数字逻辑电路与计算机组成实验

RISCV 五段流水线 CPU 设计

姓名：******

学号：******

邮箱：201220013@smail.nju.edu.cn

2021 年 12 月 31 日

目录

1 项目概览 **2**
1.1 硬件部分 . . . 2
1.2 软件部分 . . . 2
1.3 其他 . . . 3

2 硬件部分 **3**
2.1 基础五段流水线 . . . 3
2.1.1 流水线控制信号 . . . 4
2.1.2 取指模块 IF . . . 5
2.1.3 译码模块 ID . . . 5
2.1.4 执行模块 EX . . . 7
2.1.5 访存模块 MEM . . . 8
2.1.6 回写模块 WB . . . 8
2.2 冒险与 Load-Use 阻塞 . . . 9
2.3 基础流水线调试与 DEBUG . . . 11
2.4 中断、CSR 寄存器与 M 拓展指令 . . . 13
2.4.1 流水线中断控制器 client . . . 13
2.4.2 CSR 寄存器读写指令与 mret 指令 . . . 15
2.4.3 时钟中断 . . . 16
2.5 内存映射 . . . 17

3 软件部分 **18**
3.1 系统 API 函数 . . . 18
3.1.1 输出 . . . 18
3.1.2 输入 . . . 20
3.2 实现常用 std 函数 . . . 22
3.3 中断注册与处理 . . . 23
3.4 ui_mainloop 交互主函数 . . . 25
3.5 移植表达式求值 实现跑马灯 . . . 27
3.6 字符动画 . . . 27

4 参考及实验感悟 **30**

1 项目概览

1.1 硬件部分

1. **RISC-V 五段流水线 CPU**：实现了取指-译码-执行-访存-回写的流水线 CPU，包含转发与阻塞探查处理，常见数据冒险无冲刷，load-use 指令插入一次气泡。

2. **中断与 CSR 寄存器支持**：实现了流水线的中断，支持机器态、非向量、无嵌套的中断处理。添加 CSR 寄存器及 CSR 读写指令（csrrc、csrrw 等 6 条），添加 ecall、mret 指令。实现时钟中断。

3. **板载硬件支持与显示器拓展**：支持内核写入板上 6 个七段数码管与 10 个 LEDR，支持 VGA 字符的最高 256 色显示

4. **内存映射与 IO 调度**：利用内存映射实现代码区、数据区、显存、键盘数据、中断临时栈等无冲突读写

1.2 软件部分

1. **内核编写**：使用 C 语言编写内核，提供了输入输出接口、常见 std 函数（strtok、atoi、strcmp、itoa）、软件乘除法，兼具易用性与可拓展性。实现几条要求的基础指令。

2. **中断注册与处理**：使用汇编语言编写中断处理入口函数、完成状态保存与上下文切换，C 语言使用内联汇编方式编写中断注册与实际处理函数，实现了时钟中断处理。

3. **移植表达式求值、实现跑马灯功能**：移植了 PA 的表达式求值，可以实现常见的表达式求值。利用板载硬件的内存映射完成类似开发板开机时的跑马灯功能。

4. **字符动画**：基于之前写的字符动画脚本实现了字符动画。利用字符串压缩算法大幅减小存储占用，压缩率 9%，使用 128KB 内存可以播放约 30s。

1.3 其他

1. **编写调试辅助脚本**：编写了一个 python 脚本，可以交互式地将 16 进制 Riscv 指令转换为汇编，方便调试

2 硬件部分

2.1 基础五段流水线

首先贴上一张《计算机系统基础》课程的 ppt 内容，下图展示了一个基本的五段流水线 CPU 的工作流程，也即取指-译码-执行-访存-回写五个步骤，理想状态下流水想上始终有 5 个指令在并行执行。

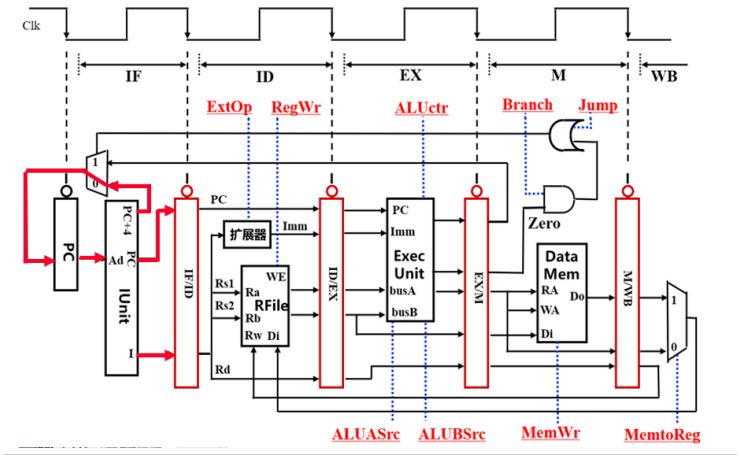


图 1: 5 段流水线原理图

本项目的流水线基本遵循了这一结构，只是在 branch 部分稍作修改，以符合实验十一的指令译码，在后续的中断支持部分添加了一些特有的部件，不过不影响当前这一部分的执行。

可以看到，流水线的设计分为 10 个主要部分，5 个流水线寄存器、5 个执行部件，我们的主要工作就是完成分别完成他们，做好他们之间的连线。这一部分在 **hardware/cpu/pipelilne.v** 中。

在具体阐述每个部件的工作之前，我们先明确一下整个项目的时序设计，整体上时序遵循存储器与通用寄存器堆上升沿读写、内部数据下降沿写入上升沿使用的逻辑，具体分配如下：

1. **上升沿**：流水线控制器、中断控制器、通用寄存器与 CSR 寄存器写入、RAM 读写
2. **下降沿**：流水段寄存器、PC、中断信号发生器

其他例如时钟中断发生器、VGA 控制器、PS2 控制器的时钟沿实际上并不那么严格。

2.1.1 流水线控制信号

流水线的行为是比较复杂的，正常情况下 5 个状态依此向前流动，也有不少特殊情况，如当 branch 跳转来临时，需要冲刷 MEM 之前的所有寄存器，如中断信号来临时，应该暂停流水线以便做好状态保存，保存之后又要冲刷掉所有流水段寄存器内容进入中断处理。因而项目设置了一个 *pipeline_status* 模块，根据若干控制信号生成 5 个流水段寄存器以及 PC 的控制信号，信号宽度 3bit，以 define 的形式写在 **hardware/cpu/define.v** 中。外界信号的响应有**优先级**，从高到低分别为：全局 reset 信号、中断信号、中断预处理的流水线暂停、跳转、load-use 阻塞、正常工作。

```
1  module pipeline_status (
2   input clr,
3   input clk,
4   input branch,
5   input stall,
6   input int_set_pl_pause,
7   input int_flag,
8
9   input [31:0] int_pc,
10
11  output [`PL_STATUS_BUS_WIDTH] pl_ctrl_pc_out,
12  output [`PL_STATUS_BUS_WIDTH] pl_ctrl_ID_out,
13  output [`PL_STATUS_BUS_WIDTH] pl_ctrl_EX_out,
14  output [`PL_STATUS_BUS_WIDTH] pl_ctrl_MEM_out,
15  output [`PL_STATUS_BUS_WIDTH] pl_ctrl_WB_out,
16
17  output reg [31:0] int_pc_out
18 );
19 //具体信号生成见代码
20 endmodule
```

2.1.2 取指模块 IF

IF 段包含了 PC 以及指令的获取，流水线的模块中并不包括 Instruction RAM，IF 段会引出地址引脚、引入读取到的地址，提供的引脚如下：

```
module IF (
  input             clr,
  input             clk,

  input      [31:0] pc,
  output     [31:0] instr,

  input      [31:0] imemdataout,
  output     [31:0] imemaddr,
  output            imemclk
);
```

PC 被分离为一个单独的模块，根据控制信号选择不同的 nextpc：

```
always @(negedge clk) begin
 if(PL_status == `PL_FLUSH)
    pc <= 0;
 else if(PL_status == `PL_PC_INT)
    pc <= nextpc_int;
 else if(PL_status == `PL_PC_BRANCH)
    pc <= nextpc_mem;
 else if(PL_status == `PL_PAUSE)
    pc <= pc;
 else if(PL_status == `PL_NORMAL)
    pc <= pc + 4;
 else
    pc <= 0;
end
```

2.1.3 译码模块 ID

IF 与 ID 之间有一个 IF_ID_reg 模块，存储流水信息，在这里只存储了指令与 PC，下面的 4 个流水段寄存器中还会存储控制信号在内的很多信息，不过大同小异，在这里给下整体框架，后续将不再赘述各个流水段寄存器的行为了。

```
module IF_ID_reg (
    input      [`PL_STATUS_BUS_WIDTH] pl_ctrl_ID,
    input             clk,
    input      [31:0] pc_in,
    input      [31:0] instr_in,
    output reg [31:0] pc_out,
    output reg [31:0] instr_out
);

always @(negedge clk) begin
    if(pl_ctrl_ID == `PL_FLUSH) begin
        pc_out <= 32'd0;
        instr_out <= 32'd0;
    end else if(pl_ctrl_ID == `PL_PAUSE) begin
        pc_out <= pc_out;
        instr_out <= instr_out;
    end else begin
        pc_out <= pc_in;
        instr_out <= instr_in;
    end
end
```

ID 模块的主要工作是根据取到的 Instruction 进行指令译码，生成控制信号 *extop, regwr, ALUAsrc, ALUBsrc, ALUctr, branch, MemtoReg, memwr, memop*（这一部分不会提及有关中断的控制信号与各个引脚）并生成立即数。控制信号生成器直接使用实验十一的 **control_gen** 模块，这里不再详述，ID 模块主体代码如下：

```
module ID (
//若干引脚
);

assign rd = instr[11:7];
assign rs1_addr = instr[19:15];
assign rs2_addr = instr[24:20];
contr_gen contr_gen_instance(...);
imm_gen imm_gen_instance(instr, extop, imm);

```

```
11    assign csr_raddr = {20'd0, imm[11:0]}; //这两行是中断支持
12    assign csr_data_out = csr_rdata;
13
14    endmodule
```

2.1.4 执行模块 EX

执行模块使用 ID_EX_reg 流水段寄存器输出的内容，传入 ALU 中进行运算并输出 aluresult，算术运算部分也是完全沿用了实验十一的相关内容，只不过两个输入需要考虑转发（处理后为 rs1_proc, rs2_proc）,EX 模块中还会并行计算出可能的两个跳转地址传入下一个流水段寄存器. 主体代码如下:

```
1     module EX (
2         //若干引脚
3     );
4
5     wire [31:0] dataa;
6     wire [31:0] datab;
7
8     wire [31:0] rs1_proc = forward_rs1[0] ? aluresult_wb :(forward_rs1[1]
          ? aluresult_mem :rs1);
9     wire [31:0] rs2_proc = forward_rs2[0] ? aluresult_wb :(forward_rs2[1]
          ? aluresult_mem :rs2);
10
11    assign rs2_forward = rs2_proc;
12    assign nextpc_pc = pc + imm;
13    assign nextpc_rs1 = rs1_proc + imm;
14    assign dataa = ALUAsrc?pc:rs1_proc;
15    assign datab = (ALUBsrc==2'b00 ? rs2_proc :(ALUBsrc==2'b01 ? imm :32'
          d4));
16    alu alu_instance(dataa, datab, ALUctr, less, zero, aluresult);
17    CSRALU csr_alu_instance(csr_data, rs1_proc, {27'd0, zimm}, csr_aluctr
          , csr_aluresult); //CSR指令专用ALU
18
19    endmodule
```

2.1.5 访存模块 MEM

访存模块也没有实例化 RAM，只是做了一个引脚的连接来调用外界的 RAM，再将输出送给下一个流水段寄存器。在 MEM 模块中还做了跳转的判断，这样可以最大限度地减少跳转带来的流水线冲刷，如果需要跳转，在 MEM 模块就会将 branch 信号送出，并给出 nextpc 的值。

```
module M (
    //若干引脚
);

assign dmemaddr = aluresult;
assign dmemdatain = rs2;
assign dmemrdclk = clk;
assign dmemwrclk = clk;
assign dmemop = memop;
assign dmemwe = memwr;

assign dmemdata = dmemdataout;

wire PCAsrc, PCBsrc;
branch_cond branch_cond_instance(less, zero, branch, PCAsrc, PCBsrc);
assign nextpc = PCBsrc?nextpc_rs1:nextpc_pc;
assign pc_branch = PCAsrc?1'b1:1'b0;
endmodule
```

2.1.6 回写模块 WB

回写模块事实上并没有作什么工作，只是根据 memtoreg 和下面的 csr2reg 选择合适的 busW 传入通用寄存器堆，通用寄存器堆会根据 regwr 选择进行写入操作。

```
module WB (
    //若干引脚
);
assign busW = csr2reg ? csr_aluresult :(MemtoReg?dmemdata:aluresult);
endmodule
```

2.2 冒险与 Load-Use 阻塞

流水线的 5 段处理大大增加硬件使用效率的同时也带来了诸多问题，流水线有三种常见冒险：

- 结构冒险：如果一条指令需要的硬件部件还在为之前的指令工作，而无法为这条指令提供服务，那就导致了结构冒险。
- 数据冒险：如果一条指令需要某数据而该数据正在被之前的指令操作，那这条指令就无法执行，就导致了数据冒险
- 控制冒险：如果现在要执行哪条指令，是由之前指令的运行结果决定，而现在那条之前指令的结果还没产生，就导致了控制冒险。

结构冒险方面，指令寄存器与数据寄存器已经被分开，通用寄存器设置为上升沿写入、下降沿有效，规避了两种结构冒险。

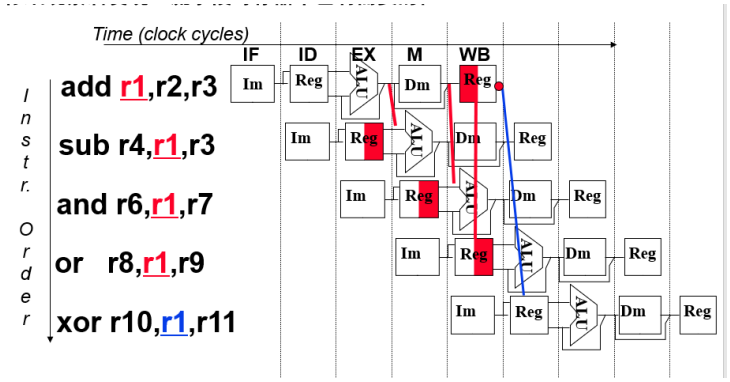


图 2: 3 种需要转发的流水线状态

数据冒险方面，当一条指令的运算结果需要写入某个寄存器，且这个寄存器的值需要被下两条中的某个取用时会发生数据冒险，因为此时正确的运算结果还没有写入寄存器，ID 阶段取出的值还是不正确的。处理这种冒险的方式有很多种，这里采用**转发**的方式解决。观察发现，冒险发生时，正确的结果已经被计算出来，虽然从 regfile 中取到的值是不正确的，我们可以从 EX-M 或者 M-WB 流水段寄存器中转发正确的结果，只需要一个模块进行判断是否需要转发即可。在 forward_detecter 模块中，根据 EX 阶段的两个寄存器地址值与 M、WB 两阶段写回的地址与使能信号，判断是否

需要转发，输出转发控制信号，优先级方面，WB 转发优先于 M 转发，代码如下：

```
module forward_detecter(
  input             regwr_mem,
  input             regwr_wb,
  input     [4 :0]  rs1,
  input     [4 :0]  rs2,
  input     [4 :0]  rd_addr_mem,
  input     [4 :0]  rd_addr_wb,
  // forward ctrl: 00 = no forward, 01 = forward from m/wb, 10 =
      forward from ex/mem
  output    [1 :0]  forward_rs1,
  output    [1 :0]  forward_rs2
 );

// forward rs1, mem first
wire tmp_rs1 = regwr_mem && (rs1 == rd_addr_mem);
assign forward_rs1[1] = rs1 != 5'h0 && tmp_rs1;
assign forward_rs1[0] = rs1 != 5'h0 && regwr_wb && !tmp_rs1 && (rs1
    == rd_addr_wb);

// forward rs2
wire tmp_rs2 = regwr_mem && (rs2 == rd_addr_mem);
assign forward_rs2[1] = rs2 != 5'h0 && tmp_rs2;
assign forward_rs2[0] = rs2 != 5'h0 && regwr_wb && !tmp_rs2 && (rs2
    == rd_addr_wb);

endmodule
```

还剩下一种数据冒险没有解决：**load-use 冒险**，寄存器写入在第四周期，而回写在下一周期，如果 load 的下一条指令需要访存出来的值，我们无妨通过转发的方式将正确的值送过去，这里我们采用的解决方案是在检测到 load-use 情况时加入一条“气泡”，ID 阶段刚结束的指令不进入 EX 阶段，而是暂停一周期，等待 WB 执行完得到正确的值。具体到代码上，就是 ID_EX_reg 的控制信号设为 flush，PC 与 IF_ID_reg 的控制信号设为 PAUSE，后面的为 NORMAL。load-use 的判定为：位于 EX 阶段的指令 mem2reg 为真，且写入地址与下一条指令的某一个寄存器地址相同。

```
module load_use_detecter (
 input          mem2reg_ex,
 input [4 :0]   rd_addr_ex,
 input [4 :0]   rs1_addr_id,
 input [4 :0]   rs2_addr_id,
 output         stall
  );
  assign stall = mem2reg_ex && ((rd_addr_ex == rs1_addr_id) || (
      rd_addr_ex == rs2_addr_id));
  endmodule
```

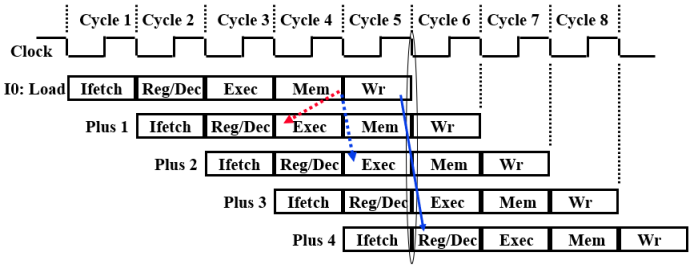


图 3: load-use 冒险的流水线状态

2.3 基础流水线调试与 DEBUG

至此，五段流水线的基本功能理论上实现完毕，不过整体逻辑、连线复杂，很容易出现各种 bug。DEBUG 方面，项目对实验十一提供的 cpu-batch 稍加修改即可适用于流水线。主要修改了 run 这个 task，原本 testbanch 会在检测到 $idataout == 32'hdead10cc$ 时退出，而流水线下有可能这条指令被 branch 冲刷掉，并没有执行完，因而需要继续执行四个周期，确认这四个周期都没有发生 branch。其他代码几乎不用改变，只需要把取到寄存器的链式引用改下，run 部分代码修改如下：

```
  task run;
    integer i;
    integer cont;
     begin
            i = 0;
            cont = 1;
```

```
            while(cont != 0) begin
                cont = 0;
                while( (idataout!=32'hdead10cc) && (i<maxcycles))
                begin
                    step();
                    i=i+1;
                end
                if(mycpu.pc_branch!=0) begin cont = 1; step(); end
                else begin
                    if(cont != 1) begin step(); if(mycpu.
                        pc_branch!=0) cont = 1; end
                    if(cont != 1) begin step(); if(mycpu.
                        pc_branch!=0) cont = 1; end
                    if(cont != 1) begin step(); if(mycpu.
                        pc_branch!=0) cont = 1; end
                    if(cont != 1) begin step(); if(mycpu.
                        pc_branch!=0) cont = 1; end
                end
            end
        end
    endtask
```

大多数 bug 还是出在转发与连线这两方面。

实验中，由于实验十一提供的 hex 代码是修改过的，通过官方 riscv-test 反汇编并不能得到，常常需要知道某串十六进制执行究竟是什么，因而编写了一个简单的 Python 脚本，其原理如下：首先编写一个只有一条指令的汇编文件，用 gcc 编译出.o 文件，脚本定位到这个指令的偏移位置，用输入的十六进制串替换它，再送给 objdump 截取汇编代码（通过 subprocess 实现）。脚本主要函数代码如下：

```
offset = 0x34
obj_src = bytearray(b"编译出的.o文件的二进制")

def decode(instr_str):
    obj = obj_src
    instr = bytearray(bytes.fromhex(instr_str))
    instr = instr[::-1] #小端
    obj[offset:offset+len(instr)] = instr
    obj = bytes(obj)
```

```
    output = open("tmp.o", "wb")
    output.write(obj)
    output.close()
    p = subprocess.Popen(["./riscv32-unknown-linux-gnu-objdump", '-d', '
        tmp.o'], stdout = subprocess.PIPE)
    msg = p.stdout.read()
    msg = msg.splitlines()[-1].decode('utf-8')
    os.remove("tmp.o")
    return msg
```

2.4 中断、CSR 寄存器与 M 拓展指令

2.4.1 流水线中断控制器 client

riscv 与 x86 的中断在核心思想上比较相近，不过具体实现上还是有颇多不同的。x86 的基地址存储在 idt 中，而 RISC 下有一个专用的 CSR 寄存器 MTVEC 用于存储基地址，根据官方手册第十章，RISCV 一共有 8 个控制状态寄存器（CSR），这里我们舍弃几个用不到的，保留 mtvec, mepc, mcause, mscratch, mstatus 五个，同时简化 mstatus 中的内容，由于我们处理非向量中断，且不嵌套，因而只保留其 mie 位（全局中断使能，第 4 位），同时也舍弃了 mie 寄存器。在标准中 CSR 寄存器地址 20 位有各种用途，这里全部设置为 0。

标准中断处理的流程如下 (经过简化)：

1. 异常指令的 PC 被保存在 mepc 中，PC 被设置为 mtvec。（对于同步异常，mepc 指向导致异常的指令；对于中断，它指向中断处理后应该恢复执行的位置。）
2. 根据异常来源设置 mcause，（并将 mtval 设置为出错的地址或者其它适用于特定异常的信息字。舍弃）
3. 把控制状态寄存器 mstatus 中的 MIE 位置零以禁用中断（并把先前的 MIE 值保留到 MPIE 中，这一过程舍弃）
4. （不做权限处理）

本项目中中断处理流程如下：

1. 中断控制器（client）检测到硬中断引脚有一个 1 且中断使能为 1 时进入中断预处理，暂停流水线、改变 int_state 状态机。
2. 根据 int_state 用状态机在几个时钟内完成相应的 csr 寄存器写入操作。软硬中断下保存 pc（如果正好在跳转，就保存处理后的跳转地址）到 mepc，然后依此保存 mstatus、mcause。mret 状态下，恢复 mstatus。
3. 将 int_flag 置 1，并给出 int_addr，流水线控制器探查到 int_flag 后进入中断相应状态，冲刷流水线，设置新的 pc，进入中断处理。

在 **hardware/cpu/client.v** 中，可以看到四个 always 语句。（代码较长这里就不贴了）

1. 第一个用组合逻辑控制 int_state 状态机
2. 第二个是 csr_state 的状态转移，默认在 CSR_STATE_IDLE 空状态，检测到 int_state 改变，保存当前 pc、mcause 并进入处理，有两种状态转移：
   (a) 软硬中断：*CSR_STATE_IDLE -> CSR_STATE_MEPC -> CSR_STATE_MSTATUS -> CSR_STATE_MCAUSE -> CSR_STATE_IDLE*
   (b) mret：*CSR_STATE_IDLE -> CSR_STATE_MRET -> CSR_STATE_IDLE*
3. 第三个根据 csr_state 进行 csr 寄存器的写入，因为一个周期只能写入一个寄存器，所以要通过这种方式实现，CSR_STATE_MEPC 状态下写 mpec 以此类推。
4. 第四个会在状态保存的最后一阶段发挥作用（即 CSR_STATE_MCAUSE 或者 CSR_STATE_MRET）将 int_flag 置 1，抛出应该跳转的 pc 值。

全局中断使能即 mstatus 寄存器的第三位。int_flag 会送给 pipeline_status 模块，生成相应的控制信号。

CSR 寄存器模块按照标准定义了几个控制寄存器，同时接收来自 pipeline 与 client 的读写命令，模块支持同时读操作（不过 client 并没有读操作），支持 cpu 和 client 同时写不同的寄存器，当二者写同一个寄存器时，会按照一定优先级，不过本项目中不应该出现这种情况。

000000000000	00000	000	00000	1110011	I ecall
000000000001	00000	000	00000	1110011	I ebreak
csr	rs1	001	rd	1110011	I csrrw
csr	rs1	010	rd	1110011	I csrrs
csr	rs1	011	rd	1110011	I csrrc
csr	zimm	101	rd	1110011	I csrrwi
csr	zimm	110	rd	1110011	I cssrrsi
csr	zimm	111	rd	1110011	I csrrci

图 4: 添加的几条指令

2.4.2 CSR 寄存器读写指令与 mret 指令

在操作系统（Kernel 下同）启动时，需要作中断注册，写入 mtvec 等 CSR 寄存器，因而还要支持 CSR 寄存器写入指令。项目实现了 ecall 指令，不过没有将输入输出处理成 ecall 的方式。

观察其指令形式，对比现有的指令执行过程，可以发现 csr 部分就是 I 型立即数，rs1 和 rd 正常解码，唯一有区别的就是 EX 模块和 WB 模块，EX 模块为了不改变原来的 ALU，项目中添加了专用的 CSR_ALU 与 csr_aluctr，WB 模块添加了 csr2reg，csr_we 两个控制信号。

根据手册，可以看出 CSR_ALU 需要三个输入：rs1，zimm，csr，输出的值被写入到某个通用寄存器中。下面是拓展后的控制信号（未提到的都为 0）：

指令	op[6:2]	func3	ExtOp	RegWr	csralu_ctr	csr2reg	csr_we
csrrw	11100	001	000	1	000	1	1
csrrs	11100	010	000	1	010	1	1
csrrc	11100	011	000	1	100	1	1
csrrwi	11100	101	000	1	001	1	1
cssrrsi	11100	110	000	1	011	1	1
csrrci	11100	111	000	1	101	1	1
ecall	11100	000	000	0	000	1	0

表 1: 拓展指令译码表

在 EX 模块中添加 CSR_ALU:

```
1 CSRALU csr_alu_instance(csr_data, rs1_proc, {27'd0, zimm}, csr_aluctr,
      csr_aluresult);
```

csraluctr	operation
000	rs1
001	zimm
010	rs1|csr
011	zimm|csr
100	r̃s1|csr
101	z̃imm|csr

表 2: CSR_ALU 控制信号

在 WB 模块添加 csraluresult 的选择：

```
1  assign busW = csr2reg ? csr_aluresult :(MemtoReg?dmemdata:aluresult);
```

流水线模块添加 csrs 和 client，按需修改几个流水段寄存器，完成中断的硬件支持。

2.4.3 时钟中断

项目中实现了一个时钟硬件，每 1 毫秒发出一次时钟中断。以时钟中断为例阐述整个中断流程：

1. Kernel 启动并完成中断注册，将全局中断使能置 1
2. 时钟满足触发条件 (这里为计数器小于等于 7) 且全局中断使能为 1，将中断引脚置为 1
3. client 检测到中断引脚，将全局中断使能置 0。保存几个 CSR 寄存器，然后将 int_flag 置 1.
4. 流水想控制器检测到 int_flag，冲刷流水线进入中断处理入口程序。
5. 中断处理入口程序取出 mscrach 的值（临时栈地址），保存若干通用寄存器到临时栈。将 mcause 作为函数参数调用中断处理程序。
6. 中断处理程序内执行处理流程。本项目中时钟中断时累加时间，并更新键盘输入。

7. 中断处理程序返回到中断入口程序，恢复通用寄存器，执行 mret 指令。

8. client 检测到 mret，进入 mret 执行，恢复 mstatus 等寄存器，将 int_flag 置 1，跳转回之前的 pc。

2.5 内存映射

硬件 IO 通过内存映射的方式进行，例如 VGA 将固定位置的内存作为显存，读取其中的 ascii 码进行显示。本项目中键盘没有采用复杂的处理，键盘会始终将按下的按键同步到内存中，支持字符与 ctrl、shift 同时按下，但不支持多个字符一起按下。内存映射模块在 **hardware/memory/memory_map.v** 中。内存分配如下：

起始地址	描述	读取模块	写入模块
0x00000000	指令 ROM	cpu	none
0x00100000	数据 RAM	cpu	cpu
0x00200000	VGA 控制信号，包含光标位置、滚屏行数	vga	cpu
0x00300000	VGA 字符 ASCII 码 RAM	vga	cpu
0x00400000	VGA 字符颜色 RAM	vga	cpu
0x00500000	键盘信息，包含按下的按键 ascii 码，与 ctrl、shift 是否按下等	cpu	keyboard
0x00700000	临时栈，在中断时使用	cpu	cpu
0x00800000	数码管与 LEDR	hardware	cpu
0x00900000	字符动画内容 ROM	cpu	none

表 3: 内存映射的分配

内存映射模块中，实例化所有存储器，根据地址的前缀将某一个存储器的写使能置一，读写口是分开的，cpu、数码管 LEDR、vga 的读取互相不冲突。内存映射模块有四个方向的读写需求：来自 cpu、vga、键盘和数码管 LEDR，因而有若干个各自的引脚。存储器方面较大的存储器（指令 ROM 和数据 RAM）采用 IP 核并且有专门的处理模块，支持 memop（与实验十一相同），键盘、VGA 控制信号等有实时读取需求，采用 reg 方式实现。

3 软件部分

硬件部分我们实现了底层，还需要操作系统来实现交互，这一部分就是完成操作系统内核的设计。这里我们采用 C 语言（以及少部分汇编与内联汇编）编写内核 Kernel，然后 RISCV32 编译环境进行编译得到 mif 文件。

在这个简易操作系统中，没有进程的概念，所有指令的执行都在内核态完成，虽然实现了 ecall，不过输入输出依然是直接调用函数而非通过 ecall 进行陷阱操作。

3.1 系统 API 函数

3.1.1 输出

输出。部分，为了方便缓冲的实现，输出采用 64*30 的大小，算上缓冲区一共 64*64 个字符，方便用位移快速运算。正如前面所说，有一块专门的显存用于存储 ascii 码，cpu 只负责写，vga 只负责读，在 **include/global.h** 中，定义了很多常量，包括各个内存区的地址前缀，显存分为两块，一个字符、一个颜色，在 syscall.c 中定义显存数组指针：

```
uint8_t *const ch_mem = (uint8_t *)VGA_CHAR_OFFSET;
uint8_t *const color_mem = (uint8_t *)VGA_COLOR_OFFSET;
```

写显存时，只需要进行普通数组操作即可。

输出时，有三个控制输出状态的变量：monitor_write_cursor 记录当前写入点在 64*64 的显存中的位置，output_front_cursor 记录退格能够退到的最前位置（例如不能删去命令提示符 $），VGAINFO 类型的结构体 vga_info，用于记录光标位置、滚屏的行数，通过 *p_vga_info = vga_info; 的方式写入。

最底层输出函数 _setc，向 addr 写入 color 颜色的 c：

```
inline static void _setc(const char c, const uint8_t color, const
    uint32_t addr)
 {
    ch_mem[addr] = c;
    color_mem[addr] = color;
 }
```

第一层封装 putc，对换行、退格、tab 做了特殊处理，并且在输出之后会判断是否需要滚动屏幕：

```
void putc(const char c, const uint8_t color)
{
    if (c == '\n')
    {
        monitor_write_cursor = (monitor_write_cursor / COL_CNT) * (
            COL_CNT) + COL_CNT;
        if (monitor_write_cursor == TOTAL_CHAR + vga_info.extra_line_cnt
            * COL_CNT)
            scroll_screen();
        if (monitor_write_cursor == TOTAL_CHAR_WITH_BUF)
            monitor_write_cursor = 0;
    }
    else if (c == '\t')
        print("   ", COLOR_WHITE);
    else if (c == '\b')
    {
        if (output_front_cursor < monitor_write_cursor)
            _erasec(--monitor_write_cursor);
    }
    else
    {
        _setc(c, color, monitor_write_cursor++);
        if (monitor_write_cursor == TOTAL_CHAR)
            scroll_screen();
        if (monitor_write_cursor == TOTAL_CHAR_WITH_BUF)
            monitor_write_cursor = 0;
    }
    _update_cursor();
}
```

再一层封装 print，输出 color 颜色的字符串 str，以及辅助函数 error。

```
 void print(const char *str, const uint8_t color)
{
    char c = *str;
    while (c)
    {
```

```
        putc(c, color);
        ++str;
        c = *str;
    }
}

inline void error(const char *msg)
{
    print("Error: ", COLOR_RED);
    print(msg, COLOR_WHITE);
}
```

lock_output_front 函数用于锁定当前位置为最前位置，不允许在此位置继续退格。

```
inline void lock_output_front() { output_front_cursor =
    monitor_write_cursor; }
```

3.1.2 输入

由于项目采用中断的方式处理键盘输入，因而 Kernel 无需循环访问键盘状态，我们在 kernel 中维护了一个 KBInfo 结构，它会在时钟中断中被更新，kb_info 被声明为 volatile，以便告诉编译器它随时会被修改：

```
 typedef struct KBInfoStruct
{
    uint8_t c;
    union
    {
        struct
        {
            uint8_t is_shift :1;
            uint8_t is_ctrl :1;
            uint8_t is_caps :1;
            uint8_t is_special :1;
            uint8_t is_error :1;
            uint8_t unused :3;
        };
        uint8_t flags;
```

```
    };
} KBInfo;
exten volatile KBInfo kb_info;
```

为了判断当前 kb_info 中的字符是否已经被取用过了，还需要一个 InputController，记录上一个读取的字符以及读取时间：

```
  typedef struct InputControllerStruct
{
    uint8_t pre_c;
    uint32_t input_time;
} InputController;
```

一个字符被 getc 读取时，input_controller 被更新，另外时钟中断时如果字符发生了变化，也会更新它，这样就可以实现 getc 函数执行时按键按下立刻响应，同一按键保持按压时每隔 *KEY_REPEAT_INTERVAL* 响应一次：

```
static inline bool _next_key_arrived()
{
    if (kb_info.c)
    {
        if (kb_info.c == input_controller.pre_c)
            return sys_time - input_controller.input_time >
                KEY_REPEAT_INTERVAL;
        else
            return kb_info.c != '\0';
    }
    else
        return false;
}
```

有了 next_key_arrived 函数的帮助，getc 函数也就很好写了，getc 的同时还要回显，即调用 putc

```
char getc()
{
    while (!_next_key_arrived())
        ; //wait input
    char c = kb_info.c;
    input_controller.pre_c = c;
    input_controller.input_time = sys_time;
```

```
    putc(c, COLOR_WHITE);
    return c;
}
```

向上封装一层 getline，将一行的输入计入 buf，需要注意退格的处理：

```
 void getline(char *buf)
{
    char c;
    int i = 0;
    while ((c = getc()) != '\n')
    {
        if (c == '\b')
        {
            if (i > 0)
                buf[--i] = '\0';
        }
        else
        {
            buf[i++] = c;
        }
    }
    buf[i] = '\0';
}
```

wait_ms 函数，延迟 n 毫秒，因为 kenel 中的 sys_time 是在中断中更新的，因而这个函数只有一句 while：

```
void wait_ms(uint32_t ms)
{
    uint32_t start_time = sys_time;
    while (sys_time - start_time < ms);
}
```

kernel/src/syscall.c 中还有一些其他函数，原理都比较简单，根据函数名和代码很好理解，不做赘述了。

3.2 实现常用 std 函数

Kerne 的编写中需要用到很多函数，由于不引入标准库，需要自己实现，如 strcmp，strtok，itoa，atoi，___mulsi3，___udivsi3，实现上参考标准实

现，其实并没有什么可说的，程序中考虑到除法和取模往往结伴出现，而除法和取模的程序实际上是相同的，因而提供了一个一次得到除法与取模结果的函数：

```
1 void _udiv_mod(uint32_t a, uint32_t b, uint32_t *res, uint32_t *remain)
```

3.3 中断注册与处理

在操作系统启动时需要将中断入口程序的 pc 值设置到 mtvec 寄存器中，并开启全局中断使能。这一部分的代码设计参考了 pa i386 中断的设计思路。代码在 **kernel/src/irq.c** 中

中断注册：在 init_CSR 函数中，使用三条内联汇编设置三个 CSR 寄存器，内联汇编使用 gcc 拓展语法，第一个字符串为若干条指令，用分号隔开，第一个冒号后面为输出声明，第二个冒号为输入声明，使用代号传输参数，”r” 表示让 gcc 选择合适的方式存储参量：

```
1  void init_CSR()
2  {
3      uint32_t addr = (uint32_t)(int_handler_entrance);
4      uint32_t tmp;
5      // set the address of the interrupt handler
6      asm volatile("mv %[tmp], %[src];"
7                   "csrw mtvec, %[tmp];"
8                       :
9                       : [src]"r"(addr), [tmp]"r"(tmp));
10     asm volatile("csrw mscratch, %[tmp];"
11                      :
12                      : [tmp]"r"(TMP_STACK_OFFSET));
13     // enable global interrupt
14     asm volatile("csrw mstatus, %[tmp];"
15                      :
16                      : [tmp]"r"(0x00000008));
17 }
```

中断处理入口函数：由于我们的硬件是非向量中断处理方式，只有一个入口函数，在入口中使用 mcause 寄存器判断中断类型，采用 RSICV 手册的标准，时钟中断号 0x80000007，ecall 中断号 0x0000000b，如果需要 ecall

还涉及到对寄存器的约定，这里不做。入口函数因为不能改变寄存器，不能使用内联汇编，因而直接采用汇编编程，写在 **kernel/src/do_irq.S** 中：

```
.global int_handler_entrance
.extern irq_handler
int_handler_entrance:
    csrrw t6, mscratch, t6
    nop
    nop
    sw a0, 0(t6)
    sw a1, 4(t6)
    sw a2, 8(t6)
    sw a3, 12(t6)
    sw a4, 16(t6)
    sw a5, 20(t6)
    sw a6, 24(t6)
    sw a7, 28(t6)
    sw x1, 32(t6)

    csrr a0, mcause //load parameter
    call irq_handler

    lw a0, 0(t6)
    lw a1, 4(t6)
    lw a2, 8(t6)
    lw a3, 12(t6)
    lw a4, 16(t6)
    lw a5, 20(t6)
    lw a6, 24(t6)
    lw a7, 28(t6)
    lw x1, 32(t6)
    csrrw t6, mscratch, t6
    mret
```

这段代码 extern 了一个实际处理函数 irq_handler，然后使用 global 导出了入口函数 int_handler_entrance，之所以有两条 nop 是因为这里涉及到 load-use 但我不想再添加 csr 的转发逻辑了。中间的 call 是一条伪指令，根据 riscv 的函数调用规则，第一个参数存在 a0 寄存器中。mscrach 中存着临时栈的位置，在这个栈中进行读写不会对原来的数据 RAM 造成影响。

实际处理函数 irq_handler：

```
void irq_handler(uint32_t mcause)
{
    if(mcause == CLOCK_INT_MCAUSE)
    {
        //clock interrupt
        sys_time++;
        KBInfo tmp_info = *p_kb_info;
        if(kb_info.c != tmp_info.c)
        {
            input_controller.pre_c = kb_info.c;
            input_controller.input_time = sys_time;
            kb_info = tmp_info;
        }
    }
    else if(mcause == ECALL_MCAUSE)
    {

    }
}
```

3.4 ui_mainloop 交互主函数

写到这里，工作已经基本完成了，下面的任务就只是用提供的结果完成交互的设计了。虽然没完全搞懂 PA AM 的全部内容，不过在设计上还是尽可能地以接口的方式分离硬件与软件，下面不再涉及到硬件直接交互，只有接口的调用。

ui_mainloop 函数中首先 getline，然后使用 strtok 进行参数分割，使用 strcmp 进入相应的交互函数。中间夹杂了两个 lock_output_front 用于锁定输出前线

```
void ui_mainloop()
{
    while (true)
    {
        print("$ ", COLOR_WHITE);
        lock_output_front();

```

```
        char line[64];
        getline(line);
        lock_output_front();

        char *cmd = strtok(line, " ");
        if (strcmp(cmd, "hello") == 0)
            hello();
        else if (strcmp(cmd, "time") == 0)
            time();
        else if (strcmp(cmd, "fib") == 0)
            fib();
        else if (strcmp(cmd, "calc") == 0)
            calc();
        else if (strcmp(cmd, "marquee") == 0)
            marquee();
        else if (strcmp(cmd, "") == 0)
            continue;
        else
            error("Command not found: ");
    }
}
```

fib 和 hello 没什么可说的，time 函数实现了实时刷新直到 ctrl+c 时退出，代码上每个循环节先退格若干次删去上次的输出（因为已经调用了 lock_output_front，因而退格多输出几个也无所谓），然后使用 itoa 输出 sys_time，接着延迟一毫秒进入下一次循环，循环条件判断 ctrl+C 是否按下：

```
void time()
{
    char s[20];
    while (!is_ctrl_c())
    {
        print("\b\b\b\b\b\b\b\b\b", COLOR_WHITE);
        itoa(sys_time, s);
        print(s, COLOR_WHITE);
        wait_ms(1);
    }
    putc('\n', COLOR_WHITE);
```

```
12 }
```

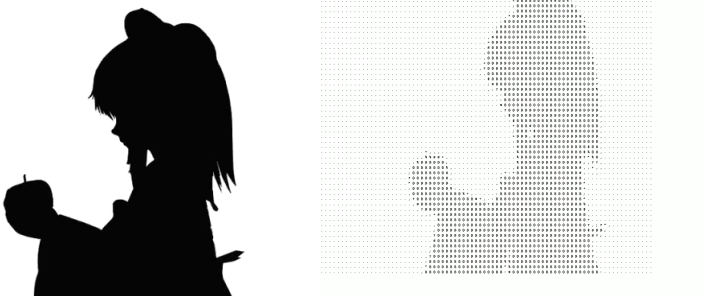
3.5 移植表达式求值 实现跑马灯

在 PA2-3 中我们实现了 monitor，其中包含了表达式求值功能，只不过里面用到了正则表达式 regex.c，为了规避正则的使用，规定本项目中表达式的数字、符号之间应该用空格隔开，这样就可以使用 strtok 进行读入，其他部分对 PA 的代码稍加修改即可，具体代码参见 **kernel/src/expr.c**，较长这里不贴了。通过预处理将字符串处理为各个 token，并区分单目运算符、双目运算符，匹配括号，计算逻辑为寻找主运算符，递归计算两边的值后作用该运算符，最后可以实现基本所有常见表达式计算。

跑马灯的实现在上面的接口铺垫下也比较好实现，六个七段数码管是一个 32 位整数数组（只有低 7 位有用，代表数码管的某一根线亮暗），ledr 的控制为一个 32 为整数，第 10 位控制 10 个 LEDR 亮暗。读写函数在 **kernel/src/syscall.c** 中。marquee 函数每 500ms 改变一次数码管与 LEDR 的值，数码管从 0 到 F 改变，ledr 从右往左逐个点亮，循环直到 ctrl+C 按下。

3.6 字符动画

这一部分的功能基于我去年写的一个脚本，可以将视频转换为字符动画，效果类似于：



(a) 原视频　　(b) 处理后

图 5: 字符动画效果图

本项目中的 VGA 显示器支持 64*30 的字符显示，对于字符动画来说勉强够用，具体的转换脚本参见代码，基本原理是用 opencv 导入视频，逐帧处理成灰度图，然后按灰度将一小块像素替换为**.:+*?%#@** 中的一个。

字符串压缩：脚本会将每一帧画面处理成一个 64*30 的字符串，我们还需要一种算法将这个字符串存储到内存中，直接存储肯定是不行的，内存占用太大，因而这里采用一个很简单的字符串压缩算法，很自然的想法是形如"aaabbccccdddd" 的字符串可以压缩为"a3b2c4d4" 的形式，这里稍作约定，用一个 Byte 存储一个结构，低 3 位为字符（以下标形式对应".:+*?%#@"），高 5 位为该字符的数量（最多为 31 个，超过了就记到下一个字节）。这个算法大幅减小了空间占用，可以达到 9% 的压缩率，只需 128KB 空间就可以显示约 30s 的字符动画，同时也非常方便读取，我们在代码中只需要一个接一个读入字节，用循环输出即可。

双缓冲技术：这是一种在游戏引擎中常常使用的技术，如果我们在打印字符时一直在一块 64*30 的区域上更改，那么经常出现一帧仅仅输出了一半就被显示，造成两帧画面的混合。双缓冲是指使用 "两块" 显示屏，一块用于展示，在另一块上准备下一帧，准备好了再切换屏幕，这样可以确保显示出的画面是稳定的。我们的硬件实现也能够支持这种技术，因为显存是 64*64 的，只要将 0 行和 32 行分别作为两块显示屏的开始行即可。在 syscall.c 中增加几个新的接口 *putc_buffer, switch_screen, switch_mode* 分别用于向缓冲区输出字符，切换两块缓冲区、在命令行状态与动画状态切换。

一个奇怪的 bug：不知道是脚本、Kernel、硬件中的哪一个有点问题，不做特殊处理下，硬件播放字符动画十几帧就会卡住，在将所有输出接口改为强制 inline 之后，可以顺利放完了，不过播放十秒左右又会出现错位现象。bug 的触发位置是稳定的，不像是中断引发的，而 riscv 官方测试集又是全部通过，流水线 cpu 应该也没有问题。

最后为了解决这个 bug，只能做一些特殊检查，生成字符串时，规定每一帧的第一个字节为零，在 Kernel 中读取显存时，如果发现第一个字节不为 0，就递减指针直到该字节为 0，然后正常读取输出。

如果需要编译我的项目查看字符动画的话，需要将 global.h 的 **#define inline ___attribute((always_inline))** 取消注释，进行 kernel 的编译。

CharDance 函数主体代码如下：

```
1 char type2char[] = {'.',':','+','*','?','%','#','@'};
```

```
2  uint32_t ms_per_frame = 33;
3  typedef struct
4  {
5      uint8_t type :3;
6      uint8_t num :5;
7  } VideoChar;
8  void chardance()
9  {
10     switch_mode(VIDEO_MODE);
11     VideoChar *p = (VideoChar *)VIDEO_OFFSET;
12     volatile uint32_t cnt_out = 0;
13     char c;
14
15     while (!is_ctrl_c())
16     {
17         uint32_t frame_start = sys_time;
18         cnt_out = 0;
19         if(*(uint8_t*)p != 0)
20         {
21             while(*(uint8_t*)p != 0)
22                 --p;
23         }
24         ++p;
25         while(cnt_out < 30*64)
26         {
27             // prepare next frame
28             volatile VideoChar vc = *p;
29             c = type2char[vc.type];
30             ++p;
31             cnt_out += vc.num;
32             for(int i = 0; i < vc.num; ++i)
33             {
34                 putc_buffer(c, COLOR_WHITE);
35             }
36         }
37         wait_ms(ms_per_frame - (sys_time - frame_start));
38         switch_screen();
39     }
40
```

```
41      switch_mode(CMD_MODE);
42  }
```

4 参考及实验感悟

项目 Github 地址：
https://github.com/WLLEGit/riscv-project

1. RISC-V-Reader-Chinese
2. RISC-V 特权指令和 CSR
3. cpu architecture - When an interrupt occurs, what happens to instructions in the pipeline? - Stack Overflow
4. 从零开始写 RISC-V 处理器【5】硬件篇（3-完）
5. quartus ii 增量编译
6. Understanding RiscV objdump - Stack Overflow