# 中国科学院大学计算机组成原理（研讨课）

# 实 验 报 告

学号：2021K8009929034　姓名：侯汝垚　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 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。<br>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 SERVE GitLab 远程仓库 master 分支（具体命令详见实验报告）。<br>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。 |

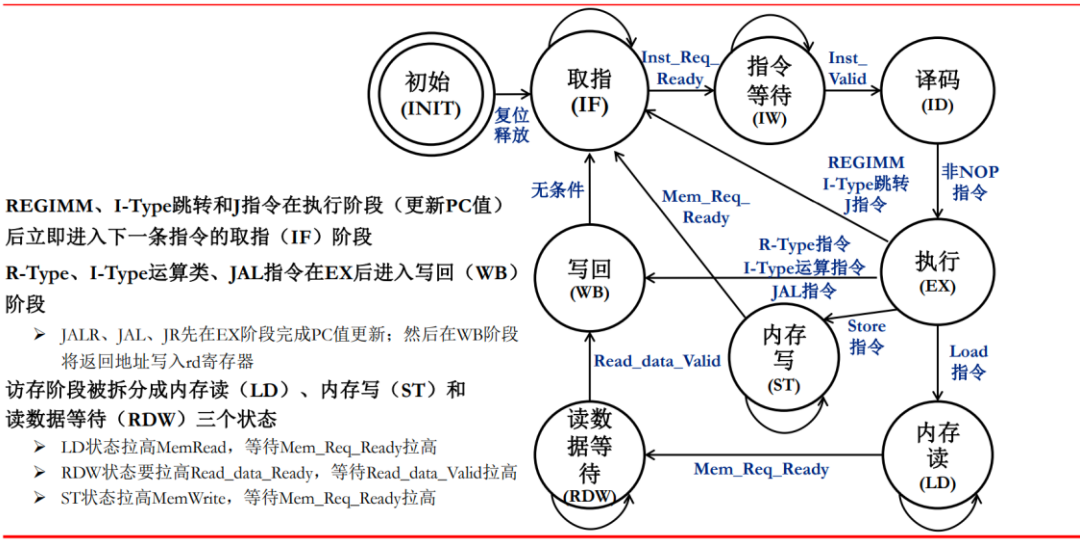
## 一、 逻辑电路结构与仿真波形的截图及说明

### 1. RTL 代码

在 prj2 中已经完成 simple_cpu 的多周期附加实验，因此本次实验只需略微修改一下状态机中的状态、相应的控制信号，然后加上三组握手信号即可。

```
localparam INIT = 9'b000000001,
           IF   = 9'b000000010,
           IW   = 9'b000000100,
           ID   = 9'b000001000,
           EX   = 9'b000010000,
           ST   = 9'b000100000,
           LD   = 9'b001000000,
           RDW  = 9'b010000000,
           WB   = 9'b100000000;
```

图表 1 状态机的 9 个状态




图表 2 状态转移图

首先将原来的五个状态变为九个状态，增加一个 IW 指令等待状态，然后将原来的访存状态 MEM 分为 ST（内存写），LD（内存读），RDW（读数据等待）状态。根据图二状态转移图修改 FSM 第二部分。

```
// FSM 2.
always @(*) begin
        case(current_state)
                INIT  : next_state = IF;
                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
                ID    : begin
                        if (NOP)
                                next_state = IF;
                        else
                                next_state = EX;
                end
```

```
                EX    : begin
                        if (J_type && ~link || REGIMM || I_type_branch)
                                next_state = IF;
                        else if (store)
                                next_state = ST;
                        else if (load)
                                next_state = LD;
                        else
                                next_state = WB;
                end
                ST    : begin
                        if (Mem_Req_Ready)
                                next_state = IF;
                        else
                                next_state = ST;
                end
                LD    : begin
                        if (Mem_Req_Ready)
                                next_state = RDW;
                        else
                                next_state = LD;
                end
                RDW   : begin
                        if (Read_data_Valid)
                                next_state = WB;
                        else
                                next_state = RDW;
                end
                WB    : next_state = IF;
                default: next_state = INIT;
        endcase
end
```

根据状态定义 CPU 需要发出的三个握手信号。

```
    // FSM 3.
    assign Inst_Req_Valid  = ~current_state[0] && current_state[1];
    assign Inst_Ready      =  current_state[0] || current_state[2];
    assign Read_data_Ready =  current_state[0] || current_state[7];
```

其余控制信号与简单多周期处理器基本一致，要注意等待内存响应时要保证各信号不变，在简单多周期从 CPU 中我的 ALUOut 每个周期都跳转，但这个实验中 ALUOut 在访存时需要保持。ALUop 同样采用两级译码，第一级在主模块通过 opcode 生成，第二级 ALUcontroler 模块中在根据 func 生成。

```
assign J_type = (op[5:1] == 5'b00001);
assign R_type = ~(|op);
assign REGIMM = (op == 6'b000001);
assign lui = (op == 6'b001111);

// unconditional jump
assign PCWrite     = current_state[2] && Inst_Valid || current_state[4] && jmp;
// conditional jump
assign PCWriteCond = current_state[4] && (REGIMM || I_type_branch);
assign MemRead     = current_state[6];
assign MemWrite    = current_state[5];
// to select the source of PC
assign PCsource[1] = current_state[4] && jmp;
assign PCsource[0] = current_state[4] && branch;
// to select the second oprand of ALU
assign ALUSrcB[1]  = current_state[5] || current_state[6] || current_state[4] && (op[3] || op[5]) || current_state[3] && branch;
assign ALUSrcB[0]  = current_state[5] || current_state[6] || current_state[2] || current_state[4] && link || current_state[3] && branch;
// to select the first oprand of ALU, RF_rdata1 or PC
assign ALUSrcA     = (current_state[4] && ~link) || (current_state[3] && ~branch);
assign RegWrite    = current_state[8];
assign IRWrite     = current_state[2] && Inst_Ready && Inst_Valid;
assign MemtoReg    = current_state[8] && load;
assign RegDst      = current_state[8] && R_type;
assign ImmDst      = current_state[8] && (op[5] || op[3]);

// ALUEx is only used in EX state
wire [2:0] ALUEx;

// first decode of ALUop by opcode, then we still need to use the ALUcontroler module to decode
assign ALUEx[2] = ~op[5] && (~op[3] || op[1] && ~op[0]);
assign ALUEx[1] = op[5] || (op[3] ^ op[2]) || REGIMM;
assign ALUEx[0] = R_type || REGIMM || ~op[5] && (~op[2] && op[1] || ~op[3] && op[2] && op[1] || op[3] && op[2] && op[0]);
```

```
// if ALUop is not 101, we can get ALUop only by op. So the input ALUop is the real ALUop
// if ALUop is 101, we must get ALUop by func because it is R-type instruction
module ALU_controler(
        input [2:0] ALUop,
        input [5:0] func,
        output [2:0] ALUcontrol
);
        wire R_type = (ALUop == 3'b101);
        wire [2:0] funcOP;
        assign funcOP[2] =  func[1] && (~func[3] || ~func[0]);
        assign funcOP[1] = ~func[2] && (~func[3] || func[1]);
        assign funcOP[0] =  func[2] && func[0] || func[3] && func[1];

        assign ALUcontrol = {3{ R_type}} & funcOP
                          | {3{~R_type}} & ALUop;

endmodule
```

```
        reg [31:0] ALUOut;
        always @(posedge clk) begin
                // ALUOut is unchanged while waiting for memory response
                if (current_state[5] || current_state[6] || current_state[7])
                        ALUOut <= ALUOut;
                else
                        ALUOut <= {32{~ShifterValid}} & ALU_result
                                | {32{ ShifterValid}} & Shifter_result;
        end
```

在握手成功时将读到的数据写入 MDR。

```
// MDR
reg [31:0] MDR;
always @(posedge clk) begin
        // Change the MDR only when the handshake is successful
        if (current_state[7] && Read_data_Ready && Read_data_Valid)
                MDR <= MemRdata;
        else
                MDR <= MDR;
end
```

其余部分与简单多周期 CPU 基本一致。

对于性能计数器选择了计算总周期、IW 等待周期、内存读周期、内存写周期、总指令数、无条件跳转数、分支指令数以及分支指令生效数。各计数器写法类似，在计数指令数时选择译码阶段计数，因为取值阶段可能需要多个周期。

```
reg [31:0] cycle_cnt, IW_cnt, memR_cnt, memW_cnt, jmp_times, branch_times, branchValid_times, Inst_num;

always @(posedge clk) begin
        if (rst) begin
                cycle_cnt <= 32'b0;
        end
        else begin
                cycle_cnt <= cycle_cnt + 32'b1;
        end
end
assign cpu_perf_cnt_0 = cycle_cnt;

always @(posedge clk) begin
        if (rst) begin
                IW_cnt <= 32'b0;
        end
        else if (current_state[2]) begin
                IW_cnt <= IW_cnt + 32'b1;
        end
        else begin
                IW_cnt <= IW_cnt;
        end
end
```

软件部分需要编写 put 函数，因宏定义中寄存器偏移量以字节为单位，所以根据 uart 定义两个 char*类型指针分别指向状态寄存器的最低字节以及 Tx_FIFO 中的字符。增加 volatile 关键字，否则编译器优化后会导致打印出的字符混乱。

```c
int puts(const char *s)
{
    // TODO: Add your driver code here
    int i = 0;

    // the lowest byte of stat_reg
    volatile unsigned char *stat_reg_l = (unsigned char*)uart + UART_STATUS;
    // Tx data in Tx FIFO
    volatile char *Tx_data = (char*)uart + UART_TX_FIFO;

    while (s[i] != '\0') {
        while (*stat_reg_l & UART_TX_FIFO_FULL)
            ;
        *(Tx_data) = s[i++];
    }
    return i;
}
```

修改 perf_cnt.c 文件，增加几个函数返回性能计数器中的数值。

```c
#include "perf_cnt.h"
#define CPU_PERF_CNT_0 (unsigned long*)0x60010000
#define CPU_PERF_CNT_1 (unsigned long*)0x60010008
#define CPU_PERF_CNT_2 (unsigned long*)0x60011000
#define CPU_PERF_CNT_3 (unsigned long*)0x60011008
#define CPU_PERF_CNT_4 (unsigned long*)0x60012000
#define CPU_PERF_CNT_5 (unsigned long*)0x60012008
#define CPU_PERF_CNT_6 (unsigned long*)0x60013000
#define CPU_PERF_CNT_7 (unsigned long*)0x60013008

unsigned long _uptime()
{
    // TODO [COD]
    //   You can use this function to access performance base related with time or cycle.
    return *CPU_PERF_CNT_0;
}

unsigned long _IW_cycles()
{
    return *CPU_PERF_CNT_1;
}

unsigned long _jmp_times()
{
    return *CPU_PERF_CNT_2;
}

unsigned long _branch_times()
{
    return *CPU_PERF_CNT_3;
}

unsigned long _branchValid_times()
{
    return *CPU_PERF_CNT_4;
}
```

修改 bench_prepare 和 bench_done 函数，使其分别让 res 存放计数器初值

和计数器运行前后的差值。

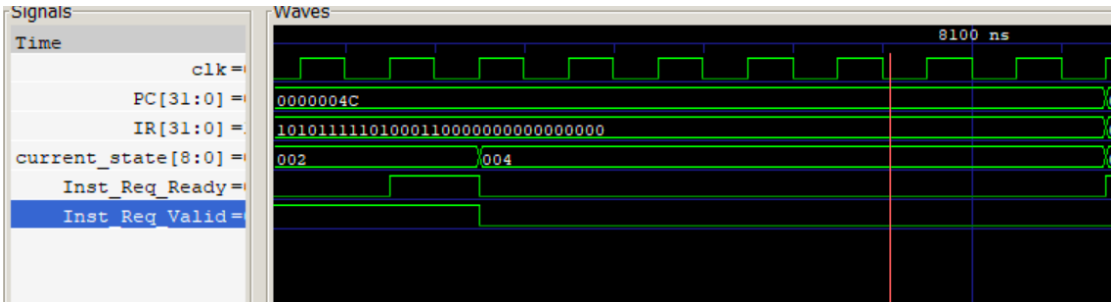
```
void bench_prepare(Result *res)
{
    // TODO [COD]
    //   Add preprocess code, record performance counters' initial states.
    //   You can communicate between bench_prepare() and bench_done() through
    //   static variables or add additional fields in `struct Result`
    res->msec              = _uptime();
    res->IW_cycles         = _IW_cycles();
    res->jmp_times         = _jmp_times();
    res->branch_times      = _branch_times();
    res->branchValid_times = _branchValid_times();
    res->memR_cycles       = _memR_cycles();
    res->memW_cycles       = _memW_cycles();
    res->instr_num         = _instr_num();
}

void bench_done(Result *res)
{
    // TODO [COD]
    //  Add postprocess code, record performance counters' current states.
    res->msec              = _uptime()            - res->msec;
    res->IW_cycles         = _IW_cycles()         - res->IW_cycles;
    res->jmp_times         = _jmp_times()         - res->jmp_times;
    res->branch_times      = _branch_times()      - res->branch_times;
    res->branchValid_times = _branchValid_times() - res->branchValid_times;
    res->memR_cycles       = _memR_cycles()       - res->memR_cycles;
    res->memW_cycles       = _memW_cycles()       - res->memW_cycles;
    res->instr_num         = _instr_num()         - res->instr_num;
}
```

最后在 bench.c 中打印结果。

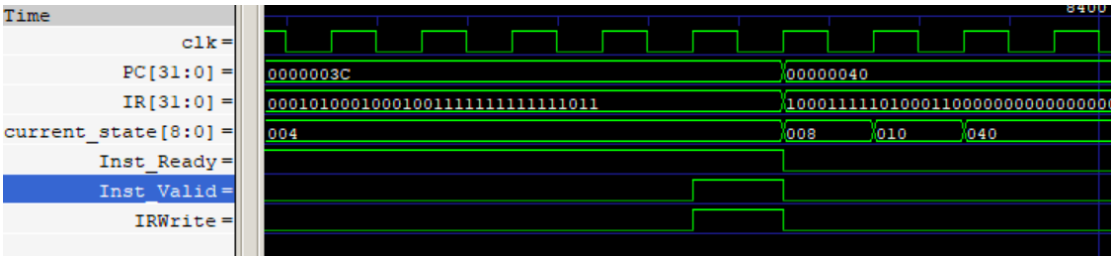
```
  printk("total cycle     : %u\n", msec);
  printk("instruction num : %u\n", instr_num);
  printk("IW cycle        : %u\n", IW_cycles);
  printk("memR cycle      : %u\n", memR_cycles);
  printk("memW cycle      : %u\n", memW_cycles);
  printk("jump times      : %u\n", jmp_times);
  printk("branch times    : %u\n", branch_times);
  printk("brachValid times: %u\n", branchValid_times);
}
```

2. 波形图测试，以访问为例

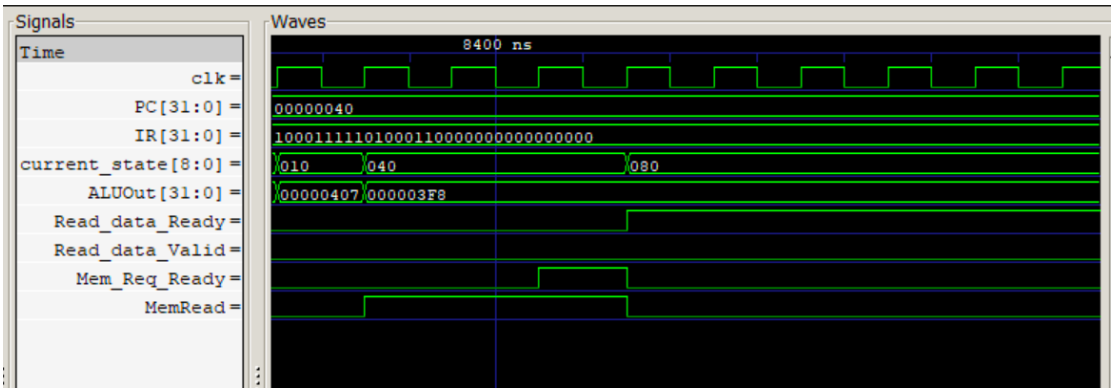
在 2 阶段，即取值阶段等待 Inst_Req_Ready 拉高，然后进入 IW 等待指令阶段



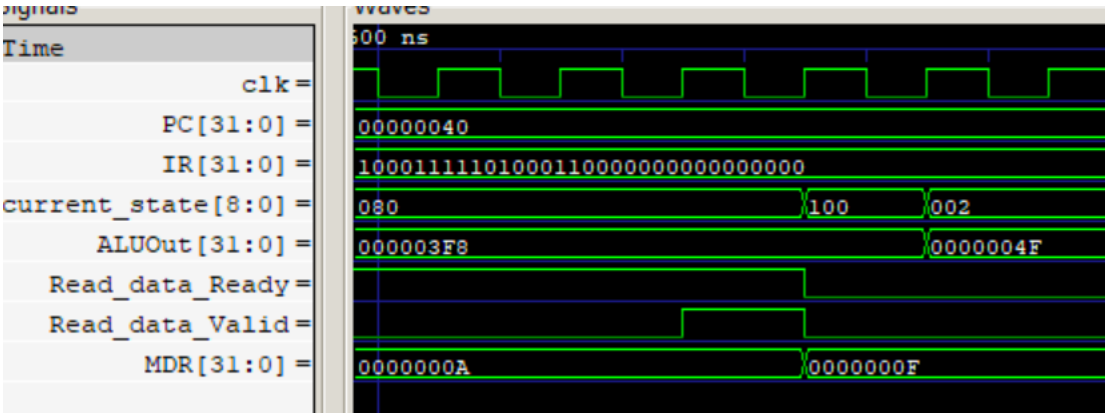

在 IW 阶段拉高 Inst_Ready，等待 Valid 信号拉高，握手成功后 IR 写使能信号 IRWrite 拉高，IR 变为当前指令，然后进入译码阶段，同时 PC 自增。




在 EX 阶段计算出地址后进入 LD 阶段（040），拉高 MemRead，等待 Ready 信号，然后进入 RDW 状态，拉高 Read_data_Ready，等待 Valid 信号。




握手成功后，输入写入 MDR。




3. 字符串打印结果

（1） Hello 打印结果

```
40  FST info: dumpfile fpga/sim_out/custom_cpu/dump.fst opened for output.
41  testing 1 2 0000003
42  faster and "cheaper"
43  deadf00d % DEADF00D
44  00000000100000000200000000300000000400000000 5
45  50 50 -50 4294967246
46  ================================================
47  Benchmark simulation passed!!!
```

（2） 性能计数器打印结果

```
[fib] Fibonacci number: * Passed.
total cycle     : 179408827
instruction num : 2525746
IW cycle        : 169418476
memR cycle      : 314528
memW cycle      : 3904
jump times      : 5220
branch times    : 398164
brachValid times: 382537
benchmark finished
```

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）

1. Puts 函数是否是无限循环?

```c
while (s[i] != '\0') {
    while (*stat_reg_l & UART_TX_FIFO_FULL)
        ;
    *(Tx_data) = s[i++];
}
```

Puts 函数中的一个循环并未修改任何变量，如果是平时编写的程序大概率会进入死循环，但是其中的访存其实访问的是外设而并非自身内存，因此这个值是可能被改变的。不过这个驱动程序应该还需要更多鲁棒的改进，比如当外设故障时可能会陷入死循环，因此我想可以设定一个计数器，超过这个次数就认为超时，跳出循环。

2. Printk 函数中似乎不支持%lu。

如果用%lu，程序会直接打印%lu，改成%u 后可以正常输出数字。

3. Puts 函数中定义的两个新变量没有加 volatile 关键字导致打印错误。

```
reset: MMIO accessed
[qsort] Quick so* Passed.
total cycle     instruction numIW cycle        memR cycle
memW cycle
jump times :branch times  7brachValid tim
bentime 10970.65ms
```

```
// the lowest byte of stat_reg
volatile unsigned char *stat_reg_l = (unsigned char*)uart + UART_STATUS;
// Tx data in Tx FIFO
volatile char *Tx_data = (char*)uart + UART_TX_FIFO;
```

如果定义 stat_reg_l 和 Tx_data 时不加 volatile，那么打印就会出错，原以为 uart 加了 volatile 后，它的效果可以延续到用 uart 定义的变量，但是后来想 uart 加 volatile 只保证用 uart 时一定访存取出这个数而不进行优化，但是 stat_reg_l 和 Tx_data 和 uart 只是数值上的关系，定义计算完后就没什么联系，因此使用 stat_reg_l 和 Tx_data 访存时可能会被编译器为了减少访存次数优化。

## 三、 对讲义中思考题（如有）的理解和回答

volatile 关键字的作用是什么？如果去掉会出现什么后果?

volatile 是一个特征修饰符，volatile 的作用是作为指令关键字，确保本条指令不会因编译器的优化而省略，且要求每次直接读值。

在本实验中访存位置是外设，因此存储的值会被外设修改，如果去掉 volatile，那么编译器优化时为了减少访存次数，对同一地址多次访存可能会优化为读取该内存的值后存入寄存器，每次访存时直接用寄存器内的值，所以当 uart 中的值改变时，原先存到寄存器内的值并不变，循环依然继续，导致打印的字符不全。根据测试来看编译器确实做了一定的优化，去掉 volatile 后打印的字符缺失了很多。

```
reset: MMIO accessed
[qsort] Quick so* Passed.
total cycle     instruction numIW cycle        memR cycle
memW cycle
jump times :branch times  7brachValid tim
bentime 10970.65ms
```

四、 在课后，你花费了大约 5 小时完成此次实验。

因为上次完成了简单处理器的多周期附加实验，这次实验轻松了很多。

五、 对于此次实验的心得、感受和建议。

本次实验主要工作量在于多周期处理器的编写，经过上个实验后对 CPU 工作原理的理解有了很大的进步，并且理论课也跟上了进度，因此实验顺手了很多。

调试过程中发现在和金标准 CPU 对比时不会对比写入数据、写使能是否正确，导致很多后面产生检测到 bug 需要一步一步向前看第一次出错的位置，希望可以像单周期 CPU 一样检测写入数据，使得大部分情况下报错位置就是有问题的指令。