# 中国科学院大学计算机组成原理（研讨课）

# 实 验 报 告

学号：2021K8009929001　　姓名：谢嘉楠　　专业：计算机科学与技术

实验序号：3　　实验名称：定制 MIPS 功能型处理器设计——真实内存、外设与性能计数器访问

一、实验目的

1. 将实验2中的单周期处理器修改为多周期处理器；
2. 硬件：改进基于理想内存的处理器，访问**真实内存**通路；
3. 软件：基于改进后的处理器访存接口，实现简单***I/O*外设访问**，支持字符串打印；
4. 硬件 + 软件：添加**性能计数器**并对复杂*benchmark*进行性能评测。

二、实验过程

1. 基于真实内存的多周期处理器的实现

与单周期处理器相比，我们实验中需要实现的多周期处理器将一条指令的执行分为几个阶段，并且每个阶段需要一个或者多个时钟周期。这样可以提高硬件的复用频率，也有利于缩短处理器的时钟周期（提高主频），而且不同指令的执行时间不需要相同，简单指令不需要等待复杂指令，也可以加快处理器的速度。而单周期中主要的组合逻辑都可以保留不变，主要需要添加的是实现状态机的同步跳转，以及更多的寄存器的加入。

(1) 添加更多寄存器

首先是需要更多的寄存器：$IR$用来存从内存中取出的指令，$MDR$用来存从内存中取出的数据，$ALUout$用来存$ALU$计算的结果。

```
// 多周期需要加入的一些寄存器
    reg  [31:0] IR; //Instruction Register
    reg  [31:0] MDR; //Memory Data Register
    reg  [31:0] PC_reg; //PC Register
    reg  [31:0] ALUout; //ALU Register 用于存储 ALU 计算的结果
```

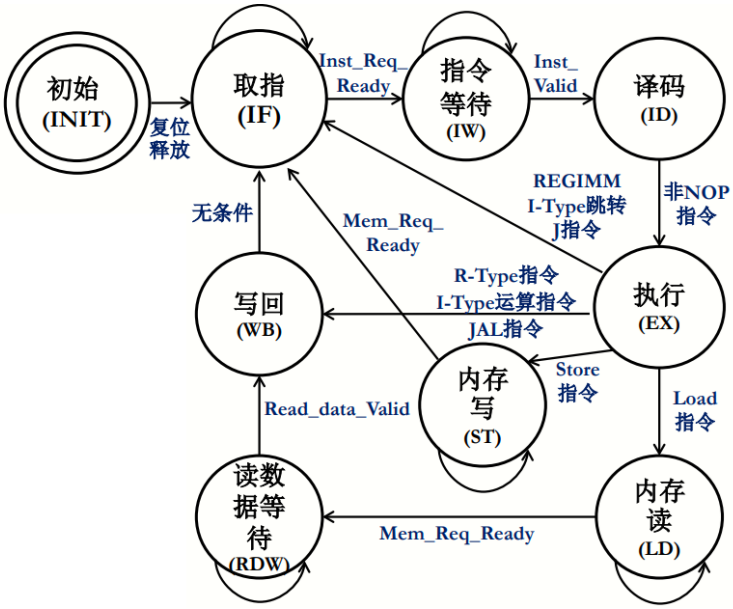
$ALUout$的加入是因为某些指令（如$branch$类指令的$beq$）需要在指令周期中的**译码和执行**阶段两次使用到$ALU$，译码阶段$ALU$的$result$在进入执行阶段时（下一个上升沿时刻）被传入$ALUout$中保存而不会消失，而在执行阶段就可以同时使用此时$ALU$的$result$和$ALUout$中的数据了。

如果不考虑$PC$的更新，大多数指令的指令周期内只使用一次$ALU$，而且$ALU$的计算一般在执行阶段完成，计算结果的使用一般在执行的下一个阶段，所以也可以直接使用$ALUout$中的数据。将计算结果存入$ALUout$还有一个好处是，寄存器内的值只有在时钟上升沿才变化，所以在使用结果的这一个时钟周期内结果一定是稳定的，不会因为$ALU$输入端的无效数据而变化。

与理论课上讲的多周期处理器相比，我们这里不需要寄存器$A$和$B$，因为每条指令都只需要从寄存器堆中至多读一次（2 个）数据，可以直接将读操作简化为异步的组合逻辑，直接在执行$(EX)$阶段读数，不需要在前一阶段译码$(ID)$时就将数据读出，传入$A$，$B$中。这样还能够简化读操作的控制条件。

(2) 状态机实现

接下来是实现状态机的控制。状态机状态的编码使用$one-hot$码，因为共有 9 个状态，所以需要 9 位位宽。因为真实内存的访问会存在一些不确定的延迟，并且只有当$CPU$和内存控制信号完成“握手”才能传输数据，所以一次读/写操作可能需要多个时钟周期，在状态机也增添了中有$IW$(指令等待)状态和$RDW$(读数据等待)状态。




通过“三段式”的方法描述状态机，思路还是很清晰的。第一段是现态*currunt_state*的更新，只需要在每次时钟上升沿将次态*next_state*的值给到*currunt_state*就可以了。

```
// 状态机第一段：描述状态寄存器 current_state 的同步状态更新
    always @(posedge clk)begin
        if(rst==1'b1)
            current_state <= INIT;
        else
            current_state <= next_state;
    end
```

第二段是对次态*next_state*的赋值，它是组合逻辑，所以应该用阻塞赋值“=”，因为次态的值要根据译码、执行的结果实时变化，需要等到第一段*always*块执行完再执行。它的值需要根据*currunt_state*和指令类型来决定。这里使用了*always* @(∗)和*case*语句来实现对各个状态的分类（如果我使用纯组合逻辑来进行分类，写起来应该会比较麻烦吧）。虽然是*always*块，但是它等同于组合逻辑。*always* @(∗)用于输出信号敏感于所有输入的情况下，即当输入信号任何一位发生变化时，该*always*块都会被重新执行。它通常用于**组合逻辑**的实现。

第三段是实现*CPU*中各个寄存器的同步赋值，是时序逻辑。*PC*，*IR*，*MDR*，*ALUout*的更新都要考虑当前状态*currunt_state*和控制信号的值。因为第二、第三段都比较长，这里就不贴过来了。

(3) 对组合逻辑部分的修改

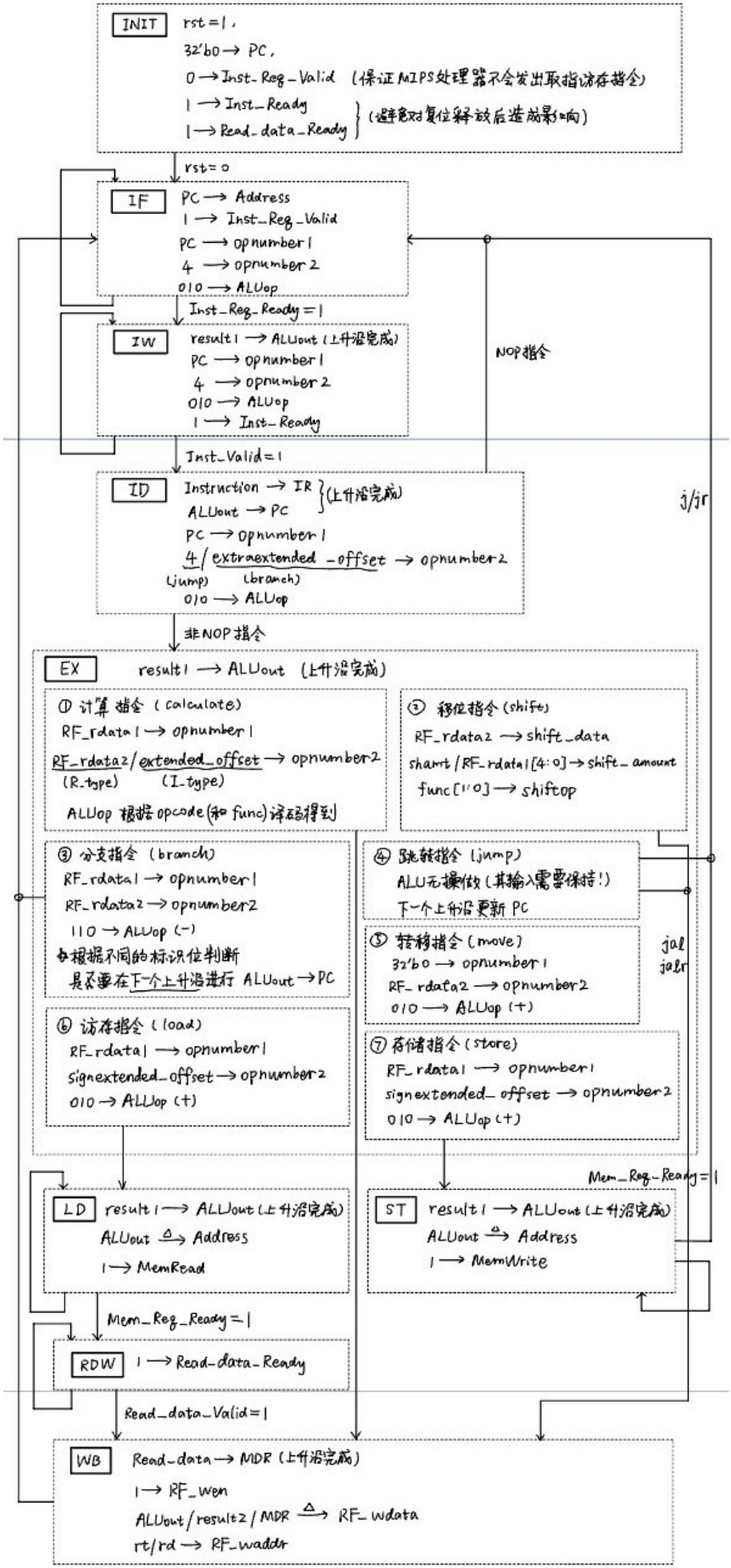
至于组合逻辑部分，大部分单周期*CPU*的代码都可以复用到这里，只要改的比较多的是*ALU*模块输入的控制。因为从单周期到多周期，需要把*PC*的加法也在*ALU*中完成，这样控制*ALU*的逻辑会变得更加复杂。所以我使用了 4 个使能信号来控制两个操作数的输入，其中*ALUsrcA*和*ALUsrcB*根据状态来分类，*ALUsrcC*和*ALUsrcD*根据指令功能来分类，然后在使用时可以将两中使能信号组合起来使用，就实现了对不同指令在不同状态下的精准分类。

```
wire [1:0] ALUsrcA      =
wire [2:0] ALUsrcB      =
wire [1:0] ALUsrcD      =
wire [2:0] ALUsrcC      =
```

组合逻辑的修改需要仔细去找需要修改的地方，想一想每个模块的控制有哪些不同，然后定位去修改。我在这个过程中还是遗漏了几个信号（比如说寄存器堆的写数据应该从*MDR*来了，不是*ReadData*），后来查看仿真波形才发现。

(4) 总结思考

我觉得实现多周期处理器，比较重要的一步是要想清楚指令的每一阶段需要完成哪些“微指令”，可以按照指令的功能来分类，这样 45 条指令就分为了 7 类，把每一类指令的整个周期过一遍，就会比较清楚。


INIT
rst=1,
32'b0 → PC,
0 → Inst_Reg_Valid (保证MIPS处理器不会发出取指令指令)
1 → Inst_Ready
1 → Read_data_Ready
(避免对复位释放后造成影响)
rst=0
IF
PC → Address
1 → Inst_Reg_Valid
PC → opnumber1
4 → opnumber2
010 → ALUop
Inst_Reg_Ready=1
IW
result1 → ALUout (上升沿完成)
PC → opnumber1
4 → opnumber2
010 → ALUop
1 → Inst_Ready
NOP指令
Inst_Valid=1
ID
Instruction → IR
ALUout → PC
(上升沿完成)
PC → opnumber1
4 / extraextended_offset → opnumber2
(jump) (branch)
010 → ALUop
j/jr
非NOP指令
EX
result1 → ALUout (上升沿完成)
① 计算指令 (calculate)
RF_rdata1 → opnumber1
RF_rdata2 / extended_offset → opnumber2
(R-type) (I-type)
ALUop 根据opcode(和func)译码得到
② 移位指令 (shift)
RF_rdata2 → shift_data
sham / RF_rdata1[4:0] → shift_amount
func[1:0] → shiftop
③ 分支指令 (branch)
RF_rdata1 → opnumber1
RF_rdata2 → opnumber2
110 → ALUop (-)
*根据不同的标识位判断
是否要在下一个上升沿进行 ALUout → PC
④ 跳转指令 (jump)
ALU无操做 (其输入需要保持!)
下一个上升沿更新 PC
⑤ 转移指令 (move)
32'b0 → opnumber1
RF_rdata2 → opnumber2
010 → ALUop (+)
jal
jalr
⑥ 访存指令 (load)
RF_rdata1 → opnumber1
signextended_offset → opnumber2
010 → ALUop (+)
⑦ 存储指令 (store)
RF_rdata1 → opnumber1
signextended_offset → opnumber2
010 → ALUop (+)
Mem_Reg_Ready=1
LD
result1 → ALUout (上升沿完成)
ALUout → Address
1 → MemRead
ST
result1 → ALUout (上升沿完成)
ALUout → Address
1 → MemWrite
Mem_Reg_Ready=1
RDW
1 → Read_data_Ready
Read_data_Valid=1
WB
Read_data → MDR (上升沿完成)
1 → RF_wen
ALUout / result2 / MDR → RF_wdata
rt/rd → RF_waddr

上图为我整理的不同功能的指令在其指令周期的各阶段的一些微指令。有了这张图，编写状态机时就只要看各个状态之间转移的条件就行了，而对新增加的寄存器和控制信号的赋值也可以在各个状态的微指令中找到。

一开始我思考时其实对于一些操作的完成时间不是很清楚。比如说$PC+4$什么时候计算，$PC+4$的结果什么时候给到$PC$，比如说分支指令的分支地址什么时候计算，分支条件又什么时候计算，再比如说每条指令$ALU$的计算结果是否会在下一阶段被使用，如果不是立刻被使用的话，$ALUout$就需要保持，或者$ALU$的输入就需要保持……

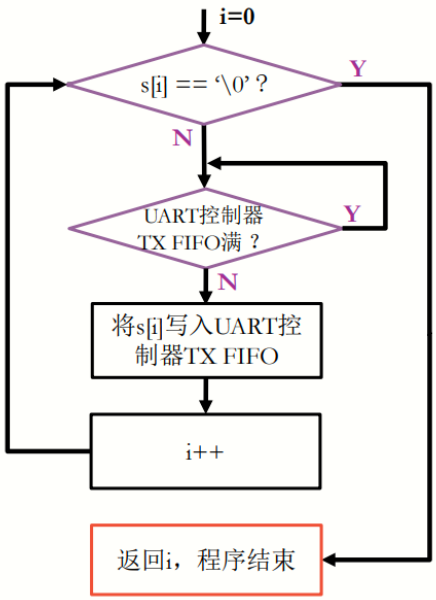
这些具体的场景在脑子里空想会很乱，把它们写出来会好很多。而且在写状态机之前这些微指令的执行时间是必须要搞清楚的，否则因为多周期硬件的复用，指令执行的时候前后就会打架，在到仿真波形中去看会花更多时间。Plus，看波形可能会遇到非常让人头疼的错误，比如说一条$store$指令的写入数据错了，但是它错误的根源竟然是很久很久之前某条指令写入寄存器的数据是错的……

我在整理上图时也发现一些问题。首先是计算指令，不管是$R_type$还是$I_type$，它的计算我是放在$EX$阶段完成的，但是因为我们这里从寄存器读操作数是异步的，而且也没有设寄存器，所以如果不考虑延迟就可以认为可以瞬间从寄存器堆中拿到操作数，然后给$ALU$去计算，这样是不是就可以把计算直接前移到译码阶段去做？但如果设有寄存器$A$，$B$来同步地更新寄存器堆中读出的数据，那就需要在译码阶段先读数据，在执行阶段计算了。

其次是$jump$跳转指令。$jal$和$jalr$指令要经过写回阶段，将$PC+8$存入寄存器，而$j$和$jr$执行阶段之后就跳回取指了。而我是在译码阶段计算$PC+8$的，因此在执行阶段$jump$指令就没有什么操作可以做，这一阶段就空出来了。但是空出来了就有一个问题，译码阶段计算出的$PC+8$在执行阶段被更新到$ALUout$寄存器中，但是在执行阶段到写回阶段的时钟上升沿，$ALUout$中的值需要保持，才能在写回阶段被使用。所以我采取的方法是在执行阶段保持$ALU$的输入，相当于重复计算$PC+8$。这个问题我在查看波形时才找到，所以印象特别深刻。

2. 字符串打印函数$puts()$的实现，简单**$I/O$外设访问**

字符串打印程序中，我们需要实现的部分就是将字符串$s$放入$UART$控制器的$TX$寄存器，让$TX$寄存器去输出到$I/O$外设，还是比较简单的。在这里，$TX$寄存器其实就是内存中的一段（32 位，4$byte$）存储空间，所以可以通过$C$语言的指针来访问。而对照着$PPT$上的这张流程图也可以很清楚地理解$puts()$函数要完成的功能。




$puts()$将字符串$s$一个字符一个字符地写入$TX$寄存器的最低 8 位上，而因为我们的内存是小端序存储的，所以$TX$的最低 8 位对应的地址也是$TX$地址段中最小的字节，即

$$(unsigned\ char*)uart + UART\_TX\_FIFO$$

这里$uart$指针是$unsigned\ int*$型的，需要将它转换成$unsigned\ char*$型，才能加上$TX$寄存器的偏移量，否则$uart$指针加 1 就相当于加 4 字节，即 4 存储地址单元，这样偏移出的地址就会有问题。当$TX$

寄存器满时，需要等待，因此我就放入了一个空的*while*循环，只要*TX*满时就一直在循环中，这样就可以实现“等待”的控制。*puts*()函数如下所示。

```
int
puts(const char *s)
{
    int i;
    unsigned char * TX_target = (unsigned char*)uart + UART_TX_FIFO;
    for(i = 0; s[i] != '\0'; i++){
        while((*(uart+2) & UART_TX_FIFO_FULL) != 0){
            ;
        }
        *TX_target = s[i];
    }
    return i;
}
```

3. 添加**性能计数器**

添加性能计数器其实并不复杂。只要在*CPU*中设一些专门的寄存器来统计这些与性能相关的量，例如时钟周期数量、指令周期数量，以及访存周期的数量等等。将这些寄存器的值赋值给内存中的接口，再通过软件访问到这些内存地址，就可以统计这些值了。

我在实验中就实现了统计时钟周期（*cycle_counter*）、指令周期（*inst_cycle_counter*）和访存周期（*men_cycle_counter*），还是挺简单的。统计时钟周期的函数如下所示。

```
unsigned long _uptime() {
  unsigned long *cc_addr = (unsigned long*)0x60010000;
  //   You can use this function to access performance counter related with time or
cycle.
  return *cc_addr;
}
void bench_prepare(Result *res) {
  //   Add preprocess code, record performance counters' initial states.
  //   You can communicate between bench_prepare() and bench_done() through
  //   static variables or add additional fields in `struct Result`
  res->msec = _uptime();
  res->mem_cycle = _read_mem_cycle();
  res->inst_cycle = _read_inst_cycle();
}
void bench_done(Result *res) {
  //  Add postprocess code, record performance counters' current states.
  res->msec = _uptime() - res->msec;
  res->mem_cycle = _read_mem_cycle() - res->mem_cycle;
  res->inst_cycle = _read_inst_cycle() - res->inst_cycle;
}
```

实验中*microbench*统计到的一组数据是:

```
cycle number:404218
mem_cycle number:1304
inst_cycle number:5231
```

这样可以估计得到这个测试程序在*MIPS*指令集下的的$CPI \approx 77$，用其他测试统计到的数据计算*CPI*也基本要在七八十左右。在上一个单周期处理器实验中，理想内存只需要一个时钟周期就可以读出或存入数据，而这次实验的真实内存就非常显著地慢好多，可见访存返回数据相对于*CPU*内部的操作是非常

慢的，往往需要许多时钟周期来完成，所以很多时候需要在$CPU$内设更多寄存器（$cache$）、或者通过编译优化来减少访存的次数，从而降低$CPI$，加快指令执行的速度。

三、实验思考

1. $printf.c$中的$UART$控制器基地址指针使用了$volatile$关键字。什么是$volatile$关键字？它有什么作用呢？我从来没用过，查了一下，大概懂了一点。

由于内存访问速度远不及$CPU$处理速度，为提高机器整体性能，编译器会对$C$程序进行编译优化。

1）硬件一级的优化：引入硬件高速缓存$Cache$，加速对内存的访问。另外在现代$CPU$中指令的执行并不一定严格按照顺序执行，没有相关性的指令可以乱序执行，以充分利用$CPU$的指令流水线，提高执行速度。

2）软件一级的优化：一种是在编写代码时由程序员优化，另一种是由编译器进行优化。编译器优化常用的方法有：将内存变量缓存到寄存器。由于访问寄存器要比访问内存单元快的多，编译器在存取变量时，为提高存取速度，编译器优化有时会先把变量读取到一个寄存器中；以后再取变量值时就直接从寄存器中取值。但在很多情况下会读取到脏数据，严重影响程序的运行效果。

$Volatile$意思是“易变的”，“易变”是因为外在因素引起的，像多线程，中断等。C 语言$volatile$关键字的作用就是告诉编译器，该变量的值可能会被意外修改，因此编译器不应该对该变量进行优化。

2. 状态机给各个状态的编码使用了独热码$(one-hot)$，即每一个编码只有一位是 1，其他位均为 0。为什么我们使用的是独热码而非二进制码或格雷码呢？

那就要从每种编码的特性上说起了，首先独热码因为每个状态只有 $1bit$ 是不同的，例如在执行到（$state == IF$）这条语句时，综合器会识别出这是一个比较器，而因为只有 1 位为 1，所以综合器会进行智能优化为$state[2] == 1'b1$，这就相当于把原本需要的 9 比特的比较器变为了 1 比特的比较器，**大大节省了组合逻辑资源**，但是付出的代价就是**状态变量的位宽需要的比较多，而我们 $FPGA$ 中组合逻辑资源相对较少，所以比较宝贵，而寄存器资源较多**，所以很完美。

而**二进制编码和格雷码的情况和独热码刚好相反**，因为它们使用了较少的状态变量，使之在减少了寄存器状态的同时无法进行比较器部分的优化，所以使用的寄存器资源较少，而使用的组合逻辑资源较多。

3. 在状态机第三段中，需要使用$current_state$作为判断条件或赋值变量，但是尽量不要使用$next_state$。我觉得这是因为$next_state$是通过组合逻辑（$always\ @(*)$）赋值的，它的值会随着$current_state$、指令类型以及标志位的变化而实时地变化。当时钟上升沿到来时，$current_state$会更新，它的更新又会导致$next_state$的变化，但是因为电路的延迟，$next_state$值的变化是稍晚于时钟上升沿的，所以如果寄存器的赋值以$next_state$作为判断条件的话，可能会导致使用瞬间采样得到的信号恰好正在变化，比如某一位的值恰好处于 0 和 1 之间的不定态，就会带来问题。

4. 关于$Verilog$中的$always\ @(*)$。$Verilog$中的$always$块是一种用于描述时序逻辑的语句块。它可以在特定的时钟信号或输入信号变化时执行一系列操作。

我们用过的$always$块有以下几种类型：

$always\ @(*)$：表示在任何输入信号变化时都会执行。

$always\ @(posedge\ clk)$：表示在时钟信号的上升沿（$positive\ edge$）时执行。

$always\ @(negedge\ clk)$：表示在时钟信号的下降沿（$negative\ edge$）时执行。

$always\ @(posedge\ clk\ or\ negedge\ reset)$：表示在时钟信号的上升沿或复位信号的下降沿时执行。

$always\ @(*)$是一种组合逻辑，它会在任何输入信号发生变化时重新执行$always$中的内容。它的作用是将输入信号直接映射到输出信号，因此不需要指定特定的输入信号。这种组合逻辑在硬件实现中通常使用多路选择器（$MUX$）来实现。$MUX$可以根据输入信号的值选择正确的输出信号。在$Verilog$中，可以使用$case$语句或$if-else$语句来实现$MUX$。