果：

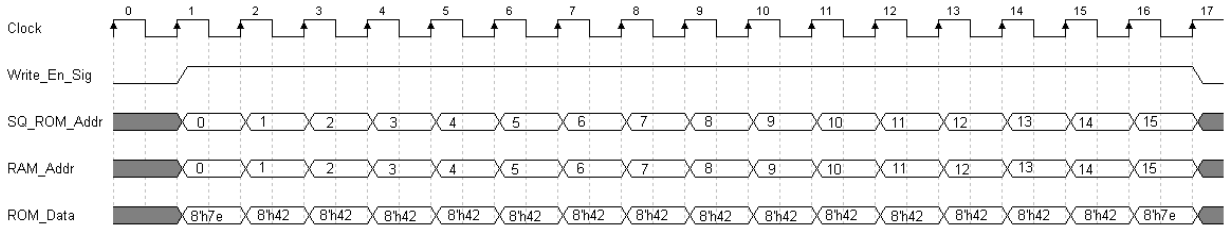


图 1.11f（预期中的时序）

在还没有正式开始解读仿真结果前，读者先来看看笔者预期中的时序。图 1.11f 是笔者预期中的是时序图，亦即 rom 模块对于时钟的消耗“为零”作为前提。就举时钟 T1 来解释吧：

我们先假设 rom 模块是不消耗任何时钟，当我们在 T1 为 rom 模块提供 SQ_ROM_Addr - 0 的时候，那么 ROM_Data 会取得“即时结果” - 8'h7e。在同一个瞬间我们也为 RAM 模块提供 Write_En_Sig 为 1 ，RAM_Addr 为 1。结果“rom 模块中地址为 0 的数据-8'h7e 就会存入 ram 模块中地址为 0 的地方。”

但是仿真结果悄悄好不是如此，由于 rom 模块实际上会消耗掉一个时钟，所以造成了 ROM_Data 的更新都比 Write_En_Sig 和 RAM_Addr 迟了一个时钟。结果造成了这三个信号的不同步。

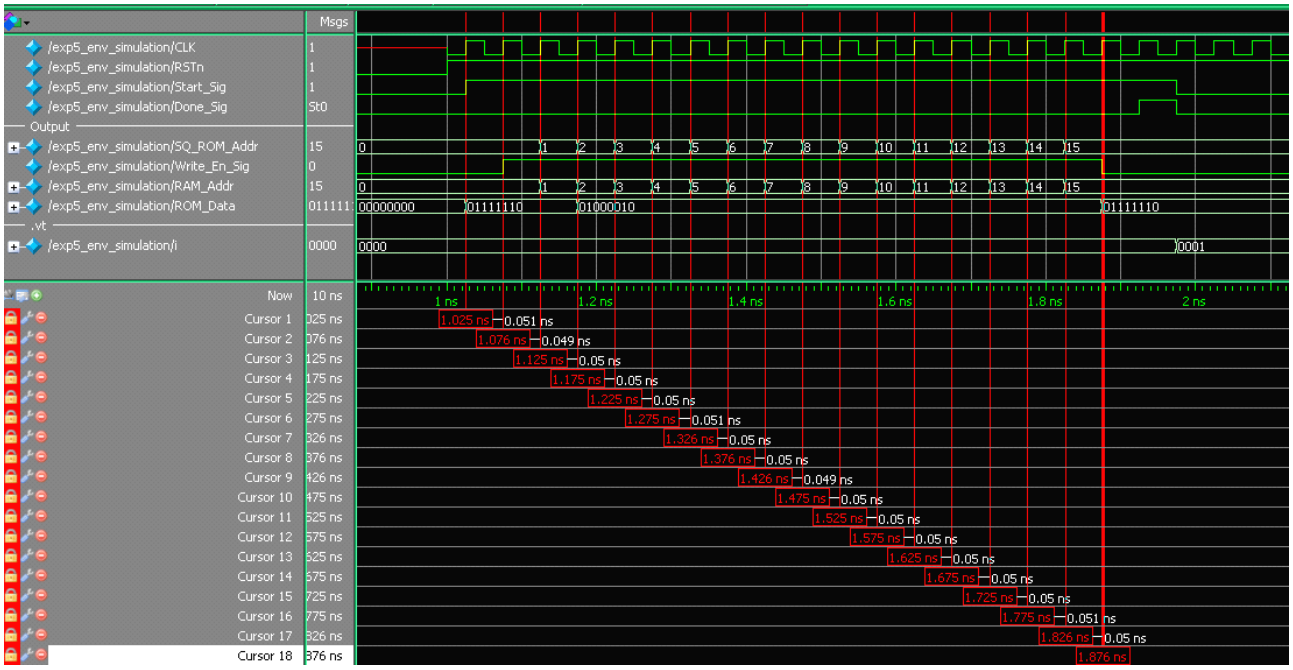


图 1.11g（实验五仿真结果）

图 1.11g 的仿真结果是有 rom 模块不小心延迟了一个时钟所造成的意外。其中光标 1 指向的是 T0，在 T0 的时候 .vt 开始拉高 Start_Sig。光标 2 是指向 T1，在这个时候 .v 检测到 Start_Sig 的过去值是逻辑 1，那么 .v 文件开始操作，亦即在 T1 的未来 .v 拉高 Write_En_Sig 和输出 SQ_ROM_Addr 和 RAM_Addr。

在这里请读者稍微注意一下，光标 1 之前所有模块内部相关的寄存器都为清零，理应驱动 ROM 模块的 SQ_ROM_Addr 也是 0。换句话说，ROM_Data 在复位的默认下的输出初值是 8'b0111_1110。

同样光标 2 指向 T1，这时候相关的未来值是：
SQ_ROM_Addr 为 0，Write_En_Sig 为 1，RAM_Addr 为 0，ROM_Data 为 8'b0111_1110。

看似这个结果和预期中的时序图很接近，实际上它存在 BUG。用前面的话来说，此时 ROM_Data 的 8'b0111_1110 是复位期间中的 8'b0111_1110，而不是在这个时间点里从 ROM 模块中读取的 8'b0111_1110。而在这个时间点中在 rom 模块中所请求地址-0 的数据 8'b0111_1110 会在下一个时间点里才有效 ...

光标 3 指向 T2，这时候相关的未来值是：
SQ_ROM_Addr 为 1，Write_En_Sig 为 1，RAM_Addr 为 1，ROM_Data 为 8'b0111_1110。

好了，这次的结果明显有错误了，而且非常露骨。其中 ROM_Data 的 8'b0111_1110，是在 T1 时从 rom 模块中地址为 0 读出的。也就是说，在 T1 中的数据 8'b0111_1110，在 T2 中更新并且生效了。说直接一点就是 T1 的数据延迟一个时钟至 T2 了。和预期中的结果相比，在 T2 时 ROM_Data 的数据应为 8'b0100_0010 ...

图 1.11h（延迟一个时钟以后的结果）

余下的 T3~T16 , ROM_Data 的未来值都是延迟了一个时钟，结果图像信息会像图 1.11h 那样。
o(╯□╰)o ~OMG

1.12 同步数据的几个想法

实验五所留下的问题事实上不是代码级的问题，它不会像在“时序篇”中般修改几行代码就可以完工，如果读者不是基于“时序”上解决问题的话，你再怎努力去修改代码，结果也一样又或者会更糟 ... 简单点说就是修改这段代码，那段代码就出现问题 ... 如此重复，最终头都大。

笔者记得自己在“时序篇”里说过，模块之间的沟通会出现延迟，基本上这不是真正的 BUG，而只是“模块的活动”在时序上安排不好而已。读者可能要问：“怎样个安排不好呢？”，答案很简单，而且都是一针见血，就是“模块的内容模糊！”和“时序概念理解模糊”。把话说白了，请读者自问：有没有看过任何一本参考书会仔细告诉你“这个时钟发生什么事？那个时钟发生什么事？时间点事件？即时事件？”

别傻了，参考书不会告诉你们这一些事儿的，很多参考书都是圈钱为目的，内容的焦点都是放在“如何完成设计”而不是“理解 Verilog HDL 语言的性质”.... 不说了，不说了！不然又要气火伤身了 ...

要解决类似问题的方法有很多：

1. 如果 rom 模块保存的对象是图像信息的话，那么可以在图像信息的上下左右方都留白，呵呵呵，这是笔者最喜欢的“骗小孩把戏”，这种简单的办法也只能使用在图像信息为前提下。

2. 如果我们无视 RTL 级的建模，那么我们可以把 rom 模块设计成为组合逻辑 ... 结果 rom 模块既遵守即时事件而不会在时序上有任何关心。但是这个方法有许多后遗症，组合逻辑一般上可以的话都要尽量的少用，尤其是 rom 这般为存储信息为目的的模块。

3. 读者还记得在“时序篇”中有一个实例，有两个流水操作的模块，因为各自的消耗时钟不同，那么笔者使用了“移位寄存器”来延迟其中一个时钟消耗较少的模块，使得两个流水操作的模块可以同步操作，同样的思想我们可以应用在其中。

4. 在前面笔者说了，如果可以显性的指向时钟，而且时钟也可以被评估的话，那么我们只要在写法上换个办法就可以了。这就是整合概念中的其中一个用意。

第一个解决方法笔者就不解释了，是小孩都会，呵呵！在这里，请原谅笔者曾经在建模篇里骗过读者的感情 ... 笔者真的很惭愧！第二个方法笔者也不演示了，没有任何学习的目的。好了我们先把焦点放在方法三，然后再看方法四。

实验六：加上过路寄存器

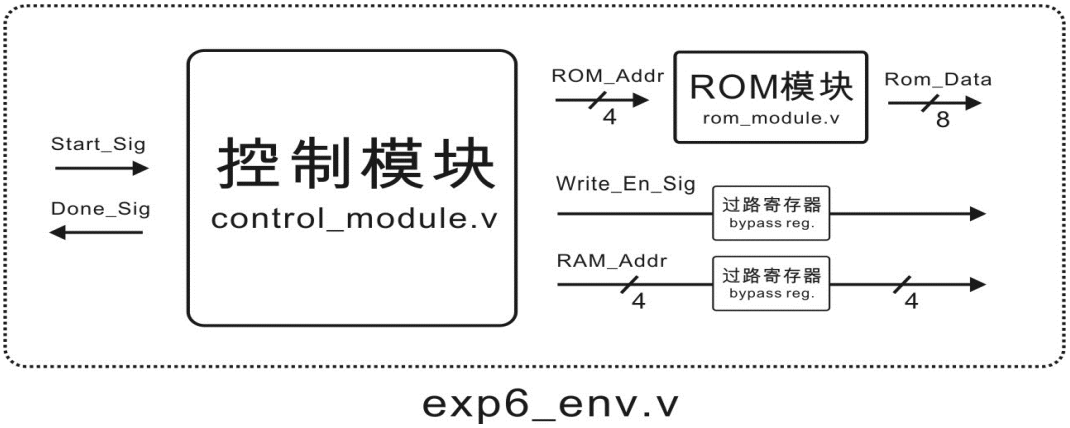


图 1.12a（加上过路寄存器）

Bypass register 称为过路|旁路寄存器，又或者称为 Clock delay register“时钟延迟寄存器”基本上它们是同一样东西，只是在作用不同称呼也不同而已。图 1.12a 是经过修改的实验五，笔者从新命名为实验六至于又为 Write_En_Sig 和 Ram_Addr 加上过路寄存器。实验六的思想基本上和“移位寄存器”一样，由于 ROM_Addr 查表 ROM 模块导致“延迟了一个时钟”，所以 Write_En_Sig 和 RAM_Addr 也需要加上一个“无用的过路寄存器”来“延迟一个时钟”，使得 Write_En_Sig , RAM_Addr 信号才能同步与 ROM_Addr 经 ROM_Data。

exp06_env.v

```
module exp6_env
(
        input CLK,
        input RSTn,

        input Start_Sig,
        output Done_Sig,

        output [7:0]ROM_Data,
        output [3:0]RAM_Addr,
        output Write_En_Sig,

        /**********************/

        output [3:0]SQ_ROM_Addr

        /*********************/
);

      /****************************/
```

```
21.
22.         wire U1_Write_En_Sig;
23.         wire [3:0]U1_RAM_Addr;
24.         wire [3:0]ROM_Addr;
25.
26.         control_module U1
27.         (
28.                 .CLK( CLK ),
29.                 .RSTn( RSTn ),
30.                 .Start_Sig( Start_Sig ),
31.                 .Done_Sig( Done_Sig ),
32.                 .ROM_Addr( ROM_Addr ),
33.                 .Write_En_Sig( U1_Write_En_Sig ),
34.                 .RAM_Addr( U1_RAM_Addr )
35.         );
36.
37.         /********************************/
38.
39.         rom_module U2
40.         (
41.               .CLK( CLK ),
42.                 .RSTn( RSTn ),
43.                 .ROM_Addr( ROM_Addr ),
44.                 .ROM_Data( ROM_Data )
45.         );
46.
47.         /********************************/
48.
49.         reg rBy1;
50.         reg [3:0]rBy2;
51.
52.         always @ ( posedge CLK or negedge RSTn )
53.               if( !RSTn )
54.                     begin
55.                             rBy1 <= 1'b0;
56.                             rBy2 <= 4'd0;
57.                     end
58.                 else
59.                     begin
60.                             rBy1 <= U1_Write_En_Sig;
61.                             rBy2 <= U1_RAM_Addr;
62.                     end
63.
64.         /***********************************/
65.
66.         assign Write_En_Sig = rBy1;
```

```
67.         assign RAM_Addr = rBy2;
68.
69.         /**************/
70.
71.         assign SQ_ROM_Addr = ROM_Addr;
72.
73.         /******************************/
74.
75.   endmodule
```

exp6_env.v 和 exp5_env.v 相比只是组合模块的内容修改了一点而已。第 33~34 行的 Write_En_Sig 和 RAM_Addr 没有直接驱动顶层的输出口，反之是连线至 U1_Write_En_Sig 和 U1_RAM_Addr。那么这两个连线的目的地是哪里呢？答案是 49~62 行的过路寄存器里。其中笔者声明了一位位宽的 rBy1 和 四位位宽的 rBy2，前者是 Write_En_Sig 的过路寄存器，后者是 rBy2 的过路寄存器。然后 rBy1 和 rBy2 分别驱动顶层的输出口（第 66~67 行）。

exp6_env.vt

大致上和 exp5_env.vt 大同小异，读者自己看着办吧。

仿真结果：

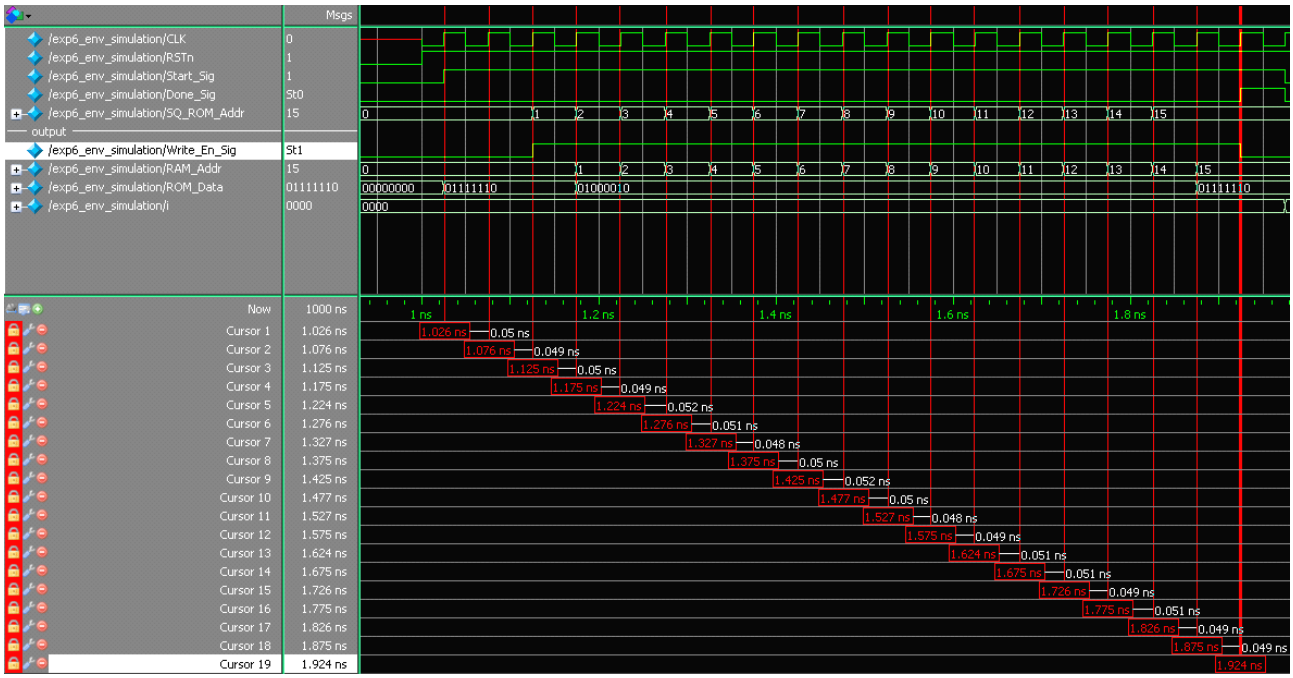


图 1.12b（实验六仿真结果）

在这里读者自己需要去比较 图 1.12b 和 图 1.11g (实验五仿真结果)了。实验六的仿真结果比起实验五的仿真结果，多消耗了一个时钟，具体请数一数光标的数目。那么是在哪一个光标上除了问题？？

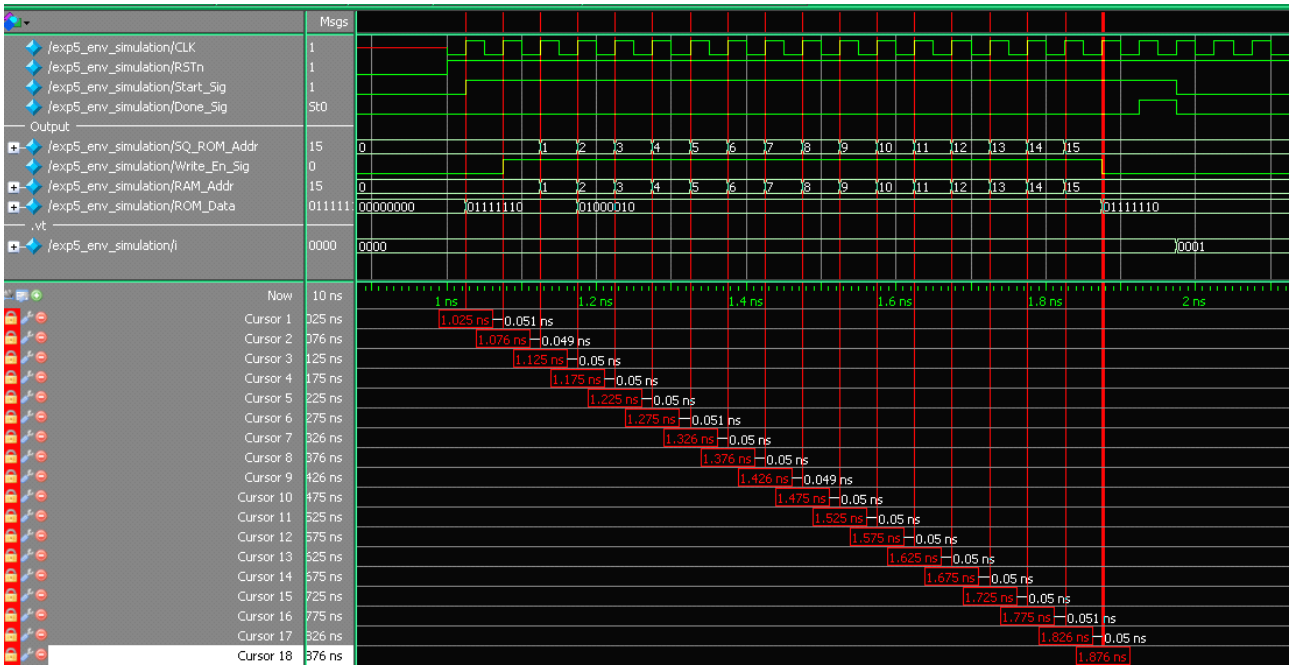
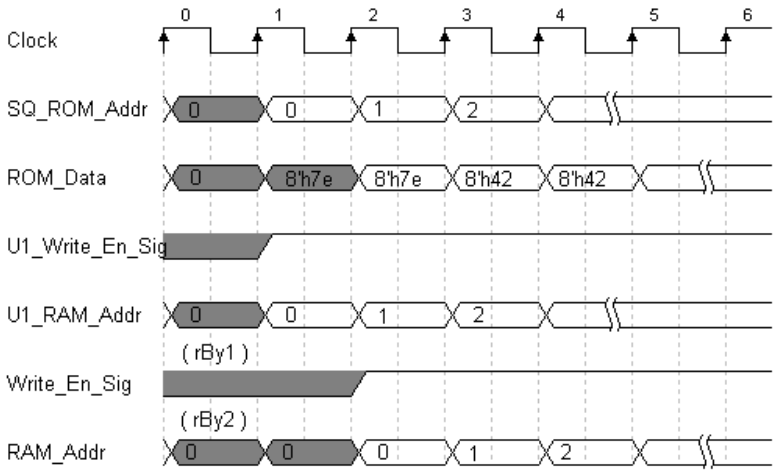


图 1.11g（实验五仿真结果）

笔者就知道读者们懒得翻页了，所以故意又复制了一张实验五的仿真结果。在图 1.11g 里，当光标数是 2 的时候 Write_En_Sig 就拉高，反之在图 1.12b 图 Write_En_Sig 是在光标 3 的时候才拉高。同样的事情也发生在 ROM_Addr。



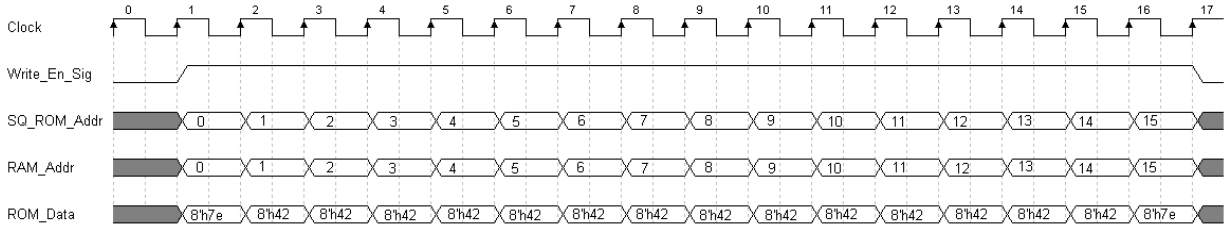
1.12c (实验六内部时序过程)

好了，笔者在实验六仿真结果的其中开了一刀，看看里边发生了什么事情。图 1.12c 是实验六内部时序的经过，灰色的填充是可以“无视”。在这里读者需要知道的就是 Write_En_Sig，RAM_Addr 和 ROM_Data 需要同步就是了。其中的 U1_Write_En_Sig 和 U1_RAM_Addr 是由 U1 产生的，Write_En_Sig 和 RAM_Addr 是由 rBy1 和 rBy2 过路寄存器来驱动，然而 rBy1 和 rBy2 则由 U1_Write_En_Sig 和 U1_RAM_Addr 来驱动。

我们先看 SQ_ROM_Addr 和 ROM_Data： 在 T1 的未来 U1 开始给 SQ_ROM_Addr 输出，

根据模块沟通的原理，ROM 消耗了一个时钟，所以 ROM_Data 取得的“真正”结果是在 T2 的过去值才产生。接下来 U1_Write_En_Sig 和 U1_RAM_Addr 同样也是由 U1 产生，它们和 SQ_ROM_Addr 也一样在同一个时间被输出。前提是 U1_Write_En_Sig 和 U1_RAM_Addr 要和 ROM_Data 同步，而不是和 SQ_ROM_Addr 同步哦。所以就利用 rBy1 和 rBy2 来延迟一步 U1_Write_En_Sig 和 U1_RAM_Addr。

在图 1.12c 中，在 T2 的时候 ROM_Addr 和 Write_En_Sig 和 RAM_Addr 几乎是同步了！这就是我们预期的结果~



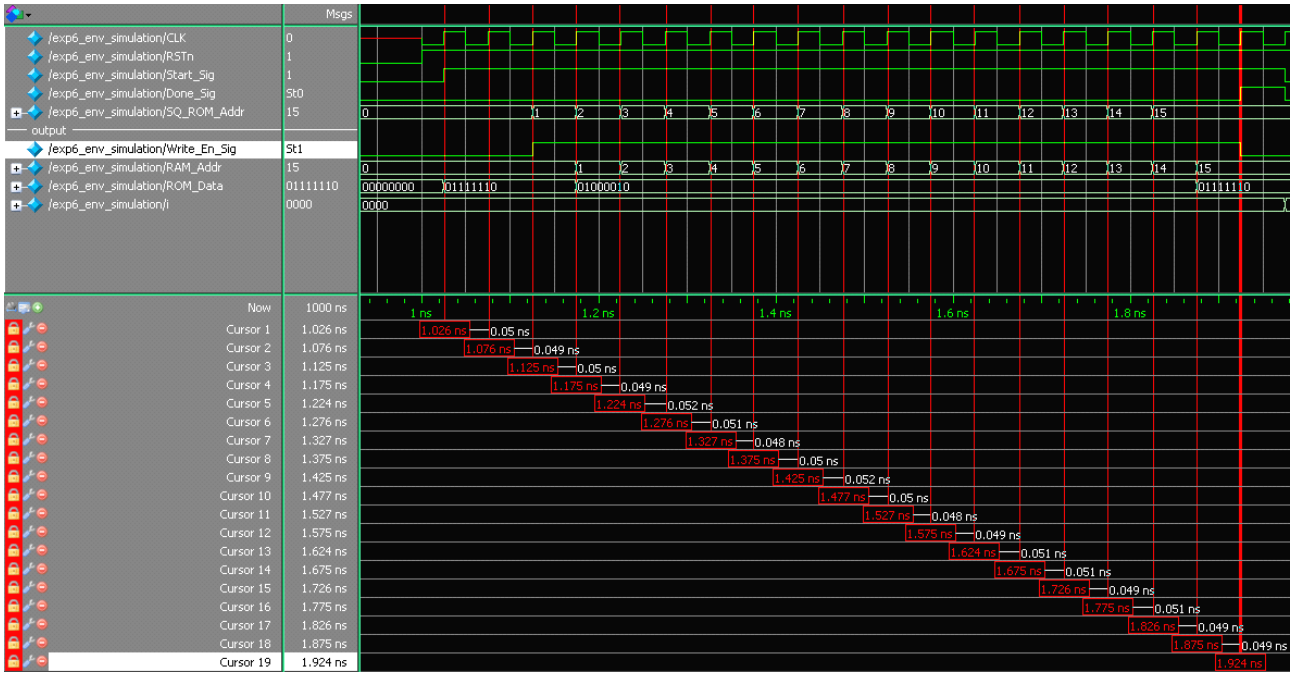


图 1.12d（预期结果和仿真结果相互比较）

请注意在仿真结果中的光标 3~19 正好和预期结果中的 T1~T16 是一致的，换句话说“预期结果中的 Write_En_Sig + RAM_Addr + ROM_Data 的同步”与“仿真结果中的 Write_En_Sig + RAM_Addr + ROM_Data 的同步”是一致的。那么我们的任务就完成了，真是可惜可贺。说一点题外话：

用过路寄存器时钟是一个被动的方法，如果 A 模块和 B 模块之间有 5 个延迟 .... 那么不是要建立 5 个移位寄存器来延迟时钟来达到同步吗？所以说，虽然用过路寄存器来延迟以达到模块之间在时序上的同步是一个好办法，但是它只是一个小方法，只有小作为而已。在一些大场面的地方小作为的东西只有“鸡助”的份而已 .... 所以我们需要考虑第四个的解决方法。

（注：组合模块的功能可不小啊，在许多方面都提到很大的帮助，如输出选择器等 ... ）

实验七：用循环操作来同步数据

在这里我们先把实验六给忘了，因为实验六是实验五其中一个解决方法而已。如此实验七也是实验五中的另一个解决方法。顺便笔者也来刷刷读者的大脑，实验六解决延迟的方法是在设计的后期间，在组合模块里添加过路寄存器，属于被动式的方法。反之，实验七解决延迟的方法是直接在写法上下功夫，属于主动式方法。好了不废话了，还是直接上源码：

control_module.v

```
module control_module
(
        input CLK,
        input RSTn,

        input Start_Sig,
        output Done_Sig,

        output [3:0]ROM_Addr,
        output Write_En_Sig,
        output [3:0]RAM_Addr
);

      /****************************/

        reg [1:0]i;
        reg [4:0]x;
        reg [4:0]y;
        reg [3:0]rROM, rRAM;
        reg isWrite;
        reg [4:0]C1;
        reg isDone;

        always @ ( posedge CLK or negedge RSTn )
            if( !RSTn )
                begin
                        i <= 2'd0;
                        x <= 5'd0;
                        y <= 5'd0;
                        rROM <= 4'd0;
                        rRAM <= 4'd0;
                        isWrite <= 1'b0;
                        C1 <= 5'd0;
                        isDone <= 1'b0;
                end
            else if( Start_Sig )
                case( i )
```

```
38.
39.                             0:
40.                             begin
41.                                 if( y == C1 ) begin y <= y + 1'b1; rROM <= y ; end
42.                                 if( x +1 == C1 ) begin x <= x + 1'b1; isWrite <= 1'b1; rRAM <= x; end
43.
44.                                 if( C1 == 17 -1 ) begin C1 <= 5'd0; x <= 5'd0; y <= 5'd0; i <= i + 1'b1; end
45.                                 else C1 <= C1 + 1'b1;
46.                             end
47.
48.                             1:
49.                             begin isWrite <= 1'b0; i <= i + 1'b1; end
50.
51.                             2:
52.                             begin isDone <= 1'b1; i <= i + 1'b1; end
53.
54.                             3:
55.                             begin isDone <= 1'b0; i <= 2'd0; end
56.
57.                         endcase
58.
59.         /****************************/
60.
61.           assign ROM_Addr = rROM;
62.           assign RAM_Addr = rRAM;
63.           assign Write_En_Sig = isWrite;
64.           assign Done_Sig = isDone;
65.
66.           /****************************/
67.
68.
69.   endmodule
```

在 16~22 行是控制模块所用的相关寄存器，和实验五相比多了 x，y，C1 (17,18 和 21 行)。寄存器 x 是用来控制 isWrite 和 rRAM（42 行），寄存器 y 是用来控制 rROM（41 行），然而 C1 就是用来计数时钟（44~45 行）。

实验五所示，rROM 的驱动目的是用作查表，以至于从 ROM 模块读出相关的数据，rRAM 和 isWrite 的驱动目的是给 RAM 模块寻址和使能。但是，由于模块沟通的关系当 rROM 查表 ROM 模块在中读出数据时就消耗一个时钟，结果从 ROM 模块中读出的数据与 rRAM 和 isWrite 的驱动造成不同步。

在实验六里，在后期中我们在组合模块里 exp6_env.v 为 rRAM 和 isWrite 的驱动源挂上过路寄存器来达到延迟一个时钟的效果，使得输出可以同步。相反的，在实验七里 control_module.v - 39~46 行，使用了循环操作的写法。

第 44 行的 if(C1 == 17 -1) 其中的 C1 == 17 表示了，这一个步骤需要 17 个时钟，那么为什么需要 17 个时钟呢？很简单，我们要循环操作 16 次，再加上 ROM 模块的沟通延迟 1 个时钟，所以结果是 17 个时钟。至于 -1 表示在这个步骤的最后一个时钟清零相关的寄存器以外，还更新步骤。

第 41 行的 if(y == C1) 表示在每一个时钟之中都执行 y 更新，然后 rROM 也更新。第 42 行的 if(x +1 == C1) 表示 x 延迟一个时钟执行，在以后的每一个时钟之中都更新 x 和 rRAM ，然而 isWrite 一直保持拉高。步骤 1（48~49 行）是用来拉低 isWrite 信号。步骤 2~3 是用来产生完成信号。

嗯 ..... 读者是不是看得头都大了？呵呵，笔者也这么认为 .... 确实单靠代码是不能完全解释时序。还是来看仿真结果。

exp7_env.vt

基本上和实验五~六的激励文本相似，读者自己看着办吧。

仿真结果：

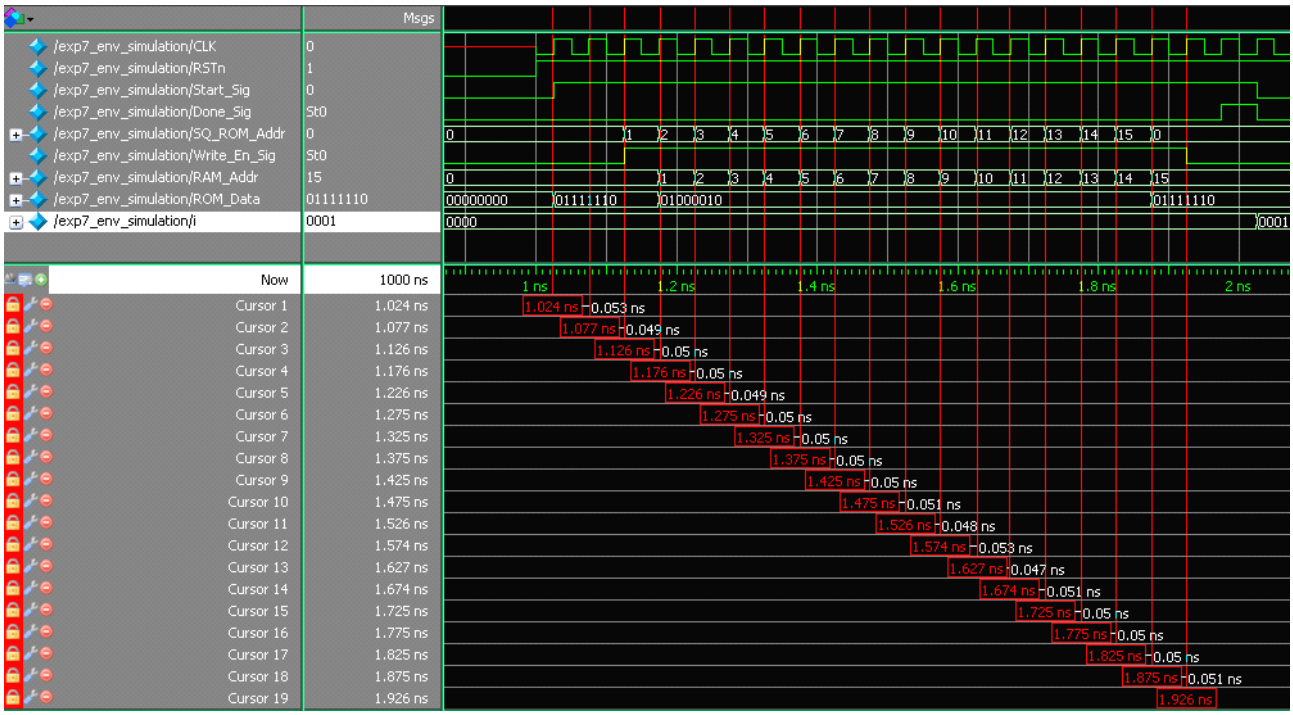


图 1.12e（实验七仿真结果）

图 1.12e 是实验七的仿真结果 ... 是不是和实验六的仿真结果一摸一样呢？虽然仿真结果均为一样，但是在解决的方法上，实验七看似更麻烦 .... 其中光标 2~18 是 .v 文件中步骤 0 的操作，数数看是不是有 17 个时钟。

在光标 1 .vt 开始使能的仿真对象，在光标 2 的时候 .v 判断 Start_Sig 的过去值为高，那么 .v 开始启动和执行步骤 0。光标 2 是步骤的 0 的第 0-CLK，光标 2~17 也就是 x 控制的所 16 次

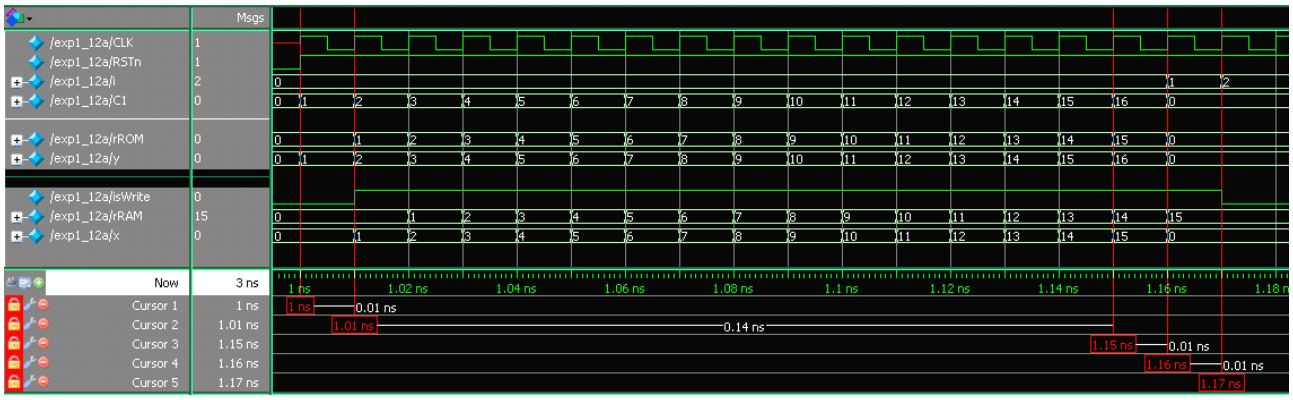
循环操作。光标 3 是步骤 0 的第 1-CLK，光标 3~18 也就是 y 控制的 16 次循环操作。那么问题来了 ....

exp1_12a.vt

```
39.                     0:
40.                     begin
41.                             if( y == C1 ) begin y <= y + 1'b1; rROM <= y[3:0]; end
42.                             if( x +1 == C1 ) begin x <= x + 1'b1; isWrite <= 1'b1; rRAM <= x[3:0]; end
43.
44.                             if( C1 == 17 -1 ) begin C1 <= 5'd0; x <= 5'd0; y <= 5'd0; i <= i + 1'b1; end
45.                             else C1 <= C1 + 1'b1;
46.                     end
47.
48.                     1:
49.                     begin isWrite <= 1'b0; i <= i + 1'b1; end
```

读者知道吗？在这一段代码之中，尤其是在步骤 0 的最后一个时钟里，到底发生额什么事情吗？我们把这小段代码再放大仿真看看：

exp1_12a.vt 的仿真结果



光标 1 是指向步骤 0 的 T0，在 T0 的时候 if(y == C1) 成立然后，y 递增，rROM 被赋予 y 的过去值，亦即 0。在 T0 的未来 y 的未来值为 1，rROM 的未来值为 0。y 的循环操作执行直到光标 3 指向的 T15。

在光标 4 指向的 T16，rROM 理应被赋予 y 的过去值，亦即 16，但是为什么在 T16 的未来，rROM 的过去值会是 0 而不是 16？答案很简单 rROM 被赋予 y[3:0]，16 的二进制是 5'b10000，所以是 4'b000。同样在 T16 的瞬间，由于 if(C1 == 17-1) 成立 y <= 5'd0; 的优先级大于 y <= y + 1'b1; 所以在 T16 的未来 y 的未来值是 0 而不是 17。

光标 2 是指向步骤 0 的 T1，在 T1 的时候 if(x+1 == C1) 成立，rRAM 被赋予 x 的过去值，同时间 isWrite 也被拉高。所以在 T1 的未来 x 的未来值为 1，rRAM 的未来值为 0。x 的循

环操作执行直到光标 4 指向的 T16。很凑巧的是在 T16 的时候 if(C1 == 17-1) 也成立，与其 y <= y + 1'b1 被执行但是优先级的关系，所以 y <= 5'd0 被优先执行。

不过有一点不一样的是 if(C1 == 17-1)的成立不包括拉低 isWrite，isWrite 更像是循环保留的操作。isWrite 在步骤 0 的 T1 被拉高（光标 2），由于它需要和 rRAM 同步，所以 isWrite 也必须拉高 16 个时钟的时间，亦即 T1~16（光标 2~4）。直到步骤 1（光标 5）的时候它才被拉低。

到目前为止，是不是觉得实验七比起实验六来得更复杂？呵呵呵~笔者也这样认为，实际上一点都不复杂，如果读者要熟悉时序的话。实验七对于读者来说一点也不复杂，只是初鸭下水会觉得困惑一点，但是游水永远都是属于鸭子的专利，同样的道理时序也是属于 Verilog HDL 语言的专利。鸭子怎么能不游水？Verilog HDL 语言怎么不关心时序？

总结：

嗯！这一本笔记真的很耗笔者的时间。笔者用了将近 2 个月去构思和准备概念，然后用去将近 2 个月的时间去编辑笔记和实例。比起其他笔记，这一本笔记更让笔者满意，因为许多早期的想法都得到证实是行得通的。不过实话说，时序确实是许多 Verilog HDL 语言新手最不喜欢的一环 .... 开始的笔者也是。但是深入以后，时序才是 Verilog HDL 语言的本质，天下。

从笔记的第一章节开始，笔者就提起“循环保留”的概念，其实这一概念的提起绝对是有目的的，它完全是针对传统的状态机而来的。传统的状态机都喜欢使用 cState 和 nState 这两个寄存器来表示当前状态，和下一个状态 .... 但是读者你知道吗？传统的状态机当从一个状态切入另一个状态就要消耗掉 2 个时钟，比起笔者爱用的“步骤”（读者可以把它看成简易状态机）多了一个时钟，而且调用也不方便。“循环保留”的就是要把状态机更新所耗的时间都节省掉。

但是笔记最大的重点还是“显性指向时钟”，几乎每个模块的操作步骤都会出现 C1 这个时钟计数器。显性指向时钟的好处，莫过于更直接把模块的内容表达出来以外，模块在时序上还可以更好的被控制，尤其要实现“循环操作”一点也不难。换句话说要实现多少个嵌入循环也没有问题 .... 但是嵌入循环不是“循环操作”的重点呀 .... 具体的内容都在笔记里，笔者就不多重复了。

好了，笔者差不多要结束这本笔记了 ... 整合的概念是一个很“有趣和神奇”的书写技巧，如果一个 Verilog HDL 语言的设计加上“建模技巧”和“整合”，可以说是屠龙刀和倚天剑都出来了 .... 哈哈哈！笔者有点老王卖瓜了 .... 好了好了，不多话了！喜欢的同学自己看着办吧，这本笔记不可能看一次就能看明白了，而且也要而外的细心和耐性。

第二章 - 时钟化和信号的长度

2.1 一个时钟一块数据的概念

距离第一章，第二章的内容同样也使用了很长的时间去构思和准备。期间笔者一直思考着另一种更直接的概念，用于表达“整合”在“建模”和“时序之间的作用和连接性。整合的概念，说白了就是一种写法，一种连接“建模概念”和“时序概念”的写法。但是如何清晰的表达它，在此我们需要借用另一个概念，亦即“一个时钟一块数据”。我们先来看一看一段简单的代码：

```
else if( Start_Sig )
        case(i)
                0:
                begin isDone <= 1'b1; i <= i + 1'b1; end
                1:
                begin isDone <= 1'b0; i <= i + 1'b1; end
                ......
......
assign Done_Sig = isDone;
```

在上面的一段代码中，如果以“建模”的角度去看的话，就是产生一个“完成信号”的高脉冲。如果以“时序”的角度去理解的话，就是 T0 的时候“决定”拉高 isDone，在 T1 的时候“决定”拉低 isDone。如果我们想用整合的角度去理解上面的代码的话，事实上是不可能的，因为整合概念的本质是一个写法，为了使这个写法更“实在和具体”一点，因此我们需要“一个时钟一块数据”的概念。

好了，如果以“一个时钟一块数据”去理解的话，那么上述的一段代码表示了“某个模块输出一个时钟长度的 Done_Sig 数据”。至于这个数据的价值（值）是逻辑 1 还是逻辑 0，基本上“一个时钟一个数据”它不感兴趣。但是为了更习惯人类脑袋的理解，我们可说成：“某个模块输出一个时钟长度的 Done_Sig 高脉冲”。

“信号或者数据如果有长度的话 ...”对！这就是一个重点，只要“某个信号或者数据”拥有长度的话，那么这个“信号或者数据”就是“非透明”的和“可以计算”的，只要“某个信号或者数据”可以计算的话，那么整合的概念就能起到作用了。

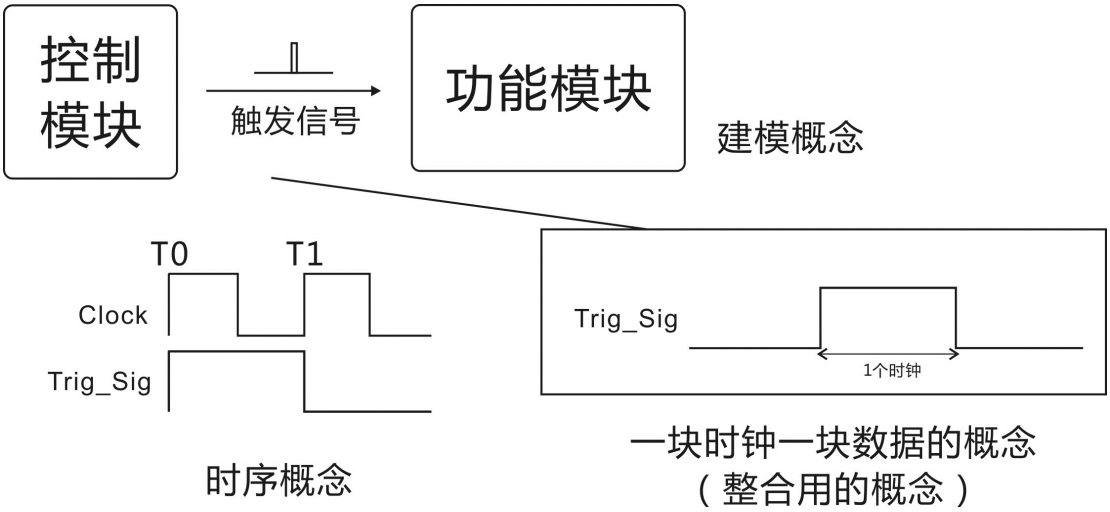


图 2.1a 建模・时序・整合之间的概念之分

图 2.1a 是 Verilog HDL 语言三个概念之间的区分，图中我们可以看到建模概念是最“大体的”，控制模块向功能模块发送一个触发信号。然后在时序的概念中，控制模块在 T0 的时候决定拉高 Trig_Sig 然后在 T1 的时候决定拉低 Trig_Sig。当我们在建模的时候，基本上我们是不会关心时序上的小细节，反之我们更关心“建模的层次”，然而当我们在分析模块的活动（时序）的时候，关心的对象会变成“模块的沟通记录”而不是“建模的层次”。

在整合的概念中，笔者为了可以把“时序概念”和“建模概念”更好的联系起来，单单在“写法”上下功夫是不足的，因为写法是非常抽闲而且空虚的东西，因此笔者需要给“这个写法”提供一个“具体的形状”，所以笔者在整合中借用“一个时钟一块数据”这个概念是必须的。

继续分析图 2.1a，当控制模块向功能模块发送一个触发信号的时候，如果我们直接用“一个时钟一块数据”来看的话：

第一我们不知道控制模块在什么时候向功能模块发送触发信号。第二我们不可能任何时候都用“时序概念”去控制模块和功能模块的沟通记录，因为这样是会累死人的。从中我们只是知道一个事实而已，就是控制模块向功能模块发送了“一个时钟长度的触发信号”。

在 Verilog HDL 的世界中，任何“模块”都依赖着“时钟”作为自身的驱动动力，换句话说“时钟就是模块的心脏”（时序概念的中心思想）。所以，模块的任何活动逃不出时钟，以前笔者在时序篇中所关心的是“某个模块在某个时间点在决定干某某事？”或者“某个模块和某个模块在某个时间点的沟通怎么了？”等。在这里，笔者不是说“时序的概念”用作细化模块有错，而是分析模块的活动记录“太累人了”（估计很多读者都有同感）。很多时候“分析时序”都是最后的工作，正如笔者在时序篇中常常挂在嘴边的口头禅“我们不可能写一行代码，然后仿真一次 ... ”，如果是这样子的话，工作会没完没了。

一些模块的设计，基本上有“xx_En_Sig”或者“xx_Start_Sig”等信号，前者可以在 RAM 模块中看到，后者可以在仿顺序操作的模块中看到。xx_En_Sig 或者 xx_Start_Sig 在某种意义上，可以看成是“模块的工资”...

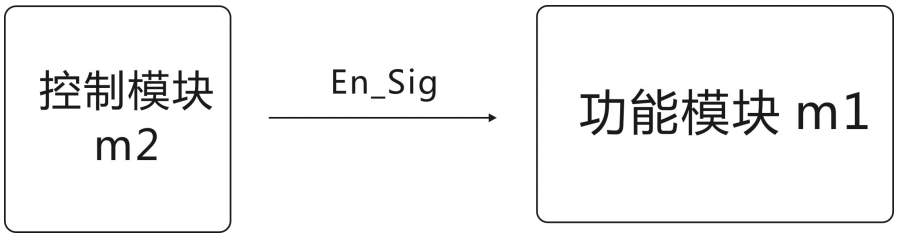


图 2.1b m2 使能 m1 的建模图。

假设有一个功能模块 m1，本身就需要消耗掉 10 个时钟来工作，而且 En_Sig 作为它的时能信号。然而控制模块 m2 就是充当使能 m1 的角色。

```
module m1 //功能模块
(
        input En_Sig,
        ......
);

        always @ ( posedge CLK ... )
            ...
        else if( En_Sig )
            case( i )
                    0,1,2,3,4,5,6,7,8:
                    begin x <= x + 1'b1；i <= i + 1'b1; end

                    9:
                    begin x <= x + 1'b1; i <= 4'd0; end
          endcase
        ......

endmodule
```

m1 功能模块的操作就是一当 En_Sig 拉高，每一个时钟沿 CLK 都递增 x。

```
module m2 //控制模块
(
        output En_Sig,
        ......
);

        reg isEn;
        ......
        always @ ( posedge CLK ... )
                ...
                else
                        case( i )
```

```
                0
                if( C1 == 10 -1 ) i <= i + 1'b1;
                else begin C1 <= C1 + 1'b1; isEn <= 1'b1; end

                /*   或者可以这样写:
                0,1,2,3,4,5,6,7,8,9:
                begin isEn <= 1'b1; i <= i + 1'b1; end
                */

                ......

            endcase
    ......
endmodule
```

控制模块 m2，为了使 m1 工作，它必须拉高 En_Sig 信号 10 个时钟。

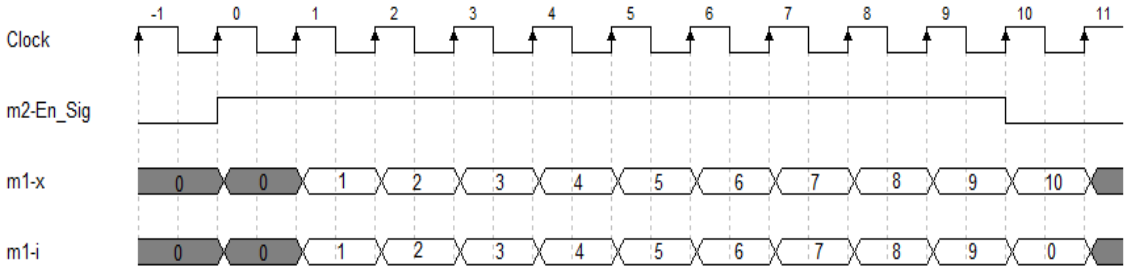


图 2.1c m2 与 m1 的活动记录

图 2.1c 是 m2 和 m1 的活动记录（时序），在时序的理解上，在 T0 的时候 m2 决定拉高 En_Sig ，然后在 T0 的未来 En_Sig 输出逻辑 1。在 T1 的时候 m1 检查到 En_Sig 的过去值是逻辑 1，那么 x 和 i 都“决定”递增，然后在 T1 的未来 x 和 i 从 0 递增为 1 。接下来的时钟都是以此类推 .... 直到 T10 的时候 m2“决定”拉低 En_Sig，然而 m1 已经进入步骤 9 并且检测到 En_Sig 的过去值是 1，这时候 m1“决定”递增 x 和清零 i。所以在 T10 的未来，En_Sig 输出逻辑 0，x 递增为 10 和 i 清零。

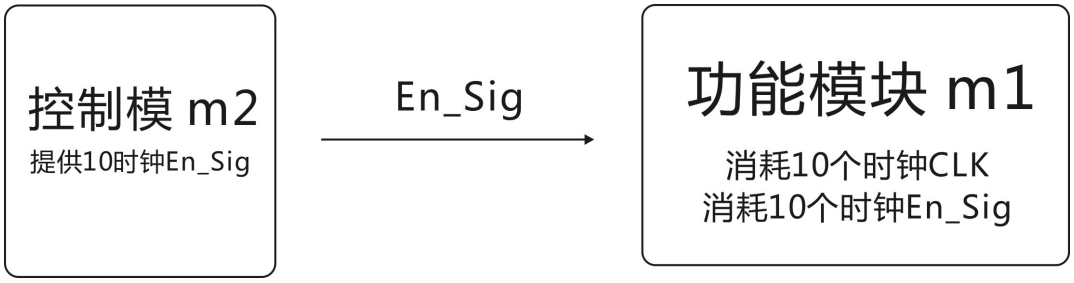


图 2.1d 从一个时钟一块数据

如果用时序的概念去理解的话，读者读起来是不是觉得很烦人？笔者也写到吐血 ... 反之从“一个时钟一块数据”的角度去理解的话：

我们知道“模块和模块之间的沟通至少需要一个时钟”这个硬道理，也就是说 En_Sig 会延迟一个时钟到达 m1。但是对于一个时钟一块数据来说，这个硬道理是“透明”的（不是不存在，而是用户不用去关心而已）。在此，从图 2.1d 中我们可以得知功能模块 m1 活动起来需要消耗 10 个时钟 CLK 和 10 个时钟 En_Sig。CLK 基本上是模块的心脏，CLK 的提供对于所有模块来说都是无限的动力资源。

但是对于 En_Sig 来说，它不是模块的心脏，它是使“模块活动”的钥匙，说得具体点“En_Sig 是 m1 的劳动工资”。换句话，如果 m2 要 m1 工作的话，m2 就必须提供“10 个时钟 En_Sig 的工资”给 m1，不然的话 m1 是绝对罢工的。

从打工仔和老板的角度来看的话，就像读者工作一个月老板就必须提供读者一个月的工资。这一天和“时序的概念”有点不同“时序的概念”有点像，老板给读者一天钱，然后读者就工作一天，其中老板还唠唠叨叨要这个，不要那个 .... 读者你说到底烦不烦呢？

“时序的概念”就是在每一个时钟里，都检查“模块之间的沟通”，因此用户必须看 m1，然后又看 m2。反之“一个时钟一块数据”，用户用不着过分把 m1 和 m2 这两个模块联系起来。用户仅需要知道 m2 不给 m1 工资，m1 就罢工，然后用户仅需要确保 m2 一方有好好给钱，m1 另一方收钱然后有好好工作就行了。

2.2 信号时钟化

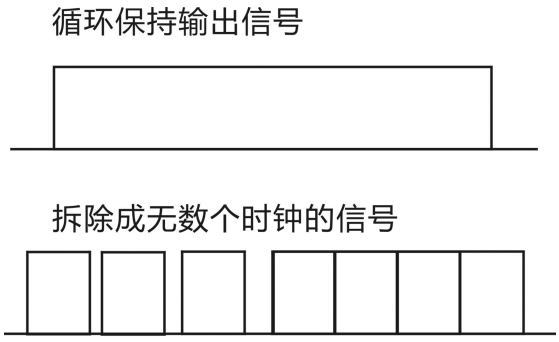


图 2.2a 把循环保留输出的信号，拆除致无数个时钟信号

在第一章-整合的概念中不是有一个叫做“循环保留”的家伙吗？循环保留，基本上就是“一个始终一块数据”。在这里，可能有同学会问：“给不给模块工资干活和循环保留有什么关系？”，这位同学难道就没有看出循环保留，实际上就是把一块又一块相同属性的数据串联起来的吗？

还记得实验三的串口发送实验吗？笔者先求得波特率的周期，然后用计数器去循环保留一帧的一位数据的周期长度。实际上，我们也可以用“一个时钟一块数据”的理解实验三的内容。

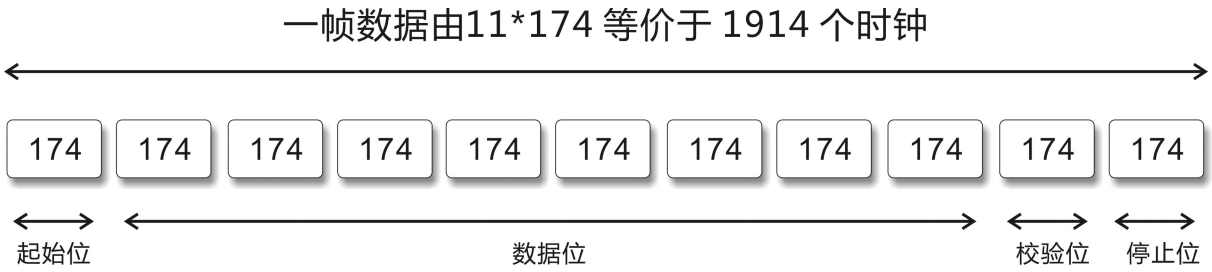


图 2.2b 由 11 组 174 个时钟组成的一帧数据

可能又有同学问到：“以 115k2 的波特率为主，一帧数据有 1914 个时钟，很普通呀！到底有什么稀奇？”对，乍看下实验三确实很普通，到实验 3 中所没有涉及的一个重点，就是“使某个长长的信号时钟化”——这一重点。如图 2.2b 所示，一帧数据可以是由 11 组由 174 个时钟组成。（抛开初始化和产生完成信号的步骤）。

图 2.2c 由 1914 个时钟组成的一帧数据

图 2.2c 是实验四中的串口发送模块，与实验三不同，它是直接由 1914 个时钟组成。

其中起始位，数据位，校验位和停止位的划分是由循环操作执行的。隐藏在实验三和实验四之间的重点就在于“只要某个信号可以时钟化，那么要怎样 YY 它都行”。
又可能有同学要问：“时钟化某个信号真的那么重要吗？”呵呵，这个不是重不重要的问题，第一读者们要明白，任何模块都依赖“时钟”这个心脏，或者说任何模块的活动都离不开时钟。对于一些“外来的数据”（有时候又称为异步信号），如果我们无法把它和模块们“同化”的话，即使设计者再怎么牛，都要费九牛二虎之力控制它。反之只要“外来数据”同化，即“时钟化”，那么“外来数据”自然而然就会成为模块们的一份子。用简单的话来说，外来数据同样可以用“时钟”这一单位去计算，评估和控制。

对于整合的概念来说，重点就是“如何去控制，计算，评估”这些“时钟”，所以说“时钟化”一个“外来的数据”，不仅会简化时序（波形图）分析，而且也会提高设计的表达能力，此外就衍生出而外的效果，如精密计数等等等。

2.3 深入了解“模块的沟通”

模块的沟通是时序篇里的一个重点，因为考虑到初学者对时序的初次认识，所以笔者简单的描述和讨论它而已。“模块的沟通”大体上的意思就是“两个模块之间如何协调操作”，更确切的意思是“一个模块如何把'信号'|'数据'传给另一个模块”。在早期间，笔者只是用时序的话来解释“模块之间到底如何沟通？”，但是不乏提到一个硬道理就是“模块和模块之间的沟通至少一个时钟”。

从这个硬道理里，我们可以知道这样一个“信息”就是“模块之间的数据|信号传输是按着时钟这个单位来计算”。“时钟信号就是模块的心脏”，同样这一句话（信息）也是笔者常常挂在嘴边。因为在 RTL 级的设计中，最好什么模块都要依赖“Clock—这个时钟信号”。

我们先看一个简单的例子：

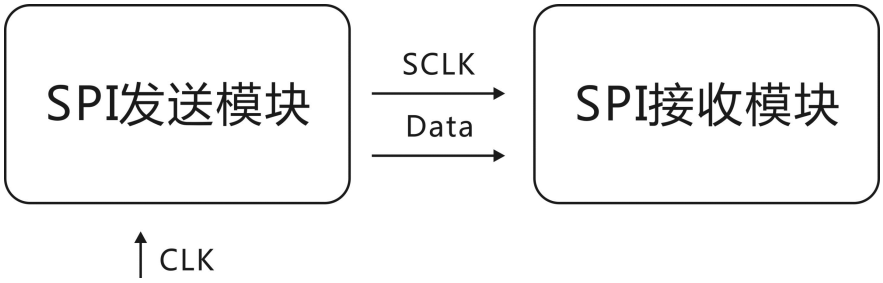


图 2.3a SPI 发送模块提供时钟给 SPI 接收模块。

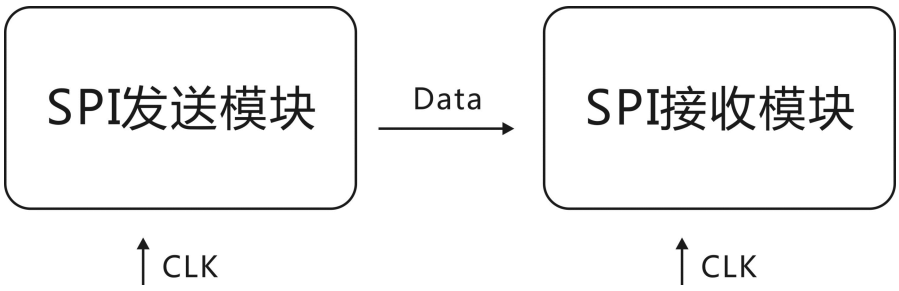


图 2.3b SPI 发送模块和 SPI 接收模块同用一个时钟。

图 2.3a 和 图 2.3b SPI 发送模块和 SPI 接收模块的内容都是一样的，唯一不同的是：图 2.3a 中的 SPI 接收模块所使用的时钟——SCLK，是由 SPI 发送模块提供的；反之图 2.3b 中的 SPI 接收模块和 SPI 发送模块同用一个 CLK。在某种意义上，图 2.3a 中的 SPI 接收模块被视为“被动者”或者“阶层级”，相反的图 2.3b 中的 SPI 接收模块和 SPI 发送模块均为“主动者”，均同一个等级，换句话说它们使用同样的“心脏”活动着。

可能有同学会问：“图 2.3a 的 SPI 接收模块到底有什么不妥？许多参考书也这样写 ...”这位同学你有所不知了，虽然图 2.3a 和图 2.3b 看似没什么两样，但是在时序分析上，图 2.3b 多么和谐和凸显出更多优势。理由很简单，图 2.3a 的两个模块我们很难把它们联系起来，反之图 2.3b 却很乐意。

接下来我们再看一个简单的例子：

```
always @ ( a_sig )
      case( a_sig )
            0000: Q .....
            ......
```

代码 2.3a

代码 2.3a 可以说在参考书里看到，其中一个典型的例子。代码大致上的意思是“当 a_sig 产生变化，根据 a_sig 变化，然后 Q 什么什么的”，话说这也是许多初学者喜欢“滥用”的写法之一，为什么笔者说滥用呢？读者尝试闭想象看看，类似的写法“像什么形状呢”？就是“组合逻辑”了....

组合逻辑基本上是触发“即时事件”，亦即无视时间点的操作。站在另一个角度来看的话，上述的代码是一个“被动者”，我们很难用 时钟信号——CLK 把它联系起来。

```
always @ ( posedge CLK )
      case( a_sig )
            0000: Q .....
            ......
```

代码 2.3b

反之上面的代码 2.3b，虽然功能上和代码 2.3a 大同小异，不过在重要的细节上和代码 2.3a 非常不同。代码 2.3b 大致的意思是“每一个时钟——CLK 的上升沿都检查 a_sig 的过去值，根据 a_sig 过去值的变化，然后 Q 决定什么什么 ...”。当读者听到“过去值”，“决定”等的字眼，自然而然会明白，这是以“时间点事件”为前提的东西。

“时间点事件”任何过程都依赖“时钟——CLK”，亦即“模块的心跳”。站在另一个角度来看，代码 2.3b 是一个“主动者”，它和时钟信号——CLK 的关系式紧密的。

可能会有同学很困惑的问道：“是谁规定‘时间点事件’就是正规的？难道‘即时事件'不可以在模块的沟通（时序）中出现吗？”这位同学，笔者很明白的你的困惑 ... 无论是建模篇还是时序篇，设计者要用“什么事件”都没有问题。原因都是非常直接的，在那时候，笔者不会去计算信号的长度，也不会过分评估时钟的消耗，笔者所关心的只是 Verilog HDL 语言的结构和如何去从模块的沟通记录中得出信息。

但是在整合篇里，时钟所占的戏份已经增长了。过多使用“组合逻辑”或者即时事件，尤其是模块之间的“沟通口”，会使得“时序”非常不和谐。在这里，读者只要明白“组合逻辑会引起即时事件，然而即时事件会无视时钟”这一信息就可以了。

之说以要把任何东西都和“时钟”都联系起来，是为了时序更好控制，时序更好解读，某快更好沟通。粗略点就像人类的世界，如果所有人共有一种语言，那么什么事情都很好解决，战争也不会发生，地球更和谐・这一道理。

==========

在前面笔者举了两个例子，都是“可以和时钟联系起来，对模块的沟通会带来好处”这些信息，接下来我们再来看看，从建模篇到时序篇，模块们到底有多少沟通的方式：

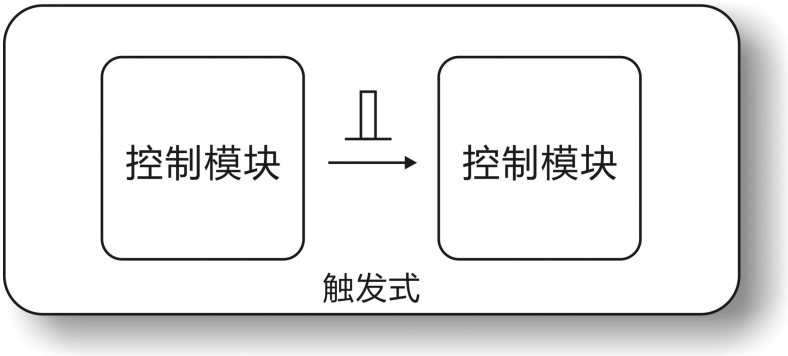


图 2.3c 触发式的沟通方式。

触发式的沟通方式可以说是模块最基础的沟通方式，触发式沟通基本上是由一个模块向一个模块发送“一个时钟长度”——“喂!”信号。从时序的角度来讲，这个“喂!”信号从一个模块到一个模块，需要一个时钟的延迟，这也是“模块的沟通”最基本和最基础的硬道理。但是从“一个时钟，一块数据”的角度来看的话，一个模块向另一个模块发送一个时钟长度的“启动——触发”信号。当某个模块接收到“启动——触发”信号以后，它就开始活动了。

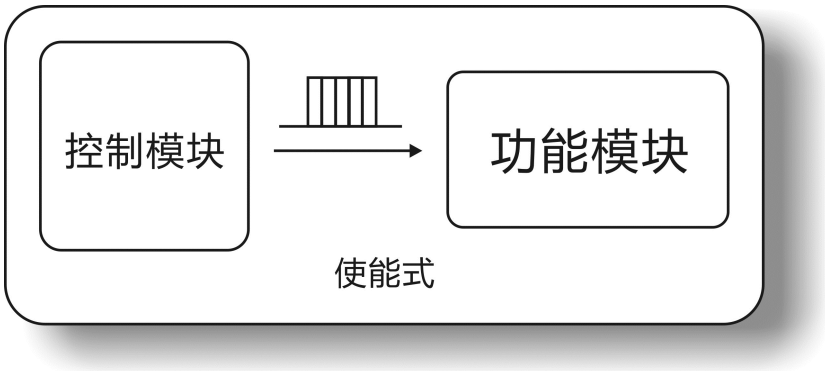


图 2.3d 使能式的沟通方式。

使能式的沟通方式基本上是某个模块向另一个模块发送“一串包含许多个时钟”——“有钱就继续工作”信号。使能式的沟通方式，一般上都不被设计者喜欢，理由很简单:
当我们要使能某个模块的时候，我们就必须清楚 “使能对象”到底“消耗多少时钟”。关于这一点，对于很多人来说是非常头疼的，基本上已经涉及了 Verilog HDL
语言的“写法”和“结构”等技巧，故它不怎么受欢迎。在某个方面上，使能式的沟通方式就像老板给用工薪水干活那样一笑。

从“一个时钟，一块角度”的角度来讲，使能信号一般上都是有“许多个时钟”，“时钟的个数”代表了使能对象的“一次性操作所消耗的时钟量”。

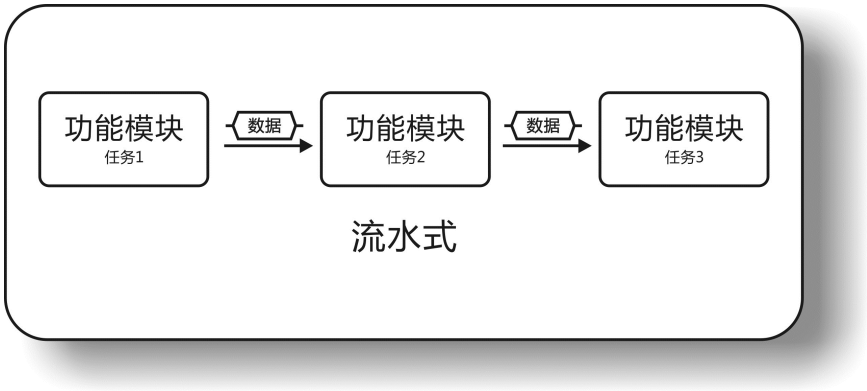


图 2.3e 流水式的沟通方式。

流水式可以说是最机械和最单调的沟通方式。它和触发式的沟通方式有几分相识，但是一个模块向另一个模块传输的是“数据”而不是“信号”。从时序上理解，数据之间的传输有一个时钟的延迟，反之从“一个时钟，一块数据”去理解的话：“上一个模块，向下一个模块发送一个时钟长度的数据”。

流水式的沟通方式原本是最适合“时间紧张”的设计，这里所谓的“时间紧张”是“每一个时钟，每一个任务，都要工作”的意思。流水式的沟通方式，数据永远只是向一个方向移动，所以说它非常适合“数据流”而不适合“控制流” ... 因为流水式的操作有潜伏期，方向单一等缺点。此外流水式操作，是一种“用空间换时间”的设计，故非常消耗资源。

除此之外，模块“普遍沟通都可以说成是流水式沟通，但是流水式的长度只是 1 而已”。就好比触发式的沟通方式，也可以说成流水式沟通，因为一个时钟长度的触发信号也是用了一个时钟去传播，不同的只是在理解上的差别而已。

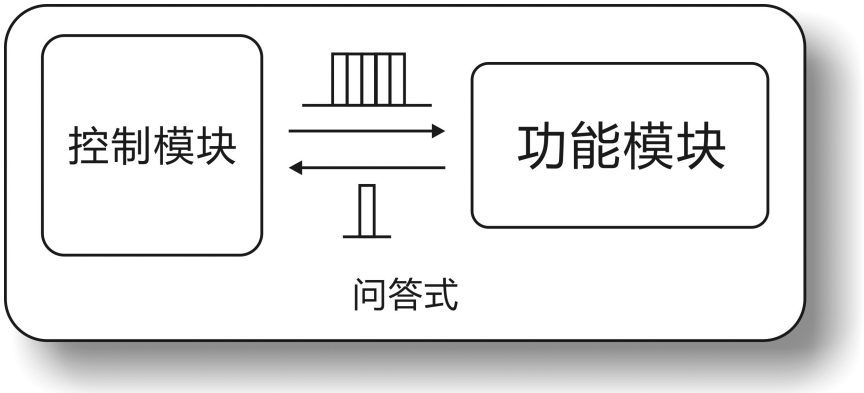


图 2.3f 问答式的沟通方式。

问答式的沟通方式估计读者都不陌生吧？笔者的任何笔记里都会出现它的声影，一个经典的例子就是 Start_Sig 和 Done_Sig。它对于笔者的“仿顺序操作”来说是一个重要的基础，此外也是“仿顺序操作——建模”中的一个模块沟通方式。问答式基本上是一个模块使能另一个模块，而且不用知道使能对象到底消耗了多少时钟。当使能对象完成一

次性的操作以后就会反馈信号给使能的模块。

如果要分析问答式的时序活动的话，是非常猥琐的 .... 但是从“一个时钟，一块数据”的角度来看的话：一个模块提供了一个模块活动所需要的“工资”，这个“工资”的“长度”是根据“使能对象自身一次性操作所消耗的时钟数”为主。当使能对象完成一次性的“工作以后”，就会反馈“一个时钟长度”的“完成信号”给使能模块。（触发式和使能式的集合体）

这种的沟通方式的好处估计不用笔者详解了吧？它除了用法简单以外，常常被视为“外部子状态机”。但是它也不是完美的，使能对象要用而外的 2 个时钟作为反馈信号的时钟消耗。

==========

到目前为止，笔者介绍了 4 中笔者所知——模块的沟通方式。除了“使能式”的沟通方式不常使用以外，其他的沟通方式使用频率也挺多的，尤其是 “问答式”。在这里，笔者秀出 4 中的沟通方式，不是纯属的给读者 “看花”而已。笔者的目的是要读者知道，“模块的沟通”在 Verilog HDL 语言设计中是一个重要的基础，然而“模块的沟通”更多时候是用“时钟”这个单位去“计算，分析和解读”。这个“时钟”的单位，就是源于“模块的心跳”亦即时钟信号——CLK。

这也是为什么笔者如此“执着”——“什么东西都要和时钟联系起来”的原因。在设计中，无论是任何形式的沟通方式，都逃不过“时钟”这个单位和概念。更重要的是，在时序上“时钟”都是以“时间点事件”的形式出现。笔者常说“即时事件”是无视“时钟”，如果把“即时事件”（组合逻辑）作为，设计的基础，模块沟通的基础，对于笔者来说是非常不和谐的。

2.4 电平检测模块的整合（即时事件在时序上的不和谐）

图 2.4a 电平检测模块

电平检测模块，看过建模篇的朋友估计一点也不陌生吧。在建模篇里，笔者常常把电平检测模块独立建模为一个模块。电平检测模块的主要功能就是检测电平的变化，然后如果电平由低变高就向 L2H_Sig 产生一个高脉冲；如果电平有高变低就向 H2L_Sig 产生一个高脉冲。（如图 2.4a 所示）

忽然间有同学问道："电平检测模块和第二章的大体又有什么关系？"这位同学果然很心急，这也是接下笔者要说明的东西。那些读过建模篇的朋友，一开始的时候可能会觉得很疑惑，为什么电平检测模块内容那么小，为什么当初不把它嵌入在其他模块中呢？读者有所不知了，电平检测模块看似简单，实际上它是披着羊皮的狼。

早期笔者没有能力解释电平检测模块在时序上的行为，所以就"一笑带过"。如今要把电平检测模块整合在某个模块里，我们就必须深入了解电平模块几个容易搞糊涂的重点。电平检测模块会随着用法不同，理解的程度也不同，而且行为也不同。

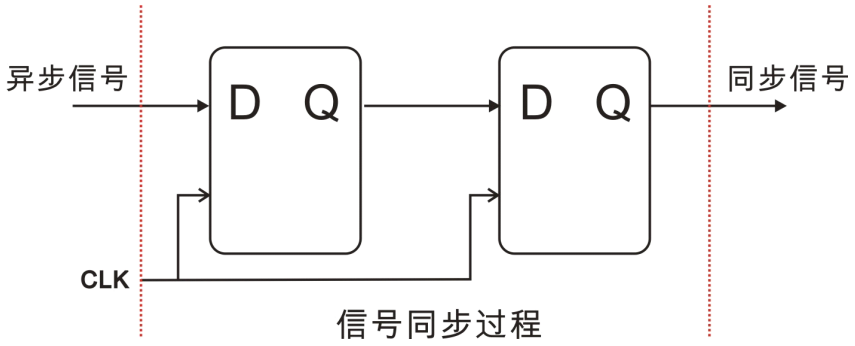


图 2.4b 异步信号同步过程

在物理的行为上，电平检测模块有同步化异步信号的能力。关于这一点，那些读过"数字系统"的同学估计都这个道理，因为 FPGA 把所有外部信号，都视为"异步信号"。（这里"异步"的定义是"频率不同"）信号"异步"的行为会引起"亚稳态"等问题。关于"亚稳态"就是信号处于"逻辑 1 和逻辑 0 的中间"，这个问题主要是寄存器的"建立时间"和"保持时间"不足。关于"建立时间"和"保持时间"笔者就不多解释了。

虽然电平检测模块有解决亚稳态发生的效果，但是更多时候，引起"亚稳态"的问题源本身是来至实际的物理路劲，更贴切的说是"物理时序不合格"。好了更多关于"亚稳

态”的故事，笔者就不谈了 ... 自己谷歌或者百度一下。

在这里，笔者需要再一次强调，在 Verilog HDL 语言的世界里，Verilog HDL 语言是完美而且无瑕疵的工具。虽然电平检测模块有同步化信号的功能，但是不是重点，然而重点是电平检测模块在“模块的沟通”中，到底有什么稀奇的部分？

实验八：电平检测模块

detect_module.v

```
module detect_module
(
        input CLK,
        input RSTn,
        input Pin_In,
        output H2L_Sig,
        output L2H_Sig,

        /*****************/
        output SQ_F1,SQ_F2
        /*****************/
);

/**********************************/

        reg F1,F2;

        always @ ( posedge CLK or negedge RSTn )
                if( !RSTn )
                        begin
                                F1 <= 1'b1;
                                F2 <= 1'b1;
                        end
                else
                        begin
                                F1 <= Pin_In;
                                F2 <= F1;
                        end

/**********************************/

          assign H2L_Sig = F2 && !F1;
          assign L2H_Sig = F1 && !F2;
```

```
34.
35.         //simulation use
36.         assign SQ_F1 = F1;
37.         assign SQ_F2 = F2;
38.
39.  /*********************************/
40.
41.  endmodule
```

上述的代码是不是很怀念？呵呵呵，笔者也觉得是。这是典型的电平检测模块的内容，第 3~11 行是端口的声明，然而第 10 行是仿真用的输出。第 16 行使寄存器声明，均为 1 位。第 18~33 行是电平检测模块的核心功能，具体的思路笔者已经在建模篇里说过了，不明白的朋友请复习去。第 35~38 行是仿真输出用，驱动源分为 F1 和 F2。

这个电平检测模块所检测的电平，在空闲的时候总是拉高电平，所以在 21~22 行的复位中需要把 F1 和 F2 复位为逻辑 1。第 32 行的 H2L_Sig 主要是当 Pin_In 的电平从高变低就会产生“一个时钟长度的高脉冲”，当 Pin_In 的电平从低变高的时候就会产生“一个时钟长度”。

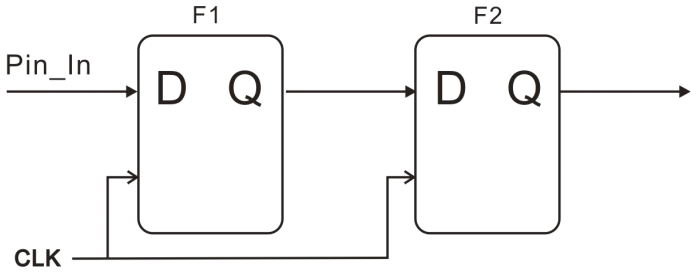


图 2.4c 2 个深度的移位寄存器

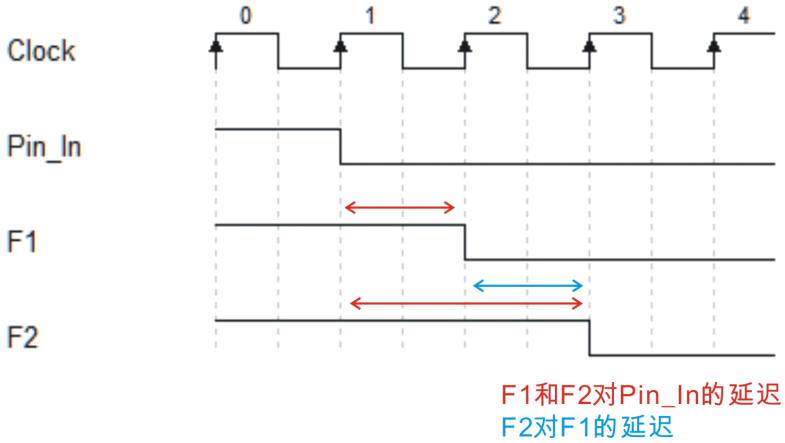


图 2.4d　F1，F2 和 Pin_In 之间的延迟

第 25~28 行基本上就是深度为 2 的移位寄存器，F1 从 Pin_In 读入数据，F2 从 F1 读入数据。然而 H2L_Sig 和 L2H_Sig 会因 F1 和 F2 的状态而产生反应。从另一个角度来看的话，F1 的结果会是 Pin_In 延迟的 1 个时钟，F2 的结果会是 Pin_In 的延迟 2 个时钟，然后 F2 的结果又是 F1 延迟的 1 个时钟。如图 2.4d 所示。

如果用时序的话来解释图 2.4d ，当 T1 的时候 F1 读取 Pin_In 的过去值，F2 读取 F1 的过去值，所以在 T1 的未来，F1 输出和 F2 均输出逻辑 1。当 T2 的时候，F1 读取 Pin_In 的过去值，F2 则读入 F1 的过去值，所以在 T2 的未来，F1 输出逻辑 0，F2 输出逻辑 1。在 T3 的时候，F1 读取 Pin_In 的过去值，F2 则读入 F1 的过去值，所以在 T3 的未来，F1 输出逻辑 0，F2 也是输出逻辑 0。

其实笔者要谈论的重点就是发生在 32~33 行的 H2L_Sig 和 L2H_Sig 的输出。笔者还记得在时序篇里，曾经有讨论过关于“精密计数”。内容中的定时器，有两个不同驱动源的输出——组合逻辑和寄存器。但那时候，笔者只是单纯的列出“即时时间”和“时间点时间”的概念而已。

读者知道吗？组合逻辑驱动的输出源，很多时候在模块的沟通记录（时序）里“很难捉拿”。就是这个“捉拿点”有点别扭的关系，导致模块和模块之间的沟通出现“不和谐。在把电平检测模块整合入某个模块之前，我们有必要清楚知道“到底哪里不和谐”呢？

detect_module.vt

```
1.    `timescale 1 ps/ 1 ps
2.    module detect_module_simulation();
3.
4.          reg CLK;
5.          reg Pin_In;
6.          reg RSTn;
7.
8.          wire H2L_Sig;
9.          wire L2H_Sig;
10.
11.         /******************/ //simulation use
12.         wire SQ_F1;
13.         wire SQ_F2;
14.         /******************/
15.
16.       /************************/
17.
18.         detect_module U1
19.         (
20.               .CLK( CLK ),
21.               .RSTn( RSTn ),
22.               .Pin_In( Pin_In ),
23.               .H2L_Sig( H2L_Sig ),
24.               .L2H_Sig( L2H_Sig ),
25.               /******************/
```

```
26.                 .SQ_F1( SQ_F1 ),
27.                 .SQ_F2( SQ_F2 )
28.                 /******************/
29.           );
30.
31.           /**********************/
32.
33.           initial
34.           begin
35.                   RSTn = 0; #1000 RSTn = 1;
36.                   CLK = 1; forever #5 CLK = ~CLK;
37.           end
38.
39.          /**********************/
40.
41.          reg [3:0]i;
42.
43.           always @( posedge CLK or negedge RSTn )
44.                 if( !RSTn )
45.                         begin
46.                                 Pin_In = 1'b1;
47.                                 i <= 4'd0;
48.                           end
49.                   else
50.                         case( i )
51.
52.                                   0,1:
53.                                    begin Pin_In <= 1'b1; i <= i + 1'b1; end
54.
55.                                    2:
56.                                    begin Pin_In <= 1'b0; i <= i + 1'b1; end
57.
58.                                    3,4:
59.                                    begin i <= i + 1'b1; end
60.
61.                                    5:
62.                                    begin Pin_In <= 1'b1; i <= i; end
63.
64.                           endcase
65.
66.           /************************************/
67.
68.   endmodule
```

上面的代码是电平检测模块的 .vt 文件。第 18~29 行是仿真对象的实例化，第 33~37 行是复位信号和时钟信号的产生。第 41~64 行是激励的经过，在复位时候 Pin_In 是赋予逻辑 1，在 T0~T1 的时候（步骤 0~1—52~53 行）拉高 Pin_In 两个时钟。在 T2 的时候（步骤 2—55~56 行）拉低 Pin_In，然后在 T3~T4 的时候（步骤 3~4—58~59 行）延迟 2 个时钟。在 T5 的时候（步骤 5—61~62 行）永远拉高 Pin_In。

仿真结果：

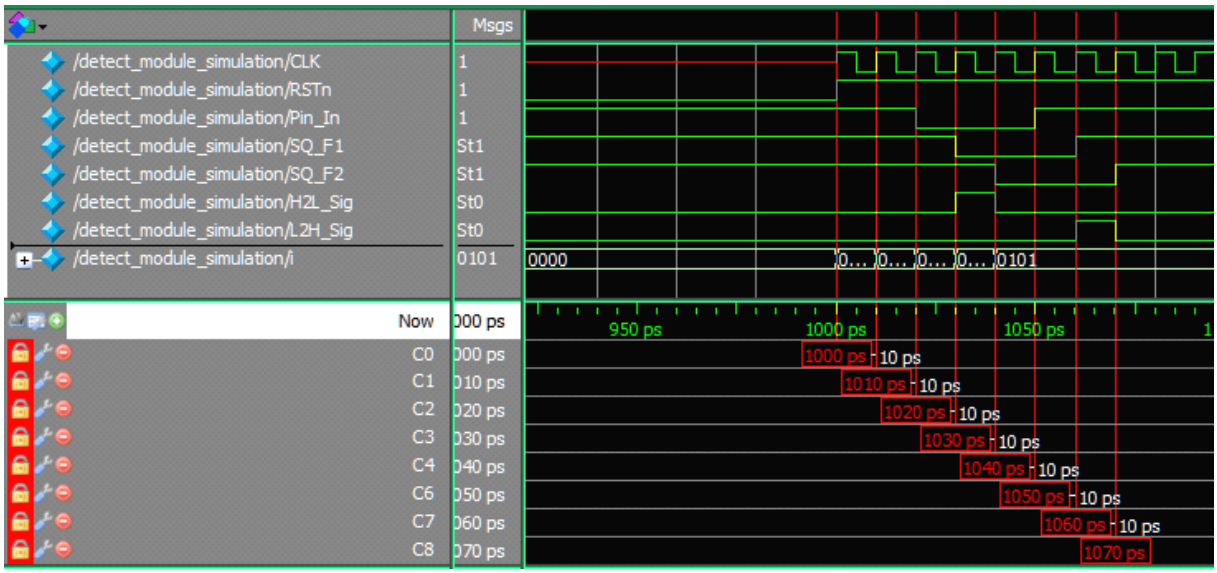


图 2.4e 实验八仿真结果

图 2.4e 是实验八的仿真结果。读者注意 Pin_In ，SQ_F1 和 SQ_F2 之间的时序，SQ_F1 延迟与 Pin_In 结果的一个时钟（C1~C2），SQ_F2 延迟与 Pin_In 结果的两个时钟（C1~C3），SQ_F2 也延迟与 SQ_F1 的一个时钟（C2~C3）。一些猥琐的过程笔者就不解释了，我们直接来看重点。

在 C4 的时候 .v 文件检查到 Pin_In 从高电平变成低电平。期间 SQ_F1 的过去值是 0，SQ_F2 的过去值是 1。在这里奇怪的事情发生了！在前面笔者说过 H2L_Sig 是由组合逻辑驱动，然而组合逻辑是不遵守“时间点事件”，不遵守“时间点事件”的信号源，永远都是“不和谐”的凶手。至于笔者为什么说 H2L_Sig 不和谐呢？

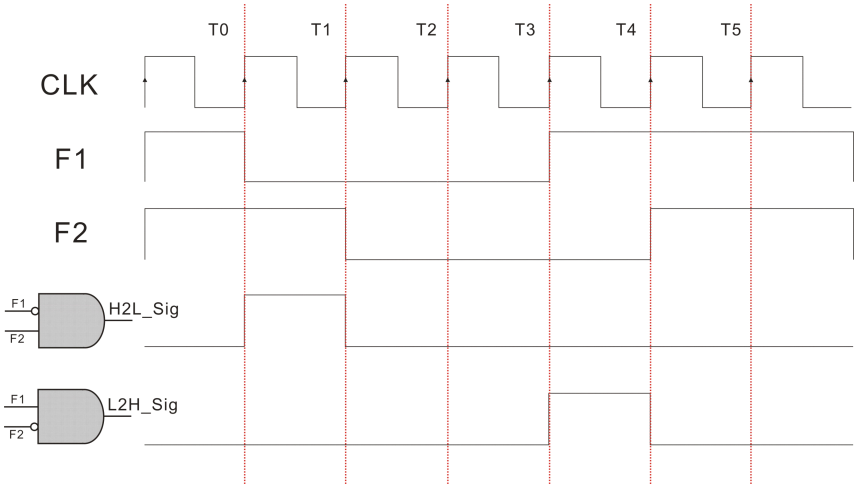


图 2.4f C2~C8 的部分放大

图 2.4f 是仿真图的部分放大，不和谐的地方时发生在 T0 和 T1，然后还有 T3 和 T4。大伙还记得“时间点事件”的概念吗？“某某判断过去值，然后‘决定’干什么，未来发生什么 ... ”。在 T0 的时候 H2L_Sig 判断 F1 和 F2 的过去值，F1 和 F2 均为 1，但是 H2L_Sig 输出一个高脉冲。在 T1 的时候 H2L_Sig 判断 F1 和 F2 的过去只，F1 为 0, F2 为 1，H2L_Sig 应该输出高脉冲，怎么在 T1 的未来 H2L_Sig 输出低电平了？

同样的问题也发生在 T3 和 T4。在 T3 的时候，L2H_Sig 判断 F1 和 F2 的过去值，F1 和 F2 均为 0,但是在 T3 的未来里 L2H_Sig 确输出一个时钟的高脉冲。在 T4 的时候 L2H_Sig 判断 F1 和 F2 的过去值，F1 为 1，F2 为 0，H2L_Sig 有理由输出高脉冲才对，怎么拉低输出了。

在这里，可能有同学会问“是不是 modelsim 有问题？如果 modelsim 没有问题，那么是不是笔者的 '时间点事件' 概念有问题？”。说实在，这些提问也是笔者早期的问题 ... 其实 modelsim 没有错，笔者的概念也没有问题。

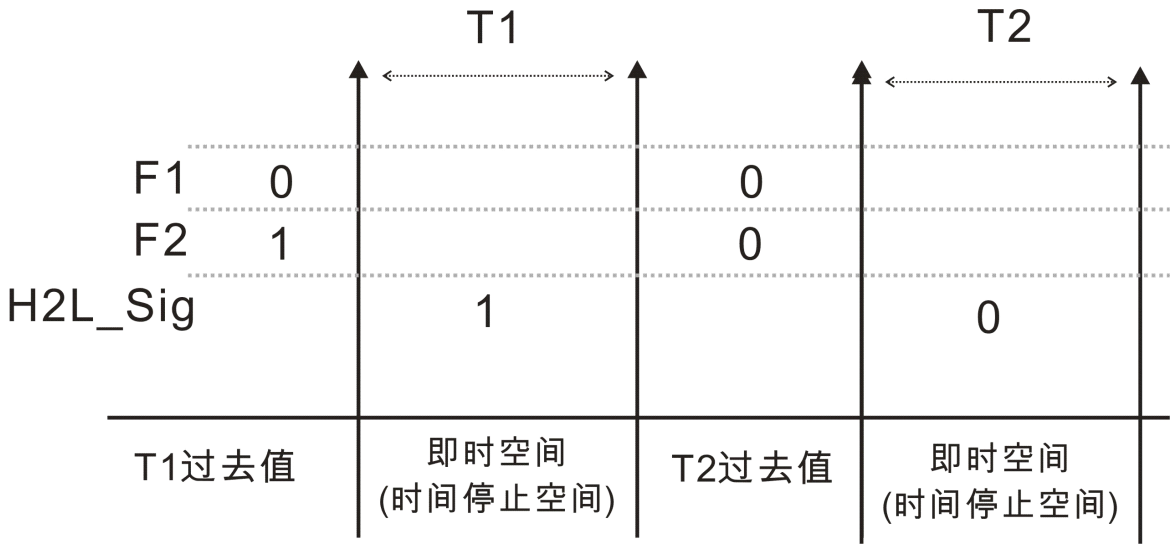


图 2.4g H2L_Sig 即时事件的概念图。

第一我们必须回忆起“组合逻辑引发即时事件”，换句话说“即时事件”是无视“时间点”。如图 2.4g 所示，在 T1 的时候 H2L_Sig 判断 F1 和 F2 的过去值，结果 F1 为 0，F2 为 1，在 T1 这个空间里，产生出即时空间（时间停止空间），然后 H2L_Sig 取得即时结果 1。在 T2 的时候，H2L_Sig 判断 F1 和 F2 的过去值它们均为 0，然后在 T2 的即时空间里，H2L_Sig 取得即时结果 0。

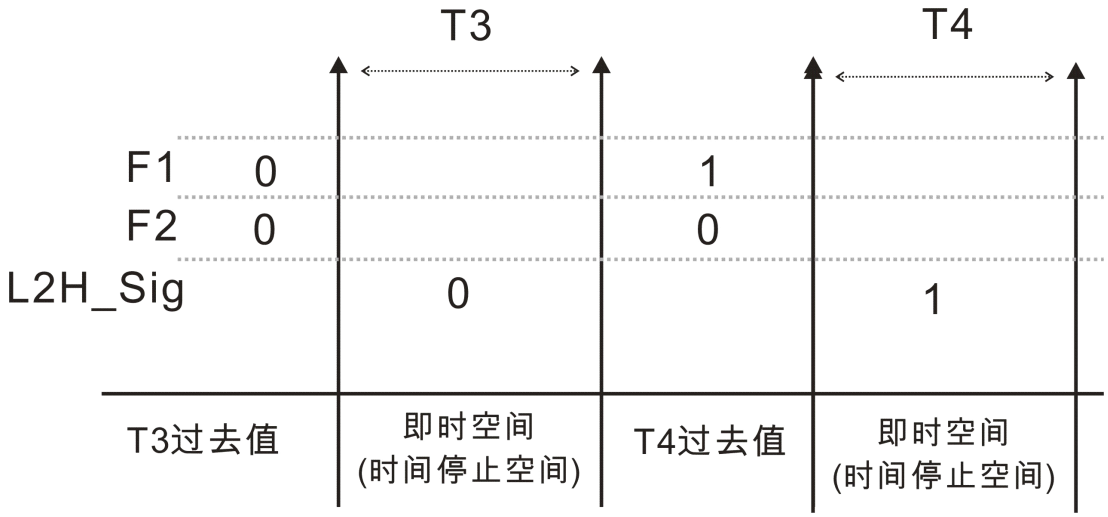


图 2.4h L2H_Sig 即时事件的概念图。

图 2.4h 是 L2H_Sig 即时事件的经过。在 T3 的时候 L2H_Sig 判断 F1 和 F2 的过去值，F1 和 F2 的结果均为 0，在 T3 的即时空间咯，L2H_Sig 取得即时结果 0。在 T2 的时候, L2H_Sig 判断 F1 和 F2 的过去值，F1 为 1，F2 为 0，然后在 T4 的即时空间里，L2H_Sig 取得即时结果 0。

不知道读者们是否已经注意到，所谓即时事件就是某个时间点里的 “时间停止空间”。用数字电子的话来说，就是“无视时钟产生的结果”，或者“在时钟沿发生的那一刻所产生的即时结果”。在这里，我们要知道 Modelsim 虽是一个仿真软件，但是它不是万能的，Modelsim 它不可能把某个“时间沿的中间拨开，然后贴上‘即时时间’的标签”。不不不！Modelsim 做不到，它也没有理由去做 ....

读者还是不明白笔者在说什么吗？好，我们换另一个角度去理解即时事件。

假设笔者把电平检测模块嵌入某个模块里：

exp2_4a.vt

```
1.    `timescale 1ps/1ps
2.    module exp2_4a();
3.
4.      reg CLK;
5.      reg RSTn;
```

```
6.      reg Pin_In;
7.
8.      /*******************************/
9.
10.     initial
11.     begin
12.             RSTn = 0; #10 RSTn = 1;
13.             CLK = 1; forever #5 CLK = ~CLK;
14.     end
15.
16.     /*******************************/
17.
18.     reg [3:0]i;
19.
20.     always @ ( posedge CLK or negedge RSTn )
21.             if( !RSTn )
22.                     begin
23.                             i <= 4'd0;
24.                             Pin_In <= 1'b1;
25.                     end
26.             else
27.                     case( i )
28.
29.                             0,1:
30.                             begin Pin_In <= 1'b1; i <= i + 1'b1; end
31.
32.                             2,3:
33.                             begin Pin_In <= 1'b0; i <= i + 1'b1; end
34.
35.                             4,5:
36.                             begin Pin_In <= 1'b1; i <= i + 1'b1; end
37.
38.                             6,7:
39.                             begin Pin_In <= 1'b0; i <= i + 1'b1; end
40.
41.                             8,9:
42.                             begin Pin_In <= 1'b1; i <= i + 1'b1; end
43.
44.                             10:
45.                             begin Pin_In <= 1'b0; i <= i; end
46.
47.                     endcase
48.
49.     /*********************************************************/
```

```
50.
51.     reg F1,F2;
52.
53.     always @ ( posedge CLK or negedge RSTn )
54.         if( !RSTn )
55.             begin
56.                 F1 <= 1'b1;
57.                 F2 <= 1'b1;
58.             end
59.         else
60.             begin
61.                 F1 <= Pin_In;
62.                 F2 <= F1;
63.             end
64.
65.     wire H2L_Sig = F2 && !F1;
66.     wire L2H_Sig = F1 && !F2;
67.
68.     /*****************/
69.
70.     reg [3:0]j;
71.     reg [1:0]rQ;
72.
73.     always @ ( posedge CLK or negedge RSTn )
74.         if( !RSTn )
75.             begin
76.                 j <= 4'd0;
77.                 rQ <= 2'd0;
78.             end
79.         else
80.             case( j )
81.
82.                 0:
83.                 if( H2L_Sig ) begin rQ <= 2'd1; j <= j + 1'b1; end
84.
85.                 1:
86.                 if( L2H_Sig ) begin rQ <= 2'd2; j <= j + 1'b1; end
87.
88.                 2:   // F2 Previos state, F1 Current state
89.                 if( F2 == 1 && F1 == 0 ) begin rQ <= 2'd3; j <= j + 1'b1; end
90.
91.                 3:   // F2 Previos state, F1 Current state
92.                 if( F2 == 0 && F1 == 1 ) begin rQ <= 2'd0; j <= j +1'b1; end
93.
```

```
94.                     4:
95.                     j <= j;
96.
97.                 endcase
98.
99.   endmodule
```

第 1~14 行使基本配置，复位时间为 10ps，时钟信号为 10ps 周期（随便填也行）。第 18~47 行是基本的 Pin_In 刺激行为。复位的时候 Pin_In 为高电平（第 24 行），然后在 T0~T1（步骤 i-0~1）拉高 Pin_In，T2~T3（步骤 i-2~3）拉低 Pin_In，T4~T5（步骤 i-4~5）拉高 Pin_In，T6~T7（步骤 i-6~7）拉低 Pin_In，T8~T9（步骤 i-8~9）拉高 Pin_In，其后就一直拉低 Pin_In。

第 51~66 行是电平检测模块（笔者懒得介绍了），F2 寄存“之前”的 Pin_In 状态，F1 则寄存“当前”的 Pin_In 状态。第 70~91 行是操作过程。第 70 行的 j 为此操作过程的步骤。

（一）在步骤 0 的时候，if 条件是利用 H2L_Sig 信号来检测 Pin_In 由高变低的变化，如果 if 条件成立就递增 j，然后 rQ 赋予 1 值；
（二）在步骤 1 的时候，if 条件是利用 L2H_Sig 信号来检测 Pin_In 由低变高的变化，如果 if 条件成立就递增 j，然后 rQ 赋予 2 值；
（三）在步骤 2 的时候，if 条件直接利用 F2 和 F1 来检测 Pin_In 由高变第的变化，如果 if 条件成立就递增 j，然后 rQ 赋予 3 值；
（四）在步骤 3 的时候，if 条件直接利用 F2 和 F1 来检测 Pin_In 由低变高的变化，如果 if 条件成立就递增 j，然后 rQ 赋予 0 值；
（五）在步骤 4 的时候，操作原地打转。

第 82~95 行的重点就是，步骤 0~1 是使用 xx_Sig 信号，反之步骤 2~3 是直接使用 F1 和 F2 寄存器。

仿真结果：

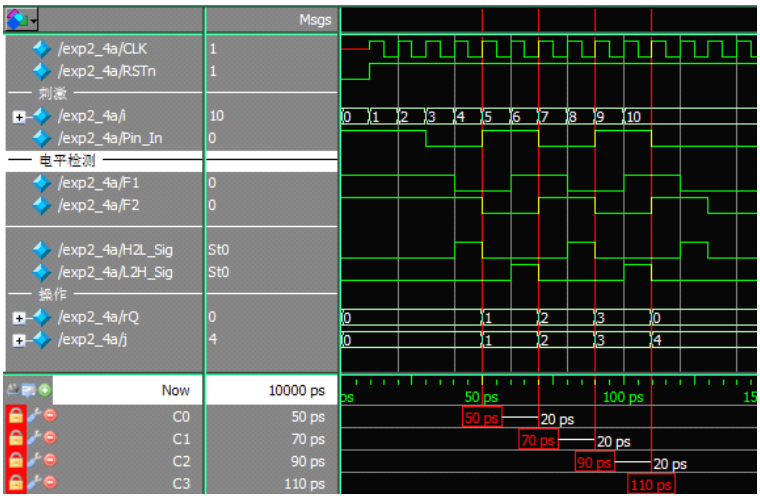


图 2.4i 仿真结果

笔者直接进入重点吧。C0 是指向 T4 步骤 i-4，同样也指向步骤 j-0。大家稍微注意一下此时的 H2L_Sig，是不是觉得很奇怪，H2L_Sig 早一个时钟拉高，这也使得步骤 j-0 的 if 条件可以检测到 Pin_In 由高变低，然后“决定”递增 j 和赋值 1 给予 rQ。所以在 T4 的未来，j 和 rQ 一同输出 1。
C1 是指向 T6 步骤 i-6，同样也指向步骤 j-1。大家稍微注意一下此时的 L2H_Sig，同样，L2H_Sig 也是早一个时钟拉高，这也使得步骤 j-1 的 if 条件可以检测到 Pin_In 由低变高，然后“决定”递增 j 和赋值 2 给予 rQ。所以在 T6 的未来，j 和 rQ 一同输出 2。

C2 是指向 T8 步骤 i-8，同样也指向步骤 j-2。在这个时间点，步骤 j-2 的 if 条件直接判断 F2 和 F1 的过去值，F2 表示 Pin_In 的之前状态，F1 表示 Pin_In 的当前状态，F2 的过去值为 1，F1 的过去值为 0，亦即 Pin_In 由高变低。if 条件成立，然后“决定”递增 j 和赋值 3 给予 rQ。所以 T8 的未来，j 和 rQ 一同输出 3。

C3 是指向 T10 步骤 i-10，同样也指向步骤 j-3。在这个时间点，步骤 j-3 的 if 条件直接判断 F2 和 F1 的过去值，F2 表示 Pin_In 的之前状态，F1 表示 Pin_In 的当前状态，F2 的过去值为 0，F1 的过去值为 1，亦即 Pin_In 由低变高。if 条件成立，然后“决定”递增 j 和赋值 0 给予 rQ。所以 T10 的未来，j 输出 3，rQ 输出 0。

exp2_4a.vt 中所包含的重要信息就是：

由于 Modelsim 软件无法绘出组合逻辑引发的“即时事件”，所以把“即时事件”的输出都提早一个时钟。H2L_Sig 和 L2H_Sig 就是例子。在这里，虽说步骤 j-0 和步骤 j-2 是等价，步骤 j-1 和步骤 j-3 的效果是等价。但是唯一不同的是，步骤 j-2 和 3 是直接判断 F2 和 F1 这个寄存器，而不是借用 H2L_Sig = F2 && !F1 和 L2H_Sig = !F2 && F1 等“组合逻辑”所取得的“即时结果”。由于 Modelsim 真的没有能力绘出“即时结果”，但是 Modelsim 又要把 H2L_Sig 和 L2H_Sig 放在波形图上，而且又不搅乱时序的正确性。所以才把 H2L_Sig 和 L2H_Sig 提早一个时钟触发。这也是笔者所谓“时序的不和谐”现象。

2.5 电平检测模块的整合（即时事件在模块沟通上的不和谐）

在 2.4 小节中，笔者详细了讨论组合逻辑——“即时事件”在时序上的不和谐。即时事件的本质是“无视时钟”，但是在波形图上（Modelsim）却绘不出“即时事件”，所以才把即时事件提前一个时钟输出。事实上，笔者对于这个问题困扰许久，那些组合模块引起的即时事件，如果把它引出来然后输出在模块之间“沟通口”，在“模块的沟通”中，是一个 BUG 来的。

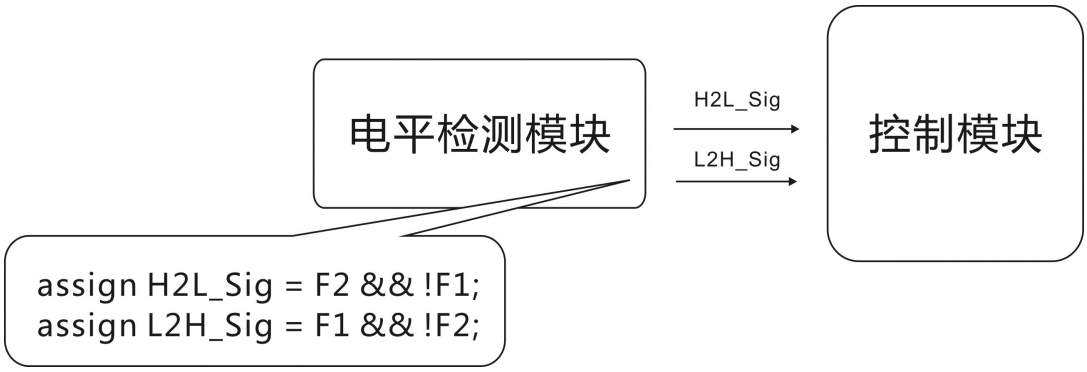


图 2.5a 电平检测模块和控制模块的（沟通）建模图。

在建模的角度上，图 2.5a 是一副再正常不过的“图形”。电平检测模块将 H2L_Sig 和 L2H_Sig 输出致控制模块的。但是当我们了解到“模块的沟通”的硬道理，我们不得不开始怀疑了 ...

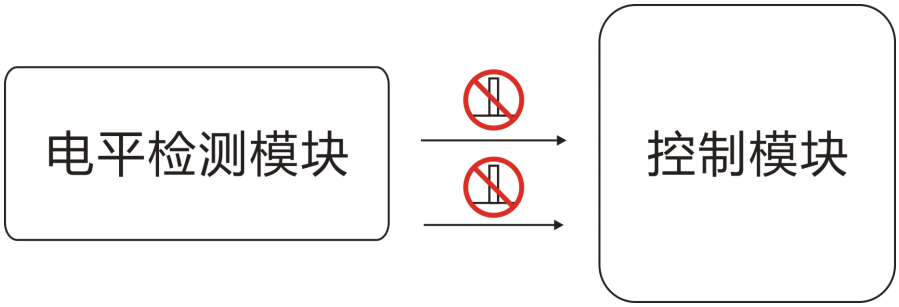


图 2.5b 电平检测模块沟通口的驱动源是组合逻辑。

因为电平检测模块向控制模块的沟通口的驱动源是组合逻辑，然而组合逻辑所触发的是“即时事件”。换句话说，图 2.5a 虽说电平检测模块向控制模块发送什么，实际上电平检测模块并没有向控制模块输出什么。在前面笔者已经是说过了，即时事件是“无视时钟”的事件，即时事件不会发送“一个时钟长度信号”等什么的 ...

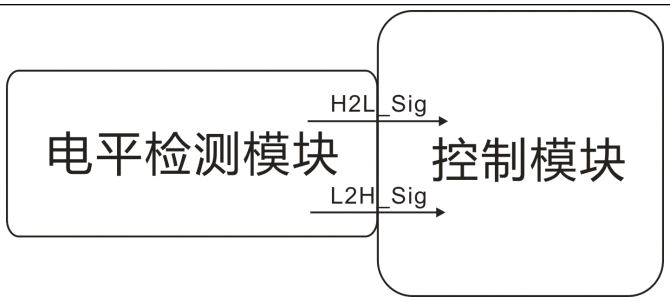


图 2.5c 与图 2.5b 等价的“图形”。

如果从另一个角度来看的话，那么电平检测模块和控制模块之间的沟通（建模图）会是如图 2.5c 所示。因为电平检测模块实际上没有向控制模块输出信号（沟通），所以它们俩个理应是贴在一起。

图 2.5d 把电平检测模块整合进入控制模块。

与其留着图 2.5c 的“囧样”，还不如彻底把电平检测模块整合进来，如图 2.5d 所示。在图 2.5d 中，电平检测模块已经是控制模块的一部分，应该说是“同化”了。

在这里，可能有些同学终于按不住性子叫道：“笔者你带我们绕了地球一大圈，最后还不是回到‘单文件主义’不是吗？”，然后继续道：“你怎么不一开始就直接告诉把电平检测模块的内容‘粘贴’到控制模块里边，不就好了吗？何必浪费自己的口水和我们的时间呢？...”

笔者说呀这位同学，不要把笔者和那种没有艺术性质的单文件主义混为一谈。首先读者需要明白到，建模是 Verilog HDL 语言的结构，时序是 Verilog HDL 语言的沟通本质，然而整合就是这两个概念结合的“轮滑剂”。当然“整合”的概念不是几行汉字就可以交代清楚的事情，如果是那样的话笔者就不写这本笔记了。

..... 好了，沉闷的话题我们先抛开不说，在这里，我们只要知道一个事实，就是：“当两个模块之间的沟通口，如果一方是由组合逻辑来驱动的话，那么就有整合的可能性。”到目前为止，我们已经讨论许多概念了，接下来笔者还是以实例来说话吧。不然的话，笔者一定会被读者怀疑致死。

2.6 实验九：串口接收模块

经过 2.1~2.3 小节的洗礼，读者是不是对“一个时钟，一块数据”的概念有点了解了。在 2.4~2.5 小节里，笔者以“电平检测模块”为例，说明了“即时事件”对模块之间的沟通产生的“不和谐”现象。在这个实验里，我们不仅要用“一个时钟，一块数据”的思维，去理解串口模块之间的沟通（数据之间的传输），此外还要用“时钟化”的概念，把电平检测模块整合在串口接收模块里。

途中读者可能会觉得很困惑 ... 这也难怪的，以往的实验里有“即时事件”又有“时间点事件”。但是在实验九里，“任何东西都是以时钟去实现”，在时序里“任何东西都以时间点去理解”。所以，读者要很耐性的慢慢去明白实验九中的每一个小细节。废话不多说，我们开始吧。

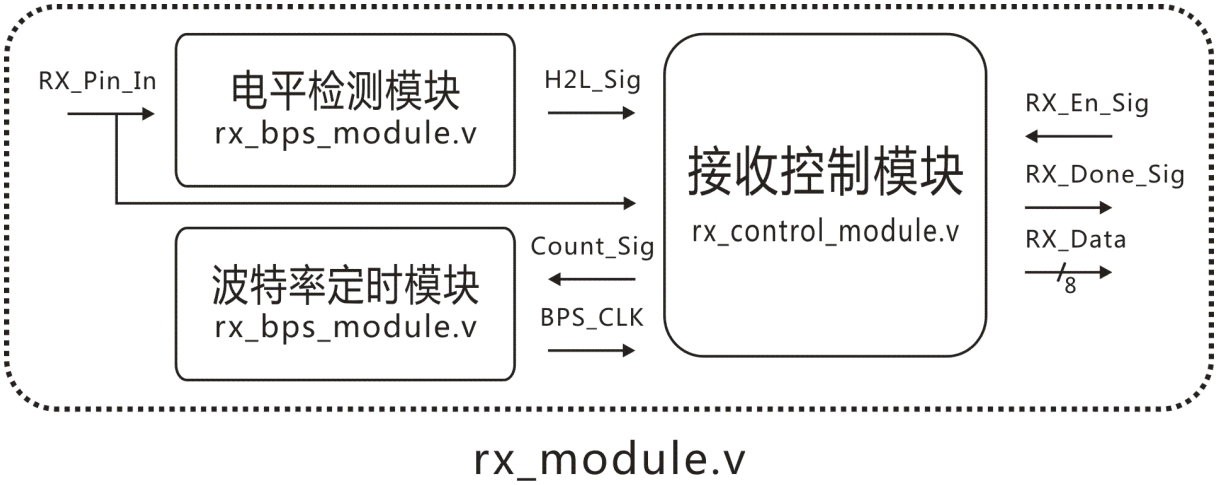


图 2.6a 早期的串口接收模块。

图 2.6a 读者是不是觉得很眼熟，而且也觉得很怀念 ... 笔者也是，这是早期的串口接收模块的“图形”。那时候的笔者不谈时序，更不谈整合（事实上早期的笔者根本就没有这些概念），咋看下还真是庞然大物，而且有点不规整 .... 说实话笔者也觉得有点不好意思，呵呵。不过话也不能这样说，如果不是“建模篇”的出现，基本上就不会衍生出那么多本优秀的笔记（老王卖瓜）。

图 2.6a 所示的 rx_module.v 是由 3 个模块组成的，接收控制模块，电平检测模块，波特率定时模块，模块的功能如各自的命名一样。

图 2.6b 串口接收模块的图形。

图 2.6b 所示的串口接收模块，基本上是把 接收控制模块，电平检测模块和波特率定时模块整合进来。在这里，可能有些同学觉得“这根本就是单文件主义”，只不过是把所有七零八落的模块拼凑在一个名为“rx_module.v”的模块而已。非也，这个模块包含的玄机，笔者会慢慢为读者们解答的。

在还没有浏览代码之前，笔者先讲解在实验中使用到的概念：

图 2.6c 串口发送模块向串口发送模块发送一帧数据

实验三和实验四的目标就是串口发送模块，以 20Mhz 的时钟频率，115k2 的波特率，一帧 9 位数据为例。那么一帧数据就需要 1914 个时钟，其中 174 时钟为一位数据的长度。当我们用“一个时钟，一块数据”的概念去理解的时候，串口发送模块只要确保完整无损的发送“一帧 1914 个时钟的数据”。从而在串口接收模块的方面，那么串口接收模块只要确保如何去“接收一帧 1914 个时钟的数据，并且处理它”。

串口发送模块的对象是以实验三为例。实验九中的串口接收模块，基本上是经过整合了，具体的内容，我们还是直接看代码吧：

rx_module.v

```
module rx_module
(
        input CLK,
        input RSTn,
        input RX_Pin_In,
        input RX_En_Sig,
        output RX_Done_Sig,
        output [7:0]RX_Data,

        /********************/

        output [7:0]SQ_C1,
        output SQ_F1,
        output SQ_F2

        /********************/
);

      /******************************/
```

```
20.
21.         reg F1,F2;    // F2 Previous State, F1 Current State
22.
23.           always @ ( posedge CLK or negedge RSTn )
24.                 if( !RSTn )
25.                         begin
26.                                     F1 <= 1'b1;
27.                                     F2 <= 1'b1;
28.                             end
29.                   else
30.                         begin
31.                                     F1 <= RX_Pin_In;
32.                                     F2 <= F1;
33.                         end
34.
35.         /**********************************/
36.
37.           parameter BPS115K2 = 8'd174, SAMPLE = 8'd87;
38.
39.           /**********************************/
40.
41.           reg [3:0]i;
42.           reg [7:0]C1;
43.           reg [7:0]rData;
44.           reg isDone;
45.
46.           always @ ( posedge CLK or negedge RSTn )
47.                 if( !RSTn )
48.                         begin
49.                                     i <= 4'd0;
50.                                     C1 <= 8'd0;
51.                                     rData <= 8'd0;
52.                                     isDone <= 1'b0;
53.                             end
54.                   else if( RX_En_Sig )
55.                         case( i )
56.
57.                                     0: //1 clock delay from F2 to i-0, other 1 clock delay from i-0 to i-1, well C1 add 2 clock
58.                                     if( F2 == 1'b0 ) begin i <= i + 1'b1; C1 <= C1 + 4'd2; end
59.
60.                                     1: // stalk and count 0's bit-start bits
61.                                     if( C1 == BPS115K2 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
62.                                     else C1 <= C1 + 1'b1;
63.
```

```
64.                             2,3,4,5,6,7,8,9: //stalk and count 1~8 data's bit , sample data at 1/2 for bps
65.                             begin
66.
67.                                 if( C1 == SAMPLE ) rData[i-2] <= F2;
68.                                 if( C1 == BPS115K2 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
69.                                 else C1 <= C1 + 1'b1;
70.
71.                             end
72.
73.                             10,11: // stalk and count last 2 bit
74.                             if( C1 == BPS115K2 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
75.                             else C1 <= C1 + 1'b1;
76.
77.                             12:
78.                             begin isDone <= 1'b1; i <= i + 1'b1; end
79.
80.                             13:
81.                             begin isDone <= 1'b0; i <= 4'd0; end
82.
83.                         endcase
84.
85.         /*******************************/
86.
87.         assign RX_Done_Sig = isDone;
88.         assign RX_Data = rData;
89.
90.         /*******************************/
91.
92.         assign SQ_C1 = C1;
93.         assign SQ_F1 = F1;
94.         assign SQ_F2 = F2;
95.
96.         /*******************************/
97.
98.   endmodule
```

第 3~8 行是该模块的输入输出口声明，第 21~33 行是电平检测模块 .... 但是，这里的电平检测模块经过整合之后，笔者不再使用 F1 和 F2 来取得电平的变化。目前 21~33 行只是一个保险而已，将异步信号同步化，为了就是避免亚稳态（不把一万只怕万一 ）。第 41~83 行是 rx_module.v 的核心功能了，是不是觉得和实验三的 tx_module.v 很相似呢？呵呵，它们都是使用同样的概念去写的（一个时钟，一块数据）。

第 41~44 行是寄存器声明，寄存器 i 用于步骤，寄存器 C1 用于计数和跟踪一帧 1914 个时钟的数据，寄存器 rData 是用于寄存从一帧 1914 个时钟的数据中所采集到的数据，最

后的 isDone 是用于产生完成信号的。在一开始的时候，所有数据复位并且清零（第 49~52 行）。

当 RX_En_Sig 拉高的时候，串口接收模块就进入就绪状态（第 54 行）。我们知道串口的一帧数据有 11 位，第零位也是起始位，状态永远都是逻辑 1。当进入步骤 0 的时候（57~58 行），if 条件检测到 F2 寄存的结果为 0，然后 i 递增，随后 C1 也递增“两个时钟”，一个时钟是从 F2 到步骤 0 的延迟，另一个时钟是从步骤 0 到步骤 1 的延迟（如果读者目前不明白这“两个时钟”的意思，请放松一点笔者会慢慢解释的）。

步骤 1（60~62 行）是用于继续“跟踪起始位的余下 n 个时钟”。在 if 条件中的 C1 == BPS115K2 -1 , BPS115k2 是第 37 行中定义的常量，亦即 115200 波特率的周期用于 20Mhz 的时钟量化数，求得公式如下：

```
串口一帧一位数据的周期 = 1 / 115200
                       = 8.6805555 ... us

20Mhz 时钟频率量化 = 8.6805555 ... us / ( 1 / 20Mhz )
                   = 173.6111 ....
                   = 174
```

然而，其中的 -1 表示 C1 清零和 i 更新 174 个时钟在里面。

步骤 2~9（64~71 行）分别是用来跟踪 一帧[1..8] 的数据。第 67 行的 if 条件是当计数器 C1 计数某位数据位的“一半”（37 行的 SAMPLE 常量声明）的时候，就采集（读取结果）与 rData。第 68~69 行的 if 条件是计数 174 个时钟以外，其中 if 条件里的 BPS115k - 1，和 61~62 行有同样的意思，亦即 C1 清零和 i 的更新发生在 174 个时钟里。

步骤 12~13（77~81 行）是产生完成信号。第 87~88 行是输出设置，第 92~93 行是仿真用得输出设置。基本上 rx_module.v 就是这么简单而已，但是唯一的问题点就是步骤 0 中的“if 判断条件”和“C1 <= C1 + 4'd2”。具体的内容，我们需要在仿真中才能知道。

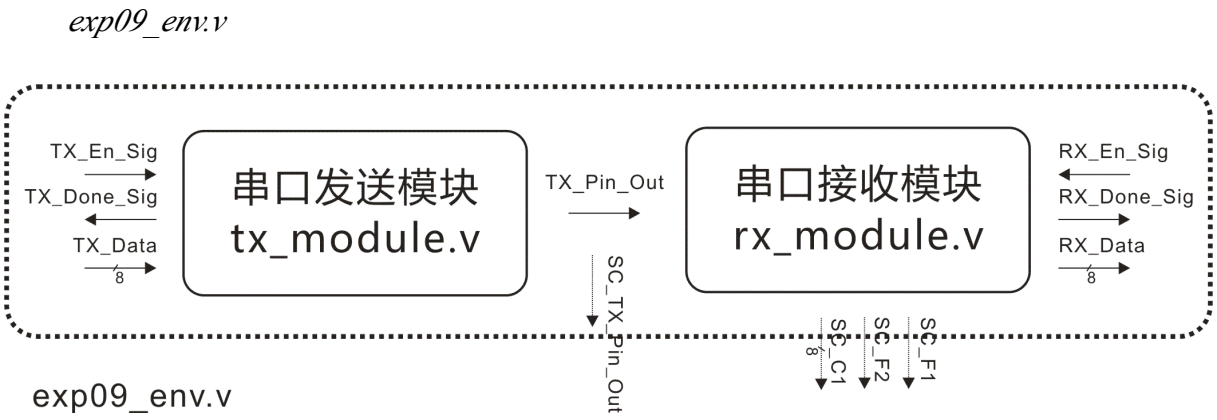


图 2.6d 实验九的仿真环境。

图 2.6d 是实验九——串口接收模块的仿真环境，关于仿真环境（虚拟环境）的概念笔者就不详谈了，自己复习时序篇的后半部。图 2.5h 中的串口接收模块是由实验三中的串口发送模块来驱动（刺激）。其中 SC_F1 , SC_F2 和 SC_C1 是从 rx_module.v 引出的，至于 SC_TX_Pin_Out 是从这个仿真环境中引出的，这些被引出的信号主要是在仿真中，用于观察。

```
module exp09_env
(
        input CLK,
        input RSTn,

        input TX_En_Sig,
        output TX_Done_Sig,
        input [7:0]TX_Data,

        output [7:0]RX_Data,
        input RX_En_Sig,
        output RX_Done_Sig,

        /****************************/

        output SQ_TX_Pin_Out,
        output [7:0]SQ_C1,
        output SQ_F1,
        output SQ_F2
);
        /****************************/

        wire TX_Pin_Out;

        tx_module U1
          (
                .CLK( CLK ),
                .RSTn( RSTn ),
                .TX_En_Sig( TX_En_Sig ),
                .TX_Done_Sig( TX_Done_Sig ),
                .TX_Data( TX_Data ),
                .TX_Pin_Out( TX_Pin_Out )
          );

          /****************************/

          rx_module U2
```

```
38.         (
39.                 .CLK( CLK ),
40.                 .RSTn( RSTn ),
41.                 .RX_Pin_In( TX_Pin_Out ),
42.                 .RX_En_Sig( RX_En_Sig ),
43.                 .RX_Done_Sig( RX_Done_Sig ),
44.                 .RX_Data( RX_Data ),
45.                 /**********************/
46.                 .SQ_C1( SQ_C1 ),
47.                 .SQ_F1( SQ_F1 ),
48.                 .SQ_F2( SQ_F2 )
49.         );
50.
51.         /****************************/
52.
53.         assign SQ_TX_Pin_Out = TX_Pin_Out;
54.
55.         /****************************/
56.
57.  endmodule
```

第 3~12 行是输入输出口的声明，第 16~19 行是仿真用输出口的声明。第 25~33 行是实验三的串口发送模块的实例化，反之第 37~49 行是实验九的串口接收模块的实例化。第 53 行是 SQ_TX_Pin_Out 的链接设置。

exp09_env.vt

```
1.   `timescale 1 ns/ 1 ps
2.   module exp09_env_simulation();
3.
4.       reg CLK;
5.       reg RSTn;
6.       reg [7:0]TX_Data;
7.       reg TX_En_Sig;
8.       reg RX_En_Sig;
9.
10.      wire [7:0]RX_Data;
11.      wire RX_Done_Sig;
12.      wire TX_Done_Sig;
13.
14.      /**********************/
15.
16.      wire SQ_TX_Pin_Out;
```

```
17.         wire [7:0]SQ_C1;
18.         wire SQ_F1;
19.         wire SQ_F2;
20.
21.       /****************************************/
22.
23.         exp09_env U1
24.         (
25.                 .CLK(CLK),
26.                 .RSTn(RSTn),
27.                 .TX_En_Sig(TX_En_Sig),
28.                 .TX_Done_Sig(TX_Done_Sig),
29.                 .TX_Data(TX_Data),
30.                 .RX_Data(RX_Data),
31.                 .RX_En_Sig(RX_En_Sig),
32.                 .RX_Done_Sig(RX_Done_Sig),
33.                 /******************************/
34.                 .SQ_TX_Pin_Out(SQ_TX_Pin_Out),
35.                 .SQ_C1( SQ_C1 ),
36.               .SQ_F1( SQ_F1 ),
37.                 .SQ_F2( SQ_F2 )
38.         );
39.
40.       /****************************************/
41.
42.         initial
43.         begin
44.                 RSTn = 0; #10 RSTn = 1;
45.                 CLK = 1; forever #25 CLK = ~CLK; // 50ns period or 20Mhz Clock Frequency
46.         end
47.
48.         /****************************************/
49.
50.           reg [3:0]i;
51.
52.           always @ ( posedge CLK or negedge RSTn )
53.                 if( !RSTn )
54.                         begin
55.                                 i <= 4'd0;
56.                                 TX_Data <= 8'd0;
57.                                 TX_En_Sig <= 1'b0;
58.                                 RX_En_Sig <= 1'b0;
59.                             end
60.                     else
```

```
61.                     case( i )
62.
63.                         0:
64.                         begin RX_En_Sig <= 1'b1; i <= i + 1'b1; end
65.
66.                         1:
67.                         if( TX_Done_Sig ) begin TX_En_Sig <= 1'b0; i <= i + 1'b1; end
68.                         else begin TX_Data <= 8'b0101_0101; TX_En_Sig <= 1'b1; end
69.
70.                         2:
71.                         if( RX_Done_Sig ) begin RX_En_Sig <= 1'b0; i <= i + 1'b1; end
72.
73.                         3:
74.                         i <= i;
75.
76.                     endcase
77.
78.         /****************************************/
79.
80.   endmodule
```

这个激励文件的内容很简单，笔者就大概的介绍一番。第 23~38 行是实验九的仿真环境（虚拟环境）的实例化，第 42~46 行是复位信号和时钟信号的刺激。第 50~76 行是激励过程，步骤 0（63~64 行）是使能串口接收模块，并且要它进入就绪状态。步骤 1（66~68 行）是使能串口发送模块，并且开始发送数据 8'b01010101。直到一帧数据发送完毕，串口发送模块会反馈完成信号，然后 67 行的 if 条件会成立，并且不使能它和更新 i。

步骤 2（70~71 行）是：当串口接收模块完成接收一帧数据的时候，它会反馈完成信号，然后 71 行的 if 条件会成立，并且不使能串口接收模块，然后更新 i。步骤 3（73~74 行）是停留而已，没有什么意义。

仿真结果：

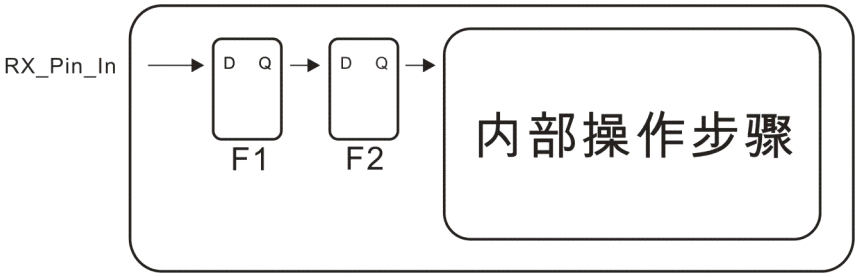


图 2.6e 串口接收模块的内部概念图。

在还没有秀出仿真图之前，读者需要知道串口接收模块基本上，再没有“电平检测模块”，

“波特率定时模块”和“接收控制模块”之分，取而代之是经过整合的电平检测模块，它变成单纯的一对移位寄存器 F1 和 F2，目的是防止亚稳态发生。那么一帧数据会从 F1 读入，然后过渡 F2，最后会进入内部的操作步骤。

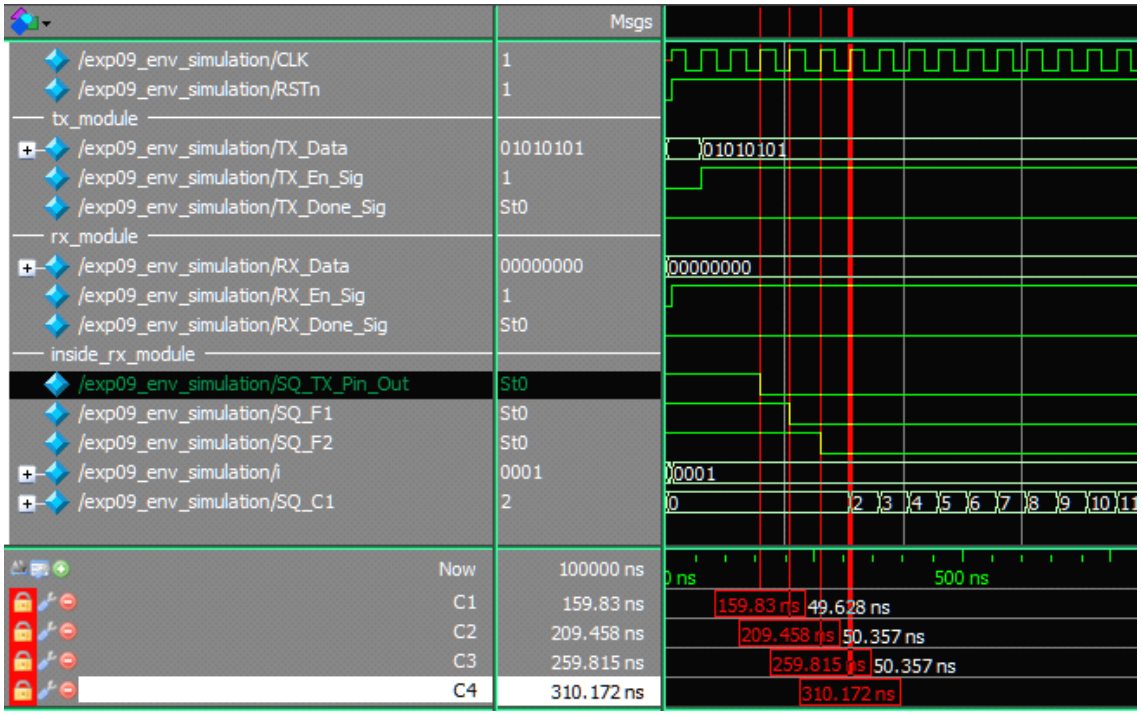


图 2.6f 仿真图・SQ_TX_Pin_Out 对 F1 延迟一个时钟，F2 对 F1 延迟一个时钟。

图 2.6f 是仿真图，注意其中的 SQ_TX_Pin_Out ，SQ_F1 和 SQ_F2 这三个信号。根据“模块沟通的硬道理”——模块之间的沟通需要至少一个时钟，所以 SQ_F1 相比 SQ_TX_Pin_Out 延迟了一个时钟（C1~C2），SQ_F2 相比 SQ_F1 延迟了一个时钟（C2~C3），串口接收模块的内部操作相比 SQ_F2 延迟了一个时钟（C3~C4）。

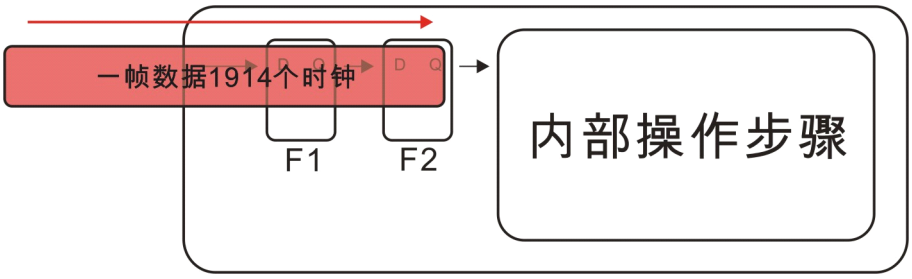


图 2.6g 一帧数据进入串口接收模块的过程。

在这里内部操作步骤“只要从 F2 读入数据”就可以了，如图 2.6g 所示。

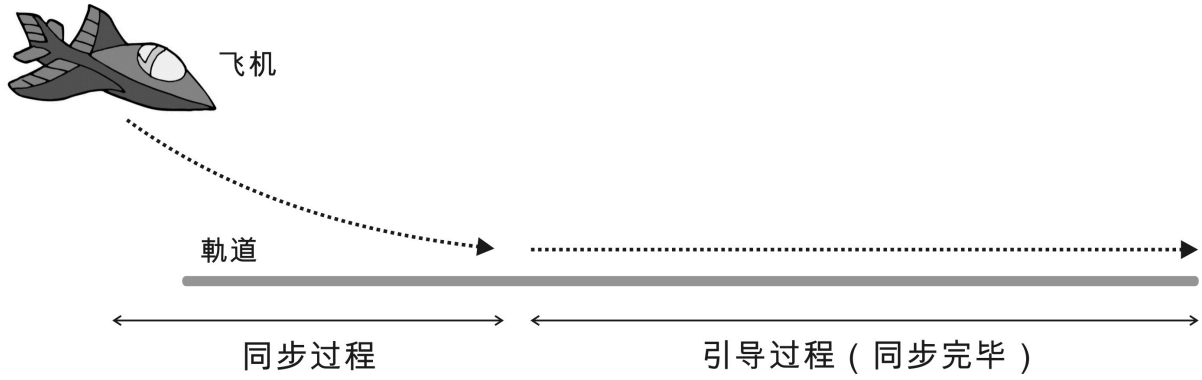


图 2.6h 同步过程的概念图。

图 2.6h 是一架飞机降落的经过。不知道读者有没有看过科幻片，或者科幻动漫之类的如：星级大战系列的电影或者高达系列的卡通。假如某个飞机要进入某个基地，飞机就需要沿着跑道降落。首先飞机会将速度调整到和跑道引导程序一样的频率，这个过程成为同步过程。当飞机的速度和引导程序的频率同步完毕，那么引导程序会自动引导飞机进入基地。

同样的道理，当一帧长长的数据从一方进入另一方的时候，如果“没有时钟信号的引导”（串口的传输协议是没有时钟信号的，反之 SPI 的传输协议却有时钟信号来引导），很多时候一帧数据的第[0]位除了有“起始”的定义以外，接收方有责任把第[0]位数据作为同步的对象。

那么，上述的内容又和实验九有什么关系呢？

```
32                  F2 <= F1;
33              end
34
35      /*********************************/
36
37      parameter BPS115K2 = 8'd174, SAMPLE = 8'd87;
38
39      /*********************************/
40
41      reg [3:0]i;
42      reg [7:0]C1;
43      reg [7:0]rData;
44      reg isDone;
45
46      always @ ( posedge CLK or negedge RSTn )
47          if( !RSTn )
48              begin
49                  i <= 4'd0;
50                  C1 <= 8'd0;
51                  rData <= 8'd0;
52                  isDone <= 1'b0;
53              end
54          else if( RX_En_Sig )
55              case( i )
56
57                  0: //1 clock delay from F2 to i-0, other 1 clock delay from i-0 to i-1, well C1 add 2
58                  if( F2 == 1'b0 ) begin i <= i + 1'b1; C1 <= C1 + 4'd2; end
59
60                  1: // stalk and count 0's bit-start bits
61                  if( C1 == BPS115K2 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
62                  else C1 <= C1 + 1'b1;
```

步骤0判断F2用了1个时钟

步骤0进入步骤1又用了1个时钟

图 2.6i　rx_module 其中一段代码。

图 2.6i 是 rx_module.v 的一段代码，这段代码有如“飞机开始进入跑道”的时候。当步骤 0 的 if 条件判断 F2 的过去值的时候，基本上“一帧 1914 个时钟的数据，已经进入跑道 1 个时钟了”。步骤 0 进入步骤 1 的时候，基本上“一帧 1914 个时钟的数据，已经进入跑道 2 个时钟了”。换句话说，当 rx_module.v 进入步骤 1 的时候“174 个时钟的起始位”的“前 2 个时钟的数据”已经流过了 .... 这也是代码中的地 58 行，C1 要预先计算 2 个时钟。

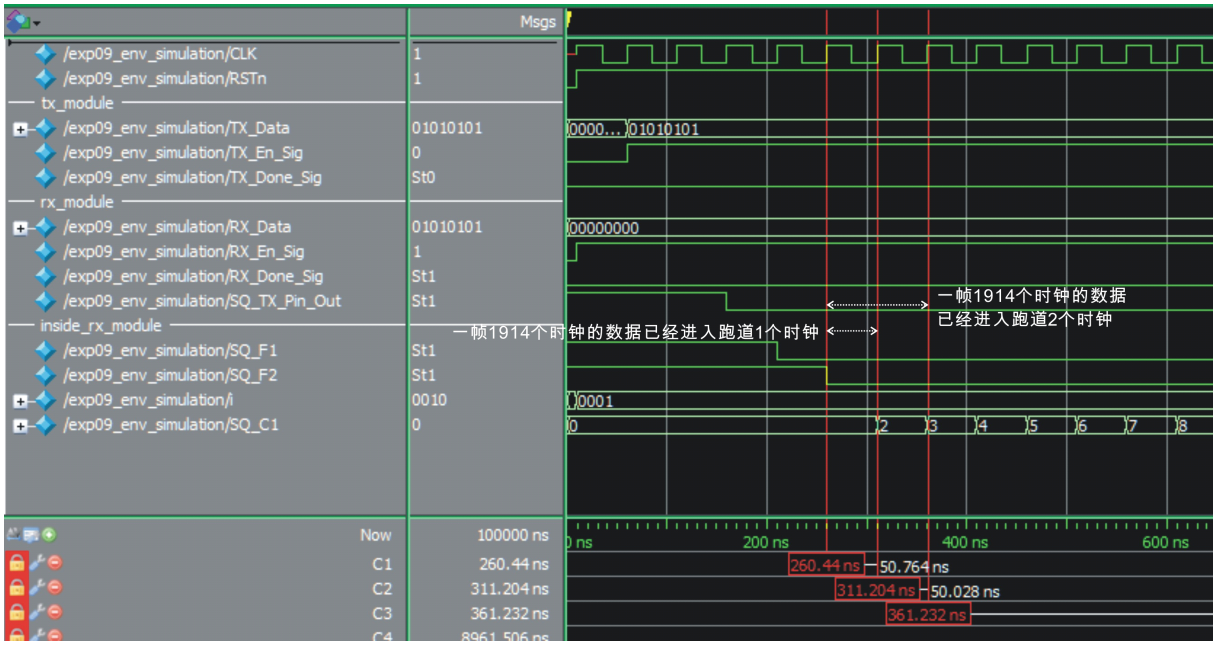


图 2.6j 对第[0]位数据的“同步计数”。

图 2.6j 就是“飞机开始进入跑道”。C1 指向是 F2 读取 F1 的过去值的时候，在 C1 的未来，F2“决定”输出数据逻辑 0。C2 是指向步骤 0 判断 F2 的过去值的时候，目前一帧 1914 个时钟的数据已经进入跑道 1 个时钟了（174 个时钟的起始位已经进入跑道一个时钟了）。步骤 0 的 if 条件判断 F2 的过去值是逻辑 0，if 条件成立，步骤 0“决定”递增 i 和为 SQ_C1 加入 2 个预先时钟。所以在 C2 的未来，C1 的值为 2，i 的值为 1。

C3 是指向 rx_module.v 已经进入步骤 1，目前一帧 1914 个时钟的数据已经进入跑道 2 个时钟了（174 个时钟的其实位已经进入跑道 2 个时钟），这时候 SQ_C1 的过去值是 2，基本上同步已经完成了，然后步骤 1 计算起始位“余下”的时钟数。

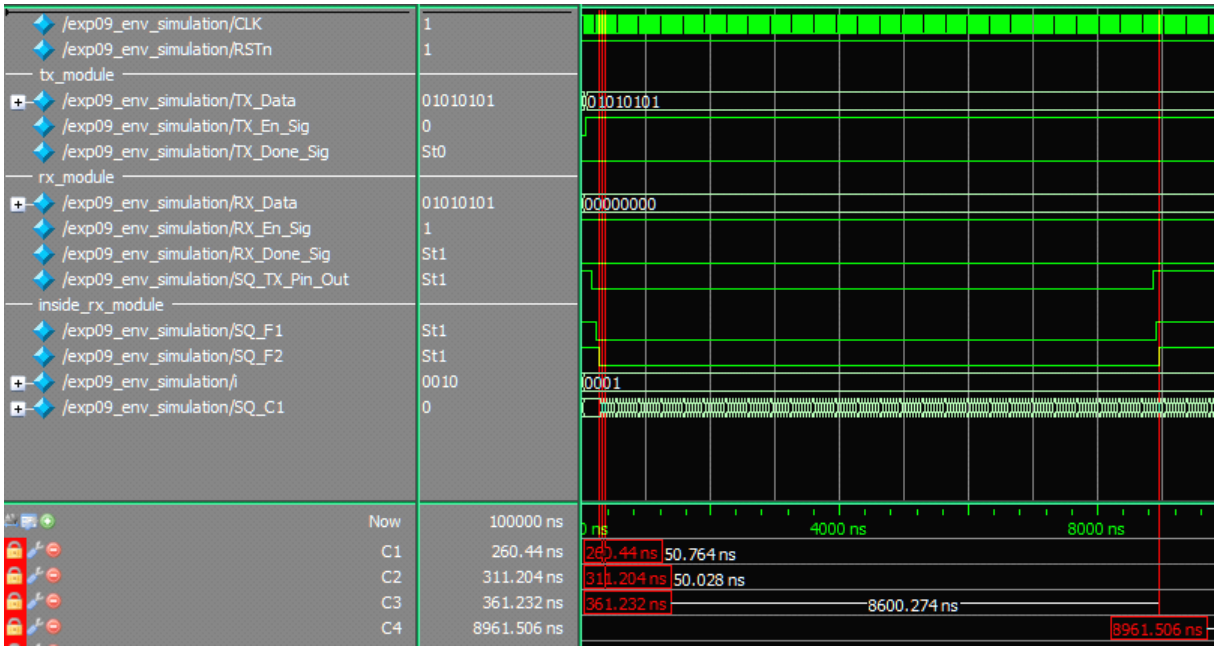


图 2.6k 一帧数据第 0 位——起始位

图 2.6k 中 C1~C4 是指向一帧数据 174 个时钟长度的第 0 位，读者只要把 C1 的时间 260.44ns 减掉 C4 的时间 8961.506ns ，得到的结果大约是 8700ns。仿真所使用的频率是 20Mhz，亦即 50ns 的时钟周期，8700ns 除与 50ns 就是 174 了。

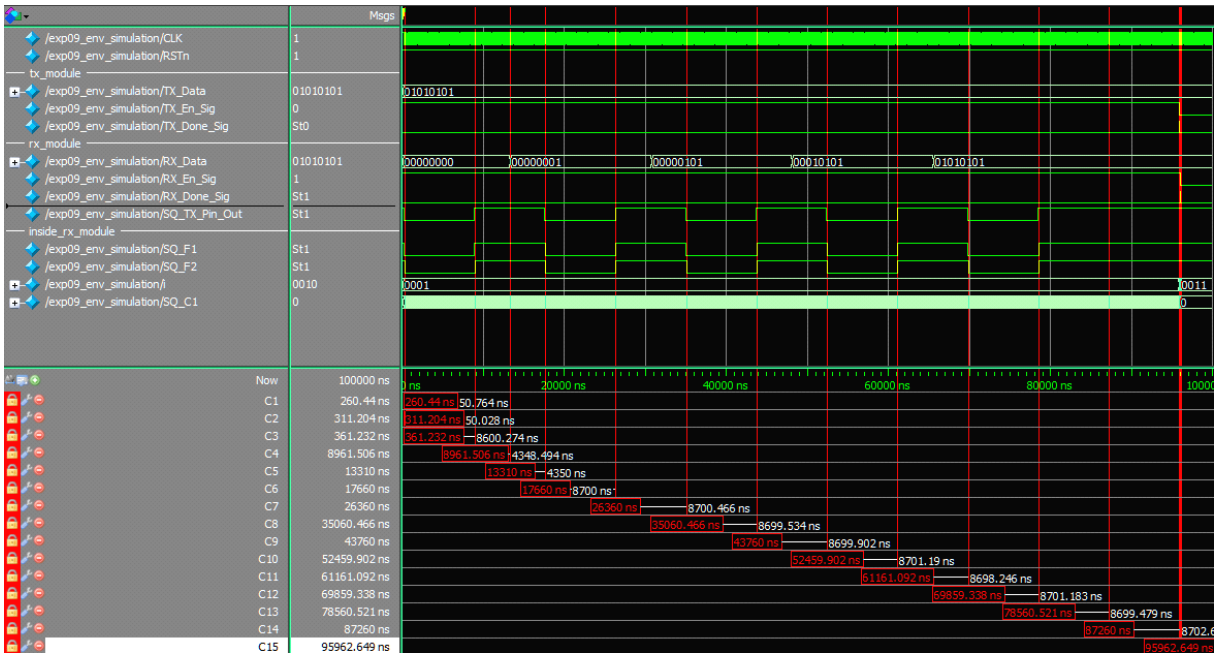


图 2.6l 一帧数据 11 位，跟踪，计数和采集的过程。

图 2.6l 是一帧数据的，跟踪，计数和采集的过程。C1~C4 是起始位，C4~C5 是[0]数据位，C5~C6 是[1]数据位，C6~C7 是[2]数据位，C7~C8 是[3]数据位，C8~C9 是[4]数据位，C9~C10 是[4]数据位，C10~C11 是[5]数据位，C11~C12 是[6]数据位，C12~C13 是[7]数据位，C13~C14 是校验位，C14~C15 是结束（停留）位。除了起始位的跟踪比较特别以

外，其余的“位”SQ_C1 都是从 1 开始计数，直到 173~0 结束。

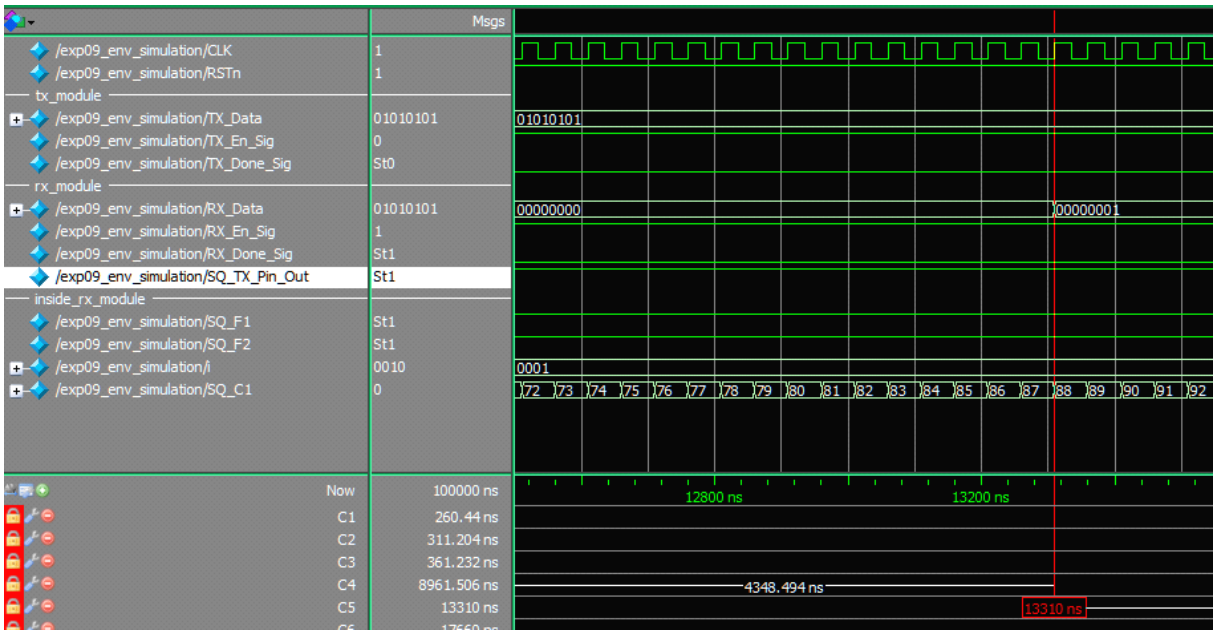


图 2.6m 每当 SQ_C1 计数到 87 个时钟数的时候就采集。

图 2.6m 是第[0]数据位的采集过程，还记得 rx_module.v 中的步骤 2~9——67 行吗？其中的代码是这样写的：

```
if( C1 == SAMPLE ) rData[ i-2 ] <= F2;
```

然而 SAMPLE 的常量是 BPS115K2 的一半，亦即 87。换句话说，每当计数器 SQ_C1 (C1) 跟踪和计数某个“位”到“第 87 个时钟”的时候，就会采集 SQ_F2 的数据。如图 2.6m 中所示，C5 指向某“位”第 87 个时钟数的时候，此时会“决定”采集 SQ_F2 的过去值，亦即逻辑 1。在 C5 的未来，RX_Data 的结果变成 8'b0000_0001。同样的事情也发生在余下的[1..7]数据位。

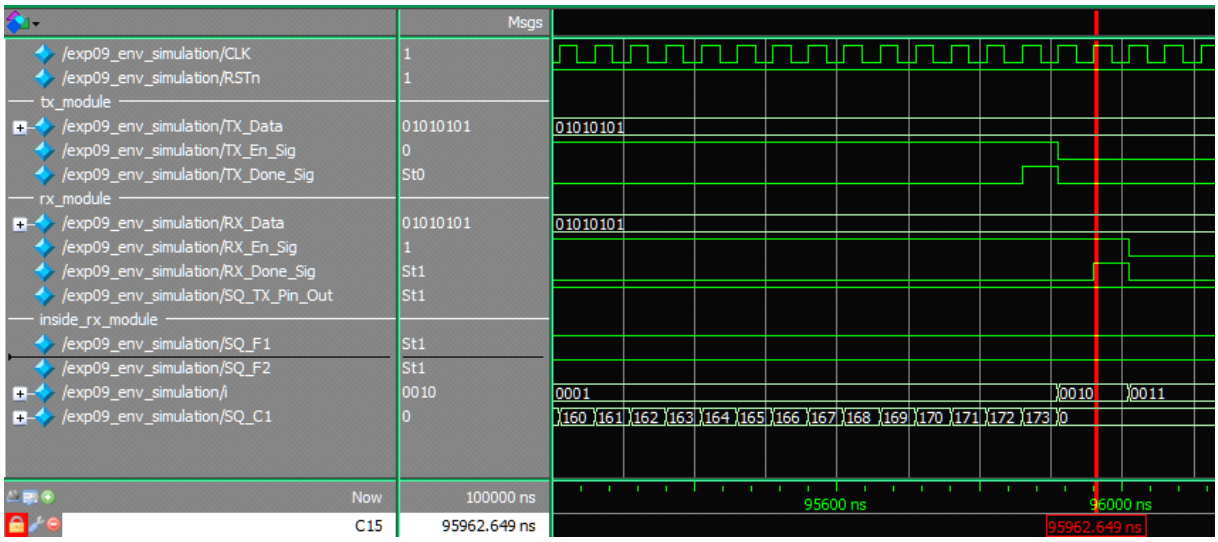


图 2.6n 串口接收模块产生完成信号。

图 2.6n 是 rx_module.v 进入步骤 12 的时候，光标 C15 之前是结束位的跟踪和计数，光标 C15 之中是步骤 12“决定”拉高 RX_Done_Sig，光标 C15 之后是 RX_Done_Sig 的产生。此时的 RX_Data 已经成功接收到由串口发送模块发送来的数据 8'b0101_0101。

到这里，笔者已经结束分析整个仿真过程了。实验九的重点就是：

用“一个时钟，一块数据”的概念去理解实验九的话，我们把“串口发送模块发送一帧 1914 个时钟的数据”看成是某一方的操作，只要我们“确保串口发送模块可以正常发送 1914 个时钟的数据”，余下就不关串口发送模块的事了。换之，串口接收模块这一方如何“正确”的去“跟踪和计数”这一帧 1914 个时钟的数据，并且采集其中数据内容就行了。

此外，实验九还有一个重点就是：

在建模篇里的串口接收模块是由 3 个模块组合而成——波特率定时模块，电平检测模块和串口接收控制模块。但是经“时钟化”和用“一个时钟，一块数据”的概念整合以后，电平检测模块变成简单的移位寄存器 F1 和 F2，目的是为了确保预防亚稳态。在这里，再没有什么“电平检测模块”，“波特率”类似的功能了，取而代之的就是“一切内容都以时钟作为模块的设计基础”。

2.7 不同执行模式的“定时器和计数器”

“定时器”这一个概念在单片机中就是“定时中断”的功能，然而“计数器”这个概念又在单片机中是“延迟，记录执行次数”等。还记得笔者在学习单片机的时候，“定时器”的“中断间隔”要越长越好，因为过于频繁的“定时中断”会消耗单片机宝贵的资源。此外假设某个定时器是 1us 的中断间隔（实际上 1us 的中断间隔是非常不好的习惯），然后可以基于这个 1us 的定时器执行计数，100 次的计数就是 100us，1000 次的计数就是 1ms 等 .... 这就是在单片机中“定时器”和“计数器”的关系。

反之，在 FPGA 的世界中“定时器”的间隔越短越好，如果间隔越长，那么消耗的逻辑资源就越多，但是这也是“量”的问题而已，目前的 FPGA 动不动就几千个逻辑资源，所以说这个问题，根本上就不是问题。定时器在 Verilog HDL 语言的表现如下：

```
always @ ( posedge CLK )
    if( C1 == T1US )
        C1 <= 5'd0;
    else
        C1 <= C1 + 1'b1;
```

同样在早期的时候，笔者为了实现更长时间的“定时器”，常常把小时间量的“定时器”，再配上“计数器”从而实现大时间量的“计数器”。Verilog HDL 语言的实现如下：

```
always @ ( posedge CLK )
    if( C1 == T1US )
        C1 <= 5'd0;
    else
        C1 <= C1 + 1'b1;

always @ ( posedge CLK )
    if( C2 == 1000 )
        C2 <= 10'd0;
    else if( C1 == T1US )
        C2 <= C2 + 1'b1;
```

定时器或者计数器可以说在 FPGA 里边使用最频密的一个功能，它的作用多到用两只手也数不完。在这里，读者可能会产生疑问：“定时器和计数器，我当然知道啦，这是基础再基础不过的东西，笔者你有没有病？为什么忽然讨论这么基础的东西？”笔者没有生病，就是它们（定时器和计数器）过于基础，从而使得很多学习者忽略一些非常重要的细节，在此笔者一个一个慢慢来解释。

读者先在心里尝试回答这个“再基础不过的问题”先 — “在 Verilog 的世界里定时器和计数器有什么不同？”嗯 .... 读者是不是觉得相似回答的出，又相似回答不出呢？或者更干脆点，就如笔者在前面说所的那样：定时器产生某个时间间隔，然后计数器可以

基于这个时间间隔，计数更大的间隔时间量 ....

不不不 .... 早期的笔者也好，还是这样回答的读者也好，事实上我们已经被“单片机的思路”主宰了，因为类似的回答是站在“单片机的世界”而不是站在“Verilog 的世界"去解释定时器和计数器的不同。好了，笔者不再捉弄读者了，笔者还是切入正题。

“定时器”这一词实际上是从单片机的世界接过来用的，因为在单片机的世界中，“时钟的概念”是透明而且是模糊的，亦即单片机的使用者对于“时钟”来说是“很难控制"（笔者没有说不能控制）。因此单片机才要借用而外的器件去产生某个特准时间的间隔，事实上无论这个器件的质量再怎么好，它说产生的定时是近似或者接近的定时结果，因为单片机中有“上一步步骤”的延迟。简单说，就是当某个定时器产生的时候，单片机的处理器要保存这个数据，查找那个中断向量，又调用哪个服务函数 ... （汗！实际上更复杂）当中断服务函数被启动以后，它已经延迟了诺干的步骤所消耗的时钟数。

笔者冒着生命危险的说道“在 Verilog 的世界里根本就没有“定时器”这东西”有的只是可以“计数时钟”的计数器而已。如果我们硬是要用 Verilog 去实现定时器的话，我们来看看会出现什么小 Bug ：

exp2_7a.vt

```
always @ ( posedge CLK or negedge RSTn )
    if( !RSTn )
        C1 <= 4'd0;
    else if( C1 == 4 )
        C1 <= 4'd0;
    else
        C1 <= C1 + 1'b1;
```

上面一段代码是再正常不过的定时器实例，该定时器是产生 4 个时钟的间隔。

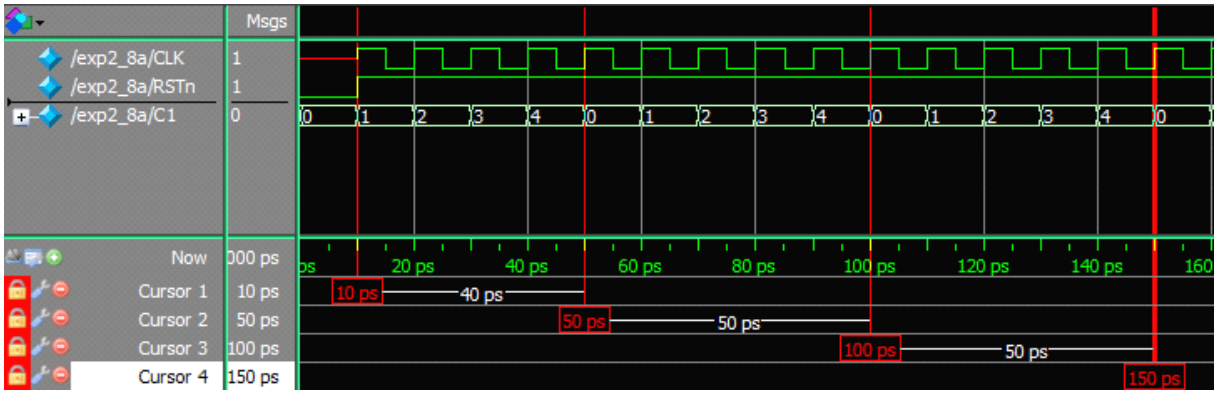


图 2.7a 实例 exp2_7a.vt 的仿真结果。

图 2.7a 是仿真结果，从中我们可看到一些小 bug 了。（以下 Cursor 简称为 C）在 C1~C2

之间是初头 4 个时钟长度的产生，随后的 C2~C3 和 C3~C4 是 5 个时钟长度的产生。但是更多的使用者面对这个小 bug，他们会这样修改 exp2_8a.vt 其中一段代码。

```
always @ ( posedge CLK or negedge RSTn )
    if( !RSTn )
        C1 <= 4'd0;
    else if( C1 == 3 )
        C1 <= 4'd0;
    else
        C1 <= C1 + 1'b1;
```

把其中 else if (C1 == 4) 改成 (C1 == 3) ，那么我们再仿真看看。

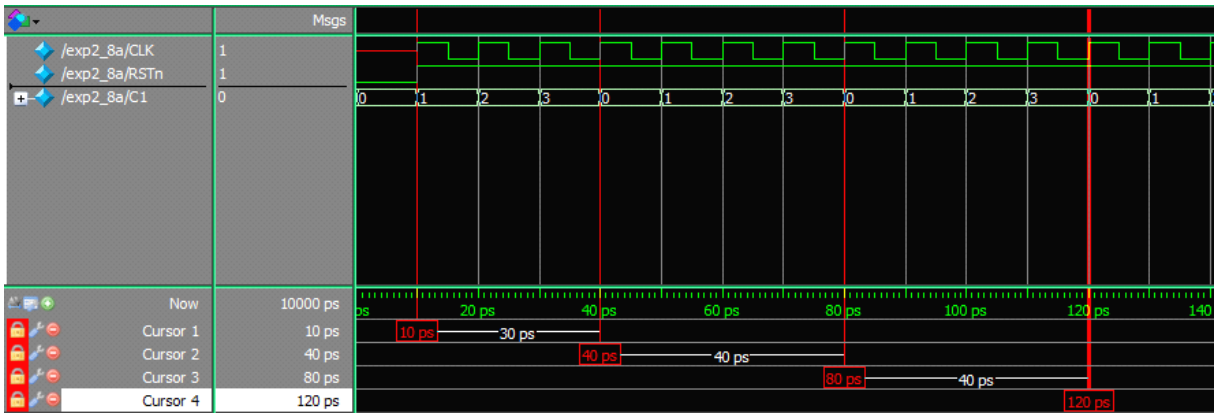


图 2.7b 经过修改 exp2_7a.vt 的仿真结果。

图 2.7b 是经过修改之后的方针结果，C1~C2 是指向初头的定时结果，亦即产生 3 个时钟长度的定时，然而 C2~C3 和 C3~C4 则是产生 4 个时钟长度的定时。C1 == 4 还是 C1 == 3 ，结果都是第一次一定和接下来的不违和。这也是许多初学者（即时是有经验的家伙）累犯的小 bug，如果实验的精度要求不高，那么则无伤大雅，反之如果是“完美者”还是“偏激狂”的实验要求，这样的答案当然不行。

读者们千万不要灰心，类似的问题之前的笔者也是常犯。我们只要换另一个思路，再次修改一下代码就行了。在某一天的半夜，笔者忽然间醒来，脑子里一直充斥这样一个问题：“为什么时钟的精度那么难控制，我到犯了什么？还是缺少什么？”，然后笔者坐在床上呆呆的盯着风扇在旋转 ...

笔者呐呐到：“事实上 FPGA 根本就没有定时器这个东西，用 Verilog 实现定时器这个想法，根本上就是对 Verilog 的世界很不公平！在 Verilog 的世界里，每一个时钟都有一个周期（时间），C1~C4 的时钟长度完全都是等价于 C2~C5 的时钟长度。时钟精度的控制，重点不是在数字上的控制，而是有效的估算时钟的消耗，和跟踪时钟它们。”

然后笔者躺在床上，进入静思的状态思考到：“如果 Verilog 的世界没有定时器这东西，那么 Verilog 需要用实现某个东西，可以有效的产生 正确的 N 个时钟，如果我可以重复的并且正确的产生 固定的 N 个时钟，那么这就不是定时器吗？不不不，都说了 Verilog

的世界里没有定时器这东西，那么余下如何正确估算 N 个时钟，然后又如何估算在 N 个时钟里触发某个事件（时间点事件）这才是 Verilog 要干的工作，更多时候‘数字’起到跟踪时钟的作用，而‘数字’本身不是完全指向时钟。”

那一天，笔者完全失眠了，笔者也因此有了“一个时钟一块数据”的概念。好了，笔者简单的总结一下思路：定时器在 Verilog 的世界里只是从单片机的世界里借来的，实际上定时器就是等介于“固定产生 N 个时钟长度”“数字只是一个跟踪的标签而已”。然后我们可以这样修改 exp2_7a.vt 成为 exp2_7b.vt:

exp2_7b.vt

```
always @ ( posedge CLK or negedge RSTn )
    if( !RSTn )
      C1 <= 4'd0;
    else if( C1 == 4 )
      C1 <= 4'd1;
    else
      C1 <= C1 + 1'b1;
```

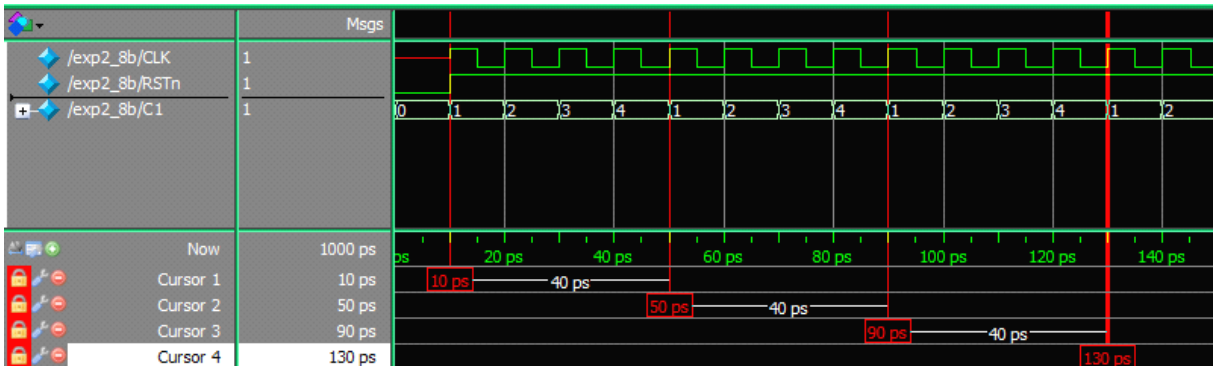


图 2.7c 实例 exp2_7b.vt 的仿真结果。

图 2.7c 是仿真结果，从中无论是 C1~C2，还是 C2~C3 或者 C3~C4, 时钟的长度都是固定的。最后笔者可以这样总结：

事实上 Verilog 的世界是没有定时器这东西，有的只有计数器，一个计数而且跟踪 N 个时钟长度的计数器。然而定时器不过是重复产生 N 个固定长度的计数器而已，换言之就是重复 某个数字范围（exp2_7b.vt 为例就是数字 1~4），然而数字只是跟踪和计数 N 个时钟长度的标签而已。继续以图 2.7c 为例，笔者可以说 C1~C2 的数字 1~4 加上 C2~C3 的 1~2 是 6 个时钟的长度，又或者说 C1~C4 的 3 次数字 1~4 总和一共是 12 个时钟的长度，但是重点就是如何去估算。

说以说 Verilog HDL 语言不是实现定时器，而是实现“类似”定时器的东西，但是本质上还是“计数器”。另外，每当计数器开始计数时，从会从 1 开始，然后致 0 结束。如

果读者窒息的观察 图 2.7c，读者就会发现图中的 C1 确实是从 1 开始计数，但是它是循环着计数，所以永远都是无法计数结束。

此外，还有一个重点就是“类似定时器”的东西在 Verilog 的世界里虽然好用，如果实验要求不挑剔为前提下。但是更多是“类似定时器”这样“一直重复计数固定 N 个时钟"的计数器不常使用，很多时候我们更需要可以使能和关闭的计数器：

```
always @ ( posedge CLK or negedge RSTn )
    if( !RSTn )
        C1 <= 4'd0;
    else if( C1 == rTimes )
        C1 <= 4'd0;
    else if( isCount )
        C1 <= C1 + 1'b1;
    else
        C1 <= 4'd0;

     ....
     case( i )

        0:
        if( C1 == rTimes ) begin isCount <= 1'b0; i <= i + 1'b1...
        else begin rTime <= 4; isCount <= 1'b1; end
```

在建模篇里，类似的写法常常出现，就是把定时器|计数器放置在核心操作的外边。实际上要实现紧密的计数，上面的写法会破绽百出（笔者又一次漂亮的抽自己一拳），因为每当启动定时器|计数器的时候，使用者很容易忽略计数定时器|计数器启动时所消耗掉的时钟。最后还是那么一句，如果实验的条件不挑剔的话，那么还可以接受。

读者尝试想象一下，读者有一个闹钟放在自己的房间外边，每当时钟响一次不耐烦的读者会离开狗窝去关闭闹钟，然后继续回狗窝睡觉。然后闹钟又响了起来，读者又一次不耐烦的离开狗窝去关闭闹钟 .... 然后闹钟又又响了 ... 正常的人早已经发飙毁掉闹钟了。上诉的事实如果反映到 Verilog 的世界也是如此，因此笔者才有把计数器“整合进来”的初衷，正如在第一章里所看到的那样。在这里，笔者可以这样说道：定时器也好，计数器也好，这些都只是概念上的标签而已，最重要的是如何精确的去跟踪和计数某个信号的长度。

2.8 计算 N 个长度的工具

在 Verilog HDL 的综合语言里充满着各种比较操作符 == != <= < > >= 。笔者依然还记得初学 Verilog HDL 的时候，笔者还天真的用 C 语言的角度去看 ... 结果真的是“最恶”（saiaku-日文是糟糕的意思）。好了，不多废话了，要如何有效的去用这些比较操作符，就得从“如何看波形图”讲起。

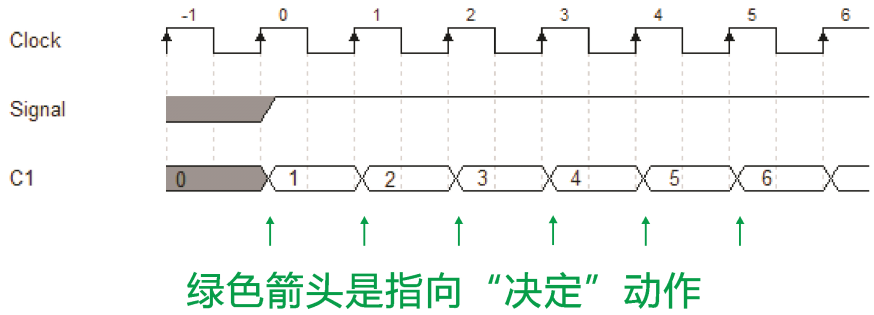


图 2.8a “决定”操作的波形图看法。

图 2.8a 的绿色箭头是指向模块的“决定”操作，这也是时序篇里的一大重点。因为任何模块的操作，我们都写在 always @ (posedge CLK) 的区域内，在 SystemVerilog 里基本上是和 always_ff（posedge CLK）等价的。如果从数字电子的基础说起，亦即就是 Flip Flop 的在时钟的上升沿触发“时间点事件”，具体的内容，读者就复习时序篇吧。

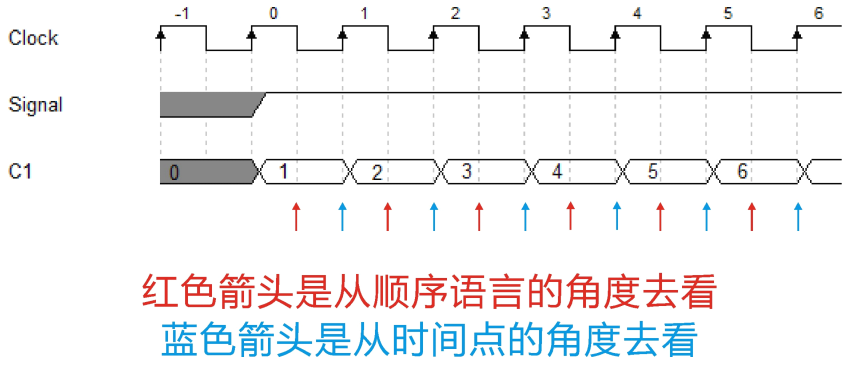


图 2.8b 各种不同角度去看计数和跟踪某个信号的长度。

当我们用某个计数器去跟踪和计数某个信号的 N 个时钟长度的时候，我们就要换另外一种看法了。如图 2.8b 所示，C1 用来跟踪和计数 Signal 信号，早期的读者就是受到单片机和顺序语言的思想主宰的关系，总是习惯用顺序语言的角度去看某个信号的长度计数。如图 2.8b 所示，红色箭头就是从顺序语言的角度去看某个信号的长度计数，事实上这是美丽而且非常严重的误会。

因为任何依赖时钟而驱动着的模块，一般上的“决定”和操作都是按照“时间点”的事件去执行。此外，在任何时候无论是现实还是波形图中，在某个时间点我们只能“参考”一些已经成为历史的过去值，而不能“预测和参考到未来值”。未来值是在某个时间的时候，根据“决定”而产生的结果。所以说，明白什么是“时间点”事件是多么重要的，

参考书才不会告诉您呢！

在这里，读者先大概回想第一章的内容，读者可能会发现，笔者很喜欢用 if(xx == xx) 去触发 if 条件的成立。继续以图 2.8b 为例，假设笔者要在 Signal 的长度为 2 的时候触发某个事件，那么笔者可以这样表示 if(C1 == 2) ... 就是这么简单。当我们要在某个信号在某个长度的时候触发某个事件是很简单的事情，但是当我们要“接收某个范围长度”的话我们就得下一点功夫了。

（还是以图 2.8b 为例）我们知道“时间点事件”只能参考过去值而已——这一个事实，如果我们要“接收某个范围的长度”我们一定基于这个道理。举个例子，假设笔者要接收 Signal 信号长度 2 到 4 的范围之内，那么笔者可以这样表示（如写法 1 所示）：

写法 1：

```
if( C1 >1 && C1 <= 4 )  触发某个事件
```

上述就是最典型的写法，但是对于许多新手来说尤其是不明白时序的家伙，这根本就是一团黑糟糟的东西。好了，读者先冷静下来想想看，我们是为了什么目的，则要用 if 条件来判断某个信号的范围长度呢？为了不就是在这个范围内除法某个事件吗？笔者这样说有点抽象了，笔者换一种写法会具体一点（如写法 2 所示）：

写法 2：

```
if( C1 == 2 || C1 == 3 || C1 == 4 )  触发某个事件
```

根本上，写法 1 和写法 2 是完全等价的，不过写法 1 和写法 2 比起来，显得有点“猥琐”难看懂（明白时序概念除外），虽然写法 2 比起写法 1 显得更人性化一点，但是写法 2 只支持小范围而已。读者可能要问“笔者有没有写法 3 呢?”有是有，但是感觉更糟糕（如写法 3 所示）：

写法 3：

```
case( i )
      0:
      begin x <= 2; i <= i + 1'b1; end

      1;
      if（C1 == x && C1 != 4 + 1）begin x <= x + 1'b1;  触发事件  ...
```

就是用第一章里循环操作的想法，用某个变量去控制某个操作，然后给予一个初值 x <= 2，然后又限制一个最高的范围 C1 != 4 + 1。读者是不是怕了吗？呵呵呵 ... 事实上无论是写法 1~2 还是写法 3，都是为了“接收某个长度的范围”而存在，我们只要明白这个道理以后就可以了。但是更多时候，使用者会比较倾向写法 1，当然笔者也是。最后，我们可以这样总结：

“在比较操作符的家族里 == 显然是使用频率最高的，此外 == 比较操作符不但可以用来用来针对第 N 个时钟触发事件，也可以用来指定某个范围的长度。反之 > && <= 是

一种用来指定某个范围长度的精简写法。至于 != >= 和 < 比较操作符在指定某个范围长度上使用的频率不见得频繁。”

好了，到目前为止读者已经了解了第二章的内容（思路）了，接下来笔者会陪同读者用第二章和第一章所学到的知识，重新设计 vga 模块和建立一个 screen 系统。

2.9 实验十：重新设计 vga 模块

笔者回味了一下，在建模篇建立 vga 模块的过程，那时候笔者单纯只用建模技巧，不问时序，不用仿真，用了一些小技巧，虽然避开了图像显示不规则（走样），但是却避开不了图像跑帧（右移几帧），跑帧的问题实际上是数据延迟 N 个时钟所导致的结果。建模篇里的 vga 实验，基本上是要读者了解建模技巧而已，然而它（实验）已经表现足够了。

说实话，要完全写好一个 vga 模块，事实上所需要的 Verilog HDL 语言技巧量一点也不少，要了解的概念也很多，如今我们有能力去设计一个更完善的 vga 模块。不知道读者还记得 vga 的时序吗？忘了？模糊？没办法，笔者再简单的刷新刷新一下读者的大脑。

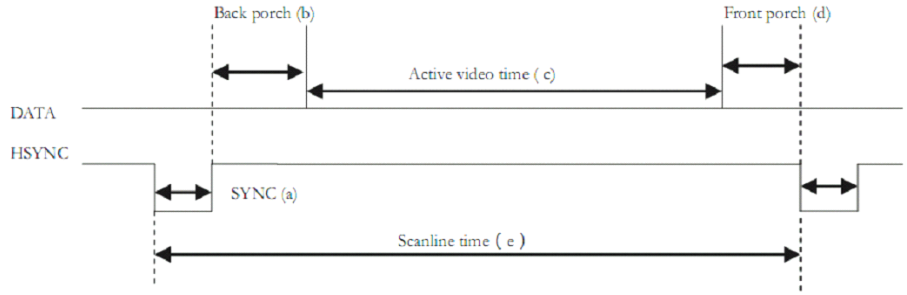


图 2.9a HSYNC 信号的时序图

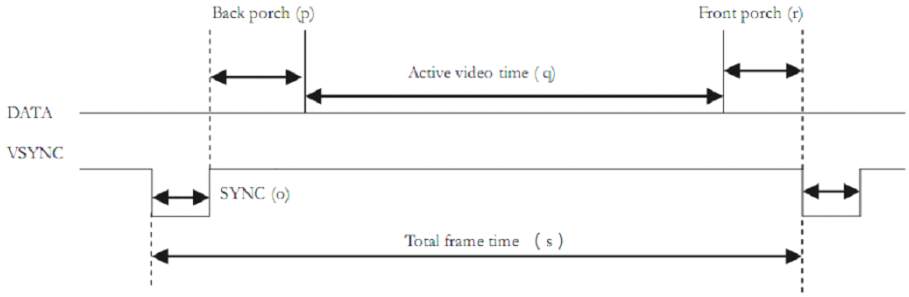


图 2.9b VSYNC 信号的时序图

图 2.9a 和图 2.9b 分别是 HSYNC 信号和 VSYNC 信号，它们构成了 vga 最基本的时序，么一个时序都有固定的“段”，如果以 800 x 600 @ 60Hz 的显示标准为例，那么详细的资料如表 2.9a 所示。

信号	a 段	b 段	c 段	d 段	e 段（总和）
HSYNC	128	88	800	40	1056
信号	o 段	p 段	q 段	r 段	s 段（总和）
VSYNC	4	23	600	1	628

表 2.9a 显示标准 800 x 600 @ 60Hz 的各段资料。

基本上除了 HSYNC 信号的 a 段和 c 段，VSYNC 信号的 o 段和 q 段以外其他的段压根儿也不重要。HSYNC 信号的 a 段是同步段，又称为起始段，c 段是列显示（扫描）的数

据有效段。然而 VSYNC 信号的 o 段也是同步段，也称为起始端，q 段是行显示（扫描）的数据有效段。HSYNC 总共有 1056 个列像素，每一个列像素的周期是 40Mhz 也是 25ns。反之 VSYNC 总共有 628 个行像素，每一个行像素的周期是 1056 个列像素，也就是 1056 * 25ns。

读者有没有觉得 vga 的时序和串口的传输协议有几分相似呢？

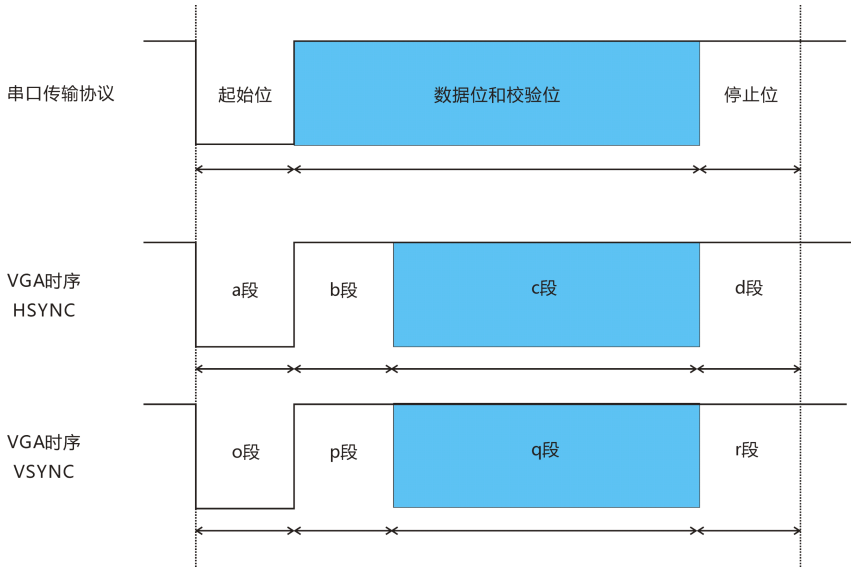


图 2.9c 串口传输协议和 VGA 时序的相识处。

如图 2.9c 所示，串口的一帧数据从低电平的起始位开始，然后致高电平的停止位结束。然而 VGA 的时序，HSYNC 吸纳后的 a 段和 VSYNC 信号的 o 段都是以低电平开始，然后 d 段和 r 段以高电平结束。

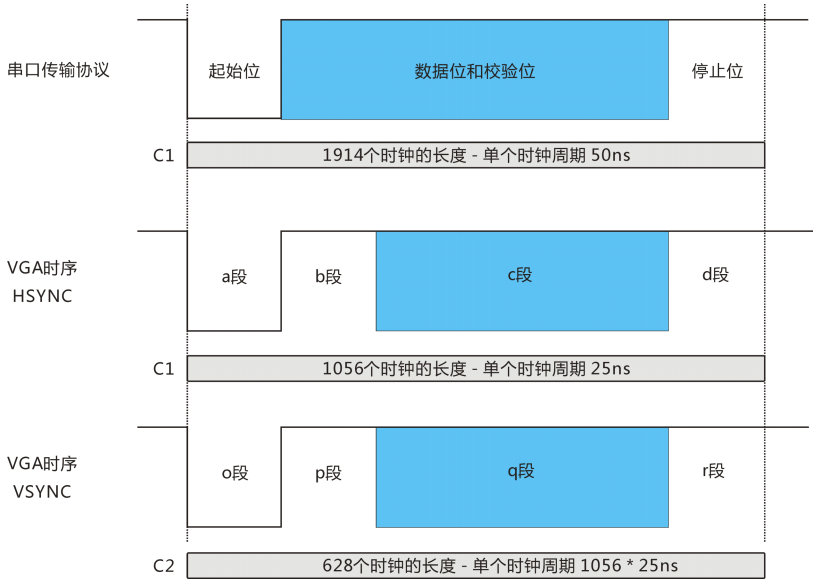


图 2.9d 用计数器去跟踪和计数每个信号的长度。

在串口的实验里，如果串口的一帧数据我们可以用 C1 去跟踪和计数的话，同样的道理，VGA 的时序，我们分别使用 C1 和 C2 去计数和跟踪 HSYNC 信号和 VSYNC 信号。不

过有点不同的是，串口中所使用的时钟周期几乎是一样的，亦即单个时钟周期是 50ns。反之在 VGA 的时序中，HSYNC 的单个时钟周期是 25ns，VSYNC 则是 1056 * 25ns，这个问题就是要考验我们，如果精确的控制 N 个时钟的长度。

sync_module.v

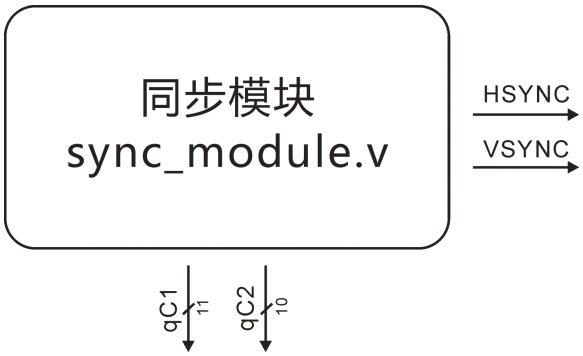


图 2.9e 同步模块的图形。

图 2.9e 是同步模块的通行，在 vga 的建模中它主要负责“显示标准”，此外还将 HSYNC 和 VSYNC 的计数器 qC1 信号和 qC2 信号反馈出来。具体的内容，我们还是来看代码吧。

```
module sync_module
(
        input CLK,
        input RSTn,
        output HSYNC,
        output VSYNC,
        output [10:0]qC1,
        output [9:0]qC2
);

        reg [10:0]C1;
        reg [9:0]C2;
        reg rH,rV;

        always @ ( posedge CLK or negedge RSTn )
            if( !RSTn )
                begin
                        C1 <= 11'd0;
                        C2 <= 10'd0;
                        rH <= 1'b1;
                        rV <= 1'b1;
                    end
```

```
23.                 else
24.                     begin
25.
26.                             if( C1 == 1056 ) rH <= 1'b0;
27.                             else if( C1 == 128    ) rH <= 1'b1;
28.                             //else if( C1 == 0 ) rH <= 1'b0;
29.
30.                             if( C2 == 628 ) rV <= 1'b0;
31.                             else if( C2 == 4    ) rV <= 1'b1;
32.                             //else if( C2 == 0 ) rV <= 1'b0;
33.
34.                             if( C2 == 628 ) C2 <= 10'd1;
35.                             else if( C1 == 1056 ) C2 <= C2 + 1'b1;
36.
37.                             if( C1 == 1056 ) C1 <= 11'd1;
38.                             else C1 <= C1 + 1'b1;
39.
40.                         end
41.
42.         /*******************************/
43.
44.           assign qC1 = C1;
45.           assign qC2 = C2;
46.
47.           assign HSYNC = rH;
48.           assign VSYNC = rV;
49.
50.           /*******************************/
51.
52.
53.   endmodule
```

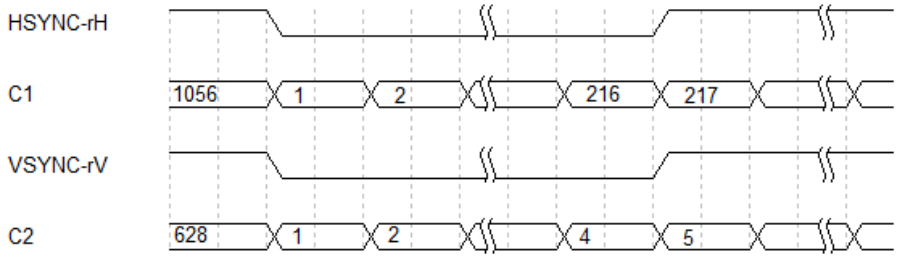
第 3~8 行是同步模块的输入输出声明，第 11~40 行是核心操作。第 11 行的 C1 是用来计数 HSYNC 信号，然而第 12 行的 C2 则是用来计数 VSYNC 信号。第 13 行的 rH 和 rV 分别是 HSYNC 和 VSYNC 信号的驱动寄存器。在复位的时候 C1 和 C2 都是清零（第 18~19 行），然而 rH 和 rV 则是拉高电平（第 20~21 行），原因很简单“HSYNC 信号和 VSYNC 信号在空闲的时候是处于高电平的”。

第 37~38 行是 C1 跟踪和计数 HSYNC 信号，由于黑金开发板的时钟频率只有 20Mhz，亦即 50ns 的单个时钟周期，所以在 vga 组合模块里需要用到 pll_module.v 将 20Mhz 倍频到 40Mhz，亦即单个时钟周期是 25ns。换句话说，同步模块是直接由 40Mhz 来驱动，也就是同步模块会以 800 * 600 @ 60Hz 为显示标准。在 37 行的 if 条件里的 C1 == 1056 表示 HSYNC 信号最大时钟数是 1056 个。当 37 行的 if 条件成立，那么 C1 会重新复制为 1，而不是清零（具体语言请看 2.7 小节）。

第 34~35 行是 C2 跟踪和计数 VSYNC 信号，它是以 C1 为基础，当 C1 计数满 1056 个时钟以后，C2 就会递增 1（35 行）。第 34 行则表示，当 C2 计数满 628 个时钟以后就赋值为 1，而不是清零。

第 26~28 行是 rH 根据 C1 的计数而相关的拉高或拉低，其中唯一让读者困惑不解的是，为什么笔者会注释掉 28 行呢？原因很简单：当同步模块开始工作的时候 C1 的复位值为 0，那么 rH 会被拉低（28 行），然后直到 C1 计数到 128 个时钟的时候（27 行）rH 会被拉高。最后当 C1 计数满 1056 个时钟以后（26 行），rH 会被拉低，C1 会从 1 开始计数。由此可见，第 28 行只有在 C1 循环计数的第一次，也是仅有的一次才被执行，余下的循环计数第 28 行就没有关系了。注释掉也就是为了省逻辑资源 ....

可能读者要问："注释掉了第 28 行，同步模块会不会受影响？" 嗯 ... 基本上影响不大，只是在 C1 的第一次循环计数中 rH 是没有操作而已，余下的循环计数 rH 就会回复工作了。同样的道理地 30~32 行的 32 行也被注释掉了，不同的是第 30~32 行是以 C2 为计数，则跟踪和计数对象是 rV。rV 在 C2 计数满 628 个时钟的以后会拉低（30 行），反之 rV 在 C2 计数到第 4 个时钟以后会拉低（31 行）。



2.9f 同步模块的大概时序图。

由于这个实验涉及到许多 quartus 自带的 ip 如 pll 模块或者 rom 模块，所以笔者就不玩仿真了。取而代之，我们只是使用第二章所学到的概念，去实现一个"主动"的设计（主动 - 不过分依赖仿真，设计者只按思路和对自己掌握 Verilog 的的信心去完成设计)。图 2.9f 是同步模块的大概时序图，读者只要明白时序和时间点事件的概念就能看懂了，笔者就不多话了（不懂得家伙要多复习）。第 44~48 行是相关的驱动和输出。

rom_module.v

rom 模块在这里的功用就是储存图像信息。实验十需要显示 128 * 128 分辨率的图像，此外 rom 模块以一个字节的基础去储存图像信息。一副 128 * 128"点阵"图像信息的容量是：

128 * 128 = 16384 bits

多少字？

16384 / 8 = 2048 words

图 2.9f.2 128 x 128 的图像信息（还是比卡丘）。

图 2.9f.2 是我们要显示的图像信息，还是笔者最爱的比卡丘（呜哇哇，实在太可爱了）。至于“一个字节的基础去储存图像信息”是什么意思？接下来，笔者会慢慢解释的。

vga_control_module.v

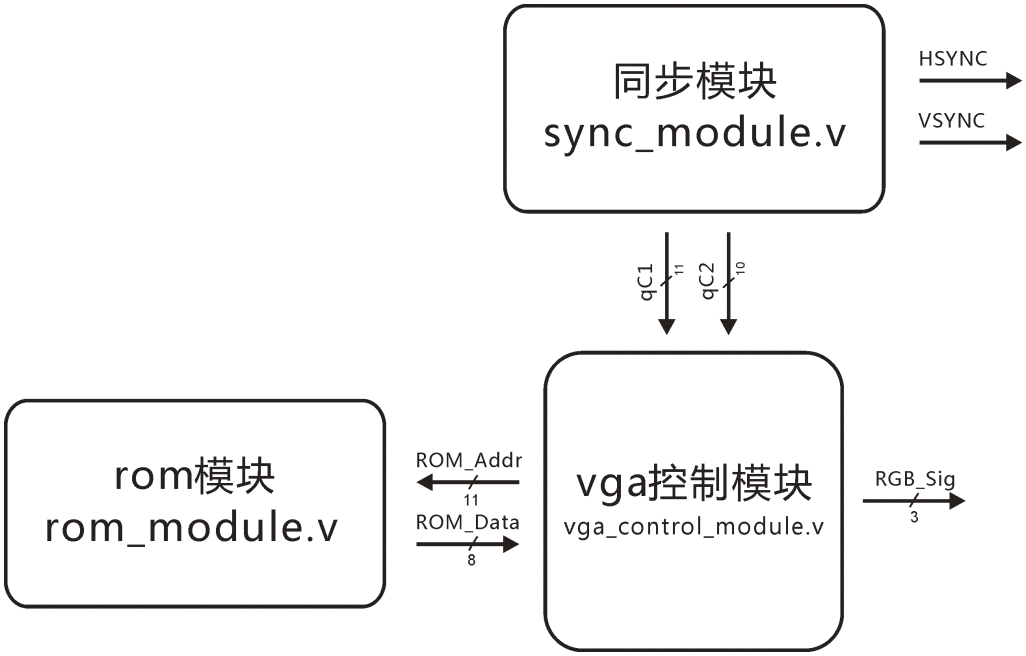


图 2.9f vga 控制模块的图形。

vga 控制模块在建模中的位置依然不变，它主要是负责图像处理和显示。如图 2.9f 所示，vga 控制模块接收来之同步模块的 qC1 和 qC2 信号，然后还从 rom 模块里读取相关的图像，最后处理图像信息讲其内容输出至 RGB_Sig。此外在实验十里的 vga 控制模块的功能已经升级了，它除了上诉的功能以外，它还可以定制显示屏幕的大小和位置。它到底是如何做到的？然而它有如何去实现和操作呢？

在这里，笔者先要求读者把“同步模块”和“vga 控制模块”是互不相干的东西，用另一句来说就是“同步模块只要负责而且正确的输出 qC1 和 qC2 即可”。反之“vga 控制模块只要负责如何要正确的接收 qC1 和 qC2 即可”，如 2.6 小节实验九的“思路”那样“串口发送模块只要正确无误发送一帧数据”，然而“串口接收模块子要确保无误的接

收一帧数据即可”。

因为在“主动”的设计中，我们不可能每时每刻都用仿真软件“一个时钟一个中的去观察两个模块如何沟通”。亦即，前一个模块操作状态和后一个模块操作状态也 okay，然后再遵守“模块沟通的硬道理”——模块的沟通至少需要一个时钟，把精力只焦距在“如何处理部分的沟通”而不是“以全局的方式观察全部的沟通”。

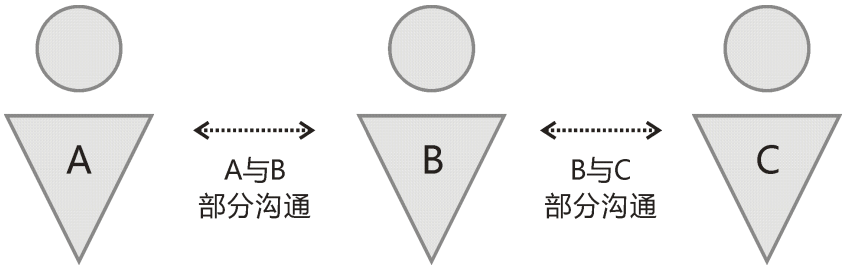


图 2.9g A，B 与 C 均为局部者，各之间都有部分的沟通。

举个具体的例子：如图 2.9h 所示，有 3 个小朋友分别是 A ，B 和 C，他们均为局部者，然而每一个小朋友的沟通都是局部性而且是部分的，亦即 A 与 B 沟通，B 与 C 沟通。

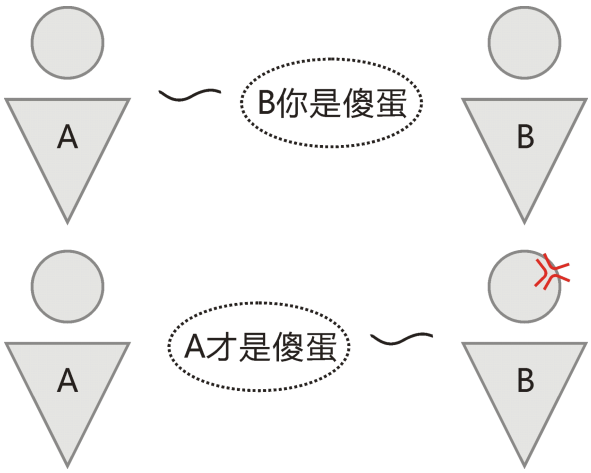


图 2.9h A 和 B 的部分沟通。

假设 A 要与 B 沟通，那么 A 只要准确的抛出话语“B 你是傻蛋”，然后这句话会在空气中摇摇摆摆的传播到 B 的耳中，这时候 B 就要细心而且是正确无误的竖起耳朵听这句 A 抛出的话语。B 经过严格的处理和分析这句话的意思，然后面红耳赤的返回到“A 才是傻蛋”，然后这句话会经空气传播，这时 A 竖起耳朵安静而且专注的聆听 B 返回的话语。

为了使 A 和 B 可以方便的沟通“A 和 B 都要说人话，而不是鬼话”，然而人话的定义“就是一串长长经过时钟化的信号”。此外“话语在空气中慢慢的传播”亦即“模块沟通所产生的延迟”，这个延迟至少需要一个时钟。现实点看，就是 A 在 T0 说话，那么 B 在 T1 才能听到话。

所以说，我们不要第三者（被动式）来监督 A 和 B 之间的沟通记录。设计者只要站在 A

的立场确保 A 可以有效把“B 你是傻蛋”抛出，换之设计者只要站在 B 的立场确保 B 有效的接收到“B 你是傻蛋”这句话。反回来，设计者又要站在 B 的立场确保 B 有效并且愤怒的抛出“A 才是傻蛋”，换之设计者又要站在 A 的立场确保 A 可以有效的接收到“A 才是傻蛋”这句话。

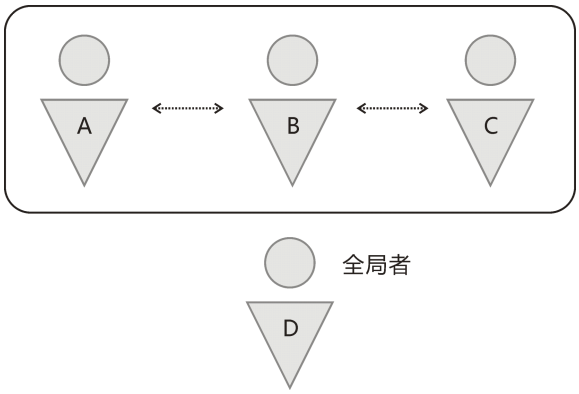


图 2.9i D—全局的观察者，监督 A，B 与 C 的所有沟通记录。

在时序篇中，或者在以往的实验中，设计者永远扮演全局的观察者 D，以局外的身份，用全知的心智（仿真工具）观察 A，B 与 C 的所有沟通记录（如图 2.9i 所示）。基本上，虽说这是最正常也是最安全的方法，对于小设计（小范围）来说还 Okay，但是对于大设计（大范围）的话，结果设计者会仿真到精神衰弱，最后会精尽人亡。

在这里可能有读者要问道：“类似局部设计的思想，真的安全吗？”只要 A，B 和 C 都说人话，亦即所有模块都遵守同一样的沟通规则：

第一，信号必须时钟化，亦即可以用“一个时钟，一块数据”去理解；
第二，沟通口（输出的一方）不要使用组合逻辑（即时事件）来驱动，因为即时事件是无视时钟的（这也是 2.4 小节所要表达的东西）。

如此一来，我们可以省下一边设计一边仿真的所带来的劳累和时间消耗。事实上设计者也可以完全不用仿真（如果他可以倒着 Verilog 来用的话）。好了，比喻和例子都有点扯远了，不明白的同学只要耐心的看下去，概念就会在脑中朦朦的显现。

==========

参数	值
分辨率	128 x 128
位置	X-0，Y-0
图像信息最大位宽	一字节（8 位）
操作方式	流水操作

表 2.9b vga 控制模块的参数表。

我们要以 2.9b 的参数表来设计 vga 控制模块。对于分辨率来说，在以往的设计中分辨率的最大是依据图像信息的位宽，也就说 64 * 64 大小的分辨率就左右着图像信息的最

大位宽为 64 位。但是根据 Quartus 的“优化”角度，位宽最大也是最佳是 64 位，虽说这不一定要去遵守的铁则，但是为了提升读者自身的 Verilog 使用能力，我们还是换一换口味来设计 vga 控制模块。由此，图像信息最大的位宽我们以一字节为基础，打个比方：假设有 0~15 的列像素，分别使用 2 字节的图像信息去填充。

最后一点，让读者感到流汗的是 vga 控制模块的操作方式。在以往的实验里，vga 控制模块也是按流水操作的方式去运作，但不同的是，那时候的设计笔者不谈时序，笔者尽可能用更多的即时事件去压缩时钟的延迟，然后也对图像信息动了一些手脚（图像最外围留空），根本就谈不上什么正式的流水操作。

```
1.  module vga_control_module
2.  (
3.          input CLK,
4.          input RSTn,
5.          input [10:0]qC1,
6.          input [9:0]qC2,
7.          output [2:0]RGB_Sig,
8.          input [7:0]ROM_Data,
9.          output [10:0]ROM_Addr
10.
11. );
12.
13.       /*************************************/
14.
15.         // Height , width, x-offset and y-offset
16.         parameter _X = 8'd128, _Y = 8'd128, _XOFF = 10'd0, _YOFF = 10'd0;
17.
18.         /*************************************/
19.
20.         reg [6:0]x,y;
21.         reg [2:0]n1,n2;
22.         reg [10:0]rAddr;
23.         reg [2:0]rRGB;
24.
25.       always @ ( posedge CLK or negedge RSTn )
26.               if( !RSTn )
27.                       begin
28.                               x <= 7'd0;
29.                               y <= 7'd0;
30.                               n1 <= 3'd0;
31.                               n2 <= 3'd0;
32.                               rAddr <= 11'd0;
33.                               rRGB <= 3'd0;
34.                           end
```

```
35.                 else
36.                     begin
37.
38.                         // step 1 : compute data address and index-n
39.                         if( (qC1 > 128 + 88 + _XOFF && qC1 <= 128 + 88 + _XOFF +_X) &&
40.                             (qC2 > 4 + 23 + _YOFF && qC2 <= 4 + 23 + _YOFF + _Y) )
41.                             begin
42.                                 x = qC1 - _XOFF - 128 - 88 - 1;
43.                                 y = qC2 - _YOFF - 4 - 23 - 1;
44.                                 rAddr <= (y << 4) + (x >> 3);
45.                                 n1 <= ( x - ( (x >> 3) << 3) ); // x & 0x07
46.                             end
47.                         else
48.                             begin n1 <= 3'd0; rAddr <= 11'd17; end
49.
50.                         // step 2 : reading data from rom
51.                         n2 <= n1;
52.
53.                         // step 3 : assign RGB_Sig
54.                         rRGB <= { ROM_Data[n2], ROM_Data[n2], ROM_Data[n2] };
55.
56.                     end
57.
58.         /****************************/
59.
60.         assign RGB_Sig = rRGB;
61.         assign ROM_Addr = rAddr;
62.
63.         /****************************/
64.
65.     endmodule
```

第 3~9 行是该模块的输入输出口的声明。第 16 行是 vga 控制模块，控制并且显示屏幕的大小和位置的常量声明。第 20~23 行是相关寄存器的声明；第 20 行的 x 和 y 寄存器是用来计算和寄存 x 和 y 位置的“即时结果”；第 21 行的 n1 和 n2 是在流水操作中，图像信息的“位寻址”的“参考数据”；第 22 行的 rAddr 是用来驱动 ROM_Addr 的寄存器；第 23 行是用来驱动 RGB_Sig 信号的寄存器。第 25~56 行是该控制模块的核心操作。

对于 vga 控制模块来说“什么时候要显示第 n 个列像素和第 n 个行像素”，同步模块-sync_module.v 完全以 qC1 和 qC2 的方式告诉它。继续以 800 * 600 @ 60Hz 的显示标准为说明，当 qC1 开始计算到 c 段，亦即 qC1 从 217 计数开始，那么 217 就表示“第 0 个列像素”。同样的道理，当 qC2 开始计数到 q 段，亦即 qC2 从 28 计数开始，那么 28 就是“第 0 个行像素”。

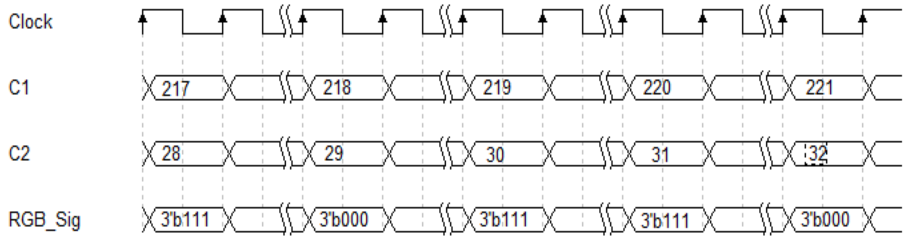


图 2.9j 一个例子的时序图。

假设一个例子，如图 2.9j 所示：

当 C1（qC1）为 217，C2（qC2）为 28 的时候，然而 RGB_Sig 的结果是 3'b111，这也表示“第 0 行第 0 列显示点”。
当 C1（qC1）为 218，C2（qC2）为 29 的时候，然而 RGB_Sig 的结果是 3'b000，这也表示“第 1 行第 1 列显示灭”。
当 C1（qC1）为 219，C2（qC2）为 30 的时候，然而 RGB_Sig 的结果是 3'b111，这也表示“第 2 行第 2 列显示点”。
当 C1（qC1）为 220，C2（qC2）为 31 的时候，然而 RGB_Sig 的结果是 3'b111，这也表示“第 3 行第 3 列显示点”。
当 C1（qC1）为 221，C2（qC2）为 32 的时候，然而 RGB_Sig 的结果是 3'b000，这也表示“第 4 行第 4 列显示点”。

第 39~40 行是“接收某段信号长度”或者说“某个范围的 N 个时钟”（这也是 2.8 小节所谈到的重点）。第 39 行是表示接收 qC1 的长度范围，亦即“接收某个长度的列像素”，反之第 40 行是表示接收 qC2，亦即“就收某个长度的行像素”。继续以 800 * 600 @ 60Hz 显示标准为例，实验十的 vga 控制模块要接收 128 宽（列）* 128 高（行）的长度范围，那么 qC1 是 217~ 344，qC2 则是 28~155。

如果我们先无视 XOFF 和 YOFF 显示位置的偏移量（亦即显示在 X-0，Y-0 的位置），那么列像素的长度是：

qC1 > 128 + 88（某个列长度的开始）&& qC1 <= 128 + 88 + 128 (某个长度的结束)

那么行像素的长度是：

qC2 > 4 + 23（某个行长度的开始）&& qC2 <= 4 + 23 + 128（某个行长度的结束）

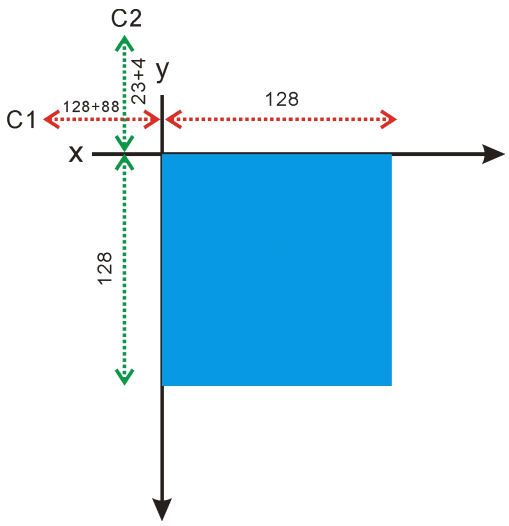


图 2.9j 显示的概念图。

图 2.9j 是笔者脑海中大概的概念图，读者自己看着办吧。第 42~43 行的 x 和 y 寄存器是为了转换 qC1 和 qC2 成为 x 和 y 像素（位置），注意 x 和 y 都是“即时事件”。举个例子，假设 qC1 和 qC2 现在的值为 300 和 100，那么（先无视 XOFF 和 YOFF）：

```
x = qC1 - XOFF - 128 - 88 -1
  = 300 - 0 - 128 - 88 -1
  = 83

y = qC2 - YOFF - 4 - 23 - 1
  = 100 - 0 - 4 - 23 - 1
  = 72
```

结果就是“当前的同步模块正在显示第 83 个列像素，第 72 个行像素”（其中 -1 代表从第 0 个行|列像素开始算起）。

第 44 行的 rAddr <= (y << 4) + (x >> 3) 却比较难解释了 ... 在前面笔者说过图像信息是以 1 字节为储存基础，我们的 rom 模块是储存一只 128 * 128 的小可爱，其中“字数”（words）到达 2048。由于 vga 控制模块同样也是显示分辨率为 128 * 128 ，

那么 1 行是：

```
128 bits = 16 words * 8 bits
         = 16 words * 1 Bytes
```

结果每一行的偏移量是 16。笔者还是画出一张表格来说话吧，单单只是以文字来表达，实在太对不起那些懒得在脑中画画的朋友。

行\列-字节	0	1	2	3	...	13	14	15
0	0	1	2	3	...	13	14	15
1	16	17	18	19	...	29	30	31
...	...	...	...	...	...	...	...	...
126	2016	2017	2018	2019	...	2029	2030	2031
127	2032	2033	2034	2035	...	2045	2046	2047

表 2.9c　128 * 128 的分辨率，图像信息以字节为基础（图像信息显示部分）。

表 2.9c 就是 128 * 128 分辨率，然后图像信息以储存基础。总结一句，一副 128 * 128 的图像信息，如果以直接为储存基础，那么一共有 2048 字 * 1 Bytes 亦即 2048 * 8 bit。然而每一行德偏移量是 16 个字。 所以说，第 44 行的 rAddr <= (y << 4) + (x >> 3), 其中 y << 4 就是把 y 个像素转换为 rom 模块中对应中“行偏移量”。至于 x >> 3 呢？

行\列-字节	0	1	2	3	4	...	6	14	15
0	8'hFF	8'hEE	8'h88	8'h77	8'h44	...	8'h11	8'h00	8h'22
...	...	...	...	...	...	...	...	...	...
行\列-字节	2032	2033	2034	2035	2036		2045	2046	2047
127	8'h01	8'h02	8'h03	8'h04	8'h05	...	8'h06	8'h07	8'h08

表 2.9d　rom 模块里第 0 行前 8 个字节（图像信息储存部分）。

假设 vga 模块要从表 2.9d 中读取数据，如果当前 y 为 0 ，x 为 0~7，亦即对应 rom 模块中第 0 字的数据。笔者简单的计算一下：

```
rAddr   = ( y<<4 ) + ( x >> 3 ) = ( 0 * 16 ) + ( 0/ 8 ) = 0 + 0 = 0
......
rAddr   = ( y<<4 ) + ( x >> 3 ) = ( 0* 16 ) + ( 7/ 8 ) = 0 + 0 = 0
```

又如果 y 为 0，x 为 120~127，那么：

```
rAddr   = ( y<<4 ) + ( x >> 3 ) = ( 0 * 16 ) + ( 120/ 8 ) = 0 + 15 = 15
......
rAddr   = ( y<<4 ) + ( x >> 3 ) = ( 0* 16 ) + ( 127/ 8 ) = 0 + 15 = 15
```

笔者再假设 y 为 127 , x 为 112~119 那么：

```
rAddr   = ( y<<4 ) + ( x >> 3 ) = ( 127 * 16 ) + ( 112/ 8 ) = 2032 + 14 = 2046
......
rAddr   = ( y<<4 ) + ( x >> 3 ) = ( 127* 16 ) + ( 119/ 8 ) = 2032 + 14 = 2046
```

嗯 ... 笔者还是绘制一张概念图来得更直接，结果如图 2.9k 所示：

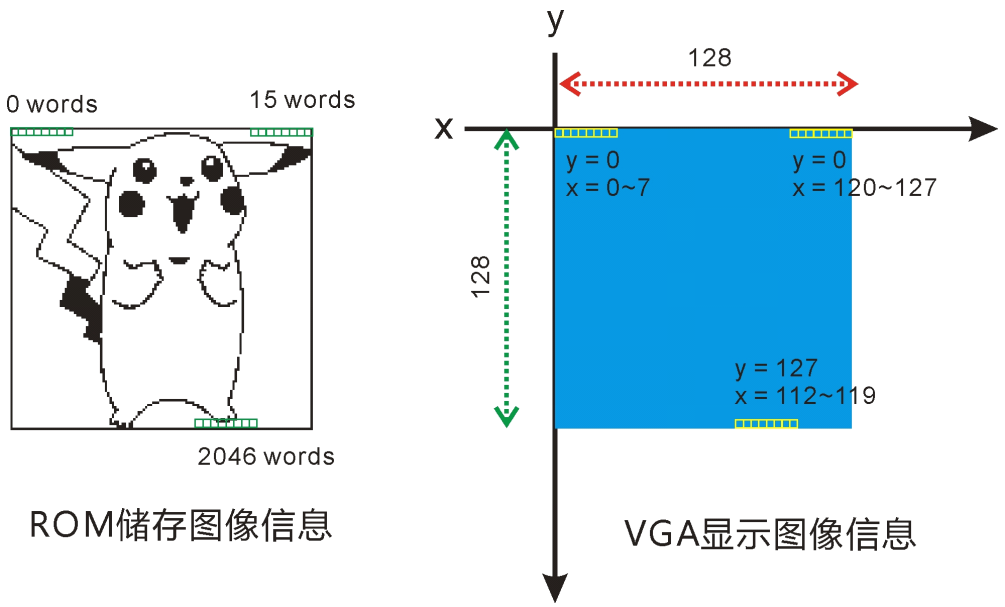


图 2.9k 按 vga 显示的 x 与 y，从 rom 读取字节的图像信息。

```
45.                                    n1 <= ( x - ( (x >> 3) << 3) ); // x & 0x07
```

第 45 行的表达式是取得“位寻址”。假设 x 为 7，y 为 0 ，那么 vga 控制模块会从 rom 模块中读取第 0 字节的图像信息。但是，读者不要忘了 vga 的列扫描方式（列填充方式）“是以一位一位”去填充，而“不是以一字节一字”去填充，所以我们计算出“当前列填充”的“位寻址”。
其中 x - (x >> 3) << 3 和 x & 0x07 是等价的，继续以 x 为 7 ，y 为 0 为例：

```
n1  = 7 - ( 7 >> 3 ) << 3
    = 7 - 0
    = 7
n1  = 8'b0000_0111 & 8'b0000_0111
    = 8'b0000_0111
```

再假设 x 为 128，y 为 127 ，那么：

```
n1  = 118 - ( 118 >>3 ) << 3
    = 118 - (118/8) * 8
    = 118 - 14 * 8
    = 118 - 112
    = 6
n1  = 8'b0111_0110 & 8'b0000_0111
    = 8'b0000_0110
```

总结说，假设 x 为 128，y 为 127，那么我们会从 rom 模块里读出第 2046 words 的图像信息（字节），x 为 128 这也表示，要从第 2046 words 图像信息中“寻址”第 6 位“点阵信息”。

==========

```
44.                                     rAddr <= (y << 4) + (x >> 3);
45.                                     n1 <= ( x - ( (x >> 3) << 3) ); // x & 0x07
46.                                 end
47.                             else
48.                                 begin n1 <= 3'd0; rAddr <= 11'd17; end
49.
50.                             // step 2 : reading data from rom
51.                             n2 <= n1;
52.
53.                             // step 3 : assign RGB_Sig
54.                             rRGB <= { ROM_Data[n2], ROM_Data[n2], ROM_Data[n2] };
```

第 47~48 行是表示“超出 C1 和 C2 长度接收的范围之外”，这时候读者可以任意寻找在 rom 模块中“任意 words 的图像信息和任意的位寻址”最重要的结果是“那一点图像信息是灭的”。

至于 50~54 行，我们要结合从 38~54 行来看，因为这是流水操作的步骤。读者还记得怎样设计流水操作吗？首先，我们要以“步骤”的方式去建立操作，如 vga 控制模块：

（一）计算出 x 和 y 对应中，rom 模块里的“第几个 words”的图像信息（第 44 行）。同是也计算出 x 对应中的“位寻址”（第 45 行）。
（二）从 rom 模块里读出图像信息（字节），然后传送“上一个位寻址”（50~51 行）。
（三）把读出来的图像信息，按“位寻址”分别驱动 Red Green 和 Blue 信号（53~54 行）。

图 2.9l vga 控制模块的流水操作。

只要知道“步骤”的操作方式以后，我们就可以建立起流水操作方式了。vga 控制模块有“3 个操作步骤”，亦即流水操作需要 3 个任务：第一个任务是计算出 x 和 y 所对应的第 words 图像信息（字节）的地址和出 x 所对应的“位寻址”；第二个任务就是从 rom 模块里取得图像信息；第三个任务就偶是按位寻址 n 驱动 RGB 信号。其中 n 是参考值。

当读者看到这里，估计脑子里已经充满着许多的问号 .... 读者可能要问：“为什么 vga 控制模块是流水操作？”嗯 ..... 笔者也不知道要如何回答，总之我们先换另一种角度去看待问题。

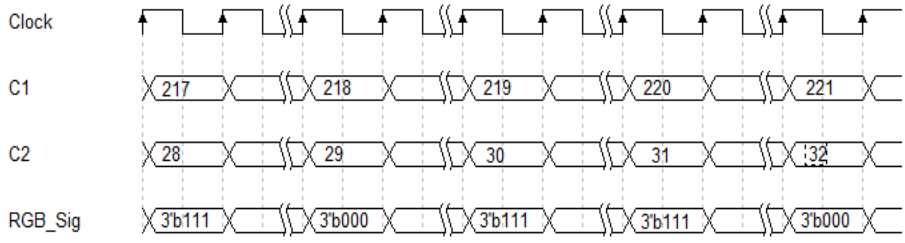


图 2.9j 一个例子的时序图。

在一个理想的状态下，当 vga 控制模块接收到 C1 和 C2 的同时会驱动相应的 RGB_Sig，换句话说就是 C1 和 C2 的结果应该和 RGB_Sig 的结果同步进行（如图 2.9j 所示），假设在某 TA 的时候，C1 为 217，C2 为 28，RGB_Sig 的结果“理应”也是在 TA 的时候和 C1 和 C2 同步输出。

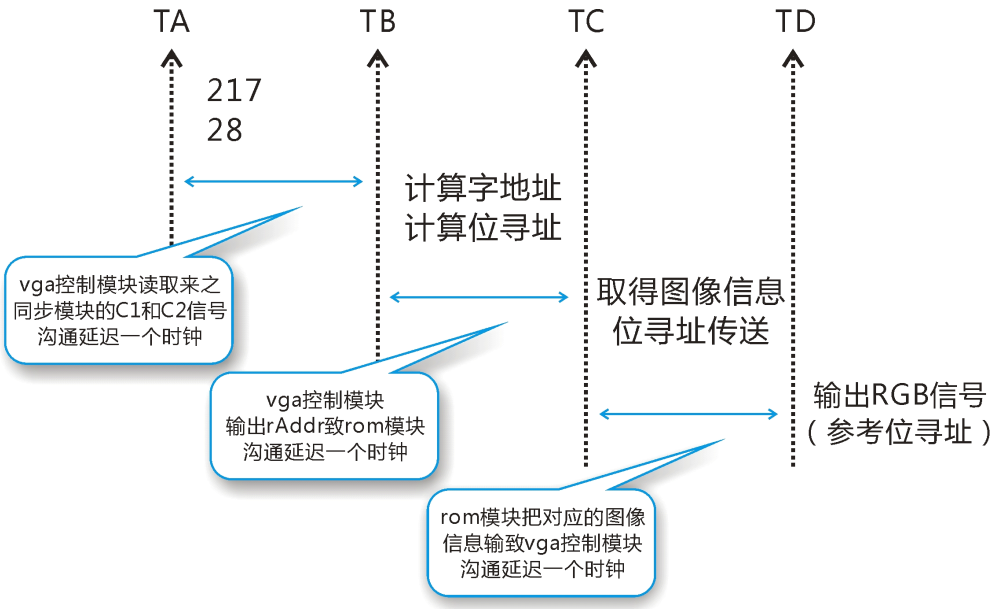


图 2.9m 从 vga 控制模块接收到 C1 和 C2 致输出 RGB 信号的延迟。

实际上 RGB 信号需要经过几个时钟的延迟才能生成，换句话说 RGB 信号无法同步和 C1 和 C2 同步输出，如图 2.9m 所示。要读懂图 2.9m 就得用“时间点事件”去解读：

在 TA 的时候同步模块输出 C1 和 C2 均为 217 和 28，然后在 TB 的时候 vga 控制模块接收到 217 和 28（第一个时钟的延迟）。于此同时 vga 控制模块计算出 rAddr 和位寻址，然后它向 rom 模块发送 rAddr 结果（第二个时钟的延迟）。在 TC 的时候，rom 模块取得 rAddr 的结果，然后输出对应的图像信息致 vga 控制模块（第三个时钟延迟）。在 TD 的时候，vga 控制模块将图像信息，然后按位寻址驱动 RGB 信号。

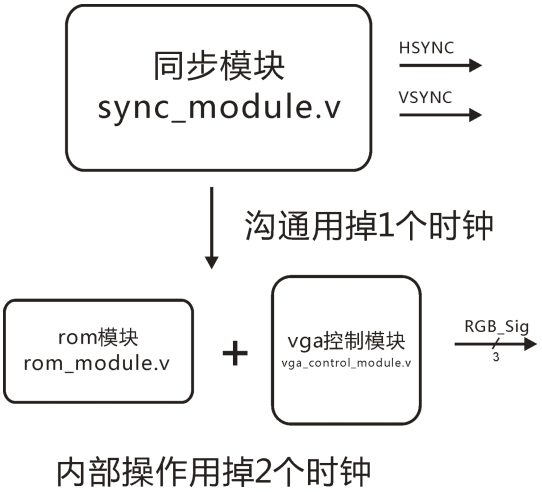


图 2.9n 在建模中看时钟延迟。

总结来说，同步模块和 vga 控制模块沟通耗掉一个时钟，再加上内部操作所耗掉的两个时钟，一共耗掉 3 个时钟。换句话说，假设同步模块当前输出 C1 和 C2 为 217 和 28，亦即同步模块当前要扫描第 0 行第 0 列，但是对性的 RGB_Sig（图像信息）需要经过 3 个时钟以后才能出现 ....

为了解决这个办法，第一我们提高 vga 控制模块的频率高于同步模块高于 3 被，亦即同步模块工作频率为 40Mhz，那么 vga 控制模块就需要 120Mhz。或者用更多的即时事件，把时钟的消耗压缩到不成人形。余下的方法我们只能使用流水操作而已。事实上，同步模块提供给 vga 控制模块的 qC1 和 qC2 是源源不断的，从另一个角度来看 qC1 和 qC2 就是同步模块提供给 vga 控制模块的“数据流”。

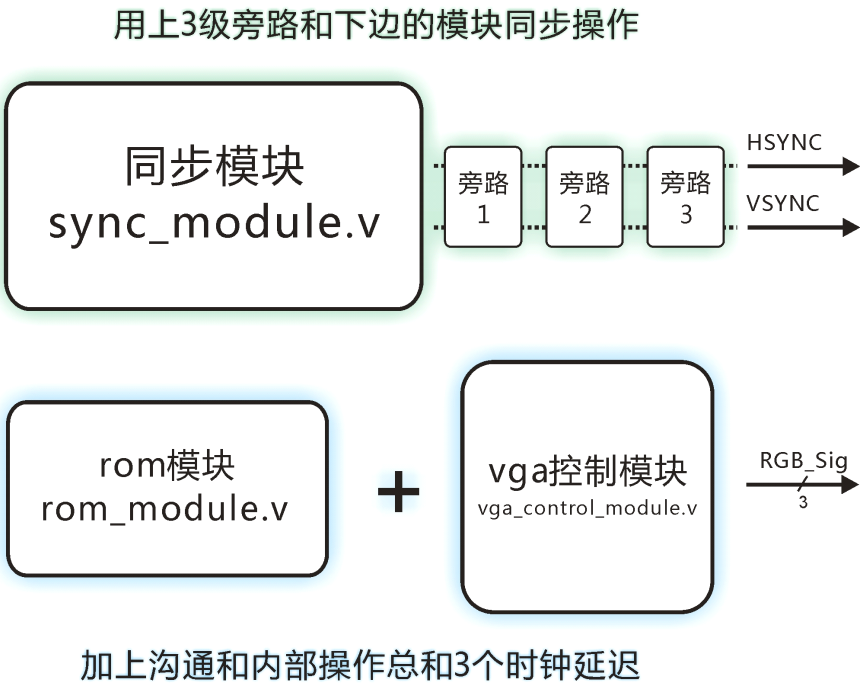


图 2.9o 用旁路寄存器来同步操作。

当某个操作时流水操作的话，那么我们就可以使用“旁路寄存器”来延迟某个信号，从而使两个或以上的信号对齐和同步。如图 2.9o 所示，同步模块用了 3 级的旁路寄存器使得和下边的模块同步操作。

这时候的读者估计已经连蛋蛋都要憋破了 ... “笔者你到底在说什么，怎么我越来越糊涂了，vga 控制模块用流水操作与 C1，C2 和 RGB_Sig 同步与用旁路寄存器来延迟 HSYNC 和 VSYNC 信号，这之间到底有什么关系？

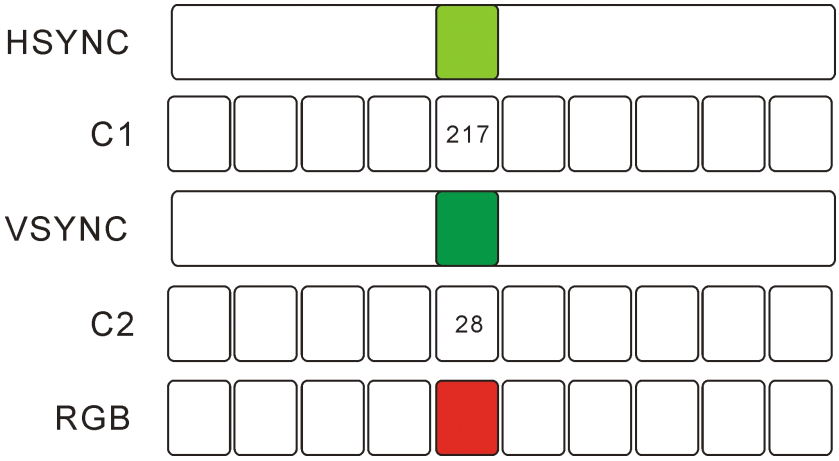


图 2.9p 信号时钟化的概念图。

打从一开始，HSYNC 和 VSYNC 信号已经被 C1 和 C2 时钟化了，或者说“C1 是可计数的 HSYNC 信号，C2 是可计数的 VSYNC 信号”。假设 HSYNC 信号有一块“浅绿色的数据”恰好对应 C1 的 217；同样 VSYNC 信号也有一块“深绿色的数据”恰好对应 C2 的 28。事实上，我们知道 C1 为 217，C2 为 28，亦即同步模块在扫描第 0 列第 0 行，然而要显示的“图像信息”是 RGB 信号中的“红色的数据”（如图 2.9p 所示）。

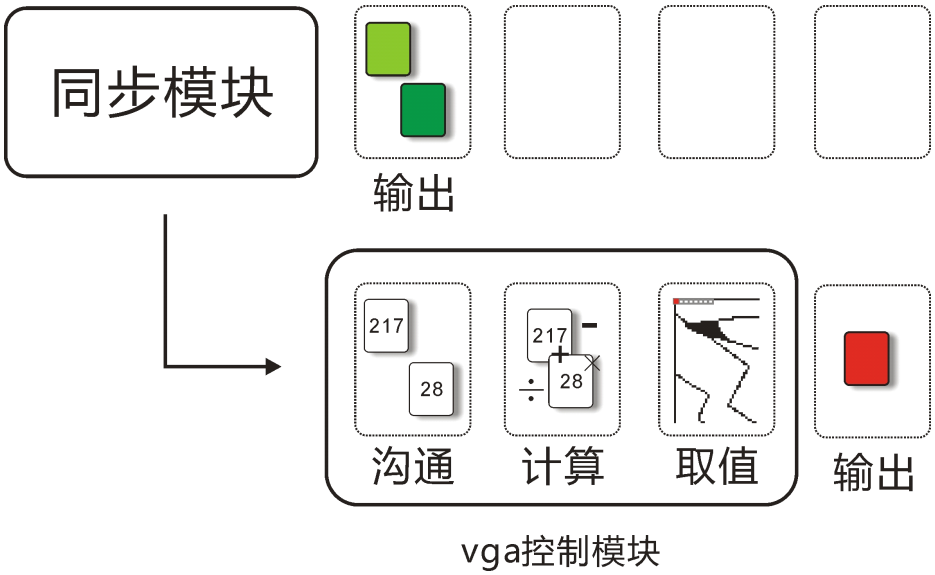
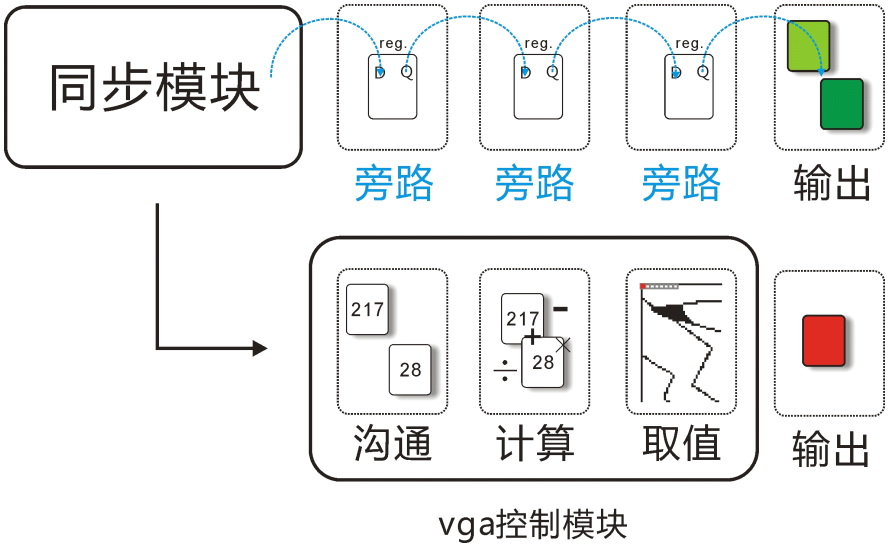


图 2.9q RGB 信号输出所经过的延迟。

事实上“浅绿色数据”和“深绿色数据“无法和“红色数据”同步输出，如图 2.9q 所示。

对于同步模块来说，它不能直接把“浅绿色数据”和“深绿色数据”输给 vga 控制模块，因为 vga 控制模块根本不认识它们，换之同步模块需要把“时钟化”后的 HSYNC 和 VSYNC 信号，亦即 C1 和 C2，输给 vga 控制模块才有用。经过时钟化后的 HSYNC 和 VSYNC 信号是“有标签”和“可操作”的。

当同步模块把“浅绿色数据”和“深绿色数据”所对应的 217 和 28 输给 vga 控制模块后，在“模块的沟通”上延迟了一个时钟；然后又在“217 和 28 计算为 rom 地址”上延迟了一个时钟；最后为了“取得相对应的图像信息”再一次延迟了一个时钟。结论，“vga 控制模块”生成“红色数据”至少需要 3 个时钟，结果造成“浅绿色数据”，“深绿色”和“红色数据”不同步。



2.9q 使用旁路寄存器同步输出 3 个不同颜色的数据。

为了使 3 个不同颜色的数据同步输出，浅绿色和深绿色数据输出之前，使用了 3 级的旁路寄存器使得和红色数据对齐（如图 2.9q 所示）。对于非流水操作来说，旁路寄存器除了延迟了数据 N 个时钟以外没有任何用处。反之对于流水操作来说，旁路寄存器用于同步数据的输出。

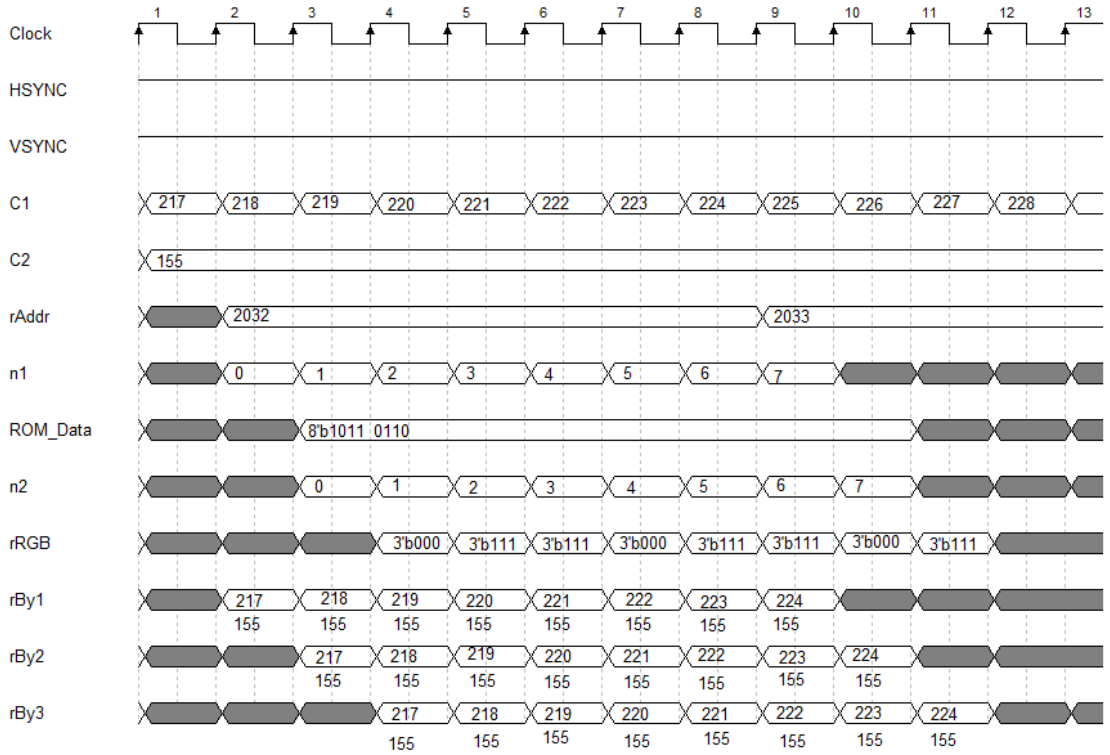


图 2.9r 一段时序经过的例子。

图 2.9r 是一个简单的例子，主要是发生在 C2 为 155，C1 为 217~224，亦即 y 为 127，x 为 0~7 之中。在图 2.9r 中 HSYNC 和 VSYNC 信号基本上已经是无意义了，因为这个信号是输出给液晶或者 CRT 看的，对于 vga 控制之模块来说 C1 和 C2 才有意义。

我们先看 vga 控制模块生成“红色数据”的经过：

（一）在 T1 的时候，同步模块开始输出一堆数据（C1 和 C2）给 vga 控制模块。
（二）在 T2 的时候，vga 控制模块开始计算 rAddr，C1 和 C2 均为 155 和 217，亦即 y 为 127，x 为 0，计算出来的结果也是 2032，结果从 rom 模块中寻找第 2032 个 words。在同一个时候也计算出 x 为 0 相对应的“位寻址-0”。
（三）在 T3 的时候，rom 模块吐出第 2032 words 所对应的图像信息—8'b1011_0110（笔者随意举例而已），在同一个时候 n1 把位寻址-0 传送给 n2（应该说 n2 读取 n1 中的位寻址-0）。
（四）在 T4 的时候，vga 控制模块根据 n2 和 ROM_Data 的结果驱动 rRGB。

接下来我们再来看，同步模块的“浅绿色数据”和“深绿色数据”被旁路的经过：

（一）在 T2 的时候 rBy1 开始读取 C1 和 C2 中的 217 和 155。
（二）在 T3 的时候 rBy2 开始读取 rBy1 的 217 和 155。
（三）在 T4 的时候 rBy3 开始读取 rBy2 的 217 和 155。

事实上 rBy1~3 应该旁路 HSYNC 和 VSYNC 的结果，可以如果笔者这样作的话图 2.9r 就没有意义了。在前面笔者已经说过了，HSYNC 和 VSYNC 基本上已经投影在 C1 和

C2 的身上，所以旁路 C1 与 C2 和旁路 HSYNC 和 VSYNC 是等价的。在图 2.9r 中，重点就只有一个，就是“在 T4 的 rBy3 和 rRGB”。这时候我们可以说，“浅绿色数据”，“深绿色数据”和“红色数据”已经同步了。

由于 vga 控制模块是流水操作，所以只要同步模块不停给它发送数据，那么数据的输出就会远远不断，如图 2.9r 所示中的 T1~T8。同步模块给 vga 控制模块发送 C1-217~224 和 C2-155，那么 vga 控制模块“在每一个时钟里”都处理这些“数据流”。

vga_module.v

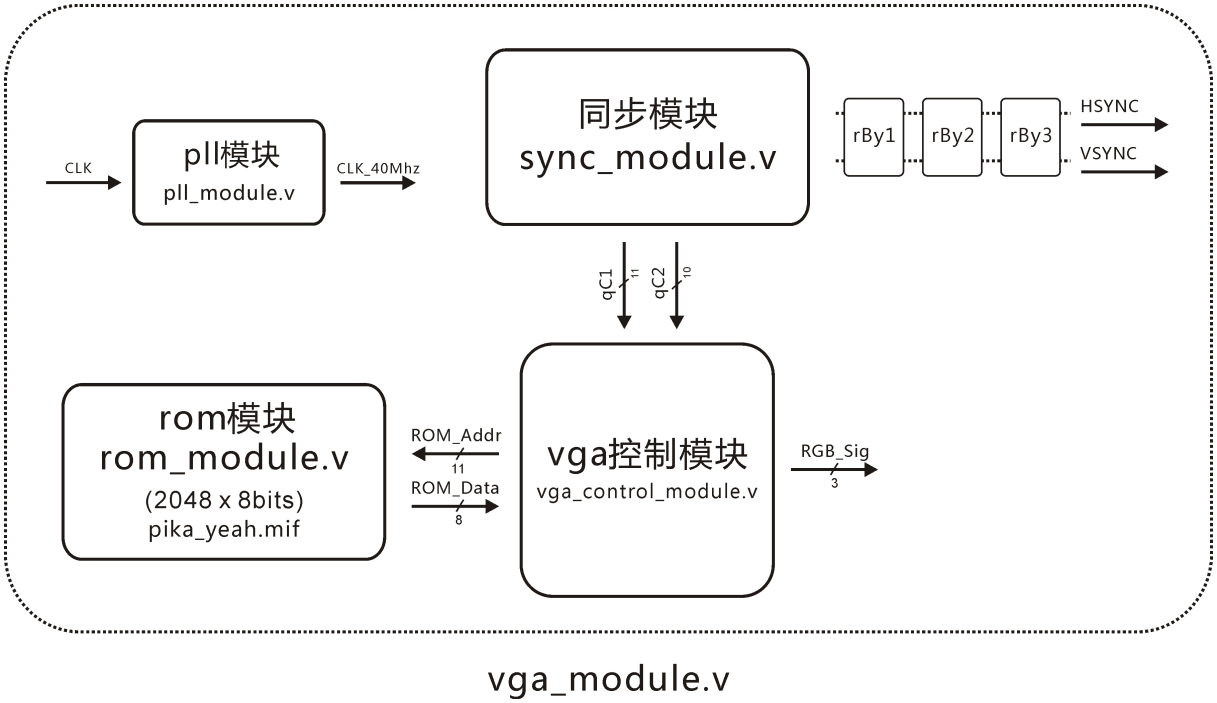


图 2.9s vga 模块的建模图。

图 2.9 是 vga_module.v 组合模块，它包含了 pll 模块，用于倍频 20Mhz 至 40Mhz；同步模块，用于配置显示标准；rom 模块，用于储存图像信息，容量为 2048 * 8bits，初始化文件时 pika_yeah.mif；vga 控制模块，流水操作，此外主要是负责图像处理，分辨率，和显示位置；3 级旁路寄存器，主要是用于同步操作。

```
module vga_module
(
        input CLK,
        input RSTn,
        output HSYNC,
        output VSYNC,
        output [2:0]RGB_Sig
);

```

```
10.         /***************************/
11.
12.           wire CLK_40Mhz;
13.
14.           pll_module     U0
15.           (
16.                .inclk0 ( CLK ),        // in 20MHz
17.                .c0 ( CLK_40Mhz )      // output 40Mhz
18.           );
19.
20.         /***************************/
21.
22.           wire [10:0]qC1;
23.           wire [9:0]qC2;
24.           wire U1_HSYNC, U1_VSYNC;
25.
26.           sync_module U1
27.           (
28.                  .CLK( CLK_40Mhz ),      // input - from U0
29.                  .RSTn( RSTn ),
30.                  .qC1( qC1 ),                // output - to U3
31.                  .qC2( qC2 ),                // output - to U3
32.                  .HSYNC( U1_HSYNC ),    // output - to bypass register
33.                  .VSYNC( U1_VSYNC )      // output - to bypass register
34.           );
35.
36.           /****************************/
37.
38.           wire [7:0]ROM_Data;
39.
40.           rom_module U2
41.           (
42.                .clock( CLK_40Mhz ),    // input - from U0
43.                .address( ROM_Addr ), // input - from U3
44.                .q( ROM_Data )               // output - to U3
45.           );
46.
47.           /***************************/
48.
49.           wire [10:0]ROM_Addr;
50.
51.           vga_control_module U3
52.           (
53.                  .CLK( CLK_40Mhz ),          // input - from U0
```

```
54.                 .RSTn( RSTn ),
55.                 .qC1( qC1 ),                    // input - from U1
56.                 .qC2( qC2 ),                    // input - from U1
57.                 .ROM_Addr( ROM_Addr ), // output - to U2
58.                 .ROM_Data( ROM_Data ), // input - from U2
59.                 .RGB_Sig( RGB_Sig )        // output - to top
60.           );
61.
62.           /***************************/
63.
64.           // sync. sync_module and vga_control_module
65.           // vga_control_module use 3 clock than sync_module
66.           // well, sync_module bypass-ing 3 shift-reg.
67.
68.           reg [1:0]rBy1,rBy2,rBy3;
69.
70.           always @ ( posedge CLK_40Mhz or negedge RSTn )
71.                 if( !RSTn )
72.                       begin
73.                                 rBy1 <= 2'b11;
74.                                 rBy2 <= 2'b11;
75.                                 rBy3 <= 2'b11;
76.                           end
77.                 else
78.                       begin
79.                                 rBy1 <= { U1_HSYNC, U1_VSYNC };
80.                                 rBy2 <= rBy1;
81.                                 rBy3 <= rBy2;
82.                           end
83.
84.          /***************************/
85.
86.          assign { HSYNC, VSYNC } = rBy3;
87.
88.          /***************************/
89.
90.   endmodule
```

除了第 64~82 行以外，读者自己看着办吧。第 64~82 行是 3 级的旁路寄存器（又可以成为 3 级的移位寄存器）。第 68 行中声明的 rBy1~3 都是 2 位位宽，为了就是寄存从同步模块中输出的 HSYNC 和 VSYNC 信号。第 73~75 行，rBy1~3 都复位为逻辑 1，原因很简单，因为 HSYNC 和 VSYNC 信号在空闲时总是拉高电平。第 79~81 行就是旁路|移位操作了。至于 81 行的 rBy3 直接驱动 HSYNC 和 VSYNC 信号（第 86 行）。

在实验十里，重点就是 vga 控制模块的流水操作和同步模块的 3 级旁路寄存器。但是最为重要的还是 C1 和 C2,因为它们都是“HSYNC 和 VSYNC 时钟化的结果”。在这里把某个信号“时钟化”成为一个长长“可计数”和“有标签”的概念真的很重要。因为这是同步数据的一个重点。

最后还有一个重点就是关于“3 位小朋友沟通” 这一概念。对于实验十来说，可能读者还不能充分的了解到其中的含义。这个概念对于设计一个“系统模块”来说是非常重要的，因为当一个设计接近“系统建模”的等级，建模量不仅会增加，最头疼的就是如何“协调全部模块的沟通”... 因此，传统的“全局者观察”的方式虽然依旧有效（就是把全部东西都联系起来然后仿真）但是会累死设计者的。

类似的方法为什么称为“被动”？答案很简单，因为我们只能等到问题浮现以后才能排错。不但而已，有时侯问题的产生是需要 2 个模块以上的沟通才能浮现。所以说，我们要反客为主，从“被动”成为“主动”。这样一来，仿真的频率不仅会降低，而且设计会变得更轻松。但是前提是“读者要舍得传统”，因为接下来的内容会不断扭曲有关 Verilog HDL 语言的传统世界。

好了,笔者也不多话了 ... 把实验下载到黑金开发板以后,读者就会看到一只大小为 128 x 128 的小可爱，会出现在屏幕 x-0 y-0 的地方。读者稍微注意一下，这只小可爱的最外围是否被一个 1pixel 宽的正方形“圈着”？这是为了强调图像信息没有跑帧以外，也暗示着“浅绿色数据”，“深绿色数据”和“红色数据”已经同步输出了。（读者稍微复习一下，第一章中的最后一个小节，自然就会明白笔者在说什么了。）

2.10 主动设计的概念和模块的语言

在 2.9 小节里，笔者说过“3 个小朋友沟通”的故事对否？这个概念基本上是从“一个时钟，一块数据”发展过来的，但是这个概念的启发确实是来至“一群小朋友在对话”，故事是这样的：

在某个傍晚，笔者在来到海边，却为一些私人的问题烦恼着，无论笔者怎么思考，都无法使问题圆滑起来。眼见头皮都要抓破了，心里说道 “算了吧”，笔者起身然后向海边的另一边走去。忽然间笔者的身旁经过一群在游戏的小朋友，一个似“老大”的家伙大声的说道：“我们现在就玩鬼游戏，陪碰到的人就是鬼 ... ”，过后其他的小朋友和身边的人讨论了起来。看着小朋友们玩闹，豁然使笔者忘了烦恼。

读者你们知道吗？人类的“对话”是一种极其落后的工具，“对话”本身就是一维像度。“对话”本身的精度问题先不论，然而“对话”也只是“单工”的，亦即同时间只能一方发送和一方接收。但是一个人“发话”，却同时可以由很多人“接收”，不过在“对话”的过程中一定会有“延迟”。也就说，这一秒发话，下一秒才能接收得到。

读者尝试想象一下，在一个演讲会中，假设读者就是演讲者，那么此时的读者只要“关心”自己的“发话”是“正确无误”。相反的，如果读者换做是听众，那么此时的读者只要“关心”自己的“听话”是“正确无误”。但是在演讲会开始之前，必须确保演讲者所讲的话，听众必须听得懂。

假设演讲者说了“大家早安，很感谢大家出席这次的演讲会 。”作为听众，有的人会把所有内容听完，有的人却只注意到 “大家早安”，有的人更过份，一句话也没有听进耳子里面。作为演讲者，听众想怎样，演讲者是奈何不了的，演讲者只要做好自己的本分，把“大家早安，很感谢大家出席这次的演讲会”说完。反之，作为听众，听众要怎么接收内容都是听众的自由，如果某段演讲内容确实对听众感兴趣，听众只要正确无误接收内容就好了。

这就是“主动设计”的基本概念。虽说这个概念是基于 “一个时钟，一块数据”，但是这个概念也是“模块沟通”的扩展。对于模块来说，模块是没有人类所谓的 “语言”，然而“一个时钟，一块数据”就是把某个信号时钟化以后，变成“有标签”和“可计数”。“有标签”类似于英文字母“a~z”，“可计数”类似于“apple”有 5 个字母。

不过模块的语言不像人类的语言那么“复杂”，因为语言的内容只能是“二进制”。举个简单的例子，假设有个模块 A 说“1”“0”“1”“1”“0”，那么 5'b01101 就是“内容”，如果模块 A 是用“并行”的方式沟通，那么语言只有 1 个时钟的长度。换之，如果模块 A 用“串行”的方式沟通，那么语言就有 5 个时钟的长度。

当某个信号被时钟化以后，下边就会出现 C1 或者 C2 之类的计数器。C1 和 C2 计数器的出现，是为了计数和跟踪某个信号，当然读者也可以说 C1 和 C2 等计数器“标签化”了某个信号。但是不是所有信号都需要时钟化，只有“串行”沟通的信号才需要时钟化，如：串口传输，或者 vga 时序中的 VSYNC 和 HSYNC。那些不用时钟化的信号，通常

都是以“并行”的方式在沟通，如 FIFO，因为并行的沟通方式只要一个时钟即可。对于模块之间的沟通都需要“一条硬道理”，就是“模块之间沟通至少需要一个时钟”。除此之外，所有模块的“沟通口”必须遵守“时间点事件”（2.4 小节的重点）。好了，笔者也不罗嗦了，实战好过理论，笔者还是用实验来说话吧。

实验十一：两个小屏幕

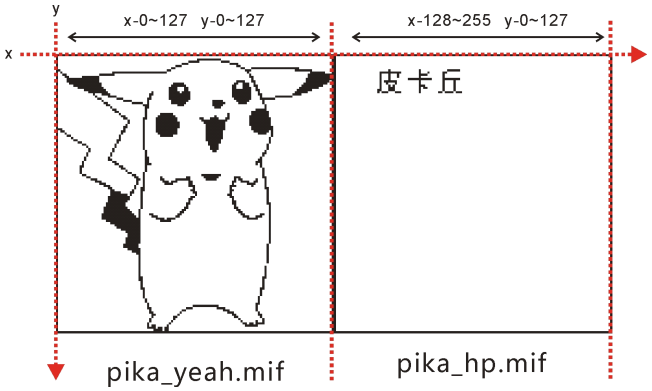


图 2.10a 实验十一要显示的结果。

在实验十里，我们成功了显示一副大小为 128 * 128 的图像在 x-0 y-0 的位置。在实验十一里，我们要在实验十的基础上，再显示另一副大小为 128 * 128 的图像在前一副图像的右边（亦即位置为 x-128 y-0），如图 2.10a 所示。

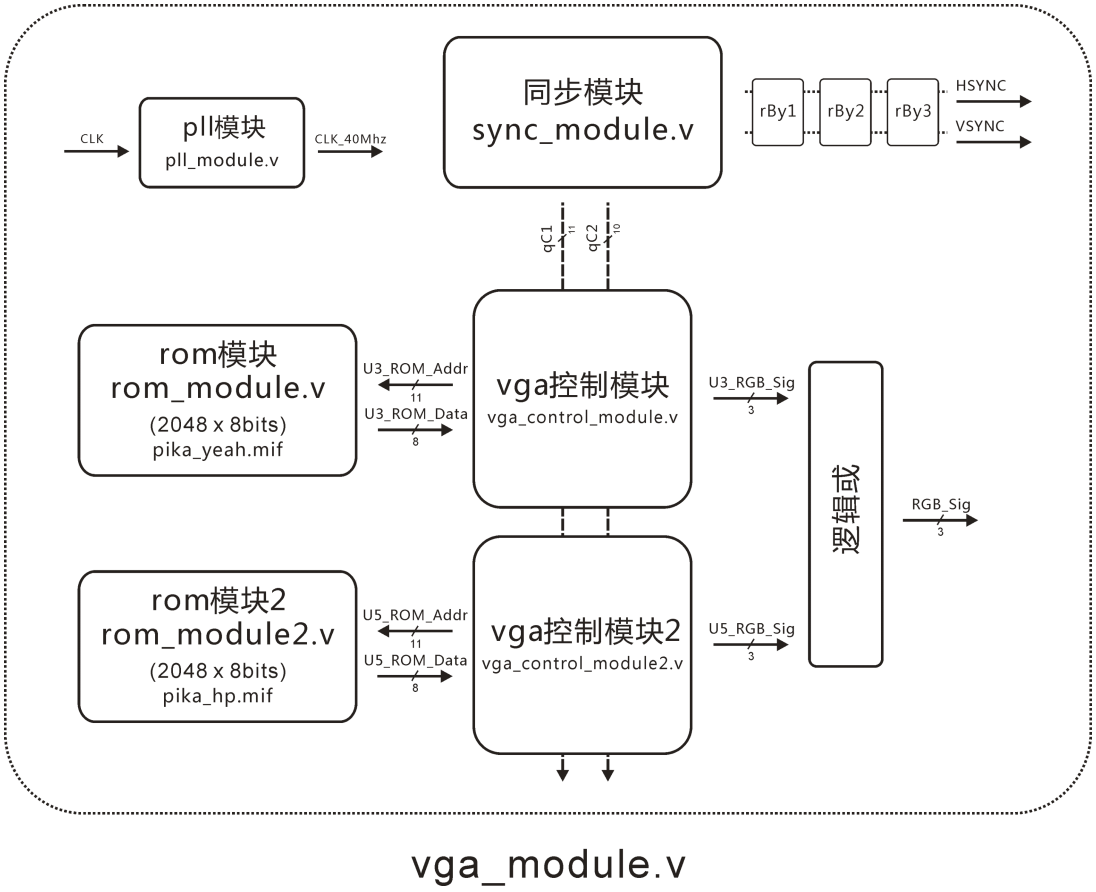


图 2.10b vga 模块的建模图。

图 2.10b 就是实验十一要实现建模。vga 模块是一个组合模块，它包含除了实验十的模块以外，它还包含 rom 模块 2 和 vga 控制模块 2。当读者看到这里，是不是要开始感叹道建模技巧极其重要呢？如果因为没有建模技巧作为基础，基本上就很难衍生其它 Verilog 的使用技巧。即使是本小节所学到的概念，如果没有建模技巧作为后盾根本就不能实现（笔者又再一次老王卖瓜了）。

图 2.10b 中，rom 模块和 vga 控制模块除了连线命名改变以外，其它照旧。换之，rom 模块 2 里边保存了 pika_hp.mif 的图像信息，容量是 2048 words * 8 bits，然而 vga 控制模块 2 控制着 128 * 128 大小的小屏幕，显示位置是 x-218，y-0。此外会使得读者疑惑的就是“逻辑或”了，逻辑或在这里有“叠加图层或者叠加屏幕”的作用，仅此而已。

好了，我们先用传统的角度去看图 2.10b 中，同步模块发送 qC1 和 qC2 给 vga 控制模块和 vga 控制模块 2 的关系。在传统中，从图 2.10b 我们只能看到 3 个呆呆的模块坐在那里，然而 qC1 和 qC2 信号代表着它们之间的关系，亦即同步模块发送信号 qC1 和 qC2 给 vga 控制模块和 vga 控制模块 2。这 3 个家伙基本上就是死鸡，或者呆子，如果没有仿真的话根本看不到它们的反应（沟通记录）。类似的方法，我们成为“被动”式。

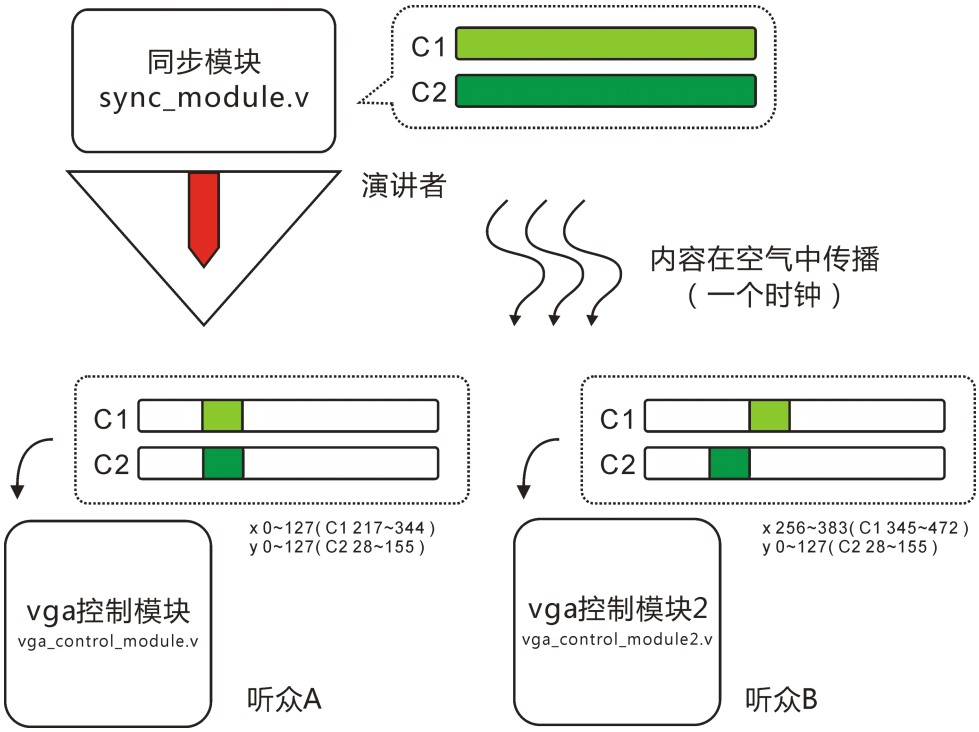


图 2.10c 同步模块和 vga 控制模块们在沟通中。

在这里，我们换一个比较生动的角度去看这 3 个家伙。如图 2.10c 所示，同步模块当起演讲者了，演讲内容就是 C1 和 C2 的全部。换之，vga 控制模块和 vga 控制模块就是听众，当演讲内容通过空气的转播到它们的耳子边，它们接收的“内容”都各有不一。读者是不是觉得生动有趣？感觉上模块就像有生命的 ... 主动设计的重点，就是“设计可以站在模块的立场上”然后“确保各个模块的工作是无误的”。

vga_control_module2.v

```
module vga_control_module2
(
        input CLK,
        input RSTn,
        input [10:0]qC1,
        input [9:0]qC2,
        output [2:0]RGB_Sig,
        input [7:0]ROM_Data,
        output [10:0]ROM_Addr

);

      /*************************************/

        // Height , width, x-offset and y-offset
```

```
16.         parameter _X = 8'd128, _Y = 8'd128, _XOFF = 10'd128, _YOFF = 10'd0;
17.
18.         /************************************/
19.
20.         reg [6:0]x,y;
21.         reg [2:0]n1,n2;
22.         reg [10:0]rAddr;
23.         reg [2:0]rRGB;
24.
25.         always @ ( posedge CLK or negedge RSTn )
26.             if( !RSTn )
27.                 begin
28.                     x <= 7'd0;
29.                     y <= 7'd0;
30.                     n1 <= 3'd0;
31.                     n2 <= 3'd0;
32.                     rAddr <= 11'd0;
33.                     rRGB <= 3'd0;
34.                 end
35.             else
36.                 begin
37.
38.                     // step 1 : compute data address and index-n
39.                     if( (qC1 > 128 + 88 + _XOFF && qC1 <= 128 + 88 + _XOFF +_X) &&
40.                         (qC2 > 4 + 23 + _YOFF && qC2 <= 4 + 23 + _YOFF + _Y) )
41.                         begin
42.                             x = qC1 - _XOFF - 128 - 88 - 1;
43.                             y = qC2 - _YOFF - 4 - 23 - 1;
44.                             rAddr <= (y << 4) + (x >> 3);
45.                             n1 <= ( x - ( (x >> 3) << 3) ); // x & 0x07
46.                         end
47.                     else
48.                         begin n1 <= 3'd0; rAddr <= 11'd17; end
49.
50.                     // step 2 : reading data from rom
51.                     n2 <= n1;
52.
53.                     // step 3 : assign RGB_Sig
54.                     rRGB <= { ROM_Data[n2], ROM_Data[n2], ROM_Data[n2] };
55.
56.                 end
57.
58.         /****************************/
59.
```

```
60.         assign RGB_Sig = rRGB;
61.         assign ROM_Addr = rAddr;
62.
63.         /****************************/
64.
65.   endmodule
```

vga_control_module2.v 和 vga_control_module.v 的内容大同小异，改变的只是第 16 行的 XOFF 常量声明而已，从 0 修改为 128。因此，在第 39~40 行的计算也改变了。第 39 行的表达式是接收 C1 的某个长度，当前的 _X 和 _XOFF 也是 128，那么结果会是：

```
qC1 > 128 + 88 + _XOFF && qC1 <= 128 + 88 + _XOFF +_X
qC1 > 128 + 88 + 128 && qC1 <= 128 + 88 + 128 + 128
qC1 > 344 && qC1 <= 472
```

亦即接收 C1 的长度是 345~472。至于 42 行的表达如何计算 x 都是从 0 开始计算，不信吗？

```
x = qC1 - _XOFF - 128 - 88 - 1;
  = 345 - 128 - 128 + 88 - 1
  = 345 - 344 - 1
  = 0。
```

vga_module.v

```
1.    module vga_module
2.    (
3.          input CLK,
4.            input RSTn,
5.            output HSYNC,
6.            output VSYNC,
7.            output [2:0]RGB_Sig
8.    );
9.
10.         /****************************/
11.
12.           wire CLK_40Mhz;
13.
14.           pll_module     U0
15.           (
16.                .inclk0 ( CLK ),          // in 20MHz
17.                .c0 ( CLK_40Mhz )      // output 40Mhz
18.           );
```

```
19.
20.         /***************************/
21.
22.           wire [10:0]qC1;
23.           wire [9:0]qC2;
24.           wire U1_HSYNC, U1_VSYNC;
25.
26.           sync_module U1
27.           (
28.                 .CLK( CLK_40Mhz ),      // input - from U0
29.                   .RSTn( RSTn ),
30.                   .qC1( qC1 ),                    // output - to U3
31.                   .qC2( qC2 ),                    // output - to U3
32.                   .HSYNC( U1_HSYNC ),    // output - to bypass register
33.                   .VSYNC( U1_VSYNC )      // output - to bypass register
34.           );
35.
36.           /***************************/
37.
38.           wire [7:0]U3_ROM_Data;
39.
40.           rom_module U2
41.           (
42.                .clock( CLK_40Mhz ),           // input - from U0
43.                .address( U3_ROM_Addr ),    // input - from U3
44.                .q( U3_ROM_Data )               // output - to U3
45.           );
46.
47.           /***************************/
48.
49.           wire [10:0]U3_ROM_Addr;
50.           wire [2:0]U3_RGB_Sig;
51.
52.           vga_control_module U3
53.           (
54.                 .CLK( CLK_40Mhz ),         // input - from U0
55.                   .RSTn( RSTn ),
56.                   .qC1( qC1 ),                    // input - from U1
57.                   .qC2( qC2 ),                    // input - from U1
58.                   .ROM_Addr( U3_ROM_Addr ),    // output - to U2
59.                   .ROM_Data( U3_ROM_Data ),      // input - from U2
60.                   .RGB_Sig( U3_RGB_Sig )           // output - to top
61.           );
62.
```

```
63.         /**************************/
64.
65.         wire [7:0]U5_ROM_Data;
66.
67.         rom_module2 U4
68.         (
69.             .clock( CLK_40Mhz ),          // input - from U0
70.             .address( U5_ROM_Addr ),    // input - from U3
71.             .q( U5_ROM_Data )              // output - to U3
72.         );
73.
74.         /**************************/
75.
76.         wire [10:0]U5_ROM_Addr;
77.         wire [2:0]U5_RGB_Sig;
78.
79.         vga_control_module2 U5
80.         (
81.             .CLK( CLK_40Mhz ),                    // input - from U0
82.             .RSTn( RSTn ),
83.             .qC1( qC1 ),                          // input - from U1
84.             .qC2( qC2 ),                          // input - from U1
85.             .ROM_Addr( U5_ROM_Addr ),       // output - to U2
86.             .ROM_Data( U5_ROM_Data ),       // input - from U2
87.             .RGB_Sig( U5_RGB_Sig )              // output - to top
88.         );
89.
90.
91.         /**************************/
92.
93.         // sync. sync_module and vga_control_module
94.         // vga_control_module use 3 clock than sync_module
95.         // well, sync_module bypass-ing 3 shift-reg.
96.
97.         reg [1:0]rBy1,rBy2,rBy3;
98.
99.         always @ ( posedge CLK_40Mhz or negedge RSTn )
100.            if( !RSTn )
101.                begin
102.                        rBy1 <= 2'b11;
103.                        rBy2 <= 2'b11;
104.                        rBy3 <= 2'b11;
105.                    end
106.            else
```

```
                    begin
                            rBy1 <= { U1_HSYNC, U1_VSYNC };
                            rBy2 <= rBy1;
                            rBy3 <= rBy2;
                        end

        /***************************/

        assign { HSYNC, VSYNC } = rBy3;
        assign RGB_Sig = U3_RGB_Sig | U5_RGB_Sig;

        /***************************/

endmodule
```

内容基本上都很直接，自己只要跟着建模图看就会明白了。

==========

实验十一基本上在很多地方已经开始“打破传统”，笔者真的很怕“保守主义者”的围攻 ... 呵呵呵。主动设计是一种比较奇怪的方法，一开始的时候读者们可能会不习惯它。因为在传统的设计中，我们都非常依赖仿真，然而主动设计强调的是“模块都用同样的语言说话”和“模块只要自顾自的完成操作”。

读者尝试回想看看，当你和朋友在说话的时候，自身是不是“只在乎”说完某段话和听完某段话（当然你的朋友也是），最重要的就是读者要用和朋友之前都听得懂的语言，而不需要第三者去监督读者和朋友之间的对话。同样的道理，在主动设计中，我们只要确保模块自身“发话或者听话良好”，用不着把所用东西都联系起来然后仿真。但是前提就是模块之间都用“模块话”在沟通，这个“模块话”就是“一个时钟，一块数据”。当然，笔者的意思不是不使用仿真，而是减少仿真的频率，因为仿真真的很耗时间和精力。好了，接下来的实验笔者会继续用同样的概念来表达实验的内容。

2.11 实验十二：皮卡丘的 HP 血条

在实验十一中，笔者在 x-128，y-0 的位置显示了一张非常单调的图像，既是在一个 128 x 128 正方形中，左上方写着“比卡丘”3 个字。接下来，我们要在实验十一的基础上，然后为图像 pika_hp.mif ，在“比卡丘”3 个字的下方显示一个“动态的 HP 血条”。

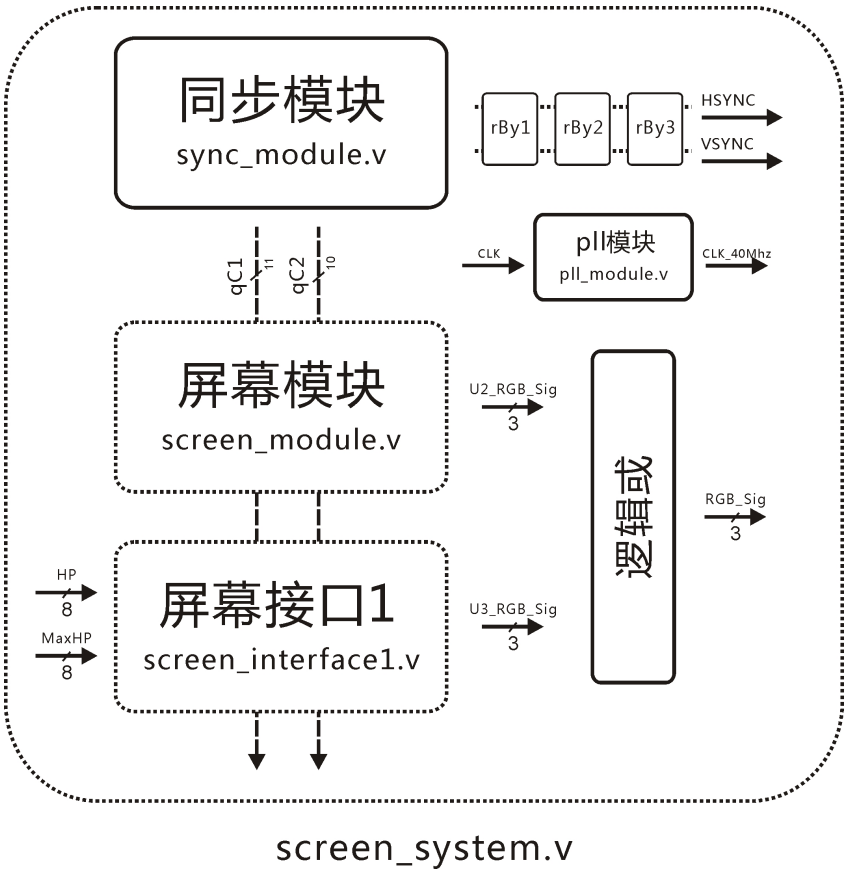


图 2.11a 屏幕系统的建模图。

图 2.11a 是该实验要建立的系统模块—screen_system.v。它从实验十一的基础上，一口进化为系统级的模块了，原因就是里边的屏幕接口 1 — screen_interface1.v。屏幕模块基本上是由之前实验中 rom 模块 + vga 控制模块组合而成的，如图 2.11b 所示。至于比较陌生的就是屏幕接口这个接口级的建模了。它主要是由实验十一中 rom 模块 2 + vga 控制模块 2 扩展而成，如图 2.11c 所示。

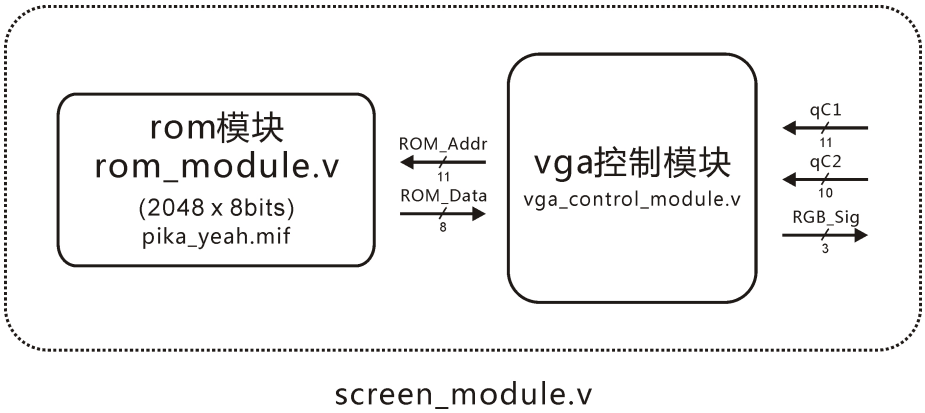


图 2.11b 屏幕模块的建模图。

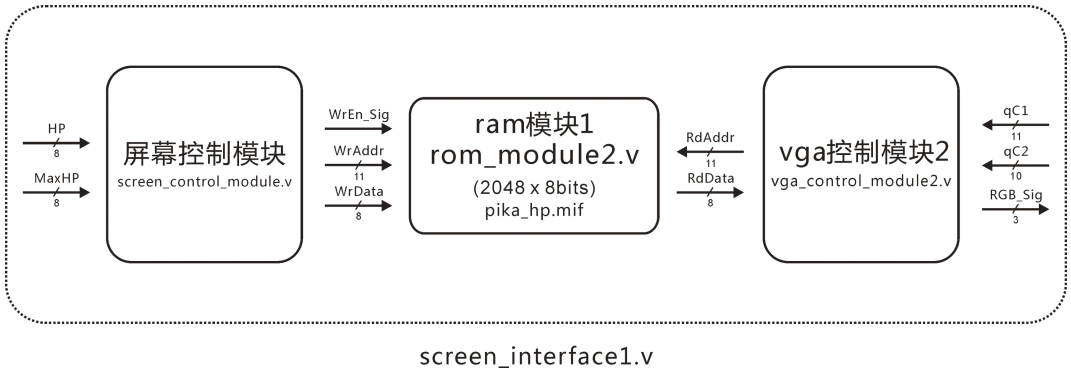


图 2.11c 屏幕接口 1 的建模图。

图 2.11c 就是屏幕接口的组合模块，它包含了 vga 控制模块 2，ram 模块 1 和屏幕控制模块。vga 控制模块 2，除了负责图像信息处理以外，还有控制 128 * 128 的分辨率和显示位置 x-128 y-0；ram 模块 1 可以看成是图像信息的储存室或者是图像信息的缓冲区；屏幕控制模块主要是计算和显示化 HP 血条。屏幕控制模块看起来很难入口，其实在设计中真的简单到不行。

ram_module1.v

```
module ram_module1
(
        input CLK,
        input RSTn,

        input WrEn_Sig,
        input [10:0]WrAddr,
        input [7:0]WrData,

        input [10:0]RdAddr,
        output [7:0]RdData
);
```

```
13.
14.         /****************************/
15.
16.           (* ramstyle = "no_rw_check , m4k" , ram_init_file = "pika_hp.mif" *) reg [7:0]RAM[2047:0];
17.
18.           reg [7:0]rData;
19.
20.           always @ ( posedge CLK or negedge RSTn )
21.                 if( !RSTn )
22.                       rData <= 8'd0;
23.                   else if( WrEn_Sig )
24.                       RAM[ WrAddr ] <= WrData;
25.                   else
26.                       rData <= RAM[ RdAddr ];
27.
28.         /*******************************/
29.
30.         assign RdData = rData;
31.
32.         /********************************/
33.
34.   endmodule
```

ram_module1 的内容基本上简单到不行了，第 3~11 行是输入输出的声明。第 16 行是储存器的声明（Verilog 不支持二维数组），大小为 2048 * 8bits。其中初始文件是 pika_hp.mif。第 20~26 行是 ram 的操作，第 23 行的 if 条件表示写操作的优先级高于读操作。

screen_control_module.v

```
1.    module screen_control_module
2.    (
3.            input CLK,
4.            input RSTn,
5.            input [7:0]HP, MaxHP,
6.
7.            output WrEn_Sig,
8.            output [10:0]WrAddr,
9.            output [7:0]WrData
10.   );
11.
12.         /****************************/
13.
```

```
14.         reg [7:0]F1,F2;
15.         reg [7:0]F3,F4;
16.
17.         always @ ( posedge CLK or negedge RSTn )
18.             if( !RSTn )
19.                 begin
20.                         F1 <= 8'd0;
21.                         F2 <= 8'd0;
22.                         F3 <= 8'd0;
23.                         F4 <= 8'd0;
24.                     end
25.             else
26.                 begin
27.                         F1 <= HP;
28.                         F2 <= F1;
29.                         F3 <= MaxHP;
30.                         F4 <= F3;
31.                     end
32.
33.       /*********************************/
34.
35.         parameter XOFF = 8'd16, YOFF = 8'd30;
36.
37.         /*********************************/
38.
39.         reg [4:0]i;
40.         reg [4:0]Go;
41.         reg [10:0]rAddr;
42.         reg [7:0]rData;
43.         reg [7:0]T1,T2,T3;
44.         reg [7:0]Result;
45.         reg [15:0]Temp;
46.         reg isWrite;
47.
48.
49.         always @ ( posedge CLK or negedge RSTn )
50.             if( !RSTn )
51.                 begin
52.                         i <= 5'd0;
53.                         Go <= 5'd0;
54.                         rAddr <= 11'd0;
55.                         rData <= 8'd0;
56.                         T1 <= 8'd0;
57.                         T2 <= 8'd0;
```

```
58.                             T3 <= 8'd0;
59.                             Result <= 8'd0;
60.                             Temp <= 16'd0;
61.                             isWrite <= 1'b0;
62.                 end
63.             else
64.                     case( i )
65.
66.                             0: // if HP or MaxHP change ...
67.                             if( F1 != F2 || F3 != F4 ) begin T1 <= HP; T2 <= MaxHP; T3 <= 8'd96; i <= i + 1'b1; end
68.
69.                             /****************************/ // Compute ( HP/MaxHP * 96% )
70.
71.                             1:    // HP * 96
72.                             if( T3 == 0 ) i <= i + 1'b1;
73.                             else begin Temp <= Temp + T1; T3 <= T3 - 1'b1; end
74.
75.                             2:    // HP * 96 / MaxHP
76.                             if( Temp < T2    ) i <= i + 1'b1;
77.                             else begin Temp <= Temp - T2; Result <= Result + 1'b1; end
78.
79.                             /*************************************************/
80.
81.                             3,4,5,6,7,8,9,10: // fill 8 time hp-row
82.                             begin T1 <= Result; T2 <= YOFF + (3-i); i <= 5'd12; Go <= i + 1'b1; end
83.
84.                             11:
85.                             begin i <= 5'd0; end
86.
87.                             /*******************************/ // Draw hp function
88.
89.                             12,13,14,15,16,17,18,19,20,21,22,23:
90.                             begin
91.                                     if( T1 >= 8 ) begin rData <= 8'hff; T1 <= T1 - 8; end
92.                                     else if( T1 == 1 ) begin rData <= 8'b00000001; T1 <= 8'd0; end
93.                                     else if( T1 == 2 ) begin rData <= 8'b00000011; T1 <= 8'd0; end
94.                                     else if( T1 == 3 ) begin rData <= 8'b00000111; T1 <= 8'd0; end
95.                                     else if( T1 == 4 ) begin rData <= 8'b00001111; T1 <= 8'd0; end
96.                                     else if( T1 == 5 ) begin rData <= 8'b00011111; T1 <= 8'd0; end
97.                                     else if( T1 == 6 ) begin rData <= 8'b00111111; T1 <= 8'd0; end
98.                                     else if( T1 == 7 ) begin rData <= 8'b01111111; T1 <= 8'd0; end
99.                                     else rData <= 8'd0;
100.
101.                                    isWrite <= 1'b1;
```

```
102.                                rAddr <= (T2<<4) + (XOFF>>3) + (i-12);
103.                                i <= i + 1'b1;
104.                            end
105.
106.                            24:
107.                            begin isWrite <= 1'b0; i <= Go; end
108.
109.                            /*****************************************************/
110.
111.                        endcase
112.
113.        /***************************************/
114.
115.        assign WrEn_Sig = isWrite;
116.        assign WrAddr = rAddr;
117.        assign WrData = rData;
118.
119.        /***************************************/
120.
121. endmodule
```

哇！读者千万不要被吓到，它虽然内容庞大，但是只要搞懂了操作步骤，再庞大的东西都是纸老虎。第 3~9 行是输入输出口的声明。第 14~31 行是用来采样 HP 和 MaxHP 信号，为了就是检测 HP 和 MaxHP 信号的改变。第 35 行是 HP 血条显示的位置的常量声明，其中 XOFF 是 16，YOFF 是 30。

第 39~46 行是相关的寄存器声明，第 39 行的 i 是步骤用途；第 40 行的 Go 是在伪函数中，返回步骤的用途；第 41 行的 rAddr 是用来驱动 WrAddr 信号；第 42 行的 rData 是用来驱动 WrData 信号。第 42 行声明的 T1~T3 是操作中所使用的暂存空间 Temp1~3；第 44 行的 Result 是用来寄存计算结果；第 45 行是另一个 Temp 空间，比起第 42 行的 T1~T3，位宽大一倍；第 46 行的 isWrite 是用来驱动 WrEn_Sig。

第 64~111 行是该控制模块的核心操作。一开始的时候清零所有寄存器（51~62 行），步骤 0（第 66~67 行）的 if 条件是用来判断 HP 和 MaxHP 信号是否发生变化？如果是的话就把 HP 寄存在 T1，MaxHP 寄存于 T2，然后 T3 赋值与 96，然后 i 递增。第 71~73 行是用来实现 HP * 96 的传统乘法器，第 75~77 行的步骤 2 是用来实现（HP * 96）/ MaxHP 的传统触发器（至于传统乘法器除法器的操作原理请复习时序篇）。

步骤 1 和步骤 2 的操作归根究底就是求出 (HP / MaxHP) 的 96 分比。可能读者要问，为什么不是百分比而是 96 分比呢？很简单呀，因为图像信息都是以字节为基础，然而 96 可以被 8 整除。 举个例子，假设 HP 为 122，MaxHP 为 255，那么它们之间的 96 分比是：

```
(122 / 255 ) * 96 = 45
```

第 81~82 行的步骤 3~10 是用来划出 8 行的 HP 血条，然而步骤 3~10 会调用第 89~107 的伪函数。该伪函数是在 ram 模块中绘出 HP 血条，具体的操作过程是如何发生呢，就让笔者一步步来解释吧。

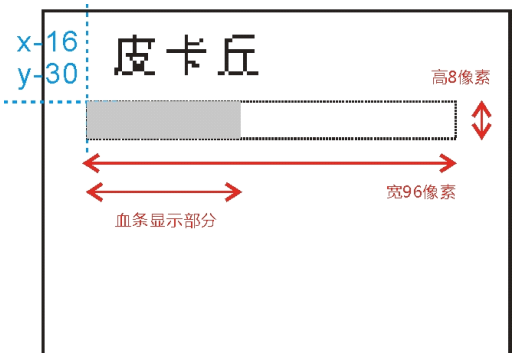


图 2.11d 血条的绘制信息。

第 35 行的常量定义就是要把 HP 血条显示在 Y-30，X-16 的地方，然而步骤 3~10 就是要绘制高位 8 个像素的血条，至于步骤 89~107 行的伪函数就要显示宽为 96 个像素和血条的显示部分。在图 2.11c 的建模图中，屏幕接口 1 的建模是按照低级建模，换句话说 ram 模块 1 充当了图像信息的缓冲空间。但是这个缓冲空间是按字节作为基础，所以说“我们如何把想象中的 HP 血条绘制出，符合缓冲空间的图像信息”。

```
81.     3,4,5,6,7,8,9,10: // fill 8 time hp-row
82.     begin T1 <= Result; T2 <= YOFF + (3-i); i <= 5'd12; Go <= i + 1'b1; end
83.
84.     11:
85.     begin i <= 5'd0; end
86.
87.     /********************************/ // Draw hp function
88.
89.     12,13,14,15,16,17,18,19,20,21,22,23:
90.     begin
91.         if( T1 >= 8 ) begin rData <= 8'hff; T1 <= T1 - 8; end
92.         else if( T1 == 1 ) begin rData <= 8'b00000001; T1 <= 8'd0; end
93.         else if( T1 == 2 ) begin rData <= 8'b00000011; T1 <= 8'd0; end
94.         else if( T1 == 3 ) begin rData <= 8'b00000111; T1 <= 8'd0; end
95.         else if( T1 == 4 ) begin rData <= 8'b00001111; T1 <= 8'd0; end
96.         else if( T1 == 5 ) begin rData <= 8'b00011111; T1 <= 8'd0; end
97.         else if( T1 == 6 ) begin rData <= 8'b00111111; T1 <= 8'd0; end
98.         else if( T1 == 7 ) begin rData <= 8'b01111111; T1 <= 8'd0; end
99.         else rData <= 8'd0;
100.
101.        isWrite <= 1'b1;
102.        rAddr <= (T2<<4) + (XOFF>>3) + (i-12);
```

```
103.                    i <= i + 1'b1;
104.                end
105.
106.                24:
107.                begin isWrite <= 1'b0; i <= Go; end
```

在 82 行中，T1 赋予 Result，T2 赋予 YOFF 亦即血条的 Y 显示位置，i 赋予伪函数的入口步骤，Go 赋予返回步骤。我们先假设 HP 为 122，MaxHP 为 255，Result（T1）为 45，当 i 进入伪函数以后，第 91 行的 if 条件成立（T1 大于等于 8），然么 rData 赋予 8'hff，然后 T1 递减 8。 在同一个时间，第 101 行的 isWrite 会被拉高，此时第 102 行的 rAddr 也计算出缓冲空间中所对应的地址，i 也会递增。

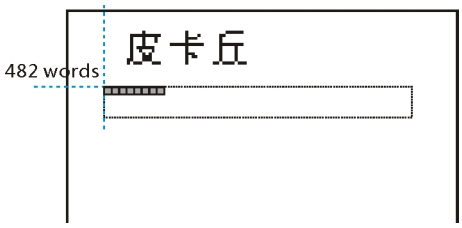


图 2.11e ram 模块 1 中对应的字地址。

估计读者的头要大了，然后问道："rAddr 中的表达式到底是有什么规律的？"第 102 中的 T2 基本上是寄存 YOFF，亦即显示位置 y 的值。此外 XOFF 也表示显示位置 x 的值。我们来简单的计算一下：

```
rAddr    = YOFF << 4 + XOFF >> 3
         = 30 * 16 + 16 / 8
         = 480 + 2
         = 482
```

结果 0xff 会写入 ram 模块 1 中第 482 个字地址（如图 2.11e 所示）。

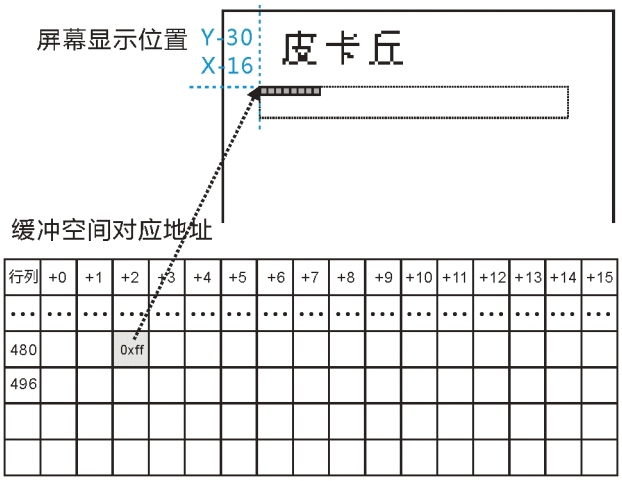


图 2.11f ram 模块里图像信息映射到屏幕。

在这里 x 和 y 是屏幕的“位置标签”，然而屏幕所显示的图像信息是储存在图像缓冲空间里（ram 模块），图像缓冲空间里的图像信息却是以字节为基础并且按字的次序排列着。在图像处理中有一个很老很臭的关键词就是“图像映射”，说简单点就是把屏幕的 x 和 y 位置标签，按图像信息的“基础”计算出在图像缓冲空间里所对应的数据。

我们的屏幕分辨率是 128 * 128，然而“基础是字节”。128 宽（列）等价于 16 words * 8 bits，一副 128 * 128 的图像等价于 128 * 16 words * 8 bits，所以行的偏移量是 16 words * 8 bits，如果无视 8bits 的话就是 16 words。第 35 行定义中的 YOFF 和 XOFF 均为 30 和 16，YOFF 表示行所以 30 * 16 words，XOFF 表示列（不要忘记了“宽”是以“字节”为基础）所以 16/ 8, 总和结果就是 482。最后 8'hff 会写入 ram 模块里的第 482 words, 然后经过“图像映射”，结果会如图 2.11f 所示。

我们继续把焦点放在 102 行，其中 i-12 就是把结果“写入相续的列中”。在此第 90~104 行会继续循环操作 11 次，亦即会把结果写入 ram 模块的 483~493 字中，至于结果是什么要依据 T1 了：

第 1 次循环 T1 大于等于 8, 8'hff 被写入 482 字地址，T1 等于 45 - 8 =37；(步骤 12)
第 2 次循环 T1 大于等于 8, 8'hff 被写入 483 字地址，T1 等于 37 - 8 = 29；(步骤 13)
第 3 次循环 T1 大于等于 8, 8'hff 被写入 484 字地址，T1 等于 29 - 8 = 21；(步骤 14)
第 4 次循环 T1 大于等于 8, 8'hff 被写入 485 字地址，T1 等于 21 - 8 = 13；(步骤 15)
第 5 次循环 T1 大于等于 8, 8'hff 被写入 486 字地址，T1 等于 13 - 8 = 5；(步骤 16)
第 6 次循环 T1 等于 5, 8'b00011111 被写入 487 字地址，T1 等于 0；(步骤 17)
第 7 次循环 T1 等于 0, 8'h00 被写入 488 字地址，T1 依然等于 0；(步骤 18)
第 8 次循环 T1 等于 0, 8'h00 被写入 489 字地址，T1 依然等于 0；(步骤 19)
第 9 次循环 T1 等于 0, 8'h00 被写入 490 字地址，T1 依然等于 0；(步骤 20)
第 10 次循环 T1 等于 0, 8'h00 被写入 491 字地址，T1 依然等于 0；(步骤 21)
第 11 次循环 T1 等于 0, 8'h00 被写入 492 字地址，T1 依然等于 0；(步骤 22)
第 12 次循环 T1 等于 0, 8'h00 被写入 493 字地址，T1 依然等于 0；(步骤 23)

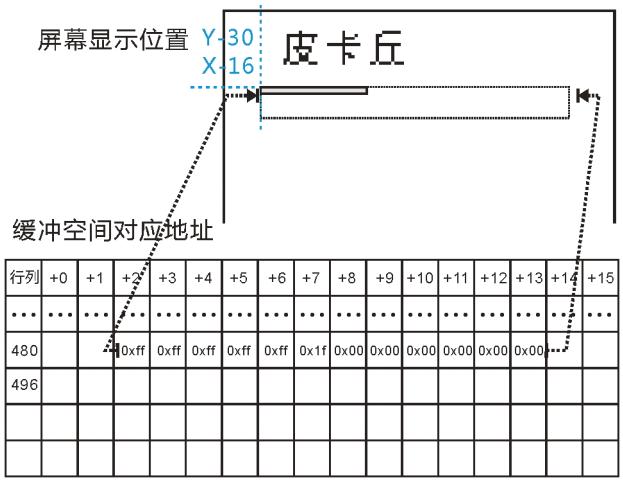


图 2.11g 一行的 HP 血条绘制完成。

就这样循环 12 次，一行的 HP 血条就完成了，如图 2.11g 所示。

```
81.        3,4,5,6,7,8,9,10: // fill 8 time hp-row
82.        begin T1 <= Result; T2 <= YOFF + (3-i); i <= 5'd12; Go <= i + 1'b1; end
```

第 81~82 行的操作循环 8 次（步骤 3~10），就是重复上述的操作 8 次，以至于绘制出一个高为 8 像素的血条。第 82 行中的 YOFF + (3-i) ，其中的 3-i 就是在 YOFF 的基础上更新“行”地址，因此：

第 1 行的 HP 血条是在屏幕的 y-30，对应 ram 模块的 480 字。（步骤 3）
第 2 行的 HP 血条是在屏幕的 y-31，对应 ram 模块的 496 字。（步骤 4）
第 3 行的 HP 血条是在屏幕的 y-32，对应 ram 模块的 512 字。（步骤 5）
第 4 行的 HP 血条是在屏幕的 y-33，对应 ram 模块的 528 字。（步骤 6）
第 5 行的 HP 血条是在屏幕的 y-34，对应 ram 模块的 544 字。（步骤 7）
第 6 行的 HP 血条是在屏幕的 y-35，对应 ram 模块的 560 字。（步骤 8）
第 7 行的 HP 血条是在屏幕的 y-36，对应 ram 模块的 576 字。（步骤 9）
第 8 行的 HP 血条是在屏幕的 y-37，对应 ram 模块的 592 字。（步骤 10）

在 81~82 行经过 8 次的循环调用伪函数以后，就会绘出一个 8 个像素高的 HP 血条了，结果如图 2.11h 所示。

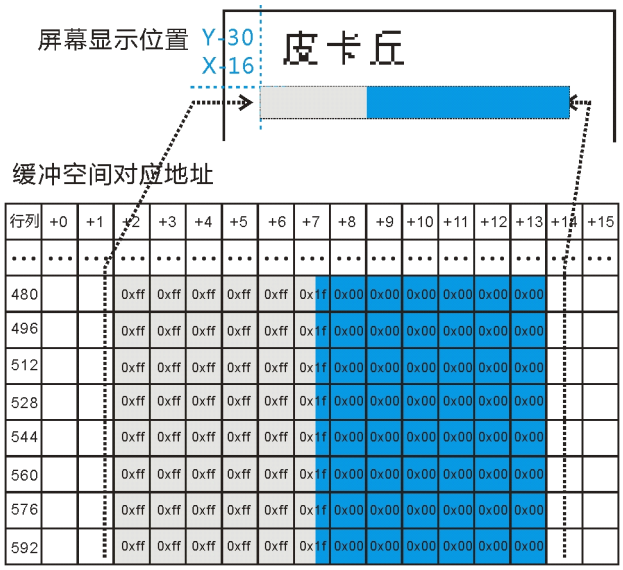


图 2.11h 一个 8 个像素高的 HP 血条已经完成。

==========

呼~终于把 HP 血条绘制的过程写完了，接下来我们要关心一下数据的同步性了。每当我们调用一次伪函数，我们就会往 ram 模块里写入 12 个时钟的数据，以 T2 为 30 (原始 YOFF)，XOFF 为 16，T1 为 45 为例。

```
89.        12,13,14,15,16,17,18,19,20,21,22,23:
90.        begin
```

```
91.                     if( T1 >= 8 ) begin rData <= 8'hff; T1 <= T1 - 8; end
92.                     else if( T1 == 1 ) begin rData <= 8'b00000001; T1 <= 8'd0; end
93.                     else if( T1 == 2 ) begin rData <= 8'b00000011; T1 <= 8'd0; end
94.                     else if( T1 == 3 ) begin rData <= 8'b00000111; T1 <= 8'd0; end
95.                     else if( T1 == 4 ) begin rData <= 8'b00001111; T1 <= 8'd0; end
96.                     else if( T1 == 5 ) begin rData <= 8'b00011111; T1 <= 8'd0; end
97.                     else if( T1 == 6 ) begin rData <= 8'b00111111; T1 <= 8'd0; end
98.                     else if( T1 == 7 ) begin rData <= 8'b01111111; T1 <= 8'd0; end
99.                     else rData <= 8'd0;
100.
101.                 isWrite <= 1'b1;
102.                 rAddr <= (T2<<4) + (XOFF>>3) + (i-12);
103.                 i <= i + 1'b1;
104.             end
105.
106.             24:
107.             begin isWrite <= 1'b0; i <= Go; end
```

步骤 12~23 是利用 12 个时钟向 ram 模块写入数据，步骤 24（第 106~107 行）是拉低 isWrite 和返回的操作。

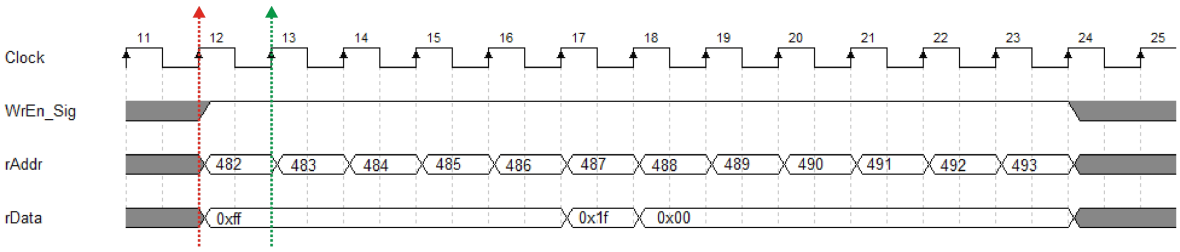


图 2.11i 步骤 12~24 的操作过程。

图 2.11i 笔者做了一些“优惠”的手脚了，读者只要把 i 直接看成“指向第几个时钟”。在 T12 的时候，该模块“决定”拉高 isWrite，输出 rAddr 为 482，rData 为 0xff。所以在 T12 的未来，isWrite 被拉高，rAddr 和 rData 的未来值为 482 和 0xff。在 T13 的时候，ram 模块检测到 isWrite 的过去值为逻辑 1 它“决定”依据 rAddr 的过去值，储存 rData 的过去值。

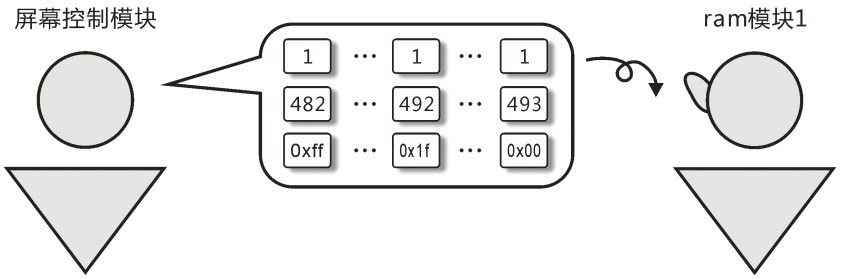


图 2.11j 屏幕控制模块和 ram 模块 1 在沟通。

以上是用“时序”来说话，用“时序”来说话基本上是一件很“烦人”的事情。假设我们把 2.11i 的时序图（时钟化）切成一个时钟一个时钟的话，那么“一个时钟，一块数据”就成为“模块的语言”了。如图 2.11j 所示，屏幕控制模块和 ram 模块 1 各自都遵守“模块沟通的硬道理”，然而屏幕控制模块只要确保自己“发话无误”，然后 ram 模块只要确保自己“听话无误”即可。一来我们可以省去画时序图。二来设计不止那么被动，三来模块也生动起来。

screen_interface1.v

```
module screen_interface1
(
        input CLK,
        input RSTn,

        input [10:0]qC1,
        input [9:0]qC2,

        input [7:0]HP,MaxHP,
        output [2:0]RGB_Sig
);

      /******************************/

        wire WrEn_Sig;
        wire [10:0]WrAddr;
        wire [7:0]WrData;

        screen_control_module U1
        (
                .CLK( CLK ),
                .RSTn( RSTn ),
                .HP( HP ),                      // input - from top
                .MaxHP( MaxHP ),                // input - from top
                .WrEn_Sig( WrEn_Sig ),      // output - to U2
                .WrAddr( WrAddr ),              // output - to U2
                .WrData( WrData )               // output - to U2
        );

        /*********************************/

        wire [7:0]RdData;

        ram_module1 U2
```

```
35.         (
36.                 .CLK( CLK ),
37.                 .RSTn( RSTn ),
38.                 .WrEn_Sig( WrEn_Sig ),      // input - from U1
39.                 .WrAddr( WrAddr ),          // input - from U1
40.                 .WrData( WrData ),          // input - from U1
41.                 .RdAddr( RdAddr ),          // input - from U3
42.                 .RdData( RdData )           // output - to U3
43.         );
44.
45.         /*********************************/
46.
47.         wire [10:0]RdAddr;
48.
49.         vga_control_module2 U3
50.         (
51.                 .CLK( CLK ),
52.                 .RSTn( RSTn ),
53.                 .qC1( qC1 ),                // input - from top
54.                 .qC2( qC2 ),                // input - from top
55.                 .RAM_Addr( RdAddr ),        // output - to U2
56.                 .RAM_Data( RdData ),        // input - from U2
57.                 .RGB_Sig( RGB_Sig )         // output - to top
58.         );
59.
60.         /*********************************/
61.
62.  endmodule
```

screen_system.v

```
1.   module screen_system
2.   (
3.           input CLK,
4.           input RSTn,
5.
6.           output HSYNC,
7.           output VSYNC,
8.           output [2:0]RGB_Sig
9.   );
10.        /***************************/
11.
12.          wire CLK_40Mhz;
```

```
13.
14.         pll_module     U0
15.         (
16.             .inclk0 ( CLK ),        // in 20MHz
17.             .c0 ( CLK_40Mhz )      // output 40Mhz
18.         );
19.
20.         /****************************/
21.
22.         wire [10:0]qC1;
23.         wire [9:0]qC2;
24.         wire U1_HSYNC, U1_VSYNC;
25.
26.         sync_module U1
27.         (
28.                 .CLK( CLK_40Mhz ),      // input - from U0
29.                 .RSTn( RSTn ),
30.                 .qC1( qC1 ),                // output - to U3
31.                 .qC2( qC2 ),                // output - to U3
32.                 .HSYNC( U1_HSYNC ),    // output - to bypass register
33.                 .VSYNC( U1_VSYNC )      // output - to bypass register
34.         );
35.
36.         /*******************************/
37.
38.         wire [2:0]U2_RGB_Sig;
39.
40.         screen_module U2
41.         (
42.                 .CLK( CLK_40Mhz ),           // input - from U0
43.                 .RSTn( RSTn ),
44.                 .qC1( qC1 ),                        // input - from U1
45.                 .qC2( qC2 ),                        // input - from U1
46.                 .RGB_Sig( U2_RGB_Sig )    // output - to top
47.         );
48.
49.         /*********************************/
50.
51.         wire [2:0]U3_RGB_Sig;
52.
53.         screen_interface1 U3
54.         (
55.                 .CLK( CLK_40Mhz ),      // input - from U0
56.                 .RSTn( RSTn ),
```

```
57.                 .qC1( qC1 ),                    // input - from U1
58.                 .qC2( qC2 ),                    // input - from U1
59.                 .HP( 8'd128 ),                  // direct driving
60.                 .MaxHP( 8'd255 ),               // direct driving
61.                 .RGB_Sig( U3_RGB_Sig )          // output - to top
62.             );
63.
64.             /**********************************/
65.
66.         reg [1:0]rBy1,rBy2,rBy3;
67.
68.             always @ ( posedge CLK_40Mhz or negedge RSTn )
69.                 if( !RSTn )
70.                     begin
71.                         rBy1 <= 2'b11;
72.                         rBy2 <= 2'b11;
73.                         rBy3 <= 2'b11;
74.                     end
75.                 else
76.                     begin
77.                         rBy1 <= { U1_HSYNC, U1_VSYNC };
78.                         rBy2 <= rBy1;
79.                         rBy3 <= rBy2;
80.                     end
81.
82.         /***************************/
83.
84.         assign { HSYNC, VSYNC } = rBy3;
85.         assign RGB_Sig = U2_RGB_Sig | U3_RGB_Sig ;
86.
87.         /***************************/
88.
89.     endmodule
```

screen_system.v 的内容基本上和图 2.11a 大同小异，需要注意的只有第 59~60 行的 HP 和 MaxHP 信号的直接驱动，在这里笔者输入 128 和 255，所以 HP 血条在屏幕上会显示一半而已。其余的，读者自己看着办吧。

==========

实验十二我们在实验十一的基础上，为第二个小屏幕添加“可变”的 HP 血条。这个 HP 血条会随着 HP 和 MaxHP 信号的更动而产生变化。实验十二其中的重点就是“图像映射”，亦即把“HP 血条的图像信息往 ram 模块的那里写去”。到目前为止，实验十二是为了实现字库而作准备的，因为要实现字库功能，数据的同步更敏感；而且“图像映射”

的计算更多；建模量更大，如果只是单纯用传统的概念去理解模块之间如何沟通的话，基本上会哭死设计者呢。

2.12 实验十三：我的小小字库

在实验十二里我们在实验十一的基础上，在 pika_hp 的图像下方实现可了变动的 HP 血条。在实验十三里，我们会以实验十二为继续，在 HP 血条的下方绘制出比卡丘的“战斗能力表”。不过有一点不同的是，我们不会去更动 screen_interface1.v 的内容 ，而是建立一个新的屏幕接口—screen_interface2.v，然后利用“叠加”的效果显示“比卡丘的战斗能力表”。

在建模篇的隐藏内容里，笔者也有为液晶 12864 建立字库写过实验。基本上在 12864 液晶里实现字库功能比起在 VGA 里容易许多。在还没有进入实验之前，笔者先介绍笔者所设计的字库，笔者取名为-初级字库。

格式化

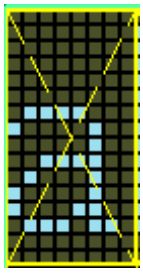


图 2.12a 小写字符 a。

每一个字符的大小（分辨率）都是 16 高* 8 宽，然而字符的位次序是按照 vga 的扫描次序，亦即逐行列填充，低位在前。图像信息会以“字节”为基础，储存在字库模块里，如图 2.12a 所示，小写字符 a 图像信息是 0x00 0x00 0x00 0x00 0x00 0x00 0x1E 0x21 0x20 0x3C 0x22 0x21 0x31 0x6E 0x00 0x00。顺便一提字母的原型是字体 MS Gothic，这是一个常用在日文系统里的字体，优点有简洁和清晰。

字库模块（rom 模块）说明

一个字符图像信息的容量是 16 * 8 bits，然而字库模块里储存了 93 个字符，所以字库模块的总容量是 93 * 16 * 8 bits，亦即 11904 bits，如果以字节计算的话是 1488 字节。如果忽略字节的存在那么每一个字符的偏移量是 16 个 words。

字符

空字符
rom 地址分别是 0~15。

abcdefghjjklmnopqrstuvwxyz

rom 地址分别是 16 ~ 431

ABCDEFGHIJKLMNOPQRSTUVWXYZ
rom 地址分别是 432 ~ 847

0123456789
rom 地址分别是 848 ~ 1007

！@#$%^&*（）
rom 地址分别是 1008 ~ 1167

-=[]¥;',./
rom 地址分别是 1168 ~ 1327

_+{}|:"<>?
rom 地址分别是 1328 ~ 1487

命令

初级字库的命令时以一字节为基础，位分配如表 2.12a

位	说明
[7]	1 位操作命令 0 为数据命令
[6..0]	如果最高位为 1 那么[6..0]为各种操作命令，反之为各种数据

表 2.12a 命令的位分配。

命令为数据的例子：

SPACE（空字符）
{1'b0, 7'd0} = 空字符

abcdefghjjklmnopqrstuvwxyz
{ 1'b0, 7'd1 } = a,
{1'b0, 7'd26} = z,

ABCDEFGHIJKLMNOPQRSTUVWXYZ
{ 1'b0, 7'd27 } = A,
{ 1'b0, 7'd52 } = Z,

0123456789
{ 1'b0, 7'd53 } = 0,
{ 1'b0, 7'd62 } = 9,

！@#$%^&*（）
{ 1'b0, 7'd63 } = !,
{ 1'b0, 7'd72 } =),

-=[]¥;',./
{ 1'b0, 7'd73 } = -,
{ 1'b0, 7'd82 } = /,

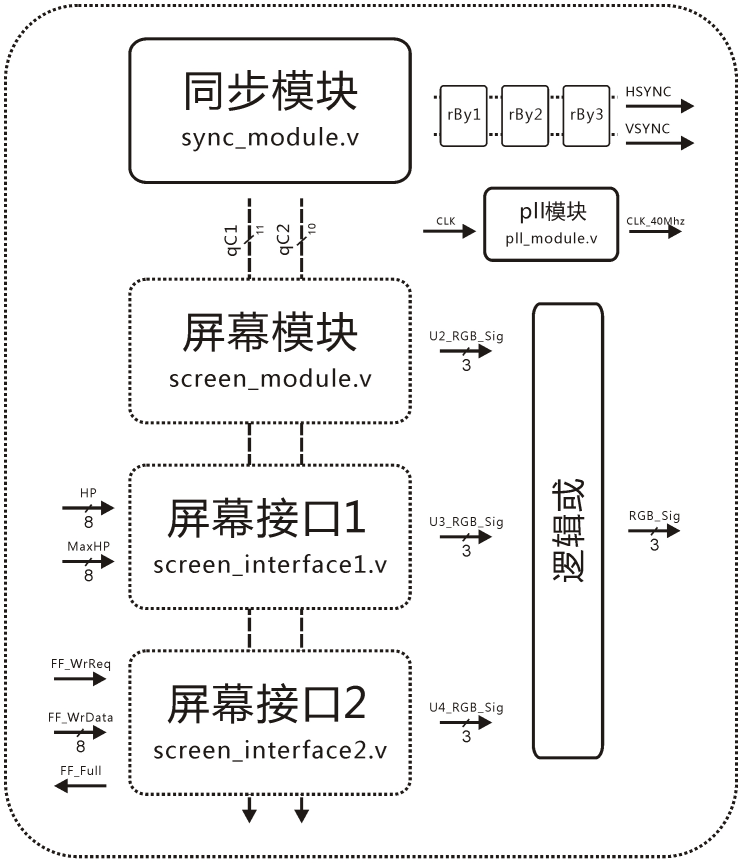
_+{}|:"<>?
{ 1'b0, 7'd83 } = _,
{ 1'b0, 7'd92 } = ?

命令为操作的例子：

{ 1'b1, 7'd1 } = <ENTER>,

初级字库的内容基本上就是这么简单而已，未来根据读者自己的需要再来扩展吧，但是在这之前还是先搞懂一些重点内容。

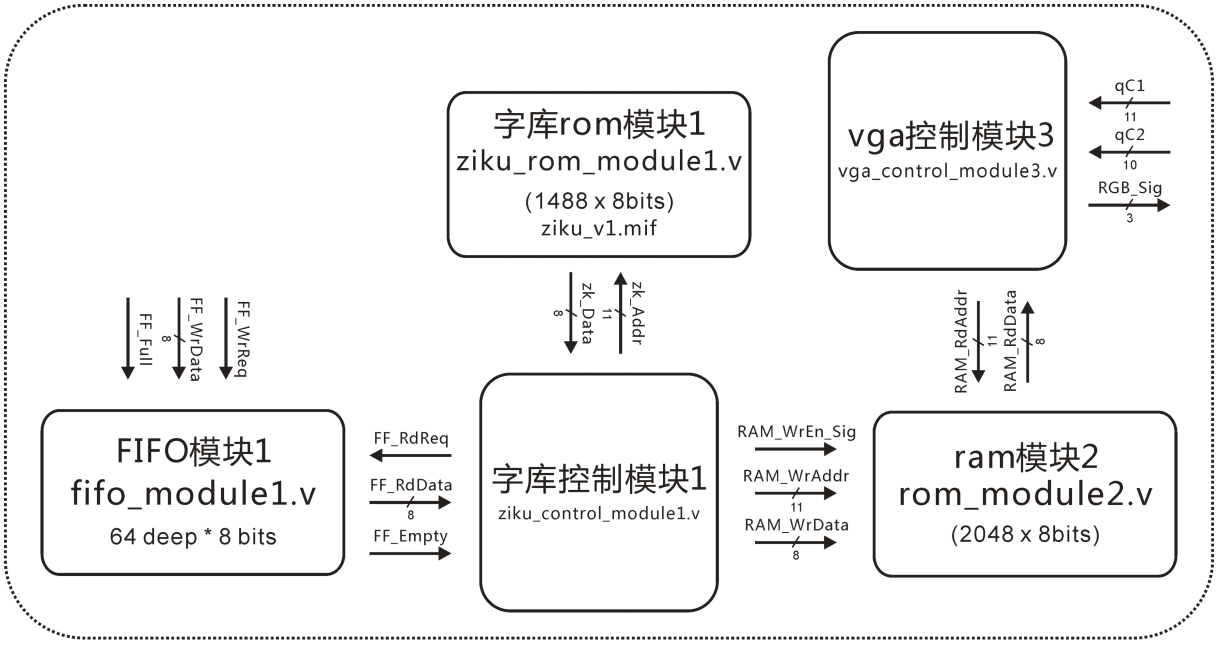
==========



screen_system.v

图 2.12b 屏幕系统（实验十三）。

图 2.12b 是实验十三的屏幕系统的建模图，它是从实验十三的基础上再添加一个屏幕接口 2。也就是说，在这里多了一个听众，听同步模块的演讲 qC1 和 qC2 ... 呵呵。



screen_interface2.v

图 2.12c 屏幕接口 2 的建模图。

图 2.12c 是组合模块 screen_interface2.v, 它包含了 vga 控制模块 3，ram 模块 2，字库控制模块 1,字库 rom 模块 1 和 FIFO 模块 1。vga 控制模块 3 的主要功能除了处理图像信息以外，还有就是控制分辨率为 128 * 128 的大小和屏幕的显示位置为 x-128，y-0；ram 模块 2 是图像信息的缓冲空间；字库 rom 模块 1 是包含了 93 字符的 rom 模块，该模块的初始化文件时 ziku_v1.mif；FIFO 模块 1 有 64 个深度，主要功能是用来缓冲命令；字库控制模块是整个屏幕接口 2 中的核心模块，它除了处理命令，读取字符的图像信息以外，太还要处理字符的在屏幕中的显示位置，和“图像映射”的计算工作。

ziku_control_module1.v

```
module ziku_control_module1
(
        input CLK,
        input RSTn,

        input [7:0]FF_RdData,
        input FF_Empty,
        output FF_RdReq,

        output [10:0]zk_Addr,
        input [7:0]zk_Data,

```

```
13.         output WrEn_Sig,
14.         output [10:0]WrAddr,
15.         output [7:0]WrData
16.
17.   );
18.
19.       /******************************/
20.
21.         parameter MASK = 8'b0111_1111, ENMASK = 8'b0111_0000;
22.
23.         /******************************/
24.
25.         reg [2:0]i;
26.         reg [4:0]x;
27.         reg [7:0]Point;
28.         reg [10:0]zkAddr, ramAddr;
29.         reg isRead;
30.         reg isWrite;
31.
32.         always @ ( posedge CLK or negedge RSTn )
33.               if( !RSTn )
34.                     begin
35.                                 i <= 3'd0;
36.                                 x <= 5'd0;
37.                                 Point <= 8'd0;
38.                                 zkAddr <= 11'd0;
39.                                 ramAddr <= 11'd0;
40.                                 isRead <= 1'b0;
41.                                 isWrite <= 1'b0;
42.                     end
43.                 else
44.                    case( i )
45.
46.                             0:
47.                             if( !FF_Empty ) begin isRead <= 1'b1; i <= i + 1'b1; end
48.
49.                             1:
50.                             begin isRead <= 1'b0; i <= i + 1'b1; end
51.
52.                             2:
53.                             if( !FF_RdData[7] ) i <= i + 1'b1; // if write character
54.                             else i <= 3'd6;
55.
56.                             /****************/ // Write Character
```

```
57.
58.                         3:
59.                         if( x == 16 ) begin Point <= MASK & (Point + 1'b1); x <= 5'd0; i <= 3'd0; end
60.                         else begin zkAddr <= (FF_RdData[6:0]<<4 )+ x; i <= i + 1'b1;end
61.
62.                         4:
63.                         begin isWrite <= 1'b1; ramAddr <= (x << 4) + ((Point>>4)<<8) + (Point & 8'h0f); i <= i + 1'b1; end
64.
65.                         5:
66.                         begin isWrite <= 1'b0; x <= x + 1'b1; i <= 3'd3; end
67.
68.                         /***************/ // Command
69.
70.                         6: // <Enter>
71.                         if( FF_RdData[6:0] == 7'd1 ) begin Point <= ENMASK & ( Point + 5'd16 ); i <= 3'd0; end
72.                         else i <= 3'd0;
73.
74.                     /**********************************************/
75.
76.                 endcase
77.
78.         /************************************/
79.
80.         assign FF_RdReq = isRead;
81.
82.         assign zk_Addr = zkAddr;
83.
84.         assign WrEn_Sig = isWrite;
85.         assign WrAddr = ramAddr;
86.         assign WrData = zk_Data;
87.
88.         /************************************/
89.
90.   endmodule
```

第 3~15 行是该模块的输入输出口声明，第 21 行是一些屏蔽用的常量声明，第 25~30 行是相关的寄存器声明。第 26 行的 x 是用于循环计数，第 27 行的 point 是用于计算显示位置，第 28 行的 zkAddr 和 ramAddr 分别是用来驱动 ram 模块 2 和字库模块的地址信号。第 29 行的 isRead 是用来驱动 FF_RdReq 信号，第 30 行的 isWrite 是用来驱动 WrEn_Sig 信号。

第 32~76 行是字库控制模块的核心操作。首先先把所有寄存器复位（第 35~41 行），然后在步骤 0~1（第 46~50 行）读入一个命令。第 52~54 行的步骤 2 是用来判断该命令是操作命令还是数据命令，如果是数据命令的话，FF_RdData 的最高位会是逻辑 0（第 53

行），然后步骤 i 会进入 58~65 行的写字符—伪函数，
否则的话（第 54 行）就会进入第 70~72 行执行操作。

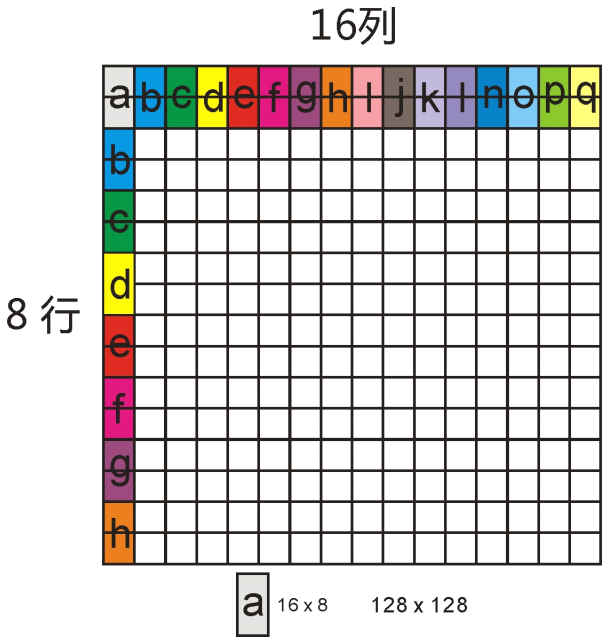


图 2.12d 分辨率 128 * 128 可容纳 n 个 16 *8 大小的字符。

要了解字库控制模块如何写入 1 个字符的图像信息，首先读者需要了解屏幕的分辨率大小，字符的图像信息大小。vga 控制模块 3 所控制的分辨率是 128 * 128 个像素，然而单个字符的图像信息就有 16 * 8 个像素。如图 2.13d 所示，128 个像素的高可以容纳 8 行，128 个像素的宽可容纳 16 个字符，总结说分辨率 128 * 128 的屏幕可以容纳 16 * 8 ，亦即 128 个字符。

行列	零	一	二	三	四	五	六	七	八	九	十	十一	十二	十三	十四	十五
零	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
一	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
二	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
三	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
四	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
五	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
六	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
七	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

表 2.12a Point 指向的 128 个显示位置。

第 37 行所声明的 Point 就是用来指向“字符的显示位置”，所以说第零行 Point 是 0~15，第七行 Point 是 112~127。第 21 行中声明的 MASK 常量 8'b0111_1111 是用来屏蔽 Point 不超过 127，屏蔽过程就是 Point 和 MASK“求与”运算，亦即 Point & MASK。

同样第 21 行中所声明的 ENMASK 常量 8'b0111_000 使用来屏蔽 Point 当遇见

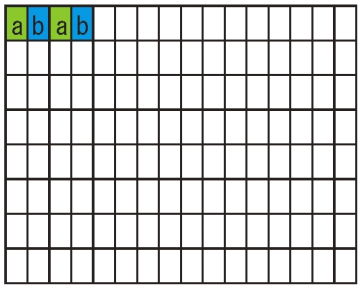
<ENTER>命令。笔者尝试想象一下，当我们在文本软件中输入<ENTER>命令以后，是不是会换行和返回开始的位置呢？ENMASK 常量基本上就是为了屏蔽 Point 的低四位，亦即返回该行的开始位置。打个比方，假设 Point 指向 10，那么当 Point 和 ENMASK 执行“求与”运算 8'b0000_1010 & 8'b01110_000，结果会是 8'd0，亦即 Point 指向该行的开始位置。

① 字库模块里字符的对应地址

	行列	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15
空字符	0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
字符a	16	00	00	00	00	00	00	1E	21	20	3C	22	21	31	6E	00	00
字符b	32	00	01	01	01	01	01	1D	23	41	41	41	41	23	1D	00	00

② 缓冲空间对应地址

行列	+0	+1	+2	+3
0	00	00	00	00
16	00	01	00	01
32	00	01	00	01
48	00	01	00	01
64	00	01	00	01
80	00	01	00	01
96	1E	1D	1E	1D
112	21	23	21	23
128	20	41	20	41
144	3C	41	3C	41
160	22	41	22	41
176	21	41	21	41
192	31	23	31	23
208	6E	1D	6E	1D
224	00	00	00	00
240	00	00	00	00
...	...	...	...	...
2032	...	...	...	...



③ 屏幕的图像信息映射

图 2.12e 字库模块对图像缓冲空间对屏幕显示的映射关系。

```
58.                     3:
59.                     if( x == 16 ) begin Point <= MASK & (Point + 1'b1); x <= 5'd0; i <= 3'd0; end
60.                     else begin zkAddr <= (FF_RdData[6:0]<<4 )+ x; i <= i + 1'b1;end
61.
62.                     4:
63.                     begin isWrite <= 1'b1; ramAddr <= (x << 4) + ((Point>>4)<<8) + (Point & 8'h0f); i <= i + 1'b1; end
64.
65.                     5:
66.                     begin isWrite <= 1'b0; x <= x + 1'b1; i <= 3'd3; end
```

一图胜千字，图 2.12e 说明了字库 rom 模块对 ram 模块 2 对屏幕显示的映射关系。步骤 3~5（第 58~66 行）就是把字库里的字符图像信息写入图像缓冲空间的操作.步骤 3 是从字库模块读出字符的图像信息，其中 FF_RdData[6:0] << 4，就是把数据命令直接转换成每一个字符的偏移量，然而 +x 就是读出陆续的 16 字节的图像信息。步骤 4 是用来计算“图像映射”以外，步骤 4 也会把步骤 3 读出来的一个字节图像信息写入图像缓冲空间里。步骤 5 是结束操作，和更新 x。

基本上在步骤 3~5 的循环操作过程中，笔者在设计上稍微利用了步骤和时序之间的关系 ... 嗯笔者越说越糊涂了，还是画一张时序图来说话吧。

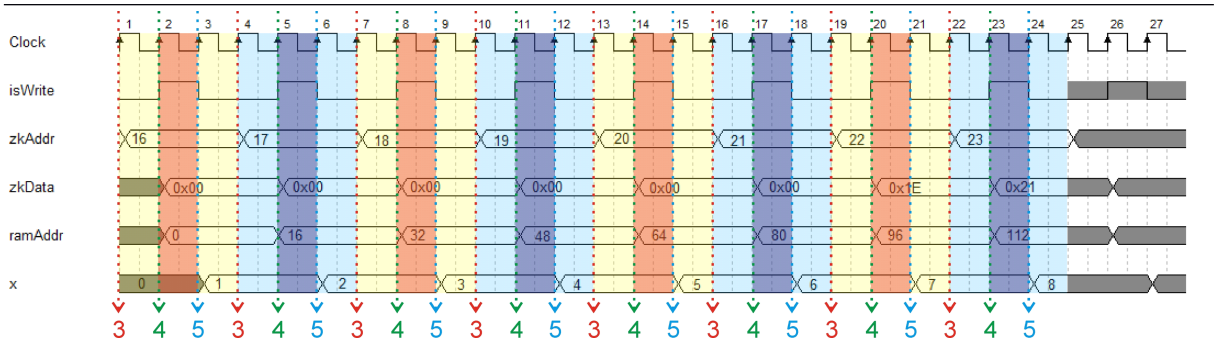


图 2.12f 步骤 3~5 前 8 次循环操作。

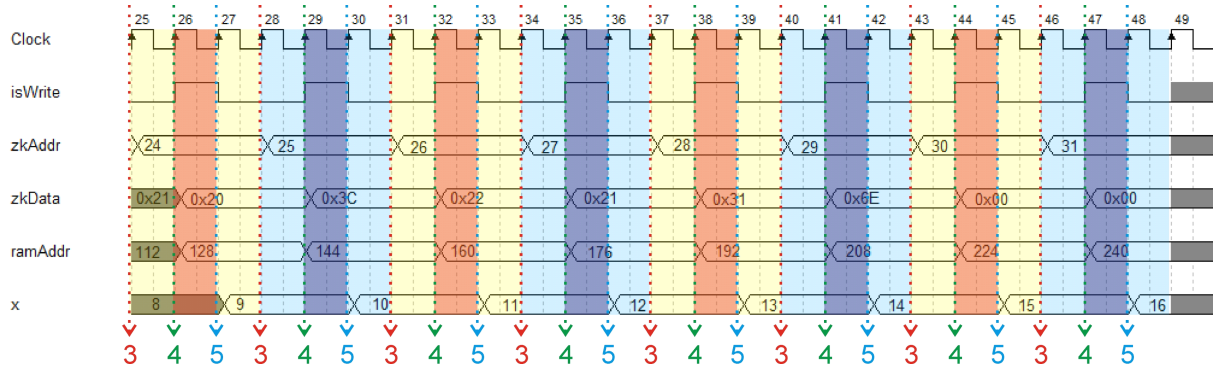


图 2.12g 步骤 3~5 后 8 次循环操作。

我们先假设从 FIFO 里读出一个数据命令 {1'b0, 7'd1}，亦即字符 a。 在 T1 的时候（步骤 3），FF_RdData[6:0] << 4 会把 7'd1 转换成字符在字库里的偏移量也就是 16，此时的 x 还是 0。所以在 T1 的未来 zkAddr 输出 16 的未来值。

在 T2 的时候，步骤 4 拉高 isWrite 和计算出图像缓冲空间的对应地址,（先无视 (Point >>4) << 8 和 Point & 0x0f）此时 x 为 0。所以在 T2 的未来，isWrite 被拉高，ramAddr 输出未来值 0。在同一个时候字库模块，检测到 zkAddr 的过去值是 16，所以在 T2 的未来它输出字符 a 第一字节的图像信息。

在 T3 的时候，步骤 5 决定拉低 isWrite 和递增 x。所以在 T3 的未来 isWrite 被拉低，x 递增为 1。在同一个时候 ram 模块 2 检测到 isWrite 被拉高，所以它决定根据 ramAddr 把 zkData 写入（注意第 89 行 zkData 直接驱动 ramData），所以在 T3 的未来图像缓冲空间中的第 0 字写入数据 0x00。步骤 3~5 经过 16 次的循环操作以后，时序过程如图 2.12f 和图 2.12g 所示。如果笔者一个一个时钟去分析图 2.12f 和图 2.12g 的话，说实话笔者一定会精尽人亡 ...

在这里，我们需要用到前面所学到的知识了，读者尝试想象有两个模块在沟通中，其中字库控制模块是发话者，然而 ram 模块 2 是听话者，图 2.12f 和图 2.12g 每一个浅颜色的矩形框都代表字库控制模块的“每一次发话”(步骤 3~5)，然而每一个深颜色的矩形框（步骤 4~5 之间），代表 ram 模块 2 每一次所要接收的话。如果我们把所有数据都切成一块一块的话，这时候 isWrite 就充当着“ram 模块 2 的接收标签”。

大约在 T49 的时候，我们已经成功把 a 字符的图像信息，从字库模块写到图像缓冲空间

中（ram 模块 2），（读者对应看看图 2.12f 和图 2.12g 中深颜色的矩形框是不是和图 2.12e 中符合呢？），这时候的 x 的过去值是 16，第 59 行的 if 条件成立 x 被清理和 Point 递增为 1。

==========

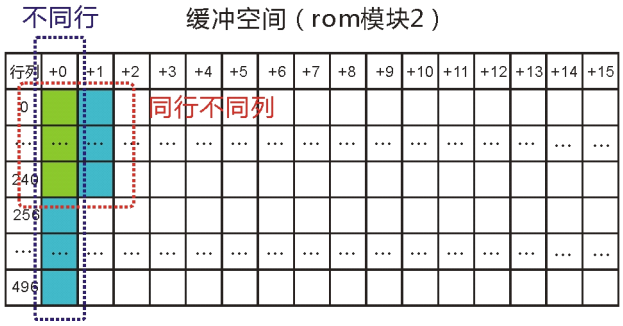


图 2.12h 同行不同列和不同行的 Point 功能区别。

在这里 Point 有 3 个功能，在屏幕的显示上 Point 具有指向字符显示的位置（第一功能）。在“图像映射”的计算中，Point 有指向同行不同列的功能（第二功能，既是现在我们要讨论的 Point & 8'h0f）。同样在“图像映射”的计算中，Point 有指向不同行的功能（第三功能，目前无视中的 (Point >> 4)<<16）。

Point 的第一功能很简单，笔者就不解释了。反之第二功能和第三功能都是在“图像映射”中发生的，对于初学或者初次碰到图像处理的朋友脑袋一定会冒烟。虽说一个分辨率 128 *128 的屏幕可以容纳 128 个 17 * 8 大小的字符，随此 Point 会指向 0~127 的范围，但是“Point 指向 0~127 的范围”这只是“字符在屏幕上显示”的了解而已。在“图像映射”的计算中，当我们成功显示一个字符以后 Point 会递增，在此 Point 会指向显示位置 1，这时所谓的“同行不同列”就发生了，在计算中，我们需要把“这个 1”添加进来，如图 2.12h 所示，区别如表 2.12b 所示。

当 point 为 0 的时候，	当 point 为 1 的时候，	字相差
第 0 个字节图像信息储存在 第 0 字。	第 0 个字节图像信息储存在 第 1 字。	+1
第 1 个字节图像信息储存在 第 16 字。	第 1 个字节图像信息储存在 第 17 字。	+1
第 2 个字节图像信息储存在 第 32 字。	第 2 个字节图像信息储存在 第 33 字。	+1
第 3 个字节图像信息储存在 第 48 字。	第 3 个字节图像信息储存在 第 49 字。	+1
第 4 个字节图像信息储存在 第 64 字。	第 4 个字节图像信息储存在 第 65 字。	+1
第 5 个字节图像信息储存在 第 80 字。	第 5 个字节图像信息储存在 第 81 字。	+1
第 6 个字节图像信息储存在 第 96 字。	第 6 个字节图像信息储存在 第 97 字。	+1
第 7 个字节图像信息储存在 第 112 字。	第 7 个字节图像信息储存在 第 113 字。	+1
第 8 个字节图像信息储存在 第 128 字。	第 8 个字节图像信息储存在 第 129 字。	+1
第 9 个字节图像信息储存在 第 144 字。	第 9 个字节图像信息储存在 第 145 字。	+1
第 10 个字节图像信息储存在 第 160 字。	第 10 个字节图像信息储存在 第 161 字。	+1
第 11 个字节图像信息储存在 第 176 字。	第 11 个字节图像信息储存在 第 177 字。	+1
第 12 个字节图像信息储存在 第 192 字。	第 12 个字节图像信息储存在 第 193 字。	+1
第 13 个字节图像信息储存在 第 208 字。	第 13 个字节图像信息储存在 第 209 字。	+1

第 14 个字节图像信息储存在 第 224 字。	第 14 个字节图像信息储存在 第 225 字。	+1
第 15 个字节图像信息储存在 第 240 字。	第 15 个字节图像信息储存在 第 241 字。	+1

表 2.12b 同行不同列。

再假设现在笔者从 FIFO 里读出另一个数据命令 {1'b0, 7'd2}，亦即字符 b。此时 Point 为 1，所以我们不可以无视第 63 行的 Point & 8'h0f 了，但是依然无视 (Point >> 4) << 8 。Point & 8'h0f 就是取 Point 的第四位亦即 0~15。

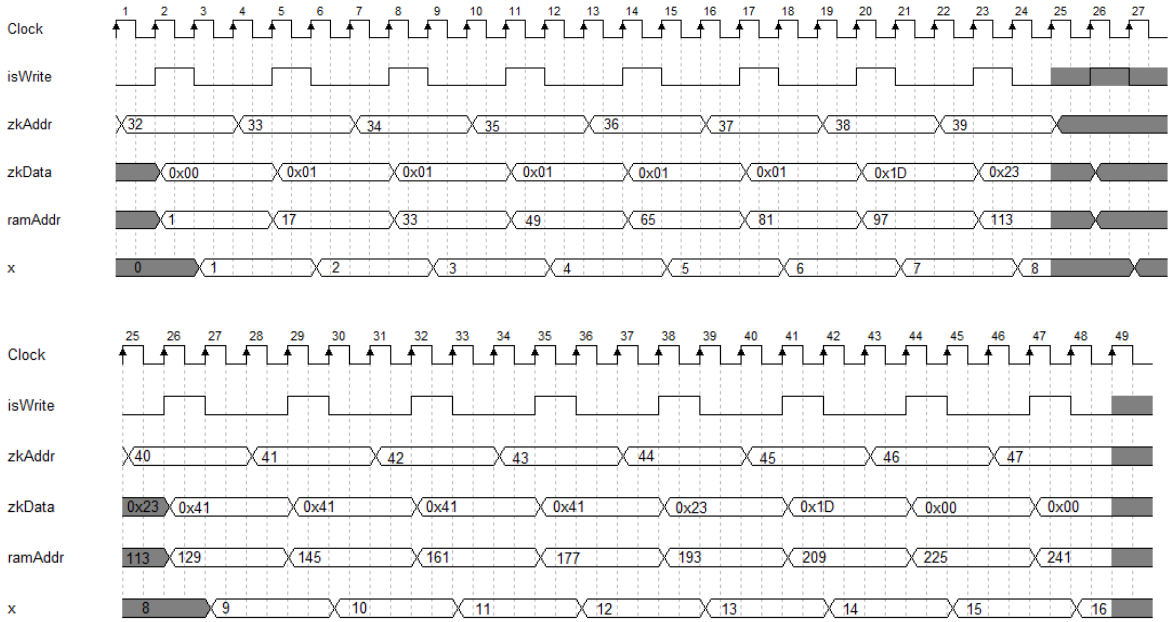


图 2.12i 把第二个字符 b 写入图像缓冲空间的时序过程。

图 2.12i 同样是发生在步骤 3~5 的 16 次循环操作，不过不同的是，这次的 Point 为 1，而且是写入字符 b。时序的理解要用老办法还是和图 2.12f 和图 2.12g 一样，自己看着办吧。大约在 T49 的时候，第二个字符 b 成功写入缓冲空间，此时 x 清理，Point 会递增为 2（读者自己看看图 2.12i 的过程是否和图 2.12e 符合呢？）。

==========

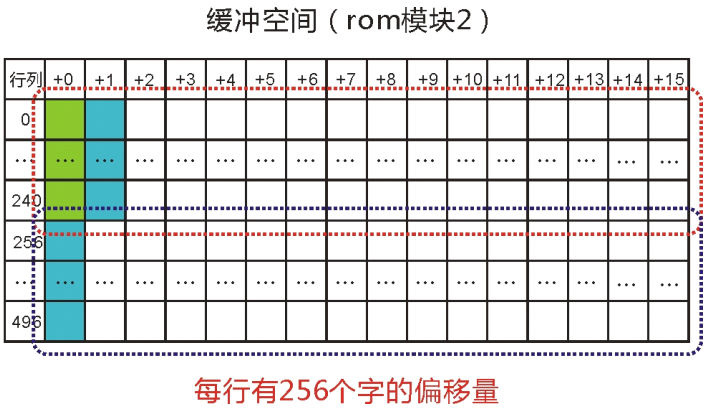


图 2.12j 每一行的偏移量。

所谓的不同行就是 Point 在不同的数字范围，以实验十三为例第零行 Point 是 0~15，第一行 Point 是 16~31，第二行 Point 是 32~47 ... 其它以此类推，更具体一点的话可以把 Point 区分为低四位和高四位来看，假设现在的 Point 指向第一行第一列，亦即 16 + 1。

```
Point 8'h 0001 0001
```

Point 的低四位是 Point 的第二功能，亦即同行不同列（这也是为什么 Point & 8'h0f 的原因）。Point 的高四位是 Point 的第三功能，亦即不同行。在屏幕上，Point 的每一行偏移值是 16，但是在“图像映射”的计算中（图像缓冲空间里），每一行的偏移量却是 256 个 words（这也是为什么 (Point >>4)<<8 的原因），如图 2.12j 所示。 我们稍微的计算一下：

```
ramAddr = (8'b0001_0001 >> 4) <<8 + (8'b0001_0001 & 8'h0f)
        = 8'b0000_0001 << 8 + 8'b0000_0001
        = 9'b1_0000_0000 + 8'b0000_0001
        = 9'b1_0000_0001 (9'd257)
```

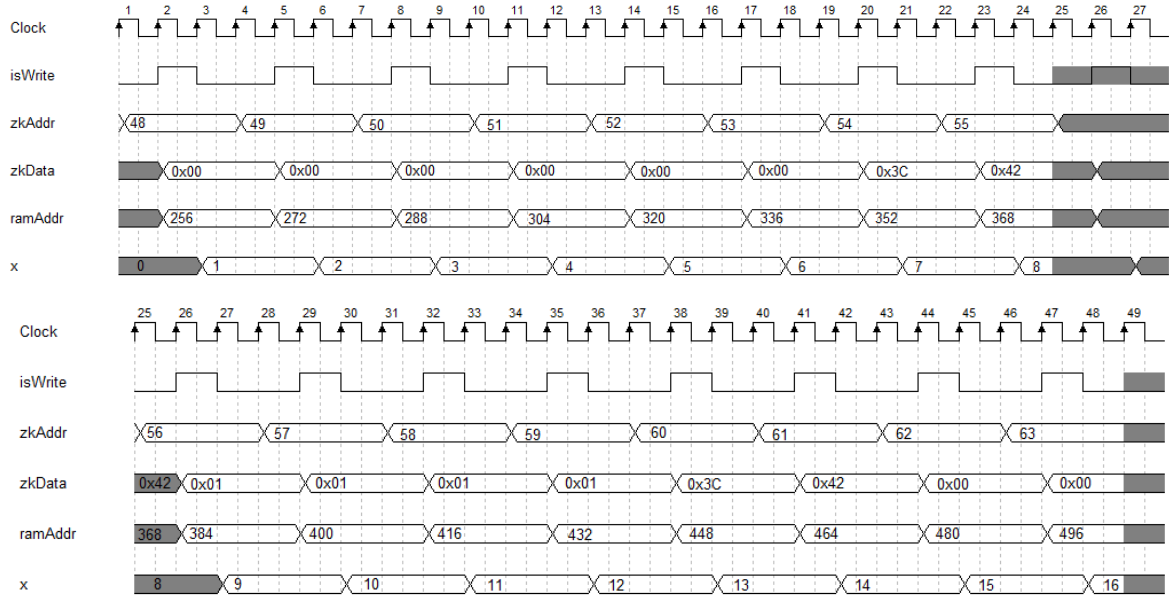


图 2.12k 当 Point 为 16，从字库里往图像缓冲空间里写字符 c 的时序过程。

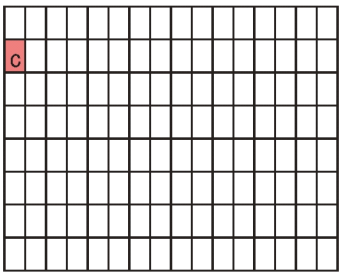
再再假设当前的 Point 为 16，笔者再再从 FIFO 里读出一个数据命令{1'b0, 7'd3}，亦即字符 C。然后步骤 3~5 经过 16 次的循环操作，产生的时序过程如图 2.12k 所示。再一次，读者要用什么方法去理解图 2.12k 随读者自己喜欢吧。

① 字库模块里字符的对应地址

	行列	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15
空字符	0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
字符a	16	00	00	00	00	00	00	1E	21	20	3C	22	21	31	6E	00	00
字符b	32	00	01	01	01	01	01	1D	23	41	41	41	41	23	1D	00	00
字符C	48	00	00	00	00	00	00	3C	42	01	01	01	01	3C	42	00	00

② 缓冲空间对应地址

行列	+0	+1	+2	+3
···	···	···	···	···
256	00	···	···	···
272	00	···	···	···
288	00	···	···	···
304	00	···	···	···
320	00	···	···	···
336	00	···	···	···
352	3C	···	···	···
368	42	···	···	···
384	01	···	···	···
400	01	···	···	···
416	01	···	···	···
432	01	···	···	···
448	3C	···	···	···
464	42	···	···	···
480	00	···	···	···
496	00	···	···	···
···	···	···	···	···



③ 屏幕的图像信息映射

图 2.12l 优惠图-对应关系。

最后就给出一张优惠图 2.12l，这张图和图 2.12k 有对应关系，很简单而已，笔者就不解释了，读者自己看着办吧。

```
70.                         6: // <Enter>
71.                         if( FF_RdData[6:0] == 7'd1 ) begin Point <= ENMASK & ( Point + 5'd16 ); i <= 3'd0; end
72.                         else i <= 3'd0;
73.
74.                     /*********************************************/
75.
76.                 endcase
77.
78.         /*************************************/
79.
80.         assign FF_RdReq = isRead;
81.
82.         assign zk_Addr = zkAddr;
83.
84.         assign WrEn_Sig = isWrite;
85.         assign WrAddr = ramAddr;
86.         assign WrData = zk_Data;
87.
88.         /*************************************/
89.
90.   endmodule
```

第 70~72 行的步骤 6 是操作命令的处理，主要经由步骤 2。在此只有一个 <Enter> 操作要处理而已（第 71 行）就是 Point 自增 16 然后屏蔽与 ENMASK（前面已经说过了）。第 80~86 行是输出和驱动，读者就自己看着办吧。

vga_control_module3.v

vga 控制模块 3 基本上和实验十二的 vga 控制模块 2 一模一样。因此实验十二的屏幕和实验十三的屏幕会相互叠加。

screen_interface2.v

```
module screen_interface2
(
        input CLK,
        input RSTn,

        input [10:0]qC1,
        input [9:0]qC2,

        input FF_WrReq,
        input [7:0]FF_WrData,
        output FF_Full,

        output [2:0]RGB_Sig
);

      /******************************/

        wire FF_Empty;
        wire [7:0] FF_RdData;

        fifo_module1 U1
        (
              .clock ( CLK ),
              .wrreq ( FF_WrReq ),      // input - from top
              .data ( FF_WrData ),      // input - from top
              .full( FF_Full ),         // output - to top
              .rdreq ( FF_RdReq ),      // input - from
              .empty ( FF_Empty ),      // output - to
              .q ( FF_RdData )          // output - to
        );

        /******************************/

        wire [7:0] zk_Data;

```

```
36.         ziku_rom_module1 U2
37.         (
38.                 .clock( CLK ),
39.                 .address( zk_Addr ),            // input - from U3
40.                 .q( zk_Data )                   // output - to U3
41.         );
42.
43.         /*********************************/
44.
45.         wire FF_RdReq;
46.         wire [10:0]zk_Addr;
47.         wire RAM_WrEn_Sig;
48.         wire [10:0]RAM_WrAddr;
49.         wire [7:0]RAM_WrData;
50.
51.         ziku_control_module1 U3
52.         (
53.                 .CLK( CLK ),
54.                 .RSTn( RSTn ),
55.                 .FF_RdData( FF_RdData ),          // input - from U1
56.                 .FF_Empty( FF_Empty ),           // input - form U1
57.                 .FF_RdReq( FF_RdReq ),           // output - to U1
58.                 .zk_Addr( zk_Addr ),               // output - to U2
59.                 .zk_Data( zk_Data ),             // input - from U1
60.                 .WrEn_Sig( RAM_WrEn_Sig ),   // output - to U4
61.                 .WrAddr( RAM_WrAddr ),        // output - to U4
62.                 .WrData( RAM_WrData )          // output - to U4
63.         );
64.
65.         /*********************************/
66.
67.         wire [7:0]RAM_RdData;
68.
69.         ram_module2 U4
70.         (
71.                 .CLK( CLK ),
72.                 .RSTn( RSTn ),
73.                 .WrEn_Sig( RAM_WrEn_Sig ),   // input - from U3
74.                 .WrAddr( RAM_WrAddr ),          // input - from U3
75.                 .WrData( RAM_WrData ),          // input - from U3
76.                 .RdAddr( RAM_RdAddr ),          // input - from U5
77.                 .RdData( RAM_RdData )           // output - to U5
78.         );
79.
```

```
80.         /*******************************/
81.
82.         wire [10:0]RAM_RdAddr;
83.
84.         vga_control_module3 U5
85.         (
86.                 .CLK( CLK ),
87.                 .RSTn( RSTn ),
88.                 .qC1( qC1 ),                    // input - from top
89.                 .qC2( qC2 ),                    // input - from top
90.                 .RAM_Addr( RAM_RdAddr ),        // output - to U4
91.                 .RAM_Data( RAM_RdData ),        // input - from U4
92.                 .RGB_Sig( RGB_Sig )             // output - to top
93.         );
94.
95.         /*******************************/
96.
97.   endmodule
```

screen_interface2.v 基本上和图 2.12c 大同小异，自己看着办吧。

demo_control_module.v

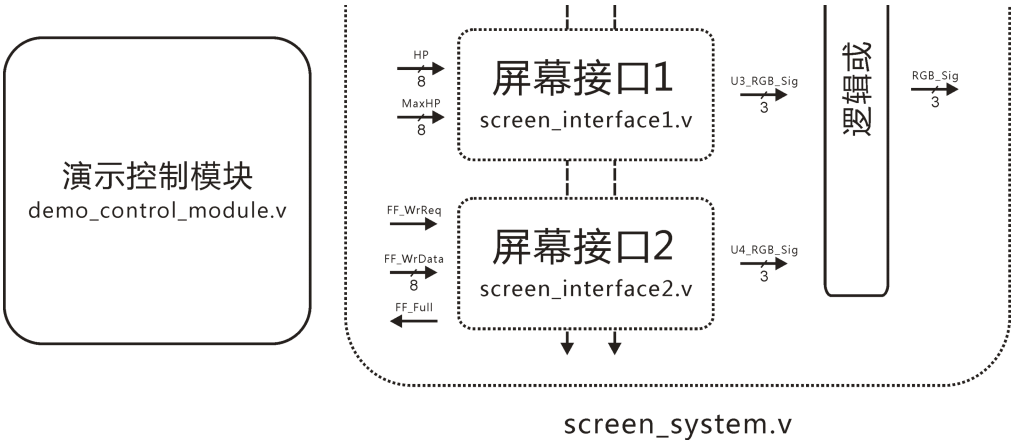


图 2.12m 演示控制模块的图形。

演示控制模块是用来驱动屏幕接口 1 和 2 的控制模块。笔者为了方便，把演示控制模块写在 screen_system.v 的下边，具体的内容我们还是来看代码吧：

```
126. module demo_control_module
127. (
128.         input CLK,
129.         input RSTn,
```

```
130.
131.        output [7:0]HP,
132.        output [7:0]MaxHP,
133.
134.        output FF_WrReq,
135.        output [7:0]FF_WrData
136. );
137.
138.      /***************************/
139.
140.        parameter SPACE = { 1'b0, 7'd0 };
141.
142.        parameter   _ATK = { 8'd0, 1'b0,7'd27, 1'b0,7'd46, 1'b0,7'd37},
143.                    _DEF = { 8'd0, 1'b0,7'd30, 1'b0,7'd31, 1'b0,7'd32 },
144.                    SATK = { 1'b0,7'd45, 1'b0,7'd27, 1'b0,7'd46, 1'b0,7'd37 },
145.                    SDEF = { 1'b0,7'd45, 1'b0,7'd30, 1'b0,7'd31 ,1'b0,7'd32 },
146.                    _96_ = { 8'd0, 1'b0,7'd61, 1'b0,7'd58, 8'd0 },
147.                    _104 = { 8'd0, 1'b0,7'd53, 1'b0,7'd62, 1'b0,7'd56 },
148.                    _287 = { 8'd0, 1'b0,7'd54, 1'b0,7'd60, 1'b0,7'd59 },
149.                    _135 = { 8'd0, 1'b0,7'd53, 1'b0,7'd55, 1'b0,7'd57 };
150.
151.        parameter ENTER = { 1'b1, 7'd1 };
152.
153.        /***************************/
154.
155.        reg [4:0]i,Go;
156.        reg [7:0]rHP, rMaxHP;
157.        reg [31:0]T1,T2;
158.        reg [7:0]rData;
159.        reg isWrite;
160.
161.        always @ ( posedge CLK or negedge RSTn )
162.            if( !RSTn )
163.                begin
164.                        i <= 5'd0;
165.                        Go <= 5'd0;
166.                        rHP <= 8'd0;
167.                        rMaxHP <= 8'd0;
168.                        T1 <= 32'd0;
169.                        T2 <= 32'd0;
170.                        rData <= 8'd0;
171.                        isWrite <= 1'b0;
172.                    end
173.            else
```

```
174.                    case( i )
175.
176.                            0: // drive U3
177.                            begin rHP <= 8'd10; rMaxHP <= 8'd200; i <= i + 1'b1; end
178.
179.                            1,2,3:    // dirve U5 and Enter 3 times
180.                            begin isWrite <= 1'b1; rData <= ENTER; i <= i + 1'b1; end
181.
182.                            4:
183.                            begin isWrite <= 1'b0; i <= i + 1'b1; end
184.
185.                            5:    // Write ATK and VALUE
186.                            begin T1 <= _ATK; T2 <= _96_; i <= 5'd10; Go <= i + 1'b1; end
187.
188.                            6: // Write DEF and VALUE
189.                            begin T1 <= _DEF; T2 <= _104; i <= 5'd10; Go <= i + 1'b1; end
190.
191.                            7: // Write SATK( Special Attack ) and VALUE
192.                            begin T1 <= SATK; T2 <= _287; i <= 5'd10; Go <= i + 1'b1; end
193.
194.                            8: // Write SDEF( Spedical Defend ) and VALUE
195.                            begin T1 <= SDEF; T2 <= _135; i <= 5'd10; Go <= i + 1'b1; end
196.
197.                            9: // Hold
198.                            i <= i;
199.
200.                            /*************************/ // Write function
201.
202.                            10,11:    // Write Space 2 times
203.                            begin isWrite <= 1'b1; rData <= SPACE; i <= i + 1'b1; end
204.
205.                            12,13,14,15: // Write Title
206.                            begin isWrite <= 1'b1; rData <= T1[31:24]; T1 <= T1 << 8; i <= i + 1'b1; end
207.
208.                            16,17: // Space x 2
209.                            begin isWrite <= 1'b1; rData <= SPACE; i <= i + 1'b1; end
210.
211.                            18,19,20,21: // Write value
212.                            begin isWrite <= 1'b1; rData <= T2[31:24]; T2 <= T2 << 8; i <= i + 1'b1; end
213.
214.                            22: // <Enter> x 1;
215.                            begin isWrite <= 1'b1; rData <= ENTER; i <= i + 1'b1; end
216.
217.                            23: // Return
```

```
218.                                begin isWrite <= 1'b0; i <= Go; end
219.
220.                                /*********************************/
221.
222.                            endcase
223.
224.        /***********************************/
225.
226.        assign HP = rHP;
227.        assign MaxHP = rMaxHP;
228.        assign FF_WrReq = isWrite;
229.        assign FF_WrData = rData;
230.
231.        /*************************************/
232.
233. endmodule
```

由于 demo_control_module 是写在 screen_system.v 的下方，所以行数是从 100 开始。第 128~135 是输入和输出口的声明；第 140 行是空数据，亦即 SPACE 的常量声明；第 142~149 都是比卡丘的战斗能力表的常量声明，第 142~145 行是战斗能力项，第 146~149 是战斗能力值。第 151 行是操作命令 ENTER 的常量声明。

第 155~159 是相关的寄存器声明。第 155 行的 i 和 Go 寄存器是步骤和返回步骤的用途；第 166 行 rHP 和 rMaxHP 是用来驱动 HP 和 MaxHP 信号的；第 157 行的 T1 和 T2 是暂存空间，主要是用来把比卡丘的站都能表写入 FIFO 里。第 158~159 行的 rData 和 isWrite 同样是用于 FIFO（看命名就知道用途了吧？）。

一开始的时候所有寄存器都清零（第 163~171 行）。步骤 0（第 176~177 行）是赋值 rHP 和 rMaxHP。步骤 1~3（第 179~180 行）主要是往 FIFO 写入 3 次 ENTER 的操作命令（换行 3 次），步骤 4 则是拉低 isWrite。第 200~217 行是向 FIFO 写入比卡丘的战斗能力表的伪函数。

步骤 10~11 是向 FIFO 写入 2 个空数据（第 202~203 行）；
步骤 12~15 是向 FIFO 写入第一项战斗能力（第 205~206 行）；
步骤 16~17 是向 FIFO 写入 2 个空数据（第 208~209 行）；
步骤 18~21 是向 FIFO 写入第一项战斗能力的值（第 211~212 行）。
步骤 22 是向 FIFO 写入一个 ENTER 操作命令（第 214~215 行）。
步骤 23 是拉低 isWrite，然后返回步骤 Go（第 217~218 行）。

步骤 5~8（第 185~195 行）就是这样调用这个伪函数，把第一项到第四项的战斗能力和战斗能力值全部向 FIFO 里写去。步骤 9 是停止操作。

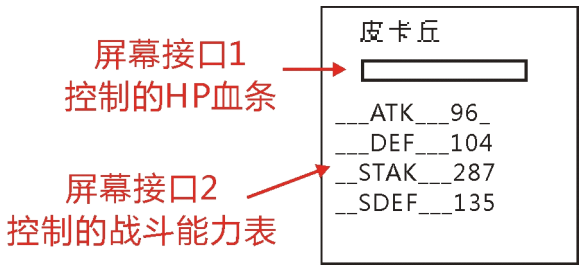


图 2.12n 实验十二和实验十三屏幕叠加的效果。

最后屏幕会如图 2.12n 般显示，亦即实验十二和实验十三会相互叠加。但是特别的是，HP 血条的控制是由屏幕接口 1 来控制，比卡丘的战斗能力表则由屏幕接口 2 来控制（低级建模的功劳）。

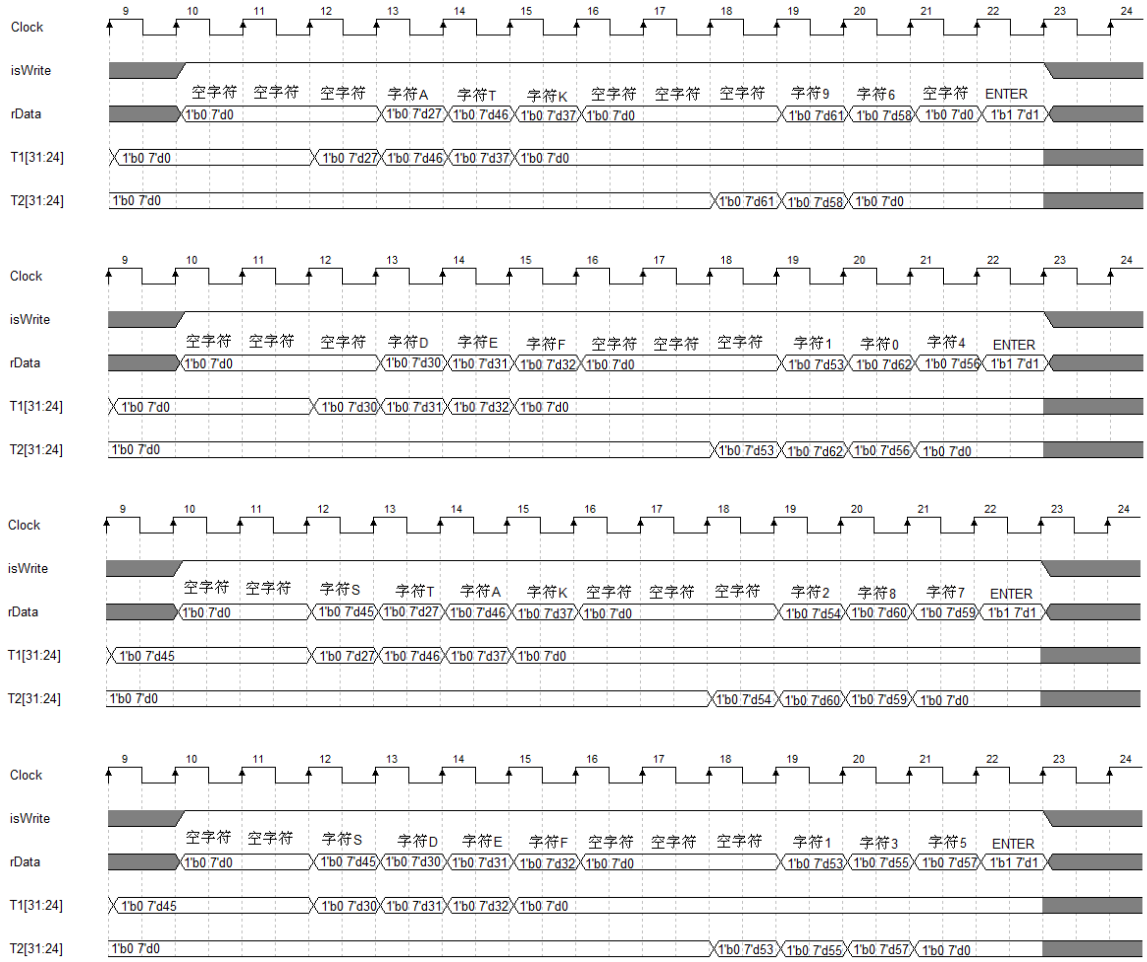


图 2.12o 比卡丘的战斗能力表向 FIFO 的写入时序过程。

图 2.12o 是比卡丘战斗能力表的写入时序过程（伪函数部分），基本上笔者已经对图动了不少“优惠”了，凡事读过时序篇的读者，对读者来说一定很容易。又或者要把信号切成一块一块来看也没有问题 ...

screen_system.v

```
1.    module screen_system
2.    (
3.            input CLK,
4.            input RSTn,
5.
6.            output HSYNC,
7.            output VSYNC,
8.            output [2:0]RGB_Sig
9.    );
10.         /***************************/
11.
12.           wire CLK_40Mhz;
13.
14.           pll_module     U0
15.           (
16.                .inclk0 ( CLK ),                // in 20MHz
17.                .c0 ( CLK_40Mhz )         // output 40Mhz
18.           );
19.
20.           /****************************/
21.
22.           wire [10:0]qC1;
23.           wire [9:0]qC2;
24.           wire U1_HSYNC, U1_VSYNC;
25.
26.           sync_module U1
27.           (
28.                   .CLK( CLK_40Mhz ),       // input - from U0
29.                   .RSTn( RSTn ),
30.                   .qC1( qC1 ),                    // output - to U3
31.                   .qC2( qC2 ),                    // output - to U3
32.                   .HSYNC( U1_HSYNC ),    // output - to bypass register
33.                   .VSYNC( U1_VSYNC )      // output - to bypass register
34.           );
35.
36.           /*******************************/
37.
38.           wire [2:0]U2_RGB_Sig;
39.
40.           screen_module U2
41.           (
42.                   .CLK( CLK_40Mhz ),           // input - from U0
```

```
43.                 .RSTn( RSTn ),
44.                 .qC1( qC1 ),                    // input - from U1
45.                 .qC2( qC2 ),                    // input - from U1
46.                 .RGB_Sig( U2_RGB_Sig )    // output - to top
47.         );
48.
49.         /*********************************/
50.
51.         wire [2:0]U3_RGB_Sig;
52.
53.         screen_interface1 U3
54.         (
55.                 .CLK( CLK_40Mhz ),          // input - from U0
56.                 .RSTn( RSTn ),
57.                 .qC1( qC1 ),                    // input - from U1
58.                 .qC2( qC2 ),                   // input - from U1
59.                 .HP( HP ),                      // input - from U5
60.                 .MaxHP( MaxHP ),            // input - from U5
61.                 .RGB_Sig( U3_RGB_Sig )    // output - to top
62.         );
63.
64.         /************************************/
65.
66.         wire [2:0]U4_RGB_Sig;
67.
68.         screen_interface2 U4
69.         (
70.                 .CLK( CLK_40Mhz ),              // input - from U0
71.                 .RSTn( RSTn ),
72.                 .qC1( qC1 ),                       // input - from U1
73.                 .qC2( qC2 ),                       // input - from U1
74.                 .FF_WrReq( FF_WrReq ),         // input - from U5
75.                 .FF_WrData( FF_WrData ),       // input- from U5
76.                 .FF_Full( ),
77.                 .RGB_Sig( U4_RGB_Sig )
78.         );
79.
80.         /*********************************/
81.
82.         wire [7:0]HP, MaxHP;
83.         wire FF_WrReq;
84.         wire [7:0]FF_WrData;
85.
86.         demo_control_module U5
```

```
87.         (
88.                 .CLK( CLK_40Mhz ),          // input - from U0
89.                 .RSTn( RSTn ),
90.                 .HP( HP ),                          // output - to U3
91.                 .MaxHP( MaxHP ),                // output - to U3
92.                 .FF_WrReq( FF_WrReq ),        // output - to U4
93.                 .FF_WrData( FF_WrData )      // output - to U4
94.         );
95.
96.
97.         /********************************/
98.
99.       reg [1:0]rBy1,rBy2,rBy3;
100.
101.        always @ ( posedge CLK_40Mhz or negedge RSTn )
102.              if( !RSTn )
103.                    begin
104.                            rBy1 <= 2'b11;
105.                            rBy2 <= 2'b11;
106.                            rBy3 <= 2'b11;
107.                        end
108.                else
109.                    begin
110.                            rBy1 <= { U1_HSYNC, U1_VSYNC };
111.                            rBy2 <= rBy1;
112.                            rBy3 <= rBy2;
113.                        end
114.
115.      /***************************/
116.
117.      assign { HSYNC, VSYNC } = rBy3;
118.      assign RGB_Sig = U2_RGB_Sig | U3_RGB_Sig | U4_RGB_Sig;
119.
120.      /***************************/
121.
122. endmodule
```

screen_system.v 基本上和图 2.12b 大同小异，笔者就不多解释了，读者自己看着办吧。

==========

哇 ... 实验内容一个比一个多，真是累死笔者了。总结的说，建立图库苦难的地方就是自定义字库，建立字库控制模块，还有就是“图像映射”等工作，不过只要一步一步来，没有什么东西是困难的。在实验十三里，读者学到的东西可不少呀 ... 除了建立字库以

外，最重要的地方—即使不依赖仿真我们也可以在自己的脑海里建立波形图。事实上，脑海里有没有建立波形图是没有被限定的，我们可以用其他的办法去看待“模块”的沟通，如：把信号切成一块一块（一个时钟一个时钟）来看 ...

2.13 实验十四：字库的屏幕扩张（自定义字符串）

实验十四要实现的内容是基于实验十三的字库，把 128 * 128 的屏幕扩张为 128 高* 256 宽。基本上实验十四的内容不难，就是修改几行代码而已，此外我们还要自定义“字符串”。

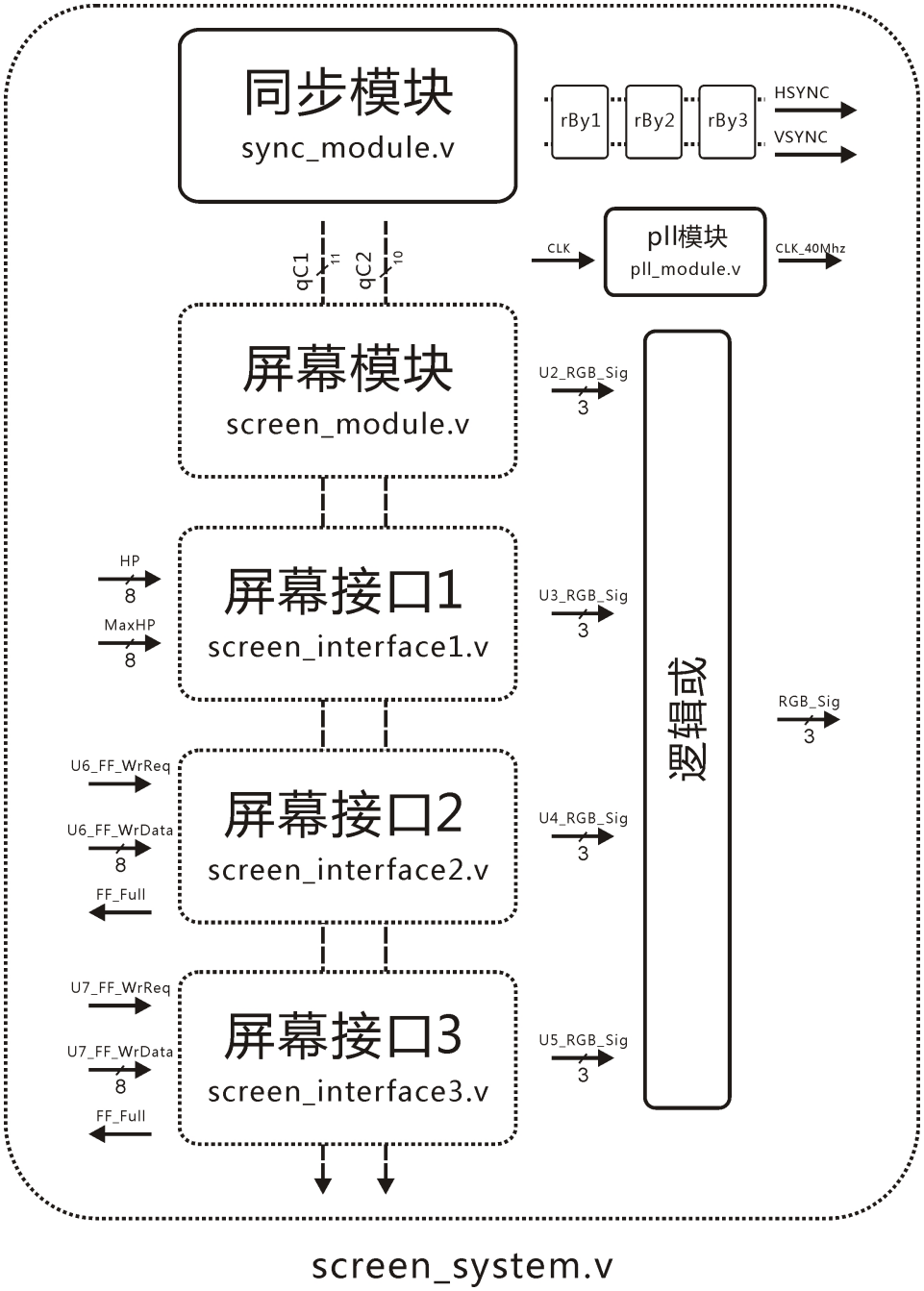


图 2.13a 实验十三屏幕系统的建模图。

图 2.13a 是实验十三的屏幕系统，它主要从实验十二的基础上再添加一个屏幕接口 3。屏幕接口 3 实际上也是一个控制字库的接口模块，不过分辨率是 128 * 256。

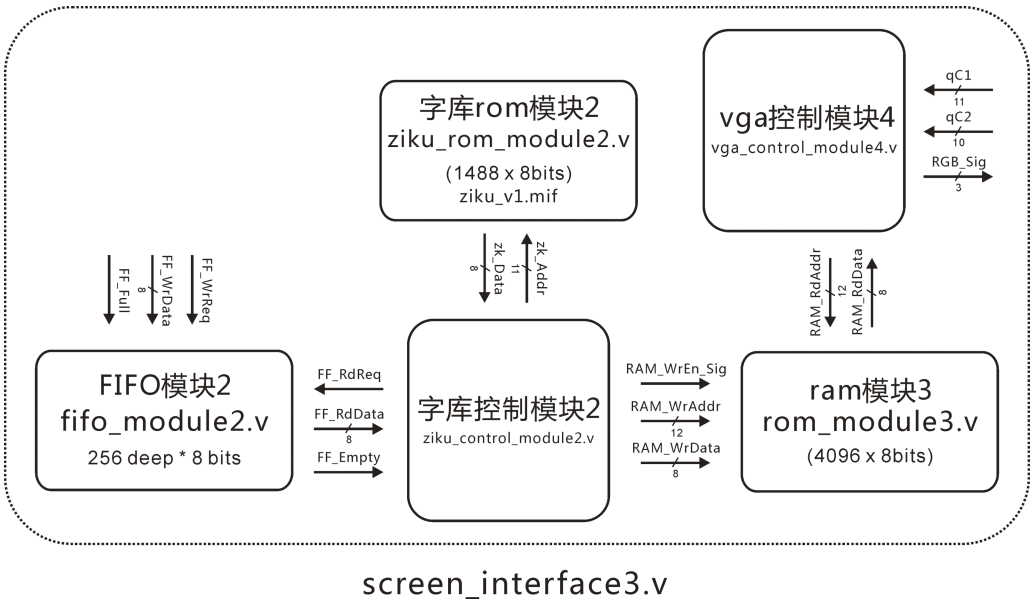


图 2.13b 屏幕接口 3 的建模图。

图 2.13b 是屏幕接口 3 的建模图，和屏幕接口 2 的内容差不多，主要是添加了 FIFO 模块的深度和 ram 模块（图像缓冲空间）的容量（128 * 256 / 8 bits）。此外 vga 控制模块 4 和字库控制模块 2 作了一些修改。具体的内容，我们还是来看代码吧：

vga_control_module4.v

```
module vga_control_module4
(
        input CLK,
        input RSTn,
        input [10:0]qC1,
        input [9:0]qC2,
        output [2:0]RGB_Sig,
        input [7:0]RAM_Data,
        output [11:0]RAM_Addr

);

      /*************************************/

        // Height , width, x-offset and y-offset
        parameter _X = 9'd256, _Y = 9'd128, _XOFF = 10'd0, _YOFF = 10'd160;

        // Character per line
        parameter CPL = 5; // 2^n , 2^5 = 32;

```

```
21.         // Character width in bits
22.         parameter CW = 3; // 2^n, 2^3 = 8;
23.
24.         /************************************/
25.
26.         reg [8:0]x,y;
27.         reg [2:0]n1,n2;
28.         reg [11:0]rAddr;
29.         reg [2:0]rRGB;
30.
31.       always @ ( posedge CLK or negedge RSTn )
32.             if( !RSTn )
33.                     begin
34.                             x <= 9'd0;
35.                             y <= 9'd0;
36.                             n1 <= 3'd0;
37.                             n2 <= 3'd0;
38.                             rAddr <= 12'd0;
39.                             rRGB <= 3'd0;
40.                     end
41.                   else
42.                       begin
43.
44.                             // step 1 : compute data address and index-n
45.                             if( (qC1 > 128 + 88 + _XOFF && qC1 <= 128 + 88 + _XOFF +_X) &&
46.                                   (qC2 > 4 + 23 + _YOFF && qC2 <= 4 + 23 + _YOFF + _Y) )
47.                                   begin
48.                                           x = qC1 - _XOFF - 128 - 88 - 1;
49.                                           y = qC2 - _YOFF - 4 - 23 - 1;
50.                                           rAddr <= (y << CPL) + (x >> CW);
51.                                           n1 <= ( x - ( (x >> 3) << 3) ); // x & 8'b0000_0111
52.                                   end
53.                             else
54.                                   begin n1 <= 3'd0; rAddr <= 11'd17; end
55.
56.                             // step 2 : reading data from rom
57.                             n2 <= n1;
58.
59.                             // step 3 : assign RGB_Sig
60.                             rRGB <= { RAM_Data[n2], RAM_Data[n2], RAM_Data[n2] };
61.
62.                         end
63.
64.       /****************************/
```

```
65.
66.         assign RGB_Sig = rRGB;
67.         assign RAM_Addr = rAddr;
68.
69.         /****************************/
70.
71.   endmodule
```

基本上 vga 控制模块 4 改变不多，在 16 行的_X 常量定义改成 256（256 宽）。第 19 和 22 行是有关屏幕和字符个数的常量声明：第 19 行的 CPL 表示 Character per line，对于 256 个像素宽的屏幕来说，单个字符的宽度用了 8 个像素，所示 256 / 8 = 32，在此常量声明是取 2 幂 5（25）的 5。第 22 行的 CW 表示单个字符的宽度，笔者说设计的初级字库的字符，字符的宽度都是占 8 个像素，因此常量声明中取了 2 幂 3（23）的 3。

第 26 和 28 行的 x，y 和 rAddr 寄存器的宽度随着屏幕的扩张也增加了。此外，重点就是 50 行了，思路基本上和实验十大同小异，亦即 y 针对行，x 针对列。对于 y（行）来说不过是从 1 行 16 * 8bits，变成 1 行 32 * 8bits，或者 1 行 16 Bytes 变成 1 行 32 Bytes。对于 x（列）来说，基本上还是以字节为基础，亦即 8 个列像素等于 1 字节。

不过设计上所针对的目的不同，实验十是针对图像信息，实验十三和实验十四都是针对字库，所以 rAddr 才会采用 (y << CPL) + (x >> CW) 类似的写法。啊！差点忘了，128*256 大小的屏幕，显示的位置是 x-0 y-160，所以第 22 行的 _YOFF 常量修改为 160。

ziku_control_module2.v

```
1.    module ziku_control_module2
2.    (
3.            input CLK,
4.            input RSTn,
5.
6.            input [7:0]FF_RdData,
7.            input FF_Empty,
8.            output FF_RdReq,
9.
10.           output [10:0]zk_Addr,
11.           input [7:0]zk_Data,
12.
13.           output WrEn_Sig,
14.           output [11:0]WrAddr,
15.           output [7:0]WrData
16.
17.   );
18.
```

```
19.         /******************************/
20.
21.           parameter MASK = 8'b1111_1111, ENMASK = 8'b1110_0000;
22.
23.           /******************************/
24.
25.           reg [2:0]i;
26.           reg [4:0]x;
27.           reg [7:0]Point;
28.           reg [11:0]ramAddr;
29.           reg [10:0]zkAddr;
30.           reg isRead;
31.           reg isWrite;
32.
33.           always @ ( posedge CLK or negedge RSTn )
34.                 if( !RSTn )
35.                         begin
36.                                     i <= 3'd0;
37.                                     x <= 5'd0;
38.                                     Point <= 8'd0;
39.                                     zkAddr <= 11'd0;
40.                                     ramAddr <= 12'd0;
41.                                     isRead <= 1'b0;
42.                                     isWrite <= 1'b0;
43.                                 end
44.                 else
45.                     case( i )
46.
47.                             0:
48.                                 if( !FF_Empty ) begin isRead <= 1'b1; i <= i + 1'b1; end
49.
50.                                 1:
51.                                 begin isRead <= 1'b0; i <= i + 1'b1; end
52.
53.                                 2:
54.                                 if( !FF_RdData[7] ) i <= i + 1'b1; // if write character
55.                                 else i <= 3'd6;
56.
57.                                 /****************/ // Write Character
58.
59.                                 3:
60.                                 if( x == 16 ) begin Point <= MASK & (Point + 1'b1); x <= 5'd0; i <= 3'd0; end
61.                                 else begin zkAddr <= (FF_RdData[6:0]<<4 )+ x; i <= i + 1'b1;end
62.
```

```
63.                         4:
64.                         begin isWrite <= 1'b1; ramAddr <= (x << 5) + ((Point>>5)<<9) + (Point & 8'h1f); i <= i + 1'b1; end
65.
66.                         5:
67.                         begin isWrite <= 1'b0; x <= x + 1'b1; i <= 3'd3; end
68.
69.                         /***************/ // Command
70.
71.                         6: // <Enter>
72.                         if( FF_RdData[6:0] == 7'd1 ) begin Point <= ENMASK & ( Point + 5'd16 ); i <= 3'd0; end
73.                         else i <= 3'd0;
74.
75.                     /*********************************************/
76.
77.                 endcase
78.
79.         /************************************/
80.
81.         assign FF_RdReq = isRead;
82.
83.         assign zk_Addr = zkAddr;
84.
85.         assign WrEn_Sig = isWrite;
86.         assign WrAddr = ramAddr;
87.         assign WrData = zk_Data;
88.
89.         /************************************/
90.
91.   endmodule
```

同样 ziku_control_module2.v 和实验十三的 ziku_control_module1.v 之间改变不多。主要是 21 行的 MASK 和 ENMASK 的常量声明，MASK 是用于屏蔽 Point 不超过屏幕可接受的最大字符个数，128 * 256 大小的屏幕可以接受 256 个字符，所以 MASK 也从原本的 8'b0111_1111 变成 8'b1111_111。ENMASK 是用于在 ENTER 操作命令中屏幕 Point，使得 Point 指向第一个位置，128 *256 的屏幕一行可以容纳 32 个字符而且最大字符的个数是 256，所以 ENMASK 也从原来的 8'b0111_0000 变成 8'b1111_0000。

此外，第 64 行也有相关的修改。还记得 Point 的第二和第三功能吗？Point 的第二功能是在“图像映射”的计算中指向同行不同列，由于 128 * 256 的屏幕一行可以容纳 32 个字符，所以 Point & 8'h0f 也修改成 Point & 8'h1f。Point 的第三功能是在“图像映射”的计算中指向不同行，原本 128 * 128 的屏幕每行的偏移量是 256 个字，但是 128 * 256 的屏幕每行的偏移量却是 512 个字，所以 (Point >> 4) << 8 也改成 (Point >> 5)<<9。最后就是 x << 5 也因一行可容纳 32 个字符而改变（具体内容请复习实验十）。

screen_interface3.v

```
1.    module screen_interface3
2.    (
3.            input CLK,
4.            input RSTn,
5.
6.            input [10:0]qC1,
7.            input [9:0]qC2,
8.
9.            input FF_WrReq,
10.           input [7:0]FF_WrData,
11.           output FF_Full,
12.
13.           output [2:0]RGB_Sig
14.   );
15.
16.         /****************************/
17.
18.           wire FF_Empty;
19.           wire [7:0] FF_RdData;
20.
21.           fifo_module2 U1
22.           (
23.                   .clock ( CLK ),
24.                   .wrreq ( FF_WrReq ),      // input - from top
25.                   .data ( FF_WrData ),      // input - from top
26.                   .full( FF_Full ),         // output - to top
27.                   .rdreq ( FF_RdReq ),      // input - from
28.                   .empty ( FF_Empty ),      // output - to
29.                   .q ( FF_RdData )          // output - to
30.           );
31.
32.           /****************************/
33.
34.           wire [7:0] zk_Data;
35.
36.           ziku_rom_module2 U2
37.           (
38.                   .clock( CLK ),
39.                   .address( zk_Addr ),      // input - from U3
40.                   .q( zk_Data )             // output - to U3
41.           );
42.
```

```
43.         /*********************************/
44.
45.         wire FF_RdReq;
46.         wire [10:0]zk_Addr;
47.         wire RAM_WrEn_Sig;
48.         wire [11:0]RAM_WrAddr;
49.         wire [7:0]RAM_WrData;
50.
51.         ziku_control_module2 U3
52.         (
53.                 .CLK( CLK ),
54.                 .RSTn( RSTn ),
55.                 .FF_RdData( FF_RdData ),        // input - from U1
56.                 .FF_Empty( FF_Empty ),          // input - form U1
57.                 .FF_RdReq( FF_RdReq ),          // output - to U1
58.                 .zk_Addr( zk_Addr ),            // output - to U2
59.                 .zk_Data( zk_Data ),            // input - from U1
60.                 .WrEn_Sig( RAM_WrEn_Sig ),      // output - to U4
61.                 .WrAddr( RAM_WrAddr ),          // output - to U4
62.                 .WrData( RAM_WrData )           // output - to U4
63.         );
64.
65.         /*********************************/
66.
67.         wire [7:0]RAM_RdData;
68.
69.         ram_module3 U4
70.         (
71.                 .CLK( CLK ),
72.                 .RSTn( RSTn ),
73.                 .WrEn_Sig( RAM_WrEn_Sig ),    // input - from U3
74.                 .WrAddr( RAM_WrAddr ),        // input - from U3
75.                 .WrData( RAM_WrData ),        // input - from U3
76.                 .RdAddr( RAM_RdAddr ),        // input - from U5
77.                 .RdData( RAM_RdData )         // output - to U5
78.         );
79.
80.         /*********************************/
81.
82.         wire [11:0]RAM_RdAddr;
83.
84.         vga_control_module4 U5
85.         (
86.                 .CLK( CLK ),
```

```
                .RSTn( RSTn ),
                .qC1( qC1 ),                    // input - from top
                .qC2( qC2 ),                    // input - from top
                .RAM_Addr( RAM_RdAddr ),        // output - to U4
                .RAM_Data( RAM_RdData ),        // input - from U4
                .RGB_Sig( RGB_Sig )             // output - to top
            );

            /******************************/

    endmodule
```

不解释了，自己看图 2.13b 吧。

string_generate_module.v

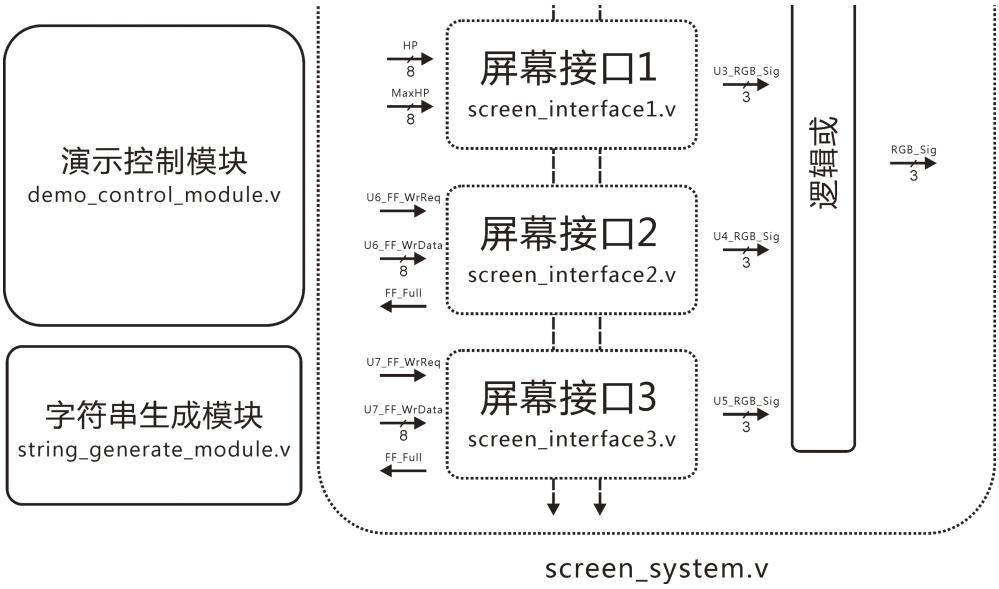


图 2.13c 字符串生成模块的图形。

字符串生成模块主用是用来驱动屏幕接口 3。该模块的里边笔者自定了自己的字符串，内容是：

“Pikachu as a electric's rat”	第一个字符串包含 27 个字符。
“where group-leaving at forest，”	第二个字符串包含 30 个字符。
“fav. food is apple.”	第三个字符串包含 19 个字符。
“Skill is [THUNDER SHOCK] !!!”	第四个字符串包含 28 个字符

对于屏幕接口 3 来说，每一行最大可以容纳 32 个字符。但是对于自定义的字符串来说，也会有自定义的结束字符，亦即每一个字符串的最后一个字符一是结束字符，典型的结

束字符时 '\0'，然而自定义的结束字符 8'hff。具体的内容我们还是来看代码吧：

```
267. module string_generate_module
268. (
269.        input CLK,
270.        input RSTn,
271.
272.        output FF_WrReq,
273.        output [7:0]FF_WrData
274. );
275.
276.        /*************************/
277.
278.        parameter   _a = 8'd1, _b = 8'd2, _c = 8'd3, _d = 8'd4, _e = 8'd5, _f = 8'd6,
279.                    _g = 8'd7, _h = 8'd8, _i = 8'd9, _j = 8'd10, _k = 8'd11, _l = 8'd12,
280.                    _m = 8'd13, _n = 8'd14, _o = 8'd15, _p = 8'd16, _q = 8'd17, _r = 8'd18,
281.                    _s = 8'd19, _t = 8'd20, _u = 8'd21, _v = 8'd22, _w = 8'd23, _x = 8'd24,
282.                    _y = 8'd25, _z = 8'd26;
283.
284.        parameter   _A = 8'd27, _B = 8'd28, _C = 8'd29, _D = 8'd30, _E = 8'd31, _F = 8'd32,
285.                    _G = 8'd33, _H = 8'd34, _I = 8'd35, _J = 8'd36, _K = 8'd37, _L = 8'd38,
286.                    _M = 8'd39, _N = 8'd40, _O = 8'd41, _P = 8'd42, _Q = 8'd43, _R = 8'd44,
287.                    _S = 8'd45, _T = 8'd46, _U = 8'd47, _V = 8'd48, _W = 8'd49, _X = 8'd50,
288.                    _Y = 8'd51, _Z = 8'd52;
289.
290.        parameter   _1 = 8'd53, _2 = 8'd54, _3 = 8'd55, _4 = 8'd56, _5 = 8'd57, _6 = 8'd58,
291.                    _7 = 8'd59, _8 = 8'd60, _9 = 8'd61, _0 = 8'd62;
292.
293.        /*
294.        parameter   _! = 8'd63, _@ = 8'd64, _# = 8'd65, _$ = 8'd66, _% = 8'd67, _^ = 8'd68,
295.                    _& = 8'd69, _* = 8'd70, _( = 8'd71, _) = 8'd72;
296.
297.        parameter   _- = 8'd73, _= = 8'd74, _[ = 8'd75, _] = 8'd76, _\ = 8'd77, _; = 8'd78,
298.                    _' = 8'd79, _, = 8'd80, _. = 8'd81, _/ = 8'd82;
299.
300.        parameter   __ = 8'd73, _+ = 8'd84, _{ = 8'd85, _} = 8'd86, _| = 8'd87, _: = 8'd88,
301.                    _" = 8'd89, _< = 8'd90, _> = 8'd91, _? = 8'd92;
302.        */
303.
304.        parameter ENTER = { 1'b1, 7'd1 };
305.
306.        parameter SPACE = { 1'b0, 7'd0 };
307.
308.        /*****************************************/
```

```
309.
310.        // Pikachu as a electric's rat - 27 words
311.        parameter STRING1 = { _P,_i,_k,_a,_c,_h,_u, 8'd0, _a,_s, 8'd0, _a, 8'd0, _e,_l,_e,_c,_t,_r,_i,_c,8'd79,_s, 8'd0 ,_r,_a,_t, 32'd0, 8'hff };
312.
313.        // where group-leaving at forest, - 30 words
314.        parameter STRING2 = { _w,_h,_e,_r,_e, 8'd0, _g,_r,_o,_u,_p,8'd73,_l,_e,_a,_v,_i,_n,_g, 8'd0, _a,_t, 8'd0, _f,_o,_r,_e,_s,_t,8'd80, 8'd0, 8'hff };
315.
316.        // fav. food is apple. - 19 words
317.        parameter STRING3 = { _f,_a,_v,8'd81, 8'd0, _f,_o,_o,_d, 8'd0, _i,_s, 8'd0, _a,_p,_p,_l,_e,8'd81, 96'd0, 8'hff };
318.
319.        // Skill is [THUNDER SHOCK] !!! - 28 words
320.        parameter STRING4 = { _S,_k,_i,_l,_l, 8'd0, _i,_s, 8'd0, 8'd75,_T,_H,_U,_N,_D,_E,_R, 8'd0, _S,_H,_O,_C,_K,8'd76, 8'd0, 8'd63,8'd63,8'd63, 24'd0, 8'hff };
321.
322.        /*********************************************/
323.
324.        reg [2:0]i;
325.        reg [2:0]Go;
326.        reg [7:0]rData;
327.        reg [255:0]rStr;
328.        reg isWrite;
329.
330.        always @ ( posedge CLK or negedge RSTn )
331.            if( !RSTn )
332.                begin
333.                    i <= 3'd0;
334.                    Go <= 3'd0;
335.                    rData <= 8'd0;
336.                    rStr <= 256'd0;
337.                    isWrite <= 1'b0;
338.                end
339.            else
340.                case( i )
341.
342.                    0:    // Write string1
343.                    begin rStr <= STRING1; i <= 3'd5; Go <= i + 1'b1; end
344.
345.                    1: // Write string2
346.                    begin rStr <= STRING2; i <= 3'd5; Go <= i + 1'b1; end
347.
348.                    2: // Write string3
349.                    begin rStr <= STRING3; i <= 3'd5; Go <= i + 1'b1; end
350.
351.                    3: // Write string4
352.                    begin rStr <= STRING4; i <= 3'd5; Go <= i + 1'b1; end
```

```
353.
354.                    4: // Hold
355.                    i <= i;
356.
357.                    /**********************/ // Write function
358.
359.                    5:
360.                    if( rStr[255:248] == 8'hff ) begin isWrite <= 1'b1; rData <= ENTER; i <= i + 1'b1; end
361.                    else begin isWrite <= 1'b1; rData <= rStr[255:248]; rStr <= rStr << 8; end
362.
363.                    6:
364.                    begin isWrite <= 1'b0; i <= Go; end
365.
366.                    /*****************************************************/
367.
368.                endcase
369.
370.        /****************************/
371.
372.        assign FF_WrReq = isWrite;
373.        assign FF_WrData = rData;
374.
375.        /****************************/
376.
377. endmodule
```

string_generate_module.v 是建立在 demo_control_module.v 和 screen_module.v 之下，所以行数超过 200 行。第 278~306 行是初级字库里所支持的字符的常量声明。第 310~320 行是四个字符串的常量声明，读者需要注意一下，虽然每一个字符的最后一个字符都是自定义的结束字符 8'hff，但是为了实现“向左对齐”，结束字符之前都插入诺干个控制符（每一个字符串都是 32 个字符-包括对齐用的控制符和结束字符）。

第 324~328 行是相关的寄存器声明。第 324 行的 i 是用于步骤；第 325 行的 Go 是用于返回步骤（伪函数）；第 326 行的 rData 是用于驱动 FIFO；第 327 行的 rStr 是用于寄存字符串；第 328 行的 isWrite 同样也是用于驱动 FIFO。

首先所有寄存器都清零（第 332~337 行），第 359~364 行是伪函数，主要是把输入字符串输入 FIFO。步骤 5（第 359~361 行）拉高 isWrite 的同时，也逐个字符输入到 FIFO（第 361 行），前提不是结束字符。当 if 条件判断到结束字符（第 360 行）则向 FIFO 输入 ENTER 操作命令。步骤 6（第 363~364 行）是拉低 isWrite 和返回步骤。

第 342~352 行的步骤 0~3 则调用伪函数（第 332~337 行）将 4 个字符串输入 FIFO 里。步骤 4 是停止操作。

screen_system.v

```
1.    module screen_system
2.    (
3.            input CLK,
4.            input RSTn,
5.
6.            output HSYNC,
7.            output VSYNC,
8.            output [2:0]RGB_Sig
9.    );
10.         /***************************/
11.
12.           wire CLK_40Mhz;
13.
14.           pll_module     U0
15.           (
16.                .inclk0 ( CLK ),             // in 20MHz
17.                .c0 ( CLK_40Mhz )      // output 40Mhz
18.           );
19.
20.           /****************************/
21.
22.           wire [10:0]qC1;
23.           wire [9:0]qC2;
24.           wire U1_HSYNC, U1_VSYNC;
25.
26.           sync_module U1
27.           (
28.                  .CLK( CLK_40Mhz ),           // input - from U0
29.                    .RSTn( RSTn ),
30.                    .qC1( qC1 ),                      // output - to U3
31.                    .qC2( qC2 ),                      // output - to U3
32.                    .HSYNC( U1_HSYNC ),      // output - to bypass register
33.                    .VSYNC( U1_VSYNC )      // output - to bypass register
34.           );
35.
36.           /*******************************/
37.
38.           wire [2:0]U2_RGB_Sig;
39.
40.           screen_module U2
41.           (
42.                  .CLK( CLK_40Mhz ),             // input - from U0
```

```
43.                 .RSTn( RSTn ),
44.                 .qC1( qC1 ),                    // input - from U1
45.                 .qC2( qC2 ),                    // input - from U1
46.                 .RGB_Sig( U2_RGB_Sig )    // output - to top
47.         );
48.
49.         /*********************************/
50.
51.         wire [2:0]U3_RGB_Sig;
52.
53.         screen_interface1 U3
54.         (
55.                 .CLK( CLK_40Mhz ),          / input - from U0
56.                 .RSTn( RSTn ),
57.                 .qC1( qC1 ),                    // input - from U1
58.                 .qC2( qC2 ),                    // input - from U1
59.                 .HP( HP ),                      // input - from U6
60.                 .MaxHP( MaxHP ),                // input - from U6
61.                 .RGB_Sig( U3_RGB_Sig )    // output - to top
62.         );
63.
64.         /************************************/
65.
66.         wire [2:0]U4_RGB_Sig;
67.
68.         screen_interface2 U4
69.         (
70.                 .CLK( CLK_40Mhz ),       // input - from U0
71.                 .RSTn( RSTn ),
72.                 .qC1( qC1 ),                            // input - from U1
73.                 .qC2( qC2 ),                            // input - from U1
74.                 .FF_WrReq( U6_FF_WrReq ),       // input - from U6
75.                 .FF_WrData( U6_FF_WrData ),     // input - from U6
76.                 .FF_Full( ),                            // ignore
77.                 .RGB_Sig( U4_RGB_Sig )          // output - to top
78.         );
79.
80.         /*********************************/
81.
82.         wire [2:0]U5_RGB_Sig;
83.
84.         screen_interface3 U5
85.         (
86.                 .CLK( CLK_40Mhz ),              // input - from U0
```

```
87.                 .RSTn( RSTn ),
88.                 .qC1( qC1 ),                          // input - from U1
89.                 .qC2( qC2 ),                          // input - from U1
90.                 .FF_WrReq( U7_FF_WrReq ),         // input - from U7
91.                 .FF_WrData( U7_FF_WrData ),     // input - from U7
92.                 .FF_Full( ),                          // ignore
93.                 .RGB_Sig( U5_RGB_Sig )            // output - to top
94.             );
95.
96.
97.             /*******************************/
98.
99.             wire [7:0]HP, MaxHP;
100.            wire U6_FF_WrReq;
101.            wire [7:0]U6_FF_WrData;
102.
103.            demo_control_module U6
104.            (
105.                .CLK( CLK_40Mhz ),                // input - from U0
106.                .RSTn( RSTn ),
107.                .HP( HP ),                        // output - to U3
108.                .MaxHP( MaxHP ),                  // output - to U3
109.                .FF_WrReq( U6_FF_WrReq ),     // output - to U4
110.                .FF_WrData( U6_FF_WrData )      // output - to U4
111.            );
112.
113.
114.            /*********************************/
115.
116.            wire U7_FF_WrReq;
117.            wire [7:0]U7_FF_WrData;
118.
119.            string_generate_module U7
120.            (
121.                .CLK( CLK_40Mhz ),                  // input - from U0
122.                .RSTn( RSTn ),
123.                .FF_WrReq( U7_FF_WrReq ),         // output - to U5
124.                .FF_WrData( U7_FF_WrData )       // output - to U5
125.            );
126.
127.            /*********************************/
128.
129.          reg [1:0]rBy1,rBy2,rBy3;
130.
```

```
131.        always @ ( posedge CLK_40Mhz or negedge RSTn )
132.            if( !RSTn )
133.                begin
134.                        rBy1 <= 2'b11;
135.                        rBy2 <= 2'b11;
136.                        rBy3 <= 2'b11;
137.                    end
138.            else
139.                begin
140.                        rBy1 <= { U1_HSYNC, U1_VSYNC };
141.                        rBy2 <= rBy1;
142.                        rBy3 <= rBy2;
143.                    end
144.
145.        /***************************/
146.
147.        assign { HSYNC, VSYNC } = rBy3;
148.        assign RGB_Sig = U2_RGB_Sig | U3_RGB_Sig | U4_RGB_Sig | U5_RGB_Sig;
149.
150.        /***************************/
151.
152. endmodule
```

screen_system.v 基本上和图 2.13a 大同小异，读者自己看着办吧。

==========

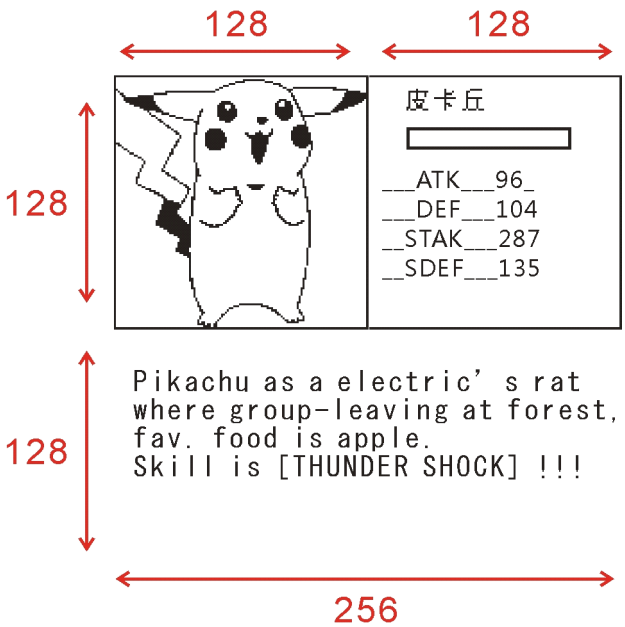


图 2.13d 实验十四的结果。

当实验十四编译成功，并且下载到开发板里。那么图 2.13d 的小可爱，HP 血条，战斗能力表和基本介绍就会出现在屏幕中。最后总结的说，Verilog HDL 语言是非常自由的，就连字符都可以自己定义，而且字符串也可以自己定义。当然实验十四的重点不在这里，从实验十到实验十四所联系的重点，都是在谈“一个时钟，一块数据”概念在设计上的应用。传统的设计方法太被动了，而且也很耗时和精力，因此笔者不得不去找更轻松的方法。没有办法呀，人类的懒惰是天性，又是文明发达的动力，越懒惰的人越有创造力 .... 呵呵呵。

总结：

“时钟是什么？”凡事学单片机的都会回答“节拍”，换之凡是学 FPGA 的都会回答“时钟沿”。读者呢？读者要说时钟提供节拍也行，读者也可以说时钟沿和时钟是等价的无论是“节拍”也好“时钟沿”也好，这是时钟的抽象概念。如果用笔者的话来说，时钟就是模块的心脏或者心跳，时钟的频率决定了周期。

时钟有如心跳那样，提供了模块节拍和动力，但是对于模块的沟通来说，模块不是用心跳或者动力来交流而是信号或者数据。如果要把信号和数据控制好，在什么都没有情况下是很困难的，所以笔者才提起“一个时钟，一块数据”的概念。

读者还记得整合的概念是什么吗？对！就是一种控制时钟和协调模块的写法（这里所说的写法不是指代码级的写法），所以我们需要一个“单位”，这个单位就“一个时钟，一块数据”或者说一个信号的长度是由几个时钟组成，某个信号用被时钟量化就称为时钟化。总之“一个时钟，一块数据”就是给时钟换上另一个“具体的外壳”。

在模块的沟通中“一个时钟，一块数据”充当了“模块语言”一个具体的实在，就像人类的人类的语言，可以用“一个字一个字”去计算那样。然而在设计中，我们应该尽量避免在模块的沟通口中使用“组合逻辑”（即时事件）来驱动，因为这会破快了“沟通的和谐”（实验八的重点）。当某个信号“有标签”和“可计数”的时候，我们就可以任意的跟踪并且同步它（实验九的重点）。

此外，“一个时钟，一块数据”最重要的一个重点就是给予了“主动设计”的概念。传统的设计，由于时钟对于我们来说是一种抽象的存在，所以我们不得不借助仿真软件来观察模块的沟通记录（被动设计）。然而“主动设计”的概念，就是把模块比喻成活生生的人在对话，它们的身份只有“发话者”和“听话者”之分，在设计的时候“设计者只要终于哪个身份去设计哪个模块”，用不着第三者来监督一切（仿真工具）。

因为对于系统级的建模来说，一边仿真一边设计是非常耗时和好精神的，设计者分分钟都会精尽人亡 .... 在此实验十~十四就是为了表达这些信息才存在的（主动设计的基本想法）。

好了，是时候结束第二章了，第二章真的耗了笔者很多的时间，每一个小节的深入“一个时钟，一块数据”的概念不断得以扩展，就这样越写越多了 .... 这一章比读者想象中绝对有趣许多，同时读者最好把之前对 Verilog 的所有认识暂时倒空，不然的话读者的大脑一定短路数次的 ... 呵呵呵。

第三章 理想时序的整合

3.1 理想时钟和实际时钟的关系

时序是什么？估计这个问题笔者已经重复了很多次 ... 如果用 Verilog HDL 语言的话来讲，时序就是模块的沟通记录，又或者是模块的活动记录，然而构成（连接）时序的关键就是时钟，用另一句来说，没有时钟就没有时序。如果是用“物理”的话来讲，时序就是物理路径上的电平变化和信号的时间表现。

在时序篇里笔者说过“Verilog 的时序”理应是理想而且没有瑕疵，反之“物理的时序”充满着破烂和不确定。举个简单的例子，Verilog 的时序有“诺干个时钟的延迟”，然而物理的时序有 1ps~1ns 等的延迟。很多同学常常犯这个概念上的错误“用 Verilog 去迎合物理时序 ... ”

读者知道吗？这是一个很严重的概念陷阱，坏处小则影响设计结果，大则影响日后的学习。一个地基不稳定的高楼，怎能直直站立呢？

就好比“乘法器”我们常常可以从一些硬件设计的参考书里看到：“这个乘法器需要用多少纳秒 ns 和多少 us 才能完成”。这些参考书所说的乘法器，实际上“全都是利用逻辑门”前拼后凑而成，然而逻辑门在 Verilog HDL 语言的定义上，是引起“即时事件”的家伙。换言之，这么大的一个“乘法器”基本上就是一大堆的逻辑门，在 Verilog 的世界里，我们无法给它评估时间，因为“即时事件”是无视时钟的存在。

所以说，这个由逻辑门拼凑出来的“乘法器”在 Verilog 的世界里，求得结果几乎是“瞬间”完成。但是在物理上 FPGA 的每一个逻辑资源，都有“最小”和“最大”的物理延迟（以目前的科技来说是皮秒到纳秒级的范围）。读者尝试想想看看，一个由许多个逻辑门拼凑出来的家伙（乘法器），到底有多少物理延迟呢？呵呵呵，笔者也不知道，因为笔者不是学 ASCI 硬件设计的料。

“物理延迟的时间”对于 Verilog HDL 语言来说是实实在在的“瞎子”，从中我们可以晓得，那些把 Verilog HDL 语言和“物理延迟的时间”强迫联系起来然后又不给于解释的参考书，有如盲人摸象或者黑暗看镜子那样“瞎搞”而已。因为“物理延迟的时间”会随着 FPGA 的工艺，温度，科技，编译器等有所改变，打个比方：一个用 Verilog HDL 语言，并且强调逻辑门所建立的乘法器，会根据不同的 FPGA 产品，延迟结果也会有所不同。

哈哈哈！读者你说好不好笑呢？但是好笑的背后往往都是学习者的怨气和淚气 .....

反而那些看过“时序篇”的朋友，可能又有一番不一样的味道，因为笔者所设计的乘法器不是以实际的时间来计算，而是用“时钟”这个单位来计算。从时序篇开始到整合篇，笔者日夜不停的强调它的重要性，不仅而已，笔者还不停的优化“时钟的概念”和对于

“控制时钟的写法”读者可知笔者为甚如此重视它么？模块所依赖的时钟是设计者唯一可以计算和控制的单位，这“理想时钟 - Ideal Clock”是模块的心脏。

在这里，可能有朋友要问：“难道就没有办法吧 Verilog HDL 语言和物理的延迟时间联系起来吗?”很遗憾，Verilog HDL 语言理应是完美而且理想的，反之物理的延迟时间是破烂不堪而且变化多端。但是天下也不至于有绝人之路，Verilog HDL 语言虽然控制不了物理的延迟时间，但是 Verilog HDL 语言就可以计算和控制实际时间。这就是这第三章节所要讨论和学习的重点。

那么重点来了，要把“Verilog 的理想时钟”和“实际时间”联系起来，其中需要什么“粘糊剂”吗？这个“粘糊剂”就是另一个重点概念，亦即“时钟周期”，理应称为“速率”更为贴切。好了，我们先看看他们之间的关系：

N 个时钟 * 时钟周期 = 实际之间。

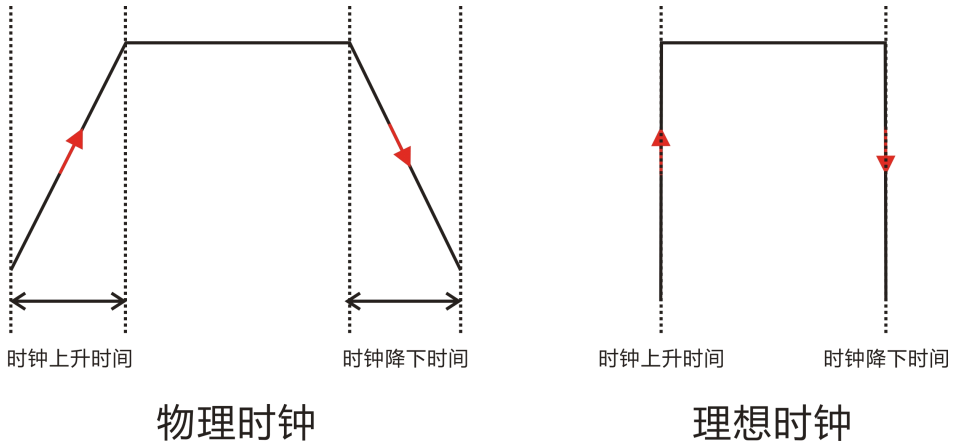


图 3.1a 物理时钟和理想时钟

“N 个时钟”在抽象的认识上是“N 个时钟沿”，但是读者需要明白的是“时钟沿是电平由高变低或者由低变高的瞬间”,这个电平变化的瞬间如果用物理的话来讲就是，时钟升起时间 Time Clock Rise，或者时钟降下时间 Time Clock Fall。如果用 Verilog 的话来说时间沿永远都是理想和瞬间的，如图 3.1a 所示。

在前面笔者已经说过 Verilog HDL 语言对于物理状态（实际时间）的控制完全是一个瞎子，所以在时序篇里，笔者永远强调的是“Verilog HDL 语言是理想而且是完美的”。

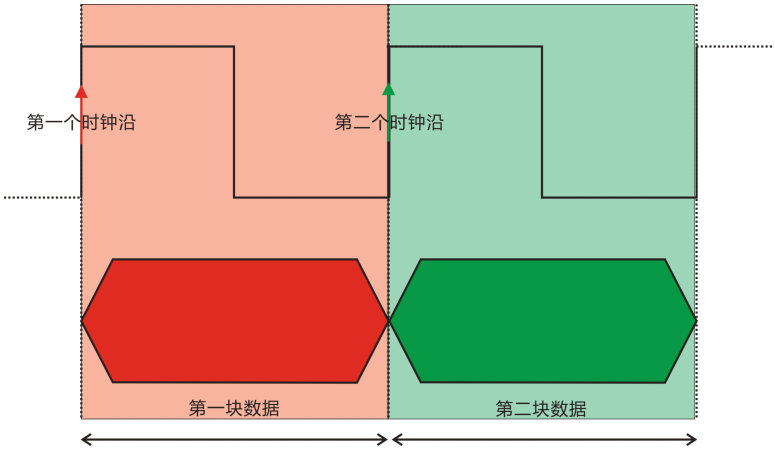


图 3.1b “一个时钟，一块数据”

图 3.1b 是“一个时钟，一块数据”的概念图，是一种用更具体的概念去认识“N 个时钟”。如图 3.1b 所示，第一个上升沿造就了一块红色的数据，然而整合红色图框的区域里，就是第一个“一个时钟，一块数据”。接下来的第二个上升沿造就了一块绿色的数据，哪个绿色图框显示了第二“一个时钟，一块数据”。

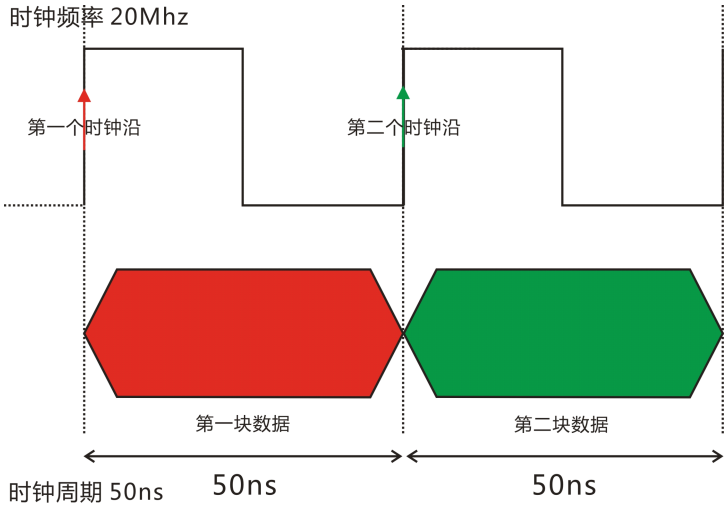


图 3.1c 时钟频率为 20Mhz，时钟周期为 50ns

好了，当“N 个时钟”有具体的概念以后，我们就可以求出实际时间了。图 3.1c 以黑金开发板的时钟频率为例，亦即 20Mhz，所以时钟周期为 50ns。图 3.1c 中有 2 个时钟，所以 2 个时钟的实际时间是 100ns。

2 个时钟 * 50ns 时钟周期 = 100ns

对于设计者来说“时钟周期”是一个固定的常数，它是由时钟频率决定。反之 N 个时钟是一个变数，是一个设计者可以控制的变数。上述的内容说是简单，理解是容易，实际上要随心所欲控制“这个变数”又不过分依赖仿真，设计者唯有在“脑海中就要建立一个清晰而且理想的时序图”。

3.2 速率和步骤的整合

“速率”—这一词，任何学习电子的都会晓得它。“速率”的定义就是数据传输的效率，在传输的定义上“速率”又指“一秒内可以传输多少位数据”，如果继续挖它的底子，说白了“速率”就是“一位数据的周期”，应该说“一位数据所停留的时间”才正确。

如果以“串口传输协议”为例，串口的速率就是波特率，然而波特率决定了一位数据的周期。反之以“SPI 传输协议”为例的话，SCL—串行时钟决定了一位数据的周期。就连 FPGA 自身也有“速率”这东西，就是 CLK—时钟信号，因为一位数据的周期完全是根据 CLK 的时钟频率。（虽然单片机也有时钟源，但是对于用户来说是透明的。）

“时钟化”的基本概念，就是用“较快的速率”（周期较短）去计数“较慢的速率”（周期较长）。但是“速率”又和步骤又有什么关系呢？读者还记得是“循环保留”吗？循环保留就是用“较快的速率”保持 n 个时钟来达到“较慢的速率”的周期。只要明白到这点，那么速率就已经和步骤扯上关系了。

在还没有进入主题之前，笔者需要敲掉的是，单单以“时钟沿”的概念去认识“N 个时钟”结果绝对会撞墙的。原因很简单“时间沿”在 Verilog 的世界里，基本上就是“瞬间”。所以说，用“一个时钟，一块数据”去认识“N 个时钟”才是上上策。

“速率”，“步骤”，“N 个时钟”和“一个时钟，一块数据”具体上是如何整合起来，成为一个和谐的写法呢？我们先来看看简单的例子：

类似的写法（用法）在前面的实验，已经隐隐约约出现很多次了，尤其是实验三|四的串口发送模块，还有实验九的串口接收模块。如实验九—— rx_module.v 中的 64~71 行：

```
64.     2,3,4,5,6,7,8,9: //stalk and count 1~8 data's bit , sample data at 1/2 for bps
65.     begin
66.
67.         if( C1 == SAMPLE ) rData[i-2] <= F2;
68.         if( C1 == BPS115K2 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
69.         else C1 <= C1 + 1'b1;
70.
71.     end
```

rx_module.v 处理并且采集数据，完全是“一个步骤，针对一位数据的跟踪和采集”。又或者实验 3—— tx_module.v 中的 43~45 行：

```
43.     1,2,3,4,5,6,7,8,9,10,11:
44.     if( C1 == B115K2 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
45.     else begin rPin <= rData[i - 1]; C1 <= C1 + 1'b1; end
```

tx_module.v 处理并且发送数据，完全也是“一个步骤，针对一位数据的计数和发送”。

在这里，可能有同学要问：“为什么非把速率和步骤扯上关系？”答案很直接。当笔者在设计某个模块的时候，笔者的要求永远只有一个，就是“提高模块的表达能力”，虽然“建模技巧”已经帮到笔者许多的大忙，但是笔者需要一种更直接，而且可以针对模块内容的写法，或者说一种表达能力强而且更能控制时序的写法。

继续以实验九的 rx_module.v 为例：

首先 rx_module.v 自身的速率（CLK—时钟信号）比起串口传输的速率远远来得高。串口传输协议典型最高的速率是 115200 ，反之 rx_module.v 操作的频率是 20Mhz。经过“时钟化”以后，只要 rx_module.v 计数 174 个时钟就等价与“115200 波特率一位数据的周期”。那么，重点来了 ...

人嘛~就是喜欢把“东西整合”癖好的动物。为了迎合我们身的天性，当我们用 Verilog HDL 语言控制时序（更贴切说是时钟）的时候，应该也考虑学会把“某位数据的操作，整合与某个步骤”，就好比 rx_module.v 其中的 64~71 行：

```
64.                 2,3,4,5,6,7,8,9: //stalk and count 1~8 data's bit , sample data at 1/2 for bps
65.                 begin
66.
67.                     if( C1 == SAMPLE ) rData[i-2] <= F2;
68.                     if( C1 == BPS115K2 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
69.                     else C1 <= C1 + 1'b1;
70.
71.                 end
```

从外部进来的一帧串口数据是由 11 位由 174 个时钟的数据组合而成。其中 rx_module.v 要的就是 [1..8] 的数据位。步骤 2~9 的存在，就是把“每一位数据位的处理，都整合为每一个步骤”。当我们可以把一个“速率”——一位数据的周期，“整合”到某一个步骤里，无论是内容的直接性，表达性，还是美观性上都是按“指数”提升。

上诉的例子只是告诉读者，如何把“速率”的处理整合在“步骤”里而已。事实上这也是第三章的冰上一角而已。究竟笔者还要表达什？不怕不怕，我们从实验中了解吧。

实验十五：IIC 实验（芯片—24LC04 驱动）

iic 实验对于笔者来说真是有爱有恨，早期为了实现用 Verilog HDL 语言驱动 24LC04 还留下不少眼泪和鼻涕。芯片——24LC04 是一片用 IIC 协议的 EEPROM，读者知道 IIC 协议的故事吗？我们姑且先大概的知道 iic 总线到底是什么。

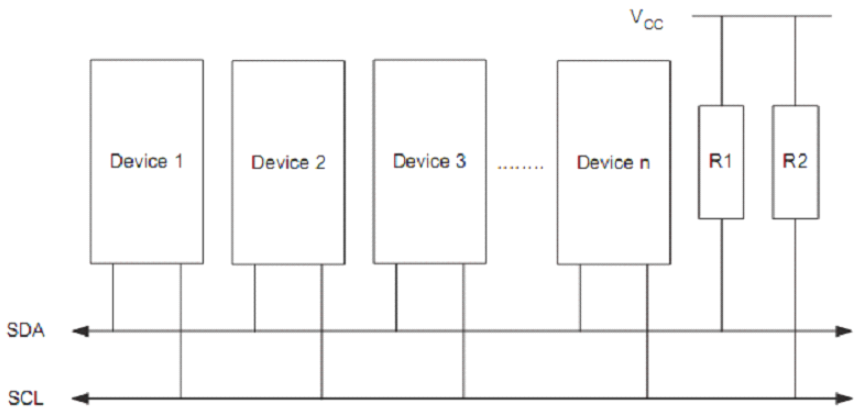


图 3.2a iic 总线概念图。

iic 协议更贴切说是 iic 总线（iic 另一个别名是 twi）。该总线有两条信号 SDA 和 SCL，SDA 信号是双向（三态）的串行数据，反之 SCL 信号时单向的串行时钟。iic 总线同样也有主机和从机之分，只要谁（使能）控制 SCL 信号，谁就是老大（主机），其它的都是小弟（从机），如图 3.2a 所示。

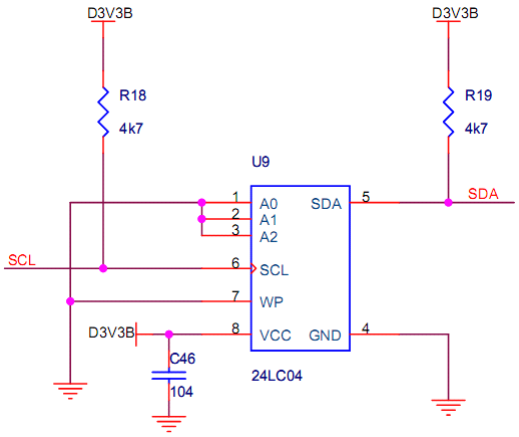


图 3.2b 设备地址（硬件地址）。

任何 iic 的从机设备都有设备地址（硬件地址）。如图 3.2b 所示，其中的 A0~A2 就是设备地址了，在这里，图 3.2b 中的设备地址是 A0~A2 — 3'b000（接地关系）。此外设备地址是 2 的 3 次幂（2^3），这也表示 iic 总线里最大的从机（设备）数是 8（但是这是以前的故事了，未来 iic 总线应该所升级吧，想知道详情的朋友自己去维基一下就行了）。

iic 总线也有“速率”这一东西，普通模式的话速率是 100kb/s ，高速模式的话速率是 400kb/s（这里的 b 是 bit 而不是 Bytes），也就是说普通模式下 一位数据的周期是 10us，反之高速模式下 一位数据的周期是 2.5us。读者只要搞明白了这个重点以后，那么 iic 设备的驱动就会变得很简单。

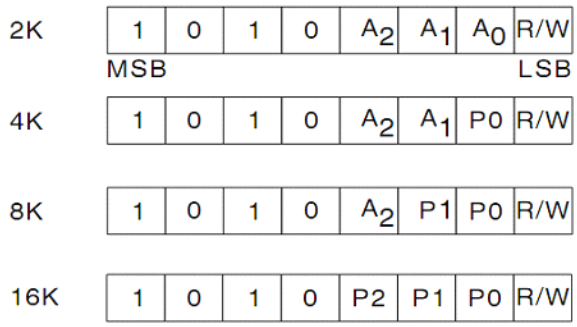


图 3.2c 设备（硬件）地址

好了，接下来笔者要谈的就是 24LC04 者芯片本身。芯片 24LCxx 后边的 xx 表示了这个 eeprom 的内部量，02 为 2k，04 为 4k，08 为 8k，其他以此类推。黑金开发板所包含的是 24LC04 亦即 4k。图 2.7c 所示的是 iic 设备的设备（硬件）地址。

[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
厂商特定的产品 iic 设备地址				iic 设备在总线上的地址			R/W

表 3.2a 设备地址。

[7..4]是厂商特定的产品的设备地址，基本上是固定的 .... 24LCxx 系列产品是 4'b1010, [3..1]会根据设备本身的 A2~A0 引脚配置而改变。[0]是，当主机访问 iic 总线上的任何设备（从机）时，[0]为逻辑 1 表示主机读设备（从机），[0]为逻辑 0 表示主机写设备（从机）。

好了，现在笔者先用一个比较“傻瓜”的方法去解释芯片 24LC04 到底是如何驱动？

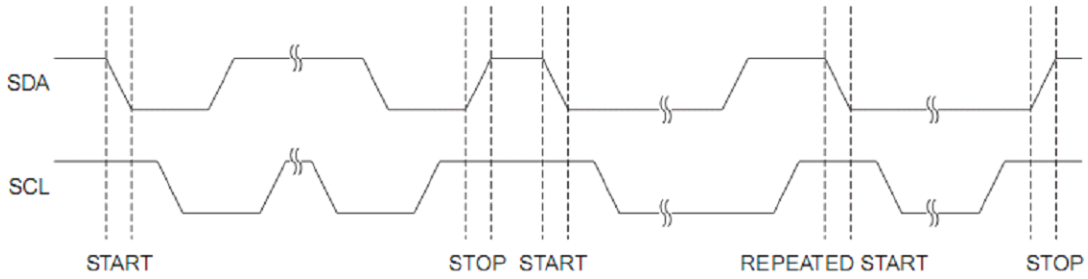


图 3.2d 起始信号和结束信号

主机和从机在 iic 总线上沟通的时候，它们之间有用类似“你好”（起始信号）和“再见”（结束信号）来建立一次又一次的沟通。如图 3.2d 所示，当某个主机要对从机访问之前，主机必须先在 iic 总线上广播（发送）“起始信号”以便表示“各位小弟给我听好！老大我要发号了”当总线上的小弟们（从机），聆听（接收）到“起始信号”，小弟们便就绪好，看看谁到底会被老大点到。起始信号的产生过程很简单，只要在 SCL 拉高之际，使 SDA 信号由高变低即可。

假设 iic 总线是普通速率，亦即 100k/b，那么我们会这样理解，以 20Mhz 频率为例，100kHz 的量化结果是 200：

(1/100k) / (1/20M) = 200

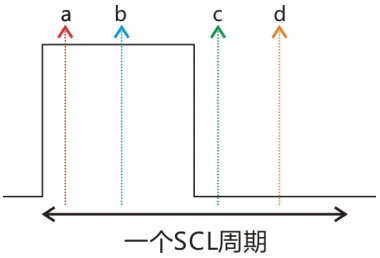


图 3.2e 一个 SCL 周期分成 4 个关键部分（网上惯用的方法）。

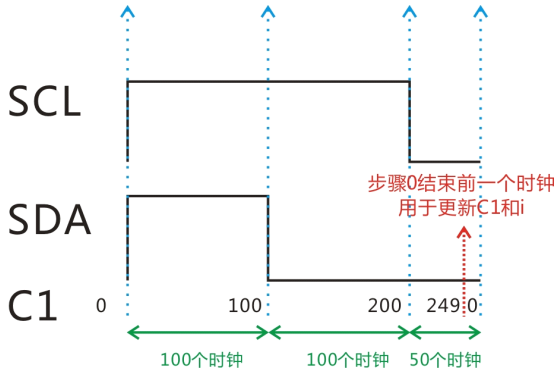
在网上，很多人都“習慣”建立一個計時器來確定 SCL 信號的產生，隨後又把一个 SCL 周期分成 4 个关键部分，如图 3.2e 所示。关键部分 a 是 SCL 信号上升沿后不久；关键部分 b 是 SCL 信号保持在高电平最中间（最稳定）的时候；关键部分 c 是 SCL 信号下降沿后不久；关键部分 d 是 SCL 信号处于低电平。這個“習慣”虽然很普通，但是寫法的伸縮性不好而且扩展性又有限 ... 所以笔者尝试了另一个更直接的写法。

在整合的写法中，信号 .... 应该说每一个“速率”或者每一个“动作”（操作）都是整合在一个“步骤”来控制，于是我们可以用 Verilog HDL 语言这样写：

```
1.      0: // Start
2.      begin
3.          if( C1 == 0 ) rSCL <= 1'b1;
4.          else if( C1 == 200 ) rSCL = 1'b0;
5.
6.          if( C1 == 0 ) rSDA <= 1'b1;
7.          else if( C1 == 100 ) rSDA <= 1'b0;
8.
9.          if( C1 == 250 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
10.         else C1 <= C1 + 1'b1;
11.     end
```

上诉的写法使用了第一章的整合思路，就是把 C1 等计数器整合在步骤 0 里。然而在理解方面使用了第二章的“一个时钟，一块数据”，因为 100kHz 的“速率”是由 200 个时钟的长度来决定。

第 9~10 行表示这个“步骤”消耗 250 个时钟，其中第 9 行的 -1 表示 C1 和 i 的变更发生在步骤 0 结束前一个时钟。第 3~4 行是用来控制 SCL 信号，第 3 行表示了 C1 等于第 0 个的时候拉高 SCL 信号，当 C1 等于第 200 个时钟的时候则拉低 SCL 信号(第四行)。第 6~7 行是用来控制 SDA 信号，当 C1 等于第 0 个时钟的时候拉高 SDA 信号，当 C1 计数到第 100 个时钟的时候而拉低 SDA 信号。



如图 3.2f 所示 步骤 0 产生起始信号。

在这里，有可能有同学要问“100Khz 的速率，经过 20Mhz 时钟的量化应该是 200 个时钟才对嘛？为什么多 50 个时钟来呢？（注意~第 9 行是 250 而不是 200）”。如图 3.2f 所示最后 50 个时钟是“保险作用”没有什么特别。至于上述代码发生的经过，图 3.2f 已经解释一切了（图 3.2f 必须用第二章所学到的概念去理解）。

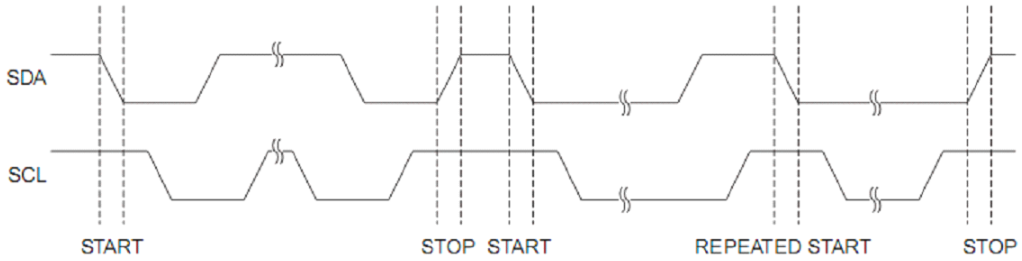


图 3.2d 起始信号和结束信号

“结束信号”对于 iic 的主从设备来说有如“我完成了”或者“结束操作 Over”等，目的是为了结束“一次性”的沟通。结束信号的产生是在 SCL 拉高之际，SDA 信号由低变高。至于 Verilog HDL 语言的实现，我们可以用同样的写法。

```
1.      4: // Stop
2.      begin
3.
4.          if( C1 == 0 ) rSCL <= 1'b0;
5.          else if( C1 == 50 ) rSCL <= 1'b1;
6.
7.          if( C1 == 0 ) rSDA <= 1'b0;
8.          else if( C1 == 100 ) rSDA <= 1'b1;
9.
10.         if( C1 == 200 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
11.         else C1 <= C1 + 1'b1;
12.     end
```

第 10~11 行表示步骤 4（结束信号的步骤）消耗 200 个时钟，其中第 10 行的 -1 则表示

C1 和 i 被更新时发生在步骤结束前的一个时钟。第 4~5 行是用来控制 SCL，反之第 7~8 行是用来控制 SDA。在 C1 计数到底 0 个时钟 SCL 信号被拉低（第 4 行），直到 C1 计数到第 50 个时钟 SCL 信号才被拉高（第 5 行）。换做是 SDA 信号，当 C1 计数到第 0 个时钟的时候被拉低，则 C1 计数到第 100 个时钟的时候 SDA 被拉高。

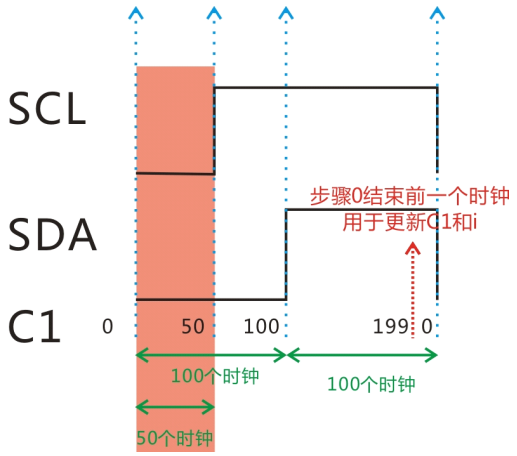


图 3.2g 结束信号产生的过程。

图 3.2g 已经说明了，一切皆发生步骤 4 的升上。读者你知道吗，iic 主从设备每一次性的沟通最关键的部分就是在于前 50 个时钟（C1 计数 0~50 之前），一个 iic 的设计稳不稳定就是发生在“红色矩形”这个地方。至于是什么原因它那么重要呢，笔者先卖个关子。在此，读者只需要直到“红色矩形”是关键的部分就是了。

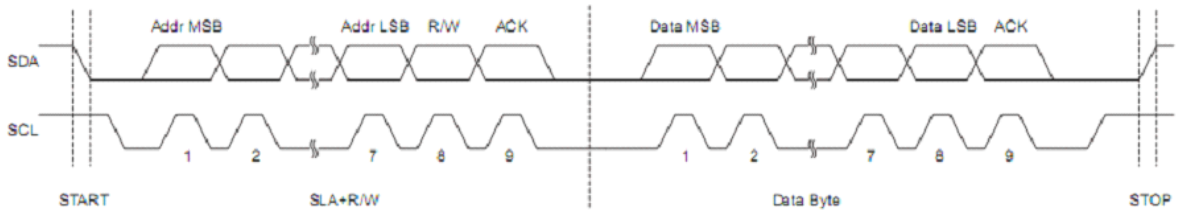


图 3.2h iic 总线的数据传输。

图 3.2h 是 iic 主从设备在 iic 总线上传输数据的时序图。如果某个主机要和从机沟通，基本上主机先在 iic 总线上广播（发送）起始信号，然后所有 iic 总线上的从机就竖着耳朵看看谁被点到。主机会先发送一字节“认人”的数据，也就是一字节“设备地址”数据，至于主机是访问的方向（是读还是写）则由设备地址的最小位（LSB）来作决定。

假设哪一个不幸的从机小家伙被主机老大“叫到”，这位从机需要“用一个速率”来应答（Acknowledge）主机，这好比“老大你叫我吗？”。当主机老大听到应答以后，那么它可以任意使唤这位从机小家伙了（读数据还是写数据）。随便一提，如果主机老大得到的应答位的结果是逻辑 0 表示它点到人了，反之它点空了。

我们先不管 iic 的主机是如何向从机写操作还是写操作，点到人还是点不到人。在此读者只要明白“每一节数据数据的传输都会随着一位应答的出现”就够了。

好了，我们可以用 Verilog HDL 语言写一个“发送一个字节”的紧接着处理一位应答信号的伪函数。在理论上“速率”是指“数据在传输中所保留的时间”，继续以 100khz 的"速率”为例，然后配合起始信号和解释信号的写法，亦即“每一位数据都整合在一个步骤里处理”，当然“应答位的处理也是整合在一个步骤里”。

```
1.          7,8,9,10,11,12,13,14:
2.          begin
3.                  isOut = 1'b1;
4.                  rSDA <= rData[14-i];
5.
6.                  if( C1 == 0 ) rSCL <= 1'b0;
7.                  else if( C1 == 50 ) rSCL <= 1'b1;
8.                  else if( C1 == 150 ) rSCL <= 1'b0;
9.
10.                 if( C1 == F100K -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
11.                 else C1 <= C1 + 1'b1;
12.         end
```

第 1 行中的步骤 7~14 分别是 8 个步骤，每一个步骤则针对每一位数据。第 3 行表示 SDA 是输出（关于‘=’往后再讲）。第 10~11 行表示每一个步骤都消耗 200 个时钟，其中第 11 行的 -1 表示 C1 和 i 的更新发生在该步骤结束前的一个时钟。第 4 行是为 SDA 赋值的动作，iic 是从最高位（MSB）开始。第 6~8 行是用来控制 SCL 信号，在 C1 计数到第 0 个时钟的时候拉低 SCL 信号；在 C1 计数到第 50 个时钟的时候拉高 SCL 信号；在 C1 计数到第 150 个时钟的时候则拉低 SCL 信号。

好吧，笔者还是用图画来说话吧 ....

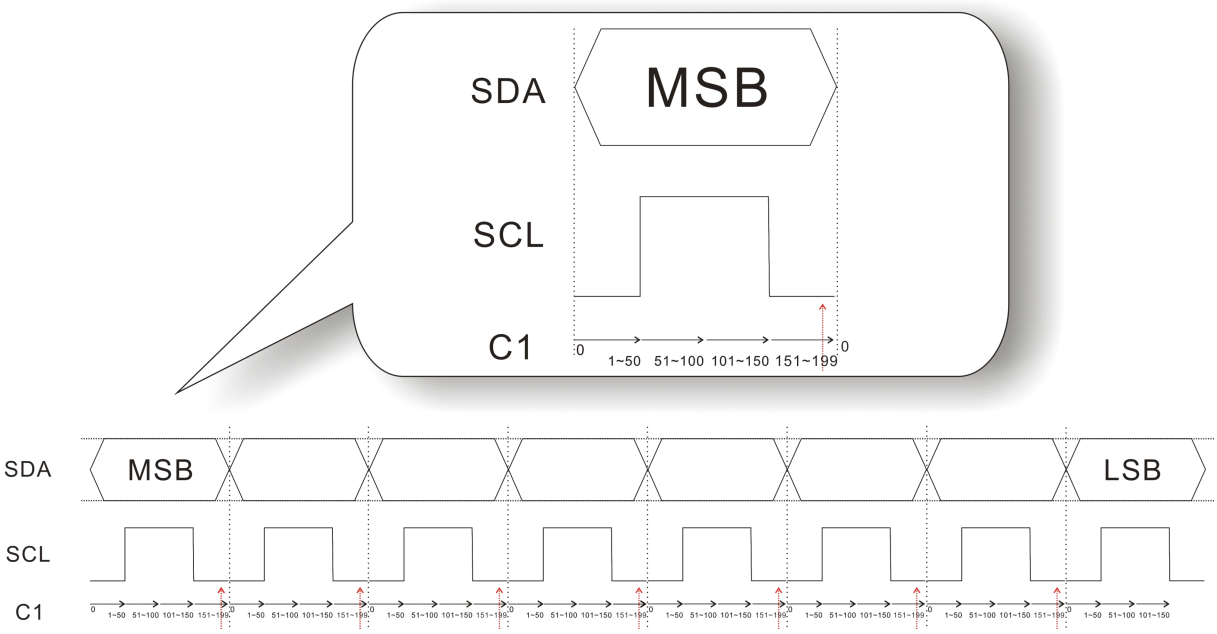


图 3.2i 一字节数据发送的过程。

俗语说得好：一图胜千言，一个真心的拥抱生动过说一万个我爱你。图 3.2i 已经把一切都解释透了，笔者就姑且添翼点睛吧。图 3.2i 中是表示一位数据发生的经过，其中一位数据停留 200 个时钟，或者说一位数据有 200 个时钟的长度。至于那个红色的小箭头则表示，在步骤结束的前一个时钟用来更新 C1 和 i。当步骤 7~14 连续发生，那么就会产生出如图 3.2i 所示的时序图。

读者是不是觉得很感叹呢？这写法也未免太直接了吧？好戏还在后头，继续用同样的写法实现应答位的处理：

```
1.      15: // waiting for acknowledge
2.      begin
3.          isOut = 1'b0;
4.          if( C1 == 100 ) isAck <= SDA;
5.
6.            if( C1 == 0 ) rSCL <= 1'b0;
7.            else if( C1 == 50 ) rSCL <= 1'b1;
8.            else if( C1 == 150 ) rSCL <= 1'b0;
9.
10.           if( C1 == 200 -1 ) begin C1 <= 8'd0; i <= Go; end
11.           else C1 <= C1 + 1'b1;
12.     end
```

第 10~11 行表示这一个步骤消耗 200 个时钟，其中 -1 是表示在该步骤结束前的一个时钟更新 C1 和 i（暂时先无视那个 Go）。步骤 3 是用来控制 SDA 是输入和输出，同样也是暂时先无视，在这里读者只要明白主机处理应答位是读操作就可以了。第 4 行表示，当 C1 计数到第 100 个时钟的时候读取应答位。第 6~8 行足够华丽了，主要是用来拉低又拉高 SCL 信号。当 C1 计数到第 0 个时钟就拉低 SCL 信号；当 C1 计数到第 50 个时钟就拉高 SCL 信号；当 C1 计数到第 150 个时钟就拉低 SCL 信号。

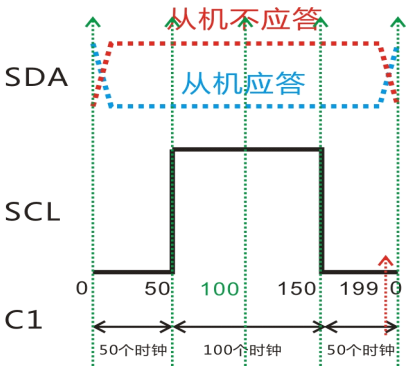


图 3.2j 应答位的处理。

上面笔者已经说过，每当主机传输一字节数据以后，随着会有一个从机的应答位。在此，SCL 依然是由主机控制，但是 SDA 数据线的方向变成输入，亦即从机反馈应答位，而主机负责接收。图 3.2j 说明了主机是如何处理应答位，当 C1 计数为 0 时就拉低 SCL；50 时拉高 SCL 信号；100 时读取应答位；150 时拉低 SCL 信号；总结，应答位也是“速

率”的其中一伙，因为应答位的处理也消耗一个时钟周期（以 100khz 的速率为例，经 20Mhz 时钟的量化是 200 个时钟的长度）。

在这里，基本内容已经充足了，接下来我们来讨论芯片 24LC04 如何实现写/读操作。

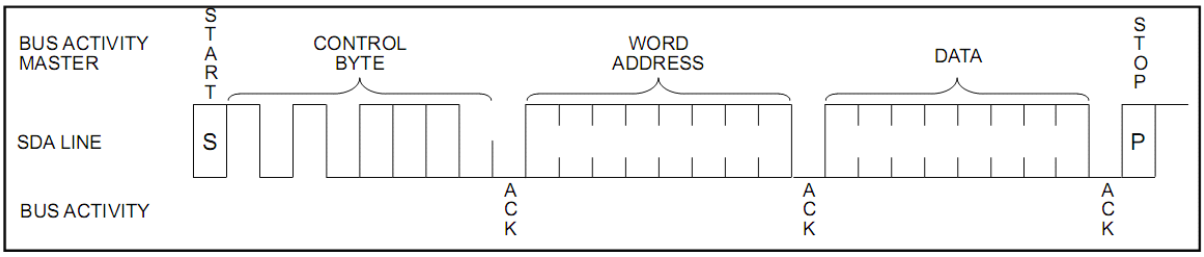


图 3.2k 芯片 24LC04 的写操作。

不知道读者有没有玩过“拼图”的游戏？图 3.2k 的写操作如同一个拼图游戏一样，笔者把所有操作都整合在自身的“步骤”里，要 24LC04 的写操作也随之简单起来。我们只要像玩拼图一样，按着“次序”把每一个“步骤”（拼图），拼接起来 .... 如图 3.2k 所示：

（一）首先我们按下起“始信号的拼图（步骤）”；
（二）然后按下“写一个字节的拼图（步骤）”，字节的内容是设备地址；
（三）接着按下“应答位的拼图（步骤）”；
（四）随后我们又按下另一个“写一字节的拼图（步骤）”，字节的内容是字地址；
（五）然后再按下“应答位的拼图（步骤）”；
（六）跟着就是按下“写一字节的拼图（步骤）”，字节内容是写入信息；
（七）随之按下一个“应答位的拼图（步骤）”；
（八）最后按下“结束位的拼图（步骤）”；

就这样芯片 24LC04 的写操作拼图完成了。图 3.2k 的关键就是次序（一）的字节内容，笔者说过设备地址的最低位决定了主机对从机的访问方向。24LC04 在写操作中，第一字节内容就是主机老大的“点人程序”，也是写方向，所以设备地址的最小位是逻辑 0 了。 余下的关键就是次序（三），（五）和（七）的应答位，如图 3.2k 所示，应答位的内容都是逻辑 0，亦即从机小弟响应主机老大，那么它们之间的沟通才能继续下去。

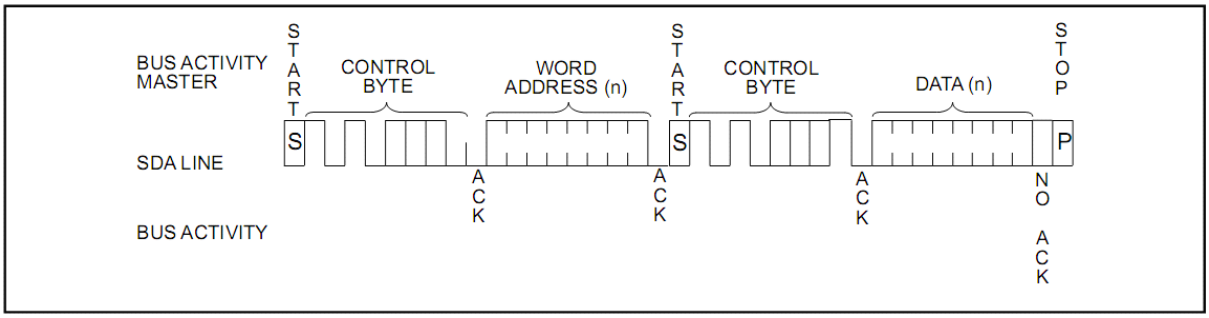


图 3.2l 芯片 24LC04 的读操作。

按照这样的思路继续实现 24LC04 的读操作。

（一）首先按下“起始信号的拼图（步骤）”；
（二）接下来按下“写下一字节的拼图（步骤）”，字节内容是设备地址；
（三）随之是按下“应答位的拼图（步骤）”；
（四）然后又是按下“写一字节的拼图（步骤）”，字节内容是字地址；

iic 设备的沟通有一个坏习惯，假设主机向从机读数据，首先主机需要写设备地址，既然是“写”设备地址的方向当然也是写，所以设备地址的最低位是逻辑 0，主机顺便“焦距”好要读哪一个信息（写字地址）。好了，主机点人完毕以后，要读取的信息也确定了，这回要向从机“转换方向”，主机需要重新发送一个起始信号，作为“转换方向”的“提醒”（如图 3.21 所示）。紧接着写一字节设备地址，然而设备地址的最低位是逻辑 1，亦即主机向从机要求“读”。

（五）跟着是按下一个“起始信号的拼图（步骤）”，作为“转换方向”的提醒；
（六）接下来又是按下一个“写一个字节的拼图（步骤）”，字节内容是设备地址，但是方向改变了。
（七）随之是按下“应答位的拼图（步骤）”；
（八）然后按下一个“读一字节的拼图（步骤）”；
（九）随后接着按下一个“应答位的拼图（步骤）”，这时候应答位的内容可以无视或者 Don't Care；
（十）最后按下一个“结束信号的拼图（步骤）”；

关于芯片 24LC04 的读操作和写操作，拼图（步骤）的内容基本上是大同小异，不同的只是读操作中次序（九）的“读一个字节”。

```
1.  19,20,21,22,23,24,25,26: // Read
2.  begin
3.      isOut = 1'b0;
4.      if( C1 == 100 ) rData[26-i] <= SDA;
5.
6.        if( C1 == 0 ) rSCL <= 1'b0;
7.        else if( C1 == 50 ) rSCL <= 1'b1;
8.        else if( C1 == 150 ) rSCL <= 1'b0;
9.
10.       if( C1 == F100K -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
11.       else C1 <= C1 + 1'b1;
12. end
```

第 1 行的步骤 19~26 表示读一个字节是由 8 个步骤组成。第 10~11 行表示每一个步骤都消耗 200 个时钟，或者说每一个步骤都停留 200 个时钟，其中的 -1 表示在该步骤结束前的一个时钟更新 C1 和 i。第 3 行是表示 SDA 处于读取状态。第 4 行是表示当 C1 计数到 100 的时候（数据最稳定的时候），从 SDA 读取数据到 rData 寄存器（MSB 开始）。第 6~8 行是针对 SCL 信号的操作，当 C1 计数到第 0 时钟的时候拉低 SCL；第 50 个时钟的时候拉高 SCL；第 150 个时钟的时候拉低 SCL。

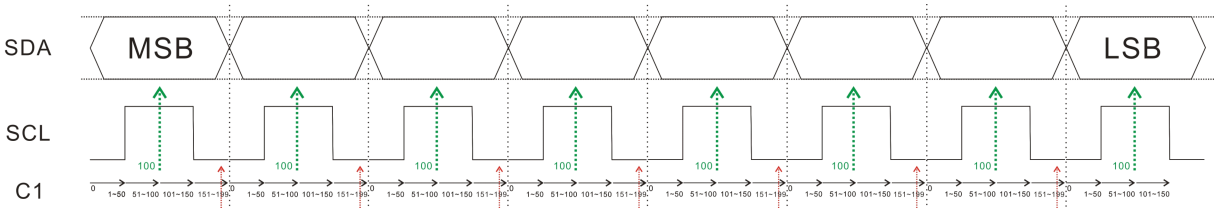


图 3.2m 一字节读取过程。

如图 3.2m 所示，步骤 18~25 联合起来就会成为图 3.2m 等如此壮观的概念图。笔者和读者也一样，同样是一个“迷信”的家伙，我们都相信数据处于正中间的时候才是最稳定的，事实上这是心理作用而已，如果用理想时序的话来讲，任何时候数据都是稳定的；如果用物理时序的话来讲，只要过了 Tsetup 和 Thold 数据保存是稳定的（关于物理时序听不懂不要紧，笔者随口说说而已）。

到目前为止，关于 iic 总线和芯片 24LC04 的拼图我们已经说了七七八八了，是时候进入建模的正题了。

iic_func_module.v

图 3.2n iic 功能模块。

图 3.2n 就是我们要建立的功能模块，这个功能模块用了命令式仿顺序操作去建立。
图 3.2n 的功能模块样子上都很眼熟，其中 Start_Sig 表示它有两个“内部功能”。至于 Addr_Sig 是指 Word Address（字地址）而不是设备地址，设备地址已经嵌入在里边了。具体的内容，我们还是来看代码吧：

```
module iic_func_module
(
        input CLK,
        input RSTn,

        input [1:0]Start_Sig,
        input [7:0]Addr_Sig,
        input [7:0]WrData,
        output [7:0]RdData,
        output Done_Sig,

        output SCL,
```

```
13.         inout SDA
14.
15.         /***************/
16.         //output [4:0]SQ_i
17.
18.   );
19.
20.         /*************************/
21.
22.         parameter F100K = 9'd200;
23.
24.         /*************************/
25.
26.         reg [4:0]i;
27.         reg [4:0]Go;
28.         reg [8:0]C1;
29.         reg [7:0]rData;
30.         reg rSCL;
31.         reg rSDA;
32.         reg isAck;
33.         reg isDone;
34.         reg isOut;
35.
36.         always @ ( posedge CLK or negedge RSTn )
37.               if( !RSTn )
38.                     begin
39.                             i <= 5'd0;
40.                             Go <= 5'd0;
41.                             C1 <= 9'd0;
42.                             rData <= 8'd0;
43.                             rSCL <= 1'b1;
44.                             rSDA <= 1'b1;
45.                             isAck <= 1'b1;
46.                             isDone <= 1'b0;
47.                             isOut <= 1'b1;
48.                         end
49.                 else if( Start_Sig[0] )
50.                     case( i )
51.
52.                             0: // Start
53.                             begin
54.                                     isOut = 1;
55.
56.                                     if( C1 == 0 ) rSCL <= 1'b1;
```

```
57.                     else if( C1 == 200 ) rSCL = 1'b0;
58.
59.                     if( C1 == 0 ) rSDA <= 1'b1;
60.                     else if( C1 == 100 ) rSDA <= 1'b0;
61.
62.                     if( C1 == 250 -1) begin C1 <= 9'd0; i <= i + 1'b1; end
63.                     else C1 <= C1 + 1'b1;
64.                 end
65.
66.                 1: // Write Device Addr
67.                 begin rData <= {4'b1010, 3'b000, 1'b0}; i <= 5'd7; Go <= i + 1'b1; end
68.
69.                 2: // Wirte Word Addr
70.                 begin rData <= Addr_Sig; i <= 5'd7; Go <= i + 1'b1; end
71.
72.                 3: // Write Data
73.                 begin rData <= WrData; i <= 5'd7; Go <= i + 1'b1; end
74.
75.                 /************************/
76.
77.                 4: // Stop
78.                 begin
79.                     isOut = 1'b1;
80.
81.                     if( C1 == 0 ) rSCL <= 1'b0;
82.                     else if( C1 == 50 ) rSCL <= 1'b1;
83.
84.                     if( C1 == 0 ) rSDA <= 1'b0;
85.                     else if( C1 == 150 ) rSDA <= 1'b1;
86.
87.                     if( C1 == 250 -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
88.                     else C1 <= C1 + 1'b1;
89.                 end
90.
91.                 5:
92.                 begin isDone <= 1'b1; i <= i + 1'b1; end
93.
94.                 6:
95.                 begin isDone <= 1'b0; i <= 5'd0; end
96.
97.                 /******************************/ //function
98.
99.                 7,8,9,10,11,12,13,14:
100.                begin
```

```
101.                                isOut = 1'b1;
102.                                rSDA <= rData[14-i];
103.
104.                                if( C1 == 0 ) rSCL <= 1'b0;
105.                                else if( C1 == 50 ) rSCL <= 1'b1;
106.                                else if( C1 == 150 ) rSCL <= 1'b0;
107.
108.                                if( C1 == F100K -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
109.                                else C1 <= C1 + 1'b1;
110.                        end
111.
112.                        15: // waiting for acknowledge
113.                        begin
114.                                isOut = 1'b0;
115.                                if( C1 == 100 ) isAck <= SDA;
116.
117.                                if( C1 == 0 ) rSCL <= 1'b0;
118.                                else if( C1 == 50 ) rSCL <= 1'b1;
119.                                else if( C1 == 150 ) rSCL <= 1'b0;
120.
121.                                if( C1 == F100K -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
122.                                else C1 <= C1 + 1'b1;
123.                        end
124.
125.                        16:
126.                        if( isAck != 0 ) i <= 5'd0;
127.                        else i <= Go;
128.
129.                        /*******************************/ // end function
130.
131.                endcase
132.
```

iic_func_module.v 的代码看是行数非常多，但只是“重复性的拼图而已”，笔者先解释上部分。第 3~13 行是该模块的输入输出口，基本上和“图形”没有什么两样。第 16 行是仿真用的输出，笔者把它注释掉了，暂时无视它吧。第 22 行是 iic 总线“速率”为 100KHz 经过 20Mhz 量化过后的结果。

第 26~34 行是相关的寄存器，i 是步骤用；Go 是伪函数的返回用；C1 是计数时钟用；rData 是数据暂存和驱动用；rSCL 是驱动 SCL 用；rSDA 是驱动 SDA 用；isAck 是暂存应答结果；isDone 是反馈完成信号用；isOut 用来决定 SDA 是输出还是输入。第 39~47 行是相关寄存器的复位操作，其中 isOut 初值为逻辑 1，则表示 SDA 在初始状态是输出；rSCL 和 rSDA 初值为 1，亦即 SCL 和 SDA 信号在“空闲时”处于高电平；isAck 初值为 1，默认状态而已，没有什么特别。

第 49 行 if 条件的 Start_Sig[0] 表示“写字节操作”，第 50~131 是该功能的全部操作。我们先看第 99~127 行的伪函数，第 99~110 行是处理“写一个字节的拼图”，第 112~123 行是处理“应答位的拼图”，至于第 125~127 行是用来判断“应答位是否为逻辑 0”。如果应答位的结果不为 0 的话（从机不应答）就重新操作（第 126 行），否则返回步骤（第 127 行）。笔者在前面已经很详细解释“写一个字节”和“应答位”等拼图的功能了，因此笔者就不重复了。

好了，先在读者把头跳到步骤 0（第 52~64 行）。当第 49 行检测到 Start_Sig[0] 拉高，该模块会先发送一个“起始信号”（第一块拼图），其中 54 行的 isOut = 1 表示，SDA 为输出状态。在这里，估计很多朋友都搞不懂 isOut <= 1 和 isOut = 1 有什么分别？读者还知道 IO 是怎样用 Verilog 去实现和控制的吗？

```
assign SDA = isOut ? rSDA : 1'bz;
```

典型的 IO 口控制都会使用 3 目操作符“a? b:c”，如果把这个稍微改头换面一下，那么：

```
if( isOut ) SDA = rSDA;
else SDA = 1'bz;
```

读者还记得“即时事件”和“时间点事件”的区别吗？在此，重点不是 SDA = rSDA 或者 SDA = 1'bz。然而重点是 if 条件里的 isOut。如果第 54 行的 isOut 我们用“<=”去赋值的话，那么 if 条件会发生“时间点事件”，时间点事件的特征，都是“在时间点做决定，在未来输出决定”，结果 SDA 的 IO 口需要迟一个时钟才能成为输出。

反之如果，第 54 行的 isOut 我们用“=”去赋值的话，if 条件会发生“即时事件”，结果 SDA 的 IO 口会立即成为输出口。

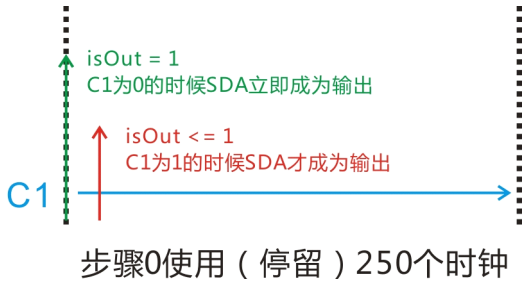


图 3.2o isOut = 1 和 isOut <= 1 的区别。

第三章的其中一个重点就是如何把“某个步骤和某个操作”整合起来，如图 3.2o 所示。图 3.2o 指出步骤 0 停留了 250 个时钟，如果 isOut <= 1 那么 SDA 会耗掉一个时钟以后，才成为输出状态。结果在步骤 0 中，SDA 输出状态只是保持 249 个时钟而已。反之，如果 isOut = 1 ，那么进入步骤 0 的一之际 SDA 会“立即”成为输出状态。结果 SDA 会以输出状态于步骤 0 的全过程，亦即在步骤 0 的 250 个时钟里 SDA 都是保持输出状态。

同样的道理也发生在 99~110 行和 112~123 行。

上述的内容不是“讲究不讲究”的问题，重点而是如何控制好每一个都可以利用的“时钟”。读者尝试想象一下，假设读者考试以 100 分为目标，如果读者只得到 99 分，笔者尝试问心一下，自己能接受吗？当然时钟控制和考试分数不能相提并论，因为前者比后者更“严格”和“敏感”，少 1 分都会酿成大错。如何把“这个 1 分”都控制好而且内容要“直接”和“清晰”，这也是在整合的概念中，其中一个重点。

话题有点扯远了 .... 第 66~67 行的步骤 1 是用调用伪函数前的准备，芯片 24LC04 写操作的第一个字节内容是“设备地址”，所以 rData 赋值为 { 4'b1010（产品系列），3'b000（总线地址），1'b0（写方向）}。至于 i 指定伪函数的入口-步骤 7，Go 赋值与从伪函数返回后的下一个步骤。

步骤 2（第 69~70 行）是写操作中的第二个字节内容，亦即“字地址”，所以 rData 赋值与 Addr_Sig。同样 i 指向伪函数的入口-步骤 7 和 Go 赋值与返回后的下一个步骤。步骤 3（第 72~73 行）则是写操作的第三个字节，内容是“写信息（写数据）”，所以 rData 赋值与 WrData，i 和 Go 同样也有步骤 1 和 2 的操作。

步骤 4（第 77~89 行）是结束信号的产生（笔者已经在前面解释过了）。步骤 5~6（第 91~95 行）是完成信号的产生。就这样，主机对从机的一次性写操作就完成了，在这里笔者所设计的 iic 功能模块是 非常“铭感”的，在一次性的操作中只要从机不给主机一个应答，写操作就会不停的一直重复，直到成功为止。

```
133.            else if( Start_Sig[1] )
134.                    case( i )
135.
136.                            0: // Start
137.                              begin
138.                                    isOut = 1;
139.
140.                                    if( C1 == 0 ) rSCL <= 1'b1;
141.                                    else if( C1 == 200 ) rSCL = 1'b0;
142.
143.                                    if( C1 == 0 ) rSDA <= 1'b1;
144.                                    else if( C1 == 100 ) rSDA <= 1'b0;
145.
146.                                    if( C1 == 250 -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
147.                                    else C1 <= C1 + 1'b1;
148.                              end
149.
150.                              1: // Write Device Addr
151.                              begin rData <= {4'b1010, 3'b000, 1'b0}; i <= 5'd9; Go <= i + 1'b1; end
152.
153.                              2: // Wirte Word Addr
```

```
154.                    begin rData <= Addr_Sig; i <= 5'd9; Go <= i + 1'b1; end
155.
156.                    3: // Start again
157.                    begin
158.                            isOut = 1'b1;
159.
160.                            if( C1 == 0 ) rSCL <= 1'b0;
161.                            else if( C1 == 50 ) rSCL <= 1'b1;
162.                            else if( C1 == 250 ) rSCL <= 1'b0;
163.
164.                            if( C1 == 0 ) rSDA <= 1'b0;
165.                            else if( C1 == 50 ) rSDA <= 1'b1;
166.                            else if( C1 == 150 ) rSDA <= 1'b0;
167.
168.                            if( C1 == 300 -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
169.                            else C1 <= C1 + 1'b1;
170.                    end
171.
172.                    4: // Write Device Addr ( Read )
173.                    begin rData <= {4'b1010, 3'b000, 1'b1}; i <= 5'd9; Go <= i + 1'b1; end
174.
175.                    5: // Read Data
176.                    begin rData <= 8'd0; i <= 5'd19; Go <= i + 1'b1; end
177.
178.                    6: // Stop
179.                    begin
180.                            isOut = 1'b1;
181.                            if( C1 == 0 ) rSCL <= 1'b0;
182.                            else if( C1 == 50 ) rSCL <= 1'b1;
183.
184.                            if( C1 == 0 ) rSDA <= 1'b0;
185.                            else if( C1 == 150 ) rSDA <= 1'b1;
186.
187.                            if( C1 == 250 -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
188.                            else C1 <= C1 + 1'b1;
189.                    end
190.
191.                    7:
192.                    begin isDone <= 1'b1; i <= i + 1'b1; end
193.
194.                    8:
195.                    begin isDone <= 1'b0; i <= 5'd0; end
196.
197.                    /******************************/ //function
```

```
198.
199.                    9,10,11,12,13,14,15,16:
200.                    begin
201.                        isOut = 1'b1;
202.
203.                        rSDA <= rData[16-i];
204.
205.                        if( C1 == 0 ) rSCL <= 1'b0;
206.                        else if( C1 == 50 ) rSCL <= 1'b1;
207.                        else if( C1 == 150 ) rSCL <= 1'b0;
208.
209.                        if( C1 == F100K -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
210.                        else C1 <= C1 + 1'b1;
211.                    end
212.
213.                    17: // waiting for acknowledge
214.                    begin
215.                        isOut = 1'b0;
216.
217.                        if( C1 == 100 ) isAck <= SDA;
218.
219.                        if( C1 == 0 ) rSCL <= 1'b0;
220.                        else if( C1 == 50 ) rSCL <= 1'b1;
221.                        else if( C1 == 150 ) rSCL <= 1'b0;
222.
223.                        if( C1 == F100K -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
224.                        else C1 <= C1 + 1'b1;
225.                    end
226.
227.                    18:
228.                    if( isAck != 0 ) i <= 5'd0;
229.                    else i <= Go;
230.
231.                    /****************************/
232.
233.                    19,20,21,22,23,24,25,26: // Read
234.                    begin
235.                        isOut = 1'b0;
236.                        if( C1 == 100 ) rData[26-i] <= SDA;
237.
238.                        if( C1 == 0 ) rSCL <= 1'b0;
239.                        else if( C1 == 50 ) rSCL <= 1'b1;
240.                        else if( C1 == 150 ) rSCL <= 1'b0;
241.
```

```
242.                                if( C1 == F100K -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
243.                                else C1 <= C1 + 1'b1;
244.                            end
245.
246.                            27: // no acknowledge
247.                            begin
248.                                isOut = 1'b1;
249.                                //if( C1 == 100 ) isAck <= SDA;
250.
251.                                if( C1 == 0 ) rSCL <= 1'b0;
252.                                else if( C1 == 50 ) rSCL <= 1'b1;
253.                                else if( C1 == 150 ) rSCL <= 1'b0;
254.
255.                                if( C1 == F100K -1 ) begin C1 <= 9'd0; i <= Go; end
256.                                else C1 <= C1 + 1'b1;
257.                            end
258.
259.                            /************************************/ // end fucntion
260.
261.                        endcase
262.
263.        /***************************************/
264.
265.        assign Done_Sig = isDone;
266.        assign RdData = rData;
267.        assign SCL = rSCL;
268.        assign SDA = isOut ? rSDA : 1'bz;
269.
270.       /****************************************/
271.
272.        //assign SQ_i = i;
273.
274.        /******************************/
275.
276. endmodule
```

第 133~261 行就是芯片 24LC04 的读操作，第 133 行的 if 条件表示，当 Start_Sig[1]被拉高以后读操作就开始了。第 199~229 行是写一个字节+应答位的处理伪函数，基本上和写操作的 99~127 行一样。

第 233~257 行是读一个字节+应答位的处理伪函数，我们先来看 233~244 行。第 235 行的 isOut = 1'b0，表示在每一个步骤里 SDA 保持输入状态为 200 个时钟（注意赋值操作符是 = 而不是 <=，具体的解释前面已经讲过了），余下的就不解释了。第 233 行中有 8 个步骤，亦即步骤 19~26（读者自己用手指数数看，是不是有 8 个？）。每一个步骤正

好处理读一字节的每一位内容。

步骤 27（第 246~257 行）是读一字节以后紧接着是“一位”应答位的处理，不同的是读一字节以后的应答位主机可以无视掉。所以步骤 27 以后，没有针对应答位的相关处理，这一点和写一字节以后的应答位有点不一样。

好了，读者先在把头移动到步骤 0（第 136~148 行），和写操作一样主机先发送一个起始信号（136~148 行），然后写一个字节的伪函数发送设备地址和字地址（步骤 1~2-第 150~154 行）。重点来了，当主机“点人完毕（发送设备地址）”和确定信息的读出“字地址”（发送字地址）以后，主机要改变方向了。

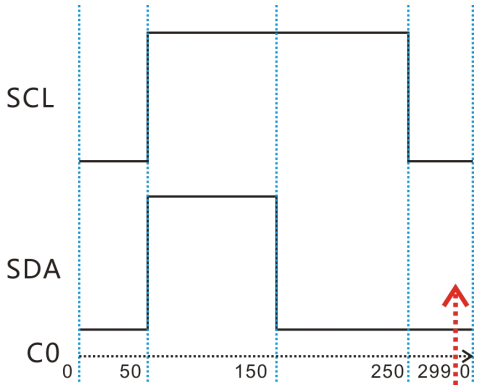


图 3.2p 改变方向的起始信号。

图 3.2p 从机访问改变方向（步骤 3）需要用到的起始信号和一开始（步骤 0）的起始信号没有多大的区别，步骤 3 的所用的起始信号 SCL 和 SDA 仅是在前面多加 50 个时钟的拉低而已，这个只是保险的动作而已。

步骤 4（第 172~173 行）是利用写一字节的伪函数重新发送已经改变方向的设备地址而已（设备地址的最低位从 0 变 1，亦即写变读）。步骤 5（第 175~176 行）则是利用读一字节的伪函数从从机那里读出早已在步骤 2（字节地址）确定好的信息。第 178~189 行是主机发送结束信号，以确保一次性的读操作已经结束。最胡步骤 7~8 发送完成信号。最后 265~268 行的输出驱动，读者自己看着办吧。

exp15_demo.v

```
1.  module exp15_demo
2.  (
3.          input CLK,
4.          input RSTn,
5.          output [3:0]LED,
6.
7.          output SCL,
8.          inout SDA
9.  );
```

```
10.
11.         /**************************/
12.
13.           wire [7:0]RdData;
14.           wire Done_Sig;
15.
16.         iic_func_module U1
17.           (
18.                   .CLK( CLK ),
19.                   .RSTn( RSTn ),
20.                   .Start_Sig( isStart ),
21.                   .Addr_Sig( rAddr ),
22.                   .WrData( rData ),
23.                   .RdData( RdData ),
24.                   .Done_Sig( Done_Sig ),
25.                   .SCL( SCL ),
26.                   .SDA( SDA )
27.           );
28.
29.           /***************************/
30.
31.           reg [3:0]i;
32.           reg [7:0]rAddr;
33.           reg [7:0]rData;
34.           reg [1:0]isStart;
35.           reg [3:0]rLED;
36.
37.         always @ ( posedge CLK or negedge RSTn )
38.                 if( !RSTn )
39.                         begin
40.                                 i <= 4'd0;
41.                                 rAddr <= 8'd0;
42.                                 rData <= 8'd0;
43.                                 isStart <= 2'b00;
44.                                 rLED <= 4'd0;
45.                             end
46.                 else
47.                         case( i )
48.
49.                                 0:
50.                                 if( Done_Sig ) begin isStart <= 2'b00; i <= i + 1'b1; end
51.                                 else begin isStart <= 2'b01; rData <= 8'h12; rAddr <= 8'd0; end
52.
53.                                 1:
```

```
54.                                 if( Done_Sig ) begin isStart <= 2'b00; i <= i + 1'b1; end
55.                                 else begin isStart <= 2'b10; rAddr <= 8'd0; end
56.
57.                                 2:
58.                                 begin rLED <= RdData[3:0]; end
59.
60.                             endcase
61.
62.         /***************************/
63.
64.           assign LED = rLED;
65.
66.           /***************************/
67.
68.   endmodule
```

上述是 iic 功能模块的测试代码。第 3~8 行是输入和输出的声明，第 16~27 行是 iic 功能模块的实例化，Start_Sig 由 isStart 驱动；Addr_Sig 由 rAddr 驱动；WrData 由 rData 驱动；至于 RdData 和 Done_Sig 连接至第 13~14 行连线声明。第 31~35 行是相关的寄存器声明，第 37~60 行是核心功能。

一开始的时候，所有相关的寄存器复位为零（第 40~44 行）。步骤 0（第 49~51 行）是把数据 8'h12 写入第 0 地址。步骤 1（第 53~55 行）是从地址 0 读出数据，然后在步骤 2 里（第 57~58 行）把输出结果赋值与 rLED（由于 LED 只有 4 位的关系大家就将就一下吧）。第 64 行表示 LED 由 rLED 驱动。

实验十五基本上就是到这里为止，在这里可能有同学要求仿真，很遗憾的是实验十五虽然看起来是很“正常”，而且运行起来也很“稳定”，但是它没有仿真的价值？为什么呢？因为实验十五犯了“理解上”的一个疏忽，这个“疏忽”到底是什么东西？而且严重到仿真也没有价值？很想知道吗？呵呵呵，下一小节读者就会知道了，这个“疏忽”联系到“速率”和“寄存器”等非常基础的知识。

3.3 iic 总线被忽略的基础知识和速率的关系。

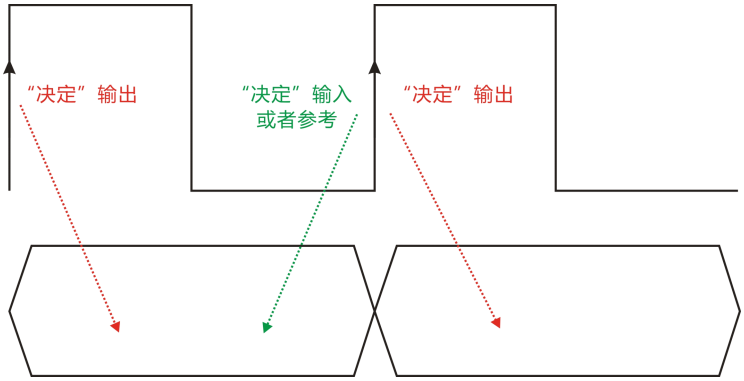


图 3.3a Verilog 理想时序的时间点事件

在实验十五笔者留下这样的一句话“我们在理解上疏忽了一个重点”，确实我们疏忽这样一个基础，但是为了避免产生困乱，在此笔者都只是以“理想时序”为例。图 3.3a 是时序篇的一个重点概念，亦即“时间点事件”，这也是 Verilog HDL 语言理想时序中某一个事件的表现（还有即时事件）。

关于时间点事件的输出“决定”，还是读入“决定”或者“参考”，这一切的一切都发生在时钟沿的“上升沿”。之所以时间点事件会在时钟的上升沿产生“各种决定”，因为模块的所有活动都是根据 always @ (posedege CLK) 这个写法，然而这不是第三章的重点。

每一个决定（上升沿）之间随之产生了第二章中所谓的“一个时钟，一块数据”具体的概念，当然这也不是第三章的重点。所谓的“速率”是指“1 秒内倒地能传输多少个位数据”，但是读者知道吗？“速率”同样也是指向，模块在时序上的活动“决定和决定之间的距离”。

笔者还是举例一个比较清晰的例子：假设黑金开发板使用 20Mhz 的时钟频率，如果笔者想要产生 100Khz 的 SCL 信号，那么笔者可以这样表达：

```
always @ ( posedge CLK or negedge RSTn)
if( !RSTn )
        begin rSCL <= 1'b0; end
else
        case( i )

                0,2,4,6,8,10,12,14:
                if( C1 == 100 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
                else rSCL <= 1'b1; C1 <= C1 + 1'b1; end

                1,3,5,7,9,11,13,15:
```

```
12.             if( C1 == 100 -1 ) begin C1 <= 8'd0; i <= i + 1'b1; end
13.             else rSCL <= 1'b0; C1 <= C1 + 1'b1; end
14.            ......
15.
16.        endcase
```

上面的一段代码中，我们使用 20Mhz 的时钟频率经过量化 100khz，结果是 200 个时钟，又或者可以这样说“20Mhz 与 100Khz 之间的速率比例是 200”。这段代码是用来产生 8 个 SCL 时钟，其中的双数步骤 0,2,4,6,8,10,12,14 (第 7~9 行) 是用来产生 SCL 的上升沿，反之单数步骤 1,3,5,7,9,11,13,15（第 11~13 行）则是用来产生 SCL 的下降沿。在此，我们可以这样下结论：“SCL 速率的产生是发生在双数步骤之间的决定”。

“速率”在某个程度上可以说是“某位数据停留了多久”。但是有一点笔者很好奇的是，为什么从数字系统（Digital System）被创造的开始，时钟的“上升沿”就成为了“决定”或者“触发”呢？这个问题估计只有火星人才知道。（有传言说数字系统不是被地球人所创造，而是被地球人所发现的外星科技）。

接下来我们来看看不同的时序图：

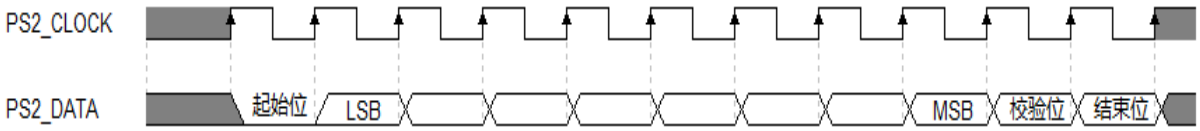


图 3.3b 主机角度的 PS2 时序图。

图 3.3b 是站在主机角度看的 PS2 时序图，PS2_DATA 的更新都是发生在 PS2_CLOCK 的上升沿，然而每一个 PS2_CLOCK 上升沿的之间决定了 PS2 的传输“速率”。如果站在从机的角度来说，那么“PS2_DATA 的有效性是发生在 PS2_CLOCK 的下降沿”。

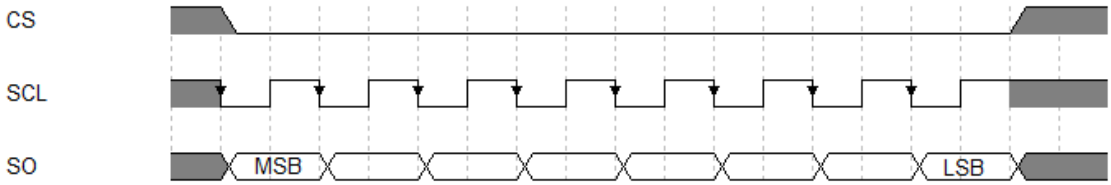


图 3.3c 主机角度的 SPI 时序图。

图 3.3c 是站在主机角度看的 SPI 时序图，SO 输出更新都是发生在 SCL 的下降沿，而不是上升沿，然而 SCL 的下降沿之间决定了 SPI 的传输“速率”。如果站在从机的角度来说的话，SO 数据的有效性是发生在 SCL 信号的上升沿。因此 SPI 传输协议就有这样一说“SCL 信号的下降沿主机更新（设置）数据，反之上升沿是从机锁存（读取）数据”。

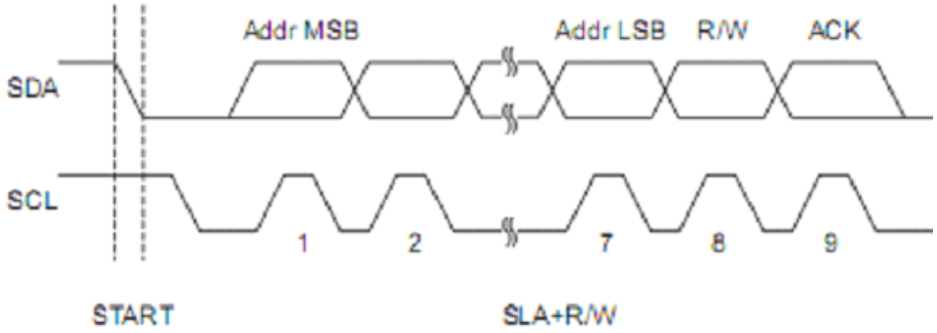


图 3.3d iic 写一个字节的时序图。

图 3.3d 是 iic 写一个字节的时序图，在此请问读者到底能不能看出图 3.3b~c 之间的区别吗？看不出？没关系 .. 笔者会慢慢解释的。一般人认为在 iic 的总线上，SDA 一位数据的“有效锁存（读取）”是发生在 SCL 信号的上升沿之后和下降沿之前，亦即 SCL 两则处于低电平，中间处于高电平。

不不不，这是一个美丽的误会 .... 如果 iic 总线真的用这样的方式去传输数据的话，那么“速率”就不适合用在它身上了，更不用说什么 100kHz 普通模式或者 400Khz 高速模式。在此可能有朋友要问：“如果这样的传输方式是一个美丽的误会，那么又如何解释 iic 传输数据的有效性呢？”好问题！

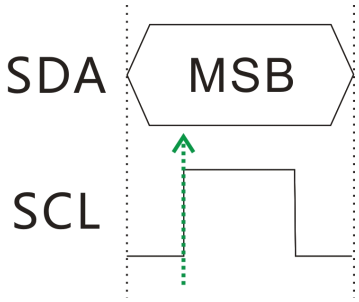


图 3.3e 一位数据的读取。

如图 3.3e 所示，SDA 其中一位数据被读取时发生在时钟的上升沿。如果继续分析的话，我们会看到-在 SCL 信号还没有产生上升沿之前 SDA 已经准备好了数据，因此对于数据的有效性来说是决定没有问题。在这里可能又有同学要问：“既然 SDA 的数据不缺乏有效性，那么这样的方式又有什么问题？”这位同学，这话就不能这样说了，有没有听过一句老人言“越是外表好看而且认为没有问题的东西实际上就是最大的问题”

读者尝试想想看：“如果 FPGA 同样使用 100Khz 的话，读者有如何去表达（写法）这样的传输方式呢？”就是因为实验十五使用 20Mhz 的频率去产生 100Khz 的频率，然而“速率”之间的比例是 200 倍，所以我们才能这样“有余”。

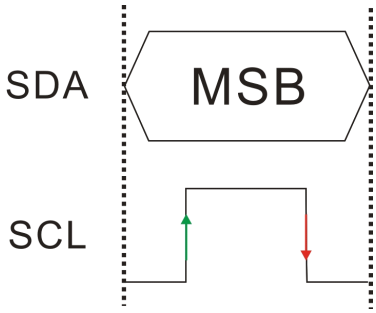


图 3.3f 上升沿读有效？还是下降沿读有效？

假设读者是一位初学者，当读者看了图 3.3f，那么读者请问："SDA 的数据 SCL 信号的上升沿读有效呢？还是下降沿读有效呢？"结果撞墙。无论是在表达的角度上，还是理解的角度上，类似的传输方式已经犯了"最基础的问题"。

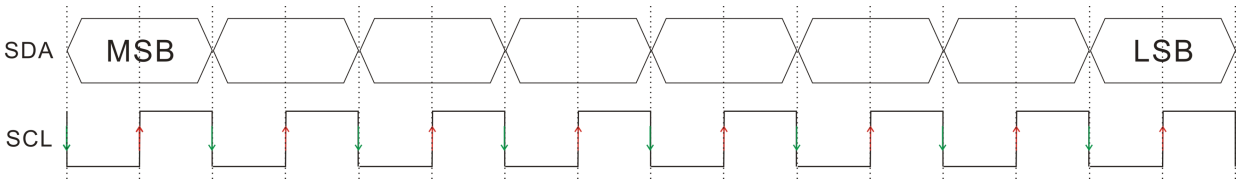


图 3.3g iic 总线主从之分，一字节数据传输的理想时序图。

读者不要忘了 iic 协议时有主从之分，某种程度上 iic 协议与 PS2 和 SPI 协议基本上没有什么两样，它们之间只是形式上的不同而已。图 3.3g 是笔者更正以后的时序图，图 3.3g 是一字节数据传输的理想时序图，在此有明显的主从之分，亦即 SCL 信号的下降沿主机设置 SDA 的数据（更新数据），SCL 信号的上升沿从机锁存 SDA 的数据（读取数据）。

看吧！更正以后的时序图是不是很和谐和"顺眼"许多了？在此，我们可以这样下结论："IIC 速率是由 SCL 两个时钟沿的之间来决定"。当读者明白这一小节其中的意思以后，仿真的工作和驱动更高速的 iic 总线才有意义。这个好比读者出国旅游，如果读者带着的人民币去马来西亚的话，再多的钱也没有用。(读者别反驳说有 Money Changer 就可以解决问题唷 ... 呵呵！)

3.4 按时序表驱动 iic 设备。

在前面一个小节（3.3），笔者已经更正了 iic 时序图存在的美丽误会，有此了解我们才可以以不同的“速率”去驱动芯片 24LC04。可以，有一点特别的是，这一回笔者会按时序表来编辑 24LC04 的驱动代码。首先读者必须了解一个重点，我们常常在手册看到的时序图或者或者时序表都是基于“物理时序的测试结果”，然而 Verilog HDL 语言的时序永远都是如此美丽和理想。

读者还记得笔者在前面说过的话吗？Verilog HDL 语言理应是完美的工具，但是使用者往往都习惯用“不完美的概念，去舞动完美的工具”，其中就有“将 Verilog 的理想时序当着物理时序来看待”这个“忽略”。至于这个“忽略”到底是如何形成的，然后又如何去解决它？估计这是一个使用者想都没有想过的问题，更何况是初学者呢？从另一方面我们又不得不按心自问：“手册上的时序图是不是 Verilog HDL 语言真正要表达的时序”— 这一问题。这写问题都是用语言回答不了的，从中我们也只能透过实验来领悟而已。

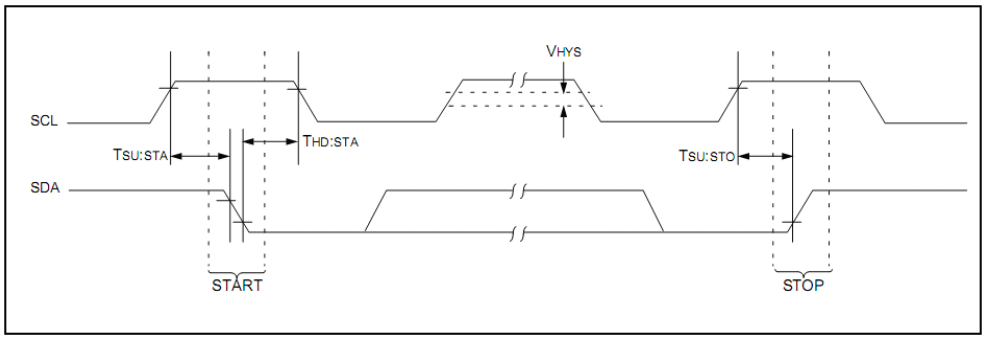
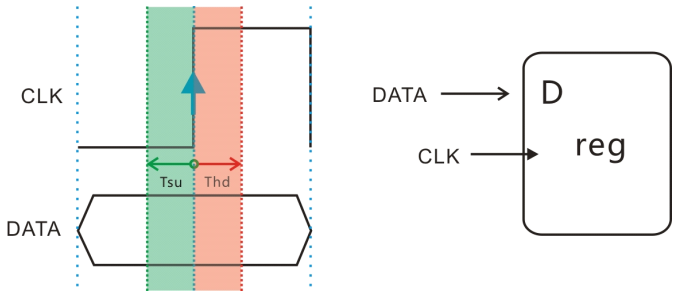


图 3.4a iic 总线—起始信号和结束信号的物理时序图。

图 3.4a 是起始信号和结束信号在物理上的表现，其中 Tsu:STA，Thd:STA 等物理时间，估计很多同学看了，就会开始头疼 ..... 不要紧，我们一步一步从最简单的基础知识开始。



3.4b Tsu 和 Thd 与寄存器的关系。

如果用寄存器的话来说，Tsu 就是时钟的上升沿之前（触发锁存之前）数据需要建立的时间，俗话是数据的热生时间。至于 Thd 是时钟的上升沿时候（触发锁存之后）数据的的保持时间，俗话叫做数据确保时间。用一个简单比喻，当一位业务员进货之前，他要

准备进货空间，账单手续等一些就绪功夫。当进货之后，业务员需要执行结账，点货等工作。在这里货物就是数据，进货前后就是 Setup 和 Hold 等意思。

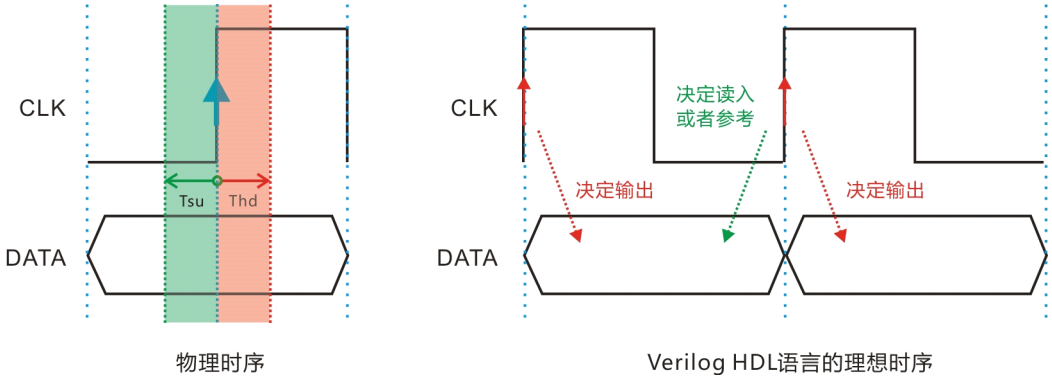


图 3.4c 物理时序和 Verilog HDL 语言的理想时序。

在这里，可能有些同学要问："笔者我已经搞不清楚什么事物理时序和 Verilog HDL 语言理想时序了 .... 救命！"这位同学不要慌，其实这是很简单的问题而已。Verilog HDL 语言的时序图是真正意义上的理想，换之我们常常在课本和手册上看到的都是物理时序图，如图 3.4c 所示。

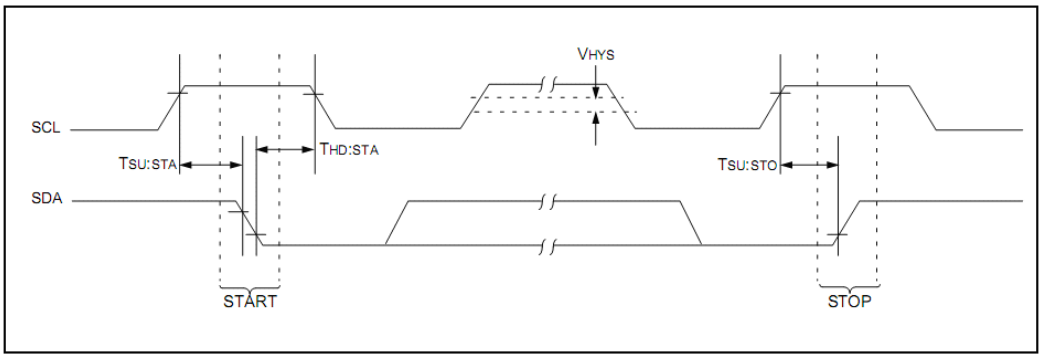


图 3.4a iic 总线—起始信号和结束信号的物理时序图。

好了，返回话题。虽说 Tsu（Time Setup）和 Thd（Time Hold）是寄存器的建立时间和保持时间，但是芯片 24LC04 却不是一个简单寄存器，然而这些都不是重点，重点就是就是在于 Tsu 和 Thd 之间的"触发"。在这里关键的"触发"是 SDA 由高变低的瞬间（图 3.4a 的左边），同样的道理也可以应用在结束信号的身上（图 3.4a 的右边）

Parameter	Symbol	STANDARD MODE		Vcc = 4.5 - 5.5V FAST MODE		Units
		Min	Max	Min	Max	
Clock frequency	FCLK	—	100	—	400	kHz
Clock high time	THIGH	4000	—	600	—	ns
Clock low time	TLOW	4700	—	1300	—	ns
SDA and SCL rise time	TR	—	1000	—	300	ns
SDA and SCL fall time	TF	—	300	—	300	ns
START condition hold time	THD:STA	4000	—	600	—	ns
START condition setup time	TSU:STA	4700	—	600	—	ns
Data input hold time	THD:DAT	0	—	0	—	ns
Data input setup time	TSU:DAT	250	—	100	—	ns
STOP condition setup time	TSU:STO	4000	—	600	—	ns

图 3.4d 芯片 24LC04 的时序表。

图 3.4d 是万恶的时序表，可说是鬼神看了都害怕，同学看了都想逃 ... 别怕别怕，它是充气娃娃而已。

Clock Frequency 就是 iic 的“速率”了，它有 100kHz 和 400Khz 之分。
Clock High Time 就是 SCL 信号处于高电平需要的时间。
Clock Low Time 是 SCL 信号处于低电平需要的时间。
Rise Time 是 SDA 信号和 SCL 信号同是电平从低变高需要的时间。
Fall Time 是 SDA 信号和 SCL 信号同是电平由高变低需要的时间。
Start Condition Hold Time 是起始信号产生时需要的保持时间。
Start Condition Setup Time 是起始信号产生的需要的建立时间。
Data Input Hold and Setup Time 先无视它。
Stop Condition Setup Time 是结束信号产生式需要的建立时间。

接下来我们需要用 20Mhz 的时钟频率去量化上述的所有“Time”，如表 3.4a 所示。

相关参数	最小	最大	N 个时钟（20Mhz）
Thigh	80	---	个 CLK
Tlow	94	---	个 CLK
Tr	---	20	个 CLK
Tf	---	6	个 CLK
Tsu:Start	80	---	个 CLK
Thd:Start	94	---	个 CLK
Tsu:Stop	80	---	个 CLK

表 3.4a 相关的时间参数经过 20Mhz 量化后的结果。

首先笔者必须强调的是，我们要用 Verilog HDL 语言的理想时序去实现芯片 24LC04 的物理时序。此外，100Khz 和 20Mhz 频率（速率）之间比例是 1:200，换句话说操作时间非常“有余”。（这一段话听不明白不要紧，以后自然而然会明白的）好了，先在开始就是 Show Time 了。

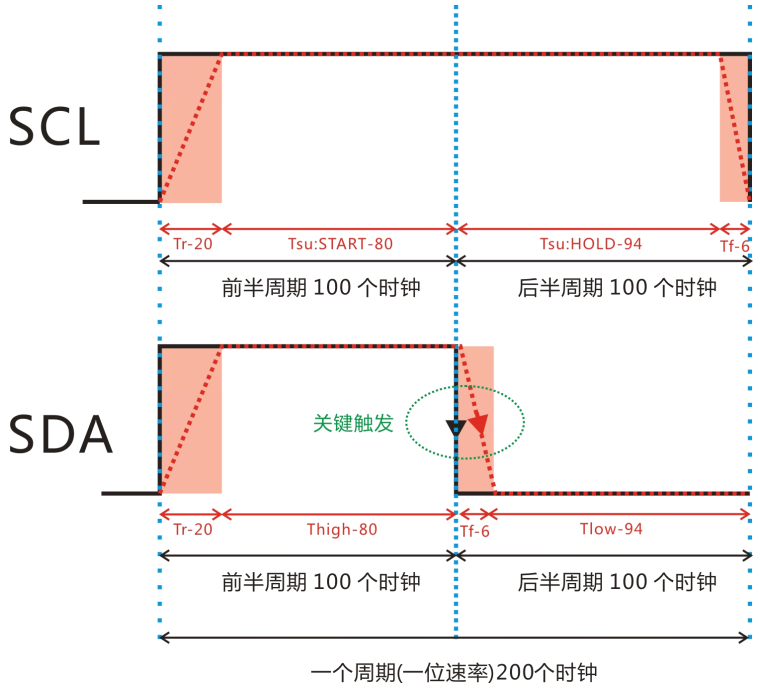


图 3.4e 理想时序实现起始信号。

图 3.4e 是不是够复杂？不 .... 一点都不复杂，读者只要将物理时序和理想时序分开来看就好了。黑线是理想时序也是 Verilog 实现的时序，反之红色虚线是物理时序。图 3.4e 全部事件参数都是使用 20Mhz 量化以后的结果来计算。我们先假设一开始的时候 SDA 和 SCL 都为低电平，随之就是产生起始信号。

在前半周期的时候，SCL 的红色虚线（物理时序）从低变高经过时间 Tr 20 个时钟，随之经过 Tsu:START 80 个时钟。在后半周期的时候，SCL 的红色虚线依然保持高电平 Thd:START 94 个时钟，然后由高变低经过 Tf 6 个时钟。当然这些都是用物理时序（红色虚线）去看时序而已，如果用 Verilog 的理想时序（黑色实线）去实线的话，从一开拉高电平到最后，一共使用了 200 个时钟。

至于 SDA 的红色虚线（物理时序）在前半周期的时候，先从低变高经过 Tr 20 个时钟，然后保持高电平 Thigh 80 个时钟。在后半周期的时候，由高变低经过 Tf 6 个时钟，然后保持低电平 Tlow 94 个时钟。如果我们用黑色实线（Verilog 的理想时序）去实现的话，从一开始 SDA 拉高，然后保持 100 个时钟（前半周期），然后拉低 SDA 又 100 个时钟（后半周期）。

在这里，请问读者是否看到其中的重点了？对于很多同学来说，令人头疼的就是 Rise Time 和 Fall Time，只要是稍微“物理”一点的时序，当电平发生电平发生变化的时候，就会看到令人厌恶的斜线 .... 尽可以的话人人都希望斜线的宽度越短越好 .... 但是这也只是关系到“物理”而已，反之 Verilog HDL 语言可以无视，或者说可以直接把斜线“套在 Verilog 的理想时序里面”。

这样说有点抽象了，笔者应该这样说 ... Rise Time 和 Fall Time 应该先换算成 N 个时钟，然后总和在 Verilog 的理想时序里面。Rise Time 可以看成是“速率的前一半部分”，

Fall Time 是“速率的后一半部分。继续引用前面的起始信号为例，100kHz 对 20Mhz 的速率比例是 200，用 20Mhz 去驱动 100Khz 的信号对于我们来说，时钟是非常有余的。速率的一半是 100，如果 SCL 和 SDA 信号的电平产生变化，那么：

SCL 信号：

（速率的前半部分）100 = Time Rise + Tsu：START
（速率的后半部分）100 = Thd :START + Time Fall

SDA 信号：

（速率的前半部分）100 = Time Rise + Thigh
（速率的后半部分）100 = Time Fall + Tlow

只要明白这个要点以后，我们可是使用同样的道理去实现结束信号。

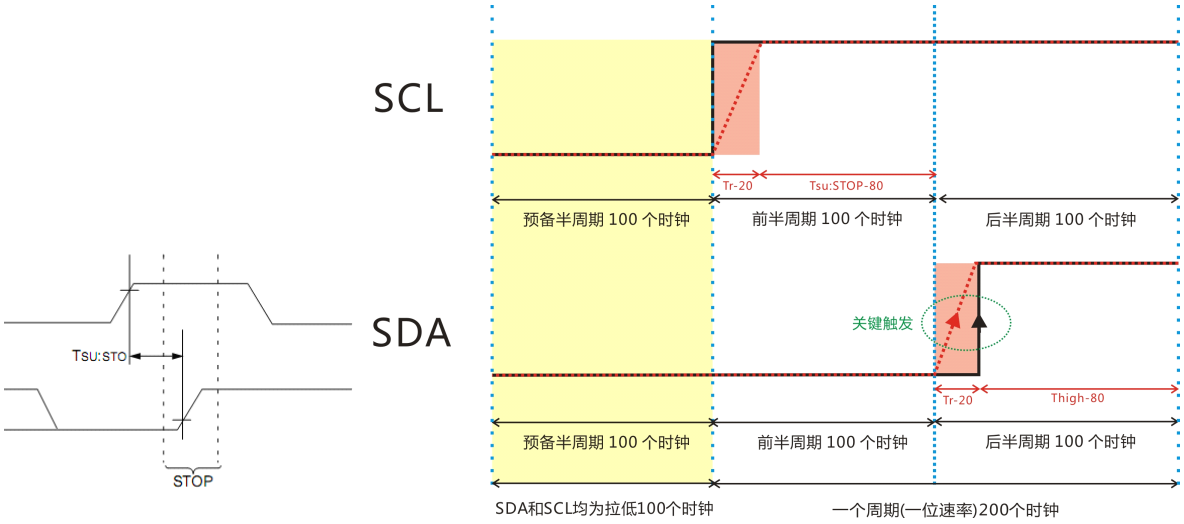


图 3.4f 结束信号。

图 3.4f 是结束信号的时序图，左图是官方的，右图是笔者画的。左图乍看下没什么，很简单而已，但是这张图充满了陷阱，读者相信吗？经过更正以后，比较接近事实的是笔者画的右图。每当主机要发结束信号的时候，SDA 和 SCL 必须先拉低半个速率，作为预备和稳定性的保险，假设速率为 100Khz，那么半个速率有 100 个时钟（基本上 1/4 个速率即可）。

当预备的半个速率经过以后，随之是结束信号的“一位速率”。首先 SCL 信号会先拉高，红色虚线为物理时序，Tr + Tsu：STOP 为 100 个时钟（一位速率的前半部分），随后会保持高电平 100 个时钟（一位速率的后半部分）。至于黑色实线（理想时序），当预备用的半个速率经过以后，黑色实线会保持 200 个时钟的高电平。

同样 SDA 信号也先拉低半个速率作为预备用，接着是结束信号的“一位速率”。红色虚线（物理时序）先经过 100 个时钟的拉低电平以后（预备用），接下来结束信号“一位

速率”的前半部分依然也保持低电平，红色虚线随后从低变高，经过 Tr + Th 100 个时钟。反之黑色实线，先拉低自己 100 个时钟作为与备用，然后又拉低自己 100 个时钟所谓结束信号—速率的前半部分，最后拉高电平 100 个时钟。

==========

分析完起始信号和结束信号之后，接下来我们就要分析写一位数据和读一位数据的物理 / 理想时序图。

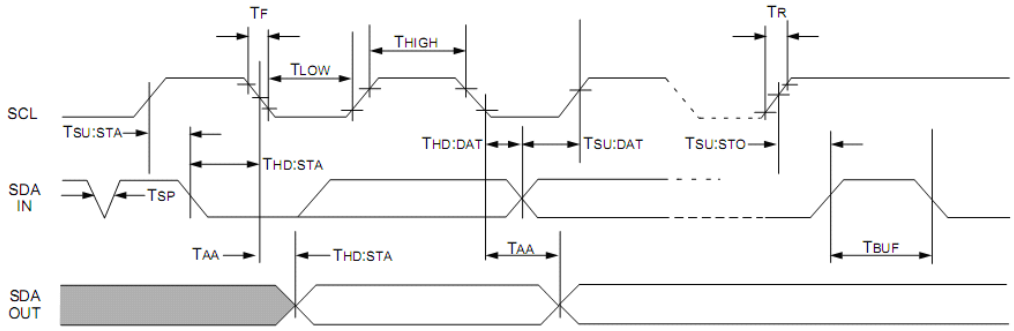


图 3.4h 官方的读一位数据和写一位数据的物理时序图。

老实说，图 3.4h 作为官方的手册真的是有点惨不忍睹？图 3.4h 是不是太过专业了，还是笔者打怪打不足（等级太低的意思）？总之，图 3.4h 对于笔者来说看不懂就是了，无视无视。不过从中，我们还可以找到一些需要的线索，亦即 Thd：DATA 和 Tsu:DATA。

Data input hold time	THD:DAT	0	—
Data input setup time	TSU:DAT	250	—
STOP condition setup time	TSU:STO	4000	—
Output valid from clock	TAA	—	3500

图 3.4i 数据有效性的相关参数。

相关参数	最小	最大	N 个时钟（20Mhz）
Tsu：DATA	5	---	个 CLK
Thd：DATA	0	---	个 CLK
TAA	---	70	个 CLK

表 3.4b 经过 20Mhz 量化。

图 3.4i 是官方手册给出数据有效性的相关参数，反之表 3.4b 是经过 20Mhz 量化的结果。至于 Tsu：DATA 是指数据被锁存之前（时间沿触发之后）所需要的建立时间（热生时间）；Thd：DATA 是指数据锁存之后（时间沿触发之后）所需要的保持时间（确保时间）；TAA 是指数据由从机出发之前需要的准备时间，和主机读操作有关系。

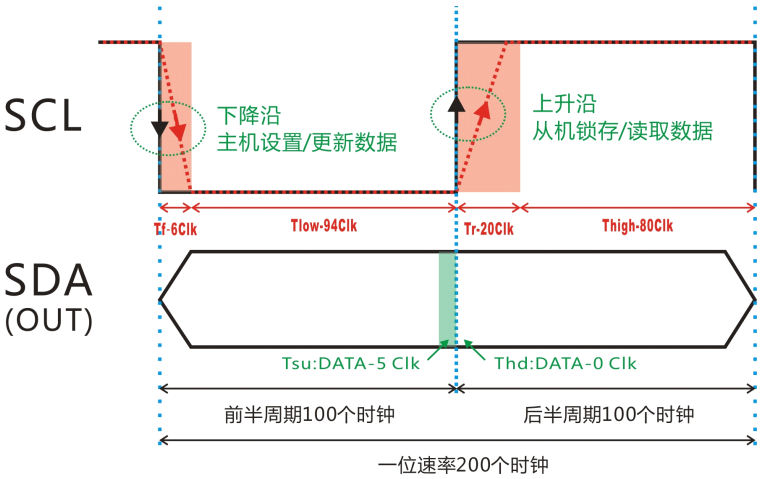


图 3.4j 主机发送一位数据从机读取一位数据（主机写操作）。

在 3.3 小节里笔者说过，官方给出的相关的时序图（数据读写操作）虽说没有错误。但是在某种程度上来说，确实有几些“忽略”，尤其在 Verilog 这一环，这些“忽略”是致命的。iic 协议和 SPI 协议有几点相识就是下降沿设置（更新）数据，上升沿锁存（读取）数据。图 3.4j 是主机发送一位数据，从机读取一位数据（俗称写操作）理想 /物理时序的混合图。

黑色实线代表 Verilog HDL 语言的理想时序，反之红色虚线是物理时序。SCL 信号产生下降沿的时候，SDA 信号就开始更新输出的数据。当经过 100 个时钟的前半周期（前半速率），SCL 信号拉高随之产生上升沿，这时候 Tsu：DATA 参数就派上用场了。对于黑色实线来说（理想时序）可以直接无视它，然而红色虚线却需要它。

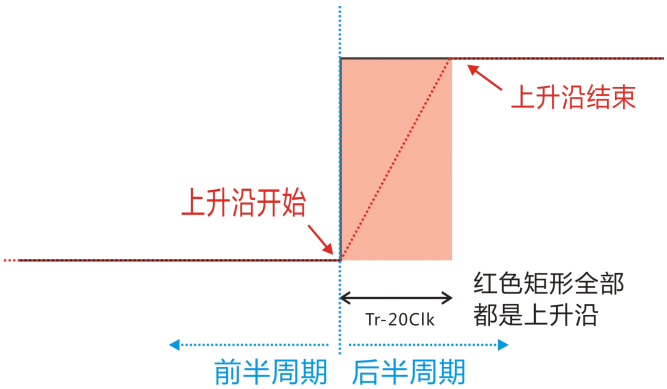


图 3.4k 上升沿从如何定义。

在这里，可能有些朋友要问：“红色虚线不像黑色实线那样直上直下而是有斜度 .... 请问如何定义红色虚线的上升沿呢？”按笔者的话来说，Tr 这一部分全是红色虚线的上升沿，更具体的话，Tr 这一部分的左边是上升沿的开始，右边是上升沿的结束。一般计算，红色虚线的上升沿是定义在“左边（上升沿的开始）”。不要问笔者为什么，笔者只是站在 Verilog 理想时序的角度去看而已。

对于 Verilog 的理想时序来说，“上升沿的触发”定义在“上升沿的开始”，至于上升沿的开始到结束之间是“一段保险时间”。但是有很多同学就是喜欢把上升沿定义在红色矩形的“中间”，不要忘了当电平处于不高也不低的时候是高祖太状态，高祖太就像一个绝缘体把两点之间的联系切掉。

哦 .... 话题又扯远了，继续分析图 3.4j。Thd:DATA 为 0 个时钟，亦即数据被从机锁存后可以忽略保持时间（或者说数据立即有效）。就这样，一位数据就成功锁存在从机里。

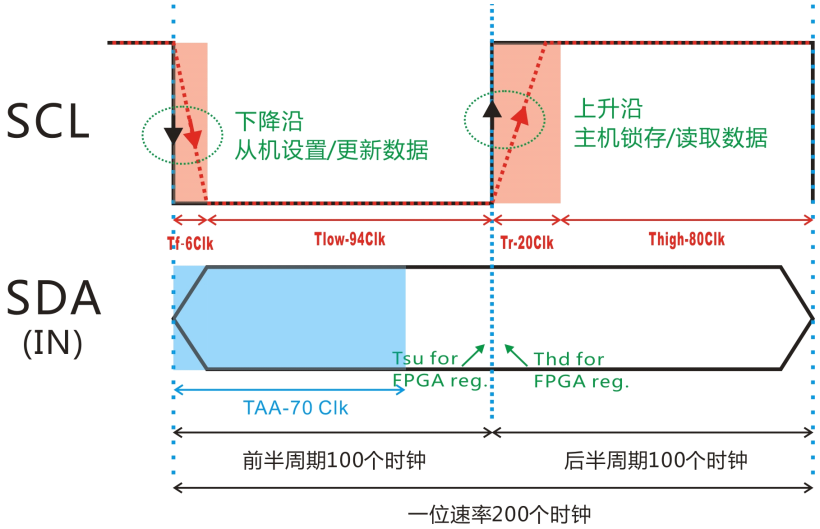


图 3.4l 从机设置（更新）数据，主机锁存（读取）数据（主机读操作）。

图 3.4l 是主机读一位数据的操作。当主机要求从从机那里读取数据的时候， SDA 的控制权就递交给从机了（SCL 的控制权依然在主机的手上）。在这里我们要明白，此时要锁存 SDA 数据的对象是 FPGA，发送（更新）SDA 数据的对象是芯片 24LC04。

首先主机（FPGA）把 SCL 信号拉下以后，从机就更新 SDA 的数据。新数据由从机出发之前需要经过 70 个时钟的 TAA，换句话说从机更新的 SDA 数据需要经过 70 个时钟以后才有效。

直到半个周期以后，主机再拉高 SCL 信号，此时会产生上升沿，在上升沿触发之前要满足 Tsu for FPGA reg，换之在上升沿之后要同样满足 Thd for FPGA reg。在此，可能有些同学犯傻，Tsu /Thd for FPGA reg 到底是从那里来呢？不要忘了，现在是 FPGA 读取芯片 24LC04 的数据，FPGA 也存在物理延迟，不过很小而已，大约是几个 ns，所以我们可以直接无视（需要更详细的资料自己去看 TimeQuest 相关的手册吧）。就这样，一位数据就成功被主机读进来。

Bus free time	TBUF	4700	—

图 3.4m Tbuf 时间参数。

相关参数	最小	最大	N 个时钟（20Mhz）
Tbuf	94	---	个时钟

表 3.4c Tbuf 经 20Mhz 量化。

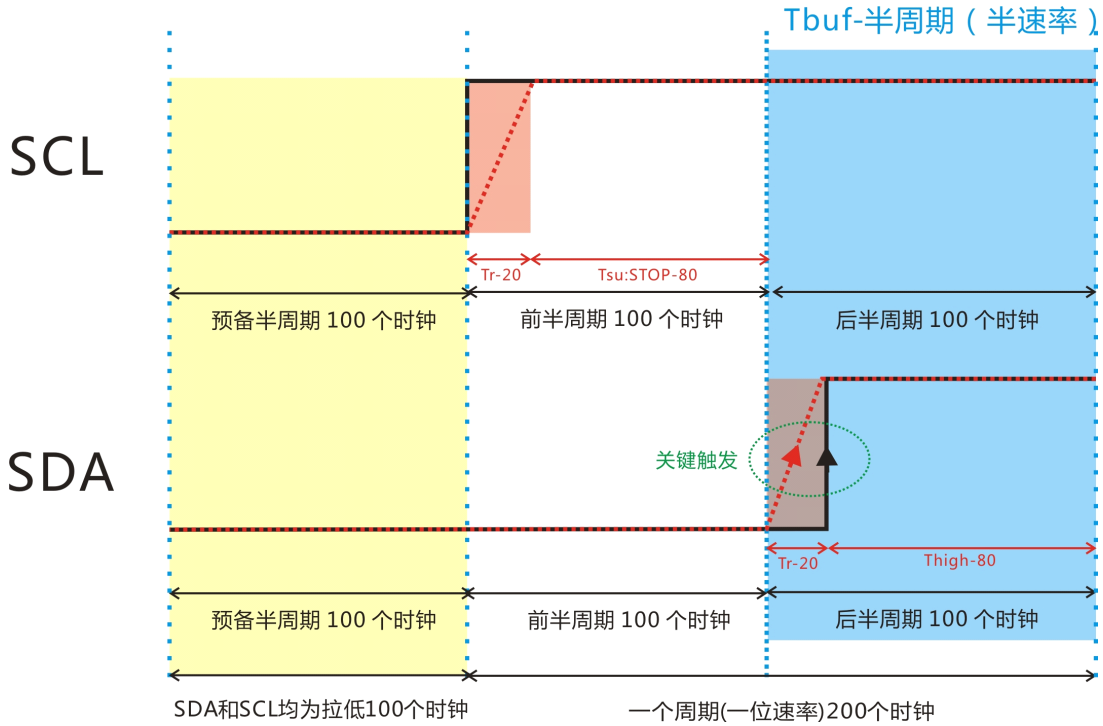


图 3.4n 结束信号和 Tbuf。

图 3.4n 是结束信号的混合（理想+物理）时序图，至于蓝色矩形是 Tbuf 时间参数。Tbuf 是指主机和从机开始新的沟通之前需要确保的时间，换句话说每当主机和从机结束沟通之间需要确保的时间，以致主机和从机可以释放对总线的控制。虽说 TBuf 在手册上要求 94 个时钟，但是为了方便设计笔者直接输入半个周期的时间（人类真的很奇怪为什么都喜欢以 10 位整除的数目）。

到目前为止，这些补充知识笔者已经解释到七七八八了，接下来还是用实验说了算。比起实验十五，实验十六完全是按照着芯片 24LC04 写程序，此外笔者不会用 Verilog HDL 语言去勉强迎合于物理时序，反而笔者会用 Verilog HDL 语言的理想时序去满足物理时序的时间要求，而不失自身的美丽和理想。

实验十六：真正意义上的速率 100Khz

图 3.4o iic 功能模块。

实验十六和实验十五拥有同样的建模图形（如图 3.4o 所示）所以读者自己看着办了，我们还是直接进入代码：

iic_func_module.v

```
module iic_func_module
(
        input CLK,
        input RSTn,

        input [1:0]Start_Sig,
        input [7:0]Addr_Sig,
        input [7:0]WrData,
        output [7:0]RdData,
        output Done_Sig,

        output SCL,
        inout SDA,

        /***************/
        output [4:0]SQ_i

);

      /*************************/

        parameter FREQ = 9'd200;

        /*************************/

        reg [4:0]i;
        reg [4:0]Go;
        reg [8:0]C1;
```

```
29.         reg [7:0]rData;
30.         reg rSCL;
31.         reg rSDA;
32.         reg isAck;
33.         reg isDone;
34.         reg isOut;
```

第 1~34 行基本上没什么改变，除了第 16 行的仿真输出被恢复以外。

```
35.
36.         always @ ( posedge CLK or negedge RSTn )
37.             if( !RSTn )
38.                 begin
39.                         i <= 5'd0;
40.                         Go <= 5'd0;
41.                         C1 <= 9'd0;
42.                         rData <= 8'd0;
43.                         rSCL <= 1'b1;
44.                         rSDA <= 1'b1;
45.                         isAck <= 1'b1;
46.                         isDone <= 1'b0;
47.                         isOut <= 1'b1;
48.                     end
49.             else if( Start_Sig[0] )
50.                 case( i )
51.
52.                         0: // Start
53.                         begin
54.                                 isOut = 1;
55.
56.                                 rSCL <= 1'b1;
57.
58.                                 if( C1 == 0 ) rSDA <= 1'b1;
59.                                 else if( C1 == 100 ) rSDA <= 1'b0;
60.
61.                                 if( C1 == FREQ -1) begin C1 <= 9'd0; i <= i + 1'b1; end
62.                                 else C1 <= C1 + 1'b1;
63.                         end
64.
65.                         1: // Write Device Addr
66.                         begin rData <= {4'b1010, 3'b000, 1'b0}; i <= 5'd7; Go <= i + 1'b1; end
67.
68.                         2: // Wirte Word Addr
69.                         begin rData <= Addr_Sig; i <= 5'd7; Go <= i + 1'b1; end
```

```
70.
71.                     3: // Write Data
72.                     begin rData <= WrData; i <= 5'd7; Go <= i + 1'b1; end
73.
74.                     /************************/
75.
76.                     4: // Stop
77.                     begin
78.                             isOut = 1'b1;
79.
80.                             if( C1 == 0 ) rSCL <= 1'b0;
81.                             else if( C1 == 50 ) rSCL <= 1'b1;
82.
83.                             if( C1 == 0 ) rSDA <= 1'b0;
84.                             else if( C1 == 150 ) rSDA <= 1'b1;
85.
86.                             if( C1 == 50 + FREQ -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
87.                             else C1 <= C1 + 1'b1;
88.                     end
89.
90.                     5:
91.                     begin isDone <= 1'b1; i <= i + 1'b1; end
92.
93.                     6:
94.                     begin isDone <= 1'b0; i <= 5'd0; end
95.
96.                     /******************************/ //function
97.
98.                     7,8,9,10,11,12,13,14:
99.                     begin
100.                            isOut = 1'b1;
101.                            rSDA <= rData[14-i];
102.
103.                            if( C1 == 0 ) rSCL <= 1'b0;
104.                            else if( C1 == 100 ) rSCL <= 1'b1;
105.
106.                            if( C1 == FREQ -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
107.                            else C1 <= C1 + 1'b1;
108.                    end
109.
110.                    15: // waiting for acknowledge
111.                    begin
112.                            isOut = 1'b0;
113.                            if( C1 == 150 ) isAck <= SDA;
```

```
114.
115.                                    if( C1 == 0 ) rSCL <= 1'b0;
116.                                    else if( C1 == 100 ) rSCL <= 1'b1;
117.
118.                                    if( C1 == FREQ -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
119.                                    else C1 <= C1 + 1'b1;
120.                            end
121.
122.                            16:
123.                            if( isAck != 0 ) i <= 5'd0;
124.                            else i <= Go;
125.
126.                            /******************************/ // end function
127.
128.                    endcase
```

第 36~47 行是相关寄存器的复位操作，还是没有改变 .... 第 49~128 行是该模块的功能 1，亦即写一字节数据。步骤 0（第 53~63 行）其中第 54 行表示 SDA 在整个步骤里为输出状态；第 56 行表示 SCL 信号在整个步骤 0 里都为拉高；第 58~59 行是 SDA 信号拉高前 100 个时钟，后拉低 100 个时钟，这一点完全符合前面的内容（Verilog 不失自身的美丽又能满足物理时间要求）；至于第 61~62 行就不解释了。

然后把焦点放在第 96~126 行的 写一字节+应答位 处理的伪函数。第 98~108 行是写一个字节的操作，它是由 8 个步骤（第 98 行）组成，每一个步骤处理相应的字节位；第 100 行的 isOut = 1 表示在该步骤 SDA 为输出状态；第 101 行表示 SDA 的数据数据，更具 i 和 rData 进行相关换算；第 103~104 行是拉低 SCL 信号 100 个时钟（前半期），然后拉高 SCL 信号 100 个时钟（后半期）。

第 110~120 是用来处理应答位。第 112 行为 SDA 在整个步骤里为输入状态；第 113 行表示在第 150 个时钟的时候读取 SDA，当时钟计数到 100 的时候 SDA 上的应答数据基本上已经有效和成熟了，笔者等到第 150 个时钟才读取应答数据是基于保险考虑。步骤 16（第 122~124 行）是返回步骤用的。

接下来读者请把头摆到步骤 1~3。步骤 1（第 64~65 行）是发送一字节的设备地址；步骤 2（第 68~69 行）是发送一字节的字地址；步骤 3（第 71~72 行）是发送一字节的数据。

步骤 4（第 76~88 行）是用来发送结束信号。在前面笔者已经说过，在结束信号还没有发送之前，SCL 和 SDA 需要同时拉低充当一段预备用的时间，在此笔者给上 1/4 周期而已（基本上已经满足了，1/2 周期的预备用时间是更安全而已）。

```
129.
130.                    else if( Start_Sig[1] )
```

```
131.                    case( i )
132.
133.                            0: // Start
134.                              begin
135.                                      isOut = 1;
136.
137.                                      rSCL <= 1'b1;
138.
139.                                      if( C1 == 0 ) rSDA <= 1'b1;
140.                                       else if( C1 == 100 ) rSDA <= 1'b0;
141.
142.                                      if( C1 == FREQ -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
143.                                       else C1 <= C1 + 1'b1;
144.                              end
145.
146.                              1: // Write Device Addr
147.                              begin rData <= {4'b1010, 3'b000, 1'b0}; i <= 5'd9; Go <= i + 1'b1; end
148.
149.                              2: // Wirte Word Addr
150.                              begin rData <= Addr_Sig; i <= 5'd9; Go <= i + 1'b1; end
151.
152.                              3: // Start again
153.                              begin
154.                                      isOut = 1'b1;
155.
156.                                      if( C1 == 0 ) rSCL <= 1'b0;
157.                                       else if( C1 == 50 ) rSCL <= 1'b1;
158.                                       else if( C1 == 250 ) rSCL <= 1'b0;
159.
160.                                      if( C1 == 0 ) rSDA <= 1'b0;
161.                                       else if( C1 == 50 ) rSDA <= 1'b1;
162.                                       else if( C1 == 150 ) rSDA <= 1'b0;
163.
164.                                      if( C1 == 300 -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
165.                                       else C1 <= C1 + 1'b1;
166.                              end
167.
168.                              4: // Write Device Addr ( Read )
169.                              begin rData <= {4'b1010, 3'b000, 1'b1}; i <= 5'd9; Go <= i + 1'b1; end
170.
171.                              5: // Read Data
172.                              begin rData <= 8'd0; i <= 5'd19; Go <= i + 1'b1; end
173.
174.                              6: // Stop
```

```
175.                    begin
176.                        isOut = 1'b1;
177.
178.                       if( C1 == 0 ) rSCL <= 1'b0;
179.                        else if( C1 == 50 ) rSCL <= 1'b1;
180.
181.                        if( C1 == 0 ) rSDA <= 1'b0;
182.                        else if( C1 == 150 ) rSDA <= 1'b1;
183.
184.                        if( C1 == 50 + FREQ -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
185.                        else C1 <= C1 + 1'b1;
186.                    end
187.
188.                    7:
189.                    begin isDone <= 1'b1; i <= i + 1'b1; end
190.
191.                    8:
192.                    begin isDone <= 1'b0; i <= 5'd0; end
193.
194.                    /******************************/ //function
195.
196.                    9,10,11,12,13,14,15,16:
197.                    begin
198.                        isOut = 1'b1;
199.
200.                        rSDA <= rData[16-i];
201.
202.                        if( C1 == 0 ) rSCL <= 1'b0;
203.                        else if( C1 == 100 ) rSCL <= 1'b1;
204.
205.                        if( C1 == FREQ -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
206.                        else C1 <= C1 + 1'b1;
207.                    end
208.
209.                    17: // waiting for acknowledge
210.                    begin
211.                        isOut = 1'b0;
212.
213.                        if( C1 == 150 ) isAck <= SDA;
214.
215.                        if( C1 == 0 ) rSCL <= 1'b0;
216.                        else if( C1 == 100 ) rSCL <= 1'b1;
217.
218.                        if( C1 == FREQ -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
```

```
219.                                else C1 <= C1 + 1'b1;
220.                            end
221.
222.                            18:
223.                            if( isAck != 0 ) i <= 5'd0;
224.                            else i <= Go;
225.
226.                            /****************************/
227.
228.                            19,20,21,22,23,24,25,26: // Read
229.                            begin
230.                                isOut = 1'b0;
231.                                if( C1 == 150 ) rData[26-i] <= SDA;
232.
233.                                  if( C1 == 0 ) rSCL <= 1'b0;
234.                                  else if( C1 == 100 ) rSCL <= 1'b1;
235.
236.                                  if( C1 == FREQ -1 ) begin C1 <= 9'd0; i <= i + 1'b1; end
237.                                  else C1 <= C1 + 1'b1;
238.                            end
239.
240.                            27: // no acknowledge
241.                            begin
242.                                isOut = 1'b1;
243.                                //if( C1 == 100 ) isAck <= SDA;
244.
245.                                  if( C1 == 0 ) rSCL <= 1'b0;
246.                                  else if( C1 == 100 ) rSCL <= 1'b1;
247.
248.                                  if( C1 == FREQ -1 ) begin C1 <= 9'd0; i <= Go; end
249.                                  else C1 <= C1 + 1'b1;
250.                            end
251.
252.                            /************************************/ // end fucntion
253.
254.                        endcase
```

至于读操作基本上和写操作的出入不多，除了转换方向的步骤以外，还有读一字节的伪函数。

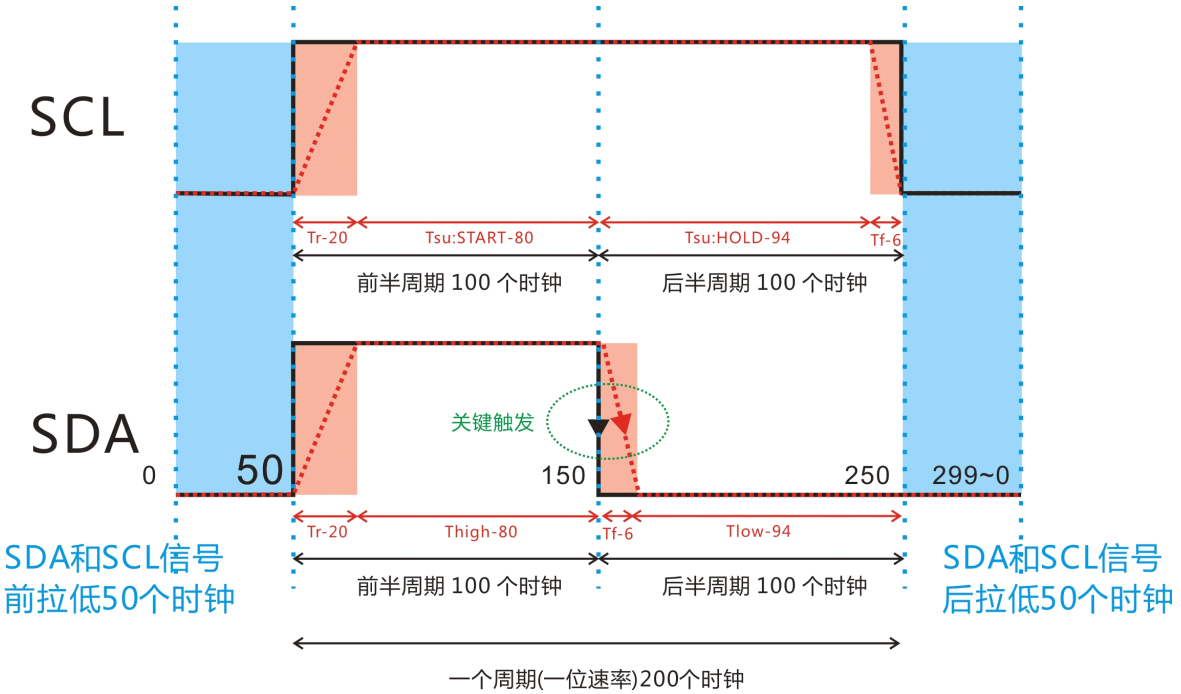


图 3.4p 主机访问从机改变方向时发送的（再）起始信号。

图 3.4p 是第 152~166 行主机改变方向用的起始信号。笔者开始犯懒了，还是直接用图画来解释更好，一图胜千言嘛。图 3.4p 是步骤 0—起始信号的加强版，亦即起始信号前后都加上拉低的 1/4 周期 SDA 和 SCL 信号。图 3.4p 已经够直接了，读者自己看着办吧。

第 228~250 行是 读一字节 + 应答位 处理的伪函数。第 228~238 行（三八~呵呵呵）是由 8 个步骤组成（第 228 行），每一个步骤对应一位数据读取操作；第 230 行表示在该步骤里，SDA 为输入状态；第 231 行表示，读数据发生在第 150 个时钟，别问笔者为什么是第 150 个时钟，而不是第 100 个时钟，说白了就是更保险而已；第 233~234 行是前 100 个时钟拉低 SCL 信号，后 100 个时钟拉高 SCL 信号。

至于步骤 27（第 240~250 行）是读一字节以后的应答位处理，这时候的它（应答位）从机有没有回应也无所谓了，所以笔者注释掉了~笑。

好了，我们简单的按着步骤，大概的了解一下 iic 功能模块第 2 功能的基本流程。首先产生起始信号（步骤 0），然后发送设备地址（步骤 1）和字地址（步骤 2），随之重新发送一个起始信号（步骤 3）用于改变访问的方向。重新发送已经改变方向的设备地址（步骤 4），接下来读取一字节数据（步骤 5），最后发送结束信号（步骤 6）。基本上，就是这么简单而已~

exp16_demo.v

```
1.    module exp16_demo
2.    (
3.            input CLK,
4.            input RSTn,
```

```
5.          output [3:0]LED,
6.
7.          output SCL,
8.          inout SDA
9.    );
10.
11.       /*************************/
12.
13.         wire [7:0]RdData;
14.         wire Done_Sig;
15.
16.       iic_func_module U1
17.         (
18.                 .CLK( CLK ),
19.                 .RSTn( RSTn ),
20.                 .Start_Sig( isStart ),
21.                 .Addr_Sig( rAddr ),
22.                 .WrData( rData ),
23.                 .RdData( RdData ),
24.                 .Done_Sig( Done_Sig ),
25.                 .SCL( SCL ),
26.                 .SDA( SDA )
27.         );
28.
29.         /*************************/
30.
31.         reg [3:0]i;
32.         reg [7:0]rAddr;
33.         reg [7:0]rData;
34.         reg [1:0]isStart;
35.         reg [3:0]rLED;
36.
37.       always @ ( posedge CLK or negedge RSTn )
38.               if( !RSTn )
39.                       begin
40.                               i <= 4'd0;
41.                               rAddr <= 8'd0;
42.                               rData <= 8'd0;
43.                               isStart <= 2'b00;
44.                               rLED <= 4'd10;
45.                           end
46.                 else
47.                       case( i )
48.
```

```
                    0:
                    if( Done_Sig ) begin isStart <= 2'b00; i <= i + 1'b1; end
                    else begin isStart <= 2'b01; rAddr <= 8'd0; rData <= 8'h0c; end

                    1:
                    if( Done_Sig ) begin isStart <= 2'b00; i <= i + 1'b1; end
                    else begin isStart <= 2'b10; rAddr <= 8'd0; end

                    2:
                    begin rLED <= RdData[3:0]; end

                endcase

        /***************************/

          assign LED = rLED;

          /***************************/

endmodule
```

这是下载到开发板测试用的代码，和 exp15_demo.v 出入不多，自己看着办吧。

iic_func_module.vt

```
`timescale 1 ns/ 1 ps
module iic_func_module_simulation();

        reg CLK;
        reg RSTn;

        reg [1:0] Start_Sig;
        reg [7:0] Addr_Sig;
        reg [7:0] WrData;

        wire Done_Sig;
        wire [7:0] RdData;
        wire SCL;

          /*************************/

          // io
        reg treg_SDA;
        wire SDA;
```

```
20.
21.         assign SDA = treg_SDA;
22.
23.         /************************/
24.
25.         wire [4:0]SQ_i;
26.
27.         /************************/
28.
29.         iic_func_module U1
30.         (
31.                 .CLK(CLK),
32.                 .RSTn(RSTn),
33.                 .Start_Sig(Start_Sig),
34.                 .Addr_Sig(Addr_Sig),
35.                 .WrData(WrData),
36.                 .Done_Sig(Done_Sig),
37.                 .RdData(RdData),
38.                 .SCL(SCL),
39.                 .SDA(SDA),
40.                 /******************/
41.                 .SQ_i( SQ_i )
42.         );
43.
44.        /***************************/
45.         initial
46.         begin
47.             RSTn = 0; #10 RSTn = 1;
48.                 CLK = 1; forever #25 CLK = ~CLK;
49.         end
50.        /******************************/
51.
52.         reg [3:0]i;
53.
54.         always @ ( posedge CLK or negedge RSTn )
55.                 if( !RSTn )
56.                         begin
57.                                 i <= 4'd0;
58.                                   Start_Sig <= 2'd0;
59.                                   Addr_Sig <= 8'd0;
60.                                   WrData <= 8'd0;
61.                             end
62.                 else
63.                         case( i )
```

```
64.
65.                             0:
66.                             if( Done_Sig ) begin Start_Sig <= 2'd0; i <= i + 1'b1; end
67.                             else begin Start_Sig <= 2'b01; Addr_Sig <= 8'b10101010; WrData <= 8'b10101010; end
68.
69.                             1:
70.                             if( Done_Sig ) begin Start_Sig <= 2'd0; i <= i + 1'b1; end
71.                             else begin Start_Sig <= 2'b10; Addr_Sig <= 8'b10101010; end
72.
73.                             2:
74.                             i <= i;
75.
76.                         endcase
77.
78.         /*************************************/
79.
80.         always @ ( posedge CLK or negedge RSTn )
81.             if( !RSTn )
82.                         treg_SDA <= 1'b1;
83.               else if( Start_Sig[0] )
84.                     case( SQ_i )
85.
86.                             15: treg_SDA = 1'b0;
87.                             default treg_SDA = 1'b1;
88.
89.                         endcase
90.             else if( Start_Sig[1] )
91.                     case( SQ_i )
92.
93.                             17: treg_SDA = 1'b0;
94.
95.                             19,20,21,22,23,24,25,26:
96.                             treg_SDA = WrData[ 26-SQ_i ];
97.
98.                             default treg_SDA = 1'b1;
99.
100.
101.                        endcase
102.
103.        /*************************************/
104.
105. endmodule
```

在实现十六的目录下还有一个名为 iic_func_module 的项目，这是用来测试

iic_func_module.v 模块，然而 iic_func_modulevt 就是激励文本。关于 IO 口如何在仿真中驱动这一点，读者就自己复习时序篇吧。

在这里，iic_func_module.v 的步骤 i 已经被引出来了。读者要知道步骤 i 在无论是在设计上还是仿真上都有许多的好处。步骤 i 在仿真中可以充当“调试跟踪”的作用，因为只要模块的那个部分出问题，步骤 i 就会指向它。此外，步骤 i 在驱动 IO 口的时候，我们还可以知道仿真对象的内部到底发生什么事情了。

第 29~42 行是仿真对象 iic_func_module.v 的实例化；第 45~49 行是复位信号和时钟的刺激产生，时钟的周期是 50ns 亦即是频率 20Mhz。第 52~76 行是仿真对象的激励过程，然而第 80~101 行是 IO 口的激励过程。

我们先看仿真对象 iic_func_module.v 的激励过程。在一开始的时候复位所有相关的寄存器（第 57~60 行）。步骤 0（第 65~67 行）是把数据 8'b10101010 写入字地址 8'b10101010。步骤 1（第 69~71 行）是把子弟之 8'b10101010 中的数据读出。步骤 2（第 73~74 行）是停止动作。

至于 IO 口的激励过程呢 ..... 第 83~89 行是用来针对 iic_func_module.v 的写功能（第 83 行）。当 i 等于 15 的时候（第 86 行）是应答位的反馈动作，所以 IO 就被驱动为逻辑 0（应答），其余的情况下 IO 都被驱动为逻辑 1（第 87 行在根本上，如果仿真对象不是将 IO 设置为输入状态，无论怎样驱动和刺激都没有用）。

第 90~101 行是用来针对 iic_func_module.v 的写功能（第 90 行）。当 i 等于 17 的时候，亦即应答位的反馈操作，所以 IO 被拉低（第 93 行）。当 i 为 19~26 的时候，是仿真对象读一字节数据的操作，i-19 是读一字节数据的最高位，反之 i-26 是读一字节数据的最低位（第 95 行）。在这里，笔者为了省事所以笔者就直接按 WrData 驱动 IO 口（第 96 行）。在其他的情况下 IO 都被拉高（第 98 行）。

在此，可能会有同学问：“为什么，在 86~98 行之间 treg_SDA 都使用 = 赋值操作符？”回这位同学，请再一次复习时序篇 ..... 就这样 iic_func_module.v 的激励过程就结束了。

仿真结果：

笔者需要先声明一下，由于笔者的机器不给力 .... 当笔者仿真 IO 口的时候，波形图会呈现红色的信号，红色信号线是逻辑 0 而不是不定状或者高祖太，这一点需要好好的注意。不然的话，别以为笔者在哪里搞错了，没法子吗 .... 笔者是穷人。

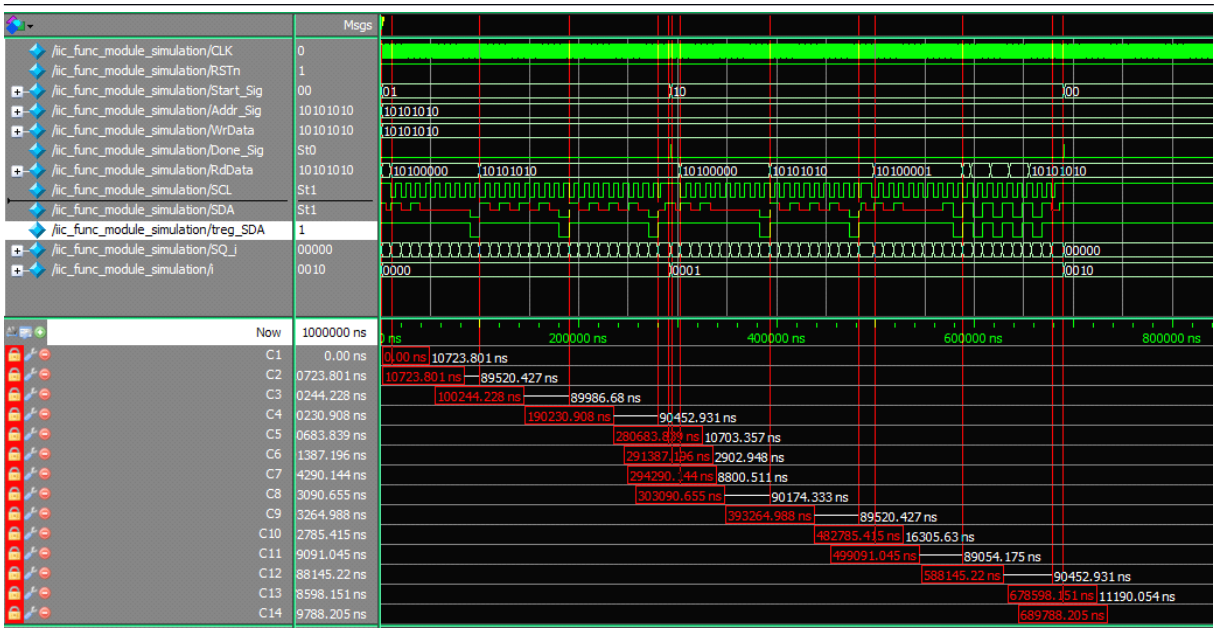


图 3.4q iic_func_module.v 的仿真结果。

“这是什么东西？密密麻麻的 ... 我要发疯了，救命呀...”，读者千万别被图 3.4q 吓到了，不要怕有笔者在，笔者会一步一步慢慢解释的。C1~C2 是写操作的开始信号；C2~C3 是写一字节的设备地址；C3~C4 是写一字节的字地址 8'b10101010；C4~C5 是写一字节的数据 8'b10101010；C5~C6 是写操作的结束信号。

C7~C8 是读操作的开始信号；C8~C9 是写一字节的设备地址；C9~C10 是写一字节的字地址 8'b10101010；C10~C11 是读操作换方向用的开始信号；C11~C12 是写一字节的设备地址，注意最低位已经从原本的逻辑 0（写访问）变成逻辑 1（读访问）；C12~C13 是读一字节数据 8'b10101010; C13~C14 是读操作的结束信号。

接下来，我们一个一个放大来看：

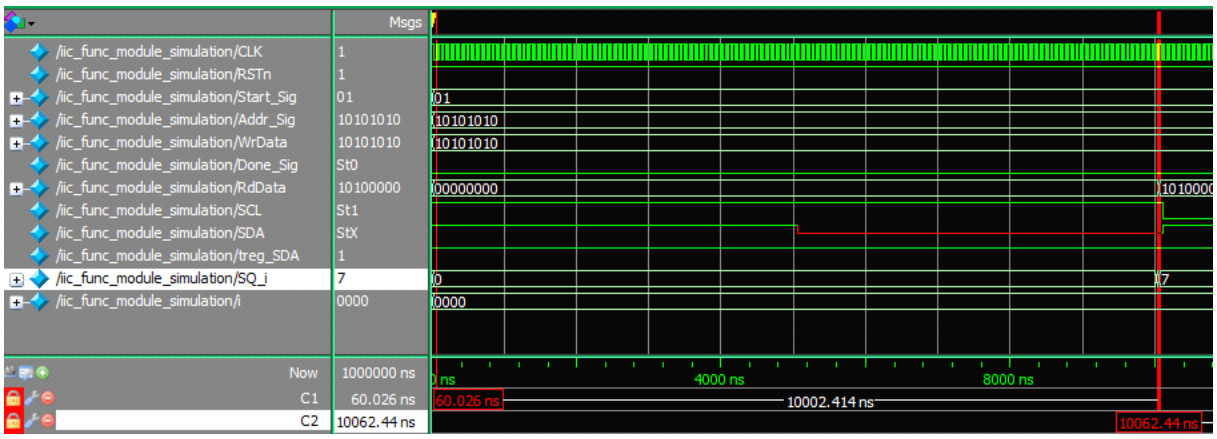


图 3.4r C1~C2-写操作的起始信号。

C1~C2 是写操作的起始信号，使用了 (C2 - C1)/50ns = 200 个时钟。

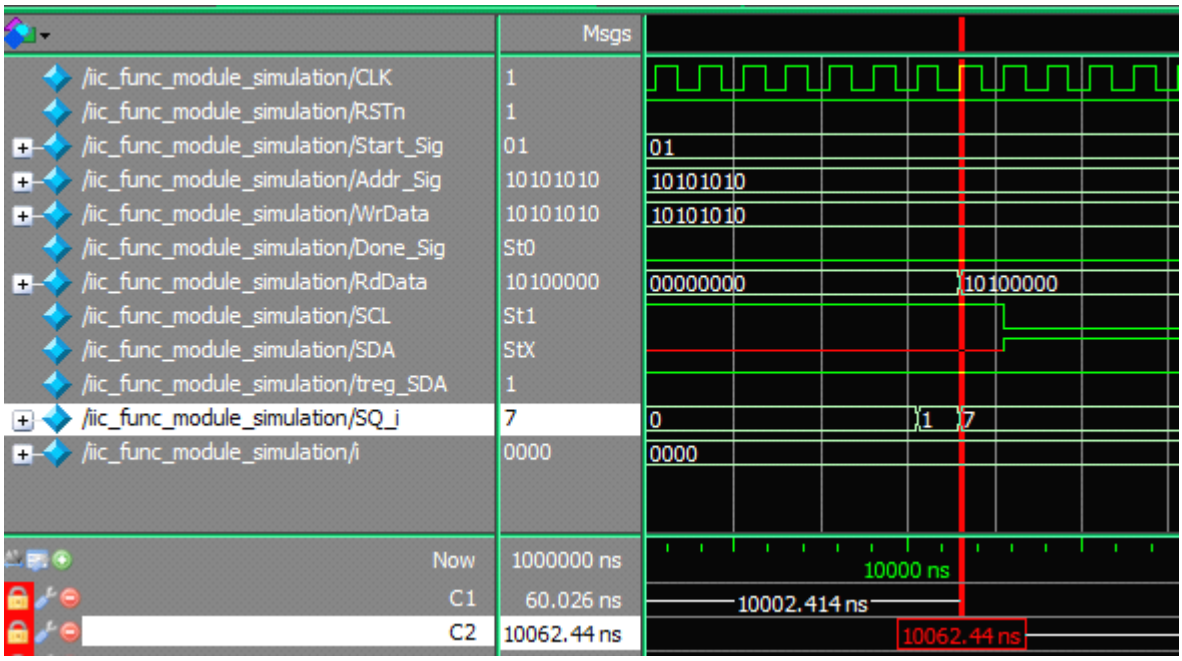


图 3.4s 进入“写一节数据 + 应答位”伪函数时用了一个时钟。

当产生完 200 个时钟的起始信号以后，该模块会用一个时钟去赋值内部的 rData 寄存器和更新 i 为 7，以便下一个时钟跳转到伪函数的步骤。

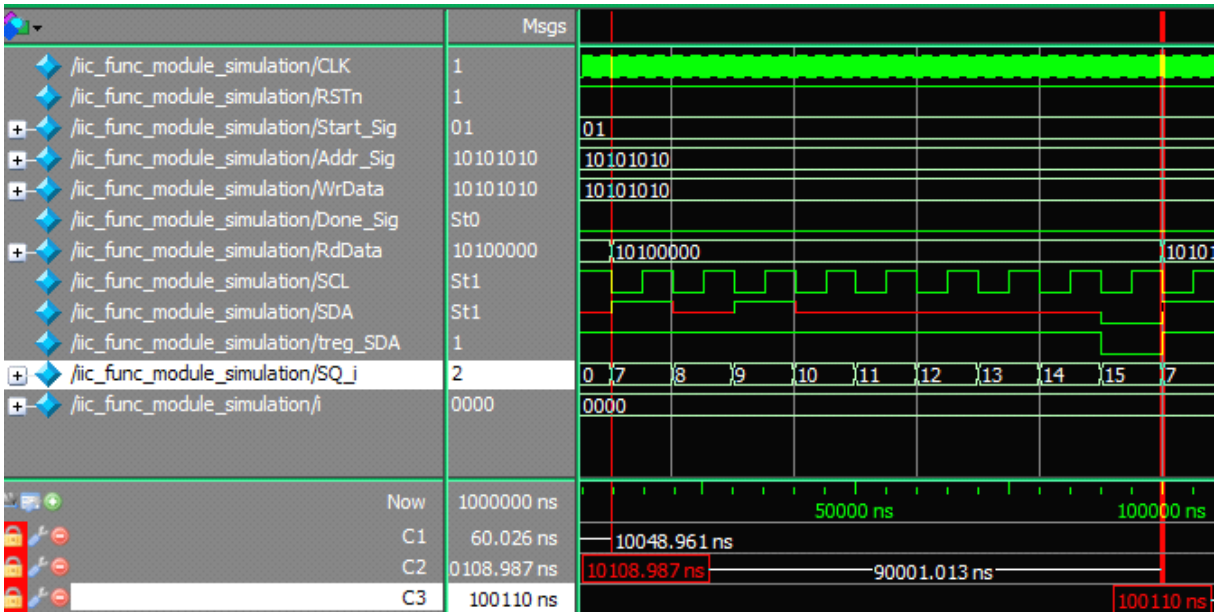


图 3.4t 写入一字节的设备地址和应答位处理。

图 3.4t 是写入一字节的设备地址，此外还包括一位应答位的处理，一共使用了（C3-C2）/50ns = 1800 个时钟 ，一字节数据加上一位应答位，总共有 9 位数据，亦即一位对应 200 个时钟。

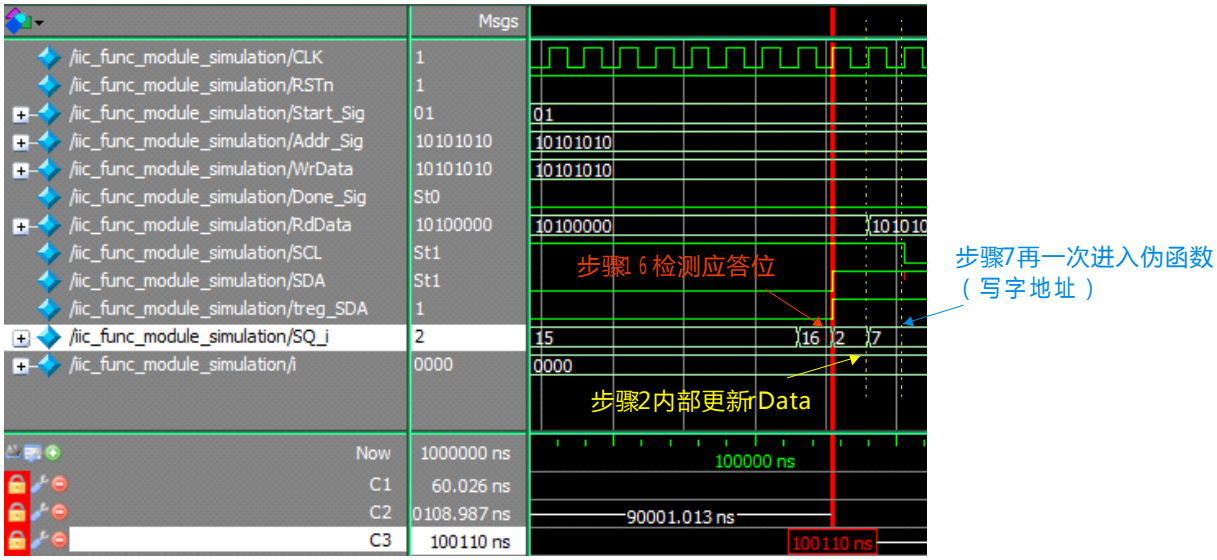


图 3.4u 伪函数的最后用一个时钟判断应答位的结果。

图 3.4u 中，光标 C3 指向的步骤是 16，亦即伪函数判断应答位的结果。其后步骤 i 会返回 2，亦即 iic_func_module.v 内部里为 rData 赋值字地址的操作，然后会再一次跳转到伪函数（步骤 7）。余下的 C3~C4 和 C4~C5 也有同样的动作。

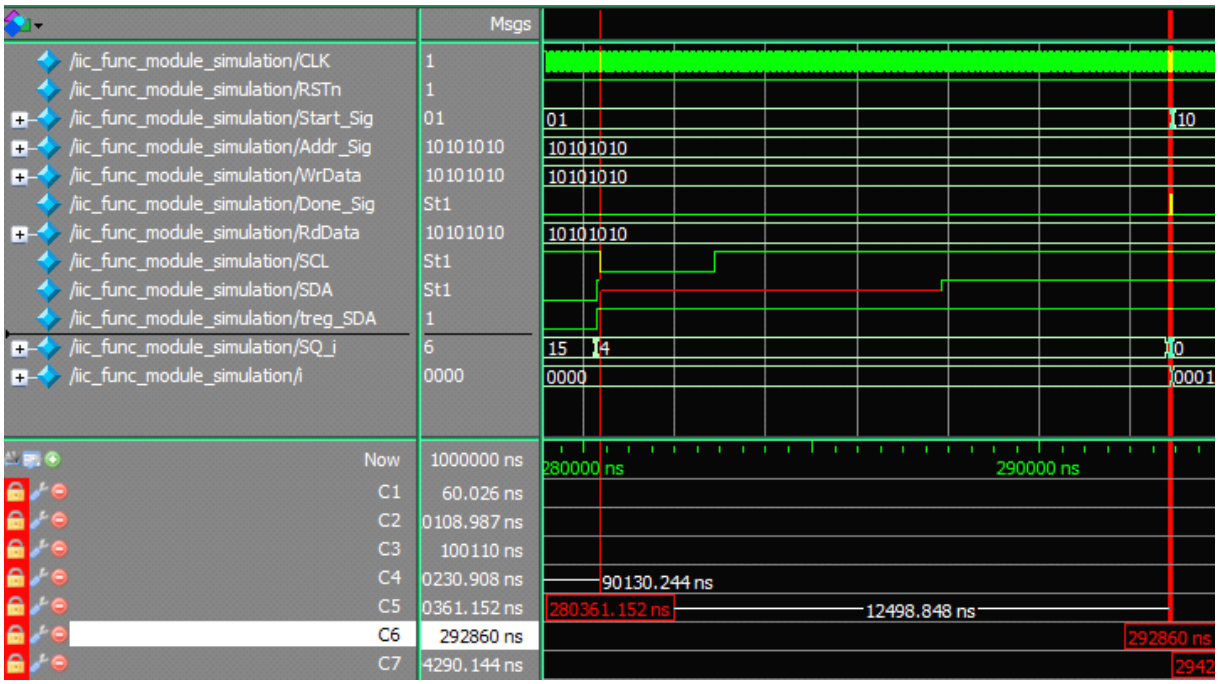


图 3.4v 写操作的结束信号。

图 3.4v 中的光标 C5~C6 是指向结束信号的产生过程（步骤 4），其中一共用了(C6-C5)/50ns = 250 个时钟。C6 的时候就是产生一个时钟的完成信号（步骤 5~6）。

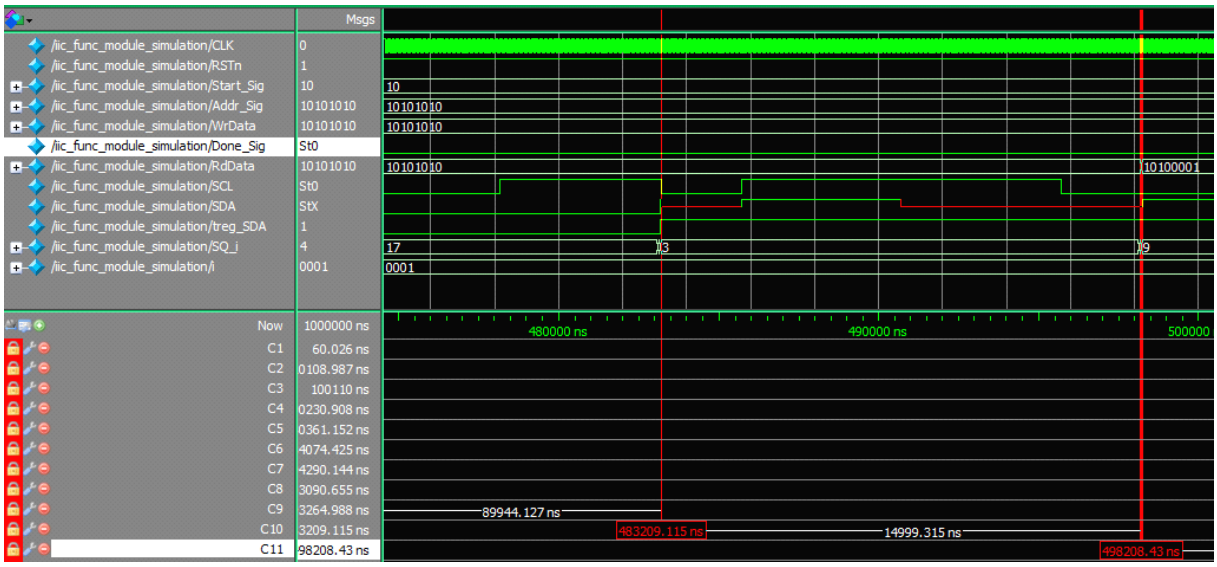


图 3.4w 主机改变访问方向时用的起始信号。

由于图像的数目有限的关系，所以笔者就直接跳到这里（基本上也是重复性的东西而已），图 3.4w 是主机改变方向是所产生的起始信号（读操作的步骤 3），其中一共用了 (C11-C10)/50ns = 300 个时钟，亦即普通起始信号的前面和后面都加上 SCL 和 SDA 信号的拉低 50 个时钟。

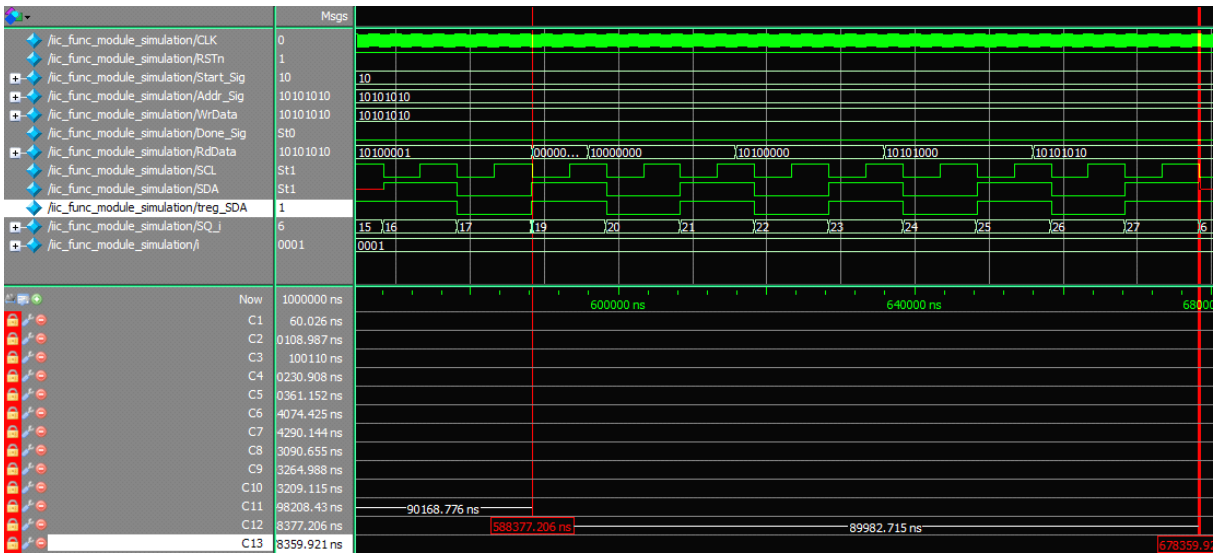


图 3.4x “读一字节+应答位”的伪函数。

图 3.4x 中的光标 C12~C13 是指向“读一字节 + 应答位”的伪函数，其中一共使用了 (C13-C12)/50ns = 1800 个时钟。这个伪函数有 9 个步骤，然而每一个步骤都有针对的相关“位”，亦即每一个步骤都使用 200 个时钟 ....

好了，波形图的解释就到此为止了。如果读者真的不明白笔者在说什么，或者怀疑笔者的解释有问题的话 .... 很抱歉，这只能说明读者还没有理解第一章和第二章的内容而已，怒笔者不重复了。

接下来，我们不借助波形图然后尝试评估在 iic_func_module.v 中，每一次的写操作和读操作一共用了多少是个时钟呢？

写操作总共用了 N 个时钟：

步骤 0，产生起始信号使用了 200 个时钟
步骤 1 ，更新 rData 为设备地址使用了 1 个时钟
步骤 7~14，写一字节设备地址使用了 1800 个时钟
步骤 15，处理一位应答位使用了 200 个时钟
步骤 16，判断应答位用了 1 个时钟
步骤 2，更新 rData 为字地址使用了 1 个时钟
步骤 7~14，写一个字节字地址使用了 1800 个时钟
步骤 15，处理一位应答位使用了 200 个时钟
步骤 16，判断应答位用了 1 个时钟
步骤 3，更新 rData 为写数据使用了 1 个时钟
步骤 7~14，写一个字节数据使用了 1800 个时钟
步骤 15，处理一位应答位使用了 200 个时钟
步骤 16，判断应答位用了 1 个时钟
步骤 4，产生结束信号使用了 250 个时钟
步骤 5~6，产生完成信号使用了 2 个时钟

N 个时钟 = 200 + 1 + 1800 + 200 + 1 + 1 + 1800 + 200 + 1 + 1 + 1800 + 200 + 1 + 250 + 2
= 6458

实际时间 = 6458 个时钟 * 50ns
= 332.9us

读操作总共用了 N 个时钟：

步骤 0，产生起始信号使用了 200 个时钟
步骤 1 ，更新 rData 为设备地址使用了 1 个时钟
步骤 9~16，写一字节设备地址使用了 1800 个时钟
步骤 17，处理一位应答位使用了 200 个时钟
步骤 18，判断应答位用了 1 个时钟
步骤 2，更新 rData 为字地址使用了 1 个时钟
步骤 9~16，写一个字节字地址使用了 1800 个时钟
步骤 17，处理一位应答位使用了 200 个时钟
步骤 18，判断应答位用了 1 个时钟
步骤 3，产生改变方向用的起始信号使用了 300 个时钟
步骤 4，更新 rData 为设备地址（读方向）使用了 1 个时钟
步骤 9~16，写一个字节字地址使用了 1800 个时钟
步骤 17，处理一位应答位使用了 200 个时钟
步骤 18，判断应答位用了 1 个时钟

步骤 5，清空 rData 用于暂存读字节数据用了 1 个时钟
步骤 19~26，读一字节数据用了 1800 个时钟
步骤 27，处理无用的应答位使用了 200 个时钟
步骤 6，产生结束信号用了 250 个时钟
步骤 7~8，产生完成信号用了 2 个时钟

```
N 个时钟  = 200 + 1 + 1800 + 200 + 1 + 1 + 1800 + 200 + 1 + 300 + 1 + 1800 + 200 + 1 + 1
            + 1800 + 200 + 250 + 2
          = 8759
实际时间  = 8759 个时钟   * 50ns
          = 437.95us
```

看！是不是很神奇呢？笔者自己也觉得非常不可思议，只要我们在写法上下功夫（整合），基本上要正确的评估某个模块使用了多个时钟是非常容易的事情而已。事实上，即时我们看得懂波形图，如果在写法缭乱没有同一的话，要准确的评估模块到底耗掉了多少个时钟是非常伤神和耗时（困难的意思） ....

好了，这一小节就到此为止了。在这小节里我们学到如何用 Verilog 的理想时序去实现物理时序而不失自身的理想和美丽。此外，我们还知道整合的概念（写法）在准确评估时钟的作用上达到举足轻重的地位。总结一句，这一小节的内容除了重要以外就是变态了 ....

3.5 非正规的速率

首先我们先简单的回忆一下，到底什么是速率呢？速率一般人都会认为是时钟的周期，当然这样的认识不是不对只是不接近答案而已。速率更确切的定义是“一位数据停留的时间”或者“一位数据的周期”，在此，我们需要明白“一位数据”所停留的时间可以由时钟的周期来决定，此外“一位数据”所停留的时间也可以由 N 个时钟的长度来决定。

打个比方 SPI 的速率是由 SCL 决定，串口传输的速率是由波特率来决定（波特率是指串口传输中一位数据所停留的时间）。前者有时钟信号来决定速率，后者则由一位数据的周期来决定速率。

在正常的情况下，正规的速率是指：“决定速率的时钟信号都有 50%的占空比”和“每一位数据的停留时间均为一样”。然而非正规的速率是指：要么“决定速率的时钟信号拥有不规则的占空比（非 50%的占空比）”，要么“每一位数据停留的时间都不均衡”。在这里，有些同学可能会觉得很好奇，为什么速率要有正规和非正规之分呢？

亲爱的同学，之所以速率有正规和非正规之分是因为“时序有理想和物理之分”的关系。在理想的时序下（这里是指 Verilog 的理想时序），所有波形图的信号都是四四方方，数据的周期都是一致，此外时钟信号都是 50%的占空比。反之在物理的时序下，信号的变化如同坐过山车一样“斜上斜下”，时钟的占空比可以随意变化等 ...

人们常说梦是理想的，现实是残酷的。在世界上，我们为了活得快乐一点，而不是夺取和消耗自然资源为代价去满足肉体上的欲望，而是如何在梦和现实之间找到平衡点，或者说是“粘糊剂”。这就好比理想时序和物理时序之间的映射，我们用不着勉强 Verilog HDL 语言去模仿物理时序，而是利用 Verilog HDL 语言自身的理想时序去满足物理时序的时间要求而不失自身的美丽。

然而非正规速率可以说是更破烂，更苛刻的物理时序。但是，无论物理时序再如何破烂不堪，我们只要找到其中的“粘糊剂（平衡点）”，我们照样可以用理想时序去满足它，实现它 .... 这一点就像古人所言那样“相由心生”，相是指我们眼前的环境或者世界，心是指我们的梦或者心里王国。只要梦美，相就会美，但是如何把美丽的梦成为实相，这就是生活的艺术 .....

实验十七：非正规速率的高速（400Khz）iic 总线

我们知道当代的 iic 总线有普通模式和高速模式，换句话说就是 100Khz 的速率和 400Khz 的速率。不过很不幸的是，前者是正规速率后者是非正规速率。那么到底是什么原因导致 iic 总线在高速模式下成为非正规速率呢？

在前面笔者已经说过，速率是指一位数据停留的时间，其中速率可以由时钟信号来决定。所以，导致 iic 总线在高速模式下成为非正规速率的凶手正是“SCL 信号”！接下来，笔

者会跟随读者在“实验的案发现场”中，寻找任何 SCL 信号“犯罪”所留下的蛛丝马迹！大家一起呐喊吧：“真相永远只有一个”！

Parameter	Symbol	STANDARD MODE		Vcc = 4.5 - 5.5V FAST MODE		Units
		Min	Max	Min	Max	
Clock frequency	FCLK	—	100	—	400	kHz
Clock high time	THIGH	4000	—	600	—	ns
Clock low time	TLOW	4700	—	1300	—	ns
SDA and SCL rise time	TR	—	1000	—	300	ns
SDA and SCL fall time	TF	—	300	—	300	ns
START condition hold time	THD:STA	4000	—	600	—	ns
START condition setup time	TSU:STA	4700	—	600	—	ns
Data input hold time	THD:DAT	0	—	0	—	ns
Data input setup time	TSU:DAT	250	—	100	—	ns
STOP condition setup time	TSU:STO	4000	—	600	—	ns
Output valid from clock	TAA	—	3500	—	900	ns
Bus free time	TBUF	4700	—	1300	—	ns

图 3.5a 芯片 24LC04 的时序表。

图 3.5a 是官方给出一芯片 24LC04 相关的时序表，左边是普通模式（100Khz 的速率），右边则是高速模式（400Khz 的速率）。首先我们先从图 3.5a 开始寻找出凶手留下的线索。在高速模式下，时钟信号的高电平（Thigh）和低电平（Tlow）保持时间为 600ns 和 1300ns，然而时钟信号的斜上（Tr）和斜下（Tf）时间均为 300ns。然后，我们把相关的时间参数用 20Mhz 去量化，结果如表 3.5a 所示。

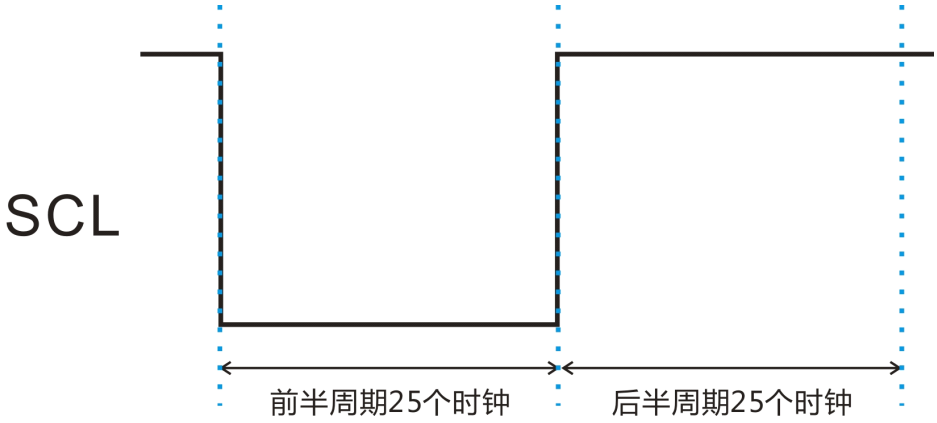
相关参数	最小	最大	N 个时钟（20Mhz）
Thigh	12	---	个时钟
Tlow	26	---	个时钟
Tr	---	6	个时钟
Tf	---	6	个时钟
Thd:START	12	---	个时钟
Tsu:START	12	---	个时钟
Thd:DATA	0	---	个时钟
Tsu:DATA	2	---	个时钟
Tsu:STOP	12	---	个时钟
TAA	---	18	个时钟
TBUF	26	---	个时钟

表 3.5a 经过频率 20Mhz 量化以后的时序表。

接下来，我们计算出 400Khz 经过 20Mhz 量化的结果：

N 个时钟 = (1/400Khz) / (1/20Mhz)
= 2.5us / 50ns
= 50 个时钟

半个周期的结果是 = 50/2
= 25 个时钟



3.5a 预期中 400Khz 速率的 SCL 信号。

当读者看到 400Khz 速率的一个周期是 50 个时钟，那么半周期自然是 25 个时钟，随后会联想到如图 3.5a 的 SCL 信号。很遗憾，这一回读者完完全全猜错了 ... 在前面笔者都说了 iic 的高速模式是一种非正规的速率，自然而安决定速率的 SCL 信号不可能是 50% 的占空比

接下来我们按表 3.5a 尝试绘出理想时序和物理时序的混合时序图 ：

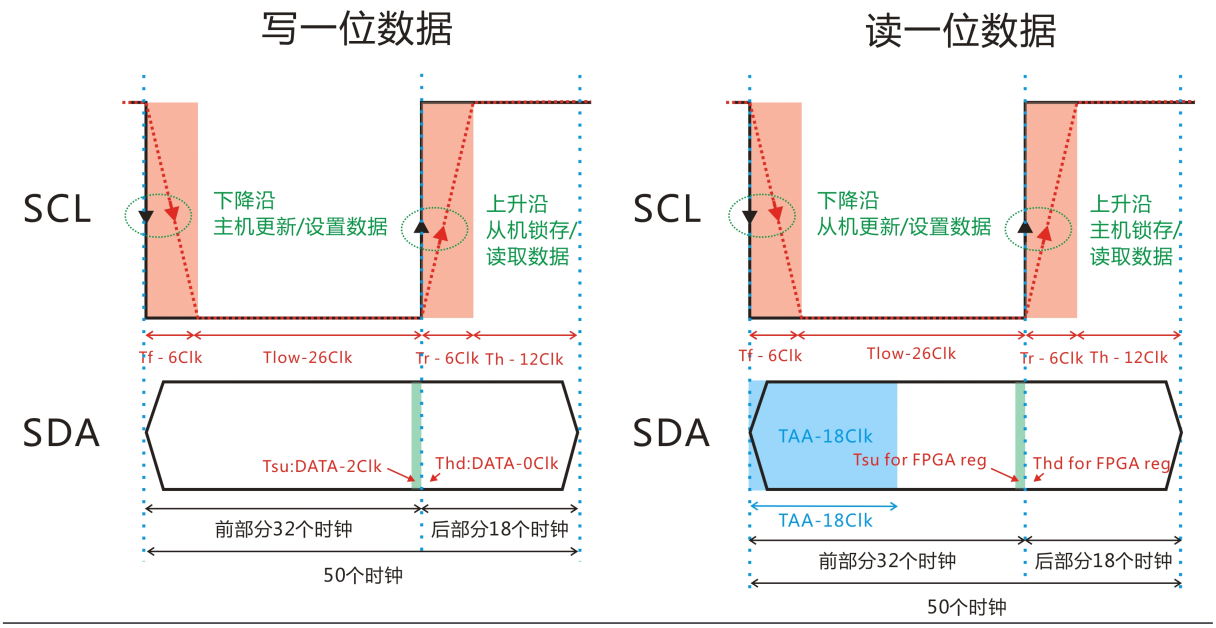


图 3.5b 写一位数据和读一位数据。

图 3.5b 是 iic 高速模式下的写一位数据和读一位数据的混合时序图（理想和物理）。红色虚线是物理时序图，反之黑色实线是理想时序图。图 3.5b 完全是按表 3.5a 来绘制，其中我们可以看到，速率的前部分占 32 个时钟，速率的后部分占 18 个时钟 ... 这已经是占空比为 36%的 SCL 时钟信号（别忘了速率是由 SCL 时钟信号来决定）。不过，一位数据的周期时间依然是保持 50 个时钟，这一点和理论上的一样。

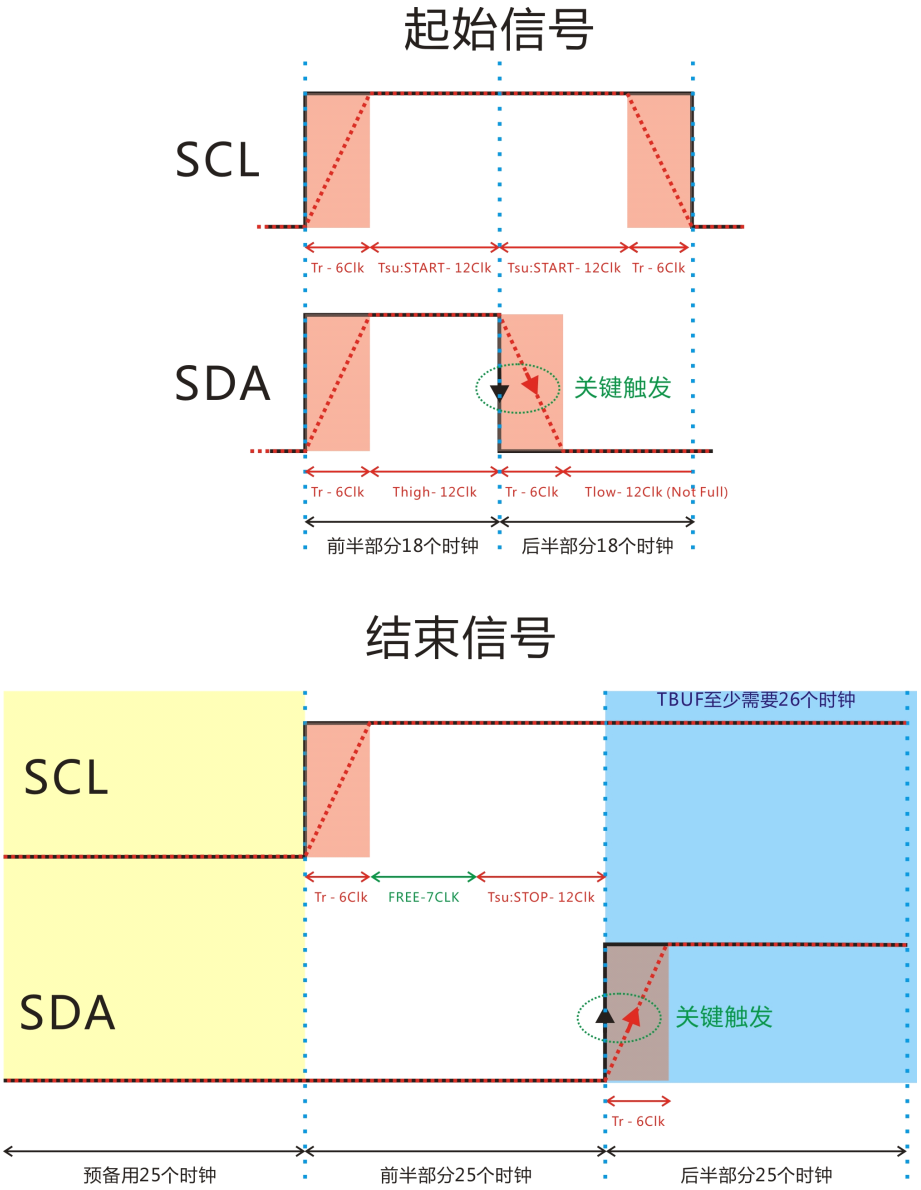


图 3.5c 起始信号和结束信号。

在这里，估计有些朋友会看到傻眼了。图 3.5c 中的起始信号和结束信号似乎有点乱来：

第一起始信号只有 36 个时钟而已，呵呵！这是笔者故意干得，目的是要读者明白，只要满足时间参数，起始信号要多短就有多短；

第二是结束信号的预备用时间和 TBUF 的时间参数。预备用时间是按理论上 1/2 速率的周期，而不是 1/4 速率的周期，为什么？答案很简单呀，为了更保险 ... 目前 iic 总线是以 400kHz 的速率在运动，这就好比读者驾驶超过 100 Km/h 一样。还记得笔者说过什么吗？一点关于 iic 驱动的好坏就在乎这一段预备用的时间。

第三，按照习惯结束信号的前半部分应该是 18 个时钟才对 .... 呵呵，这也故意的！只要满足了 Tsu:STOP 和 Tr 等时间参数以外，多几个时钟也不要紧。

最后有关 TBUF 时间少了一个时钟（时序表指出是 26 个时钟，图 3.5 中笔者只满足 25 个时钟而已），这一点读者就无视吧，关于 TBUF 这个时间参数比较松散，少了一个时钟也不会少了一根骨头（不会坏了大事的意思）。好了，我们还是直接来看代码吧：

iic_func_module.v

```
1.    module iic_func_module
2.    (
3.            input CLK,
4.            input RSTn,
5.
6.            input [1:0]Start_Sig,
7.            input [7:0]Addr_Sig,
8.            input [7:0]WrData,
9.            output [7:0]RdData,
10.           output Done_Sig,
11.
12.           output SCL,
13.           inout SDA
14.
15.           /***************/
16.           //output [4:0]SQ_i
17.
18.   );
19.
20.         /*************************/
21.
22.           parameter FREQ = 7'd50; // 400kHz
23.
24.           /*************************/
25.
26.           reg [4:0]i;
27.           reg [4:0]Go;
28.           reg [6:0]C1;
29.           reg [7:0]rData;
```

```
30.         reg rSCL;
31.         reg rSDA;
32.         reg isAck;
33.         reg isDone;
34.         reg isOut;
35.
36.         always @ ( posedge CLK or negedge RSTn )
37.             if( !RSTn )
38.                 begin
39.                     i <= 5'd0;
40.                     Go <= 5'd0;
41.                     C1 <= 7'd0;
42.                     rData <= 8'd0;
43.                     rSCL <= 1'b1;
44.                     rSDA <= 1'b1;
45.                     isAck <= 1'b1;
46.                     isDone <= 1'b0;
47.                     isOut <= 1'b1;
48.                 end
```

第 1~49 行是都是一些琐碎的事情，如输入输出口声明，建立相关寄存器，复位县官寄存器。其中需要注意的是第 22 行的常量声明，由于现在要驱动的 iic 总线是高速模式，亦即 400Khz，经过 20Mhz 量化以后是 50 个时钟。

```
49.     else if( Start_Sig[0] )
50.                 case( i )
51.
52.                     0: // Start
53.                     begin
54.                         isOut = 1;
55.
56.                         rSCL <= 1'b1;
57.
58.                         if( C1 == 0 ) rSDA <= 1'b1;
59.                         else if( C1 == 18 ) rSDA <= 1'b0;
60.
61.                         if( C1 == 36 -1) begin C1 <= 7'd0; i <= i + 1'b1; end
62.                         else C1 <= C1 + 1'b1;
63.                     end
64.
65.                     1: // Write Device Addr
66.                     begin rData <= {4'b1010, 3'b000, 1'b0}; i <= 5'd7; Go <= i + 1'b1; end
67.
68.                     2: // Wirte Word Addr
```

```
69.                     begin rData <= Addr_Sig; i <= 5'd7; Go <= i + 1'b1; end
70.
71.                     3: // Write Data
72.                     begin rData <= WrData; i <= 5'd7; Go <= i + 1'b1; end
73.
74.                     /*************************/
75.
76.                     4: // Stop
77.                     begin
78.                             isOut = 1'b1;
79.
80.                             if( C1 == 0 ) rSCL <= 1'b0;
81.                             else if( C1 == 25 ) rSCL <= 1'b1;
82.
83.                             if( C1 == 0 ) rSDA <= 1'b0;
84.                             else if( C1 == 50 ) rSDA <= 1'b1;
85.
86.                             if( C1 == 25 + 50 -1 ) begin C1 <= 7'd0; i <= i + 1'b1; end
87.                             else C1 <= C1 + 1'b1;
88.                     end
89.
90.                     5:
91.                     begin isDone <= 1'b1; i <= i + 1'b1; end
92.
93.                     6:
94.                     begin isDone <= 1'b0; i <= 5'd0; end
95.
```

第 49~94 行是写功能，修改的有：步骤 0 的起始信号（第 54~62 行）；步骤 4（第 77~88 行）的结束信号；其余的如同实验十六一样。至于步骤 0 和步骤 4 被修改过什么，一切尽在图 3.5c 之中（自己看着办的意思）。

```
96.                     /******************************/ //function
97.
98.                     7,8,9,10,11,12,13,14:
99.                     begin
100.                            isOut = 1'b1;
101.                            rSDA <= rData[14-i];
102.
103.                            if( C1 == 0 ) rSCL <= 1'b0;
104.                            else if( C1 == 32 ) rSCL <= 1'b1;
105.
106.                            if( C1 == FREQ -1 ) begin C1 <= 7'd0; i <= i + 1'b1; end
107.                            else C1 <= C1 + 1'b1;
```

```
108.                    end
109.
110.                    15: // waiting for acknowledge
111.                    begin
112.                            isOut = 1'b0;
113.                            if( C1 == 40 ) isAck <= SDA;
114.
115.                            if( C1 == 0 ) rSCL <= 1'b0;
116.                            else if( C1 == 32 ) rSCL <= 1'b1;
117.
118.                            if( C1 == FREQ -1 ) begin C1 <= 7'd0; i <= i + 1'b1; end
119.                            else C1 <= C1 + 1'b1;
120.                    end
121.
122.                    16:
123.                    if( isAck != 0 ) i <= 5'd0;
124.                    else i <= Go;
125.
126.                    /******************************/ // end function
127.
128.            endcase
129.
```

第 96~128 行是 写一字节 + 应答位 处理的伪函数，其中被修改的有步骤 7~14（第 98~108 行）的写一字节操作和步骤 15（第 110~120）的应答位处理的操作。在此读者只要明白主机和从机之间如何读一位数据和写一位数据，然后再配合图 3.5b 作为后盾，自然而然会看懂被修改的地方（还是自己看着办的意思）。

```
130.        else if( Start_Sig[1] )
131.                case( i )
132.
133.                    0: // Start
134.                    begin
135.                            isOut = 1;
136.
137.                            rSCL <= 1'b1;
138.
139.                            if( C1 == 0 ) rSDA <= 1'b1;
140.                            else if( C1 == 18 ) rSDA <= 1'b0;
141.
142.                            if( C1 == 36 -1 ) begin C1 <= 7'd0; i <= i + 1'b1; end
143.                            else C1 <= C1 + 1'b1;
144.                    end
145.
```

```
146.                    1: // Write Device Addr
147.                    begin rData <= {4'b1010, 3'b000, 1'b0}; i <= 5'd9; Go <= i + 1'b1; end
148.
149.                    2: // Wirte Word Addr
150.                    begin rData <= Addr_Sig; i <= 5'd9; Go <= i + 1'b1; end
151.
152.                    3: // Start again
153.                    begin
154.                            isOut = 1'b1;
155.
156.                            rSCL <= 1'b1;
157.
158.                            if( C1 == 0 ) rSDA <= 1'b1;
159.                            else if( C1 == 18 ) rSDA <= 1'b0;
160.
161.                            if( C1 == 36 -1 ) begin C1 <= 7'd0; i <= i + 1'b1; end
162.                            else C1 <= C1 + 1'b1;
163.                    end
164.
165.                    4: // Write Device Addr ( Read )
166.                    begin rData <= {4'b1010, 3'b000, 1'b1}; i <= 5'd9; Go <= i + 1'b1; end
167.
168.                    5: // Read Data
169.                    begin rData <= 8'd0; i <= 5'd19; Go <= i + 1'b1; end
170.
171.                    6: // Stop
172.                    begin
173.                            isOut = 1'b1;
174.
175.                           if( C1 == 0 ) rSCL <= 1'b0;
176.                            else if( C1 == 25 ) rSCL <= 1'b1;
177.
178.                            if( C1 == 0 ) rSDA <= 1'b0;
179.                            else if( C1 == 50 ) rSDA <= 1'b1;
180.
181.                            if( C1 == 25 + 50 -1 ) begin C1 <= 7'd0; i <= i + 1'b1; end
182.                            else C1 <= C1 + 1'b1;
183.                    end
184.
185.                    7:
186.                    begin isDone <= 1'b1; i <= i + 1'b1; end
187.
188.                    8:
189.                    begin isDone <= 1'b0; i <= 5'd0; end
```

```
190.
```

第 130~189 行是读功能的基本操作，其中被修改的有步骤 0（第 133~144）的起始信号；步骤 3（第 152~163 行）主机改变方向用的起始信号；步骤 6（第 171~183 行）的结束信号。基本上步骤 0 的起始信号和步骤 6 的结束信号被修改的地方和写功能完全一样。至于步骤 3 主机改方向涌的起始信号，完全是从步骤 0（第 133~144 行）拷贝下来，在此笔者故意拿掉了“前后被拉低的一段时间”（读者请回忆实验十六）。如果读者担心这样作会不会导致驱动失败 ... 放一百个大心，这样写也是没有问题的。

```
191.  /******************************/ //function
192.
193.                    9,10,11,12,13,14,15,16:
194.                    begin
195.                        isOut = 1'b1;
196.
197.                        rSDA <= rData[16-i];
198.
199.                        if( C1 == 0 ) rSCL <= 1'b0;
200.                        else if( C1 == 32 ) rSCL <= 1'b1;
201.
202.                        if( C1 == FREQ -1 ) begin C1 <= 7'd0; i <= i + 1'b1; end
203.                        else C1 <= C1 + 1'b1;
204.                    end
205.
206.                    17: // waiting for acknowledge
207.                    begin
208.                        isOut = 1'b0;
209.
210.                        if( C1 == 40 ) isAck <= SDA;
211.
212.                        if( C1 == 0 ) rSCL <= 1'b0;
213.                        else if( C1 == 32 ) rSCL <= 1'b1;
214.
215.                        if( C1 == FREQ -1 ) begin C1 <= 7'd0; i <= i + 1'b1; end
216.                        else C1 <= C1 + 1'b1;
217.                    end
218.
219.                    18:
220.                    if( isAck != 0 ) i <= 5'd0;
221.                    else i <= Go;
222.
223.                    /****************************/
224.
```

第 191~221 行是读功能的 写一字节 + 应答位 处理的伪函数，和写功能的伪函数没什么两样，自己看着办吧。

```
225.                    19,20,21,22,23,24,25,26: // Read
226.                    begin
227.                            isOut = 1'b0;
228.
229.                            if( C1 == 40 ) rData[26-i] <= SDA;
230.
231.                            if( C1 == 0 ) rSCL <= 1'b0;
232.                            else if( C1 == 32 ) rSCL <= 1'b1;
233.
234.                            if( C1 == FREQ -1 ) begin C1 <= 7'd0; i <= i + 1'b1; end
235.                            else C1 <= C1 + 1'b1;
236.                    end
237.
238.                    27: // no acknowledge
239.                    begin
240.                            isOut = 1'b1;
241.                            //if( C1 == 40 ) isAck <= SDA;
242.
243.                            if( C1 == 0 ) rSCL <= 1'b0;
244.                            else if( C1 == 32 ) rSCL <= 1'b1;
245.
246.                            if( C1 == FREQ -1 ) begin C1 <= 7'd0; i <= Go; end
247.                            else C1 <= C1 + 1'b1;
248.                    end
249.
250.                    /*************************************/ // end fucntion
251.
252.            endcase
253.
```

第 225~252 行是 读一字节 + 应答位 处理的操作，被修改的地方有步骤 19~26（第 225~236 行）的读一字节操作，还有步骤 27（第 238~248 行）的应答位处理操作（这个步骤可以无视）。修改的内容只要看懂图 3.5b 自然而然会明白其中意义（也是自己看着办的意思）。

```
254.  /***************************************/
255.
256.          assign Done_Sig = isDone;
257.          assign RdData = rData;
258.          assign SCL = rSCL;
259.          assign SDA = isOut ? rSDA : 1'bz;
```

```
260.
261.        /*************************************/
262.
263.          //assign SQ_i = i;
264.
265.          /******************************/
266.
267. endmodule
```

第 254~267 行就是一些输出的驱动而已 ....

exp17_demo.v

这是下载到开发板测试用的代码，基本上和实验十五~十流一样，修改的地方不多，所以笔者就不贴出代码了。还有实验十七笔者也懒得去仿真了，核心内容基本上和实验十六，仿真了也只是换汤不换药 ...

==========

这一小节要讨论的内容基本上就是这些东西而已，速率之所以有正规和非正规之分，这些全部都取决与硬件自身的设计，作为使用者我们什么也作不了，只能随波即流。此外，在 iic 总线的普通模式和高速模式之间还有一个相互兼容的微妙关系。基本上，官方给出的时序表我们要遵守也可以，不遵守也可以，只遵守一点也可以，时序表的作用只是标准的参考性而已，而不是要死头死脑般去遵守。

除了这一点以外，iic 总线在普通模式和高速之间操作时可以相互“兼容”。说得具体就是，实验十六的步骤 0（起始信号）和实验十七的步骤 0（起始信号）可以相互交换 ... 其他的步骤也是如此。这句话听了是不是觉得太神，太不可思议了？事实就是如此，这好比另个同样形状的拼图，一个是红色，另一个是蓝色，可是 iic 总线在乎的只是形状而不是颜色，随便一个颜色的拼图只要安插在正确的地方，iic 就可以运作。

关于这一点，信不信由己，反正笔者已经成功测试了 ... 至于稳定性方面笔者还不知道非正规速率的影子在许多的硬件上常常会出现，所以多少也要认识一下。好了这一小节的内容就到此为止了。

3.6 不同世界的模块

在低级建模的固有风格里，每一个模块都离不开“时钟”，或者说时钟是模块需要消耗的动力。但是有一点读者必须晓得的是，我们所建立的模块全部都是纳入 FPGA 的内部。读者尝试想像看，如果模块建立在 FPGA 的外部到底会是什么样的情形呢？

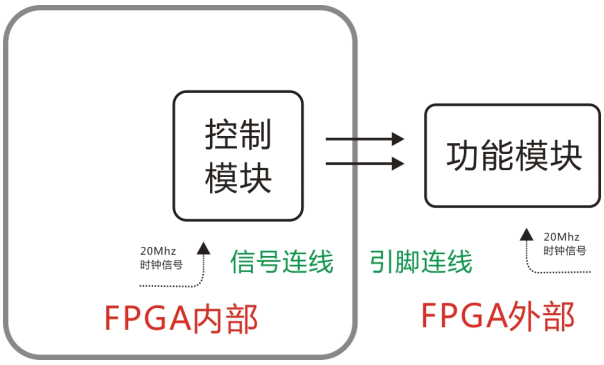


图 3.6a 不同世界的模块。

如图 3.6a 所示，图中有两个模块，一个是建立在 FPGA 内部的控制模块，另一个是建立在 FPGA 外部的功能模块。内部的控制将输出连线至内部的引脚配置，然后由内部的引脚经过 IO 口连线至外部的功能模块。假设两个模块都使用相同的时钟频率，可能有同学那样想：“如果两个模块都使用相同的时钟频率，那么应用在理想时序(Verilog 的理想很思绪）的时钟点事件是否可以被实现与它们之间呢？”

对！这就是这一小节要讨论的内容，此外这也是 TimeQuest 静态时序分析最初的问题。很遗憾的是，笔者不打算在这里谈论有关 TimeQuest 的问题。为什么呢？原因很简单呀，谁叫这本笔记名为 Verilog HDL 那些事儿嘛。

随后可能又有同学会那样想：“图 3.6a 中的功能模块到底是什么样的硬件资源呢？”.... 哦，这家伙就是臭名昭著的 SDRAM 了。SDRAM 基本上有 3 个要完成的功课，第一是基本功能的驱动；第二是附加功能的实现；第三是静态时序分析。读者千万不要被吓到，在这里我们只设计第一项功课而已，其外的两项功课已经超过讨论的范围了，笔者只会针对它们然后提出问题而已。

话说回了 SDRAM 到底有什么样的基本功能要我们去实现的呢？此外，我们如何在不同世界的模块之间应用 Verilog 的理想时序呢？等，这些笔者都会一一陪同读者去揭晓。

实验十八（一） SDRAM 基本功能介绍

SDRAM 的作为基本上就是为处理器提供一个庞大，经济而且高效的临时空间。当然这些也只是发生在顺序操作中的事情，在并行操作中如 Verilog HDL 语言，SDRAM 的作为就有点尴尬了。第一 SDRAM 的内部操作是没有固定的时钟，第二 SDRAM 一次只能

为一个控制器/处理器服务而已 .... 无论是第一还是第二个原因，都是并行操作（建模中）最大的问题。啊！批评有点过分了 ....

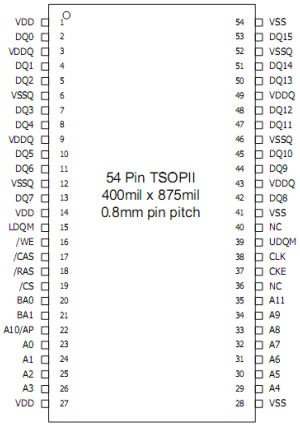


图 3.6b 引脚分布

SYMBOL	TYPE	DESCRIPTION
CLK	Clock	The system clock input. All other inputs are registered to the SDRAM on the rising edge of CLK
CKE	Clock Enable	Controls internal clock signal and when deactivated, the SDRAM will be one of the states among power down, suspend or self refresh
$\overline{CS}$	Chip Select	Enables or disables all inputs except CLK, CKE, UDQM and LDQM
BA0, BA1	Bank Address	Selects bank to be activated during $\overline{RAS}$ activity Selects bank to be read/written during $\overline{CAS}$ activity
A0 ~ A11	Address	Row Address: RA0 ~ RA11, Column Address: CA0 ~ CA7 Auto-precharge flag: A10
$\overline{RAS}$, $\overline{CAS}$, $\overline{WE}$	Row Address Strobe, Column Address Strobe, Write Enable	$\overline{RAS}$, $\overline{CAS}$ and $\overline{WE}$ define the operation Refer function truth table for details
UDQM, LDQM	Data Input/Output Mask	Controls output buffers in read mode and masks input data in write mode
DQ0 ~ DQ15	Data Input / Output	Multiplexed data input / output pin
VDD / VSS	Power Supply / Ground	Power supply for internal circuits and input buffers
VDDQ / VSSQ	Data Output Power / Ground	Power supply for output buffers
NC	No Connection	No connection

表 3.6a 引脚说明。

图 3.6b 是 SDRAM 的引脚分布，表 3.6a 是引脚说明。CLK 是用来提供时钟；CKE 是用来使能 CLK 信号；CS 是用来使能 SDRAM；BA0~1 是用来选择内部的资源库；A0~A11 列选和行公用的地址信号；RAS,CAS,WE 等信号配合 CKE 和 CS 信号可以组成相关的命令；UDQM，LDQM 是数据掩码；DQ0~DQ15 是数据信号；

首先笔者先解释信号 BA0~1 和信号 A0~A11 它们之间的关系：

黑金开发板上使用的 SDRAM 是 HY57V641620E(L/S)T(P)-7，其中的特征有：支持频率达到 143Mhz；64Mbit 的空间；SDRAM 内部有 4 个资源库，每一个资源库的寻址深度是 2^{12} 行 * 2^{8} 列，亦即 1M 的寻址范围。然而 SDRAM 一字的数据宽度是 16 位，所以：

SDRAM 的空间 = 4（资源库-Bank）× 2^{12} 行 × 2^{8} 列（寻址深度）× 16 bits
= 4 × 1048576 × 16 bits
= 4 × 1048576 / 1024 × 16 bits
= 4 × 1024 K / 1024 × 16bits
= 4 × 1M × 16 bits
= 64Mbits 或者 67,108,864 bits

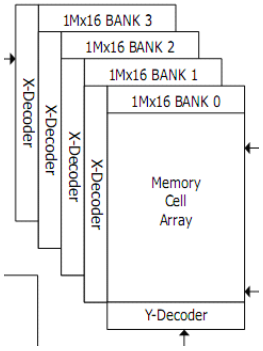


图 3.6c 拥有 4 个 1M x 16 位 资源库的 SDRAM。

BA0~1 (BA - 2^2) 是用来选择哪一个资源库, 然而 1M 是 2^{12} 行 × 2^8 列的寻址深度。至于每一个寻址都指向一字为 16 位的空间。笔者再强调一下，行寻址和列寻址共享使用 A0~A11 的信号，至于具体的过程是如何发生，笔者会在后面讲解的。

相关命令	常量名	CKE	CS	RAS	CAS	WE
Mode Register Set	_LMR	1	0	0	0	0
No Operation	_NOP	1	0	1	1	1
Bank Active	_ACT	1	0	0	1	1
Read with Auto Precharge	_RD	1	0	1	0	1
Write with Auto Precharge	_WR	1	0	1	0	0
Auto Refresh	_AR	1	0	0	0	1
Precharge/Precharge All Bank	_PR	1	0	0	1	0

注意 CS，RAS，CAS 和 WE 等信号都是低电平有效。

表 3.6b SDRAM 常用的命令。

CKE，CS，CAS，RAS 和 WE 这五个信号经过组合，会形成不同作用的命令，常用的命令如表 3.6b 所示。读者是不是看不明白这些命令是干什么的吗？没关系，在此只要先了解表 3.6b 是 SDRAM 常用的几个命令就可以了。

最有有关一点就是 UDQM 和 LDQM 数据掩码信号，官方的定义是：在写操作的时候如果 UDQM 和 LDQM 拉高，写入数据会被屏蔽，亦即 DQ0~DQ15 的写入大门会被关闭，写入失败或者被屏蔽码替代（屏蔽码一般都是 0x00 或者 0xff）；在读操作的时候如果 UDQM 和 LDQM 被拉高，读出数据会被屏蔽为高祖太，亦即 DQ0~DQ15 的读出大门会被关闭，读出失败或者会读出屏蔽码（屏蔽码一般都是 0x00 或者 0xff）。用简单的话来说 UDQM 和 LDQM 可以充当数据写入和读出的钥匙。

==========

在还没有进入正题之前，我们先来考虑这样一个问题：“SDRAM 到底有没有速率？”答案是非常肯定的，SDRAM 也有速率，然而 SDRAM 的速率是由 CLK 时钟信号来决定。不过“SDRAM 的速率”和“iic 总线的速率”之间区别的是“速率的比例”。

iic 总线的速率经过 20Mhz 的频率量化以后，普通模式得到 200 的比例值（亦即一个 100kHz 的速率等于 200 个 20Mhz 的速率），高速模式得到 50 的比例值（亦即一个 400Khz 的速率等于 50 个 20Mhz 的速率），那么 SDRAM 呢？SDRAM 的速率和 20Mhz 的频率之间是 1:1 的比例值，话句话说 SDRAM 使用着 20Mhz 频率的时钟信号。所以驱动 SDRAM 不像驱动 iic 设备那样，可以使用的时钟不是那么“有余”。

在这里，可能有同学要问：“SDRAM 有没有存在物理时序的时间要求吗？”有！这一定有，不过是 0.5ns, 1.8ns 和 6.0ns 等 Verilog 无法涉及到的时间单位。读者听模糊了吗？好，笔者具体的说明一下 ...

假设我们使用 20Mhz 的时钟频率去驱动 SDRAM，那么 20Mhz 时钟频率最小的时间周

期当然是 50ns。在此，这个 50ns 就成为了我们可以使用的最小单位。这个情况就好比读者去慈善晚会，每一张慈善票价都是以 5 元作为最小单位，然后读者想要买果水喝，正好果水的售价是 4 元 ... 在这样不得已的情况下，读者只能用一张 5 元的慈善票去换取一杯 4 元的果水，其中 1 元是不能被折回的。

所以说，如果 SDRAM 有 7ns 或者 12ns 等时间要求，在 20Mhz 时钟频率的使用情况下，我们都必须以 50ns 去满足（呜呜~真是亏大本 ... ）。此外，还有一点非常麻烦的是，SDRAM 有“两层”的物理时序/时间要求。

“两层？这到底是什么一回事？”估计已经有同学按不了性子破口大问了 .... 冷静点这位同学，很多东西不是表面上看得那么简单，也不是骨子里想的那么复杂，最重要就是如何发现问题和关联性而已。具体的过程到底是如何呢？不要怕，笔者会一步一步陪同读者去认识的。

==========

SDRAM 初始化：

每当 SDRAM 正式被使用之前，它和其他硬件一样都需要经过初始化。SDRAM 初始化的过程如下：

一、发送 _PR（Precharge）命令。
二、满足 tRP 时间要求 至少 20ns。
三、发送 _AR（Auto Refresh）命令。
四、满足 tRFC（tRPC 又名 tRCC）时间要求 至少 63ns。
五、发送 _AR（Auto Refresh）命令。
六、满足 tRFC（tRPC 又名 tRCC）时间要求 至少 63ns。
七、发送 _LMR（Lode Mode Register）命令和相关配置信息。
八、满足 tMRD 时间要求 至少 1 个时钟。

Precharge 命令是指“释放所有资源库”。每当我们要求 SDRAM 往哪一个地址写入什么数据的时候，在寻址发生之际，某个 Bank（资源库）的某行（Row）某列（Column）会被选中（锁定）。在这里 Precharge 的命令就是取消寻址中被选中的资源库（解除锁定），然而 tRP 就是取消锁定的时间要求。

Auto Refresh 命令是指“内部数据自动刷新”。SDRAM 逻辑资源本质就是电容，但是电容会随着时间的流逝会发生“泄电”的现象，亦即逻辑内容流失。在这里刷新的意思就是恢复电容的内容，所谓的自动刷新就是“恢复所有资源库，所有行，所有列电容的内容”。然而 tRFC（tRPC 又名 tRCC）就是自动刷新的时间要求（当然也有手动刷新的指令，就是恢复某资源库，某行，某列特定电容的内容）。

Mode Register

BA1	BA0	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	0	0	0	OP Code	0	0	CAS Latency			BT	Burst Length		

OP Code

A9	Write Mode
0	Burst Read and Burst Write
1	Burst Read and Single Write

Burst Type

A3	Burst Type
0	Sequential
1	Interleave

CAS Latency

A6	A5	A4	CAS Latency
0	0	0	Reserved
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	Reserved

Burst Length

A2	A1	A0	Burst Length	
			A3 = 0	A3=1
0	0	0	1	1
0	0	1	2	2
0	1	0	4	4
0	1	1	8	8
1	0	0	Reserved	Reserved
1	0	1	Reserved	Reserved
1	1	0	Reserved	Reserved
1	1	1	Full Page	Reserved

图 3.6d Mode Register 的相关配置信息。

Lode Mode Register 命令是用来配置 SDRAM 的行为，如图 3.6d 所示。SDRAM 有三种的读写模式，一字读写（Single），突发读写（Burst）和页读写（Page），亦即写满整个资源库。其中 A[2..0]就是设置突发读写的长度，默认下是 4/8。Burst Type 和突发读写有关，默认下是逻辑 0。（在这里我们不会涉及突发读写，因为这是 SDRAM 的第二项功课）

CAS Latency 又称为读潜伏期，亦即数据从 SDRAM 读出之前需要等待的时间，单位为 N 个时钟，默认下为 2。OP Code 是指 SDRAM 的读写行为，逻辑 0 是突发读和写，逻辑 1 是突发读单写，默认下为 0。关于这一点，读者必须知道的是，除非 SDRAM 的驱动作业涉及第二项和第三项的功课，不然的话 OP Code 是 0 是 1,都不影响。因为第一项功课所涉及的读写行为和模式都是一字读写（Single Read and Write）。最后的 tMRD 就是配置 SDRAM 行为所需要用到的时间。

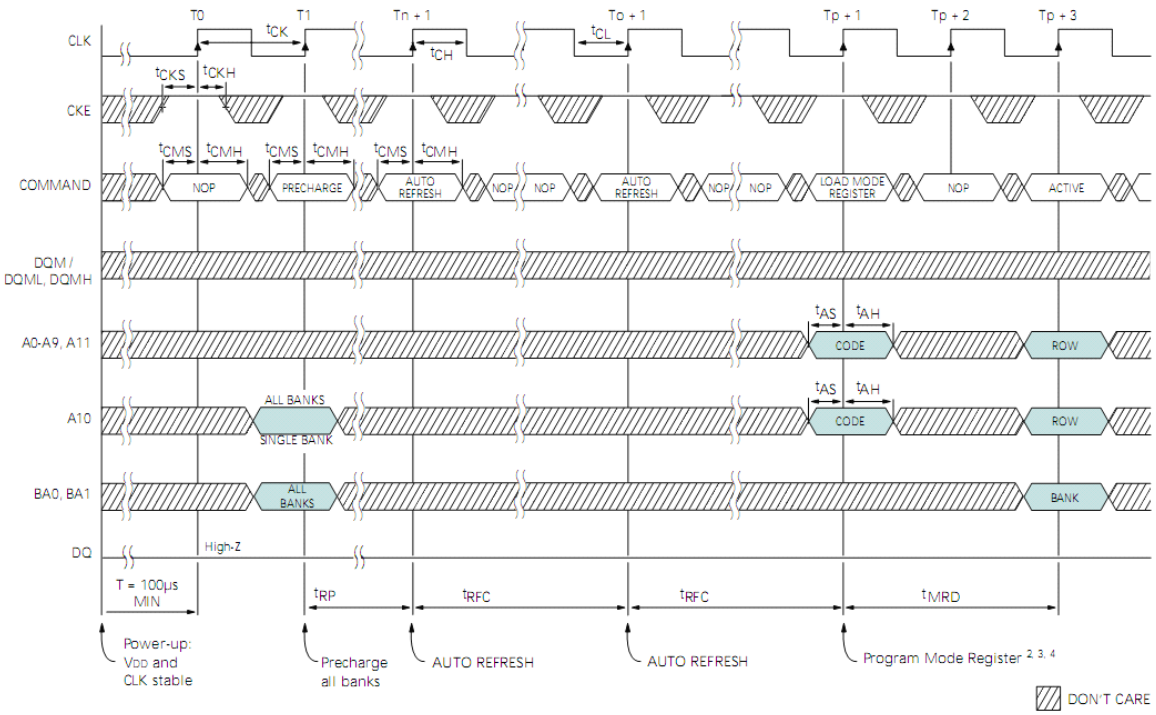


图 3.6e SDRAM 初始化的物理时序图。

图 3.6e 是 SDRAM 初始化的时候所产生的物理时序图，晕了吗？图中 CKE 信号时 Command 信号里的其中一伙，在前笔者已经说过了 Command 信号是由 CKE, CS, RAS, CAS 和 WE 等信号组合而成。同样笔者在前面也说过 SDRAM 有分为两层的时间要求，第一层就是 tRP，tRPC 和 tMRD 等；第二层是 tCMS，tCMH，tAS 和 tAH 等。

第一层称为 SDRAM 操作的基本时间要求；第二层称为硬件自身行为的时间要求（或者路径的延迟时间）。SDRAM 的第一项功课除了基本驱动以外，还有懂的如何去满足第一层的时间要求。反之第二层的时间要求往往都是 1ns，0.8ns 和 4.5ns 等物理时序固有的时间要求，这些都是 SDRAM 的第二项和第三项功课。

在这里，读者需要明白的是，如何“解决”第二层的时间要求，基本上 Verilog HDL 语言已经无能为力了，因为已经超过 Verilog HDL 语言的范围。除非 Verilog HDL 语言所使用的时钟信号周期含有 1ns 级或者 1ps 级，这样的话我们可以套用实验十五~十六学到的知识，用更小的速率去量化更大的速率，取得一定范围的速率比例，不然的话就不要妄想去解决它们 .... 用其他的话来讲，要“解决”第二层的时间要求就必须借助静态时序分析的力量了。在此我们只能“满足”而已（满足又是什么意思呢？）。

哦，话题有点扯远了 ... 图 3.6e 中还给予我们这样一个信息：“SDRAM 在时钟信号的下降沿设置（更新）数据；时钟信号的上升沿（读取）锁存数据 。”那么问题来了，在目前的情况下我们的 FPGA 使用 20Mhz 的频率去驱动 SDRAM 硬件，在此 FPGA 同也是 20Mh，亦即速率的比例是 1:1。

除非我们在 Verilog HDL 语言里使用“双沿触发”，要不然的话我们根本实现不了“时钟信号的下降沿设置（更新）数据；时钟吸纳后的上升沿锁存（读取）数据 ”。可是笔者

又不想使用双沿触发，因为双沿触发容易埋下许多后遗症 ....

==========

补充知识：

在一般的情况下 Verilog 都很乐意把时钟信号的上升沿作为事件的触发 always @ (posedge CLK)。换之双沿触发是指，Verilog 不仅仅是使用时钟信号的上升沿去触发事件，同时也会使用时钟信号的下降沿去触发事件 always @ (posedge CLK or negedge CLK)。

双沿触发有一个假象就是“倍频”，其实不然。双沿触发的主意只是在一个时钟内利用时钟沿的上升沿和下降沿去触发 2 次事件而已。虽然双沿触发可以带来一倍的触发效果，但是双沿触发也是名副其实的双剑刃。在一般的情况，默认下我们都习惯用时钟信号的上升沿作为触发，距离下一次事件的触发就需要等到下一个时钟沿才行，因此数据保持的时间基本上和时钟信号的周期一样（速率的基础概念）。

当我们使用双沿触发的时候，亦即我们必须减半数据的保持时间。这就好比我们得到一倍的利润，同时我们又减半了安全性。不仅而已，双沿触发不但难控制，而且还会大幅度降低代码的表达能力。总结的说，双沿触发不是不可使用，双沿触发就像是一个回报高，同时也伴随着巨大风向的赌博，除非是狗急跳墙的情况，尽可以的话宁可寻找其他的出路。所以，读者在使用双沿触发之前要三思。

==========

“这个也不行，那个也不可以 .... 笔者到底要我怎么办？”嗯，冷静点这位朋友，这个难题也是这一小节里面我们要讨论的东西。为了在学习 TimeQuest 的时候留下更多的伏笔，我们应该善用“Verilog HDL 语言的理想时序”来解决这个难题，这是一个很有趣的思想：

小时候我们常常会想到：“如果我可以天天开心就好了... 如果我和明星一样漂亮就好了 ... 如果我和爱因斯坦一样聪明就好了... ”。可是当我们长大后，我们会受到身边人，环境或者常识等所束缚，如：“天天开心是不可能的因为现实是残酷的 ... 你那么胖不能成为明星 ... 爱因斯坦的出现率是几亿份之一 ...”。

小时候的我们就是无邪，无论什么时候思考的方向都是理想的状态。可是在长大的过程中，许多思想渐渐地开始不理想，而且越来越破烂不堪 .... 笔者偷偷告诉读者，这些都只是“概念上”的问题而已，如果读者懂得抛开传统的概念，带上“理想概念”这顶帽子，很多事情都很容易被圆滑。这也是学习 Verilog HDL 语言的时候需要跨过的另一个“难题”。

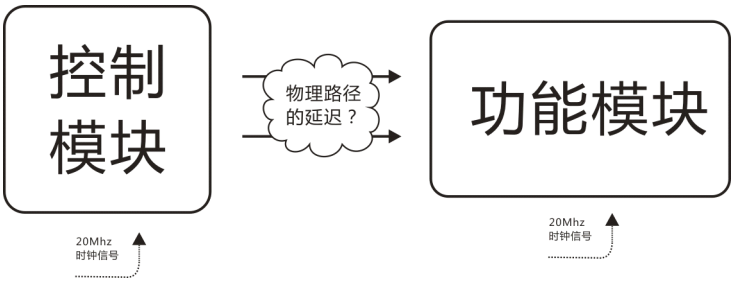


图 3.6f 内部的控制模块和外部的功能模块在沟通。

在图 3.6f 中控制模块是 FPGA 的内部模块，也就是我们要编辑的控制器。图 3.6f 中的功能模块是外部的 SDRAM 模块，也就是我们的驱动对象。无论是控制模块还是功能模块，碍手碍脚的就是“内部和外部之分”这一概念。当我们把“内部和外部之分”这个概念抛开以后，随之会衍生一个问题，那就“物理路径的延迟”一存在。

好了，精彩的推理部分来了。控制模块和功能模块都使用相同的时钟频率，亦即我们可以应用 Verilog 理想时序中的“时间点事件”。其中的问题是我们不知道“物理路径确切的延迟数值”，在此我们可以换另一种概念去思考问题。

假设读者要从 A 点旅行到 B 点，然而旅费可能是 100~300 元之间。一般上，我们不可能带足 300 元去旅行的对吗？但是作为保险我们最终会带上 5000 元或者更多。在此 5000 元就成为理想的值。换另一个角度来说，只要我们的口袋里带有 5000 元，基本上我们不会去考虑“旅费到底足不足？”这一问题。因为 5000 元作为理想值，已近覆盖掉（满足）100~300 元这个芝麻的小数目了。

同样的道理，“物理路径的确切延迟数值是多少？”这个不是我们要思考的问题，只要我们把时序当成是一个理想的状态，无论物理路径的延迟是多少，理想值都可以把它覆盖掉（满足它）。20Mhz 时钟频率拥有 50ns 的时钟周期，这个 50ns 就是笔者所谓的理想值，至于物理路径的延迟一般上都是几纳秒作为单位。对于 50ns 这个理想值来说几纳秒就是一个芝麻的小数目，不足以担心。

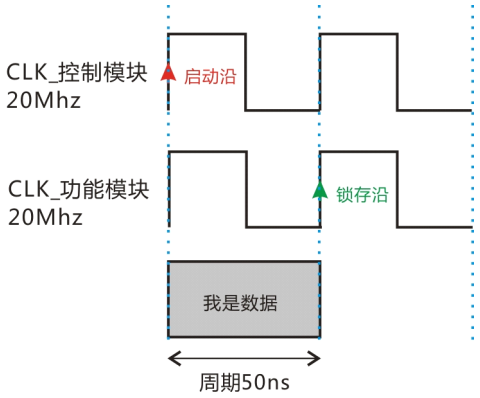


图 3.6g 没有物理路径延迟的理想时序。

笔者还是具体的讲解一下。图 3.6g 的“启动沿”是指控制模块在 T0 的时候数据的上升

沿；反之“锁存沿”是指功能模块在 T1 的时候读取数据的上升沿。（注：启动沿和锁存沿是 TimeQuest 里的居民）。图中是没有任何路径延迟的理想时序，由于控制模块和功能模块使用同样的时钟，所以理想时序的时间点事件就可以成立 。（注意那个理想值 50ns）

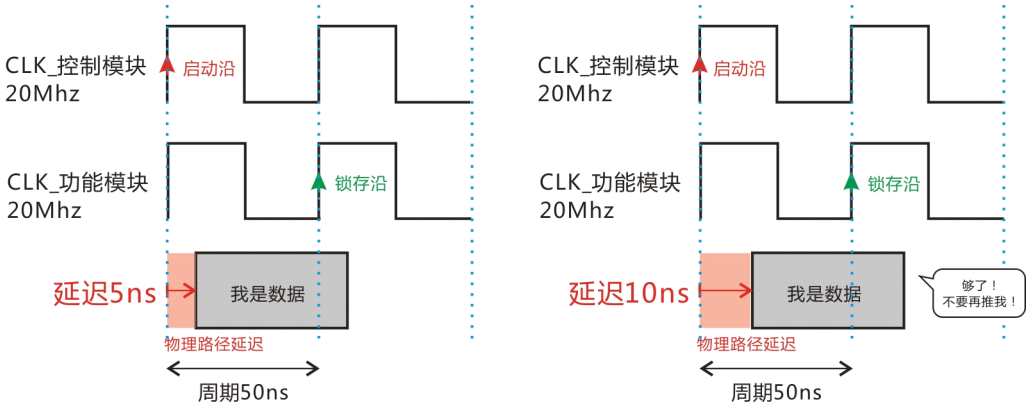


图 3.6h 用物理时序的角度看待时序和物理路径延迟的关系。

图 3.6h 是理想时序把延迟（物理路径）加入计算的时序图。无论物理路径的延迟再怎么推数据，锁存沿都都能触碰（读取）到数据。在这里，读者们需要注意的是“唯有物理时序的看法下，延迟会把数据推向右边”。换句话图 3.6h 是物理时序的表现，或者说是站在物理时序的角度下看待“时序和物理路径延迟的关系”。

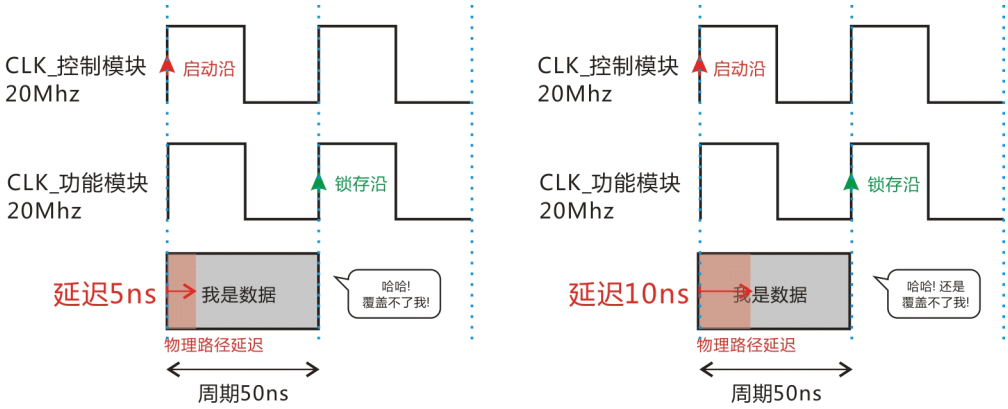


图 3.6i 用理想时序的角度去看待时序和物理路径延迟的关系。

反之，如果读者站在理想时序的角度去看待“物理路径延迟和时序之间的关系”，我们非常乐意使用“覆盖”，而不是“推向”的概念。如图 3.6i 所示，无论“红色矩形（物理路径的延迟）”如何覆盖数据，只要数据还有路出身体的一方，红色矩形永远坏不了大事（已经满足）。这也是笔者在前面所说的“只要口袋有 5000 元，就用不着担心芝麻的 100~300 元，因为 5000 元是一个理想值了”。（同样的概念也应用在实验十六和十七）。

实际上，物理路径的延迟绝对不是图 3.6h~i 中所指那样，5ns~10ns 等那么巨大。一般上如果设计者没有故意的话，物理路径的延迟很难超过 5ns。决定物理路径的延迟数值是有很多因素的，如：连线材料，连线质量，连线距离/宽度，湿度，温度，生锈 ... 等。设计者往往只能经过评估得到可以接受的数值而已。

读者看到这里估计多少也有感觉了吧？Verilog HDL 语言的理想时序是一个见人打人见熊打熊的真武器，不但可以套用在模块的沟通上，同样还可以套用在物理时序延迟的身上。其中图 3.6i 所使用的“覆盖”的概念也可以称为理想时序和物理时序之间的粘糊剂（平衡点）。

其外，图 3.6h 中所使用的“推向”概念常常可以在静态时序分析的或者其他硬件的时序图里看到。反之图 3.6i 所使用的“覆盖”概念皆可以在静态时序分析里看到，同样可以在 Verilog 理想时序的世界里看到。

哎呀，话题又不小心扯那么远了，想知道更多相关的内容，读者自己去翻看 TimeQuest 或者其他和静态时序分析的资料吧。在此，读者只需要明白，FPGA 和 SDRAM 之间，只要彼此使用相同的时钟频率，理想时序中所使用的“时间点事件”就可以应用在它们的身上，就足够了。

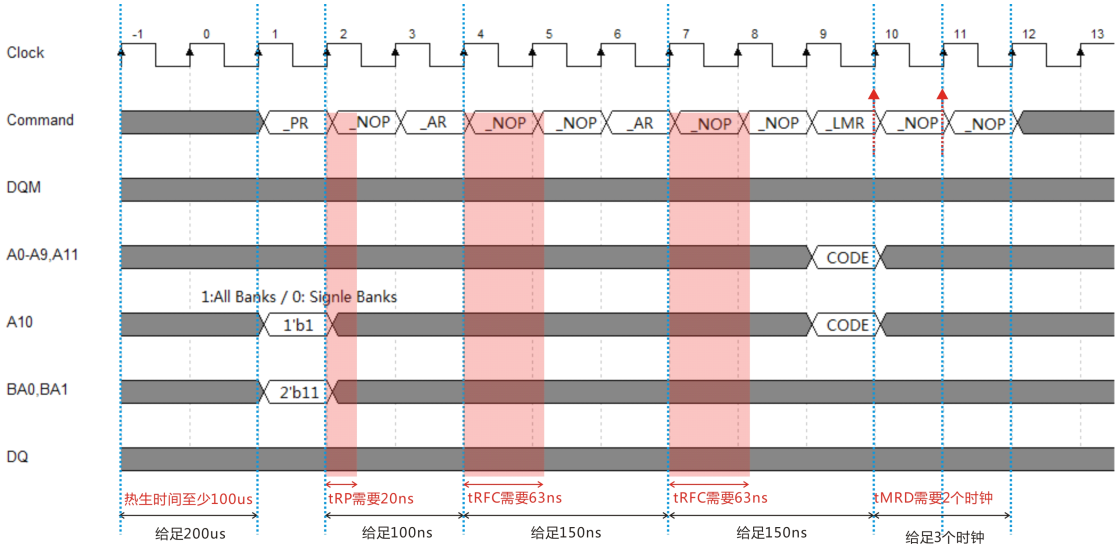


图 3.6j SDRAM 初始化的理想时序图。

图 3.6j 是经过笔者美化—SDRAM 初始化的理想时序图，与图 3.6e 官方给出的物理时序稍微比较一下的话，是不是有天跟地的区别呢？首先，笔者需要强调一下，在此图 3.6j 和图 3.6e 之间没有对错之分，只是出发点不同而已。图 3.6e 是官方经过测试以后给出实际（接近）的物理时序图，反之图 3.6f 是 Verilog HDL 语言需要产生出的理想时序图。顺便再强调一下，接下来的时序图我们需要从“两方”去解读：一是驱动方（FPGA）；二是被驱动方（SDRAM）。

首先 SDRAM 需要至少 100us 的热生时间（SDRAM 上电以后所需要的初始化时间），在此笔者用 200us 来满足它。

在 T1 的时候驱动方（FPGA）发送命令 Precharge，其中 A10 和 BA0~1 伴随相关数据。根据手册当 Precharge 命令发送的时候，如果 A10 为逻辑 1，SDRAM 的所有资源库都会被释放，此时 BA0~1 的选择无效。反之，如果 A10 为逻辑 0，SDRAM 会根据 BA0~1 的选择去释放相关的资源库。此时，被驱动方（SDRAM）没有相关的操作。

在 T2 的时候 FPGA 发送空命令，为了就是给 SDRAM 相关的操作时间。
。此时，SDRAM 接收到命令 Precharge，然后 SDRAM 开始执行释放操作，相关操作需要消耗时间 tRP -20ns。

在 T3 的时候，FPGA 发送 Auto Refresh 命令给 SDRAM，相关的操作需要消耗时间 tRPC - 63ns。此时 SDRAM 已经完成 Precharge 的操作。

在 T4 的时候，FPGA 发送空命令给 SDRAM，为了就是给 SDRAM 相关操作时间。此时 SDRAM 接收到 Auto Refresh 的命令，然后它开始执行相关的工作。

在这里，请问读者有没有看出一种固定的操作模式？当两个模块在沟通的时候，它们之间都离不开一个硬道理，那就是“模块之间的沟通，至少需要 1 个时钟”。然而在图 3.6j 中所示，FPGA 和 SDRAM 之间沟通用的 1 个时钟延迟，却被笔者利用于满足 SDRAM 相关操作的时间要求。举个例子：

在 T2 的时候，SDRAM 接收到在 T1 从 FPGA 发送过来的 Precharge 命令。SDRAM 距离接收，FPGA 下一个发送过来的 Auto Refresh 命令是在 T4。在 T2~T4 之间，SDRAM 拥有 2 个时钟，亦即 100ns。其中 T2~T3 是 FPGA 给予的空时间（_NOP 命令），换之 T3~T4 则是巧妙的利用了 SDRAM 和 FPGA 之间沟通的延迟时间（一个时钟），充当 SDRAM 在执行 Precharge 操作时的“有余（空）”时间。因此在 T2~T4 之间，SDRAM 拥有 100ns 的时间，去完成 20ns 的 Precharge 工作。

同样的故事也发生在 T4~T6 和 T7~T9。在 T4（T7）的时候，SDRAM 接收 FPGA 在 T3（T6）的时候发送过来的 Auto Refresh 命令，然后 SDRAM 开始执行相关的操作。T4~T5（T7~T8）是 FPGA 给足的两个空时间（NOP 命令）。换之 T6~T7（T9~T10）则是，SDRAM 利用了“距离下一个新命令到来之前，把一个时钟的延迟沟通充当操作时间”。因此在 T4~T6（T7~T9）之间，SDRAM 拥有 150ns 的时间，去完成 63ns 的 Auto Refresh 操作。

在 T9 的时候 FPGA 发送了 Load Mode Register 命令，这个命令伴随相关的 CODE 操作。在此，CODE 是完全对应图 3.6d，其中 A11~A0 的值是：

```
A11~A0 = {
            4'd0 ,
            1'b0，（Op - Burst Read and Burst Write）
            2'd0，
            3'b010，（CAS Latency - 2）
            1'b0，（Burst Type - Sequential）
            3'b010（Burst Length - 4）
        };
```

在 T10 的时候 SDRAM 接收到 FPGA 在 T9 的时候发送过来的 _LMR 命令和相关 CODE 数据。在这里，SDRAM 执行 Lode Mode Register 操作所需要的时间单位是 N 个时钟，而不是多少 ns。根据手册，Lode Mode Register 操作需要消耗 tMRD - 2 个时钟，其中

时钟的消耗计数时从 SDRAM 读到命令 _LMR 开始算起。因此 T10 为 tMRD 消耗的第一个时钟，随之 T11 则是 tMRD 消耗的第 2 个时钟。

如图 3.6j 所示，FPGA 在 T10 的时候给足一个空时间（_NOP 命令），随后 FPGA 又在 T11 的时候给足另一个空时间（_NOP 命令）。事实上 FPGA 只要给足一个空时间即可（即 T10 这个空命令），但是笔者心太软了，所以笔者多给一个空时间（即 T11 这个空命令）。就这样，SDRAM 的初始化过程就完成了 ... 真是可喜可贺！

定时自动刷新：

SDRAM 资源逻辑的本质就是电容，电容是随着时间的流逝它会不小心泄掉自身的内容。因此 SDRAM 必须每隔一段时间刷新这些电容，使得电容的内容可以得到恢复。刷新有分手动和自动刷新，在这里我们不管前者。黑金搭配的 SDRAM 是 64M bits（4 个资源库 × 1M 寻址深度 × 16 bits）。

根据手册，64M bits 的 SDRAM 每隔 64ms 需要完成 4096 次的自动刷新。也就说，每隔 15.625us 我们必须给 SDRAM 发送一次 Auto Refresh 的命令。不过为了保险起见，我们还是每隔 15us 就给 SDRAM 发送一次 Auto Refresh 命令，这样作的目的除了保险以外，还有偏爱整数的原因。

Auto Refresh 的操作基本上有两种，一种是严谨的（按官方手册要求的手段），另一种是普通的（民间习惯的手段）操作方法。

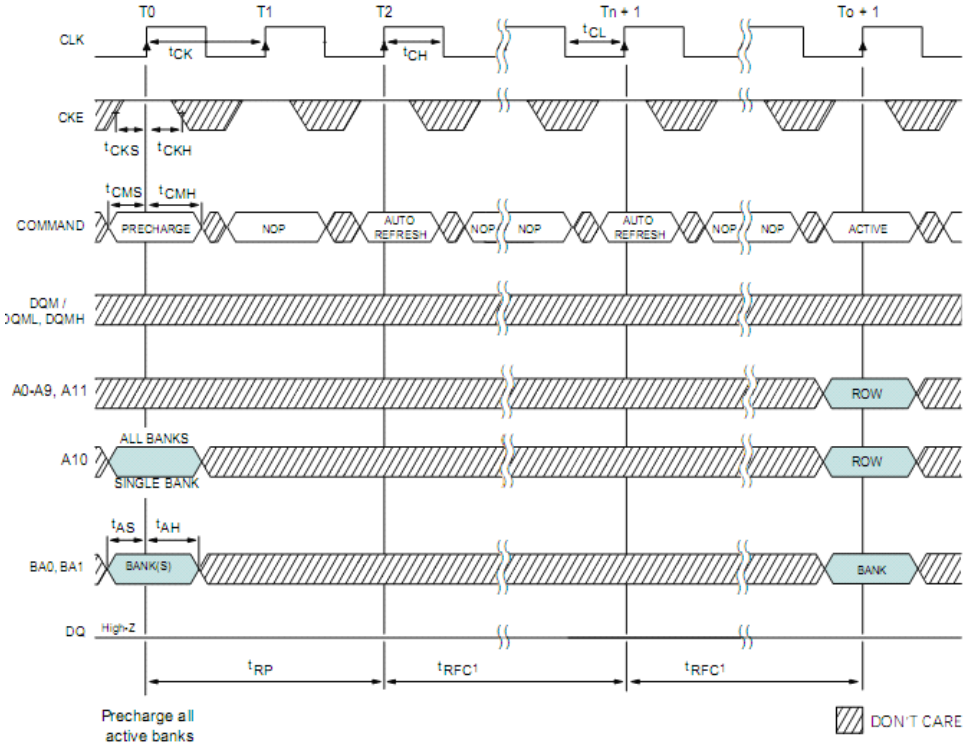


图 3.6k 官方给出的自动刷性过程（物理时序图）。

图 3.6k 中的大意是这样：

一、先发送 Precharge 命令，命 SDRAM 释放所有资源库。
二、相关操作需要消耗时间 tRP - 20ns。
三、发送 Auto Refresh 命令，命 SDRAM 刷新内部逻辑的内容。
四、相关操作需要消耗时间 tRFC - 63ns、
五、发送 Auto Refresh 命令，命 SDRAM 刷新内部逻辑的内容。
六、相关操作需要消耗时间 tRFC - 63ns。

读者有没有觉得有似曾相识的感觉？对，这就是简化以后的初始化过程。如果我们不遵守官方给出的操作步骤，那么我们可以（打从一开始笔者就不打算遵守了:P）....

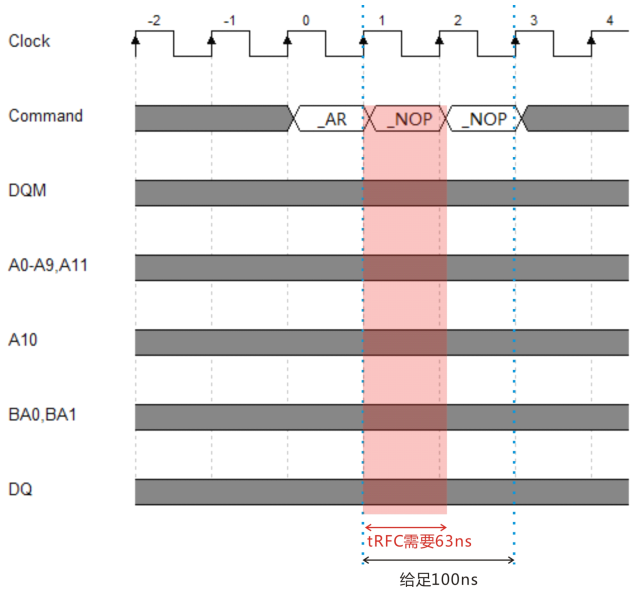


图 3.6l 自动刷新的理想时序图。

图 3.6l 是经过修改以后—自动刷新的理想时序图，此外这也民间惯用的手段。经过简化以后，自动刷新的操作只剩下发送命令 Auto Refresh 以外和满足 tRFC 的时间要求而已。

在 T0 的时候 FPGA 向 SDRAM 发送命令 Auto Refresh。在 T1 的时候 SDRAM 接受到，来至 FPGA 在 T0 的时候所发送的 Auto Refresh 命令，然后开始执行相关的操作。FPGA 随之在 T1 和 T2 发送两个空命令给 SDRAM，使得它有足够的操作时间。就这样，SDRAM 有 100ns 的时间去完成 63ns 的自动刷新操作。

有些同学觉得奇怪，然后问道："笔者那个每隔 15us 自动刷新的操作去哪了？"耐心点这位朋友，在这里我们只要明白"自动刷新是什么？"，"自动刷新如何发生？"等内容就足够。至于"每隔 15us 自动刷新"这是往后的工作，放心！我们一定会遇见的。

写一字操作：

在前面笔者说过 SDRAM 至少有 3 中的读写模式——单读写，突发读写和页读写。突发读写是指，我们给 SDRAM 指定一个地址以后，随着 Clock 增加和根据 Burst Length 的设置，SDRAM 的内部会自动递增被指定的地址。反之页读写，当我们给 SDRAM 指定一个地址以后，随着 Clock 增加，SDRAM 会以这个指定地址为基础把相关的页全部写/读完，除非遇见停止命令。一般上 1 页的范围就是 1 行的所有列，1 页的范围会随着 SDRAM 的架构有所改变。

至于单读写，就每一次读写操作，都会按指定的地址读写 1 字的数据（小知识：一字是指某个硬件的总线位宽，如 SDRAM 的数据总线位宽是 16 位，所以 SDRAM 一字数据是指 16 位数据的意思）。笔者再强调一下，在这里我们只是学习 SDRAM 的第一项功课而已，亦即我们所设计的读写模式是单读写。好了，少罗嗦了，我们进入正题吧：

SDRAM 的读写有“With Auto Precharge”和“Without Auto Precharge”之分。With Auto Precharge 的意思是指，每当一次性的读写发生以后，SDRAM 自动释放相关的资源库，好让下一次的读写操作更为方便。反之 Without Auto Precharge 却没有（意思是手动释放相关的资源库，或者不要）。Without Auto Precharge 明显是师父级的玩意，我们这些懒小鬼就选择 With Auto Precharge 吧，简单又省事。

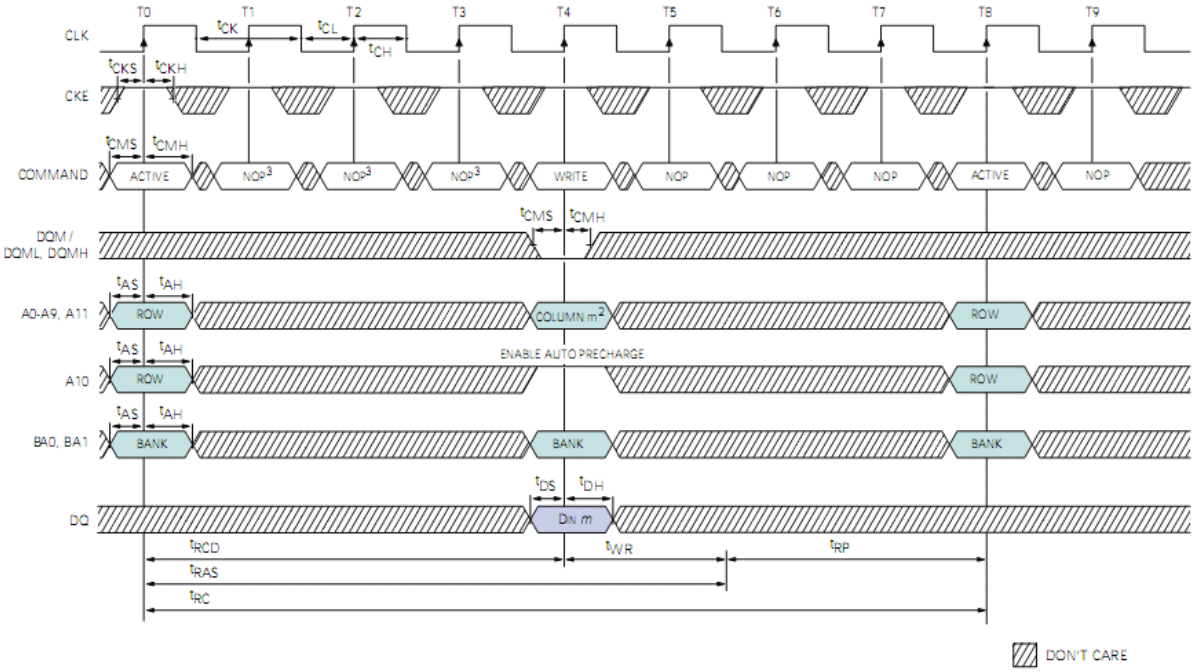


图 3.6m 写一字数据操作的物理时序图。

图 3.6m 是官方给出的写一字数据操作的物理时序图，操作过程大体上是这样的：

一、发送 Active 命令，伴随相关库（Bank）和行（Row）地址信息。
二、满足 tRCD 时间要求，至少 20ns。
三、发送 Write 命令，伴随相关库（Bank）和列（Column）地址；A10 拉高代表 With Auto Precharge；随之也伴随要写入的一字数据。这时候 DQM 必须拉低。
四、满足 tWR（又名 tDPL）时间要求，至少 2 个时钟。

五、满足 tRP 时间要求，至少 20ns。

Active 命令的基本功能是：激活或者选择某个资源库（BA1~0 的库地址）和某行（A11~0 的行地址）。当某个资源库和某行被选择以后，接下来的 Write 命令，会按 A7~A0 的列地址把数据写入（注：这里的 BANK 地址依然不变）。大家注意到那个 A10 被拉高的情形吗？这就是提醒 SDRAM 这次的读/写为 With Auto Precharge。

当经过时间 tWR（又名 tDPL，以 N 个时钟为单位）以后，一字数据就成功被写入。随后 SDRAM 开始执行 Auto Precharge 的操作，亦即释放当前相关的资源库选择。最后经过 tRP 以后（Auto Precharge 的操作完成），一次性的写一字数据操作已经完成。

在这里，唯一让读者捉头发的就是 DQM 这个信号了。很多设计为了方便，始终把 DQM 拉低，DQM 在很多时候可以看成是数据从 SDRAM 读出或者写入的钥匙（拉低有效）。所以说我们要充分的使用它，除了读写操作以外，基本上 DQM 信号一直被拉高，唯有在数据从 SDRAM 读出或者写入的时候，DQM 信号才被拉低（有点像 ram 模块的 En_Sig）。

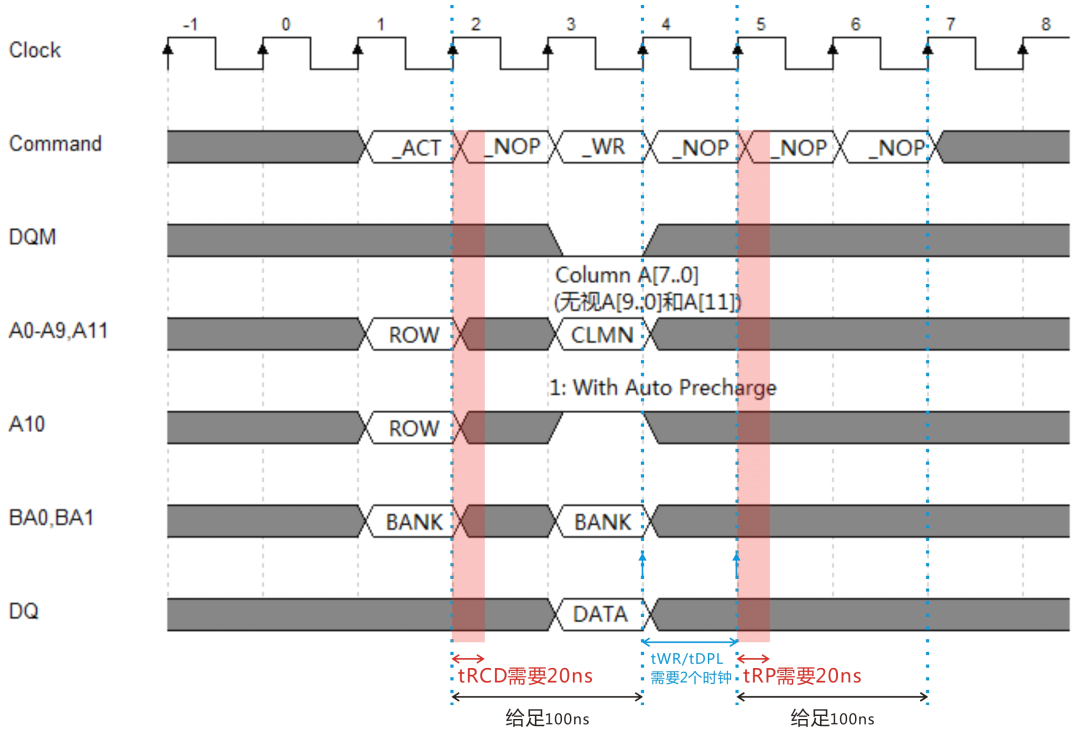


图 3.6n 写一字数据操作的理想时序图。

图 3.6n 是写一字数据操作的理想时序图，乍看下是不是顺眼许多了？呵呵，笔者真懂得老王卖瓜 .... 废话少说，我们开始分析这幅理想时序图。

在 T1 的时候 FPGA 向 SDRAM 发送 Active 命令，随后的是相关 Row 和 Bank 地址信息。

在 T2 的时候 FPGA 向 SDRAM 发送一个空命令，亦即给一个空时间以便 SDRAM 拥有足够的时间执行 Active 操作。在同一个时候，SDRAM 接收到来至 FPGA 在 T1 发送过来的 Active 命令，然后 SDRAM 开始执行相关操作。

在 T2~T3 之间，SDRAM 一共拥有 100ns 的时间去完成 tRCD - 20ns 的 Active 操作。其中 T2 是 SDRAM 给予的空时间，T3 是巧妙利用了两个模块之间沟通的一个时钟延迟，充当空时间。

在 T3 的时候，FPGA 向 SDRAM 发送 Write 命令，随之拉低 DQM 一个时钟，拉高 A10 一个时钟，随后也伴随着相关列地址信息-CLMN，和写入数据-DATA。FPGA 在接下来的 3 个时间（T4~T6）给 SDRAM 发送 3 个空时间，以至于 SDRAM 有足够的时间完成相关操作。

在 T4 的时候，SDRAM 接收到 FPGA 在 T3 所发送过来的 Write 命令和相关的信息，SDRAM 开始执行相关的操作。完成 Write 命令需要满足 tWR/tDPL 两个时钟，tWR/tDPL 从 Write 被接收那一刻算起（T4）。

在 T5 的时候，Write 操作所要求（消耗）的第二个 tWR/tDPL 时钟已经满足，然后 SDRAM 开始执行 Auto Precharge 操作。在 T5~T6 之间，SDRAM 一共给予 100ns 的时间去满足 Auto Precharge 操作所消耗的 20ns（笔者很慷慨）。就这样，一次性的写一字数据就完成了。

读一字数据操作：

读数据操作可以说是 SDRAM 最为关键的一步，SDRAM 驱动失败七不离十就是它搞的鬼。它葫芦里到底卖什么危险的药，我们一步一步去发掘吧。

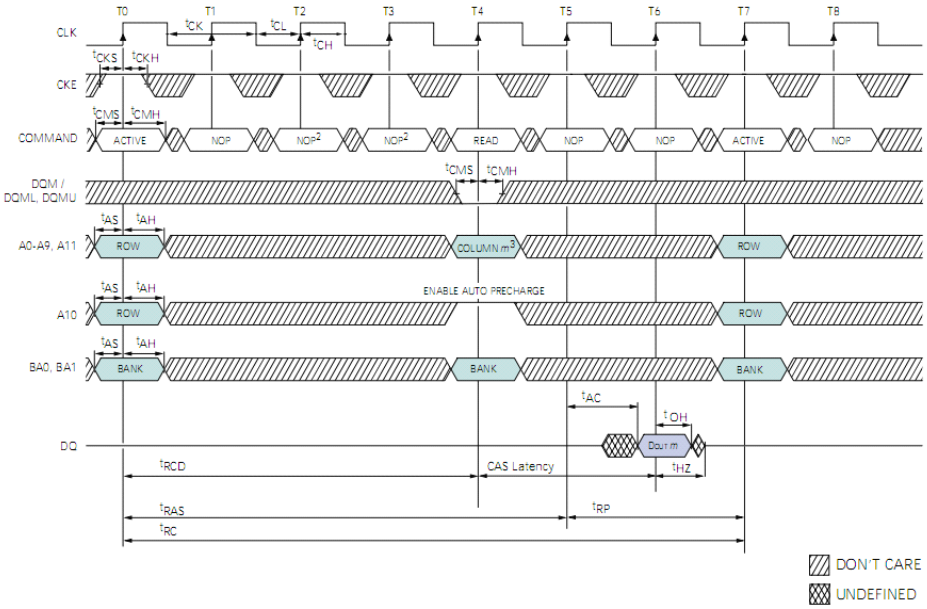


图 3.6o 读一字数据操作的物理时序图。

图 3.6o 是手册给出的读一字数据操作的物理时序图，大致的操作过程如下：

一，发送 Active 命令，伴随行（Row）和资源库（Bank）地址。
二，满足时间要求 tRCD - 20ns。
三，发送 Read 命令，伴随列（Column）和资源库（Bank）地址，其中 DQM 拉低一个时钟，然后拉高 A10 一个时钟，表示读操作后自动释放资源库（With Auto Precharge）。
四，满足时间要求 CAS Latency 2 个时钟。
五，CAS Latency 满足之后，接下来满足时间要求 tAC - 6ns，然后读取数据。
五，CAS Latency 满足之后，接下来满足时间要求 tRP - 20ns。

读一字数据和写一字数据一样，首先发送 Active 命令，选择要从那个资源库和那个行，然后满足相关的时间。接下来发送 Read 命令，根据 Column-列地址信息，指定从那个列读取数据。在此需要拉低 DQM 一个时钟和拉高 A10 一个时钟，前者允许数据从 SDRAM 的 IO 口读出，后者是提醒 SDRAM 读一字数据以后记得释放相关的资源库。最后满足 CAS Latency 2 个时钟（SDRAM 初始化的时候预先配置）。

当 SDRAM 满足 2 个时钟的 CAS Latency，有两个动作同时发生。第一是满足 tAC - 6ns 然后读取数据；第二是满足 With Auto Precharge 所需要的 tRP - 20ns 时间要求。

在这里可能有同学要问："笔者那个 tAC 到底是什么东西？"根据手册的说明，tAC 就是 Access Time From Clock，亦即时钟沿过后所需要的有效时间。如果用傻瓜的话来讲就是：当数据从 SDRAM 的 IO 口读出的时候，由于受到硬件本身的影响，数据需要经过 tAC 这个时间以后才能读出。接下来我们来看 tAC 这个凶手如何干案 ....

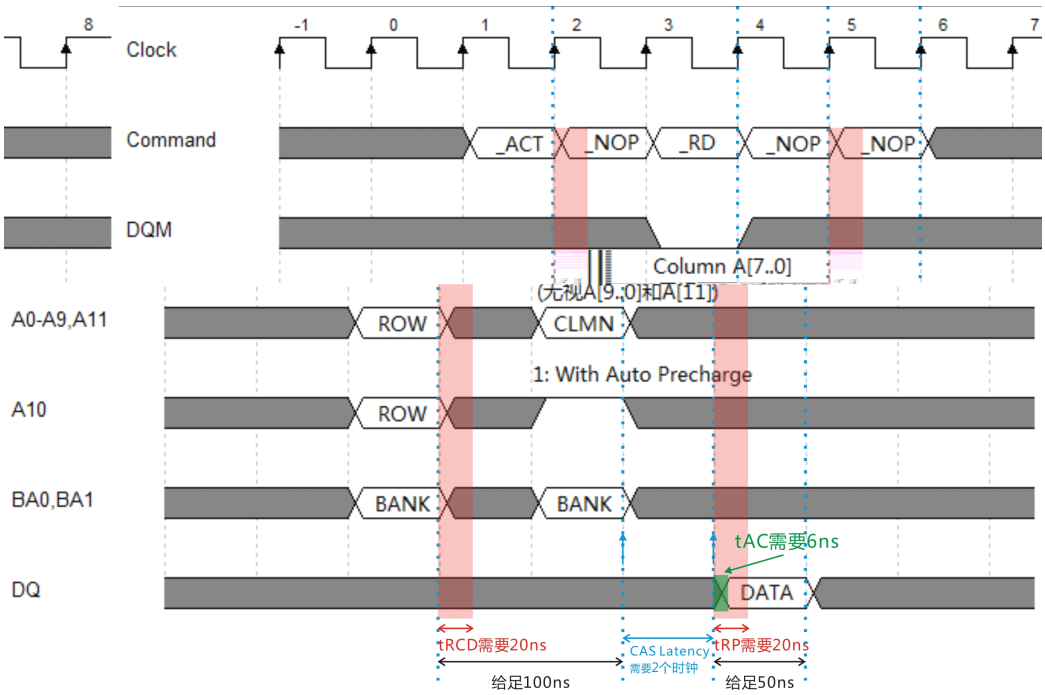


图 3.6p 读一字数据操作的理想时序图。

图 3.6p 是读一字数据操作的理想时序图。基本上图 3.6p 和 3.6n 没有什么两样，唯一让我们困惑的是 CAS Latency 之后的 tAC 时间要求和 tRP 时间要求。正如笔者在前面所说的那样，当 CAS Latecny 时间被满足以后，tAC 和 tRP 时间要求实际上“同时发生”的是两个动作来。tRP 是 Auto Precharge 需要满足的时间要求，这点和写一字操作一样。反之 tAC 则表示，数据“有效”从 SDRAM 读出的时候需要经过的时间，这一点和芯片 24LC04 的 tAA 有几分相似。

在 T1 的时候 FPGA 向 SDRAM 发送 Active 命令，伴随行和资源库的地址信息。

在 T2 的时候 FPGA 向 SDRAM 发送空命令，以至于 SDRAM 有足够的时间完成 Active 操作。在同一个时候，SDRAM 接收到 FPGA 在 T1 发送过来的 Active 命令和相关信息，然后 SDRAM 开始指向相关的操作。

SDRAM 拥有 100 个 ns 的时间（T2~T3）来完成，需要消耗 tRCD - 20ns 的 Active 操作。其中，T2 是 FPGA 发送过来的空指令；换之 T3 是巧妙利用了 FPGA 向 SDRAM 发送下一个命令之间存在的模块沟通延迟（一个时钟）。

在 T3 的时候 FPGA 向 SDRAM 发送 Read 命令，伴随列和资源库的地址信息。此外 FPGA 也拉低 DQM 一个时钟（允许 SDRAM 读出数据）和拉高 A0 一个时钟（读一字数据以后执行释放相关资源库）。 FPGA 随后也发送两个空之间（T4~T5），以至于 SDRAM 有足够的时间完成相关的操作。

在 T4 的时候 SDRAM 读到 FPGA 在 T3 的时候所发送过来的 Read 命令和相关信息。随后 SDRAM 开始指向相关的操作。Read 命令需要满足 CAS Latency 两个时钟，CAS Latecny 的第一个被满足的时钟从 T4 算起，然后 CAS Latency 的第二个时钟在 T5 被满足。

当 CAS Latecny 的时间要求被满足以后，SDRAM 同时发生两个动作，第一就是需要满足 tAC-6ns 时间要求，然后读取数据；第二是满足 tRP - 20ns 的时间要求。我们先来看第一个动作：

SDRAM 的 tAC 时间要求和芯片 24LC04 的 tAA 时间要求基本相似，如果用速率的话来说，iic 总线的速率由 SCL 信号来决定（假设是普通模式-100Khz），SDRAM 的速率则由 CLK 信号来决定（这里是 20Mhz）。前者的数据停留时间是 10us，后者的数据停留时间是 50ns。

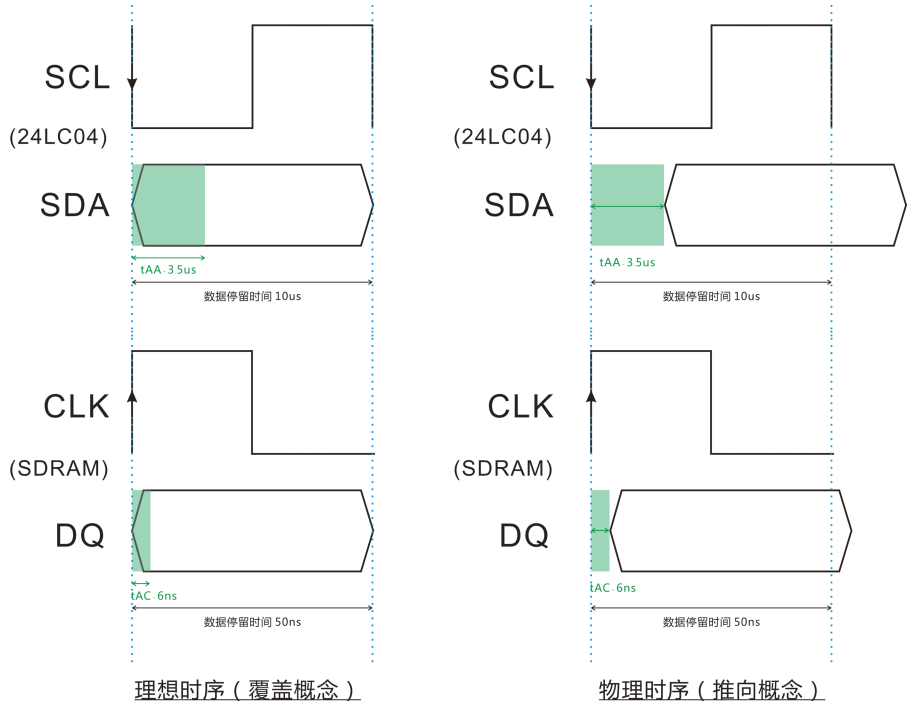


图 3.6q 芯片 24LC04—tAA 和 SDRAM—tAC 之间的关系。

图 3.6q 可说是笔者的大优惠，左图是理想时序的覆盖概念，右图是物理时序的推向概念。上图是芯片 24LC04 数据被读出的时候受到 tAA 时间要求的影响，下图是 SDRAM 数据被读出的时候受到 tAC 时间要求的影响。

读者在这里看到没有，SDRAM 的使用频率假设是 20Mhz，那么数据的停留的时间是 50ns，然而 tAC 的覆盖率不到 20%。所以说，tAC-6ns 的时间要求对于停留时间为 50ns 的数据（时钟频率 20Mhz）来说影响不大。

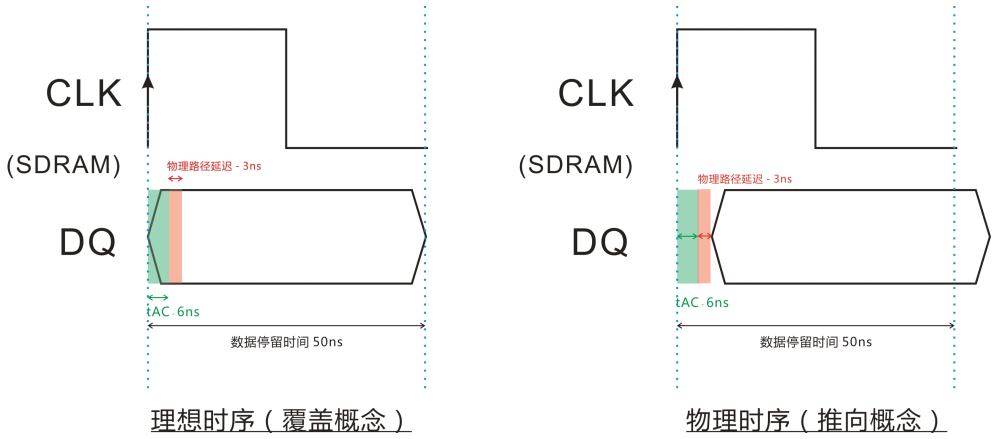


图 3.6r tAC 时间要求再物理路径延迟。

基于图 3.6q 中的下图（SDRAM），如果再加上物理路径延迟的话（假设为 3ns），结果到底会是什么样的呢？答案如图 3.6r 所示，tAC 加上物理路径的延迟，总和覆盖率也是不足数据停留时间的 20% ... 根本就是不疼不痒。我们尝试换一个情况去思考同样的问题，如果 SDRAM 使用的频率是 100Mhz 而不是 20Mhz 的话，结果会是怎么样的？

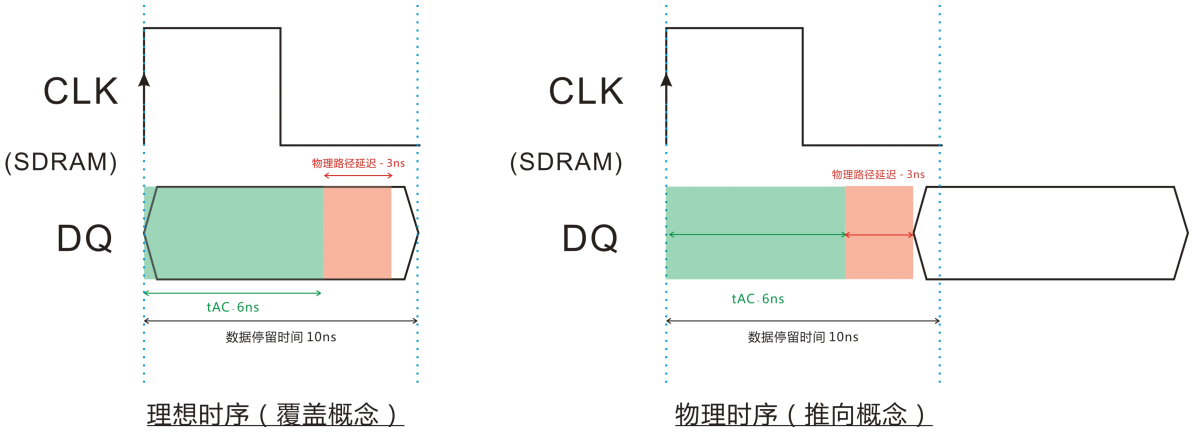


图 3.6s SDRAM 的频率从 20Mhz 变成 100Mhz。

读者看到了吗？tAC 加上物理路径的延迟，总和覆盖率是数据停留时间的 90%。当覆盖率超过 50%的时候就有执行静态时序分析的必要了，更何况是 90%？汗，不知不觉又讲到火星里去了 .... 我们还是回到主题吧。这下读者明白了吗？只要 tAC 的覆盖率还不足数据停留的时间 20%，基本上我们无视它。

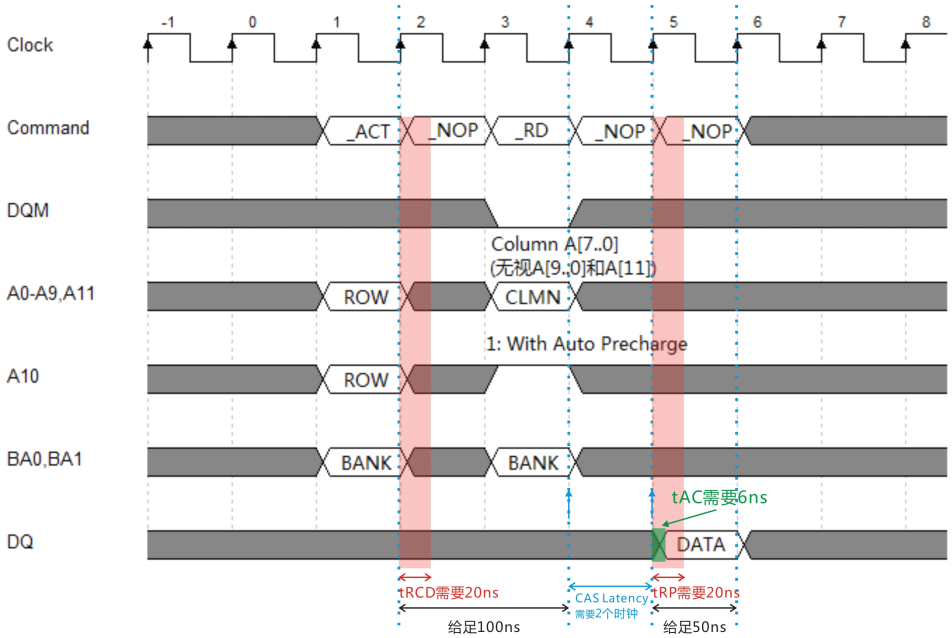


图 3.6p 读一字数据操作的理想时序图。

继续图 3.6p。在 T5 的时候，从 SDRAM 读出的数据经过 tAC 的时间要求以后，读出的一字数据已经有效。然后在 T6 的时候 FPGA 就可以读取数据了。

同样发生在 T5 的时候，SDRAM 的内部开始执行 Auto Precharge 操作，以至于释放相关的资源库。其中 SDRAM 拥有 50ns 的空时间（FPGA 发送的空命令），去完成消耗 tRP - 20ns 的操作。就这样，一次性的一字数据读操作就完成了。

3.7 时间要求与理想时序之间的平衡点

在 3.6 小节里，我们留下这样的内容：

（一）两个模块之间如果处于不同的外部（内部和外部），恰好两个模块使用同样的时钟频率，那么理想时序的“时间点事件”就可以套用在它们的身上。
（二）如果（一）成立的话，那么外部和内部的模块之间就有“物理路径”的存在。
（三）除了“物理路径”存在以外，还有“硬件本身”存在“时间要求”。
（四）“硬件的时间要求”或者“物理路径的延迟”对于 Verilog HDL 语言来说只是概念上的问题而已。Verilog HDL 语言用不着和这些小细节斤斤计较，换之是如何用“理想值”去满足这些小细节。
（五）“理想值”基本上和“速率”有直接的关系，理想值可以是一个速率或者两个速率以上来总和（给足几个时钟的意思）。
（六）理想时序里，时间要求和物理路径的延迟会“覆盖”数据。反之在物理时序里它们会把数据“推向”某个方向。
（七）理想值虽然可以满足第一层或者第二层的时间要求，但是第二层的时间要求往往都是太小了，如果设计不挑剔的话，时间要求的“覆盖率”不超过 50% 的话，还是可以接受的。（安全范围是 1~20%）

（一）到（四）很容易明白而已，至于（五）中所谓用几个速率总和结算理想值，可以看成是用几个没用的时钟，给足操作需要消耗的时间 。（六）和（七）是理想时序里的“覆盖”概念，我们知道 Verilog HDL 语言是理想的工具，假设时钟频率是 20Mhz 的话，那么我们可以使用 20Mhz 去量化任何低于这个频率的速率。

相反的，如果某个时间要求是 1ns 或者 5ns 等，20Mhz 时钟频率无法量化也无法也无法计算，因为 20Mhz 的时钟周期是 50ns。因此“覆盖”概念就成立了，首先 50ns 就是“覆盖率”的理想值，如果这些 1ns 或者 5ns 等时间要求覆盖理想值不足 50%的话，基本上我们可以无视。即使覆盖率是超过了 50%，如果设计不挑剔的话还是可以接受的，否则的话只有执行静态时序分析的份了。

还有其他种类的时间要求，如 60ns 或者 130ns 等。20Mhz 频率的时钟周期是 50ns，单单一个时钟是满足不了 60ns 或者 130ns，那么我们会用上 2 个或者 3 个以上的“空时钟”总和理想值给足这些时间要求。

1ns 和 60ns 同为时间要求，但是 1ns 是第二层的时间要求，60ns 是第一层的时间要求。（第一层时间要求的定义一般是操作所需要的时间，或者是巨大的时间要求；第二层时间要求的定义一般是硬件本身的时间要求，或者是极小的时间要求）

上述的内容就这笔者所谓的“时间要求与理想时序之间的平衡点”。 在笔者的眼中 Verilog HDL 语言是完美而且理想的工具，Verilog 的理想时序只能用 N 个时钟去计算而已。“时间要求”更多时候是指向硬件，或者称为“物理时间要求”，时间要求的种类可以很多：硬件操作所需要的时间，路径的延迟时间，硬件本身的时间参数等。

读者需要明白，这些“物理时间要求”不能用 N 个时钟去计算，唯一黏糊（平衡）这两者之间（理想时序和物理时间要求）的办法，就是“用理想时序去覆盖物理时序”（满足）。具体一点的说法是，用 理想值/时序 去满足物理时序/时间要求。这就好比，1~9 元的东西都用 10 元去购买；11~19 元的东西都用 2 个 10 元去购买。可以被 10 整除的值称为理想值，不收余钱的慷慨行为就称为理想时序。

实验十八（二）SDRAM 基本功能驱动

sdram_init_module.v

图 3.7a SDRAM 初始化模块的图形。

图 3.7a 中所示是 SDRAM 初始化模块。SDRAM 初始化功能拥有独立的模块，原因很简单，就是初始化模块使用次数只有 SDRAM 上电这一次而已。该模块有 Start_Sig 和 Done_Sig，它们用来启动模块和反馈完成；SDRAM_CMD 信号时用来发送命令；SDRAM_BA 是用来发送 Bank，Column 和 row 等地址信号，然而位分配是[13:12] BA1~0, [11:0] A11~A0。

```
module sdram_init_module
(
        input CLK,
        input RSTn,

        input Init_Start_Sig,
        output Init_Done_Sig,

        output [4:0]SDRAM_CMD, // [4]CKE , [3]CSn, [2]RAS, [1]CASn, [0]WEn
        output [13:0]SDRAM_BA // [13:12]BA , [11:0]Addr

);

      /****************************************/

      parameter T200US = 12'd4000;

      /*********************************/
```

```
19.
20.         parameter _INIT = 5'b01111, _NOP = 5'b10111, _ACT = 5'b10011, _RD = 5'b10101, _WR = 5'b10100,
21.                     _BSTP = 5'b10110, _PR = 5'b10010, _AR = 5'b10001, _LMR = 5'b10000;
22.
23.         /*********************************/
24.
25.         reg [3:0]i;
26.         reg [11:0]C1;
27.         reg [4:0]rCMD;
28.         reg [13:0]rBA;
29.         reg isDone;
30.
31.         always @ ( posedge CLK or negedge RSTn )
32.                 if( !RSTn )
33.                         begin
34.                                 i <= 4'd0;
35.                                 C1 <= 12'd0;
36.                                 rCMD <= _NOP;
37.                                 rBA <= 14'h3fff; // rAddr[13:12] = BA, rAddr[11:0] = Addr, all reset by 1
38.                                 isDone <= 1'b0;
39.                         end
40.                 else if( Init_Start_Sig )
41.                         case( i )
42.
43.                                 /*********************************************/
44.
45.                                 0:    // Delay 200us (mininum 100us)
46.                                 if( C1 == T200US -1 ) begin C1 <= 12'd0; i <= i + 1'b1; end
47.                                 else begin C1 <= C1 + 1'b1; end
48.
49.                                 /*******************************************/
50.
51.                                 1: // Send Precharge Command
52.                                 begin rCMD <= _PR; rBA <= 14'h3fff; i <= i + 1'b1; end
53.
54.                                 2: // Send 1 nop clock for tRP 20ns
55.                                 begin rCMD <= _NOP; i <= i + 1'b1; end
56.
57.                                 3: // Send Auto Refresh Command
58.                                 begin rCMD <= _AR; i <= i + 1'b1; end
59.
60.                                 4,5: // Send 2 nop clock for tRFC 63ns
61.                                 begin rCMD <= _NOP; i <= i + 1'b1; end
62.
```

```
63.                     /*********************************************/
64.
65.                     6: // Send Auto Refresh Command
66.                     begin rCMD <= _AR; i <= i + 1'b1; end
67.
68.                     7,8: // Send 2 nop clock for tRFC 63ns
69.                     begin rCMD <= _NOP; i <= i + 1'b1; end
70.
71.                     9: // Send LMR command : Burst Read & Write, 3'b010 mean CAS latecy = 2, Sequential, 8 burst length
72.                     begin rCMD <= _LMR; rBA <={ 4'd0, 1'b0, 2'd0, 3'b010, 1'b0, 3'b011 }; i <= i + 1'b1; end
73.
74.                     10,11: // Send 2 nop clock for tMRD
75.                     begin rCMD <= _NOP; i <= i + 1'b1; end
76.
77.                     /*******************************************/
78.
79.                     12:
80.                     begin isDone <= 1'b1; i <= i + 1'b1; end
81.
82.                     13:
83.                     begin isDone <= 1'b0; i <= 4'd0; end
84.
85.                     /*******************************************/
86.
87.                 endcase
88.
89.         /*********************************/
90.
91.         assign SDRAM_CMD = rCMD;
92.         assign SDRAM_BA = rBA;
93.         assign Init_Done_Sig = isDone;
94.
95.        /**********************************/
96.
97.   endmodule
```

第 16 行是 200us 的常量声明。第 20~21 行是相关命令的常量声明。第 25~29 行是相关寄存器的声明和建立。在一开始的时候（第 34~38 行），所有寄存器都清零，除了 rCMD 赋予_NOP 命令和 rBA 全部赋予 1。

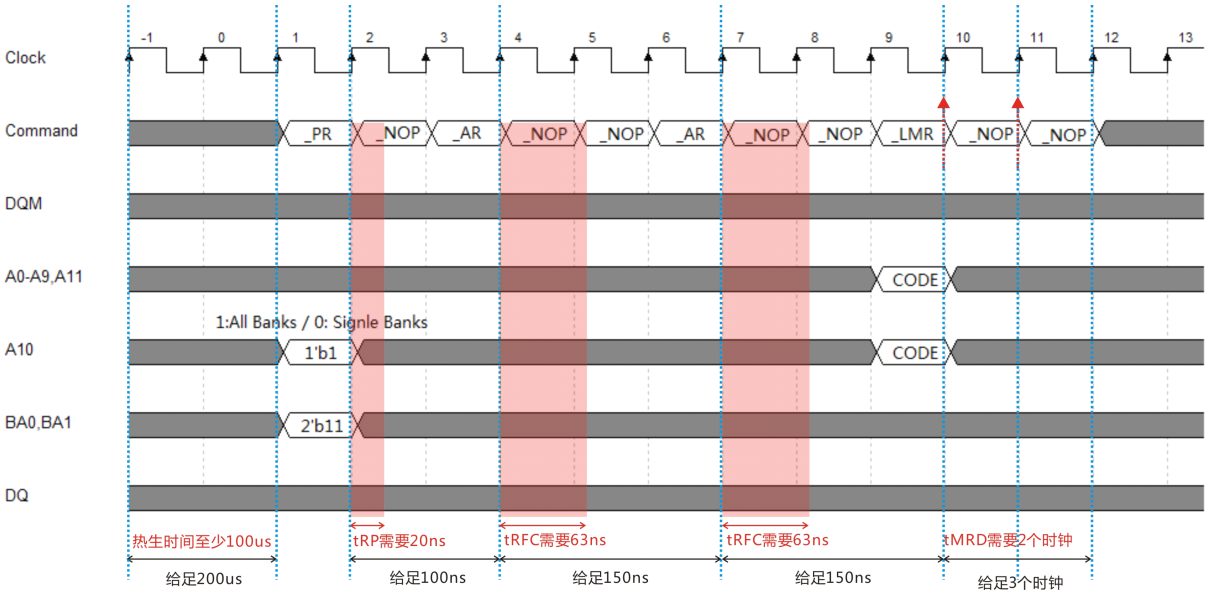


图 3.7b SDRAM 初始化的理想时序图。

如果 Init_Start_Sig 被拉高(第 40 行) ... 首先，步骤 0 会延迟 200us（第 45~47 行）。接下来步骤 1~11 会按图 3.7b 的时序图发生（笔者犯懒了，不清楚的同学自己复习 3.6 小节吧）。最后步骤 12~13（第 79~83 行）会产生完成信号。

iic_func_module.v

图 3.7c SDRAM 功能模块的图形。

图 3.7c 是 SDRAM 的功能模块，其中 Func_Start_Sig 有 3 位位宽，亦即该模块有 3 个基本功能，位分配分别是：[2]自动刷新，[1]读一字操作，[0]写一字操作。其中 Func_Done_Sig 是用来反馈读操作和写操作的完成，换之 AR_Done_Sig 是用来反馈自动刷新的完成。为什么读写操作和自动刷新操作的完成信号要独立开来呢？笔者这样作自然有其他的用意。

BRC_Addr 是库，行和列等组合的地址信号，位分配分别是：[21:20]库-Bank, [19:8]行-Row, [7:0]列-Column，因此 BRC_Addr 也等价与 2^{22} 的寻址深度。WrData 和 RdData 作用如命名般一样。在输出方面 SDRAM_BA 和 SDRAM_BA 的作用和初始化模块一样。至于 SDRAM_LDQM 和 SDRAM_UDQM 信号是 DQM 信号的个别信号。其他的如 SDRAM_DATA 是数据的 IO 信号。

```
1.    module sdram_func_module
2.    (
3.            input CLK,
4.            input RSTn,
5.
6.            input [2:0]Func_Start_Sig,
7.            output Func_Done_Sig,
8.            output AR_Done_Sig,
9.
10.           input [21:0]BRC_Addr,
11.           input [15:0]WrData,
12.           output [15:0]RdData,
13.
14.           output [4:0]SDRAM_CMD, // [4]CKE , [3]CSn, [2]RAS, [1]CASn, [0]WEn
15.           output [13:0]SDRAM_BA, // [13:12]BA , [11:0]Addr
16.           inout [15:0]SDRAM_DATA,
17.
18.           output SDRAM_LDQM,
19.           output SDRAM_UDQM
20.
21.   );
22.
23.         /************************************/
24.
25.         parameter _INIT = 5'b01111, _NOP = 5'b10111, _ACT = 5'b10011, _RD = 5'b10101, _WR = 5'b10100,
26.                     _BSTP = 5'b10110, _PR = 5'b10010, _AR = 5'b10001, _LMR = 5'b10000;
27.
28.         /*********************************/
29.
30.         reg [4:0]i;
31.         reg [9:0]C1;
32.         reg [4:0]rCMD;
33.         reg [13:0]rBA;
34.         reg [1:0]rDQM;
35.         reg [15:0]rData;
36.         reg isOut;
37.         reg isDone;
38.         reg isAR;
39.
40.         always @ ( posedge CLK or negedge RSTn )
41.                 if( !RSTn )
42.                         begin
43.                                 i <= 5'd0;
```

```
44.                     rCMD <= _NOP;
45.                     rBA <= 14'h3fff;
46.                     rDQM <= 2'b11;
47.                     rData <= 16'd0;
48.                     isOut <= 1'b1;
49.                     isDone <= 1'b0;
50.                 end
```

第 25~26 行是相关 SDRAM 命令的常量声明。第 30~38 行是相关寄存器的声明。第 43~39 行是复位动作，其中 rCMD 为 _NOP（第 44 行）；rBA 和 rDQM（第 45~46）均为全部置一；一开始的时候 SDRAM 为输出状态（第 48 行）。

```
51.     else if( Func_Start_Sig[2] )
52.                 case( i )
53.
54.                     /*********************************/
55.
56.                     0: // Send Auto Refresh Command
57.                     begin rCMD <= _AR; i <= i + 1'b1; end
58.
59.                     1,2: // Send 2 nop clock for tRFC - 63ns
60.                     begin rCMD <= _NOP; i <= i + 1'b1; end
61.
62.                     /*********************************/
63.
64.                     3:   // generte done signal
65.                     begin isAR <= 1'b1; i <= i + 1'b1; end
66.
67.                     4:
68.                     begin isAR <= 1'b0; i <= 5'd0; end
69.
70.                     /************************************/
71.
72.                 endcase
```

第 51~72 行是自动刷新功能，Func_Start_Sig[2] 拉高的时候开始执行（第 51 行）。

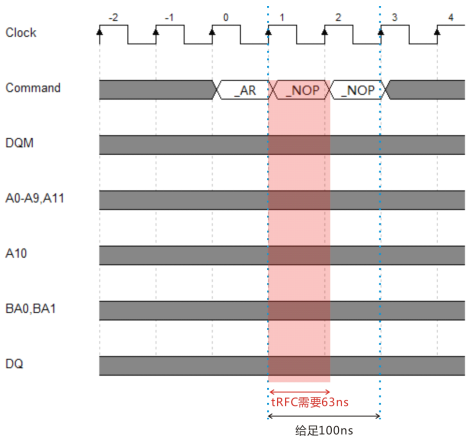


图 3.7d 自动刷新功能的理想时序图。

步骤 0~2（第 56~60 行）会按图 3.7d 的时序图发生，随后步骤 3~4（第 64~68 行）产生完成信号。

```
73.     else if( Func_Start_Sig[1] )
74.                 case( i )
75.
76.                     0: // Set IO to input and clear reg. rData
77.                     begin isOut <= 1'b0; rData <= 16'd0; i <= i + 1'b1; end
78.
79.                     1: // Send Active command , Bank address and Row address
80.                     begin rCMD <= _ACT; rBA <= BRC_Addr[21:8]; i <= i + 1'b1; end
81.
82.                     2: // Send 1 nop clock for tRCD - 20ns
83.                     begin rCMD <= _NOP; i <= i + 1'b1; end
84.
85.                     /****************************************/
86.
87.                 3: // Send Read Command and Row address, pull down DQM 1 clock, 4'b0100 mean A0=1 with auto precharge.
88.                 begin rCMD <= _RD; rBA <= { BRC_Addr[21:20], 4'b0100, BRC_Addr[7:0]}; rDQM <= 2'b00; i <= i + 1'b1; end
89.
90.                     4,5: // Send 1 nop clock for CAS Latency, pull up DQM;
91.                     begin rCMD <= _NOP; rDQM <= 2'b11; i <= i + 1'b1; end
92.
93.                     /*****************************************/
94.
95.                     6: // Read Data
96.                     begin rData <= SDRAM_DATA; i <= i + 1'b1; end
97.
98.                     /*****************************************/
99.
100.                    7: // generate Done signal
```

```
101.                    begin isDone <= 1'b1; i <= i + 1'b1; end
102.
103.                    8:
104.                    begin isDone <= 1'b0; i <= 5'd0; end
105.
106.                    /*****************************************/
107.
108.                endcase
```

第 73~108 行是 SDRAM 的读一字功能，该功能被执行是 Func_Start_Sig[1] 被拉高的时候（第 73 行）。步骤 0（第 76~77 行）是设置 io 口为输入状态，然后清零 rData。

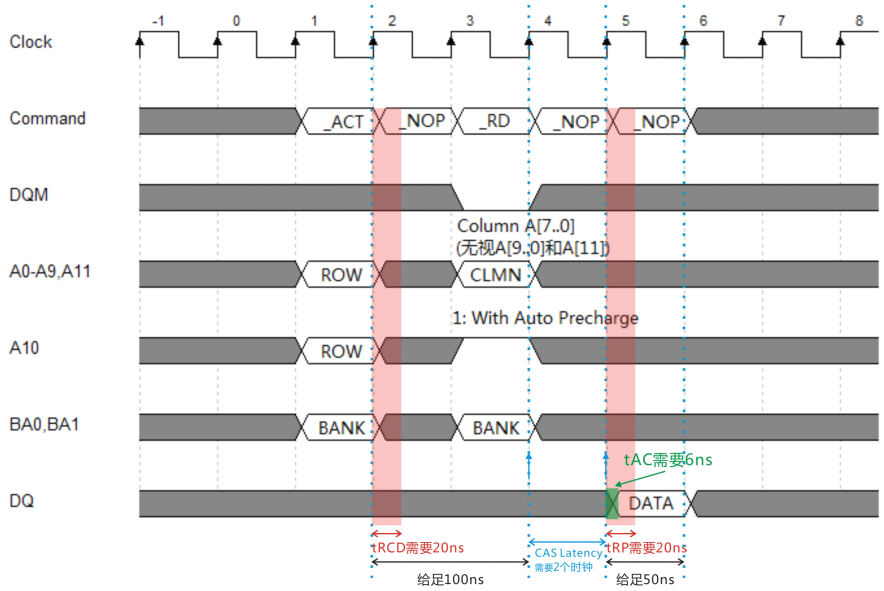


图 3.7e SDRAM 读一字操作的理想时序图。

步骤 1~6（第 79~96 行）会按图 3.7e 发生，随后步骤 7~8（第 100~104 行）会产生完成信号。

```
109.    else if( Func_Start_Sig[0] )
110.                case( i )
111.
112.                    /*************************************/
113.
114.                    0: // Set IO to output
115.                    begin isOut <= 1'b1; i <= i + 1'b1; end
116.
117.                    1: // Send Active Command, bank address and row address
118.                    begin rCMD <= _ACT; rBA <= BRC_Addr[21:8]; i <= i + 1'b1; end
119.
120.                    2: // Send 1 nop clock for tRCD - 20ns
121.                    begin rCMD <= _NOP; i <= i + 1'b1; end
```

```
122.
123.                    /*********************************************/
124.
125.                    3: // negedge FPGA update cmd Write and BRC Addr with auto precharge
126.            begin rCMD <= _WR; rBA <= { BRC_Addr[21:20], 4'b0100, BRC_Addr[7:0] }; rDQM <= 2'b00; i <= i + 1'b1; end
127.
128.                    4,5,6: // Send 3 nop clock for tDPL and tRP
129.                    begin rCMD <= _NOP; rDQM <= 2'b11; i <= i + 1'b1; end
130.
131.                    /*********************************************/
132.
133.                    7: // generate done signal
134.                    begin isDone <= 1'b1; i <= i + 1'b1; end
135.
136.                    8:
137.                    begin isDone <= 1'b0; i <= 5'd0; end
138.
139.                    /*********************************************/
140.
141.            endcase
```

第 109~141 行是 SDRAM 的写一字操作，当 Func_Start_Sig[0] 被拉高的时候开始工作（第 109 行）。步骤 0（第 114~115 行）是设置 io 为输出状态。

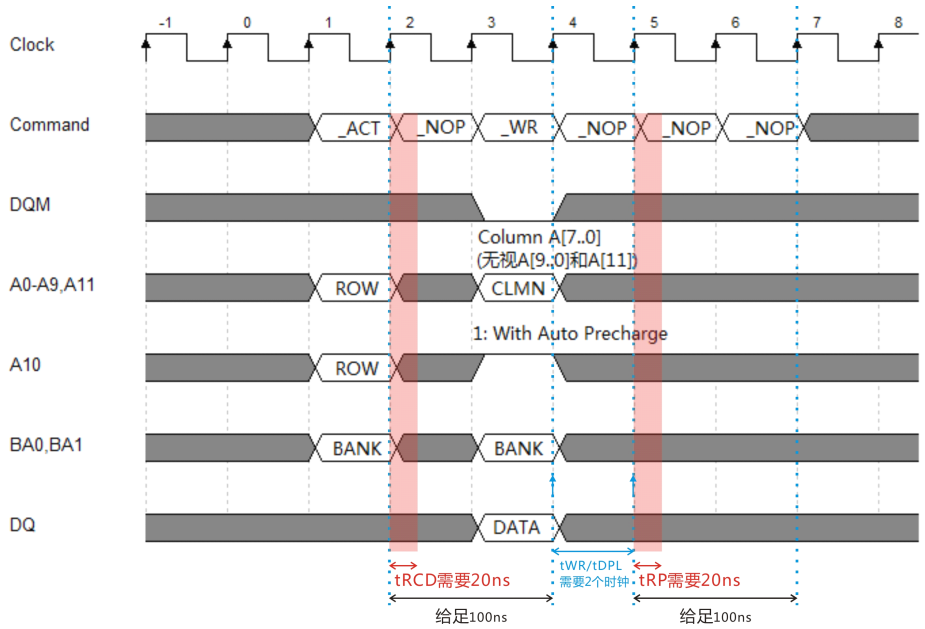


图 3.7f 写一字操作的理想时序图。

接下来的步骤 1~6（第 117~129 行）会按着图 3.7f 发生，随后步骤 7~8 会产生完成信号（第 133~137 行）。

```
142.
143.        /***********************************/
144.
145.          assign SDRAM_CMD = rCMD;
146.          assign SDRAM_BA = rBA;
147.          assign { SDRAM_LDQM, SDRAM_UDQM } = rDQM;
148.          assign SDRAM_DATA = isOut ? WrData : 16'hzzzz;
149.          assign RdData = rData;
150.          assign Func_Done_Sig = isDone;
151.          assign AR_Done_Sig = isAR;
152.
153.        /************************************/
154.
155. endmodule
```

第 142~155 行相关的输出配置，自己看着办吧。

sdram_ctrl_module.v

图 3.7g SDRAM 控制模块的图形。

图 3.7g 是 SDRAm 控制模块的图形，左边的信号 WrEn_Sig，RdEn_Sig，Done_Sig 和 Busy_Sig 等是用于上一层模块调用。其中 WrEn_Sig 和 RdEn_Sig 如命名般，用来使能写操作和读操作；Done_Sig 是用来反馈 WrEn_Sig 和 RdEn_Sig 的操作完成；Busy_Sig 是用来反馈 SDRAM 上电后初始化的进度，如果处于高电平即表示 SDRAM 还在初始化。

右边的信号用来控制 SDRAM 初始化模块和 SDRAM 功能模块，其中 Init_Start_Sig 是用来使能 SDRAM 初始化模块，Init_Done_Sig 是用来反馈初始化的完成。Func_Start_Sig 是用来使能 SDRAM 功能模块的 3 中功能，然而 Func_Done_Sig 是反馈写操作和读操作的完成。换之 Ar_Done_Sig 是用反馈自动刷新的完成。具体的内容，我们还是来看代码吧：

```
1.    module sdram_control_module
2.    (
```

```
3.          input CLK,
4.          input RSTn,
5.
6.          input WrEN_Sig,
7.          input RdEN_Sig,
8.          output Done_Sig,
9.          output Busy_Sig,
10.
11.         input Init_Done_Sig,
12.         input Func_Done_Sig,
13.         input AR_Done_Sig,
14.
15.         output Init_Start_Sig,
16.         output [2:0]Func_Start_Sig
17.
18.   );
19.
20.       /*********************************/
21.
22.         parameter T15US = 9'd100;
23.
24.         /*********************************/
25.
26.         reg [21:0]C1;
27.         reg isRef;
28.
29.         always @ ( posedge CLK or negedge RSTn )
30.                 if( !RSTn )
31.                         begin C1 <= 22'd0; isRef <= 1'b0; end
32.                 else if( isRef && AR_Done_Sig )
33.                         begin C1 <= 22'd0; isRef <= 1'b0; end
34.                 else if( C1 == T15US )
35.                         begin C1 <= 22'd0; isRef <= 1'b1; end
36.                 else if( !isRef )
37.                         C1 <= C1 + 1'b1;
38.
39.         /************************************/
40.
41.         reg [2:0]i;
42.         reg [2:0]isStart; //[2] Auto Refresh , [1] Read Action, [0] Write Action
43.         reg isInit;
44.         reg isBusy;
45.         reg isDone;
46.
```

```
47.         always @ ( posedge CLK or negedge RSTn )
48.             if( !RSTn )
49.                 begin
50.                         i <= 3'd5;              // Initial SDRam at first
51.                         isStart <= 3'b000;
52.                         isInit <= 1'b0;
53.                         isBusy <= 1'b1;
54.                         isDone <= 1'b0;
55.                     end
56.             else
57.                 case( i )
58.
59.                         0: // IDLE state
60.                         if( isRef ) begin isStart <= 3'b100; i <= 3'd1; end
61.                         else if( RdEN_Sig ) begin isStart <= 3'b010; i <= 3'd2; end // Write enable
62.                         else if( WrEN_Sig ) begin isStart <= 3'b001; i <= 3'd2; end // Read enable
63.
64.                         1: // Auto Refresh Done
65.                         if( AR_Done_Sig ) begin isStart <= 3'd0; i <= 3'd0; end
66.
67.                         2: // Func Done
68.                         if( Func_Done_Sig ) begin isStart <= 3'd0; i <= i + 1'b1; end
69.
70.                         /*************************************/
71.
72.                         3: // Generate Done Signal
73.                         begin isDone <= 1'b1; i <= i + 1'b1; end
74.
75.                         4:
76.                         begin isDone <= 1'b0; i <= 3'd0; end
77.
78.                         /*****************************************/
79.
80.                         5: // Initial SDRam
81.                         if( Init_Done_Sig ) begin isBusy <= 1'b0; isInit <= 1'b0; i <= 3'd0; end
82.                         else begin isBusy <= 1'b1; isInit <= 1'b1; end
83.
84.                     endcase
85.
86.         /*************************************/
87.
88.         assign Init_Start_Sig = isInit;
89.         assign Func_Start_Sig = isStart;
90.         assign Done_Sig = isDone;
```

```
91.         assign Busy_Sig = isBusy;
92.
93.         /************************************/
94.
95.   endmodule
```

第 3~16 行是该模块的输入输出声明。第 22 行是 15us 的常量声明。第 26~37 行是 15us 的计数器，每当 C1 计数到 15us（第 34 行），那么 isRef 就会被拉高（第 35 行）。直到收到 Ar_Done_Sig 的反馈信号，否则计数器不重新开工。

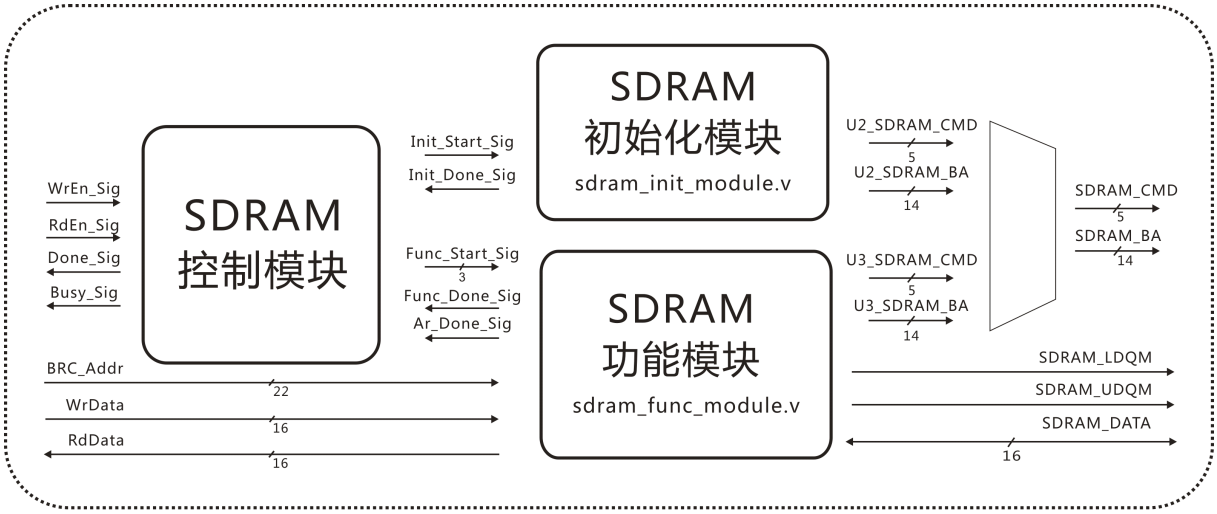
第 41~45 行是相关的寄存器声明，寄存器的命名都很直接 ... 第 47~84 行是控制模块的核心功能。首先所有寄存器都清零（第 50~54 行），除了步骤 i 初值为 5（第 50 行）和 isBusy 初值被置一（第 53 行）。步骤 5 是使能 SDRAM 初始化模块，亦即初始化 SDRAM 操作。isBusy 一开被置一是为了告诉上一层模块 SDRAM 在初始化当中 ...

当步骤 5（第 80~82 行）初始化的动作完成以后，SDRAM 初始化模块会反馈完成信号（第 81 行），与此同时 isBusy 会被拉低，i 会返回步骤 0。步骤 0（第 59~62 行）可以看成是待机的状态：如果 isRef 被拉高（第 60 行）isStart 会赋值与 3'b100，然后步骤 i 会指向步骤 1（第 64~65 行）等待自动刷新的操作完成，最后不反馈往哪成信号然后返回步骤 0（第 65 行）。

如果此时 RdEn_Sig 信号被拉高（第 61 行），isStart 会赋值与 3'b010，然后步骤 i 会指向步骤 2（第 67~68 行）等待读操作的完成（第 68 行），最后会反馈完成信号（第 72~76 行）然后返回步骤 0。如果 WrEn_Sig 信号被拉高（第 62 行）会发生同样的动作，不同的只是 isStart 的赋值是 3'b001 而不是其他（第 62 行）。

第 88~91 行是相关的输出驱动 ... 读者自己看着办吧。

sdram_module.v



sdram_module.v

图 3.7h sdram 模块的建模图。

图 3.7h 是 sdram 模块的建模图，基本上它也没有什么特别，只要读者的联想能力够强的话 ... 不过，笔者姑且还是简单的解释一番。SDRAM 控制模块的 WrEn_Sig，RdEn_Sig, Done_Sig，Busy_Sig 和 SDRAM 功能模块的 BRC_Addr，WrData，RdData 等信号，是该组合模块（sdram_module）的输入。

反之，SDRAM 初始化模块和 SDRAM 功能模块的共有的 SDRAM_CMD 与 SDRAM_BA 信号，联合 SDRAM 功能模块的 SDRAM_LDQM，SDRAM_UDQM，SDRAM_DATA 等信号作为该组合模块的输出。其中输出选择器用来协调 SDRAM_CMD 和 SDRAM_BA 信号的输出。

```
1.    module sdram_module
2.    (
3.            input CLK,
4.            input RSTn,
5.
6.            input WrEN_Sig,
7.            input RdEN_Sig,
8.            output Done_Sig,
9.            output Busy_Sig,
10.
11.           input [21:0]BRC_Addr,
12.           input [15:0]WrData,
13.           output [15:0]RdData,
14.
15.           output [4:0]SDRAM_CMD,
16.           output [13:0]SDRAM_BA,
17.           inout [15:0]SDRAM_DATA,
18.
19.           output SDRAM_LDQM,
20.           output SDRAM_UDQM
21.
22.   );
23.
24.         /**********************************/
25.
26.           wire Init_Start_Sig;
27.           wire [2:0]Func_Start_Sig;
28.           wire [15:0]Func_Write_Data;
29.
30.           wire Init_Done_Sig;
31.           wire Func_Done_Sig;
32.           wire AR_Done_Sig;
```

```
33.
34.         sdram_control_module U1
35.           (
36.                   .CLK( CLK ),
37.                   .RSTn( RSTn ),
38.                   .WrEN_Sig( WrEN_Sig ),                    // input - from top
39.                   .RdEN_Sig( RdEN_Sig ),                     // input - from top
40.                   .Done_Sig( Done_Sig ),                     // output - to top
41.                   .Busy_Sig( Busy_Sig ),                    // output - to top
42.                   .Init_Done_Sig( Init_Done_Sig ),          // input - from U2
43.                   .Func_Done_Sig( Func_Done_Sig ),          // input - from U3
44.                   .AR_Done_Sig( AR_Done_Sig ),              // input - from U3
45.                   .Init_Start_Sig( Init_Start_Sig ),        // output - to U2
46.                   .Func_Start_Sig( Func_Start_Sig )         // output - to U3
47.           );
48.
49.           /**********************************/
50.
51.           wire [4:0]U2_SDRAM_CMD;
52.           wire [13:0]U2_SDRAM_BA;
53.
54.           sdram_init_module U2
55.           (
56.                   .CLK( CLK ),
57.                   .RSTn( RSTn ),
58.                   .Init_Start_Sig( Init_Start_Sig ),                // input - from U1
59.                   .Init_Done_Sig( Init_Done_Sig ),                  // output - to U1
60.                   .SDRAM_CMD( U2_SDRAM_CMD ),                       // output - to selector
61.                   .SDRAM_BA( U2_SDRAM_BA )                          // output - to selector
62.           );
63.
64.           /**********************************/
65.
66.           wire [15:0]Func_Read_Data;
67.           wire [4:0]U3_SDRAM_CMD;
68.           wire [13:0]U3_SDRAM_BA;
69.
70.           sdram_func_module U3
71.           (
72.                   .CLK( CLK ),
73.                   .RSTn( RSTn ),
74.                   .Func_Start_Sig( Func_Start_Sig ),                // input - from U1
75.                   .Func_Done_Sig( Func_Done_Sig ),                  // output - to U1
76.                   .AR_Done_Sig( AR_Done_Sig ),
```

```
                .BRC_Addr( BRC_Addr ),                          // input - from top
                .WrData( WrData ),                              // input - from top
                .RdData( RdData ),                              // output - to top
                .SDRAM_CMD( U3_SDRAM_CMD ),                     // output - to selector
                .SDRAM_BA( U3_SDRAM_BA ),                       // output - to selector
                .SDRAM_DATA( SDRAM_DATA ),                      // output - to top
                .SDRAM_LDQM( SDRAM_LDQM ),
                .SDRAM_UDQM( SDRAM_UDQM )
        );

        /***********************************/

        //Selector of SDRAM_CMD pin and SDRAM_BA pin

        reg [4:0]rCMD;
        reg [13:0]rBA;

        always @ ( * )
              if( Init_Start_Sig ) begin rCMD = U2_SDRAM_CMD; rBA = U2_SDRAM_BA; end
                else if( Func_Start_Sig ) begin rCMD = U3_SDRAM_CMD; rBA = U3_SDRAM_BA; end
                else begin rCMD = 5'bxxxxx; rBA = 14'bxxxxxxxxxxxxxx; end

        assign SDRAM_CMD = rCMD;
        assign SDRAM_BA = rBA;

        /***********************************/

        //assign SQ_Init_Start_Sig = Init_Start_Sig;
        //assign SQ_Init_Done_Sig = Init_Done_Sig;
        //assign SQ_Op_Start_Sig = Op_Start_Sig;
        //assign SQ_Op_Done_Sig = Op_Done_Sig;

        /***********************************/

endmodule
```

基本上，代码的内容和图 3.7h 大同小异而已，读者自己看着吧。

sdram_demo.v

```
module sdram_demo
(
        input CLK,
        input RSTn,
```

```
5.
6.          output SDRAM_CLK,
7.          output [4:0]SDRAM_CMD,
8.          output [13:0]SDRAM_BA,
9.          inout [15:0]SDRAM_DATA,
10.
11.         output SDRAM_UDQM,
12.         output SDRAM_LDQM,
13.
14.         output [3:0]LED
15.   );
16.
17.           /*******************************/
18.
19.           wire Done_Sig;
20.           wire Busy_Sig;
21.           wire [15:0]RdData;
22.
23.           sdram_module U1
24.           (
25.                 .CLK( CLK ),
26.                 .RSTn( RSTn ),
27.                 .WrEN_Sig( isWrite ),
28.                 .RdEN_Sig( isRead ),
29.                 .Done_Sig( Done_Sig ),
30.                 .Busy_Sig( Busy_Sig ),
31.                 .BRC_Addr( rBRC ),
32.                 .WrData( rData ),
33.                 .RdData( RdData ),
34.                 .SDRAM_CMD( SDRAM_CMD ),
35.                 .SDRAM_BA( SDRAM_BA ),
36.                 .SDRAM_DATA( SDRAM_DATA ),
37.                 .SDRAM_UDQM( SDRAM_UDQM ),
38.                 .SDRAM_LDQM( SDRAM_LDQM )
39.           );
40.
41.           /***********************************/
42.
43.           parameter T3S = 26'd60000000, T1S = 26'd20000000;
44.
45.           /***********************************/
46.
47.           reg [6:0]i;
48.           reg [25:0]C1;
```

```
49.         reg [4:0]x;
50.         reg [15:0]rData;
51.         reg [21:0]rBRC;
52.         reg [3:0]rLED;
53.         reg isWrite;
54.         reg isRead;
55.
56.         always @ ( posedge CLK or negedge RSTn )
57.             if( !RSTn )
58.                 begin
59.                     i <= 7'd0;
60.                     C1 <= 26'd0;
61.                     x <= 5'd1;
62.                     rData <= 16'd0;
63.                     rBRC <= 22'd0;
64.                     rLED <= 4'd0;
65.                     isWrite <= 1'b0;
66.                     isRead <= 1'b0;
67.                 end
68.             else
69.                 case( i )
70.
71.                     0: // waiting SDRAM initial
72.                     if( !Busy_Sig ) i <= i + 1'b1;
73.
74.                     1:
75.                     if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
76.                     else begin isWrite <= 1'b1; rData <= 16'h8421; rBRC <= 22'd0; end
77.
78.                     2:
79.                     if( Done_Sig ) begin isRead <= 1'b0; i <= i + 1'b1; end
80.                     else begin isRead <= 1'b1; rBRC <= 22'd0; end
81.
82.                     3:
83.                     begin rLED <= RdData[3:0]; i <= i + 1'b1; end
84.
85.                     4:
86.                     if( C1 == T1S -1) begin C1 <= 26'd0; i <= i + 1'b1; end
87.                     else C1 <= C1 + 1'b1;
88.
89.                     5:
90.                     if( Done_Sig ) begin isRead <= 1'b0; i <= i + 1'b1; end
91.                     else begin isRead <= 1'b1; rBRC <= 22'd0; end
92.
```

```
93.                     6:
94.                     begin rLED <= RdData[7:4]; i <= i + 1'b1; end
95.
96.                     7:
97.                     if( C1 == T1S -1) begin C1 <= 26'd0; i <= i + 1'b1; end
98.                     else C1 <= C1 + 1'b1;
99.
100.                    8:
101.                    if( Done_Sig ) begin isRead <= 1'b0; i <= i + 1'b1; end
102.                    else begin isRead <= 1'b1; rBRC <= 22'd0; end
103.
104.                    9:
105.                    begin rLED <= RdData[11:8]; i <= i + 1'b1; end
106.
107.                    10:
108.                    if( C1 == T1S -1) begin C1 <= 26'd0; i <= i + 1'b1; end
109.                    else C1 <= C1 + 1'b1;
110.
111.                    11:
112.                    if( Done_Sig ) begin isRead <= 1'b0; i <= i + 1'b1; end
113.                    else begin isRead <= 1'b1; rBRC <= 22'd0; end
114.
115.                    12:
116.                    begin rLED <= RdData[15:12]; i <= i + 1'b1;end
117.
118.                  /****************************************/
119.
120.                    13:
121.                    if( C1 == T1S -1) begin C1 <= 26'd0; i <= i + 1'b1; end
122.                    else C1 <= C1 + 1'b1;
123.
124.                    14:
125.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
126.                    else begin isWrite <= 1'b1; rData <= 16'd1; rBRC <= 22'd1; end
127.
128.                    15:
129.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
130.                    else begin isWrite <= 1'b1; rData <= 16'd2; rBRC <= 22'd2; end
131.
132.                    16:
133.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
134.                    else begin isWrite <= 1'b1; rData <= 16'd3; rBRC <= 22'd3; end
135.
136.                    17:
```

```
137.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
138.                    else begin isWrite <= 1'b1; rData <= 16'd4; rBRC <= 22'd4; end
139.
140.                    18:
141.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
142.                    else begin isWrite <= 1'b1; rData <= 16'd5; rBRC <= 22'd5; end
143.
144.                    19:
145.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
146.                    else begin isWrite <= 1'b1; rData <= 16'd6; rBRC <= 22'd6; end
147.
148.                    20:
149.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
150.                    else begin isWrite <= 1'b1; rData <= 16'd7; rBRC <= 22'd7; end
151.
152.                    21:
153.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
154.                    else begin isWrite <= 1'b1; rData <= 16'd8; rBRC <= 22'd8; end
155.
156.                    22:
157.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
158.                    else begin isWrite <= 1'b1; rData <= 16'd9; rBRC <= 22'd9; end
159.
160.                    23:
161.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
162.                    else begin isWrite <= 1'b1; rData <= 16'd10; rBRC <= 22'd10; end
163.
164.                    24:
165.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
166.                    else begin isWrite <= 1'b1; rData <= 16'd11; rBRC <= 22'd11; end
167.
168.                    25:
169.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
170.                    else begin isWrite <= 1'b1; rData <= 16'd12; rBRC <= 22'd12; end
171.
172.                    26:
173.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
174.                    else begin isWrite <= 1'b1; rData <= 16'd13; rBRC <= 22'd13; end
175.
176.                    27:
177.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
178.                    else begin isWrite <= 1'b1; rData <= 16'd14; rBRC <= 22'd14; end
179.
180.                    28:
```

```
181.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
182.                    else begin isWrite <= 1'b1; rData <= 16'd15; rBRC <= 22'd15; end
183.
184.                    /************************************/
185.
186.                    29:
187.                    if( C1 == T1S -1) begin C1 <= 26'd0; i <= i + 1'b1; end
188.                    else C1 <= C1 + 1'b1;
189.
190.                    30:
191.                    if( x == 16 ) begin x <= 5'd0; i <= i + 7'd3; end
192.                    else if( Done_Sig ) begin isRead <= 1'b0; i <= i + 1'b1; end
193.                    else begin isRead <= 1'b1; rBRC <= {17'd0, x}; end
194.
195.                    31:
196.                    begin rLED <= RdData[3:0]; x <= x + 1'b1; i <= i + 1'b1; end
197.
198.                    32:
199.                    if( C1 == T1S -1) begin C1 <= 26'd0; i <= i - 7'd2;; end
200.                    else C1 <= C1 + 1'b1;
201.
202.                    33:
203.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
204.                    else begin isWrite <= 1'b1; rData <= 16'd0; rBRC <= 22'd0; end
205.
206.                    34:
207.                    if( Done_Sig ) begin isRead <= 1'b0; i <= i + 1'b1; end
208.                    else begin isRead <= 1'b1; rBRC <= 22'd0; end
209.
210.                    35:
211.                    begin rLED <= RdData[3:0]; end
212.
213.                endcase
214.
215.        /***************************/
216.
217.        assign SDRAM_CLK = CLK;
218.        assign LED = rLED;
219.
220.        /***************************/
221.
222. endmodule
```

上面的代码是用来测试 SDRAM 模块用的，不要看那么长就退缩，说实在它只是充气娃

娃而已。第 2~14 行是输入输出的声明，其中 SDRAM_CLK 最为醒目（第 6 行），它直接由 CLK 驱动（第 217 行）。第 19~39 行是相关连线声明和 SDRAM 模块的实例化，WrEn_Sig 和 RdEn_Sig 分别由 isWrite 和 isRead 寄存器来驱动（第 27~28 行）；BRC_Addr 和 WrData 分别由 rBRC 和 rData 寄存器来驱动（第 31~32 行）。

第 43 行是 3 秒和 1 秒的常量声明。第 47~54 行是相关的寄存器声明，其中 x 作为循环操作使用，rLED 用来驱动四位的 LED 输出。一开始的时候所有的寄存器的清零（第 59~66 行），除了 x 赋值为 1，用于其他目的（第 61 行）。步骤 0 是等待 SDRAM 的初始化结束，步骤 1 是将数据 16'h8421 写入地址 22'd0。

步骤 2~12 是重复读出地址 22'd0 的数据，然后每秒从低四位到高四位将数据输出致 LED。步骤 13 用于延迟 1 秒，没有什么其他用义。步骤 14~28 是将数据 16'd1~15 分别写入地址 22'd1~22'd15。步骤 29 用来延迟 1 秒，没有什么特殊的意义。步骤 30~32 是用循环的方式，将地址 22'd1~15 的数据按次序读出，然后输出致 LED（x 的初值为 1 的用意就是在这里）。步骤 33 是将数据 16'd0 写入地址 22'd0，然后读出，最后将低四位输出致 LED。

步骤 1~12 是用来确定一字数据写入和读出的正确性，步骤 14~32 是用来确定数据连续写入和读出的稳定和正确性。到目前为止，SDRAM 的基本功能的驱动程序，基本上我们已经编辑完成了 ... 读者可以将程序编译然后下载到板子里看看效果。

在此笔者需要强调的是，笔者不作仿真了，如果读者可以看懂 3.6~3.7 小节的内容 .... 笔者可以肯定，波形图已经出现在读者自己的脑海里了。这不是笔者犯懒的借口，在前面笔者说过了，采用主动设计是为了减少对仿真的依赖。因为模块经过主动设计以后，内容都非常直接和清晰，宛如波形图活生生的在脑海里运动着 .... 从第三章的一开始，笔者就暗地里用“主动设计”的概念去完成模块的建立，在不知不觉中读者已经慢慢的吸收和习惯。

3.8 高速率的风险

笔者再简单的复习一下有关的速率的概念和定义。速率是指数据传输的速度，更具体一点是指一位数据所停留的时间。然而数据可以由某个时钟信号来决定，或者其他的。如果时钟信号越高，速率自然而然会越高，与此同时数据所停留的时间就会越低。当两个模块在传输数据的时候，我们当然希望速率越高越好，如果两个模块同样在 FPGA 里，那么问题就没有大不了。

反之，如果两个模块是存在与不同的世界（外部和内部），过高的速率往往会使得一位数据的周期（停留时间）变窄，打个比方：假设我们用 FPGA 去驱动 SDRAM，然而 FPGA 和 SDRAM 使用同样的时钟频率，亦即 100Mhz。100Mhz 的时钟周期是 10ns，FPGA 和 SDRAM 之间的传输速率（一位数据的停留时间）随之也是 10ns。以目前的科技，总和 FPGA 硬件的时间要求，物理路径的延迟，SDRAM 硬件的（第二层）时间要求 ... 超过 6ns 是非常简单的事情。

对于停留 10ns 的数据来说，如果被总时间要求覆盖超过 50%的话，那么我们就可以断定存在着隐藏风险。这话要怎么说呢？总和时间要求，基本上可以由 3 个因素组成，亦即驱动硬件的时间要求，物理路径，和被驱动硬件的时间要求。不要忘了，这 3 个因素会随着环境的变化而有所改变。随便一个夏天伴随毒辣的太阳（温度），或者随便一个雨季来个五月雨（湿度），上面的 3 个因素都会“起伏跳舞”。

当然，这些都是 Verilog HDL 语言不能涉及的地方。既然不关 Verilog HDL 语言的事情，为什么我们还要谈论它呢？呵呵 ... 事实上，这个小节我们依然要谈论有关“概念”的问题。笔者一直不停的唠叨这，完美和理想的工具，无论从什么样的角度去看它，它还是完美和理想。一个理想而且完美的工具，绝对不可能从上面看下来完美，然而从下看看上去却不理想。

同样的道理，无论物理路径有多么差劲，无论硬件的时间要求如何过分，我们不可否定 Verilog HDL 语言是完美而且理想的工具——这一事实。继续引用上面的例子：

在 SDRAM 的驱动过程，当时钟频率从 20Mhz 变成 100Mhz 以后，速率改变，时间要求的覆盖率也会随之增大。网络上有一个常用的解决方法，就是 SDRAM 的时钟信号，比起 FPGA 的时钟信号，提前 75° 的相位。以上这句话是什么意思呢？

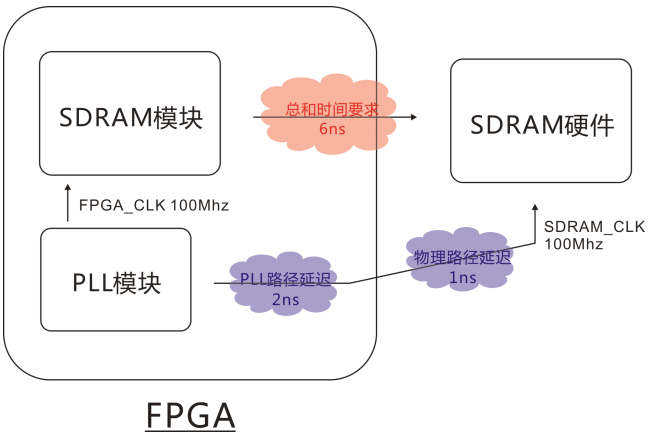


图 3.8a 总和 SDRAM 模块，PLL 模块和 SDRAM 硬件的简易图形。

黑金开发板所使用的时钟源是 20Mhz，如果要倍频成为 100Mhz，当然少不了 PLL 模块。当时钟源倍频过后，然后经过相关的配置，结果如图 3.8a 所示。图中红色云形表示 3 个因数的总和时间要求；换之，蓝色云形却是 FPGA 内部的 PLL 路径延迟，和 FPGA 外部的 SDRAM_CLK 的物理路径延迟。

在这里，有些同学可能觉得很困惑，SDRAM_CLK 的物理路径延迟，无论 SDRAM_CLK 使用的频率是 20Mhz 还是 100Mhz，延迟早就存在。为什么笔者不在 3.6 和 3.7 小节声明，反而在这小节提起呢？原因很简单，SDRAM_CLK 物理路径的延迟因素，如果 PLL 模块没有出现的话，可以纳为红色云形的一部分。或者说 SDRAM_CLK 在 20Mhz 的情况下，SDRAM_CLK 物理路径的延迟所给予的影响根本就是不疼不痒。

目前为止，无论是 FPGA_CLK 还是 SDRAM_CLK 时钟频率均为 100Mhz，随之数据所停留的时间也从原本的 50ns 变成 10ns。由于一位数据所停留的时间变窄了，所以我们不得不重新看待，第二层时间要求和延迟因数给数据所带来的影响（覆盖率）。
（注，接下来的内容，时间要求都是指第二层时间要求，亦即是指驱动和被驱动设备硬件的本身，还有物理路径等因素）

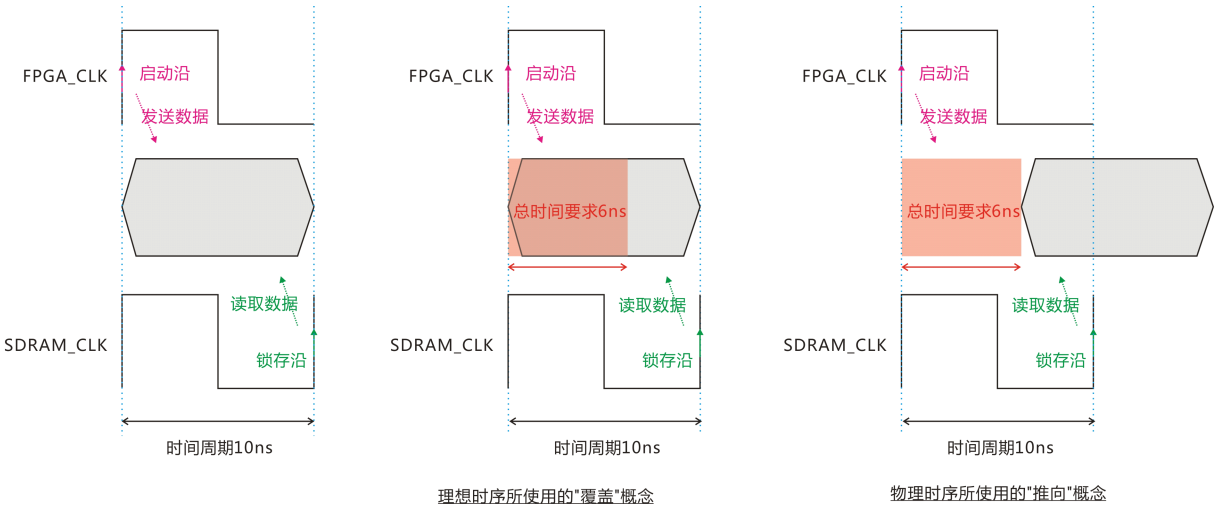


图 3.8b 理想时序和物理时序所使用的不同概念。

在还没有进入问题之前，我们先简单的复习一下，理想时序和物理时序之间在满足时间要求的时候所使用的概念之分。图 3.8b 的最左边是没有任何时间要求的理想时序；中间是涉及时间要求的理想时序，其中所使用的概念是时间要求覆盖数据，任何信号都没有被移动；图 3.8b 最右边是涉及时间要求的物理时序，它所使用的概念是时间要求把数据推向一个方向，结果数据向右移动诺干个时间。

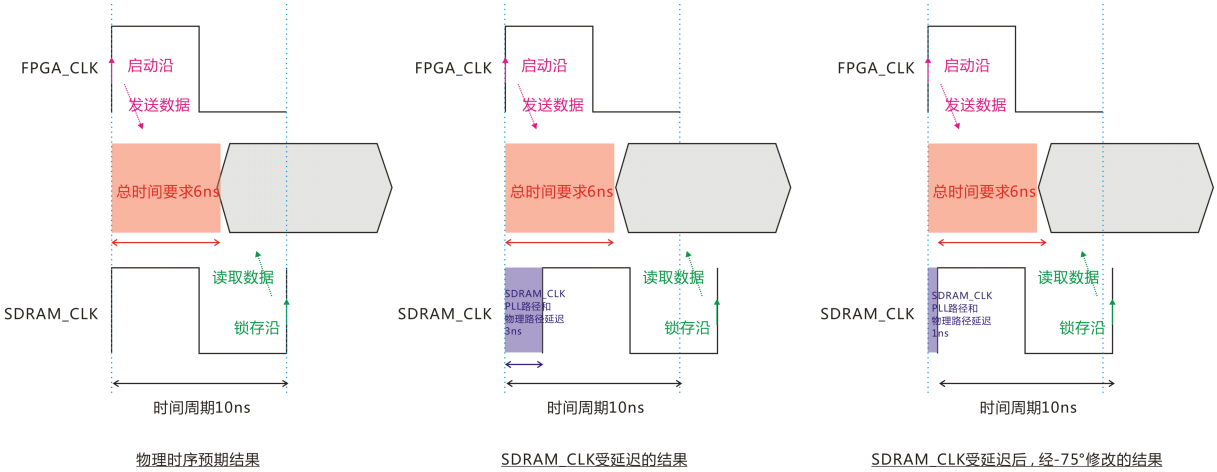


图 3.8c 物理时序，SDRAM_CLK 受 PLL 路径和物理路径的影响，然后经修改-75° 相位的结果。

图 3.8c 是物理时序加入 SDRAM_CLK 受 PLL 路劲和物理路径的影响。图中最左边是预期的结果，亦即没有考虑 SDRAM_CLK 的任何延迟因数；图 3.8c 的中间是 SDRAM_CLK 受 PLL 路径和物理路径影响的结果，结果 SDRAM_CLK 向右边推向 3ns，然而这个 3ns 是 PLL 路劲和物理路径的总和的时间。图 3.8c 的最右边是 SDRAM_CLK 经-75° 相位的修改的结果，亦即将 SDRAM_CLK 向左拉向大约 2ns。计算过程如下：

-75° /360° × 10ns = -2.08333333ns

大家有没有看到，图中尤其是最右边的物理时序图。当 SDRAM 经过 -75° 的修改以后，3 个信号（FPGA_CLK，数据和 SDRAM_CLK）都失去的对齐性。当中 SDRAM 经-75° 的相位修改，在网络中称为黄金 -75 度，实际上就是把 SDRAM_CLK 向左边拉 2ns，目的就是为了恢复预期的结果（图 3.8c 的左图）和合格静态时序的分析。

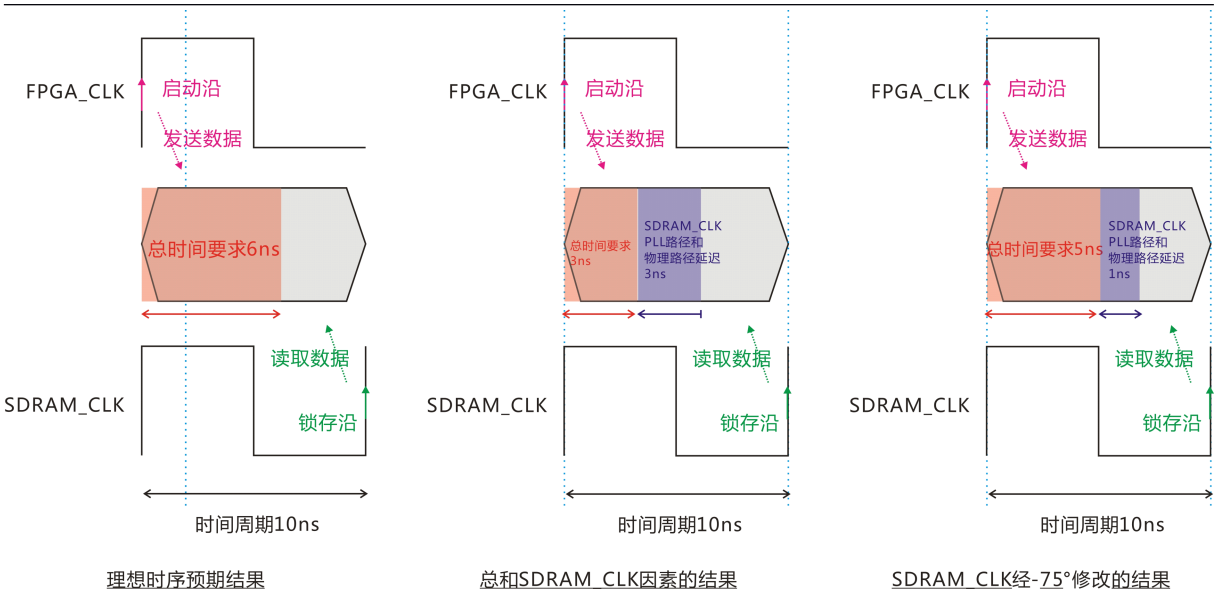


图 3.8d 理想时序，SDRAM_CLK 受 PLL 路径和物理路径的影响，然后经修改-75° 相位的结果。

图 3.8d 中所示是理想时序，SDRAM_CLK 受 PLL 路径和物理路径的延迟结果。首先左图是理想时序的预期结果，亦即不涉及 SDRAM_CLK 因数的预期结果。图 3.8d 的中间则是，数据受 SDRAM_CLK 的 PLL 路径和物理路径的覆盖结果，覆盖率是总和时间要求 ＋PLL 路径和物理路径的延迟，新总和时间要求也从原本的 6ns 变成 3ns，覆盖率也从原本的 60%变成 30%。

图 3.8d 的右图是 SDRAM_CLK 受-75° 修改以后的结果，亦即新总和时间从原来的 3ns 变成 5ns，覆盖率也从原本的 30% 变成 50%。大家有没有看到图 3.8c 和图 3.8d 之间的区别吗？无论图 3.8d 怎么修改和被动，3 个信号（FPGA_CLK，数据和 SDRAM_CLK）依然没有失去对齐性。

==========

在这里，笔者的重点不是要讨论多少覆盖率，多少延迟时间等这些琐碎的东西，笔者这样做是要给读者一个感知的认识，和告诉读者-75° 到底有什么作用而已。换之，笔者是要告诉读者重点就是物理时序和理想时序之间的区别。

如图 3.8c 和图 3.8d 所示，图 3.8c 的物理时序经过各种时间要求的影响以后，信号的对齐性已经被破坏。相反的，图 3.8d 的理想时序，无论各种时间要求如此苛刻的挑衅和影响，信号的对齐性依然无损。

上诉的内容也告诉我们这样一个信息， Verilog HDL 语言绝对实现不了图 3.8c，然而 Verilog HDL 语言却可以实现图 3.8d。因为图 3.8c 不是 Verilog HDL 语言这个理想工具要实现的破烂时序，反之图 3.8d 才是 Verilog HDL 语言这个理想工具愿意实现的时序。

如果读者打从一开始以图 3.8c 作为使用 Verilog HDL 语言的基本概念，估计读者很快就会碰见学习瓶颈，不仅而已还有许多学习的后遗症在等着读者去“享受”。图 3.8d 是物理时序和理想时序之间的“平衡点”，也是“平衡”的概念，既不失 Verilog HDL 语言的

理想本质，又可以涉及和满足物理的时间要求，岂不是一举两得，活得乐悠悠呢？

此外，还有一个重要的信息的是：驱动 SDRAM 无论是使用 20Mhz 还是 100Mhz 的频率也好，Verilog 理想时序的编写和理解方法始终如一。用简单的话来说：不同时钟频率有如不同的背景而已，Verilog HDL 语言要干的工作就是如何实现 SDRAM 和 FPGA 之间的“沟通模式”。无论背景如何改变，SDRAM 和 FPGA 之间的“沟通模式”还是一样。

理想的沟通模式依然是理想的沟通模式，任何在沟通上出现的时间要求都不是 Verilog HDL 语言可以解决的了的工作。Verilog HDL 语言能做的就是时时刻刻提供“理想”的沟通而已。那些在沟通上出现的时间要求，自然而然有“其他工具”去负责。话说白就是：Verilog HDL 语言只负责驱动，时间要求它虽然可以涉及和满足，但是它不能负责和解决。反之这些活儿（负责解决时间要求）都是静态时序分析工具的工作。

啊 .... 说了那么一大堆的话，笔者的手指也累坏了。

实验十九 ：SDRAM—100Mhz 基本驱动

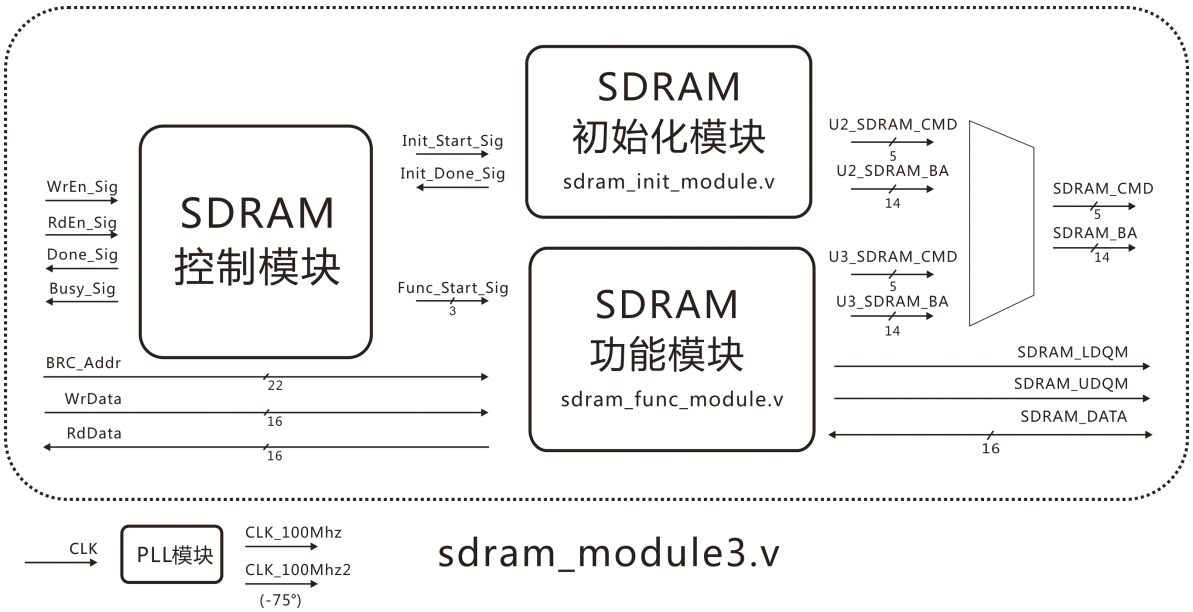


图 3.8e 100Mhz 驱动的 SDRAM 模块的建模图形。

图 3.8e 是基于实验十八修改而来的，图中的 SDRAM 模块在外部加了一个 PLL 模块。PLL 模块的功能主要是把 20Mhz 的 CLK 信号，倍频致两个 100Mhz 的信号，其中 CLK_100Mhz 是给内部用的时钟信号，反之经过修改-75°相位的 CLK_100Mhz2 是用来驱动外部的 SDRAM 硬件。

很多同学都不明白，为什么驱动 SDRAM 硬件的时钟信号—CLK_100Mhz2 需要经过-75°的修改呢？原因有如笔者在前面所讲的那样，-75°的修改时为了补偿 PLL 模块的内部路径和外部物理路径的延迟。此外，该模块还做了一些修改，就是更换了 sdram 功能

模块的沟通方式，从原本的问答式沟通变成使能式沟通。

sdram_initial_module.v

该文件基本上修改的地方不多，由于现在驱动 SDRAM 的频率是 100Mhz，所以 sdram 初始化模块也做了想对应的修改。

```
1.    module sdram_init_module
2.    (
3.            input CLK,
4.            input RSTn,
5.
6.            input Init_Start_Sig,
7.            output Init_Done_Sig,
8.
9.            output [4:0]SDRAM_CMD, // [4]CKE , [3]CSn, [2]RAS, [1]CASn, [0]WEn
10.           output [13:0]SDRAM_BA // [13:12]BA , [11:0]Addr
11.
12.   );
13.
14.         /****************************************/
15.
16.         parameter T200US = 15'd20000;
17.
18.         /*********************************/
19.
20.         parameter _INIT = 5'b01111, _NOP = 5'b10111, _ACT = 5'b10011, _RD = 5'b10101, _WR = 5'b10100,
21.                     _BSTP = 5'b10110, _PR = 5'b10010, _AR = 5'b10001, _LMR = 5'b10000;
22.
23.         /*********************************/
24.
```

第 1~24 行，除了相关的输入和输出的声明以外，还有 200us 的常量声明，其外就是一些命令的常量声明。

```
25.         reg [4:0]i;
26.         reg [14:0]C1;
27.         reg [4:0]rCMD;
28.         reg [13:0]rBA;
29.         reg isDone;
30.
31.         always @ ( posedge CLK or negedge RSTn )
32.               if( !RSTn )
```

```
33.                 begin
34.                     i <= 5'd0;
35.                     C1 <= 15'd0;
36.                     rCMD <= _NOP;
37.                     rBA <= 14'h3fff; // rAddr[13:12] = BA, rAddr[11:0] = Addr, all reset by 1
38.                     isDone <= 1'b0;
39.                 end
```

第 25~39 行是相关的寄存器声明，还有一些复位的初始化操作。

```
40.   else if( Init_Start_Sig )
41.                 case( i )
42.
43.                     /*********************************************/
44.
45.                     0:    // delay 100~200us
46.                     if( C1 == T200US -1 ) begin C1 <= 15'd0; i <= i + 1'b1; end
47.                     else begin C1 <= C1 + 1'b1; end
48.
49.                     /*********************************************/
50.
51.                     1: // Send Precharge Command
52.                     begin rCMD <= _PR; rBA <= 14'h3fff; i <= i + 1'b1; end
53.
54.                     2: // Send 1 nop clock for tRP-20ns
55.                     begin rCMD <= _NOP; i <= i + 1'b1; end
56.
57.                     3: // Send Auto Refresh Command
58.                     begin rCMD <= _AR; i <= i + 1'b1; end
59.
60.                     4,5,6,7,8,9: // Send 6 nop clock for tRFC - 63ns
61.                     begin rCMD <= _NOP; i <= i + 1'b1; end
62.
63.                     /*********************************************/
64.
65.                     10: // Send Auto Refresh Command
66.                     begin rCMD <= _AR; i <= i + 1'b1; end
67.
68.                     11,12,13,14,15,16: // Send 6 nop clock for tRFC - 63ns
69.                     begin rCMD <= _NOP; i <= i + 1'b1; end
70.
71.                 17: // Send LMR Cmd. Burst Read & Write,    3'b010 mean CAS latecy = 2, Sequential, 8 burst length
72.                     begin rCMD <= _LMR; rBA <={ 4'd0, 1'b0, 2'd0, 3'b010, 1'b0, 3'b011 }; i <= i + 1'b1; end
73.
```

```
74.                     18,19: // Send 2 nop CLK for tMRD
75.                     begin rCMD <= _NOP; i <= i + 1'b1; end
76.
77.                     /*******************************************/
78.
79.                     20: // Generate done signal
80.                     begin isDone <= 1'b1; i <= i + 1'b1; end
81.
82.                     21:
83.                     begin isDone <= 1'b0; i <= 5'd0; end
84.
85.                     /*******************************************/
86.
87.                 endcase
```

第 40~87 行是初始化的相关操作，步骤 0~19 完全按图 3.8f 中的时序发生，步骤 20~21 是产生完成信号。修改的地方不多，只是按 100Mhz 的时钟周期，重新分配给足时间而已。

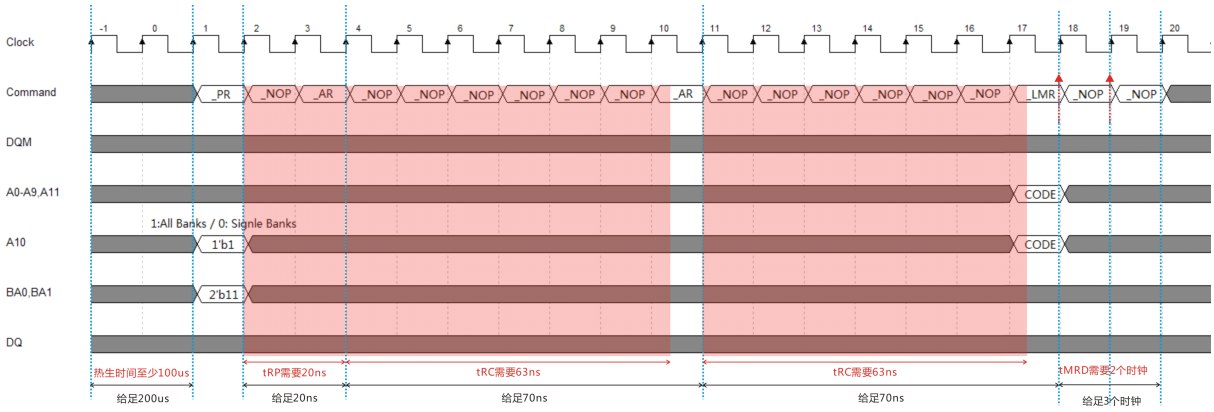


图 3.8f 100Mhz 驱动 SDRAM，初始化操作的理想时序图。

```
88.
89.         /*********************************/
90.
91.         assign SDRAM_CMD = rCMD;
92.         assign SDRAM_BA = rBA;
93.         assign Init_Done_Sig = isDone;
94.
95.        /**********************************/
96.
97.   endmodule
```

第 88~97 行是相关的输出驱动配置，很简单而已，自己看着办吧。

sdram_func_module.v

同样 sdram 功能模块修改的地方也不多，除了按 100Mhz 的时钟周期重新分配给足时间以外，还改变了 sdram 功能模块的控制方法，亦即除去 Start_Sig 和 Done_Sig 等问答的沟通方式，取而代之是用使能式的方法（需要几个时钟工作，就给几个时钟）。

```
module sdram_func_module
(
        input CLK,
        input RSTn,

        input [2:0]Func_Start_Sig,

        input [21:0]BRC_Addr,
        input [15:0]WrData,
        output [15:0]RdData,

        output [4:0]SDRAM_CMD, // [4]CKE , [3]CSn, [2]RAS, [1]CASn, [0]WEn
        output [13:0]SDRAM_BA, // [13:12]BA , [11:0]Addr
        inout [15:0]SDRAM_DATA,

        output SDRAM_LDQM,
        output SDRAM_UDQM

);

      /************************************/

      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;

      /*********************************/

      reg [3:0]i;
      reg [9:0]C1;
      reg [4:0]rCMD;
      reg [13:0]rBA;
      reg [1:0]rDQM;
      reg [15:0]rData;
      reg isOut;

      always @ ( posedge CLK or negedge RSTn )
            if( !RSTn )
```

```
38.                 begin
39.                     i <= 4'd0;
40.                     rCMD <= _NOP;
41.                     rBA <= 14'h3fff;
42.                     rDQM <= 2'b11;
43.                     rData <= 16'd0;
44.                     isOut <= 1'b1;
45.                 end
```

第 1~45 行是输入输出的声明；命令的常量声明；相关寄存器的声明；最后就是相关的复位操作，其中 rCMD 的初值为 _NOP 命令，rDQM 拉高，还有 isOut 设置为 1。

```
46.         else if( Func_Start_Sig[2] )
47.                 case( i )
48.
49.                     /*********************************/
50.
51.                     0: // Send Auto Refresh Command
52.                     begin rCMD <= _AR; i <= i + 1'b1; end
53.
54.                     1,2,3,4,5,6,7: // Send 6 nop Clk for tRFC-63
55.                     begin rCMD <= _NOP; i <= i + 1'b1; end
56.
57.                     /*********************************/
58.
59.                     8:    // return i to 0
60.                     begin i <= 4'd0; end
61.
62.                     /*************************************/
63.
64.                 endcase
```

第 46~64 行是自动刷新的功能。这里有 9 个步骤，其中步骤 0~7 是自动刷新的操作，步骤 8 是用来返回操作，结果需要 9 个时钟。因此在拉高 Func_Start_Sig[2] 的时候，确保 Func_Start_Sig[2]的驱动源给足 9 个时钟的高脉冲。具体的操作自己看图 3.8g 吧。

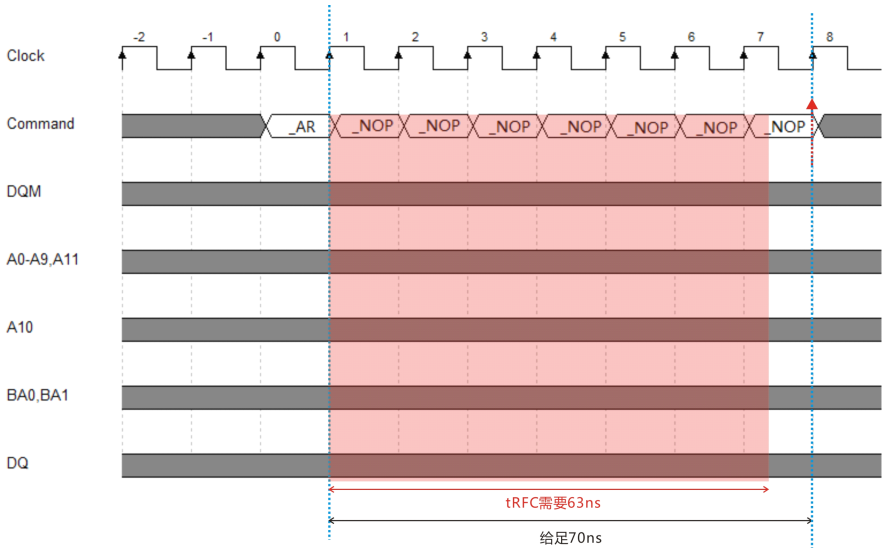


图 3.8g 100Mhz 驱动 SDRAM，自动刷新的理想时序图。

```
else if( Func_Start_Sig[1] )
                case( i )

                    0: // Set IO to input state
                    begin isOut <= 1'b0; rData <= 16'd0; i <= i + 1'b1; end

                    1: // Send Active command with Bank and Row address
                    begin rCMD <= _ACT; rBA <= BRC_Addr[21:8]; i <= i + 1'b1; end

                    2: // Send 1 nop Clk for tRCD-20ns
                    begin rCMD <= _NOP; i <= i + 1'b1; end

                    /****************************************/

                    3: // Send Read command and column address, pull down DQM clock and pull up A10 to auto precharge.
                begin rCMD <= _RD; rBA <= { BRC_Addr[21:20], 4'b0100, BRC_Addr[7:0]}; rDQM <= 2'b00; i <= i + 1'b1; end

                    4,5: // Send 2 nop Clk for CAS Latency
                    begin rCMD <= _NOP; rDQM <= 2'b11; i <= i + 1'b1; end

                    /*****************************************/

                    6: // Read Data
                    begin rData <= SDRAM_DATA; i <= i + 1'b1; end

                    /******************************************/

                    7: // return i to 0
```

```
93.                     begin i <= 4'd0; end
94.
95.                     /*****************************************/
96.
97.                 endcase
```

第 65~97 行是读一字数据的操作，步骤 0 是设置 IO 口输入，步骤 1~6 是读一字数据的相关步骤，步骤 7 是返回的操作，因此读一字数据的功能需要 8 个时钟来操作。

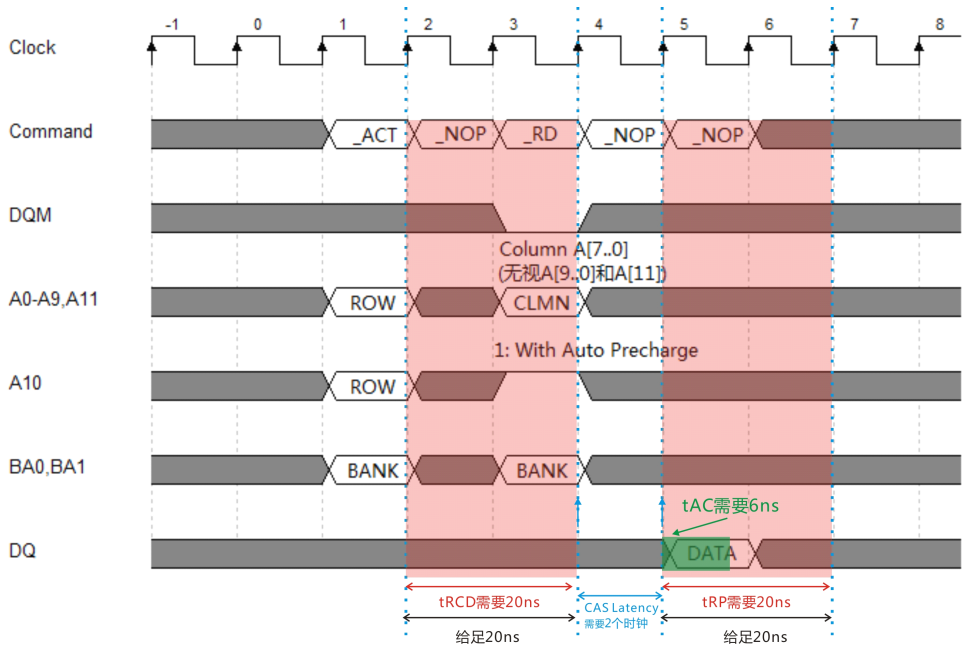


图 3.8h 100Mhz 驱动 SDRAM，读一字的数据的理想时序图。

步骤 1~6 的发生过程如图 3.8h 所示，大家注意那个 tAC 把数据覆盖 60%了 ... 如果设计不苛刻的话，是不要紧的。

```
98.     else if( Func_Start_Sig[0] )
99.                 case( i )
100.
101.                    /*****************************************/
102.
103.                    0: // Set IO to output State
104.                    begin isOut <= 1'b1; i <= i + 1'b1; end
105.
106.                    1: // Send Active Command with Bank and Row address
107.                    begin rCMD <= _ACT; rBA <= BRC_Addr[21:8]; i <= i + 1'b1; end
108.
109.                    2: // Send 1 nop Clk for tRCD-20ns
110.                    begin rCMD <= _NOP; i <= i + 1'b1; end
111.
```

```
112.                    /*********************************************/
113.
114.                    3: // Send Write command with row address, pull down DQM 1 clk and pull up A10 1 clk to Auto Precharge
115.                begin rCMD <= _WR; rBA <= { BRC_Addr[21:20], 4'b0100, BRC_Addr[7:0] }; rDQM <= 2'b00; i <= i + 1'b1; end
116.
117.                    4,5,6,7: // Send 4 nop clk for tDPL/tWR and tRP
118.                    begin rCMD <= _NOP; rDQM <= 2'b11; i <= i + 1'b1; end
119.
120.                    /*********************************************/
121.
122.                    8: // return i to 0
123.                    begin i <= 4'd0; end
124.
125.                    /*********************************************/
126.
127.                endcase
128.
```

第 98~127 行是写一字数据的操作，步骤 0 是用来设置 IO 口为输出，步骤 1~7 是写一字数据的操作，步骤 8 是用来返回。因此写一字操作一共用了 9 个时钟，所以 Func_Start_Sig[0]被拉高的时候，驱动源要确保给足 9 个时钟。具体的操作过程如图 3.8i 所示。

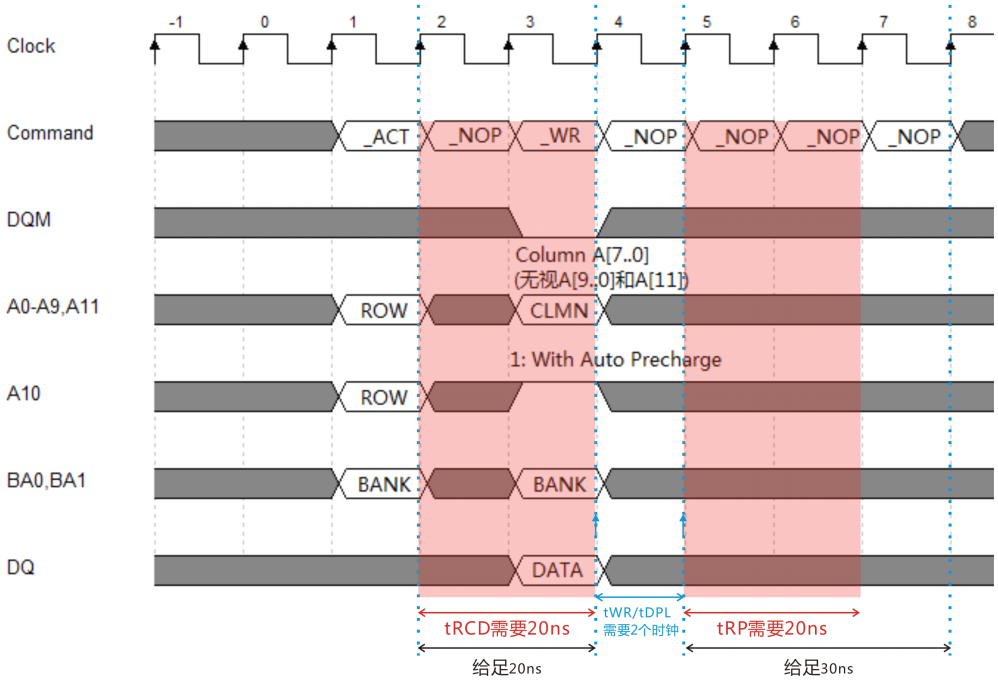


图 3.8i 100Mhz 驱动 SDRAM，写一字操作的理想时序图。

```
129. /*************************************/
130.
```

```
131.        assign SDRAM_CMD = rCMD;
132.        assign SDRAM_BA = rBA;
133.        assign SDRAM_DATA = isOut ? WrData : 16'hzzzz;
134.        assign { SDRAM_LDQM, SDRAM_UDQM } = rDQM;
135.        assign RdData = rData;
136.
137.      /***********************************/
138.
139. endmodule
```

步骤 129~139 是输出的相关配置，读者自己看着办吧。

sdram_control_module.v

比起实验十八的 sdram 控制模块，实验十九的 sdram 控制模块整合了一点，压缩了一点。具体上它有什么改变？我们直接来看代码吧：

```
1.    module sdram_control_module
2.    (
3.          input CLK,
4.          input RSTn,
5.
6.          input WrEN_Sig,
7.          input RdEN_Sig,
8.          output Done_Sig,
9.          output Busy_Sig,
10.
11.         input Init_Done_Sig,
12.         output Init_Start_Sig,
13.
14.         output [2:0]Func_Start_Sig
15.
16.   );
17.
18.       /*********************************/
19.
20.         parameter T15US = 11'd1500;
21.
22.         /*********************************/
23.
```

第 3~14 行是该模块的输入输出声明，注意看！是不是少了 Func_Done_Sig 和 AR_Done_Sig 呢？因为 sdram 功能模块从问答式沟通（Start_Sig 和 Done_Sig 形式）变

成使能式沟通（En_Sig 形式）。第 20 行是 15us 的常量声明。

```
24.         reg [2:0]i;
25.         reg [10:0]C1;
26.         reg [4:0]C2;
27.         reg [4:0]Used;
28.         reg [2:0]rStart;
29.         reg [2:0]isStart; //[2] Auto Refresh , [1] Read Action, [0] Write Action
30.         reg isInit;
31.         reg isBusy;
32.         reg isDone;
33.
34.         always @ ( posedge CLK or negedge RSTn )
35.             if( !RSTn )
36.                 begin
37.                     i <= 3'd6;              // Initial SDRam at first
38.                     C1 <= 11'd0;
39.                     C2 <= 5'd0;
40.                     Used <= 5'd0;
41.                     rStart <= 3'b000;
42.                     isStart <= 3'b000;
43.                     isInit <= 1'b0;
44.                     isBusy <= 1'b1;
45.                     isDone <= 1'b0;
46.                 end
47.             else
```

第 24~32 行是相关的寄存器声明，C1 用于计数自动刷新的时间；C2 用于计数 sdram 功能模块的使能时间；Used 用来寄存 sdram 功能模块各 3 种操作的消耗时间；rStart 用来寄存 Func_Start_Sig 的命令；isStart 则是用来驱动 sdram 功能模块。第 37~45 行是复位操作，除了 isBusy 初值被置一和步骤 i 赋值为 6 以外，其他的寄存器都清零。isBusy 一开始被拉高是用来反馈给上一层模块知道，初始化还在进行中。相比步骤 i 被赋初值为 6，则是使能 sdram 初始化模块。

```
48.         case( i )
49.
50.                 0: // IDLE state
51.                 if( C1 >= T15US ) begin C1 <= 11'd0; Used <= 5'd9; rStart <= 3'b100; i <= 3'd1; end
52.                 else if( RdEN_Sig ) begin C1 <= C1 + 1'b1; Used <= 5'd8; rStart <= 3'b010;i <= 3'd3; end
53.                 else if( WrEN_Sig ) begin C1 <= C1 + 1'b1; Used <= 5'd9; rStart <= 3'b001;i <= 3'd3; end
54.                 else begin C1 <= C1 + 1'b1; end
55.
56.             /***************************/
57.
```

```
58.                     1: // Auto Refresh Done , 9 clock on this step
59.                     if( C2 == Used -1 ) begin C2 <= 5'd0; i <= i + 1'b1; end
60.                     else begin isStart <= rStart; C2 <= C2 + 1'b1; end
61.
62.                     2: // 1 clock one this step
63.                     begin isStart <= 3'd0; C1 <= C1 + 1'b1; i <= 3'd0; end
64.
65.                     /***************************/
66.
67.                     3: // Read and Write Done
68.                     if( C2 == Used -1) begin C2 <= 5'd0; C1 <= C1 + Used; i <= i + 1'b1; end
69.                     else begin isStart <= rStart; C2 <= C2 + 1'b1; end
70.
71.                     /*************************************/
72.
73.                     4: // Generate Done Signal
74.                     begin isStart <= 3'd0; isDone <= 1'b1; C1 <= C1 + 1'b1; i <= i + 1'b1; end
75.
76.                     5:
77.                     begin isDone <= 1'b0; C1 <= C1 + 1'b1; i <= 3'd0; end
78.
79.                     /*****************************************/
80.
81.                     6: // Initial SDRam using 21 clock
82.                     if( Init_Done_Sig ) begin isBusy <= 1'b0; isInit <= 1'b0; C1 <= C1 + 1'b1; i <= 3'd0; end
83.                     else begin isBusy <= 1'b1; isInit <= 1'b1; end
84.
85.                     /******************************************/
86.
87.                 endcase
```

第 48~87 行是该控制模块的核心操作。步骤 6 是一开始的时候，会进入的初始化 SDRAM 的步骤，然后返回步骤 0（第 81~83 行）。读者有没有觉得奇怪，感觉上是不是少了什么东西似地？对，就是少了自动刷新用的计数器。事实上它不是不见了，而是被整合了 ...

步骤 0 可以看成待机状态，如果没有什么用事，C1 会一直累加（第 54 行）。知道第 51 行的 if 条件成立为止，亦即 C1 计数等于或超过 15us 的常量。C1 会被清零，Used 被赋值为 9（sdram 功能模块中，自动刷新的功能需要消耗 9 个时钟），rStart 被赋值为 3'b100（sdram 功能模块中自动刷新的使能命令），最后 i 会指向步骤 1。

步骤 1（第 58~60 行）是用来使能自动刷新功能，其中等价为 3'b100 的 isStart 会被拉高 9 个时钟（第 60 行）。在这里读者一定会感到困惑，为什么第 59 行的 if 条件中有-1？if 条件中的 Used 代表这个步骤停留多少个时钟，其中 -1 表示：在步骤结束的前一个时钟之前清零 C2 和更新步骤 i。这是整合概念的一个基础写法。

步骤 2（第 62~63 行）的作用则是拉低 isStart，模式上有点像使能 FIFO。当步骤 1 执行完毕以后，我们知道一次性的自动刷新已经完成。当步骤进入 2 的时候，C1 就开始为下一次性的自动刷新开始计数，所以第 63 行才有 C1 <= C1 + 1'b1。最后 i 会返回步骤 0。

现在我们又回到待机状态，如果没有其他操作 C1 会一直累加，然后重复上述的操作。假设在这个时候，第 52 行的 if 条件成立（读操作），Used 被赋值为 8（读操作需要消耗 8 个时钟），rStart 被赋值为 3'b010（读操作的使能命令），然后 i 指向步骤 3。在此，从步骤 0 转移到步骤 3 需要 1 个时钟，所以 C1 为自动刷新的计数累加 1。

步骤 3（第 67~69 行）是读操作和写操作被使能的步骤，isStart 会被赋值为 rStart 的结果，C2 用来计数使能所使用的时钟。第 67 行的 if 条件中的 Used 表示该步骤停留多少个时钟（使能多少个时钟），其中 -1 表示 C2 的清零，步骤 i 的更新为 4，此外还为自动刷新的计数更新等操作，都发生在结束前的一个时钟。

在这里读者要明白，当 sdram 控制模块使能 sdram 功能模块的读操作还是写操作的时候，自动刷新的计数依然在发生着。每当完成读操作和写操作以后，我们必须为自动刷新的计数更新相关的值。因此才有 C1 <= C1 + Used。在此，步骤 3 充当使能读操作或者写操作多少个时钟，反之步骤 4 除了产生完成信号以外还有拉低 isStart 的作用（不使能）。

步骤 4~5（第 73~77 行）是用来产生完成信号，在产生完成信号的过程一共消耗 2 个时钟。因此我们需要为自动刷新的计数累加两个时钟，其中一个时钟在步骤 4（第 73~74 行），另一个时钟在步骤 5（第 76~77 行）。最后步骤会返回 0。如果此时的 C1 累加结等于或超过 15us 的常量，那么会进入自动刷新的操作 ... 反之会在待机状态中继续累加。

到目前为止，读者是不是觉得很有意思，就算没有建立针对自动刷新的计数器，只要懂得整合的概念和精密的控制时钟。在不知不觉中，我们依然可以为计数激动刷新的周期。

```
88.
89.         /****************************************/
90.
91.         assign Init_Start_Sig = isInit;
92.         assign Func_Start_Sig = isStart;
93.         assign Done_Sig = isDone;
94.         assign Busy_Sig = isBusy;
95.
96.         /****************************************/
97.
98.   endmodule
```

第 88~98 行是输出的相关配置 ... 读者自己看着办吧。

sdram_module3.v

```
1.  module sdram_module3
2.  (
3.          input CLK,
4.          input RSTn,
5.
6.          input WrEN_Sig,
7.          input RdEN_Sig,
8.          output Done_Sig,
9.          output Busy_Sig,
10.
11.         input [21:0]BRC_Addr,
12.         input [15:0]WrData,
13.         output [15:0]RdData,
14.
15.         output [4:0]SDRAM_CMD,
16.         output [13:0]SDRAM_BA,
17.         inout [15:0]SDRAM_DATA,
18.
19.         output SDRAM_LDQM,
20.         output SDRAM_UDQM
21.
22. );
23.
24.       /***********************************/
25.
26.         wire Init_Start_Sig;
27.         wire [2:0]Func_Start_Sig;
28.         wire [15:0]Func_Write_Data;
29.
30.         wire Init_Done_Sig;
31.
32.       sdram_control_module U1
33.         (
34.                 .CLK( CLK ),
35.                 .RSTn( RSTn ),
36.                 .WrEN_Sig( WrEN_Sig ),                  // input - from top
37.                 .RdEN_Sig( RdEN_Sig ),                  // input - from top
38.                 .Done_Sig( Done_Sig ),                  // output - to top
39.                 .Busy_Sig( Busy_Sig ),                  // output - to top
40.                 .Init_Start_Sig( Init_Start_Sig ),      // output - to U2
41.                 .Func_Start_Sig( Func_Start_Sig ),      // output - to U3
42.                 .Init_Done_Sig( Init_Done_Sig )
```

```
43.         );
44.
45.         /**********************************/
46.
47.         wire [4:0]U2_SDRAM_CMD;
48.         wire [13:0]U2_SDRAM_BA;
49.
50.         sdram_init_module U2
51.         (
52.                 .CLK( CLK ),
53.                 .RSTn( RSTn ),
54.                 .Init_Start_Sig( Init_Start_Sig ),    // input - from U1
55.                 .Init_Done_Sig( Init_Done_Sig ),
56.                 .SDRAM_CMD( U2_SDRAM_CMD ),                // output - to selector
57.                 .SDRAM_BA( U2_SDRAM_BA )                   // output - to selector
58.         );
59.
60.         /**********************************/
61.
62.         wire [15:0]Func_Read_Data;
63.         wire [4:0]U3_SDRAM_CMD;
64.         wire [13:0]U3_SDRAM_BA;
65.
66.         sdram_func_module U3
67.         (
68.                 .CLK( CLK ),
69.                 .RSTn( RSTn ),
70.                 .Func_Start_Sig( Func_Start_Sig ),    // input - from U1
71.                 .BRC_Addr( BRC_Addr ),                     // input - from top
72.                 .WrData( WrData ),                         // input - from top
73.                 .RdData( RdData ),                         // output - to top
74.                 .SDRAM_CMD( U3_SDRAM_CMD ),                // output - to selector
75.                 .SDRAM_BA( U3_SDRAM_BA ),                  // output - to selector
76.                 .SDRAM_DATA( SDRAM_DATA ),                 // output - to top
77.                 .SDRAM_LDQM( SDRAM_LDQM ),
78.                 .SDRAM_UDQM( SDRAM_UDQM )
79.
80.         );
81.
82.         /**********************************/
83.
84.         //Selector of SDRAM_CMD pin and SDRAM_BA pin
85.
86.         reg [4:0]rCMD;
```

```
87.         reg [13:0]rBA;
88.
89.         always @ ( * )
90.               if( Init_Start_Sig ) begin rCMD = U2_SDRAM_CMD; rBA = U2_SDRAM_BA; end
91.                 else if( Func_Start_Sig ) begin rCMD = U3_SDRAM_CMD; rBA = U3_SDRAM_BA; end
92.                 else begin rCMD = 5'bxxxxx; rBA = 14'bxxxxxxxxxxxxxx; end
93.
94.         assign SDRAM_CMD = rCMD;
95.         assign SDRAM_BA = rBA;
96.
97.         /***********************************/
98.
99.   endmodule
```

上面的代码是组合模块 sdram_module3.v，内容基本上和图 3.8e 的建模图大同小异。

sdram_demo3.v

```
1.    module sdram_demo3
2.    (
3.          input CLK,
4.          input RSTn,
5.
6.          output SDRAM_CLK,
7.          output [4:0]SDRAM_CMD,
8.          output [13:0]SDRAM_BA,
9.          inout [15:0]SDRAM_DATA,
10.
11.         output SDRAM_UDQM,
12.         output SDRAM_LDQM,
13.
14.         output [3:0]LED
15.   );
16.
17.         /********************************/
18.
19.         wire CLK_100Mhz;
20.         wire CLK_100Mhz2;
21.
22.         pll_module U1
23.         (
24.                     .inclk0 ( CLK ),
25.                     .c0 ( CLK_100Mhz ),
```

```
26.                 .c1 ( CLK_100Mhz2 )
27.         );
28.
29.         /*******************************/
30.
31.         wire Done_Sig;
32.         wire Busy_Sig;
33.         wire [15:0]RdData;
34.
35.         sdram_module3 U2
36.         (
37.             .CLK( CLK_100Mhz ),
38.             .RSTn( RSTn ),
39.             .WrEN_Sig( isWrite ),
40.             .RdEN_Sig( isRead ),
41.             .Done_Sig( Done_Sig ),
42.             .Busy_Sig( Busy_Sig ),
43.             .BRC_Addr( rBRC ),
44.             .WrData( rData ),
45.             .RdData( RdData ),
46.             .SDRAM_CMD( SDRAM_CMD ),
47.             .SDRAM_BA( SDRAM_BA ),
48.             .SDRAM_DATA( SDRAM_DATA ),
49.             .SDRAM_UDQM( SDRAM_UDQM ),
50.             .SDRAM_LDQM( SDRAM_LDQM )
51.         );
52.
53.         /***********************************/
54.
55.         parameter T3S = 29'd300000000, T1S = 29'd100000000;
56.
57.         /***********************************/
58.
59.         reg [6:0]i;
60.         reg [28:0]C1;
61.         reg [4:0]x;
62.         reg [15:0]rData;
63.         reg [21:0]rBRC;
64.         reg [3:0]rLED;
65.         reg isWrite;
66.         reg isRead;
67.
68.         always @ ( posedge CLK_100Mhz or negedge RSTn )
69.             if( !RSTn )
```

```
70.                 begin
71.                     i <= 7'd0;
72.                     C1 <= 29'd0;
73.                     x <= 5'd1;
74.                     rData <= 16'd0;
75.                     rBRC <= 22'd0;
76.                     rLED <= 4'd0;
77.                     isWrite <= 1'b0;
78.                     isRead <= 1'b0;
79.                 end
80.             else
81.                 case( i )
82.
83.                     0: // waiting SDRAM initial
84.                     if( !Busy_Sig ) i <= i + 1'b1;
85.
86.                     1:
87.                     if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
88.                     else begin isWrite <= 1'b1; rData <= 16'h8421; rBRC <= 22'd0; end
89.
90.                     2:
91.                     if( Done_Sig ) begin isRead <= 1'b0; i <= i + 1'b1; end
92.                     else begin isRead <= 1'b1; rBRC <= 22'd0; end
93.
94.                     3:
95.                     begin rLED <= RdData[3:0]; i <= i + 1'b1; end
96.
97.                     4:
98.                     if( C1 == T1S -1) begin C1 <= 29'd0; i <= i + 1'b1; end
99.                     else C1 <= C1 + 1'b1;
100.
101.                    5:
102.                    if( Done_Sig ) begin isRead <= 1'b0; i <= i + 1'b1; end
103.                    else begin isRead <= 1'b1; rBRC <= 22'd0; end
104.
105.                    6:
106.                    begin rLED <= RdData[7:4]; i <= i + 1'b1; end
107.
108.                    7:
109.                    if( C1 == T1S -1) begin C1 <= 29'd0; i <= i + 1'b1; end
110.                    else C1 <= C1 + 1'b1;
111.
112.                    8:
113.                    if( Done_Sig ) begin isRead <= 1'b0; i <= i + 1'b1; end
```

```
114.                    else begin isRead <= 1'b1; rBRC <= 22'd0; end
115.
116.                    9:
117.                    begin rLED <= RdData[11:8]; i <= i + 1'b1; end
118.
119.                    10:
120.                    if( C1 == T1S -1) begin C1 <= 29'd0; i <= i + 1'b1; end
121.                    else C1 <= C1 + 1'b1;
122.
123.                    11:
124.                    if( Done_Sig ) begin isRead <= 1'b0; i <= i + 1'b1; end
125.                    else begin isRead <= 1'b1; rBRC <= 22'd0; end
126.
127.                    12:
128.                    begin rLED <= RdData[15:12]; i <= i + 1'b1;end
129.
130.                  /****************************************/
131.
132.                    13:
133.                    if( C1 == T1S -1) begin C1 <= 29'd0; i <= i + 1'b1; end
134.                    else C1 <= C1 + 1'b1;
135.
136.                    14:
137.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
138.                    else begin isWrite <= 1'b1; rData <= 16'd1; rBRC <= 22'd1; end
139.
140.                    15:
141.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
142.                    else begin isWrite <= 1'b1; rData <= 16'd2; rBRC <= 22'd2; end
143.
144.                    16:
145.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
146.                    else begin isWrite <= 1'b1; rData <= 16'd3; rBRC <= 22'd3; end
147.
148.                    17:
149.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
150.                    else begin isWrite <= 1'b1; rData <= 16'd4; rBRC <= 22'd4; end
151.
152.                    18:
153.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
154.                    else begin isWrite <= 1'b1; rData <= 16'd5; rBRC <= 22'd5; end
155.
156.                    19:
157.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
```

```
158.                    else begin isWrite <= 1'b1; rData <= 16'd6; rBRC <= 22'd6; end
159.
160.                    20:
161.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
162.                    else begin isWrite <= 1'b1; rData <= 16'd7; rBRC <= 22'd7; end
163.
164.                    21:
165.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
166.                    else begin isWrite <= 1'b1; rData <= 16'd8; rBRC <= 22'd8; end
167.
168.                    22:
169.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
170.                    else begin isWrite <= 1'b1; rData <= 16'd9; rBRC <= 22'd9; end
171.
172.                    23:
173.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
174.                    else begin isWrite <= 1'b1; rData <= 16'd10; rBRC <= 22'd10; end
175.
176.                    24:
177.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
178.                    else begin isWrite <= 1'b1; rData <= 16'd11; rBRC <= 22'd11; end
179.
180.                    25:
181.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
182.                    else begin isWrite <= 1'b1; rData <= 16'd12; rBRC <= 22'd12; end
183.
184.                    26:
185.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
186.                    else begin isWrite <= 1'b1; rData <= 16'd13; rBRC <= 22'd13; end
187.
188.                    27:
189.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
190.                    else begin isWrite <= 1'b1; rData <= 16'd14; rBRC <= 22'd14; end
191.
192.                    28:
193.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
194.                    else begin isWrite <= 1'b1; rData <= 16'd15; rBRC <= 22'd15; end
195.
196.                    /************************************/
197.
198.                    29:
199.                    if( C1 == T1S -1) begin C1 <= 29'd0; i <= i + 1'b1; end
200.                    else C1 <= C1 + 1'b1;
201.
```

```
202.                    30:
203.                    if( x == 16 ) begin x <= 5'd0; i <= i + 7'd3; end
204.                    else if( Done_Sig ) begin isRead <= 1'b0; i <= i + 1'b1; end
205.                    else begin isRead <= 1'b1; rBRC <= {17'd0, x}; end
206.
207.                    31:
208.                    begin rLED <= RdData[3:0]; x <= x + 1'b1; i <= i + 1'b1; end
209.
210.                    32:
211.                    if( C1 == T1S -1) begin C1 <= 29'd0; i <= i - 7'd2;; end
212.                    else C1 <= C1 + 1'b1;
213.
214.                    33:
215.                    if( Done_Sig ) begin isWrite <= 1'b0; i <= i + 1'b1; end
216.                    else begin isWrite <= 1'b1; rData <= 16'd0; rBRC <= 22'd0; end
217.
218.                    34:
219.                    if( Done_Sig ) begin isRead <= 1'b0; i <= i + 1'b1; end
220.                    else begin isRead <= 1'b1; rBRC <= 22'd0; end
221.
222.                    35:
223.                    begin rLED <= RdData[3:0];    end
224.
225.                    /****************/
226.
227.                endcase
228.
229.        /**************************/
230.
231.        assign SDRAM_CLK = CLK_100Mhz2;
232.        assign LED = rLED;
233.
234.        /***************************/
235.
236. endmodule
```

实验十九 sdram_demo3.v 的内容和实验十八 sdram_demo.v 的内容基本上改变不大，除了第 19~27 行的 pll 实例化以外，还有就是修改了第 55 行的常量声明。测试的方式依然是先测试 16 位数据的正确性，然后测试连续读和写的正确性，读者自己看着办吧。

==========

就这样，SDRAM 的实验就到此为止了 .... 读者别这样舍不得，笔者也是舍不得 ... 逼近受到页数的限制，笔者不被允许继续写下去了。此外，越深入讨论就会越偏离笔记所

制定的范围，和牺牲更多的脑细胞和头发。美味的佳肴都是一吃不到 3，高手切磋都是点到即止，重点讨论也是如此。

在这一小节里，我们学习到为什么 sdram 倍频到 100Mhz 的时候，就要把 SDRAM_CLK 修改为 -75° 相位。当然，这小节最重要的内容依然是关于理想时序的概念。这一小节的结论是：无论环境怎样改变，理想的工具（Verilog HDL 语言）还是理想，运用的效果也是理想（Verilog 的理想时序）。Verilog HDL 语言虽然可以涉及和满足自身以外的工作（第二层时间要求），但是却不能解决它 ...

总结：

啊 ... 终于写完了这一长篇怪兽，真是一节比一节更难写。在第三章里我们讨论了有关“速率”的概念，速率这东西本来就不稀有，只是许多参考书敷衍行事而已。明白“速率”是什么东西，能助于我们更了解 Verilog 理想时序的本质，然后从理想时序和物理时序之间需找平衡点（粘糊剂）。

关于这一点非常重要的，速率的定义就是一位数据停留的时间，这也是“覆盖”概念的所需要用到的基础知识。如此类推，“覆盖”概念又是寻找平衡点的基础知识 ... 很遗憾的是，很多同学在一开始往死胡同的方向。他们不知道有理想时序的存在，更不知道速率和理想时序之间的关系。

当他们们学到某个程度的时候，心里就会悬起一种“连接不上”的感觉，因为他们找不到 Verilog 和物理时序之间的平衡点。读者不要忘了 Verilog HDL 语言是一种用 N 个时钟去计算的理想时序，而不是用多少 ns 或者 ps 去计算的物理时序。打从心里没有“理想时序”这一概念的存在，根本就找不到所谓的“平衡点”。这也是没有把 Verilog HDL 语言当成理想工具来看待的结果 ... 在这里读者要明白，理想时序的概念可以兼容物理时序的概念，反过来物理时序的概念却兼容不了理想时序的概念。

旅行有 5000 元，用不着怕 400~500 元小钱的花费，这就好比理想时序兼容物理时序的情况。换之，就算我们知道旅行最多只需要花费 500 元，当我们带足 500 元去旅行，心里就会有一种莫名的不安 .... 担心这个，担心那个 ... 这就好比只晓得物理，却不晓得理想时序。笔者所谓的“黏糊/平衡点”就是那种，旅行不用担心预期的花费不够，也不怕意外的花费。因为只要有 5000 元在手，有没有而外的花费，旅行都可以在“不用担心”的状态下顺利完成。当然，笔者的意思不是钱多就好办事 ... 这只是比喻而已。

寻找理想时序和物理时序的平衡点还有一个重点，有如实验十八和实验十九的情况。SDRAM 的驱动从 20Mhz 变成 100Mhz，对于物理时序来说，时序图中的信号早已经失去的对齐性。反之，理想时序只是更动数据的覆盖而已，时序图中的信号依然保持着对齐性。

第四章 浮点数那些事儿

在整合篇的第一章到第三章，中心内容不过与“精密控时”... 具体的经过，包括“整合概念”的基本经过一段长时间的研究，笔者终于迎来整合篇的下半部了，整合篇的第一章到第三章由于受到页数的限制，此外笔者也在第一章到第三章用光了资料 ... 说实在非常郁闷，收集资料是在编辑笔记中最耗时间和精力的一段过程，期间不仅要大量过滤和理解那些莫名奇妙的资料，而且还要不停的吸收断断续续资料，时不时会发生会消化不良或者大脑短路等现象。好了，牢骚就先发到这里 ... 我们继续 Verilog HDL 语言的学习吧。

接下来的内容，同样也是爬满了刺猬和毛毛虫，问读者有没有勇气和笔者一同走下去了？呵呵，答案是绝对的 ... 那些打开笔记的读者，在某种程度上已经有了相当的觉悟。嗯，好样！深呼吸，集中精神，放松神经，让我们继续走下去。

4.1 闻风色变的浮点数

当我们听到定点立即就会联想到定点整数和定点小数，打个比方（如代码 4.1.1 所示）：

```
定点整数 7₁₀    = (符号位)111₂
定点小数 7.7₁₀ = (符号位)111.111₂
```

代码 4.1.1 定点整数和小数的表示方法

定点整数一般上都简称为“整数”；定点小数一般上不常使用，除了一些特殊的场合。在 c 语言里，整数是正规的“数据格式”，相比之下那些有小数的数字都会用浮点数来表示。“浮点数是什么”估计只有蛋疼的同学才会问，学过 c 语言的朋友都知道，浮点数是（代码 4.1.2 所示）：

```
float pi;          //声明
pi = 3.142;        //赋值
if( pi == 3.142 ) //判断
pi = pi * 2.3;     //运算
```

代码 4.1.2 浮点数在 C 语言（高级语言）中的表示和操作

对呀！关于浮点数的大小姨妈就只有这些东西而已，难道还有其他吗？这个现象如同我们知道某个产生，却不知道某个经过。在高级语言的世界里，我们当然可以一只眼开一只眼关，让这些猥琐的经过都交给编译器去干，自己在旁翘脚等着收钱就是了。呵呵！高级语言和编译器多么便利呀，这也是为什么美国人的 1/3 人口都是肥胖者。

以前的笔者也是如此，今天的读者也是如此，从 C 语言过渡到 Verilog HDL 语言，不知不觉之间养成贪图效率和便利的小胖子了。直到某一天，笔者被某个 Verilog 医生检验出患有“肥胖病症”（懒惰病），笔者晴天霹雳一般，知道自己已经接近踩入不可救药的边缘了。笔者诚心祷告“掌管 Verilog 的神呀，请救救那个病入膏肓的自己吧 ...”，

那一夜，Verilog 之神在梦里对笔者说：

“孩子，在 Verilog 的世界里高级语言是 KFC、是 McDoland、是 PizzaHut、是不则不扣的快餐。‘食’用虽然方便又美味，但是它们都是毒，不但危害健康，而且还会麻痹大脑。孩子你应该放眼在 Verilog 的料理上，Verilog 料理的烹饪方法不仅煎炸一种而已，而是多样百出。此外，Verilog 料理所强调的不只是某个程度的便利和美味而已，除了色香味具备以外，还有营养均衡。”

神明永远不会告诉人们直接的含义，笔者不明白 Verilog 之神这一番对话，然后笔者焦虑的反问道：“神呀！孩子还是不明白 ...”

Verilog 之神抚摸了笔者的头，继续话到：“孩子，你是聪慧的。你既然懂得建模技巧，同样也可以明白这一番话 ... 去研究浮点数吧，孩子！记得用 Verilog 的角度去研究浮点数 ...”

就这样，笔者从梦中惊醒。笔者打从心底明白，这是 Verilog 之神给笔者的一个指引。笔者面对整数乘法器的时候曾经那样想过，笔者只要在文本编辑中输入“*”，那么乘法器——更确切说乘法的组合逻辑会经过编译器而形成，写法如下（如代码 4.1.3 所示）：

```
reg [7:0]rA,rB;
reg [15:0]rData;

always @ ( posedge CLK ... )
    ......
    rData <= rA * rB;
```

代码 4.1.3 在 Verilog 中实现最简单的乘法

在代码 4.1.3 中，rA 和 rB 均为乘法的输入，它们拥有同样的 8 位位宽。更具最简单的整数乘法规则，rData 输出的位宽为 rA 和 rB 的总和。笔者尝试的问自己，“*”是如何实现乘法操作的？那时候的笔者还是菜鸟一只，笔者翻查了许多的资料才知道，“*”乘法操作符，一是用 FPGA 设备自身的乘法硬件来实现；二是用组合逻辑以逻辑门的方法门来实现。前者消耗宝贵的乘法硬件，后者是消耗大量的组合逻辑。

那些有接触过 ASCI 设计的朋友应该知道，随着输入（操作数 a 和 b）的位宽增加，逻辑门的会使用越来越多，而且“组合逻辑的层级”会越来越长，组合逻辑的延迟也会越来越久。相反的，如果读者是用 C 语言来实现整数乘法的话（如代码 4.1.4 所示）...

```
int a,b,data;

data = a * b;
```

代码 4.1.4 C 语言的乘法实现

如同代码 4.1.4 所示那样，就是几段代码而已，至于乘法操作时如何发生，而且如何设计（硬件的？还是软件的？）使用者是被屏蔽的猪头。在此，笔者尝试站在 Verilog 的角度继续思考：

第一个问题：“如果某个乘法的输入同样是 8 位宽，输出一定要 16 位宽吗？”

第二个问题："Verilog 的乘法只有一种方法而已吗？（硬件还是软件）"

从中笔者明白到 Verilog 是没有固定性，Verilog 有即时事件和时间点事件之分（这就是创作时序篇的灵感）。笔者越往后想，感觉越是无底层，头脑越是发疼 ... 忽然间笔者明白到一为什么那么多人会喜欢快餐式的高级语言，它是"快餐美味方便"，很可惜 Verilog HDL 语言不是快餐。

如果 Verilog HDL 语言不是快餐的话，那么那些患有快餐上瘾症的使用者该怎么办？(根据资料显示,KFC 和 McDoland 等煎炸和高脂肪的肉类食物会使食用者上瘾)。简单呀,只要调用官方的 ip/lpm 或者消耗更多的逻辑资源，什么乘法都能实现呀。在学习的经过笔者遇到一种现象，原本调用官方 ip/lpm 是一件再普通为过的事情，和食用快食一样偶尔没什么关系，如果长时间养成习惯的话就糟糕了。

我们必须知道，ip/lpm 作为官方产品的一部分一定存在版权的问题，但是官方又不能把所有内容收的死死。一般情况下 ip/lpm 的内容都是"半开半关"，读者尝试打开其中一个 ip/lpm 开看看，读者一定会被内容吓死 ... 理由笔者不用说了，任何人都有类似的经验。所以说，如果读者打算从官方的 ip/lpm 偷学几招盖世功夫还是省了吧，看死不偿命，就算看懂了价值性也是太低 ... 这有点像快食的黑幕，我们明知快食危害健康，却无法抗拒它 ...呵呵，话题有点扯远了。

那些从时序篇过来的朋友,多少都知道要设计一个乘法器不是想象中的简单,也不是想象中的苦难。但是问题就在于，设计短整型乘法器已经够猥琐了，换作浮点数无非不是更猥琐吗？在这里，可能有同学会回答："用 ip!"... 如果真的觉得这是最好决定，那么就这样作吧！随后可以把这本笔记关闭。

看来，我们已经接近这个章节的重点了，我们先看几段代码（代码 4.1.5）：

```
1.  float FloatMulti( float A, float B ){ ... }; //函数声明和定义
2.
3.  main()
4.  {
5.      float pi;    //浮点数变量声明
6.      float Temp;
7.
8.      pi = 3.142;                    //赋值
9.      Temp = pi * pi;                //直接运行浮点数的乘法操作
10.     Temp = FloatMulti( pi , pi ); //调用函数实现浮点数的乘法操作
11. }
```

代码 4.1.5 C 语言中实现浮点数运算

代码 4.1.5 是一段很简单的 C 语言代码，第 9 行是直接使用"*"操作符实现乘法的运算，换之第 10 行我们调用函数实现乘法元算。第 8~9 行的运算结果基本上都是一模一样。当我们点击"编译"的一瞬间，一些不为人知的肮脏工作在台面下发生着。我们这些使用者就像笨蛋一样，浮点数是如何执行和运算我们亚丁儿也不知道。

有些同学估计会反驳道："C 语言就是这样使用的啦！那里有问题？"正确！原本 C 语言的发明就

是为了实现代码的可移植性，使用者的工作就是驾驭语言。当然，这些事情都是发生在高级语言的世界里。换作 Verilog HDL 语言，难道我们又妄想编译器帮我们实现一切吗？在扫盲文里，笔者已经说过 Verilog HDL 语言在层次上是低于 C 语言（高级语言），不过这也是大众的一般认识而已。

笔者还说过，Verilog HDL 语言是非常自由的语言，甚至是自由到没有结构，相比 C 语言（高级语言）从出生以来就有固定的结构。如果 Verilog HDL 语言没有自定义的结构，必然是依赖编译器和 ip。反之，如果 Verilog HDL 语言有自定义的结构，我们就可以按着自定义的结构来设计专属的 ip。

继续把问题往后推的话，要实现浮点数的加减乘除运算——这个问题一定会把小屁孩吓得失禁。参考书？不要笑坏笔者的肚子，参考书都有影子没实体；网页？维基百科？博客？这些来源是笔者视为最好的参考资料，但是大多数的内容都是断断续续，内容有限；权威手册？怒笔者会消化不良，不过一些关键的内容还可以看懂。

上述的内容就是笔者要面对的问题 ... 更糟的是，它们都和 Verilog HDL 语言一点关系也没有。如果笔者没有建模技巧，如果笔者更不懂 Verilog HDL 的理想时序，笔者只能说是雪上加霜而已。浮点数不是妖怪，它之所以让人闻风色变是背后的不知，不足和不安。不懂得 Verilog 世界的自然规则是不知（Verilog 的建模技巧和理想时序）；没有足够的资料是不足。最直接的问题还是自身的心境，起伏不定的心境容易产生不安。

读者明白了吧，这就是在 Verilog 的世界里，所看到的“自然现象”。高级语言和 Verilog 的世界各自都有自身的自然规则，反自然者死，自由顺从则小苗亦然茁壮成长，流水也会无孔不穿。当我们在不同世界里互相“呼唤信息”的时候，一定要非常谨慎，不要忘了自己是在那个世界去窥视那个世界，绝对不能打破自身存在在那个世界的自然规则。
Verilog 之神为什么把笔者指引到这种窘境？笔者不懂，但是笔者相信神明的指明。

4.2 浮点数和单精度格式

在 4.1 小节里笔者说了一大堆废话的目的只有一个，那就是让读者有思想准备。因为任何人一旦听到“浮点数”这 3 个字，就会莫名的焦虑起来，笔者也是如此。首先我们先了解一个简单的问题：“什么是浮点数？”这个问题看似简单，实际上隐藏着许多信息。

```
float pi;
pi = 3.142;
```

代码 4.2.1 一个实例

如代码 2.1.1 所示，我们先声明一个浮点数的变量 pi，然后将 3.142 赋值给 pi。在此，我们会产生一个错觉，不知不觉的情况下我们会把 3.142 认为浮点数。嗯，这是浮点数最普遍的认知错误。我们还是从数学的角度去了解“什么是浮点数”。

3.142 是以 10 进制表示“拥有小数”的数字：3 是整数；.142 是小数。如果换成浮点数的话：

```
Number x B^Exp ( B:Base 进制 Exp:Exponent 价码)

1.  3.142 x 10^0
2.  0.3142 x 10^1
3.  31.42 x 10^-1
```

普通的浮点数，数字一定会伴随价码(Exponent),如第一行的 3.142×10^0，“3.142”就是数字，“10^0”就是价码。随着小数点左移/右移，价码会递增或者递减，无论小数怎么移动“数字的基本值”都不会改变。在计算机里浮点数有特定的表示方式：

```
3.142E0
0.3142E+1 / 0.3142E1
31.42E-1
```

在前面笔者已经说过了，既然我们现在是站在 Verilog 的世界去理解浮点数。所以说，那一套用在高级语言世界里的“自然规则”，当然不能套用在 Verilog 的世界里。假设我们要为某个寄存器赋予浮点数的值，如代码 4.2.2 所示：

```
1.  reg [31:0]rPi
2.  rPi <= 32'b0_10000000_10010010001011010000111
3.  rPi <= 3.142;
```

代码 4.2.2 Verilog HDL 语言里浮点数的表示方法。

第 1 行是 32 寄存器的声明；第 2 行是 3.142 按 ieee 单精度格式赋值；反之第 3 行如同高级语言那样赋值。第 2 行的赋值方法明显是遵守 Verilog 的“自然规则”，至于第 3 行是打破自然规则和强硬的赋值方法。第 3 行的方法在 Quartus II 版本 9.0c 以上（估

计版本 7.0 以上已经可以了）已经支持了，但是唯一不足的是编译器会按 ieee 双精度的规定格式去赋值。

什么是 ieee 的单精度和双精度格式，估计那些学过 C 语言的同学都略懂一二，为了使内容更全面一点，笔者还是简单的介绍一下。IEEE 对于浮点数的规定格式如表 4.2.1 所示。（笔者忘了是哪一个 IEEE 标准）：

表 4.2.1 单精度和双精度浮点数的格式。

浮点数	符号 Sign	价码 Exponent	尾数 Mantissa
单精度 32 位宽	[31]-1 位	[30..23]-8 位	[22..0]-23 位
双精度 64 位宽	[63]-1 位	[62..52]-11 位	[51..0]-52 位

看到这里，估计有些同学已经慌了，已经开始不晓得什么在打什么。事实上，理解单精度和双精度是大一的功课，尤其是“计算机组成原理”哪一环里。不知道这些同学是如何“睡过”大一或者及格考试？笔者真的很好奇 ... 笔者举个简单的例子，就以 3.142 为例：

```
3.142 转换为单精度：

  3₁₀     = 0b11
.142₁₀    = 0b00100100010110100001110 0（实际上是非常的长，因此笔者只摘取前 24 位）

定点小数的表示方法为  = 0b11.00100100010110100001110 0      （小数源）
浮点数的表示方法为    = 0b11.00100100010110100001110 0E+0
                    = 0b1.100100100010110100001110 0E+1   （浮点数源）
```

到目前为止“0b1.100100100010110100001110 0E+1”可以称为 2 进制的“浮点数源”，虽然我们已经得到“浮点数源”，但这并不表示我们的工作已经结束。我们还要把“浮点数源”套在单精度的格式里。在此之前，我们先看数学是如何表示“浮点数”。

```
浮点数      = (Sign x -1) Mantissa^Exponent

 (中文)     = (符号位 x -1) 尾数^价码

3.142       =  0b (0 x -1) 1.100100100010110100001110 0^1
            =  0b1.100100100010110100001110 0^1

计算机      =  0b1.100100100010110100001110 0E+1
```

单精度格式的变量还没有赋值之前，它有固定的“默认空壳”。

```
[ 0][0111111][00000000000000000000000]  //默认空壳
[31][-30:23-][---------22:0----------]
[符][-价码--][---------尾数----------]
```

这个“默认空壳”除了符号位，和尾数以外，最让人匪夷所思的就是价码。价码的默认值不是 0 而是 0b01111111，亦即 8'd127。假设我们要把 3.142 的“浮点数源”套入这个空壳里边，让我们瞧瞧在套入的过程中会发生什么事？

```
浮点数源 = 0b1.10010010001011010000011100E+1


默认空壳的符号位[31]      = 0
默认空壳的价码[30..23]    = 0b01111111 + (E+1)
                          = 0b01111111 + 1
                          = 0b10000000
默认空壳的尾数[22..0]     = .[10010010001011010000111]00  //小数点后的 23 位


因此 3.142 的单精度表示是 = 32'b0_10000000_10010010001011010000111
```

看到了吧，过程就是这样简单而已 ... “呜呜呜呜”读者别发出这样的声音，笔者知道读者在忧虑什么。读者是不是想问：“为什么空壳的价码是 0b0111111？”和“为什么空壳的尾数只取浮点数源小数点后的 23 位？”。嗯~这是说来话长的故事，首先笔者不是单精度的发明人，当然不知道作者葫芦里卖什么药？

但是从使用者的角度来说，空壳的价码之所以是 0b0111111 是为了简化计算。至于空壳的尾数取“浮点数源”小数点后的 23 位，作者自身称为“隐藏位技术”。因为，任何的“浮点数源”都是“1.**1110**E-10 或者 -1.**0111**E+3”等格式。无论浮点数源是负值还是正直结果都是 1.XXXXE±X。因此小数点前的 0b1 可见是一个常量，在使用者的眼里它的存在是不会被动摇，没有表示出来也不用紧。

说几句讽刺的话，任何“XX”只要在后面加上“技术”二字，“XX”无论是什么简单的原理，加上“技术”二字后，发音都会无比响量。我们常常在文章里都会看见“什么什么产品佩戴 XX 技术”或者“什么什么公司有 XX 技术”... 当我们看到“XX 技术”，在心底里就会无比崇拜和敬畏。哎~现在的人类真是可悲，原因就是我们不知道“XX”到底是什么，才会觉得“XX”很厉害，很伟大。佛家说过一句“无知就是罪过”，笔者深深感受得到。

笔者再举个例子，假设笔者想把 -0.2012 用单精度表示：

```
先取得 -0.2012 的二进制表示：
-0.2012₁₀ = 0b-0.00110011100000011101011111 0（尾数太长了，仅摘取前小数点后的 27 位）


把 0b-0.00110011100000011101011111 0 换成浮点数源：
0b-0.00110011100000011101011111 0 = 0b-1.10011100000011101011111 0E-3


把浮点数源套入单精度的空壳里：
默认空壳的符号位[31] = 1 (浮点数源为负值)
默认空壳的价码[30:24]     = 0b01111111 + (E-3)
```

```
                    = 0b0111111 - 3
                    = 0b01111100
默认空壳的尾数[22:0]      = .[10011100000011101011111]0 （摘取浮点数源小数点后的23位）


-0.2012的当精度表示 = 32'b1_01111100_10011100000011101011111
```

好了，单精度的浮点数表示方法就是那么简单而已，笔者总结一下大概的步骤：

（一）把十进制的数字转换为二进制。
（二）把二进制的数字转换为浮点数源。
（三）根据浮点数源的情况，把相关的内容套入单精度-默认空壳的相关地方。

在这里，有些同学可能会上面的任务太过于巨大，如同拯救地球那么困难。确实如此，当我们使用高级语言的时候（如 C 语言），我们都会过度依赖编译器。还记得笔者之前说过的“快食上瘾症”吗？人类对于自身认为困难的东西，潜意识中会累积大量的焦虑。身体智能为了释放这些焦虑，身体智能会影响大脑智能，然后大脑智能会以“纵欲”的方式去释放焦虑，过度的纵欲最后会变成上瘾。快食上瘾症，就是人们为了释放心理焦虑，然后以满足食欲的方式去执行。基本上素食很难上瘾，唯有肉食，尤其是油腻腻的肉食食物 ...

不知不觉又离题了！笔者所要告知读者的是 ... 在 Verilog 的世界里，过度依赖编译器就是一种“上瘾症”的现象。打从那一天我们决定和 Verilog HDL 语言打交道的时候，我们就必须遵守 Verilog 世界的“自然规则”，所以说和二进制和谐相处是注定的事情。如果读者使用编译器，读者就看不到任何和二进制有关的结果。

为了融合在 Verilog 的自然里，我们要常常与二进制相处，相处久了自然而然会发现二进制的可爱之处。当然笔者非要读者在白纸和铅笔上完成转换工作，笔者建议可以使用一些计算工具，来简化转换工作的困难度。就当给自己一个训练，要征服浮点数之前就要先征服二进制。当二进制的存在不再碍眼的时候，再使用一些快捷的工具也不为迟。

```
[ 0][0111111111][0000000000000000000000000000000000000000000000000000]
[63][--62:52---][-------------------------51:0-----------------------]
[符][---价码---][-------------------------尾数-----------------------]
```

双精度的默认空壳

至于双精度和单精度之间是大同小异，双精度比起单精度“精度的表示更为准确”和“数目表示的范围更广阔”以外，基本上任何转换步骤都一样。好了，笔者已经把这小节的重点都已经讨论一番，在此双精度不是这小节的主角，有关单精度读者只要先有个概念就行了，往后会继续深入的。

美食一不到三过多就会厌腻，学习也是一样，重点都是点到即止，不然会学到头昏脑胀。当时机到了，该遇的问题自会相遇，该懂的内容自然会懂。

4.3 浮点数加减法

单精度浮点数的加减法比起当单精度浮点数的乘除法来得更复杂一点。其中涉及到诸多的补码和正码之间的转换工作。我们先简单的回忆一下什么是正码？什么是补码？假设我们有 4 位宽的整数，为了把它区分正与负，我们会在最高位用“符号（Sign）”来表示，如下所示：

```
0111₂ = 7₁₀     // 正码
1000₂           // 取反，反码
1001₂ = -7₁₀    // 加 1，补码
```

读者是否还记得，在我们设计各种乘法器的时候，我们知道有的乘法器可融入补码和正码一起运算，而有些乘法器仅能正码和正码之间一起运算而已。对于单精度表示的浮点数来说，在表示的时候“只显示正码”；加减运算的时候则“正码和补码一起参与”。这是实现补码加减运算的第一问题点。

至于第二个问题点是关于单精度浮点数的运算过程和数学上的浮点数运算过程是非常不同，这句话又何解呢？别急，往后读者就会知晓的。最后一个问题点就是精度的控制，如常常受忽略的 10 进度和 2 进制之间“精度认识”的不同就是最好的一个例子。

```
A = 32'b0_01110100_11101101110000011100101 0
B = 32'b0_01110100_11101101110000011100100 1
A == B == 0.0009417
```

在 2 进制的认识上 A 和 B 是不一样的结果，反之在 10 进制的认识上 A 和 B 的结果均为 0.0009417。为什么会发生这样的情形呢？原因很简单，因为单精度的尾数受到位宽限制的关系。除此之外，在浮点数加减乘除的运算过程中“尾数常常会被移动”，很多软件运算会把“失去精度”作为提升运算速度/节省空间的条件。

基本上要注意的内容差不多就是这些问题而已，Verilog HDL 语言在“位操作”上拥有很强的优势，因此在设计浮点数运算的时候，比起其他高级语言“伸缩性”会更好。

Q1. A = 6.751, B = 4.832, A + B = ?

```
6.751 转换为单精度表示的浮点数：

6.751₁₀ = 0b110.110000000100000110001001001101 11 （尾数太长，只摘取一部分）
浮点数源 = 1.101100000001000001100010010011 01111E+2

默认空壳的符号位[31]   = 0
默认空壳的价码[30:23]  = 0b0111111 + 2₁₀ = 0b0b10000001
默认空壳的尾数[22:0]   =.[101100000001000001100 01]00100110111
```

```
单精度表示的浮点数是 = 32'b0_10000001_10110000000100000110001
4.832 转换为单精度表示的浮点数：

4.832₁₀ = 0b100.01100001110010101100000010000011 （尾数太长，只摘取一部分）
浮点数源 = 0b1.00011000011100101011000000100000 11E+2

默认空壳的符号位[31]  = 0
默认空壳的价码[30:23]  = 0b0111111 + 2₁₀ = 0b0b10000001
默认空壳的尾数[22:0]   = .[00011000011100101011000]00010000011

单精度表示的浮点数是 = 32'b0_10000001_00011000011100101011000
```

在这里，我们已经得到 A 和 B 的单精度格式的浮点数，接下来我们要实现加法了。

步骤 1：操作数预处理

“操作数预处理”是任何一个单精度浮点数实现运算之前的热身运动。根据运算目的（加减还是乘除）操作数预处理的动作也不同。事实上，不同的参考书也有不同的内容，但是众多的参考书都是以“优化”为目的，作为理解是非常不适合的。所以读者还是按笔者的方法来学习吧。

```
reg [56:0]rA,rB;

rA <= { A[31],A[30:23],2'b01,A[22:0],23'd0 };
rB <= { B[31],B[30:23],2'b01,B[22:0],23'd0 };
```

代码 4.3.1 单精度浮点数，加减运算前的预操作。

首先声明位宽为 57 的 rA 和 rB 寄存器，rA 寄存单精度浮点数 A 的预操作结果；rB 则寄存单精度浮点数 B 的预操作结果。

rA 和 rB 各个位定义如表 4.3.1 所示：

符号位[56]	价码[55:48]	隐藏位[47:46]	尾数[45:23]	尾数补偿[22:0]
A[31]	A[30:23]	2'b01	A[22:0]	23'd0

表 4.3.1 寄存器 rA，rB 的相关位定义。

在这里，可能有同学要问：“隐藏位为什么要用 2 位来表示呢？”答案很简单，是用来寄存隐藏位和隐藏位的进位结果，此外也为了简化结果调整的工作（步骤 6）。“那么 23 位宽的尾数补偿是用来干嘛？”在前面笔者已经说过了，在运算的过程中尾数常常会发生移动，尾数补偿就是予防尾数右移的时候浮点数失去原来的精度。

结果 rA 和 rB 的值为：

```
rA = 57'b0_10000001_01_101100000001000000110001_000000000000000000000000
rB = 57'b0_10000001_01_000110000111001010110000_000000000000000000000000
```

步骤 2：价码对齐

浮点数在实现加减之前，两个浮点数的价码之间需要对齐，对齐的规则是“向大对齐”，这样说有点模糊，笔者还是举个数学上的例子吧：

```
A = 5.237 x10^3 , B = 3.142 x10^-2, A+B = ?

A.Exp - B.Exp = 3 - (-2) = 5  //A 的价码比 B 的价码大 5 个位数

B = 3.142 x10^-2        // B 向 A 对齐之前，B 的尾数必须右移 5 位。
  = .3142 x10^-1        // 右移 1 位
  = .03142 x10^-0       // 右移 2 位
  = .003142 x10^1       // 右移 3 位
  = .0003142 x10^2      // 右移 4 位
  = .00003142 x10^3     // 右移 5 位

B' = 0.00003142x10^3    // B'位 B 向 A 对齐之后的结果。

A + B' = 5.237x10^3 + 0.00003142x10^3 = 5.23703142x10^3
```

上述的例子是浮点数以 10 进制在数学的运算过程，对于 2 进制的单精度浮点数来说，过程会比较直接一点。接下来，我们看如何用 Verilog HDL 语言来实现。

首先先计算出 A 与 B 的代码差，如代码 4.3.2 所示：

```
reg [9:0]rExp,rExpDiff;

begin
    rExp = A[30:23] - B[30:23];
    if( rExp[8] ) rExpDiff <= ~rExp + 1'b1;
    else rExpDiff <= rExp;
end
```

代码 4.3.2 计算出 A 与 B 的价码差。

寄存器 rExp 和 rExpDiff 均为 10 位，rExp 寄存 A 与 B 的价码差；rExpDiff 则寄存相差位数。寄存器 rExp 之所以要声明为 10 位位宽，是为了判断价码差的符号位。如果 rExp[8]为 1 的话，说明价码差位负值，换句话说 A 的价码小于 B；如果 rExp[8]为 0，说明 A 的价码大于还是等于 B。总结一下：

```
if rExp[8] == 1 then A.Exp < B.Exp;
if rExp[8] == 0 then A.Exp >= B.Exp;
```

稍微把 A 与 B 的价码差计算一下：

```
rExp    = A[30:23] - B[30:23]
        = 8'b10000001 - 8'b10000001
        = 10'b00_00000000

rExp 的结果为 0 表示 A 和 B 的价码相同。
rExpDiff 为 0。
```

最后，我们根据 rExp[8] 的符号位，将 rExp 的结果赋值与 rExpDiff。rExpDiff 是用来寄存价码差的结果，为了实现移位，理应取得正值。

```
begin
  if( rExp[8] == 1 ) begin rA[47:0] <= rA[47:0] >> rExpDiff; rA[55:48] <= rB[55:48]; end
  else begin rB[47:0] <= rB[47:0] >> rExpDiff; rB[55:48] <= rA[55:48]; end
end
```

代码 4.3.3 价码对齐操作。

代码 4.3.3 是价码对齐的动作，其中 if(rExp[8] == 1) 表示 A 的价码小于 B 的价码，所以 A 必须向 B 对齐，rA 的尾数（包括补偿尾数）即 rA[47..0] 向右移 rExpDiff 位。然后 rA 的价码即，rA[55:48]赋予 rB 的价码。反之 B 必须向 A 对齐，rB 包括补偿尾数移动 rExpDiff 位（rB[47:0] >> rExpDiff），然后 rB 的价码赋予 rA 的价码（rB[55:48] << rA[55:48]）。

在 Q1 中，A 和 B 的价码差位 0，rA 和 rB 既没有变化：

```
rA = 57'b0_10000001_01_101100000001000000110001_000000000000000000000000
rB = 57'b0_10000001_01_000110000111001010110 00_000000000000000000000000
```

步骤 3：尾数（包括尾数补偿）运算预处理

什么是运算预处理？未执行加减运算之前，我们还要根据 rA 和 rB 的符号位，把 rA 和 rB 的尾数（包括尾数补偿）转换成补码与否。具体的过程到底是怎么一会事呢？我们来看代码：

```
reg [48:0]TempA,TempB;

begin
      TempA <= rA[56] ? { rA[56], (~rA[47:0] + 1'b1) } : { rA[56], rA[47:0] };
      TempB <= rB[56] ? { rB[56], (~rB[47:0] + 1'b1) } : { rB[56], rB[47:0] };
```

```
end
```

代码 4.3.4 尾数运算前处理。

如代码 4.3.4 所示，TempA 寄存 rA 的尾数（包括隐藏位和尾数补偿），TempB 寄存 rB 的尾数（包括隐藏位和尾数补偿）。TempA 会根据 rA 的符号位，把 rA 的尾数（包括隐藏位和尾数补偿）转换为补码与否。Temp 也有同样的动作。那么 Q1 中的 A 和 B 经过步骤 3 会变成怎么样呢？

```
rA = 57'b0_10000001_01_1011000000010000011000 1_00000000000000000000000
rB = 57'b0_10000001_01_0001100001110010101100 0_00000000000000000000000

rA[56] == 0
rB[56] == 0

TempA =49'b0_01_1011000000010000011000 1_00000000000000000000000
TempB =49'b0_01_0001100001110010101100 0_00000000000000000000000
```

步骤 4：加减法操作

```
reg [48:0]Temp

begin Temp <= TempA + TempB;  end
```

代码 4.3.5 加减法操作

代码 4.3.5 是加减法操作，步骤 3 和步骤 4 的存在是为了提高代码的表达能力和修改能力（还记得 IIC 的实验吗？不同速率之间的步骤互换也不会影响实验结果？)。因此，加法和减法之间的操作只是更换其中的运算符而已。

继续计算 Q1 中 A 和 B 的结果：

```
TempA =49'b0_01_1011000000010000011000 1_00000000000000000000000
TempB =49'b0_01_0001100001110010101100 0_00000000000000000000000

TempA + TempB = ?

s hh m  // S-Sign   H-Hidden Bit   M-Mantissa
0_01_1011000000010000011000 1000000000000000000000000
0_01_0001100001110010101100 0000000000000000000000000
0_10_1100100010000011000100 1000000000000000000000000

Temp = 49'b0_10_1100100010000011000100 1000000000000000000000000
```

读者记住注释中的 s h m 一行字，s 表示符号位；一个 h 表示一个隐藏位，两个 h 则是

两位隐藏位；m 表示尾数（包括尾数补偿）。这 3 个字的作用可大可小，尤其在手动运算的时候，会提高不少表达能力。

最后 Temp 的结果为 49'b0_10_1100100010000011000100100000000000000000000000。

步骤 5：结果预处理

步骤 5 中的“结果预处理”是重要的一环，在前面笔者说过单精度的浮点数是以“正直”表示，在加减运算后如果结果取得赋值的话，我们需要正直化处理。此外还要取得结果的符号位，和更新 rExp 的结果。

```
begin
        isSign <= Temp[48];
        if( Temp[48] == 1'b1) Temp <= ~Temp + 1'b1; // change M be unsigned
        rExp <= {2'b00, rA[55:48]}; // or rB[55:48] , change rExp withbe rA.Exp or rB.Exp
end
```

代码 4.3.6 结果预处理。

代码 4.3.6 很简单而已，isSign 赋予 Temp[48]，Temp[48]则是加减结果的符号位。然后判断 Temp[48]的符号位，如果为 1 的就正直化结果。最后更新 rExp 为 rA 的价码 (rB 的价码也是等价)。我们继续运算 Q1 的问题看看：

```
Temp = 49'b0_10_1100100010000011000100100000000000000000000000

Temp[48] == 0 then
isSign = 0;
Temp 没有变化。
rExp = 10'b00_10000001
```

步骤 6: 结果调整

单精度格式的浮点数，尤其是尾数，在运行过后结果需要经过调整。那么调整是什么？我们又要怎样调整？嗯，这是一个号问题，我们先看一个简单的例子：

```
0b 01.111111E+2  //第一组尾数
0b 10.111111E+2  //第二组尾数
0b 11.111111E+2  //第三组尾数
0b 00.111111E+2  //第四组尾数
0b 00.011111E+2  //第五组尾数
```

假设我们有 4 组尾数，除了第一组尾数以外，其余 4 组尾数都不适合“浮点数源”的格式，我们知道“浮点数源”的格式都是 0b1.XXXX，所谓的调整就是将结果调整到适合

“浮点数源”的格式。

那么其余 3 组尾数的调整过程会是：

```
0b 10.111111E+2 = 0b 1.0111111E+3  // 第二组 尾数右移 1 位，价码加 1
0b 11.111111E+2 = 0b 1.1111111E+3  // 第三组 尾数右移 1 位，价码加 1
0b 00.111111E+2 = 0b 1.11111  E+1  // 第四组 尾数左移 1 位，价码减 1
0b 00.011111E+2 = 0b 1.1111   E+0  // 第五组 尾数左移 2 位，价码减 2
```

比起这个简单的例子，在实际的调整中我们会遇见更多位宽和更大范围的调整。

```
begin
        if( Temp[47:46] == 2'b10 || Temp[47:46] == 2'b11) begin Temp <= Temp >> 1; rExp <= rExp + 1'b1; end
        else if( Temp[47:46] == 2'b00 && Temp[45] ) begin Temp <= Temp << 1; rExp <= rExp - 5'd1; end
        else if( Temp[47:46] == 2'b00 && Temp[44] ) begin Temp <= Temp << 2; rExp <= rExp - 5'd2; end
        else if( Temp[47:46] == 2'b00 && Temp[43] ) begin Temp <= Temp << 3; rExp <= rExp - 5'd3; end
        else if( Temp[47:46] == 2'b00 && Temp[42] ) begin Temp <= Temp << 4; rExp <= rExp - 5'd4; end
        else if( Temp[47:46] == 2'b00 && Temp[41] ) begin Temp <= Temp << 5; rExp <= rExp - 5'd5; end
        else if( Temp[47:46] == 2'b00 && Temp[40] ) begin Temp <= Temp << 6; rExp <= rExp - 5'd6; end
        else if( Temp[47:46] == 2'b00 && Temp[39] ) begin Temp <= Temp << 7; rExp <= rExp - 5'd7; end
        else if( Temp[47:46] == 2'b00 && Temp[38] ) begin Temp <= Temp << 8; rExp <= rExp - 5'd8; end
        else if( Temp[47:46] == 2'b00 && Temp[37] ) begin Temp <= Temp << 9; rExp <= rExp - 5'd9; end
        else if( Temp[47:46] == 2'b00 && Temp[36] ) begin Temp <= Temp << 10; rExp <= rExp - 5'd10; end
        else if( Temp[47:46] == 2'b00 && Temp[35] ) begin Temp <= Temp << 11; rExp <= rExp - 5'd11; end
        else if( Temp[47:46] == 2'b00 && Temp[34] ) begin Temp <= Temp << 12; rExp <= rExp - 5'd12; end
        else if( Temp[47:46] == 2'b00 && Temp[33] ) begin Temp <= Temp << 13; rExp <= rExp - 5'd13; end
        else if( Temp[47:46] == 2'b00 && Temp[32] ) begin Temp <= Temp << 14; rExp <= rExp - 5'd14; end
        else if( Temp[47:46] == 2'b00 && Temp[31] ) begin Temp <= Temp << 15; rExp <= rExp - 5'd15; end
        else if( Temp[47:46] == 2'b00 && Temp[30] ) begin Temp <= Temp << 16; rExp <= rExp - 5'd16; end
        else if( Temp[47:46] == 2'b00 && Temp[29] ) begin Temp <= Temp << 17; rExp <= rExp - 5'd17; end
        else if( Temp[47:46] == 2'b00 && Temp[28] ) begin Temp <= Temp << 18; rExp <= rExp - 5'd18; end
        else if( Temp[47:46] == 2'b00 && Temp[27] ) begin Temp <= Temp << 19; rExp <= rExp - 5'd19; end
        else if( Temp[47:46] == 2'b00 && Temp[26] ) begin Temp <= Temp << 20; rExp <= rExp - 5'd20; end
        else if( Temp[47:46] == 2'b00 && Temp[25] ) begin Temp <= Temp << 21; rExp <= rExp - 5'd21; end
        else if( Temp[47:46] == 2'b00 && Temp[24] ) begin Temp <= Temp << 22; rExp <= rExp - 5'd22; end
        else if( Temp[47:46] == 2'b00 && Temp[23] ) begin Temp <= Temp << 23; rExp <= rExp - 5'd23; end
        // else do nothing if( Temp[47:46] == 2'b01 )
end
```

代码 4.3.7 结果调整，范围 Temp[47] ~ Temp[23]。

我们知道 Temp[48]是符号位，符号位不参与调整。因此 Temp[47:46]~Temp[45:23]之间就成为了调整的平衡点，换个角度说“浮点数源”格式的“假定”小数点就在两者之间，既是“Temp[47:46].Temp[45:23]”。（关于假定小数点的解释在下一个小节）

为了更清楚的表达，我们先假设 4 个情况：

```
   s hh m                    s hh m                    s hh m                    s hh m
49'b0_01_1.....           49'b0_01_0.....           49'b0_00_1.....           49'b0_00_01.....
49'b0_01_1..... +         49'b0_01_0..... +         49'b0_00_1..... +         49'b0_00_01..... +
===========               ===========               ===========               ===========
49'b0_11_0......          49'b0_10_0......          49'b0_01_0......          49'b0_00_10......
```

上面的 4 个情况告诉我们一个事实，当 A 和 B 的尾数作加操作的时候，2 位位宽的隐藏位足够应付隐藏位的进位结果。在实际的操作中结果远远不会那么理想，我们需要执行更大范围的调整工作，如：

```
   s hh m
49'b0_11_xxxxxxxxxxxxxxxxxxxxxxxx x // 隐藏位为 2'b11，尾数需要右移 1 位，价码加 1
49'b0_10_xxxxxxxxxxxxxxxxxxxxxxxx x // 隐藏位为 2'b10，尾数需要右移 1 位，价码加 1

   s hh m
49'b0_00_[1]xxxxxxxxxxxxxxxxxxxxxx x // 隐藏位为 2'b00，[45]位为 1，尾数需要左移 1 位，价码减 1
49'b0_00_0[1]xxxxxxxxxxxxxxxxxxxxx x // 隐藏位为 2'b00，[44]位为 1，尾数需要左移 2 位，价码减 2
49'b0_00_00[1]xxxxxxxxxxxxxxxxxxxx x // 隐藏位为 2'b00，[43]位为 1，尾数需要左移 3 位，价码减 3
......
49'b0_00_00000000000000000000[1]x x // 隐藏位为 2'b00，[24]位为 1，尾数需要左移 22 位，价码减 22
49'b0_00_0000000000000000000[1] x // 隐藏位为 2'b00，[23]位为 1，尾数需要左移 23 位，价码减 23

   s  hh   m
49'b0_0[1]_xxxxxxxxxxxxxxxxxxxxxxx x // 最终的结果
```

除了结果的隐藏位为 2'b01 以外，其外都需要调整：

如果结果的隐藏位为 2'b11 那么尾数需右移 1 位，然后价码加 1；
如果结果的隐藏位为 2'b10 那么尾数需右移 1 位，然后价码加 1；
如果结果的隐藏位为 2'b00，但是第 45 位为 1，那么需尾数左移 1 位，然后价码减 1;
如果结果的隐藏位为 2'b00，但是第 44 位为 1，那么需尾数左移 2 位，然后价码减 2;
如果结果的隐藏位为 2'b00，但是第 43 位为 1，那么需尾数左移 3 位，然后价码减 3;
....
如果结果的隐藏位为 2'b00，但是第 24 位为 1，那么尾数需左移 22 位，然后价码减 22;
如果结果的隐藏位为 2'b00，但是第 23 位为 1，那么尾数需左移 23 位，然后价码减 23。

经过调整以后，最终的结果都会和“浮点数源”的格式 1.xxxxx 一致。读者需要注意的是，尾数发生左移还是右移的时候价码也会随着改变。此外，可能有同学要问：“为什么不调整[23]位之后的尾数呢？”这是一个好问题，一般上只调整范围只要达到尾数的第[23]位已经够用了，一些挑剔的设计可能会把尾数补偿都考虑到调整的范围内。

家下来，我们来看看 Q1 中 A 和 B 的操作结果，经过调整后会变成什么样子？

```
Temp = 49'b0_10_1100100010000011000100100000000000000000000000
rExp = 10'b00_10000001

    s hh m
49'b0_10_1100100010000011000100100000000000000000000000 // 隐藏位 2'b01，尾数右移 1 位，价码加 1

49'b0_10_1100100010000011000100100000000000000000000000 // 之前，rExp = 10'b00_10000001
49'b0_01_0110010001000001100010010000000000000000000000 // 右移 1 位， rExp + 1
49'b0_01_0110010001000001100010010000000000000000000000 // 之后，rExp = 10'b00_10000010
```

步骤 7：输出和格式化

步骤 7 主要的动作就是把调整后的结果，以单精度浮点数格式化，并且判断价码溢出，或者尾数零值等问题。如果没有任何问题就把格式化以后的结果输出，否则就输出“默认空壳”和相关的错误信息。

我们先来讨论什么是价码溢出：

单精度格式的浮点数，价码的位宽为 8，然后价码从 8'b0111111 之间摆动不定。如果价码为 8'b1000000 那么该浮点数源的价码为 E+1；反之如果价码为 8'b01111110，那么该浮点数源的价码为 E-1。换句话说，正直的价码会从 8'b10000000~8b11111111，亦即 E+1 ~ E+128；同样负值的价码从 8'b01111110 ~ 8'b00000000，亦即 E-1 ~ E-127。
（对于一些指令格式来说赋值的价码从 8'b01111110 ~ 8'b00000001 而已，其中 8'b00000000 是保留给零值使用）

价码溢出可分为两种，亦即价码上溢和价码下溢。价码上溢就是正直的价码超过 E+128，价码下溢则是负值的价码超过 E-127。价码溢出的问题，最可能发生是在尾数的调整之后，因为尾数在调整的时候，价码也会随着改变。我们为了判定价码是否溢出？上溢还是下溢？因此，我们为 rExp 声明 10 位宽寄存器。

假设 rExp[9:8]为 2'b00，即表示价码没有溢出；rExp[9:8]为 2'b01，即表示价码为上溢；rExp[9:8]为 2'b11，即表示价码为下溢。喜欢抓头的同学可能又要问：“为什么认定 rExp[9:8]等于 2'b01 就是发生价码上溢，其他亦然？”原因很简单，笔者举个简单的演示(表 4.3.2)：

上溢 = 10'b00_11111111 + 10'b00_00000001 = 10'b01_00000000	上溢 = 10'b00_11111111 + 10'b00_00000011 = 10'b01_00000010
无溢出 = 10'b00_00001111 + 10'b00_00100000 = 10'b00_00011111	
下溢 = 10'b00_00000000 - 10'b00_00000001 = 10'b00_00000000 + 10b11_111111111 补	下溢 = 10'b00_00000000 - 10'b00_00000111 = 10'b00_00000000 + 10b11_11111001 补

= 10'b11_111111111	= 10'b11_11111001

表 4.3.2 价码溢出的表示

如果价码正的发生溢出，我们该怎么办才好呢？别慌，我们只要反馈价码上溢，或者下溢的错误信息，然后把输出指定为默认空壳就可以了。

那么，什么是尾数零值呢？

尾数零值仅是一个假想的错误状态，一般上单精度的浮点数无法表示 0 值，0 值在浮点数中表示无穷大。尾数零值还有另一个可能是：当浮点数执行运算过后可能会发生尾数全零的状态，举个例子 1.5E+2 减 1.5E+2。

除此之外，当运算结果使得精度非常非常小的时候，甚至超过单精度可以曾受的范围之内，我们也可以看成是尾数零值。如果发生了尾数零值，我们该怎么办呢？传统中，处理器的指令系统会反馈 zero 或者 infinity 的错误指令。所以笔者有样学样，果真发生尾数零值，反馈尾数零值的错误信息，然后输出默认空壳。

说一点体外话！在笔者的理解中，0 值等价与“精度超过格式可承受的范围”。以此类推，在无限的左移调整中，价码迟早会发生下溢 ... 举个具体的例子：

```
0 = 0.001                                          //单精度有余承受
0 = 0.00000000000000001                            //单精度可以承受
0 = 0.00000000000000000000000001                   //只要左移尾数，单精度勉强承受
0 = 0.00000000000000000000000000 xxxxx 1           //已经超过单精度可以承受的范围
```

精度恢复，四舍五入？

如果调整以后的结果，价码没有溢出或者尾数又不是零值。换言之，调整结果是单精度格式可以承受的浮点数结果。在输出之前，我们还要执行四舍五入的操作。对于拥有 10 个手指头的人类来说，四舍五入的概念很简单，换做 2 进制的话又会怎么样呢？

假设小数只要求 3 位数，那么小数第 3 位之后的数目就要执行四舍五入，举个例子：

```
22/7     = 3.14285714285714285714
         = 3.143
```

对于 2 进制来说四舍五入等价于“0 舍 1 入”，举个例子(同要小数之后的 3 个位数而已)：

```
1.1110111 = 1.111
1.0001111 = 1.001
```

可能有些同学很好奇，为什么尾数输出之前要执行四舍五入呢？原因很简单，就是恢复精度嘛（也可是说是保护精度流失）。如果要执行四舍五入，那么这个小数点以后的“某个定位”又在哪里？单精度浮点数的尾数是 23 位，对应 Temp[45 ... 23]，“某个定位”

就是 Temp[22]。用简单的话来说，如果 Temp[22]的值是 1 的话，那么 Temp[45 ... 23]就加 1，反之不然。

```
begin
    if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end          // E Overflow
    else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end     // E Underflow
    else if( Temp[46:23] == 24'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // M Zero
    else if( Temp[22] == 1'b1 ) rResult <= { isSign, rExp[7:0], Temp[45:23] + 1'b1 };    // okay with normalised
    else rResult <= { isSign, rExp[7:0], Temp[45:23] }; // okay without normalise
end
```

代码 4.3.8 结果输出之前的处理。

代码 4.3.8 中有 5 个分支，前两个分支是判断价码是否溢出；第 3 个分支是判断尾数是否零值。如果任何一个分支的条件成立，就拉高相关的错误信息，然后输出默认空壳。第 4 个分支是用来检测 Temp[22]的值是否为一？如果是的话，就往 Temp[45:23]中加 1。然后按单精度的格式输出结果。最后一个分支和第 4 个分支大同小异，只是没有执行四舍五入的动作而已。

好了，我们把 Q1 的题目运算一下：

```
Temp = 49'b0_01_01100100010000011000100[1]00000000000000000000
rExp = 10'b00_10000010
isSign = 0

Temp[22] is 1 then 23'b01100100010000011000100 + 1 ,equal 23'b01100100010000011000101
输出结果  = 32'b0_10000010_01100100010000011000101
```

我们稍微验证一下：

```
6.751 + 4.382 = 11.133

11.133₁₀          = 0b1011.00100010000011000100100110110100
浮点数源          = 0b1.01100100010000011000100100110110100 E+3

默认空壳[31]          = 0
默认空壳[30:23]       = 0b01111111 + 3 = 0b10000010
默认空壳[22:0]        = .[01100100010000011000100]100110111

Real Result           = 32'b0_10000010_01100100010000011000100
```

看！结果是不是已经接近了！？很好 ... 在前面，笔者已经说过二进制浮点数和十进制浮点数之间会存在不同的“精度认识”。在二进度的认识上，两个结果不一样，但是在 10 进制的认识上，运算结果是正确的。举个例子：

```
32'b0_10000010_01100100010000011000101  Test Result 11.1330003
32'b0_10000010_01100100010000011000100  Real Result 11.1329994
                                        True Result 11.133
```

Test Result 是两个操作数 6.741 和 4.382 执行一系列运算以后的结果；
Real Result 是预期结果 11.133 直接转换成单精度的结果；
True Result 是 10 进制中预期的结果。

在此 Test Result 和 Real Result 之间有所相差，这是著名的精度问题。但是在 10 进制的认识上，它们都是正确的。

==========

最后，笔者稍微总结一下一系列的运算步骤：

（一）操作数预处理：从单精度格式中恢复成浮点数源。
（二）价码对齐：价码向大对齐，取得价码差，然后执行对齐操作。
（三）尾数运算预处理：根据操作数的符号位，处理尾数。
（四）加减操作：尾数执行加减操作。
（五）结果预处理：取得运算结果的符号位，更新 rExp 和正值化运算结果
（六）结果调整：调整运算结果，适合于浮点数源的格式。
（七）输出和格式化：检测价码溢出，尾数零值等错误，此外执行四舍五入。

基本上，单精度格式的浮点数要完成加减运算就是以上 7 个步骤而已。好了，这一小节不多就到这里了。可能读者还觉得例子不够，想吃更多，读者姑且先让大脑休息一会儿。下一个小节，笔者会用更多的实例来消除读者的迷惑。

4.4 实定和假定小数点的困惑

首先笔者插入一个不相干的小节，根据笔者学习的经验，一开始接触浮点数的时候，尤其是二进制的尾数移动那个的环节 … 刚开始的时候感觉有点“怪怪”的。对于拥有十只脚趾头的人类来说，从小到大的数学教育都是基于 10 进制，即时长大了从事软件相关的行业也很少直接与底层的二进制打交道。

我们先来看看一个例子（例子 4.4.1 所示）：

```
0b 1.1001 E+2
0b 11.001 E+1
0b 110.01 E+0
```

例子 1.4.1 例子一

当我们理解上述的例子，大脑很习惯的看着小数点移动，作为书写的形式，我们可以在白纸或者文本编辑上表示小数点的存在，当我们移动小数点的时候，有“实际”的小数点在发生移动，在此简称“实定移动”。

相反的，我们不能使用 Verilog HDL 语言来表示小数点的存在，我们只能“假定”从某个范围，到某个范围之间的中心点就是小数点的存在，如下代码所示（例子 4.4.2 所示），笔者以 4.3 小节中出现的 Temp[48:0]为例；

符号位	隐藏位	小数点	尾数	尾数补偿
Temp[48]	Temp[47:46]	.	Temp[45:23]	Temp[22:0]

例子 4.4.2 例子二

在代码 4.4.2 中，Temp 寄存器是用来寄存运算结果，它的位宽为 49 位。然而 Temp 本身没有牺牲其中的一位表示小数点的存在。在此“小数点”在二进制中是一个“假定”存在，它的存在是介于 Temp[47:46] 和 Temp[45:23]之间。如果小数点不存在，自然不能实现所谓的“小数点移动”，我们最多仅能左移或者右移 Temp 的二进制值，在此简称二进制的“假定位移”。

那么，这种“怪怪”的感觉到底是怎么一回事呢？我们再来看一个例子（例子 4.4.3）：

```
“实定位移”和“假定位移”发生右移的时候：

0b 1.1001 E+2 before Exp = 2
0b 11.001 E+1 小数点右移 1 位
0b 11.001 E+1 after E = 1

0b 0_11_10000000000000000000000() before Exp = 0
0b 0_01_11000000000000000000000(0) 二进制值右移 1 位，Exp = Exp + 1
0b 0_01_11000000000000000000000(0) after Exp = 1
```

```
“实定位移”和“假定位移”发生左移的时候：

0b 1.1001  E+2 before Exp = 2
0b 0.11001 E+3 小数点左移 1 位
0b 0.11001 E+3 after Exp = 3

0b 0_00_11100000000000000000000(1)  before  Exp = 1
0b 0_01_11000000000000000000001() 二进制值左移 1 位，Exp = Exp - 1
0b 0_01_11000000000000000000001() after Exp = 0
```

例子 4.4.3 “实定位移”和“假定位移”的区别。

例子 4.4.3 告诉了我们一个事实，“实定位移”和“假定位移”的小数点在发生移动的时候会产生“相反的结果”，这话又何解？如例子 4.4.3 所示，在右移的过程中“实定位移”会使价码递减，反之“假定位移”会使价码递增；在发生左移的时候“实定位移”会使价码递增，“假定位移”则会使价码递减。

哇！这真是一个多么可怕的发现，老人说：“疏忽小细节会毁了一切”，原来这种“怪怪”的感觉就是这么一回事。侥幸笔者福气十足，读者也是 ... 不然的话，心里深处会一直受这种“怪怪”的感觉折磨，有如一只看不见的针，不停的忘心里头扎，直到把我们逼疯。

最后，随便讽刺一下那些所谓的“权威”的参考书，如果小细节都不见得，结果会如老人所说：“疏忽小细节会毁了一切”，再权威也会胯下。

4.5 浮点数加减法实例

在还没有进入实验之前，为了尽可以消除读者对浮点数的迷惑，接下来笔者会举例大量的例子。在此之前，我们稍微复习一下单精度浮点数加减预算的几个步骤：

（一）操作数预处理就是从单精度的格式中恢复浮点数；
（二）价码对齐是浮点数在执行加减运算之前的重要步骤；
（三）尾数运算预处理是根据操作数的符号位，是否对尾数转换为补码形式；
（四）加减处理是尾数的加减运算；
（五）结果预处理，取得结果的符号位，和正值化运算结果；
（六）结果调整就是将尾数调整到适合“浮点数源”的格式。
（七）输出和格式化中除了反馈相关的错误信息以外，还有四舍五入的操作。

Q2：A = 3.65 , B = -7.4 , A + B = ?

```
A = 32'b0_10000000_11010011001100110011010
B = 32'b1_10000001_11011001100110011001101

价码对齐：
Exp - B.Exp = 0b10000000 - 0b10000001
            = 128 - 129
            = -1  (负值表示 B 的价码比 A 的价码大，所以 A 向 B 看齐)

$ A.Exp = B.Exp and A.M >> 1

h m
1 11010011001100110011010    before
0 11101001100110011001101(0)  >>1 （括号为尾数补偿）
0 11101001100110011001101(0)  after , A = A'

尾数运算预处理：
$ B.Sign is 1 , while ~B.m + 1（B 的符号位为 1 必须取得补码形式）

s hh m
0 01 11011001100110011001101 before
1 10 00100110011001100110010 ~
1 10 00100110011001100110011 +1
1 10 00100110011001100110011 after, B = B'

加减操作：
e        s hh m
10000001 0 00 11101001100110011001101(0)  A'
10000001 1 10 00100110011001100110011     B' +
```

```
10000001 1 11 00010000000000000000000(0)   sum

结果预处理：
$ isSign = sum.s = 1
$ sum.s is 1，while ~sum.m + 1 （运算结果的符号位为 1，需要正值化）

s hh m
1 11 00010000000000000000000(0) before
0 00 11101111111111111111111(1) ~
0 00 11110000000000000000000(0) +1
0 00 11110000000000000000000(0) sum'

结果调整：
hh is 2'b00, while sum'.m << 1 and sum'.exp - 1

$ sum'.exp = 8'b10000001 - 1 = 8'b10000000

s hh m
0 00 11110000000000000000000(0) before
0 01 1110000000000000000000()  <<1
0 01 111000000000000000000    after, sum"

输出和格式化：
32'b1_10000000 11100000000000000000000 Test Tesult -3.75
32'b1_10000000 11100000000000000000000 Real Result -3.75
                                       True Result -3.75
```

上面是 Q2 简化运算（手动）的过程，读者是不是非常不习惯呢？别慌，笔者会一步一步慢慢解释的。首先，笔者直接用工具取得操作数 A 和 B 单精度表示的浮点数，笔者直接省略了转换过程。对于读者来说，只要明白 4.2~4.3 小节，那么用不着重复了（页数有限的关系）。

接下来的动作是价码对齐，如果取得的代码差是负直的话，即表示 A 小于 B。然后 A 的尾数右移一位，A 的价码赋予 B 的价码，A 重新命名为 A'。在执行加减操作之前，操作数会根据自身的符号位来决定是否把尾数转换成补码形式？很不幸，恰好操作数 B 的符号位为 1，因此 B 的尾数必须转换位补码形式。B 重新命名为 B'。

此时运行 A' + B' 的操作，得到运行结果后，将 sum 的符号位赋予 isSign（Temp[48]），然后检测 isSign 是否为 1，如果是就正值化 sum 的尾数。sum 重新命名为 sum'。随后我们必须把结果调整致适合“浮点数源”的格式，我们判断 hh 的值（隐藏位）(Temp[47:46])，取得 2'b00，此时小数点后的第一位为 1（Temp[47:46] == 2'b00 && Temp[45]），结果必须左移 1 位，和递减 sum'的价码。sum' 重新命名为 sum"。最后把结果输出。

说实在，简化运算模式习惯以后，在表达浮点数运算全过程的时候是非常方便的，如果读者还不能习惯的话，不要勉强自己 … 慢慢来。

Q3: A = -3.142 , B = 2.002 , A+B = ？

```
A = 32'b1_10000000_10010010001011010000111
B = 32'b0_10000000_00000000010000011000101

价码对齐：
A.exp - B.exp = 0b10000000 - 0b10000000 = 0

尾数运算预处理：
$ A.sign is 1 , while ~A.m + 1'b1

s hh m
0 01 10010010001011010000111 before
1 10 01101101110100101111000 ~
1 10 01101101110100101111001 +1
1 10 01101101110100101111001 after, A = A'

加减操作：
e        s hh m
10000000 1 10 01101101110100101111001 A'
10000000 0 01 00000000010000011000101 B  +
10000000 1 11 01101110000101000111110 sum

结果预处理：
$ isSign = sum.s = 1
$ sum.s is 1, while ~sum.m + 1

s hh
1 11 01101110000101000111110 before
0 00 10010001111010111000001 ~
0 00 10010001111010111000010 +1
0 00 10010001111010111000010 after, sum = sum'

结果调整：
$ sum'.hh is 2'b00, while sum'm << 1 and sum'.exp - 1

sum".exp = 8'b10000000 - 1 = 8'b01111111

hh m
00 10010001111010111000010 before
01 00100011110101110000100 <<1
```

```
01 0010001111010111000010 after, sum' = sum"

输出和格式化：
32'b1_01111111_0010001111010111000010 Test Result -1.139998
32'b1_01111111_0010001111010111000010 Real Result -1.1399999
                                      True Result -1.14
```

Q4：A = -3.142，B = -11.519，A + B = ?

```
A = 32'b1_10000000_10010010001011010000111
B = 32'b1_10000010_01110000100110111010011

价码对齐：
A.exp - B.exp = 0b1000000 - 10000010 = -2

$ -2 while, A.m >> 2 and A.exp = B.exp

A.exp = B.exp = 0b10000010

hh m
01 10010010001011010000111(00) before
00 01100100100010110100001(11) >> 2
00 01100100100010110100001(11) after, A = A'

运算预处理：
$ A'.sign and B.sign is 1, while ~A.m + 1, ~B.m + 1

s hh
0 00 01100100100010110100001(11) before
1 11 10011011011101001011110(00) ~
1 11 10011011011101001011111(00) +1
1 11 10011011011101001011111(00) after, A' = A"

s hh
0 01 01110000100110111010011 before
1 10 10001111011001000101100 ~
1 10 10001111011001000101101 +1
1 10 10001111011001000101101 +1 after, B = B'

加减运算:
e         s hh m
10000010 1 11 10011011011101001011111(00) A"
10000010 1 10 10001111011001000101101(00) B' +
10000010 1 10 00101010110110010001100(00) sum
```

```
结果预处理：
$ isSign = sum.s = 1
$ sum.s is 1, while ~sum.m + 1'b1

s hh m
1 10 00101010110110010001100(00) before
0 01 11010101001001101110011(11) ~
0 01 11010101001001101110100(11) +1
0 01 11010101001001101110101(00) after, sum = sum'

结果调整：
$ sum'.hh is 2'b01, while do nothing ...

输出和格式化：
32'b1_10000010_11010101001001101110101 Test Result -14.6610002
32'b1_10000010_11010101001001101110101 Real Result -14.6610002
                                       True Result -14.661
```

Q4 的重点就在于 A 和 B 操作数均为负值（符号位），在运算预处理时双方的尾数都需要取得补码的形式。在加减预算过后，尤其是结果预处理的阶段，由于运算结果是负值（符号位），所以结果的尾数必须正值化（因为单精度浮点数是以正直表示）。

Q5：A = 2112.2012，B = -3.1429，A + B = ？

```
A = 32'b0_10001010_00001000000001100111000
B = 32'b1_10000000_10010010010010101000110

价码对齐：
A.exp - B.exp = 0b10001010 - 0b10000000 = 10₁₀
$ 10 mean A.exp > B.exp, while B.m >> 10 and B.exp = A.exp

B.exp = A.exp = 0b10001010

hh M
01 10010010010010101000110()           before
00 0000000001100100100100 1(0101000110) >> 10
00 0000000001100100100100 1(0101000110) after, B = B'

运算预处理：
$ B'.sign is 1, while ~B'.m + 1

s hh M
0 00 0000000001100100100100 1(0101000110) before
```

```
1 11 1111111100110110110110(1010111001) ~
1 11 1111111100110110110110(1010111010) +1
1 11 1111111100110110110110(1010111010) after, B' = B"

加减操作：
e        s hh m
10001010 0 01 0000100000001100111000(0000000000) A'
10001010 1 11 1111111100110110110110(1010111010) B" +
10001010 0 01 0000011110100001110111 0(1010111010) sum

结果预处理：
$ isSign = sum.s = 0

结果调整：
$ sum.hh is 2'b01, while do nothing ...

输出和格式化：
Temp[22] is 1, Temp[45:23] + 1

m
0000011110100001110111 0(1010111010) before
00000111101000011101111             after

32'b0_10001010_00000111101000011101111 Test Result 2109.0583496
32'b0_10001010_00000111101000011101111 Real Result 2109.0583496
                                       True Result 2109.0583
```

Q5 的重点就在于 A 和 B 操作数之间的价码差很大，因此在价码对齐的步骤里，B 尾数的一部分都填入致尾数补偿。然后，在最后的步骤——输出和格式化，由于尾数补偿最高位（Temp[22]）为 1 所以运算结果必须执行四舍五入。

Q6: A = 33.0451 , B = -20.9466 , A + B = ?

```
A = 32'b0_10000100_00001000010111000101111
B = 32'b1_10000011_01001111001001010100011

价码对齐：
A.exp - B.exp = 0b10000100 - 0b10000011 = 1₁₀

$ 1 mean A.exp > B.exp, while B.exp >> 1 and B.exp = A.exp

B.exp = A.exp = 0b10000100

hh m
```

```
01 0100111100100101010001 1() before
10 1010011110010010101000 1(1) >>1
10 1010011110010010101000 1(1) after, B = B'

运算预处理：
$ B'.s is 1, while ~B'.m + 1

s hh M
0 00 1010011110010010101000 1(1) before
1 11 0101100001101101010111 0(0) ~
1 11 0101100001101101010111 0(1) +1
1 11 0101100001101101010111 0(1) after, B' = B"

加减操作：
e        s hh M
10000100 0 01 0000100001011100010111 1(0) A
10000100 1 11 0101100001101101010111 0(1) B" +
10000100 0 00 0110000011001001101110 1(1) sum

结果预处理：
$ isSign = sum.s = 0

结果调整：
$ sum.hh is 2'b00, and Temp[44] is 1, while sum.m << 2 and sum.exp - 2

sum'.exp = 0b10000100 - 2 = 0b10000010

00 0110000011001001101110 1(1) before
01 1000001100100110111011 0(0) <<2
01 1000001100100110111011 0(0) after, sum = sum'

输出和格式化：
32'b0_10000010_10000011001001101110110 Test Result 12.0985012
32'b0_10000010_10000011001001101110101 Real Result 12.0985021
                                       True Result 12.0985
```

Q6 没有什么特别之处，除了在结果调整中，隐藏位为 2'b00，恰好尾数补偿的最高第二位（Temp[44]）为 1，所以尾数必须左移 2 位，然后价码 -2。输出结果中 Test Result 和 Real Result 的精度有点微差，但是结果都是正确的范围之内。

Q7：A = 7.751 , B = 5.382 , A + B = ?

```
A = 32'b0_10000001_11110000000100000110001
B = 32'b0_10000001_01011000011100101011000
```

```
对齐操作：
A.exp - B.exp = 0b10000001 - 0b10000001 = 0

运算预处理：
$ do nothing ...

加减操作：
e        s hh m
10000001 0 01 11110000000100000110001 a
10000001 0 01 0101100001110010101100 0 b +
10000001 0 11 0100100010000011000100 1 sum

结果预处理：
$ isSign = sum.s = 0;

结果调整：
$ sum.hh is 2'b10, while sum.m >> 1 and sum.exp + 1

sum'.exp = 0b10000001 + 1 = 0b10000010

hh
11 01001000100000110001001     before
01 10100100010000011000100(1)  >>1
01 10100100010000011000100(1)  after, sum = sum'

输出和格式化：
$ Temp[22] is 1, while Temp[45:23] + 1

m
10100100010000011000100(1) before
10100100010000011000101(0) after, sum' = sum"

32'b0_10000010_10100100010000011000101 Test Result 13.1329994
32'b0_10000010_10100100010000011000101 Real Result 13.1329994
                                       True Result 13.133
```

Q7 和 Q1 一样，没有什么特别。

Q8：A = 1.9999997，B = -1.9999998, A + B = ?

```
A = 32'b0_01111111_11111111111111111111110
B = 32'b1_01111111_11111111111111111111111
```

```
对齐操作：
A.exp - B.exp = 0b01111111 - 0b01111111 = 0

运算预处理：
$ B.s is 1, while ~B.m + 1

s hh m
0 01 11111111111111111111111 before
1 10 00000000000000000000000 ~
1 10 00000000000000000000001 +1
1 10 00000000000000000000001 after, B = B'

加减操作：
e        s hh m
01111111 0 01 11111111111111111111110 A
01111111 1 10 00000000000000000000001 B' +
01111111 1 11 11111111111111111111111 sum

结果预处理：
$ isSign = sum.s = 1
$ sum.s is 1, while ~sum.m + 1

s hh m
1 11 11111111111111111111111 before
0 00 00000000000000000000000 ~
0 00 00000000000000000000001 +1
0 00 00000000000000000000001 after, sum = sum'

结果调整：
$ sum'.hh is 2'b00 and Temp[23] is 1, while sum'.m << 23 and sum'.exp - 23

sum".exp = 0b01111111 - 23 = 0b01101000

hh
00 00000000000000000000001 before
01 00000000000000000000000 << 23
01 00000000000000000000000 after, sum' = sum"

输出和格式化：
32'b1_01101000_00000000000000000000000 Test Result -0.0000001
32'b1_01101000_00000000000000000000000 Real Result -0.0000001
                                       True Result -0.0000001
```

Q7 是一个极度挑剔的问题，然而这个问题的重点就在于“结果调整”这个步骤里，当结

果预处理以后，我们得到的隐藏位为 2'b00，此外小数点以后的第 23 位，亦即 Temp[23] 的值为 1，因此尾数必须左移 23 位，然后价码减去 23。这个现象就涉及“结果调整”即步骤 6 的设计了。如果隐藏位为 2'b00，作为一般的处理我们只是检查致小数点后的最低位而已，并且不涉及继续往后检测之后的尾数补偿。人算不如天算，我们不知道什么时候加减的结果，不但隐藏位为 2'b00，而且小数点以后的 23 位全 0，然而在尾数补偿里巧好存在着某个值为 1 ... 给个具体一点的例子：

```
hh m                          m backup
00_00000000000000000000000(00000000000000000000001)
```

对于这种问题，我们只有两个解决方法而已。第一方法就是反馈”尾数零值“的错误，还记得笔者讲过什么嘛吗？尾数零值基本上有两个可能，值无限大或者超过格式的承载能力。对于单精度浮点数的格式来说，小数点以后 23 位全 0 也表示了“精度超过承载范围”，反馈尾数零值错误也是情有可原。

第二个方法就是把步骤 6 的“结果调整”范围延伸到尾数补偿的最后，如代码 4.5.1 所示。这是一种吃力不讨好的方法，我们为了那么一个小小的结果，而不惜耗费庞大的逻辑资源。

```
else if( Temp[47:46] == 2'b00 && Temp[23] ) begin Temp <= Temp << 23; rExp <= rExp - 5'd23; end
...
else if( Temp[47:46] == 2'b00 && Temp[1] ) begin Temp <= Temp << 45; rExp <= rExp - 6'd45; end
else if( Temp[47:46] == 2'b00 && Temp[0] ) begin Temp <= Temp << 46; rExp <= rExp - 6'd46; end
```

代码 4.5.1 延长结果调整的范围到尾数补偿。

好了，这一小节也差不多了，估计经过这小节的洗礼以后，读者和二进制的认识是不是更上一层楼呢？

4.6 实验二十：浮点数加法器

正是久违的实验，我们做了那么多准备就是实现单精度浮点数的加法器。读者知道吗？据说单精度浮点数加/减法器，比起单精度浮点数乘/除更难实现？笔者倒不觉得，这话只是懒人的借口而已 ... 因为只要搞定了单精度的原理，所有功夫都简单了。不过说老实话，单精度浮点数乘发起确实有点难度。好了，我们进入正题吧：

图 4.6.1 浮点数价码模块图形。

图 4.6.1 是这个实验要建立的浮点数加法模块。Start_Sig 和 Done_Sig 是控制信号，作为启动和反馈完成，不过不同的只是 Done_Sig 信号有 4 位位宽，它们分别为 { isOver, isUnder, isZero, isDone }。Done_Sig[3:1] 是用来反馈错误信息，价码上溢，价码下溢，尾数零值等；Done_Sig[0] 才是传统上的完成信号。其他的如 A 和 B 是 32 位宽的操作数输入信号，Result 则是 32 位宽的输出结果。其本上就是这些而已 ...

float_add_module.v

```
1.    module float_add_module
2.    (
3.        input CLK, RSTn,
4.        input [31:0]A,B,
5.        output [31:0]Result,
6.
7.        input Start_Sig,
8.        output [3:0]Done_Sig,
9.
10.       /*****************/
11.       output [56:0]SQ_rA,SQ_rB,
12.       output [48:0]SQ_Temp,SQ_TempA,SQ_TempB,
13.       output [9:0]SQ_rExp,
14.       output [7:0]SQ_rExpDiff
15.
16.   );
```

第 1~16 行是该模块的谁输出声明，第 3~8 行的信号基本上和图 4.6.1 大同小异，至于第 11~14 行是仿真用的信号。

```
17.     /***************************************/
18.
19.         reg [3:0]i;
20.         reg [56:0]rA,rB;            // [56]Sign, [55:48]Exponent, [47:46]Hidden Bit, [45:23]Mantissa [22:0]M'Backup
21.         reg [48:0]Temp;             // [48]M'sign, [47:46]Hidden Bit, [45:23]M, [22:0]M'Backup
22.         reg [48:0]TempA,TempB; //[48]M'sign, [47:46]Hidden Bit, [45:23]M, [22:0]M'Backup
23.         reg [31:0]rResult;
24.         reg [9:0]rExp;              //[9:8] Overflow or underflow check, [7:0] usuall exp.
25.         reg [7:0]rExpDiff;       //Different between A.Exp and B.Exp
26.         reg isSign;
27.         reg isOver;         // exp overflow error feedback
28.         reg isUnder;        // exp underflow error feedback
29.         reg isZero;         // m zero error feedback
30.         reg isDone;
```

第 19~30 行是相关的寄存器声明。rA 和 rB 除了寄存操作数 A 和 B 以外，它还用来恢复隐藏位[47:46]，并且还添加了尾数补偿的空间 [22:0]。Temp 寄存器是用来寄存 TempA 和 TempB 的运算结果，话说 TempA 和 TempB 操作数运算前（运算预处理）寄存用的寄存器，[48]为符号位，[47:46]是隐藏位，[45:23]是尾数，[22:0]是尾数补偿。

rResult 是用来驱动 Result 信号；rExp 是用来寄存价码差以外，它还有后期等工作；rExpDiff 是用来寄存价码差的正直，价码对齐的时候被应用；isSign 是用来寄存运算结果（加减操作）的符号位；isOver 是价码上溢的标志，isUnder 是价码下溢的标志，isZero 是尾数零值的表示。

```
31.
32.         always @ ( posedge CLK or negedge RSTn )
33.             if( !RSTn )
34.                 begin
35.                     i <= 4'd0;
36.                     rA <= 57'd0;
37.                     rB <= 57'd0;
38.                     TempA <= 49'd0;
39.                     TempB <= 49'd0;
40.                     Temp <= 49'd0;
41.                     rResult <= 32'd0;
42.                     rExp <= 10'd0;
43.                     rExpDiff <= 8'd0;
44.                     isOver <= 1'b0;
45.                     isUnder <= 1'b0;
46.                     isZero <= 1'b0;
47.                     isDone <= 1'b0;
48.                 end
```

第 35~47 行是相关的复位操作，所有寄存器都清零。

```
49.         else if( Start_Sig )
50.             case( i )
51.
52.                 0: // Initial A,B and other reg.
53.                 begin
54.                     rA <= { A[31], A[30:23], 2'b01, A[22:0], 23'd0 };
55.                     rB <= { B[31], B[30:23], 2'b01, B[22:0], 23'd0 };
56.
57.                     isOver <= 1'b0; isUnder <= 1'b0; isZero <= 1'b0;
58.                     i <= i + 1'b1;
59.                 end
60.
61.                 1: // if rExp[9..8] is 1, mean A.Exp small than B.Exp
62.                    // while rExp[9..8] is 0, mean A.Exp large than B.Exp or same.
63.                 begin
64.                     rExp = A[30:23] - B[30:23];
65.
66.                     if( rExp[8] == 1 ) rExpDiff <= ~rExp[7:0] + 1'b1;
67.                     else rExpDiff <= rExp[7:0];
68.                     i <= i + 1'b1;
69.                 end
```

第 52~59 行是步骤 0，亦即操作数预处理。第 54~55 行是操作 A 和 B 的预处理操作，第 57 行则清零 3 个错误标志寄存器。第 61~69 行是对齐操作中取价码差，第 64 行 rExp 取得 A 价码减去 B 价码的即时结果。第 66~67 行则取得价码差的正直，其中 if(rExp[8] == 1) 则表示，如果价码差是赋值的话 ...

```
70.
71.                 2:   // if A < B; A.M move and A.E = B.E, else opposite act;
72.                 begin
73.                     if( rExp[8] == 1 ) begin rA[47:0] <= rA[47:0] >> rExpDiff; rA[55:48] <= rB[55:48]; end
74.                     else begin rB[47:0] <= rB[47:0] >> rExpDiff; rB[55:48] <= rA[55:48]; end
75.                     i <= i + 1'b1;
76.                 end
77.
78.                 3: // Modify TempA and TempB. with sign
79.                 begin
80.                     TempA <= rA[56] ? { rA[56], (~rA[47:0] + 1'b1) } : { rA[56], rA[47:0] };
81.                     TempB <= rB[56] ? { rB[56], (~rB[47:0] + 1'b1) } : { rB[56], rB[47:0] };
82.                     i <= i + 1'b1;
83.                 end
84.
85.                 4: // Addition
86.                 begin Temp <= TempA + TempB; i <= i + 1'b1; end
```

第 71~76 行价码对齐中“向大看齐”，亦即较小价码的操作数，除了移动尾数以外，还要更新自身的价码。第 73 行表示，如果价码差为负值即是 A.exp < B.exp，A 的尾数位移后则更新自身的价码；第 74 行则相反动作。第 78~83 行是运算预处理操作，该步骤会根据操作数的符号化，是否将尾数转换为补码形式，并且寄存在操作空间 TempA 和 TempB。第 85 行是加减操作。

```
87.
88.                 5: // modify result
89.                 begin
90.                     isSign <= Temp[48];
91.                     if( Temp[48] == 1'b1) Temp <= ~Temp + 1'b1; // change M be postive
92.                     rExp <= {2'b00, rA[55:48]}; // or rB[55:48] , change rExp withbe rA.Exp or rB.Exp
93.                     i <= i + 1'b1;
94.                 end
```

第 87~84 行是结果预处理，isSign 寄存操作结果的符号位（第 90 行）；第 91 行的根据符号位，把结果正直化；第 92 行则是更新 rExp。

```
95.
96.                 6: // Check M'hidden bit and modify to 2'b01
97.                 begin
98.                     if( Temp[47:46] == 2'b10 || Temp[47:46] == 2'b11) begin Temp <= Temp >> 1; rExp <= rExp + 1'b1; end
99.                     else if( Temp[47:46] == 2'b00 && Temp[45] ) begin Temp <= Temp << 1; rExp <= rExp - 5'd1; end
100.                    else if( Temp[47:46] == 2'b00 && Temp[44] ) begin Temp <= Temp << 2; rExp <= rExp - 5'd2; end
101.                    else if( Temp[47:46] == 2'b00 && Temp[43] ) begin Temp <= Temp << 3; rExp <= rExp - 5'd3; end
102.                    else if( Temp[47:46] == 2'b00 && Temp[42] ) begin Temp <= Temp << 4; rExp <= rExp - 5'd4; end
103.                    else if( Temp[47:46] == 2'b00 && Temp[41] ) begin Temp <= Temp << 5; rExp <= rExp - 5'd5; end
104.                    else if( Temp[47:46] == 2'b00 && Temp[40] ) begin Temp <= Temp << 6; rExp <= rExp - 5'd6; end
105.                    else if( Temp[47:46] == 2'b00 && Temp[39] ) begin Temp <= Temp << 7; rExp <= rExp - 5'd7; end
106.                    else if( Temp[47:46] == 2'b00 && Temp[38] ) begin Temp <= Temp << 8; rExp <= rExp - 5'd8; end
107.                    else if( Temp[47:46] == 2'b00 && Temp[37] ) begin Temp <= Temp << 9; rExp <= rExp - 5'd9; end
108.                    else if( Temp[47:46] == 2'b00 && Temp[36] ) begin Temp <= Temp << 10; rExp <= rExp - 5'd10; end
109.                    else if( Temp[47:46] == 2'b00 && Temp[35] ) begin Temp <= Temp << 11; rExp <= rExp - 5'd11; end
110.                    else if( Temp[47:46] == 2'b00 && Temp[34] ) begin Temp <= Temp << 12; rExp <= rExp - 5'd12; end
111.                    else if( Temp[47:46] == 2'b00 && Temp[33] ) begin Temp <= Temp << 13; rExp <= rExp - 5'd13; end
112.                    else if( Temp[47:46] == 2'b00 && Temp[32] ) begin Temp <= Temp << 14; rExp <= rExp - 5'd14; end
113.                    else if( Temp[47:46] == 2'b00 && Temp[31] ) begin Temp <= Temp << 15; rExp <= rExp - 5'd15; end
114.                    else if( Temp[47:46] == 2'b00 && Temp[30] ) begin Temp <= Temp << 16; rExp <= rExp - 5'd16; end
115.                    else if( Temp[47:46] == 2'b00 && Temp[29] ) begin Temp <= Temp << 17; rExp <= rExp - 5'd17; end
116.                    else if( Temp[47:46] == 2'b00 && Temp[28] ) begin Temp <= Temp << 18; rExp <= rExp - 5'd18; end
117.                    else if( Temp[47:46] == 2'b00 && Temp[27] ) begin Temp <= Temp << 19; rExp <= rExp - 5'd19; end
118.                    else if( Temp[47:46] == 2'b00 && Temp[26] ) begin Temp <= Temp << 20; rExp <= rExp - 5'd20; end
119.                    else if( Temp[47:46] == 2'b00 && Temp[25] ) begin Temp <= Temp << 21; rExp <= rExp - 5'd21; end
120.                    else if( Temp[47:46] == 2'b00 && Temp[24] ) begin Temp <= Temp << 22; rExp <= rExp - 5'd22; end
121.                    else if( Temp[47:46] == 2'b00 && Temp[23] ) begin Temp <= Temp << 23; rExp <= rExp - 5'd23; end
```

```
122.                    //else do nothing, can extend the hidden bit check area
123.
124.                    i <= i + 1'b1;
125.                end
```

第 96~125 行是结果调整，至于要不要继续扩展隐藏位的搜索就自己看着办（第 122 行）。

```
126.
127.                7: //error check and format result in float format
128.                begin
129.                    if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end          // E Overflow
130.                    else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end     // E Underflow
131.                    else if( Temp[46:23] == 24'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end    // M Zero
132.                    else if( Temp[22] == 1'b1 ) rResult <= { isSign, rExp[7:0], Temp[45:23] + 1'b1 };    // okay with normalised
133.                    else rResult <= { isSign, rExp[7:0], Temp[45:23] };                                  // okay without normalise
134.                    i <= i + 1'b1;
135.                end
136.
137.                8:
138.                begin isDone <= 1'b1; i <= i + 1'b1; end
139.
140.                9:
141.                begin isDone <= 1'b0; i <= 4'd0; end
142.
143.            endcase
```

第 127~135 行是输出和格式化，第 129 行是检测价码上溢；第 130 行是检测价码下溢；第 131 行是检测尾数零值；第 132 行是四舍五入的操作；第 133 行则是不同的输出。步骤 8~9 是用来产生完成信号。

```
144.
145.        /*************************************/
146.
147.        assign Done_Sig = { isOver, isUnder, isZero, isDone };
148.        assign Result = rResult;
149.
150.        /*************************************/
151.
152.        assign SQ_rA = rA;
153.        assign SQ_rB = rB;
154.        assign SQ_Temp = Temp;
155.        assign SQ_TempA = TempA;
156.        assign SQ_TempB = TempB;
157.        assign SQ_rExp = rExp;
```

```
158.        assign SQ_rExpDiff = rExpDiff;
159.
160.        /****************************************/
161.
162.  endmodule
```

第 147~148 行是相关的输出驱动，第 152~158 行则是仿真用的驱动输出。基本上浮点数加法模块的功能就是这么简单而已。

float_add_module.vt

```
1.    `timescale 1 ps/ 1 ps
2.    module float_add_module_simulation();
3.
4.          reg CLK;
5.          reg RSTn;
6.
7.          reg Start_Sig;
8.          reg [31:0] A;
9.          reg [31:0] B;
10.
11.         wire [3:0]   Done_Sig;
12.         wire [31:0]   Result;
13.
14.         /******************/
15.
16.       wire [56:0]SQ_rA,SQ_rB;
17.       wire [48:0]SQ_Temp,SQ_TempA,SQ_TempB;
18.       wire [9:0]SQ_rExp;
19.       wire [7:0]SQ_rExpDiff;
20.
21.         /**********************************/
22.
23.         float_add_module U1
24.         (
25.                 .CLK( CLK ),
26.                 .RSTn( RSTn ),
27.                 .A( A ),
28.                 .B( B ),
29.                 .Result( Result ),
30.                 .Start_Sig( Start_Sig ),
31.                 .Done_Sig( Done_Sig ),
32.                 .SQ_rA(SQ_rA),
33.                 .SQ_rB(SQ_rB),
```

```
34.                 .SQ_Temp( SQ_Temp ),
35.                 .SQ_TempA( SQ_TempA ),
36.                 .SQ_TempB( SQ_TempB ),
37.                 .SQ_rExp( SQ_rExp ),
38.                 .SQ_rExpDiff( SQ_rExpDiff )
39.         );
40.
41.         /***********************************/
42.
43.         initial
44.         begin
45.                 RSTn = 0; #10 RSTn = 1;
46.                 CLK = 0; forever #5 CLK = ~CLK;
47.         end
48.
49.         /***********************************/
50.
51.         reg [3:0]i;
52.
53.         always @ (    posedge CLK or negedge RSTn )
54.                 if( !RSTn )
55.                         begin
56.                           A <= 32'd0;
57.                           B <= 32'd0;
58.                           Start_Sig <= 1'b0;
59.                           i <= 4'd0;
60.                       end
61.             else
62.                       case( i )
63.
64.                                 0: //A=3.65, B= -7.4, A+B = ?
65.                                 if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
66.                         else begin A <= 32'b0_10000000_11010011001100110011010; B <= 32'b1_10000001_11011001100110011001101; Start_Sig <= 1'b1; end
67.
68.                                 1: //Exp undeflow check
69.                                 if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
70.                         else begin A <= 32'b0_00000000_01010000101101000101101; B <= 32'b1_00000000_00010000101100001000111; Start_Sig <= 1'b1; end
71.
72.                                2: //A=1.9999997, B=-1.9999998 , A+B =?
73.                                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
74.                         else begin A <= 32'b0_01111111_11111111111111111111110; B <= 32'b1_01111111_11111111111111111111111; Start_Sig <= 1'b1; end
75.
76.                                 3: //Exp Overflow
77.                                 if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
78.                         else begin A <= 32'b0_11111111_11111111111111111111111; B <= 32'b0_11111111_11111111111111111111111; Start_Sig <= 1'b1; end
```

```
                    4: //A= -12.558, B= -7.309 , A+B =?
                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
            else begin A <= 32'b11000001010010001110110110010001; B <= 32'b11000000111010011110001101010100; Start_Sig <= 1'b1; end

                    5: //A= 111.7762, B= 302.4409 , A+B =?
                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
            else begin A <= 32'b01000010110111111000110101101010; B <= 32'b01000011100101110011100001101111; Start_Sig <= 1'b1; end

                    6: //A= 2112.2012, B= -2002.2012 , A+B =?
                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
            else begin A <= 32'b01000101000001000000001100111000; B <= 32'b11000100111110100100011001110000; Start_Sig <= 1'b1; end

                      7:
                      i <= i;

                endcase

endmodule
```

以上是浮点数加法模块的激励文件，激励文件的书写风格笔者就不重复了，除了关键字 $display 以外，其他都不陌生。$display 是验证语言，主要是用来输出结果，用法和 C 语言的 printf 类似。顺便补上一句，$display 函数在时序的效果上和组合逻辑一样，都是即时事件。步骤 0，2，4，5，6 是用验证测该模块的通常运作，步骤 1 和 3 则是用来测试价码上溢和下溢的运作。

仿真结果：

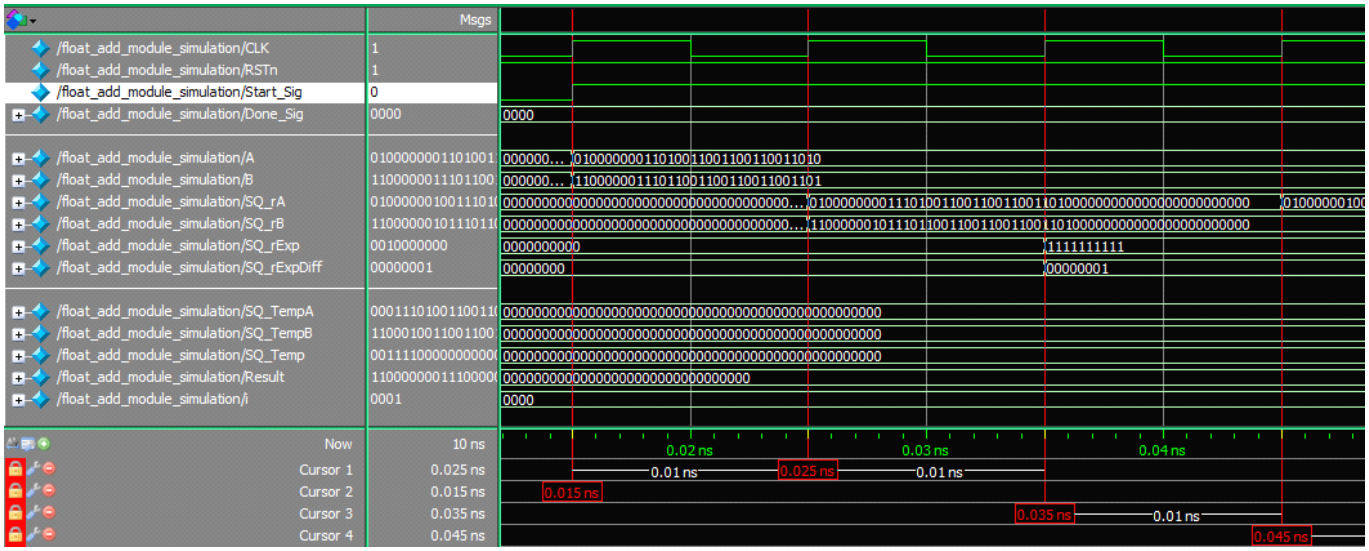


图 4.6.2 仿真流程 1

图 4.6.2 是 3.65 -7.4 的运算过程，C1（以下 Curosr 简称为 C）表示激励文件吧操作数 A 和 B 输

入以后，并且拉高 Start_Sig，在 C1 的未来 32'b0_10000000_11010011001100110011010 和 32'b1_10000001_11011001100110011001101 被输出。在 C2 的时候是操作数预处理，浮点数加法模块读入该过去值，在 C2 的未来输出操作数的预操作结果。在 C3 的时候是取价码差（记得价码差的取得是“即时事件”）结果是 8'b11111111，既是-1 也是 A.exp < B.exp。在 C3 的未来，输出正直化以后的价码差。

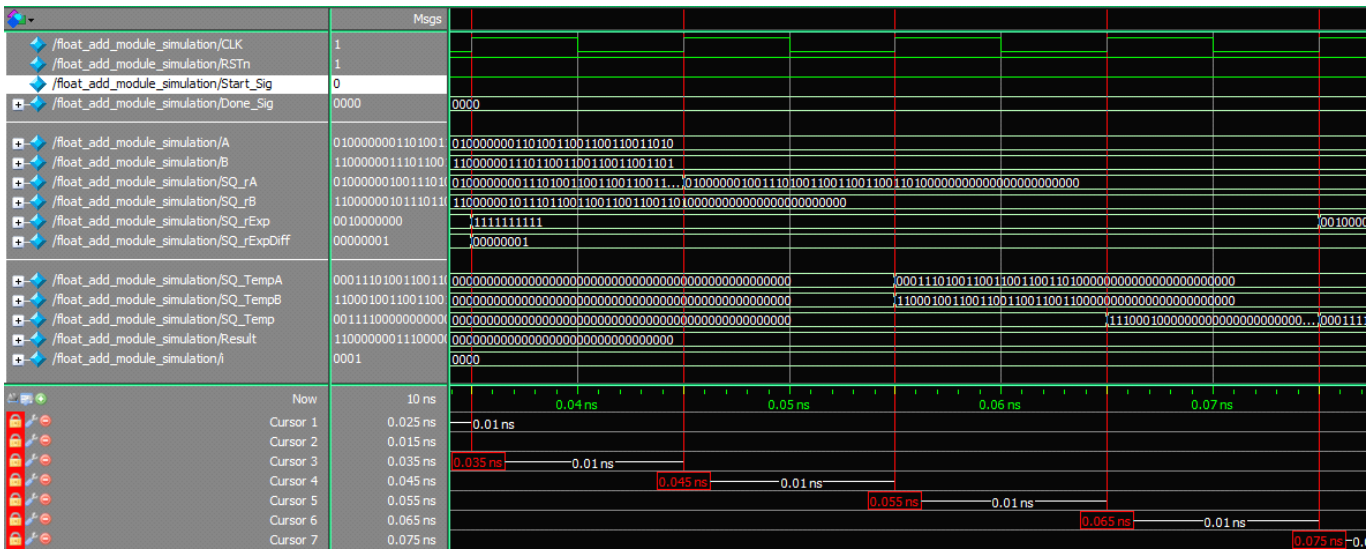


图 4.6.3 仿真流程 2

C4 的时候是“价码向大对齐”的操作，并且更新相关的价码。在 C4 的未来值中，我们知道 rExp[8] 为 1 亦即 A.exp < B.exp，因此决定 A 的尾数右移 rExpDiff 在 C4 的过去值（亦即 1）。所以在 C4 的未来，A 的尾数不但右移 1 位，而且价码的更新结果也和 B 一样。C5 的时候是运算预处理，根据 rA 和 rB 在 C5 中过去的结果，rB 带有符号位，所以 rB 的尾数必须取得补码形式，所以在 C5 的未来，rA 和 rB 的预处理结果会寄存在操作空间 TempA 和 TempB 中。C6 的时候是加减运算，C6 的未来中所输出的值是 TempA 和 TempB 在 C6 过去中相加的结果。

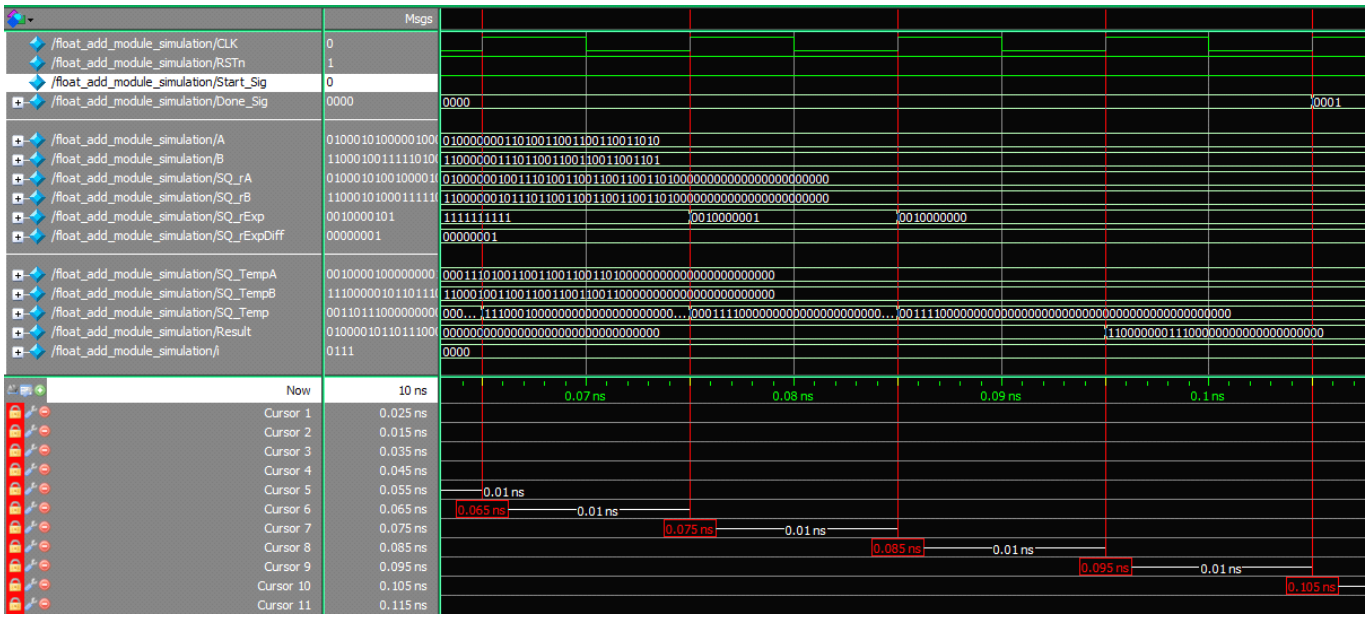


图 4.6.4 仿真流程 3

C7 的时候是结果预处理，isSign 取得在 C7 中 Temp[48]的过去值，亦即是 1。此外，根据过去值 Temp[48]的结果，该值是 1 所以正值化 Temp，在 C7 的未来，被输出的 Temp 为正值化的结果，并且更新 rExp 的结果。C8 的时候是结果调整，Temp[47:46]的过去值为 2'b00（隐藏位），所以在 C8 的未来输出尾数左移一位的结果，其中 rExp 被减去 1。

C9 的时候是输出和个时候，rExp 的过去值为 10'b00_10000000，其中 rExp[9:8] 不是 2'b01 或者 2'b11，所以不发生价码溢出的问题。此外，Temp[22]的过去值也不是 1，所以也不对尾数四舍五入。在 C9 的未来，输出 isSign 为 1，rExp 为 8'b10000000，和尾数为 Temp[45:23]的结果。在这次的加减预算中，没有发生任何错误，所以在 C9 的未来也不输出任何错误反馈。至于 C10~C11 是完成信号的产生经过，怒笔者不解释了。

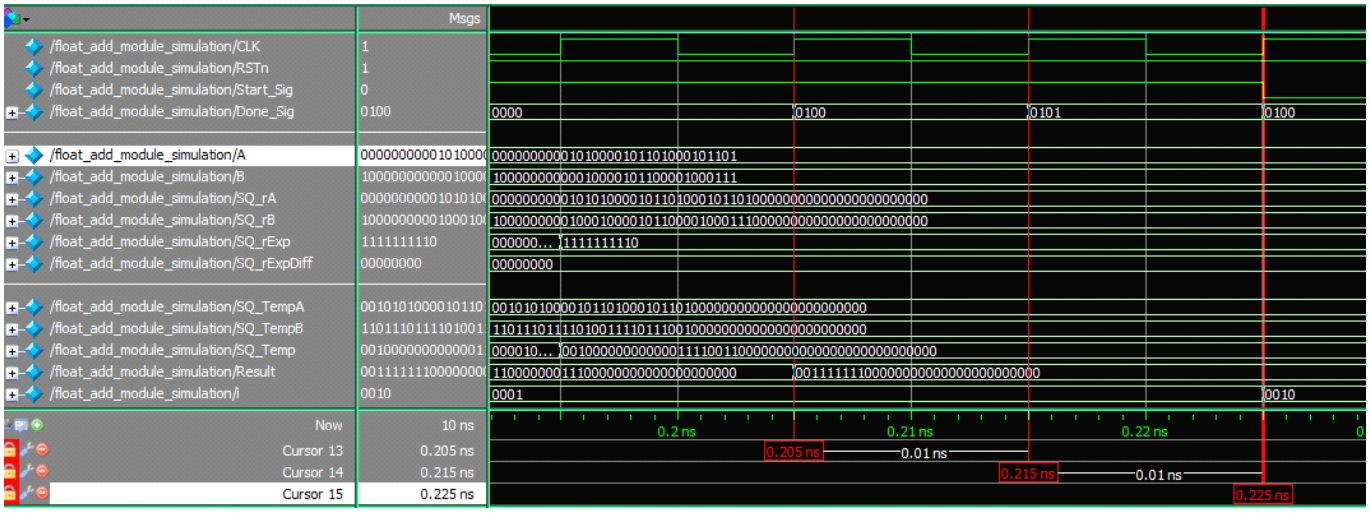


图 4.6.5 价码下溢。

至于图 4.6.5 是激励中故意引起的价码下溢，如 C13 指向的地方，rExp[9:8]的过去值为 2'b11，因此在 C13 的未来判断为价码下溢，拉高 isUnder 的标志（Done_Sig[2]），并且输出默认空壳的值。

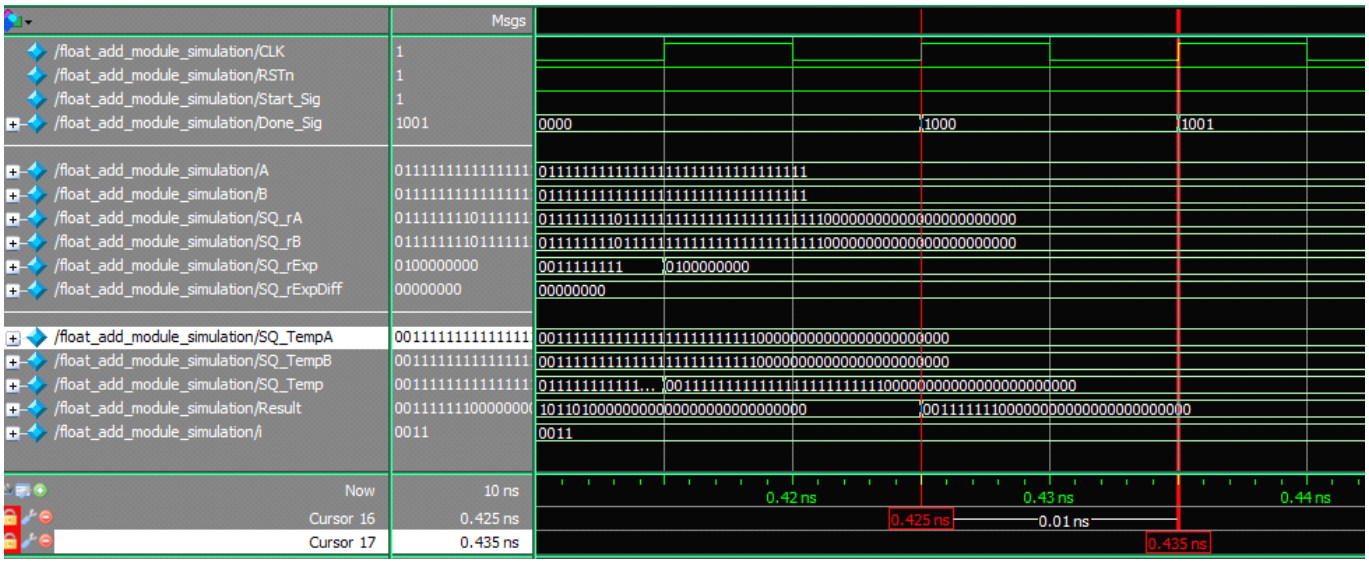


图 4.6.6 价码上溢。

图 4.6.6 同为笔者故意引起的价码上溢，C17 所指向的地方 rExp[9:8]的过去值为 2'b10，因此断定为价码上溢，所以在 C17 的未来，拉高价码上溢的标志（Done_Sig[3]），并且输出默认空壳。关于尾数零值错误信息的产生经过也是大同小异而已。往后还有 -12.558 -7.309，111.7762 + 302.4409 和 2112.2012 -2002.2012 的运算过程也差不多而已，在此笔者就不加说明了，作为补偿笔者就简化运算一下最后一题的过程。

```
Q9: A= 2112.2012 , B = -2002.2012, A + B = ?
```

```
A = 32'b0_10001010_00001000000001100111000
B = 32'b1_10001001_11110100100011001110000

价码对齐：
A.exp - B.exp = 0b10001010 - 0b10001001 = 1₁₀

$ 1 mean A.exp > B.exp while, B.m >> 1 and B.exp = A.exp;

hh m
01 11110100100011001110000()  before
00 11111010010001100111000(0) >> 1
00 11111010010001100111000(0) after, B = B'

运算前预处理：
$ B'.s is 1, while ~B'.m + 1

s hh m
0 00 11111010010001100111000(0)  before
1 11 00000101101110011000111(1) ~
1 11 00000101101110011001000(0) +1
1 11 00000101101110011001000(0) after, B' = B"

加减运算：
e        s hh m
10001010 0 01 00001000000001100111000    A
10001010 1 11 00000101101110011001000(0)  B" +
10001010 0 00 00001101110000000000000(0) sum

结果预处理：
$ isSign = sum.s = 0

结果调整：
$ sum.hh is 2'b00 and Temp[41] is 1, while sum.m << 5 and sum.exp - 5
```

```
sum'.exp = 0b10001010 - 5 = 0b10000101


hh m
00 00001101110000000000000(0) before
01 10111000000000000000000(0) << 5
01 10111000000000000000000(0) after, sum = sum'


输出和格式化：
32'b0_10000101_10111000000000000000000 Test Result 110
32'b0_10000101_10111000000000000000000 Real Result 110
                                       True Result 110
```

基本上，实验 1 中浮点数加法模块的运算过程就是这样而已，没有什么特别需要强调的地方。只要把最基本的概念都搞懂了，设计单精度浮点数的加法器一点也不困难。好了，这个实验二十到此为止把，下一个小节是浮点数减法器的实验，大体上和浮点数加法器大同小异而已，因为笔者在实验二十已经买下一些微调的种子，所以只要在相关的步骤动一下手脚就行了。

4.7 实验二十一：浮点数减法器

图 4.7.1 浮点数减法模块图形。

图 4.7.1 是浮点数减法模块的图形，信号的定义上基本上和浮点数加法模块一模一样，甚至内容也是大同小异，讲多无用我们还是直接来看代码吧。

float_sub_module.v

```
module float_sub_module
(
        input CLK, RSTn,
        input [31:0]A,B,
        output [31:0]Result,

        input Start_Sig,
        output [3:0]Done_Sig,

        /****************/
        output [56:0]SQ_rA,SQ_rB,
        output [48:0]SQ_Temp,SQ_TempA,SQ_TempB,
        output [9:0]SQ_rExp,
        output [7:0]SQ_rExpDiff

);
        /*************************************/

      reg [3:0]i;
      reg [56:0]rA,rB;   // [56]Sign, [55:48]Exponent, [47:46]Hidden Bit, [45:23]Mantissa [22:0]M'resolution
      reg [48:0]Temp;     // [48]M'sign, [47:46]Hidden Bit, [45:23]M, [22:0]M'resolution
      reg [48:0]TempA,TempB; //[48]M'sign, [47:46]Hidden Bit, [45:23]M, [22:0]M'resolution
      reg [31:0]rResult;
      reg [9:0]rExp;        //[9:8] Overflow or underflow check, [7:0] usuall exp.
      reg [7:0]rExpDiff; //Different between A.Exp and B.Exp
      reg isSign;
      reg isOver;
      reg isUnder;
```

```
29.     reg isZero;
30.     reg isDone;
31.
32.     always @ ( posedge CLK or negedge RSTn )
33.         if( !RSTn )
34.             begin
35.                 i <= 4'd0;
36.                 rA <= 57'd0;
37.                 rB <= 57'd0;
38.                 TempA <= 49'd0;
39.                 TempB <= 49'd0;
40.                 Temp <= 49'd0;
41.                 rResult <= 32'd0;
42.                 rExp <= 10'd0;
43.                 rExpDiff <= 8'd0;
44.                 isOver <= 1'b0;
45.                 isUnder <= 1'b0;
46.                 isZero <= 1'b0;
47.                 isDone <= 1'b0;
48.             end
49.         else if( Start_Sig )
50.             case( i )
51.
52.                 0: // Initial and resulted out Signature
53.                 begin
54.                     rA <= { A[31], A[30:23], 2'b01, A[22:0], 23'd0 };
55.                     rB <= { B[31], B[30:23], 2'b01, B[22:0], 23'd0 };
56.
57.                     isOver <= 1'b0; isUnder <= 1'b0; isZero <= 1'b0;
58.                     i <= i + 1'b1;
59.                 end
60.
61.                 1: // if rExp[9..8] is 1, mean A.Exp small than B.Exp
62.                    // while rExp[9..8] is 0, mean A.Exp large than B.Exp or same.
63.                 begin
64.                     rExp = A[30:23] - B[30:23];
65.
66.                     if( rExp[8] == 1 ) rExpDiff <= ~rExp[7:0] + 1'b1;
67.                     else rExpDiff <= rExp[7:0];
68.                     i <= i + 1'b1;
69.                 end
70.
71.                 2:   // if A < B; A.M move and A.E = B.E, else different act;
72.                 begin
73.                     if( rExp[8] == 1 ) begin rA[47:0] <= rA[47:0] >> rExpDiff; rA[55:48] <= rB[55:48]; end
```

```
            else begin rB[47:0] <= rB[47:0] >> rExpDiff; rB[55:48] <= rA[55:48]; end
            i <= i + 1'b1;
        end

        3: // Modified TempA and TempB with own signed
        begin
            TempA <= rA[56] ? { rA[56], (~rA[47:0] + 1'b1) } : { rA[56], rA[47:0] };
            TempB <= rB[56] ? { rB[56], (~rB[47:0] + 1'b1) } : { rB[56], rB[47:0] };
            i <= i + 1'b1;
        end

        4: // Subtraction
        begin Temp <= TempA - TempB; i <= i + 1'b1; end

        5:
        begin
            isSign <= Temp[48];
            if( Temp[48] == 1'b1) Temp <= ~Temp + 1'b1; // change M be unsigned
            rExp <= {2'b00, rA[55:48]}; // or rB[55:48] , change rExp withbe rA.Exp or rB.Exp
            i <= i + 1'b1;
        end

        6: // Check M'hidden bit and modify in 2'b01
        begin
            if( Temp[47:46] == 2'b10 || Temp[47:46] == 2'b11) begin Temp <= Temp >> 1; rExp <= rExp + 1'b1; end
            else if( Temp[47:46] == 2'b00 && Temp[45] ) begin Temp <= Temp << 1; rExp <= rExp - 5'd1; end
            else if( Temp[47:46] == 2'b00 && Temp[44] ) begin Temp <= Temp << 2; rExp <= rExp - 5'd2; end
            else if( Temp[47:46] == 2'b00 && Temp[43] ) begin Temp <= Temp << 3; rExp <= rExp - 5'd3; end
            else if( Temp[47:46] == 2'b00 && Temp[42] ) begin Temp <= Temp << 4; rExp <= rExp - 5'd4; end
            else if( Temp[47:46] == 2'b00 && Temp[41] ) begin Temp <= Temp << 5; rExp <= rExp - 5'd5; end
            else if( Temp[47:46] == 2'b00 && Temp[40] ) begin Temp <= Temp << 6; rExp <= rExp - 5'd6; end
            else if( Temp[47:46] == 2'b00 && Temp[39] ) begin Temp <= Temp << 7; rExp <= rExp - 5'd7; end
            else if( Temp[47:46] == 2'b00 && Temp[38] ) begin Temp <= Temp << 8; rExp <= rExp - 5'd8; end
            else if( Temp[47:46] == 2'b00 && Temp[37] ) begin Temp <= Temp << 9; rExp <= rExp - 5'd9; end
            else if( Temp[47:46] == 2'b00 && Temp[36] ) begin Temp <= Temp << 10; rExp <= rExp - 5'd10; end
            else if( Temp[47:46] == 2'b00 && Temp[35] ) begin Temp <= Temp << 11; rExp <= rExp - 5'd11; end
            else if( Temp[47:46] == 2'b00 && Temp[34] ) begin Temp <= Temp << 12; rExp <= rExp - 5'd12; end
            else if( Temp[47:46] == 2'b00 && Temp[33] ) begin Temp <= Temp << 13; rExp <= rExp - 5'd13; end
            else if( Temp[47:46] == 2'b00 && Temp[32] ) begin Temp <= Temp << 14; rExp <= rExp - 5'd14; end
            else if( Temp[47:46] == 2'b00 && Temp[31] ) begin Temp <= Temp << 15; rExp <= rExp - 5'd15; end
            else if( Temp[47:46] == 2'b00 && Temp[30] ) begin Temp <= Temp << 16; rExp <= rExp - 5'd16; end
            else if( Temp[47:46] == 2'b00 && Temp[29] ) begin Temp <= Temp << 17; rExp <= rExp - 5'd17; end
            else if( Temp[47:46] == 2'b00 && Temp[28] ) begin Temp <= Temp << 18; rExp <= rExp - 5'd18; end
            else if( Temp[47:46] == 2'b00 && Temp[27] ) begin Temp <= Temp << 19; rExp <= rExp - 5'd19; end
            else if( Temp[47:46] == 2'b00 && Temp[26] ) begin Temp <= Temp << 20; rExp <= rExp - 5'd20; end
```

```
119.                    else if( Temp[47:46] == 2'b00 && Temp[25] ) begin Temp <= Temp << 21; rExp <= rExp - 5'd21; end
120.                    else if( Temp[47:46] == 2'b00 && Temp[24] ) begin Temp <= Temp << 22; rExp <= rExp - 5'd22; end
121.                    else if( Temp[47:46] == 2'b00 && Temp[23] ) begin Temp <= Temp << 23; rExp <= rExp - 5'd23; end
122.                    //else do nothing
123.
124.                    i <= i + 1'b1;
125.                end
126.
127.                7: //error check and decide final result
128.                begin
129.                    if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end       // E Overflow
130.                    else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // E Underflow
131.                    else if( Temp[46:23] == 24'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // M Zero
132.                    else if( Temp[22] == 1'b1 ) rResult <= { isSign, rExp[7:0], Temp[45:23] + 1'b1 }; // okay with normalised
133.                    else rResult <= { isSign, rExp[7:0], Temp[45:23] }; // okay without normalise
134.                    i <= i + 1'b1;
135.                end
136.
137.                8:
138.                begin isDone <= 1'b1; i <= i + 1'b1; end
139.
140.                9:
141.                begin isDone <= 1'b0; i <= 4'd0; end
142.
143.            endcase
144.
145.    /*************************************/
146.
147.    assign Done_Sig = { isOver, isUnder, isZero, isDone };
148.    assign Result = rResult;
149.
150.    /****************************************/
151.
152.    assign SQ_rA = rA;
153.    assign SQ_rB = rB;
154.    assign SQ_Temp = Temp;
155.    assign SQ_TempA = TempA;
156.    assign SQ_TempB = TempB;
157.    assign SQ_rExp = rExp;
158.    assign SQ_rExpDiff = rExpDiff;
159.
160.    /*****************************************/
161.
162. endmodule
```

以上是该模块的内容，除了第 85~86 行的加减操作以外，其余的内容都和浮点数加法模块一模一样。至于修改的内容，也是从原本的“+”变成“-”而已。在这里笔者再简单的复习一下 +/- 与整数之间的关系。事实上，编译器在编译 “-”运算符的时候，它会把接续的整数（二进制值）转换为补码形式，也就是我们常作的“取反再加一”。

float_sub_module.vt

```
1.    `timescale 1 ps/ 1 ps
2.    module float_sub_module_simulation();
3.
4.          reg CLK;
5.          reg RSTn;
6.
7.          reg Start_Sig;
8.          reg [31:0] A;
9.          reg [31:0] B;
10.
11.         wire [3:0]   Done_Sig;
12.         wire [31:0]   Result;
13.
14.         /******************/
15.
16.       wire [56:0]SQ_rA,SQ_rB;
17.       wire [48:0]SQ_Temp,SQ_TempA,SQ_TempB;
18.       wire [9:0]SQ_rExp;
19.       wire [7:0]SQ_rExpDiff;
20.
21.         /*********************************/
22.
23.         float_sub_module U1
24.         (
25.                 .CLK( CLK ),
26.                 .RSTn( RSTn ),
27.                 .A( A ),
28.                 .B( B ),
29.                 .Result( Result ),
30.                 .Start_Sig( Start_Sig ),
31.                 .Done_Sig( Done_Sig ),
32.                 .SQ_rA(SQ_rA),
33.                 .SQ_rB(SQ_rB),
34.                 .SQ_Temp( SQ_Temp ),
35.                 .SQ_TempA( SQ_TempA ),
36.                 .SQ_TempB( SQ_TempB ),
37.                 .SQ_rExp( SQ_rExp ),
```

```
38.                 .SQ_rExpDiff( SQ_rExpDiff )
39.         );
40.
41.         /***********************************/
42.
43.         initial
44.         begin
45.                 RSTn = 0; #10 RSTn = 1;
46.                 CLK = 0; forever #5 CLK = ~CLK;
47.         end
48.
49.         /***********************************/
50.
51.         reg [4:0]i;
52.
53.         always @ (    posedge CLK or negedge RSTn )
54.                 if( !RSTn )
55.                         begin
56.                           A <= 32'd0;
57.                           B <= 32'd0;
58.                           Start_Sig <= 1'b0;
59.                           i <= 5'd0;
60.                     end
61.             else
62.                     case( i )
63.
64.                             0: //A=3.65, B=-7.4, A-B = ?
65.                             if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
66.             else begin A <= 32'b0_10000000_11010011001100110011010; B <= 32'b1_10000001_11011001100110011001101; Start_Sig <= 1'b1; end
67.
68.                             1: //Exp undeflow check
69.                             if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
70.             else begin A <= 32'b0_00000000_01010000101101000101101; B <= 32'b1_00000000_00010000101100001000111; Start_Sig <= 1'b1; end
71.
72.                            2: //A=1.9999997, B=-1.9999998 , A-B =?
73.                            if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
74.             else begin A <= 32'b0_01111111_11111111111111111111110; B <= 32'b1_01111111_11111111111111111111111; Start_Sig <= 1'b1; end
75.
76.                             3: //Exp Overflow
77.                             if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
78.            else begin A <= 32'b0_11111111_11111111111111111111111; B <= 32'b0_11111111_11111111111111111111111; Start_Sig <= 1'b1; end
79.
80.                            4: //A=78992.118, B=-90116.402, A-B =?
81.                            if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
82.            else begin A <= 32'b01000111100110100100100000001111; B <= 32'b11000111101100000000001000110011; Start_Sig <= 1'b1; end
```

```
                    5: //A=6992.330, B=1271.004, A-B =?
                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
        else begin A <= 32'b01000101110110101000001010100100; B <= 32'b01000100100111101110000000100001; Start_Sig <= 1'b1; end

                    6: //A=-1.5032, B=-8.3309, A-B =?
                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
        else begin A <= 32'b10111111110000000110100011011100; B <= 32'b11000001000001010100101101011110; Start_Sig <= 1'b1; end

                    7: //A=-88.8888, B=-10.9944, A-B =?
                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
        else begin A <= 32'b11000010101100011100011100010001; B <= 32'b11000001001011111110100100010000; Start_Sig <= 1'b1; end

                    8: //A=0.001584, B=0.063197, A-B =?
                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
        else begin A <= 32'b00111010110011111001111000111000; B <= 32'b00111101100000010110110101101110; Start_Sig <= 1'b1; end

                    9: //A=-613.931, B=27.11402, A-B =?
                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
        else begin A <= 32'b11000100000110010111101110010110; B <= 32'b01000001110110001110100110000011; Start_Sig <= 1'b1; end

                    10: //A=44.4444, B=68.3333, A-B =?
                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
        else begin A <= 32'b01000010001100011100011100010001; B <= 32'b01000010100010001010101010101010; Start_Sig <= 1'b1; end

                    11:
                      i <= i;

                endcase

endmodule
```

步骤 0 与 2 基本和 float_add_module.vt 是一样的操作，但是随着操作符的不同，运算结果页随着不同。

仿真结果：

至于详细的仿真过程，基本上和实验二十是换汤不换药，笔者就不重复了，读者自己看着办吧。换之，笔者把运算的结果打印出来给读者作以下比较（说实话以下有几组操作数是非常的挑剔）：

```
Q1: A = 3.65, B = -7.4 , A - B = ?
```

```
A = 32'b0_10000000_11010011001100110011010
B = 32'b1_10000001_11011001100110011001101
```

```
32'b0_10000010_01100001100110011001101 Test Result 11.0500001
32'b0_10000010_01100001100110011001101 Teal Result 11.0500001
                                       True Result 11.05
```

Q2: A = 1.9999997 , B = -1.9999998 , A - B = ?

```
A = 32'b0_01111111_11111111111111111111110
B = 32'b1_01111111_11111111111111111111111

32'b0_10000000_11111111111111111111111 Test Result 3.9999997
32'b0_10000000_11111111111111111111111 Real Result 3.9999997
                                       True Result 3.9999997
```

Q3: A = 78992.118 , B = -90116.402 , A - B = ?

```
A = 32'b0_10001111_00110100100100000001111
B = 32'b1_10001111_01100000000001000110011

32'b0_10010000_01001010010010100100001 Test Result 169108.5156250
32'b0_10010000_01001010010010100100001 Real Result 169108.5156250
                                       True Result 169108.52
```

关于这个问题，读者需要注意的地方就是精度问题了。A - B 的结果显示 Test Result 和 Real Result 已经开始偏远 True Result 了。原因是——单精度格式已经是勉强承载运算过后的结果。用一句傻瓜的话来说，8 个十进制位数（包括小数），单精度浮点数对它的支持已经出现“负荷”的现象了。

Q4: A = 6992.330 , B = 1271.004 , A - B = ?

```
A = 32'b0_10001011_10110101000001010100100
B = 32'b0_10001001_00111101110000000100001

32'b0_10001011_01100101100101010011100 Test Result 5721.3261718
32'b0_10001011_01100101100101010011100 Real Result 5721.3261718
                                       True Result 5721.326
```

比起 Q3，Q4 的 Test Result 和 Real Result 显得更接近 True Result。用傻瓜的话来说，7 个十进制的位数（包括小数），对于单精度格式来说是最大的承载极限。

Q5: A = -1.5032 , B = -8.3309 , A - B = ?

```
A = 32'b1_01111111_10000000110100011011100
B = 32'b1_10000010_00001010100101101011110
```

```
32'b0_10000001_10110100111110010000101 Test Result 6.8277001
32'b0_10000001_10110100111110010000101 Real Result 6.8277001
                                       True Result 6.8277
```

```
Q6: A = -88.8888 , B = -10.9944 , A - B = ?
```

```
A = 32'b1_10000101_01100011100011100010001
B = 32'b1_10000010_01011111110100100010000

A = 32'b1_10000101_00110111100100111101111 Test Result -77.8944015
B = 32'b1_10000101_00110111100100111101111 Real Result -77.8944015
                                           True Result -77.8944
```

对于拥有 6 个十进制位数的结果（包括小数），是最佳的状态。

```
Q7: A = 0.001584 , B = 0.063197 , A - B = ?
```

```
A = 32'b0_01110101_10011111001111000111000
B = 32'b0_01111011_00000010110110101101110

32'b1_01111010_11111000101110111101010 Test Result -0.0616130
32'b1_01111010_11111000101110111101010 Real Result -0.0616130
                                       True Result -0.061613
```

即只有小数的情况下，6 个位数的十进制（小数而已）的结果，单精度格式的支持还不错。

```
Q8: A = -613.931 , B = 27.11402 , A - B = ?
```

```
A = 32'b1_10001000_00110010111101110010110
B = 32'b0_10000011_10110001110100110000011

32'b1_10001000_01000000100001011100010 Test Result -641.0450439
32'b1_10001000_01000000100001011100010 Real Result -641.0450439
                                       True Result -641.04502
```

Q8 中 A 和 B 的操作数，10 进制的位数已经不是同侪了，运行以后的结果 ... Test Result 和 Real Result 都偏离 True Result。果然拥有 7 个十进制的位数（包括小数），单精度格式不好支持。

```
Q9: A = 44.4444 , B = 68.3333 , A - B = ?
```

```
A = 32'b0_10000100_01100011100011100010001
B = 32'b0_10000101_00010001010101010101010

32'b1_10000011_01111110001110010000110 Test Result -23.8889274
```

```
32'b1_10000011_01111110001110001111000 Real Result -23.8889007
                                       True Result -23.8889
```

结果再一次的显示，6 个十进制位数（包括小数）是单精度格式在加减操作中，Test Result 和 Real Result 是最接近 True Result 的。不过比较愕然的是，即使 Test Result 中最后几个尾数 和 Real Result 不同，但是它们取得的结果都和 True Result 差不多。

到目前为止我们对单精度浮点数的认识已经有一定的程度了，因此对于实现加减运算是绝对没有问题的。关于单精度格式有一定的承载能力，如果用 10 进制的尾数作为标准的话，我们可以得出一个结论，那就是拥有 6 个十进制位数（包括小数）的操作数，可以取得最接近 True Result 的加减运算结果。

4.8 浮点乘法和陷阱

什么是浮点数的乘法呢？我们先从数学的角度上去认识它：

$$
\begin{aligned}
5.0 \times 10^{2} \times -2.2 \times 10^{3} &= (1)5.0 \times 10^{2} \times (-1)2.2 \times 10^{3} \\
&= (1 \times -1)\ 5.0 \times 2.2 \times 10^{2+3} \\
&= (-1)11 \times 10^{5} \\
&= -11 \times 10^{5}
\end{aligned}
$$

在传统的数学上，浮点数的乘法可以分成 3 个部分，就是符号位运算（正负运算），数目运算（尾数运算）和价码运算。当我们把同样的概念带入 Verilog 的世界，同样的概念只能发挥一半的功能而已。如果我们一意孤行使用相同的概念，以 2 进制的方式来实现浮点乘法的话，结果必定天珠。这就是著名的浮点乘法陷阱。

在前面，笔者已经说过那一套用在我们认知的常识里，是绝对不适合用在 Verilog 的世界里。我们认识里，数字是以 10 进制为基础，浮点数不存在精度问题的问题。另一个比较明显的例子，就是 4.4 小节里笔者所讨论过的“实定移位”和“假定移位”。至于浮点乘法如何在两个不同的世界里产生违和感，以致掉入它的陷阱里！？接下来，就让我们慢慢去发现 ...

首先让我们先从数据的角度上看二进制运行浮点数乘法的运算？

```
1.110E+2 × -1.011E+3 = (1)1.110E+2 × (-1)1.011E+3
                     = (1)( -1) 1.110 × 1.011 E(2+3)
                     = (-1) 1.0011010 E+5
                     = -1.0011010 E+5
```

符号位 = 1 (-1) = -1	1110 1011 × 1110 11100 000000 1110000 + 尾数 10011010	价码 = 2 + 3 = 5

```
结果 =  (符号位)尾数 E(价码)
     =  (-1)1.0011010E(5)
     =  -1.0011010E+5
```

对于浮点数按二进制方式来执行乘法运算，同样可以用传统数学的方式实现，亦即符号位，尾数和价码分 3 部分来运算。不过不同的是尾数在累加过程中，小数点没有被包含进来。在这里有学同学可能会问“既然尾数在累加过程中不包含小数点，那么在结果中怎样去定义小数点的存在？”嗯！这确实是重点问题。

在前面笔者已经讲过了，在二进制的运算中小数点的存在是“假定”，而不是书写形式上的“实定”。对于“假定”的小数点来说它是被“夹在某个范围之间”，作为前提条件我们需要某个“固定”的位宽。如上面的例子中，操作数 A 和 B 均为 4 位宽，那么这个“固定”的空间就是 A 和 B 的总和。最后“假定”的小数点就是“夹在最高位和最高第二位”之间。具体一点来说，假设 A 和 B 的乘法结果寄存在名为 Temp 的空间里，那么 Temp[7] 和 Temp[6] 之间就是假定的小数点。

读到这里读者估计已经认识到最基本“浮点乘法”在十进制和二进制之间的区别。更深一层的认识，接下来我们要实现单精度格式的浮点数乘法。为了预防一些同学发生失忆，我们在来复习一下单精度的格式：

```
[ 0][0111111][00000000000000000000000]  //默认空壳
[31][-30:23-][---------22:0----------]
[符][-价码--][---------尾数----------]
```

单精度格式可以分为 3 部分，亦即符号位[31]，价码[30~23]和尾数[22:0]。默认空壳中，除了符号位和尾数以外（全零），价码的值是 8'b01111111（在专业的用词中，8'b0111111 称为价码的偏移量）。在尾数中还包含一个“隐藏位”，在单精度格式中“隐藏位”是没有被表示出来的。

普通下，我们都会按照浮点数的乘法执行习惯，把符号位，价码和尾数分为 3 个部分来运算。结合我们对单精度浮点数的加减运算的经验，大致的步骤如下：

（一）操作数预处理：恢复单精度浮点数中的隐藏位；
（二）运算：符号位操作，价码运算操作，尾数运算操作独立运行；
（三）调整结果：将运算结果调整至适合浮点数源的格式；
（四）输出和格式化：从尾数中摘取前半部分，然后检测是否执行四舍五入。

乍看下，乘法运算的过程比起加减运算来得更简单，不过单是这样空口白谈感觉还是很抽象的，笔者几个简单的例子吧：

Q1: A = 2.5 , B = 5 , A × B = ?

```
A = 32'b0_10000000_01000000000000000000000

B = 32'b0_10000001_01000000000000000000000


操作数预处理：


A' = 32'b0_10000000_10100000000000000000000

B' = 32'b0_10000001_10100000000000000000000


运算：


sum.s     = A'.s ^ B'.s

          = 0 ^ 0

          = 0
```

```
sum.exp   = A'.exp + B'.Exp
          = 8'b10000000 + ( 8b'10000001 - 127 )
          = 128 + ( 129 - 127)
          = 128 + 2
          = 130
          = 0b10000010

sum.m     = A'.m * B'.m
          = 24'b101000000000000000000000 * 24'b101000000000000000000000
          = 48'b011001000000000000000000000000000000000000000000

结果调整：
$ sum.m[47] is 0, while sum.m << 1

011001000000000000000000000000000000000000000000 before
110010000000000000000000000000000000000000000000  << 1
110010000000000000000000000000000000000000000000 after, sum = sum'

输出和格式化：
$ Take out sum'.m[46:24], if sum'.m[23] is 1  sum'.m[46:24] + 1

32'b0_10000010_10010000000000000000000 Test Result 12.5
32'b0_10000010_10010000000000000000000 Real Result 12.5
                                       True Result 12.5
```

步骤一：操作数预处理

首先是操作数预处理，乘法运算和加减运算不一样的是，乘法运算没有“价码对齐”这一个步骤，除了不用移动尾数以外，更不需要而外的空间来补偿尾数因位移使失去的部分。此外，还有一点的是，隐藏位的恢复也取得 1 位而已，Verilog HDL 语言的实现如代码 4.8.1 所示：

```
begin
    rA <= { A[31], A[30:23], 1'b1, A[22:0]};
    rB <= { B[31], B[30:23], 1'b1, B[22:0]};
end
```

代码 4.8.1 操作数预处理

步骤二：运算

下来的操作时 3 个部分——符号位，价码和尾数同时运算。符号位的运算可以用亦或取得；价码的运算也可以用简单的加减运算符取得；至于尾数的乘法运算，笔者为了很好说明，笔者先用默认的“*”运算符取得尾数 A 和 B 的相乘结果。Verilog HDL 语言的实现如代码 4.8.2 所示：

```
begin
    isSign <= A[31] ^ B[31];

    BDiff = rB[31:24] - 8'd127; // + (~8'd127 + 1)
    rExp <= rA[31:24] + BDiff; // A.Exp + B.Exp

    Temp <= rA[23:0] * rB[23:0];
end
```

代码 4.8.2 运算

步骤 3：结果调整

运算以后我们要调整结果，这一点和加减运算一样。因为尾数在运算以后，结果不一定是适合“浮点数源”的格式。至于“假定”的小数点是介于“最高位和最高第二位之间”。其外有一点不同的是，乘法运算在结果调整中，调整范围不必加减运算的大，举个例子：

```
QI：                        QII：                       QIII：                      QIV：
    4'b1111                     4'b1000                     4'b0100                     4'b0100
    4'b1111 x                    4'b1000 x                   4'b1111 x                   4'b1000 x
8'b11100001 before          8'b01000000 before          8'b00111100 before          8'b00100000 before
8'b11100001 after           8'b10000000 << 1            8'b11110000 << 2            8'b10000000 << 2
                            8'b10000000 after           8'b11110000 after, Exp - 1  8'b10000000 after, Exp - 1
```

我们假设操作数 A 和 B 均为 4 位宽，那么相乘结果最大的可能必然是 8 位宽，至于“假定”小数点必是 [7]~[6]之间。在上面 4 个例子中，笔者取出最有可能得到的 4 大结果。

一、首先是 QI 的结果，其中最高位为 1 所以不用调整；
二、QII 的结果中最高位值为 0，所以必须左移 1 位；
三、QIII 的结果中，[7:6]都为 0，所以必须左移 2 位，然后价码减 1；
四、QIV 的结果中，[7:6]双双还是为 0，所以必须左移 2 位，然后价码减 1。

从上面 4 个结果中，Q1 不用调整，价码理所当然不用更改，至于 Q2 的结果再调整中（左移 1 位），价码同样不用被更改。但是 Q3 和 Q4 的结果再调整中（左移 2 位），价码必须被调整为减 1。在此可能有很多同学会问 WHY！不要问 WHY 了，这是隐藏在乘法运算中的陷阱之一。操作数无论是 4 位宽还是 24 位宽，这个陷阱同样存在，所以读者们要多多醒目！如果要用 Verilog HDL 语言来实现的话，如代码 4.8.3 所示：

```
begin
    if( Temp[47] == 1'b1 ) begin Temp <= Temp; end // do nothing
    else if( Temp[46] == 1'b1 ) begin Temp <= Temp << 1; end
    else if( Temp[45] == 1'b1 ) begin Temp <= Temp << 2; rExp <= rExp - 1'b1;end
end
```

代码 4.8.3 结果调整。

步骤 4：输出和格式化

在这个步骤中，我们同样要检查价码溢出或者尾数零值等错误信息以外，我们要四舍五入尾数的结果。价码溢出和尾数零值的检测过程，笔者就不多说了，基本上和前两个实验都一样。不过在执行四舍五入的时候，概念会有所不同。在输出之前，我们只能从尾数的运算结果中摘取一部分而已，假设这个运算空间为 48 位宽的 Temp，Temp[47]必定隐藏位，然而 Temp[46:24]就是我们要摘取的部分，如果 Temp[23]的值为 1 的话，那么我们必须为 Temp[46:24]加 1。

Verilog HDL 语言实现的过程如代码 4.8.4 所示：

```
begin
    if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end       // E Overflow
    else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // E Underflow
    else if( Temp[47:24] == 24'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // M Zero
    else if( Temp[23] == 1'b1 ) rResult <= { isSign, rExp[7:0], Temp[46:24] + 1'b1 }; // okay with normalised
    else rResult <= { isSign, rExp[7:0], Temp[46:24] }; // okay without normalise
end
```

代码 4.8.4 输出和格式化。

==========

到目前为止，如果读者只以为浮点乘法只有一个陷阱而已，那么读者就是大错特错了。第二个陷阱更隐蔽，更危险，它也是在许多浮点数乘法器设计上所犯的问题。笔者再举一个例子：

```
Q2: A = 2 , B = 2 ,  A×B = ?
```

先插入一段废话，尽管 Q2 是一个小学生的问题，但是在此刻它是一个无比的经典，读者要好好的铭记于心。至于这个连小学生都会的问题，到底会出现什么样的陷阱？让我们慢慢去发掘吧！（以下的过程中笔者把操作数预处理简化掉）

```
A = 32'b0_10000000_00000000000000000000000

B = 32'b0_10000000_00000000000000000000000


运算：


sum.s = A.s ^ B.s = 0 ^ 0 = 0


sum.exp = A.exp + B.exp
        = 0b10000000 + ( 0b10000000 - 127 )
        = 0b10000001
```

```
sum.m = A.m * B.m
      = 24'b100000000000000000000000 * 24'b100000000000000000000000
      = 48'b010000000000000000000000000000000000000000000000


结果调整：
$ sum.m[47] is 0, while sum.m << 1

48'b010000000000000000000000000000000000000000000000 before
48'b100000000000000000000000000000000000000000000000 << 1
48'b100000000000000000000000000000000000000000000000 after, sum = sum'

输出和格式化：
32'b0_10000001_00000000000000000000000 Test Result 4
32'b0_10000001_00000000000000000000000 Real Result 4
                                       True Result 4
```

从 Q2 的运算结果中，我们看到 2 * 2 也存在第一个陷阱。在结果调整中，如果尾数必须右移一位才能达到浮点数源的格式，按常理来说我们应该讲价码递减为 1。如果我们真的把价码递减，那么 True Result 就会和 Real Result 不同的结果。

Q3：A = 3.142 , B = 3.142 , A × B = ？

```
A = 32'b0_10000000_10010010001011010000111
B = 32'b0_10000000_10010010001011010000111

运算：
sum.s = A.s ^ B.s = 0 ^ 0 = 0

sum.exp = A.exp + B.exp
        = 0b10000000 + ( 0b10000000 - 127 )
        = 0b10000001

sum.m = A.m * B.m
      = 24'b110010010001011010000111 * 24'b110010010001011010000111
      = 48'b100111011111010001100001111110010111101100110001

结果调整：
$ sum.m[47] is 1, while do nothing

输出和格式化：
$ sum.m[23] is 1, while sum.m[46:24] + 1

h m
```

```
1 00111011111010001100001 before
1 00111011111010001100010 +1
1 00111011111010001100010 after, sum = sum'

32'b0_10000001_00111011111010001100010 Test Result 4.9360818
32'b0_10000010_00111011111010001100010 Real Result 9.8721637
                                       True Result 9.872164
```

Q3 我们用和 Q2 一样的运算步骤，可是得出的乘法运算结果 Test Result 却和 Real Result 差十万八千里，此外 Real Result 和 True Result 的结果又差不多，因此我们更能肯定 Test Result 出现错误了。然后我们再比较一下 Test Result 和 Real Result 的二进制值，从中我们可以发现 Test Result 的价码比起 Real Result 的价码少了一位。这又是另一个浮点乘法的阴森陷阱！

为了解决这个问题，我们必须重新审视之前有关浮点乘法的运算步骤。关于之前浮点乘法概念，我们视符号运算，价码运算和尾数运算是 3 方可以独立运行的。这个观点在传统的数学角度上，是绝对没有问题，换之在 Verilog 里，尤其是二进制的世界是绝对有问题的。既然我们已经知道问题是发生在概念或者观点上，但是我们要如何解决问题呢？

问题源于概念切口自然直入概念，所以我们可以这样思考："符号运算是一个独立的，价码运算和尾数运算又另一个独立的 ... "因此，乘法运算再也不是 3 方法的，而是两方独立运行。借此我们可以利用浮点运算自身的微妙能力，我们知道浮点数的小数点是可以跳来跳去不是吗？举个例子：

$$5.1 \times 10^3 \times 2.2 \times 10^2 = 5.1 \times 10^3 \times 0.22 \times 10^3$$

1.101E+3 × 1.011 E+2 = 1.101E+3 × 0.1101 E+3

上面的例子中，笔者都是把操作数 B 的价码递增，然后小数点左移 1 位（二进制值右移一位），结果的基本值还是等价的。我们可以利用这个微妙的移位动作来解决 Q3 的陷阱问题。不过不是什么问题组（值操作数 A 和操作数 B）都可以使用此方法，唯有某个"特殊条件"下，才能使用这个方法。所以呢，我们必须建立某个逻辑来判断这个"特殊条件"。最后，我们必须重新整理一下之前的乘法步骤。

（一）操作数预处理：恢复单精度浮点数中的隐藏位；
（二）符号位运算：符号位运算操作独立运行；
（三）移位预处理：根据某个"特殊条件"，把操作数 B 的小数点左移 1 位，然后价码递增 1；
（四）价码和尾数运算：价码运算操作和尾数运算操作独立运行；
（五）调整结果：将运算结果调整至适合浮点数源的格式；
（六）输出和格式化：从尾数中摘取前半部分，然后检测是否执行四舍五入。

这个特殊条件有 2 个，笔者先说其中一个，那就是："操作数组价码等价，但是双反尾数同不等于 0"。操作组是指操作数 A 和 B，如果它们之间的价码等价，然后尾数又不等于 0，那么移位预处理就可以执行了（注意在判断过程不包括隐藏位，但是在移位过程中却包括隐藏位）。如果用 Verilog HDL 语言来表示的话，代码如 4.8.5 所示：

```
if( rA[31:24] == rB[31:24] && ( rA[22:0] != 23'd0 & rB[22:0] != 23'd0 ) )
```

```
begin rB[31:24] <= rB[31:24] + 1'b1; rB[23:0] <= rB[23:0] >> 1; end
```

代码 4.8.5 移位预处理。

我们用新的运算步骤再一次运算 Q3 的问题看看：

Q3b：A = 3.142，B = 3.142，A × B = ？

```
A = 32'b0_10000000_10010010001011010000111
B = 32'b0_10000000_10010010001011010000111

符号位运算：
sum.s = A.s ^ B.s = 0 ^ 0 = 0

移位预处理：
$ A.exp == B.exp && A.m and B.m not a zero ( no include hidden bit ),
  while B.m >> 1 and B.exp + 1 (include hidden bit )

B'.exp = 0b10000000 + 1 = 0b10000001

h m
1 10010010001011010000111 before
0 11001001000101101000011 >> 1  //无视尾数补偿
0 11001001000101101000011 after, B = B'

价码和尾数运算：
sum.exp = A.exp + B'.exp
        = 0b10000000 + ( 0b10000001 - 127 )
        = 0b10000010

sum.m = A.m * B.m
      = 24'b110010010001011010000111 * 24'b011001001000101101000011
      = 48'b010011101111101000110000100110000011001001010101

结果调整：
$ sum.m[47] is 0, while sum.m << 1

h  m
0 100111011111010001100001001100000110010010101010 1 before
1 001110111110100011000010011000001100100101010 << 1
1 001110111110100011000010011000001100100101010 after, sum = sum'

输出和格式化：
$ sum.m[23] is 0, while do nothing
```

```
32'b0_10000010_00111011111010001100001 Test Result 9.8721628
32'b0_10000010_00111011111010001100010 Real Result 9.8721637
                                       True Result 9.872164
```

当我们重新用不同的概念和步骤运行 Q3 的问题以后，最终求得的 Test Result 和 Real Result 差不多。不过操作数 B 在移位预处理的途中，笔者无视尾数的补偿，小小的 1 位尾数而已，用不着那么大惊小怪，有没有补偿它对结果没有多大影响。

在此，可能有些同学要求能不能把“特殊条件”的判断逻辑简化一下？

```
if( rA[31:24] == rB[31:24] && ( rA[22:0] != 23'd0 & rB[22:0] != 23'd0 ) )   // before  正版
if( rA[31:24] == rB[31:24] && rA[22:0] == rB[22:0] )                        // after   简化版
```

那些同学可能因由 Q3b 操作组中“操作数 A 和操作数 B 同样”的关系才会那么想 ... 事实上，那是万万不可的动作！不相信吗？我们再运算一下 Q2 的问题（2*2）就知道结果了。

```
Q2b：A = 2，B = 2，A × B = ？
```

```
A = 32'b0_10000000_00000000000000000000000
B = 32'b0_10000000_00000000000000000000000

浮点位运算：
sum.s = A.s ^ B.s = 0 ^ 0 = 0

移位预处理：
$ A.exp == B.exp && A.m == B.m，while B.m << 1 and B.exp + 1

B'.exp = 0b10000000 + 1
       = 0b10000001

h m
1 00000000000000000000000 before
0 10000000000000000000000 >> 1
0 10000000000000000000000 after, B = B'

价码和尾数运算：
sum.exp = A.exp + B'.exp
        = 0b10000000 + ( 0b10000001 - 127 )
        = 0b10000010

sum.m = A.m * B'.m
      = 24'b100000000000000000000000 * 24'b010000000000000000000000
      = 48'b001000000000000000000000000000000000000000000000
```

```
结果调整：
$ sum.m[47] is 0, while sum.m << 2 and sum.exp - 1

sum'.exp = 0b10000010 - 1 = 0b10000001

h m
0 01000000000000000000000000000000000000000000 before
1 00000000000000000000000000000000000000000000 << 2
1 00000000000000000000000000000000000000000000 after, sum = sum'

32'b0_0b10000001_00000000000000000000 Test Result 4
32'b0_0b10000001_00000000000000000000 Real Result 4
                                      True Result 4
```

同学得意的说："笔者！看吧！特殊条件的判断逻辑简化以后 Q2 的结果还不是一样正确吗？"少得意了这位同学 … 我们再运算另一个问题看看。

Q4：A = 273.757，B = 483.265，A × B = ？

```
A = 32'b0_10000111_00010001110000011100101
B = 32'b0_10000111_11100011010000111101100

符号位预算：
sum.s = A.s ^ B.s = 0

移位预处理：
$ A.exp == B.exp but A.m != B.m, while do nothing ... //简化判断条件无法成立

价码和尾数运算：=
sum.exp = A.exp + B.exp
        = 0b10000111 + ( 0b10000111 - 127 )
        = 0b10001111

sum.m = A.m * B.m
      = 24'b100010001110000011100101 *  24'b111100011010000111101100
      = 48'b100000010011001001001011001101000101100000011100

结果调整：
$ sum.m[47] is 1, while do nothing ...

输出和格式化：
32'b0_10001111_00000010011001001001011 Test Result 66148.5859375
32'b0_10010000_00000010011001001001011 Real result 132297.1718750
                                       True result 132297.176605
```

从 Q4 的结果看来，我们就可以得知结果 Test Result 已经大量偏移 Real Result 和 True Result 的值。从 Test Result 和 Real Result 的区别中，我们非常清楚的看到它们之间价码相差 1。因此我们可以断定简化版的判断逻辑（特殊条件用）只能运行部分功能而已，实际上是抬不了场面。接下来，我们用回完整版的判断逻辑再一次运算 Q4 看看。

Q4b：A = 273.757，B = 483.265，A × B = ？

```
A = 32'b0_10000111_00010001110000011100101
B = 32'b0_10000111_11100011010000111101100

符号位预算：
sum.s = A.s ^ B.s = 0

移位预处理：
$ A.exp == B.exp and A.m != 0 and B.m != 0 （not include hidden bit），
  while B.m << 1 ( include hiddent bit ) and B.exp + 1

B'.exp = 0b10000111 + 1 = 0b10001000

h m
1 11100011010000111101100 before
0 11110001101000011110110 >> 1
0 11110001101000011110110 after, B = B'

价码和尾数运算：
sum.exp = A.exp + B'.exp
        = 0b10000111 + ( 0b10001000 - 127 )
        = 0b10010000

sum.m = A.m * B.m
      = 24'b100010001110000011100101 *  24'b011110001101000011110110
      = 48'b010000001001100100100101100110100010110000001110

结果调整：
$ sum.m[47] is 0, while sum.m << 1

h m
0 100000100110010010010110011010001011000000011100 before
1 000000100110010010010110011010001011000000111000 << 1
1 000000100110010010010110011010001011000000111000 after, sum = sum'

输出和格式化：
32'b0_10010000_00000010011001001001011 Test Result 132297.1718750
32'b0_10010000_00000010011001001001011 Real Result 132297.1718750
```

```
True Result 132297.176605
```

当我们用回正版的判断逻辑（特殊条件用）Q4 取得的结果 Test Result 已经和 Real Result 一样了。不过它们距离 True Result 还是有点精度上的差别。结论，我们可以证明简化版的判断逻辑只是鹌鹑而已，用回正版的东西才是王道。

==========

在前面笔者声明过“特殊条件”有两个，其中一个是操作数组价码等价，但是双方的尾数都不等于 0 时。另一个“特殊条件”就是“操作组的符号位均不同”，移位预处理同样会被执行。。第二个“特殊条件”用 Verilog HDL 语言来表达更简单，如代码 4.8.5 所示：

```
if( rA[31:24] == rB[31:24] && ( rA[22:0] != 23'd0 & rB[22:0] != 23'd0 ) || rA[32] ^ rB[32] )
      begin rB[31:24] <= rB[31:24] + 1'b1; rB[23:0] <= rB[23:0] >> 1; end
```

代码 4.8.5 第二会引起移位预处理的特殊条件。

“哦 ... 纳尼!? (⊙o⊙)”，读者别用这样的表情看笔者，笔者所说的话都是事实。不信吗？那好，我们再运算其他问题看看。

```
Q5 A = 6.3565 , B = -0.0063 , A × B = ?
```

```
A = 32'b0_10000001_10010110110100001110011
B = 32'b1_01110111_10011100111000000111011

价码运算：
sum.s = A.s ^ B.s= 0 ^ 1 = 1

移位预处理：
$ sum.s is 1,while B.m << 1 and B.exp + 1

B'.exp = 0b01110111 + 1 = 0b01111000

h m
1 10011100111000000111011 before
0 11001110011100000011101 >> 1
0 11001110011100000011101 after, B = B'

价码和尾数运算：
sum.exp = A.exp + B'.exp
        = 0b10000001 + ( 0b01111000 - 127 )
        = 129 + ( 120 - 127 )
        = 122
        = 0b01111010
```

```
sum.m = A.m * B'm
      = 24'b110010110110100001110011 * 24'b011001110011100000011101
      = 48'b010100100000001110011100001010001111110100000111

结果调整：
$ sum.m[47] is 0, while sum.m << 1

h m
0 101001000000011100111000010100011111101000001 11 before
1 010010000000111001110000101000111111010000011 110 << 1
1 010010000000111001110000101000111111010000011 110 after, sum = sum'

输出和格式化：
32'b1_01111010_01001000000011100111000 Test Result -0.0400459
32'b1_01111010_01001000000011100111001 Real Result -0.0400459
                                       True Result -0.04004595
```

读者看到了吧，当操作数 A 和 B 之前的符号位异或运算以后，如果是 1 那么移位预处理就要执行了。如 Q5 所示 A 和 B 的符号位预算结果是 1，经过移位预处理以后，最终求得的 Test Result 和 Real Result 差不多。

在 Verilog 的世界里，尤其是二进制实现乘法运算，这里那里到处都充满陷阱。在这个小节里，我们知道乘法运算又 2 个陷阱，第一是在结果调整中，另一个是移位预处理中。这两个陷阱是非常隐蔽的，这也使得浮点数乘法器设计的难度大大提升。笔者还记得在网上冲浪寻找资源的时候，笔者发现很多网站的内容都无视这个问题，此外相关的论文也显得更疲软了，笑~

笔者不知道著名的有关 Pentium 在浮点数运算的 BUG 是不是和这些陷阱有关系？但是笔者非常可定的是，要发现陷阱就要勇敢去踩陷阱，作为发现陷阱的带价就是付出一大半的 HP（勇者斗恶魔经典的精神）。

4.9 实验二十二：浮点数乘法器

在前一个小节里，笔者不但详细讨论过在浮点乘法会出现的隐蔽陷阱，此外还讨论浮点乘法的基本概念和过程。总结下，浮点乘法的大概步骤如下：

（一）操作数预处理：恢复单精度浮点数中的隐藏位；
（二）符号位运算：符号位运算操作独立运行；
（三）移位预处理：根据某个“特殊条件”，把操作数 B 的小数点左移 1 位，然后价码递增 1；
（四）价码和尾数运算：价码运算操作和尾数运算操作独立运行；
（五）调整结果：将运算结果调整至适合浮点数源的格式；
（六）输出和格式化：从尾数中摘取前半部分，然后检测是否执行四舍五入。

在实验二十二里我们就是按照这些步骤去设计一个浮点数乘法模块。

图 4.9.1 浮点数乘法模块图形。

图 4.9.1 是浮点数乘法模块的图形，至于信号之间的定义，基本上和之前的几个实验都一样。在此笔者就不罗嗦了，我们直接来看代码吧。

float_multi_module.v

```
module float_multi_module
(
    input CLK, RSTn,
    input [31:0]A,B,
    output [31:0]Result,

    input Start_Sig,
    output [3:0]Done_Sig,

    /*****************/

    output [32:0]SQ_rA,SQ_rB,
    output [47:0]SQ_Temp,
    output [9:0]SQ_rExp,SQ_BDiff

);
```

第 1~9 行是该模块的相关输入输出声明，至于第 12~14 行是仿真用的输出。

```
17.  /*************************************/
18.
19.      reg [3:0]i;
20.      reg [32:0]rA,rB;   // [32]Sign, [31:24]Exponent, [23]Hidden Bit, [22:0]Mantissa
21.      reg [47:0]Temp;     // [48]M'sign, [47:46]Hidden Bit, [45:23]M, [22:0]M'resolution
22.      reg [31:0]rResult;
23.      reg [9:0]rExp,BDiff;        //[9:8] Overflow or underflow check, [7:0] usuall exp.
24.      reg isSign;
25.      reg isOver;
26.      reg isUnder;
27.      reg isZero;
28.      reg isDone;
29.
30.      always @ ( posedge CLK or negedge RSTn )
31.          if( !RSTn )
32.              begin
33.                  i <= 4'd0;
34.                  rA <= 33'd0;
35.                  rB <= 33'd0;
36.                  Temp <= 48'd0;
37.                  rResult <= 32'd0;
38.                  rExp <= 10'd0;
39.                  BDiff <= 10'd0;
40.                  isOver <= 1'b0;
41.                  isUnder <= 1'b0;
42.                  isZero <= 1'b0;
43.                  isDone <= 1'b0;
44.              end
```

第 19~28 行是相关的寄存器声明，rA 和 rB 是用来寄存操作数 A 和 B 的隐藏位恢复结果，因此位宽声明为 32；Temp 是用来寄存相乘的结果；至于其他的寄存器，大意上和之前的几个实验都一样。

```
45.          else if( Start_Sig )
46.              case( i )
47.
48.                  0: // Initial and resulted out Signature
49.                  begin
50.                      rA <= { A[31], A[30:23], 1'b1, A[22:0]};
51.                      rB <= { B[31], B[30:23], 1'b1, B[22:0]};
52.                      isSign <= A[31] ^ B[31];
53.
```

```
54.                 isOver <= 1'b0; isUnder <= 1'b0; isZero <= 1'b0;
55.                 i <= i + 1'b1;
56.             end
57.
58.             1:
59.             begin
60.                 if( rA[31:24] == rB[31:24] && ( rA[22:0] != 23'd0 & rB[22:0] != 23'd0 ) || rA[32] ^ rB[32] )
61.                     begin rB[31:24] <= rB[31:24] + 1'b1; rB[23:0] <= rB[23:0] >> 1; end
62.                 i <= i + 1'b1;
63.             end
```

第 48~56 行是操作数预处理，除了从操作数 A 和 B 恢复隐藏位以外，还有清零相关的错误标志寄存器（第 54 行）。第 58~63 行是移位预处理，注意第 60 行是移位预处理的判断条件。在此笔者再强调一下，在判断图中隐藏位是不参与，反之在判断条件成立后，隐藏位在尾数 B 的移位却参与。

```
64.
65.             2: // if rExp[9..8] is 1, mean A.Exp small than B.Exp
66.                 // while rExp[9..8] is 0, mean A.Exp large than B.Exp or same.
67.             begin
68.                 BDiff = rB[31:24] - 8'd127; // + (~8'd127 + 1)
69.                 rExp <= rA[31:24] + BDiff; // A.Exp + B.Exp
70.                 i <= i + 1'b1;
71.             end
72.
73.             3:
74.             begin
75.                 Temp <= rA[23:0] * rB[23:0];
76.                 i <= i + 1'b1;
77.             end
78.
79.             4: // Check M'hidden bit
80.             begin
81.                 if( Temp[47] == 1'b1 ) begin Temp <= Temp; end // do nothing
82.                 else if( Temp[46] == 1'b1 ) begin Temp <= Temp << 1; end
83.                 else if( Temp[45] == 1'b1 ) begin Temp <= Temp << 2; rExp <= rExp - 1'b1;end
84.
85.                 i <= i + 1'b1;
86.             end
```

第 65~71 行是价码运算，第 73~77 行是尾数运算，至于运算过程在 4.8 小节里已经解释过了，在此读者自己看着办。第 79~86 行是结果调整，比起加减操作，乘法操作在该步骤显得更简单，不过读者不要大意，这里是某个陷阱点。

```
87.
```

```
88.                     5: //error check and decide final result
89.                     begin
90.                         if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end       // E Overflow
91.                         else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // E Underflow
92.                         else if( Temp[47:24] == 24'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // M Zero
93.                         else if( Temp[23] == 1'b1 ) rResult <= { isSign, rExp[7:0], Temp[46:24] + 1'b1 }; // okay with normalised
94.                         else rResult <= { isSign, rExp[7:0], Temp[46:24] }; // okay without normalise
95.                         i <= i + 1'b1;
96.                     end
97.
98.                     6:
99.                     begin isDone <= 1'b1; i <= i + 1'b1; end
100.
101.                    7:
102.                    begin isDone <= 1'b0; i <= 4'd0; end
103.
104.                endcase
105.
106.        /*************************************/
```

第 88~96 行是输出和格式化，内容基本上和之前的实验大同小异，除了第 92 行的“假定”小数点的位置不同以外，还有就是第 93~94 行之间尾数的位置。用简单的话来说，假定小数点夹在 Temp[47]~Temp[46]之间，然而输出用的尾数是 Temp[46 .. 24]，最后四舍五入的判断位置是 Temp[23]。

```
107.
108.        assign Done_Sig = { isOver, isUnder, isZero, isDone };
109.        assign Result = rResult;
110.
111.        /*************************************/
112.
113.          assign SQ_rA = rA;
114.          assign SQ_rB = rB;
115.          assign SQ_rExp = rExp;
116.          assign SQ_BDiff = BDiff;
117.          assign SQ_Temp = Temp;
118.
119.        /*************************************/
120.
121.  endmodule
```

第 108~109 行是相关的输出驱动，第 113~117 行是仿真用的相关输出驱动。

float_multi_module.vt

```
1.    `timescale 1 ps/ 1 ps
2.    module float_multi_module_simulation();
3.
4.          reg CLK;
5.          reg RSTn;
6.
7.          reg [31:0] A;
8.          reg [31:0] B;
9.          reg Start_Sig;
10.
11.         wire [3:0]   Done_Sig;
12.         wire [31:0]   Result;
13.
14.         /*************************/
15.
16.         wire [9:0]   SQ_BDiff;
17.         wire [47:0]   SQ_Temp;
18.         wire [32:0]   SQ_rA;
19.         wire [32:0]   SQ_rB;
20.         wire [9:0]   SQ_rExp;
21.
22.         /*************************/
23.
24.         float_multi_module U1
25.           (
26.               .CLK(CLK),
27.                 .RSTn(RSTn),
28.                 .A(A),
29.                 .B(B),
30.                 .Result(Result),
31.                 .Start_Sig(Start_Sig),
32.                 .Done_Sig(Done_Sig),
33.                 /************************/
34.                 .SQ_BDiff(SQ_BDiff),
35.                 .SQ_Temp(SQ_Temp),
36.                 .SQ_rA(SQ_rA),
37.                 .SQ_rB(SQ_rB),
38.                 .SQ_rExp(SQ_rExp)
39.         );
40.
41.       /****************************/
42.
43.       initial
```

```
44.         begin
45.                 RSTn = 0; #10 RSTn = 1;
46.                 CLK = 0; forever #5 CLK = ~CLK;
47.         end
48.
49.       /******************************/
50.
51.       reg [5:0]i;
52.
53.         always @ (    posedge CLK or negedge RSTn )
54.                 if( !RSTn )
55.                     begin
56.                       A <= 32'd0;
57.                       B <= 32'd0;
58.                       Start_Sig <= 1'b0;
59.                       i <= 6'd0;
60.                   end
61.             else
62.                   case( i )
63.
64.                           0: //A=2.5    , B=5, A+B = ?
65.                           if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
66.                           else begin A <= 32'b01000000001000000000000000000000; B <= 32'b01000000101000000000000000000000; Start_Sig <= 1'b1; end
67.
68.                         1:
69.                         begin
70.                               $display("A = %b and rA = %b", A,SQ_rA);
71.                               $display("B = %b and rB = %b", B,SQ_rB);
72.                               $display("A.m * B.m = %b", SQ_Temp);
73.                               $display("Result = %b", Result);
74.                               i <= i + 1'b1;
75.                         end
76.
77.                         2: //3.142 * 3.142
78.                         if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
79.                         else begin A <= 32'b01000000010010010001011010000111; B <= 32'b01000000010010010001011010000111; Start_Sig <= 1'b1; end
80.
81.                         3:
82.                         begin
83.                               $display("A = %b and rA = %b", A,SQ_rA);
84.                               $display("B = %b and rB = %b", B,SQ_rB);
85.                               $display("A.m * B.m = %b", SQ_Temp);
86.                               $display("Result = %b", Result);
87.                               i <= i + 1'b1;
88.                         end
```

```
89.
90.                 4: //12.662 * 4.903
91.                 if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
92.                 else begin A <= 32'b01000001010010101001011110001101; B <= 32'b01000000100111001110010101100000; Start_Sig <= 1'b1; end
93.
94.                 5:
95.                 begin
96.                         $display("A = %b and rA = %b", A,SQ_rA);
97.                         $display("B = %b and rB = %b", B,SQ_rB);
98.                         $display("A.m * B.m = %b", SQ_Temp);
99.                         $display("Result = %b", Result);
100.                        i <= i + 1'b1;
101.                end
102.
103.                6: // 2 * 2
104.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
105.                else begin A <= 32'b01000000000000000000000000000000; B <= 32'b01000000000000000000000000000000; Start_Sig <= 1'b1; end
106.
107.                7:
108.                begin
109.                        $display("A = %b and rA = %b", A,SQ_rA);
110.                        $display("B = %b and rB = %b", B,SQ_rB);
111.                        $display("A.m * B.m = %b", SQ_Temp);
112.                        $display("Result = %b", Result);
113.                        i <= i + 1'b1;
114.                end
115.
116.                8: // 3.142 * 2
117.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
118.                else begin A <= 32'b01000000010010010001011010000111; B <= 32'b01000000000000000000000000000000; Start_Sig <= 1'b1; end
119.
120.                9:
121.                begin
122.                        $display("A = %b and rA = %b", A,SQ_rA);
123.                        $display("B = %b and rB = %b", B,SQ_rB);
124.                        $display("A.m * B.m = %b", SQ_Temp);
125.                        $display("Result = %b", Result);
126.                        i <= i + 1'b1;
127.                end
128.
129.                10: // 73.767 * 83.266
130.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
131.                else begin A <= 32'b01000010100100111000100010110100; B <= 32'b01000010101001101000100000110001; Start_Sig <= 1'b1; end
132.
133.                11:
```

```
134.                    begin
135.                        $display("A = %b and rA = %b", A,SQ_rA);
136.                        $display("B = %b and rB = %b", B,SQ_rB);
137.                        $display("A.m * B.m = %b", SQ_Temp);
138.                        $display("Result = %b", Result);
139.                        i <= i + 1'b1;
140.                    end
141.
142.                    12: // 83.266 * 83.266
143.                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
144.                    else begin A <= 32'b01000010101001101000100000110001; B <= 32'b01000010101001101000100000110001; Start_Sig <= 1'b1; end
145.
146.                    13:
147.                    begin
148.                        $display("A = %b and rA = %b", A,SQ_rA);
149.                        $display("B = %b and rB = %b", B,SQ_rB);
150.                        $display("A.m * B.m = %b", SQ_Temp);
151.                        $display("Result = %b", Result);
152.                        i <= i + 1'b1;
153.                    end
154.
155.                    14: // 1024 * 256
156.                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
157.                    else begin A <= 32'b01000100100000000000000000000000; B <= 32'b01000011100000000000000000000000; Start_Sig <= 1'b1; end
158.
159.                    15:
160.                    begin
161.                        $display("A = %b and rA = %b", A,SQ_rA);
162.                        $display("B = %b and rB = %b", B,SQ_rB);
163.                        $display("A.m * B.m = %b", SQ_Temp);
164.                        $display("Result = %b", Result);
165.                        i <= i + 1'b1;
166.                    end
167.
168.                    16: // 33.116 * 16.558
169.                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
170.                    else begin A <= 32'b01000010000001000111011011001001; B <= 32'b01000001100001000111011011001001; Start_Sig <= 1'b1; end
171.
172.                    17:
173.                    begin
174.                        $display("A = %b and rA = %b", A,SQ_rA);
175.                        $display("B = %b and rB = %b", B,SQ_rB);
176.                        $display("A.m * B.m = %b", SQ_Temp);
177.                        $display("Result = %b", Result);
178.                        i <= i + 1'b1;
```

```
179.                end
180.
181.                18: // 0.00416 * 0.00014
182.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
183.                else begin A <= 32'b00111011100010000101000010011100; B <= 32'b00111001000100101100110011110111; Start_Sig <= 1'b1; end
184.
185.                19:
186.                begin
187.                    $display("A = %b and rA = %b", A,SQ_rA);
188.                    $display("B = %b and rB = %b", B,SQ_rB);
189.                    $display("A.m * B.m = %b", SQ_Temp);
190.                    $display("Result = %b", Result);
191.                    i <= i + 1'b1;
192.                end
193.
194.                20: // 0.125 * 0.125
195.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
196.                else begin A <= 32'b00111110000000000000000000000000; B <= 32'b00111110000000000000000000000000; Start_Sig <= 1'b1; end
197.
198.                21:
199.                begin
200.                    $display("A = %b and rA = %b", A,SQ_rA);
201.                    $display("B = %b and rB = %b", B,SQ_rB);
202.                    $display("A.m * B.m = %b", SQ_Temp);
203.                    $display("Result = %b", Result);
204.                    i <= i + 1'b1;
205.                end
206.
207.                22: // 0.0868 * 0.0868
208.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
209.                else begin A <= 32'b00111101101100011100010000110011; B <= 32'b00111101101100011100010000110011; Start_Sig <= 1'b1; end
210.
211.                23:
212.                begin
213.                    $display("A = %b and rA = %b", A,SQ_rA);
214.                    $display("B = %b and rB = %b", B,SQ_rB);
215.                    $display("A.m * B.m = %b", SQ_Temp);
216.                    $display("Result = %b", Result);
217.                    i <= i + 1'b1;
218.                end
219.
220.                24: // 0.0868 * 0.01085
221.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
222.                else begin A <= 32'b00111101101100011100010000110011; B <= 32'b00111100001100011100010000110011; Start_Sig <= 1'b1; end
223.
```

```
224.                    25:
225.                    begin
226.                            $display("A = %b and rA = %b", A,SQ_rA);
227.                            $display("B = %b and rB = %b", B,SQ_rB);
228.                            $display("A.m * B.m = %b", SQ_Temp);
229.                            $display("Result = %b", Result);
230.                            i <= i + 1'b1;
231.                    end
232.
233.                    26: // 0.0078125 * 0.0057488
234.                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
235.                    else begin A <= 32'b00111100000000000000000000000000; B <= 32'b00111011101111000110000011010001; Start_Sig <= 1'b1; end
236.
237.                    27:
238.                    begin
239.                            $display("A = %b and rA = %b", A,SQ_rA);
240.                            $display("B = %b and rB = %b", B,SQ_rB);
241.                            $display("A.m * B.m = %b", SQ_Temp);
242.                            $display("Result = %b", Result);
243.                            i <= i + 1'b1;
244.                    end
245.
246.                      28: // 3.1828 * -0.0063
247.                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
248.                    else begin A <= 32'b01000000010010111011001011111111; B <= 32'b10111011110011100111000000111011; Start_Sig <= 1'b1; end
249.
250.                    29:
251.                    begin
252.                            $display("A = %b and rA = %b", A,SQ_rA);
253.                            $display("B = %b and rB = %b", B,SQ_rB);
254.                            $display("A.m * B.m = %b", SQ_Temp);
255.                            $display("Result = %b", Result);
256.                            i <= i + 1'b1;
257.                    end
258.
259.                    30: // 6.631 * 0.25
260.                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
261.                    else begin A <= 32'b01000000110101000011000100100111; B <= 32'b00111110100000000000000000000000; Start_Sig <= 1'b1; end
262.
263.                    31:
264.                    begin
265.                            $display("A = %b and rA = %b", A,SQ_rA);
266.                            $display("B = %b and rB = %b", B,SQ_rB);
267.                            $display("A.m * B.m = %b", SQ_Temp);
268.                            $display("Result = %b", Result);
```

```
269.                    i <= i + 1'b1;
270.                end
271.
272.                32: // 6.3565 * -0.0063
273.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
274.                  else begin A <= 32'b01000000110010110110100001110011; B <= 32'b10111011110011100111000000111011; Start_Sig <= 1'b1; end
275.
276.                33:
277.                begin
278.                    $display("A = %b and rA = %b", A,SQ_rA);
279.                    $display("B = %b and rB = %b", B,SQ_rB);
280.                    $display("A.m * B.m = %b", SQ_Temp);
281.                    $display("Result = %b", Result);
282.                    i <= i + 1'b1;
283.                end
284.
285.                34: // 273.757 * 483.265
286.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
287.                else begin A <= 32'b01000011100010001110000011100101; B <= 32'b01000011111100011010000111101100; Start_Sig <= 1'b1; end
288.
289.                35:
290.                begin
291.                    $display("A = %b and rA = %b", A,SQ_rA);
292.                    $display("B = %b and rB = %b", B,SQ_rB);
293.                    $display("A.m * B.m = %b", SQ_Temp);
294.                    $display("Result = %b", Result);
295.                    i <= i + 1'b1;
296.                end
297.
298.                36: // 3.1828 * 2
299.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
300.                else begin A <= 32'b01000000010010111011001011111111; B <= 32'b01000000000000000000000000000000; Start_Sig <= 1'b1; end
301.
302.                37:
303.                begin
304.                    $display("A = %b and rA = %b", A,SQ_rA);
305.                    $display("B = %b and rB = %b", B,SQ_rB);
306.                    $display("A.m * B.m = %b", SQ_Temp);
307.                    $display("Result = %b", Result);
308.                    i <= i + 1'b1;
309.                end
310.
311.                48:
312.                i <= i;
313.
```

```
314.                    endcase
315.
316.  endmodule
```

读者千万不要被 float_multi_module.vt 激励文本的长度被吓到，笔者在验证浮点乘法模块的时候，不知觉中就写到那么长了，撤了有觉得可惜，所以就保留了下来。激励文本的风格没有多大变化，不过类似第 69-75 行的步骤会常常出现，主要是打印恢复以后的操作数 rA 和 rB，随之打印相乘的结果，最后打印单精度格式的结果。

仿真结果：

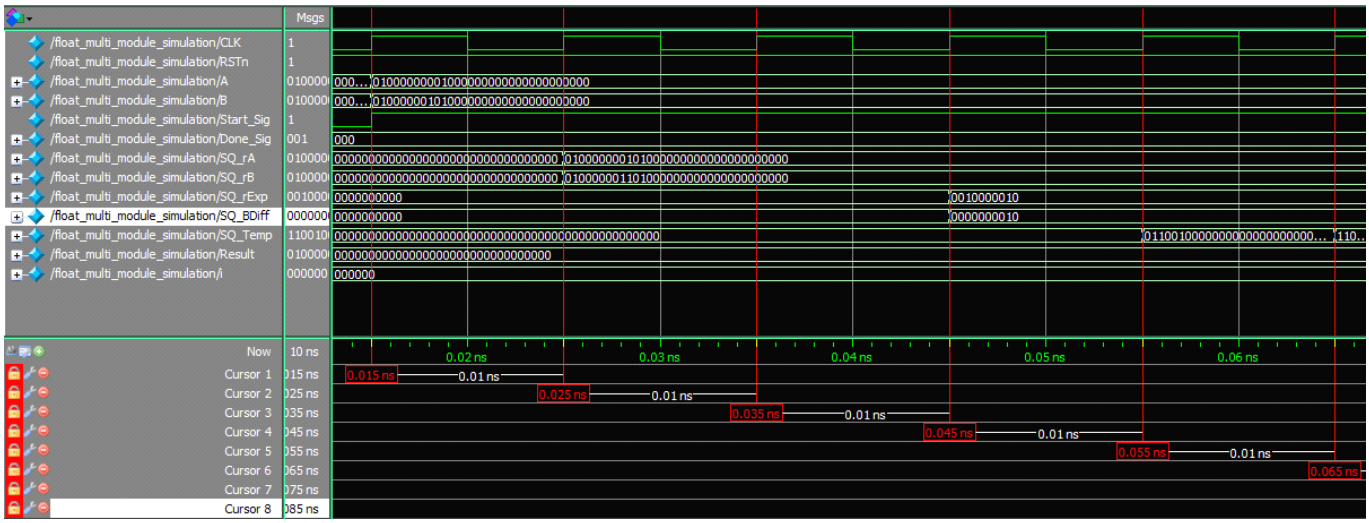


图 4.9.2 仿真过程一。

图 4.9.2 中 C1 是指向乘法模块的启动开始。在 C2 的时候是操作数预处理和符号位运算，该模块寄存并且恢复操作数 A 和 B，所以在 C2 的未来输出恢复以后的 rA 和 rB，与此同时也取得符号位结果。在 C3 的时候是移位预处理，此时特殊条件不成立，因此操作数 rB 没有发生移位预处理。在 C4 的时候是价码运算，所以在 C4 的未来输出价码结果。在 C5 的时候是尾数运算，rA 和 rB 的尾数决定相乘，所以在 C5 的未来输出相乘结果。

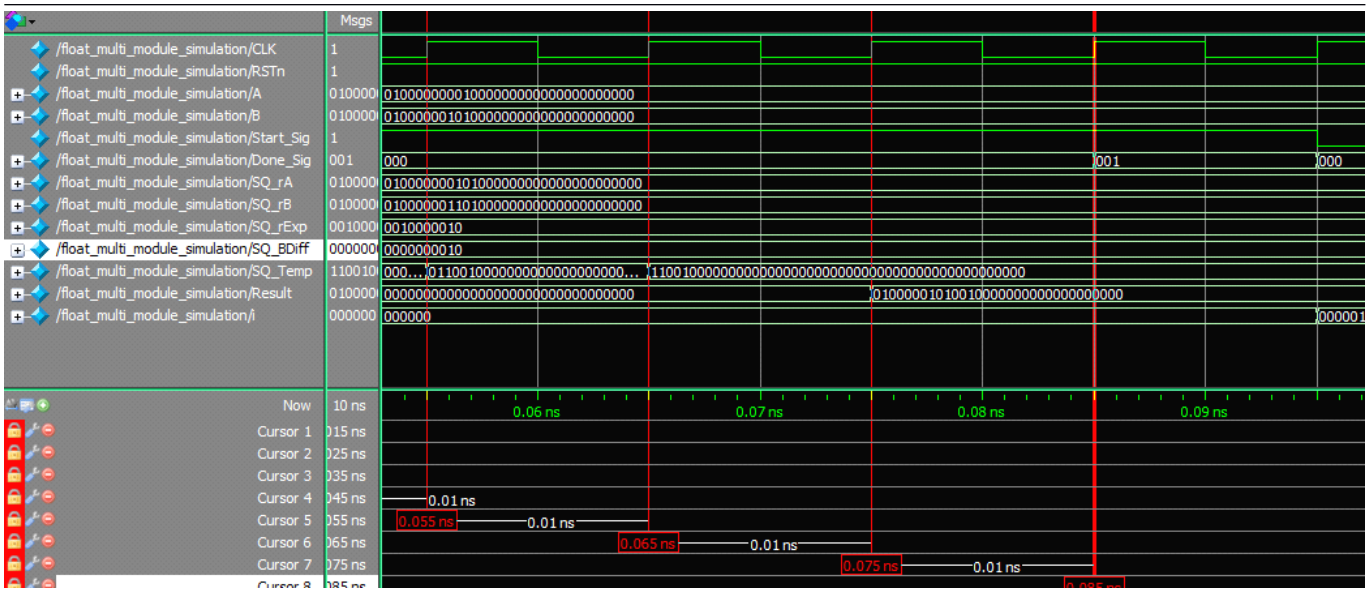


图 4.9.3 仿真过程 2。

在 C6 的时候是结果调整，此时 Temp[47]的过去值为 0，所以在 C6 的未来输出 Temp 右移一位以后的结果。在 C7 的时候是输出和格式化，各种错误反馈没有发生，最后在 C7 的未来输出单精度格式的结果。以上的仿真过程是没有处罚特殊条件的情形，接下来我们来看看相反情形的仿真过程。

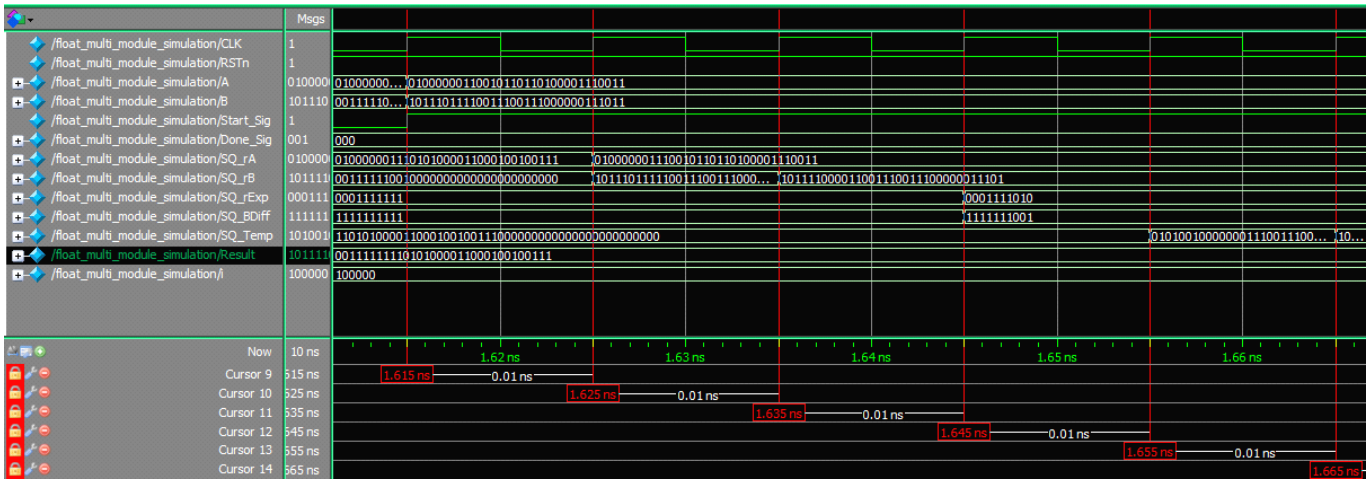


图 4.9.4 仿真过程 3。

图 4.9.4 是测试操作数组符号位为不一样的情况，亦即特殊条件之一的情况。C9 指向该模块使能的开始。C10 是操作数预处理和符号位运算，所以在 C10 的未来输出操作数预处理以后的结果，同时也取得符号位运算的结果。C11 是移位预处理，这时候特殊条件成立了，所以在 C11 的未来输出右移一位和价码加 1——操作数 B 的结果。

C12 是价码运算，所以在 C12 的未来输出运算以后的价码结果。C13 是尾数运算，所以在 C13 的未来输出相乘结果。

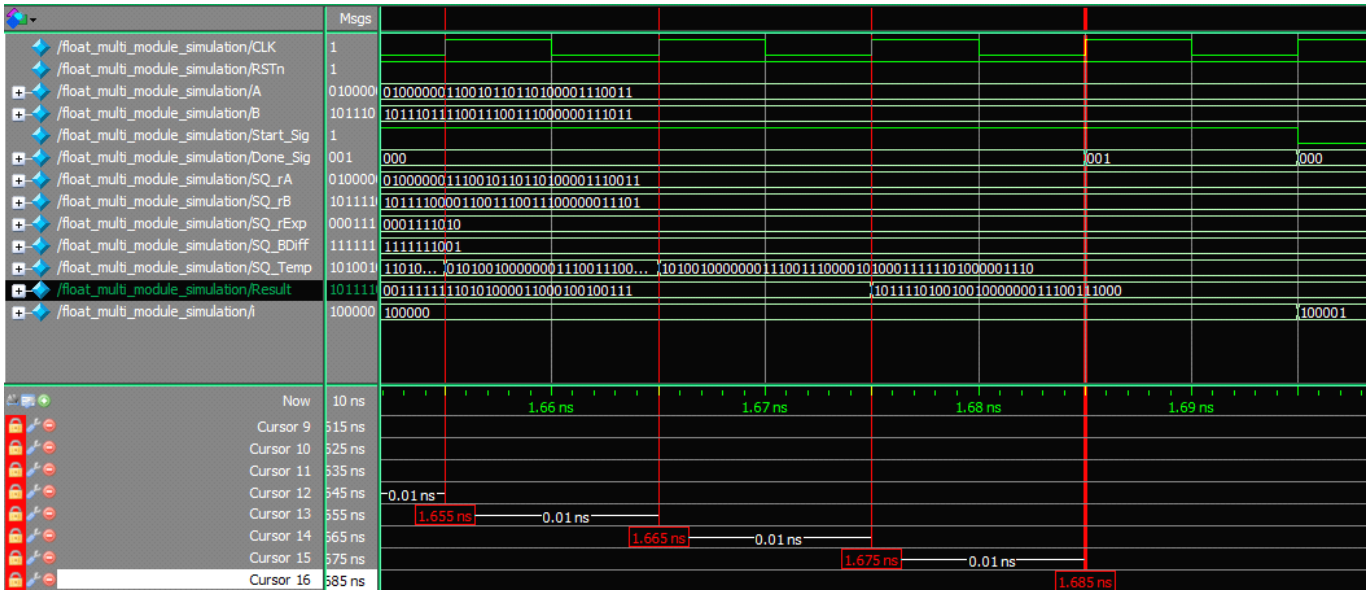


图 4.9.5 仿真过程 4。

在 C14 的时候是结果调整，根据 Temp 的过去值 Temp[47]为 0，所以在 C14 的未来输出右移一位后的 Temp。C15 是输出和格式化，任何错误反馈没有发生，所以在 C15 的未来输出单精度格式的结果。有发生移位预处理和无发生移位预处理是根据特殊状况/条件而定，不过在运算过程中变化不是很大。

这个实验的激励过程好长好长，所以笔者就不将仿真过程一一贴出来，读者自己看着办吧，作为补偿笔者很乐意将所以验证结果贴处理。

Q1: A = 2.5 , B = 5 , A×B = ?

```
A = 32'b0_10000000_01000000000000000000000

B = 32'b0_10000001_01000000000000000000000


32'b0_10000010_10010000000000000000000 Test Result 12.5

32'b0_10000010_10010000000000000000000 Real Result 12.5

                                       True Result 12.5
```

Q1 这题很简单，没有什么好说的。

Q2: A = 3.142 , B = 3.142 , A×B = ?

```
A = 32'b0_10000000_10010010001011010000111

B = 32'b0_10000000_10010010001011010000111


32'b0_10000001_00111011111010001100010 Test Result 9.8721637

32'b0_10000001_00111011111010001100010 Real Result 9.8721637

                                       True Result 9.872164
```

对于十进制操作组的在 4 位以上的时候（包括小数），单精度浮点数的相乘结果显得有点勉强了。此外，Q2 是特殊条件之一。

Q3: A = 12.662 , B = 4.903 , A×B = ？

```
A = 32'b0_10000010_10010101001011110001101
B = 32'b0_10000001_00111001110010101100000

32'b0_10000100_11110000101001110111111 Test Result 62.0817832
32'b0_10000100_11110000101001111000000 Real Result 62.0817871
                                       True Result 62.081786
```

十进制操作组位数不一致，一个是 5 位（包含小数），另一位是 4 位（包含小数）得到的单精度结果还是可以接受。

Q4: A = 2 , B = 2 , A×B = ？

```
A = 32'b0_10000000_00000000000000000000000
B = 32'b0_10000000_00000000000000000000000

32'b0_10000001_00000000000000000000000 Test Result 4
32'b0_10000001_00000000000000000000000 Real Result 4
                                       True Result 4
```

Q5: A = 73.676 , B = 83.266 , A×B = ？

```
A = 32'b0_10000101_00100111000100010110100
B = 32'b0_10000101_01001101000100000110001

32'b0_10001011_01111111111001001000011 Test Result 6142.2827148
32'b0_10001011_01111111111001001000100 Real Result 6142.2832031
                                       True Result 6142.283022
```

对于十进制的操作数拥有 5 位以上（包括小数），单精度格式在精度方面已经显得有点承受不了了。此外，Q5 是特殊条件之一。

Q6: A = 83.266 , B = 83.266 , A×B = ？

```
A = 32'b0_10000101_01001101000100000110001
B = 32'b0_10000101_01001101000100000110001

32'b0_10001011_10110001010100111001111 Test Result 6933.2260742
32'b0_10001011_10110001010100111010000 Real Result 6933.2265625
```

```
                                        True Result 6933.226756
```

Q6 也是特殊条件之一。

Q7: A = 1024 , B = 256 , A×B = ？

```
A = 32'b0_10001001_00000000000000000000000
B = 32'b0_10000111_00000000000000000000000

32'b0_10010001_00000000000000000000000 Test Result 262144
32'b0_10010001_00000000000000000000000 Real Result 262144
                                        True Result 262144
```

Q8: A = 33.116 , B = 16.558 , A×B = ？

```
A = 32'b0_10000100_00001000111011011001001
B = 32'b0_10000011_00001000111011011001001

32'b0_10001000_0001001000101010110110 1 Test Result 548.3347778
32'b0_10001000_0001001000101010110110 0 Real Result 548.3347167
                                        True Result 548.334728
```

Q8 也是特殊条件之一。

Q9: A = 0.00416 , B = 0.00014 , A×B = ？

```
A = 32'b0_01110111_00010000101000010011100
B = 32'b0_01110010_00100101100110011110111

32'b0_01101010_00111000101011000111001 Test Result 0.0000005000
32'b0_01101010_00111000101011000111001 Real Result 0.0000005000
                                        True Result 0.0000005824
```

对于有 5 位十进制的小数来说，得到的单精度结果，精度已经越来越偏离了，或者说已经超过单精度格式可以承受的程度。

Q10: A = 0.125 , B = 0.125 , A×B = ？

```
A = 32'b0_01111100_00000000000000000000000
B = 32'b0_01111100_00000000000000000000000

32'b0_01111001_00000000000000000000000 Test Result 0.015625
32'b0_01111001_00000000000000000000000 Real Result 0.015625
                                        True Result 0.015625
```

Q11: A = 0.0868 , B = 0.0868 , A×B = ？

```
A = 32'b0_01111011_01100011100010000110011
B = 32'b0_01111011_01100011100010000110011

32'b0_01110111_11101101110000111001000 Test Result 0.0009417
32'b0_01110111_11101101110000111001001 Real Result 0.0009417
                                       True Result 0.00094178
```

仅有 4 位的十进制小数，得到的结果还可以接受。Q11 也是特殊条件之一。

Q12: A = 0.0078125 , B = 0.0057488 , A×B = ？

```
A = 32'b0_01111000_00000000000000000000000
B = 32'b0_01110111_01111000110000011010001

32'b0_01110000_01111000110000011010001 Test Result 0.0000449
32'b0_01110000_01111000110000001101110 Real Result 0.0000449
                                       True Result 0.0000449125
```

Q12 是非常挑剔的操作组，得到的结果 无论是 Real Result 还是 Test Result，它们都已经是最大极限了。

Q13： A = 3.1828 , B = -0.0063 , A×B = ？

```
A = 32'b0_10000000_10010111011001011111111
B = 32'b1_01110111_10011100111000000111011

32'b1_01111001_01001000100001101010110 Test Result -0.0200516
32'b1_01111001_01001000100001101010110 Real Result -0.0200516
                                       True Result -0.02005164
```

Q13 是特殊条件之一。

Q14: A = 6.631 , B = 0.25 , A×B = ？

```
A = 32'b0_10000001_10101000011000100100111
B = 32'b0_01111101_00000000000000000000000

32'b0_01111111_10101000011000100100111 Test Result 1.6577500
32'b0_01111111_10101000011000100100111 Real Result 1.6577500
                                       True Result 1.65775
```

Q15: A = 6.3565 , B = -0.0063 , A×B = ？

```
A = 32'b0_10000001_10010110110100001110011
B = 32'b1_01110111_10011100111000000111011

32'b1_01111010_01001000000011100111000 Test Result -0.0400459
32'b1_01111010_01001000000011100111001 Real Result -0.0400459
                                       True Result -0.04004595
```

Q16: A = 273.757 , B = 483.265 , A×B = ？

```
A = 32'b0_10000111_00010001110000011100101
B = 32'b0_10000111_11100011010000111101100

32'b0_10010000_00000010011001001001011 Test Result 132297.1718750
32'b0_01110000_01111000110000001101110 Real Result 132297.1718750
                                       True Result 132297.176605
```

Q16 同样也是挑剔的操作数，操作组都拥有 6 位十进制（包括小数），得到的结果，精度偏离的程度只能用掉眼泪来形容。Q16 是特殊条件之一。

Q17: A = 3.1828 , B = 2 , A×B = ？

```
A = 32'b0_10000000_10010111011001011111111
B = 32'b0_10000000_00000000000000000000000

32'b0_10000001_10010111011001011111111 Test Result 6.3656001
32'b0_10000001_10010111011001011111111 Real Result 6.3656001
                                       True Result 6.3656
```

Q17 没有什么特别，操作组价码同是，但是其中一个操作数的尾数（不包含隐藏位）是全零，所以不引起移位预处理。最后得到的 Test Result 不但接近 Real Result，而且也和 True Result 的偏度不大。

==========

呼，先松一口气 ... 在 17 组的操作组中，我们都可以晓得所有 Test Result 都非常接近 Real Result 和 True Result，不过根据操作数的十进制位数，精度也会有所不同。在没有包括小数的情况下，十进制位数多少都不会严重影响结果，反之在包含小数的情况下，唯有 3 位十进制小数才是单精度格式可以发挥最佳的状态。

最后我们可以简单的总结一下，浮点数乘法器在理论或者概念上，它显得非常容易设，这是许多人最容易大意的现象。实际上浮点数乘法器存在 2 个隐蔽的陷阱，稍微不注意问题就大条了。因此，隐蔽的陷阱结果提升了设计的难度。最后说几句讽刺的话，很多参考书都不把这些问题（浮点乘法的陷阱）考虑进去，可见参考书的态度是非常随便。

4.10 浮点除法和精度流失

浮点数除法运算比起，加减和相乘运算可以说是最简单的一个。浮点除法在数学的形式上和浮点乘法的运算过程差不多，不过是反着方向而已。笔者举个简单的例子：

$$
\begin{aligned}
12 \times 10^4 / -3 \times 10^2 &= (1)12 \times 10^4 \ / (-1)3 \times 10^2 \\
&= (1)(-1)\ 12/3 \times 10^{4-2} \\
&= (-1)\ 4 \times 10^2 \\
&= -4 \times 10^2
\end{aligned}
$$

浮点除法可以把 符号运算，价码运算和尾数运算分开 3 个独立运算来执行。在此有些同学可能会担心浮点除法和浮点乘法那样存在陷阱 ... 嗯，实际上浮点除法也存在陷阱，不过比起浮点乘法的陷阱，阴险度倒是低了不少。那么浮点除法的陷阱是什么，笔者先卖个关子，在往后的讨论中，那时候笔者再说。

不过在执行除法运算的时候，我们常常会碰上典型的现象？我们来看这样一个例子：

$A = 24 \times 10^3$, $B = 30 \times 10^4$

$$
\begin{aligned}
24 \times 10^3 / 30 \times 10^4 &= 24 \times 10^3 / 3 \times 10^5 \\
&= 24 / 3 \times 10^{3-5} \\
&= 8 \times 10^{-2} \\
&= 0.08
\end{aligned}
$$

$$
\begin{aligned}
24 \times 10^3 / 30 \times 10^4 &= 24 \times 10^3 / 0.3 \times 10^6 \\
&= 24 / 0.3 \times 10^{3-6} \\
&= 80 \times 10^{-3} \\
&= 0.08
\end{aligned}
$$

$$
\begin{aligned}
24 \times 10^3 / 30 \times 10^4 &= 24 \times 10^3 / 0.03 \times 10^7 \\
&= 24 / 0.03 \times 10^{3-7} \\
&= 800 \times 10^{-4} \\
&= 0.08
\end{aligned}
$$

从上面的例子中，如果操作数 A 与 B 要相除，其中操作数 B 的尾数大于操作数 A 的尾数，如果真的发生相除的话，那么结果一定会产生余数。对于浮点除法来说，相除以后的余数如果越大，那么商数的精度就会下降。因此在浮点除法里，操作数 A 的尾数永远必须必操作数 B 的尾数大。所以，我们会利用小数点的移位，将操作数 A 的尾数变大。

那么问题来了，在 A 的尾数变大其中，我们没有一个规定尾数要变多大？浮点数左移 1 位，2 位还是 3 位呢？。对于在书写形式上的十进制来说，操作数 A 和 B 是没有固定的尾数，因此小数点位移的范围是非常的大。不过相反的，在二进制的世界里，浮点数可以套入在某个格式中，就以单精度格式为话题。

我们知道单精度格式的尾数，包括隐藏位一共是 24 位，因此只要把操作数 A 的尾数左移 24 位，无论是什么样的操作组，我们都可以断定操作数 A 的尾数一定大过操作数 B 的尾数。笔者举个具体一点的例子：

```
A = 24'b100000000000000000000001
B = 24'b100000000000000000000000

A/B = Quotient = 1'b1 , Reminder   = 1'b1
```

在上述的例子中，操作数 A 比起操作数 B 大，在 A 和 B 相除以后，我们会得到商数为 1，余数为 1 的结果。对于浮点数来说，余数的存在某个程度上就是精度流失的现象。因此我们必须把 A 变大，预防余数的产生。

```
A = 48'b100000000000000000000001000000000000000000000000E-24
B = 48'b000000000000000000000000100000000000000000000000E+0

A/B = Quotient = 25'b1000000000000000000000010 , Reminder  = 0
```

途中，我们把 A 左移 24 位然后和 B 相除，相除以后取得商数为 25'b1000000000000000000000010 余数为 0 的结果。在此我们的问题不是要怎样处理 A 移位以后的价码，而我们主要的任务就是竟可能消除余数而已。

接下来我们来看一个更具体的例子：

```
Q1: A = 10 , B = 3 , A / B = ?
```

```
A = 32'b0_10000010_01000000000000000000000
B = 32'b0_10000000_10000000000000000000000

操作数预处理：
A = 33'b0_10000010_101000000000000000000000
B = 32'b0_10000000_110000000000000000000000

运算：
sum.s = A.s ^ B.s = 0 ^ 0 = 0

sum.exp = A.exp - B.exp
        = 0b10000010 - ( 0b10000000 - 127 )
        = 0b10000010 - 1
        = 0b10000001

sum.m = A.m / B.m
      = 24'b101000000000000000000000 / 24'b110000000000000000000000
```

```
      = 0, Reminder is 1101010101010101 ...
输出和格式化：
$ 由于求得的商值为 0，所以结果反馈尾数零值的错误。
```

在 Q1 的运算中，由于操作数 A 的尾数小于操作数 B 的尾数，所以相除之后所求的商数为 0，因此也造成尾数零值的错误发生，为此我们需要把操作数的尾数加大，我们把 A 的尾数加大一点点再来运算 Q1 看看：

```
Q1b: A = 10，B = 3，A / B = ?
```

```
A = 32'b0_10000010_01000000000000000000000
B = 32'b0_10000000_10000000000000000000000

操作数预处理：
A = 33'b0_10000010_101000000000000000000000
B = 32'b0_10000000_110000000000000000000000

运算：
sum.s = A.s ^ B.s = 0 ^ 0 = 0

sum.exp = A.exp - B.exp
        = 0b10000010 - ( 0b10000000 - 127 )
        = 0b10000010 - 1
        = 0b10000001

sum.m = A.m / B.m
      = { A.m, 4'd0 } / { 4'd0, B.m }
      = 28'b1010000000000000000000000000 / 28'b0000110000000000000000000000
      = 28'b0000000000000000000000001101, reminder is 0101010101...

结果调整：
$ sum.m[3] is 1, while sum.m << 24 and sum.exp -1

sum'.exp = 0b10000001 - 1 = 0b10000000

h m
0 0000000000000000000000001101 before
1 1010000000000000000000000000 << 24
1 1010000000000000000000000000 after, sum = sum's

输出和格式化：
32'b0_10000000_10100000000000000000000 Test Result 3.2500000
32'b0_10000000_10101010101010101010101 Real Result 3.3333332
                                       True Result 3.333...
```

Q1b 中的 Test Result 距离 Real Result 虽然已经接近了，但是精度的流失极为严重。不管了，我们把马力加到最大，尽可以把操作数 A 的尾数加到最大，然后在一次运算 Q1 看看。

```
Q1C: A = 10 , B = 3 , A / B = ?
```

```
A = 32'b0_10000010_01000000000000000000000
B = 32'b0_10000000_10000000000000000000000

操作数预处理：
A = 33'b0_10000010_101000000000000000000000
B = 32'b0_10000000_110000000000000000000000

运算：
sum.s = A.s ^ B.s = 0 ^ 0 = 0

sum.exp = A.exp - B.exp
        = 0b10000010 - ( 0b10000000 - 127 )
        = 0b10000010 - 1
        = 0b10000001

sum.m = A.m / B.m
      = { A.m, 24'd0 } / { 24'd0, B.m }
      = 48'b101000000000000000000000000000000000000000000000 / 48'b000000000000000000000000110000000000000000000000
      = 48'b000000000000000000000000110101010101010101010101, reminder is 0101010101...

结果调整：
$ sum.m[23] is 1, while sum.m << 24 and sum.exp -1

sum'.exp = 0b10000001 - 1 = 0b10000000

h m
0 000000000000000000000000110101010101010101010101 before
1 101010101010101010101010000000000000000000000000 << 24
1 101010101010101010101010000000000000000000000000 after, sum = sum's

输出和格式化：
32'b0_10000000_10101010101010101010101 Test Result 3.3333332
32'b0_10000000_10101010101010101010101 Real Result 3.3333332
                                       True Result 3.333...
```

Q1c 的预算结果中，我们得到的 Test Result 和 Real Result 一模一样，所以结果是正确的。在这里，估计许多同学都看到头冒金星，完全看不懂运算细节。同学们别着急，向把运算的细节放在一边去。为此笔者只是想证明一件事就是，尽可以减少余数的产生，虽然 10/3 可能有算不完的余数，但是

我们要尽最大的努力和最大的可能避免余数过多而导致精度流失的问题。

如 Q1b 的结果，操作数 A 的马力只开到 4 位档而已，所以造成 Test Result 严重的精度流失。反之在 Q1c 的运算结果中，操作数 A 的马力开到最大，亦即 24 位档，最终 Test Result 的结果已经接近 Real Result。这并不是所谓精度的流失没有发生，而是压缩到最低的程度而已。事实上 24 位大的马力不是最大档，而马力是没有上限的，马力越大精度的流失可以压缩越低，作为代价逻辑资源的消耗就会越高。不过对于单精度可是来说，24 位档的马力已经足够了，多了就是浪费。

==========

根据 Q1c 的运算过程，浮点除法的大概步骤如下所示：

（一）操作数预处理：恢复操作数 A 和 B；
（二）运算：符号位运算，价码运算和尾数运算独立运行，尾数运算中可以考虑马力的档位；
（三）结果调整：调整运算结果适合浮点数源的格式；
（四）输出和格式化：检测任何错误信息，执行四舍五入操作和输出结果。

步骤一：操作数预处理

浮点除法和浮点乘法的操作数预处理步骤，内容没有变化。该步骤的目的就是恢复隐藏在操作数中的符号位。Verilog HDL 语言的实现如代码 4.10.1 所示：

```
begin
    rA <= { A[31], A[30:23], 1'b1, A[22:0] };
    rB <= { B[31], B[30:23], 1'b1, B[22:0] };
end
```

代码 4.10.1 操作数预处理。

步骤二：运算

浮点除法比起浮点乘法所存在的陷阱，由于阴险程度一点也比不上，所以符号位，价码和尾数可以独立运算。至于 Verilog HDL 语言的实现如代码 4.10.2 所示。在代码 4.10. 2 所示中操作数 A 的尾数左移 24 位，对于单精度格式的来说这种程度的马力已经足够满足运算。

```
begin
      isSign <= A[31] ^ B[31];

      BDiff = rB[31:24] - 8'd127;
      rExp <= rA[31:24] - BDiff;

      Temp <= { rA[23:0] , 24'd0} / { 24'd0 , rB[23:0] };
end
```

代码 4.10.2 运算。

步骤三：结果调整

还记得在浮点乘法中出现的陷阱之一吗？假设“假定”小数点介于 Temp[47]~Temp[46]之间，在结果调整中，如果 Temp[47] 为 1 结果调整没有发生；再如果 Temp[46]为 1，结果就要左移 1 位，可是价码没有变化；再再如果 Temp[45]为 1，结果就要左移 2 位，此时价码就要 -1。同样的陷阱也出现在浮点除法之中，不过只是换了一件衣服而已。笔者举个例子：

假设操作数 A 和 B 均为 4 位，操作数 A 最佳的马力档位是 4，那么相除结果一定是 8 位。然后假设“假定”小数点介于结果的 [7]~[6]之间。

QI：	QII：	QIII：	QIV：
A = 4'b1111	A = 4'b1111	A = 4'b1000	A = 4'b1000
B = 4'b1000	B = 4'b1111	B = 4'b1000	B = 4'b1111
8'b11110000	8'b11110000	8'b10000000	8'b10000000
8'b00001000 ÷	8'b00001111 ÷	8'b00001000 ÷	8'b00001111 ÷
8'b00011110 before	8'b00010000 before	8'b00010000 before	8'b00001001 before
8'b11110000 << 3	8'b10000000 << 3	8'b10000000 << 3	8'b10010000 << 4 , exp -1
8'b11110000 after	8'b10000000 after	8'b10000000 after	8'b10010000 after

笔者列出四个最大的可能会出现的状况，在 QI 的操作组中，A 左移 4 位，然后在运算之后，得到的答案为 8'b00011110，亦即[4] 为 1。在结果调整中，左移 3 位价码不变；在 QII 的操作组中，执行同样的动作，得到的运算结果是 8'b00010000，亦即[4] 为 1。在结果调整中，左移 3 位价码不变；在 QIII 的运算结果中，取得 8'b00010000，亦即[4] 为 1，结果左移 3 位，价码不变；在最后的 QIV 运算中，求得结果为 8'b00001001，亦即[3]为 1，结果左移 4 位，然后价码递减 1。

如果操作数从 4 位变成 24 位，马力为 24 档位，然后根据 QI~QIV 之间的情况。那么，QI~QIII 的运算结果中，[24]一定为 1，然后结果左移 23 位。至于 QIV 的运算结果，[23]一定为 1，然后结果左移 24 位，价码随之减一。为了给结果调整买份保险，我们可以在 [24 ... 23] 的前后加入调整的范围。Verilog HDL 语言的实现如代码 4.10.3 所示：

```
begin
        if( Temp[25] == 1'b1 ) begin Temp <= Temp << 22; end
        else if( Temp[24] == 1'b1 ) begin Temp <= Temp << 23; end
        else if( Temp[23] == 1'b1 ) begin Temp <= Temp << 24; rExp <= rExp - 1'b1; end
end
```

代码 4.10.3 结果调整。

在代码 4.10.3 中，if(Temp[25] == 1'b1) 是一个保险措施，虽然笔者知道在运算之后，调整范围一定会发生在 Temp[24] ~ Temp[23] 之间。不过为了安心，笔者还是买下这份绝对安心保险（意思是不可能得到赔偿的保险）。

步骤四：输出和格式化

```
begin
        if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end
        else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end
          else if( Temp[47:24] == 24'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end
          else rResult <= { isSign, rExp[7:0], Temp[46:24] };
 end
```

代码 4.10.4 输出和格式化。

在输出和格式化这个步骤中，浮点乘法和浮点除法的内容基本上是大同小异，因为浮点除法和浮点乘法都一样，我们“假定”小数点介于 Temp[47]~Temp[46] 之间。不过有一点读者会吓到的是，浮点除法少了四舍五入这个操作，为什么呢？有两种原因，第一个原因是经过结果调整以后，Temp[23]~Temp[0] 很大可能是全零。另一个原因是，在尾数运算途中保护精度的措施已经最大发挥了，所以四舍五入操作显得有点鹌鹑，因此就被节了。

==========

到目前为止，读者的心是否还是忐忑？头脑是否迷糊中？这是理所当然的事情，因为浮点除法运算比起其他的运算，它虽然容易但是显得有点特殊。不过不要紧，只要我们简单的手动运算一下，有个感知的认识以后，这些“讨厌”的感觉自然而然会消失。

（注意在接下来的手动运算中，笔者会省略掉操作数预处理这个操作）

Q2：A = 6.32，B = 0.26，A / B = ?

```
A = 32'b0_10000001_10010100011110101110001
B = 32'b0_01111101_00001010001111010111000

运算：
sum.s = A.s ^ B.s = 0 ^ 0 = 0

sum.exp   = A.exp - B.exp
          = 0b10000001 - ( 0b01111101 - 127 )
          = 0b10000001 - ( -2 )
          = 0b10000011

sum.m     = A.m / B.m
          = { A.m, 24'd0 } / { 24'd0, B.m }
          = 48'b110010100011110101110001 ... 0 /  48'b0 ... 100001010001111010111000
          = 48'b000000000000000000000000110000100111011000101000 0

结果调整：
$ sum.m[24] is 1, while sum.m << 23
```

```
h m
0 00000000000000000000000110000100111011000101000 before
1 10000100111011000101000000000000000000000000000 << 23
1 10000100111011000101000000000000000000000000000 after, sum = sum'

输出和格式化：
32'b0_10000011_10000100111011000101000 Test Result 24.3076934
32'b0_10000011_10000100111011000100111 Real Result 24.3076915
                                       True Result 24.307692
```

Q3：A = 3.48 , B = 10 , A / B = ?

```
A = 32'b0_10000000_10111101011100001010010
B = 32'b0_10000010_01000000000000000000000

运算：
sum.s = A.s ^ B.s = 0 ^ 0 = 0

sum.exp   = A.exp - B.exp
          = 0b10000000 - ( 0b10000010 - 127 )
          = 0b10000000 - ( 3 )
          = 0b01111101

sum.m     = A.m / B.m
          = { A.m, 24'd0 } / { 24'd0, B.m }
          = 48'b101111010111000010100 10 ... 0 /  48'b0 ... 101000000000000000000000
          = 48'b000000000000000000000000101100100010110100001110 0

结果调整：
$ sum.m[24] is 1, while sum.m << 23

h m
0 000000000000000000000010110010001011010000111 00 before
1 011001000101101000011100000000000000000000000 00 << 23
1 011001000101101000011100000000000000000000000 00 after, sum = sum'

输出和格式化：
32'b0_01111101_01100100010110100001110 Test Result 0.3479999
32'b0_01111101_01100100010110100001110 Real Result 0.3479999
                                       True Result 0.348
```

Q4：A = 1.442 , B = 11.515 , A / B = ?

```
A = 32'b0_01111111_01110001001001101110101
B = 32'b0_10000010_01110000011110101110001

运算：
sum.s = A.s ^ B.s = 0 ^ 0 = 0

sum.exp   = A.exp - B.exp
          = 0b01111111 - ( 0b10000010 - 127 )
          = 0b01111111 - (  3 )
          = 0b01111100

sum.m     = A.m / B.m
          = { A.m, 24'd0 } / { 24'd0, B.m }
          = 48'b101110001001001101110101 ... 0 /  48'b0 ... 101110000011110101110001
          = 48'b000000000000000000000000100000000011101111000010 0

结果调整：
$ sum.m[24] is 1, while sum.m << 23

h m
0 000000000000000000000100000000011101111000010 0 before
1 000000000111011110000100000000000000000000000 << 23
1 000000000111011110000100000000000000000000000 after, sum = sum'

输出和格式化：
32'b0_01111100_00000000011101111000010 Test Result 0.1252279
32'b0_01111100_00000000011101111000010 Real Result 0.1252279
                                       True Result 0.12522796352...
```

Q5：A = 1000 , B = 0.5 , A / B = ?

```
A = 32'b0_10001000_11110100000000000000000
B = 32'b0_01111110_00000000000000000000000

运算：
sum.s = A.s ^ B.s = 0 ^ 0 = 0

sum.exp   = A.exp - B.exp
          = 0b10001000 - ( 0b01111110 - 127 )
          = 0b10001000 - (  -1 )
          = 0b10001001

sum.m     = A.m / B.m
          = { A.m, 24'd0 } / { 24'd0, B.m }
```

```
         = 48'b111110100000000000000000 ... 0 /  48'b0 ... 100000000000000000000000

         = 48'b000000000000000000000000111110100000000000000000

结果调整：
$ sum.m[24] is 1, while sum.m << 23

h m
0 000000000000000000000000111110100000000000000000 before
1 111101000000000000000000000000000000000000000000 << 23
1 111101000000000000000000000000000000000000000000 after, sum = sum'

输出和格式化：
32'b0_10001001_11110100000000000000000 Test Result 2000
32'b0_10001001_11110100000000000000000 Real Result 2000
                                       True Result 2000
```

Q6：A = 3.142857 , B = -0.723093 , A / B = ?

```
A = 32'b0_10000000_10010010010010010010010
B = 32'b1_01111110_01110010001110010011111

运算：
sum.s = A.s ^ B.s = 0 ^ 1 = 1

sum.exp   = A.exp - B.exp
          = 0b10000000 - ( 0b01111110 - 127 )
          = 0b10000000 - (  -1 )
          = 0b10000001

sum.m     = A.m / B.m
          = { A.m, 24'd0 } / { 24'd0, B.m }
          = 48'b110010010010010010010010 ... 0 /  48'b0 ... 101110010001110010011111
          = 48'b000000000000000000000000100010110001010111000110 0

结果调整：
$ sum.m[24] is 1, while sum.m << 23

h m
0 000000000000000000000000100010110001010111000110 0 before
1 000101100010101110001100000000000000000000000000 << 23
1 000101100010101110001100000000000000000000000000 after, sum = sum'

输出和格式化：
32'b1_10000001_00010110001010111000110 Test Result -4.3464078
```

```
32'b0_10001001_00010110001010111000110 Real Result -4.3464078
                                       True Result -4.3464077
```

经过 Q2~Q6 这五题的运算以后，读者有什么感想呢？从中我们可以看到从 Q2~Q6，操作数组内容越来越挑剔，不过取得的 Test Result 都非常接近 Real Result。基本上，浮点除法的运算过程就是这样而已，只要把基本概念搞清楚以后就没有什么好怕的。

4.11 实验二十三：浮点数除法模块

图 4.11.1 浮点数除法模块图形。

图 4.11.1 是实验二十三里要建立的浮点数除法模块。至于信号之间的定义，基本上和前几个实验二十样，没有什么号重复的。我们还是直接来看代码吧：

float_divide_module.v

```
1.    module float_divide_module
2.    (
3.        input CLK, RSTn,
4.        input [31:0]A,B,
5.        output [31:0]Result,
6.
7.        input Start_Sig,
8.        output [3:0]Done_Sig,
9.
10.       /*****************/
11.
12.       output [32:0]SQ_rA,SQ_rB,
13.       output [47:0]SQ_Temp,
14.       output [9:0]SQ_rExp,SQ_BDiff
15.
16.   );
```

第 3~8 行是该模块的输入输出声明，反之第 12~14 行是仿真用的输出。

```
17.     /****************************************/
18.
19.       reg [3:0]i;
20.       reg [32:0]rA,rB;   // [32]Sign, [31:24]Exponent, [23]Hidden Bit, [22:0]Mantissa
21.       reg [47:0]Temp;     // [47] Hidden-A [46:24] Mantissa-A , [23] Hidden-B [22:0]Mantissa-B
22.       reg [31:0]rResult;
23.       reg [9:0]rExp,BDiff;        //[9:8] Overflow or underflow check, [7:0] usuall exp.
24.       reg isSign;
```

```
25.     reg isOver;
26.     reg isUnder;
27.     reg isZero;
28.     reg isDone;
29.
30.     always @ ( posedge CLK or negedge RSTn )
31.         if( !RSTn )
32.             begin
33.                 i <= 4'd0;
34.                 rA <= 33'd0;
35.                 rB <= 33'd0;
36.                 Temp <= 48'd0;
37.                 rResult <= 32'd0;
38.                 rExp <= 10'd0;
39.                 BDiff <= 10'd0;
40.                 isOver <= 1'b0;
41.                 isUnder <= 1'b0;
42.                 isZero <= 1'b0;
43.                 isDone <= 1'b0;
44.             end
```

第 19~28 行是相关的寄存器声明，功能上和浮点数乘法模块大同小异，不过 Temp 是寄存相除以后的结果。第 33~43 行是复位动作，所有寄存器都清零。

```
45.         else if( Start_Sig )
46.             case( i )
47.
48.                 0: // Initial and resulted out Signature
49.                 begin
50.                     rA <= { A[31], A[30:23], 1'b1, A[22:0] };
51.                     rB <= { B[31], B[30:23], 1'b1, B[22:0] };
52.
53.                     isSign <= A[31] ^ B[31];
54.                     isOver <= 1'b0; isUnder <= 1'b0; isZero <= 1'b0;
55.                     i <= i + 1'b1;
56.                 end
57.
58.                 1: // if rExp[9..8] is 1, mean A.Exp small than B.Exp
59.                    // while rExp[9..8] is 0, mean A.Exp large than B.Exp or same.
60.                 begin
61.                     BDiff = rB[31:24] - 8'd127; // + (~8'd127 + 1)
62.                     rExp <= rA[31:24] - BDiff; // A.Exp - B.Exp
63.                     i <= i + 1'b1;
64.                 end
```

```
65.
66.                 2: // Divide and A.m << 24
67.                 begin
68.                     Temp <= { rA[23:0] , 24'd0} / { 24'd0 , rB[23:0] };
69.                     i <= i + 1'b1;
70.                 end
```

第 49~56 行是操作数预处理和符号位运算；第 58~64 行是价码运算；第 66~70 行是尾数运算。虽然说，符号位运算，价码运算和尾数运算等操作都可以压缩在同一个步骤里运行，但是为了统一和前面几个实验二十样的风格，因此笔者免了。上述的操作基本上和 4.10 小节里解释过的一样，在此笔者就不重复了。

```
71.
72.                 3: // Check M'hidden bit
73.                 begin
74.                     if( Temp[25] == 1'b1 ) begin Temp <= Temp << 22; end
75.                     else if( Temp[24] == 1'b1 ) begin Temp <= Temp << 23; end
76.                     else if( Temp[23] == 1'b1 ) begin Temp <= Temp << 24; rExp <= rExp - 1'b1; end
77.
78.                     i <= i + 1'b1;
79.                 end
80.
81.                 4: //error check and decide final result
82.                 begin
83.                     if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end       // E Overflow
84.                     else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // E Underflow
85.                     else if( Temp[47:24] == 24'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // M Zero
86.                     else rResult <= { isSign, rExp[7:0], Temp[46:24] }; // okay without normalise
87.                     i <= i + 1'b1;
88.                 end
```

第 72~79 行是结果调整，然而第 81~88 行是输出和格式化。同样，笔者也不重复解释了。

```
89.
90.                 5:
91.                 begin isDone <= 1'b1; i <= i + 1'b1; end
92.
93.                 6:
94.                 begin isDone <= 1'b0; i <= 4'd0; end
95.
96.             endcase
97.
98.     /*************************************/
99.
```

```
100.        assign Done_Sig = { isOver, isUnder, isZero, isDone };
101.        assign Result = rResult;
102.
103.        /****************************************/
104.
105.          assign SQ_rA = rA;
106.          assign SQ_rB = rB;
107.          assign SQ_rExp = rExp;
108.          assign SQ_BDiff = BDiff;
109.          assign SQ_Temp = Temp;
110.
111.        /****************************************/
112.
113.  endmodule
```

余下的步骤 5~6 是产生完成信号，至于第 100~101 行是相关的输出驱动。最后第 105~109 行是仿真用的相关输出驱动。

float_divide_module.vt

```
1.    `timescale 1 ps/ 1 ps
2.    module float_divide_module_simulation();
3.
4.          reg CLK;
5.          reg RSTn;
6.
7.          reg [31:0] A;
8.          reg [31:0] B;
9.          reg Start_Sig;
10.
11.         wire [3:0]   Done_Sig;
12.         wire [31:0]   Result;
13.
14.         /**************************/
15.
16.         wire [9:0]   SQ_BDiff;
17.         wire [47:0]   SQ_Temp;
18.         wire [32:0]   SQ_rA;
19.         wire [32:0]   SQ_rB;
20.         wire [9:0]   SQ_rExp;
21.
22.         /**************************/
23.
24.         float_divide_module U1
```

```
25.         (
26.             .CLK(CLK),
27.              .RSTn(RSTn),
28.              .A(A),
29.              .B(B),
30.              .Result(Result),
31.              .Start_Sig(Start_Sig),
32.              .Done_Sig(Done_Sig),
33.              /************************/
34.              .SQ_BDiff(SQ_BDiff),
35.              .SQ_Temp(SQ_Temp),
36.              .SQ_rA(SQ_rA),
37.              .SQ_rB(SQ_rB),
38.              .SQ_rExp(SQ_rExp)
39.       );
40.
41.     /*****************************/
42.
43.     initial
44.       begin
45.              RSTn = 0; #10 RSTn = 1;
46.              CLK = 0; forever #5 CLK = ~CLK;
47.       end
48.
49.     /*****************************/
50.
51.     reg [5:0]i;
52.
53.       always @ (    posedge CLK or negedge RSTn )
54.              if( !RSTn )
55.                   begin
56.                     A <= 32'd0;
57.                     B <= 32'd0;
58.                     Start_Sig <= 1'b0;
59.                     i <= 6'd0;
60.                 end
61.           else
62.                 case( i )
63.
64.                         0: // 3.142 / 3.142
65.                         if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
66.                       else begin A <= 32'b01000000010010010001011010000111; B <= 32'b01000000010010010001011010000111; Start_Sig <= 1'b1; end
67.
68.                         1:
69.                         begin
```

```
70.                     $display("A = %b and rA = %b", A,SQ_rA);
71.                     $display("B = %b and rB = %b", B,SQ_rB);
72.                     $display("A.m / B.m = %b", SQ_Temp);
73.                     $display("Result = %b", Result);
74.                     i <= i + 1'b1;
75.                 end
76.
77.                 2: // 10 / 3
78.                 if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
79.                 else begin A <= 32'b01000001001000000000000000000000; B <= 32'b01000000010000000000000000000000; Start_Sig <= 1'b1; end
80.
81.                 3:
82.                 begin
83.                     $display("A = %b and rA = %b", A,SQ_rA);
84.                     $display("B = %b and rB = %b", B,SQ_rB);
85.                     $display("A.m / B.m = %b", SQ_Temp);
86.                     $display("Result = %b", Result);
87.                     i <= i + 1'b1;
88.                 end
89.
90.                 4: // 6.32 / 0.26
91.                 if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
92.                 else begin A <= 32'b01000000110010100011110101110001; B <= 32'b00111110100001010001111010111000; Start_Sig <= 1'b1; end
93.
94.                 5:
95.                 begin
96.                     $display("A = %b and rA = %b", A,SQ_rA);
97.                     $display("B = %b and rB = %b", B,SQ_rB);
98.                     $display("A.m / B.m = %b", SQ_Temp);
99.                     $display("Result = %b", Result);
100.                    i <= i + 1'b1;
101.                end
102.
103.                6: // 3.48 / 10
104.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
105.                else begin A <= 32'b01000000010111101011100001010010; B <= 32'b01000001001000000000000000000000; Start_Sig <= 1'b1; end
106.
107.                7:
108.                begin
109.                    $display("A = %b and rA = %b", A,SQ_rA);
110.                    $display("B = %b and rB = %b", B,SQ_rB);
111.                    $display("A.m / B.m = %b", SQ_Temp);
112.                    $display("Result = %b", Result);
113.                    i <= i + 1'b1;
114.                end
```

```
115.
116.                    8: // 1.442 / 11.515
117.                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
118.                    else begin A <= 32'b00111111101110001001001101110101; B <= 32'b01000001001110000011110101110001; Start_Sig <= 1'b1; end
119.
120.                    9:
121.                    begin
122.                            $display("A = %b and rA = %b", A,SQ_rA);
123.                            $display("B = %b and rB = %b", B,SQ_rB);
124.                            $display("A.m / B.m = %b", SQ_Temp);
125.                            $display("Result = %b", Result);
126.                            i <= i + 1'b1;
127.                    end
128.
129.                    10: // 1000 / 0.5
130.                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
131.                  else begin A <= 32'b01000100011110100000000000000000; B <= 32'b0_01111110_00000000000000000000000; Start_Sig <= 1'b1; end
132.
133.                    11:
134.                    begin
135.                            $display("A = %b and rA = %b", A,SQ_rA);
136.                            $display("B = %b and rB = %b", B,SQ_rB);
137.                            $display("A.m /B.m = %b", SQ_Temp);
138.                            $display("Result = %b", Result);
139.                            i <= i + 1'b1;
140.                    end
141.
142.                    12: // 1000 / 4.5
143.                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
144.                   else begin A <= 32'b01000100011110100000000000000000; B <= 32'b01000000100100000000000000000000; Start_Sig <= 1'b1; end
145.
146.                    13:
147.                    begin
148.                            $display("A = %b and rA = %b", A,SQ_rA);
149.                            $display("B = %b and rB = %b", B,SQ_rB);
150.                            $display("A.m /B.m = %b", SQ_Temp);
151.                            $display("Result = %b", Result);
152.                            i <= i + 1'b1;
153.                    end
154.
155.                    14: // 176.9218 / 1146.3369
156.                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
157.                  else begin A <= 32'b01000011001100001110101111111011; B <= 32'b01000100100011110100101011001000; Start_Sig <= 1'b1; end
158.
159.                    15:
```

```
                    begin
                        $display("A = %b and rA = %b", A,SQ_rA);
                        $display("B = %b and rB = %b", B,SQ_rB);
                        $display("A.m /B.m = %b", SQ_Temp);
                        $display("Result = %b", Result);
                        i <= i + 1'b1;
                    end

                    16: // 3.142857 / -0.723093
                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
                    else begin A <= 32'b01000000010010010010010010010; B <= 32'b10111111001110010001110010011111; Start_Sig <= 1'b1; end

                    17:
                    begin
                        $display("A = %b and rA = %b", A,SQ_rA);
                        $display("B = %b and rB = %b", B,SQ_rB);
                        $display("A.m / B.m = %b", SQ_Temp);
                        $display("Result = %b", Result);
                        i <= i + 1'b1;
                    end

                    18: // 0.0625 / 0.25
                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
                    else begin A <= 32'b00111101100000000000000000000000; B <= 32'b00111110100000000000000000000000; Start_Sig <= 1'b1; end

                    19:
                    begin
                        $display("A = %b and rA = %b", A,SQ_rA);
                        $display("B = %b and rB = %b", B,SQ_rB);
                        $display("A.m /B.m = %b", SQ_Temp);
                        $display("Result = %b", Result);
                        i <= i + 1'b1;
                    end

                    20:
                    i <= i;

                endcase

endmodule
```

激励文本的书写风格基本上没有什么改变，怒笔者懒得解释了。我们还是直接来看仿真结果吧。

仿真结果：

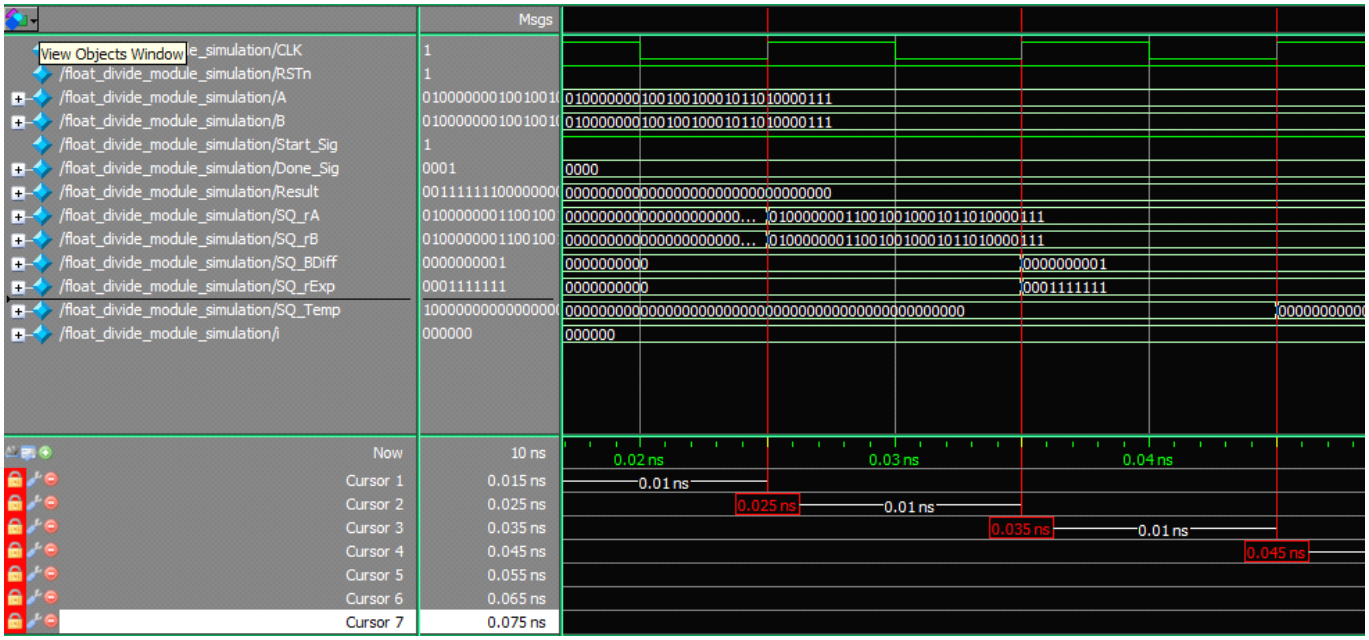


图 4.11.2 仿真过程 1。

如图 4.11.2 所示，在 C1 的时候该模块被使能。C2 的时候是操作数预处理和符号位运算，所以在 C2 的未来输出被回复的 rA 和 rB，此外也取得符号运算结果。在 C3 的时候是价码操作，所以在 C3 的未来输出价码运算的结果。

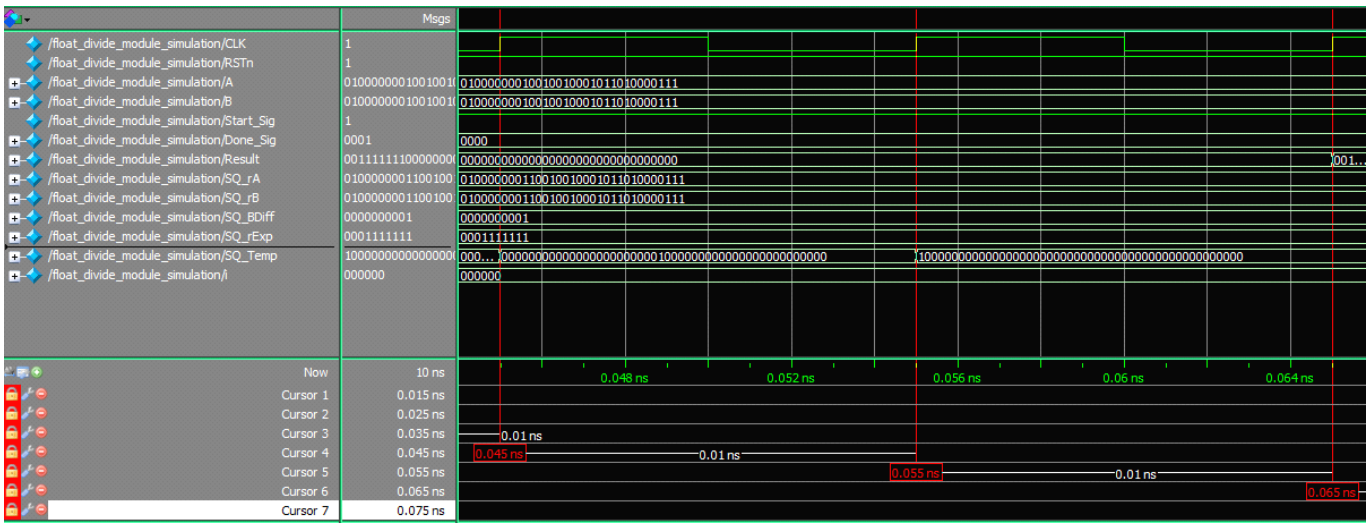


图 4.11.3 仿真过程 2。

在 C4 的时候是尾数运算，A 的尾数被拉大 24 档马力然后和 B 运算，所以在 C4 的未来输出运算的结果。在 C5 的时候是结果调整，Temp[24]的过去值为 1，所以在 C5 的未来输出被左移 24 位的 Temp。

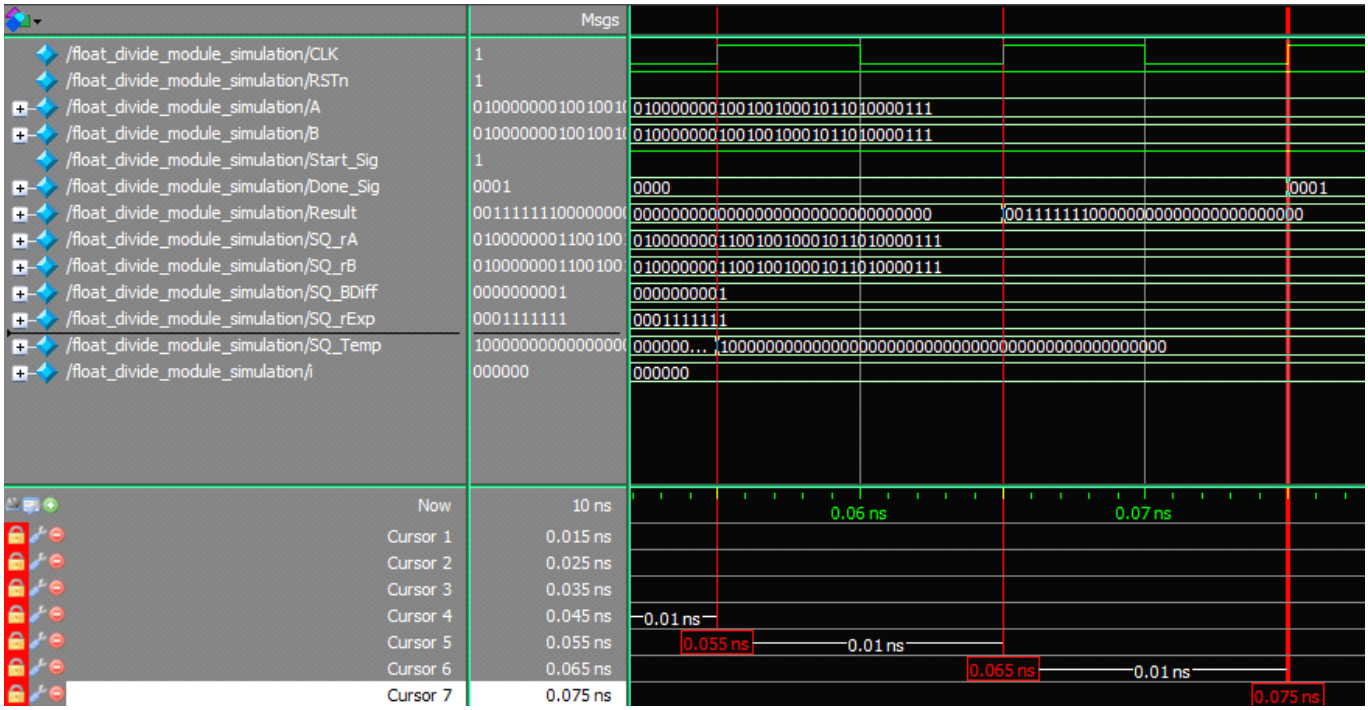


图 4.11.4 仿真过程 3。

在 C6 的时候是输出和格式化，所以在 C6 的未来输出相关的结果。基本上，浮点除法运算是总多运算之中最简单，步骤也是最单调的。接下来还是老习惯，笔者不解释所有仿真过程，作为补偿笔者给出所有运算的验证结果。

Q1: A = 3.142 , B = 3.142 , A / B = ?

```
A = 32'b0_10000000_10010010001011010000111
B = 32'b0_10000000_10010010001011010000111

32'b0_01111111_00000000000000000000000 Test Result 1
32'b0_01111111_00000000000000000000000 Real Result 1
                                       True Result 1
```

Q2: A = 10 , B = 3 , A / B = ?

```
A = 32'b0_10000010_01000000000000000000000
B = 32'b0_10000000_10000000000000000000000

32'b0_10000000_10101010101010101010101 Test Result 3.3333332
32'b0_10000000_10101010101010101010101 Real Result 3.3333332
                                       True Result 3.3333333...
```

Q3: A = 6.32 , B = 0.26 , A / B = ?

```
A = 32'b0_10000001_10010100011110101110001
```

```
B = 32'b0_01111101_00001010001111010111000

32'b0_10000011_10000100111011000101000 Test Result 24.3076934
32'b0_10000011_10000100111011000100111 Real Result 24.3076915
                                       True Result 24.30769
```

Q4: A = 3.48 , B = 10 , A / B = ?

```
A = 32'b0_10000000_10111101011100001010010
B = 32'b0_10000010_01000000000000000000000

32'b0_01111101_01100100010110100001110 Test Result 0.3479999
32'b0_01111101_01100100010110100001110 Real Result 0.3479999
                                       True Result 0.348
```

Q5: A = 1.442 , B = 11.515 , A / B = ?

```
A = 32'b0_01111111_01110001001001101110101
B = 32'b0_10000010_01110000011110101110001

32'b0_01111100_00000000011101111000010 Test Result 0.1252279
32'b0_01111100_00000000011101111000010 Real Result 0.1252279
                                       True Result 0.12522796352...
```

Q6: A = 1000 , B = 0.5 , A / B = ?

```
A = 32'b0_10001000_11110100000000000000000
B = 32'b0_01111110_00000000000000000000000

32'b0_10001001_11110100000000000000000 Test Result 2000
32'b0_10001001_11110100000000000000000 Real Result 2000
                                       True Result 2000
```

Q7: A = 1000 , B = 4.5 , A / B = ?

```
A = 32'b0_10001000_11110100000000000000000
B = 32'b0_10000001_00100000000000000000000

32'b0_10000110_10111100011100011100011 Test Result 222.2222137
32'b0_10000110_10111100011100011100100 Real Result 222.2222290
                                       True Result 222.2222222....
```

Q8: A = 176.9218 , B = 1146.3369 , A / B = ?

```
A = 32'b0_10000110_01100001110101111111011
B = 32'b0_10001001_00011110100101011001000

32'b0_01111100_00111100000101001101101 Test Result 0.1543366
32'b0_01111100_00111100000101001101101 Real Result 0.1543366
                                       True Result 0.154336652
```

Q9: A = 3.142857 , B = -0.723093 , A / B = ?

```
A = 32'b0_10000000_10010010010010010010010
B = 32'b1_01111110_01110010001110010011111

32'b1_10000001_00010110001010111000110 Test Result -4.3464078
32'b1_10000001_00010110001010111000110 Real Result -4.3464078
                                       True Result -4.3464077
```

Q10: A = 0.0625 , B = 0.25 , A / B = ?

```
A = 32'b0_01111011_00000000000000000000000
B = 32'b0_01111101_00000000000000000000000

32'b0_01111101_00000000000000000000000 Test Result 0.25
32'b0_01111101_00000000000000000000000 Real Result 0.25
                                       True Result 0.25
```

Q1~Q10 基本上也没有什么好说的，因为每一题的 Test Result 都非常接近 Real Result，无论操作组拥有几位十进制的尾数都不会使精度偏离得很厉害。答案很简单，因为浮点除法所采取保护精度的方法和其他的浮点运算不同。实验再一次证明，对于单精度格式来说，执行除法运算的时候，“24 位档”是最理想的马力程度。

总结：

哇~这一章是终于结束了！从收集资料到编辑笔记，笔者真的做了很长的准备功夫。在这里有些同学可能很遗憾，因为本章没有出现任何和精密控时的内容。读者千万不要怀疑，本章的内容也是围绕着整合的思想。不过比较不同的是，本章的目的是在思想和思路上做好准备。那些从顺序语言过渡而来的朋友，在许多情况下（都是不知不觉）他们都无法抛弃一些顺序语言在头脑里扎根已久的运行模式。笔者希望透过本章可以彻底并且狠狠地为顽固的读者敲出一条可以活动的裂缝。

笔者和读者都一样，在接触 Verilog HDL 语言一段时间后，一直无法摆脱我们在这个世界的认知或者常识。我们的现实世界是基于十个手指头，同样高级语言也是基于十个手指头。相反的，主宰 Verilog 世界里的自然规则是二进制。说实话，要完全的适应 2 进制是一个非常漫长的过程，因为在生理上大脑会抗拒。

就在那个神奇的夜晚 Verilog 之神给笔者指点，神明为笔者指出一条有效并能适应二进制的捷径，那就是透过学习浮点数的运算。从本章的内容中，我们可以知晓浮点数运算在书写形式上，或者十进制的模式上，都和二进制有很大的区别，就以浮点乘法为例，那些被隐藏的陷阱就是我们不熟悉的一面。

当然神明的指引不只是要笔者适应二进制，更贴切说是适应 Verilog 世界的自然规则。神明要笔者和 Verilog 的自然融合成为和谐的一致，然后按着自然在这个世界不停的创造。此外，本章还是为后续的几个章节做好准备。笔者常说实验的目的是其次，最重要的是思路 ... 读者还记得时序篇里的整数乘法和整数除法吗？它们在接下来的章节中，不是帮助读者更能理解 Verilog HDL 语言的理想时序吗？

好了，差不多要结束本章的内容了。在结束之前，笔者希望读者可以认真去理解浮点数四中不同的操作，因为浮点数在 Verilog HDL 语言是另一个很重要的大门。想想看有多少人被摘住在这个大门之外？很可能是我们使用高级语言太多了，有一种理论说过“如果人类过度依赖某个方便的技术，那么人类某个天然的功能就会退化”。高级语言也是同样的道理，因为很多运算在使用者不知道的情况下进行。

说白了就是“在高级语言中晓得浮点数，实际上一点也不浮点数是虾米”。高级语言给人的感觉是越用回脑经越懒得思考，反之 Verilog HDL 语言越用脑经越好使。此外，笔者经过这一次完成的笔记，非常惊讶到 ... 原来有关浮点数的资料真的少得可怜，不仅而已很多资料根本不沾边。不说了，说了都使人忧心。

第五章 优化和平衡

5.1 Verilog HDL 优化的思考

当读者听到优化这一词，读者会联想到什么？“提升某个东西的发挥或者产生？”假设我们想优化某个又 Verilog 的模块，那么读者该怎么办？从顺序语言过度到 Verilog HDL 语言，笔者一直放不下之前的认知。在顺序语言的世界里，优化的意思就是“提升运算的速度”或者说“降低运算的时钟消耗”。

如果要提升运算速度，一来更换精简的算法；二来就是用空间换速度。在此我们已经开始接触到问题点了？“优化”这一词的意义难道就是提高速度，或者降低时钟的消耗数而已吗？还是 .... 在 Verilog 的世界里，“优化”这一词的比起顺序语言的“优化”，意义的范围来得更广。

尤其说是“优化”还不如说“平衡”来得更贴切。根据笔者自己的理解，Verilog HDL 语言的优化可以分为 3 类“用空间换时钟”，“用时钟换空间”还是双赢的局面。实际上双赢的局面很难实现，我们即不增加空间又可以减低时钟的消耗，唯一一个可能性就是减低运算的质量。本章我们要讨论的内容就是围绕着顺序语言和 Verilog HDL 语言各自的优化。

在上一章里， 笔者设计了 4 个和浮点数相关的运算模块， 实际上这 4 个模块是一个 Prototype——初模型而已。这话何解呢？初模型什么优化都不是，它仅仅是为了验证理解才被建立而成。初模型在建立之其，笔者只考虑如何提高模块的表达能力而已（这又是不是另一种的优化吗？）

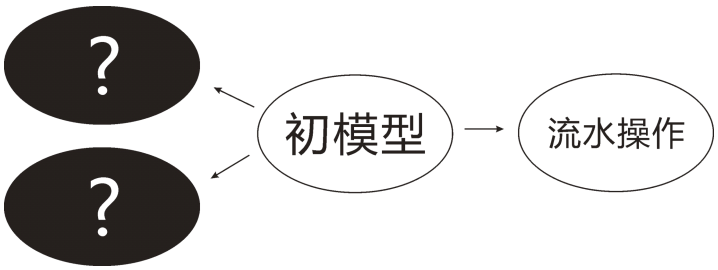


图 5.1.1 初模型的加工。

如图 5.1.1 所示，初模型可以加工成为可以支持流水操作的模块。同样的道理，只要明白了其他的细节，初模型也可以加工成为不同运行模式的模块。至于两个黑色的椭圆形就是本章要涉及的其他加工工作，具体笔者先不过于细谈。在此读者只要明白初模型的重要性就可以了。

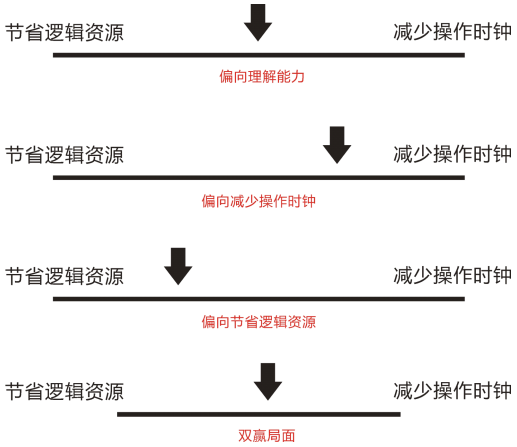


图 5.1.2 平衡的概念。

根据笔者自己的理解， 如图 5.1.2 所示，当我们偏向模块的理解能力的时候，偏向箭头保持在中间点；当我们偏向减少操作时钟的时候，偏向箭头就会向右移动，因此我们需要花费更多的逻辑资源；相反的，如果偏向的选择是节省逻辑资源，那么箭头会向反方向移动，因此时钟的消耗会随之提升。如果什么都不是的话，那么黑色的线条就会缩短，最终会得到双赢的局面。

在这里，初模型成为一个参考的标准 ...从另一个角度来想初模型可以是某个程度的理想模型。图 5.1.2 还隐藏着另一个重要的信息，那就是：无论箭头有没有移动，还是黑色线条有没有缩短。在建模的角度里，任何动作最终都会达到一个平衡的局面，最重要还是看设计者下单，然后要买什么货。

可能有许多同学无法接受笔者的观点或者无法理解的想法，这种感觉就像在水中的鱼问天上的鸟："天空是怎样的？"无论鸟怎样告诉鱼天空是这样，是那样，鱼儿无法用在水里生活的认识去想象和理解，鱼儿唯有抛开这些认识才能理解鸟儿的话 ...

在某天夜晚笔者带着问题进入梦乡，在梦里笔者看到 Verilog 之神在耍杂技 ... 神明手里拿着一杯水，神明问笔者："水会使你会联想什么？"，笔者当然会回答大海，河川等液态的水。神明以不可思议的能力把加热又降温。水一会儿变成一团水蒸汽，一会儿又变成冰块。神明又问笔者："如果我再一次提及水，那么你又会不会联想到水蒸气和冰块呢？别忘了水的本质一直没有改变过"。啊！一声，笔者似乎明白什么了 ...

水对普通人来说，液态是一种习惯的认知；水对南北极来说，固态是一种习惯的认知；水对天空来说，气态是一种习惯的认知。对于 Verilog HDL 语言来说，它不局限与任何形态 ... 但是对于笔者来说"强调模块的能力"是 Verilog HDL 语言应有的默认形态（笔者所谓的初模型）。如果某个模块以初模型作为基础，它可以变成流水操作或者其他形态的模块，但是由始至终该模块的功能依然没有被改变到。

如果我们把 Verilog HDL 语言比拟成水的话，以笔者的理解"重视表达能力"是液态；"重视节省逻辑资源"是固态；"重视减少时钟操作"是气态。当然 Verilog HDL 语言不可能仅限于 3 个形态而已，在此笔者只是作一个假设而已（最近有一个科技新闻说物质有第 4 态的存在，据说是 Plasma 体）。

不同形态的模块与这小节的优化和平衡之间又有什么关系呢？首先我们来考虑情况一：“假设有一个设计，它重视表达能力，然后又重视减少操作时钟，其中前者优先过后者。”如果按照笔者的习惯，笔者会先建立初模型，然后慢慢偏向减少操作时钟的优化，以上的动作会一直重复，慢慢会达到一个平衡的状态。又假设情况而是一个逻辑资源紧张的设计，结果我们必须偏向节省逻辑资源的优化。

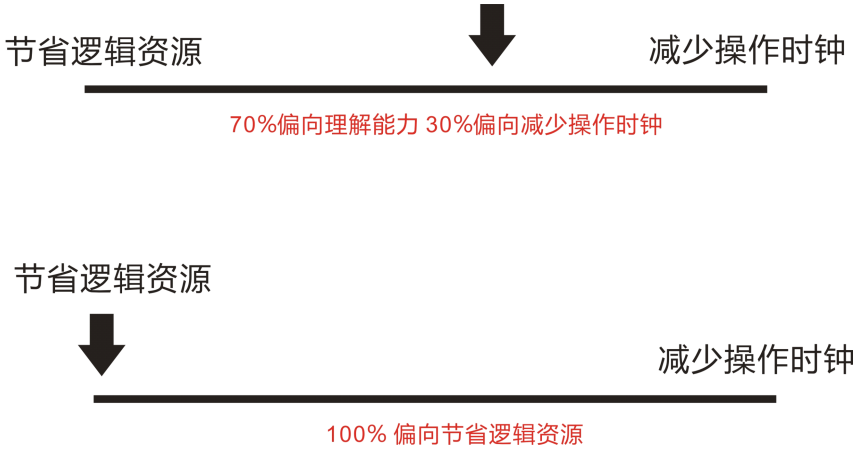


图 5.1.3 两个情况。

如图 5.1.3 所示，在情况一里最终我们会看到一个“70%重视表达能力，30%重视减少操作时钟”的平衡状态，这个设计的内容一眼就看得懂，不过只是在减少操作时钟的优化上效果不大。在反方面，减少操作时钟意味着增加逻辑资源的消耗。在情况二里，最终我们会看到一个“接近 100%重视节省逻辑资源”的平衡状态，这个设计可以用很少的逻辑资源实现预期的功能，不过在表达能力上，时钟的消耗上都是差强人意。图 5.1.3 只不过显示“两个优化的方向”而已，事实上还有更多考虑的方向，如“表达能力”就无法在图 5.1.3 显示。

笔者常常从参考书中会看到，什么什么更好的优化方案 ... 等等！说实在，这些内容都只是严重偏向一方而已，先不说它们不了解 Verilog HDL 语言，更不用讲所谓的优化是没有从多方作考虑。读者是否有过这样的经历？当我们按着某个目的去修改某个设计的内容，越修改感觉越糟糕 ... 虽然目的已经达成 ...结果内容却面目全非呢？

Verilog HDL 语言不像高级语言那样受到硬件资源的限制，所以优化的方向不只是一味减少操作的时钟。Verilog HDL 语言就像水一样，非常自由，没有任何限制，因此才会引起使用者的头疼。不过水在什么时候都有一个默认形态，Verilog HDL 语言也一样，笔者把“模块的表达能力”作为默认形态，读者又如何呢？

一个模块可以 100%偏向表达能力，也可以 60%偏向时钟的利用率（流水操作），也可以 70%偏向节省逻辑资源，又或者 80%偏向减少操作时钟。最终的决定是什么，该模块的功能始终如一，任何方向的优化和偏向都会达到一种平衡的状态，问题只是能不能接受而已？在笔者的老话中“好的设计不应该自己看得懂而已”，从中就可以知道笔者的口味是什么？没法子吗，笔者作的都是学习笔记，模块不偏向表达能力那么应该偏向什么？

好了，在这一小节我们从 Verilog HDL 语言的角度上讨论了有关优化和平衡的内容。在这里，笔者希望读者可以放开一下已经习惯的认知，尤其是在学习顺序语言的时候所学到的东西。Verilog HDL 语言真的是一个非常自由的语言，自由到让我们不知所措和头脑发疼。因此用 Verilog 设计的模

块到底有多少种形态？目前的笔者也知道其中几个而已。

当然，上述全部的内容都是基于笔者个人的理解而已，接受不接受，认同不认同，读者自己看着办吧。果真接受不了，就一笑而过然后关闭这本笔记吧。不要强迫自己干不喜欢的事情，最终受伤害的都是自己，伟大和荣耀一样都是人生的浮云，用不着执着那些假相，保全和健康才是最重要。

5.2 偏向节省逻辑资源的优化

第一章的实验里所实现的 4 个浮点数加，减，成和除运算器，实际上以偏向表达能力的设计。不过这 4 个设计都而外的“重”，浮点数的加减和除法器经过编译以后，逻辑资源的消耗都是吓死人。逻辑资源的消耗基本上都是超过 2000 . . . 这不是一个小数目，就连浮点数乘法器也用了 7 个乘法硬件，如果把 7 个乘法硬件用逻辑资源来计算的话，7 * 300 亦即消耗 2100 个逻辑资源左右。

到底是什么原因使它们变成如此笨重呢？我们先来看看是什么原因使浮点数加法器变成那么沉重？基本上有两个原因，第一个原因当然是尾数补偿的空间所致，第二个原因就是结果调整中庞大的 if 组合逻辑。首先我们先考虑一下优化的方案，以至于取得一个可接受的平衡结果 . . . 就这样吧“60%的表达能力，40%的节省空间”，感觉如何呢读者？

实验二十四：轻量版-浮点数加法模块

在这个实验里，我们要用循环来取代步骤 6-结果调整的组合逻辑，废话少说 . . . 我们还是直接来看代码吧。

float_add_module_light.v

```
module float_add_module_light
(
        input CLK, RSTn,
        input [31:0]A,B,
        output [31:0]Result,

        input Start_Sig,
        output [3:0]Done_Sig,

        /*****************/
        output [56:0]SQ_rA,SQ_rB,
        output [48:0]SQ_Temp,SQ_TempA,SQ_TempB,
        output [9:0]SQ_rExp,
        output [7:0]SQ_rExpDiff

);
```

基本上，模块的输入和输出的声明没有改变到。

```
        /*************************************/

        reg [3:0]i;
```

```
20.     reg [56:0]rA,rB;            // [56]Sign, [55:48]Exponent, [47:46]Hidden Bit, [45:23]Mantissa [22:0]M'Backup
21.     reg [48:0]Temp;             // [48]M'sign, [47:46]Hidden Bit, [45:23]M, [22:0]M'Backup
22.     reg [48:0]TempA,TempB; //[48]M'sign, [47:46]Hidden Bit, [45:23]M, [22:0]M'Backup
23.     reg [31:0]rResult;
24.     reg [9:0]rExp;              //[9:8] Overflow or underflow check, [7:0] usuall exp.
25.     reg [7:0]rExpDiff;      //Different between A.Exp and B.Exp
26.     reg [4:0]C1;
27.     reg isSign;
28.     reg isOver;         // exp overflow error feedback
29.     reg isUnder;        // exp underflow error feedback
30.     reg isZero;         // m zero error feedback
31.     reg isDone;
32.
33.     always @ ( posedge CLK or negedge RSTn )
34.         if( !RSTn )
35.             begin
36.                 i <= 4'd0;
37.                 rA <= 57'd0;
38.                 rB <= 57'd0;
39.                 TempA <= 49'd0;
40.                 TempB <= 49'd0;
41.                 Temp <= 49'd0;
42.                 C1 <= 5'd0;
43.                 rResult <= 32'd0;
44.                 rExp <= 10'd0;
45.                 rExpDiff <= 8'd0;
46.                 isOver <= 1'b0;
47.                 isUnder <= 1'b0;
48.                 isZero <= 1'b0;
49.                 isDone <= 1'b0;
50.             end
```

在 17~50 行中，其中在 26 行添加了 C1 的计数器，然后在 42 行复位它。

```
51.         else if( Start_Sig )
52.             case( i )
53.
54.                 0: // Initial A,B and other reg.
55.                 begin
56.                     rA <= { A[31], A[30:23], 2'b01, A[22:0], 23'd0 };
57.                     rB <= { B[31], B[30:23], 2'b01, B[22:0], 23'd0 };
58.
59.                     isOver <= 1'b0; isUnder <= 1'b0; isZero <= 1'b0; C1 <= 5'd0;
60.                     i <= i + 1'b1;
```

```
61.                 end
62.
63.                 1: // if rExp[9..8] is 1, mean A.Exp small than B.Exp
64.                    // while rExp[9..8] is 0, mean A.Exp large than B.Exp or same.
65.                 begin
66.                     rExp = A[30:23] - B[30:23];
67.
68.                     if( rExp[8] == 1 ) rExpDiff <= ~rExp[7:0] + 1'b1;
69.                     else rExpDiff <= rExp[7:0];
70.                     i <= i + 1'b1;
71.                 end
72.
73.                 2:   // if A < B; A.M move and A.E = B.E, else opposite act;
74.                 begin
75.                     if( rExp[8] == 1 ) begin rA[47:0] <= rA[47:0] >> rExpDiff; rA[55:48] <= rB[55:48]; end
76.                     else begin rB[47:0] <= rB[47:0] >> rExpDiff; rB[55:48] <= rA[55:48]; end
77.                     i <= i + 1'b1;
78.                 end
79.
80.                 3: // Modify TempA and TempB. with sign
81.                 begin
82.                     TempA <= rA[56] ? { rA[56], (~rA[47:0] + 1'b1) } : { rA[56], rA[47:0] };
83.                     TempB <= rB[56] ? { rB[56], (~rB[47:0] + 1'b1) } : { rB[56], rB[47:0] };
84.                     i <= i + 1'b1;
85.                 end
86.
87.                 4: // Addition
88.                 begin Temp <= TempA + TempB; i <= i + 1'b1; end
89.
90.                 5: // modify result
91.                 begin
92.                     isSign <= Temp[48];
93.                     if( Temp[48] == 1'b1) Temp <= ~Temp + 1'b1; // change M be postive
94.                     rExp <= {2'b00, rA[55:48]}; // or rB[55:48] , change rExp withbe rA.Exp or rB.Exp
95.                     i <= i + 1'b1;
96.                 end
```

随之也在步骤 0 中初始化 C1 为零值。

```
97.
98.                 6: // checking hidden bit by C1
99.                 if( C1 == 6'd25 ) i <= i + 1'b1;
100.                else if( Temp[47-C1] ) i <= i + 1'b1;
101.                else C1 <= C1 + 1'b1;
```

```
102.
103.            7: // Modify result
104.            begin
105.                if( C1 == 0 ) begin Temp <= Temp >> 1; rExp <= rExp + 1'b1; end
106.                // else if( C1 == 1 ) do nothing
107.                else begin Temp <= Temp << ( C1 - 1 ); rExp <= rExp - ( C1 - 1'b1 ); end
108.                i <= i + 1'b1;
109.            end
```

步骤 6~7（第 98~109 行）就是关键的步骤了。首先在步骤 6 里，C1 是以递增的方式来寻找最接近假定小数点的 1 值。第 99 行的 if 条件是表示，这个循环最大为 25 次，当循环超过 25 次后就进入下一个步骤；第 100 行是用来寻找最接近假定小数点的 1，其中 Temp[47-C1]则表示从 Temp 的最高位开始搜索 1，如果发现 1 值就进入下一个步骤；第 101 行是用来递增 C1，只要在 Temp 中找不到 1 值，C1 就会不停递增。

至于步骤 6 的具体含义是什么呢？首先我们决定寻找隐藏位（1 值）的范围是 Temp[47..23]，所以 C1 从 0~24 之间递增。当 C1 递增到 25，亦即循环结束，此外也表示在 Temp[47..23]之间没有隐藏位（也可以说是尾数 0 值）。如果在 Temp[47..23]之间找到隐藏位（1 值），那么 C1 就会指向哪里。用 C 语言来表示的话，大约这这个感觉：

```
for( C1 = 0; C1 < 25; C1++ )    if ( Temp[ 47-C1 ]) break;
```

第 103~109 行是关键的步骤，在这个步骤里我们参考 C1 的结果 ...

如果 C1 为 0，亦即 Temp[47:46]的结果为 2'b11 或者 2'b10，所以 Temp 右移 1 位，然后价码加 1；
如果 C1 为 1，亦即 Temp[47:46]的结果为 2'b01，没有操作直接进入下一个步骤；
如果 C1 为 2 或者以上，亦即 Temp[47:46]的结果为 2'b00，然后在 Temp[45..23] 某个地方一定存在 1 值，最后 Temp 和 rExp 执行相关的操作。

```
110.
111.            8: //error check and format result in float format
112.            begin
113.                if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end            // E Overflow
114.                else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end   // E Underflow
115.                else if( Temp[46:23] == 24'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // M Zero
116.                else if( Temp[22] == 1'b1 ) rResult <= { isSign, rExp[7:0], Temp[45:23] + 1'b1 }; // okay with normalised
117.                else rResult <= { isSign, rExp[7:0], Temp[45:23] }; // okay without normalise
118.                i <= i + 1'b1;
119.            end
120.
121.            9:
122.            begin isDone <= 1'b1; i <= i + 1'b1; end
123.
124.            10:
```

```
125.                    begin isDone <= 1'b0; i <= 4'd0; end
126.
127.                endcase
128.
129.        /*************************************/
130.
131.        assign Done_Sig = { isOver, isUnder, isZero, isDone };
132.        assign Result = rResult;
133.
134.        /****************************************/
135.
136.        assign SQ_rA = rA;
137.        assign SQ_rB = rB;
138.        assign SQ_Temp = Temp;
139.        assign SQ_TempA = TempA;
140.        assign SQ_TempB = TempB;
141.        assign SQ_rExp = rExp;
142.        assign SQ_rExpDiff = rExpDiff;
143.
144.        /*****************************************/
145.
146.  endmodule
```

第 110~146 行没有什么改变 ...

==========

经过编译以后，逻辑资源消耗的结果为 1241，比起实验二十中的 2068，一共节省了 827 个逻辑资源。在这里，可能有同学会说："笔者能不能再节省逻辑资源？"回答是"能是能"，但是这回我们要把什么东西作为代价呢？"浮点数的精度"... 就知道读者会这样说。平常如果不是特殊的情况，我们绝对是不可降低一个设计的质量，相反的情况就像国内的山寨或者无量商品一样 ... 想钱想疯了，为了钱连人命都可以节省掉!!!

不过笔者也用不着那么认真，这只是一个实验而已 ....

实验二十五：超轻量版-浮点数加法模块

float_add_module_light2.v

```
1.    module float_add_module_light2
2.    (
3.        input CLK, RSTn,
4.        input [31:0]A,B,
5.        output [31:0]Result,
```

```
6.
7.          input Start_Sig,
8.          output [3:0]Done_Sig,
9.
10.         /*****************/
11.         output [33:0]SQ_rA,SQ_rB,
12.
13.         output [25:0]SQ_Temp,SQ_TempA,SQ_TempB,
14.         output [9:0]SQ_rExp,
15.         output [7:0]SQ_rExpDiff
16.
17.   );
```

输入输出口基本上么有改变，不过仿真用的输出口（第 13 行）的位宽却减少了。

```
18.       /*************************************/
19.
20.       reg [3:0]i;
21.       reg [33:0]rA,rB;              // [33]Sign, [32:25]Exponent, [24:23]Hidden Bit, [22:0]Mantissa
22.       reg [25:0]Temp;             // [25]M'sign, [24:23]Hidden Bit, [22:0]M
23.       reg [25:0]TempA,TempB; // [25]M'sign, [24:23]Hidden Bit, [22:0]M
24.       reg [31:0]rResult;
25.       reg [9:0]rExp;             //[9:8] Overflow or underflow check, [7:0] usuall exp.
26.       reg [7:0]rExpDiff;       //Different between A.Exp and B.Exp
27.       reg [4:0]C1;
28.       reg isSign;
29.       reg isOver;        // exp overflow error feedback
30.       reg isUnder;       // exp underflow error feedback
31.       reg isZero;        // m zero error feedback
32.       reg isDone;
33.
34.       always @ ( posedge CLK or negedge RSTn )
35.           if( !RSTn )
36.               begin
37.                   i <= 4'd0;
38.                   rA <= 34'd0;
39.                   rB <= 34'd0;
40.                   TempA <= 26'd0;
41.                   TempB <= 26'd0;
42.                   Temp <= 26'd0;
43.                   C1 <= 5'd0;
44.                   rResult <= 32'd0;
45.                   rExp <= 10'd0;
46.                   rExpDiff <= 8'd0;
```

```
47.                     isOver <= 1'b0;
48.                     isUnder <= 1'b0;
49.                     isZero <= 1'b0;
50.                     isDone <= 1'b0;
51.                 end
```

第 20~32 行是相关的寄存器声明，在 21 行寄存操作数用的 rA 和 rB，基本上已经减掉尾部补偿的空间，亦即从原本的 57 位宽变成 34 位宽。此外，操作空间 Temp，TempA 和 TempB 随着减掉尾数补偿的空间，位宽也从原本的 49 位变成 26 位。第 37~50 行是相关的复位操作。

```
52.             else if( Start_Sig )
53.                 case( i )
54.
55.                     0: // Initial A,B and other reg.
56.                     begin
57.                         rA <= { A[31], A[30:23], 2'b01, A[22:0] };
58.                         rB <= { B[31], B[30:23], 2'b01, B[22:0] };
59.
60.                         isOver <= 1'b0; isUnder <= 1'b0; isZero <= 1'b0; C1 <= 5'd0;
61.                         i <= i + 1'b1;
62.                     end
63.
64.                     1: // if rExp[9..8] is 1, mean A.Exp small than B.Exp
65.                        // while rExp[9..8] is 0, mean A.Exp large than B.Exp or same.
66.                     begin
67.                         rExp = A[30:23] - B[30:23];
68.
69.                         if( rExp[8] == 1 ) rExpDiff <= ~rExp[7:0] + 1'b1;
70.                         else rExpDiff <= rExp[7:0];
71.                         i <= i + 1'b1;
72.                     end
```

在步骤 0 里，rA 和 rB 对操作数的恢复动作也无视了尾数补偿（第 57~58 行）。

```
73.
74.                     2:   // if A < B; A.M move and A.E = B.E, else opposite act;
75.                     begin
76.                         if( rExp[8] == 1 ) begin rA[24:0] <= rA[24:0] >> rExpDiff; rA[32:25] <= rB[32:25]; end
77.                         else begin rB[24:0] <= rB[24:0] >> rExpDiff; rB[32:25] <= rA[32:25]; end
78.                         i <= i + 1'b1;
79.                     end
80.
81.                     3: // Modify TempA and TempB. with sign
82.                     begin
```

```
83.                 TempA <= rA[33] ? { rA[33], (~rA[24:0] + 1'b1) } : { rA[33], rA[24:0] };
84.                 TempB <= rB[33] ? { rB[33], (~rB[24:0] + 1'b1) } : { rB[33], rB[24:0] };
85.                 i <= i + 1'b1;
86.             end
87.
88.             4: // Addition
89.             begin Temp <= TempA + TempB; i <= i + 1'b1; end
```

步骤 2（第 74~79 行）是价码对齐的其中一个操作，随着尾数补偿的消失该步骤也作出相关的修改。同样的原因也发生在步骤 3（第 81~86 行）里——运算预处理，因此相关修改也进行着。具体的细节笔者就不罗嗦了，读者自己看吧。

```
90.
91.             5: // modify result
92.             begin
93.                 isSign <= Temp[25];
94.                 if( Temp[25] == 1'b1) Temp <= ~Temp + 1'b1; // change M be postive
95.                 rExp <= {2'b00, rA[32:25]}; // or rB[55:48] , change rExp withbe rA.Exp or rB.Exp
96.                 i <= i + 1'b1;
97.             end
98.
99.             6: // checking hidden bit by C1
100.            if( C1 == 6'd25 ) i <= i + 1'b1;
101.            else if( Temp[24-C1] ) i <= i + 1'b1;
102.            else C1 <= C1 + 1'b1;
103.
104.            7: // Modify result
105.            begin
106.                if( C1 == 0 ) begin Temp <= Temp >> 1; rExp <= rExp + 1'b1; end
107.                // else if( C1 == 1 ) do nothing
108.                else begin Temp <= Temp << ( C1 - 1 ); rExp <= rExp - ( C1 - 1'b1 ); end
109.                i <= i + 1'b1;
110.            end
```

步骤 5 是结果预处理，由于失去了尾数补偿相应的位宽，运算结果的符号位变成 Temp[25]（第 93 行），运算结果的价码也变成 Temp[32:25]。步骤 6~7（第 99~110 行）是循环型的结果调整操作，读者注意第 101 行的 if 条件，Temp[47-C1]修改为 Temp[24-C1]。至于其他的，都和上一个实验二十样。

```
111.
112.            8: //error check and format result in float format
113.            begin
114.                if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end          // E Overflow
```

```
115.                    else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end    // E Underflow
116.                    else if( Temp[23:0] == 24'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // M Zero
117.                    else rResult <= { isSign, rExp[7:0], Temp[22:0] }; // okay without normalise
118.                    i <= i + 1'b1;
119.                end
```

步骤 8 是输出和格式化，基本上它已经舍弃四舍五入的操作 ... 当放弃尾数补偿的那一瞬间，四舍五入的操作基本上已经失去意义。同样步骤 8 也做了相关的修改，如第 116 行的 if 判断，还有第 117 行的赋值里，相关的 Temp 作了修改。

```
120.
121.                9:
122.                begin isDone <= 1'b1; i <= i + 1'b1; end
123.
124.                10:
125.                begin isDone <= 1'b0; i <= 4'd0; end
126.
127.            endcase
128.
129.    /***************************************/
130.
131.    assign Done_Sig = { isOver, isUnder, isZero, isDone };
132.    assign Result = rResult;
133.
134.    /***************************************/
135.
136.    assign SQ_rA = rA;
137.    assign SQ_rB = rB;
138.    assign SQ_Temp = Temp;
139.    assign SQ_TempA = TempA;
140.    assign SQ_TempB = TempB;
141.    assign SQ_rExp = rExp;
142.    assign SQ_rExpDiff = rExpDiff;
143.
144.    /***************************************/
145.
146. endmodule
```

最后第 120~146 行，基本上没有任何改变 ....

当编译顺利通过以后，实验二十五超轻量版的浮点数加法器比起实验二十四轻量版的浮点数加法器，逻辑资源的消耗从原本的 1241 变成 445，亦即少了 796 个逻辑资源。实验二十五的浮点数加法器虽然在节省的排名中高高在上，但是它也付出相关的带价。在仿真的结果中，其中比较挑剔的两个问题，亦即 “-12.558 加 -7.309” 和“111.7762 加 302.4409”，取得的运算结果如表 5.2.1 所

示：

表 5.2.1 有尾数补偿和五尾数补偿的结果比较。

操作组	True Result	尾数补偿（实验二十/二十四）	无尾数补偿（实验二十五）
-12.558 加 -7.309	19.867	-19.8670005	-19.8669986
111.7762 加 302.4409	414.2171	414.2171020	414.2170715

读者看到了吗？这就是实验二十五要付出的带价，它在运算中精度发生流失... 虽然在某种程度上，如果条件不挑剔的话还是可以接受的。最后可能读者要问："我们能不能按某个程度的位宽上保留尾数补偿呢？"这是很好的想法，不过"某个程度"的范围到底是多少，这是一个很暧昧的问题，除了不停验证和测试以外，最终才可以得到"某个程度"的平衡点。这个伟大的任务就交给读者吧！在此笔者已经把重点讲完了 ... 累了，睡觉去了。

5.3 无意义的 for 循环

原本这一小节我们要建立一个轻量版的浮点数乘法器，不过一些原因笔者先插入和主题不怎么相干的内容，那就是关于 for 循环的话题。如果笔者说 for 循环在 Verilog 里是无意的关键字，不知道有多少人可以接受？笔者还有可能乱说话而遭受血光之灾。笔者这样说，笔者当然有绝对的理由和证据，并且可以用实验来证明。

笔者曾经在整合篇（上）第一章里讨论过，相关用 Verilog HDL 语言去模仿 For 循环的行为，然后将其中的好处整合竟来 ... 模仿归模仿，如果我们还是没有把本质和重要的细节理解清楚，再怎么整合也是无意义的？读者可能觉得笔者有点自打嘴巴似地的口吻，其实故事是这样的 ...

在整合篇（上）完成不久后某一天下雨的下午，笔者从窗户外观察随着屋檐滴下的小雨点，滴滴的一声又一声落下，于是笔者在心理暗自数了起来，不知不觉进入梦乡了。在梦中笔者惊吓 Verilog 之神坐在笔者的旁边，神明察觉了笔者的惊讶，神明做了一个安静的手势然后有把手指往下指。笔者惊呆了，笔者的座下是一个庞大的自然场景 ...

那是水的循环，水从大海里蒸发成为水蒸气，然后累积在天空成了云朵，经过大风推呀推，云朵一边移动体积一边变大最终变成乌云，随着乌云从天而降成为密雨，大地被滋润。大自然的伟大和运动深深把笔者感动了，过一会儿神明开口说话：

“一个完整的水循环有好几个周期，水变成水蒸气有固定的周期，水蒸气变成云朵有固定的周期，云朵变成乌云有固定的周期，乌云变成密雨也有固定的周期，雨水降落大地也有一定的周期。不停的周期时而运动时而重复成为循环 ... ”笔者尝试回答什么，只见神明笑一笑又说：“ 孩子，你已经知道答案了 ...”从梦中醒来的那一刻，笔者似乎领悟到什么。

这是一个神奇又荒唐的梦，不过这些都不是重点 ... 直觉要笔者找到什么，笔者翻查存放在电脑里的参考书，“没有 ... 没有”果真什么也没有发现，最后“啊”一声，终于明白那个梦里的信息。当笔者在浏览参考书的时候，无论是那一本书它们都没有告诉笔者“for 到底是为了什么在执行循环？”，然而“for 又是由什么组成？”。

这些看似很幼稚的小问题，而且答案又是理所当然，它到底有什么研究的价值？在这里笔者和读者都犯了一个严重的认知错误，亦即用顺序语言去思考 for 循环。首先我们必须知道，顺序语言的基础是指令，诺干指令组成是一个步骤，最后诺干步骤形成函数。在顺序语言里 for 语言就是用来控制步骤的循环，话说白点 for 是执行步骤或者函数的重复操作 ... 别忘了，一个步骤到底有几个指令，那是使用者不被允许知道的。

当我们在 Verilog HDL 语言里用 for 循环，我们到底是控制“时钟周期呢？步骤周期呢？还是功能周期呢？”。

时钟周期就是单个步骤所消耗时钟数，大致的表现有：

```
0:
```

```
    begin
        //动作
        if( C1 == 25 -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
        else C1 <= C1 + 1'b1;
    end
```

步骤周期是指两个步骤以上的循环，大致的表现如下：

```
    0:
    begin  执行动作一；i <= i + 1'b1; end

    1:
    begin  执行动作二; X <= X + 1'b1; i <= i + 1'b1; end

    2:
    if( X == 4 ) i <= i + 1'b1;
    else i <= i - 2'b2;
```

功能周期是指某个功能的循环操作，如：

```
//功能模块
if( Start_Sig )
    case( i )
        0: ...
        ...
        7: isDone <= 1'b1; i <= i + 1'b1;
        8: isDone <= 1'b0; i <= 4'd0;
    endcase

//调用功能模块
case( i )
    0，1，2，3:
    if( Done_Sig) begin isStart <= 1''b0; i <= i + 1'b1; end
    else isStart <= 1'b1;
```

```
for(  控制变量初始化；判断条件/逻辑；控制变量操作  ) {功能操作}
```

此外，在顺序语言里 for 的本质是由诺干步骤形成的，亦即“控制变量初始化，判断条件，循环控制变量操作，还有功能操作”一个 for 关键字里边就夹杂那么多步骤。如果我们深入思考这个问题，神明呀！那是一个非常危险的东西。因为一次性的 for 循环操作到，底用了多少时钟，我们根本不知道？不仅而已，for 循环是建立在顺序操作之上，如果建模技巧没有仿顺序操作的支持，我们到底要怎样模仿 for 循环！?

经过深思熟虑以后，笔者得到一个结论，那就是：“关键字 for 在 Verilog HDL 语言是无意义的存在，

无论是综合语言还是验证语言哪一方”。其他的关键字还有：如 while，do ... while，break，continue, loop 和 goto 等 ... 笔者可以用同样的理由要它们立即枪毙！只要我们明白 Verilog HDL 语言的理想时序，只要建模技巧可以支持仿顺序操作，一个简单的 i 就可以实现一切功能，而且控制时钟的效果还而外的好。

读者读到这里，可能会认为笔者彻底疯了 ... 因为那些关键字是经历多年的经典，任何语言都会出现它们的身影，笔者以几个简单的理由就抹杀它们在 Verilog 的地位，这样的行为除了疯狂还有什么呢？笔者不否认自身的疯狂和自大，但是事实就是如此，笔者一向对事不对人（关键字）。

在这里，我们必须认真思考同质不同样的问题：

```
for( C1 = 0; C1 < 10; C1++ ) if( Temp[C1] == 1 ) break;
```

这一段的代码的意思是：如果 Temp 中的 C1 寻址结果是 1 的话就跳出 for 循环。其中 C1 可以在任何时候都跳出 for 循环，如果这只是单纯的顺序操作的话，那么问题就无大碍。相反的，如果这是一个和精密控时有关的话 ... 换句话说，for 循环到底是用 1 个时种，2 个时钟，还是 n 个时钟完成一次性的操作？对于精密控时来说，无疑是一个严重的打击 ... 因为操作周期是不固定的，这个问题会严重影响我们评估一个模块的消耗时钟数。

读者：“这么说，实验二十四的设计即不是有问题了吗？”不，这句话又不能这样说，实际上实验二十四的设计到底有没有问题，还要视情况和设计的目的而论。在此，笔者只想提醒读者，不可以忽视任何会引出问题的小细节。在还没有进入实验之前，我们再来研究研究看看，在精密控时 中 for 循环要注意什么呢？

实验二十六：轻量版-浮点数加法模块・改

注意，实验二十六的 Verilog 文件命名虽然和实验二十五一样，但是实验二十五是超级轻量版的浮点数加法器，然而实验二十六则是从实验二十四中的基础上，在一些微妙的细节上做了一些修改，所以读者不要搞混了。

float_add_module_light2.v

```
100.            6:
101.            begin
102.                if( Temp[47-C1] ) X[5] <= 1'b1;
103.                else if( !X[5] ) X[4:0] <= X[4:0] + 1'b1;
104.
105.                if( C1 == 25 -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
106.                else C1 <= C1 + 1'b1;
107.            end
108.
109.            7: // Modify result
110.            begin
111.                if( X[4:0] == 0 ) begin Temp <= Temp >> 1; rExp <= rExp + 1'b1; end
```

```
112.                    // else if( X[4:0] == 1 ) do nothing
113.                    else begin Temp <= Temp << ( X[4:0] - 1'b1 ); rExp <= rExp - ( X[4:0] - 1'b1 ); end
114.                    i <= i + 1'b1;
115.                end
```

一些重复性的代码笔者就不贴了。基本上在实验二十六中，除了步骤 6 和 7 以外，其他的步骤都没有改变。C1 依然是 5 位宽的寄存器，它用来控制时钟；X 是 6 位宽的寄存器 X[5]主要是针对隐藏位的标志位，X[4:0]则是用来计数用。第 105~106 行表示该步骤用了 25 个时钟，其中在结束的前一个时钟清零 C1 和更新步骤 i。第 102 行的 if 条件则是用来检测隐藏位——值 1，如果发现 1 值则设置 X[5]这个隐藏位，否则（第 103 行）X[4:0]就递增。

我们假设 Temp[47:46]为 2'b00，则 Temp[45] 为 1，大致的过程如下：

C1 为 0 的时候，发现 Temp[47]为 0，则 X[4:0]递增，X[4:0]为 1；
C1 为 1 的时候，发现 Temp[46]为 0，则 X[4:0]递增，X[4:0]为 2；
C1 为 2 的时候，发现 Temp[45]为 1，则 X[5]设置，X[4:0]不变；
C1 为 3 的时候，Temp[44]是什么一点以不重要，因为 X[5]已经为 1，X[4:0]不变；
...
C1 为 24 的时候，Temp[23]是什么一点也不重要，X[5]不变，X[4:0]不变，第 105 行的 if 条件成立 C1 清零，进入下一个步骤，最终我们得到 X[4:0]为 2。

步骤七的结果调整中用来调整尾数和价码，基本上和实验二十四的步骤七大同小异。第 111 行表示，如果 X[4:0]为 0，亦即隐藏位的 1 值在 Temp[47]中，结果尾数右移 1 位，价码加 1；第 112 行表示，如果 X[4:0]为 1 则，亦即隐藏位的 1 值在 Temp[46]中，结果什么也没有发生；如果 X[4:0]不是 0 也不是 1，那么第 113 的代码就会执行，那么 1 值隐藏在 Temp[45..23] 有（或者不在 Temp 里——尾数零值）。尾数会左移 X[4:0]被减一以后的位数，同样价码会减去 X[4:0]被减一以后的值。

我们继续用同样的例子来测试看看。此时 X 为 2，第 111~112 行的条件不成立，所以第 113 行的代码被执行，Temp 被左移 1 位，然和价码被减去 1。就这样，步骤 6~7 的操作就完成了 ... 读者是不是觉得有点微妙呢？呵呵，感觉确实如此。

```
56.                0: // Initial A,B and other reg.
57.                begin
58.                    rA <= { A[31], A[30:23], 2'b01, A[22:0], 23'd0 };
59.                    rB <= { B[31], B[30:23], 2'b01, B[22:0], 23'd0 };
60.
61.                    isOver <= 1'b0; isUnder <= 1'b0; isZero <= 1'b0; C1 <= 5'd0; X <= 6'd0;
62.                    i <= i + 1'b1;
63.                end
```

让我们综合思考一下，for 循环一般上又 4 个部分组成，控制变量的初始化发生在步骤 0 里，如 61 行所示。判断条件/逻辑则是第 105~106 行，至于控制变量操作同样也在 105~106 行里。最后的功能操作则是 105~106 行。在第 105~106 行的功能操作中，我们还用而外的变量 X 一边充当功能操作中的主角，一边又充当功能的控制变量 ... 感觉是不是越来越微妙了？

仿真结果：

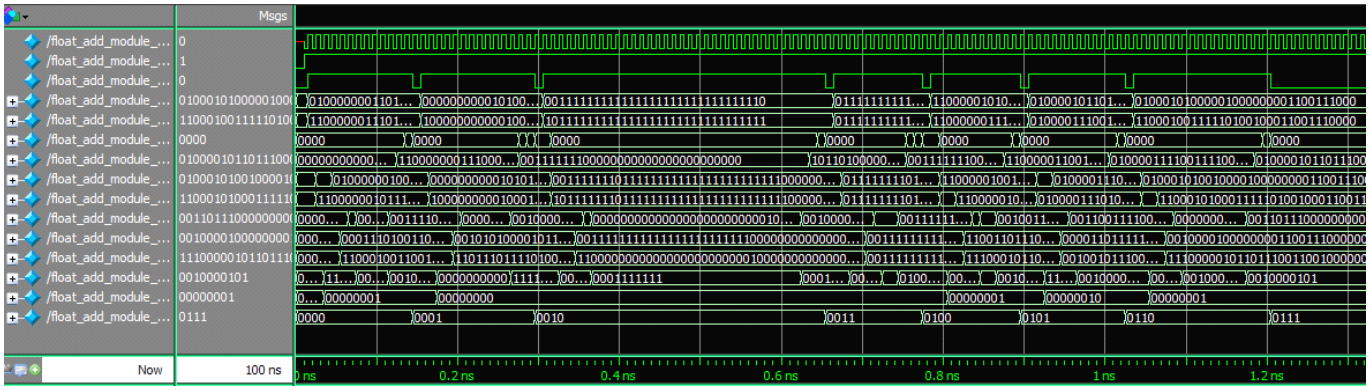


图 5.3.1 实验二十四的仿真结果。

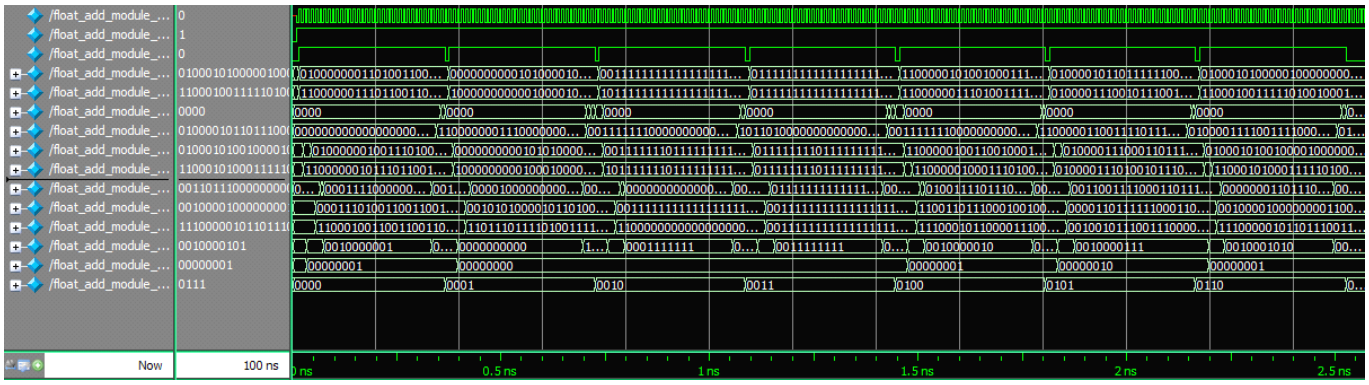


图 5.3.2 实验二十六的仿真结果。

图 5.3.1 是实验二十四的仿真结果，然而图 5.3.2 是实验二十六的仿真结果。不知道读者是否看到其中的不同吗？实验二十六里，一次性的操作都拥有固定的时钟消耗数；反之实验二十四里，一次性的操作都没有固定的时钟消耗数。虽然两个实验都实现同样的功能操作和实验结果，但是它们之间却有微妙的细节分差。

==========

这个小节的最后，笔者到底要表达什么？读者到底要理解什么？问题和答案都很直接，Verilog 真的需要委屈自己去迁就顺序语言的 for 循环吗？难道我们不能理解 for 循环的结构，然后用 Verilog 自身的方式去实现自己专属的 for 循环吗？如果这个理由是真确的，那么在 Verilog HDL 语言里 for，while 和 break 等关键字，它们的存在是不是失去了意义呢？

笔者不希望把自己的想法强加在读者身上，最终的答案需要读者自己去思考。不过有一点很明显的是，笔者的立场永远都是以 Verilog 为原点，然后俯视所有问题，所以笔者绝对无视这些循环用的关键字。嗯！笔者有点激动了 ...

5.4 软件操作和循环操作的疑惑

当我们在 C 语言上实现乘法操作的时候，如果处理器里包含乘法器硬件资源，结果会乘法器里执行，这是俗称的硬件操作。如果处理器不包含乘法器的话，那么浮点乘法会以软件操作执行。那么问题来了，什么是“软件操作”呢？首先我们必须清楚，C 语言是由汇编语言组成，然而汇编语言又是由某个指令系统组成，间接来说 C 语言的基础就是指令。

软件操作是指由指令完成的操作，举个例子：假设笔者要实现某个乘法操作，如果处理器没有乘法器的话，那么就没有相关的乘法指令。因此笔者必须用加法指令不停重复累加操作而达到乘法的效果。如果有算法，笔者可以根据算法，然后用相关的指令建立指令函数（如 Booth 算法）。无论笔者有用算法还是没用算法，为了取得乘法操作的结果，笔者是逃不过循环操作的命运。

（那些没有读过时序篇的朋友可能无法理解这一段内容的意思；为了实现乘法而不停执行累加操作称为传统乘法；Booth 算法则是实现乘法操作的算法；两者之间都有典型的循环操作）

不知何时在不知不觉中？软件操作和循环操作被挂上等价的代号，亦即软件操作 = 循环操作。这是浑浊的理解：软件的基础是指令；硬件的基础就是物理的东西（晶体管之类的东西），然而循环操作只是某种“操作模式”。

对于 Verilog HDL 语言来说，它到底是软件还是硬件呢？实际上 Verilog HDL 语言的基础是固件，因此它同时间拥有软件和硬件的本质，如果勉强分类的话 Verilog HDL 语言是偏向硬件（扫盲文里有详细的分类）。软件操作和硬件操作的操作模式都是非常有限制的，反之固件的操作模式是非常多样化。好奇的同学可能要问：“那么 Verilog 到底有多少种的操作模式？”

嗯，这个问题真的把笔者问倒了 ... Verilog HDL 语言到底有多少种的操作模式？首先必须根据设计者的设计倾向和习惯，不同的设计倾向和习惯都有可能衍生不同的操作模式。不过根据笔者目前的手腕可以举例的操作模式并不多，如图 5.4 所示：

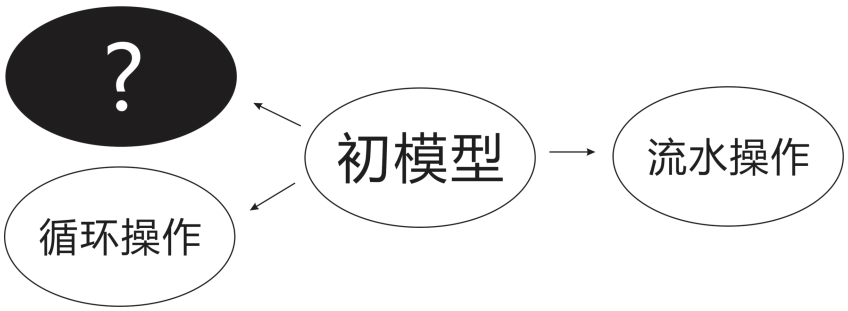


图 5.4.1 笔者所知道的操作模式。

操作模式是一个比较死硬的代名词，笔者更喜欢用“形态”来称呼，读者可能会问为什么？读者千万别被手指的形状局限了手指指向的认识。对于笔者来说 Verilog HDL 语言是非常自由的，它最接近的形象就是“水”，用“形态”来替代“操作模式”这支手指，我们更能认识 Verilog HDL 语言。所以，读者们就迁就一下笔者的意见和任性。

在图 5.4.1 中，初模型就是以“理解/表达能力”为重的形态，也是笔者的默认形态，不同的其他形态都可以从初模型出发。

（一）流水操作是一个以“时间利用率”为重的形态，具体的内容请参考时序篇，在此笔者就不重复了；
（二）循环操作是一个以“节省资源”为重的形态，更多时候它的理解非常接近软件操作。对于处理器来说硬件资源是非常贵重和稀少的，所以处理器才会以循环的方式去利用它们。
（三）关于“？”到底是怎样的一个形态，目前先卖个关子。

```
100.        6:
101.        begin
102.            if( Temp[47-C1] ) X[5] <= 1'b1;
103.            else if( !X[5] ) X[4:0] <= X[4:0] + 1'b1;
104.
105.            if( C1 == 25 -1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
106.            else C1 <= C1 + 1'b1;
107.        end
```

实验二十四~六的设计就是循环操作的例子，更具体一点如实验二十六—float_add_module_light2.v 其中的步骤 6（第 100~107 行），我们重复使用了 第 102~103 行的资源 25 次。在某种程度上循环操作是用时间换空间的操作。

这小节我们已经讲到的重点了，最后我们可以这样作结论：循环操作是 Verilog HDL 语言的其中一个形态（操作模式），虽然它不是软件操作，不过它和软件操作的目的非常类似，亦即以循环的方式重复使用同一资源。在 Verilog HDL 语言的角度里，循环操作有效重复利用某个步骤，从而达到节省逻辑资源的效果。

5.5 实验二十七：轻量版-浮点数乘法模块

在这个小节里，我们要建立一个轻量版的浮点数乘法器，因此首先我们要在实验二十二里先找到使浮点数乘法器臃肿的问题。

```
73.                 3:
74.                 begin
75.                     Temp <= rA[23:0] * rB[23:0];
76.                     i <= i + 1'b1;
77.                 end
```

之所以实验二十二的浮点乘法器会如此臃肿，原因是发生在步骤 3 的第 75 行。笔者很久以前就一直强调过，除非遇见无可奈何的情况下，不然的话我们不随便使用“*”，“/”，“%”等操作符，因为它们会把逻辑资源都吃光，其中“*”就是用大量逻辑资源来建立乘法法器的饿鬼。从 75 行中我们可以看到，Temp 求得的结果是 24 位 乘以 24 位。

如果我们的 FPGA 设备内嵌乘法器资源的话，那么“*”这个饿鬼会根据操作 A 和操作 B 的总位宽把诺干的乘法器资源吃掉。在实验二十二里，它把 7 个乘法器资源吃掉，如果换做逻辑资源来换算的话，大约是 7 * 300 = 2100 个左右（实际上更多）。既然我们知道这是“*”问题的所在，那么我们该怎么办呢？呵呵，在此我们就必须用到时序篇里那些拥有循环操作模式的乘法器了。

笔者还记得曾经许多同学曾针对时序篇的乘法器问了相关的问题：“笔者，既然 Verilog HDL 语言里已经有了相关的 * 操作符，为什么我们还要浪费精力去学习不同算法/模式的乘法器呢？”那时候的笔者以理解时钟为理由回答了这个问题。但是直觉告诉笔者，在时序篇里所学到的乘法器有很大的隐情，只是时机未到而已。

好了，现在是点秋香的时候，到底哪一种的乘法器会被选中呢？当当，幸运儿是时序篇里—实验二十三的 booth_multiplier_module2.v。至于什么原因它被选中呢？理由很简单，它最符合这本笔记的内容。
如果读者忘了 booth 算法是如何运算的话，请立即回家复习一下，在此怒读者不再重复同样的内容。

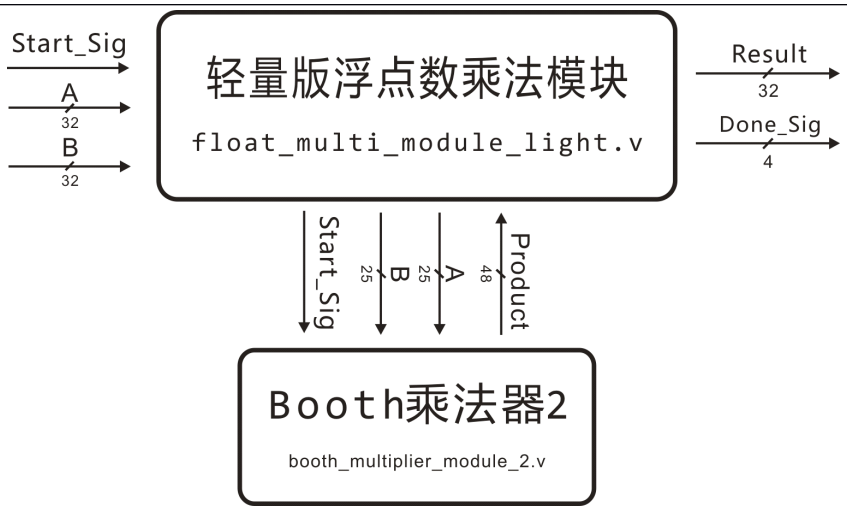


图 5.5.1 轻量版浮点数乘法模块建模图。

朋友，看到图 5.5.1 是不是被吓了一跳？图 5.5.1 是从实验二十二的基础发展而来，其中“*”操作符被 Booth 乘法器 2 所取代。首先围绕着轻量版浮点数乘法模块的输入输出信号的作用和实验二十二一模一样，不过作用于 booth 乘法器 2 之间的信号却和以往在时序篇-实验二十三里的 booth 乘法器 2 差不多。搞不明白的同学就抓抓头皮思考一会儿吧，那是建模篇和时序篇的基础知识。

在这里有同学会问：“在实验二十二里 * 操作符好像是实现 24 × 24 位宽的乘法操作 ...”真是犀利的眼睛，我们还是来看看 booth 乘法器 2 的代码吧：

booth_multiplier_module_2.v

```
1.    module booth_multiplier_module_2
2.    (
3.            input CLK,
4.            input RSTn,
5.
6.            input Start_Sig,
7.            input [24:0]A,
8.            input [24:0]B,
9.
10.           output [47:0]Product
11.
12.   );
```

第 1~12 行是该模块的输入输出，具体的连线请看图 5.5.1。虽然说 booth 乘法器 2 是替代 rA[23:0] * rB[23:0]的乘法操作，理应第 7~8 行的操作 A 和 B 位宽是 24 位才对呀？为什么这里声明为 25 位呢？读者不要忘了一个重点，booth 算法是包含符号位来运算的，然而 rA[23:0] * rB[23:0]是不包含符号位。为了迎合 booth 算法，操作 A 和 B 的[24]位是后来填上的符号位，都是 0 值。

```
13.           /*************************/
```

```
14.
15.         reg [4:0]i;
16.         reg [24:0]a;   // [24]Sign ,A result
17.         reg [24:0]s;   // [24]Sign ,reverse result of A
18.         reg [50:0]p;   // operation register
19.         reg [24:0]Diff1;
20.         reg [24:0]Diff2;
21.
22.         always @ ( posedge CLK or negedge RSTn )
23.             if( !RSTn )
24.                 begin
25.                         i <= 5'd0;
26.                         a <= 25'd0;
27.                         s <= 25'd0;
28.                         p <= 51'd0;
29.                         Diff1 <= 25'd0;
30.                         Diff2 <= 25'd0;
31.                     end
```

第 15~20 行是相关的寄存器声明，寄存器 a 是用来寄存操作数 a 的正码；寄存器 s 是用来寄存操作 b 的补码；p 是操作空间，位宽取得是 操作数 A 和 B 的位宽总和，然后再加上 1；Diff1 和 Diff2 是用来取得不同即时结果的寄存器。第 25~31 行是相关的复位操作。

```
32.             else if( Start_Sig )
33.                 case( i )
34.
35.                     0:
36.                     begin
37.                         a <= A;
38.                         s <= ( ~A + 1'b1 );
39.                         p <= { 25'd0 , B , 1'b0 };
40.                         Diff1 <= 25'd0;
41.                         Diff2 <= 25'd0;
42.
43.                         i <= i + 1'b1;
44.                     end
45.
46.                     1,2,3,4,5,6,7,8,9,10,11,12,13,14,
47.                     15,16,17,18,19,20,21,22,23,24,25:
48.                     begin
49.                         Diff1 = p[50:26] + a;
50.                         Diff2 = p[50:26] + s;
51.
52.                         if( p[1:0] == 2'b01 ) p <= { Diff1[24] , Diff1 , p[25:1] };
```

```
53.                                     else if( p[1:0] == 2'b10 ) p <= { Diff2[24] , Diff2 , p[25:1] };
54.                                     else p <= { p[50] , p[50:1] };
55.
56.                                     i <= i + 1'b1;
57.                                 end
58.
59.                                 26:
60.                                 begin i <= 5'd0; end
61.
62.                             endcase
63.
64.         /************************/
65.
66.          assign Product = p[48:1];
67.
68.          /************************/
69.
70.     endmodule
```

步骤 0 是用来初始化相关的寄存器，然后第 46~57 行就是关键的 booth 算法的操作了。我们知道 booth 算法有一个怪习惯，就是—它会按操作数的 N 个位宽而循环 N 次。在这里，操作数 A 和 B 均为 25 位宽，所以步骤 1 循环 25 次。第 59~69 行是步骤 i 返回的操作，关于这个步骤很重要，少了它整个功能模块会跑飞。

第 66 行 Product 的输出驱动，在这里同样也有怪点。按照 booth 算法的习惯，Product 的驱动理应是 p 空间的[50:1]，不过为什么是 p 空间的[48:1]。笔者举个简单的例子：

假设操作数 A 和 B 均为 25'b0_$$$$$$$$$$$$$$$$$$$$$$$$$ ，它们在相乘以后一定会取得 50'b00_$$$$$$$$$$$$$$$$$$$

```
3.          input CLK, RSTn,
4.          input [31:0]A,B,
5.          output [31:0]Result,
6.
7.          input Start_Sig,
8.          output [3:0]Done_Sig,
9.
10.         /*****************/
11.
12.         output [32:0]SQ_rA,SQ_rB,
13.         output [47:0]SQ_Temp,
14.         output [9:0]SQ_rExp,SQ_BDiff,
15.         output [3:0]SQ_i
16.    );
```

读 3~15 行是相关的输出输入声明，基本上和实验二十二一样。

```
17.           /**************************************/
18.
19.        reg [3:0]i;
20.        reg [32:0]rA,rB;   // [32]Sign, [31:24]Exponent, [23]Hidden Bit, [22:0]Mantissa
21.        reg [47:0]Temp;     // [48]M'sign, [47:46]Hidden Bit, [45:23]M, [22:0]M'resolution
22.        reg [31:0]rResult;
23.        reg [9:0]rExp,BDiff;        //[9:8] Overflow or underflow check, [7:0] usuall exp.
24.        reg isSign;
25.        reg isOver,isUnder,isZero;
26.        reg isDone;
27.        reg isStart;   // Enable the booth multiplier
28.        reg [4:0]C1;   // Counting Enbaled clock
29.
```

第 19~28 行是相关的寄存器声明，其中 27~28 行是用来控制 booth 乘法器 2。

```
30.         /**************************************/
31.
32.         wire [47:0]Product;
33.
34.         booth_multiplier_module_2 U1
35.         (
36.                .CLK( CLK ),
37.                .RSTn( RSTn ),
38.                .Start_Sig( isStart ),
39.                .A( {1'b0,rA[23:0]} ),
40.                .B( {1'b0,rB[23:0]} ),
```

```
41.             .Product( Product )
42.         );
43.
```

第 32~41 行是 booth 乘法器 2 的实例化，其中第 39 行的 A 又 rA[23:0] 带上 符号位 0 驱动；第 40 行的 B 也有相同的动作。笔者再啰嗦一下，booth 算法是包含符号位运算，如果单单只是 rA[23:0] rB[23:0]，booth 算法是无法正常运算的 ... 所以第 39~40 行的动作是绝对必须的。

```
44.     /****************************************/
45.
46.     always @ ( posedge CLK or negedge RSTn )
47.         if( !RSTn )
48.             begin
49.                 i <= 4'd0;
50.                 rA <= 33'd0;
51.                 rB <= 33'd0;
52.                 Temp <= 48'd0;
53.                 rResult <= 32'd0;
54.                 rExp <= 10'd0;
55.                 BDiff <= 10'd0;
56.                 isOver <= 1'b0;
57.                 isUnder <= 1'b0;
58.                 isZero <= 1'b0;
59.                 isDone <= 1'b0;
60.                 isStart <= 1'b0;
61.                 C1 <= 5'd0;
62.             end
```

第 49~61 行是寄存器相关的复位操作。

```
63.         else if( Start_Sig )
64.             case( i )
65.
66.                 0: // Initial and resulted out Signature
67.                 begin
68.                     rA <= { A[31], A[30:23], 1'b1, A[22:0]};
69.                     rB <= { B[31], B[30:23], 1'b1, B[22:0]};
70.                     isSign <= A[31] ^ B[31];
71.
72.                     isOver <= 1'b0; isUnder <= 1'b0; isZero <= 1'b0;
73.                     i <= i + 1'b1;
74.                 end
75.
76.                 1:
```

```
77.                 begin
78.                     if( rA[31:24] == rB[31:24] && ( rA[22:0] != 23'd0 & rB[22:0] != 23'd0 ) || rA[32] ^ rB[32] )
79.                         begin rB[31:24] <= rB[31:24] + 1'b1; rB[23:0] <= rB[23:0] >> 1; end
80.                     i <= i + 1'b1;
81.                 end
82.
83.                 2: // if rExp[9..8] is 1, mean A.Exp small than B.Exp
84.                     // while rExp[9..8] is 0, mean A.Exp large than B.Exp or same.
85.                 begin
86.                     BDiff = rB[31:24] - 8'd127; // + (~8'd127 + 1)
87.                     rExp <= rA[31:24] + BDiff; // A.Exp + B.Exp
88.                     i <= i + 1'b1;
89.                 end
```

第 66~89 行，步骤 1~3 的内容基本上和实验二十二一样。

```
90.
91.                 3: // Enable boothe multiplier
92.                 if( C1 == 27-1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
93.                 else begin C1 <= C1 + 1'b1; isStart <= 1'b1; end
94.
95.                 4:
96.                 begin Temp <= Product; isStart <= 1'b0; i <= i + 1'b1; end
97.
```

反之第 91~93 行的步骤 3，是用来启动和使能 booth 乘法器 2，booth 乘法器 2 需要 27 个时钟完成一次性的乘法运算，所以 isStart 就拉高 27 个时钟。其中第 92 行 if 条件里的 -1 表示在结束前的一个时钟，清零 C0 和递增步骤 i，别忘了此时是 isStart 还在拉高。第 95~96 行的步骤 4，是用来拉低 isStart (关闭 booth 乘法器 2)，然后顺便将结果赋值与 Temp。

```
98.                 5: // Check M'hidden bit
99.                 begin
100.                    if( Temp[47] == 1'b1 ) begin Temp <= Temp; end // do nothing
101.                    else if( Temp[46] == 1'b1 ) begin Temp <= Temp << 1; end
102.                    else if( Temp[45] == 1'b1 ) begin Temp <= Temp << 2; rExp <= rExp - 1'b1;end
103.
104.                    i <= i + 1'b1;
105.                end
106.
107.                6: //error check and decide final result
108.                begin
109.                    if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end       // E Overflow
110.                    else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // E Underflow
111.                    else if( Temp[47:24] == 24'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // M Zero
```

```
                    else if( Temp[23] == 1'b1 ) rResult <= { isSign, rExp[7:0], Temp[46:24] + 1'b1 }; // okay with normalised
                    else rResult <= { isSign, rExp[7:0], Temp[46:24] }; // okay without normalise
                    i <= i + 1'b1;
                end

                7:
                begin isDone <= 1'b1; i <= i + 1'b1; end

                8:
                begin isDone <= 1'b0; i <= 4'd0; end

            endcase

    /**************************************/

    assign Done_Sig = { isOver, isUnder, isZero, isDone };
    assign Result = rResult;

    /**************************************/

      assign SQ_rA = rA;
      assign SQ_rB = rB;
      assign SQ_rExp = rExp;
      assign SQ_BDiff = BDiff;
      assign SQ_Temp = Temp;
      assign SQ_i = i;

    /***************************************/

endmodule
```

好了，接下来的操作笔者就不重复了，不明白的同学可以复习一下实验二十二。

仿真结果：

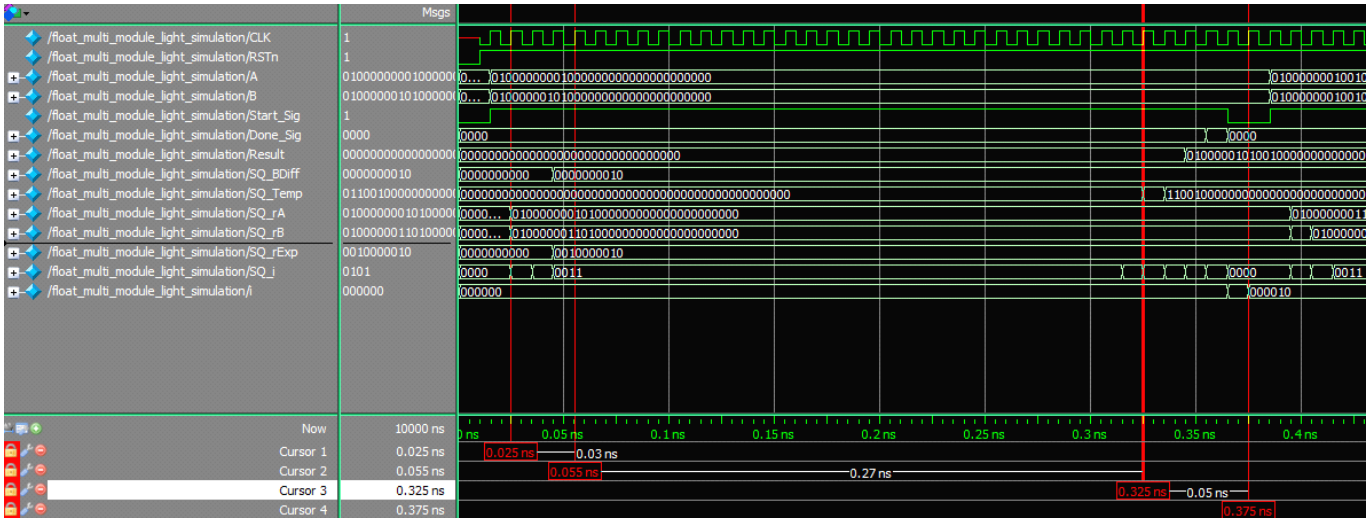


图 5.5.2 仿真结果。

激励文件的内容还是和实验二十二一样。图 5.5.2 是仿真结果，光标 C1~C4 之间的距离表示了该浮点乘法的一次性操作，总共用了 3+27+5 = 35 个时钟（一个时钟是 10ps）。其中 C2~C3 是 booth 乘法器占去的 27 个时钟，然而 C1~C2 和 C3~C4 的总和是 float_multi_module_light.v 占去的 8 个时钟。

==========

在仿真的结果中，我们得到的运算结果和实验二十二一样，不过不同的是 ... 比起实验二十二，实验二十七的浮点数乘法器消耗更多的时钟。最为弥补的效果，经过编译以后实验二十七的逻辑资源消耗数仅是 399 而已（实验二十二的逻辑资源消耗数预估是 2100 左右）。逻辑资源之间的消耗比率下了 n 倍，真是是可怕的倍差。

资源消耗数为 399 和时钟消耗数为 35，在某个程度来说实验二十七的浮点数乘法模块是一个性价比很好的东西。用 Verilog 的眼光来看，实验二十七倾向节省逻辑资源方面的优化，恰到某种程度的平衡。
在此，我们不牺牲输出质量和模块的表达能力 ... 不过，那些极端的家伙可能还不满意在节省逻辑资源上的优化，对它们还有“空间还能再节省”等~怨~念。

5.6 实验二十八：超轻量版-浮点数乘法模块

在这个小节里笔者会实现在实验二十七里留下的那些怨念（空间还能再节省）。以实验二十七为基础，如果要继续减少逻辑资源的消耗，我们应该从哪里下手呢？冷血的读者们估计已经发现了，就是在输出质量方面下手 ... 亦即减少输出的精度来提升节省空间的优化。实际上这是万万不可的赌博，风险非常高，不过我们只是做实验而已，试试一下也无妨。

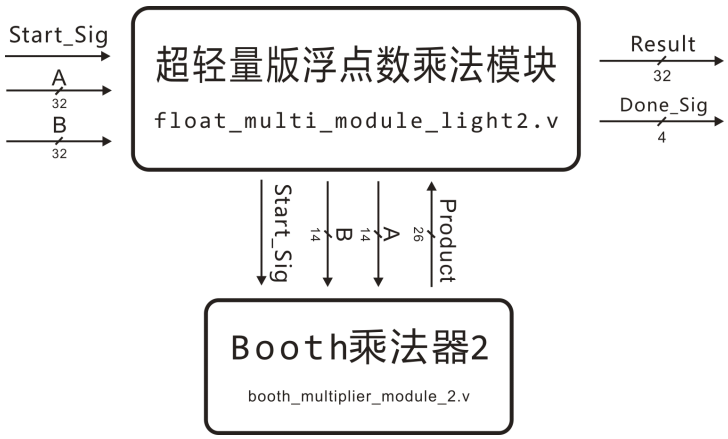


图 5.6.1 超轻量版浮点数乘法模块建模图。

图 5.6.1 是超轻量版浮点数乘法模块的建模图，比起上一个实验哦建模图（图 5.5.1）在浮点数乘法模块和 booth 乘法器 2 之间的连线，尤其是 A，B 和 Product，位宽已经减少了不少。A 和 B 摘去了 11 位位宽，从而 Prodcut 也丢失 22 位的位宽。换一句话说，我们已经牺牲掉某个程度的精度，在这里，有同学会问："浮点数乘法不像浮点数加减法那样，拥有尾数补偿 ... 如果要牺牲某个部分的精度，我们该从哪里下手呢？ "

这是一个很好的问题，最有效的方法就是直接在操作数 A 和 B 的尾数下手。单精度浮点数的尾数，一般都不是 23 位的吗？我们只要牺牲尾数后面的某个部分就可以了，举个例子：

```
原本：
A.m                    24'b111111111111111111111111
B.m                    24'b111111111111111111111111  ×
48'b111111111111111111111110000000000000000000000001

失去精度：
A.m                    24'b111111111111100000000000
B.m                    24'b111111111111100000000000  ×
48'b111111111110000000000000100000000000000000000000
```

在上诉的例子中，原本一方的操作 A 和 B 的尾数（包括隐藏位）全是 1 值，在相乘之后会得到相关的答案。反之在失去精度一方，我们尾数后面 11 位全部弃之为 0，然后再以相乘就会得到不同的结果。这就是在实验二十八中要实现的操作，具体的内容我们还是直接来看代码吧。

booth_multiplier_module_2.v

```
module booth_multiplier_module_2
(
        input CLK,
        input RSTn,

        input Start_Sig,
        input [13:0]A,
        input [13:0]B,

        output [25:0]Product

);
    /************************/

        reg [3:0]i;
        reg [13:0]a;   // [13]Sign ,A result
        reg [13:0]s;   // [13]Sign ,reverse result of A
        reg [28:0]p;   // operation register
        reg [13:0]Diff1;
        reg [13:0]Diff2;

        always @ ( posedge CLK or negedge RSTn )
            if( !RSTn )
                begin
                        i <= 4'd0;
                        a <= 14'd0;
                        s <= 14'd0;
                        p <= 29'd0;
                        Diff1 <= 14'd0;
                        Diff2 <= 14'd0;
                    end
            else if( Start_Sig )
                case( i )

                    0:
                        begin
                            a <= A;
                            s <= ( ~A + 1'b1 );
                            p <= { 14'd0 , B , 1'b0 };
                            Diff1 <= 14'd0;
                            Diff2 <= 14'd0;

```

```
43.                                     i <= i + 1'b1;
44.                                 end
45.
46.                                 1,2,3,4,5,6,7,8,9,10,11,12,13,14:
47.                                 begin
48.                                     Diff1 = p[28:15] + a;
49.                                     Diff2 = p[28:15] + s;
50.
51.                                     if( p[1:0] == 2'b01 ) p <= { Diff1[13] , Diff1 , p[14:1] };
52.                                     else if( p[1:0] == 2'b10 ) p <= { Diff2[13] , Diff2 , p[14:1] };
53.                                     else p <= { p[28] , p[28:1] };
54.
55.                                     i <= i + 1'b1;
56.                                 end
57.
58.                                 15:
59.                                 begin i <= 4'd0; end
60.
61.                             endcase
62.
63.         /************************/
64.
65.         assign Product = p[26:1];
66.
67.         /************************/
68.
69.     endmodule
```

由于操作数 A 和 B 的位宽做了不少的减少，所以 booth 乘法器 2 也必须执行相关的修改。具体上什么内容被修改时换汤不换药的，读者自己看这着办吧。不过笔者可以优惠一点，操作数 A 和 B 的位宽均为 14 位，其中最高位是符号位 ... 换句话说，只有 13 位参与运算，最终一次性的乘法操作总共用了 16 个时钟。再啰嗦一次，关于第 58~59 行的步骤 15 是步骤的返回操作。它非常重要，少了它功能会跑飞

float_multiplier_module_light2.v

```
1.    module float_multi_module_light2
2.    (
3.        input CLK, RSTn,
4.        input [31:0]A,B,
5.        output [31:0]Result,
6.
7.        input Start_Sig,
8.        output [3:0]Done_Sig,
```

```
9.
10.         /*****************/
11.
12.         output [21:0]SQ_rA,SQ_rB,
13.         output [25:0]SQ_Temp,
14.         output [9:0]SQ_rExp,SQ_BDiff,
15.         output [3:0]SQ_i
16.     );
```

第 3~15 行是相关的输入输出声明，基本上没有什么改变 ...

```
17.         /*************************************/
18.
19.         reg [3:0]i;
20.         reg [21:0]rA,rB;    // [21]Sign, [20:13]Exponent, [12]Hidden Bit, [11:0] [22..11]Mantissa
21.         reg [25:0]Temp;     // Mantissa Multi Result
22.         reg [31:0]rResult;
23.         reg [9:0]rExp,BDiff;        //[9:8] Overflow or underflow check, [7:0] usuall exp.
24.         reg isSign;
25.         reg isOver,isUnder,isZero;
26.         reg isDone;
27.         reg isStart;    // Enable the booth multiplier
28.         reg [4:0]C1;    // Counting Enbaled clock
29.
```

第 19~28 行是相关的寄存器声明，第 20 行的 rA 和 rB，由于笔者减去其中的 11 位，所以它们也行原本 33 位变成 22 位。换句话说，笔者将操作数 A 与 B 的尾数[11:0]都砍掉！哎~多么冷血的人；第 21 行的 Temp 也从原本的 48 位变成 26 位；其他的依然依旧。

```
30.         /*************************************/
31.
32.         wire [25:0]Product;
33.
34.         booth_multiplier_module_2 U1
35.         (
36.             .CLK( CLK ),
37.             .RSTn( RSTn ),
38.             .Start_Sig( isStart ),
39.             .A( {1'b0,rA[12:0]} ),
40.             .B( {1'b0,rB[12:0]} ),
41.             .Product( Product )
42.         );
43.
```

等价 32~42 行是 booth 乘法器 2 的实例化，其中第 39~40 行业做了相关的修改，rA 和 rB 它们也从原本的 [23:0] 变成 [12:0]，但是符号位依旧。

```
44.         /****************************************/
45.
46.         always @ ( posedge CLK or negedge RSTn )
47.             if( !RSTn )
48.                 begin
49.                     i <= 4'd0;
50.                     rA <= 22'd0;
51.                     rB <= 22'd0;
52.                     Temp <= 26'd0;
53.                     rResult <= 32'd0;
54.                     rExp <= 10'd0;
55.                     BDiff <= 10'd0;
56.                     isOver <= 1'b0;
57.                     isUnder <= 1'b0;
58.                     isZero <= 1'b0;
59.                     isDone <= 1'b0;
60.                     isStart <= 1'b0;
61.                     C1 <= 5'd0;
62.                 end
```

第 49~61 行是相关的寄存器声明。

```
63.         else if( Start_Sig )
64.                 case( i )
65.
66.                     0: // Initial and resulted out Signature
67.                     begin
68.                         rA <= { A[31], A[30:23], 1'b1, A[22:11]};
69.                         rB <= { B[31], B[30:23], 1'b1, B[22:11]};
70.                         isSign <= A[31] ^ B[31];
71.
72.                         isOver <= 1'b0; isUnder <= 1'b0; isZero <= 1'b0;
73.                         i <= i + 1'b1;
74.                     end
75.
```

在步骤 0（第 64~74 行）里中的第 68~69 行，rA 和 rB 再也不是寄存尾数的[22:0]，而是尾数的[22:11]。

```
76.                     1:
77.                     begin
```

```
78.                 if( A[30:23] == B[30:23] && ( A[22:0] != 23'd0 & B[22:0] != 23'd0 ) || A[31] ^ B[31] )
79.                     begin rB[20:13] <= rB[20:13] + 1'b1; rB[12:0] <= rB[12:0] >> 1; end
80.                 i <= i + 1'b1;
81.             end
82.
83.             2: // if rExp[9..8] is 1, mean A.Exp small than B.Exp
84.                 // while rExp[9..8] is 0, mean A.Exp large than B.Exp or same.
85.             begin
86.                 BDiff = rB[20:13] - 8'd127; // + (~8'd127 + 1)
87.                 rExp <= rA[20:13] + BDiff; // A.Exp + B.Exp
88.                 i <= i + 1'b1;
89.             end
90.
```

第 76~81 行的步骤 1 和第 83~89 行的步骤 2，由于 rA 和 rB 产生了改变，所以也作了相关的修改，读者自己看着办吧。

```
91.             3: // Enable boothe multiplier
92.             if( C1 == 16-1 ) begin C1 <= 5'd0; i <= i + 1'b1; end
93.             else begin C1 <= C1 + 1'b1; isStart <= 1'b1; end
94.
95.             4:
96.             begin Temp <= Product; isStart <= 1'b0; i <= i + 1'b1; end
97.
```

步骤 3 是用来使能 booth 乘法器 2，根据 Booth 算法的特性，随着位宽改变循环操作的次数也会随之减少。其中 booth 乘法器 2 需要 16 个时钟，因此 isStart 也拉高 16 个时钟。步骤 4 是用来拉低和关闭 booth 乘法器，并且将结果寄存在 Temp 里。

```
98.             5: // Check M'hidden bit
99.             begin
100.                if( Temp[25] == 1'b1 ) begin Temp <= Temp; end // do nothing
101.                else if( Temp[24] == 1'b1 ) begin Temp <= Temp << 1; end
102.                else if( Temp[23] == 1'b1 ) begin Temp <= Temp << 2; rExp <= rExp - 1'b1;end
103.
104.                i <= i + 1'b1;
105.            end
106.
107.            6: //error check and decide final result
108.            begin
109.                if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end       // E Overflow
110.                else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // E Underflow
111.                else if( Temp[25:2] == 24'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // M Zero
112.                else if( Temp[1] == 1'b1 ) rResult <= { isSign, rExp[7:0], Temp[24:2] + 1'b1 }; // okay with normalised
```

```
113.                    else rResult <= { isSign, rExp[7:0], Temp[24:2] }; // okay without normalise
114.                    i <= i + 1'b1;
115.                end
116.
```

至于步骤 5 和 6 在 Temp 的位宽上也作了相应的修改，读者自己理解吧 ... 很简单而已，都是换汤不换药的工作。

```
117.                7:
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
124.
125.    /***************************************/
126.
127.    assign Done_Sig = { isOver, isUnder, isZero, isDone };
128.    assign Result = rResult;
129.
130.    /***************************************/
131.
132.      assign SQ_rA = rA;
133.      assign SQ_rB = rB;
134.      assign SQ_rExp = rExp;
135.      assign SQ_BDiff = BDiff;
136.      assign SQ_Temp = Temp;
137.      assign SQ_i = i;
138.
139.    /***************************************/
140.
141. endmodule
```

最后的第 117~137 行基本上没有什么改变~笑。

仿真结果：

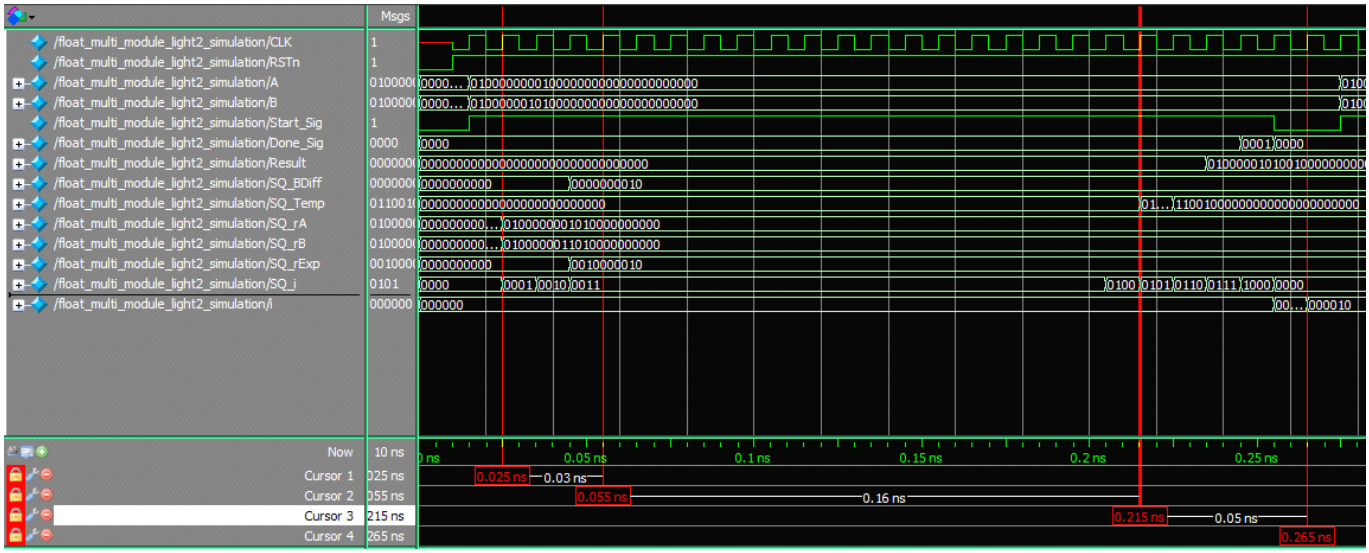


图 5.6.2 仿真结果。

图 5.6.2 是仿真结果，它同样用了和实验二十二一样的激励文件。光标 C1~C4 是一次性浮点数乘法模块所使用的时间，一共是 3+16+5 = 24 个时钟（一个时钟是 10ps）。其中 C2~C3 的 16 个时钟是 booth 乘法器 2 所占去的时间，其余的 C1~C2 + C3~C4 是浮点数乘法模块所占去的时间。实验二十八比起实验二十七无论是时钟消耗数的减少，当经过编译以后逻辑资源的消耗数也从原本的 399 减少为 298。眼见这是可喜可贺的结局 ... 实际上可怕的后遗症却留在后边！

表 5.6.1 失去精度和未失去精度之间的结果比较。

问题	True Result	未失去精度	失去精度
0.0078125 * 0.0057488	0.0000449125	0.0000449	0.0000449
3.1828 * -0.0063	-0.02005164	-0.0200516	-0.0200504
6.631 * 0.25	1.65775	1.6577500	1.6577148
6.3565 * -0.0063	-0.04004595	-0.0400459	-0.0400454
273.757 * 483.265	132297.176605	132297.1718750	132289.6875000

表 5.6.1 是一部分的运算结果，自己打开雪亮的双眼看看吧，尤其是 273.757 * 483.265 的运算结果最为显眼的，其中失去精度和未失去精度的结果之间有很大的差别 ... 貌似其他的还可以接受。嗯，这也是没有办法的事情，我们为了进一步节省空间，这点的代价是不可不付出的。

5.7 实验二十九：轻量版-浮点数除法模块

在这个实验里，基于实验二十三的浮点数除法模块，我们要建立一个轻量版的浮点数除法模块。首先我们必须找到使浮点数除法模块臃肿的原因。具体上，驱使浮点数除法模块臃肿有两个原因：第一原因就是保护精度的“马力”问题；第二问题就是“/”操作符。浮点除法在保护精度的措施上，与其他浮点运算有点不同，因此我们就不对“马力”下手了。换之，“/”操作符就要遭殃。

接下来我们需要回顾时序篇，寻找任何有循环操作模式的除法器 ... 想了想，只有 streamlined_divider_module.v 适合而已。我们先简单的复习一下该除法器的特性：它和 booth 算法一样，随着操作数的位宽增加，循环次数也会增加；此外，它还是包含符号位来运算。至于详细的功能，读者自己复习一下时序篇吧。

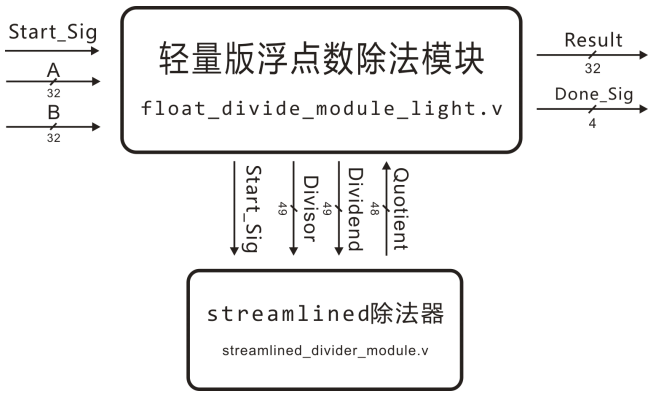


图 5.7.1 轻量版浮点数除法模块的建模图。

图 5.7.1 的建模图是实验二十九里要建立的东西，图 5.7.1 与轻量版浮点数乘法模块的建模图相比，似乎自是字面上的修改而已！？原本 Streamlined 除法器在完成一次性的除法以后，它会反馈商数—Quotient 和余数—Reminder。由于实验二十九里用不着余数，所以就把 Reminder 的输出连线无视了。图 5.7.1 基本上没有什么好说，但是除了 Dividend 和 Divisor 的位宽以外。原本 Dividend 和 Divisor 的位宽均为 24 才是，为了压缩精度的流失，在不得已的情况下才把位宽的马力放大到 48 位。其中[49]位是为了配合 Streamlined 除法器的特性才加上去 符号位（0 值）。

streamlined_divider_module.v

```
module streamlined_divider_module
(
        input CLK,
        input RSTn,

        input Start_Sig,
        input [48:0]Dividend,
        input [48:0]Divisor,

        output [47:0]Quotient,
```

```
11.         output [47:0]Reminder
12.     );
```

第 3~11 行是该模块的输入输出声明。

```
13.         /*****************************/
14.
15.         reg [1:0]i;
16.         reg [49:0]s;
17.         reg [97:0]Temp;
18.         reg [97:0]Diff;
19.         reg [5:0]C1;
20.
21.         always @ ( posedge CLK or negedge RSTn )
22.             if( !RSTn )
23.                 begin
24.                     i <= 2'd0;
25.                     s <= 50'd0;
26.                     Temp <= 98'd0;
27.                     Diff <= 98'd0;
28.                     C1 <= 6'd0;
29.                 end
```

第 15~19 行是相关的寄存器声明，其中第 19 行的 C1 是为了计时用，至于第 24~28 行是复位操作。

```
30.             else if( Start_Sig )
31.                 case( i )
32.
33.                     0:
34.                     begin
35.                         s <= Divisor[48] ? { 1'b1, Divisor } : { 1'b1 , ~Divisor + 1'b1 };
36.                         Temp <= Dividend[48] ? { 49'd0 , ~Dividend + 1'b1 } : { 49'd0 , Dividend };
37.                         Diff <= 98'd0;
38.                         i <= i + 1'b1;
39.                     end
40.
41.                     1:
42.                     begin
43.                         Diff = Temp + { s , 48'd0 };
44.
45.                         if( Diff[97] ) Temp <= { Temp[96:0] , 1'b0 };
46.                         else Temp <= { Diff[96:0] , 1'b1 };
47.
48.                         if( C1 == 49 -1) begin C1 <= 6'd0; i <= i + 1'b1; end
```

```
49.                                 else C1 <= C1 + 1'b1;
50.                             end
51.
```

步骤 0 是相关的初始化操作，具体内容和时序篇里的实验二十八差不多，由于被乘数和乘数的位宽发生改变，所以步骤 0 的内容也要发生换汤不换药的修改。其中笔者直接略掉了 isNeg 的取得运算，原因很简单，被乘数和乘数任何时候都是正直，所以 isNeg 的取得运算就显得鹌鹑了。步骤 1 是主要的循环操作，被乘数和乘数的位宽均为 49 位，所以它必须循环操作 49 次。在此，笔者不再滥用步骤 i 了，取而代之笔者用 C1 控制步骤 1 的循环次数。

```
52.                         2:
53.                         begin i <= 2'd0; end
54.
55.                         endcase
56.
57.         /*********************************/
58.
59.             assign Quotient = Temp[47:0];
60.             assign Reminder = Temp[96:49];
61.
62.             /**********************************/
63.
64.     endmodule
```

第 52~53 行的步骤 2 主要是用来返回步骤，这行很重要，少了它功能会跑飞。至于第 59~60 行是商数和余数的输出驱动，商数对应 Temp[48:0]，其中笔者把 Temp[48]的符号位省略掉了，因为 Temp 中的符号位是无用的；然而余数对应 Temp[97:49]，其中 Temp[97]的符号位被略掉，原因一样。最后，关于一次性除法操作一共用了 51 个时钟。

float_divide_module_light.v

```
1.      module float_divide_module_light
2.      (
3.          input CLK, RSTn,
4.          input [31:0]A,B,
5.          output [31:0]Result,
6.
7.          input Start_Sig,
8.          output [3:0]Done_Sig,
9.
10.         /*****************/
11.
12.         output [32:0]SQ_rA,SQ_rB,
13.         output [47:0]SQ_Temp,
```

```
14.         output [9:0]SQ_rExp,SQ_BDiff
15.
16.     );
```

第 3~8 行是相关的出入输出声明，至于第 12~15 行是仿真用的输出声明。

```
17.         /**************************************/
18.
19.         reg [3:0]i;
20.         reg [32:0]rA,rB;    // [32]Sign, [31:24]Exponent, [23]Hidden Bit, [22:0]Mantissa
21.         reg [47:0]Temp;     // [47] Hidden-A [46:24] Mantissa-A , [23] Hidden-B [22:0]Mantissa-B
22.         reg [31:0]rResult;
23.         reg [9:0]rExp,BDiff;        //[9:8] Overflow or underflow check, [7:0] usuall exp.
24.         reg [5:0]C1;
25.         reg isStart;
26.         reg isSign;
27.         reg isOver,isUnder,isZero,isDone;
28.
```

第 19~27 行是相关的寄存器声明，其中第 24 行的 C1 是用控制 Streamlined 除法器的计时用，然而第 25 行的 isStart 是用来使能该除法器。

```
29.         /**********************/
30.
31.         wire [47:0]Quotient;
32.
33.         streamlined_divider_module U1
34.         (
35.             .CLK( CLK ),
36.             .RSTn( RSTn ),
37.             .Start_Sig( isStart ),
38.             .Dividend( { 1'b0, rA[23:0], 24'd0 } ),
39.             .Divisor( { 1'b0, 24'd0, rB[23:0] } ),
40.             .Quotient( Quotient ),
41.             .Reminder()
42.         );
43.
```

第 31~42 行是 streamlined_divider_module 的实例化，第 38 行是被除数的驱动，其中 rA[23:0]拉大 24 位宽的马力，随之又补上无用的符号位 1'b0。第 39 行是除数的驱动，它是有 rB[23:0] 再加上无用的符号位。至于第 41 行的余数，它被宫了。

```
44.         /**********************/
45.
```

```
46.     always @ ( posedge CLK or negedge RSTn )
47.         if( !RSTn )
48.             begin
49.                 i <= 4'd0;
50.                 rA <= 33'd0;
51.                 rB <= 33'd0;
52.                 Temp <= 48'd0;
53.                 rResult <= 32'd0;
54.                 rExp <= 10'd0;
55.                 BDiff <= 10'd0;
56.                 C1 <= 6'd0;
57.                 isStart <= 1'b0;
58.                 isOver <= 1'b0;
59.                 isUnder <= 1'b0;
60.                 isZero <= 1'b0;
61.                 isDone <= 1'b0;
62.             end
```

第 49~61 行是相关的复位操作。

```
63.         else if( Start_Sig )
64.             case( i )
65.
66.                 0: // Initial and resulted out Signature
67.                 begin
68.                     rA <= { A[31], A[30:23], 1'b1, A[22:0] };
69.                     rB <= { B[31], B[30:23], 1'b1, B[22:0] };
70.
71.                     isSign <= A[31] ^ B[31];
72.                     isOver <= 1'b0; isUnder <= 1'b0; isZero <= 1'b0;
73.                     i <= i + 1'b1;
74.                 end
75.
76.                 1: // if rExp[9..8] is 1, mean A.Exp small than B.Exp
77.                    // while rExp[9..8] is 0, mean A.Exp large than B.Exp or same.
78.                 begin
79.                     BDiff = rB[31:24] - 8'd127; // + (~8'd127 + 1)
80.                     rExp <= rA[31:24] - BDiff; // A.Exp - B.Exp
81.                     i <= i + 1'b1;
82.                 end
83.
```

步骤 0~1 基本上和实验二十三的内容一模一样，恕笔者不重复了。

```
84.                 2: // Divide and A.m << 24
85.                 if( C1 == 51 -1) begin C1 <= 6'd0; i <= i + 1'b1; end
86.                 else begin C1 <= C1 + 1'b1; isStart <= 1'b1; end
87.
88.                 3:
89.                 begin Temp <= Quotient; isStart <= 1'b0; i <= i + 1'b1; end
90.
```

第 84~86 行的步骤 2 是用来使能 streamlined 除法器，并且即时 ... 第 88~89 行的步骤 3，除了关闭该除法器以外，它还将取得的商数寄存在 Temp 里。

```
91.                 4: // Check M'hidden bit
92.                 begin
93.                     if( Temp[25] == 1'b1 ) begin Temp <= Temp << 22; end
94.                     else if( Temp[24] == 1'b1 ) begin Temp <= Temp << 23; end
95.                     else if( Temp[23] == 1'b1 ) begin Temp <= Temp << 24; rExp <= rExp - 1'b1; end
96.
97.                     i <= i + 1'b1;
98.                 end
99.
100.                5: //error check and decide final result
101.                begin
102.                    if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end       // E Overflow
103.                    else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // E Underflow
104.                    else if( Temp[47:24] == 24'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // M Zero
105.                    else rResult <= { isSign, rExp[7:0], Temp[46:24] }; // okay without normalise
106.                    i <= i + 1'b1;
107.                end
108.
109.                6:
110.                begin isDone <= 1'b1; i <= i + 1'b1; end
111.
112.                7:
113.                begin isDone <= 1'b0; i <= 4'd0; end
114.
115.            endcase
116.
117.    /****************************************/
118.
119.    assign Done_Sig = { isOver, isUnder, isZero, isDone };
120.    assign Result = rResult;
121.
122.    /****************************************/
123.
```

```
124.        assign SQ_rA = rA;
125.        assign SQ_rB = rB;
126.        assign SQ_rExp = rExp;
127.        assign SQ_BDiff = BDiff;
128.        assign SQ_Temp = Temp;
129.
130.      /****************************************/
131.
132.  endmodule
```

接下来的第 91~128 行，内容还是和原来的实验二十三一样。

仿真结果：

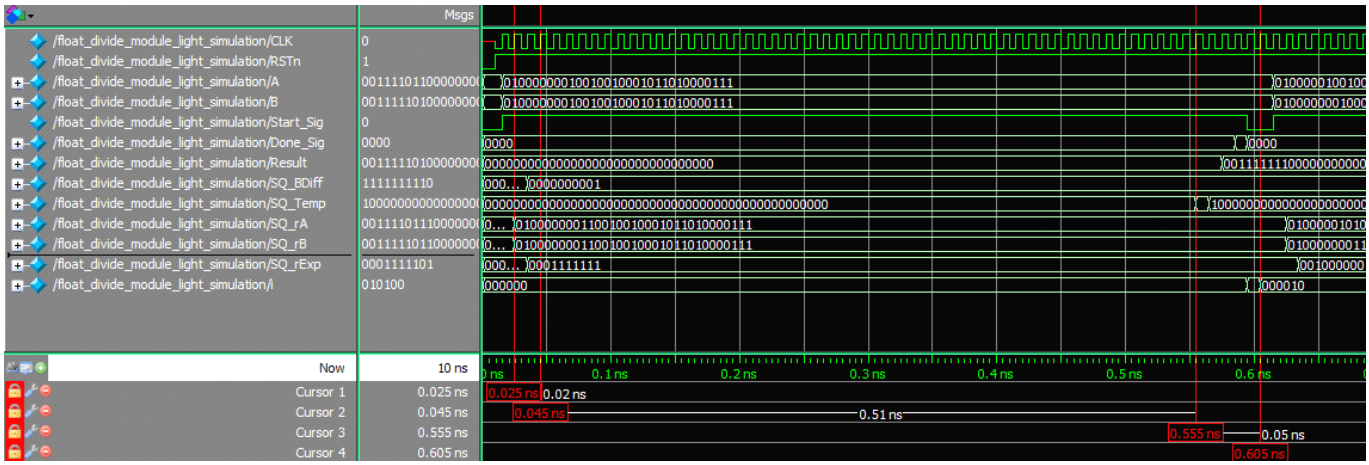


图 5.7.2 仿真结果。

图 5.7.2 是仿真结果，光标 C1~C4 是一次性的浮点除法操作，一共用了 2 + 51 + 5 = 58 个时钟（一个时钟 10ps）。其中 C2~C3 是 streamlined 除法器的一次性操作，一共占了 51 个时钟，至于 C1~C2 和 C3~C4 总和占了 7 个时钟。作为普通用的浮点除法器来说，58 个时钟消耗多少也算慢了一点，不过考虑到浮点除法精度保护措施的特殊性，为了不失精度，这也是在所难免的。

经过编译以后，逻辑资源的消耗比起实验二十三的 2057，实验二十九的结果是 433 ... 又是夸张的空间节省比率。如果多消耗一点时钟可以得到如此的结果，作为一个倾向节省逻辑资源的优化，结果是平衡而且是可以接受的。接下来，笔者就不作丢精度来提高空间节省的实验了，原因笔者在前面已经说了，浮点除法保护精度的措施和其他浮点运算不一样。不过读者自己可以尝试看看 ... 加油！

==========

在此，基本上有关“循环操作”形态的讨论已经到此为止了，循环操作原本就是软件操作专用的，对于使用软件操作的处理器来说，嵌入多一个硬件资源会导致一动牵万山的结果，所以重复使用一个硬件资源才是上上之策。反过来站在 Verilog HDL 语言的角度来思考，要建立多一个硬件资源是非常简单的事，不过考虑到逻辑资源的限制 ... 很多时候我们不得不采取循环操作的操作模式，用

时钟换空间，亦即重复使用同一个步骤。

当然循环操作采用范围一点也不广，除了相关的算法以外，一般上它出场的机会不见得那么多。不过对于 Verilog 来说，循环操作是不失一个好东西尤其是空间紧张的环境下。最后有关循环操作的实现，实际上一点都不难，只要我们有一个“初模型”适当的修改一下内容就行了。在此，我们也可以说“初模型倾向节省逻辑资源”的优化，实际上是从一种形态的模式向另一中形态的模式转化而已。

在最后，我们可以得到这写结论：循环操作可以从初模型转换；节省逻辑资源的优化有两个，一是重复使用一个步骤，二是降低产生质量（在狗急跳墙的情况下）。此外，我们还知道从初模型到循环操作的转换不可能是 100%的，因为我们还要考虑其他因数，以致优化结果达到一种平衡的结局。举个例子来说，我们不可能为了节省逻辑资源，而丢失输出质量，或者降低模块的表达能力，虽然在某种程度上我们是赚到的，但是失去大于取得，结果还是吃大亏。

所以说，当我们要执行某种倾向的优化时，应该平衡考量 ... 往往把模块的表达能力放在第一位，输出质量第二位，第三才是优化的程度。我们是人，任何时候我们都要清楚自己在干什么，做错以后惭愧以后，还可以重新来过，结果和意图都不重要。人是如此，Verilog 是如此，模块也是如此。

5.8 再谈步骤和时钟的整合

步骤和时钟的整合原本是整合概念的基本，举个例子：

```
case( i )
    0:
    begin Act0 <= Act0 + 1'b1; i <= i + 1'b1; end

    1:
    begin Act1 <= Act1 + 1'b1; i <= i + 1'b1; end

    ....
```

从上面的例子中，我们知道步骤 0 和步骤 1 中 Act0 与 Act1，无论怎么看它们，我们都知道它们是互相独立的。两个互相独立的东西各自都使用一个时钟都完成递增的动作，为何两个独立的操作不能再一个时钟里边完成呢？如：

```
case( i )
    0:
    begin Act0 <= Act0 + 1'b1; Act1 <= Act1 + 1'b1; i <= i + 1'b1; end
    ...
```

虽然在外人的眼里，任何独立的操作都可以整合在相同的时钟/步骤之内。不过要细谈类似的问题，我们不可以单单只是从外表上去判断一个整合的操作。这话要怎么讲呢？笔者再举个例子：当我们在用餐的时候，我们虽然知道一边嚼着饭，当然可以一边喝水 ... 实际上嚼饭和喝水是独立的操作。但是从几千年前，先祖大大们就告诉子孙，水要等到饭吃完后才能饮用，不然会伤到胃。

祖先的智慧是绝对有科学根据的，首先我们知道口腔是人体消化系统的一部分，我们在口腔里嚼饭时，从口腔各处会分泌各种消化液开始第一段的消化工作。如果此时喝水，会降低消化液的密度时而降低消化效率。此外我们的食道拥有缓冲食物的作用，亦即给胃起到一定操作时间。喝水会减少缓冲作用，因此胃部可用的操作时间一定会被减少。最后，当食物在胃中等待被消化的时候，胃部会分泌消化液来增加酸性（降低 pH 值）。如果此时喝水，就会提高 pH 值 ... 人体的胃部是非常智能的，它会分泌更多的消化液来保持环境的酸性。假设那个人是胃部是非常敏感的话，那么肚子会阵阵作疼，最后就会冲向厕所。

当然，把相关独立的步骤整合起来没有人体中那么严重 ... 笔者只是举个接近事实的例子而已。转回正题，类似的情况可以用浮点除法来说话：

```
48.         0: // Initial and resulted out Signature
49.         begin
50.             rA <= { A[31], A[30:23], 1'b1, A[22:0] };
51.             rB <= { B[31], B[30:23], 1'b1, B[22:0] };
```

```

            isSign <= A[31] ^ B[31];
            isOver <= 1'b0; isUnder <= 1'b0; isZero <= 1'b0;
            i <= i + 1'b1;
        end

        1: // if rExp[9..8] is 1, mean A.Exp small than B.Exp
           // while rExp[9..8] is 0, mean A.Exp large than B.Exp or same.
        begin
            BDiff = rB[31:24] - 8'd127; // + (~8'd127 + 1)
            rExp <= rA[31:24] - BDiff; // A.Exp - B.Exp
            i <= i + 1'b1;
        end

        2: // Divide and A.m << 24
        begin
            Temp <= { rA[23:0] , 24'd0} / { 24'd0 , rB[23:0] };
            i <= i + 1'b1;
        end
```

上面一段代码是实验二十三 float_divide_module.v 中的步骤 0~2。从理论来说，符号预算，价码运算和尾数运算是保持独立关系的，因为它们可以实现整合，如：

```
        0:
        begin
            rA <= { A[31], A[30:23], 1'b1, A[22:0] };
            rB <= { B[31], B[30:23], 1'b1, B[22:0] };

            isOver <= 1'b0; isUnder <= 1'b0; isZero <= 1'b0;
            i <= i + 1'b1;
        end

        1:
        begin
            isSign <= A[31] ^ B[31];

            BDiff = rB[31:24] - 8'd127; // + (~8'd127 + 1)
            rExp <= rA[31:24] - BDiff; // A.Exp - B.Exp

            Temp <= { rA[23:0] , 24'd0} / { 24'd0 , rB[23:0] };
            i <= i + 1'b1;
        end
```

上面一段代码是经过整合的结果，步骤 0 是操作数预处理，然而步骤 1 是 3 个运算同时发生。

从整合的结果中，我们减少了一个时钟 ... 不过，从上面几个小节的学习，我们学会懂得用平衡来思考问题和意图。此时的浮点除法，已经无法和浮点加减法，浮点乘法等之间实验的步骤，在理解上无法同一起来了。如果只是从片面的角度来看待问题，这样的整合结果是无伤大碍；反之，如果从全面的角度来看待问题的话，结局我们是得还是失呢？当然我们可以更过分去整合上面的代码，如：

```
0:
begin
    isSign <= A[31] ^ B[31];

    BDiff = B[30:23] - 8'd127; // + (~8'd127 + 1)
    rExp <= A[30:23] - BDiff; // A.Exp - B.Exp

    Temp <= { 1'b1, A[22:0] , 24'd0} / { 24'd0 , 1'b1, B[22:0] };

     isOver <= 1'b0; isUnder <= 1'b0; isZero <= 1'b0;
     i <= i + 1'b1;
end
```

我们可以直接无视操作数预处理的步骤，然后 3 个独立的运算直接在同一个步骤和时钟内并行执行，结果整合的程度有进一步提高。如果我们只是设计单个浮点数除法器而已的话（片面），那是 Happy Ending 的结局；反之，如果设计是围绕：浮点数加法器，浮点数减法器，浮点数乘法器和浮点数除法器的话（全面），结果就也不能这样算了。因为我们无法使用同一种理解的方式，去设计 4 种浮点数的运算器。这种感觉如同得一失三。

不知道读者是否明白笔者在讲什么？能否接受笔者的想法？不过不要紧，每个人都有自己的看法，每个人都有一套衡量平衡感的道具。虽然在步骤和时钟的整合上取得最大的程度是一件好事，不过有时候我们还是要考虑其他的因数。追求平衡才是王道，不知道读者有没有听过这句话呢？

整合相关独立的操作在一个步骤或者在时钟里，是小偏向减少操作时钟的优化。实际上还有更过分的整合行为，那是一种极端偏向减少时钟的优化。任何操作都可以压缩在一个步骤里，不过那是带有风险的赌博行为。除非有了必死的觉悟，不然的话是不推荐好孩子去尝试 .....

5.9 组合逻辑和即时操作

在时序篇里，我们可以知道在理想时序中有“时间点事件”和“即时事件”两种不同的发生事件。时间点事件说白了是时间沿触发的时序事件，特征有过去值和未来值之分，它是 RTL 级时序的主要。反之，即时事件是由组合逻辑引起的时序事件，特征是无视时钟的存在。用笔者的话来说，即时事件是无法反映在正规的时序图上，因为即时事件是发生在时间点之中（即时空间）。

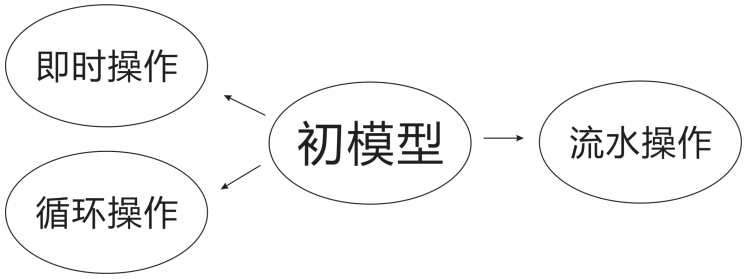


图 5.9.1 各种形态的操作模式。

那么，什么是即时操作呢？即时操作时另一种形态的操作模式，它同样会引发即时事件，不过效果要大一些。读者可能觉得很奇怪：“组合逻辑能引起即时事件，即时操作也能引起事件，它们之间到底有什么不同呢？”，当笔者一谈起组合逻辑，读者是不是会联想到 logic gate，half adder/full adder，flip-flop/ RS latch，比较器等这些在课本上学到的东西？

我们在课本上所学到的东西确确实实也是基于组合逻辑，但是它们只能成为小功能的东西，所以也只能引发小效果的即时事件。当我们谈到即时操作，虽然它的基础也是组合逻辑，但是比起那些小功能的家伙，“操作”这两个字可以给我们想象到“有一连串步骤”性质的东西，亦即拥有步骤性质的大功能组合逻辑。

那么，何为小功能和大功能呢？

```
always @ ( A or B ) //  组合逻辑 1
      case( {A,B} )
            2'b00: C = 4'b0001;
            2'b01: C = 4'b0010;
            2'b10: C = 4'b0100;
            2'b11: C = 4'b1000;

always @ ( * ) //  组合逻辑 2
      if( Start_U1 ) rQ = Q_U1;
      else if( Start_U2 ) rQ = Q_U2;
      else rQ = 1'bx;
```

小功能的组合逻辑除了在课本所学到的东西以外，一般上常见的小功能组合逻辑如上述几个模样。组合逻辑 1 好像是加码器的一种，根据 A 和 B 输入信号的变化，输出相关的加码结果。组合逻

辑 2 是低级建模中专用的输出选择器，功能笔者就不重复了。

在这里先插上不相干的话题：

```
1.  always @ ( A or B )
2.  always @ ( * )
3.  always @ ( posedge CLK )
4.  always @ ( posedge CLK or A or B )
```

笔者在扫盲文里曾经提到，组合逻辑级设计常见的开头 always 有如第 1~2 行，至于第 3 行的 always 是 RTL 级设计中常常出现的。至于第 4 行的 always 是人妖的行为，我们始终分不出它到底是组合逻辑级设计还是 RTL 级的设计？笔者一般上都非常不推荐这样的写法，如果我们要在 RTL 级设计中嵌入组合逻辑级的设计，我们还有更好的选择。

（回来话题）除此之外，小功能的组合逻辑一眼看下，它给人一种非常单纯又简单的感觉，换之大功能的组合逻辑又是什么一个样呢？笔者之所以说它“大”它并不一定很大，然而“大”是指“有步骤性质”的组合逻辑。换句话说，我们可以把整个初模型向即时操作的形态转化。读者：“笔者有没有搞错！你说整个初模型转换为组合逻辑！？”差不多是这样的意思 ... 在此，读者是否可以想象得到，一个从初模型转致即时操作的模块，在某种程度上是不是“大”功能的组合逻辑呢？

即时操作基本上和流水操作没有什么不同，前者是组合逻辑级的设计，后者则是 RTL 级的设计，这样说读者是否可以明白一些呢？不过即时操作和流水操作翟然是不同的东西，但是它们非常相似，如果读者懂得把初模型转至流水操作，自然而然懂得把初模型转至即时操作。此外，即时操作更偏向组合逻辑级的设计一些 ... 很遗憾，有关组合逻辑的东西笔者真的不想讲太多，因为现在还不是时机，不过一些零星而不伤大雅的内容还是可以讨论。

5.10 即时操作和建模

时序篇里我们曾经学过“流水操作和建模”，在这个小节里我们要学即时操作和建模之间的关系。为什么即使操作又和建模有关系呢？建模的好处笔者就不谈了，即使操作和流水操作非常的相识 ... 我们知道流水操作，每一个任务都可以用一个“模块任务”来代表。有时候同样的流水任务会重复出现许多次，如果用代码表达重复性的流水任务，不仅会使代码臃肿，而且表达能力也会低下不少。因此，多次实例化“任务模块”，不仅使外观好看不少，而且还可以使表达能力更接近具体。

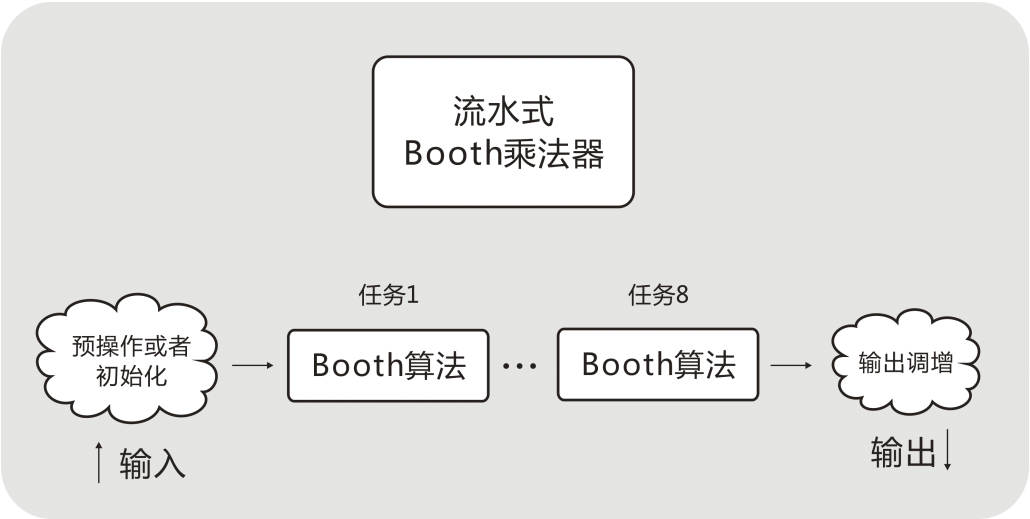


图 5.10.1 流水式 Booth 乘法器。

举个具体的例子，时序篇里的实验二十三—Booth 乘法器 2，经过向流水操作的转换以后，如图 5.10.1 所示。图 5.10.1 是时序篇里，实验三十三的流水操作的建模图（详细的内容请复习建模篇）。其中输入和输出的调整又组合逻辑来负责，反之从任务 1~8 是 8 次的重复 Booth 算法运算。在此，我们可以任务 1 为模型，然后建立一个任务模块，最后实例化 8 次。大致的代码如下：

```
module pipeline_booth_multiplier_module ( CLK，A，B，Product ) ;   //流水式 Booth 乘法器

    wire Out_T1;   //实例化任务模块 1
    Booth_Task T1( .In_T1( {A,B} )，Out_T1 )；

    wire Out_T2;   //实例化任务模块 2
    Booth_Task T2( In_T1，Out_T2 )；

    ....                //重复同样的动作

    wire Out_T8     //实例化任务模块 8
    Booth_Task T8( In_T7，Out_T8 ) ;

    assign Product = Out_T8;     //  输出驱动
```

```
endmodule

module Booth_Task( CLK, In, Out, ...); ... endmodule    //建立任务模块
```

笔者在前面说过，流水操作和即时操作差就差在前者是 RTL 级设计，后者则是组合逻辑级设计。换句话说，前者是时间点事件，后者是即时操作。用傻瓜的话来讲，前者有 CLK（时钟）后者则无 Clock。在这里，许多同学可能会有疑问："即时操作没有 CLK 它怎能像流水操作呢？"笔者已经说了，流水操作和即时操作在形式上相似而已，但是构筑即时操作的原理和方式却和流水操作非常不同。

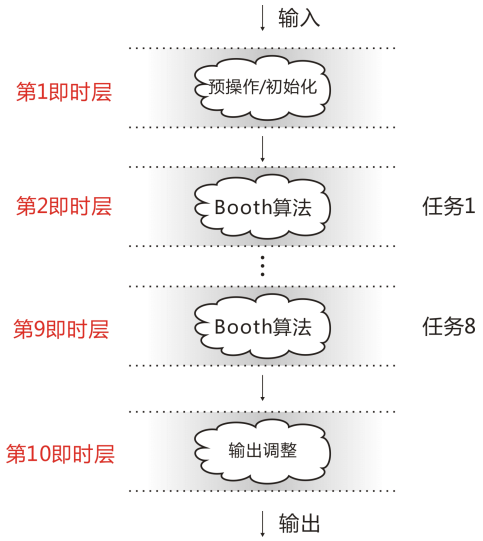


图 5.10.2 即时操作的即时层。

即时操作虽然没有时钟—CLK，但是它利用"即时层"来代表每一个"即时时间"的前后。图 5.10.2 也是流水式 Booth 乘法器转换位即时操作以后的建模图，在此我们可以清楚的看到，第 1 个即时层是预操作/初始化，第 2~9 个即时层（8 个即时层）是 booth 算法的 8 个任务模块，然而第 10 即时层则是输出调整。其中每一个任务都由"云状"的图形包裹着，在普遍的认识中"云状"的图形代表组合逻辑，这意味着什么？

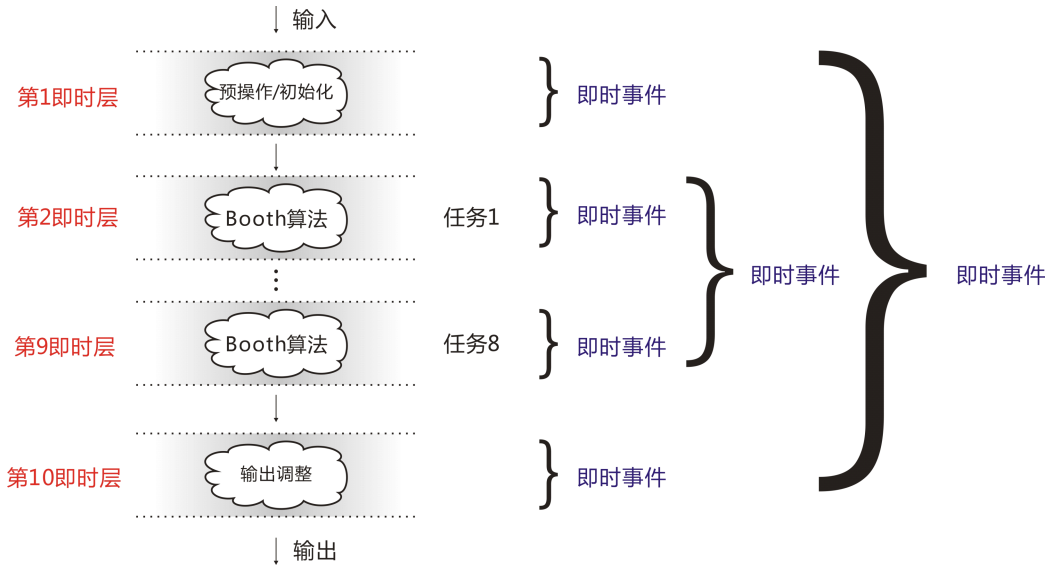


图 5.10.3 即时操作，即时层，即时事件的关系。

我们知道在理想时序中，即时事件的引起是无视时钟的存在，又或者说即时事件在一个时钟内完成。换句话说，如图 5.10.3 所示，任一一段很小的时间层都会引起一个即时事件，其外 Booth 算法中一任务 1~8 的 8 个即时层综合也是引起一个即时事件。Booth 算法的即时操作，全部总和起来的 10 个即时层，结果也是引起一个即时事件。把话说白一点，随便我们叠加几个或者几十个即时层，最终结果都会引起一个即时事件，而且任何即时事件都会在一个时钟内完成。

（在此，笔者需要强调一下：上述内容只是讲述即时事件在理想时序中的行为而已，如果把环境改为物理时序的话，那么理解就不同了。如果问为什么？怎样？什么？等问题，在后面的章节中笔者会慢慢讲解的，在这里我们只关心即时事件，即时操作，即时层和理想时序之间的关系而已）

“天呀！即时操作真的那么霸道！？”嗯，这是铁一般的事实 ... 好了，一些关键的内容我们已经了解了，接下来笔者还是用实验来说话吧。不过在实验开始之前，笔者还是先简单的讲解一下，一些有关组合逻辑的简单知识。

一般上组合逻辑的声明有一下几种：

```
1.  Temp = A + B;
2.  always @ ( a or b ) .... ;
3.  always @ ( * ) .... ;
4.  wire Q = A||B;
5.  assign Output = RegA == 4'd10 ? 1'b1 : 1'b0;
6.  assign Output = A && B && C;
```

（一）第一种，组合逻辑建成使用了关键的“=”阻塞赋值，用理想时序的话来讲就是使用了引起即时事件的赋值操作符。
（二）第二种，主要是在 always 的敏感区域里声明了相关的东西。
（三）第三种，和第二种差不多，不过使用 * 代表“以下是组合逻辑”的意思。
（四）第四种，wire 类型的 Q 主要由非纯 RTL 级的寄存器驱动，亦即 Q 由组合逻辑驱动。

（五）第五和六种，与第四种差不多，输出驱动源主要不是 纯 RTL 级的寄存器，亦即 Output 由组合逻辑驱动。

基本上，以上 6 中情况都会在不知不觉中声明到组合逻辑 ... 事实上组合逻辑的声明范围更为广阔，不过以上六种更能影响时序的变现而已。具体的内容，以后有机会笔者会慢慢讨论的。在此先知道个大概先 ...

实验三十：即时式 Booth 乘法器

```
module booth_multiplier_module_2 ( ... );
......
                    0:
                    begin
                        a <= A;
                        s <= ( ~A + 1'b1 );
                        p <= { 8'd0 , B , 1'b0 };
                        Diff1 <= 8'd0;
                        Diff2 <= 8'd0;

                        i <= i + 1'b1;
                    end

                    1,2,3,4,5,6,7,8:
                    begin
                        Diff1 = p[16:9] + a;
                        Diff2 = p[16:9] + s;

                        if( p[1:0] == 2'b01 ) p <= { Diff1[7] , Diff1 , p[8:1] };
                        else if( p[1:0] == 2'b10 ) p <= { Diff2[7] , Diff2 , p[8:1] };
                        else p <= { p[16] , p[16:1] };

                        i <= i + 1'b1;
                    end
 ......
 assign Product = p[16:1];
```

首先，我們先來回憶一下 8 位 booth 乘法器 2 的一些重要步驟。步驟 0 是预处理或者预操作的动作，亦即取得 A 的正直和 S 亦即操作数 A 负值的补码形式，随后初始化一下 P 空间。步骤 1~8 是重复 8 次的 Booth 算法运算，最后一个不起眼的步骤就是 Product 的输出驱动。

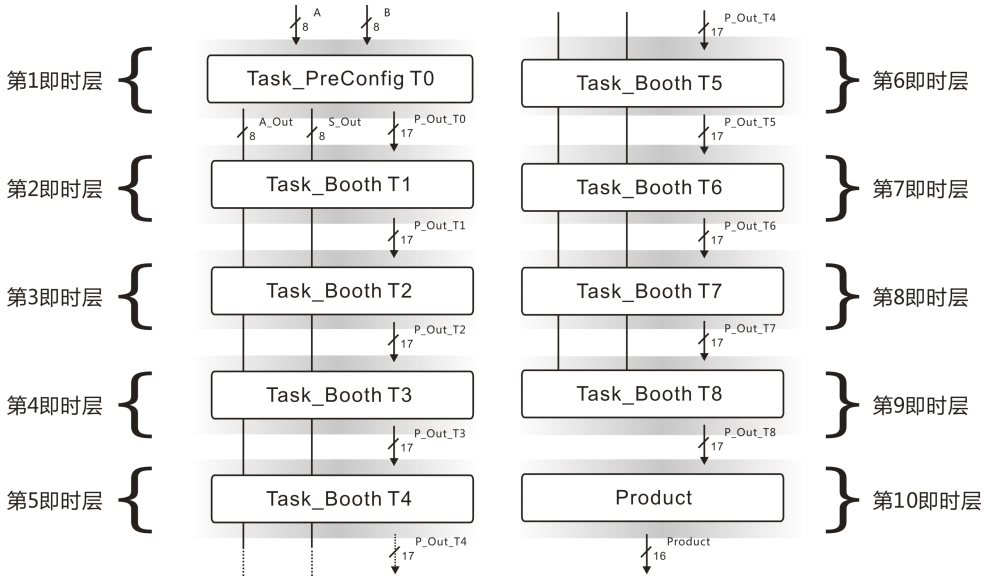


图 5.10.4 即时式 Booth 乘法器的建模图。

图 5.10.4 是即时式 Booth 乘法器的建模图，其中有两个任务模块，第一就是 Task_PreConfig，第二就是 Task_Booth，至于 Product 只是单纯的输出驱动。其中我们可以看到 Task_PreConfig 任务模块被实例化 1 次，反之 Task_Booth 任务模块被实例化 8 次。Task_PreConfig 的作用是初始化或者预操作操作数 A 与 B，功能如同 booth_multiplier_module_2 的步骤 0 一样，就是取得 A 和 S 以后，还将初始化以后的 P 空间输出，既是 A_Out，S_Out 和 P_Out_T0。

Task_Booth 是执行 Booth 算法运算的任务模块，它将输入的 P 运算以后，将结果 P 输出给下一个同样的任务模块，亦即 P_Out_T0，P_Out_T1 ... P_Out_T8。最后的 Product 很单纯，它由 P_Out_T8[16:1]驱动。好了，我们只要按照图 5.10.4 的图形建立 immediate_booth_multiplier_module 既可以。（至于为什么图 5.10.4 不是云状的图形 ... 云状的图形要绘制起来有点不方便，不过读者只要记得每一个即时层的任务都的基础都是组合逻辑就可以了）

immediate_booth_multiplier_module.v

```
module Task_PreConfig
(
        input [7:0]A,B,
        output [7:0]A_Out,S_Out,
        output [16:0]P
);

        assign A_Out = A;
        assign S_Out = ( ~A + 1'b1 );
        assign P = { 8'd0 , B , 1'b0 };

endmodule
```

第 42~53 行是 Task_PreConfig 任务模块的建立，内容很简单，自己看着办吧。

```
55.  module Task_Booth
56.  (
57.      input [16:0]P_In,
58.      input [7:0]A_In,S_In,
59.
60.      output [16:0]P_Out
61.  );
62.     /***************/
63.
64.      wire [7:0]Diff1 = P_In[16:9] + A_In;
65.      wire [7:0]Diff2 = P_In[16:9] + S_In;
66.
67.      reg [16:0]P;
68.
69.      always @ (*)
70.          if( P_In[1:0] == 2'b01 ) P <= { Diff1[7] , Diff1 , P_In[8:1] };
71.          else if( P_In[1:0] == 2'b10 ) P <= { Diff2[7] , Diff2 , P_In[8:1] };
72.          else P <= { P_In[16] , P_In[16:1] };
73.
74.     assign P_Out = P;
75.
76.  endmodule
```

第 55~74 行是 Task_Booth 任务模块的建立。第 57~60 行是该模块的输入和输出，其中第 57 行是上一个 P 空间的输出，第 58 行则是 A_In 和 S_In 的输入，至于第 60 行是经过运算以后的 P 空间输出。第 64~65 行除了声明 Diff1 和 Diff2 的连线以外，Diff1 和 Diff2 赋值与结果。

第 64~65 行是声明组合逻辑的其中一个办法，它被初学者滥用的情况非常严重，为什么这样说呢？笔者换另一种的方式来说吧 ... 低级建模的基础无疑就是 RTL 级设计的基础，一开始尤其是还不明白时序的时候，第 64~65 行的写法感觉会非常快捷。人就是这样的动物，越方便的东西越是容易上瘾，最终会变成依赖和滥用的情况 ... 此时我们要认真思考，自己的设计是 RTL 为主组合逻辑为次，还是傻傻分不清楚呢？

笔者不是在说空话，事实证明十个 Verilog 的设计，不离八九 ... 会有一大堆 wire 这个 wire 那个的声明，感觉就像蜘蛛网一样，混乱到爆。笔者不是在踩他人，笔者只是在说事实而已 ... 组合逻辑是设计的双刃剑，虽然它非常好用，可是它也伴随相当的危险。过度使用 wire 声明一大堆组合逻辑，会大大降低模块的表达能力，这种情况尤其是“单文件主义”会非常明显。

笔者曾经在扫盲文里指出，wire 类型给我们的感觉就像一条直直的长线，人类的思维非常偏好面向对象，因此 wire 类型一般只在模块和模块之间的连线才会被使用到。除了一些特殊的情况，我们才会使用 wire 来声明组合逻辑。好了！好了！笔者暂不说，不然又要激动了 ... 读者以后只要多加注意一下就行了，什么时候都要以模块的表达能力为先。

第 67 行是 p 寄存器的声明，其中第 69~72 行是组合逻辑的声明，它对应的功能如同 booth_multiplier_module_2 的步骤 1~8，亦即 booth 算法的运算。P_In 经过运算以后，成为新的 P，然后经由 P_Out 输出。

```
1.    module immediate_booth_multiplier_module
2.    (
3.            input [7:0]A,
4.            input [7:0]B,
5.            output [15:0]Product
6.    );
7.          /********************/
8.
9.          wire [7:0]A_Out,S_Out;
10.         wire [16:0]P_Out_T0;
11.
12.         Task_PreConfig T0( A,B, A_Out, S_Out, P_Out_T0 );
13.
14.         wire [16:0]P_Out_T1;
15.         Task_Booth T1( P_Out_T0, A_Out, S_Out, P_Out_T1 );
16.
17.         wire [16:0]P_Out_T2;
18.         Task_Booth T2( P_Out_T1, A_Out, S_Out, P_Out_T2 );
19.
20.         wire [16:0]P_Out_T3;
21.         Task_Booth T3( P_Out_T2, A_Out, S_Out, P_Out_T3 );
22.
23.         wire [16:0]P_Out_T4;
24.         Task_Booth T4( P_Out_T3, A_Out, S_Out, P_Out_T4 );
25.
26.         wire [16:0]P_Out_T5;
27.         Task_Booth T5( P_Out_T4, A_Out, S_Out, P_Out_T5 );
28.
29.         wire [16:0]P_Out_T6;
30.         Task_Booth T6( P_Out_T5, A_Out, S_Out, P_Out_T6 );
31.
32.         wire [16:0]P_Out_T7;
33.         Task_Booth T7( P_Out_T6, A_Out, S_Out, P_Out_T7 );
34.
35.         wire [16:0]P_Out_T8;
36.         Task_Booth T8( P_Out_T7, A_Out, S_Out, P_Out_T8 );
37.
38.           assign Product = P_Out_T8[16:1];
39.
```

```
40.    endmodule
```

第 3~5 行是 immediate_booth_multiplier_module.v 的输入输出声明；第 9~10 行是 Task_PreConfig 输出的连线声明，然而第 12 行既是 Task_PreConfig 的实例化，命名为 T0；第 14 行的 P_Out_T1 是 Task_Booth T1 输出的连线声明，然而第 15 行则是实例化，命名为 T1；其下的如第 17~36 行，都是重复第 14~15 行的动作，不过实例化和输出连线等的命名不同而已。第 38 行是 Prodcut 的输出驱动，驱动源则是 P_Out_T8[16:1]。

一个既时操作的 booth 算法乘法器 2，基本上就是这么简单而已，我们只要按着图 5.10.4 建模就行。

immediate_booth_multiplier_module.vt

```
1.     `timescale 1 ps/ 1 ps
2.     module immediate_booth_multiplier_module_simulation();
3.
4.         reg CLK;
5.         reg RSTn;
6.
7.         reg [7:0]A;
8.         reg [7:0]B;
9.
10.        wire [15:0]Product;
11.
12.            /****************************/
13.
14.        immediate_booth_multiplier_module U1
15.            (
16.                .A(A),
17.                .B(B),
18.                .Product(Product)
19.        );
20.
21.            /****************************/
22.
23.        initial
24.        begin
25.            RSTn = 0; #10; RSTn = 1;
26.            CLK = 0; forever #5 CLK = ~CLK;
27.        end
28.
29.        /****************************/
30.
31.        reg [3:0]i;
32.
```

```
33.     always @ ( posedge CLK or negedge RSTn )
34.         if( !RSTn )
35.                 begin
36.                         i <= 4'd0;
37.                         A <= 8'd0;
38.                         B <= 8'd0;
39.             end
40.             else
41.                 case( i )
42.
43.                         0: // A = 2, B = 4
44.                         begin A <= 8'd2; B <= 8'd4; i <= i + 1'b1; end
45.
46.                         1:
47.                         begin $display( "0b%b * 0b%b = 16'b%b", A,B,Product ); i <= i + 1'b1; end
48.
49.                         2: // A = 3, B = 4
50.                         begin A <= 8'd3; B <= 8'd4; i <= i + 1'b1; end
51.
52.                         3:
53.                         begin $display( "0b%b * 0b%b = 16'b%b", A,B,Product ); i <= i + 1'b1; end
54.
55.                         4: // A = 127, B = 127
56.                         begin A <= 8'd127; B <= 8'd127; i <= i + 1'b1; end
57.
58.                         5:
59.                         begin $display( "0b%b * 0b%b = 16'b%b", A,B,Product ); i <= i + 1'b1; end
60.
61.                         6: // A = 9, B = 9
62.                         begin A <= 8'd9; B <= 8'd9; i <= i + 1'b1; end
63.
64.                         7:
65.                         begin $display( "0b%b * 0b%b = 16'b%b", A,B,Product ); i <= i + 1'b1; end
66.
67.                         8: // A = -9, B = 9
68.                         begin A <= 8'b11110111; B <= 8'd9; i <= i + 1'b1; end
69.
70.                         9:
71.                         begin $display( "0b%b * 0b%b = 16'b%b", A,B,Product ); i <= i + 1'b1; end
72.
73.                         10: // A = -127, B = 127
74.                         begin A <= 8'b10000001; B <= 8'd127; i <= i + 1'b1; end
75.
76.                         11:
77.                         begin $display( "0b%b * 0b%b = 16'b%b", A,B,Product ); i <= i + 1'b1; end
```

```
78.
79.                                 12: // A = 66, B = -111
80.                                 begin A <= 8'd66; B <= 8'b10010001; i <= i + 1'b1; end
81.
82.                                 13:
83.                                 begin $display( "0b%b * 0b%b = 16'b%b", A,B,Product ); i <= i + 1'b1; end
84.
85.                                 14:
86.                                 i <= i;
87.
88.                     endcase
89.
90.         /***************************/
91.
92.   endmodule
```

上面是该模块的激励文件，第 14~19 行是该模块的实例化，注意这个模块本身就是一个大功能的组合逻辑，组合逻辑其中一个特征就是用不着 CLK 信号作为心跳。第 33~88 行是相关的刺激行为，具体的结果，我们从仿真中来观察吧。

仿真结果：

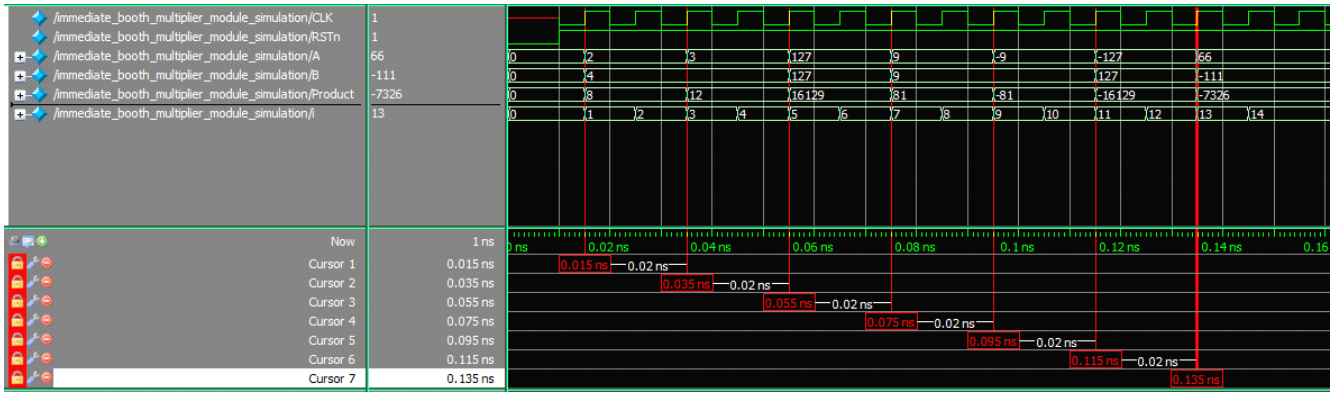


图 5.10.5 仿真结果。

图 5.10.5 是仿真结果，C1~C7 分别指向不同步骤的刺激，我们先来看看第一个的刺激结果。C1 是指向步骤 0，也就是第 1 个时钟，亦即 T0。读者请注意，操作数 A 和 B 中的 2 与 4 都是用 "<="，所以在 T0 的未来，输出符合情理的 2 与 4 的未来值。那么问题来了，在 T1 的未来同样也输出 2 * 4 的未来值，亦即 8 ... 关于这点，我们就开始困惑了？

还记得整合篇（上）第二篇某个小节中所说过的“即时结果在理想时序上的不和谐现象”吗？内容的大意就是“Modelsim 无法显示即时结果 ... ”，这个问题目前正是 Modelsim 面对使用者的尴尬之情。这是一个非常非常小的细节，如果不理解什么是即时事件，基本上不会看出其中的所以然。笔者还是画张图来说话吧：

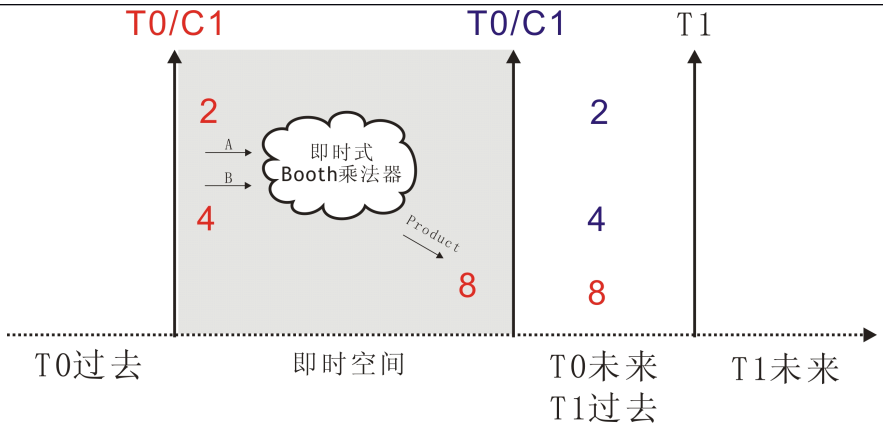


图 5.10.6 又一个即时事件的概念图。

即时事件的概念就是在即时空间里，某种的即时结果产生了，然而这个即时结果在波形图中是显示不出来的（Modelsim 又中招）。其实图 5.10.5 中，C1/T0 所产生的现象可以用图 5.10.6 来解释 ... 读者还记得，我们在 .vt 文件中实例化了即时式 Booth 乘法器吗？然而这个大功能的组合逻辑的两个输入正好连接至 A 与 B 寄存器。

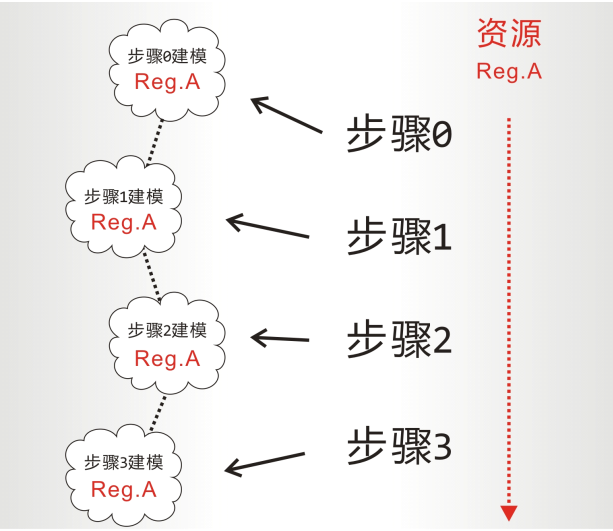
事实上，在 T0/C1 中发生了两种事件：第一种事件就是即时事件，第 44 行的 A <= 8'd2; B <= 8'd4; 对于即时式 Booth 乘法器来说等价与 A = 8'd2; B = 8'd4，所以在 T0/C1 中扩展出即使空间，然后 A 与 B 输入致即时式 Booth 乘法器，然后产生即使结果 8（红色）；在同一个时钟里也发生了正规的时间点事件，亦即 A <= 8'd2; B <= 8'd4; 输入的“决定”在 T0/C1 的未来中输出（蓝色）。读者是否能明白其中的微妙之处吗？

呜 ~ 用简单的话来说，首先读者要明白即使事件在理想时序中的定义，亦即“无视时钟取得结果”。然而 A <= 8'd2; B <= 8'd4; 引起时间点事件的同时又引起即时事件，恰好两个事件显示在时序中，然后又恰好 Modelsim 无法显示即时结果 ... 因为所以必有原因，结果如图 5.10.5 所示的咯。好了，读者不要再学猴子抓头了，往后我们会更深入讨论的，在此只要先理解个大概就好。

其余的 C2~C7 是不同的刺激输入与不同的反应输出，但是同样的现象也发生在它们的身上。最后从仿真结果中，我们可以看出 即时式 Booth 乘法器 求得的任何结果都正确无误，因此我们可以下结论说，“即时操作和建模”的概念是行得通的。

5.11 当笨蛋的组合逻辑遇上笨蛋的编译器

諸葛亮的一支空城計把敌人的大军吓跑，这个故事告訴我們聪明反被聪明误。当我们在思考和理解组合逻辑的時候，我們不能用高级语言的“高级思考”去理解它们，尤其是在组合逻辑建模的时候。组合逻辑是非常逻辑和机械，通俗点就是笨蛋一个 ... 同样它的好朋友编译器也是笨蛋一名，笨蛋配笨蛋不可能变成聪明。要和笨蛋作朋友，我们最好把自己弄成笨蛋一点，这话怎么说呢？



（即时操作）　　（仿顺序操作）

图 5.11.1 组合逻辑的资源概念

我们先来讲资源吧 ... 如图 5.11.1 所示，右边一方是仿顺序操作，从步骤 0~3 一共有 4 个步骤。假设有一个资源称为 Reg.A。对于仿顺序来说 Reg.A 可以在不同的步骤里共享，当笨蛋的编译器在编译的时候，不会引起问题资源被用来用去的问题。换之，假设我们把右边的仿顺序操作转化为左边的即时操作，那么问题就来了。

对于即时操作的建模来所，每一个模块都是独立的东西，然而都拥有各自的资源 Reg.A。因此当我们在建模的时候，务必告诉编译器每一个模块里都有一个 Reg.A（寄存器声明） ... 否则的话，一旦按下该死的编译按钮，编译器这个笨蛋是不知道要为每一个模块建立单独的资源 Reg.A，结果会反馈非常多的 Warning 信息 ....

接下来我们再来谈谈组合逻辑和编译器的智能，假设在仿顺序操作中有这样一条代码：

```
if(Temp[47:46] == 2b00 && Temp[45] == 1'b1) begin .... end
else if(Temp[47:46] == 2b00 && Temp[44] == 1'b1) begin .... end
```

当我们按下编译按钮的时候，基本上编译器不会反馈什么夸张的信息，如果换做即时操作的话 .... 即时操作的建模本质上就是组合逻辑的建模，组合逻辑是一个将逻辑和头脑简单的笨蛋。如果在即时操作中重复上述一段代码，那么结果可大条了！这话又怎么说呢？别忘了，从一开始笔者就一直

说编译器和组合逻辑一样都是一群笨蛋的好兄弟，一个思想简单，一个思想直接。当读者运气较好的时候，可能不会出现什么可怖的反馈信息，不过那一天读者正好是狗屎运，读者就会看到看不明白的反馈信息 ....

因此，我们要用老子的“道”的道理，养老虎就要随着老虎的习性，同样为了迎合笨蛋，我们的思想也要变简单起来。那么，上面一段代码可以这样写：

```
wire is1 = (Temp[47:46] == 2b00 && Temp[45] == 1'b1);
wire is2 = (Temp[47:46] == 2b00 && Temp[44] == 1'b1);

always @ ( * )
    if( is1 ) begin ... end
    else if( is2 ) begin ... end
```

首先我们先要 if 条件的判断内容在外边建立，在此为 is1 和 is2，然后才在 always @ (*) 应用 is1 和 is2。用笨蛋的话来说，我们必须事先要告诉笨蛋“外边有 is1 和 is2 两种可能性”，然后又在 always 里告诉笨蛋“如果遇见 is1 会发生这些 ... 如果遇见 is2 会发生那些”。就这样，笨蛋的编译器和组合逻辑就会在没有烦恼的环境下开心工作。

此外，我们还要将任何可能会发生的事情都告诉笨蛋两兄弟 ... 何解？举个例子：

```
if( Temp[47:46] == 2'b00 ) begin rData <= A; ... end
else begin rData <= B; ... end
```

在这里，我们知道 Temp[47:46] 会出现 4 中可能性，亦即 2'b00, 2'b01, 2'b10 还有 2'b11。其中我们在 if 条件里强调可能性—2'b00 的发生，其他的 3 中可能性都由 else 触发。对于普通的理解来说这种的理想想法非常正确，不过对于笨蛋来说，可能性有两种，亦即“会发生”和“不会发生”。换句话说，假设在 Temp[47:46]中，会发生 2'b00, 2'b01 和 2'b10 这 3 种可能性，然而不会发生 2'b11 这种可能性，那么在 else 里是不包裹 2'b11，明白吗？

所以我们要改一下写法告诉笨蛋：

```
wire is1 = (Temp[47:46] == 2'b00);
wire is2 = (Temp[47:46] == 2'b01);
wire is3 = (Temp[47:46] == 2'b10);

reg rData;

always @ ( * )
    if( is1 ) rData = A;
    else if( is2 ) rData = B;
    else if( is3 ) rData = B;
    else rData = x;
```

哇！代码变成更直接和笨蛋了 ... 首先，我们在外边告诉笨蛋会发生 3 中可能性，亦即 2'b00, 2'b01 和 2'b10。然后在 always 根据不同的可能性为 rData 赋值，至于 else 是输出选择器的象征写法。在这里，可能有些同学抓狂了："既然 is2 和 is3 这两种可能性都会使得 rData 被赋予 B 值，为什么还要故意拆成 is2 和 is3 ，而不是放在 else 里？"

喂喂！这位同学，要记得诸葛亮—空城计的教训，聪明反被聪明误呀！笔者有两种理由：从编译器的性质来讲，这样的写法可以避免编译器干傻事 ... 编译器不会喜欢喜欢就优化掉这个，或者那个；从建模的角度来讲，这种写法有关输出选择器的性质，对于输出选择器来说，else 往往都用来填 x。

最后，组合逻辑还有一个"怪癖"，就是它不可以怀疑自己的东西，再举个例子：

```
module Task_XYZ
(
	input [1:0]A_In,B_In
	output [1:0]A_Out
);

wire is1 = B_In == 2'b01;
wire is2 = B_In == 2'b10;

reg [1:0]A;

always @ ( * )
	if( A == 2'b00) A = ....
	else if( is1 ) A = ....
	else if( is2 ) A = ....
	else A = 2'dx;

assign A_Out= A;

endmodule
```

笔者声明了一个组合逻辑的模块，名为 Task_XYZ。输入输出有 A_In, B_In 和 A_Out。在此被声明的寄存器 A 就是该组合逻辑的东西，读者打起十二分精神听好"绝对！绝对！不要怀疑它！"。如 if(A == 2'b00) 就是某个怀疑的动作，这个动作会使得组合逻辑发神经不工作。呵呵~开玩笑而已，其实这是某种将输出反馈到输入的写法 ... 其中涉及到许多物理上的问题，如缓冲，锁存等，这些问题会使得静态时序分析变成非常难合规。具体的内容笔者也是不清楚，不过直觉告诉笔者这是危险的行为，要尽量避免。

有关"笨蛋"的内容就到这里为止，实际上"笨蛋们"还有许多微妙的故事 ... 有些微妙的细节会让设计者自身大笑几天，呵呵呵。好了，好了！读者就先了解这些吧，往后有机会我们会继续讨论的。

5.12 实验三十一：即时式-浮点数加法模块

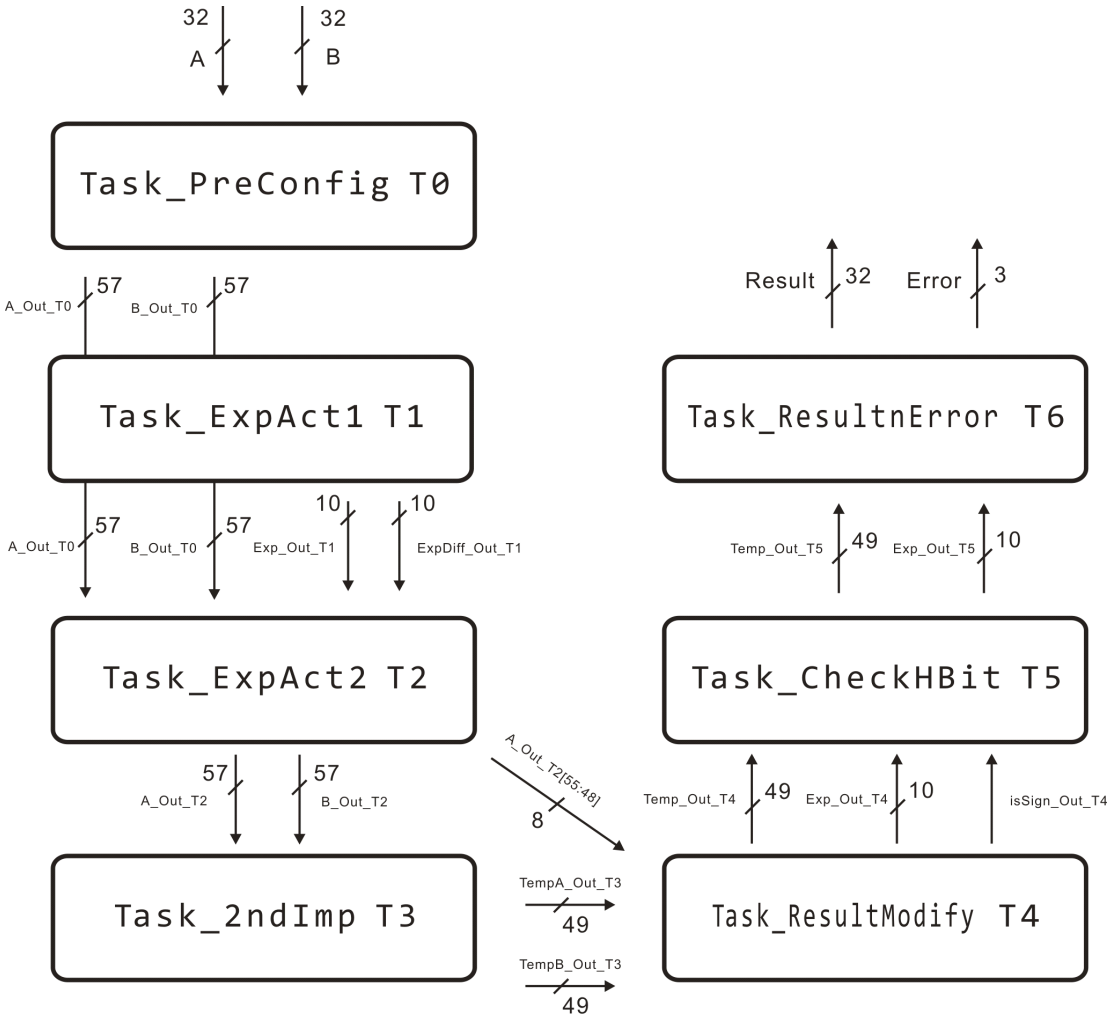


图 5.12.1 即时式-浮点数加法模块的建模图。

图 5.12.1 是按照实验二十中的浮点数加法模块，转换为即时操作的形态，也就是说：任何模块都是即时模块|组合逻辑。T0 是操作数预处理；T1~T2 是价码的对齐和操作；T3 尾数运算预处理；T4 是整合了加减操作和结果预处理；T5 是结果调整；T6 是输出个格式化。好了，现在我们来看逐个模块吧：

immediate_float_add_module.v

```
module Task_PreConfig // i0
(
    input [31:0]A_In,B_In,
    output [56:0]A_Out,B_Out
);
    assign A_Out = { A_In[31], A_In[30:23], 2'b01, A_In[22:0], 23'd0 };
    assign B_Out = { B_In[31], B_In[30:23], 2'b01, B_In[22:0], 23'd0 };
```

```
74.
75.    endmodule
```

第 67~75 行是操作数预处理的组合逻辑|即时模块，第 69~70 行是输入输出的声明，其中第 70 行的 A_Out 和 B_Out 是经过预处理以后的结果。第 72~73 行是预处理的操作，亦即填上隐藏位和尾数补偿。

```
77.    module Task_ExpAct1 //i1
78.    (
79.        input [56:0]A_In, B_In,
80.        output [9:0]Exp_Out,ExpDiff_Out
81.    );
82.
83.        wire [9:0]Exp = A_In[55:48] - B_In[55:48];
84.
85.        assign Exp_Out = Exp;
86.        assign ExpDiff_Out = Exp[8] ? ~Exp + 1'b1 : Exp;
87.
88.    endmodule
```

第 79~80 行是该模块的输入输出声明，第 83 行是取得价码差的动作，然而第 85 行是输出价码差。至于第 86 行也是输出价码差，不过是移位用的价码差（正值化）。

```
90.    module Task_ExpAct2 // i2
91.    (
92.        input [56:0]A_In, B_In,
93.        input [9:0]Exp_In,ExpDiff_In,
94.
95.        output [56:0]A_Out,B_Out
96.    );
97.
98.        wire is1 = (Exp_In[8] == 1);
99.        wire [56:0]A1 = { A_In[56], B_In[55:48], A_In[47:0] >> ExpDiff_In };
100.       wire [56:0]A2 = A_In;
101.       wire [56:0]B1 = { B_In[56], A_In[55:48], B_In[47:0] >> ExpDiff_In };
102.       wire [56:0]B2 = B_In;
103.
104.       assign A_Out = is1 ? A1 : A2;
105.       assign B_Out = !is1 ? B1 : B2;
106.
107.   endmodule
```

第 92~95 行是该模块的输入输出声明。第 98 行是取得价码差的条件（1 为 A 的价码小于 B 的价码）。第 99 行的 A1 是取得 A 价码小于 B 价码的处理结果，然而第 100 行的 A2 取得 A_In 不变的

结果。第 101 行的 B1 取得 B 价码小于 A 价码的处理结果，然而第 102 行的 B2 取得 B_In 不变的结果。最后的 104 行的 A_Out 根据 is1 的条件决定输出 A1 还是 A2，同样第 105 行的 B_Out 根据 is1 的条件决定输出 B1 还是 B2。

```
109. module Task_2ndImp // i3
110. (
111.     input [56:0]A_In,B_In,
112.     output [48:0]TempA,TempB
113. );
114.
115.     assign TempA = A_In[56] ? { A_In[56], (~A_In[47:0] + 1'b1) } : { A_In[56], A_In[47:0] };
116.     assign TempB = B_In[56] ? { B_In[56], (~B_In[47:0] + 1'b1) } : { B_In[56], B_In[47:0] };
117.
118. endmodule
```

第 111~112 行是输入输出的声明，则第 115 行是 TempA 输出的正值化，反之第 116 是 TempB 输出的正值化。

```
120. module Task_ResultModify // i4~i5
121. (
122.     input [7:0]A_Exp_In,
123.     input [48:0]TempA_In,TempB_In,
124.
125.     output isSign_Out,
126.     output [9:0]Exp_Out,
127.     output [48:0]Temp_Out
128. );
129.
130.         wire [48:0]AddResult = TempA_In + TempB_In;
131.
132.         assign isSign_Out = AddResult[48];
133.         assign Temp_Out = AddResult[48] ? ~AddResult + 1'b1 : AddResult;
134.         assign Exp_Out = { 2'b00, A_Exp_In };
135.
136. endmodule
```

第 122~127 行是该模块的输入输出声明，第 130 的 AddResult 取得 TempA + TempB 的结果（即时式-浮点数减法模块只是修改这行而已），然后在第 132 行取得符号位，第 133 行取得加法结果的正值化，第 134 行则是输出新的价码。

```
138. module Task_CheckHBit // i6
139. (
140.     input [9:0]Exp_In,
141.     input [48:0]Temp_In,
```

```
142.
143.        output [9:0]Exp_Out,
144.        output [48:0]Temp_Out
145.    );
146.
147.        wire is1 = ( Temp_In[47:46] == 2'b10 || Temp_In[47:46] == 2'b11);
148.        wire is2 = ( Temp_In[47:46] == 2'b00 && Temp_In[45] );
149.        wire is3 = ( Temp_In[47:46] == 2'b00 && Temp_In[44] );
150.        wire is4 = ( Temp_In[47:46] == 2'b00 && Temp_In[43] );
151.        wire is5 = ( Temp_In[47:46] == 2'b00 && Temp_In[42] );
152.        wire is6 = ( Temp_In[47:46] == 2'b00 && Temp_In[41] );
153.        wire is7 = ( Temp_In[47:46] == 2'b00 && Temp_In[40] );
154.        wire is8 = ( Temp_In[47:46] == 2'b00 && Temp_In[39] );
155.        wire is9 = ( Temp_In[47:46] == 2'b00 && Temp_In[38] );
156.        wire is10 = ( Temp_In[47:46] == 2'b00 && Temp_In[37] );
157.        wire is11 = ( Temp_In[47:46] == 2'b00 && Temp_In[36] );
158.        wire is12 = ( Temp_In[47:46] == 2'b00 && Temp_In[35] );
159.        wire is13 = ( Temp_In[47:46] == 2'b00 && Temp_In[34] );
160.        wire is14 = ( Temp_In[47:46] == 2'b00 && Temp_In[33] );
161.        wire is15 = ( Temp_In[47:46] == 2'b00 && Temp_In[32] );
162.        wire is16 = ( Temp_In[47:46] == 2'b00 && Temp_In[31] );
163.        wire is17 = ( Temp_In[47:46] == 2'b00 && Temp_In[30] );
164.        wire is18 = ( Temp_In[47:46] == 2'b00 && Temp_In[29] );
165.        wire is19 = ( Temp_In[47:46] == 2'b00 && Temp_In[28] );
166.        wire is20 = ( Temp_In[47:46] == 2'b00 && Temp_In[27] );
167.        wire is21 = ( Temp_In[47:46] == 2'b00 && Temp_In[26] );
168.        wire is22 = ( Temp_In[47:46] == 2'b00 && Temp_In[25] );
169.        wire is23 = ( Temp_In[47:46] == 2'b00 && Temp_In[24] );
170.        wire is24 = ( Temp_In[47:46] == 2'b00 && Temp_In[23] );
171.        wire is25 = ( Temp_In[47:46] == 2'b01 );
172.        wire is26 = ( Temp_In[47:46] == 2'b00 && Temp_In[45:23] == 23'd0 );
173.
174.        wire [48:0]Temp1 = Temp_In >> 1;
175.        wire [48:0]Temp2 = Temp_In << 1;
176.        wire [48:0]Temp3 = Temp_In << 2;
177.        wire [48:0]Temp4 = Temp_In << 3;
178.        wire [48:0]Temp5 = Temp_In << 4;
179.        wire [48:0]Temp6 = Temp_In << 5;
180.        wire [48:0]Temp7 = Temp_In << 6;
181.        wire [48:0]Temp8 = Temp_In << 7;
182.        wire [48:0]Temp9 = Temp_In << 8;
183.        wire [48:0]Temp10 = Temp_In << 9;
184.        wire [48:0]Temp11 = Temp_In << 10;
185.        wire [48:0]Temp12 = Temp_In << 11;
186.        wire [48:0]Temp13 = Temp_In << 12;
```

```
187.    wire [48:0]Temp14 = Temp_In << 13;
188.    wire [48:0]Temp15 = Temp_In << 14;
189.    wire [48:0]Temp16 = Temp_In << 15;
190.    wire [48:0]Temp17 = Temp_In << 16;
191.    wire [48:0]Temp18 = Temp_In << 17;
192.    wire [48:0]Temp19 = Temp_In << 18;
193.    wire [48:0]Temp20 = Temp_In << 19;
194.    wire [48:0]Temp21 = Temp_In << 20;
195.    wire [48:0]Temp22 = Temp_In << 21;
196.    wire [48:0]Temp23 = Temp_In << 22;
197.    wire [48:0]Temp24 = Temp_In << 23;
198.    wire [48:0]Temp25 = Temp_In;
199.    wire [48:0]Temp26 = Temp_In;
200.
201.    wire [9:0]Exp1 =   Exp_In + 1'b1;
202.    wire [9:0]Exp2 =   Exp_In - 5'd1;
203.    wire [9:0]Exp3 =   Exp_In - 5'd2;
204.    wire [9:0]Exp4 =   Exp_In - 5'd3;
205.    wire [9:0]Exp5 =   Exp_In - 5'd4;
206.    wire [9:0]Exp6 =   Exp_In - 5'd5;
207.    wire [9:0]Exp7 =   Exp_In - 5'd6;
208.    wire [9:0]Exp8 =   Exp_In - 5'd7;
209.    wire [9:0]Exp9 =   Exp_In - 5'd8;
210.    wire [9:0]Exp10 =   Exp_In - 5'd9;
211.    wire [9:0]Exp11 =   Exp_In - 5'd10;
212.    wire [9:0]Exp12 =   Exp_In - 5'd11;
213.    wire [9:0]Exp13 =   Exp_In - 5'd12;
214.    wire [9:0]Exp14 =   Exp_In - 5'd13;
215.    wire [9:0]Exp15 =   Exp_In - 5'd14;
216.    wire [9:0]Exp16 =   Exp_In - 5'd15;
217.    wire [9:0]Exp17 =   Exp_In - 5'd16;
218.    wire [9:0]Exp18 =   Exp_In - 5'd17;
219.    wire [9:0]Exp19 =   Exp_In - 5'd18;
220.    wire [9:0]Exp20 =   Exp_In - 5'd19;
221.    wire [9:0]Exp21 =   Exp_In - 5'd20;
222.    wire [9:0]Exp22 =   Exp_In - 5'd21;
223.    wire [9:0]Exp23 =   Exp_In - 5'd22;
224.    wire [9:0]Exp24 =   Exp_In - 5'd23;
225.    wire [9:0]Exp25 = Exp_In;
226.    wire [9:0]Exp26 = Exp_In;
227.
228.    reg [48:0]Temp;
229.    reg [9:0]Exp;
230.
231.    always @ ( * )
```

```
232.            if( is1 ) begin Temp = Temp1; Exp = Exp1; end
233.            else if( is2 ) begin Temp = Temp2; Exp = Exp2; end
234.            else if( is3 ) begin Temp = Temp3; Exp = Exp3; end
235.            else if( is4 ) begin Temp = Temp4; Exp = Exp4; end
236.            else if( is5 ) begin Temp = Temp5; Exp = Exp5; end
237.            else if( is6 ) begin Temp = Temp6; Exp = Exp6; end
238.            else if( is7 ) begin Temp = Temp7; Exp = Exp7; end
239.            else if( is8 ) begin Temp = Temp8; Exp = Exp8; end
240.            else if( is9 ) begin Temp = Temp9; Exp = Exp9; end
241.            else if( is10 ) begin Temp = Temp10; Exp = Exp10; end
242.            else if( is11 ) begin Temp = Temp11; Exp = Exp11; end
243.            else if( is12 ) begin Temp = Temp12; Exp = Exp12; end
244.            else if( is13 ) begin Temp = Temp13; Exp = Exp13; end
245.            else if( is14 ) begin Temp = Temp14; Exp = Exp14; end
246.            else if( is15 ) begin Temp = Temp15; Exp = Exp15; end
247.            else if( is16 ) begin Temp = Temp16; Exp = Exp16; end
248.            else if( is17 ) begin Temp = Temp17; Exp = Exp17; end
249.            else if( is18 ) begin Temp = Temp18; Exp = Exp18; end
250.            else if( is19 ) begin Temp = Temp19; Exp = Exp19; end
251.            else if( is20 ) begin Temp = Temp20; Exp = Exp20; end
252.            else if( is21 ) begin Temp = Temp21; Exp = Exp21; end
253.            else if( is22 ) begin Temp = Temp22; Exp = Exp22; end
254.            else if( is23 ) begin Temp = Temp23; Exp = Exp23; end
255.            else if( is24 ) begin Temp = Temp24; Exp = Exp24; end
256.            else if( is25 ) begin Temp = Temp25; Exp = Exp25; end
257.            else if( is26 ) begin Temp = Temp26; Exp = Exp26; end
258.            else begin Temp = 49'bx; Exp = 10'bx; end
259.
260.      assign Exp_Out = Exp;
261.      assign Temp_Out = Temp;
262.
263.  endmodule
```

第 140~144 行是该模块的输入输出声明，千万不要被接下来庞大的代码行吓到 ... 这些都是举例给傻瓜兄弟知道的任何可能性而已。第 147~172 是 26 种有关隐藏位的可能性，is1 是 Temp_In[47:46]等价与 2'b10 和 2'b11 的情况；is2~24 是[47:46]为 2'b00，然而隐藏位可能存在在[45 ..23]中；is25 则是正常结果的可能性；is26 则是尾数零值的可能性。 第 174~199 是相关的 Temp 可能性操作；第 201~226 是相关的 Exp 可能性操作；

第 228~229 行则声明了 Temp 和 Exp 的寄存器，然而第 231~258 行是庞大的有优先级的输出选择器，它输出选择任何 26 中有关 Temp 和 Exp 受操作的可能性。最后第 258 行是输出选择器的标志性特征。

```
265.  module Task_ResultnError // i7
```

```
266.  (
267.      input isSign_In,
268.      input [9:0]Exp_In,
269.      input [48:0]Temp_In,
270.
271.      output [2:0]Error,
272.      output [31:0]Result
273.  );
274.
275.      wire isZero = ( Temp_In[46:23] == 24'd0 );
276.      wire isUnder = ( Exp_In[9:8] == 2'b11 );
277.      wire isOver = ( Exp_In[9:8] == 2'b01 );
278.
279.      wire isNormalise = Temp_In[22];
280.      wire [31:0]Result1 = { isSign_In, Exp_In[7:0], Temp_In[45:23] + 1'b1 };
281.      wire [31:0]Result2 = { isSign_In, Exp_In[7:0], Temp_In[45:23] };
282.
283.      assign Error = { isOver, isUnder, isZero };
284.      assign Result = isNormalise ? Result1 : Result2;
285.
286.  endmodule
```

第 267~272 行是相关的输入输出声明。第 275~277 行取得 isZero，isUnder 还有 isOver 三种错误标志，然而第 279 行则是取得发生四舍五入的可能性。第 280 行是发生四舍五入正规化以后的结果，第 281 行则是没有发生四舍五入的正规化以后的结果。第 283 行是错误的反馈输出，第 284 行行则是 Result 的输出。

```
1.    module immediate_float_add_module
2.    (
3.        input [31:0]A,B,
4.
5.        output [31:0]Result,
6.        output [2:0]Error,
7.
8.        output [56:0]SQ_A_Out_T0,SQ_B_Out_T0, // T0
9.        output [9:0]SQ_Exp_Out_T1,SQ_ExpDiff_Out_T1, // T1
10.       output [56:0]SQ_A_Out_T2,SQ_B_Out_T2,
11.       output [48:0]SQ_TempA_Out_T3,SQ_TempB_Out_T3,
12.       output SQ_isSign_Out_T4,
13.       output [9:0]SQ_Exp_Out_T4,
14.       output [48:0]SQ_Temp_Out_T4,
15.       output [9:0]SQ_Exp_Out_T5,
16.       output [48:0]SQ_Temp_Out_T5
17.
```

```
18.  );
19.      /*************************************/
20.
21.      wire [56:0]A_Out_T0,B_Out_T0;
22.      Task_PreConfig T0( A, B, A_Out_T0, B_Out_T0 );
23.
24.      wire [9:0]Exp_Out_T1,ExpDiff_Out_T1;
25.      Task_ExpAct1 T1( A_Out_T0, B_Out_T0, Exp_Out_T1, ExpDiff_Out_T1 );
26.
27.      wire [56:0]A_Out_T2, B_Out_T2;
28.      Task_ExpAct2 T2( A_Out_T0, B_Out_T0, Exp_Out_T1, ExpDiff_Out_T1, A_Out_T2, B_Out_T2 );
29.
30.      wire [48:0]TempA_Out_T3,TempB_Out_T3;
31.      Task_2ndImp T3( A_Out_T2, B_Out_T2, TempA_Out_T3, TempB_Out_T3 );
32.
33.      wire isSign_Out_T4;
34.      wire [9:0]Exp_Out_T4;
35.      wire [48:0]Temp_Out_T4;
36.      Task_ResultModify T4( A_Out_T2[55:48], TempA_Out_T3, TempB_Out_T3, isSign_Out_T4, Exp_Out_T4, Temp_Out_T4 );
37.
38.      wire [9:0]Exp_Out_T5;
39.      wire [48:0]Temp_Out_T5;
40.      Task_CheckHBit T5( Exp_Out_T4, Temp_Out_T4, Exp_Out_T5, Temp_Out_T5 );
41.
42.      wire [2:0]Error_T6;
43.      wire [31:0]Result_T6;
44.      Task_ResultnError T6( isSign_Out_T4, Exp_Out_T5, Temp_Out_T5, Error_T6, Result_T6    );
45.
46.      assign Error = Error_T6;
47.      assign Result = Result_T6;
48.
49.      /*******************************/
50.
51.      assign SQ_A_Out_T0 = A_Out_T0;
52.      assign SQ_B_Out_T0 = B_Out_T0;
53.      assign SQ_Exp_Out_T1 = Exp_Out_T1;
54.      assign SQ_ExpDiff_Out_T1 = ExpDiff_Out_T1; // T1
55.      assign SQ_A_Out_T2 = A_Out_T2;
56.      assign SQ_B_Out_T2 = B_Out_T2;
57.      assign SQ_TempA_Out_T3 = TempA_Out_T3;
58.      assign SQ_TempB_Out_T3 = TempB_Out_T3;
59.      assign SQ_isSign_Out_T4 = isSign_Out_T4;
60.      assign SQ_Exp_Out_T4 = Exp_Out_T4;
61.      assign SQ_Temp_Out_T4 = Temp_Out_T4;
62.      assign SQ_Exp_Out_T5 = Exp_Out_T5;
```

```
63.         assign SQ_Temp_Out_T5 = Temp_Out_T5;
64.
65.     endmodule
```

第 3~6 行则是即时式-浮点数加法模块的输入输出声明，第 8~16 行则是仿真用的相关输出。第 21~47 行是根据图 5.12.1 的建模操作，具体的内容自己看着办吧。第 51~63 行是仿真用的相关输出驱动。

基本上实验三十一就是这些而已，实验三十一唯一难搞懂的地方就是 5.11 小节里的内容。如果读者用聪明的脑袋去理解实验三十一的话，大脑会短路数十次。换之，如果读者用笨蛋的理解方式去思考实验三十一的内容的话，任何一段代码都符合笨蛋的常理。即时式操作和流水式操作大致上非常相似，前者是基于组合逻辑级，后者则是基于 RTL 级。然而两者都有许多相似的地方如：每个任务模块都拥有各自的资源，一个是即时层，一个是流水层等。唯即时式操作的任务模块是比较笨蛋而已，哈哈。

immediate_float_add_module.vt

```
1.      `timescale 1 ps/ 1 ps
2.      module immediate_float_add_module_simulation();
3.
4.              reg CLK;
5.              reg RSTn;
6.
7.              reg [31:0] A;
8.              reg [31:0] B;
9.
10.             wire [31:0]    Result;
11.             wire [2:0] Error;
12.
13.         wire [56:0]SQ_A_Out_T0,SQ_B_Out_T0; // T0
14.         wire [9:0]SQ_Exp_Out_T1,SQ_ExpDiff_Out_T1; // T1
15.         wire [56:0]SQ_A_Out_T2,SQ_B_Out_T2;
16.         wire [48:0]SQ_TempA_Out_T3,SQ_TempB_Out_T3;
17.         wire SQ_isSign_Out_T4;
18.         wire [9:0]SQ_Exp_Out_T4;
19.         wire [48:0]SQ_Temp_Out_T4;
20.         wire [9:0]SQ_Exp_Out_T5;
21.         wire [48:0]SQ_Temp_Out_T5;
22.
23.             /***********************************/
24.
25.             immediate_float_add_module U1
26.             (
```

```
27.                 .A( A ),
28.                 .B( B ),
29.                 .Result( Result ),
30.                 .Error( Error ),
31.                 /********************/
32.                 .SQ_A_Out_T0( SQ_A_Out_T0 ),
33.                 .SQ_B_Out_T0( SQ_B_Out_T0 ),
34.                 .SQ_Exp_Out_T1( SQ_Exp_Out_T1 ),
35.                 .SQ_ExpDiff_Out_T1( SQ_ExpDiff_Out_T1 ),
36.                 .SQ_A_Out_T2( SQ_A_Out_T2 ),
37.                 .SQ_B_Out_T2( SQ_B_Out_T2 ),
38.                 .SQ_TempA_Out_T3( SQ_TempA_Out_T3 ),
39.                 .SQ_TempB_Out_T3( SQ_TempB_Out_T3 ),
40.                 .SQ_isSign_Out_T4( SQ_isSign_Out_T4 ),
41.                 .SQ_Exp_Out_T4( SQ_Exp_Out_T4 ),
42.                 .SQ_Temp_Out_T4( SQ_Temp_Out_T4 ),
43.                 .SQ_Exp_Out_T5( SQ_Exp_Out_T5 ),
44.                 .SQ_Temp_Out_T5( SQ_Temp_Out_T5 )
45.         );
46.
47.         /************************************/
48.
49.         initial
50.         begin
51.                 RSTn = 0; #10 RSTn = 1;
52.                 CLK = 0; forever #5 CLK = ~CLK;
53.         end
54.
55.         /************************************/
56.
57.         reg [3:0]i;
58.
59.         always @ (    posedge CLK or negedge RSTn )
60.                 if( !RSTn )
61.                         begin
62.                           A <= 32'd0;
63.                           B <= 32'd0;
64.                           i <= 4'd0;
65.                       end
66.             else
67.                       case( i )
68.
69.                               0: //A=3.65, B= -7.4, A+B = ?
70.                               begin A <= 32'b01000000011010011001100110011010; B <= 32'b11000000111011001100110011001101;    i <= i + 1'b1; end
71.
```

```
72.                 1:
73.                 begin
74.                   $display( "SQ_A_Out_T0 = 57'b%b",SQ_A_Out_T0 );
75.                   $display( "SQ_B_Out_T0 = 57'b%b",SQ_B_Out_T0 );
76.                   $display( "SQ_Exp_Out_T1 = 10'b%b",SQ_Exp_Out_T1 );
77.                   $display( "SQ_ExpDiff_Out_T1 = 10'b%b",SQ_ExpDiff_Out_T1 );
78.                  $display( "SQ_A_Out_T2 = 57'b%b",SQ_A_Out_T2 );
79.                  $display( "SQ_B_Out_T2 = 57'b%b",SQ_B_Out_T2 );
80.                  $display( "SQ_TempA_Out_T3 = 49'b%b",SQ_TempA_Out_T3 );
81.                  $display( "SQ_TempB_Out_T3 = 49'b%b",SQ_TempB_Out_T3 );
82.                  $display( "SQ_isSign_Out_T4 = 1'b%b",SQ_isSign_Out_T4 );
83.                  $display( "SQ_Exp_Out_T4 = 10'b%b",SQ_Exp_Out_T4 );
84.                  $display( "SQ_Temp_Out_T4 = 49'b%b",SQ_Temp_Out_T4 );
85.                  $display( "SQ_Exp_Out_T5 = 10'b%b",SQ_Exp_Out_T5 );
86.                  $display( "SQ_Temp_Out_T5 = 49'b%b",SQ_Temp_Out_T5 );
87.                  $display( "Result1 = %32b",Result );
88.                  $display( "Error = 3'b%b", Error );
89.                  i <= i + 1'b1;
90.             end
91.
92.                 2: //Exp undeflow check
93.                 begin A <= 32'b00000000000101000010110100010110 1; B <= 32'b10000000000001000010110000100011 1; i <= i + 1'b1; end
94.
95.                 3:
96.                 begin
97.                   $display( "SQ_A_Out_T0 = 57'b%b",SQ_A_Out_T0 );
98.                   $display( "SQ_B_Out_T0 = 57'b%b",SQ_B_Out_T0 );
99.                   $display( "SQ_Exp_Out_T1 = 10'b%b",SQ_Exp_Out_T1 );
100.                  $display( "SQ_ExpDiff_Out_T1 = 10'b%b",SQ_ExpDiff_Out_T1 );
101.                 $display( "SQ_A_Out_T2 = 57'b%b",SQ_A_Out_T2 );
102.                 $display( "SQ_B_Out_T2 = 57'b%b",SQ_B_Out_T2 );
103.                 $display( "SQ_TempA_Out_T3 = 49'b%b",SQ_TempA_Out_T3 );
104.                 $display( "SQ_TempB_Out_T3 = 49'b%b",SQ_TempB_Out_T3 );
105.                 $display( "SQ_isSign_Out_T4 = 1'b%b",SQ_isSign_Out_T4 );
106.                 $display( "SQ_Exp_Out_T4 = 10'b%b",SQ_Exp_Out_T4 );
107.                 $display( "SQ_Temp_Out_T4 = 49'b%b",SQ_Temp_Out_T4 );
108.                 $display( "SQ_Exp_Out_T5 = 10'b%b",SQ_Exp_Out_T5 );
109.                 $display( "SQ_Temp_Out_T5 = 49'b%b",SQ_Temp_Out_T5 );
110.                 $display( "Result2 = %32b",Result );
111.                 $display( "Error = 3'b%b", Error );
112.                 i <= i + 1'b1;
113.            end
114.
115.                4: //A=1.9999997, B=-1.9999998 , A+B =?
116.                begin A <= 32'b00111111111111111111111111111110;  B <= 32'b10111111111111111111111111111111;  i <= i + 1'b1; end
```

```
117.
118.                    5:
119.                    begin
120.                      $display( "SQ_A_Out_T0 = 57'b%b",SQ_A_Out_T0 );
121.                      $display( "SQ_B_Out_T0 = 57'b%b",SQ_B_Out_T0 );
122.                      $display( "SQ_Exp_Out_T1 = 10'b%b",SQ_Exp_Out_T1 );
123.                      $display( "SQ_ExpDiff_Out_T1 = 10'b%b",SQ_ExpDiff_Out_T1 );
124.                     $display( "SQ_A_Out_T2 = 57'b%b",SQ_A_Out_T2 );
125.                     $display( "SQ_B_Out_T2 = 57'b%b",SQ_B_Out_T2 );
126.                     $display( "SQ_TempA_Out_T3 = 49'b%b",SQ_TempA_Out_T3 );
127.                     $display( "SQ_TempB_Out_T3 = 49'b%b",SQ_TempB_Out_T3 );
128.                     $display( "SQ_isSign_Out_T4 = 1'b%b",SQ_isSign_Out_T4 );
129.                     $display( "SQ_Exp_Out_T4 = 10'b%b",SQ_Exp_Out_T4 );
130.                     $display( "SQ_Temp_Out_T4 = 49'b%b",SQ_Temp_Out_T4 );
131.                     $display( "SQ_Exp_Out_T5 = 10'b%b",SQ_Exp_Out_T5 );
132.                     $display( "SQ_Temp_Out_T5 = 49'b%b",SQ_Temp_Out_T5 );
133.                     $display( "Result3 = %32b",Result );
134.                     $display( "Error = 3'b%b", Error );
135.                     i <= i + 1'b1;
136.                  end
137.
138.                    6: //Exp Overflow
139.                    begin A <= 32'b01111111111111111111111111111111; B <= 32'b01111111111111111111111111111111; i <= i + 1'b1; end
140.
141.                    7:
142.                    begin
143.                      $display( "SQ_A_Out_T0 = 57'b%b",SQ_A_Out_T0 );
144.                      $display( "SQ_B_Out_T0 = 57'b%b",SQ_B_Out_T0 );
145.                      $display( "SQ_Exp_Out_T1 = 10'b%b",SQ_Exp_Out_T1 );
146.                      $display( "SQ_ExpDiff_Out_T1 = 10'b%b",SQ_ExpDiff_Out_T1 );
147.                     $display( "SQ_A_Out_T2 = 57'b%b",SQ_A_Out_T2 );
148.                     $display( "SQ_B_Out_T2 = 57'b%b",SQ_B_Out_T2 );
149.                     $display( "SQ_TempA_Out_T3 = 49'b%b",SQ_TempA_Out_T3 );
150.                     $display( "SQ_TempB_Out_T3 = 49'b%b",SQ_TempB_Out_T3 );
151.                     $display( "SQ_isSign_Out_T4 = 1'b%b",SQ_isSign_Out_T4 );
152.                     $display( "SQ_Exp_Out_T4 = 10'b%b",SQ_Exp_Out_T4 );
153.                     $display( "SQ_Temp_Out_T4 = 49'b%b",SQ_Temp_Out_T4 );
154.                     $display( "SQ_Exp_Out_T5 = 10'b%b",SQ_Exp_Out_T5 );
155.                     $display( "SQ_Temp_Out_T5 = 49'b%b",SQ_Temp_Out_T5 );
156.                     $display( "Result4 = %32b",Result );
157.                     $display( "Error = 3'b%b", Error );
158.                     i <= i + 1'b1;
159.                end
160.
161.                    8: //A= -12.558, B= -7.309 , A+B =?
```

```
                    begin A <= 32'b11000001010010001110110110010001; B <= 32'b11000000111010011110001101010100; i <= i + 1'b1; end

                    9:
                    begin
                      $display( "SQ_A_Out_T0 = 57'b%b",SQ_A_Out_T0 );
                      $display( "SQ_B_Out_T0 = 57'b%b",SQ_B_Out_T0 );
                      $display( "SQ_Exp_Out_T1 = 10'b%b",SQ_Exp_Out_T1 );
                      $display( "SQ_ExpDiff_Out_T1 = 10'b%b",SQ_ExpDiff_Out_T1 );
                     $display( "SQ_A_Out_T2 = 57'b%b",SQ_A_Out_T2 );
                     $display( "SQ_B_Out_T2 = 57'b%b",SQ_B_Out_T2 );
                     $display( "SQ_TempA_Out_T3 = 49'b%b",SQ_TempA_Out_T3 );
                     $display( "SQ_TempB_Out_T3 = 49'b%b",SQ_TempB_Out_T3 );
                     $display( "SQ_isSign_Out_T4 = 1'b%b",SQ_isSign_Out_T4 );
                     $display( "SQ_Exp_Out_T4 = 10'b%b",SQ_Exp_Out_T4 );
                     $display( "SQ_Temp_Out_T4 = 49'b%b",SQ_Temp_Out_T4 );
                     $display( "SQ_Exp_Out_T5 = 10'b%b",SQ_Exp_Out_T5 );
                     $display( "SQ_Temp_Out_T5 = 49'b%b",SQ_Temp_Out_T5 );
                     $display( "Result5 = %32b",Result );
                     $display( "Error = 3'b%b", Error );
                     i <= i + 1'b1;
                end

                    10: //A= 111.7762, B= 302.4409 , A+B =?
                    begin A <= 32'b01000010110111111000110101101010; B <= 32'b01000011100101110011100001101111; i <= i + 1'b1; end

                    11:
                    begin
                      $display( "SQ_A_Out_T0 = 57'b%b",SQ_A_Out_T0 );
                      $display( "SQ_B_Out_T0 = 57'b%b",SQ_B_Out_T0 );
                      $display( "SQ_Exp_Out_T1 = 10'b%b",SQ_Exp_Out_T1 );
                      $display( "SQ_ExpDiff_Out_T1 = 10'b%b",SQ_ExpDiff_Out_T1 );
                     $display( "SQ_A_Out_T2 = 57'b%b",SQ_A_Out_T2 );
                     $display( "SQ_B_Out_T2 = 57'b%b",SQ_B_Out_T2 );
                     $display( "SQ_TempA_Out_T3 = 49'b%b",SQ_TempA_Out_T3 );
                     $display( "SQ_TempB_Out_T3 = 49'b%b",SQ_TempB_Out_T3 );
                     $display( "SQ_isSign_Out_T4 = 1'b%b",SQ_isSign_Out_T4 );
                     $display( "SQ_Exp_Out_T4 = 10'b%b",SQ_Exp_Out_T4 );
                     $display( "SQ_Temp_Out_T4 = 49'b%b",SQ_Temp_Out_T4 );
                     $display( "SQ_Exp_Out_T5 = 10'b%b",SQ_Exp_Out_T5 );
                     $display( "SQ_Temp_Out_T5 = 49'b%b",SQ_Temp_Out_T5 );
                     $display( "Result6 = %32b",Result );
                     $display( "Error = 3'b%b", Error );
                     i <= i + 1'b1;
                end

```

```
                12: //A= 2112.2012, B= -2002.2012 , A+B =?
                begin A <= 32'b01000101000001000000001100111000; B <= 32'b11000100111110100100011001110000;i <= i + 1'b1; end

                13:
                begin
                  $display( "SQ_A_Out_T0 = 57'b%b",SQ_A_Out_T0 );
                  $display( "SQ_B_Out_T0 = 57'b%b",SQ_B_Out_T0 );
                  $display( "SQ_Exp_Out_T1 = 10'b%b",SQ_Exp_Out_T1 );
                  $display( "SQ_ExpDiff_Out_T1 = 10'b%b",SQ_ExpDiff_Out_T1 );
                  $display( "SQ_A_Out_T2 = 57'b%b",SQ_A_Out_T2 );
                  $display( "SQ_B_Out_T2 = 57'b%b",SQ_B_Out_T2 );
                  $display( "SQ_TempA_Out_T3 = 49'b%b",SQ_TempA_Out_T3 );
                  $display( "SQ_TempB_Out_T3 = 49'b%b",SQ_TempB_Out_T3 );
                  $display( "SQ_isSign_Out_T4 = 1'b%b",SQ_isSign_Out_T4 );
                  $display( "SQ_Exp_Out_T4 = 10'b%b",SQ_Exp_Out_T4 );
                  $display( "SQ_Temp_Out_T4 = 49'b%b",SQ_Temp_Out_T4 );
                  $display( "SQ_Exp_Out_T5 = 10'b%b",SQ_Exp_Out_T5 );
                  $display( "SQ_Temp_Out_T5 = 49'b%b",SQ_Temp_Out_T5 );
                  $display( "Result7 = %32b",Result );
                  $display( "Error = 3'b%b", Error );
                  i <= i + 1'b1;
            end

                14:
                  i <= i;

            endcase

endmodule
```

上述的代码是用来测试 immediate_float_add_module，写法还是老样子，读者自己看着吧？

仿真结果：

由于仿真图没有任何意义 ... 笔者直接将打印结果列出来让读者自己参考吧。

```
A=3.65, B= -7.4, A+B = ?
# SQ_A_Out_T0 = 57'b010000000011101001100110011001101000000000000000000000000
# SQ_B_Out_T0 = 57'b110000001011101100110011001100110100000000000000000000000
# SQ_Exp_Out_T1 = 10'b1111111111
# SQ_ExpDiff_Out_T1 = 10'b0000000001
# SQ_A_Out_T2 = 57'b010000001001110100110011001100110100000000000000000000000
# SQ_B_Out_T2 = 57'b110000001011101100110011001100110100000000000000000000000
# SQ_TempA_Out_T3 = 49'b0001110100110011001100110100000000000000000000000
```

```
# SQ_TempB_Out_T3 = 49'b1100010011001100110011001100000000000000000000000
# SQ_isSign_Out_T4 = 1'b1
# SQ_Exp_Out_T4 = 10'b0010000001
# SQ_Temp_Out_T4 = 49'b0001111000000000000000000000000000000000000000000
# SQ_Exp_Out_T5 = 10'b0010000000
# SQ_Temp_Out_T5 = 49'b0011110000000000000000000000000000000000000000000
# Result1 = 1100000001110000000000000000000
# Error = 3'b000
```

Exp undeflow check

```
# SQ_A_Out_T0 = 57'b000000000010101000010110100010110100000000000000000000000
# SQ_B_Out_T0 = 57'b100000000010001000010110000100011100000000000000000000000
# SQ_Exp_Out_T1 = 10'b0000000000
# SQ_ExpDiff_Out_T1 = 10'b0000000000
# SQ_A_Out_T2 = 57'b000000000010101000010110100010110100000000000000000000000
# SQ_B_Out_T2 = 57'b100000000010001000010110000100011100000000000000000000000
# SQ_TempA_Out_T3 = 49'b0010101000010110100010110100000000000000000000000
# SQ_TempB_Out_T3 = 49'b1101110111101001110111100100000000000000000000000
# SQ_isSign_Out_T4 = 1'b0
# SQ_Exp_Out_T4 = 10'b0000000000
# SQ_Temp_Out_T4 = 49'b0000100000000000011110011000000000000000000000000
# SQ_Exp_Out_T5 = 10'b1111111110
# SQ_Temp_Out_T5 = 49'b0010000000000001111001100000000000000000000000000
# Result2 = 0111111100000000000011110011000
# Error = 3'b010
```

A=1.9999997, B=-1.9999998 , A+B =?

```
# SQ_A_Out_T0 = 57'b001111111011111111111111111111111000000000000000000000000
# SQ_B_Out_T0 = 57'b101111111011111111111111111111111100000000000000000000000
# SQ_Exp_Out_T1 = 10'b0000000000
# SQ_ExpDiff_Out_T1 = 10'b0000000000
# SQ_A_Out_T2 = 57'b001111111011111111111111111111111000000000000000000000000
# SQ_B_Out_T2 = 57'b101111111011111111111111111111111100000000000000000000000
# SQ_TempA_Out_T3 = 49'b0011111111111111111111111100000000000000000000000
# SQ_TempB_Out_T3 = 49'b1100000000000000000000000100000000000000000000000
# SQ_isSign_Out_T4 = 1'b1
# SQ_Exp_Out_T4 = 10'b0001111111
# SQ_Temp_Out_T4 = 49'b0000000000000000000000000100000000000000000000000
# SQ_Exp_Out_T5 = 10'b0001101000
# SQ_Temp_Out_T5 = 49'b0010000000000000000000000000000000000000000000000
# Result3 = 1011010000000000000000000000000
# Error = 3'b000
```

Exp Overflow

```
# SQ_A_Out_T0 = 57'b0111111110111111111111111111111110000000000000000000000
# SQ_B_Out_T0 = 57'b0111111110111111111111111111111110000000000000000000000
# SQ_Exp_Out_T1 = 10'b0000000000
# SQ_ExpDiff_Out_T1 = 10'b0000000000
# SQ_A_Out_T2 = 57'b0111111110111111111111111111111110000000000000000000000
# SQ_B_Out_T2 = 57'b0111111110111111111111111111111110000000000000000000000
# SQ_TempA_Out_T3 = 49'b0011111111111111111111111110000000000000000000000
# SQ_TempB_Out_T3 = 49'b0011111111111111111111111110000000000000000000000
# SQ_isSign_Out_T4 = 1'b0
# SQ_Exp_Out_T4 = 10'b0011111111
# SQ_Temp_Out_T4 = 49'b0111111111111111111111111000000000000000000000000
# SQ_Exp_Out_T5 = 10'b0100000000
# SQ_Temp_Out_T5 = 49'b0011111111111111111111111110000000000000000000000
# Result4 = 00000000011111111111111111111111
# Error = 3'b100
```

A= -12.558, B= -7.309 , A+B =?

```
# SQ_A_Out_T0 = 57'b1100000100110010001110110110010001000000000000000000000
# SQ_B_Out_T0 = 57'b1100000010111010011110001101010100000000000000000000000
# SQ_Exp_Out_T1 = 10'b0000000001
# SQ_ExpDiff_Out_T1 = 10'b0000000001
# SQ_A_Out_T2 = 57'b1100000100110010001110110110010001000000000000000000000
# SQ_B_Out_T2 = 57'b1100000100011101001111000110101010000000000000000000000
# SQ_TempA_Out_T3 = 49'b1100110111000100100110111100000000000000000000000
# SQ_TempB_Out_T3 = 49'b1110001011000011100101011000000000000000000000000
# SQ_isSign_Out_T4 = 1'b1
# SQ_Exp_Out_T4 = 10'b0010000010
# SQ_Temp_Out_T4 = 49'b0100111101110111110011101100000000000000000000000
# SQ_Exp_Out_T5 = 10'b0010000011
# SQ_Temp_Out_T5 = 49'b0010011110111011111001110110000000000000000000000
# Result5 = 11000001100111101110111110011110
# Error = 3'b000
```

A= 111.7762, B= 302.4409 , A+B =?

```
# SQ_A_Out_T0 = 57'b0100001010110111111000110101101010000000000000000000000
# SQ_B_Out_T0 = 57'b0100001110100101110011100001101111000000000000000000000
# SQ_Exp_Out_T1 = 10'b1111111110
# SQ_ExpDiff_Out_T1 = 10'b0000000010
# SQ_A_Out_T2 = 57'b0100001110001101111110001101011010100000000000000000000
# SQ_B_Out_T2 = 57'b0100001110100101110011100001101111000000000000000000000
# SQ_TempA_Out_T3 = 49'b0000110111111000110101101010000000000000000000000
# SQ_TempB_Out_T3 = 49'b0010010111001110000110111100000000000000000000000
# SQ_isSign_Out_T4 = 1'b0
```

```
# SQ_Exp_Out_T4 = 10'b0010000111
# SQ_Temp_Out_T4 = 49'b0011001111000110111100100110000000000000000000000
# SQ_Exp_Out_T5 = 10'b0010000111
# SQ_Temp_Out_T5 = 49'b0011001111000110111100100110000000000000000000000
# Result6 = 01000011110011110001101111001010
# Error = 3'b000
```

```
A= 2112.2012, B= -2002.2012 , A+B =?
# SQ_A_Out_T0 = 57'b010001010010000100000000110011100000000000000000000000000
# SQ_B_Out_T0 = 57'b110001001011111010010001100111000000000000000000000000000
# SQ_Exp_Out_T1 = 10'b0000000001
# SQ_ExpDiff_Out_T1 = 10'b0000000001
# SQ_A_Out_T2 = 57'b010001010010000100000000110011100000000000000000000000000
# SQ_B_Out_T2 = 57'b110001010001111101001000110011100000000000000000000000000
# SQ_TempA_Out_T3 = 49'b0010000100000000110011100000000000000000000000000
# SQ_TempB_Out_T3 = 49'b1110000010110111001100100000000000000000000000000
# SQ_isSign_Out_T4 = 1'b0
# SQ_Exp_Out_T4 = 10'b0010001010
# SQ_Temp_Out_T4 = 49'b0000000110111000000000000000000000000000000000000
# SQ_Exp_Out_T5 = 10'b0010000101
# SQ_Temp_Out_T5 = 49'b0011011100000000000000000000000000000000000000000
# Result7 = 01000010110111000000000000000000
# Error = 3'b000
```

==========

实验三十一几乎是看打印结果 ... 口头说明真的作不了什么。在这个实验里，我们从实验二十里的浮点数加法模块的初模型，转换为即时操作的形态。在转换的过程中，我们依据每隔步骤建立一个组合逻辑/即时模块。从理解上，实验三十一有 7 个任务模块，亦即有 7 个即时层。根据即时操作的原理，无论多少叠加多少即时层，都会会引发即时事件。也就是说，即时层的长度和使用 N 个时钟操作时没有缘分的，因为即时事件是无视时钟的操作。

5.13 组合逻辑的压力

在前面，笔者说过即时操作在理想时序上是引发即时事件，多少即时层的即时操作都会在一个时钟内完成。如果我们把内容放过来看的话，即时事件，或者说组合逻辑到底如何“影响物理时序”呢？在此之前，我们需要先搞明白什么是组合逻辑的压力。

即时操作的基础就是组合逻辑，然而即时操作所建立的模块都是大功能的组合逻辑，凡事组合逻辑都有两方面的问题，亦即“面积布局”和“延迟压力”。延迟压力比较容易理解，它是指任何一个逻辑门对输入信号造成的输出延迟，如图 5.13.1 所示：

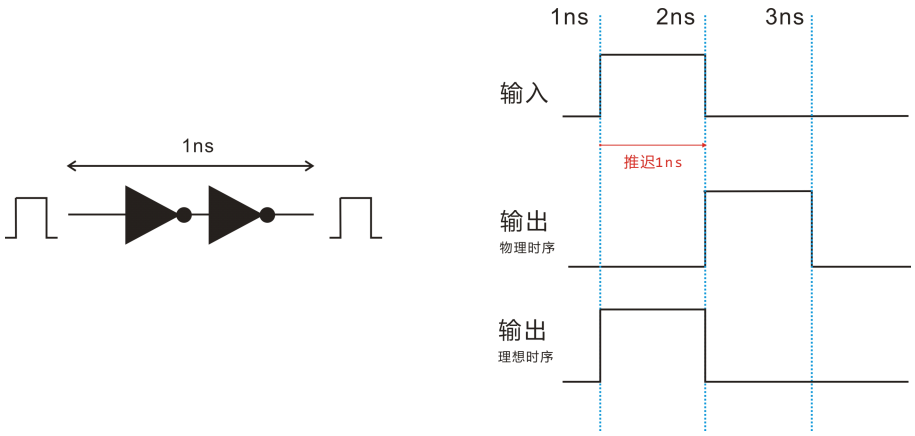


图 5.13.1 组合逻辑的延迟压力在不同时序上的变现。

图 5.13.1 是一对串联的取反逻辑门，假设一个门的延迟为 0.5ns 的话，那么一对串联哦取反门就会造成 1ns 的延迟压力。如 5.13.1 的右图所示，假设一个高脉冲在 1ns 发送，对于物理时序来说高脉冲经过延迟 1ns 以后，在 2ns 中出发。反之在理想时序来说，组合逻辑的延迟压力会直接被无视。呵呵，这是学习 Verilog HDL 语言，初学者们常常搞混的地方 ... 它们认为 Verilog HDL 语言和普通的硬件一样会显示物理时序。

笔者没有说用 Verilog HDL 语言描述的逻辑门，没有任何延迟。根据笔者的理解，凡事用 Verilog HDL 语言建立的模块，只要还没有下载到板子的一天，它的时序完全是理想的。然而用理想时序的思考方式，才能完全了解 Verilog HDL 语言的行为。至于程序下载到板子以后，FPGA 的内部到底有什么延迟，一般上都可以不用关心，因为编译器都会自动优化物理时序（如一对串联的取反门，会直接优化为一条直线）。除非编译器警告物理时序违规什么的，或者设计者自己皮痒要窥探物理时序，此时我们才会进行静态时序分析。

关于面积布局是什么，这一点比较难解释 ... 不过笔者可以用比较的方法来说明。有一些读者常常觉得非常奇怪，为什么笔者都不喜欢用 Verilog HDL 语言自带的 * / % 等操作符？早期的理由是这些操作符贪食的原因，现在的理由是“Verilog HDL 语言自带 * / % 等操作符只有一种操作模式而已”这话何解呢？读者还记得在校学习数字系统的时候，一个 4 位的乘法器是由什么组成的呢？对，就是有全加器，半加器，代位全加器或者半加器的，具体如图 5.13.2 所示：

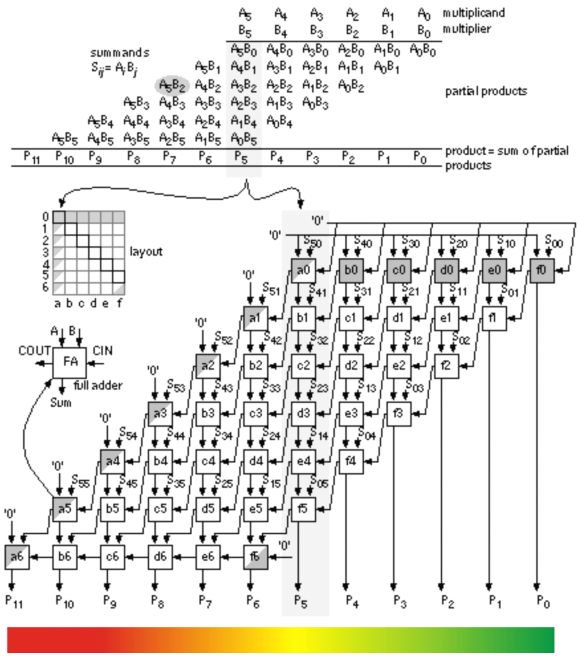


图 5.13.2 组合逻辑型的阵列乘法器。

图 5.13.2 是非常典型—由组合逻辑带进位的全加器建立而成的乘法器。对于笔者来说典型的乘法器太硬了，不但建模时不方便，而且也不支持循环操作的形态，这种不可爱的东西，能让笔者爱上它吗？当然不行啦，因此笔者太会力挺拥有算法性质的乘法器。它不但可以指定为初模型，它也能转换成流水操作，循环操作和转换成即时操作等。这样百变的小家伙，难道不可爱吗？就像比卡丘一样。不过拥有算法性质的乘法器比起典型的乘法器相比，确实更消耗组合逻辑，这一点与“表达能力”的带价带来以外，还有“面积布局”。

读者注意到图 5.13.2 下方的血条了吗？越右边就是颜色越接近红色，越左边颜色越接近绿色。那么这个血条的颜色代表什么呢？读者请注意图中的 P0 结果，为了取得 P0 的结果，输入信号只要进入一个组合逻辑而已，换之越右边的 PX 经过的组合逻辑会越多，因此延迟压力就会越大。这就是组合逻辑在面积布局上不均衡上所带来不平衡的延迟压力。

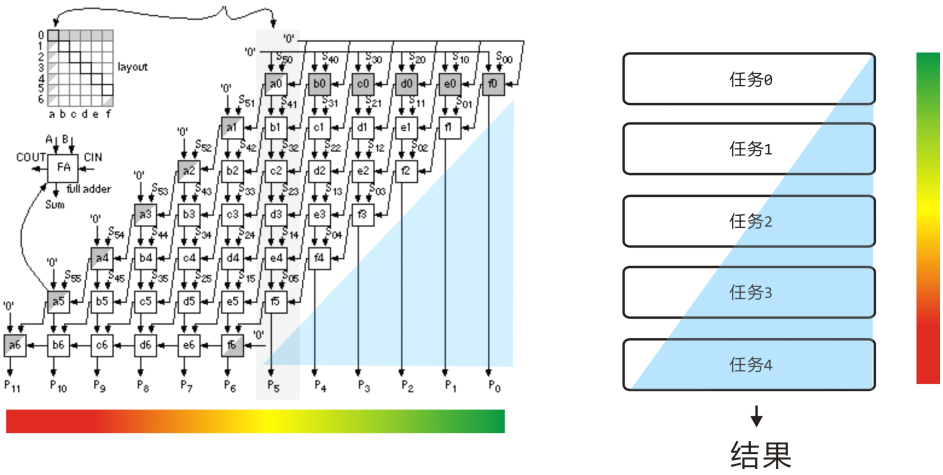


图 5.13.3 面积布局和延迟压力。

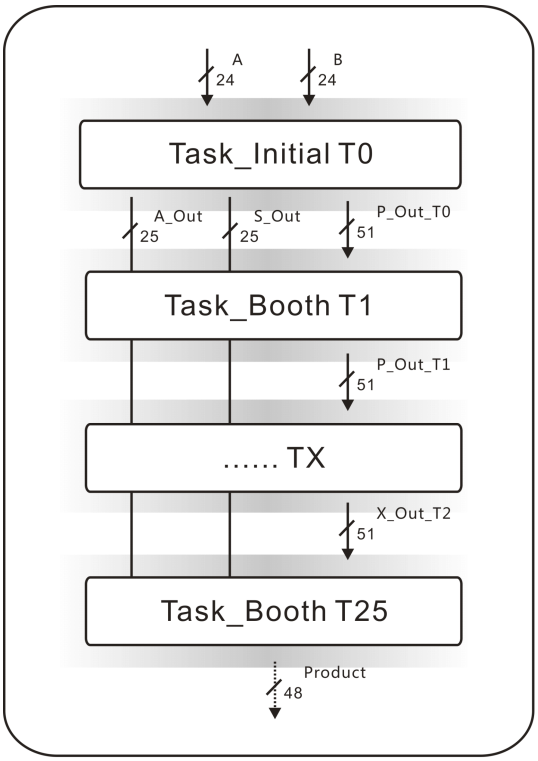
如图 5.13.3 所示，右边的图假设是 即时式 4 位的 booth 算法乘法器，除了任务 0 的初始化步骤以外，其余的任务 1~4 是取得结果的算法操作过程。我们稍微对比一下两张图，右边的建模图，在立于任务 0 的时候，延迟压力呈现绿色，以此类推 ... 当抵达任务 4 的时候，取得的结果均呈现红色的延迟压力，不过是平衡的结果。相比之下，左边的建模图取得的结果是拥有不平衡的延迟压力。

至于蓝色透明的三角体就是在面积布局上所消耗的逻辑资源示意图，由于由于左边的建模图省下部分的资源，所以造就节省逻辑资源，产生延迟压力不平衡的结果。反之右边，虽然消耗了部分的逻辑资源，不过带起了提高表达能力，建模方便，和延迟压力平衡的结果。

在这里可能有同学要问："知道这些对学习 Verilog HDL 语言有什么关系？"虽然在理想时序的角度上，组合逻辑怎样生怎样死确实不关我们的事。不过有关即时操作的任何信息，笔者都有义务表达出来，读者也有义务了解一下，这一切都是在参考书上看不到的内容。还记得这章笔记的中心内容是什么吗？对就是"平衡"。读者千万要记得，即时操作的基础是组合逻辑，组合逻辑虽然好用，但是组合逻辑是双剑刃，用不好倒会受伤。

至于延迟压力和不平衡的延迟压力在物理时序与理想时序上到底有什么影响？笔者先卖个关子，后面笔者会继续讨论的。我们先来建立一个 24 位的即时式-Booth 算法乘法器，在后面的实验里我们会用到。

实验三十二：即时式-Booth 算法乘法器（24 位）



immediate_booth_multiplier_module.v

图 5.13.4 实验三十二的建模图

图 5.13.4 是实验三十二里要建立的 immediate_booth_multiplier_module，基本上，它和实验三十相差不多，改变的除了 A 与 B 的输入位宽之外，还有相对应的任务模块数。

immediate_booth_multiplier_module.v

```
module Task_Initial
(
        input [24:0]A_In,B_In,
        output [24:0]A_Out,S_Out,
        output [50:0]P_Out
);

        assign A_Out = A_In;
        assign S_Out = ( ~A_In + 1'b1);
        assign P_Out = { 25'd0 , B_In , 1'b0 };

endmodule
```

Task_Initial 和实验三十的 Task_PreConfig 的功能基本上都一样，不过是在命名上区别以外，还有相关的位宽修改。

```
module Task_Booth
(
        input [50:0]P_In,
        input [24:0]A_In,S_In,

        output [50:0]P_Out
);

        wire [24:0]Diff1 = P_In[50:26] + A_In;
        wire [24:0]Diff2 = P_In[50:26] + S_In;

        /******************************/

        wire is1 = (P_In[1:0] == 2'b01);
        wire is2 = (P_In[1:0] == 2'b10);
        wire is3 = (P_In[1:0] == 2'b00 || P_In[1:0] == 2'b11);

        /*******************************/

        reg [50:0]P;

        always @ ( * )
```

```
            if( is1 ) P = { Diff1[24] , Diff1 , P_In[25:1] };
            else if( is2 ) P = { Diff2[24] , Diff2 , P_In[25:1] };
            else if( is3 ) P = { P_In[50] , P_In[50:1] };
            else P = 51'dx;

        assign P_Out = P;

endmodule
```

Task_Booth 是 booth 算法的处理模块，除了位宽的改变以外，还有就是采用了适合笨蛋的写法，呵呵。

```
module immediate_booth_multiplier_module
(
        input [23:0]A,B,
        output [47:0]Product
);

        wire [24:0]A_Out_T0,S_Out_T0;
        wire [50:0]P_Out_T0;
        Task_Initial T0( {1'b0, A}, {1'b0, B}, A_Out_T0, S_Out_T0, P_Out_T0 );

        wire [50:0]P_Out_T1;
        Task_Booth T1( P_Out_T0, A_Out_T0, S_Out_T0, P_Out_T1 );

        wire [50:0]P_Out_T2;
        Task_Booth T2( P_Out_T1, A_Out_T0, S_Out_T0, P_Out_T2 );

        wire [50:0]P_Out_T3;
        Task_Booth T3( P_Out_T2, A_Out_T0, S_Out_T0, P_Out_T3);

        wire [50:0]P_Out_T4;
        Task_Booth T4( P_Out_T3, A_Out_T0, S_Out_T0, P_Out_T4);

        wire [50:0]P_Out_T5;
        Task_Booth T5( P_Out_T4, A_Out_T0, S_Out_T0, P_Out_T5);

        wire [50:0]P_Out_T6;
        Task_Booth T6( P_Out_T5, A_Out_T0, S_Out_T0, P_Out_T6);

        wire [50:0]P_Out_T7;
        Task_Booth T7( P_Out_T6, A_Out_T0, S_Out_T0, P_Out_T7);

        wire [50:0]P_Out_T8;
```

```
33.	Task_Booth T8( P_Out_T7, A_Out_T0, S_Out_T0, P_Out_T8);
34.
35.	wire [50:0]P_Out_T9;
36.	Task_Booth T9( P_Out_T8, A_Out_T0, S_Out_T0, P_Out_T9);
37.
38.	wire [50:0]P_Out_T10;
39.	Task_Booth T10( P_Out_T9, A_Out_T0, S_Out_T0, P_Out_T10);
40.
41.	wire [50:0]P_Out_T11;
42.	Task_Booth T11( P_Out_T10, A_Out_T0, S_Out_T0, P_Out_T11);
43.
44.	wire [50:0]P_Out_T12;
45.	Task_Booth T12( P_Out_T11, A_Out_T0, S_Out_T0, P_Out_T12);
46.
47.	wire [50:0]P_Out_T13;
48.	Task_Booth T13( P_Out_T12, A_Out_T0, S_Out_T0, P_Out_T13);
49.
50.	wire [50:0]P_Out_T14;
51.	Task_Booth T14( P_Out_T13, A_Out_T0, S_Out_T0, P_Out_T14);
52.
53.	wire [50:0]P_Out_T15;
54.	Task_Booth T15( P_Out_T14, A_Out_T0, S_Out_T0, P_Out_T15);
55.
56.	wire [50:0]P_Out_T16;
57.	Task_Booth T16( P_Out_T15, A_Out_T0, S_Out_T0, P_Out_T16);
58.
59.	wire [50:0]P_Out_T17;
60.	Task_Booth T17( P_Out_T16, A_Out_T0, S_Out_T0, P_Out_T17);
61.
62.	wire [50:0]P_Out_T18;
63.	Task_Booth T18( P_Out_T17, A_Out_T0, S_Out_T0, P_Out_T18);
64.
65.	wire [50:0]P_Out_T19;
66.	Task_Booth T19( P_Out_T18, A_Out_T0, S_Out_T0, P_Out_T19);
67.
68.	wire [50:0]P_Out_T20;
69.	Task_Booth T20( P_Out_T19, A_Out_T0, S_Out_T0, P_Out_T20);
70.
71.	wire [50:0]P_Out_T21;
72.	Task_Booth T21( P_Out_T20, A_Out_T0, S_Out_T0, P_Out_T21);
73.
74.	wire [50:0]P_Out_T22;
75.	Task_Booth T22( P_Out_T21, A_Out_T0, S_Out_T0, P_Out_T22);
76.
77.	wire [50:0]P_Out_T23;
```

```
78.         Task_Booth T23( P_Out_T22, A_Out_T0, S_Out_T0, P_Out_T23);
79.
80.         wire [50:0]P_Out_T24;
81.         Task_Booth T24( P_Out_T23, A_Out_T0, S_Out_T0, P_Out_T24);
82.
83.         wire [50:0]P_Out_T25;
84.         Task_Booth T25( P_Out_T24, A_Out_T0, S_Out_T0, P_Out_T25);
85.
86.         assign Product = P_Out_T25[48:1];
87.
88.     endmodule
```

第 3~4 行是相关的输入和输出声明，第 7~84 行是相关的实例化动作，基本上和建模图差不多，自己看着吧。

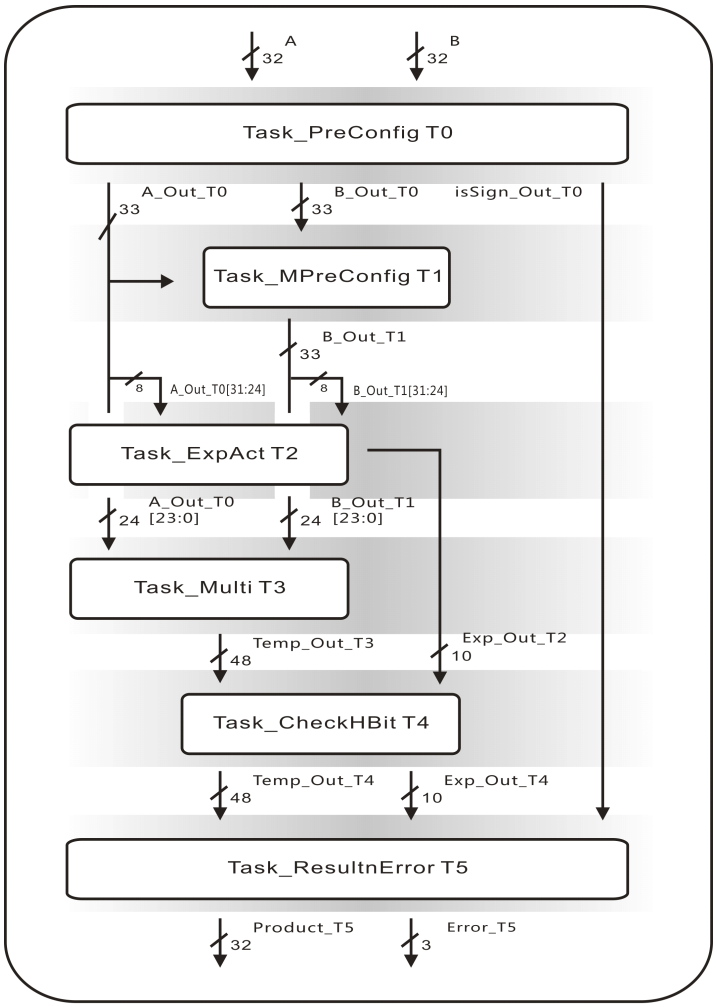
immediate_booth_multiplier_module.vt

```
1.      `timescale 1 ps/ 1 ps
2.      module immediate_booth_multiplier_module_simulation();
3.
4.          reg CLK;
5.          reg RSTn;
6.
7.          reg [23:0]A;
8.          reg [23:0]B;
9.
10.         wire [47:0]Product;
11.
12.         /*************************/
13.
14.         immediate_booth_multiplier_module U1
15.           (
16.                 .A(A),
17.                 .B(B),
18.                 .Product( Product )
19.         );
20.
21.         /****************************/
22.
23.         initial
24.           begin
25.                 RSTn = 0; #10 RSTn = 1;
26.                 CLK = 0; forever #5 CLK = ~CLK;
```

```
27.          end
28.
29.        /******************************/
30.
31.        reg [5:0]i;
32.
33.          always @ (    posedge CLK or negedge RSTn )
34.                if( !RSTn )
35.                     begin
36.                       A <= 24'd0;
37.                       B <= 24'd0;
38.                       i <= 6'd0;
39.                  end
40.              else
41.                  case( i )
42.
43.                              0:
44.                              begin A <= 24'd300; B <= 24'd400; i <= i + 1'b1; end
45.
46.                              1:
47.                              begin $display( " %b * %b = %b ", A, B, Product ); i <= i + 1'b1; end
48.
49.                              2:
50.                             begin A <= 24'HFFFFFF; B <= 24'HFFFFFF; i <= i + 1'b1; end
51.
52.                              3:
53.                              begin $display( " %b * %b = %b ", A, B, Product ); i <= i + 1'b1; end
54.
55.                             4:
56.                              i <= i;
57.
58.                       endcase
59.
60.   endmodule
```

激励文件也写得非常简单，主要是测试 300 * 400 的结果，然后再测试全 1 值相乘的结果 … 仿真图就不贴出来丢脸了，读者独自自己去乐吧~笑。实验三十二非常简单而已，主要是从实验三十的基础上，用更笨蛋的方法去编辑即时式-booth 算法乘法器而已。

*实验三十三：即时式-浮点数乘法模块（Verilog 自带的 *操作符）*



immediate_float_multi_module.v

图 5.13.5 即时式浮点数加乘法模块的建模图。

当读者看到图 5.13.5 头皮会不会发麻一会呢？这个看似蜘蛛网的东西就是实验三十三的建模图（图形），虽然图形之间的连线麻烦了一点，笔者绘图的时候也差点吐血。不过实际上，代码的编辑可是非常简单而已，废话少说我们一个一个任务模块来浏览吧：

immediate_float_multi_module.v

```
module Task_PreConfig
(
        input [31:0]A_In,B_In,
        output [32:0]A_Out,B_Out,
        output isSign_Out
);

```

```
63.         assign A_Out = { A_In[31], A_In[30:23], 1'b1, A_In[22:0] };
64.         assign B_Out = { B_In[31], B_In[30:23], 1'b1, B_In[22:0] };
65.         assign isSign_Out = A_In[31] ^ B_In[31];
66.
67.     endmodule
```

Task_PreConfig 表示操作数预处理的任务模块，其中第 58~60 行是该模块的输入输出声明，第 63~65 行是预处理以后的结果输出和驱动。

```
69.     module Task_MPreConfig
70.     (
71.         input [32:0]A_In,B_In,
72.         output [32:0]B_Out
73.     );
74.
75.         wire [23:0]BM1 = B_In[23:0] >> 1'b1;
76.         wire [7:0]BE1 = B_In[31:24] + 1'b1;
77.         wire [32:0]B1 = { B_In[32], BE1, BM1};
78.
79.         wire isMove = A_In[31:24] == B_In[31:24] && ( A_In[22:0] != 23'd0 & B_In[22:0] != 23'd0 )
80.                 || A_In[32] ^ B_In[32];
81.
82.         assign B_Out = isMove ? B1 : B_In;
83.
84.     endmodule
```

Task_MPreConfig 是对应移位预处理的任务模块，第 71~72 行是相关的输入输出声明；第 75 行先取得移位预处理中，尾数右移一位的结果；第 76 行是先取得移位预处理中，价码加一的结果；第 77 行是先取得，第 75~76 行整合的结果（包裹符号位）。第 77 行是取得移位的可能性，至于第 82 行根据 isMove 的结果来决定驱动 B_Out 是第 77 行的 B1 还是 B_In 呢？

```
86.     module Task_ExpAct
87.     (
88.         input [7:0]A_Exp_In, B_Exp_In,
89.         output [9:0]Exp_Out
90.     );
91.           wire [9:0]BDiff = B_Exp_In - 8'd127;
92.
93.           assign Exp_Out = A_Exp_In + BDiff;
94.
95.     endmodule
```

Task_ExpAct 是取得新价码操作的任务模块，第 88~89 行是相关的输入输出声明，第 91 行先取得价码差，然后在输出 Exp_Out 之前，将 A_Exp_In 和 BDiff 相加。

```
97.   module Task_Multi
98.   (
99.       input [23:0]A_M_In,B_M_In,
100.      output [47:0]Temp_Out
101.  );
102.
103.    assign Temp_Out = A_M_In * B_M_In;
104.
105.  endmodule
```

这个任务模块就不多解释了，在 103 行中笔者先贪方便，用系统 Verilog HDL 语言自带的 * 操作符。

```
107.  module Task_CheckHBit
108.  (
109.      input [9:0]Exp_In,
110.      input [47:0]Temp_In,
111.
112.      output [9:0]Exp_Out,
113.      output [47:0]Temp_Out
114.  );
115.
116.    wire is1 = Temp_In[47];
117.    wire is2 = Temp_In[46];
118.    wire is3 = Temp_In[45];
119.
120.      wire [47:0]Temp1 = Temp_In;
121.      wire [47:0]Temp2 = Temp_In << 2'd1;
122.      wire [47:0]Temp3 = Temp_In << 2'd2;
123.
124.      wire [9:0]Exp1 = Exp_In;
125.      wire [9:0]Exp2 = Exp_In;
126.      wire [9:0]Exp3 = Exp_In - 1'b1;
127.
128.      reg [47:0]Temp;
129.      reg [9:0]Exp;
130.
131.      always @ (*)
132.        if( is1 ) begin Temp = Temp1; Exp = Exp1; end
133.        else if( is2 ) begin Temp = Temp2; Exp = Exp2; end
134.        else if( is3 ) begin Temp = Temp3; Exp = Exp3; end
135.        else begin Temp = 48'dx; Exp = 10'dx; end
136.
```

```
137.    assign Temp_Out = Temp;
138.    assign Exp_Out = Exp;
139.
140.  endmodule
```

Task_CheckHBit 是结果调整的任务模块，第 109~113 行是相关的输入输出声明；第 116~118 行先取得有关隐藏位的可能性；第 120~122 行先取得任何尾数移位的可能性；第 124~126 行先取得任何价码加减的可能性；第 131~135 行更具隐藏位的可能性，再来赋值 Temp 和 Exp，最后在 137~138 行将结果输出。

```
142.  module Task_ResultnError
143.  (
144.      input isSign_In,
145.      input [9:0]Exp_In,
146.      input [47:0]Temp_In,
147.
148.      output [2:0]Error,
149.      output [31:0]Result
150.  );
151.
152.      wire isZero = ( Temp_In[47:24] == 24'd0 );
153.      wire isUnder = ( Exp_In[9:8] == 2'b11 );
154.      wire isOver = ( Exp_In[9:8] == 2'b01 );
155.
156.      wire isNormalise = Temp_In[23];
157.      wire [31:0]Result1 = { isSign_In, Exp_In[7:0], Temp_In[46:24] + 1'b1 };
158.      wire [31:0]Result2 = { isSign_In, Exp_In[7:0], Temp_In[46:24] };
159.
160.      assign Error = { isOver, isUnder, isZero };
161.      assign Result = isNormalise ? Result1 : Result2;
162.
163.  endmodule
```

Task_ResultnError 就是对应输出和格式化的任务模块，第 144~149 行是相关的输入输出声明；第 152~154 行取得不同的错误标志；第 156 行取得四舍五入的可能性；第 157 行取得经过四舍五入后的规格化结果，然而第 158 行是正常规格化的结果。最后的第 160 行将 3 种错误标志输出，第 161 行则根据四舍五入的可能性来决定输出结果是 Result1 还是 Result2。

```
1.    module immediate_float_multi_module
2.    (
3.        input [31:0]A,B,
4.        output [31:0]Result,
5.        output [2:0]Error,
6.
```

```
7.          /******************/
8.
9.          output [32:0]SQ_A_Out_T0,SQ_B_Out_T0,
10.         output SQ_isSign_Out_T0,
11.         output [32:0]SQ_B_Out_T1,
12.         output [9:0]SQ_Exp_Out_T2,
13.         output [47:0]SQ_Temp_Out_T3,
14.         output [9:0]SQ_Exp_Out_T4,
15.         output [47:0]SQ_Temp_Out_T4
16.
17.   );
18.
19.         wire [32:0]A_Out_T0,B_Out_T0;
20.         wire isSign_Out_T0;
21.         Task_PreConfig T0( A, B, A_Out_T0, B_Out_T0, isSign_Out_T0 );
22.
23.         wire [32:0]B_Out_T1;
24.         Task_MPreConfig T1( A_Out_T0, B_Out_T0, B_Out_T1 );
25.
26.         wire [9:0]Exp_Out_T2;
27.         Task_ExpAct T2( A_Out_T0[31:24], B_Out_T1[31:24], Exp_Out_T2 );
28.
29.         wire [47:0]Temp_Out_T3;
30.         Task_Multi T3( A_Out_T0[23:0], B_Out_T1[23:0], Temp_Out_T3 );
31.
32.         wire [9:0]Exp_Out_T4;
33.         wire [47:0]Temp_Out_T4;
34.         Task_CheckHBit T4( Exp_Out_T2, Temp_Out_T3, Exp_Out_T4, Temp_Out_T4 );
35.
36.         wire [2:0]Error_T5;
37.         wire [31:0]Result_T5;
38.         Task_ResultnError T5( isSign_Out_T0, Exp_Out_T4, Temp_Out_T4, Error_T5, Result_T5 );
39.
40.         assign Error = Error_T5;
41.         assign Result = Result_T5;
42.
43.         /*******************************/
44.
45.         assign SQ_A_Out_T0 = A_Out_T0;
46.         assign SQ_B_Out_T0 = B_Out_T0;
47.         assign SQ_isSign_Out_T0 = isSign_Out_T0;
48.         assign SQ_B_Out_T1 = B_Out_T1;
49.         assign SQ_Exp_Out_T2 = Exp_Out_T2;
50.         assign SQ_Temp_Out_T3 = Temp_Out_T3;
51.         assign SQ_Exp_Out_T4 = Exp_Out_T4;
```

```
52.         assign SQ_Temp_Out_T4 = Temp_Out_T4;
53.
54.     endmodule
```

第 3~5 行是该模块的输入输出声明，第 9~15 行是仿真用的输出。第 19~38 行则是根据图 5.13.5 来实例化和建模，读者自己看着办吧。最后的 45~52 行同是仿真用的驱动输出。

immediate_float_multi_module.vt

```
1.      `timescale 1 ps/ 1 ps
2.      module immediate_float_multi_module_simulation();
3.
4.              reg CLK1;
5.              reg RSTn;
6.
7.              reg [31:0] A;
8.              reg [31:0] B;
9.
10.             wire [31:0]Result;
11.             wire [2:0]Error;
12.
13.             /*************************/
14.
15.             wire [32:0]SQ_A_Out_T0,SQ_B_Out_T0;
16.           wire SQ_isSign_Out_T0;
17.           wire [32:0]SQ_B_Out_T1;
18.           wire [9:0]SQ_Exp_Out_T2;
19.           wire [47:0]SQ_Temp_Out_T3;
20.           wire [9:0]SQ_Exp_Out_T4;
21.           wire [47:0]SQ_Temp_Out_T4;
22.
23.             /*************************/
24.
25.             immediate_float_multi_module U1
26.               (
27.                     .A(A),
28.                     .B(B),
29.                     .Result(Result),
30.                     .Error( Error ),
31.                     /*****************/
32.                    .SQ_A_Out_T0 ( SQ_A_Out_T0 ),
33.                    .SQ_B_Out_T0 ( SQ_B_Out_T0 ),
34.                    .SQ_isSign_Out_T0( SQ_isSign_Out_T0 ),
```

```
35.                 .SQ_B_Out_T1( SQ_B_Out_T1 ),
36.                 .SQ_Exp_Out_T2( SQ_Exp_Out_T2 ),
37.                 .SQ_Temp_Out_T3( SQ_Temp_Out_T3 ),
38.                 .SQ_Exp_Out_T4( SQ_Exp_Out_T4 ),
39.                 .SQ_Temp_Out_T4( SQ_Temp_Out_T4 )
40.           );
41.
42.         /****************************/
43.
44.         initial
45.           begin
46.                 RSTn = 0; #10 RSTn = 1;
47.                 CLK1 = 0; forever #5 CLK1 = ~CLK1;
48.           end
49.
50.         /****************************/
51.
52.         reg [3:0]i;
53.
54.           always @ (    posedge CLK1 or negedge RSTn )
55.                 if( !RSTn )
56.                       begin
57.                         A <= 32'd0;
58.                         B <= 32'd0;
59.                         i <= 4'd0;
60.                     end
61.               else
62.                     case( i )
63.
64.                             0: // 2.5 * 5
65.                             begin A <= 32'b01000000001000000000000000000000; B <= 32'b01000000101000000000000000000000; i <= i + 1'b1; end
66.
67.                            1:
68.                            begin
69.                                 $display( "32'b%32b * 32'b%32b ", A, B );
70.                                 $display( "SQ_A_Out_T0 = 33'b%33b ",SQ_A_Out_T0 );
71.                                 $display( "SQ_B_Out_T0 = 33'b%33b ",SQ_B_Out_T0 );
72.                                 $display( "isSign_Out_T0",SQ_isSign_Out_T0 );
73.                                  $display( "SQ_B_Out_T1 = 33'b%33b",SQ_B_Out_T1 );
74.                                  $display( "SQ_Exp_Out_T2 = 10'b%10b",SQ_Exp_Out_T2 );
75.                                  $display( "SQ_Temp_Out_T3 = 48'b%48b",SQ_Temp_Out_T3 );
76.                                  $display( "SQ_Exp_Out_T4 = 10'b%10b",SQ_Exp_Out_T4 );
77.                                  $display( "SQ_Temp_Out_T4 = 48'b%48b",SQ_Temp_Out_T4 );
78.                                 $display( "Result = 32'b%32b",Result    );
79.                                 i <= i + 1'b1;
```

```
80.                 end
81.
82.                 2: //3.142 * 3.142
83.                 begin A <= 32'b01000000010010010001011010000111; B <= 32'b01000000010010010001011010000111; i <= i + 1'b1; end
84.
85.                 3:
86.                 begin
87.                     $display( "32'b%32b * 32'b%32b ", A, B );
88.                     $display( "SQ_A_Out_T0 = 33'b%33b ",SQ_A_Out_T0 );
89.                     $display( "SQ_B_Out_T0 = 33'b%33b ",SQ_B_Out_T0 );
90.                     $display( "isSign_Out_T0",SQ_isSign_Out_T0 );
91.                     $display( "SQ_B_Out_T1 = 33'b%33b",SQ_B_Out_T1 );
92.                     $display( "SQ_Exp_Out_T2 = 10'b%10b",SQ_Exp_Out_T2 );
93.                     $display( "SQ_Temp_Out_T3 = 48'b%48b",SQ_Temp_Out_T3 );
94.                     $display( "SQ_Exp_Out_T4 = 10'b%10b",SQ_Exp_Out_T4 );
95.                     $display( "SQ_Temp_Out_T4 = 48'b%48b",SQ_Temp_Out_T4 );
96.                     $display( "Result = 32'b%32b",Result    );
97.                     i <= i + 1'b1;
98.                 end
99.
100.                4: //12.662 * 4.903
101.                begin A <= 32'b01000001010010101001011110001101; B <= 32'b01000000100111001110010101100000; i <= i + 1'b1; end
102.
103.                5:
104.                begin
105.                    $display( "32'b%32b * 32'b%32b ", A, B );
106.                    $display( "SQ_A_Out_T0 = 33'b%33b ",SQ_A_Out_T0 );
107.                    $display( "SQ_B_Out_T0 = 33'b%33b ",SQ_B_Out_T0 );
108.                    $display( "isSign_Out_T0",SQ_isSign_Out_T0 );
109.                    $display( "SQ_B_Out_T1 = 33'b%33b",SQ_B_Out_T1 );
110.                    $display( "SQ_Exp_Out_T2 = 10'b%10b",SQ_Exp_Out_T2 );
111.                    $display( "SQ_Temp_Out_T3 = 48'b%48b",SQ_Temp_Out_T3 );
112.                    $display( "SQ_Exp_Out_T4 = 10'b%10b",SQ_Exp_Out_T4 );
113.                    $display( "SQ_Temp_Out_T4 = 48'b%48b",SQ_Temp_Out_T4 );
114.                    $display( "Result = 32'b%32b",Result    );
115.                    i <= i + 1'b1;
116.                end
117.
118.                6: // 2 * 2
119.                begin A <= 32'b01000000000000000000000000000000; B <= 32'b01000000000000000000000000000000; i <= i + 1'b1; end
120.
121.                7:
122.                begin
123.                    $display( "32'b%32b * 32'b%32b ", A, B );
124.                    $display( "SQ_A_Out_T0 = 33'b%33b ",SQ_A_Out_T0 );
```

```
125.                    $display( "SQ_B_Out_T0 = 33'b%33b ",SQ_B_Out_T0 );
126.                    $display( "isSign_Out_T0",SQ_isSign_Out_T0 );
127.                    $display( "SQ_B_Out_T1 = 33'b%33b",SQ_B_Out_T1 );
128.                    $display( "SQ_Exp_Out_T2 = 10'b%10b",SQ_Exp_Out_T2 );
129.                    $display( "SQ_Temp_Out_T3 = 48'b%48b",SQ_Temp_Out_T3 );
130.                    $display( "SQ_Exp_Out_T4 = 10'b%10b",SQ_Exp_Out_T4 );
131.                    $display( "SQ_Temp_Out_T4 = 48'b%48b",SQ_Temp_Out_T4 );
132.                    $display( "Result = 32'b%32b",Result    );
133.                    i <= i + 1'b1;
134.                end
135.
136.                8: // 3.142 * 2
137.                begin A <= 32'b01000000010010010001011010000111; B <= 32'b01000000000000000000000000000000; i <= i + 1'b1; end
138.
139.                9:
140.                begin
141.                    $display( "32'b%32b * 32'b%32b ", A, B );
142.                    $display( "SQ_A_Out_T0 = 33'b%33b ",SQ_A_Out_T0 );
143.                    $display( "SQ_B_Out_T0 = 33'b%33b ",SQ_B_Out_T0 );
144.                    $display( "isSign_Out_T0",SQ_isSign_Out_T0 );
145.                    $display( "SQ_B_Out_T1 = 33'b%33b",SQ_B_Out_T1 );
146.                    $display( "SQ_Exp_Out_T2 = 10'b%10b",SQ_Exp_Out_T2 );
147.                    $display( "SQ_Temp_Out_T3 = 48'b%48b",SQ_Temp_Out_T3 );
148.                    $display( "SQ_Exp_Out_T4 = 10'b%10b",SQ_Exp_Out_T4 );
149.                    $display( "SQ_Temp_Out_T4 = 48'b%48b",SQ_Temp_Out_T4 );
150.                    $display( "Result = 32'b%32b",Result    );
151.                    i <= i + 1'b1;
152.                end
153.
154.                10: // 73.767 * 83.266
155.                begin A <= 32'b01000010100100111000100010110100; B <= 32'b01000010101001101000100000110001; i <= i + 1'b1; end
156.
157.            11:
158.                begin
159.                    $display( "32'b%32b * 32'b%32b ", A, B );
160.                    $display( "SQ_A_Out_T0 = 33'b%33b ",SQ_A_Out_T0 );
161.                    $display( "SQ_B_Out_T0 = 33'b%33b ",SQ_B_Out_T0 );
162.                    $display( "isSign_Out_T0",SQ_isSign_Out_T0 );
163.                    $display( "SQ_B_Out_T1 = 33'b%33b",SQ_B_Out_T1 );
164.                    $display( "SQ_Exp_Out_T2 = 10'b%10b",SQ_Exp_Out_T2 );
165.                    $display( "SQ_Temp_Out_T3 = 48'b%48b",SQ_Temp_Out_T3 );
166.                    $display( "SQ_Exp_Out_T4 = 10'b%10b",SQ_Exp_Out_T4 );
167.                    $display( "SQ_Temp_Out_T4 = 48'b%48b",SQ_Temp_Out_T4 );
168.                    $display( "Result = 32'b%32b",Result    );
169.                    i <= i + 1'b1;
```

```
                end

                12: // 83.266 * 83.266
                begin A <= 32'b01000010101001101000100000110001; B <= 32'b01000010101001101000100000110001; i <= i + 1'b1; end

                13:
                begin
                    $display( "32'b%32b * 32'b%32b ", A, B );
                    $display( "SQ_A_Out_T0 = 33'b%33b ",SQ_A_Out_T0 );
                    $display( "SQ_B_Out_T0 = 33'b%33b ",SQ_B_Out_T0 );
                    $display( "isSign_Out_T0",SQ_isSign_Out_T0 );
                     $display( "SQ_B_Out_T1 = 33'b%33b",SQ_B_Out_T1 );
                     $display( "SQ_Exp_Out_T2 = 10'b%10b",SQ_Exp_Out_T2 );
                     $display( "SQ_Temp_Out_T3 = 48'b%48b",SQ_Temp_Out_T3 );
                     $display( "SQ_Exp_Out_T4 = 10'b%10b",SQ_Exp_Out_T4 );
                     $display( "SQ_Temp_Out_T4 = 48'b%48b",SQ_Temp_Out_T4 );
                    $display( "Result = 32'b%32b",Result    );
                    i <= i + 1'b1;
                end

                14:
                i <= i;

            endcase

endmodule
```

激励文件的编辑风格还是老样子 ... 其他的读者自己看着办吧。

仿真结果：

基本上实验三十三的仿真图是没有意义的，所以笔者就不贴图了，换之笔者将打印信息帖出来：

```
2.5 * 5
# 32'b01000000001000000000000000000000 * 32'b01000000101000000000000000000000
# SQ_A_Out_T0 = 33'b010000000101000000000000000000000
# SQ_B_Out_T0 = 33'b010000001101000000000000000000000
# isSign_Out_T00
# SQ_B_Out_T1 = 33'b010000001101000000000000000000000
# SQ_Exp_Out_T2 = 10'b0010000010
# SQ_Temp_Out_T3 = 48'b011001000000000000000000000000000000000000000000
# SQ_Exp_Out_T4 = 10'b0010000010
# SQ_Temp_Out_T4 = 48'b110010000000000000000000000000000000000000000000
```

```
# Result = 32'b01000001010010000000000000000000
```

3.142 * 3.142

```
# 32'b01000000010010010001011010000111 * 32'b01000000010010010001011010000111
# SQ_A_Out_T0 = 33'b010000000110010010001011010000111
# SQ_B_Out_T0 = 33'b010000000110010010001011010000111
# isSign_Out_T00
# SQ_B_Out_T1 = 33'b010000001011100100100010110100001 1
# SQ_Exp_Out_T2 = 10'b0010000010
# SQ_Temp_Out_T3 = 48'b010011101111101000110000100110000011001001010101
# SQ_Exp_Out_T4 = 10'b0010000010
# SQ_Temp_Out_T4 = 48'b100111011111010001100001001100000110010010101010
# Result = 32'b01000001000111011111010001100001
```

12.662 * 4.903

```
# 32'b01000001010010101001011110001101 * 32'b01000000100111001110010101100000
# SQ_A_Out_T0 = 33'b010000010110010101001011110001101
# SQ_B_Out_T0 = 33'b010000001100111001110010101100000
# isSign_Out_T00
# SQ_B_Out_T1 = 33'b010000001100111001110010101100000
# SQ_Exp_Out_T2 = 10'b0010000100
# SQ_Temp_Out_T3 = 48'b011111000010100111011111011101011111010111100000
# SQ_Exp_Out_T4 = 10'b0010000100
# SQ_Temp_Out_T4 = 48'b111110000101001110111110111010111110101111000000
# Result = 32'b01000010011110000101001110111111
```

2 * 2

```
# 32'b01000000000000000000000000000000 * 32'b01000000000000000000000000000000
# SQ_A_Out_T0 = 33'b010000000100000000000000000000000
# SQ_B_Out_T0 = 33'b010000000100000000000000000000000
# isSign_Out_T00
# SQ_B_Out_T1 = 33'b010000000100000000000000000000000
# SQ_Exp_Out_T2 = 10'b0010000001
# SQ_Temp_Out_T3 = 48'b010000000000000000000000000000000000000000000000
# SQ_Exp_Out_T4 = 10'b0010000001
# SQ_Temp_Out_T4 = 48'b100000000000000000000000000000000000000000000000
# Result = 32'b01000000100000000000000000000000
```

3.142 * 2

```
# 32'b01000000010010010001011010000111 * 32'b01000000000000000000000000000000
# SQ_A_Out_T0 = 33'b010000000110010010001011010000111
# SQ_B_Out_T0 = 33'b010000000100000000000000000000000
# isSign_Out_T00
# SQ_B_Out_T1 = 33'b010000000100000000000000000000000
```

```
# SQ_Exp_Out_T2 = 10'b0010000001
# SQ_Temp_Out_T3 = 48'b011001001000101101000011100000000000000000000000
# SQ_Exp_Out_T4 = 10'b0010000001
# SQ_Temp_Out_T4 = 48'b110010010001011010000111000000000000000000000000
# Result = 32'b01000000110010010001011010000111
```

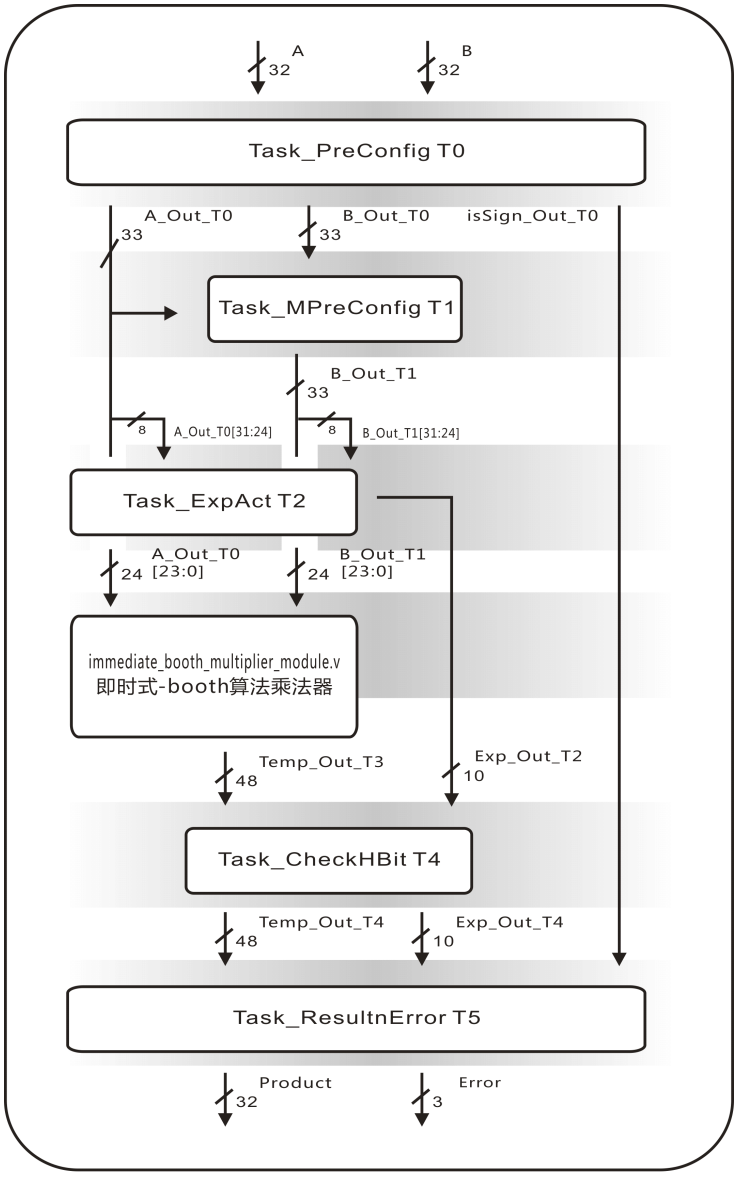
73.767 * 83.266

```
# 32'b01000010100100111000100010110100 * 32'b01000010101001101000100000110001
# SQ_A_Out_T0 = 33'b010000101100100111000100010110100
# SQ_B_Out_T0 = 33'b010000101101001101000100000110001
# isSign_Out_T00
# SQ_B_Out_T1 = 33'b010000110010100110100010000011000
# SQ_Exp_Out_T2 = 10'b0010001100
# SQ_Temp_Out_T3 = 48'b001011111111110010010000100000001010000011100000
# SQ_Exp_Out_T4 = 10'b0010001011
# SQ_Temp_Out_T4 = 48'b101111111111001001000010000000101000001110000000
# Result = 32'b01000101101111111111001001000010
```

83.266 * 83.266

```
# 32'b01000010101001101000100000110001 * 32'b01000010101001101000100000110001
# SQ_A_Out_T0 = 33'b010000101101001101000100000110001
# SQ_B_Out_T0 = 33'b010000101101001101000100000110001
# isSign_Out_T00
# SQ_B_Out_T1 = 33'b010000110010100110100010000011000
# SQ_Exp_Out_T2 = 10'b0010001100
# SQ_Temp_Out_T3 = 48'b001101100010101001110011101011001100100010011000
# SQ_Exp_Out_T4 = 10'b0010001011
# SQ_Temp_Out_T4 = 48'b110110001010100111001110101100110010001001100000
# Result = 32'b01000101110110001010100111001111
```

输出结果基本上都无误，一些精度失真没有什么大不了，笔者都在第一章里解释过了。在实验三十三里我们使用了 Verilog HDL 语言自带的 * 操作符来实现乘法，那么在下一个实验里，我们会用回自家设计的 即时式-booth 算法乘法来替代实验三十三里的 Task_Multi。读者请打起十二分精神，连续 3 个实验相连是前所未有的情况，笔者也在燃烧中。



immediate_float_multi_module.v

图 5.13.6 实验三十四的建模图。

图5.13.6基本上和图5.13.5相差不大,除了用即时式-Booth算法乘法器来替代之前的Task_Multi以外 . . . 其他的什么也没有修改。

immediate_float_multi_module.v

```
module immediate_float_multi_module2
(
    input [31:0]A,B,
    output [31:0]Result,
    output [2:0]Error,
```

```
6.
7.          /*****************/
8.
9.          output [32:0]SQ_A_Out_T0,SQ_B_Out_T0,
10.         output SQ_isSign_Out_T0,
11.         output [32:0]SQ_B_Out_T1,
12.         output [9:0]SQ_Exp_Out_T2,
13.         output [47:0]SQ_Temp_Out_T3,
14.         output [9:0]SQ_Exp_Out_T4,
15.         output [47:0]SQ_Temp_Out_T4
16.
17.   );
18.
19.         wire [32:0]A_Out_T0,B_Out_T0;
20.         wire isSign_Out_T0;
21.         Task_PreConfig T0( A, B, A_Out_T0, B_Out_T0, isSign_Out_T0 );
22.
23.         wire [32:0]B_Out_T1;
24.         Task_MPreConfig T1( A_Out_T0, B_Out_T0, B_Out_T1 );
25.
26.         wire [9:0]Exp_Out_T2;
27.         Task_ExpOperate T2( A_Out_T0[31:24], B_Out_T1[31:24], Exp_Out_T2 );
28.
29.         wire [47:0]Temp_Out_T3;
30.         immediate_booth_multiplier_module T3
31.         (
32.               .A( A_Out_T0[23:0] ),
33.               .B( B_Out_T1[23:0] ),
34.               .Product( Temp_Out_T3 )
35.         );
36.
37.         wire [9:0]Exp_Out_T4;
38.         wire [47:0]Temp_Out_T4;
39.         Task_CheckHBit T4( Exp_Out_T2, Temp_Out_T3, Exp_Out_T4, Temp_Out_T4 );
40.
41.         wire [2:0]Error_T5;
42.         wire [31:0]Result_T5;
43.         Task_ResultnError T5( isSign_Out_T0, Exp_Out_T4, Temp_Out_T4, Error_T5, Result_T5 );
44.
45.         assign Error = Error_T5;
46.         assign Result = Result_T5;
47.
48.         /******************************/
49.
50.         assign SQ_A_Out_T0 = A_Out_T0;
```

```
51.         assign SQ_B_Out_T0 = B_Out_T0;
52.         assign SQ_isSign_Out_T0 = isSign_Out_T0;
53.         assign SQ_B_Out_T1 = B_Out_T1;
54.         assign SQ_Exp_Out_T2 = Exp_Out_T2;
55.         assign SQ_Temp_Out_T3 = Temp_Out_T3;
56.         assign SQ_Exp_Out_T4 = Exp_Out_T4;
57.         assign SQ_Temp_Out_T4 = Temp_Out_T4;
58.
59.     endmodule
```

基本上修改的地方有第 30~35 行而已，我们调用实验三十二中所建立的即时式-Booth 算法乘法器来充当原本的 Task_Multi 以外 .. 就是这样而已。至于激励文件，仿真结果都和实验三十三一样，笔者就不多重复了，坐不下的同学，自己自行代开程序来看看吧。

==========

在实验结束之前，我们来谈谈节省逻辑资源和减少操作时钟的问题吧。当我们发誓要减少操作时钟的时候，如果条件允许的话，我们会选择极端性的即时操作。话说当编译器编译结束以后，逻辑资源消耗和实验二十七的轻量版-浮点数乘法器相比，前者消耗 2576 个资源逻辑，后者则消耗 399 个逻辑资源。这是一个天与地之间的差别，人类的形态就是非常奇怪 ~ 当我们像着魔般偏向某个方向的优化，相对方向的优化就会起伏很大。实验二十七消耗的时钟数不少，但是节省资源；实验三十四的时钟消耗数勉强可以记为 1 个时钟，反之逻辑资源的消耗却很大。

不过再怎样优化，这也是平衡的一种表现，在灵修中平衡也可以看成代价 ... 为了得到什么，就要付出什么。这一点和钢炼金术士的哲学非常相似，不过现实中到底有没有贤者之石可以打破这种平衡呢？呵呵，笔者还在摸索当中 ...

5.14 延迟压力和时序上的微妙关系 1

这个小节估计是本章中最后一个重点了，我们常常在书中看到组合逻辑越大延迟压力就越大，在物理时序上数据会根据延迟压力的时间，然后往右边推向多少。当然这是我们对延迟和物理时序上的普遍认识，今天我们要从更多方面去思考延迟压力和时序上的微妙性。

图 5.14.1 无所事事的组合逻辑。

组合逻辑是一个被动的东西，如图 5.14.1 所示，在没有任何东西的驱动之下，组合逻辑就像傻子一样，站在一边不停问自己“我要干什么？”。为了给组合逻辑工作，我们要设驱动源给它，那么驱动源到底是什么呢？驱动源一般上有两种，一个是组合逻辑级的驱动源，另一个是 RTL 级的驱动源。

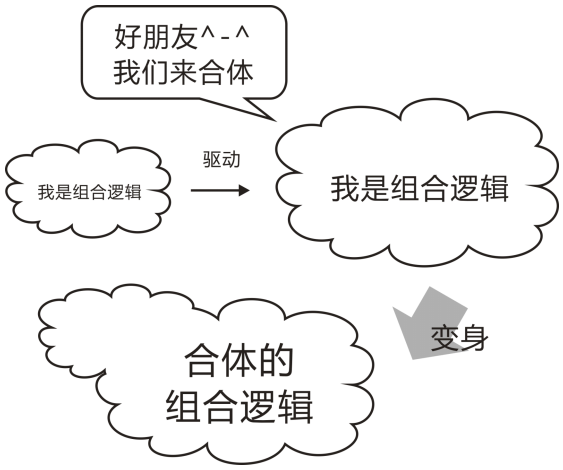


图 5.14.2 合体以后的组合逻辑。

如图 5.14.2 所示，如果驱动组合逻辑的驱动源也是组合逻辑，那么两个组合逻辑就会合体，换句话说，组合逻辑的延迟压力就会累加。因此，组合逻辑驱动组合逻辑，无论是理想还时序还是物理时序都没有任何作用的。所以说，当驱动源唯有 RTL 级时，那么分析延迟压力在时序上的影响才有意义。

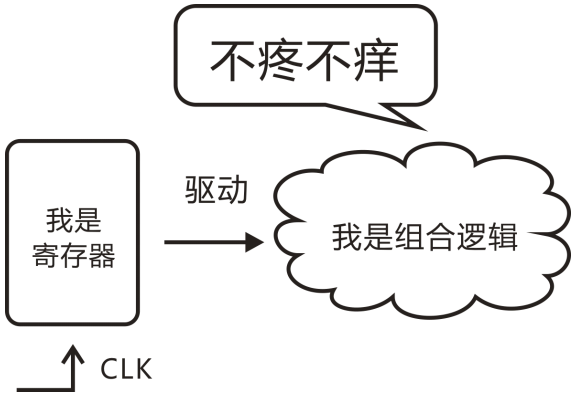


图 5.14.3 被单个 RTL 级驱动源驱动的组合逻辑。

首先我们先简单复习一下什么是 RTL 级设计？RTL 级设计最具象征性就是某个寄存器，或者模块有 CLK 这条尾巴。如图 5.14.3 所示，组合逻辑被 RTL 级的（寄存器）驱动源驱动 ... 不过，组合逻辑好像不疼不痒 ... 所以说，至少前面一个 RTL 级的寄存器，后面又有一个 RTL 级的寄存器，那么事情才会有意义。

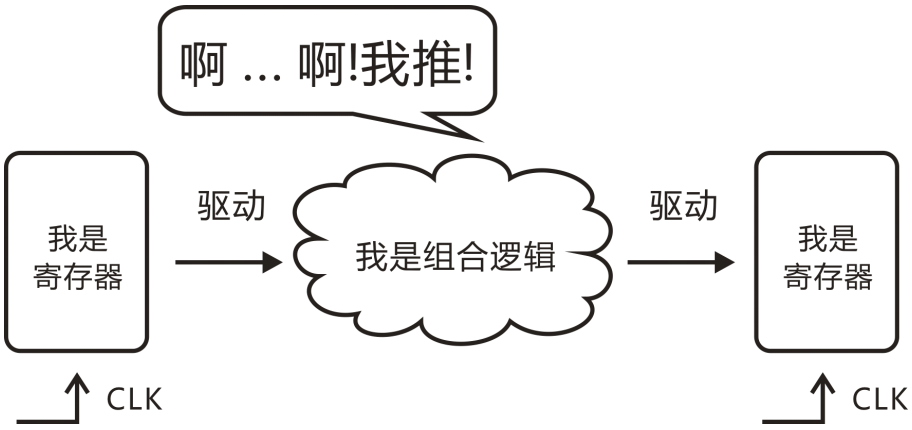


图 5.14.4 组合逻辑前后夹在 RTL 级的寄存器里。

如图 5.14.4 所示，当组合逻辑前后都被 RTL 级的寄存器包围着，那么最基本的时序图就成立了，为什么话这么说呢？

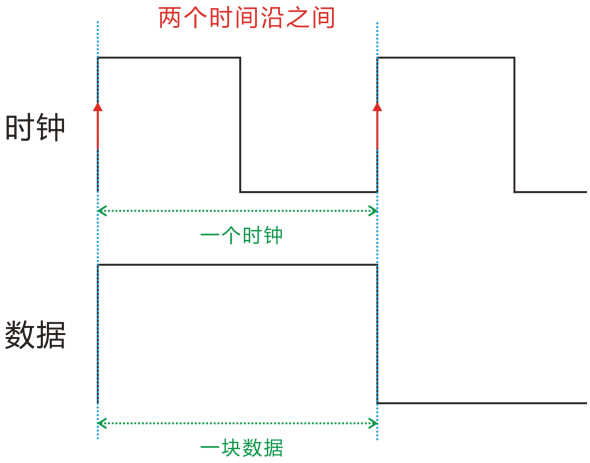


图 5.14.5 一个时钟一块数据最基本的概念。

当有两个时钟沿的存在时，那么两个时间沿的之间“一个时钟，一块数据”的概念就产生了。到目前为止，我们了解到，如果组合逻辑没有被夹在两个 RTL 级的寄存器之间，或者说是夹在两个时钟沿的之间，那么时序图就不会存在，时序分析也没有意义了。只要时序图存在，那么组合逻辑和时序之间的微妙关系就开始产生了。我们先从 TimeQuest （静态时序分析）的角度看看组合逻辑的延迟压力如何影响时序。

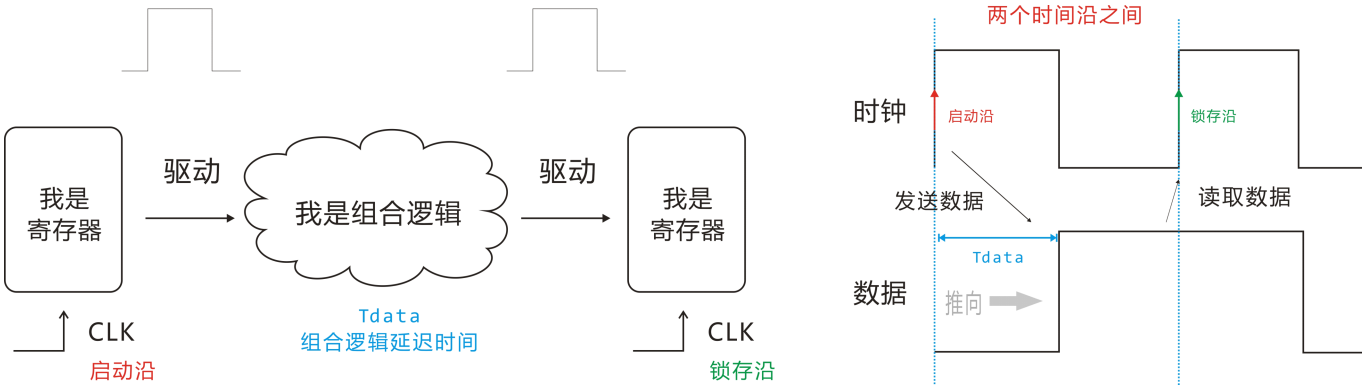


图 5.14.6 静态时序分析的基本思路。

图 5.14.6 就是组成 TimeQuest 静态时序分析的最基本“模型”。当驱动源的寄存器在时钟的上升沿发送数据，那么这个时钟沿就成为启动沿。然后数据会经过组合逻辑的 Tdata 延迟，最终数据会被向右推向诺干时钟。当锁存方的寄存在在时钟的上升沿锁存数据，那么这个时钟沿就成为锁存沿。TimeQuest 就是用这个基本模型来分析设计到底有没有发生时序违规。

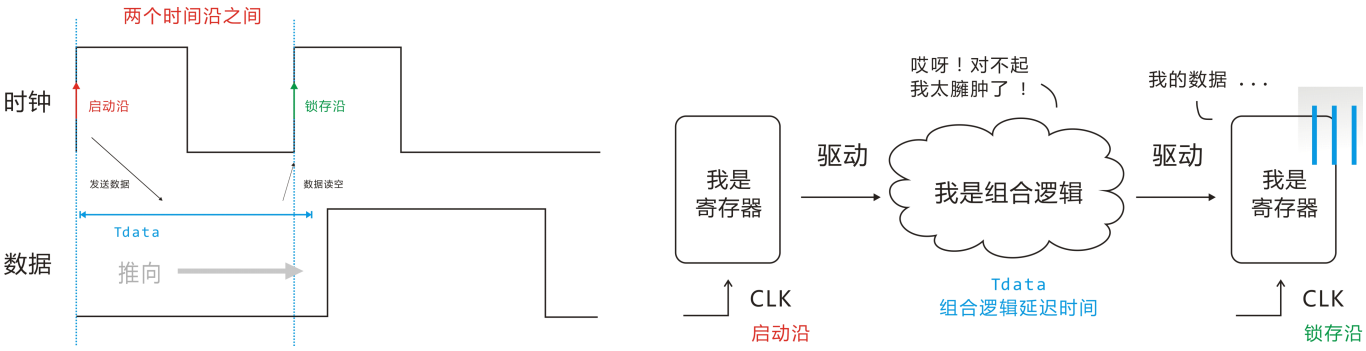


图 5.14.7 时序违规的概念图。

那么什么是时序违规呢？如图 5.14.7 所示，如果组合逻辑太臃肿了，换句话说组合逻辑的延迟压力太大的话，那么数据就会被推向荷兰。如果此时的锁存方，在锁存沿里读空的话，那么时序违规就发生了。上面的一段故事是发生在 TimeQuest 的身上，然而 TimeQuest 的静态时序分析是基于“物理时序”而不是“理想时序”... 实际上，这种典型的理解方式是非常死板而且无趣的。

此外，我们也知道在 Verilog 的理想时序里，信号的位置是不会更动，组合逻辑也不会把数据推向右边 ... 那么我们应该用什么样的方式去理解，组合逻辑的延迟压力和理想时序之间的微妙关系呢？

笔者先问读者："吃饭要不要时间？大便要不要时间？工作要不要时间？"答案是肯定的，什么事情，任何操作肯定需要时间，那么思路就成了吗 ... 慢着，笔者到底在说什么呢？其实，我们可以把组合逻辑看成似"数据的加工工厂"，然而数据的加工过程又称为"加工时间"，至于数据有没有"足够的加工时间"就要看我们的一个时钟的长度够不够宽了。哈哈，在这里读者肯定听迷糊了...

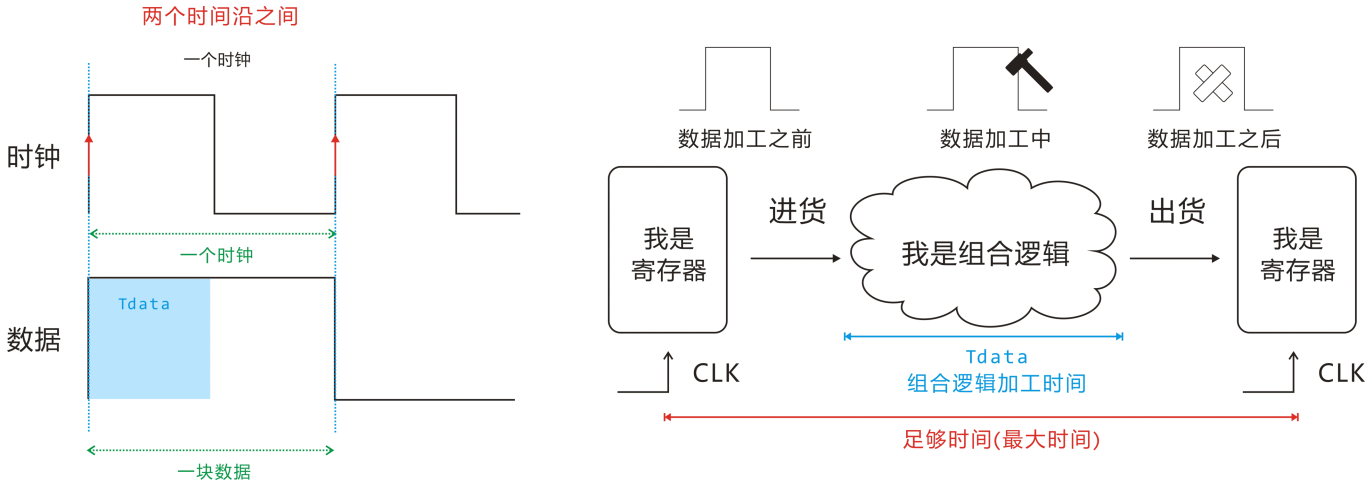


图 5.14.8 数据加工的概念图。

首先，先请读者关上眼睛仔细回忆整合篇上的所有内容，尤其是第二章和第三章。在整合篇上的第二章中，笔者曾经说过一个时钟是建立在两个时钟沿之间，然而"一个时钟"比起"时钟沿"更有具体的样子。有了"一个时钟"以后，自然而然就有"一块数据"。在整合篇上的第三章中，笔者说过，在理想时序中数据不会失去对齐性，此外可以用覆盖的概念取代推向的概念。

如图 5.14.8 所示，当充当驱动源的寄存器向组合逻辑进货的时候，数据的加工就开始了。其中 Tdata 就是加工所需的时间，至于足够时间（最大时间）就是一个时钟所停留的时间。数据的加工时间到底有多久，那么 Tdata 在一块数据中的覆盖率就会多大。只要最大时间减下加工时间以后还有余数，那么时序就是合格的。读者现在是不是彻底服了吧？我们只是用不同的理解方式，不但没有丢失信号之间的对齐性，然而有可以实现理想时序和延迟压力之间的微妙关系。

==========

我们先暂时休息一下，估计有许多朋友开始在抓头了 ...

某个同学问："笔者你不是说过组合逻辑所引起的即时事件不会在理想时序中显示吗？又或者说即时事件不是无视时钟吗？怎么这些东西和上述的内容是不是有点矛盾了？"不！一点都不矛盾，首先我们必须明白 Modelsim 是无法显示即时事件，此外 Verilog 也没有能力控制组合逻辑的延迟压力是多少。这些都是物理时序的东西 ... Verilog 无法插入一只脚去搅局。虽然 Verilog 无法搅局，但是这并不代表 Verilog 就没有关系。

身为 Verilog HDL 语言的使用者，我们或多或少都有义务去理解这些内容。有充分理解 Verilog HDL 语言一滴一事后，我们才能尽情的发挥 Verilog 这把楚龙刀。笔者劝读者耐心点，往后一步一步踏入 Verilog 的世界里，读者就会发现任何矛盾的背后却有微妙的连接和共同性。

就像生物学家说“植物和动物是不一样的”还是说”植物和动物是一样的”，无论怎么说都有矛盾。动物会移动，植物不会移动所以“植物和动物是不一样”是成立的。动物和植物都使用相同的化学能量，所以“植物和动物是一样的”也可以成立。到底结局是矛盾还是不矛盾，问题的背后只有了解和不了解而已。

5.15 延迟压力和时序上的微妙关系 2

在 5.13 小节里笔者曾经所过组合的面积布局会给组合逻辑的压力带来直接的影响不是吗？那么到底是什么样的影响，我们在这个小节里会仔细的讨论。然后我们又会讨论怎样用 Verilog 而不是 TimeQuest 等静态时序分析的工具去解决组合逻辑延迟压力过大的问题等。在此之前笔者希望读者做好十分的心里准备，因为接下来的内容会有非常大的冲击力。

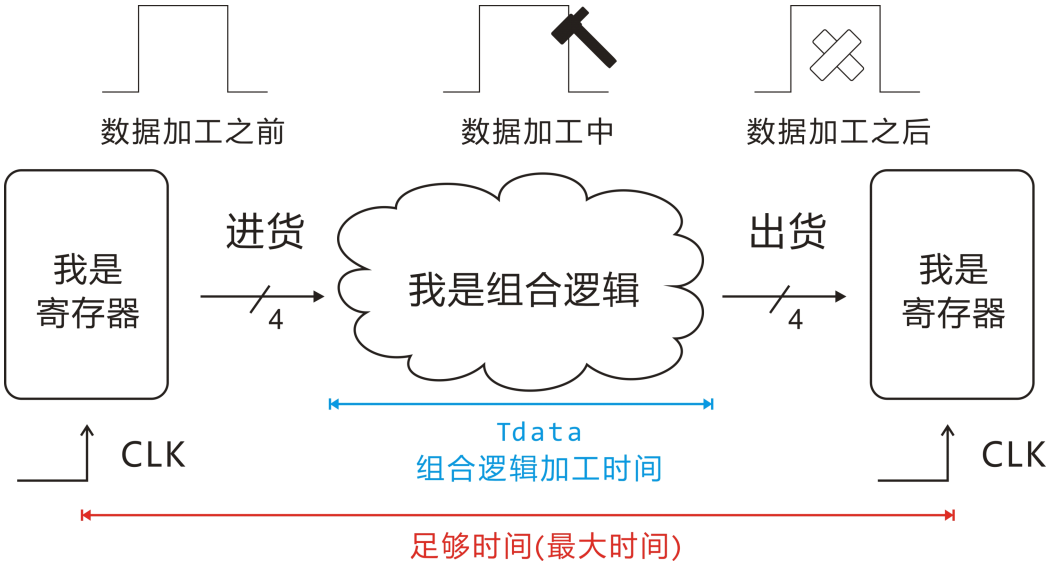


图 5.15.1 N 个数据加工的概念图。

在前一个小节，我们都假设驱动方的寄存器的位宽都是 1，如果我们把位宽从 1 变成 4 的话，那么整个数据的加工过程到底会发生什么事情呢？如图 5.15.1 所示，在两个 RTL 级的寄存器之间，进货和出货的位宽都是 4 位。假如我们把左边的寄存器当着 4 位的典型乘法器（阵列式乘法器），那么在时序上会出现什么有趣的现象呢？

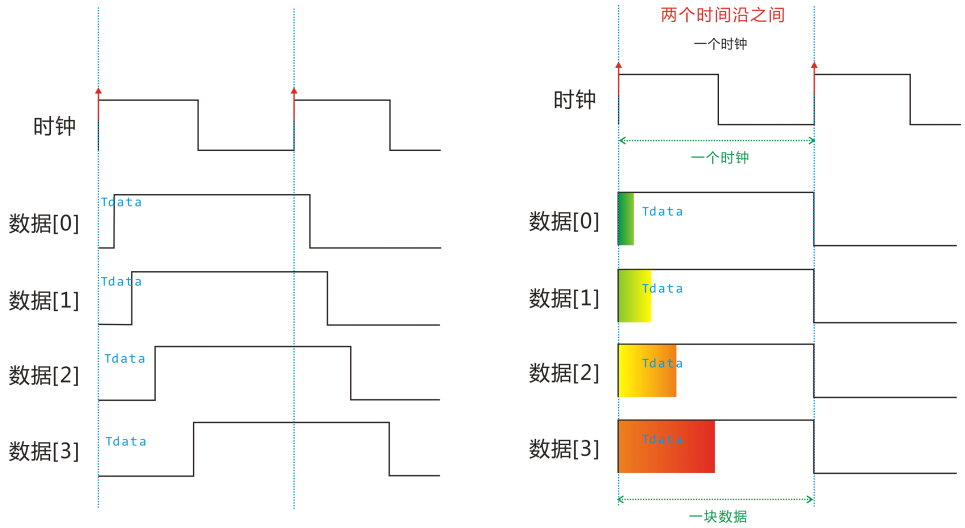


图 5.15.2 典型乘法器在时序上的表现。

图 5.15.2 是 4 位典型乘法器在时序上的变现，左边是物理时序是用来参考而已，看看就好。反之右图 4 位典型乘法器因延迟压力的不均匀，然后在理想时序上引起的现象。从图中我们可以看到，在右边的理想时序中，不同位之间的信号都有某个颜色特征的延迟压力。颜色绿延迟压力越小，颜色越红延迟压力越大，然而根据阵列式乘法器的本质，[0]位的延迟压力最小，换之[3]位的延迟压力最大。4 个不同位的信号最后在理想时序中，产生不均匀和平衡的延迟压力。

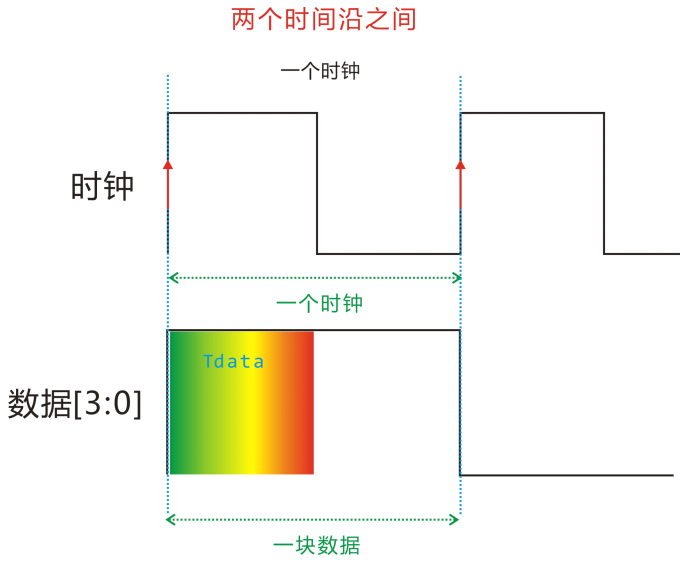


图 5.15.3 典型乘法器 4 位信号叠加。

如果把 4 位信号叠加，那么延迟压力会呈现漂亮的彩虹颜色 ... 呵呵！事实上，越是漂亮（不均衡）的延迟压力，对 Verilog 越是不健康，换之对 TimeQuest 来讲延迟压力俊不均衡结果都是一样。这话要怎么说呢？好了，现在我们把 4 位的典型乘法器，换成 4 位的 booth 乘法器 ... 然后再看看它怎样影响理想时序图。

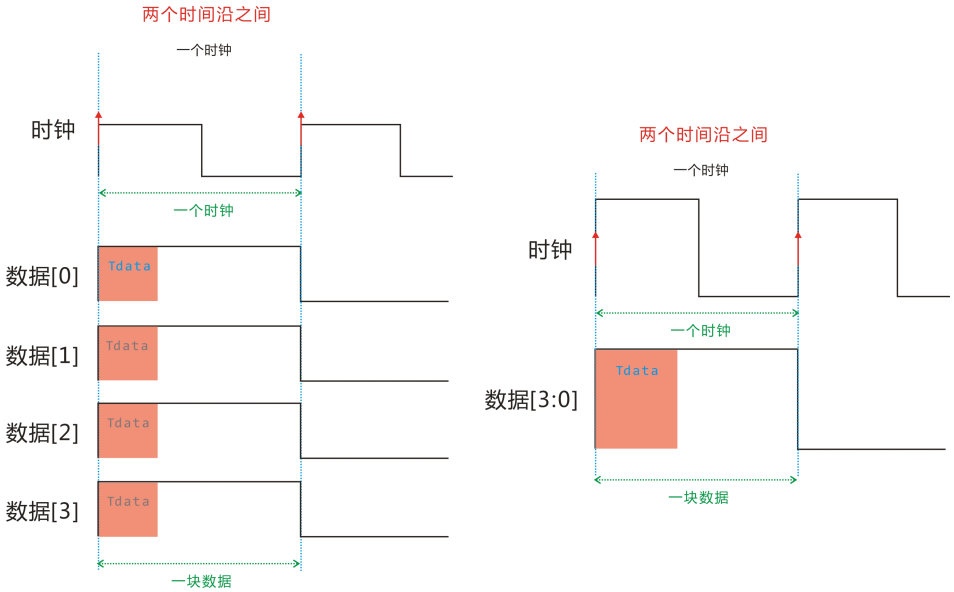
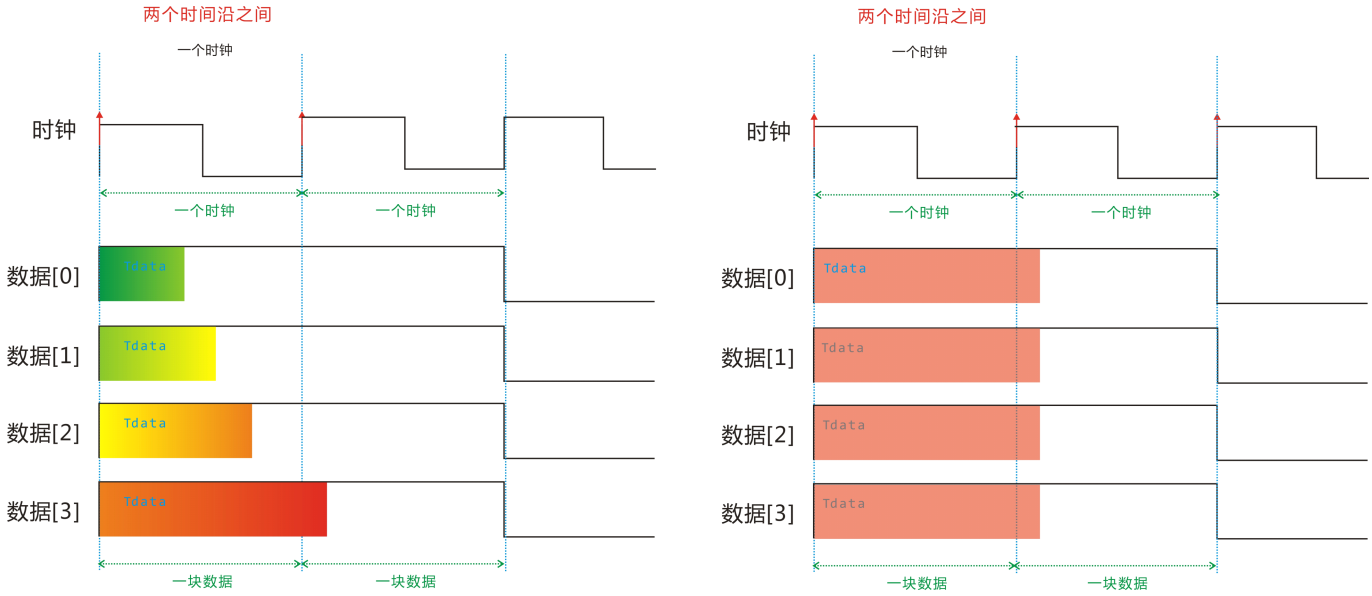


图 5.15.4 Booth 乘法器在理想时序上的表现。

图 5.15.4 的左边是 4 位信号展开的结果，然而右边的图式 4 位信号叠加的结果。读者自己看到了吧，无论是将 4 位结果展开还是叠加，它的表现依然美丽。如果读者问“这番美丽”对延迟压力会带来怎样的效果？那么，读者就问到重点了。我们假设一个情况“如果阵列式乘法器或者 Booth 乘法器的延迟压力大过一个时钟的最大时间的话 ... ”，我们来看看拥有均衡的延迟压力和不均衡的延迟压力之间，是如何被解决的。



阵列式乘法器
(不均匀的延迟压力)

Booth乘法器
(均匀的延迟压力)

图 5.15.5 当延迟压力大过的时候・解决方法 1。

当两个乘法器由于组合逻辑引起的压力过大，导致在一个时钟内无法完成数据加工，换句话说就是加工时间大过最大时间。在这里我们不谈 TimeQuest 如何结局延迟压力过大的问题，反之我们尝试用 Verilog 来结局问题。如果一个时钟不足应付数据加工的操作，从逻辑上我们可以再给足另一个时钟来应付余下不足的时间，结果如图 5.15.5 所示。这个问题读者是不是觉得很眼熟？对！类似的问题曾经发生在 SDRAM 的实验里。

那么，我们如何在 Verilog HDL 语言上下手呢？

```
wire [3:0]Product;
booth_multiplier_module U1( rA,rB,Product);

reg [3:0]rA,rB,rData;

always @ ( posedge CLK ... )
...
case( i )
        0:
        begin rA <= 4'd3; rB <= 4'd5; i <= i + 1'b1; end

        1:
```

```
13.         begin rData <= Product; i <= i + 1'b1; end;
14.
15.         ....
```

第 1 行我们声明了 Booth 乘法器的输出连线-Product；第 2 行是 Booth 乘法器的实例化；第 4 行是相关的寄存器声明，其中 rA 驱动 Booth 乘法器的 A 方，rB 则驱动 B 方，rData 是用来寄存 Product 的结果。在步骤 0 里我们给 Booth 乘法器输入数据 3 * 5，然而我们给了一个时钟，然后在步骤 1 里取得 Product 的结果，最后赋值与 rData 中。

上面一段代码是给予 1 个时钟的最大时间，让 Booth 乘法器来加工 A 和 B 的数据。如果一个时钟的时间足够与数据的加工时间，那么结果当然 Okay。反之，如果一个时钟的时间不足与应付数据的加工时间的话，那么我们又该怎样修改？

```
1.      wire [3:0]Product;
2.      booth_multiplier_module U1( rA,rB,Product);
3.
4.      reg [3:0]rA,rB,rData;
5.
6.      always @ ( posedge CLK ... )
7.      ...
8.      case( i )
9.          0:
10.         begin rA <= 4'd3; rB <= 4'd5; i <= i + 1'b1; end
11.
12.         1：
13.         i <= i + 1'b1;
14.
15.         2:
16.         begin rData <= Product; i <= i + 1'b1; end;
17.
18.         ....
```

我们在步骤 1 里插入一个没有的空步骤，实际上就是给 Booth 乘法器多一个空时间来应付数据的加工。这个方法虽然非常简单和好用，但是它也有不足的地方。从代码的理解上，我们知道 Booth 乘法器一定拥有足够的时间来应付数据的加工，但是傻瓜的编译器就不知道。无论我们怎样编译，编译器一定会反馈许多烦人的警告信息 … 如果我们无法无视编译器的警告信息，那么我们又如何下手来解决问题呢？这时候，拥有建模技巧底子的乘法器就会突出优势 …

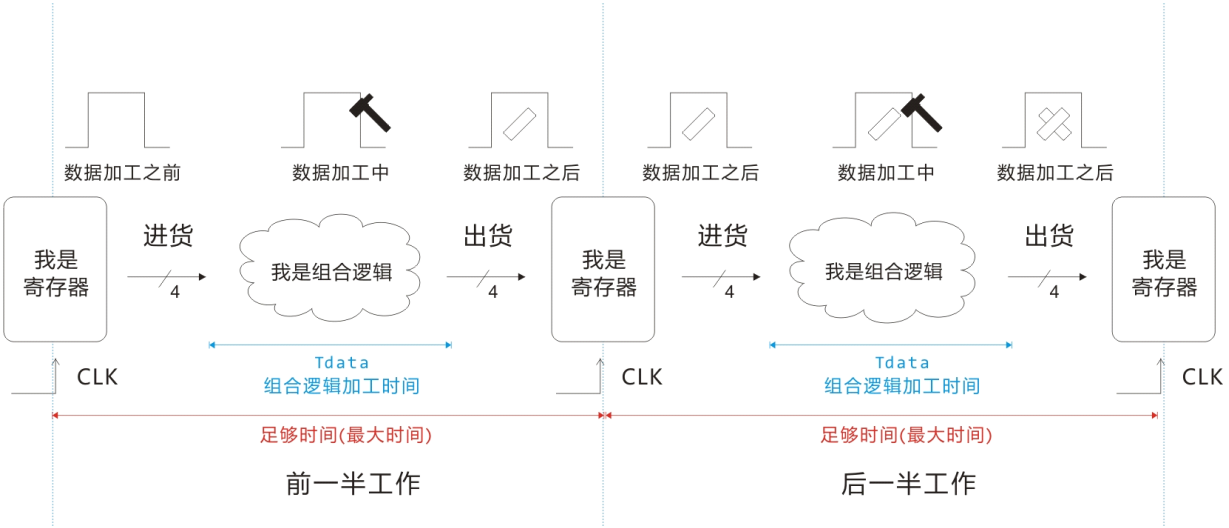


图 5.15.6 将数据的加工划分几个阶段。

如图 5.15.6 所示，我们可以用另一种方法—就是把数据的加工划分几个阶段。图中，我们把数据的加工经过划分为 2 个阶段，亦即前一段，后一段。在此，任何组合逻辑越是面积分布均匀，越是延迟压力平衡，然而拥有多个任务模块的话 ... 基本上是完全胜任这种方法。用残忍的话来说，典型的阵列式乘法器可要被淘汰了，换之 Booth 乘法器等拥有算法性质，里边又有多个任务模块的组合逻辑是压倒性的站上冠军宝座。

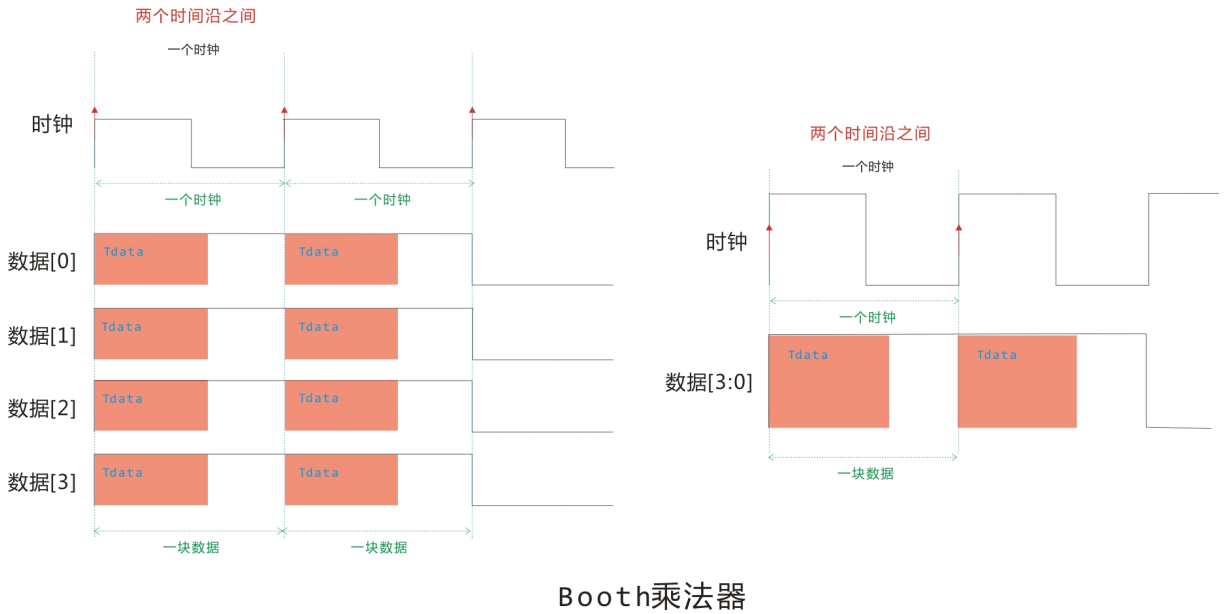


图 5.15.7 将数据加工划分几个阶段在理想时序上的表现。

图 5.15.7 是将组合逻辑划分两半，然后在理想时序上引起的表现。在此，我们可以看到延迟压力的均衡给划分过程带来极大的便利，无论是将每一位信号展开还是重叠。当然，我们只是在逻辑上来思考问题而已，等下会以 Booth 乘法器为例子，将它开刀分尸，咔咔咔！

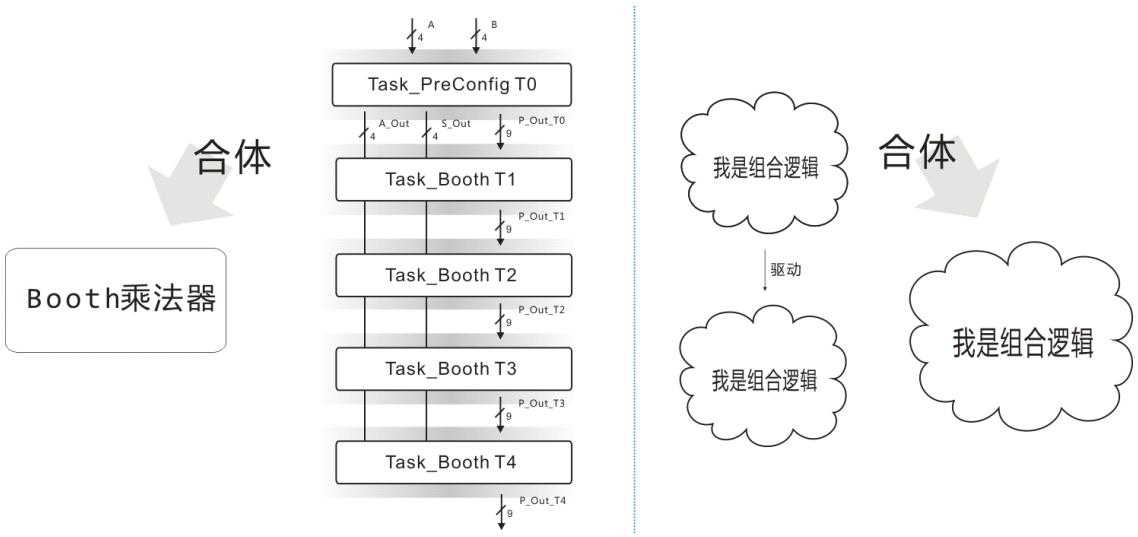


图 5.15.8 组合逻辑合体。

在前面笔者曾经说过，如果组合逻辑的驱动源是组合逻辑，那么两个组合逻辑基本上可以看成一体，同样延迟压力也会叠加，不过前提是要看组合逻辑的面积布局均不均匀。如图 5.15.8 所示，Booth 乘法器就是其中一个例子，其中 T0~T5 五个任务模块都是组合逻辑，如果全部任务模块相互驱动 ... 基本上 Booth 乘法器可以看成是 5 个任务模块的合体。

既然它们可以合体，同样也可以分离 ... 我们可以用 Verilog HDL 语言这样写：

```
module Booth_Multiplier_A
(
   input [3:0]A, B,
   output [3:0] A_Out, S_Out,
   output [8:0]P_Out
  );

wire [3:0]A_Out_T0，S_Out_T0;
wire [8:0]P_Out_T0;
Task_Initial T0( A, B, A_Out_T0, S_Out_T0, P_Out_T0 )

wire [3:0]A_Out_T1，S_Out_T1;
wire [8:0]P_Out_T1;
Task_Booth T1( A_Out_T0, S_Out_T0, A_Out_T1, S_Out_T1, P_Out_T1 );

wire [3:0]A_Out_T2, S_Out_T2;
wire [8:0]P_Out_T2;
Task_Booth T2(A_Out_T0, S_Out_T0, A_Out_T2, S_Out_T2, P_Out_T2 );

assign A_Out = A_Out_T2;
assign B_Out = B_Out_T2;
assign P_Out = P_Out_T2;

```

```
endmodule
```

```
module Booth_Multiplier_B
(
   input [3:0] A_In, S_In,
   input [8:0]P_In,
   output [7:0]Product
  );

wire [3:0]A_Out_T3，S_Out_T3;
wire [8:0]P_Out_T3;
Task_Booth T3( A_In, S_In, A_Out_T3, S_Out_T3, P_Out_T3 );

wire [3:0]A_Out_T4，S_Out_T4;
wire [8:0]P_Out_T4;
Task_Booth T4( A_Out_T3, S_Out_T3, A_Out_T4, S_Out_T4, P_Out_T4 );

assign Product = P_Out_T4[8:1];

endmodule
```

先将 Booth 乘法器划分为两个部分，前半部分包含 T0~T2，后半部分则包含 T3~T4。

```
wire [3:0]A_Out_U1, B_Out_U1;
wire [8:0]P_Out_U1;
Booth_Multiplier_A U1( rA, rB, A_Out_U1, B_Out_U1, P_Out_U1 );

wire [7:0]Product
Booth_Multiplier_B U2( rA_Out, rB_Out, Product );

reg [3:0]rA_Out,rB_Out;
reg [8:0]rP_Out;
reg [3:0] rA,rB;
reg [7:0]rData;

always @ ( posedge CLK ... )
...
case( i )
        0:
        begin rA <= 4'd3; rB <= 4'd5; i <= i + 1'b1; end

        2:
        begin rA_Out <= A_Out_U1; rB_Out <= B_Out_U1; rP_Out = P_Out_U1; i <= i + 1'b1; end

```

```
22.         3:
23.         begin rData <= Product; i <= i + 1'b1; end
24.         ......
25.   ......
```

然后我们把 Booth_Multiplier_A 和 Booth_Multiplier_B 在某个模块里调用。第 1~3 行先把前者实例化，然后在第 4~5 行把后者实例化。在这里，读者需要注意的是，第 1~2 行 Booth_Multiplier_A 的输出信号没有直接驱动第 6 行的 Booth_Multiplier_B。而是在步骤 2 中，用 rA_Out，rB_Out 和 rP_Out 寄存器作为中转站。最后在步骤 3 中将结果赋值与 rData。

==========

“笔者 ... 夭寿了，真的要死了！哇 ... 全部吸收不了，反而要吐了！”哈哈哈，估计同学们都会有这种感觉不是吗？从 5.13~5.15 小节里，笔者所讨论的内容不是要读者全部都吃下去，然而读者只要读着读着然后有个“框子”就行了。因为即时操作的基础就是组合逻辑，然而和组合逻辑有关的内容实在是太多了，目前的笔者也在学习当中。

笔者还是那句话，如果要使用即时操作，多少也要了解一下有关组合逻辑的知识。很多初学者和初学的笔者一样，都觉得组合逻辑如同课本所说的那样，几个逻辑门就成事了。如果组合逻辑在 RTL 级的设计中出现，那么情况就不一样了。里面有很多我们无法了解黑幕在隐藏着，等待傻呼呼的猎物自投罗网。

最后，我们再简单的总结一下吧：

时序分析一般上都是发生在两个 RTL 级的寄存器，然后中间夹着一个组合逻辑。组合逻辑随着功能和建模的增加，延迟压力也会随之成长，其中面积的不均俊不均衡会直接影响延迟压力的均衡。在物理时序中，延迟压力会将信号向右推向，换之在理想时序中，延迟压力会覆盖某快数据。组合逻辑在理想时序中有如数据的加工工厂，加工过程就是数据所需要的加工时间。

如果一个时钟不足应付加工时间的话，那么理论上可以多给与一个空时间，这一点和 SDRAM 的实验非常相似。不过笨蛋的编译器可能会反馈相关的时序违规信息，即使时序是安全的。为此，我们可以把一个组合逻辑分尸成几个小块，然后用更多的 RTL 级寄存器作为中转站，因而将延迟压力分化在不同的阶段。

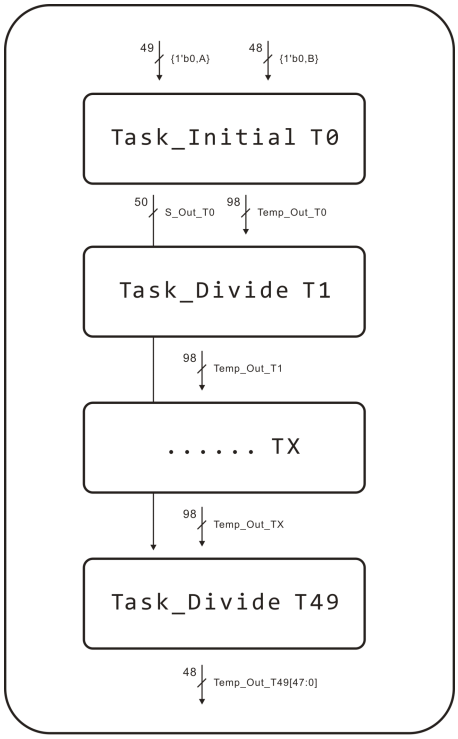
最后笔者还是重复那句老话，组合逻辑虽然好用，但它是双刃剑用不好倒回受伤，所以理 解 5.13~5.15 小节是必须的功课。从 5.13~5.15 小节里我们可以认识到组合逻辑种种不为人知的一面，组合逻辑无论好坏，这些都是它的性质。最后的最后，我们来聊一些不相关的话题：

既然我们知道，一个时钟的时间就是我们所谓的时钟周期，亦即频率越高时钟周期就越小，因此一个时钟可以给予组合逻辑加工的时间也会越来越挑剔。如果读者是追求高频率的家伙，那么就要慎用组合逻辑或者即时操作，不然的话，读者会看到非常可怖的警告信息。至于组合逻辑的延迟压力是 Verilog 都无法参与的东西，我们唯有在组合逻辑的建模上，尽可以将组合逻辑划分为更多任务模块。有需要的话，组合逻辑可以轻松的划分几个分段来工作。

5.16 与即时操作拼耐性

不是所有即时操作都那么简短，有时候我们会遇见非常沉长的任务模块，举个例子：在实验二十九里我们不是建立过循环操作的除法器吗？这个除法器为了处理 49 位位宽的数据，因此重新循环 49 次。在此，如果这个除法器要转换为即时操作的话 ... 呵呵，我们就要和即时操作的建模拼耐性了。

实验三十五：即时式-Streamlined 除法器



immediate_divide_module.v

图 5.16.1 即时式-Streamlined 除法器的建模图。

图 5.16.1 就是该模块的建模图，其中只有两种任务模块，亦即 Task_Initial 和 Task_Divide。其中非常吓人的是，Task_Divide 可是有 49 个那么多 ... 汗，这不是要杀人吗？读者不要立即就泄气了，学习 Verilog HDL 语言最重要的就是耐性，难得有个实验可以磨练自己的耐性，怎么还愁眉苦脸呢？至于 Streamlined 除法器的功能，笔者也懒得介绍了，我们还是直接来看代码吧：

immediate_divide_module.v

```
module Task_Initial
(
        input [48:0]A_In,B_In,
        output [49:0]S_Out,
        output [97:0]Temp_Out
);

```

```
171.      assign S_Out = B_In[48] ? { 1'b1, B_In } : { 1'b1 , ~B_In + 1'b1 };
172.      assign Temp_Out = A_In[48] ? { 49'd0 , ~A_In + 1'b1 } : { 49'd0 , A_In };
173.
174.  endmodule
```

第 166~168 行是该模块的输入输出声明。此外，第 171 行是 S_Out 的输出和驱动，至于第 172 行是 Temp_Out 的输出和驱动。

```
176.  module Task_Divide
177.  (
178.        input [49:0]S_In,
179.        input [97:0]Temp_In,
180.        output [97:0]Temp_Out
181.  );
182.
183.        wire [97:0]Diff = Temp_In + { S_In, 48'd0 };
184.
185.        assign Temp_Out = Diff[97] ? { Temp_In[96:0] , 1'b0 } : { Diff[96:0] , 1'b1 };
186.
187.  endmodule
```

第 178~180 行就是该模块的输入输出声明。第 183 行是先取得 Diff 的结果，然而第 185 行更具 Diff 的符号位，再来决定哪个输出和驱动 Temp_Out。

```
1.    module immediate_divide_module
2.    (
3.          input [47:0]A,B,
4.          output [47:0]Quotient
5.    );
6.
7.          wire [49:0]S_Out_T0;
8.          wire [97:0]Temp_Out_T0;
9.          Task_Initial T0( { 1'b0,A }, { 1'b0,B }, S_Out_T0, Temp_Out_T0 );
10.
11.         wire [97:0]Temp_Out_T1;
12.         Task_Divide T1( S_Out_T0, Temp_Out_T0, Temp_Out_T1 );
13.
14.         wire [97:0]Temp_Out_T2;
15.         Task_Divide T2( S_Out_T0, Temp_Out_T1, Temp_Out_T2 );
16.
17.         wire [97:0]Temp_Out_T3;
18.         Task_Divide T3( S_Out_T0, Temp_Out_T2, Temp_Out_T3 );
19.
20.         wire [97:0]Temp_Out_T4;
```

```
21.     Task_Divide T4( S_Out_T0, Temp_Out_T3, Temp_Out_T4 );
22.
23.     wire [97:0]Temp_Out_T5;
24.     Task_Divide T5( S_Out_T0, Temp_Out_T4, Temp_Out_T5 );
25.
26.     wire [97:0]Temp_Out_T6;
27.     Task_Divide T6( S_Out_T0, Temp_Out_T5, Temp_Out_T6 );
28.
29.     wire [97:0]Temp_Out_T7;
30.     Task_Divide T7( S_Out_T0, Temp_Out_T6, Temp_Out_T7 );
31.
32.     wire [97:0]Temp_Out_T8;
33.     Task_Divide T8( S_Out_T0, Temp_Out_T7, Temp_Out_T8 );
34.
35.     wire [97:0]Temp_Out_T9;
36.     Task_Divide T9( S_Out_T0, Temp_Out_T8, Temp_Out_T9 );
37.
38.     wire [97:0]Temp_Out_T10;
39.     Task_Divide T10( S_Out_T0, Temp_Out_T9, Temp_Out_T10 );
40.
41.     wire [97:0]Temp_Out_T11;
42.     Task_Divide T11( S_Out_T0, Temp_Out_T10, Temp_Out_T11 );
43.
44.     wire [97:0]Temp_Out_T12;
45.     Task_Divide T12( S_Out_T0, Temp_Out_T11, Temp_Out_T12 );
46.
47.     wire [97:0]Temp_Out_T13;
48.     Task_Divide T13( S_Out_T0, Temp_Out_T12, Temp_Out_T13 );
49.
50.     wire [97:0]Temp_Out_T14;
51.     Task_Divide T14( S_Out_T0, Temp_Out_T13, Temp_Out_T14 );
52.
53.     wire [97:0]Temp_Out_T15;
54.     Task_Divide T15( S_Out_T0, Temp_Out_T14, Temp_Out_T15 );
55.
56.     wire [97:0]Temp_Out_T16;
57.     Task_Divide T16( S_Out_T0, Temp_Out_T15, Temp_Out_T16 );
58.
59.     wire [97:0]Temp_Out_T17;
60.     Task_Divide T17( S_Out_T0, Temp_Out_T16, Temp_Out_T17 );
61.
62.     wire [97:0]Temp_Out_T18;
63.     Task_Divide T18( S_Out_T0, Temp_Out_T17, Temp_Out_T18 );
64.
65.     wire [97:0]Temp_Out_T19;
```

```
66.     Task_Divide T19( S_Out_T0, Temp_Out_T18, Temp_Out_T19 );
67.
68.     wire [97:0]Temp_Out_T20;
69.     Task_Divide T20( S_Out_T0, Temp_Out_T19, Temp_Out_T20 );
70.
71.     wire [97:0]Temp_Out_T21;
72.     Task_Divide T21( S_Out_T0, Temp_Out_T20, Temp_Out_T21 );
73.
74.     wire [97:0]Temp_Out_T22;
75.     Task_Divide T22( S_Out_T0, Temp_Out_T21, Temp_Out_T22 );
76.
77.     wire [97:0]Temp_Out_T23;
78.     Task_Divide T23( S_Out_T0, Temp_Out_T22, Temp_Out_T23 );
79.
80.     wire [97:0]Temp_Out_T24;
81.     Task_Divide T24( S_Out_T0, Temp_Out_T23, Temp_Out_T24 );
82.
83.     wire [97:0]Temp_Out_T25;
84.     Task_Divide T25( S_Out_T0, Temp_Out_T24, Temp_Out_T25 );
85.
86.     wire [97:0]Temp_Out_T26;
87.     Task_Divide T26( S_Out_T0, Temp_Out_T25, Temp_Out_T26 );
88.
89.     wire [97:0]Temp_Out_T27;
90.     Task_Divide T27( S_Out_T0, Temp_Out_T26, Temp_Out_T27 );
91.
92.     wire [97:0]Temp_Out_T28;
93.     Task_Divide T28( S_Out_T0, Temp_Out_T27, Temp_Out_T28 );
94.
95.     wire [97:0]Temp_Out_T29;
96.     Task_Divide T29( S_Out_T0, Temp_Out_T28, Temp_Out_T29 );
97.
98.     wire [97:0]Temp_Out_T30;
99.     Task_Divide T30( S_Out_T0, Temp_Out_T29, Temp_Out_T30 );
100.
101.    wire [97:0]Temp_Out_T31;
102.    Task_Divide T31( S_Out_T0, Temp_Out_T30, Temp_Out_T31 );
103.
104.    wire [97:0]Temp_Out_T32;
105.    Task_Divide T32( S_Out_T0, Temp_Out_T31, Temp_Out_T32 );
106.
107.    wire [97:0]Temp_Out_T33;
108.    Task_Divide T33( S_Out_T0, Temp_Out_T32, Temp_Out_T33 );
109.
110.    wire [97:0]Temp_Out_T34;
```

```
111.	Task_Divide T34( S_Out_T0, Temp_Out_T33, Temp_Out_T34 );
112.
113.	wire [97:0]Temp_Out_T35;
114.	Task_Divide T35( S_Out_T0, Temp_Out_T34, Temp_Out_T35 );
115.
116.	wire [97:0]Temp_Out_T36;
117.	Task_Divide T36( S_Out_T0, Temp_Out_T35, Temp_Out_T36 );
118.
119.	wire [97:0]Temp_Out_T37;
120.	Task_Divide T37( S_Out_T0, Temp_Out_T36, Temp_Out_T37 );
121.
122.	wire [97:0]Temp_Out_T38;
123.	Task_Divide T38( S_Out_T0, Temp_Out_T37, Temp_Out_T38 );
124.
125.	wire [97:0]Temp_Out_T39;
126.	Task_Divide T39( S_Out_T0, Temp_Out_T38, Temp_Out_T39 );
127.
128.	wire [97:0]Temp_Out_T40;
129.	Task_Divide T40( S_Out_T0, Temp_Out_T39, Temp_Out_T40 );
130.
131.	wire [97:0]Temp_Out_T41;
132.	Task_Divide T41( S_Out_T0, Temp_Out_T40, Temp_Out_T41 );
133.
134.	wire [97:0]Temp_Out_T42;
135.	Task_Divide T42( S_Out_T0, Temp_Out_T41, Temp_Out_T42 );
136.
137.	wire [97:0]Temp_Out_T43;
138.	Task_Divide T43( S_Out_T0, Temp_Out_T42, Temp_Out_T43 );
139.
140.	wire [97:0]Temp_Out_T44;
141.	Task_Divide T44( S_Out_T0, Temp_Out_T43, Temp_Out_T44 );
142.
143.	wire [97:0]Temp_Out_T45;
144.	Task_Divide T45( S_Out_T0, Temp_Out_T44, Temp_Out_T45 );
145.
146.	wire [97:0]Temp_Out_T46;
147.	Task_Divide T46( S_Out_T0, Temp_Out_T45, Temp_Out_T46 );
148.
149.	wire [97:0]Temp_Out_T47;
150.	Task_Divide T47( S_Out_T0, Temp_Out_T46, Temp_Out_T47 );
151.
152.	wire [97:0]Temp_Out_T48;
153.	Task_Divide T48( S_Out_T0, Temp_Out_T47, Temp_Out_T48 );
154.
155.	wire [97:0]Temp_Out_T49;
```

```
156.        Task_Divide T49( S_Out_T0, Temp_Out_T48, Temp_Out_T49 );
157.
158.        /*************************/
159.
160.        assign Quotient = Temp_Out_T49[47:0];
161.
162.  endmodule
```

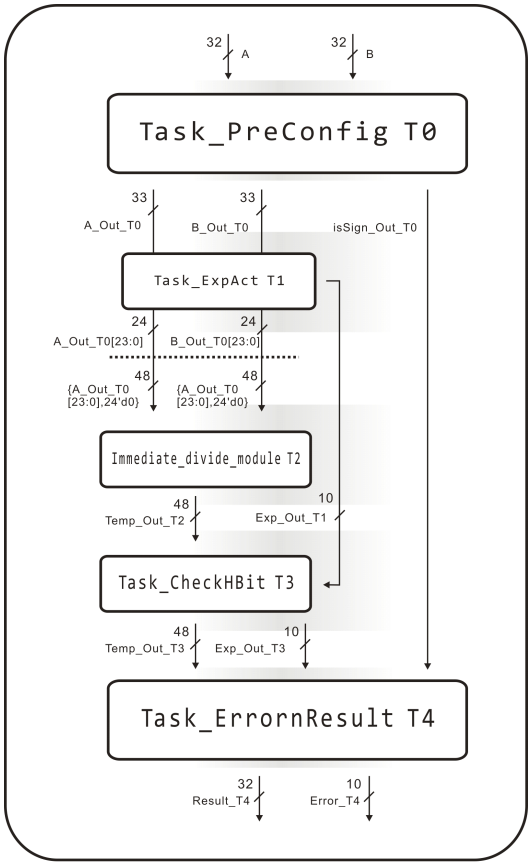
第 1~162 行看似非常简单，基本上就是重复性的工作而已 ... 有时候笔者也真的喜欢循环操作，因为只要控制相关的时钟和循环次数即可，代价再大笔者也愿意，最起码笔者用不着像傻子一样，重复实例化 Task_Divide 从 T1~T49 .... 啊拉！别看它这样，笔者足足耗了 30 分钟左右来编辑这些代码。呜呜 ... 苦呀苦，再喊啊拉！

第 1~162 行的建模和连线和图 5.16.1 差不多，读者自己看着办吧。此外，笔者也不仿真了，都是一些重复性的工作。如果读者有兴趣就自己打开仿真看看吧，文件什么的都已经准备好了。

==========

实验三十六：即时式-浮点数除法模块（整合实验三十五）

既然我们在实验三十五建立了即时式的 streamlined 除法器，因而在实验三十六里，我们不再使用 Verilog HDL 语言自带的 / 操作符了。



immediate_float_divide_module.v

图 5.16.2 即时式-浮点数除法模块的建模图。

图 5.16.2 就是实验三十六要建立的模块。基本上，无论是即时式或者循环式的浮点数除法模块差别的只是操作模式上而已，具体的内容几乎是换汤不换药。我们还是直接来看代码吧：

immediate_float_divide_module.v

```
32.  module Task_PreConfig
33.  (
34.        input [31:0]A_In,B_In,
35.
36.        output isSign_Out,
37.        output [32:0]A_Out,B_Out
38.  );
39.
40.      assign isSign_Out = ( A_In[31] ^ B_In[31] );
41.      assign A_Out = { A_In[31], A_In[30:23], 1'b1, A_In[22:0] };
42.      assign B_Out = { B_In[31], B_In[30:23], 1'b1, B_In[22:0] };
43.
44.  endmodule
```

Task_PreConfig 是对应操作数预处理的步骤以外，还取得符号位的结果。第 34~37 行是该模块的输出输出声明；第 40 行是取得符号位的运算结果；第 41 行是取得 A_Out，第 42 行取得 B_Out 亦即预处理后的操作数 A 和 B。

```
46.  module Task_ExpAct
47.  (
48.      input [32:0]A_In,B_In,
49.      output [9:0]Exp_Out
50.  );
51.
52.      wire [9:0]BDiff = B_In[31:24] - 8'd127;
53.
54.      assign Exp_Out = A_In[31:24] - BDiff;
55.
56.  endmodule
```

Task_ExpAct 对应价码运算的步骤，第 48~49 行则是该模块的输入输出声明。第 52 行取得 B 的价码差，然后在 54 行将 B 的价码差与 A 的价码运算，并且输出和驱动。至于下一个任务模块则是即时式-Streamlined 除法器，我们先略过。

```
58.  module Task_CheckHBit
59.  (
60.      input [9:0]Exp_In,
61.      input [47:0]Temp_In,
62.
63.      output [9:0]Exp_Out,
64.      output [47:0]Temp_Out
65.  );
66.
67.      reg [47:0]Temp;
68.      reg [9:0]Exp;
69.
70.      always @ ( * )
71.          if( Temp_In[25] == 1'b1 ) begin Temp = Temp_In << 5'd22; Exp = Exp_In; end
72.          else if( Temp_In[24] == 1'b1 ) begin Temp = Temp_In << 5'd23; Exp = Exp_In; end
73.          else if( Temp_In[23] == 1'b1 ) begin Temp = Temp_In << 5'd24; Exp <= Exp_In - 1'b1; end
74.          else begin Temp = 48'dx; Exp = 10'dx; end
75.
76.      assign Exp_Out = Exp;
77.      assign Temp_Out = Temp;
78.
79.  endmodule
```

Task_CheckHBit 对应结果调整的步骤，第 60~64 行着则是该任务模块的输入输出声明。第 67~68

行则声明了 Temp 和 Exp ... 由于浮点数除法比较特别，笔者就直接略过笨蛋的写法，直接在输出选择器中（第 71~73 行）根据 Temp_In 的结果来赋值 Temp 和 Exp；最后在第 76~77 则驱动相关的输出。

```
81.  module Task_ErrornResult
82.  (
83.      input isSign_In,
84.      input [9:0]Exp_In,
85.      input [47:0]Temp_In,
86.
87.      output [31:0]Result,
88.      output [2:0]Error
89.  );
90.
91.      wire isOver = ( Exp_In[9:8] == 2'b01 );
92.      wire isUnder = ( Exp_In[9:8] == 2'b11 );
93.      wire isZero = ( Temp_In[47:24] == 24'd0 );
94.
95.      assign Result = { isSign_In, Exp_In[7:0], Temp_In[46:24] };
96.      assign Error = { isOver, isUnder, isZero };
97.
98.  endmodule
```

Task_ErrornResult 对应输出和格式化的步骤。第 83~88 行是相关的输出和输入声明；第 91~93 行则取得 isOver，isUnder 和 isZero 等错误信息的标志。最后在 95 行驱动 Result ，在 96 行驱动 Error。

```
1.   module immediate_float_divide_module
2.   (
3.       input [31:0]A,B,
4.
5.       output [31:0]Result,
6.       output [2:0]Error
7.   );
8.
9.       wire isSign_Out_T0;
10.      wire [32:0]A_Out_T0,B_Out_T0;
11.      Task_PreConfig T0( A, B, isSign_Out_T0, A_Out_T0, B_Out_T0 );
12.
13.      wire [9:0]Exp_Out_T1;
14.      Task_ExpAct T1( A_Out_T0, B_Out_T0, Exp_Out_T1 );
15.
16.      wire [47:0]Temp_Out_T2;
17.      immediate_divide_module T2( {A_Out_T0[23:0], 24'd0}, {24'd0,B_Out_T0[23:0]}, Temp_Out_T2 );
```

```
18.
19.         wire [9:0]Exp_Out_T3;
20.         wire [47:0]Temp_Out_T3;
21.         Task_CheckHBit T3( Exp_Out_T1, Temp_Out_T2, Exp_Out_T3, Temp_Out_T3 );
22.
23.         wire [31:0]Result_T4;
24.         wire [2:0]Error_T4;
25.         Task_ErrornResult T4( isSign_Out_T0, Exp_Out_T3, Temp_Out_T3, Result_T4, Error_T4 );
26.
27.         assign Result = Result_T4;
28.         assign Error = Error_T4;
29.
30.   endmodule
```

第 1~30 行是根据图 5.16.2 来设计的，其他的笔者就不说了，至于第 16~17 行则是用了即时式-Streamlined 除法器来替代 Verilog HDL 语言自带的 / 操作符。

immediate_float_divide_module.v

```
1.    `timescale 1 ps/ 1 ps
2.    module immediate_float_divide_module_simulation();
3.
4.          reg CLK;
5.          reg RSTn;
6.
7.          reg [31:0] A;
8.          reg [31:0] B;
9.
10.         wire [2:0] Error;
11.         wire [31:0] Result;
12.
13.         /*************************/
14.
15.         immediate_float_divide_module U1
16.           (
17.                 .A(A),
18.                 .B(B),
19.                 .Result(Result),
20.                .Error( Error )
21.         );
22.
23.       /****************************/
24.
```

```
25.         initial
26.         begin
27.                 RSTn = 0; #10 RSTn = 1;
28.                 CLK = 0; forever #5 CLK = ~CLK;
29.         end
30.
31.        /******************************/
32.
33.        reg [5:0]i;
34.
35.         always @ (   posedge CLK or negedge RSTn )
36.                 if( !RSTn )
37.                     begin
38.                       A <= 32'd0;
39.                       B <= 32'd0;
40.                       i <= 6'd0;
41.                   end
42.             else
43.                   case( i )
44.
45.                           0: // 3.142 / 3.142
46.                           begin A <= 32'b01000000010010010001011010000111; B <= 32'b01000000010010010001011010000111; i <= i + 1'b1; end
47.
48.                           1:
49.                           begin $display(" %b / %b = %b",A,B,Result ); i <= i + 1'b1; end
50.
51.                           2: // 10 / 3
52.                           begin A <= 32'b01000001001000000000000000000000; B <= 32'b01000000010000000000000000000000; i <= i + 1'b1; end
53.
54.                           3:
55.                             begin $display(" %b / %b = %b",A,B,Result ); i <= i + 1'b1; end
56.
57.                           4: // 6.32 / 0.26
58.                           begin A <= 32'b01000000110010100011110101110001; B <= 32'b00111110100001010001111010111000; i <= i + 1'b1; end
59.
60.                           5:
61.                           begin $display(" %b / %b = %b",A,B,Result ); i <= i + 1'b1; end
62.
63.                           6: // 3.48 / 10
64.                           begin A <= 32'b01000000010111101011100001010010; B <= 32'b01000001001000000000000000000000; i <= i + 1'b1; end
65.
66.                           7:
67.                             begin $display(" %b / %b = %b",A,B,Result ); i <= i + 1'b1; end
68.
69.                           8: // 1.442 / 11.515
```

```
                    begin A <= 32'b00111111101110001001001101110101; B <= 32'b01000001001110000011110101110001;    i <= i + 1'b1; end

                    9:
                    begin $display(" %b / %b = %b",A,B,Result ); i <= i + 1'b1; end

                    10: // 1000 / 0.5
                    begin A <= 32'b01000100011110100000000000000000; B <= 32'b0_01111110_00000000000000000000000;    i <= i + 1'b1; end

                    11:
                    begin $display(" %b / %b = %b",A,B,Result ); i <= i + 1'b1; end

                    12: // 1000 / 4.5
                    begin A <= 32'b01000100011110100000000000000000; B <= 32'b01000000100100000000000000000000;    i <= i + 1'b1; end

                    13:
                    begin $display(" %b / %b = %b",A,B,Result ); i <= i + 1'b1; end

                    14: // 176.9218 / 1146.3369
                    begin A <= 32'b01000011001100001110101111111011; B <= 32'b01000100100011110100101011001000;    i <= i + 1'b1; end

                    15:
                    begin $display(" %b / %b = %b",A,B,Result ); i <= i + 1'b1; end

                    16: // 3.142857 / -0.723093
                    begin A <= 32'b01000000010010010010010010010010; B <= 32'b10111111001110010001110010011111;    i <= i + 1'b1; end

                    17:
                    begin $display(" %b / %b = %b",A,B,Result ); i <= i + 1'b1; end

                    18: // 0.0625 / 0.25
                    begin A <= 32'b00111101100000000000000000000000; B <= 32'b00111110100000000000000000000000;    i <= i + 1'b1; end

                    19:
                    begin $display(" %b / %b = %b",A,B,Result ); i <= i + 1'b1; end

                    20:
                    i <= i;

                endcase

endmodule
```

激励文件的编辑风格还是一如既往 ... 具体的内容读者自己看着办吧，都是老朋友了。

仿真结果：

即时操作的仿真图是没有意义的，笔者还是直接把打印结果贴出来比较明智：

```
3.142 / 3.142 = 1
#  01000000010010010001011010000111 / 01000000010010010001011010000111 = 00111111100000000000000000000000

10 / 3 = 3.33333...
#  01000001001000000000000000000000 / 01000000010000000000000000000000 = 01000000010101010101010101010101

6.32 / 0.26 =  24.3076934
#  01000000110010100011110101110001 / 00111110100001010001111010111000 = 01000001110000100111011000101000

3.48 / 10 =  0.3479999
#  01000000010111101011100001010010 / 01000001001000000000000000000000 = 00111110101100100010110100001110

1.442 / 11.515 =  0.1252279
#  00111111101110001001001101110101 / 01000001001110000011110101110001 = 00111110000000000011101111000010

1000 / 0.5 =  2000
#  01000100011110100000000000000000 / 00111111000000000000000000000000 = 01000100111110100000000000000000

1000 / 4.5 =  222.2222137
#  01000100011110100000000000000000 / 01000000100100000000000000000000 = 01000011010111100011100011100011

176.9218 / 1146.3369 =  0.1543366
#  01000011001100001110101111111011 / 01000100100011110100101011001000 = 00111110000111100000101001101101

3.142857 / -0.723093 = -4.3464078
#  01000000010010010010010010010010 / 10111111001110010001110010011111 = 11000000100010110001010111000110

0.0625 / 0.25 =  0.2500000
#  00111101100000000000000000000000 / 00111110100000000000000000000000 = 00111110100000000000000000000000
```

==========

到目前为止，基本上我们已经完所有即时操作的转换实验，即时式-浮点数除法模块事实上是最耗逻辑资源的 ... 没办法嘛~谁叫它特殊了一点，读者猜猜看，一共用了多少个逻辑资源？呵呵，是吓死人的 4790，这就是代价，我们唯有接受而已 ... 嗯！忽然间笔记觉得自己越来越喜欢循环操作了。

5.17 浅谈嵌套 if

在笔记结束之前，我们来谈谈有关嵌套 if 的内容。有读过建模篇的朋友都知道，笔者所称的低级建模是不推荐 if 的嵌套使用。理由很简单，过度的使用嵌套 if 是使代码的解读性大打折扣，不过小规模的嵌套 if 还是接受的。当然这也是站在建模的角度去思考嵌套 if 而已，在前面既然我们学了那么多有关 组合逻辑的基础知识，笔者就不差用组合逻辑的角度，再深入探讨嵌套 if。

假设我们有这样的一段代码：

```
If( A && B && C ) begin .... end;   //写法一

If( A ) begin                       //写法 2
    If( B ) begin
        If( C ) begin
      ...
end
   end
       end
```

上面一段代码有两个不同的 if 写法，第一种是 A，B 与 C 取 AND 运算，亦即其中一个不是逻辑 1 的话，if 条件就不会成立。反之第二种的 if 写法，也有同样的功能。乍看下，这只是不伤大雅的两种不同写法而已，不过比起第二种写法，第一种写法更养眼一些。如果从组合逻辑的角度来看的话，两种写法会产生不同数量的即时层，这话何解呐？

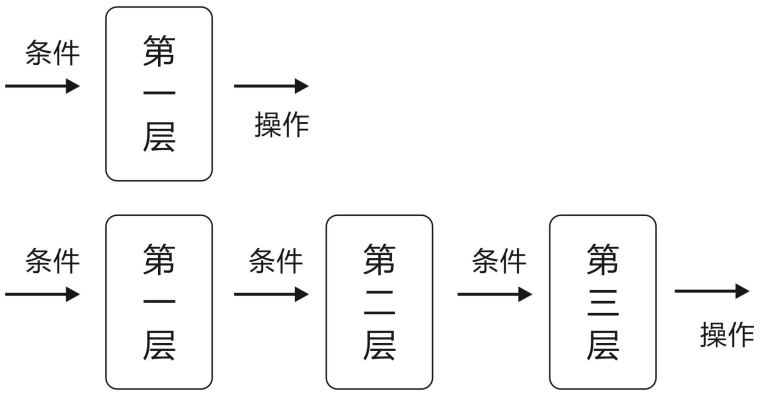


图 5.17.1 从组合逻辑的角度去看嵌套 if。

越是嵌套更多的 if，那么会创建更多的的即时层，如图 5.17.1 所示。图中，第一种 if 写法只创建了一个即时层而已，换之第二种 if 写法创建了 3 个即时层。根据组合逻辑的原理，组合逻辑如果驱动组合逻辑的话，那么延迟压力会叠加。如果延迟压力过大的话，会发生甚么后果，读者自己可是非常清楚。

我们再来换个 if 的写法看看：

```
if( A ) begin ... end                          //写法一
else if( B ) begin ... end
else if( C ) begin ... end
else if( A && B ) begin ... end
else if( A && C ) begin ... end
else if( B&& C ) begin ... end
else if( A && B && C ) begin ... end

if( A ) begin                                  //写法 2
        ....
        if( B ) begin
                ....
                if( C ) begin
                        ....
                        end
                end
        end
else if( B ) begin
        ....
        if( C ) begin
                ....
                end
        end
else if( C ) begin
        ....
        end
```

假设我们要差分更多 A 与 B 和 C 的可能性，第一种写法我们以无嵌套的方式讲六种可能性都列出来，换之第二种写法，以嵌套的方式性来表示六种可能。从风格的角度来看，第二种写法已经给风格的表达能力带来非常大的打击，此外也给编辑，修改，仿真等行为都带来极大的不便。如果从组合逻辑的角度来看：

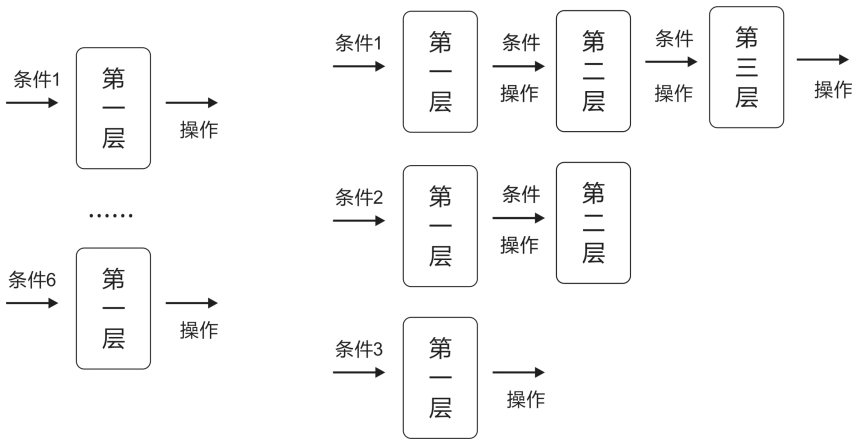


图 5.17.2 多条件的嵌套 if 与组合逻辑。

如图 5.17.2 所示，第一种写法会在第一楼得即时层里会根据 6 个不同条件，然后路由到相关的操作。换之，第二种写法会产生 3 个第一层的即时层，2 个第二层的即时层，和 1 个第三层的即时层，再来根据不同的条件，然后路由到不同的操作里。事实上，多条件的嵌套 if 是如何影响即时层，我们是有目共睹的。由于即时层的层次不同，第二种写法会产生非常不均匀的延迟压力，换之第一种写法会产生非常均匀的延迟压力。延迟压力如果不均匀，读者可知道后果会发生甚么。

再来，笔者举个更挑剔的嵌套 if：

```
if( A) begin
      i <= i + 1'b1;
      else if( i == 12 ) begin
              i <= 4'd0;
                              end
      B <= C;
      if( C == 2 ) begin
              B <= 4'd0;
                      end
      end
```

有谁可以告诉笔者，以上这段代码是甚么鬼东西吗？我们先不论它的风格有多丑陋，不过读者知道吗？Verilog HDL 语言和一些高级语言不同，那些高级语言丑陋的代码只会影响表达能力而已，反之丑陋的代码会直接影响 Verilog HDL 语言硬模块（亦即编译以后的结果）的生成和输出效果？

当读者看到这里，希望读者可以醒悟一下，贪图嵌套 if 的各种后果。嵌套 if 就像使用快餐一样，有便利的效果，不过缺点也和快餐一样，最终会搞垮模块的健康。笔者有时候真的不明白，那些出于专家教授手中的参考书，为甚么就看不到这些简单的道理和细节呐？难道专家教授都被快餐迷魂了嘛？

既然我们知道嵌套 if 的危险，那么我们应该用甚么办法去替代呐？最后的办法就是利用更多的时钟把嵌套 if 的内容分化到更多步骤（状态）里，这也是笔者常常作得事情。举个简单的例子：

```
if( A )
    if( B )
          if( C)

      case( i )

            0:
            if( A ) begin ... i <= i + 1'b1; end

            1:
            if( B ) begin ... i <= i + 1'b1; end
```

```
        2:
        if( C ) begin ... end
```

如果我们遇上 if(A) 有非常大的操作的话，那么我们应该把 A 的操作打包在另一个模块里，如：

```
    case( i )

        0
        if( A ) i <= i + 1'b1;

        1:
        if( Done_Sig ) begin isStart <= 1'b0; i <= i + 1'b1; end
        else isStart <= 1'b1;

        2:
        if( B ) begin .... i <= i + 1'b1; end
        ...
```

说到底，低级建模的样子又出现了 ... 事实上嵌套 if 是从高级语言发展过来的东西，高级语言只有指令和步骤，其他如时钟，建模，组合逻辑等概念均没有。我们不一定要 100%照搬高级语言的东西到 Verilog 的世界里，还记得第一章的大题嘛（不同世界的自然）？我们可以用其他的方式去模范高级语言，或者干脆建立一个全新而且适合 Verilog 的东西。

说实话，我们可以利用低级建模的活用性来模仿似嵌套 if 功能的东西，然而又不似嵌套 if 的东西（把话说了一大圈，我们又扯到建模技巧的重要性了）。低级建模是笔者为 Verilog HDL 语言建立的最低级结构，话说好听一点就是 Verilog HDL 语言的基础结构。不过这个低级建模，笔者自认是用来认识和学习 Verilog HDL 语言而已，它还有其它不足的地方。

最后，笔者还是奉上那句话：

读者要干甚么就干甚么，笔者没有权力去阻止读者。笔者唯一能做的就像老人一样，在旁不停唠叨而已。人都有自由意识嘛，想干甚么都是自己的事情，不过干任何事情都要付出代价，好事好报，坏事坏报，仅此而已。在最后笔者基本没有什么好补充了，笔者跟着神明的指引去游玩了不同操作模式之间的转换之后，神明果然不会说谎，笔者真的理解到 Verilog HDL 语言更多不为人知的一面。

然后在某个夜里，笔者又见到 Verilog 之神了，神明还是一如既往让人猜不出想干什么，笔者开口讲到：“我已经了解到 Verilog HDL 语言多态性的操作模式，估计...”话还没有说完，神明用手指指着下面。

哇！笔者压抑不住震撼 ... 惊呆了！然后大喊：“太强了，太强了 ... ”。
Verilog 之神微笑道：“呵呵，孩子这是你接下来的旅程，是不是非常刺激和有趣？”

笔者忽然间又犯了退缩的老毛病，这条路实在太刺激了，笔者担心自己无法承受。

Verilog 之神又说："孩子，自信一点 ... 不要给自己时间的限制，要像水一样在凹处不停累积自己，直到一天一定会溢出。"梦醒后，笔者的脑袋充溢者琳琅满目的想法，笔者假想了有关下一章笔记的种种可能性和内容，越想越激动，越激动脑经越清醒 ... 就这样，那一夜笔者又失眠了，哈哈。

总结：

这一章（本）笔记又是某个程度的大作，除了思考不同方向的优化以外，最重要是学会使用用平衡的心态去俯视全部过程。如偏向减少操作时钟的优化，一定要付出而外的逻辑资源；如果偏向节省逻辑资源的优化，那么就要用更多的操作时钟；不过一般的设计，笔者都比较钟情于模块的表达能力，那是一种看不到的优化方向。

我们常常在参考书里都会把优化偏向一个方向而已，更多时候都是把“减少操作时钟”当成主要的优化方向，这是一种高级语言借助过来的思想。对于处理器来说，硬件资源是非常宝贵的东西，因此高级语言都会重复使用它们，因此高级语言的优化思考就是用更短的操作时钟达到同样的效果。举个例子：用 10 个时钟重复利用一个资源得到效果，因此可以借助算法，N 个时钟重复利用一个资源得到同样的效果，结果优化了 N 个时钟。不过这些优化，许多情况都是借助更大的存储空间才能达到目的。

对于 Verilog 来说，只要 FPGA 的允许的话，我们可以拥有无限的资源。因此优化的思想也不应该仅限于单个方向而已。因此，随着这种想法的展开，我们会遇见不同形态的操作模式，这也是 Verilog HDL 语言不为人知的另一面。目前笔者可见的操作模式有 初模型，流水操作，循环操作和即时操作。从不同操作模式之间的转换，我们必须需要一个中转站，那就是初模型的重要性。

其中最为伤脑筋还是即时操作，因为组合即时操作的基础是组合逻辑，然而笔者的笔记都是以 RTL 级设计为基础。在各种即时操作的转换实验中，笔者也只能慢慢认识组合逻辑级的设计。（说实在笔者在这方面还是很菜，需要慢慢爬才行不然跌倒可疼）组合逻辑可有许多不得不说的事情，好多内容都是被参考书无视掉 .....

最后还是重复这章笔记的大题“优化和平衡”：高级语言适合极端，然而 Verilog HDL 语言却适合平衡。在 Verilog 的世界里很多事情不是不可以干，只是干了就要付出代价，因为 Verilog HDL 语言是由许多微妙的东西构建而成，至于结果划不划算就得看情况而定。好了，这章笔记也写的挺累的，虽然内容非常精简，不过信息也蛮大 ...

Verilog 之神呀！为甚么每次指引的路，走过以后都会无比震撼和疲倦的呢？神明要笔者学习浮点数以后，又要笔者转换来转换去 .... 其中笔者似乎领悟到甚么似地？Verilog HDL 语言真的可以用水来形容，因为水是百变，Verilog HDL 语言既不是也一样吗？读者说是不是？有没有？

结束语

《Verilog 那些事儿》系列的笔记终于结束了，不知道要伤心还是要高兴。说实在笔者心理有些舍不得，却有放下石头的矛盾感觉。那些一直追从《Verilog 那些事儿》的朋友，估计也和笔者拥有同样的感觉。原本在计划里《Verilog 那些事儿》还有数学篇，可是经过笔者深思熟虑以后，觉得数学篇已经超过初学的范围，而且也不符合黑金的产品，所以就作罢了。

呵呵，这只是正当的借口而已 ... 说实话，笔者多少也有点累了，自己也已经忘记花费多少时间和精力在 《Verilog 那些事儿》里，《Verilog 那些事儿》当初也是黑金社区邀请的因缘才会开始。经过一段时间以后，《Verilog 那些事儿》得到许多朋友的认同，心理多少也有安慰。当然也有例外的朋友。

不过就算怎样，笔者也要面对现实的问题。《Verilog 那些事儿》的结束，除了短暂的满足感以外，随之而来的就是多年里累积的疲倦感和抗拒感，这些负面情绪对笔者的身体来说是不可忽视的毒药。可能会有朋友关心《Verilog 那些事儿》结束以后，还会不会有后续甚么的？如高级建模 ... 嗯~关于这一点笔者不能断定甚么，虽然脑子里确实有其他点子，但是有没有精力，时间和理由，笔者需要好好三思和衡量。简单说，就是看时机。

好了不说了，男儿结束就要潇洒一点，最后非常感谢一直以来都喜爱《Verilog 那些事儿》的朋友。笔者真心希望，如果对 Verilog 充满热情的话，就应该好好研究一下这个怪家伙，发扬它是大家的任务！接下来要走的路，稍微提示：

一、学习 Verilog 的验证语言
二、学习 NIOS II 系统
三、学习 静态时序分析
四、学习 系统级的 HDL 语言

笔者的接下来的选择是甚么？猜猜看吧，哈哈！

笔者博客：

http://blog.ednchina.com/akuei2
http:// www.cnblogs.com/akuei2