中国科学院大学计算机组成原理实验课

实 验 报 告

学号： 2020K8009926006 姓名： 游昆霖 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

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）。

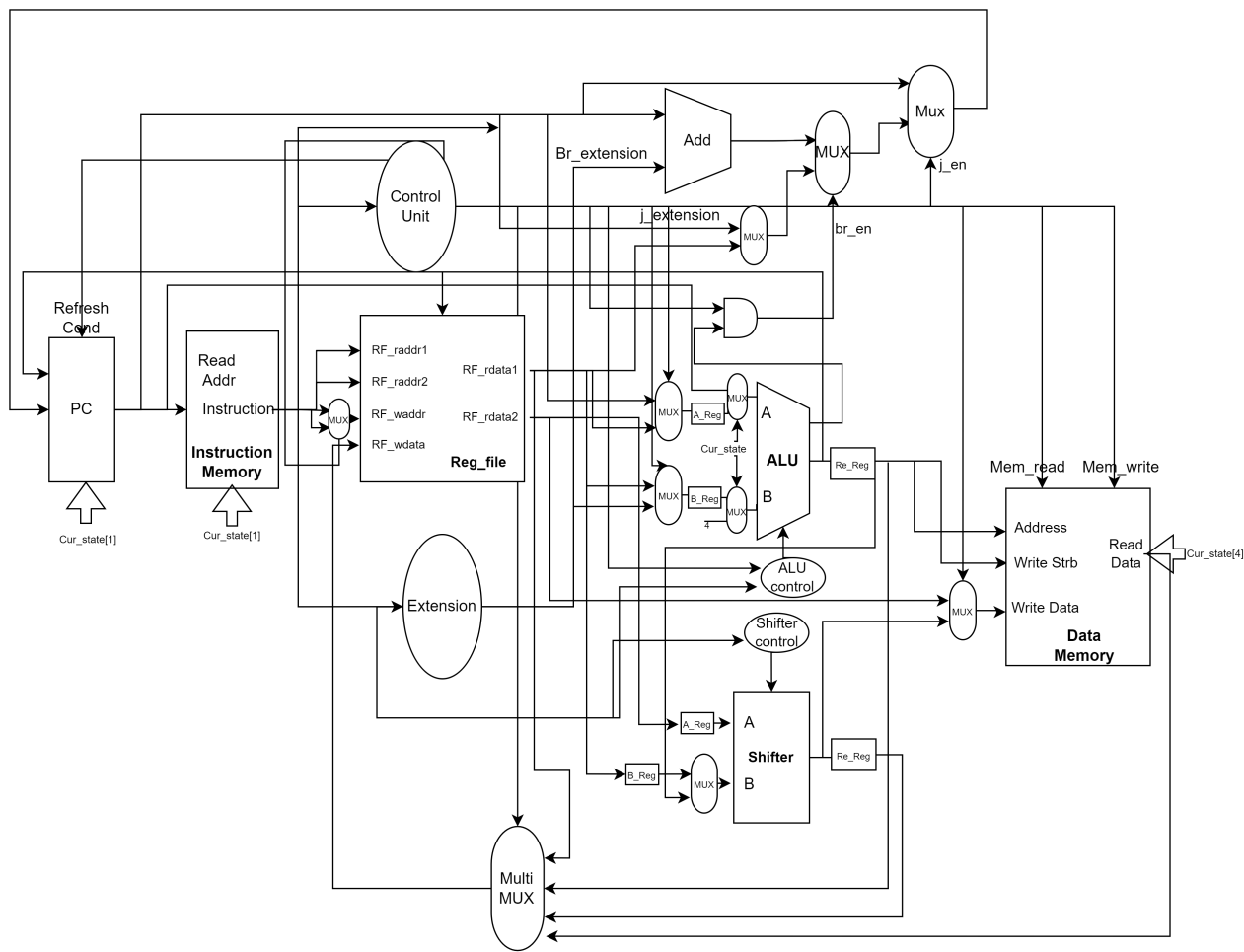
注3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

1. 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）

\* 说明：MIPS定制处理器基于prj2实现的多周期处理器进行修改，其中reg\_file.v alu.v shifter.v可直接复用，cpu部分除状态机及相关部分代码外可直接复用，以下仅对修改部分和新增功能进行说明。

1. Custom\_cpu.v 代码说明

逻辑结构图：和prj2实现的多周期处理器结构基本一致。



1. 状态机部分

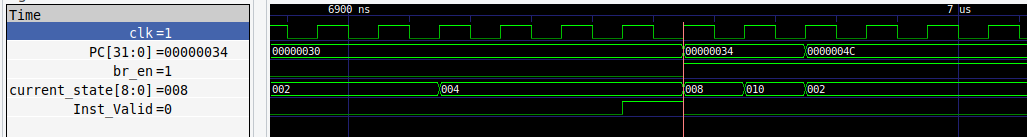
|  |
| --- |
| //define the state of machine by one-hot          localparam      RST     =9'b000000001,                          IF      =9'b000000010,                          IW      =9'b000000100,                          ID      =9'b000001000,                          EX      =9'b000010000,                          LD      =9'b000100000,                          ST      =9'b001000000,                          RDW     =9'b010000000,                          WB      =9'b100000000;  //define signal for State Machine          //set 1 to Inst\_Ready and Read\_data\_Ready to prevent Error          assign Inst\_Req\_Valid   = current\_state[1] ;    //IF          assign Inst\_Ready       = current\_state[0] | current\_state[2] ;//RST IW          assign Read\_data\_Ready  = current\_state[0] | current\_state[7] ;//RST RDW  //State Machine          //Part I:      always @ (posedge clk) begin         if(rst) begin             current\_state <= RST;         end         else begin             current\_state <= next\_state;         end      end          //Part II: Combinatorial logic          always @ (\*)          begin                  case(current\_state)                          RST: begin                                  next\_state = IF ;                          end                          IF:     begin                                  if (Inst\_Req\_Ready)                                          next\_state = IW;                                  else                                          next\_state = IF;                          end                          IW: begin                                  if (Inst\_Valid)                                          next\_state = ID;                                  else                                          next\_state = IW;                          end                          //NOP: Instr = 32'b0                          ID: begin                                  if(~|Instruction\_tmp)                                          next\_state = IF ;                                  else                                          next\_state = EX ;                          end                          //EX->IF:                                  //REGIMM        2: opcode[5:0]=000001                                  //I\_branch      4: opcode[5:2]=0001                                  //J             1: opcode[5:0]=000010                          //EX->WB                                  //R\_Type        18: opcode[5:0]=6'b0 include JALR,JR //JR: rs->GPR[0]                                  //I\_calc        7: opcode[5:3]=001 include LUI                                  //JAL           1: opcode[5:0]=000011                          //EX->LD                                  //load          7: opcode[5:3]=100                          //EX->ST                                  //store         5: opcode[5:3]=101                          EX: begin                                  if((opcode[5:0]==6'b000001) | (opcode[5:2]==4'b0001) | (opcode[5:0]==6'b000010) )                                          next\_state = IF;                                  else if((~|opcode)| (opcode[5:3]==3'b001) | (opcode[5:0]==6'b000011))                                          next\_state = WB;                                  else if(opcode[5:3]==3'b100)                                          next\_state = LD;                                  else if(opcode[5:3]==3'b101)                                          next\_state = ST;                                  else                                          next\_state = RST;                          end                          LD: begin                                  if(Mem\_Req\_Ready)                                          next\_state = RDW;                                  else                                          next\_state = LD;                          end                          ST: begin                                  if(Mem\_Req\_Ready)                                          next\_state = IF;                                  else                                          next\_state = ST;                          end                          RDW: begin                                  if(Read\_data\_Valid)                                          next\_state = WB;                                  else                                          next\_state = RDW;                          end                          WB: begin                                  next\_state = IF;                          end                          default: begin                                  next\_state = RST;                          end                  endcase          end            //Part III: deal with output          //one always module to deal one reg          always @ (posedge clk) begin                  if(Inst\_Ready & Inst\_Valid) //after response                          Instruction\_tmp <= Instruction ;          end            always @ (posedge clk) begin                  if(Read\_data\_Ready & Read\_data\_Valid)                          Read\_data\_tmp <=Read\_data ;          end            always @ (posedge clk) begin                  if(current\_state[1])//IF                          PC\_tmp <= PC;          end            always @ (posedge clk) begin                  if (rst) begin                          PC <= 32'b0;                  end                  else if(current\_state[2] & Inst\_Valid & ~rst) begin                          PC <= ALU\_Result; //PC+4                          //IW, consider the cycle before ID                  end                  else if(current\_state[4] & ~rst)begin                          if(br\_en)                                  PC   <= ALU\_Result\_tmp;                                  //PC <= br\_tar;                          else if(j\_en)                                  PC <= j\_tar;                          else                                  PC <= PC;                          //PC <= br\_en ? br\_tar : j\_en ? j\_tar : PC ;                          //j or br OP refresh in next IF, judge by EX                          //note that the default result is PC4                  end          end |

状态机第一部分改变了状态复位值，设置从RST状态进行释放。

状态机第二部分相对prj2多周期处理器新增了取指和访存部分针对真实内存的等待阶段，并添加了相应握手信号。同时考虑到写内存只要求内存闲时即可写；但是读内存在内存闲时完成握手后，还需要等待内存返回数据，故将读写内存状态进行分离，并添加RDW读等待状态。

状态机第三部分针对握手信号和状态的改变进行了调整。当相应握手信号均为高时，将Instruction和Read\_data进行寄存，PC在IF阶段进行寄存。同时在一次指令周期中，PC在IW阶段进行默认更新(+4)，且考虑到IW可能有多拍，因此添加Inst\_Valid使得PC只在IW阶段的最后一拍更新；对于成功的跳转指令，PC将在EX阶段进行额外的一次更新。

相应波形说明：



光标所指位置为IW（004）阶段的最后一拍，PC完成顺序更新；由于该条指令为条件跳转指令，在EX（010）阶段PC再次更新为跳转值。

1. 访存相应控制信号修改

|  |
| --- |
| //mem control          assign MemRead  = current\_state[5];  //LD //100          assign MemWrite = current\_state[6];  //ST //101 |

由于只有LOAD类型指令可进入LD阶段，只有STORE类型指令可进入ST阶段，将控制信号赋值为对应阶段即可。

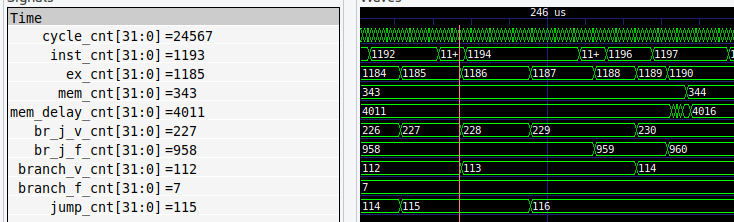
1. 性能统计相关信号

|  |
| --- |
| //counter          //set specific name to distinguish: (v means victory, f means failure)                  //cycle, ins, mem(num of visit mem), delay,                  //branch\_v, branch\_f,jump\_v,jump\_f            reg [31:0]      cycle\_cnt;              //count the num of clock          reg [31:0]      inst\_cnt;               //count the num of obtained instruction          reg [31:0]      ex\_cnt;                 //count the num of execution          reg [31:0]      mem\_cnt;                //count the num of mem visit          reg [31:0]      mem\_delay\_cnt;          //count the num of delay of mem,including read and write          reg [31:0]      br\_j\_v\_cnt;             //count the num of PC not refresh for PC4          reg [31:0]      br\_j\_f\_cnt;             //count the num of PC refresh for PC4, including instr not br or j          reg [31:0]      branch\_v\_cnt;           //count the num of successful branch          reg [31:0]      branch\_f\_cnt;           //count the num of failed branch          reg [31:0]      jump\_cnt;               //count the num of jump (always success)            assign cpu\_perf\_cnt\_0   = cycle\_cnt;          assign cpu\_perf\_cnt\_1   = inst\_cnt;          assign cpu\_perf\_cnt\_2   = ex\_cnt;          assign cpu\_perf\_cnt\_3   = mem\_cnt;          assign cpu\_perf\_cnt\_4   = mem\_delay\_cnt;          assign cpu\_perf\_cnt\_5   = br\_j\_v\_cnt;          assign cpu\_perf\_cnt\_6   = br\_j\_f\_cnt;          assign cpu\_perf\_cnt\_7   = branch\_v\_cnt;          assign cpu\_perf\_cnt\_8   = branch\_f\_cnt;          assign cpu\_perf\_cnt\_9   = jump\_cnt;            always @ (posedge clk) begin                  if(rst)                          cycle\_cnt <= 32'b0;                  else                          cycle\_cnt <= cycle\_cnt + 32'b1;          end            always @ (posedge clk) begin                  if(rst)                          inst\_cnt <= 32'b0;                  else if(current\_state[1]) //IF                          inst\_cnt <= inst\_cnt + 32'b1;          end            always @ (posedge clk) begin                  if(rst)                          ex\_cnt <= 32'b0;                  else if(current\_state[4])//EX                          ex\_cnt <= ex\_cnt + 32'b1;          end            always @ (posedge clk) begin                  if(rst)                          mem\_cnt <= 32'b0;                  else if(current\_state[5] | current\_state[6]) //ST LD                          mem\_cnt <= mem\_cnt + 32'b1;          end            always @ (posedge clk) begin                  if(rst)                          mem\_delay\_cnt <= 32'b0;                  else if(( (current\_state[5]|current\_state[6]) & ~Mem\_Req\_Ready )|( current\_state[7] & ~Read\_data\_Valid )) //LD ST ~Mem\_Req\_Ready -- RDW ~Read\_data\_Valid                          mem\_delay\_cnt <= mem\_delay\_cnt + 32'b1;          end          //state = EX 4          //Branch 4 :                    opcode[5:2]=0001          //REGIMM 2 :                    opcode[5:0]=000001          //jump   4 : J JAL              opcode[5:1] =00001                    //JR JALR             opcode[5:0] = 000000 & func[5:1]=00100          always @ (posedge clk) begin                  if(rst)                          br\_j\_v\_cnt <= 32'b0;                  else if(current\_state[4] & ( ((opcode[5:2]==4'b0001 | opcode[5:0]==6'b000001) & br\_en) | (opcode[5:1]==5'b00001 | (opcode[5:0]==6'b000000 & func[5:1]==5'b00100)) ))                          br\_j\_v\_cnt <= br\_j\_v\_cnt + 32'b1;  //branch\_suc or jump          end            always @ (posedge clk) begin                  if(rst)                          br\_j\_f\_cnt <= 32'b0;                  else if(current\_state[4] & ~( ((opcode[5:2]==4'b0001 | opcode[5:0]==6'b000001) & br\_en) | (opcode[5:1]==5'b00001 | (opcode[5:0]==6'b000000 & func[5:1]==5'b00100)) ))                          br\_j\_f\_cnt <= br\_j\_f\_cnt + 32'b1;  //branch\_fail          end            always @ (posedge clk) begin                  if(rst)                          branch\_v\_cnt <= 32'b0;                  else if(current\_state[4] & ((opcode[5:2]==4'b0001 | opcode[5:0]==6'b000001) & br\_en))                          branch\_v\_cnt <= branch\_v\_cnt + 32'b1;          end            always @ (posedge clk) begin                  if(rst)                          branch\_f\_cnt <= 32'b0;                  else if(current\_state[4] & ((opcode[5:2]==4'b0001 | opcode[5:0]==6'b000001) & ~br\_en))                          branch\_f\_cnt <= branch\_f\_cnt + 32'b1;          end            always @ (posedge clk) begin                  if(rst)                          jump\_cnt <= 32'b0;                  else if(current\_state[4] & (opcode[5:1]==5'b00001 | (opcode[5:0]==6'b000000 & func[5:1]==5'b00100)))                          jump\_cnt <= jump\_cnt + 32'b1;          end |

本次实验设置了十个性能统计计数器，分别为cycle\_cnt（时钟周期数统计）、 inst\_cnt（取得指令数统计）、ex\_cnt（执行指令数统计）、 mem\_cnt（访存次数统计）、mem\_delay\_cnt（访存延迟统计） br\_j\_v\_cnt（PC跳转次数统计）、 br\_j\_f\_cnt（PC顺序更新次数统计）、branch\_v\_cnt（条件跳转成功次数统计）、 branch\_f\_cnt（条件跳转失败次数统计）、 jump\_cnt（无条件跳转次数统计）。

以上计数器在复位阶段置零，并在满足对应条件的时钟上升沿更新。连接计数器和对应输出端口即可。

相应波形说明：



某一时刻的性能计数器统计值，其中ex\_cnt = br\_j\_v\_cnt + br\_j\_f\_cnt 、 br\_j\_v\_cnt = branch\_v\_cnt + jump\_cnt，均符合预期。

1. 外设控制器访问 puts.c代码说明

|  |
| --- |
| /\*=================================================================   \* puts: send characters in input string to UART TX FIFO in order   \* @s: input string   \*   \* Return: return the actual string length that has been sent out   \*=================================================================   \*/  int  puts(const char \*s)  {          //TODO: Add your driver code here          //UART control unit:                  //UART\_TX\_FIFO : offset from TX to uart                  //UART\_STATUS  : offset from STAT to uart          //wait until STAT is not full, then send a characters          //use volatile to provide stable access to a particular address          /\*          \* OPTIMIZE:          \* reverse cond of while to cond of if, save while inside          \* (char \*)(base+1) is equal to (char \*)base + 4, save shifter          \*/          int i=0;          while (s[i]){                  if (!( \*((volatile char \*)uart + UART\_STATUS) & UART\_TX\_FIFO\_FULL))                          \*((volatile char \*)uart + UART\_TX\_FIFO)=s[i++];          }          return i;  } |

对于输入的字符串s，需将其逐位打印，并且返回成功打印的字符数，因此设置i作为字符数组下标进行循环枚举。

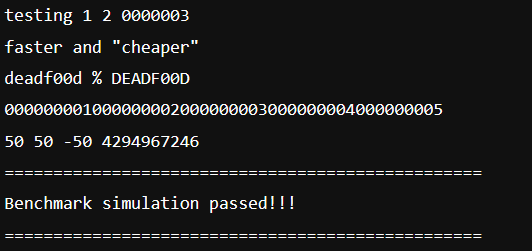
由于TX FIFO寄存器只可写不可读，且需要更改32位中的最低8位，因此考虑将uart从int型指针强制类型转化为char型指针，使其指向的数据为1字节。同时添加volatile关键字阻止编译器对访问变量的代码优化，保证系统总是从其所在内存读取数据，从而稳定访问该地址。

将类型转换后的uart加上寄存器偏移地址即得TX FIFO和STAT\_REG的绝对地址。取对应地址的值就可得到TX FIFO和STAT\_REG的最低字节。

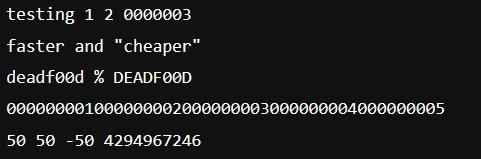
将STAT\_REG的最低字节和UART\_TX\_FIFO\_TULL（1000）进行按位与即可得到最低第3位为对应值，其余位为0的数。将其取反后作为条件，即可保证只有STAT\_REG最低第3bit为0，才可打印字符s[i].

将TX FIFO的最低字节赋值为s[i]，即实现了将待发送的8bit字符写入TX FIFO寄存器。

仿真运行结果：



上板运行结果：



1. 性能统计相关软件代码说明
2. Perf\_cnt.h 说明

|  |
| --- |
| typedef struct Result {          int pass;          unsigned long msec;             //cycle\_cnt          unsigned long inst\_cnt;          unsigned long ex\_cnt;          unsigned long mem\_cnt;          unsigned long mem\_delay\_cnt;          unsigned long br\_j\_v\_cnt;          unsigned long br\_j\_f\_cnt;          unsigned long branch\_v\_cnt;          unsigned long branch\_f\_cnt;          unsigned long jump\_cnt;  } Result; |

在结构体中添加性能计数器相关统计变量。

1. Perf\_cnt.c 说明

|  |
| --- |
| #include "perf\_cnt.h"  //The addr of specific perf\_cnt:          //perf\_cnt\_0  ->  0x60010000          //perf\_cnt\_1  ->  0x60010008          //perf\_cnt\_2  ->  0x60011000          //...  //The following 10 func is side-by-side  unsigned long \_uptime() {    // TODO [COD]    //   You can use this function to access performance counter related with time or cycle.          unsigned long \*msec\_addr;          msec\_addr = (unsigned long \*)0x60010000;          return \*msec\_addr;  }  unsigned long \_inst\_freq() {          unsigned long \*inst\_cnt\_addr;          inst\_cnt\_addr = (unsigned long \*)0x60010008;          return \*inst\_cnt\_addr;  }  unsigned long \_ex\_freq() {          unsigned long \*ex\_cnt\_addr;          ex\_cnt\_addr = (unsigned long \*)0x60011000;          return \*ex\_cnt\_addr;  }  unsigned long \_mem\_freq() {          unsigned long \*mem\_cnt\_addr;          mem\_cnt\_addr = (unsigned long \*)0x60011008;          return \*mem\_cnt\_addr;  }  unsigned long \_mem\_delay\_freq() {          unsigned long \*mem\_delay\_cnt\_addr;          mem\_delay\_cnt\_addr = (unsigned long \*)0x60012000;          return \*mem\_delay\_cnt\_addr;  }  unsigned long \_br\_j\_v\_freq() {          unsigned long \*br\_j\_v\_cnt\_addr;          br\_j\_v\_cnt\_addr = (unsigned long \*)0x60012008;          return \*br\_j\_v\_cnt\_addr;  }  unsigned long \_br\_j\_f\_freq() {          unsigned long \*br\_j\_f\_cnt\_addr;          br\_j\_f\_cnt\_addr = (unsigned long \*)0x60013000;          return \*br\_j\_f\_cnt\_addr;  }  unsigned long \_branch\_v\_freq() {          unsigned long \*branch\_v\_cnt\_addr;          branch\_v\_cnt\_addr = (unsigned long \*)0x60013008;          return \*branch\_v\_cnt\_addr;  }  unsigned long \_branch\_f\_freq() {          unsigned long \*branch\_f\_cnt\_addr;          branch\_f\_cnt\_addr = (unsigned long \*)0x60014000;          return \*branch\_f\_cnt\_addr;  }  unsigned long \_jump\_freq() {          unsigned long \*jump\_cnt\_addr;          jump\_cnt\_addr = (unsigned long \*)0x60014008;          return \*jump\_cnt\_addr;  }  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->inst\_cnt         = \_inst\_freq();    res->ex\_cnt           = \_ex\_freq();    res->mem\_cnt          = \_mem\_freq();    res->mem\_delay\_cnt    = \_mem\_delay\_freq();    res->br\_j\_v\_cnt       = \_br\_j\_v\_freq();    res->br\_j\_f\_cnt       = \_br\_j\_f\_freq();    res->branch\_v\_cnt     = \_branch\_v\_freq();    res->branch\_f\_cnt     = \_branch\_f\_freq();    res->jump\_cnt         = \_jump\_freq();  }  void bench\_done(Result \*res) {    // TODO [COD]    //  Add postprocess code, record performance counters' current states.    res->msec             = \_uptime()             - res->msec;    res->inst\_cnt         = \_inst\_freq()          - res->inst\_cnt;    res->ex\_cnt           = \_ex\_freq()            - res->ex\_cnt;    res->mem\_cnt          = \_mem\_freq()           - res->mem\_cnt;    res->mem\_delay\_cnt    = \_mem\_delay\_freq()     - res->mem\_delay\_cnt;    res->br\_j\_v\_cnt       = \_br\_j\_v\_freq()        - res->br\_j\_v\_cnt;    res->br\_j\_f\_cnt       = \_br\_j\_f\_freq()        - res->br\_j\_f\_cnt;    res->branch\_v\_cnt     = \_branch\_v\_freq()      - res->branch\_v\_cnt;    res->branch\_f\_cnt     = \_branch\_f\_freq()      - res->branch\_f\_cnt;    res->jump\_cnt         = \_jump\_freq()          - res->jump\_cnt;  } |

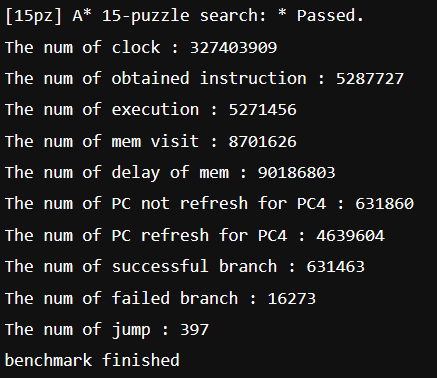
添加性能计数器相关函数，每个函数访问该性能计数器对应位置，返回其中的值。Bench\_prepare函数在运行benchmark前调用，将每个性能计数器的初始值存储于结构体变量中，Bench\_done函数在benchmark后调用，获取当前每个性能计数器的值，并减去存储于结构体中的初始值，即可得到该benchmark运行过程中的性能统计结果。

1. Bench.c 说明

|  |
| --- |
| for (int i = 0; i < ARR\_SIZE(benchmarks); i ++) {      Benchmark \*bench = &benchmarks[i];      current = bench;      setting = &bench->settings[SETTING];      const char \*msg = bench\_check(bench);      printk("[%s] %s: ", bench->name, bench->desc);      if (msg != NULL) {        printk("Ignored %s\n", msg);      } else {    unsigned long msec      = ULONG\_MAX;    unsigned long inst\_cnt    = ULONG\_MAX;    unsigned long ex\_cnt      = ULONG\_MAX;    unsigned long mem\_cnt     = ULONG\_MAX;    unsigned long mem\_delay\_cnt = ULONG\_MAX;    unsigned long br\_j\_v\_cnt    = ULONG\_MAX;    unsigned long br\_j\_f\_cnt    = ULONG\_MAX;    unsigned long branch\_v\_cnt  = ULONG\_MAX;    unsigned long branch\_f\_cnt  = ULONG\_MAX;    unsigned long jump\_cnt    = ULONG\_MAX;    int succ = 1;        for (int i = 0; i < REPEAT; i ++) {          Result res;          run\_once(bench, &res);          printk(res.pass ? "\*" : "X");          succ &= res.pass;          if (res.msec    < msec)   msec    = res.msec;          if (res.inst\_cnt  < inst\_cnt)   inst\_cnt  = res.inst\_cnt;          if (res.ex\_cnt    < ex\_cnt)   ex\_cnt    = res.ex\_cnt;          if (res.mem\_cnt   < mem\_cnt)  mem\_cnt   = res.mem\_cnt;          if (res.mem\_delay\_cnt   < mem\_delay\_cnt)mem\_delay\_cnt = res.mem\_delay\_cnt;          if (res.br\_j\_v\_cnt  < br\_j\_v\_cnt)   br\_j\_v\_cnt  = res.br\_j\_v\_cnt;          if (res.br\_j\_f\_cnt  < br\_j\_f\_cnt)   br\_j\_f\_cnt  = res.br\_j\_f\_cnt;          if (res.branch\_v\_cnt  < branch\_v\_cnt) branch\_v\_cnt  = res.branch\_v\_cnt;          if (res.branch\_f\_cnt  < branch\_f\_cnt) branch\_f\_cnt  = res.branch\_f\_cnt;          if (res.jump\_cnt  < jump\_cnt)   jump\_cnt  = res.jump\_cnt;        }        if (succ) printk(" Passed.\n");        else printk(" Failed.\n");        pass &= succ;        // TODO [COD]        //   A benchmark is finished here, you can use printk to output some information.        //   `msec' is intended indicate the time (or cycle),        //   you can ignore according to your performance counters semantics.      printk ("The num of clock : %u\n" , msec);      printk ("The num of obtained instruction : %u\n" , inst\_cnt);      printk ("The num of execution : %u\n" , ex\_cnt);      printk ("The num of mem visit : %u\n" , mem\_cnt);      printk ("The num of delay of mem : %u\n" , mem\_delay\_cnt);      printk ("The num of PC not refresh for PC4 : %u\n" , br\_j\_v\_cnt);      printk ("The num of PC refresh for PC4 : %u\n" , br\_j\_f\_cnt);      printk ("The num of successful branch : %u\n" , branch\_v\_cnt);      printk ("The num of failed branch : %u\n" , branch\_f\_cnt);      printk ("The num of jump : %u\n" , jump\_cnt);      }    } |

对每个benchmark，运行run\_once函数，将对应性能统计结果存储于结构体变量Res中，当其值小于默认值（最大值）时，将性能统计值赋值为结构体中所存储的值。然后调用printk函数将所有性能统计结果打印。

性能统计结果：（以15pz为例）



1. 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）

问题1：状态机中赋值条件未覆盖所有情况

在本次实验测试中五个阶段测试均可通过，并未发现错误。在后续实验prj4中通过硬件仿真加速工具发现状态机第二部分的一个case中，if未写else，形成了latch。

问题2：延迟降低

验收过程中，常老师提到可以用与或电路代替三目运算，也即将电路从级联改为树形连接，从而降低延迟。将代码中三目运算层级大于2的均替换为与或门，降低了延迟。

问题3：命名歧义

在CPU设计中，对PC、Instruction及ALU和shifter的操作数和结果进行寄存。原文件将某wire型信号延后一拍的寄存器命名为xxx\_reg，和vivado编译中对于reg型变量添加的后缀重复，容易产生歧义，故将后缀改为xxx\_tmp。

1. 对讲义中思考题（如有）的理解和回答

volatile 可保证系统总是从其所在内存读取数据，阻止编译器对访问变量的代码优化，从而提供对特殊地址的稳定访问，即被volatile修饰的变量修改后会直接写回内存，其他操作访问该变量将直接从内存中读取数值。由此保证了某变量修改后全局可见，且保证了多线程下相关代码的执行顺序。

第二部分提供的puts代码由于对于寄存器的读写均针对绝对地址，且只在一次操作完成，稳定性较好。其他线程对寄存器的改变对不会影响一次读或写寄存器得到的结果，因此即使删除了volatile关键字，仍在仿真和上板阶段都得到了正确的结果。但在与其他同学的讨论中发现，对于某些puts函数实现方法，删除volatile会导致打印信息不全的问题。

1. 在课后，你花费了大约\_\_\_\_\_5\_\_\_\_\_\_小时完成此次实验。
2. 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）

心得感受：

本次实验基于prj2的多周期处理器，针对真实内存访问进行了一定优化。硬件部分实现难度较低，主要需要加深对于握手信号的理解。在外设可控制器访问阶段，需要了解不同寄存器的读写权限，在特定条件下进行指定位数字符的传送。性能计数器的逻辑思路较为简单，且重复性较强，只需针对欲统计数值添加即可。

建议：

在prj2多周期处理器的基础上，本次实验难度较小，虽然实现了软硬件的协同，但从需要完成的代码部分来看，其联系仍较为割裂。建议适当加深对外设控制器、打印和性能计数函数等的讲解和要求，如当puts函数将字符写入TX FIFO后，该字符将被如何处理以使对应位置可接收下一字符。如bench.c如何对调用各函数最终实现将性能统计结果打印在日志中。

另外，在本次实验中，发现多次上板运行报错，需要重复retry才可通过，在消耗大量时间的同时也导致了更多无用的板卡占用。希望能进一步加强FPGA稳定性，减少不同并行任务的相互影响。

致谢：

感谢常老师在代码优化上的建议（详见问题2、3）。