南京大学

数电实验十二：计算系统

课程名称：数字逻辑与计算机组成实验

姓名：孙文博 ， 丁柯玮

学号：201830210 ， 201830192

班级：数电一班

邮箱：201830210@smail.nju.edu.cn

实验时间：2022.5.25 – 2022.6.11

目录

一、 实验目的

本实验的目标是在 DE10-Standard 开发板的 FPGA 上实现一个简单的计算系统，能够运行简单的指令，并处理一定量的输入输出。在所有功能开发完毕后，能够实现基本的 terminal 功能，即键盘输入命令，并在显示器上输出结果。

二、 实验环境

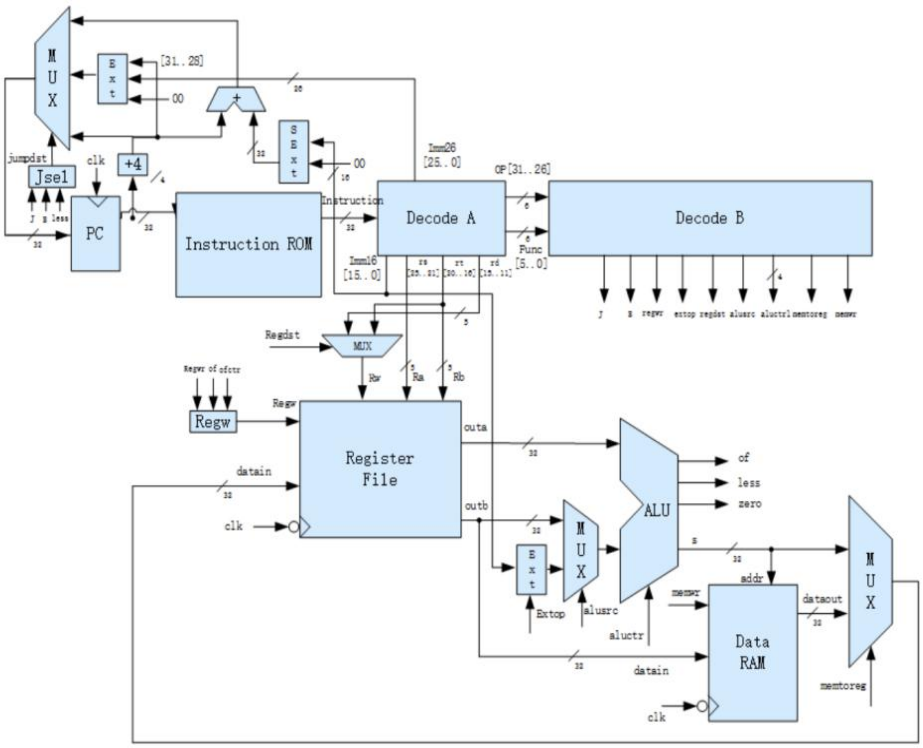
设计\编译环境：Quartus(Quartus Prime 17.1)Lite Edition

FPGA 芯片：Cyclone II 5CSXFC6D6

三、 实验原理

1． 总体架构

我们在头歌平台上已经完成了计算机硬件的核心 CPU 部分。在本实验中，我们将整合之前完成的外设部件，如 PS/2 键盘，VGA 显示器等，让 CPU 真正能与外设进行交互，这样就可以构成一个真正可以运行的计算机系统。主体部分 CPU 的结构如图：



主要组成部分包括指令计数器（PC）、二选一多路选择器、解码器、寄存器、ALU，另外有主存用于只读数据的存储和程序运行时栈的支持。

2. CPU 部分

- **寄存器**

CPU 内部寄存器与 MIPS 指令集要求的寄存器实现相同，一共有 32 个 32 位的寄存器，寄存器地址宽度为 5bit。注意 0 号寄存器的内容始终为 0。

- **指令**

CPU 指令部分实现了 MIPS 架构中除左移右移以外的算数和逻辑操作指令，部分 Load-Store 指令和分支跳转指令。指令的实现主要参

考实验手册十一中的指令表。由于系统实现过程中没有更多需要，因此没有对指令集进行过多拓展。 指令使用 256 个单元的 32 位 ROM 进行存储，系统运行时在 decode 模块进行指令解码之后转入相应流程执行。

3. 主存部分

主存部分使用 IP 核自动生成的 OnChipMemory 来实现，一共有 65536 个 32 位存储单元，按照字节编址，存储地址从 0x1000 0000 开始。主存一共划分为 5 个部分，各个部分大小和功能如下：

1. **VGA 缓存部分**：0x1000 0000 - 0x1000 20D0（共 8400 字节），CPU 可以通过对这一区域进行读写来改变屏幕显示内容；

2. **键盘缓冲区**：0x1000 20D0 - 0x1000 20D4（共 4 字节），键盘输入有效时输入对应的 ASCII 码将会被送入内存该地址处。

3. **输入缓存**：0x1000 2100 - 0x1000 2300（共 512 字节），CPU 将输入的字符串保存到该区域，待以后进行字符串比较和命令解析使用。

4. **只读数据段**：0x1000 2400 - 0x1000 2A00（共 1536 字节），用于存放常量和字符串，例如 "Hello World!" 字符串等。

5. **栈区**：0x7FFF F800 - 0x7FFF FFFF（共 8192 字节），用于为函数的过程调用提供支持。

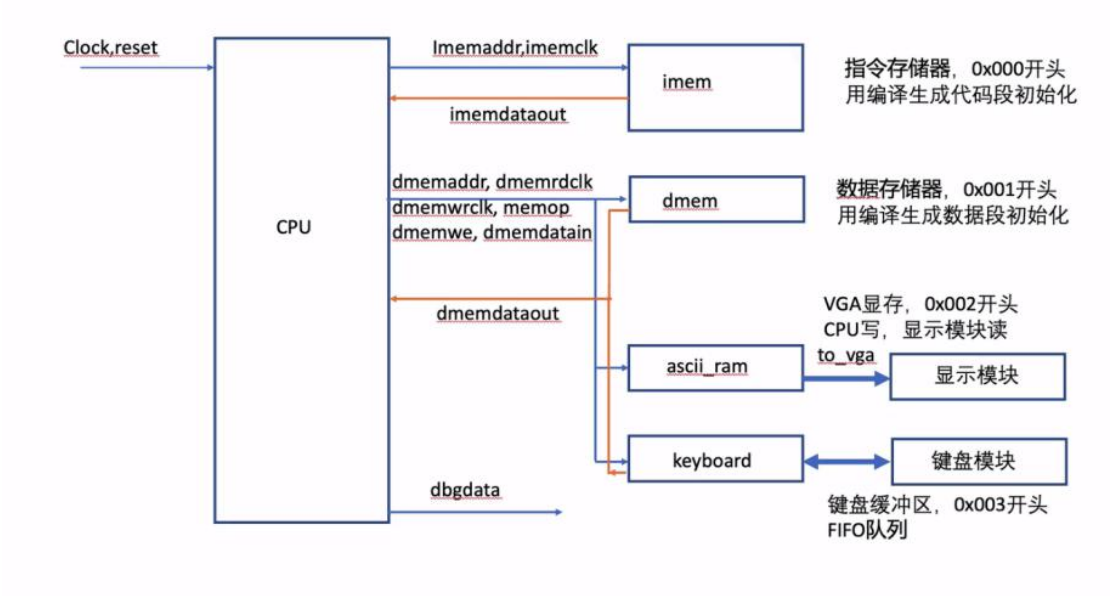
4. 各种外设

外设部分主要是键盘和显示器。本实验中外设的问题总体来说相对少一点，一方面是因为前面的实验多次用到键盘和显示器，可以实现代码复用，另一方面是本实验中大部分内容可以通过软件来实现，因此只需要编写合适的软件程序来完成各种字符显示、颜色更改等行为，这就简化了硬件的设计。

四、 实验步骤及代码实现

1. 硬件部分

首先，展示硬件的基本框架图如下，接下来我们分别将介绍各个部分的实现步骤。



1.1 单周期 CPU 模块

在实验 11 中，我们已经实现了一个基于 PV32I 指令集体系下的

单周期的指令执行 CPU 模块，此处我们只需沿用即可，具体的原理可参照实验 11 的实现报告以及实验手册。

CPU 的具体顶层文件代码如下：

```
module cpu(
    input    clock,
    input    reset,
    output [31:0] imemaddr,
    input  [31:0] imemdataout,
    output   imemclk,
    output [31:0] dmemaddr,
    input  [31:0] dmemdataout,
    output [31:0] dmemdatain,
    output   dmemrdclk,
    output   dmemwrclk,
    output [2:0] dmemop,
    output   dmemwe
    );
```

其余的 CPU 实现代码沿用实验 11 即可，不再赘述。

CPU 接入系统的调用模块代码：

```
cpu my_cpu(
.clock(cpu_clk),
.reset(1'b0),
.imemaddr(imemaddr),
.imemdataout(imemdataout),
.imemclk(imemclk),
.dmemaddr(dmemaddr),
.dmemdataout(ddata),
.dmemdatain(dmemdatain),
.dmemrdclk(dmemrdclk),
.dmemwrclk(dmemwrclk),
.dmemop(dmemop),
.dmemwe(dmemwe)
);
```

1.2 指令存储器的实现

指令存储器（instruction memory，简称 imem）用于存储所有的指令码，此处我们利用 verilog 的 IP 核生成方法生成了一个只读 RAM 来作为 imem，代号 0x000，其接入模块如下：

```
imem_rom my_imem(
.address(imemaddr[17:2]),
.clock(imemclk),
.q(imemdataout)
);
```

其作用便是通过 CPU 输入的 imemaddr （imem 取码地址）来到 imem 中取出相应的指令，然后反馈至 CPU 中执行。

1.3 数据存储器的实现

接下来要实现一个可读可写的 RAM 来作为我们存储数据的数据存储器（date memory,简称为 dmem）。这类存储器可以存储 ALU 模块运算后的数据，也可以将自身的数据传至 keyboard、CPU 等其余模块。其具体的调用模块如下：

```
147  mem my_dmem(
148  .addr(dmemaddr),
149  .dataout(dmemdataout),
150  .datain(dmemdatain),
151  .rdclk(dmemrdclk),
152  .wrclk(dmemwrclk),
153  .memop(dmemop),
154  .we(datawe)
155  );
156
```

具体模块的内部实现代码如下：

```
1   module mem(
2       input  [31:0] addr,
3       output reg [31:0] dataout,
4       input  [31:0] datain,
5       input  rdclk,
6       input  wrclk,
7       input [2:0] memop,
8       input we
9   );
10
11  wire [31:0] tempout;
12
13
14  wire[14:0]address=addr[16:2];
15  wire [31:0]in_data=datain<<(8*addr[1:0]);
16  reg[3:0]byteena_a;
17  wire [31:0]out_data=tempout>>(8*addr[1:0]);
18
19  dmem_ram ram_1(.byteena_a(byteena_a),.data(in_data),
20  .rdaddress(address),.rdclock(rdclk),
21  .wraddress(address),.wrclock(wrclk),
22  .wren(we),.q(tempout));
23
24  always@(*)begin
25      case(memop[1:0])
26      2'b10:begin//字节
27          case(addr[1:0])
28              2'b00:byteena_a=4'b1111;
29              default:byteena_a=4'b0000;
30          endcase
31      end
32      2'b01:begin//半字节
33          case(addr[1:0])
34              2'b00:byteena_a=4'b0011;
35              2'b10:byteena_a=4'b1100;
36              default:byteena_a=4'b0000;
37          endcase
38      end
39      2'b00:begin//四分之一字节
40          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
```

这样，dataout 便是我们执行后生成的数据，用于输出。

1.4 外设-键盘的接入及实现

在之前实验 7 我们已经实现了一个 ps2 键盘的功能，通过 ps2 键盘上对应按下的键码转换 ASCII 码，并将 ASCII 码数据传入 dmem 中处理，最后送达 CPU 进行取指令操作。此次实验我们只需要将其作为子模块接入我们的计算机系统中即可。

首先展示顶层调用键盘的模块代码：

```
keyboard_basic my_keyboard(
.clk(CLOCK_50),
.ps2_clk(PS2_CLK),
.ps2_data(PS2_DAT),
.data(keyboarddata)
);
```

内部代码和实验 7 报告中所展示的截图一致，具体不再赘述。

1.5 外设-VGA 映射实现

VGA 模块和键盘模块类似，沿用实验 8 和实验 9 的综合成果即可，

按照手册上描述的 30*70 的格式，利用模运算来实现滚屏功能中坐标

的重置。不过我们实现的 VGA 采取了覆盖的方式来滚屏，未能加入清屏的实现。

具体的调用代码图如下：

```
98  vga vga_top(
99  .VGA_CLK(VGA_CLK),
100 .CLOCK_50(CLOCK_50),
101 .we(vgawe),
102 .writedata(dmemdatain[7:0]),
103 .writeaddress(dmemaddr[11:0]),
104 .VGA_BLANK_N(VGA_BLANK_N),
105 .VGA_VS(VGA_VS),
106 .VGA_HS(VGA_HS),
107 .VGA_B(VGA_B),
108 .VGA_G(VGA_G),
109 .VGA_R(VGA_R),
110 .color(color_num),
111 .vga_mem_clk(dmemwrclk),
112 .vga_offset(vga_offset)
113 );
114
```

内部的代码和之前实验代码类似，如下图：

```
1   module vga(
2       input VGA_CLK,
3       input CLOCK_50,
4       input we,
5       input [3:0] color,
6       input vga_mem_clk,
7       input [7:0] writedata,
8       input [11:0] writeaddress,
9       output                  VGA_BLANK_N,
10      output                  VGA_VS,
11      output                  VGA_HS,
12      output          [7:0]   VGA_B,
13      output          [7:0]   VGA_G,
14      output          [7:0]   VGA_R,
15      input [31:0] vga_offset
16  );
17  wire [7:0]asc2;
18  wire [11:0]font;
19  wire [9:0]haddr;
20  wire [9:0]vaddr;
21  reg [11:0]vgadata;
22  reg [11:0]asc2_address;
23  reg [11:0]block_addr;
24
25  //reg reset;
26
27  vga_ctrl my_vga(
28  .pclk(VGA_CLK),//25MHz时钟
29  .reset(1'b0),//reset
30  .vga_data(vgadata),//vgadata[11:0]
31  .h_addr(haddr),
32  .v_addr(vaddr),
33  .hsync(VGA_HS),//列同步信号
34  .vsync(VGA_VS),//行同步信号
35  .valid(VGA_BLANK_N),//消隐信号
36  .vga_r(VGA_R),
37  .vga_g(VGA_G),
38  .vga_b(VGA_B)
39  );
40
```

```
40
41  reg [2:0] command_prompt=0;
42 always @(CLOCK_50)begin
43      block_addr <= (((vaddr >> 4)+vga_offset)%30) * 70 + ((haddr) / 9);
44  //  if(((vaddr >> 4)+vga_offset)%30==0)
45  //    reset = 1;
46  //  else
47  //    reset = 0;
48
49      asc2_address <= (asc2 << 4) + (vaddr % 16);
50      if(font[haddr % 9] == 1'b1&&haddr<=629) begin
51        if(color==3'b000)
52           vgadata<=12'b000000000000;
53        if(color==3'b001)
54           vgadata<=12'b000000001111;
55        if(color==3'b010)
56           vgadata<=12'b111100000000;
57        if(color==3'b011)
58           vgadata<=12'b111100001111;
59        if(color==3'b100)
60           vgadata<=12'b000011110000;
61        if(color==3'b101)
62           vgadata<=12'b000011111111;
63        if(color==3'b110)
64           vgadata<=12'b000011111111;
65        if(color==3'b111)
66           vgadata<=12'b111111111111;
67      end
68      else
69        vgadata<=12'b000000000000;
70 end
71  wire [11:0] backgrounddata;
72 vga_ram ram1(
73  .rdaddress(block_addr),
74  .wraddress(writeaddress),
75  .rdclock(CLOCK_50),
76  .wrclock(vga_mem_clk),
77  .data(writedata),
78  .wren(we),
79  .q(asc2)
```

```
80  );
81 vga_rom rom1(
82  .address(asc2_address),
83  .clock(CLOCK_50),
84  .q(font)
85  );
86  /*
87  backgroun rom2(
88  .address({haddr, vaddr[8:0]}),
89  .clock(CLOCK_50),
90  .q(backgrounddata)
91  );*/
92  endmodule
```

1.6 CPU-VGA 映射实现

我们实现的 VGA 显存由 IP 核生成，采用外设只读，CPU 只写的模式。我们先看一下具体的调用模块代码：

```
72 vga_ram ram1(
73  .rdaddress(block_addr),
74  .wraddress(writeaddress),
75  .rdclock(CLOCK_50),
76  .wrclock(vga_mem_clk),
77  .data(writedata),
78  .wren(we),
79  .q(asc2)
80  );
```

其中， writedata,writeaddress,vga_mem_clk,we 是从 cpu 传过来的，下面是顶层模块对 vga 模块的调用代码图：

```
98  vga vga_top(
99  .VGA_CLK(VGA_CLK),
100 .CLOCK_50(CLOCK_50),
101 .we(vgawe),
102 .writedata(dmemdatain[7:0]),
103 .writeaddress(dmemaddr[11:0]),
104 .VGA_BLANK_N(VGA_BLANK_N),
105 .VGA_VS(VGA_VS),
106 .VGA_HS(VGA_HS),
107 .VGA_B(VGA_B),
108 .VGA_G(VGA_G),
109 .VGA_R(VGA_R),
110 .color(color_num),
111 .vga_mem_clk(dmemwrclk),
112 .vga_offset(vga_offset)
113 );
114
```

其中，.color（color_num）以及之后的几行是后加入的，是为了在字符显示屏幕上实现改变字符颜色的函数功能所需要的接口。

然后，滚屏的实现，我们在内存地址为 0x00800000 的存储单元里面存储 vga_offset,vga 显示模块从显存这一行作为起始行读，读到最后一行后，坐标 block_addr 就循环回第一行的首部，直到读出全部 30 行。具体就体现在下面这一行代码：

```
43  |    block_addr <= (((vaddr >> 4)+vga_offset)%30) * 70 + ((haddr) / 9);
```

其中 offset 是记录最近 30 行代码用的偏移量，最终 block_num 通过模运算回到起始位置。

1.7 CPU-键盘 映射实现

根据手册上给出的实现原理，键盘采取循环缓冲区的方式实现。

我们首先分配可以存放 16 或 32 个扫描码（4 字节）的内存空间，同时分别设置头指针 head 和尾指针 tail。此时，键盘硬件作为数据生成者负责写入缓冲区。而 CPU 是数据消费者，负责从缓冲区中读取数据。在这个数据结构中，头指针只有 CPU 会写入，尾指针和缓冲区只有键盘会写入，因此可以将 CPU 只写和外设只写的区域分开，不会读写冲突。键盘在每次收到一个新的按键时，读取 head 和

tail 两个指针，如果 tail=head-1，说明缓冲区已满，不能写入。否则，键盘就在 tail 处写入按键对应的扫描码，然后将 tail+1。

对于 CPU 来说，每次需要检查有无键盘输入时，可以首先读取 head 和 tail，如果 head==tail，说明缓冲区为空，没有键盘输入，CPU 可以直接返回继续其他工作。如果 head 和 tail 不相等，CPU 将 head 指针指向的扫描码拷贝入自己的内存中，再将 head+1，表明已经读取了该扫描码。这样，我们可以用缓冲区记录一部分按键，让 CPU 在忙的时候不会丢失按键，同时也可以实现非阻塞式读取按键。CPU 读取扫描码后可以用软件对通码和断码进行处理，并进行扫描码到 ASCII 的转换。

我们实现了 buffer 模块。在 buffer 模块内部，根据 keyboarddata 判断如果是输入了一个新的字符，就往缓冲区 tail 所指的地方写入字符，tail++ 。

对 CPU 来说，buffer 和数据存储器一样就是一个存储器件，buffer 根据 cpu 提供的时钟和读使能进行判断，如果 buffer 不为空（head!=tail），就返回对应的 asc2 码，head++，否则返回 8'b0，在软件中对根据读到的东西进行判断。

下面展示 buffer 部分的实现代码：

```
module buffer(
input [7:0] keyboarddata,
input re,
input clock,
output reg [7:0] bufferout,
input rdclk
);

reg [7:0] head=0;
reg [7:0] tail=0;
reg we=1;
reg [7:0] wrdata;
reg [7:0] buffer [7:0];
reg [7:0] lastinput;
always @ (posedge clock) begin
   buffer[tail][7:0]<=keyboarddata[7:0];
   if(keyboarddata!=lastinput && keyboarddata != 0) begin
      if(tail==3'b111)
         tail<=0;
      else
         tail<=tail+1;
   end
   lastinput=keyboarddata;
end
always @ (posedge rdclk) begin
if(re) begin
   if(head==tail)
      bufferout<=8'b0;
   else begin
      bufferout<=buffer[head];
      if(head==3'b111)
         head<=0;
      else
         head<=head+1;
   end
end
end
endmodule
```

1.8 CPU-其他外设 映射实现

接下来几个外设的实现原理很简单，具体看看代码即可。

- color

color 的调用模块代码如下：

```
 assign colorwe=(dmemaddr[31:20]==12'h004)?dmemwe:1'b0;
color color_mem(
.we(colorwe),
.wrclk(dmemwrclk),
.wrdata(dmemdatain[3:0]),
.q(color_num)
);
```

具体用于字体颜色的更改功能，color 的数据存储器中读取的是字体颜色的对应信息。

- 数码管

数码管的实现方法也是找一个内存单元存储相关信息，然后根据需求输出对应的单元内容即可，具体代码如下：

```
197   wire hexwe;
198   assign hexwe=(dmemaddr[31:20]==12'h007)?dmemwe:1'b0;
199   reg [7:0] hex [7:0];
200  always @ (posedge dmemwrclk) begin
201      if(hexwe)
202          hex[dmemaddr[3:0]][3:0]=dmemdatain[3:0];
203  end
204   bcd7seg bcd0(hex[0][3:0],HEX0);
205   bcd7seg bcd1(hex[1][3:0],HEX1);
206   bcd7seg bcd2(hex[2][3:0],HEX2);
207   bcd7seg bcd3(hex[3][3:0],HEX3);
208   bcd7seg bcd4(hex[4][3:0],HEX4);
209   bcd7seg bcd5(hex[5][3:0],HEX5);
210
```

2. 软件部分

2.1 基本环境

要实现软件部分的功能，我们需要安装 riscv32-unknown-elf 工具链，然后利用老师所给的 Makefile 代码，在我们的虚拟机 Linux 环境上进行编译。

另外，我们还需要一个 sections.ld 文件，来规定可执行文件的地址映射，该文件利用手册上的样例代码即可。

2.2 VGA 映射实现

VGA 的映射实现，本质上是访问 VGA 显示字符的内存部分，因此我们定义如下地址：

```
#define VGA_START 0x00200000

#define VGA_MAXLINE 30

#define VGA_MAXCOL 70

#define OFFSET 0x00800000
```

- **vga_init（）函数**

该函数的作用是将屏幕清屏。由于 VGA 字符存储方式不同，我

们修改了部分的参数，本质上是将 VGA 显存模块清零代码实现如下：

```
char* vga_start = (char*) VGA_START;
int vga_line=0;
int vga_ch=0;
int *offset = (int*)OFFSET;
void vga_init(){
   vga_line = 0;
   vga_ch =0;
   *offset = 0;
   for(int i=0;i<VGA_MAXLINE;i++)
      for(int j=0;j<VGA_MAXCOL;j++)
         vga_start[ (i * 70)+j ] =0;
   }
```

- **putch（）函数（打印字符）**

该函数的作用是在特定位置上打印出我们需要的字符，本质上其实就是往 VGA 内存里面写数据，之后许多拓展功能，如 echo、fib 函数都基于这个函数。代码实现如下：

```
void putch(char ch) {
   if(ch==0x08) //backspace
   {
      ..
```

```
        ..
    }
        if(ch==0x0A) //enter
    {
        ..
        ..
    }
    vga_start[ (vga_line * 70)+vga_ch] = ch;
    vga_ch++;
    if(vga_ch>=VGA_MAXCOL){
        vga_line++;
        vga_ch = 0;
    }
    return;
}
```

- **滚屏函数**

滚屏函数, 相当于 VGA 模块在读显存时加上一个 offset 值来保证显示的字符画面是最近输入的 30 行内容，代码实现如下：

```
if(ch=='\n' || ch == 0x0d) //enter
{
    if(off > 0 || (vga_line == VGA_MAXLINE - 1 && off == 0))
    {
```

```
   off++;
   *offset = off;
   }
   if(vga_line < VGA_MAXLINE - 1) vga_line++;
   else vga_line = 0;
   for(int j = 0;j<VGA_MAXCOL;j++)
   {
      vga_start[(vga_line * 70) + j] = 0;
   }
   vga_ch = 0;
   return;
}
vga_start[ (vga_line * 70)+vga_ch] = ch;
vga_ch++;
if(vga_ch>=VGA_MAXCOL)
{
   for(int j=0;j<VGA_MAXCOL;j++)
      vga_start[ ( vga_line * 70)+j ] =0;
   if(off > 0) {
      off++;
      *offset = off;
   }
```

```
    vga_line++;
    vga_ch = 0;
}
```

2.3 keyboard 映射实现

本质上，keyboard 映射就是读取 keyboard 内存中对应的数据，因此规定 keyboard 地址如下：

```
#define KEY_START 0x00300000
```

- **getchar（）函数**

getchar（）函数的功能就是从键盘数据的对应位置取出键盘的输入信息，代码实现如下：

```
char* key_start = (char*) KEY_START;
char getchar(){
    char ch = *keystart;
    return ch;
}
```

2.4 指令执行

这部分是查找命令并执行的模块，将记录的输入字符串与已经定义好的命令字符串做对比，若相同，则进行相应操作，否则输出“unknown command”，具体代码实现如下：

```
void make_command(char* command)
```

```
{
   if(command == hello_command)
   {
      putstr(hello);
      return;
   }
   ..（依次类推）
   ..
}
```

2.5　辅助函数

- **判断指令是否有效**

```
int substr(char* cmd, char* type) {
   for(int i = 0;type[i] != '\0';i++){
      if(type[i] != cmd[i])
         return 0;
   }
   return 1;
}
```

- **字符串转型 int**

在计算斐波那契数列的函数 fib（）中，我们会出现整数运算，最终得出的结果也是整数型 int，但是打印在屏幕上的是字符串，因此这里我们定义两个函数 i2a（）函数和 a2i（）函数：

```
void i2a(int num, char* str, unsigned int radix) {
   static const char index[] =
   "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   unsigned unum;
   int i = 0, j, k;
   if (radix == 10 && num < 0)
   {
      unum = (unsigned)(-num);
      str[i++] = '-';
   }
   else unum = (unsigned)num;
   do
   {
      str[i++] = index[unum % (unsigned)radix];
      unum /= radix;
   } while (unum);
   str[i] = '\0';
   if (str[0] == '-') k = 1;
   else k = 0;
```

```
    char temp;
    for (j = k; j <= (i - k - 1) / 2; j++)
    {
        temp = str[j];
        str[j] = str[i - j - 1];
        str[i - j - 1] = temp;
    }
}
int a2i(char s[]) {
    int i, n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9'; i++)
        n = 10 * n + (s[i] - '0');
    return n;
}
```

3. 软硬件映射

硬件上，对于 cpu 来说，所有的模块其实都是一个个存储器，不管是数据存储器还是指令存储器，显存还是键盘缓冲，在存储器写时钟上升沿到来的时候，我只把要写入的数据送给所有可以写的模块，但是高 12 位匹配的模块的写使能为 1；在存储器读时钟上升沿到来的时候，所有可以读的模块，都把对应地址的内容发给 CPU，CPU 根据地址高 12 位选择我要的数据即可。

```
77  always @ (*) begin
78      if(dmemaddr[31:20]==12'h001)
79          ddata=dmemdataout;
80      else if(dmemaddr[31:20]==12'h003)
81          ddata=bufferout;
82      else if(dmemaddr[31:20]==12'h005)
83          ddata=time_cnt;
84      else
85          ddata=32'b0;
86  end
```

五、 实验结果

1. 思考题

本次数电大实验无思考题。

2. 上板验收

经过两星期的不懈努力后，终于实现了一个相对完善的计算机系统（虽然还有些 bug 没有修复），可以上板检验我们的成果了!

本次实验实现了全部的基本功能，包括：

1. 接受键盘输入并回显在屏幕上；
2. 支持换行、删除、滚屏等操作；
3. 命令分析：根据键盘输入命令执行对应的子程序，并将执行结果输出到屏幕上：
   - 打入 hello，显示 Hello World!
   - 打入 time，显示时间
   - 打入 fib n，计算斐波那契数列并显示结果
   - 打入未知命令，输出 Unknown Command

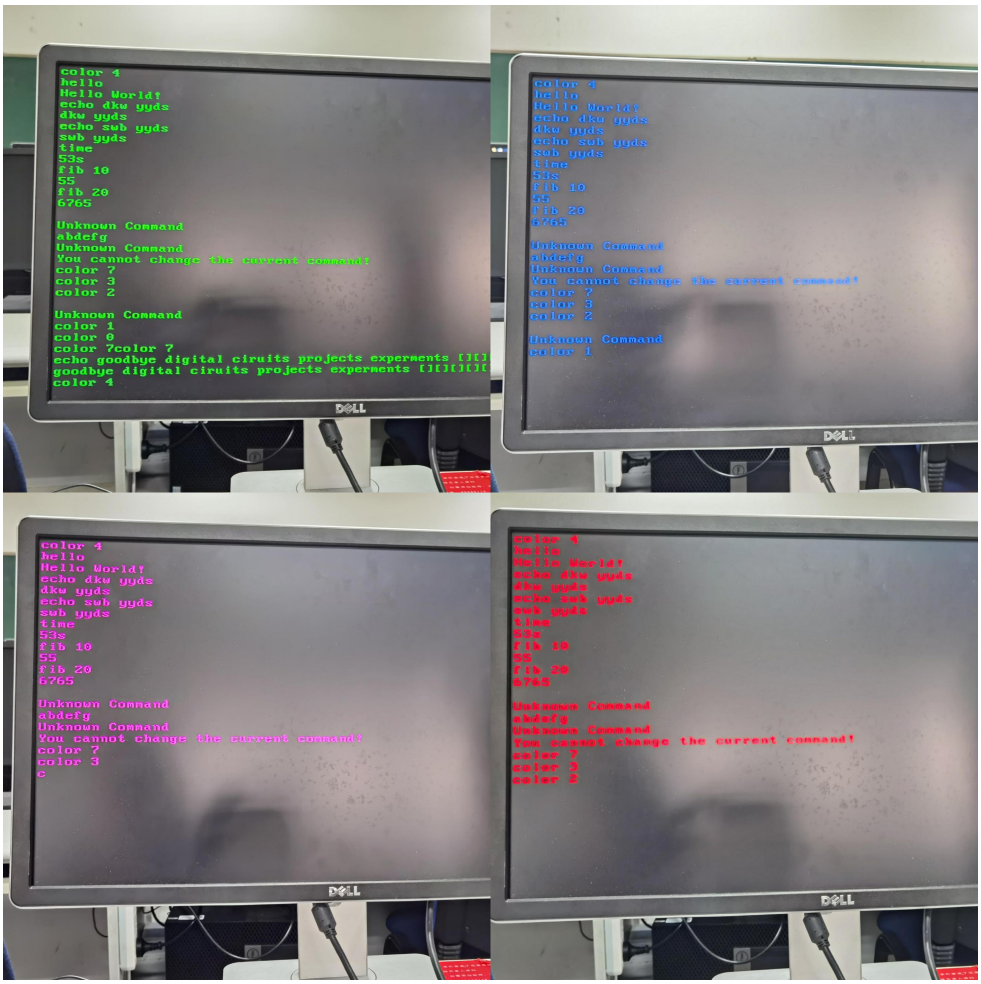
此外实现的拓展功能有：

1. 实现了一个 echo 函数，功能是重复输入字符串；

2. 支持对开发板 LED 灯、七段显示数码管的控制；

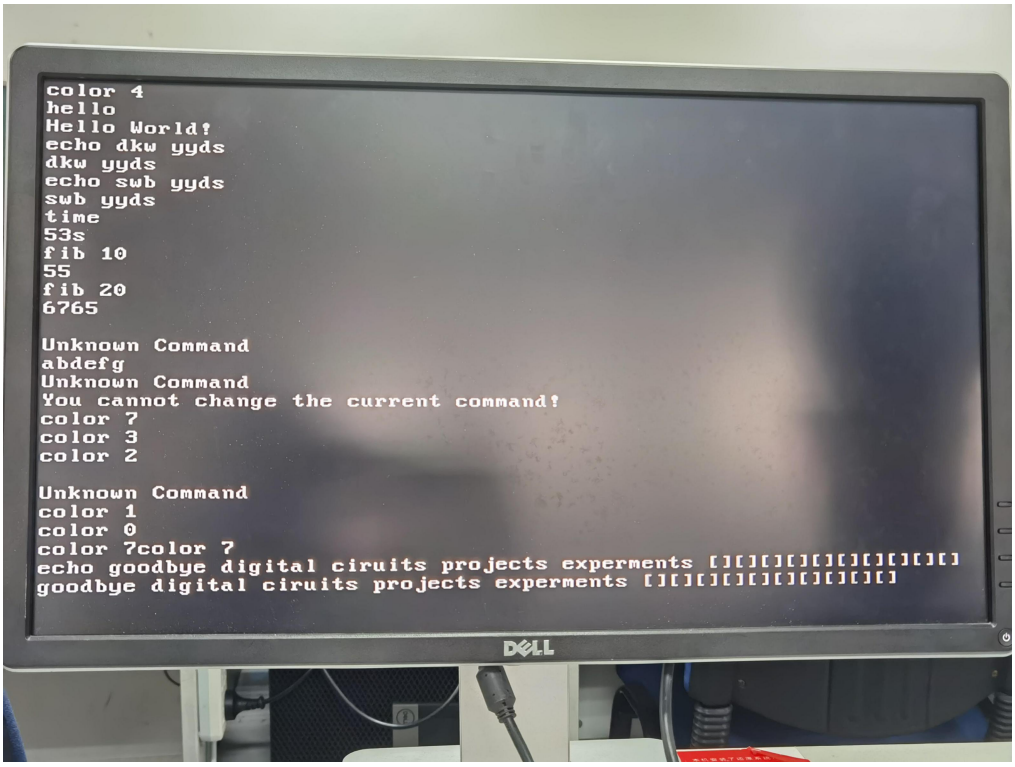
3. 实现了对字体颜色的控制，共有红黄绿蓝紫黑白等几种颜色。

上板验收的照片如下：



六、 问题与解决方法

本次实验中遇到的最大问题其实是**显示模块的设计方法问题**。



在之前的实验如显示器实验、字符交互实验中，显示模块的所有功能完全是通过硬件实现的，例如字符交互模块中命令提示符是通过每开辟一个新的行就用硬件向显存中写入固定内容的方法来实现的。因此开始做这个实验的时候十分自然地想用硬件的方式实现例如换行、命令提示符等功能。

但是随着实验的进行，用硬件进行编码的问题也逐渐暴露出来，一个是 bug 比较多，另一个是与计算机系统的实现目标不统一。后来想到，既然已经实现了计算机系统的软件部分，不如各种问题都使用软件编程实现，使用软件实现比硬件实现顺畅了很多。

七、 总结与反思

本次实验是数电实验的最终章，它不仅综合了前几次小实验中计

时器，ALU 等相关知识，还运用了七八九三个大实验中键盘与显示器的交互功能，当然最重要最核心的 CPU 则是基于十和十一两次头歌测试中的 Verilog 代码实现。随着最后一次验收结束，也意味着这学期的数电实验已经画上了一个圆满的句号。

当然整个实验并不是一帆风顺的，尤其是硬件部分出现的 bug，往往需要花很多时间上板进行 debug，不过也为以后的学习工作积累了一份经验。无论如何，这都是一段美妙而充满挑战的过程，也许是未来我们在计算机领域做各种工作，或是做各种科研的写照。那么就带着这份期待去迎接下一座高山吧，加油!

最后再次感谢一路以来相伴的老师和助教们，感谢参与编写实验手册的每一个人，感谢搭档 dkw 大佬，你们都是这段旅途的引路人，respect!

——by 孙文博

作为计科中重要的一门硬件实践课，数电实验陪我做过了大二下的全程，从单个加法器、时钟，到之后的字符显示界面，最后再到计算系统综合设计，我对于计算机内部的组成和工作原理有了更为深刻的了解，这是对上学期所学的数电理论课的知识的拓展、升华。

数电实验给我最大的收获，很大程度上是 verilog 给我带来的硬件语言编码的思考。曾经我一度认为硬件的工作效率仅仅取决于软件的算法好坏和硬件的材料与性能，但是最后阶段的综合性实验让我清楚了软硬件交互之间存在的一些问题，有些高效的算法不一定能够适

配相应的硬件。虽然看起来很高效，但是 compile 时会消耗大量的时间，软件语言不一定适应硬件语言。我们不能仅仅关注一些热门的编程领域，真正的设计应该从软硬件的综合方面考量。

另外，老师和助教，也教会了我，想要作为真正优秀的人才，我们应当重视深挖一些问题、重视原理性的探究。许多时候我们可能按照手册上的指示草草了事，但是老师和助教在验收时提出的一些问题，有时候恰恰反映了我们思考的不足，还有每次手册上提出的一些思考题，都很好地帮助了我们去思考整个实验内部的运行逻辑，帮助我们发现实现细节上的漏洞。

总之，在课程的尾声之际，衷心感谢各位老师、助教，还有陪我一路走来的搭档 swb 同学，希望我们能在旅途的末尾收获知识，收获友谊，也希望数电实验这门课程在未来熠熠生辉!

——by 丁柯玮