数字逻辑与计算机组成实验

实验十二　计算机系统

201870049 朱卓然
201850137 肖　磊

June 9, 2022

Contents

1 实现功能

1.1 硬件部分

- 实现了具有 RISC-V 指令集的单周期 CPU.
- 实现了 VGA, 键盘, LED 灯等外设的调度
- 实现了通过内存映射方式完成 CPU 与外设通信.
- 实现了 256KB 指令存储器.
- 实现了 128KB 数据存储器.

1.2 软件部分

- 用 c 语言实现了一个简易操作系统.
- 用软件完成了对键盘, 显存的控制.
- 能够解析简单的命令, 以及完成简单的表达式计算.
- 实现了一些 c 库, 如 memcpy, strcpy.

1.3 交互部分

- 通过地址映射, 软件可获得键盘缓冲区的数据.
- 通过地址映射, 软件可向显存中输出字符.
- 通过地址映射, 软件可控制 LED 灯.
- 通过地址映射, 软件可获得时间.
- 通过地址映射, 软件可操控 VGA 起始行号寄出器.

2 组内分工

2.1 朱卓然同学

- 工作量约 30 小时, 主要负责软件部分.
- 完成软件环境的搭建, 编写了 Makefile 以及 sections.ld.
- 完成了 c 库函数, strcmp, strcpy, itoa, atoi, memset.
- 完成了命令解析.
- 完成了表达式求值.

2.2 肖磊同学

- 工作量约 30 小时, 主要负责硬件部分.
- 实现了具有 RISC-V 指令集的单周期 CPU.
- 实现了 4096 字节的显存, 完成了显存的地址映射.
- 提出键盘缓冲区的构想并实现, 完成键盘的地址映射.
- 实现了时钟, 并为软件提供时钟接口

3 硬件部分

3.1 单周期 CPU 的设计

3.1.1 PC

周期开始的下降沿将同时用于写入 PC 寄存器和读取指令存储器. 由于指令存储器要在下降沿进行读取操作, 而 PC 的输出要等到下降沿后才能更新, 所以不能拿 PC 的输出做为指令存储器的地址. 可以采用 PC 寄存器的输入, NextPC 来做为指令存储器的地址。该信号是组合逻辑，一般在上个周期末就已经准备好. 相应的代码如下:

```
wire [31:0] NextPC;
reg [31:0] PC;
assign imemaddr = (reset) ? 0 : NextPC;
assign dbgdata = (reset) ? 0 : PC;
assign imemclk = ~true_clock;
always @(negedge true_clock) begin
    if (reset)
        PC <= 0;
    else
        PC <= NextPC;
end
assign NextPC = genPC(Branch, zero, less, PC, imm, busA);
```

3.1.2 译码器

将指令读出后便可以使用组合逻辑进行译码, 可以产生控制信号, 寄存器读写地址以及立即数. 相应的代码如下:

```
wire [31:0] instr;
wire [6:0] opcode;
wire [4:0] rs1;
wire [4:0] rs2;
wire [4:0] rd;
wire [2:0] func3;
wire [6:0] func7;
assign instr = imemdataout;
assign opcode = instr[6:0];
assign rs1 = instr[19:15];
assign rs2 = instr[24:20];
assign rd = instr[11:7];
assign func3 = instr[14:12];
assign func7 = instr[31:25];
```

控制信号的生成主要由组合逻辑 case 语句实现, 对不同的 opcode 产生不同的控制信号即可, 代码过于冗长, 在这里不放出, 参考 RISC-V 控制信号列表即可.

3.1.3 寄存器堆

寄存器读可以通过组合逻辑直接读取即可. 当寄存器读地址由译码产生后, 直接读取两个源寄存器数据, 与立即数准备好, 一起送进 ALU 输入端. 寄存器写在下降沿进行, 这样在下一个周期数据就是最新的了. 相应的代码如下:

```
module regfile(
        input  [4:0]  ra,
        input  [4:0]  rb,
        input  [4:0]  rw,
        input  [31:0] wrdata,
        input  regwr,
        input  wrclk,
        output [31:0] outa,
        output [31:0] outb
);
```

```
11
12          reg [31:0] regs[31:0];
13          assign outa = regs[ra];
14          assign outb = regs[rb];
15          always @ (posedge wrclk) begin
16                  if(regwr)
17                          regs[rw]<=(rw==5'b00000)?32'b0:wrdata;
18          end
19
20  endmodule
```

3.1.4 ALU

ALU 也是组合逻辑电路, 在输入端准备好数据后就直接开始计算, 由于数据存储器的读地址也是 ALU 来计算的, 所以 ALU 的输出结果要在时钟周期的上升沿之前就准备好, 通过组合逻辑我们可以较为容易的实现这一点. 相关代码如下:

```
1   module alu(
2           input [31:0] dataa,
3           input [31:0] datab,
4           input [3:0]  ALUctr,
5           output less,
6           output zero,
7           output reg [31:0] aluresult);
8
9   //add your code here
10  wire cf;
11  wire of;
12  wire zf;
13  wire [31:0] datatemp;
14
15  assign less=aluresult[0];
16  assign zero=(ALUctr==4'b0010||ALUctr==4'b1010)?zf:~(|aluresult);
17
18  alu_s my_alu(dataa,datab,ALUctr,datatemp,cf,zf,of);
19
20  always @(*)
21  begin
22          case(ALUctr)
23                  4'b0000:aluresult=datatemp;
24                  4'b1000:aluresult=datatemp;
25                  4'b0001,4'b1001:aluresult=(dataa<<datab[4:0]);
26                  4'b0101:aluresult=(dataa>>datab[4:0]);
27                  4'b1101:aluresult=(($signed(dataa))>>>datab[4:0]);
28                  4'b0010:begin
29                                          if(dataa[31]==1&&datab[31]==0)aluresult=1;
30                                          else if(dataa[31]==0&&datab[31]==1)aluresult=0;
31                                          else
32                                          begin
33                                                  if(dataa<datab)aluresult=1;
34                                                  else aluresult=0;
35                                          end
36                                  end
37                  4'b1010:begin
38                                          if(dataa<datab)aluresult=1;
39                                          else aluresult=0;
40                                  end
41                  4'b0011,4'b1011:aluresult=datab;
42                  4'b0100,4'b1100:aluresult=dataa^datab;
43                  4'b0110,4'b1110:aluresult=dataa|datab;
44                  4'b0111,4'b1111:aluresult=dataa&datab;
45          endcase
46  end
47
48  endmodule
```

3.1.5 数据存储器

数据存储器读在上升沿进行, 其地址已经由 ALU 计算好, 数据存储器写在下降沿进行, 这样就不会相互干扰. 同时, 由于需要支持单字节以及双字节读写, 所以我们的数据存储器采用 IP 核的双口 RAM, 同时加上 byte enable 信号. 相应的代码如下:

```
module mem(
        input  [31:0] addr,
        output reg [31:0] dataout,
        input  [31:0] datain,
        input  rdclk,
        input  wrclk,
        input [2:0] memop,
        input we
);
wire [31:0] tempout;wire[14:0]address=addr[16:2];
wire [31:0]in_data=datain<<(8*addr[1:0]);
reg[3:0]byteena_a;
wire [31:0]out_data=tempout>>(8*addr[1:0]);

dmem_ram ram_1(byteena_a, in_data, address, rdclk, address, wrclk, we, tempout);

always@(*)begin
        case(memop[1:0])
        2'b10:begin//字节
                case(addr[1:0])
                2'b00:byteena_a=4'b1111;
                default:byteena_a=4'b0000;
                endcase
        end
        2'b01:begin//半字节
                case(addr[1:0])
                2'b00:byteena_a=4'b0011;
                2'b10:byteena_a=4'b1100;
                default:byteena_a=4'b0000;
                endcase
        end
        2'b00:begin//四分之一字节
                case(addr[1:0])
                2'b00:byteena_a=4'b0001;
                2'b01:byteena_a=4'b0010;
                2'b10:byteena_a=4'b0100;
                2'b11:byteena_a=4'b1000;
                default:byteena_a=4'b0000;
                endcase
        end
        default:byteena_a=4'b0000;
        endcase
end

always@(*)begin
        case(memop)
        3'b000:dataout={{24{out_data[7]}},
        out_data[7:0]};
        3'b001:dataout={{16{out_data[15]}},
        out_data[15:0]};
        3'b100:dataout={24'b0,
        out_data[7:0]};
        3'b101:dataout={16'b0,
        out_data[15:0]};
        default:dataout=out_data;
        endcase
end

endmodule
```

3.2 外设管理及内存映射

3.2.1 内存映射

- 0x00000000 - 0x000FFFFF 的地址空间分配给指令存储器.
- 0x00100000 - 0x001FFFFF 的地址空间分配给数据存储器.
- 0x00200000-0x002FFFFF 的地址空间分配给显存.
- 地址 0x00300000 作为键盘缓冲区.
- 地址 0x00400000 作为 VGA 起始行号寄存器.
- 地址 0x00500000 作为定时器.
- 地址 0x00600000 - 0x00600009 分配给 LED0 - LED9.

3.2.2 显存

CPU 只写型, 采用 IP 核双口 RAM 实现, 给显存分配 4096Byte 正好表示 64 行 64 列, 每一个 Byte 对应一个 ASCII 码, 由软件负责写入. 当地址高 12 位为 0x002 时, 在 CPU 时钟下降沿写入 ASCII 码. 同时设置一个 32 位起始行号寄存器, 当读写地址高 12 位为 0x004 时, 对其进行读写控制. 通过起始行号可以实现 VGA 滚屏功能. 相关代码如下:

```
assign vga_en = (dmemaddr[31:20] == 12'h002) ? dmemwe : 1'b0;
assign vreg_en = (dmemaddr[31:20] == 12'h004) ? dmemwe : 1'b0;
assign rd_addr = ((((v_addr >> 4) + vga_reg) % 64) << 6) + h_addr / 9;
VRAM video(dmemdatain[7:0], rd_addr, CLOCK_50, dmemaddr[11:0], dmemwrclk, vga_en,
    rd_ascii_key);

always @(posedge dmemwrclk) begin
        if (vreg_en)
                vga_reg <= dmemdatain;
        else
                vga_reg <= vga_reg;
end
```

3.2.3 键盘

CPU 只读型, 采用循环缓冲区实现, 分配一个能够存储 16 个 8bit 的 buffer, 设置读写指针 r_ptr 和 w_ptr, 键盘硬件只写, 在 buffer[w_ptr] 写入 ASCII 码, 然后 $w\_ptr + 1$; CPU 只读, 当 $r\_ptr == w\_ptr$ 时, 说明缓冲区为空, 返回 0, 否则读取 ASCII 码, $r\_ptr + 1$. 具体代码如下:

```
assign kbd_rden=(dmemaddr[31:20] == 12'h003) ? 1'b1 : 1'b0;
always @(posedge dmemrdclk)
begin
        if(kbd_rden)
        begin
                if (r_ptr==w_ptr)
                        bufferout<=32'h0;
                else
                begin
                        bufferout<={24'h0, buffer[r_ptr]};
                        r_ptr<=r_ptr+1;
                end
        end
end
```

3.2.4 定时器

CPU 只读型, 生成一个 1HZ 的时钟驱动定时器寄存器, 时钟上升沿数值 +1, 由此可以得到开机的时间.

```
wire TIME_CLK;
reg [31:0] TIME_CNT;
always @(posedge TIME_CLK)
    TIME_CNT <= TIME_CNT + 1'b1;
clkgen #(1) my_timeclk(CLOCK_50, SW[0], 1'b1, TIME_CLK);
```

3.2.5 LED

CPU 只写型, 用地址 0x00600000 - 0x00600009 分别代表 LED0 - LED9, 使用一个数组和这个九个 LED 一一对应, 每次 CPU 要写入的时候获取 CPU 传递的低位偏移量即可. 相应代码如下:

```
reg led_reg [0:9];
assign led_en = (dmemaddr[31:20] == 12'h006) ? dmemwe : 1'b0;
assign LEDR[0] = led_reg[0];
assign LEDR[1] = led_reg[1];
assign LEDR[2] = led_reg[2];
assign LEDR[3] = led_reg[3];
assign LEDR[4] = led_reg[4];
assign LEDR[5] = led_reg[5];
assign LEDR[6] = led_reg[6];
assign LEDR[7] = led_reg[7];
assign LEDR[8] = led_reg[8];
assign LEDR[9] = led_reg[9];
always @(posedge dmemwrclk)
    if (led_en)
        led_reg[dmemaddr[3:0]] = ~led_reg[dmemaddr[3:0]];
```

3.2.6 CPU 从数据总线获取的数据

由于 CPU 并不能感知到硬件的存在, 所以在读取数据存储器时, 需要对 CPU 选择的不同地址提供不同硬件的数据. 由于给不同硬件分配了不同的地址空间, 因此可以将读地址对应的所有内存块一次性读出, 再通过地址高 12 位选择正确的输出.

```
always @(*) begin
        case (dmemaddr[31:20])
        12'h001: tr_data_out = dmemdataout;
        12'h003: tr_data_out = bufferout;
        12'h004: tr_data_out = vga_reg;
    12'h005: tr_data_out = TIME_CNT;
        endcase
end
```

4 软件部分

4.1 对外设的处理

4.1.1 键盘

操作系统的主函数首先应当是停留在键盘缓冲区, 等待键盘的输入, 直到获得了一个回车符, 那么开始处理输入的指令. 其逻辑可表示为: 判断当前缓冲区的字符是否为 0, 如果是 0, 不执行任何操作, 继续循环; 如果非 0, 在屏幕上打印这个字符, 如果非 0 且为回车, 那么跳出循环, 开始处理输入.

4.1.2 VGA

主要实现一个 putch 函数, 向屏幕上打印指定字符. 打印字符时需要判断两种特殊字符: 回车和退格. 以及需要判断几种特殊情况: 是否需要将起始行号寄存器加一, 是否需要退回到上一行或进入到下一行, 核心代码如下:

```
void putch(const char ch) {
    if (ch == ENTER || ch == '\n') {
        vga_ch = 0;
        if (vga_line == 63)
            vga_line = 0;
        else
            vga_line += 1;
        if (*vga_reg == 63)
            *vga_reg = 0;
        else if (vga_scroll != 0)
            *vga_reg = *vga_reg + 1;
    } else if (ch == BACKSPACE) {
        if (vga_ch == 0) {
            if (judge != 0) {
                vga_ch = 63;
                if (*vga_reg == 0 && vga_scroll != 0)
                    *vga_reg = 63;
                else if (vga_scroll != 0)
                    *vga_reg = *vga_reg - 1;
                if (vga_line == 0)
                    vga_line = 63;
                else
                    vga_line -= 1;
            }
        } else {
            vga_ch -= 1;
        }
        vga_start[(vga_line << 6) + vga_ch] = 0;
    } else {
        vga_start[(vga_line << 6) + vga_ch] = ch;
        if (vga_ch == 63) {
            vga_ch = 0;
            if (vga_line == 63)
```

```
                vga_line = 0;
            else
                vga_line += 1;
            if (*vga_reg == 63)
                *vga_reg = 0;
            else if (vga_scroll)
                *vga_reg = *vga_reg + 1;
        } else
            vga_ch += 1;
    }
    if (vga_line == 29)
        vga_scroll = 1;
}
```

4.2 库函数的实现

为了处理命令的方便, 实现了四个 c 库函数: my_strcmp, my_memset, my_itoa, my_atoi. 这 strcmp 是字符串比较, 主要是为了解析命令而服务. memset 是设置内存, 有时候需要清空一个字符串, 需要使用 memset. itoa 和 atoi 分别是将整数转化为字符串数据以及将字符串转化为整型数据, 分别是为了在屏幕上打印整数和处理一些键盘输入的字符.

4.3 命令解析

主要通过自己实现的 c 库函数 my_strcmp 实现的, 根据输入字符串的不同, 有不同的处理逻辑.

```
size_t decoder(char *instr) {
    if (my_strcmp(instr, "hello") == 0)
        return HELLO;
    if (my_strcmp(instr, "time") == 0)
        return TIME;
    if (my_strcmp(instr, "clear") == 0)
        return CLEAR;
    char t = instr[4];
    instr[4] = 0;
    if (my_strcmp(instr, "fib ") == 0) {
        instr[4] = t;
        return FIB;
    }
    if (my_strcmp(instr, "LED ") == 0) {
        instr[4] = t;
        return LIGHT;
    }
    instr[4] = t;
    if (check(instr))
        return CALC;
    if (instr[98] != 0)
        return OVERFLOW;
```

```
24 }
```

4.4 hello 命令

实现较为容易, 调用封装好的 putstr 函数即可. putstr("Hello World!
n");

4.5 time 命令

首先需要从定时器中获得数据. 注意, 定时器中存储的是开机到现在的秒数, 所以需要将秒数转化为时分秒. 这个较为容易, 通过一些整除和模运算等操作完成即可. 然后将整型数据通过实现好的 c 库函数 my_itoa 转化为字符串, 调用 putstr 打印到屏幕上.

4.6 fib n 命令

首先调用已经实现好的 c 库函数 my_atoi, 将字符数据转化为整型数据. 然后用子函数 fibonacci 计算一下第 n 项即可.

4.7 clear 命令

此命令较为容易, 将显存全部置 0, 然后将行号列号以及初始行号寄存器都置 0 即可.

4.8 LED n 命令

较为容易, 两行 c 代码即可实现.

```
1 int num = my_atoi(instr + 4);
2 *(char *)(0x600000 + num) = 0;
```

4.9 表达式计算

4.9.1 合法性检查

首先需要对输入的表达式进行合法性检查, 我们选择用 FSM 有限状态机来实现. 可以先对字符进行编码, 编码的方式如下图所示.

单词符号	种别编码
+	1
-	2
*	3
/	4
数字	5
(	6
)	7

Figure 1: 字符编码

接着我们可以构建 FSM 的状态转换图, 这样通过判断字符结束时的状态就可以知道表达式是否合法.

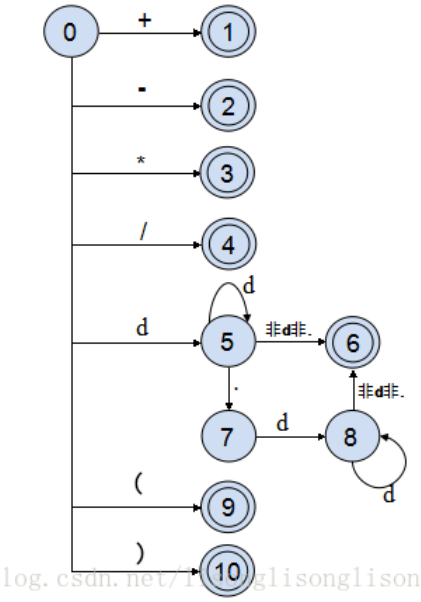

Figure 2: 状态转换图

4.9.2 表达式求值

若表达式是合法的, 那我们就可以开始对表达式进行求值. 首先将用户入的中缀表达式转化为后缀表达式. 在把中缀转后缀的过程中, 需要考虑操作符的优先级. 我们需要利用一个栈（存放操作符）和一个输出字符串 Output, 从左到右读入中缀表达式:

1. 如果字符是操作数, 将它添加到 Output.
2. 如果字符是操作符, 从栈中弹出操作符, 到 Output 中, 直到遇到左括号或优先级较低的操作符 (并不弹出). 然后把这个操作符 push 入栈.
3. 如果字符是左括号, 无理由入栈.
4. 如果字符是右括号, 从栈中弹出操作符, 到 Output 中, 直到遇到左括号.（左括号只弹出, 不放入输出字符串）
5. 中缀表达式读完以后, 如果栈不为空, 从栈中弹出所有操作符并添加到 Output 中.

得到了后缀表达式以后, 对后缀表达式的求值就变得非常简单了. 只需要使用一个栈, 从左到右读入后缀表达式:

1. 如果字符是操作数, 把它压入堆栈.
2. 如果字符是操作符, 从栈中弹出两个操作数, 执行相应的运算, 然后把结果压入堆栈.（如果不能连续弹出两个操作数, 说明表达式不正确）
3. 当表达式扫描完以后, 栈中存放的就是最后的计算结果.

5 遇到的问题以及解决办法

- 在编写 Makefile 时, 一开始总是不能成功 make, 查阅相关资料后发现是由于 Makefile 对缩进的要求比较严格, 如果不正确的缩进将无法正确编译.
- 刚开始上板的时候, 发现 CPU 连执行 putstr("Hello World") 都做不到. 我们首先验证了显存是否能正确输出, 发现显存并没有问题, 那么问题出在 CPU 对显存进行写入的时候. 于是将 CPU 的时钟信号由 CLOCK_50 换成 KEY[0], 通过按按键进行" 单步调试". 我们先将 PC 输出到七段数码管, 并使用 objdump 对生成的二进制文件进行反汇编, 观察 PC 的行为与汇编代码描述的行为是否一致. 验证后发现 PC 无误, CPU 在正确的执行每一条指令. 然而 CPU 并不能正确写入显存, 检查代码后发现是 CPU 从数据存储器读数据的时序出现了错误, 读出的数据慢了一个周期, 自然不能写入正确数据. 将 KEY[0] 换回 CLOCK_50 后又不能写入显存了, 于是推断出可能是时钟过快, 将时钟降频为 250000HZ 后能正确输出 hello world.
- 测试键盘时, 发现 CPU 大部分时间能正确读取键盘的输入, 但是会偶尔出现键盘输入的字符与 CPU 打印在屏幕上的字符不同的情况. 猜测可能是同时写入读取的冲突的问题, 因为键盘写入和 CPU 读取是由不同的信号操控的, 于是修改时序, 成功修改掉这个 bug.

6 实验结果

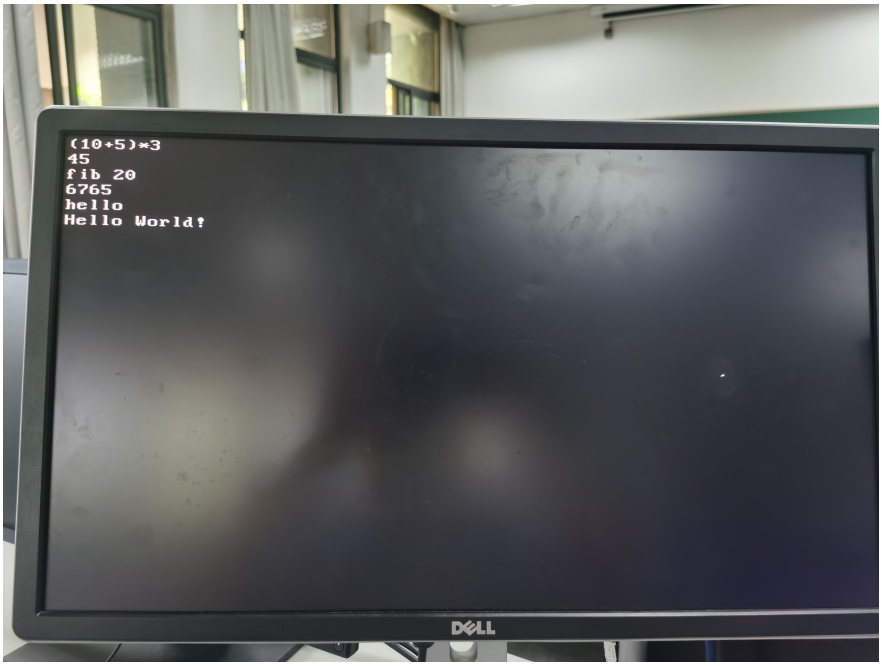

Figure 3: 测试 1

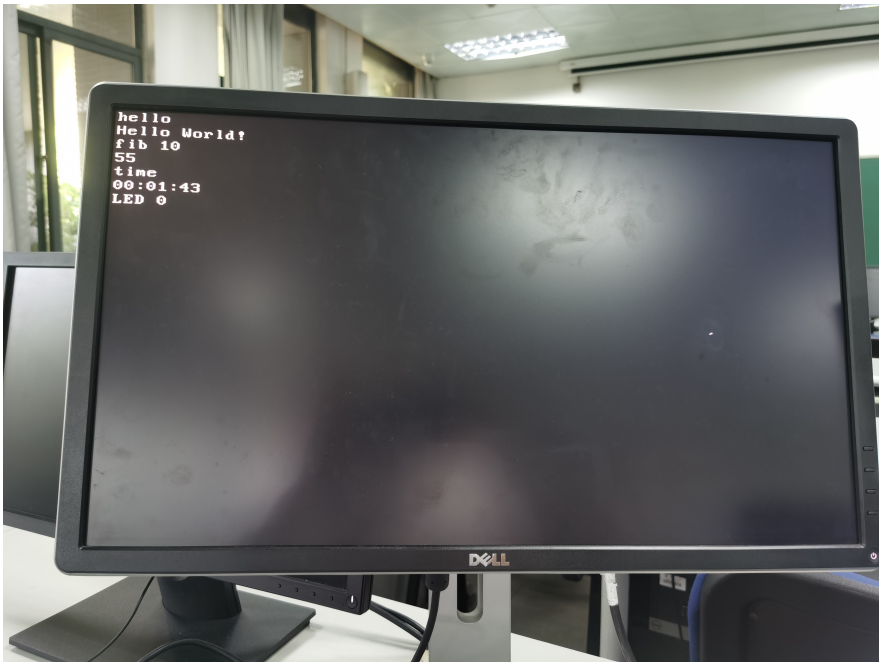

Figure 4: 测试 2

7 实验启示

- 存储器尽量使用 IP 核实现, 这样可以大幅度降低编译的时间
- 应当先保证硬件的正确性再上板, 而不是硬件和软件交互再一起之后上板 debug. 遇到 bug 可以使用仿真 debug, 这样可以较为容易的发现 bug 出现的原因.
- 一定要多交流, 有时你想不到的, 或许别人就有一个很好的解决方案, 互相查漏补缺, 效率是很高的.