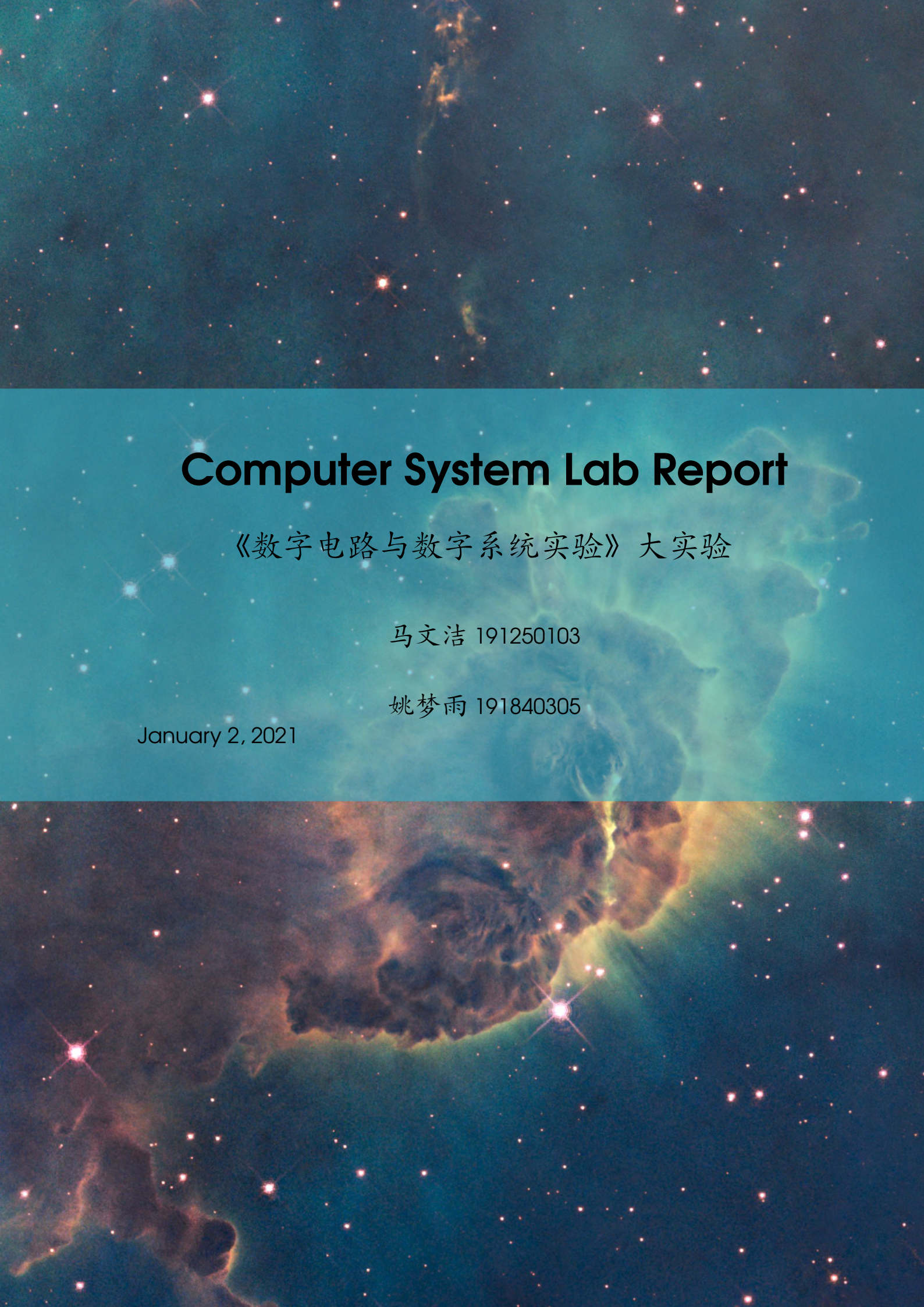


Computer System Lab Report

《数字电路与数字系统实验》大实验

马文洁 191250103

姚梦雨 191840305

January 2, 2021

Fall term 2020, Nanjing University

This project was done by two students together and obtained acceptance by Teacher Wang on December 28, 2020.

Completed in December 2020

Contents

List of Figures

List of Tables

Listings

CHAPTER 1

Overview

在本次实验中，我们利用 Quartus 和 DE-10 开发板，完成了 5 级流水线 CPU 的设计；在此基础上，结合输入输出，利用 CPU 运行一些简单的软件，实现了一个基础的命令行程序。

我们将对软件和硬件部分分别进行介绍，并且在附录 A 和 B 中分别给出了项目目录结构以及使用的一个脚本。

以下是实现的功能概览。

1.1 硬件部分

1. 实现了五级流水线 *CPU*。运用流水线 CPU 的基本原理，并在时序上做了一些改动，实现了取值-译码-执行-访存-写回；
2. 实现了 ALU、仿存和跳转/分支的基本指令；
3. 键盘、VGA 和音频之间的协调工作；
4. 在数据 RAM 中对数据段，输入输出段以及栈空间进行合理划分；
5. 待补充

1.2 输入输出部分

1. 利用键盘向 CPU 传送单个字符，以及 shift 或 caps 键与单个字符组合的大写/符号字符；
2. 在屏幕上显示 CPU 计算出的需要输出的字符，实现了显示光标，区分颜色以及屏幕上移等功能；
3. 实现了 CPU 控制音频模块播放不同的音乐；

1.3 软件部分

1. 通过 MIPS 汇编编写并生成机器码作为指令代码段文件;
2. 实现了与硬件接口之间的交互，轮询接口状态，从键盘缓冲器中读取字符并判断跳转至命令分析阶段;
3. 命令分析阶段完成了字符串的匹配并跳转执行相关子程序，执行结果返回硬件;
4. 子程序包括:

- 输入 hello: 显示 HELLO 艺术字;
- 输入 help: 显示命令提示信息;
- 输入 fib: 输入十六进制数字 n，显示屏显示第 n 个 Fibonacci 数的十六进制形式;
- 输入 piano: 输入数字 $i \in [1,3]$, 播放第 i 首歌曲;
- 输入 display: 输入十六进制数字，七段数码管上显示对应数字;
- 输入 LEDon: 输入数字 $i \in [0,9]$, 打开第 i 个 LED;
  输入 LEDoff: 输入数字 $i \in [0,9]$, 关闭第 i 个 LED;
- 输入 restart: 重新启动并恢复初始状态;
- 输入未知命令: 显示未知命令提示 Unknown Command;

CHAPTER 2

Division of Labor

本实验由两位同学合作完成，许多模块都是共同完成，大致任务分配如下所示：

2.1 姚梦雨同学

- 完成 CPU 的译码模块的编写，对 ALU 模块进行后期修改，以及对整个 CPU 模块的综合；
- 完成汇编指令的软件部分的编写；
- 完成音频模块的编写，实现了按照命令播放不同歌曲的功能；整合了键盘模块的功能，实现了向 CPU 传送字符的功能；
- 完成对于七段数码管以及 LED 的控制。
- 共同完成最后的测试-调试任务。

2.2 马文洁同学

- 完成 CPU 的取指令、pc 跳转模块的编写，以及 ALU 模块的初期编写，对 CPU 各个模块进行整合；
- 完成汇编指令的系统部分和输入输出部分的编写，完成调试阶段需要的多个测试汇编的编写；
- 完成 VGA 模块的编写，实现了控制字符颜色，屏幕位置，显示命令提示符以及光标等功能；
- 完成对于欢迎界面的设计，以及对于 CPU 中的数据 RAM 的设计与生成；
- 共同完成最后的测试-调试任务。

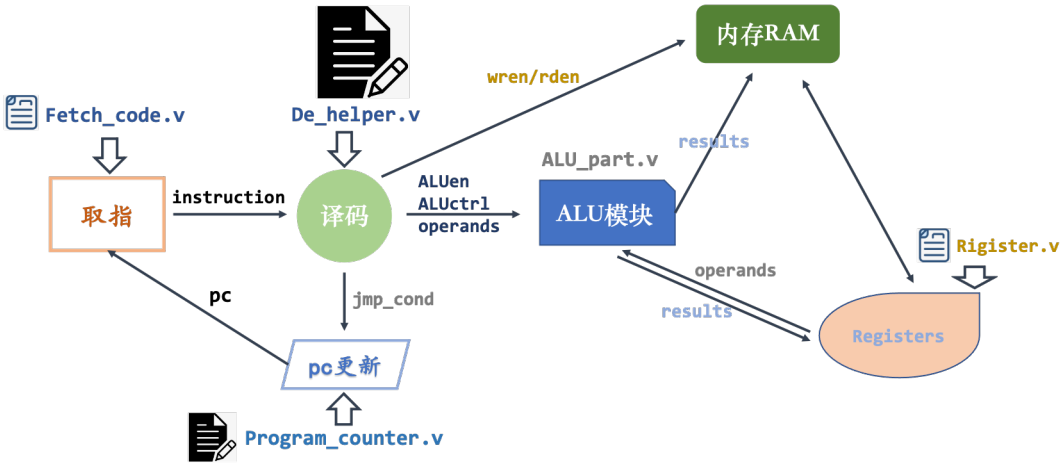
CHAPTER 3

Hardware

3.1 流水线 CPU 的实现

五级流水线 CPU 的基本思想即为，将 CPU 的工作分为：取指-译码-执行-访存-写回一共 5 个阶段，在我们的实验中，我们对于 CPU 的实现的流程图如下所示：下面

Figure 3.1: 五级流水线 CPU 流程图.



就分阶段地对于 CPU 的实现进行介绍。

3.1.1 取指令

取指令阶段是在 `Fetch_code.v` 中实现。这一阶段是整个 CPU工作的基础，只有在指令被正确取出的前提下，才能取考虑之后的操作是否正确。为了减轻 CPU 中数据 `RAM` 的工作负担，也避免时序上与预期产生偏差，我们将代码段（MIPS 汇编生成的机器码）存储在一个单独的 `ROM` —— `instrtable` 中。

`Fetch_code` 模块中，输入为时钟信号 `clk` 和一个相对的 pc 值 `instr_addr`，输出为在这个时钟周期需要执行的 32 位指令的机器码。之所以称作是相对的 pc 值，

是因为在 Mars 汇编器中，指令的编号默认从 0x400000 开始，而 `instrtable` 中的指令却默认从 0 开始，所以传进来的 pc 与真实的 pc 相差一个固定的值。

`instrtable` 使用生成的指令机器码 `overall.txt` 进行初始化。在实际的执行中，一个指令占据 4 个字节，所以 pc 指针一定是 4 的倍数，而在本模块的 `ROM` 中，因为一个寄存器即为 32 位，所以指令的真正地址是 $\frac{pc}{4}$. 故使用 `instr_addr[11:2]` 进行寻址，得到对应的指令代码传出即可。

代码如下所示：

```
module Fetch_code(
    input clk,
    output [31:0] instr,
    input [13:0] instr_addr
);
reg [31:0] instrtable [1023:0];
initial
begin
    $readmemh("overall.txt", instrtable);
end
wire [9:0] pc_in_rom;

assign pc_in_rom = instr_addr[11:2];
assign instr = instrtable[pc_in_rom];
// 一个指令4个字节，除以4以后取指令，地址为10位数
endmodule
```

Listing 3.1: module Fetch_code.

3.1.2 指令译码

取完指令后，得到一个指令的机器码 `instr` ，下面就需要对指令进行译码。在我们的实验中，本身使用到的指令较少，所以译码阶段也做了简化。在对 CPU 进行测试的过程中，我们发现按照之前的实现方式，实际执行的结果与预期结果有差距，所以对译码阶段进行修改，因此出现了和手册中不相符的一些译码。这样做只有在简单的实验，只包含简单指令的程序之中才能正确工作，若涉及大型程序及实验，需要更加系统的设计。

译码的功能主要在 `De_Helper.v` 中实现。本模块的主要输入输出引脚如下所示：

```
module De_Helper( // only the decode part, no valid signal here
        input [31:0] instr,
        output reg [3:0] jmp_cond,
        output reg RorI, // 0:R type; 1:I type(use immediate). If not used, I as a
            ↪ default type
        output [4:0] rs,
        output [4:0] rt,
        output [4:0] rd,
        output [15:0] imm,
        output [4:0] shamt,
        output reg [3:0] ALUctrl,
        output reg ALUen,
        output reg Memtoreg, //1:load from mem
        output reg Memwr, //1:load to mem
        output [25:0] pcimm
);
```

Listing 3.2: input/output of decode.

在输入的引脚中，`instr` 较为显然，是从上一阶段得到的需要译码的指令机器码。译码阶段不需要时钟信号，只要传进来的机器码有所变化，就会再一次执行所有操作。

输出 `jmp_cond` 表示跳转状况，这一输出主要为了 `program_counter` 模块服务，我们对不同的跳转方式进行编码，这些编码在宏模块 `ALU.v` 中给出：

```
‘define NOP 4’b0000
‘define BLEZ 4’b0001
‘define BEQ  4’b0010
‘define BNE 4’b0011
‘define J   4’b0100
‘define JAL 4’b0101
‘define JR 4’b0110
```

Listing 3.3: Type of jmp_cond.

其中 `NOP` 表示不需要跳转，在这种情况下 pc 直接加上 4 即可。剩下的几种条件和相应的指令对应起来，代表着几种不同的跳转情况，这在 `program_counter.v` 中有具体的实现。

第二个输出参数 `RorI` 就像注释里说的那样，表示这一条指令是否为 I type，如果是的画该输出为 1，否则输出为 0。

输出引脚 `rs` ， `rt` ， `rd` 分别表示两个操作数寄存器的编号和目的寄存器的编号，这个直接从指令中取出即可； `op` 表示操作码， `func` 表示 `ALU` 操作的类型， `imm` 表示 I type 指令的操作数，而 `pcimm` 表示 pc 直接跳转时所需要的参考地址，这一输出将在 `program_counter.v` 中使用。 `shamt` 与手册中的定义一致，用于移位操作。以上这些参数只需要直接从传入的机器码中取出即可。

如下所示：

```
assign op  = instr[31:26];
assign func = instr[5:0];
assign rs  = instr[25:21];
assign rt  = instr[20:16];
assign rd  = (op == 6'h3) ? 31 : instr[15:11];
assign imm = instr[15:0];
assign pcimm = instr[25:0];
assign shamt = instr[10:6];
```

Listing 3.4: Basic decoding.

ALUen和ALUctrl主要用于 `ALU_part.v` 模块的运算，它代表是否需要ALU计算以及具体的计算类型。

Memtoreg和Memwr分别代表是否是从内存写入寄存器，以及是否需要写入内存。

下面以几个具体的指令译码为例子，解释一下本模块的具体实现。当 `op` 为 0 时，大部分为 `ALU` 操作：

```
case(op)
6'd0:
///////////////R-type//////////////////////
begin
        if(func == 6'h08)///jr 寄存器
        begin
                RorI<=1'b0;
                ALUen <= 1'b0;
                Memtoreg <= 1'b0;
                Memwr <= 1'b0;
```

```
11                  jmp_cond <= 'JR;
12          end
13          else
14          begin
15                  RorI <= 1'b0;
16                  ALUen <= 1'b1;
17                  Memtoreg <= 1'b0;
18                  Memwr <= 1'b0;
19                  jmp_cond<='NOP;
20  ////////Judge the func////////
21                  case(func)
22                  6'h20: ALUctrl <= 'ADD_op;
23                  6'h21: ALUctrl <= 'ADDU_op;
24                  6'h22: ALUctrl <= 'SUB_op;
25                  6'h23: ALUctrl <= 'SUBU_op;
26                  6'h27: ALUctrl <= 'NOR_op;
27                  6'h2a: ALUctrl <= 'SLT_op;
28                  6'h2b: ALUctrl <= 'SLTU_op;
29                  6'h00: ALUctrl <= 'SLL_op;
30                  6'h03: ALUctrl <= 'SAR_op;
31                  endcase
32          end
33  end
```

Listing 3.5: R type ALU.

在 `op` 为 0 的时候，需要对 `func` 进行判断来译码。

当func为 0 时，这个时候为 `jr` 指令，那么自然：ALUen为 0，同时是 R type，不需要写入寄存器，也不需要读取/写入内存。这时将jmp_cond设置为 `'JR` 。

除此之外，在其它的func条件下，均为寄存器类型的ALU操作，所以首先设置 `ALUen <= 1` ；同时因为是寄存器类型，所以 `RorI` 为 0，也不需要操作内存。ALU 操作主要执行运算，所以也不需要跳转，跳转条件为 `'NOP` ；

接下来只需要根据对应的func对计算的类型进行赋值即可。ALU的不同类型同样在文件 `ALU.v` 中给出。

当然，ALU 类型的指令除了寄存器类型以外，还有立即数类型，以下是一个 I type 的

ALU 运算的指令译码例子：

```
6'h8: begin
    RorI <= 1'b1;
    ALUen <= 1'b1;
    Memtoreg <= 1'b0;
    Memwr <= 1'b0;
    ALUctrl <= 'ADD_op;
    jmp_cond<='NOP;
end
```

Listing 3.6: I type ALU.

在这种情况下，与上一种不同的仅仅在于，RorI被赋值为高电平，目的是当进行计算时，选取立即数作为第二操作数而不是寄存器 `rt` 。

以上主要为 `ALU` 类型的指令，下面介绍访存类型的指令译码。

在涉及内存的指令译码中，最关键的即为两个引脚 `Memtoreg` 和 `Memwr` ：

```
6'h23: begin
    RorI <= 1'b1;// rt 作地址
    Memtoreg <= 1'b1;
    Memwr <= 1'b0;
    ALUen <= 1'b0;
    jmp_cond<='NOP;
end
6'h2b: begin
    RorI <= 1'b1;
    Memtoreg <= 1'b0;
    Memwr <= 1'b1;
    ALUen <= 1'b0;
    jmp_cond<='NOP;
end
```

Listing 3.7: Mem type decoding.

很容易知道，以上两个指令分别为 `lw` 和 `sw` ，分别表示从寄存器写入内存，和

从内存写入寄存器。

对于跳转类型的指令，最重要的是给跳转条件赋予正确的类型，同时，对于该跳转地址计算所需要的参数，也需要译码得出。例如直接跳转类型：

```
6'h2: begin  //j
    RorI <= 1'b0;
    ALUen <= 1'b0;
    Memtoreg <= 1'b0;
    Memwr <= 1'b0;
    jmp_cond <= 'J;
end
```

Listing 3.8: Jump type decoding.

3.1.3 指令执行

正如前面所分析的那样，我们并没有写一个专门的执行模块，因为对于不同类型的指令，它的执行操作不一样，执行方式也不一样。

指令类型有计算,跳转和访存等,所以我们在 `ALU_part.v` 模块进行计算,在 `program_counter.v` 模块中进行 pc 跳转。

对于计算模块，即 `ALU_part.v` ，并没有什么需要解释的，只需要按照译码的类型进行相应的计算即可。

下面来介绍一下 pc 模块。

```
wire [31:0] seq_pc,signed_offset;
initial begin
    pc_state = 32'h400000;
end
assign signed_offset = {{14{J_branch_offset[15]}},J_branch_offset,2'b0};
assign seq_pc = pc_state + 4;
always @ (posedge pc_clk)
begin
    if (reset) pc_state <= 32'h400000;
```

```
    else begin case (jmp_type)
        ......//根据不同的跳转类型给pc赋予对应的值，此处不再赘述。
        default : pc_state <= seq_pc;
        endcase
    end
end
```

Listing 3.9: program counter.

`program.v` 模块主要负责指令结束后 pc 的跳转位置。根据代码可以看出，我们对于 pc 的控制取决于传入的 `jmp_cond` 和 `cmp_res` ，对于每一种跳转条件均给予考虑。在分支跳转的情况中，我们需要考虑比较结果，如果满足，则 pc 的位置跳转到 `seq_pc` 加上带符号偏移量的位置，否则顺序执行。对于直接跳转，则直接跳转到指定的位置。对于不需要跳转的情况，则 pc 变化为 `seq_pc` 。

需要注意的一点是，对于给出的关于 pc 跳转的立即数，在分支跳转中是有符号数，且所有的立即数都需要左移两位。

3.1.4 内存与数据 RAM

我们使用 RAM 来实现内存的功能，并通过它来实现访存功能。下表展示了在内存中数据段的划分方式。

Table 3.1: 数据段的划分.

地址空间	存储内容	详细
0x0-0x500	键盘数据存储空间	键盘 RAM, A port 输入
0x500-0x1000	欢迎界面的信息	事先存储，.mif 初始化
0x1000-0x1500	一些帮助信息	.mif 初始化
0x1500-0x1700	一些指令的输出信息	事先存储
0x1700-0x2000	指令名称	用于 `_strcmp` 阶段进行比较
0x2000-0x3000	软件计算信息	CPU 计算过程中使用
0x3000-	堆栈空间	CPU 运行过程中的栈空间

对于数据RAM，我们使用真双口RAM，A port 留给键盘与 CPU 进行交互，一部分留给 CPU 对于内存进行读写。

3.1.5 寄存器模块

在寄存器模块中，完成了最后的写回阶段。Registers.v 中使用 32 个寄存器实现，主要代码如下：

```
reg [31:0] regs_all [31:0];
integer i;
initial
begin
    for (i = 0; i <= 31; i = i + 1)
    begin
        regs_all[i] = 0;
    end
end
always @ (posedge clk)
begin
    if (wr_en) regs_all[wr_addr] <= wr_data;
end
```

Listing 3.10: Register.v 主要实现.

3.2 指令实现

本实验中，我们主要实现了如下所示的指令：

Table 3.2: 实现的指令.

指令类型	指令罗列
运用 ALU 计算	add addi sub subi and andi or ori xor xori slt sll sra
访存	lw sw
分支/跳转	beq bne ble blez jal j jr

以上指令的执行过程主要是按照上一模块中介绍的五个阶段取执行，在 CPU_overall.v 中，我们对于以上的五个阶段进行了整合，下面是对于一些关键引脚的赋值：

```
assign rtnum = RorI ? imm : rd_data2;
assign rsnum = rd_data1;
assign wr_data = ALUen ? rdnum :
                 jmp_cond == `JAL ? pc + 4 :
                 Memtoreg ?mem_fetch : // mem_fetch是从内存中取出的
                 0;
assign wr_en = (ALUen | jmp_cond == `JAL | Memtoreg); // 是否需要写入寄存器
assign mem_addr = rd_data1 + imm; // 但是实际上，imm用的都是0
assign mem_save = rd_data2; // 写入内存;
assign wrmem_en = (Memwr) & (~Memtoreg); // CPU是否需要写入内存，用于B port
assign rdmem_en = ~ wrmem_en;
assign wr_addr = (Memtoreg|RorI) ? rt : rd;
assign cmp_res = (rsnum == rtnum) ? 0 : (rsnum < rtnum) ? 1 : 2;
```

Listing 3.11: CPU 主模块对于一些引脚的分配.

这些引脚会传入各个模块。

其中 `wr_en` 和 `wr_data` 对应的是寄存器模块，表示是否有寄存器需要写入，以及具体写入的值为多少。

而 `mem_addr` 和 `mem_save` ， `wrmem_en` 则对应的是内存，表示内存是否需要写入，写入的地址，写入的值具体是多少，都会通过 CPU 计算出。

`rt_num` 和 `rs_num` 分别表示传给 ALU 进行计算的操作数 1 和操作数 2。在立即数类型下，操作数 2 为立即数。

3.2.1 计算指令

计算指令主要有以下所列出的这些，他们主要通过 ALU_part.v 来实现。

在译码阶段，给出 ALU_en 和 ALU_ctrl 传入到主模块，这决定了是否使用 ALU 以及进行何种运算。在主模块中，根据译码阶段给出的信息。再从寄存器模块取出对应的操作数。计算结束后再次根据 `wr_en` 决定是否写会寄存器。

```
assign rd_data1 = regs_all[rd_addr1];
assign rd_data2 = regs_all[rd_addr2];
```

Listing 3.12: 寄存器读取数据.

Table 3.3: 计算指令.

指令名称	操作	指令名称	操作
`add`	rd← rs + rt	`addi`	rd← rs + imm
`sub`	rd← rs - rt	`subi`	rd← rs - imm
`and`	rd← rs & rt	`andi`	rd← rs & imm
`or`	rd← rs | rt	`ori`	rd← rs | imm
`xor`	rd← rs ⊕ rt	`xori`	rd← rs ⊕ imm
`sll`	rd← rs « shamt	`sra`	rd← rs » shamt
`slt`	rd← $signed(rs) < $signed(rt)		

3.2.2 访存指令

访存指令有以下两种:

Table 3.4: 访存指令.

指令名称	操作
`sw $n, imm($m)`	将 $n 中的值存储到 $m + imm 的位置
`lw $n, imm($m)`	将 $m + imm 的位置内存中的值加载到 $n 寄存器中

通过以下的内存模块实现:

```
data_ram rd_wr_data(
        .address_a(key_wr_addr), .clock_a(wr_clk), .data_a(wr_ram_data),
        .wren_a(key_wr_en),
        .address_b(mem_addr), .clock_b(clk50), .data_b(mem_save),
        .wren_b(wrmem_en), .q_b(mem_fetch)
); // 添加一个真双口RAM, 一个给键盘一个给CPU
```

Listing 3.13: 数据 RAM 的访问存储.

3.2.3 跳转指令

跳转指令主要通过上文交代的 `program_counter.v` 模块计算得出。

主要实现的跳转指令如下所示：

Table 3.5: 跳转指令指令.

指令名称	操作
`j imm`	`pc` ← `{seq_pc[31:28],imm, 2'b0};`
`jr $n`	`pc` ← `$n;`
`jal imm`	`pc` ← `{seq_pc[31:28],imm, 2'b0};`
`beq rs, rt, offset`	`pc` ← `seq_pc + signed_offset` if rs = rt
`bne rs, rt, offset`	`pc` ← `seq_pc + signed_offset` if rs ≠ rt
`ble rs, rt, offset`	`pc` ← `seq_pc + signed_offset` if rs ≤ rt

实际的 MIPS 指令集中并没有 `ble` 指令，但是在 Mars 汇编器中是允许存在的，它将被编译成以下两条指令：

```
slt $1, $2, $3
beq $1, $0, imm
```

所以我们在汇编中使用了 `ble` 这条指令，它可以使得一些跳转条件更方便表示。

对于 `jr` 指令，我们将从寄存器模块中取出对应的跳转寄存器，然后传入 pc 模块，根据跳转条件判断是否要跳到该寄存器存储的位置。

对于 `jal` 指令，我们不仅要执行与 `j` 指令相同的操作，还需要将返回地址存入 `$ra` 寄存器，这个时候我们要求在主模块将 `wr_addr` 赋值为 31，并要求寄存器写入使能即可。

对于分支跳转指令，我们将会通过 ALU 计算出是否需要跳转，并将结果传入 pc 模块。

3.3 IO

本模块介绍实验中使用的输入输出模块是如何与 CPU 进行交互的。

为了简化实验，我们并没有实现系统的 MMIO，而是选择通过寄存器与输入输出设备进行交互，如何交互将会在软件阶段给予介绍。这里给出寄存器模块中的一些引脚赋值：

```
assign vga_addr = (regs_all[21] << 6) + (regs_all[21] << 2) + (regs_all[21] << 1) +
    ↪ regs_all[22];
assign vga_ascii = regs_all[4][10:0];
assign vga_refresh = (regs_all[23] != 0 && regs_all[4][7:0] != 8'h0d);
assign cursor_en = (regs_all[6] != 0);
assign row = regs_all[16][9:0];
assign ram_wr_addr = regs_all[28]; //gp: 写入字符串的位置
assign piano_en = (regs_all[24] == 2);
```

Listing 3.14: 寄存器与 IO 交互.

从这里导出的 `ram_wr_addr` 表示键盘的写入地址，它将会传入数据 RAM 的 A 端口，用于键盘传入的字符的写入。

从这里导出的 `piano_en` 表示音频模块的使能，它将会直接传入音频模块进行控制。

从这里导出的其它引脚，均用来控制 VGA:

- `vga_addr` 表示写入显存的地址，用来在屏幕上输出字符；
- `vga_ascii` 表示写到屏幕上的字符；
- `vga_refresh` 表示是否需要写入到屏幕上；
- `cursor_en` 表示当前是否需要显示光标，因为在我们的实验中，开机界面是不需要显示光标的；
- `row` 表示屏幕上移了多少行。

在 VGA 中，也利用这些引脚来实现一定的功能：

```
assign new_ascii = (vga_addr != wr_addr) ? ascii : (cursor && cursor_en)
? 8'h5f : 8'h0;
```

这里将利用光标使能和光标时钟来显示光标。

我们还在屏幕上输出了不同的颜色的字符。具体的实现方案是，对于 8 位字符码，我们扩展到 11 位，高三位表示该要显示的颜色序号，具体的颜色我们根据序号存储在 VGA 模块的寄存器中：

```
color_list[7] = 24'h6d6875;
color_list[6] = 24'h2a9d8f;
```

```
color_list[5] = 24'h264653;
color_list[4] = 24'h7ea7;
color_list[3] = 24'hbb4c30;
color_list[2] = 24'hc47335;
color_list[1] = 24'hff;
color_list[0] = 24'h264653;
```

Listing 3.15: 字符颜色列表.

如下所示更改字符颜色:

```
assign vga_color = color_list[new_ascii[10:8]];
wire font_valid = (font_data[dx] == 1) ? 1 : 0;
assign vr = font_valid ? vga_color[23:16] : bg_r;
assign vg = font_valid ? vga_color[15:8] : bg_g;
assign vb = font_valid ? vga_color[7:0] : bg_b;
```

Listing 3.16: 控制字符颜色.

3.4 CPU TESTS

在一开始的运行过程中，我们遇到了各种各样的问题，于是编写了较多的测试文件，对项目的各个模块进行测试。

测试使用汇编在目录 `CPU_overall\instructions` 目录下，列表如下:

Table 3.6: 测试文件.

硬件	软件
hex.asm	welcome.asm;
read_key.asm	audio_new.asm
key_vga.asm	fibbonacci.asm
vga_test.asm	fib_key_vga.asm
audio_test.asm	print_res.asm

3.5 遇到的问题和解决方法

3.5.1 与 IO 之间的交互

起初我们的输入输出设备并不能正常工作。而在最开始我们并不知道是哪个地方除了问题：硬件，IO 还是汇编代码。

解决方法：我们主要采取逐步排查的方法，在涉及对 CPU 进行排查的过程，将 CPU 的时钟改为 `KEY[0]` ，方便我们观察系统的变化。

- 脱离 CPU 对输入输出设备进行单独调试，若没有问题，进入下一步骤；
- 编写更简单的汇编代码，即测试代码，脱离其它软件以及过程的干扰，若不存在问题进入下一步骤；
- 将 CPU 的时钟调整为 `KEY[0]` ，将指令的地址、操作数，计算结果导出到七段数码管上。因为数码管数量有限，所以优先选择关键引脚导出，再利用 `LED` 等，高效利用资源调试。

3.5.2 VGA 显示不够清晰

在最开始，我们的屏幕上不能显示稳定的、没有阴影的字符；在后来的软件测试过程中，经常因为少量汇编代码的变动而使得字符出现阴影。

解决方法：我们在调试过程中采取了这样的策略：

- 简化 `VGA.v` 模块中的计算与操作，将对于坐标的计算加入到像素点坐标变化的过程中，而不是根据像素点进行乘除法操作；
- 在编写汇编代码的过程中，尽量不要在较短的距离内修改同一个寄存器的值。

3.5.3 CPU 无法正确计算简单的程序

最开始时，系统甚至无法完成显示 `HEX` 的功能。

解决方法：将一段小程序执行过程中涉及的指令依次排查，按照取指-译码-执行-访存-写回的过程依次检查各个模块，按照手册进行检查。

最终发现，我们对于一些指令的操作数以及目的数的赋值有误。

再进行一系列的测试后，我们将所有的指令进行分配，每人排查一部分，努力确保每一条指令都能正确执行。

3.6 得到的启示

机器永远是对的。

在进行这个实验的过程中，我们经常会不理解开发板的迷惑操作，较为复杂的时序电路是我们所不能理解的，但最终经过调试，都是我们不小心而犯下的错误。

团队合作项目一定要加强沟通。

在最初的调试过程中，我们发现，许多所谓的 bug 都是源于没有沟通好各自负责的模块之间的协调，各个模块之间的 API 没有对应上。

积极的RTFM，STFW。

在实验过程中，关于指令实现问题，都需要寻找官方的 **MIPS manual** ，按照手册执行，而不是根据字面意思进行猜测。
关于 verilog 和 Quartus 的问题都可以在网络上寻找到答案。

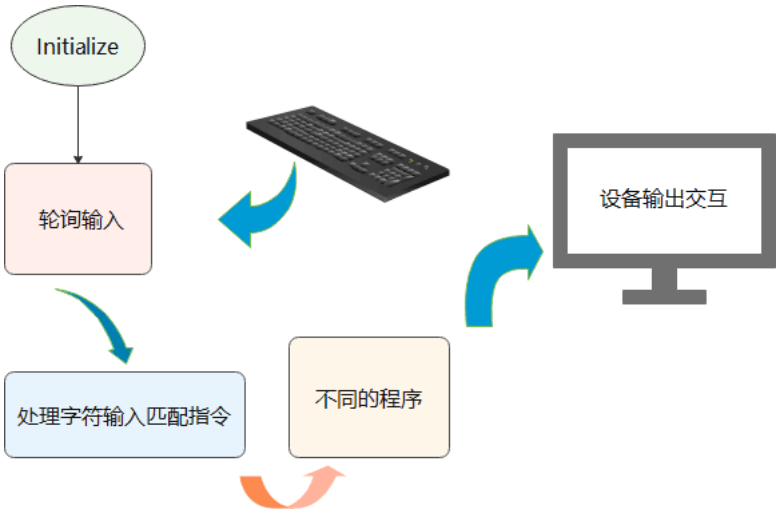
CHAPTER 4

Software

4.1 软件实现流程

软件内部实现的流程主要分为：初始化-硬件输入-程序执行-输出转化-硬件交互。我们对于软件实现流程的流程图主要如下图所示：

Figure 4.1: 软件流程图.



下面就分阶段地对软件的实现进行介绍。

4.1.1 软件内部信息初始化与约定

这一阶段我们主要对一些寄存器进行清零操作，并确定特殊寄存器来存储固定的值，以供后续使用。其中具有特殊功能的寄存器主要如下所示：

其中，`$s3,$s4,$t9` 存储回车、换行、退格的 ASCII 码供键盘缓冲器判断使用；`$t7` 存储 1 作为 `$sp,$gp` 执行过程中的增量（减量）。注意这里由于 `ROM` 生成的特点，`$sp,$gp` 取下一个值的增量为地址的 1 而非数据宽度 4。`$t6` 存储 '0'

的 ASCII 码便于后续字符与操作数之间的转换。 `$s5` 存储的是 VGA 行， `$s6` 存储的是 VGA 列。 `$sp` 是栈指针初始位置， `$gp` 记录当前键盘输入的字符位置。

```
# 一些常数
addi $s3, $zero, 0x0d # 存储回车键
addi $s4, $zero, 0x0a # 存储换行键
addi $t9, $zero, 0x08 # 存储的是退格键
addi $t7, $zero, 1
addi $t6, $zero, 0x30 # '0'的ASCII码
# 特殊目的寄存器
add $s5, $zero, $zero
add $s6, $zero, $zero
addi $sp, $zero, 0x6000 # 一片空旷的，供栈空间使用的内存空间
add $gp, $zero, $zero # 键盘输入的字符
```

Listing 4.1: State Initialization.

除此之外，我们对 `HEX` 和 `LED` 的初始值也通过对应寄存器传出指令进行了初始化，在打开开机界面前我们进行了 VGA 起始地址和须 clear 的屏幕的大小并调用刷新屏幕函数 `_clear_screen` ，刷新后通过提供相应的字符开始位置和字符个数调用 `_print` 函数显示欢迎界面。

4.1.2 轮询硬件输入

这里我们通过轮询键盘输入对特定输入进行检索并对应的跳转到其他函数处理。 首先是初始的 `Press any key to continue` ，这里我们通过一个 `_wait_loop` 函数等待一个键盘输入，轮询到输入则进入到主页面并将该输入清除。

```
_wait_loop:
lw $a0, ($gp)
beq $a0, $zero, _wait_loop
sw, $zero, ($gp) # 进入主页面，将键盘RAM清零
```

Listing 4.2: Wait for a key to continue.

进入主页面后通过 `show_prompt` 函数显示命令提示符，进入 `_read_key` 函数中等待命令输入。这里将键盘输入存入缓冲区（ `RAM` 中的一块位置），处理时从缓冲区取出并对一些特殊的输入如 shift,backspace 进行特殊处理，检测到 backspace 则跳转至 `_Backspace` 函数进行相应的清除工作（包括调整 `_write` 写入 VGA 的字符和 `$gp` 的内容与位置）后返回。对于其他字符，我们通过 `_write` 函数写入 VGA 的

RAM 中，检测到回车键时跳转至 _handle_cmd 函数进行对应字符与命令的匹配和处理。 对键盘的轮询处理即 _read_key 函数如下所示。

```
_read_key:
lw $a0, ($gp)
beq $a0, $zero, _read_key
addi $t0, $zero, 0x10 # shift键
beq $a0, $t0, _read_key
beq $a0, $t9, _Backspace
add $gp, $gp, $t7
_handle_key:
jal _write
j _read_key
# 是Backspace开始处理
_Backspace:
beq $gp, $zero, _read_key # 在最开始，不能退回上一行了
add $a0, $zero, $zero # 退格相当于先退一格，然后在那个位置写上空白
sub $s6, $s6, $t7
add $s7, $zero, $t7
add $s7, $zero, $zero
sw $zero, ($gp)
sub $gp, $gp, $t7
sw $zero, ($gp)
j _read_key
```

Listing 4.3: Read and handle the key.

其中 _write 函数会将对应字符写入 VGA 的 RAM 中，该函数将在 4.1.6 节具体介绍。

4.1.3 命令匹配过程

我们将对应的命令字符事先存放在 RAM 中并对对应地址的字符与缓冲区中的键盘输入进行一一比对后跳转执行相应的子程序，未比对成功则默认跳转至相应函数调用 _print 输出 Unknown Command 。

对应的我们编写了 _strcmp 函数进行命令的比对，其中以回车键作为比较结束的标志。

4.1.4 子程序的执行

对于子程序的执行，我们大致分为两种，一种是无需进行操作数输入的：`hello, help, fail`，其中`fail`是出现`Unknown Command`时调用的子程序；另一种是需要等待操作数的输入的：`fib, LEDon, LEDoff, display, piano`。`restart`作为特殊的子程序直接清空键盘缓冲器并回到初始化状态。

第一种，即直接进行信息输出，以 hello 为例，我们通过设置对应字符在`RAM`中的起始位置和字符数量直接调用`_print`进行输出，由于无需等待操作数，输出后直接调用`main`等待下一个命令。

```
_hello:
addi $a0, $zero, 0x1500
addi $a1, $zero, 196
jal _print
jal _cleargp # 打印的字符串里面有回车，不需要j _newline
j main # 等待下一个命令
```

Listing 4.4: hello.

第二种则分为两个阶段，第一个阶段进行命令的读入与匹配，保存命令号后继续调用`_read_key`读取操作数，根据命令号跳转至相应的函数处理操作数，其中`_atoi`函数将把操作数输入时输入的字符转换为操作数，该函数将在 4.1.5 节具体介绍。

以`fib`为例，首先在`_handle_cmd`中通过`_strcmp`匹配对应字符。

```
# fib
addi $a1, $zero, 0x1750 #fib命令在RAM中的起始地址
jal _strcmp
bne $v0, $zero, _fib_cmd # 设置s2=1
```

Listing 4.5: fib-judge.

接着跳转至`_fib_cmd`函数进行命令号的设置（`$s2`固定存储命令号，为 0 时表示无当前进行的命令），后通过`_cleargp`清除缓冲区的字符命令，跳转至`_read_key`读入操作数（直接跳转至`_read_key`不调用`show_prompt`展示命令提示符）。

```
_fib_cmd:
addi $s2, $zero, 1 # 置1等待参数
jal _cleargp
j _read_key
```

Listing 4.6: fib_cmd.

上层读到回车后再次回到 `_handle_cmd` 判断是否有命令号，有则跳转到 `_op_handle` 如下（其中如果判断没有输入操作数则继续调用 `_read_key` 读入），这里通过 `_atoi` 进行输入字符向数字的转化，转化的数字参数存储在固定寄存器 `$t5` 中：

```
_op_handle:
lw $t0, ($zero)
beq $t0, $s3, _void_op
jal _atoi
add $t5, $zero, $v0 # t5中存放参数
addi $t0, $zero, 0x1
beq $s2, $t0, _fib
addi $t0, $zero, 0x2
beq $s2, $t0, _piano
addi $t0, $zero, 0x3
beq $s2, $t0, _LEDon
addi $t0, $zero, 0x4
beq $s2, $t0, _LEDoff
addi $t0, $zero, 0x5
beq $s2, $t0, _hex
_void_op:
add $gp, $zero, $zero
sw $zero, ($gp)
j _read_key
```

Listing 4.7: _op_handle.

通过判断命令号跳转到相应的子程序处理操作数（这里是 `_fib` 进行 Fibonacci 数的计算），计算后通过 `_itoa` 进行数字向字符的转化后调用 `_print` 输出对应的结果。

```
_fib:
add $t0, $zero, $t5  # 数字参数在t5里
add $t1, $zero, $zero
addi $t2, $zero, 0x1
xor $t3, $t3, $t3
_fibo_loop:
beq $t0, $zero, _fido_ret
sub $t0, $t0, $t7
add $t3, $t2, $t1
add $t1, $zero, $t2
add $t2, $zero, $t3
j _fibo_loop
```

```
_fido_ret:
add $v0, $zero, $t1
add $t5, $zero, $v0
add $a1, $zero, $zero
jal _itoa
addi $a0, $zero, 0x2000 #从2000开始打印
jal _print
jal _newline
j main
```

Listing 4.8: Calculation of Fibonacci number.

执行完成后回到 `main` 中清空命令号和操作数寄存器 `$s2, $t5` 并调用 `show_prompt` 输出命令提示符，继续轮询键盘输入判断字符。

4.1.5 输入输出间形式的转换

如上节所述，处理操作数的子程序需要将对应的字符转换为对应的数字，处理后需要将数字转换为字符写入 VGA 的 `RAM` 中，所以我们分别提供了 `_atoi` 和 `_itoa` 的汇编代码进行输入输出形式之间的转换。

说明：这里为了处理的方便，我们规定输入输出均为十六进制数。

4.1.6 与硬件之间的交互

上述过程主要描述了软件内部工作原理，与硬件输入之间的交互则通过轮询键盘输入地址寄存器 `$gp` 中的值取得键盘输入缓冲器中的值并进行对应的处理。

而与硬件输出的交互主要包括：VGA 字符显示，LED 的输出，HEX 的显示和 piano 的播放。

字符的显示由 `_print`（输出既定起始位置开始既定数量的字符）和 `_write`（输出一个字符）完成。换行操作由 `_newline` 完成，屏幕的滚动通过 `_moveon` 完成。

LED 的输出，HEX 的显示，piano 的播放均通过在硬件中判断命令号寄存器 `$s2` 的值和操作数寄存器 `$t5` 的值进行对应的处理以输出显示。

除此之外，在处理键盘缓冲区字符时，也需要与硬件的合理交互。

具体地，需要每次处理命令后将键盘缓冲区地址 `$gp` 清零并将缓冲区字符通过 `_cleargp` 函数清空，为下次命令字符的读入做准备。

4.2 遇到的问题和解决方法

4.2.1 寄存器约定与协调

4.2.2 程序之间的协同

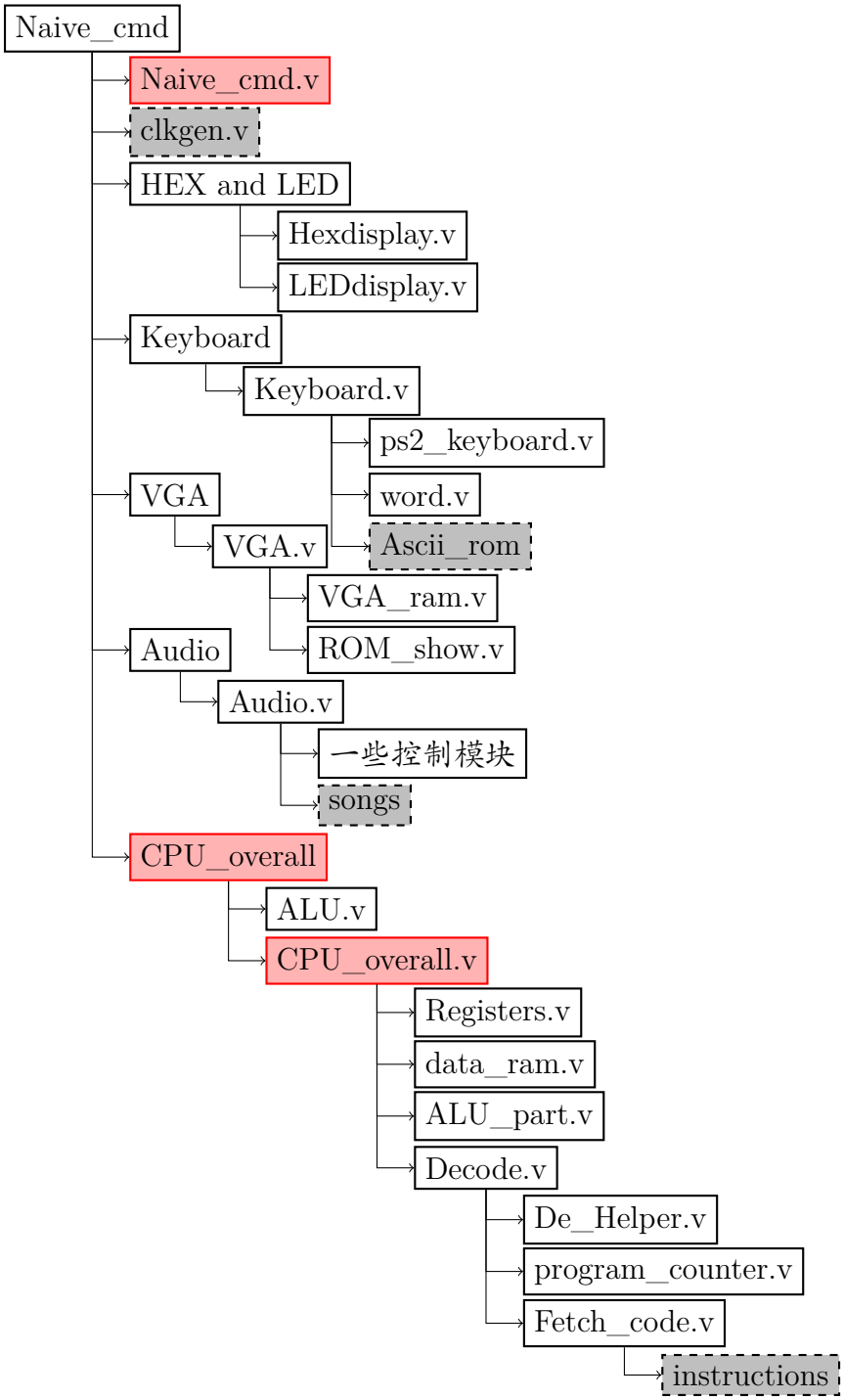
4.3 得到的启示

APPENDIX A

File Structure of the Project

本附录用于展示实验目录的文件结构。下面的结构图展示了项目文件中的关键文件，以及他们之间的所属关系。项目名称为 **Naive_cmd**，意味较为简陋、不够成熟的命令行。

文件的上下级关系表示某个文件属于某一个目录，或者上级模块调用了下级模块，即调用与被调用的关系。


Naive_cmd
Naive_cmd.v
clkgen.v
HEX and LED
Hexdisplay.v
LEDdisplay.v
Keyboard
Keyboard.v
ps2_keyboard.v
word.v
VGA
VGA.v
Ascii_rom
VGA_ram.v
ROM_show.v
Audio
Audio.v
一些控制模块
songs
CPU_overall
ALU.v
CPU_overall.v
Registers.v
data_ram.v
ALU_part.v
Decode.v
De_Helper.v
program_counter.v
Fetch_code.v
instructions

APPENDIX B

Scripts of Special Purpose

因为每次都要打开 Mars 先编译再导出机器码较为繁琐，所以我们提供了一个从 Mars 中获取机器码的脚本，如下所示：

```
import mars.*;
import java.util.*;

public class MarsCompiler {
    public static void main(String... args) throws Exception {
        if (args.length != 1) {
            System.err.println("Usage: java MarsCompiler input");
            System.exit(1);
            }

        Globals.initialize(false);
        MIPSprogram program = new MIPSprogram();
        program.readSource(args[0]);

        ErrorList errors = null;

        try {
            program.tokenize();
            errors = program.assemble(new ArrayList(Arrays.asList(program)), true,
                ↪ true);
            }
        catch (ProcessingException e) {
            errors = e.errors();
```

```
            }

        if (errors.errorsOccurred() || errors.warningsOccurred()) {
            for (ErrorMessage em : (ArrayList<ErrorMessage>)errors.getErrorMessages()
                ↪ ) {
                System.err.println(String.format("[%s] %s@%d:%d %s",
                em.isWarning() ? "WRN" : "ERR",
                em.getFilename(), em.getLine(), em.getPosition(),
                em.getMessage()));
                }
            System.exit(2);
        }

        for (ProgramStatement ps : (ArrayList<ProgramStatement>)program.
            ↪ getMachineList())
        System.out.println(String.format("%08x", ps.getBinaryStatement()));
        }
}
```

Listing B.1: 提取代码脚本.

这里运用了 Mars 作为 `java` 语言的特点，利用了本身固有的类来导出机器码。只需要在命令行执行以下的指令：

```
javac --class-path MARS4_5.jar -Xlint:unchecked MarsCompiler.java
java -cp MARS4_5.jar MarsCompiler overall.asm > overall.txt
```

.

即可将机器码导出到指定的文件中。以上为 Macbook + iTerm2 + zsh 下的指令，Windows 下可能有区别。