

# 中国科学院大学

## 《计算机组成原理(研讨课)》实验报告

姓名 朱夏楠 学号 2022K8009929031 专业 计算机科学与技术  
实验项目编号 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 SERVE CloudIDE 的 `/home/serve-ide/cod-lab/reports` 目录下(注意: reports 全部小写)。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` 推送到实验课 SERVE 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 Verilog HDL 关键代码段 {包含注释} 及其对应的逻辑电路结构图 {自行画图, 推荐用 PPT 画逻辑结构框图后保存为 PDF, 再插入到 L<sup>A</sup>T<sub>E</sub>X. 中}、相应信号的仿真波形和信号变化的说明等)

#### ● 真实内存访问通路设计。

在上一个实验项目中, 处理器核从理想内存中取得数据, 访问内存存在一个周期内即可完成。但对于真实的内存, 从内存中读取数据或是向内存中存数时消耗的时间较多, 需要多个时钟周期实现读/写的操作。因此, 需要在上个实验项目完成的多周期处理器的基础上进行改进, 使其可以访问真实内存。

对于实验项目二的多周期处理器, 复位信号未拉高时, 处理器在每一个时钟周期所处的状态与下一个时钟周期的状态都不相同。而对于真实内存的情况, 在对内存的读写操作完成前, 处理器无法转到下一个状态。为此, 需要根据访问内存的需要原先的五状态 FSM 的基础上添加新的状态。

处理器核在取指和执行 load 与 store 指令时需要访问内存, 因此, 考虑在处理器核和真实内存之前设置四条访问通道, 每个通道中的请求和应答信号成功握手后, 处理器即可转入下一个状态, 四条通道分别为指令请求发送通道、指令应答接收通道、数据请求发送通道、数据应答接收通道。因此, 可在 INIT 状态和译码状态之间添加取指状态和等待指令状态, 并将内存访问阶段拆分为 store 和 load, 在执行 load 时, 添加内存读状态和读数据等待状态, 在执行 store 时, 添加内存写状态。

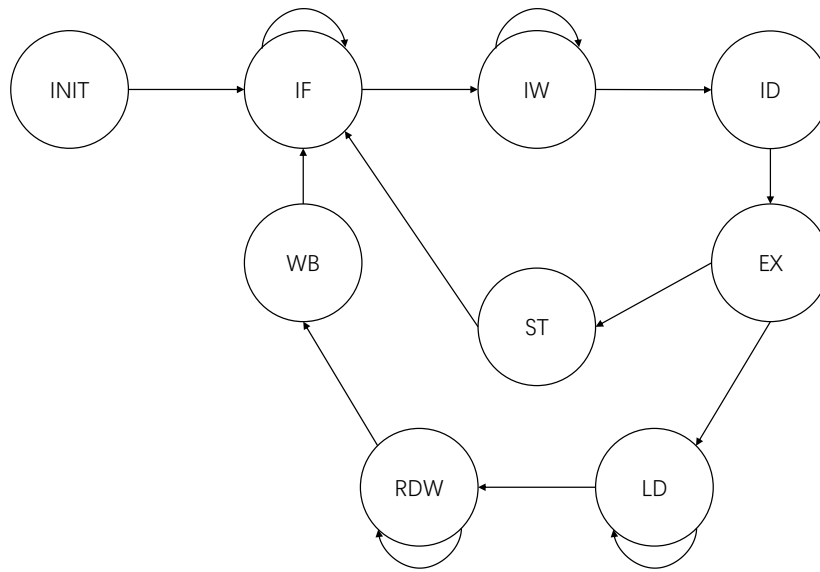


图 1: 定制处理器状态转移图

处理器取指时,向内存发出 `Inst_Valid` 信号,并停在 IF 状态等待内存发出 `Inst_Req_Ready` 信号,两信号握手成功后,处理器认为内存可发出指令,进入 IW 状态,并将 `Inst_Ready` 信号拉高,等待内存发出 `Inst_Valid` 信号,将内存发出的数据存入 IR 中,进入 ID 状态。处理器执行 store 指令时,处理器从 EX 状态转向 ST 状态,并将 `MemWrite` 信号拉高,在信号握手成功之前,一直保持在 ST 状态,直到内存发送出 `Mem_Req_Ready` 信号,处理器认为数据已成功写入内存,该指令执行完成,转到对下一条指令的取指阶段。处理器执行 load 指令时,处理器从 EX 状态转向 LD 状态,并将 `MemRead` 信号拉高,直到内存发出 `Mem_Req_Ready` 信号,处理器认为内存可以发出数据,转入 RDW 状态并拉高 `Read_Data_Ready` 信号,随后在接收到内存发出的 `Read_Data_Valid` 后,接收内存传入的数据,并将其存入相应的寄存器中,随后转入 WB 阶段。

```

/*
localparam INIT    = 9'b000000001, //0
    IF            = 9'b000000010, //1
    IW            = 9'b000000100, //2
    ID            = 9'b000001000, //3
    EX            = 9'b000010000, //4
    ST            = 9'b000100000, //5
    LD            = 9'b001000000, //6
    RDW           = 9'b010000000, //7
    WB            = 9'b100000000; //8
*/

assign MemRead    = current_state[6];
assign MemWrite   = current_state[5];
assign Inst_Req_Valid = current_state[1];
assign Inst_Ready  = current_state[2] | current_state[0];
assign Read_data_Ready = current_state[7] | current_state[0];
  
```

#### • 字符串打印功能实现。

本实验将 UART 的发送端口向处理器暴露出来,因此可以使用 C 语言编写驱动程序,实现终端打印字符串

的功能。

在本次实验中, I/O 端口和内存采用统一编址, 实验框架中给出了 UART 控制器的发送队列寄存器和状态寄存器的地址, 可据此将字符串的每一个字节依次发送出去。根据状态寄存器的发送队列状态标志位判断 UART 发送队列是否填满, 若是, 则不断循环直到队列出现空闲, 可将新的一个字节送入发送队列中。

```
int puts(const char *s)
{
    /*
    #define UART_TX_FIFO  0x04
    #define UART_STATUS  0x08
    #define UART_TX_FIFO_FULL (1 << 3)

    volatile unsigned int *uart = (void *)0x60000000;
    */
    int i = 0;
    while(s[i]!='\0')
    {
        //队列满则不断做循环
        while(((volatile unsigned *)((void *)uart+UART_STATUS)) & UART_TX_FIFO_FULL)
            ;
        *((char *)uart+UART_TX_FIFO) = s[i++];        //将字符放入发送队列寄存器
    }
    return i;
}
```

#### • 硬件性能计数器的实现及访问。

硬件性能计数器实现的基本逻辑是在 verilog 文件中添加寄存器用于计数, 根据性能计数器需要的功能不同, 寄存器在处理器执行一些指令或是处于某些状态时进行计数, 再由软件在特定的时刻将这些值读出来。以统计运算过程中指令条数的计数器的实现为例, 寄存器在程序开始执行时置零, 在每一次成功取指后, 处理器进入译码状态, 每当进入该状态, 则计数寄存器的值加一, 软件驱动程序则在程序开始运行和运行结束时分别获取寄存器内的数据, 两者相减得到运行所执行的总指令数。

```
#include "perf_cnt.h"
//指令计数器的可见地址
volatile unsigned long * const cpu_perf_cnt_1 = (void *)0x60010008;
unsigned long _instr_num() {
    return *cpu_perf_cnt_1;
}

void bench_prepare(Result *res) {
    res->msec = _uptime();
    res->instrnum = _instr_num();        //程序开始时获取指令计数器中的数据
    res->instrreq = _instr_req_cycle();
    res->instrvalid = _instr_valid_cycle();
    res->memsreq = _mems_req_cycle();
    res->memlreq = _meml_req_cycle();
    res->memvalid = _mem_valid_cycle();
    res->jumpnum = _jump_num();
    res->branchnum = _branch_num();
}
```

```

}

void bench_done(Result *res) {
    res->msec = _uptime() - res->msec;
    res->instrnum = _instr_num() - res->instrnum; //程序结束时获取执行的指令条数
    res->instrreq = _instr_req_cycle() - res->instrreq;
    res->instrvalid = _instr_valid_cycle() - res->instrvalid;
    res->memsreq = _mems_req_cycle() - res->memsreq;
    res->memlreq = _meml_req_cycle() - res->memlreq;
    res->memvalid = _mem_valid_cycle() - res->memvalid;
    res->jumpnum = _jump_num() - res->jumpnum;
    res->branchnum = _branch_num() - res->branchnum;
}

```

## 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)

### • 定制 MIPS 处理器设计问题。

本次实验中处理器需要访问真实内存,基于多周期处理器设计的状态机不再适用,具体来讲,原本在多周期处理器取指和访存阶段的控制信号现在被拆散到取指、指令等待、内存读、内存写、读数据等待这几个阶段,且 PC 值按照多周期处理器的写法也会出现错误。我在实验中所遇到的问题主要就是未能考虑到以上情况导致行为仿真出现异常。

首先为 PC 值的异常问题。在多周期处理器中,取指阶段的下一个阶段就是译码阶段,因此可以在取指阶段更新 PC 值使其顺序指向下一条指令,但在本次实验中,处理器会在取指阶段等待内存发出的信号,待握手成功后再进入下一阶段,因此会导致 PC 值不断增加,取指发生错误。解决方案为修改 PCWrite 的赋值逻辑,从原先的取指阶段置 1 变为处于指令等待阶段且 Inst\_Valid 信号为 1 时再置 1。这样一来,在处理器成功接收内存发送的指令指令后,PC 寄存器值更新,下一个阶段进入译码,此时即为期望的 PC 值。

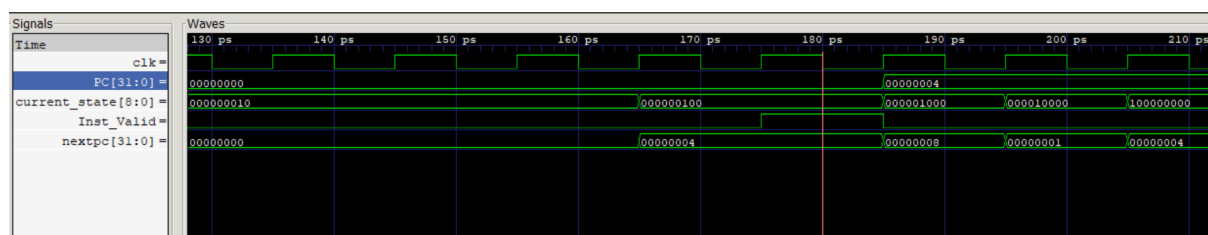


图 2: 如图所示,PC 在 IW 阶段,Inst\_Valid 拉高时更新

对于 load 和 store 类指令,需要在内存读阶段将 MemRead 信号拉高,在内存写阶段将 MemWrite 信号拉高,而在执行、内存写、内存读、读数据等待这几个阶段应保持 ALU 计算出的访问内存的地址正确。

### • UART 控制器驱动程序问题。

这部分主要出现的问题为未考虑到指针类型对偏移量的影响导致出现死循环,以及对 volatile 关键字理解不足导致驱动程序出现打印不全的问题,后者将在思考题部分进行具体的说明。

该 C 程序中,uart 指针所指向的类型为 unsigned int,在判断队列是否有空闲时,需要进行 while 循环检查发送队列标志位是否为 1,若为 1,则一直进行循环。若要取到标志位所在内存位置的值,需要以 uart 为基址再加上偏移量,进行此运算时需要将指针类型变为指向任意类型指向 char 类型的指针,否则地址偏移量会与预期不

同,可能会出现卡死在循环中的情况。表现出的结果为行为仿真时在终端有输出值的测试集会不停运行,而不出现输出值。

### 三、 对讲义中思考题(如有)的理解和回答

- 思考题:C 语言 volatile 关键字。

volatile 翻译为易变的,在 C 语言中加入该关键字,会告诉编译器不要优化所涉及的代码。一个简单的例子如下

```
volatile int a;
int b;
int func1()
{
    int i = 0;
    for(a=0;a<10;a++)
        i++;
    return i;
}
int func2()
{
    int i = 0;
    for(b=0;b<10;b++)
        i++;
    return i;
}
```

在 Ubuntu22.04 上使用版本为 11.4.0 的 gcc 对以上代码进行 O1 优化,生成 x86-64 汇编代码,会发现对于 func1,每一次对 a 进行累加和比较操作时,都会从内存中存储 a 值的位置将值更新到寄存器中,而对于 func2,编译器优化时认为 b 的值不会被其他程序改变,因此认为 for 循环是无用的,选择将其优化,直接将立即数 10 赋给 b 和 i。

具体到本次实验中,实验要求编写 UART 驱动程序,由于 UART 发送队列可能出现队列被填满的情况,因此需要在每次将一个字符送入发送队列之前,检查发送队列状态标志位是否置 1,若是,则需等待直到队列出现空闲。为实现等待,需要插入空语句循环,每次循环开始时,检查标志位是否为 0。若如下形式编写程序

```
int
puts(const char *s)
{
    //TODO: Add your driver code here
    /*
    #define UART_TX_FIFO  0x04
    #define UART_STATUS  0x08
    #define UART_TX_FIFO_FULL (1 << 3)

    volatile unsigned int *uart = (void *)0x60000000;
    */
    int i = 0;
    while(s[i]!='\0')
    {
        while(*((char *)uart+UART_STATUS) & UART_TX_FIFO_FULL) //判断是否队列是否有空闲
```

```

    ;
    *((char *)uart+UART_TX_FIFO) = s[i++];
}
return i;
}

```

可以看到,此时 while 循环的判断条件为取出标志位所在地址的值,与掩码标志位做与运算。然而对 uart 做强制类型转换时,未标注为 volatile 的指针,编译器认为该程序未对标志位所在地址的值进行更改,因此进入循环时只将该值取出一次,而开始时,该位置的标志位应为 0,因此此后的每次检验都会认为队列为空。新字符在队列满时即填入队列,将原先的值覆盖,导致 fpga 仿真阶段,终端打印的数据不全。

```

32 Executing "step_script" stage of the job script
33 $ bash ./fpga/design/ucas-cod/run/simple_cpu/fpga_run.sh $BENCH_SUITE $TARGET_DESIGN $CPU_ISA
34 RUNNER_CNT = 3
35 Completed FPGA configuration
36 Launching hello benchmark...
37 tggetattr: Inappropriate ioctl for device
38 reset: before MMIO access...
39 reset: MMIO accessed
40 axi_firewall_unblock: firewall error status: 00000000
41 main: before DDR accessing...
42 main: DDR accessed...
43 reset: before MMIO access...
44 reset: MMIO accessed
45 testing 1 2 0000fastndeadf00d % DEADF0000000010000000050 50 -50 429496time 10733.69ms
46 reset: before MMIO access...
47 reset: MMIO accessed
48 ./software/workload/ucas-cod/benchmark/simple_test/hello/mips/elf/hello passed
49 Hit good trap
50 pass 1 / 1
51 Cleaning up project directory and file based variables
52 Job succeeded

```

图 3: 如图所示,第 45 行输出显示不全

因此应采用如下方法编写程序,每次比较时,都会从内存更新标志位的值

```

int
puts(const char *s)
{
    //TODO: Add your driver code here
    /*
    #define UART_TX_FIFO 0x04
    #define UART_STATUS 0x08
    #define UART_TX_FIFO_FULL (1 << 3)

    volatile unsigned int *uart = (void *)0x60000000;
    */
    int i = 0;
    while(s[i]!='\0')
    {
        while(((volatile unsigned *)((void *)uart+UART_STATUS)) & UART_TX_FIFO_FULL)
        ;
    }
}

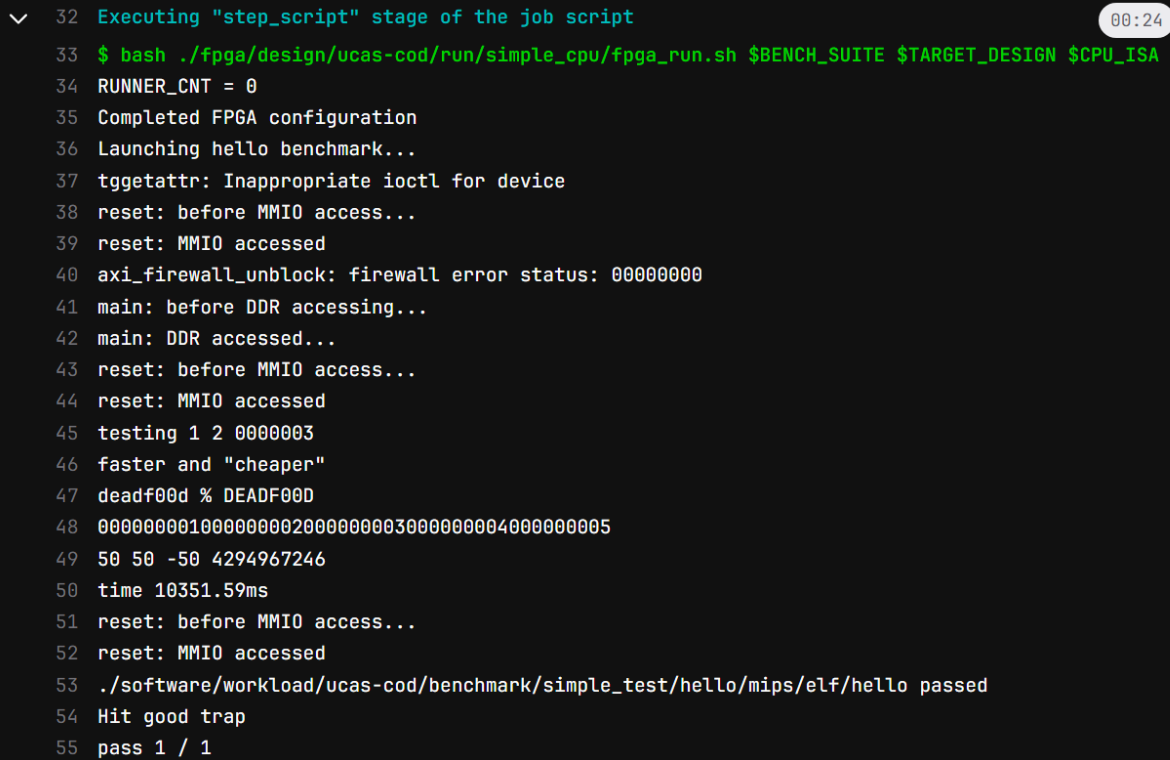
```

```

    *((char *)uart+UART_TX_FIFO) = s[i++];
}
return i;
}

```

输出结果正确



```

32 Executing "step_script" stage of the job script
33 $ bash ./fpga/design/ucas-cod/run/simple_cpu/fpga_run.sh $BENCH_SUITE $TARGET_DESIGN $CPU_ISA
34 RUNNER_CNT = 0
35 Completed FPGA configuration
36 Launching hello benchmark...
37 tggetattr: Inappropriate ioctl for device
38 reset: before MMIO access...
39 reset: MMIO accessed
40 axi_firewall_unblock: firewall error status: 00000000
41 main: before DDR accessing...
42 main: DDR accessed...
43 reset: before MMIO access...
44 reset: MMIO accessed
45 testing 1 2 0000003
46 faster and "cheaper"
47 deadf00d % DEADf00D
48 000000001000000002000000003000000004000000005
49 50 50 -50 4294967246
50 time 10351.59ms
51 reset: before MMIO access...
52 reset: MMIO accessed
53 ./software/workload/ucas-cod/benchmark/simple_test/hello/mips/elf/hello passed
54 Hit good trap
55 pass 1 / 1

```

图 4: 此时输出正确

在课后,你花费了大约\_\_\_\_10\_\_\_\_小时完成此次实验。