Episode 6 Secrets of FSM

——《你可能不知道的verilog提高班指南》By TA-马子睿

前言

大家好！本周我们大家体会了一个有限状态机的编写。我们在理论课中讲述的序列检测状态机是一种状态机最简单的应用，我们讲mealy型、moore型，两段式、三段式（**不要告诉我有一段式，没有，万万没有！**），你有没有感到奇怪：状态机就状态机了，搞来那么多形式，这不找麻烦呢吗？

如果你有这种想法，那么证明你已经对数字电路有了一些重要的思考。在本辑中，我将结合我自己对状态机的理解，深入剖析各类状态机的特点以及利弊，并回答你可能对状态机产生的各类问题。

目录

状态机究竟是什么?

曾有一位伟人说过：**计算机就是一台状态机**。何出此言呢？因为计算机很清楚“它的境遇”，并可以做出“相应的回应”，这就是状态机的本质。抽象的来说，状态机**描述了其所控制电路的一种状态，使得电路须按照它的想法来执行计算**。说白了，就是它指导了电路的思想。就像合唱团的指挥一样，他知道现在该男低音唱，给了男低音一个手势，那么现在整个合唱团就进入了“男低音唱”这个状态。过了一会，男低音的乐章结束了，该女高音唱了，那么指挥知道这件事情，他给男低音一个手势令其闭嘴，并给了女高音一个手势让她们唱，现在整个合唱团就进入了“女高音唱”这个状态——而不管怎样，他始终享有对整个合唱团的控制权，这就是状态机的思想。

状态机拥有一个存储器，**里面存储了整个电路调控的密码和核心——当前状态**。这个状态作为寄存器的一个输出信号，调控了一众多选器，让他们有了我们所希望的输出，而这些输出就让当前电路做出了对应的计算。因此，**状态机的每一个状态事实上都有它自己的意义**，这也就是为什么我们要在verilog中给每一个状态的编码赋予一个参数（比如S0，S1，S2，S3.....），这就是为了让我们能够理解这个状态究竟在执行什么样的计算。

总而言之，**状态机是一种常见的控制单元，它决定了它所负责的部分的电路执行哪些计算和操作**，这也能看出状态机的思想是符合我们在《计算系统概论》中学习的**“抽象”**的计算机思想。状态机在许许多多的模块中都有重大的作用，不过，它也制约了很多电路的性能，我们将在后面一一解读。

Moore与Mealy：性能到底谁说了算?

在开始这个讨论之前，我们需要了解一下：**什么是性能**。

毫无疑问，性能的高低可以使用**完成这项工作的总时间**来衡量。而对于总时间，我们往往采用以下公式：

$$\text{总时间} = \frac{\text{完成工作所需的时钟周期}}{\text{时钟频率}}$$

显然，总时间越长，性能越低，为了提高性能，我们要尽可能减少工作所需的时钟周期，也要尽可能提升时钟频率。

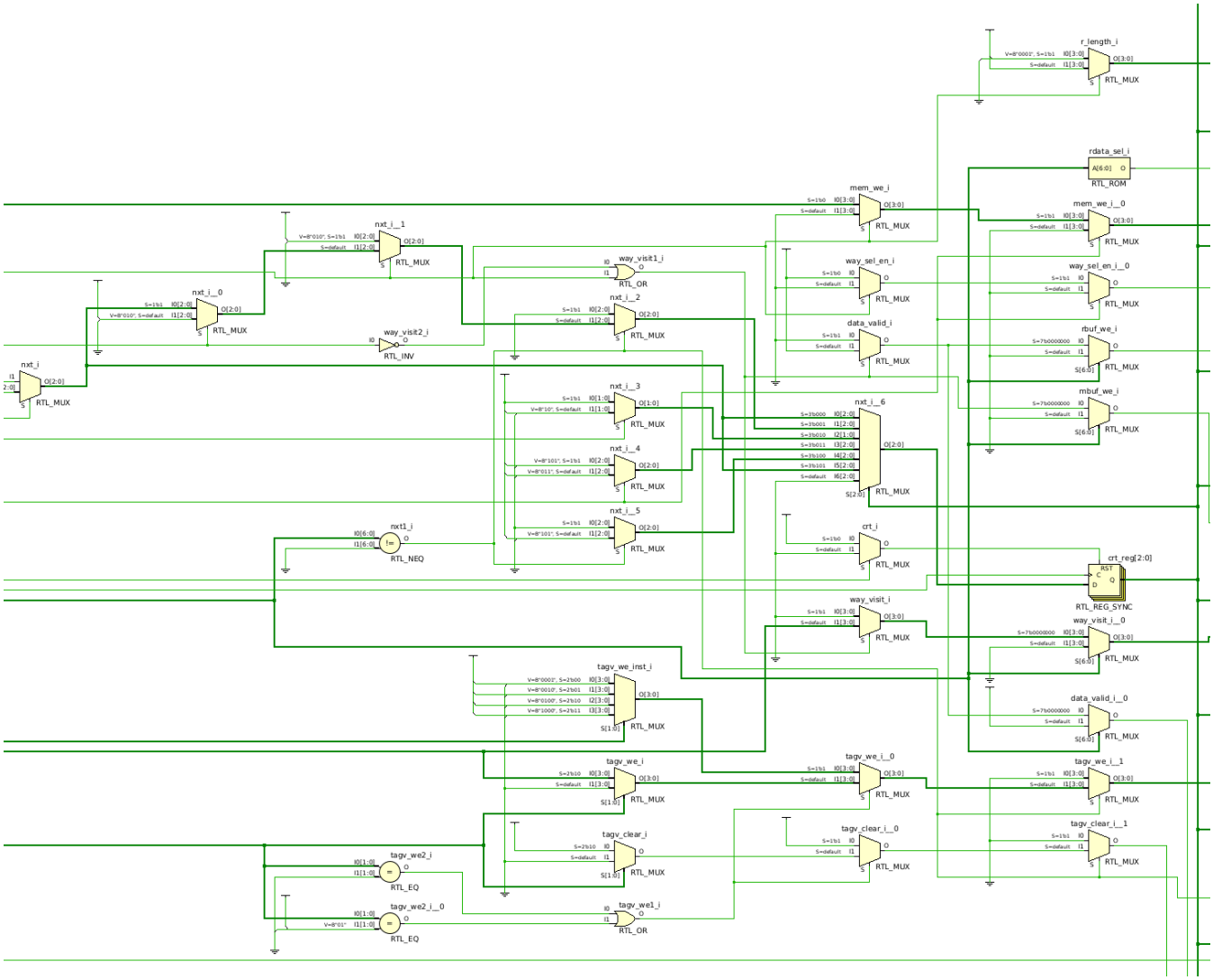
所需时钟周期显然是设计的策略问题，而制约时钟频率的往往是最长组合通路延时。这二者往往是互相制约的。比如：有一种乘法器的设计可以只用1个时钟周期算完乘法，最大时钟频率达到125MHz。如果你想设计一个性能比它好的2周期乘法器，那么你的时钟频率必须要达到250MHz以上。而**越复杂的设计，虽然可以减少所用的时钟周期，但往往也引入了更长的数据通路，导致时钟频率的降低**，所以这二者往往是互相制约的。

对于多状态的状态机，我们在使用verilog代码去描述它时，可能是如下面这样的：

```
always @(*) begin
    case(crt)
    IDLE: begin
        if(cacop_en) nxt = CACOP_COPE;
        else if(valid) nxt = LOOKUP;
        else nxt = IDLE;
    end
    LOOKUP: begin
        if(exception != 0) nxt = IDLE;
        else if(uncache) nxt = REPLACE;
        else if(cache_hit) begin
            if(cacop_en) nxt = CACOP_COPE;
            else if(valid) nxt = LOOKUP;
            else nxt = IDLE;
        end
        else nxt = REPLACE;
    end
    REPLACE: begin
        if(r_rdy_AXI) nxt = REFILL;
        else nxt = REPLACE;
    end
    REFILL: begin
        if(fill_finish) nxt = EXTRA_READY;
        else nxt = REFILL;
    end
    CACOP_COPE: begin
        if(exception != 0) nxt = IDLE;
        else nxt = EXTRA_READY;
    end
    EXTRA_READY: begin
        if(cacop_en) nxt = CACOP_COPE;
        else if(valid) nxt = LOOKUP;
        else nxt = IDLE;
    end

    default: nxt = IDLE;
    endcase
end
```

虽然它看上去复杂，但我们通过慢慢阅读，总能理解。但对于其状态转换的电路图：



这是令人发指的。我们画卡诺图真值表总可以体会到，构建一个状态机的逻辑并不容易。verilog行为级描述的背后是其编译器高负载的工作。因此，Mealy型和Moore型分别从不同角度，给出了一些思想。

Mealy型：我比你快一个周期就能反应！

为什么输出和输入有关就可以快一个周期呢？这是因为，在时钟上升沿来到的那一刻，你的输入是什么就是什么了，总不能说上升沿到来后再改变输入会对当前状态有影响吧？与其这样，**我为什么一定要搞一个接收状态？我直接根据接收状态的前一个状态与输入的数据来判断输出数据并直接输出不就好了吗？反正你下一个周期肯定就要变成接收状态了嘛！**

是的，在大家熟知的众多序列检测状态机中，Mealy型就是比Moore型少一个周期。但是相应的，我们看到了mealy型的输出判断多了一个if，这个差别可能大家体会不到，但如果是这样呢？

```
LOOKUP: begin
    if(exception_temp == 0) begin
        if(cache_hit && !uncache) begin
            data_valid = 1;
            cacop_ready = 1;
            cache_ready = 1;
            if(valid || cacop_en) rbuf_we = 1;
        end
        if(op == READ) begin
            if(is_atom_rbuf) llbit_set = 1;
        end
        else begin
```

```
            if(is_atom_rbuf) llbit_clear = 1;
            if(!is_atom_rbuf || llbit_rbuf) begin
                mem_en          = hit;
                mem_we          = mem_we_normal;
                dirty_we        = hit;
                w_dirty_data    = 1;
            end
        end
    end
    else data_valid = 1;
end
```

如此多层的if，事实上他们都是对状态寄存器输出的处理，无疑在状态寄存器的输出后又增加了过多的组合延时。

仔细想想我们可以发现，状态机的输出事实上完全依赖于一个以状态编码为选择信号的n选一选择器（n就是总状态数+default），这样的多选器显然很令人咋舌——加上这个多选器前面魔鬼一般的组合逻辑，延迟不可想象对吗？这就是Mealy型的特点：**利用“抢周期”的手段高效解决问题，却引入了一条很长的组合通路。**

Moore型：跑的那么快，你不怕摔着么？

对于我们说的Mealy型状态输出的组合通路问题，Moore型给出了一个很好的解答：**我的输出只需要用当前状态作为选择信号来进行输出就可以。**这听上去是个很不错的想法，但事实上，它却引入了更多的状态，也就引入了更多输入的多选器，这同样增加了组合延迟，只是一个更多输入的选择器，比更多更长的选择器链，可能拥有更好的性能。

在很多情况下，Moore型状态机的工作周期会多一些，因为它引入了更多的状态，但由于其输出绝对依赖当前状态，因此输出逻辑会非常非常简单，有助于时序的优化。但Moore型很有可能多引入状态，从而使整个电路处理时钟周期更多，造成性能的损失。

如何选择Mealy型和Moore型？

- Mealy型在针对输入很少的情况下很有用，因为这时状态转换时的if嵌套不会太多，而且Mealy型的状态机往往能最小化状态集。
- Moore型在针对输入多的情况下比较有用，我们可以通过多设状态来极小化输出逻辑，这是因为，状态机并不是一个孤立的部分，它的输出往往还要连往其他模块。如果输出信号的生成逻辑更简单，那么它就可以让后续通路的组合延时在允许范围内更长。

总而言之，**Mealy型致力于节省时钟周期数来提高性能，而Moore型致力于减少组合延迟来提高性能。**我们在实际设计中，可以在根据自己的考虑，来选择恰当的状态机。

为什么两段式依然有存在的意义？

我们都知道，一段式状态机已经被淘汰在了历史的长河中，但两段式状态机为什么还有它存在的意义呢？我们来深刻剖析一下：

两段式和三段式的区别在哪里？一个最最重要的区别就是，**三段式以寄存器作为输出，而两段式以组合电路作为输出。**两段式最大的问题在于，**组合电路产生的输出造成了潜在的组合时延问题。**而三段式通过把组合信号寄存一级的方法，最大程度上为后续需要此输出信号的电路提供了时延保障。但是，也正是**这一级寄存器**，导致了电路可能达不到我们所期望的效果。

有同学可能会说：既然要锁存一级，我们直接看next_state来决定输出信号不就好了嘛！在简单的设计中，这个事情确实是很好的解决方案。但是，对于复杂的设计，**看下一个状态**的策略可能并不适用。

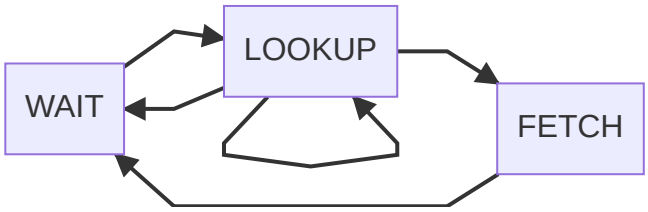
我们再来深刻了解一下状态机：**状态机是将整个电路的状态唯一确定下来的模块。**它的优点是**确定**，缺点是**唯一**。我们在做硬件设计的时候，总是尽可能挖掘其中的并行性能，而状态机却在一定程度上减少了这种可能——因为它太符合人类的思维了。

流水化买书

考虑我们生活中的一个情景——**流水化买书**：你现在是一个报刊亭的售卖员。报刊亭有两个窗口：在A窗口，顾客向你投递了一封信，信里有他想要的刊物名，且这个窗口只能容纳下一封信（多扔进来的信会永久覆盖掉之前放在这里的信）。你会取出这封信，拆开看书名，**如果这本书正好在报刊亭里有，你就把书放到B窗口，然后顾客从这里立刻取走；但假如这本书里报刊亭没有，你要立刻离开报刊亭，驱车十几里到最近的仓库取书。**那么你的所有服务的状态有哪些呢？

- WAIT：等顾客来
- LOOKUP：在报刊亭内找书，只需“一个时钟周期”就可以找到，
- FETCH：去报刊亭拿书

这个状态机如下：



一次取书活动的完结标志是：**书被放在了B窗口**，这可以看做一个**book_ok**输出信号。

这里我们考虑用一个三段式状态机来描述这个问题时出现的问题：**假如你的报刊亭巨大无比，里面存储了比西图还多的书，那么永远不会进入FETCH这个状态，只会在WAIT和LOOKUP这两个状态之间切换。**这时我们来考虑一个问题：**如果顾客一直在来，我们取走每封信后，难道在你查找完书前，还不能让顾客继续投信？**那A窗口显然没有发挥它最大的效用。

因此我们应当这样做：在顾客一直来的时候，你应该允许顾客在你拿走一封信后立刻投递，这样你就可以一直在LOOKUP状态中，而不必每次查找完书后都等“一个时钟周期”。当然，顾客怎么知道可以往里投信而不会覆盖掉之前那个人的信呢？很简单！**只要前一个顾客的前一个顾客的书出现在了窗口B，那么他知道你这个周期一定会拿走前一个顾客的信，你也就可以同时趁机投递一封信。**

问题来了：如何产生book_ok这个信号？条件是：**你进入了LOOKUP状态，且书被找到了。**问题就在这里：**书被找到**这个信号，是你进入了LOOKUP状态才能产生的！在WAIT状态，你不能未卜先知，没拆信就明白了用户的需求。

看一下我们**看下一个状态**的三段式状态机：第一个时钟周期，你在WAIT状态，同时这时候你看到了一封信，你下一个状态是LOOKUP，但则个时候书还没有开始被找，也就是没有被找到，因此，你不能产生book_ok信号。下一个周期，你进入了LOOKUP状态，同时这时候没有下一个人来投递了，你的下一个状态是WAIT，**但是由于你只看下一个状态，这个时候书被找到了，你却不能发出book_ok信号！**糟了，明明找到了书却没有告诉顾客你找到了，顾客会在B窗口等一万年！

解决的办法只有一个：构造一个**看当前状态**的三段式状态机。可是，这是一个输出需要等一个时钟周期的状态机，相当于你找到了书时候，顾客并不能立刻看到book_ok信号，相当于你这个报刊亭里有另外一个人，你找到书之后需要交给他，他在下一个周期时无脑把你交给的书放到窗口B。我们来考虑一下下面这种情况：

- 第一个时钟周期，顾客1把信投递到窗口A

- 第二个时钟周期，你打开了顾客1的信，顾客2同时把信投递到窗口B，发现他想要的书你们这里并没有（这时内部的两个人已经知道了书籍缺失，并且下一个周期就将进入FETCH状态），这时顾客3看到窗口B没有书，他觉得你们没找到顾客1的书，因此也没有处理顾客2的信，所以不敢往里继续投信。
- 第三个时钟周期，一切都被阻塞住，你们去图书馆找书了。

一切看起来那么完美，可是稍微变化一下：

- 第二个时钟周期，顾客2把信投递到窗口B，你同时打开了顾客1的信，发现你们这里有他需要的书，你找到了书并把它交给你的同伴，这时顾客3看到窗口B没有书，他觉得你们没找到顾客1的书，因此也没有处理顾客2的信，所以不敢往里继续投信。
- 第三个时钟周期，你打开了顾客2的信，这时又找到了顾客2的书，同时顾客3看到了顾客1的书出现在了窗口B，他决定下个周期把信投进窗口A，但你此时没有看到新的信，因此下一个周期将进入等待状态
- 第四个时钟周期，你等着新的信来，同时顾客3把信投入了窗口A。

有没有发现问题：顾客3盲等了一个周期。你肯定会说：顾客3真是个胆小鬼，大胆一点放就好了！反正这个周期你们了处理顾客2的信！。但是！**如果这个周期你们真的没找到顾客1的书，你们走了，没有看到顾客2的信，顾客3大胆扔信，就会覆盖掉顾客2的信，这就导致了你们从图书馆回来之后永远不会看到顾客2的信了！**

上面这个例子暴露出的问题是：**如果状态转换（WAIT到LOOKUP）后输入信号（书被找到了）才能到达，那么三段式状态机就失去了它的优势，甚至你无脑看下个状态会引发严重错误！**

上面这个问题怎么完美解决呢？**用两段式状态机**。不要让你的输出过一级寄存器**（也就是开除你的同伴），这样顾客3在第三个时钟周期前就可以看到你放在窗口B的书**，这样他会觉得：**哦哦！顾客1的需求被满足了，他这个周期肯定会拿走顾客2的信！我投递就好！**同样的，这样不会引发错误：顾**客3没看到你放在窗口B的书，这证明你真的没找到顾客1的书，也就不会处理顾客2的信件，他也不会继续投递。**

好了，上面这例子就说明了，两段式状态机**抢周期**的思路，真的可以解决**流水化买书**的问题！它的实际应用就是**高速缓存**：报刊亭里卖书的你就是高速缓存的控制单元，顾客就是一条一条的访问存储指令，报刊亭就是高速缓存，仓库就是一个更大的存储器。

结语

在本辑中，我们通过几个例子，来分析了不同状态机的应用场景。在实际应用中，我们总是尽可能的使用三段式状态机，因为它可以最大程度优化时序。但依然如我们之前所说：总的性能不仅由时序决定，更由一个操作需要使用的时钟周期决定。当我们可以把3个时钟周期用两段式状态机变成2个时钟周期，除非我们可以真的把时钟周期从100MHz提升到166.67MHz，否则，两段式状态机依然是我们最好的选择。