中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2020K8009907032　姓名：唐嘉良　专业：计算机科学与技术

实验序号：03　　实验名称：定制 MIPS 功能型处理器设计

注 1：撰写此 Word 格式实验报告后以 PDF 格式保存在~/COD-Lab/reports 目录下。文件命名规则：prjN.pdf，其中“prj”和后缀名“pdf”为小写，“N”为 1 至 4 的阿拉伯数字。例如：prj1.pdf。PDF 文件大小应控制在 5MB 以内。此外，实验项目 5 包含多个选做内容，每个选做实验应提交各自的实验报告文件，文件命名规则：prj5-projectname.pdf，其中“-”为英文标点符号的短横线。文件命名举例：prj5-dma.pdf。具体要求详见实验项目 5 讲义。
注 2：使用 git add 及 git commit 命令将实验报告 PDF 文件添加到本地仓库 master 分支，并通过 git push 推送到 GitLab 远程仓库 master 分支（具体命令详见实验报告）。
注 3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

一、 逻辑电路结构与仿真波形的截图及说明（比如关键 RTL 代码段{包含注释}及其对应的逻辑电路结构图、相应信号的仿真波形和信号变化的说明等）

本实验无需修改 alu 以及 reg_file。由于本实验没有什么至关重要的波形说明，debug 的时候波形图也未能很好地帮助到我，本部分不放波形图。

本实验主体部分是 custom_cpu.v 代码，我们要完成的任务就是将实验 2 的单周期处理器的主体进行修改，加入状态机将其转变为多周期处理器，并且进行一些多周期的适用性修改。

首先，为实现状态机，我们先用 verilog 语言中的参数类型定义所有状态的独热码（one-hot code），并且考虑到状态机的状态转移，定义 current_state 和 next_state 两个变量，分别代表当前状态和下一状态。如下所示：

```
// TODO: Please add your custom CPU code here
//define the state
reg [8:0] current_state;
reg [8:0] next_state;
reg [31:0] PC_old;
reg [31:0] Instruction_valid;
reg [31:0] Read_data_valid;//the valid read_data,not the shake hands signal
//encode the state----one hot code
localparam INIT    = 9'b000000001,
           IF      = 9'b000000010,
                   IW      = 9'b000000100,
                   ID      = 9'b000001000,
                   EX      = 9'b000010000,
                   LD      = 9'b000100000,
                   ST      = 9'b001000000,
                   RDW     = 9'b010000000,
                   WB      = 9'b100000000;
```

可以看到我还另外定义了 PC_old、Instruction_valid、Read_data_valid 三个变量（它们分别与 PC、Instruction 以及 Read_data 具有同样的位宽，其作用将在后面详细说明）。

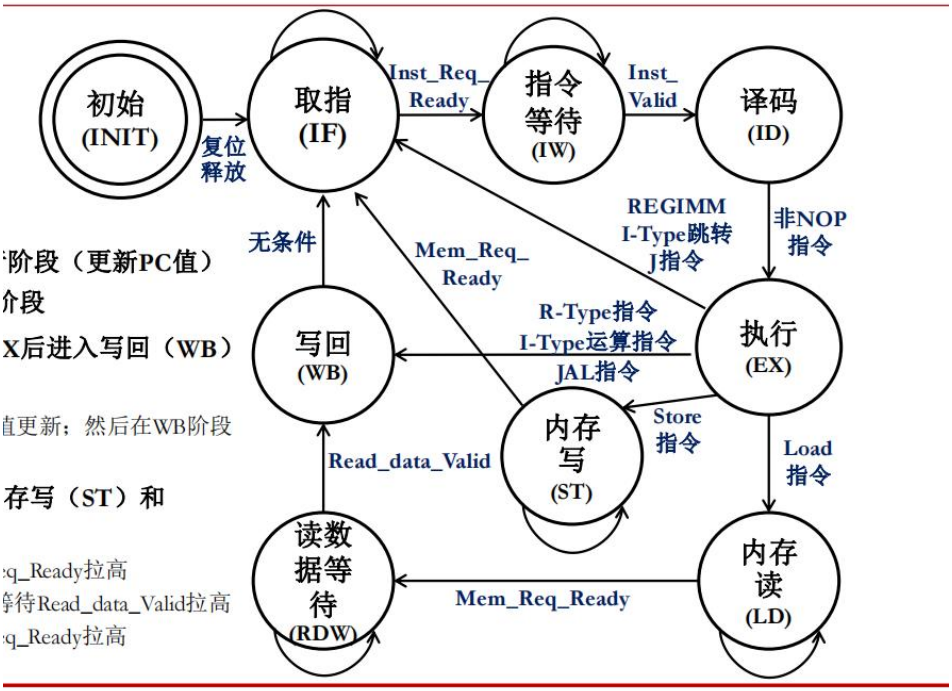
注意到，我没有采用宏定义而是参数类型来定义状态机状态。事实上对于目前的实验框架来说，二者都是可行的，但是参数定义对于含有状态机的项目框架来说更为严谨——因为`define 宏定义在编译时自动替换整个设计中所定义的宏，而 localparameter 仅仅定义模块内部的参数，不会与模块外部的其它状态机混淆——如果两个模块的 FSM 均含有 INIT 这个名称的状态，那么宏定义会混淆而 localpapameter 并不会。

如下所示是 FSM 第一段的代码。由于我采取三段式写法，按照 PPT 所示，第一段应当为状态机的状态转移时序逻辑，即在每个时钟上升沿都进行状态转移，如有复位信号则无条件将状态机状态恢复至初始状态 INIT。

```
111  //The clk logic of state
112  always @(posedge clk) begin
113          if(rst == 1'b1)
114              current_state <= INIT;
115          else
116              current_state <= next_state;
117  end
```

接下来是 FSM 第二段的代码。这一段的主要工作在于确定状态机的下一状态，采用连续赋值 always @(*)来实现。以 IF 为例，如果当前处于 IF 状态，那么要根据 Inst_Req_Ready 信号是否到来选择停留在 IF 还是进入 IW 等待接收状态，这时就采用 if-else 逻辑判断。如果 Inst_Req_Ready 信号到来，那么 next_state 就是 IW，如果尚未到来，那么仍然停留在 IF。其它状态是同理的，根据 PPT 上的状态转移说明（如下图），可以轻易地写出第二段的代码。

这里我采取了一个小技巧：在 EX 状态下有四种转移，而我将最难判断的到达 WB 的转移放在最后一个互斥的 else 块中，剩余三个转移都属于较好写出条件的。这样减少了出错可能，同时简化了代码。



```
//The calculate of next_state
always @(*) begin
    case(current_state)
            INIT:
                    next_state = IF;
                IF:
                    begin
                                if(Inst_Req_Ready) begin
                                        next_state = IW;
                                end
                                else begin
                                        next_state = IF;
                                end
                        end
                IW:
            begin
                                if(Inst_Valid) begin
                                        next_state = ID;
                                end
                                else begin
                                        next_state = IW;
                                end
                        end
                ID://'nop' is all 0 (32'b0)
                    begin
                                if(Instruction_valid[31:0] == 32'b0) begin
                                        next_state = IF;
                                end
                                else begin
                                        next_state = EX;
                                end
                        end
                EX:
```

```
                end
            EX:
                begin
                        if(ins == `LOAD) begin
                                next_state = LD;
                        end
                        else if(ins == `STORE) begin
                                next_state = ST;
                        end
                        else if(ins == `BRANCH || opcode[5:0] == 6'b000010) begin
                                next_state = IF;
                        end
                        else begin
                                next_state = WB;
                        end
                    end
            LD:
                begin
                        if(Mem_Req_Ready) begin
                                next_state = RDW;
                        end
                        else begin
                                next_state = LD;
                        end
                    end
            ST:
                begin
                        if(Mem_Req_Ready) begin
                                next_state = IF;
                        end
                        else begin
                                next_state = ST;
                        end
                    end
            RDW:
                begin
                        if(Read_data_Valid) begin
                                next_state = WB;
                        end
                        else begin
```

```
182                     end
183                 RDW:
184                     begin
185                                 if(Read_data_Valid) begin
186                                         next_state = WB;
187                                 end
188                                 else begin
189                                         next_state = RDW;
190                                 end
191                     end
192                 WB:
193                     next_state = IF;
194                 default:
195                     next_state = INIT;
196         endcase
197 end
```

此外，我还注意到了一个细节：在 INIT 状态下，Inst_Ready 和 Read_data_Read 信号都要拉高，PPT 中所说是“避免对复位释放后产生影响”（如下图），根据我的理解，这应当是由于 rst==1 时真实内存会释放一些无用数据，若 CPU 不接受，则无用数据将会一直占据内存端口，等到 rst==0 时 CPU 将会读出无用数据而非所需数据（指令）。

最后是 FSM 第三段的代码。第三段描述主要工作在于确定 output 信号，这里我选择采用组合逻辑，仍然是根据 ppt 上的 output 信号说明，在相应状态下拉高相应的 output 信号。

这里有一个比较需要考虑的点：在其它状态下，这些 output 信号是否要拉高？事实上，当且仅当我们真正需要实现该信号对应的功能时，才需要将其拉高

——因为真实内存的具体实现方式我们是不了解的，这样考虑有助于减少错误的可能。

```
198 //The output decided by fsm
199 assign Inst_Req_Valid = (current_state == IF);//only when we want to get next ins do we send valid
200 assign Inst_Ready = (current_state == INIT || current_state == IW);
201 assign MemRead = (current_state == LD);
202 assign MemWrite = (current_state == ST);
203 assign Read_data_Ready = (current_state == RDW || current_state == INIT);
```

考虑握手机制，我们向内存发出请求后必须等到取指或者取数握手成功才能获取所需信号——这是与理想内存最大的不同点，而且注意到握手机制下内存的 Instruction 以及 Read_data 端口输出的信号不再一直是我们所需信号，因此需要 Instruction_valid 和 Read_data_valid 寄存器存放正在使用的指令或数据（这里解释了前文我声明这两个 reg 变量的用意），它们在时钟上升沿、握手成功的时候都会从真实内存的 Instruction 以及 Read_data 端口获取信号传入 CPU 进行处理。其余情况，这两个有效指令与有效数据寄存器均不变，不论真实内存的指令和数据端口如何变化，于是 CPU 得以稳定处理所取指令以及数据。

于是，所有单周期 CPU 中使用的 Instruction 以及 Read_data 信号均需要修改为 Instruction_valid 和 Read_data_valid 信号。

```
358 //shake hands bhv--whether renew the Instruction
359 always @(posedge clk) begin
360                 if(Inst_Ready && Inst_Valid) begin
361                         Instruction_valid <= Instruction;//shake hands successfully, get instruction
362                 end
363                 else begin
364                         Instruction_valid <= Instruction_valid;//failed,remain the instruction
365                 end
366         end
367
368 //shake hands bhv--whether renew the Read_data
369 always @(posedge clk) begin
370                 if(Read_data_Ready && Read_data_Valid) begin
371                         Read_data_valid <= Read_data;//shake hands successfully, get instruction
372                 end
373                 else begin
374                         Read_data_valid <= Read_data_valid;//failed,remain the instruction
375                 end
376         end
377
```

关于 PC 的变化，我们的处理如下图。当 rst 复位信号到来时复位 PC，当取出的是 NOP 指令并且当前状态为 ID 时让 PC=PC+4，因为我们即将进入下一个指令周期，而 NOP 指令的内容恰是什么也不做。当状态机处于 EX 执行阶段的时候，要根据不同指令对 PC 的改变来修改 PC 的值，这里如果是 Jump 控制信号拉高，说明接下来即将跳转执行，而 PC 也应相应地赋值成我们所计算出来的跳转地址 Jaddr。如果不是跳转执行，则判断下一条指令是否是分支执行，如果是，那么 PCSrc 拉高，PC 赋值为所计算出的分支地址 PC_BRANCH。如果都不是，那么应当正常地取出下一条顺序指令地址，即 PC=PC+4。由于多周期时序，我们还必须增加一个 PC 不变的情况，用以在其它情况下保持 PC 的值。（尽管这不是必要的，还是可以让代码看得更清晰）

```
365  //Clk posedge behavior:get the next PC
366  always @(posedge clk) begin
367                  if(rst) begin
368                          PC <= 32'b0;//reset PC
369                  end
370                  else if(Instruction_valid == 32'b0 && current_state == ID) begin
371                          PC <= PC_next;//NOP
372                  end
373          else if(current_state == EX) begin
374              PC <= (Jump? J_addr: (PCSrc? PC_BRANCH: PC_next));//prepare to fetch next instruction
375                  end
376                  else begin
377                          PC <= PC;
378                  end
379          end
380
```

此外，我们还需要对其它控制信号进行一些适用性修改。

```
267  assign RF_wen = ~(ins == `BRANCH || opcode[5:3] == 3'b101 || opcode == 6'b000010
268                 || (ins == `JUMPR && func == 6'b001000) || (opcode == 6'b000000 && func[5:0] == 6'b001011 && Zero == 1)
269                                 || (opcode == 6'b000000 && func[5:0] == 6'b001010 && Zero == 0) || current_state != WB);//must be in state of WB
270  assign RF_waddr = (opcode == 6'b000011)? 31:((opcode == 6'b000000 && func == 6'b001000)? 0:((RegDst)? rd:rt));//'jal' needs REG_31,'jr' needs REG_0,others stay still
271  assign RF_wdata = (ins == `CAL || ins == `CALI)? ALU_res:
272                                          ((ins == `SHIFT || ins == `SHIFTV)? shifter_res:
273                                          ((opcode == 6'b000011 || (ins == `JUMPR && func[0] == 1'b1))? PC_old + 8:
274                                          ((opcode[5:3] == 3'b100)? load_res:
275                                          ((opcode == 6'b000000 && func[5:1] == 5'b00101)? rdata1:lui_extend))));
276  //Shifter ports
```

如上图，由于我们在执行阶段修改了 PC 的值，为了执行 jal 和 jalr 指令，我们又需要将原先的 PC+8 存储在相应寄存器中（reg 31 or rd），因此引入

PC_old 进行旧 PC 的储存，并修改 RF_wdata 中的 PC+8 为 PC_old+8，以便能够写入正确的地址。

另外，RF_wen 也需要进行修改。由于我们的状态机只可能在 WB 状态下产生写回操作，还需要增添一个 current_state 是否为 WB 的判断。

这里有一个非常细节的地方！jr 指令默认写入 0 号寄存器（本实验是这么要求的，但是指令手册并没有说明）因此 RF_waddr 需要加上 jr 指令的判断以确定写回寄存器是否是 0 号寄存器。一开始我并没有意料到 jr 指令需要写回 0 号寄存器，后来 debug 的时候无论如何都无法发现错误所在。好在后来有同学（匿名，不知道是谁）提醒了我。

最后是考虑 PC_old 的赋值逻辑。

```
401 //old PC
402         always @(posedge clk) begin
403                 if(current_state == IF) begin
404                         PC_old <= PC;
405                 end
406                 else begin
407                         PC_old <= PC_old;
408                 end
409         end
410
```

如上图，我们选择在进入 IF 周期的时候就将 PC 旧值储存下来（事实上只要在 EX 之前存储就可以，因为 PC_old 的设计仅仅针对 jal 和 jalr，它们均是在 EX 阶段修改了 PC 值），其他情况 PC_old 均保持不变。这样，每进入一个指令周期，我们都会在 IF 阶段存储该指令的 PC 值。

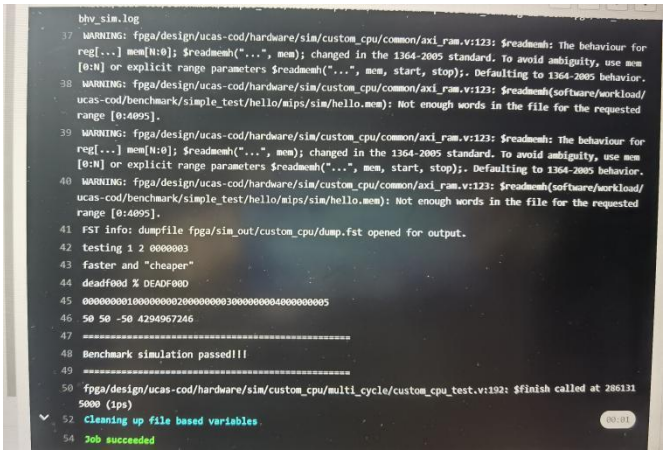
到此，我们已经成功完成了一个多周期的 CPU，接下来是软件部分的编写。

```
250 puts(const char *s)
251 {
252         int i = 0;
253         while(s[i]){
254                 while(*((volatile char*)uart + UART_STATUS) & UART_TX_FIFO_FULL){
255                         ;
256                 }
257                 *((volatile char*)uart + UART_TX_FIFO) = s[i++];
258         }
259         return i;
260         //TODO: Add your driver code here
261 }
262
```

```
46 #define UART_TX_FIFO            0x04
47 #define UART_STATUS             0x08
48 #define UART_TX_FIFO_FULL       (1 << 3)
49
50 volatile unsigned int *uart = (void *)0x60000000;
```

如上图，我们为实现打印字符串功能，先利用基址 uart 找到 0x60000008 地址的寄存器（队列状态寄存器），根据里面的内容判断队列是否为满。只有当队列不满的时候（内层 while 循环结束）才执行入队操作，将字符写入发送队列入口寄存器(地址为 0x60000004),并 i++进入下一个外层 while 循环。当 s[i]为 NULL 即字符串已经打印完毕的时候，我们退出循环，返回字符串长度 i（此时 i 就是字符串长度）。

下面是打印成功的照片：



最后我们添加性能计数器。

```
428  //The number of clk cycle
429  reg [31:0] cycle_cnt;
430  always @(posedge clk) begin
431          if(rst) begin
432                  cycle_cnt <= 32'd0;
433          end
434          else begin
435                  cycle_cnt <= cycle_cnt + 32'd1;
436          end
437  end
438  assign cpu_perf_cnt_0 = cycle_cnt;
439
440  reg [31:0] ins_cnt;
441  always @(posedge clk) begin
442          if(rst) begin
443                  ins_cnt <= 32'd0;
444          end
445          else if(Inst_Ready && Inst_Valid) begin
446                  ins_cnt <= ins_cnt + 1;
447          end
448          else begin
449                  ins_cnt <= ins_cnt;
450          end
451  end
452
453          assign cpu_perf_cnt_1 = ins_cnt;
454  ////////////////////////////////////
455  endmodule
```

如上图，我们写了两个性能计数器：一个统计时钟周期数，另一个统计指令数。为统计前者，我们在复位信号拉低的每个时钟上升沿给时钟周期数寄存器加 1，并用 assign 语句将其连到 0 号性能计数器上；为统计后者，我们在每次取指令握手成功后都给指令数寄存器加 1，并用 assign 语句将其连到 1 号性能计数器上。

接下来是对软件的修改。

perf_cnt.h 264 Bytes

```
1
2  #ifndef __PERF_CNT__
3  #define __PERF_CNT__
4
5  #ifdef __cplusplus
6  extern "C" {
7  #endif
8
9  typedef struct Result {
10         int pass;
11         unsigned long msec;
12         unsigned long ins;
13         ////////////////////
14 } Result;
15
16 void bench_prepare(Result *res);
17 void bench_done(Result *res);
18
19 #endif
```

在 perf_cnt.h 中，我们对 Result 结构体进行修改，增加 ins 变量（对应指令数计数器）。原先的 mesc 变量则对应时钟周期计数器。

```c
#include "perf_cnt.h"
////////////////////////////////////
unsigned long _uptime() {
  // TODO [COD]
  //   You can use this function to access performance counter related with time or cycle.
  volatile unsigned long *Cycle_cnt = (volatile unsigned long*)0x60010000;
  return *Cycle_cnt;
}

unsigned long _upinsnum() {
  // TODO [COD]
  //   You can use this function to access performance counter related with time or cycle.
  volatile unsigned long *ins_cnt = (volatile unsigned long*)0x60010008;
  return *ins_cnt;
}
//////////////////////////////

void bench_prepare(Result *res) {
  // TODO [COD]
  //   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->ins = _upinsnum();
  ////////////////////////////////
}

void bench_done(Result *res) {
  // TODO [COD]
  //  Add postprocess code, record performance counters' current states.
  res->msec = _uptime() - res->msec;
  res->ins = _upinsnum() - res->ins;
  ////////////////////////////////////
}

```

在 perf_cnt.c 中，我们利用 up_time 和 up_insnum 函数分别统计时钟周期数和指令数（根据计数器地址访问其中内容）。并且利用 bench_prepare 函数来记录初始数值，以便在 bench_done 函数中统计改变数目。

```
66  int main() {
67    int pass = 1;
68
69    _Static_assert(ARR_SIZE(benchmarks) > 0, "non benchmark");
70
71    for (int i = 0; i < ARR_SIZE(benchmarks); i ++) {
72      Benchmark *bench = &benchmarks[i];
73      current = bench;
74      setting = &bench->settings[SETTING];
75      const char *msg = bench_check(bench);
76      printk("[%s] %s: ", bench->name, bench->desc);
77      if (msg != NULL) {
78        printk("Ignored %s\n", msg);//**************************reference of printk
79      } else {
80        unsigned long msec = ULONG_MAX;
81        unsigned long ins = ULONG_MAX;
82        ////////////////////////////////
83        int succ = 1;
84        for (int i = 0; i < REPEAT; i ++) {
85          Result res;
86          run_once(bench, &res);
87          printk(res.pass ? "*" : "X");
88          succ &= res.pass;
89          if (res.msec < msec) msec = res.msec;
90          if (res.ins < ins) ins = res.ins;
91          ////////////////////////
92        }
93
92        }
93
94        if (succ) printk(" Passed.\n");
95        else printk(" Failed.\n");
96
97        pass &= succ;
98
99        // TODO [COD]
100       //   A benchmark is finished here, you can use printk to output some informantion.
101       //   `msec' is intended indicate the time (or cycle),
102       //   you can ignore according to your performance counters semantics.
103       printk("The number of clk cycles is %u\n",msec);
104       printk("The number of instructions is %u\n",ins);
105       ////////////////
106     }
107   }
108
```

在 bench.c 中，我们首先在打////////的地方添加了对称的操作，即

```
unsigned long msec = ULONG_MAX;
unsigned long ins = ULONG_MAX;
```

和

```
if (res.msec < msec) msec = res.msec;
if (res.ins < ins) ins = res.ins;
```

在最后，我们利用 printk 函数打印性能计数器的值。printk 的格式我是参考上面的 printk 的代码的。

```
[sieve] Eratosthenes sieve: * Passed.
The number of clk cycles is 728393
The number of instructions is 16478
benchmark finished
time 18.51ms
reset: before MMIO access...
reset: MMIO accessed
./software/workload/ucas-cod/benchmark/simple_test/microbench/mips/elf/sieve passed
Hit good trap
Launching ssort benchmark...
tggetattr: Inappropriate ioctl for device
reset: before MMIO access...
reset: MMIO accessed
axi_firewall_unblock: checking firewall status...
axi_firewall_unblock: firewall status: 00000000
main: before DDR accessing...
main: DDR accessed...
reset: before MMIO access...
reset: MMIO accessed
[ssort] Suffix sort: * Passed.
The number of clk cycles is 32132154
The number of instructions is 728289
benchmark finished
time 1024.27ms
reset: before MMIO access...
reset: MMIO accessed
./software/workload/ucas-cod/benchmark/simple_test/microbench/mips/elf/ssort passed
Hit good trap
pass 9 / 9
Cleaning up project directory and file based variables
Job succeeded
```

如上图，这里展现了最终的性能计数器输出，可以看到，它们成功统计了时钟周期数以及指令数。发现指令数远小于时钟周期数，这与我们的预期是相符合的（因为真实内存访存速度慢，而且这是一个非流水线实现的多周期 CPU）。

二、 实验过程中遇到的问题、对问题的思考过程及解决方法（比如 RTL 代码中出现的逻辑 bug,逻辑仿真和 FPGA 调试过程中的难点等）

硬件实现方面，遇到的问题主要在于控制信号的多周期适用性修改，尤其是三大 RF 信号（因为本代码是以单周期 CPU 代码为主体的，由于多周期和状态

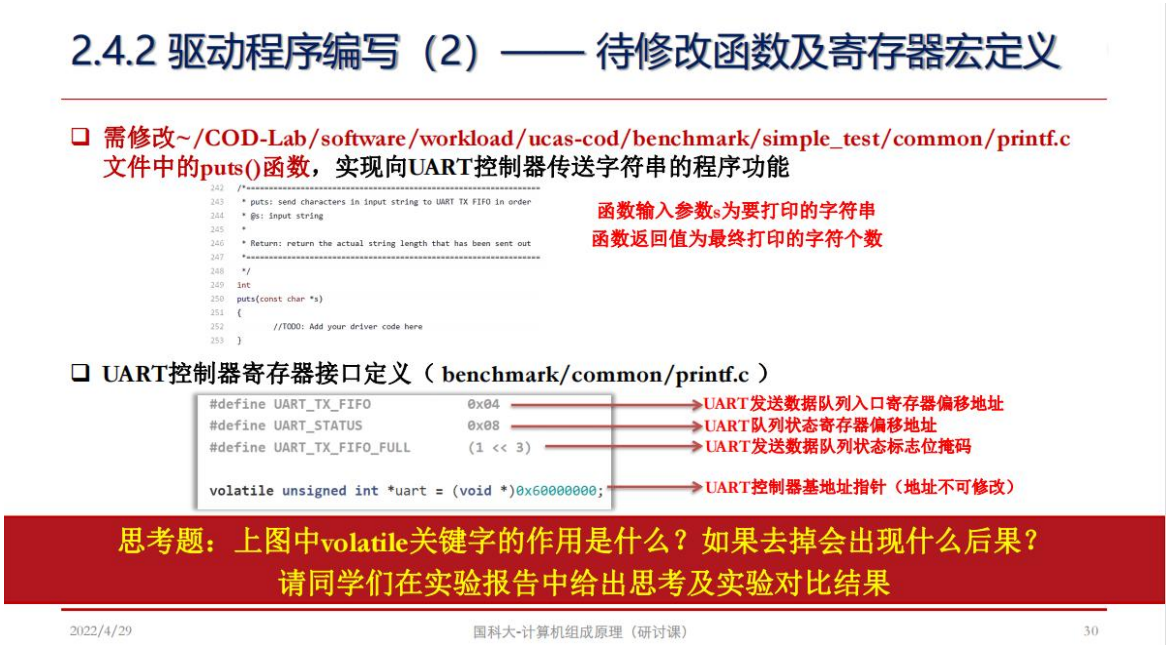
机的特性，我们必须对其进行适用性修改）！

由于我们在执行阶段修改了 PC 的值，为了执行 jal 和 jalr 指令，我们又需要将原先的 PC+8 存储在相应寄存器中（reg 31 or rd），因此引入 PC_old 进行旧 PC 的储存，并修改 RF_wdata 中的 PC+8 为 PC_old+8，以便能够写入正确的地址。

另外，由于我们的状态机只可能在 WB 状态下产生写回操作，还需要对 RF_wen 增添一个 current_state 是否为 WB 的判断。

这里有一个非常细节的地方：jr 指令默认写入 0 号寄存器（本实验是这么要求的，但是指令手册并没有说明！）因此 RF_waddr 需要加上 jr 指令的判断以确定写回寄存器是否是 0 号寄存器。一开始我并没有意料到 jr 指令需要写回 0 号寄存器，后来 debug 的时候无论如何都无法发现错误所在（对应波形图和错误信号五花八门、千奇百怪）。好在后来有同学（匿名，不知道是谁）提醒了我。

三、对讲义中思考题（如有）的理解和回答



volatile 关键词的本意是“多变的”，作用在于避免编译器对其进行优化（由于访存操作的时间成本较大，编译器为避免重复访存，会将对同一地址的访存操作次数减少到 1，尽管该地址所存的数值是变化的，这样就无法实现连续访存），以便我们能够时刻跟踪内存地址所存数值的变化情况。

如果去掉，那么访存读数将固定，无法读到最新的数值，输出也将出错。

四、在课后，你花费了大约___12___小时完成此次实验。

硬件耗时 10h，软件耗时 2h。（忽视 serve 云平台运行时间）

总共耗时 24h。（算上 serve 云平台运行时间）

五、对于此次实验的心得、感受和建议（比如实验是否过于简单或复杂，是否缺少了某些你认为重要的信息或参考资料，对实验项目的建议，对提供帮助的同学的感谢，以及其他想与任课老师交流的内容等）

希望下次能早一点将比较全的课件放到课程网站上，此次实验软件部分我对照 v6 版本课件竟无从下手，后来才发现 v8 课件加入了软件部分的具体操作指导，而 v6 里是缺失的。

整个实验是不算非常复杂的，因为硬件部分的代码都是基于单周期处理器的，而软件部分只要弄清楚原理则并不复杂。关键在于一些细节！尤其是控制信号的修改以及 PC 的跳转等，这些细节处的 bug 很难发现，即便利用波形图也难以看出来。我就因为 jr 不知道要写到 0 号寄存器而卡住很久（事实上指令手册和课件都没有说 jr 要写到 0 号寄存器，这属于缺失的信息）。

感谢 19 级的高学长，在多周期的 PC_old 修改上给了我帮助。