

基于FPGA的文件系统shell

PB20061229 吕泽龙

PB20000156 徐亦昶

主要功能及实现思路

继承lab05的流水线cpu及指令扩展。

用pdu与cpu进行IO交互，并把板载数码管显示换成vga屏幕显示。支持vga上的模拟键盘
cpu提供新的总线，提供子程序执行。

编写汇编代码，完成//to be done

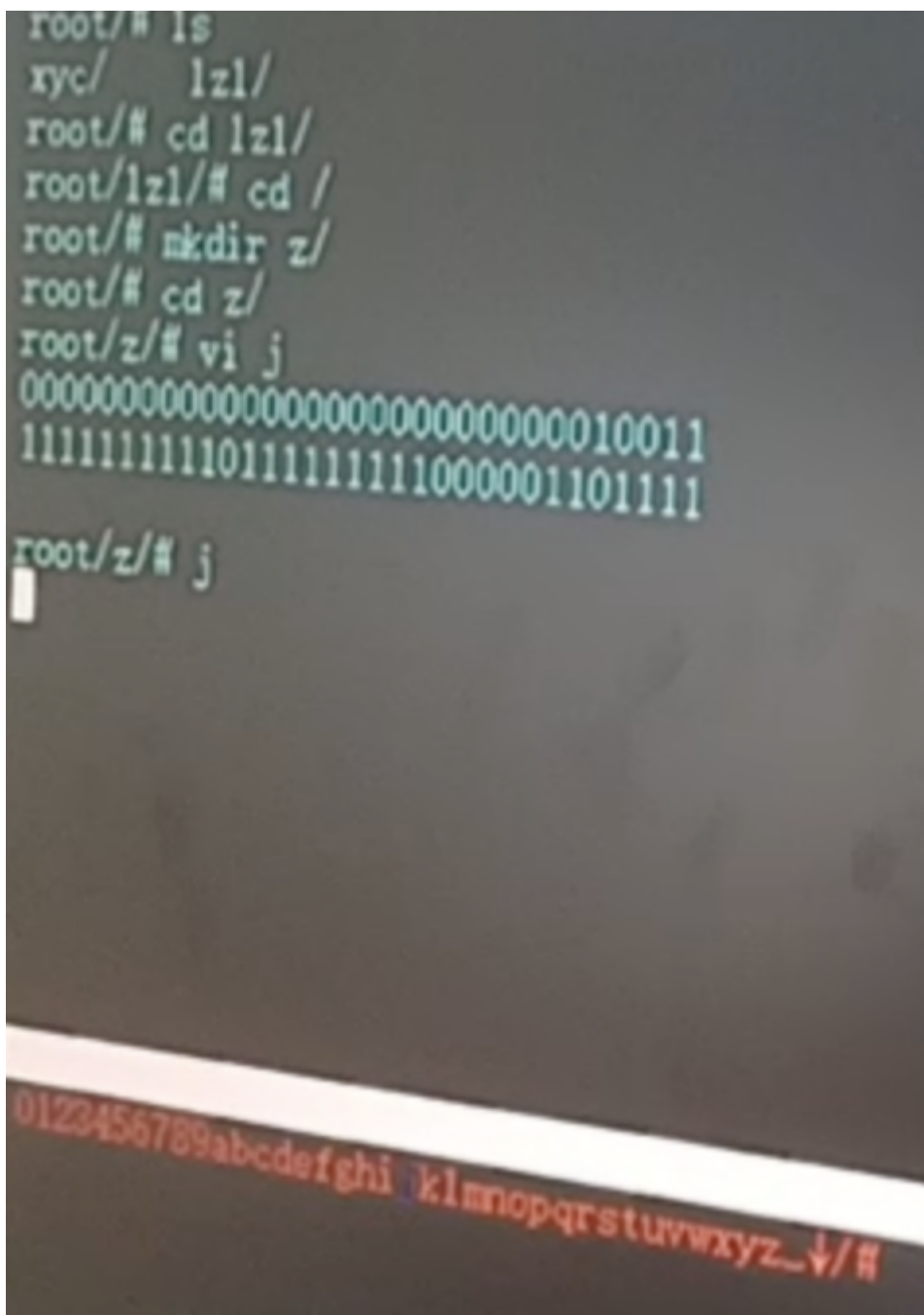
最终实现vga屏幕显示的shell文件系统

项目框架及分工

1. verilog(吕泽龙负责)
 - a. cpu
 - b. vga
 - c. pdu
2. assemble
 - a. 目录树实现
 - b. I/O实现
 - c. 各种指令

实现效果

如下图，在终端中写了一段死循环的代码并运行成功。



verilog 部分

流水线cpu及指令扩展

lab05, 吕泽龙已经检查过, 略

vga控制模块

实现思路

480*640的分辨率。

将整个屏幕分成30 * 80的大块，每个大块含有16 *8个像素点。

前20行是命令行显示的内容，vga内部用分布式RAM内存存储，指存储大块对应的信息编号1-40，这些信息编号映射成0-9，a-z，空格换行/,#，而后再开一个分布式ROM内存储存这些编号对应信息的像素点阵。

扫描时前20行先确定大块位置提取RAM的信息编号，再用信息编号和大块相对位置提取像素点。

21行显示一行白线区分键盘和命令行。

22行显示键盘。

换行时开启刷屏状态机。

以上时大致的实现思路。

输入输出端口

输入端口：clk,I_rst_n（输入时钟和rst）

current_vpos和current_hpos（外部光标所在的位置

ChoseCh（当前键盘输入显示位置

refresh_signal(置1时，下个时钟周期开启刷屏模式，会实现屏幕换行，并且刷屏时busy忙线上拉高电平)

wa, wd, we, 屏幕写入编号。

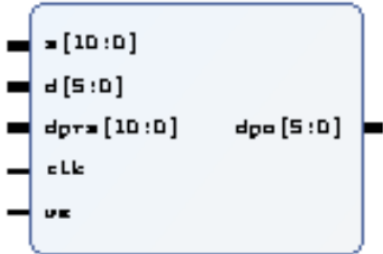
输出端口：busy忙线，表示正在刷屏，O_rgb,O_hs/vs, vga控制输出。

```
module vga_driver #(
parameter VGA_LINE = 20
)(
    input clk, //100MHZ
    input I_rst_n , // ???
    input [4:0]current_vpos,
    input [6:0]current_hpos,
    input [5:0]ChoseCh,
    input refresh_signal,
    output reg [3:0] O_red , // VGA????
    output reg [3:0] O_green , // VGA????
    output reg [3:0] O_blue , // VGA????
    output O_hs , // VGA?????
    output O_vs , // VGA?????
    input [10:0] wa_r,
    input [5:0] wd_r,
    input we_r,
    output busy
);
```

存储模块

vga_cmd:双端口RAM，深度：20*80<=2048，位宽40<=64

☐ Show disabled ports



The diagram shows a rectangular block representing the VGA Command Memory. On the left side, there are four input ports: a 10-bit address bus labeled 'a[10:0]', a 5-bit data bus labeled 'd[5:0]', a 10-bit dual-port address bus labeled 'dpra[10:0]', and a clock input labeled 'clk'. On the right side, there is one output port: a 5-bit dual-port data bus labeled 'dpo[5:0]'.

Component Name

memory config

Port config

RST & Initia

Options

Depth [16 - 65536]

Data Width [1 - 1024]

Memory Type

Memory Type

☐ ROM

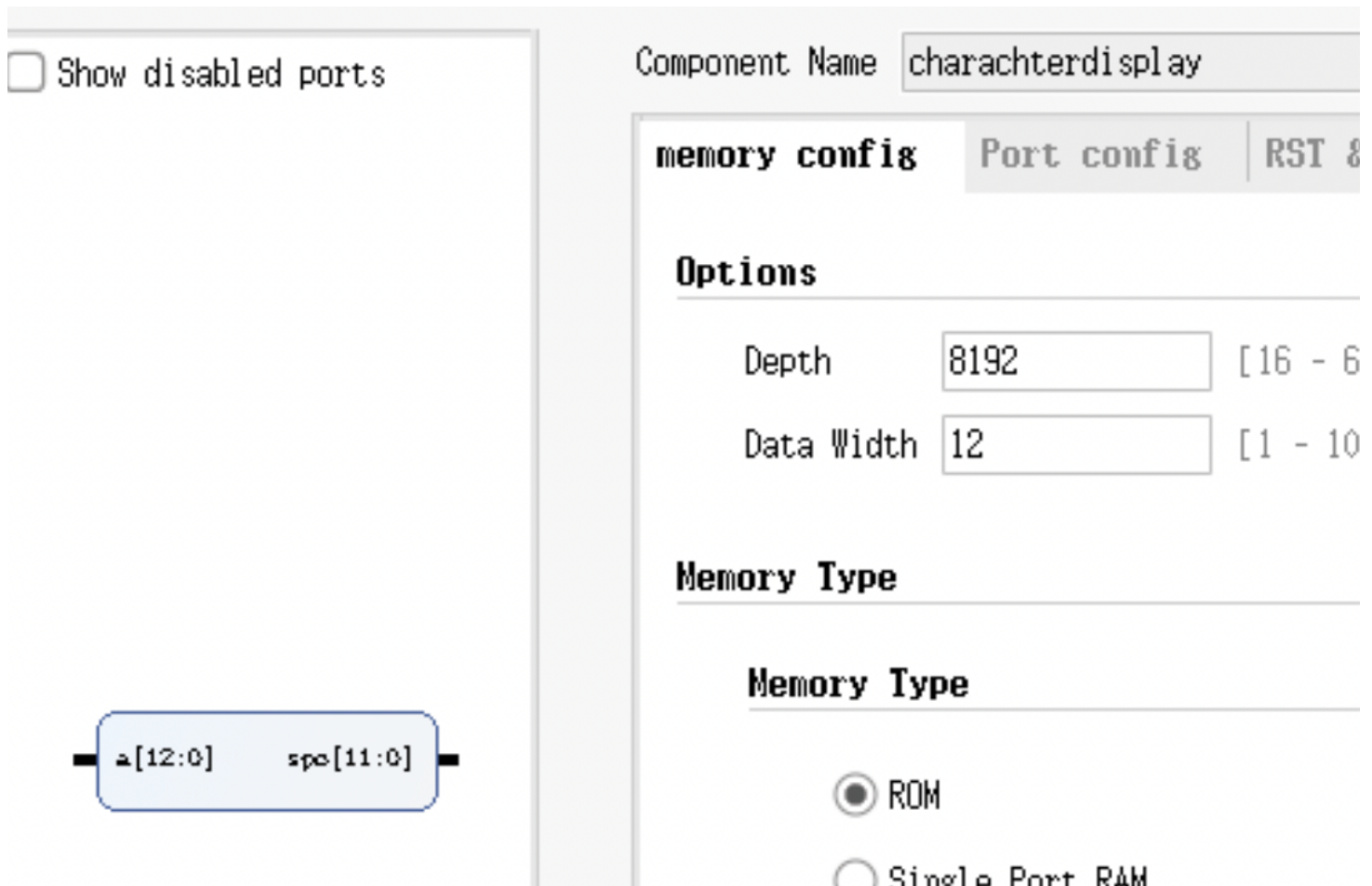
☐ Single Port RAM

☒ Simple Dual Port RAM

☐ Dual Port RAM

Select the Memory Type

characterdisplay,ROM,深度 $40 \times 16 \times 8 \leq 8192$,位宽rgb各4位为12位



```
vga_cmd r1(wa, wd, ra0, clk, we, rd0); //register file for vga
charachterdisplay r2(ra1, rd1);
```

获取各种位置信息

R_h_cnt和R_v_cnt是当前vga的垂直和平行计数器，减去开始的PLUSE就是对应的像素点位置，再根据像素点位置可以获得大块小块信息。按照存储位置模拟即可。

显示部分的代码就略去了，只是按照顺序显示对应的像素。

```
assign H_pos = (R_h_cnt - (C_H_SYNC_PULSE + C_H_BACK_PORCH)) >> 3;
assign V_pos = (R_v_cnt - (C_V_SYNC_PULSE + C_V_BACK_PORCH)) >> 4;
assign isROM = V_pos == VGA_LINE + 1 && H_pos <= 40;
assign hpos = (R_h_cnt - (C_H_SYNC_PULSE + C_H_BACK_PORCH)) & 7;
assign vpos = (R_v_cnt - (C_V_SYNC_PULSE + C_V_BACK_PORCH)) & 15;
assign ra0_r = V_pos * 80 + H_pos;
assign ra1 = (((isROM ? H_pos : rd0) << 7) + (vpos << 3) + hpos);
assign display_data = !isROM && V_pos >= VGA_LINE ? 0 : rd1;
```

刷屏实现

设计一个状态及，平时为0状态待机，开启signal之后1状态读取，2状态写入，3状态移动指针。读取和写入就是把下一行内容读取而后写入到上一行中。由此实现屏幕滚动

```
assign busy = refresh_state != 0;
initial refresh_state = 0;

always@(posedge clk) begin
    if(refresh_signal) begin
        refresh_vpos <= 0;
        refresh_hpos <= 0;
        refresh_state <= 1;
    end
    else if(refresh_state == 1) begin
        ra0_b <= (refresh_vpos + 1) * 80 + refresh_hpos;
        we_b <= 0;
        refresh_state <= 2;
    end
    else if(refresh_state == 2) begin
        wa_b <= refresh_vpos * 80 + refresh_hpos;
        we_b <= 1;
        wd_b <= refresh_vpos + 1 == VGA_LINE ? 0 : rd0;
        refresh_state <= 3;
    end
    else if(refresh_state == 3) begin
        if(refresh_hpos >= 79) begin
            refresh_hpos <= 0;
            if(refresh_vpos + 1 >= VGA_LINE)
                refresh_state <= 0;
            else begin
                refresh_vpos <= refresh_vpos + 1;
                refresh_state <= 1;
            end
        end
        else begin
            refresh_hpos <= refresh_hpos + 1;
            refresh_state <= 1;
        end
    end
end
end
```

pdu控制vga

当cpu的io_we上拉的时候，会把io_data输入，并且下拉input_rdy准备位（这部分没有改动），当input_rdy低电平的时候，把io_data传入vga，并且判断是不是换行来处理光标的位置和刷屏信息。处理结束后立刻上拉input_rdy即可。

同时，通过button处理ChoseCh来选择键盘，这部分的状态机比较简单，就不放在报告里了。

```

else if (!input_rdy_r) begin
    input_rdy_r <= 1;
    if(input_data_raw == 38) begin//换行
        vga_hpos <= 0;
        if(vga_vpos + 1 >= VGA_LINE)
            refresh_signal <= 1;
        else vga_vpos <= vga_vpos + 1;
        vga_we <= 0;
    end
    else begin
        vga_wd <= input_data_raw;
        vga_we <= 1;
    end
end
end
else if(vga_we == 1) begin
    vga_we = 0;
    vga_hpos = vga_hpos + 1;
end
end

```

cpu支持子程序执行

新设置了三条线10040000, 10040004, 10040008, 传输方式和IO交互差不多。

10040004传输给Sub_pcbias,10040008传输给Sub_pcret, 表示子程序执行起始和终止位置相对内存基址的偏移字节数。

当Sw指令操作10040000时, 产生一个和EX_Branch类似的跳转信号Sub_run, 启动子程序的执行。

记录下启动的Ex_pc地址。

执行过程中IF阶段从内存取指令。

pc默认从40000开始, 但是从相对偏移位置取指令。

结尾要留三个气泡, 防止子程序结尾的beq被跳回正常程序打断。

下面是修改过后的取指令模块和pcWrite寄存器修改流程, Ex和Mem部分也有小部分的修改。不放在报告里了。

```

always@(posedge clk) begin
    if(Sub_ready) begin
        Sub_run <= 1;
        Sub_pcrec <= EX_pc + 4;
        pc <= 32'h40000;
    end
    else if(pcWrite) begin
        if(Sub_run && MEM_dproa_r >= Sub_pcret + 16) begin
            //led <= pc;
            pc <= Sub_pcrec;
            Sub_run <= 0;
        end
    else
        pc <= pcin;
    end
end
module IF_Block(input [31:0]pc, output [31:0]instruction, input Sub_run, input [31:0]Mem
    wire [31:0]instruction_address_t, instruction_t;
    assign instruction_address_t = (pc - 32'h40000);
    dist_mem_gen_0 InstructionMemory(instruction_address_t[10:2], instruction_t);
    assign instruction = Sub_run ? Mem_instruction : instruction_t;
endmodule

```

汇编部分

数据段结构

- 0~4ff: 子程序
- 500~6ff: MMIO地址、当前路径指针和目录名相关字符串常量
- 700往后: 目录树、预留0x100 byte空位、命令字符串、预留100个空位作为当前完整工作目录名字字符串、其他指针。

代码段实现如下:


```
.data
.word 0x00100413
.word 0x008404b3
.word 0x00100413
.word 0xfe941ae3
.space 0x4f0
kbd_st:.word 0xffff0000
kbd_data:.word 0xffff0004
display_st:.word 0xffff0008
display_data:.word 0xffff000c
current_dir:.word 0x10010700
strings:
.word 0x00000023 #prompt
.word 0x00000020
.word 0x00000000
.word 0x00000072 #root/
.word 0x0000006f
.word 0x0000006f
.word 0x00000074
.word 0x0000002f
.word 0x00000000
.word 0x00000078 #xyc/
.word 0x00000079
.word 0x00000063
.word 0x0000002f
.word 0x00000000
.word 0x0000006c #lzl/
.word 0x0000007a
.word 0x0000006c
.word 0x0000002f
.word 0x00000000
.word 0x00000074 #test
.word 0x00000065
.word 0x00000073
.word 0x00000074
.word 0x00000000
.space 0x18c
root: #0x10010700
.word 0x10010520
.word 0x00000001 #is a directory
.word 0x10010754 #xyc
.word 0x10010784 #lzl
.word 0x00000000 #end list
.space 0x40
.word 0x10010538 #xyc
.word 0x00000001 #is a directory
.word 0x100107b0 #test(under the directory)
.word 0x00000000
.space 0x20
.word 0x1001054c #lzl
```

```

.word 0x00000001 #is a directory
.word 0x00000000
.space 0x20
.word 0x10010560 #test
.word 0x00000000 #is a program
.word 0x10010000 #Start address
.word 0x1001000c #End address
.word 0x00000000
.space 0x100
const_str:
.word 0x0000006c #ls
.word 0x00000073
.word 0x00000000
.word 0x00000063 #cd
.word 0x00000064
.word 0x00000000
.word 0x00000076 #vi
.word 0x00000069
.word 0x00000000
.word 0x0000006d #mkdir
.word 0x0000006b
.word 0x00000064
.word 0x00000069
.word 0x00000072
.word 0x00000000
.space 100 #current_full_dir
buffer:.word 0x10010f00
prompt_str:.word 0x10010514
current_full_dir:.word 0x10010900
ls_str:.word 0x100108c4
cd_str:.word 0x100108d0
vi_str:.word 0x100108dc
mkdir_str:.word 0x100108e8
current_dir_ptr:.word 0x10010510
base:.word 0x10010000 #Base address of data segment
subproc_addr_ptr:.word 0x10040000 #Related to the mechanism of CPU
next_index_addr:.word 0x100107c4 #next address of the file to be created
next_program_addr:.word 0x10010010
next_filename_addr:.word 0x10010574

```

目录树实现

使用树形结构来保存目录。对于目录，存储结构如下：

目录名字符串首地址、1（表示目录）、子节点地址1（指向子目录）、子节点地址2、……、子节点地址n、0（表示结束）。

对于可执行文件，存储结构如下：

文件名字符串首地址、0（表示文件）、代码第一行在数据段的位置、代码最后一行在数据段的位置、

0。

初始目录树为：root/->xyz/->test,root/->lzl/, 这里使用->表示父子关系。

I/O实现

使用MMIO来进行输入/输出。先写getchar和putchar作单字符的输入输出，再写scanf和printf作为字符串的输入输出。

```

#getchar()
getchar:
getchar_LOOP:
lw t0, keybd_st
lw t0, (t0)
beq t0, x0, getchar_LOOP
lw a0, keybd_data
lw a0, (a0)
jalr x0, 0(ra)
#scanf
scanf:
addi sp, sp, -16
sw a0, 12(sp)
sw ra, 8(sp)
sw s0, 4(sp) #s0 will store the base address
sw s1, (sp) #temporary
add s0, a0, x0
SCANF_LOOP:
jal getchar
jal putchar
addi s1, x0, 0xa
beq a0, s1, EXIT_SCANF
sw a0, (s0)
addi s0, s0, 4
beq x0, x0, SCANF_LOOP
EXIT_SCANF:
sw x0, (s0)
lw s1, (sp)
lw s0, 4(sp)
lw ra, 8(sp)
lw a0, 12(sp)
addi sp, sp, 16
jalr x0, 0(ra)

#putchar
putchar:
PUTCHAR_LOOP:
lw t0, display_st
lw t0, (t0)
beq t0, x0, PUTCHAR_LOOP
lw t0, display_data
sw a0, (t0)
jalr x0, 0(ra)
#printf
printf:
addi sp, sp, -12
sw a0, 8(sp)
sw ra, 4(sp)
sw s0, (sp) #s0 will store the base address
addi s0, a0, 0

```

```
PRINTF_LOOP:
lw a0,(s0)
beq a0,x0,EXIT_PRINTF
jal putchar
addi s0,s0,4
beq x0,x0,PRINTF_LOOP
EXIT_PRINTF:
lw s0,(sp)
lw ra,4(sp)
lw a0,8(sp)
addi sp,sp,12
jalr x0,0(ra)
```

字符串函数

实现了strcmp、strcpy、strcat、strlen以便于后期做字符串处理。

#strcmp, 0 stands for equal, -1 not equal

strcmp:

addi sp,sp,-8

sw s0,4(sp)

sw s1,(sp)

addi s0,a0,0

addi s1,a1,0

STRCMP_LOOP:

lw t0,(s0)

lw t1(s1)

add t2,t0,t1 #t2=0 means equal

beq t2,x0,STRCMP_RETURN_EQUAL

beq t0,x0,STRCMP_RETURN_NEQUAL

beq t1,x0,STRCMP_RETURN_NEQUAL

bne t0,t1,STRCMP_RETURN_NEQUAL

addi s0,s0,4

addi s1,s1,4

beq x0,x0,STRCMP_LOOP

STRCMP_RETURN_EQUAL:

addi a0,x0,0

beq x0,x0,EXIT_STRCMP

STRCMP_RETURN_NEQUAL:

addi a0,x0,-1

EXIT_STRCMP:

lw s1,(sp)

lw s0,4(sp)

addi sp,sp,8

jalr x0,0(ra)

#strcat

strcat:

addi sp,sp,-4

sw a0,(sp)

addi t0,a0,0

addi t1,a1,0

FIND_END:

lw t2,(t0)

addi t0,t0,4

bne t2,x0,FIND_END

addi t0,t0,-4

STRCAT_LOOP:

lw t2,(t1)

sw t2,(t0)

addi t0,t0,4

addi t1,t1,4

bne t2,x0,STRCAT_LOOP

lw a0,(sp)

addi sp,sp,4

jalr x0,0(ra)

```

#strcpy
strcpy:
addi sp,sp,-4
sw a0,(sp)
addi t0,a0,0
addi t1,a1,0
STRCPY_LOOP:
lw t2,(t1)
sw t2,(t0)
addi t0,t0,4
addi t1,t1,4
bne t2,x0,STRCPY_LOOP
lw a0,(sp)
addi sp,sp,4
jalr x0,0(ra)

```

```

#strlen
strlen:
addi t0,x0,0
STRLEN_LOOP:
lw t1,(a0)
beq t1,x0,EXIT_STRLEN
addi a0,a0,4
addi t0,t0,1
beq x0,x0,STRLEN_LOOP
EXIT_STRLEN:
addi a0,t0,0
jalr x0,0(ra)

```

命令分割

本shell运行命令中出现一个空格。调用split可以进行分割。返回的a0是第一个字符串，a1是第二个字符串。原字符串的空格位置会被改为0。注意如果不包含空格，则a1返回0。

```
#split
split:
addi t0,a0,0
FIND_SPACE:
lw t1,(t0)
beq t1,x0,NOT_FOUND
addi t1,t1,-32
addi t0,t0,4
bne t1,x0,FIND_SPACE
addi t0,t0,-4
sw x0,(t0)
addi a1,t0,4
beq x0,x0,SPLIT_EXIT
NOT_FOUND:
addi a1,x0,0
beq x0,x0,SPLIT_EXIT
SPLIT_EXIT:
jalr x0,0(ra)
```

ls指令

遍历当前目录树下的直接子节点，依次打印出它们的文件名。


```

#ls
ls:
addi sp,sp,-12
sw a0,8(sp)
sw s0,4(sp)
sw ra,(sp)
addi a0,a0,8
addi s0,a0,0
LS_LOOP:
lw a0,(s0)
beq a0,x0,EXIT_LS
lw a0,(a0)
jal printf
addi s0,s0,4
addi a0,x0,32 #Print space
jal putchar
jal putchar
jal putchar
jal putchar
beq x0,x0,LS_LOOP
EXIT_LS:
addi a0,x0,0xa #Print \n
jal putchar
lw ra,(sp)
lw s0,4(sp)
lw a0,8(sp)
addi sp,sp,12
jalr x0,0(ra)

```

cd指令

可以切换工作目录。注意目录名必须以/结尾，否则无法进行匹配。

```

#cd
cd:
addi sp,sp,-16
sw a1,12(sp)
sw a0,8(sp)
sw s0,4(sp)
sw ra,(sp)
lw t0,(a0)
lw t1,4(a0)
addi t0,t0,-47
beq t0,x0,RET_CHECK
beq x0,x0,NORMAL_CD
RET_CHECK:
beq t1,x0,RET
beq x0,x0,EXIT_CD
RET:
jal cd_back
beq x0,x0,EXIT_CD
NORMAL_CD:
lw s0,current_dir
addi s0,s0,8
FIND_POS:
lw a0,8(sp)
lw a1,(s0)
beq a1,x0,EXIT_CD
lw a1,(a1)
jal strcmp
addi s0,s0,4
bne a0,x0,FIND_POS
addi s0,s0,-4 #a0 is the address of the dir
lw t0,current_dir_ptr #change the current directory
lw s0,(s0)
sw s0,(t0)
lw a0,current_full_dir
addi a1,s0,0
lw a1,(a1)
jal strcat
EXIT_CD:
lw ra,(sp)
lw s0,4(sp)
lw a0,8(sp)
lw a1,12(sp)
addi sp,sp,16
jalr x0,0(ra)

```

运行子程序

根据目录树读出子程序的起始和结束地址后，将它们分别写入0x0x10040004和0x10040008，再向0x10040000中用sw随便写入一个数，CPU便会控制子程序执行。

```

#run
run:
addi sp,sp,-16
sw a1,12(sp)
sw a0,8(sp)
sw s0,4(sp)
sw ra,(sp)
lw s0,current_dir
addi s0,s0,8
FIND_SUBPROC:
lw a0,8(sp)
lw a1,(s0)
beq a1,x0,EXIT_RUN
lw a1,(a1)
jal strcmp
addi s0,s0,4
bne a0,x0,FIND_SUBPROC
addi s0,s0,-4
lw s0,(s0)
lw t0,base
lw t1,8(s0) #begin address
lw t2,12(s0) #end address
sub t1,t1,t0
sub t2,t2,t0
lw s0,subproc_addr_ptr
sw t1,4(s0)
addi x0,x0,0 #To make sure t1 is actually stored in pipeline
addi x0,x0,0
sw t2,8(s0)
addi x0,x0,0
addi x0,x0,0
sw x0,(s0) #enable subprocess
EXIT_RUN:
lw ra,(sp)
lw s0,4(sp)
lw a0,8(sp)
lw a1,12(sp)
addi sp,sp,16
jalr x0,0(ra)

```

vi指令

模仿了Linux下的vi，可以用01二进制码编写可执行文件。指令格式为vi <文件名>（实际上没有尖括号）。

```

#vi
vi:
addi sp,sp,-16
sw a1,12(sp)
sw a0,8(sp)
sw s0,4(sp)
sw ra,(sp)
lw s0,current_dir
addi s0,s0,8
FIND_INDEX:
lw t0,(s0)
addi s0,s0,4
bne t0,x0,FIND_INDEX
addi s0,s0,-4
lw t1,next_index_addr
sw t1,(s0)
sw x0,4(s0) #Expand the directory list
lw t0,next_filename_addr
lw t1,next_index_addr
sw t0,(t1)
addi a1,a0,0
addi a0,t0,0
addi sp,sp,-4
sw t1,(sp)
jal strcpy
jal strlen
addi a0,a0,1
slli a0,a0,2
la t0,next_filename_addr
lw t1,(t0)
add t1,t1,a0
sw t1,(t0)
lw s0,next_program_addr
lw t1,(sp)
addi sp,sp,4
sw x0,4(t1) #Indicate a file
sw s0,8(t1) #Begin address
VI_LOOP:
lw a0,buffer
jal scanf
lw t0,(a0)
beq t0,x0,EXIT_VI_LOOP
jal str_to_bin
sw a0,(s0)
addi s0,s0,4
beq x0,x0,VI_LOOP
EXIT_VI_LOOP:
addi s0,s0,-4 #Now s0 is the end address
lw t1,next_index_addr
sw s0,12(t1)

```

```

la t0,next_index_addr
lw t1,(t0)
addi t1,t1,4
sw t1,(t0)
la t0,next_program_addr
addi s0,s0,4
sw s0,(t0)
lw ra,(sp)
lw s0,4(sp)
lw a0,8(sp)
lw a1,12(sp)
addi sp,sp,16
jalr x0,0(ra)

```

运行vi后，shell会找到下一个空闲的文件存储位置并存入文件基本信息。str_to_bin可以将表示二进制数的字符串转化为对应数字。

```

#str_to_bin
str_to_bin:
addi t1,x0,0
STR_TO_BIN_LOOP:
lw t0,(a0)
beq t0,x0,EXIT_STR_TO_BIN
addi t0,t0,-48
addi t2,x0,9
slli t1,t1,1
addi a0,a0,4
add t1,t1,t0
beq x0,x0,STR_TO_BIN_LOOP
EXIT_STR_TO_BIN:
addi a0,t1,0
jalr x0,0(ra)

```

mkdir指令

可以创建目录。

```

#mkdir
mkdir:
addi sp,sp,-16
sw a1,12(sp)
sw a0,8(sp)
sw s0,4(sp)
sw ra,(sp)
lw s0,current_dir
addi s0,s0,8
MKDIR_FIND_INDEX:
lw t0,(s0)
addi s0,s0,4
bne t0,x0,MKDIR_FIND_INDEX
addi s0,s0,-4
lw t1,next_index_addr
sw t1,(s0)
sw x0,4(s0) #Expand the directory list
lw t0,next_filename_addr
lw t1,next_index_addr
sw t0,(t1)
addi a1,a0,0
addi a0,t0,0
jal strcpy
jal strlen
addi a0,a0,1
slli a0,a0,2
la t0,next_filename_addr
lw t1,(t0)
add t1,t1,a0
sw t1,(t0) #update next_filename_addr
lw t1,next_index_addr
addi t0,x0,1
sw t0,4(t1) #Indicate a directory
la t0,next_index_addr
addi t1,t1,36
sw t1,(t0)
lw ra,(sp)
lw s0,4(sp)
lw a0,8(sp)
lw a1,12(sp)
addi sp,sp,16
jalr x0,0(ra)

```

cd /

可以工作目切换到根目录。

```
#cd /
cd_back:
addi sp,sp,-12
sw a1,8(sp)
sw a0,4(sp)
sw ra,(sp)
lw t0,current_dir_ptr #change to root directory
la t1,root
sw t1,(t0)
lw a0,current_full_dir
lw a1,current_dir
lw a1,(a1)
jal strcpy
lw ra,(sp)
lw a0,4(sp)
lw a1,8(sp)
addi sp,sp,12
jalr x0,0(ra)
```

主程序

首先进行指令输入，然后调用split判断参数的个数。对每种情形依次对支持的指令匹配，如果匹配成功则执行，否则进行下一次输入。

```

#main
main:
lw a0,current_full_dir
lw a1,current_dir
lw a1,(a1)
jal strcat
MAIN_LOOP:
# Print the working directory and prompt character(#)
lw a0,current_full_dir
jal printf
lw a0,prompt_str
jal printf
lw a0,buffer
jal scanf
jal split
addi s0,a1,0 #save a1
beq a1,x0,COMMAND_ONE_ARG
lw a1,cd_str
jal strcmp
addi a1,s0,0 #restore a1
beq a0,x0,CD_IMPL
lw a0,buffer
lw a1,vi_str
jal strcmp
addi a1,s0,0 #restore a1
beq a0,x0,VI_IMPL
lw a0,buffer
lw a1,mkdir_str
jal strcmp
addi a1,s0,0 #restore a1
beq a0,x0,MKDIR_IMPL
beq x0,x0,MAIN_LOOP
CD_IMPL:
addi a0,a1,0
jal cd
beq x0,x0,MAIN_LOOP
VI_IMPL:
addi a0,a1,0
jal vi
beq x0,x0,MAIN_LOOP
MKDIR_IMPL:
addi a0,a1,0
jal mkdir
beq x0,x0,MAIN_LOOP
COMMAND_ONE_ARG:
addi s0,a0,0
lw a1,ls_str
jal strcmp
beq a0,x0,LS_ONEARG_IMPL
addi a0,s0,0

```



```
jal run #Run subprocess
addi a0,x0,10 #Print \n
jal putchar
beq x0,x0,MAIN_LOOP
LS_ONEARG_IMPL: #implement ls with only one argument
lw a0,current_dir
jal ls
beq x0,x0,MAIN_LOOP
EXIT:
```

.text段入口代码

调整堆栈指针后直接跳转到主函数。

```
.text
lw sp,base
addi sp,sp,1000
addi sp,sp,1000
addi sp,sp,1000
addi sp,sp,1000
jal main
```

实验总结

本次实验收获良多，加深了计算机组成原理的各方面的素质和理解。

感谢老师，感谢助教对我们实验提供的帮助。

辛苦了！