实验十二　计算机系统

2021 年秋季学期

"Next came a very large set of matched volumes containing reference materials: One contained designs for thousands of sleeve bearings, another for computers made of rods, still another for energy storage devices, and all of them were ractive so that she could use them to design such things to her own specifications. Then there were more books on the general principles of putting such things together into systems."

– "The Diamond Age: A Young Lady's Illustrated Primer", Neal Stephenson

12.1 实验目标

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机系统，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，希望能够完成基本的 terminal 功能，即键盘输入命令，并在显示器上输出结果。以下内容只是设计参考，本实验同学们可以自由发挥，完成自己设想的各种计算机功能。具体实现时可以根据自己的兴趣选做一部分或者进行裁剪和修改。

12.2 硬件部分

我们在上一个实验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，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。CPU 与外设的通信建议使用内存映射方式。即预先规定好特定的内存地址会对应到特定的外设的输入或输出上去，CPU 用普通的 load/store 命令来对这些内存地址进行读写来控制外设。这种方式下 CPU 硬件并不需要关心哪些内存地址是对应外设，哪些是对应数据 RAM。硬件只要将正确的数据地址和读写信号放在总线上，并将总线上的数据正确取回即可。具体数据的含义可以由软件来进行处理。而外设只需要关注自己对应的地址空间的一小块存储即可。

12.2.1 外设内存映射

CPU 的数据地址为 32 位，其可寻址空间可以达到 4G Byte。我们前一个实验实现的数据存储只占据了 128K Byte 的空间，因此我们可以将 CPU 的数据寻址空间进行重新规划来同时满足数据存储和外设通信的要求。

首先，可以将高 12 位地址为全零的空间，即 0x00000000 至 0x000fffff 分配给我们的指令存储器。 指令存储需要16位地址，256KB，指令存储器以0x000开头

其次，可以将高 12 位地址为 0x001 的地址空间分配给数据存储器，用于程序正常运行时需要的常量、全局变量、堆及栈空间。当然在我们的系统中指令存储器和数据存储器是分开的，实际上我们可以统一进行编址。

最后，可以分别为每个外设分配特定的空间，以地址高 12 位区分。例如，以 0x002 开头的地址可以分配给显示器，0x003 开头的地址可以分配给内存，依次类推。

CPU 在读写内存时，会将需要访问的 32 位地址放在数据地址总线上。此时，我们可以根据 CPU 访问地址的高 12 位来判断要使用具体那一片内存。在写入时，只将对应内存的写使能置为有效。在读取的时候，可以从多片不同的存储器中同时读取，最终根据高 12 位地址选择合适的数据放在 CPU 的数据总线上即可。基本思路与实验十中分四片 RAM 来提供 32bit 数据有些类似。

不同的地址区间可以采用不同的方式来实现。例如，数据存储段可以用实验十中的大容量 M10K 存储来实现。对于外设来说，其存储容量一般比较小，所以可以用较为自由的手写 RAM 方式来实现，只要注意读写时钟控制即可。

12.2.2 外设内存读写设计

映射到外设的内存一般会需要两个以上的读写端口（即两套地址线），一套供 CPU 使用，一套供外设使用。此时可能会出现读写冲突的情况。所以我们需要对不同的外设对存储空间的读写要求进行分析。典型的外设存储可能包含以下几种类型：

- **CPU 只读型**：此类外设主要包括定时器，或者是板上的开关等等。这类外设对应的空间比较小，一般只有 32 或 64bit。我们只需要将外设对应的 wire 型变量在时钟上升沿赋值给特定寄存器，在 CPU 读取地址匹配时将该寄存器的内容放置在 CPU 的数据总线上即可。
- **CPU 只写型**：此类外设主要是输出设备，例如显示器、LED 或七段显示

等。此时，CPU 总是在上升沿写入数据，外设可以根据自己的逻辑，每次下降沿读取对应的数据放置在自己的寄存器里用于驱动外设。如果有必要，在存储容量小时也可以使用类似寄存器堆的实现方式来非同步读取。

- **CPU 同时需要读写型**：这时同一周期内，CPU 和外设不能同时写入同一地址。此类情况非常少见，一般都可以通过设计将特定空间划分为 CPU 只读或只写区域。具体设计可以参考后面的键盘空间设计。

12.2.3 常见外设实现

下面简单介绍一下板上常见外设的实现方式。

LED: CPU 只负责写入，可以只用 32bit 单个寄存器实现。将此寄存器的对应比特直接 assign 给 LED 即可。CPU 软件在自身的存储空间保留一份 LED 的当前状态，在需要改变 LED 亮暗时可以对自身的副本进行操作，然后直接将副本 SW 至 LED 对应的地址。

七段数码管：CPU 只负责写入七段数码管的 BCD 码，同 LED 类似，也可以直接用 32bit 寄存器实现。

定时器：CPU 只读型。如果需要提供毫秒或微秒量级的定时器，可以用系统时钟分频实现 1 毫秒或 1 微秒的定时信号。通过计数器累加，并在每个时钟的上升沿将数据写入对应的定时器内存空间中。定时器宽度可以是 32 位或 64 位的。CPU 在需要访问定时器时可以直接用 load 指令读取对应定时器数据，这样就可以获取从开机至当前经过的毫秒或微秒数。该功能可以用于实现时钟、计算程序运行时间，有兴趣的同学还可以考虑用最小堆或时间轮的方法来实现操作系统中的各种定时功能。

开关：CPU 只读型，只需要将开关连到特定寄存器即可通过 Load 指令读取对应的开关状态。

显示器：CPU 只写型。可以为显示器分配一定量的字符显存，每 8bit 对应一个 ASCII 码。例如要支持 64 行 ×64 列的字符缓存只需要 4096Byte 即可。此类存储需要支持 CPU 不同位宽的 sb,sh,sw 指令。除此之外，可以再单独分配一些控制寄存器对应的内存空间。例如，可以分配一个起始行号寄存器，方便实现滚屏操作；也可以分配一个颜色控制寄存器来控制字符和背景颜色。CPU 在自身时钟的上升沿写入显存和显示控制寄存器。而显示器部分类似我们实验九中的功能，只需要负责从显存中读取 ASCII 码，并且正确输出即可。为实现滚屏等功能，显示器硬件可以从给定的一个起始行号开始读取 ASCII 码，随后显

示起始行号后 30 行的内容（行号可以是模 64 循环的）。这样就可以通过改变起始行号方便地实现滚屏。注意，建议显示器只忠实地显示显存中的 ASCII 字符，滚屏清零等逻辑由 CPU 软件实现。显示器读取显存可以用自己的 25MHz 时钟，如果 CPU 无法达到这个频率对于实现上影响也不是很大。即 CPU 写入可以比较慢，显示器仍然可以按自己的步伐来读取显存。

图 12-1: 显存组织方式示例

键盘: 键盘建议使用循环缓冲区的方式来实现。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。具体软件实现可以

自行设计。如果系统只有对 ASCII 码的读取需求，也可以在缓冲区中直接放置 ASCII 码，即扫描码到 ASCII 码的转换由硬件完成。此方案比软件转换的方案灵活性要差一些。

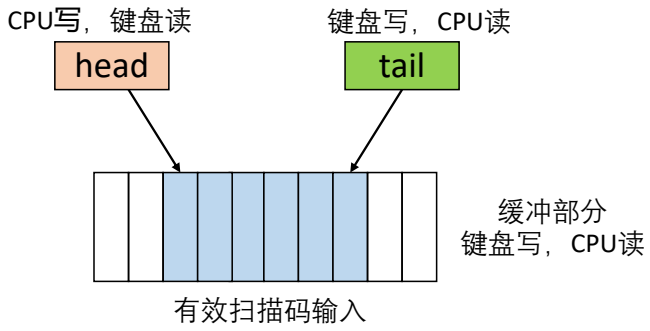

图 12-2: 键盘缓冲区组织方式示例

12.2.4 内存映射的具体实现

本实验设计中的数据 RAM 都是以双口方式实现的，即 RAM 提供单独的读地址、读数据输出口、读时钟，以及写地址、写数据输入口和写时钟。这种情况下 RAM 的读和写操作是完全分离的，可以分别单独进行设计。我们可以用单独的 RAM 或者寄存器来实现不同外设的数据存储空间。针对不同的外设的读写要求，将外设的数据线、地址线及 CPU 的数据线、地址线分别连接到对应外设的数据模块的读/写接口上即可。以数据存储为例，数据存储只需要与 CPU 相连，所以我们将读写地址、时钟和 memop 这些信号均直接与 CPU 相连。

```
1    dmem datamem(.rdaddr(drdaddr),     //CPU读地址
2                 .dataout(ddataout),   //数据存储读数据输出
3                 .wraddr(dwraddr),     //CPU写地址
4                 .datain(ddatain),     //CPU写数据输入
5                 .rdclk(drdclk),       //CPU读时钟
6                 .wrclk(dwrclk),       //CPU写时钟
7                 .memop(dop),          //CPU数据操作类型
8                 .we(datawe));         //数据存储写使能。
```

为了实现地址空间的分配，我们需要在 CPU 读写以 0x001 开头的地址时才对数据存储进行操作。因此，数据存储中有两个信号没有直接和 CPU 直连。对于写操作，只需要控制写使能就能确定是否需要对当前存储进行写操作，所以

可以先比较地址再确定是否写使能：

```
assign datawe=(dwraddr[31:20]==12'h001)? dwe:1'b0;
```

这句话判断写地址高 12 位是否为 0x001，如果是则写使能按照 CPU 输出的写使能 dwe 设置，否则总是不写入。对于读操作，我们可以将各个内存块读取的数据根据 CPU 读地址的高位来进行选择：

```
assign ddata=(drdaddr[31:20]==12'h001)? ddataout:
             ((drdaddr[31:20]==12'h003)? keymemout :32'b0 );
```

这里在地址高位为 0x001 时选择了数据存储的输出，为 0x002 时选择了键盘的输出。显示器由于 CPU 不用读取，所以没有设置。

12.3 软件部分

在实现了 CPU 及对外设的内存映射后，我们可以开始编写软件系统。我们建议大家使用 RISC-V 的 C 语言工具链对软件进行编译，这样可以使用 C 语言来编写整体代码。

12.3.1 编译过程介绍

我们需要利用上个实验中安装过的 riscv32-unknown-elf 工具链来编译系统软件。请自行编写合适的 Makefile 进行编译，我们提供了一个简单的 makefile 示例如下：

```
default: all

XLEN ?= 32
RISCV_PREFIX ?= riscv$(XLEN)-unknown-elf-
RISCV_GCC ?= $(RISCV_PREFIX)gcc
GCC_WARNS := -Wall -Wextra -Wconversion -pedantic -Wcast-qual -Wcast-align
    -Wwrite-strings
RISCV_GCC_OPTS ?= -static -mcmodel=medany -fvisibility=hidden -Tsections.ld
    -nostdlib -nolibc -nostartfiles ${GCC_WARNS}
RISCV_OBJDUMP ?= $(RISCV_PREFIX)objdump --disassemble-all --disassemble-zeroes
    --section=.text --section=.text.startup --section=.text.init
    --section=.data
RISCV_OBJCOPY ?= $(RISCV_PREFIX)objcopy -O verilog
```

```
RISCV_HEXGEN ?= 'BEGIN{output=0;}{ gsub("\r","",$$(NF)); if ($$1 ~/@/)
    {if ($$1 ~/@00000000/) {output=code;} else {output=1- code;};
    gsub("@","0x",$$1); addr=strtonum($$1); if (output==1)
    {printf "@%08x\n",(addr%262144)/4;}}
    else {if (output==1) { for(i=1;i<NF;i+=4)
    print $$(i+3)$$(i+2)$$(i+1)$$i;}}}'
RISCV_MIFGEN ?= 'BEGIN{printf "WIDTH=32;\nDEPTH=%d;\n\nADDRESS_RADIX=HEX;
    \nDATA_RADIX=HEX;\n\nCONTENT BEGIN\n",depth ; addr=0;} {
    gsub("\r","",$$(NF)); if ($$1 ~/@/) { sub("@","0x",$$1);
    addr=strtonum($$1);} else {printf "%04X : %s;\n", addr, $$1;
    addr=addr+1;}} END{print "END\n";}'

SRCS := $(wildcard  *.c)
OBJS := $(SRCS:.c=.o)
EXEC := main

.c.o:
    $(RISCV_GCC) -c $(RISCV_GCC_OPTS)  $< -o $@

${EXEC}.elf : $(OBJS)
    ${RISCV_GCC}  ${RISCV_GCC_OPTS} -e entry $(OBJS) -o $@
    ${RISCV_OBJDUMP} ${EXEC}.elf > ${EXEC}.dump

${EXEC}.tmp: ${EXEC}.elf
    $(RISCV_OBJCOPY) $<  $@

${EXEC}.hex: ${EXEC}.tmp
    awk -v code=1 $(RISCV_HEXGEN) $< > $@
    awk -v code=0 $(RISCV_HEXGEN) $< > ${EXEC}_d.hex

${EXEC}.mif: ${EXEC}.hex
    awk -v depth=65536 ${RISCV_MIFGEN} $< > $@
    awk -v depth=32768 ${RISCV_MIFGEN} ${EXEC}_d.hex > ${EXEC}_d.mif

.PHONY: all clean

all: ${EXEC}.mif
```

```
51
52 clean:
53     rm -f *.o
54     rm -f *.dump
55     rm -f *.tmp
56     rm -f *.elf
57     rm -f *.hex
58     rm -f *.mif
```

Makefile 所定义的内容主要包括：

- 第 3-6 行，定义工具链编译命令及 gcc 的告警级别。
- 第 8-9 行，定义 gcc 链接的参数，这里注意使用了静态链接，并且关闭了所有标准库的链接。除此之外，我们还使用了 sections.ld 来对二进制文件的各个段地址进行了规定，具体参见后面对 sections.ld 文件的解释。
- 第 10-13 行，类似上个实验中的内容，定义 OBJDUMP 和 OBJCOPY 的参数
- 第 14-19 行，利用 awk 对输出的十六进制文本文件进行转换，生成 verilog 可以读取的 4 字节 hex 文件。同时将代码段和数据段分开。
- 第 20-24 行，利用 awk 对 hex 文件进行改写，生成 Quartus 支持的 mif 文件分别初始化指令存储器和数据存储器。
- 第 26-35 行，扫描目录下所有.c 文件，预备生成所有对应的.o 文件，最后用所有的.o 文件来链接生成 main.elf 二进制执行文件。
- 第 37-46 行，在生成 main.elf 之后，将二进制文件分多步转换成对应的 main.mif 和 main_d.mif 来初始化指令存储器和数据存储器。
- 第 48-58 行，定义 make 的基本操作，make clean 清除所有输出文件。

同学们可以自行学习和了解 Makefile 的编写规则，按实际工程的需求对 Makefile 进行改写。

在 Make 过程中，我们利用 section.ld 文件来规定可执行文件的地址映射，该文件具体的示例如下：

```
1 ENTRY(entry)
2 OUTPUT_FORMAT("elf32-littleriscv")
3
4 SECTIONS {
5     . = 0x00000000;
6     .text : {
```

```
7           *(entry)
8           main.o (.text)
9           *(.text*)
10          *(text_end)
11      }
12      etext = .;
13      _etext = .;
14      . = 0x00100000;
15      .rodata : {
16          *(.rodata*)
17      }
18      .data : {
19          *(.data)
20      }
21      edata = .;
22      _data = .;
23      .bss : {
24      _bss_start = .;
25          *(.bss*)
26          *(.sbss*)
27          *(.scommon)
28      }
29      _stack_top = ALIGN(1024);
30      . = _stack_top + 1024;
31      _stack_pointer = .;
32      end = .;
33      _end = .;
34      _heap_start = ALIGN(1024);
35  }
```

该文件主要规定了二进制可执行文件的地址分配。首先，第 1 行说明了程序入口为 entry 函数（事实上在我们的系统里可以不用规定）。从第 5 行开始，我们规定了代码和数据的排布方式。最重要的是.text 代码段的规定。在硬件中我们规定了 reset 后从 0x00000000 地址开始执行，所以需要规定代码段从 0x00000000 开始，同时我们要把我们的入口函数 entry 放在代码段的开始位置，这是在第 7 行中规定的。后续的函数顺序可以自行定义。

在第 14 行规定了数据段的开始地址是 0x00100000，这和我们的硬件规定

是一致的。在 MakeFile 里，我们也会专门利用 awk 提取数据段内容（默认非代码段的数据全部放入数据存储），统一放入 main_d.mif 中用于初始化数据存储。

对于内存映射的输出输出地址空间我们不需要进行初始化，所以在二进制可执行文件中没有体现。

12.3.2 Hello World 示例

我们提供了简单的 Hello World 代码供大家参考，其中的 main.c 内容如下：

```
1  #include "sys.h"
2
3  char hello[]="Hello World!\n";
4
5  int main();
6
7  //setup the entry point
8  void entry()
9  {
10     asm("lui sp, 0x00120"); //set stack to high address of the dmem
11     asm("addi sp, sp, -4");
12     main();
13 }
14
15 int main()
16 {
17     vga_init();
18     putstr(hello);
19     while (1)
20     {
21     };
22     return 0;
23 }
```

main 文件首先包含了本系统自定义的头文件。第 3 行中用全局变量保存了 Hello World 字符串，该数据在编译后会放在数据段内。

程序入口：在代码的第 7-13 行，我们定义了一个入口 entry 函数。该函

数主要作用是初始化系统堆栈，并调用 main 函数。在链接过程中我们通过 section.ld 来将该函数置于起始 0x00000000 地址。使用 entry 函数的主要目的是初始化堆栈指针，在操作系统调用 main 函数前是会初始化 sp 的。但是我们的裸机程序中 sp 启动时总是全零，如果不初始化，main 函数第一句调整 sp 就会将 sp 变为 0xfffffff0，访问到我们地址空间不存在的部分。

```
00000030 <main>:
  30:   ff010113            addi    sp,sp,-16
  34:   00112623            sw  ra,12(sp)
  38:   00812423            sw  s0,8(sp)
  3c:   01010413            addi    s0,sp,16
  40:   1f8000ef            jal ra,238 <vga_init>
  44:   00100517            auipc   a0,0x100
  48:   fbc50513            addi    a0,a0,-68 # 100000 <hello>
  4c:   324000ef            jal ra,370 <putstr>
  50:   0000006f            j   50 <main+0x20>
```

而 entry 函数强制通过汇编重置了 sp，使其位于我们数据段的顶端 0x0011fffc 处，这样就保证了代码运行过程中堆栈是可以正常访问的。当然，当 main 函数返回 entry 之后系统状态会不正常，所以要求 main 函数不返回，在内部通过死循环 Halt。

```
00000000 <entry>:
   0:   ff010113            addi    sp,sp,-16
   4:   00112623            sw      ra,12(sp)
   8:   00812423            sw      s0,8(sp)
   c:   01010413            addi    s0,sp,16
  10:   00120137            lui     sp,0x120
  14:   ffc10113            addi    sp,sp,-4 # 11fffc <_end+0x1
     f7fc>
  18:   018000ef            jal     ra,30 <main>
  1c:   00000013            nop
  20:   00c12083            lw      ra,12(sp)
  24:   00812403            lw      s0,8(sp)
  28:   01010113            addi    sp,sp,16
  2c:   00008067            ret
```

main 函数执行的内容比较简单，主要是初始化 VGA 缓存，并且调用库函

数输出 Hello World，完成后进入死循环。

在 sys.h 头文件里，我们规定了 VGA 缓存的起始地址，VGA 行列数量等基本常数和部分库函数。同学们可以自己编写输入输出库函数或者移植 PA 中的部分代码。

```
#define VGA_START    0x00200000
#define VGA_LINE_O   0x00210000
#define VGA_MAXLINE  30
#define LINE_MASK    0x003f
#define VGA_MAXCOL   70


void putstr(char* str);
void putch(char ch);

void vga_init(void);
```

在 sys.c 中，我们实现了部分输出的库函数，包括 putch, putstr 等等。

```
#include "sys.h"

char* vga_start = (char*) VGA_START;
int   vga_line=0;
int   vga_ch=0;

void vga_init(){
    vga_line = 0;
    vga_ch =0;
    for(int i=0;i<VGA_MAXLINE;i++)
        for(int j=0;j<VGA_MAXCOL;j++)
            vga_start[ (i<<7)+j ] =0;
}

void putch(char ch) {
  if(ch==8) //backspace
  {
      //TODO
      return;
  }
```

```
21    if(ch==10) //enter
22    {
23        //TODO
24        return;
25    }
26    vga_start[ (vga_line<<7)+vga_ch] = ch;
27    vga_ch++;
28    if(vga_ch>=VGA_MAXCOL)
29    {
30       //TODO
31    }
32    return;
33  }
34
35  void putstr(char *str){
36      for(char* p=str;*p!=0;p++)
37        putch(*p);
38  }
```

在编译输出 main.mif 后，我们可以利用 Quartus 的 In-System Memory Content Editor 来更新指令存储器，不需要每次修改软件后再重新编译整个硬件，可以节省不少时间。由于我们的数据存储器是双口 RAM，Quartus 不支持对双口 RAM 进行实时修改。所以，如果代码中全局变量发生了变化，需要重新更新 main_d.mif 编译整个硬件。如果同学们需要实时对数据段进行更新，建议可以修改硬件，增加一个只读的数据存储，在 main 函数起始时通过 memcopy 把只读的存储拷贝到实际的数据存储器中去。

12.3.3 未实现指令的执行

RV32I 指令中没有常用的乘法指令，但是 gcc 在编译过程中如果遇到整数乘法会直接调用软件乘法函数 __mulsi3 来通过软件完成整数乘法操作。如果在代码中需要用到整数乘除法操作，可以参考 gcc 中的 __mulsi3 实现 (https://github.com/gcc-mirror/gcc/blob/master/libgcc/config/epiphany/mulsi3.c)，将对应的函数链接到主程序中即可。

```
1  unsigned int __mulsi3(unsigned int a, unsigned int b) {
2      unsigned int res = 0;
```

```
    while (a) {
        if (a & 1) res += b;
        a >>= 1;
        b <<= 1;
    }
    return res;
}

unsigned int __umodsi3(unsigned int a, unsigned int b) {
    unsigned int bit = 1;
    unsigned int res = 0;

    while (b < a && bit && !(b & (1UL << 31))) {
        b <<= 1;
        bit <<= 1;
    }
    while (bit) {
        if (a >= b) {
            a -= b;
            res |= bit;
        }
        bit >>= 1;
        b >>= 1;
    }
    return a;
}

unsigned int __udivsi3(unsigned int a, unsigned int b) {
    unsigned int bit = 1;
    unsigned int res = 0;

    while (b < a && bit && !(b & (1UL << 31))) {
        b <<= 1;
        bit <<= 1;
    }
    while (bit) {
        if (a >= b) {
```

```
            a -= b;
            res |= bit;
        }
        bit >>= 1;
        b >>= 1;
    }
    return res;
}
```

12.4 实验验收要求

12.4.1 上板验收

本实验以分组方式进行，只需要进行上板验收，实验验收时需要演示完整的计算机系统。其中必须要实现的功能包括：

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。

以上的计算机系统已经具有二十世纪八十年代末个人计算机的基本能力了，这样的系统可以完成很多有趣的功能。感兴趣的同学还可以自行思考，实现其他附加功能，附加功能可以考虑但不限于以下内容。

硬件可选实现的功能：

1. 实现五段流水线 CPU
2. 扩展 CPU 支持的指令类型，可以支持系统调用、乘除法或自定义指令
3. 支持简单的中断机制
4. 增加外设种类，支持对板载 LED、七段显示等控制
5. 增加显示器复杂度，提供简化的图形界面

软件可选实现的功能:

1. 输入简单表达式，如 (9+6*6)-3，输出结果
2. 实现简单的 C 语言库函数，移植其他课程，例如 PA 中的 AM 的代码
3. 编译运行流行的 benchmark，对 CPU 进行性能评估