Lab11 计算机系统设计

计算机系统包括处理器、存储器和输入输出部件。本次实验完成最基本的计算机系统，即通过键盘输入命令，在基于 FPGA 的 CPU 软核中进行处理，在显示器上输出结果，类似一个命令行 terminal 处理窗口。

本次实验内容可作为基础设计内容，进行提升和创新设计，开发新的实验内容。

一、实验目的

1. 掌握计算机系统中外部设备的调试方法。
2. 学习计算机系统设计的基本方法。
3. 学习Verilator仿真工具的使用方法。
4. 学会使用RISC-V GNU工具链。

二、实验环境

1. Vivado 开发环境
2. Xilinx A7-100T 实验板
3. RARS 汇编和 RISC-V 实时模拟器
4. Ripes 汇编编辑器和 RISC-V 模拟器
5. Verilator 仿真工具

三、实验原理

1、硬件部分

在本实验中，将整合之前完成的输入输出设备部件，让 CPU 能够稳定可靠地与外设进行信息交互，能够构成一个真正可运行的计算机系统。

CPU 与外设的通信可以使用内存映射方式，即预先定义特定的内存地址映射到特定外设的输入或输出端口，CPU 通过 load/store 命令对这些内存地址进行读写操作来控制外设。在这种方式下 CPU 不需要区分

哪些内存地址是对应外设，哪些是对应数据存储器。硬件部件将需要处理的数据地址和读写信号放在总线上，并将总线上的数据取回即可。每个外设只需要关注自己映射的一小块存储空间即可。

外设内存映射

RV32I 的数据地址为 32 位，其可寻址空间可以达到 4GB。需要将 CPU 的数据寻址空间进行规划，能够同时满足数据存储和外设数据交互的需求。下面提供一种简单的存储空间编址划分方法。

首先，将高 12 位地址为全零的空间，即 0x000 00000 至 0x000 fffff 分配给指令存储器，也就是指令存储段的最大存储空间是 1MB。

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存储需要的常量、全局变量、堆及栈等数据空间。在实际的计算机系统中指令存储器和数据存储器是可以分开的，但是可以进行统一编址。

最后，将高 12 位其它地址分别为每个外设分配特定的空间，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键盘，依次类推。

CPU 在读写内存时，将需要访问的 32 位地址放在地址总线上，此时，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。

不同的地址段的存储空间可以采用不同的方式来实现。例如，数据存储器可以用大容量块存储器来实现。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分布式 RAM 方式来实现，只要注意读写时钟控制即可。

不同读写类型的内存段设计

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，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：主要包括定时器、实验板上的拨档开关等。这类外设对应的空间比较小，一般只有 32 或 64 位，只需要将外设对应的 wire 型变量在时钟信号有效时赋值给特定寄存器，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：主要是输出设备，例如显示器、LED 或七段数码管等。CPU 总是在时钟信号有效时写入数据，外设可以根据自己的时钟频率，在时钟信号有效时读取对应的数据，放置在自己的寄存器中，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 读写型：此类外设的存储空间，CPU 既能读又能写。比如键盘接口的存储空间设计。需要特别考虑读写有效时序，防止读写冲突以及数据冒险的情形。

C 语言中有一个重要的关键字 volatile 用于防止编译器对代码进行过度编译优化。例如，对于 CPU 只读型变量，对应地址的内存有可能会被外设改变。CPU 对外设轮询时可能是使用 while(data==0) {...}

这样的循环来等待外设输入的。但是，“智能” 的编译器在打开 O2 之类优化的选项时会试图优化这段代码。当循环内部没有改变 data 这个变量时，编译器可能认为这个循环是没有用的，只需要做一次判断。这样，“优化” 后的二进制代码就和原始代码的本意不一致了。在这种情况下，需要在 data 这个变量的申明中增加 volatile 修饰，说明该变量有可能被外界改变，避免编译优化出问题。

常见外设接口的实现

下面简单介绍实验板上常用外设的实现方式。

LED：CPU 只负责写入，可以只用一个 32 位寄存器实现。将此寄存器的对应比特直接映射到 LED 指示灯即可。CPU 系统在自身的存储空间通过特定寄存器保留一份 LED 的当前状态，在需要改变 LED 显示时修改特定寄存器进行操作，然后直接将该寄存器内容写入 SW 到至 LED 对应的地址中。

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

定时器：CPU 只读型。如果需要提供毫秒 ms 或微秒量级 us 的定时器，可以用系统时钟分频来实现。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。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，以及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。

拨档开关：CPU 只读型，只需要将开关输入引脚映射到特定寄存器，CPU 可通过 Load 指令读取对应的特定寄存器，从而读取到开关状态。

显示器：CPU 只写型。考虑到实验板上 FPGA 中的实际存储空间有限，可以只为显示器分配字符显示空间，每 8 位对应一个 ASCII 码。例如要支持 40 行×80 列的字符缓存只需要 3200 个字节即可。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。而显示器控制逻辑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，这样就可以通过改变起始行号方便地实现滚屏。注意，显示器只负责显示显存中的 ASCII 字符，滚屏、清屏等逻辑由上层滚屏控制软件实现。

键盘：通过使用循环缓冲区的方式来实现。首先分配可以存放 16 或 32 个扫描码的内存空间。同时，分别设置头指针 head 和尾指针 tail。此时，键盘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 等于 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，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，如图 11.1 所示。

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。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。

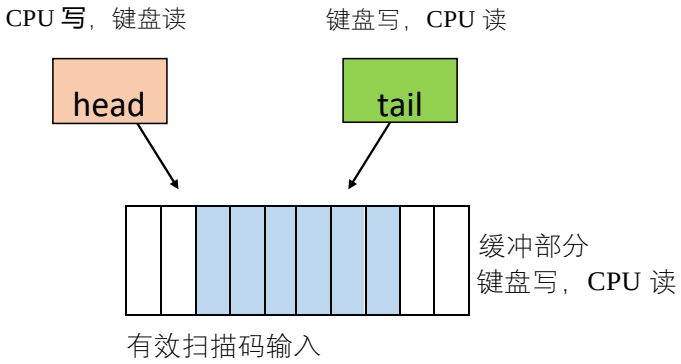

图 11.1　键盘缓冲区组织方式示例

此外，NexysA7-100T 实验板上还有 DDR 存储器、麦克风、三维陀螺仪、温度仪、网口等输入输出设备可以使用。

内存映射的实现

本实验设计中的数据 RAM 都是以双端口方式实现的，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，可以分别进行设计。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，所以将读写地址、时钟和 memop 这些信号均直接与 CPU 相连。

```
datamem dmem(.rdaddr(drdaddr),        //CPU 读地址
             .dataout(ddataout),     //数据存储器读数据输出
             .wraddr(dwraddr),      //CPU 写地址
             .datain(ddatain),       //CPU 写数据输入
             .rdclk(drdclk),         //CPU 读时钟信号
             .wrclk(dwrclk),         //CPU 写时钟信号
             .memop(dop),            //CPU 数据操作类型
             .we(datawe));           //数据存储器写使能
```

为了实现地址空间的分配，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;
```

这个语句判断写地址高 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，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 0x003 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。

2、软件部分

在定义了 CPU 及对外设的内存映射以及接口信号后，就需要编写软件系统。可以直接使用 Verilog 编程来实现；也可以使用 C 语言来编写整体代码，通过 RISC-V 的 C 语言工具链对软件进行编译。

编译过程介绍

需要提前安装 riscv32-unknown-elf 工具链来编译系统软件。然后编写合适的 Makefile 进行编译，这里提供了一个简单的 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)$$i;}}}'
RISCV_COEGEN ?= 'BEGIN{printf "memory_initialization_radix=16;\n
                       memory_initialization_vector=\n"; addr=0;} { gsub("\r","", $$(NF));
                        if ($$1 ~/@/) {gsub("@","0x",$$1);addr=strtonum($$1);
                          for(;addr<temp;addr++){print "00000000,";}} else {printf "%s,\n",
```

```
24.                         $$1; addr=addr+1;}} END{print ”0;\n”;}’
25.
26.
27.  SRCS := $(wildcard *.c)
28.  OBJS := $(SRCS: .c=.o)
29.  EXEC := main
30.
31.  .c.o:
32.     $(RISCV_GCC)   -c   $(RISCV_GCC_OPTS)   $<   -o   $@
33.
34.  ${EXEC}.elf   :   $(OBJS)
35.             ${RISCV_GCC}${RISCV_GCC_OPTS} -e entry $(OBJS) -o $@
36.             ${RISCV_OBJDUMP}   ${EXEC}.elf   >   ${EXEC}.dump
37.
38.  ${EXEC}.tmp :   ${EXEC}.elf
39.             $(RISCV_OBJCOPY) $ <   $@
40.
41.  ${EXEC}.hex :   ${EXEC}.tmp
42.          awk -v code=1 $(RISCV_HEXGEN) $<     >   $@
43.          awk -v code=0 $(RISCV_HEXGEN) $<     >   ${EXEC}_d.hex
44.
45. ${EXEC}.coe:   ${EXEC}.hex
46.          awk ${RISCV_COEGEN}   $<     >   $@
47.          awk ${RISCV_COEGEN}   ${EXEC}_d.hex   >   ${EXEC}_d.coe
48.
49.  .PHONY: all clean
50.
51.  all: ${EXEC}.coe
52.
53.  clean:
54.      rm -f *.o
55.      rm -f *.dump
56.      rm -f *.tmp
57.      rm -f *.elf
58.      rm -f *.hex
59.      rm -f *.coe
```

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，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，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 Vivado 支持的 coe 文件分别初始化指令存储器和数据存储器。注意，在生成 IP 核中的 BRAM 时，可以指定使用 coe 文件进行初始化，但是每次 coe 文件更改后需要重新生成 IP 核，在打包 IP 核的时候 Vivado 会将 coe 文件转换为 mif 文件，编译的时候用 mif 初始化内存。所以感兴趣的同学也可以考虑直接生成 mif 文件 hack 进 IP 核存储初始化文件。

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 36-45 行，在生成 main.elf 之后，将二进制文件分多步转换成对应的 main.coe 和 main_d.coe 来初始化指令存储器和数据存储器。

第 46-55 行，定义 make 的基本操作，make clean 清除所有输出文件。

可以在线学习和了解 Makefile 的编写规则，按实际需求对 Makefile 文件进行改写。

在 Make 过程中，利用 sections.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（在本次实验系统里可以不用规定）。从第 5 行开始，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，同时要把入口函数 entry 放在代码段的开始位置，这是在第 6 行中规定的。后续的函数顺序可以自行定义。

在第 13 行规定了数据段的开始地址是 0x00100000，这和硬件规定是一致的。在 MakeFile 里，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地址空间由于不需要进行初始化，因此在二进制可执行文件中没有体现。

Hello World 代码示例

提供了 Hello World 代码供参考，其中的 main.c 内容如下：

```
1.  #include ”sys.h”
2.  char hello[]=”Hello World!\n”;
3.  int main();
4.  //setup the entry point
5.  void entry()
6.  {
7.      asm(”lui sp, 0x00120”);   //设置栈段初始地址
8.      asm(”addi sp, sp, -4”);
9.      main();
10. }
11. int main()
12. {
13.    vga_init();
14.    putstr(hello);
15.    while (1)
16.    {
17.    };
18.    return 0;
```

19. }

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：代码的第 4-10 行定义了一个入口 entry 函数。该函数主要作用是初始化系统堆栈，并调用 main 函数。在链接过程中通过 sections.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，将访问到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    #00000 <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 main 函数返回 entry 之后系统状态就会不正常，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+0x1f7fc>
    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，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。可以自行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_MAXLINE     60
#define VGA_MAXCOL      80
void putstr(char* str);
void putch(char ch);
void vga_init(void);
```

在 sys.c 中，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;
      }
    if(ch==10) //enter
    {
        //TODO
        return;
     }
    vga_start[ (vga_line <<7 ) + vga_ch]= ch;
    vga_ch++;
    if(vga_ch>=VGA_MAXCOL)
    {
          //TODO
    }
    return;
 }
```

```
void putstr(char *str){
        for(char* p=str; *p!=0; p++)
        putch(*p);
}
```

如果需要实时对数据段进行更新，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。

乘除法指令的执行

RV32I 指令中没有常用的乘除法指令，如果在代码中需要用到整数乘除法操作，可以参考如下的函数设计，将对应的函数链接到主程序中即可。

```
unsigned int __mulsi3(unsigned int a, unsigned int b) {
    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;
}
```

3、调试工具

随着实验代码规模的不断扩大，编译及调试将花费越来越多的时间。因此，为了更好地完成综合实验，需要合理地开发调试工具。在综合实验中，每次硬件编译可能会耗费十几分钟到半小时不等，而且软件修改往往也需要重新编译。除此之外，硬件故障由于内部信号的不可见的特点，往往花费时间修改并编译后还不一定能迅速寻找到故障点。在本节中，将介绍通过串口来实现软件下载，CPU 重启等功能，并提供了示例代码。

串行接口将数据逐个比特地在单条线路上进行传输。这种传输模式由于使用到的数据线数量较少，因而在计算机和数字设备中得到广泛应用，特别是在嵌入式设备中通常缺乏常用的输入输出端口，往往选择使用串口来对设备进行调试和设置。在系统中实现一个简单的串口控制器往往能够大大减轻开发负担。

串行接口通常使用 9 针的 RS232 接口。主要通过接收数据 RXD 引脚、发送数据 TXD 引脚和地线 GND 传输。在接线时需要将收端和发送端的 RXD 和 TXD 的端口交叉连接，形成双向数据链路。串口传输一般支持全双工通信，即收发两端可以同时收发数据。

现代计算机已经使用 USB 接口取代了串行接口。本实验主要针对通用异步收发传输器 UART 来进行设计。UART 采用异步串行方式通信，收发两端不需要时钟同步。在异步条件下，接收端需要自己产生时钟来读取数据。典型的串口通信帧结构如图 11.2 所示：

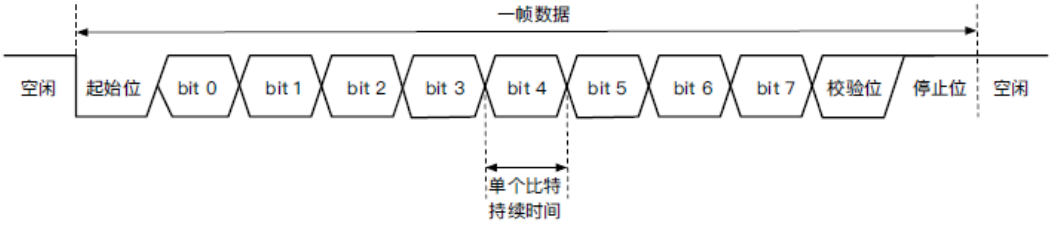

图 11.2 串口通信帧结构

类似于 PS/2 接口，每一帧的起始也是通过高电平变为低电平来指示，后续可以跟随 8 位数据位、一位校验位和一位高电平的停止位。由于是异步通信，接收端需要自己产生时钟。异步串行通信的时钟是由通信速率来决定的，一般称为波特率，也就是每秒钟传输的 bit 数。常用的串口通信波特率有 9600bps、115200bps 等等。除了波特率，串口通信一般还可以设置各种通信参数，包括数据位的位数、是否有校验位、停止位的长度等等。这些参数需要通信双方在通信前预先约定好。例如设置为 115200bps、8 位数据、奇校验及 1 位停止位。

在 Nexys A7-100T 实验板中，串口是通过 USB-UART 桥接芯片 FT2232 来实现的。串口和用于 FPGA 程序下载的 USB 接口共用一根线，互不干扰。也就是说在下载 bitstream 流文件后，FPGA 可以直接和电脑进行串口通信。串口的 TXD、RXD 及 CTS、RTS 分别连接 FPGA 的 C4、D4、D3 和 E5 接口。其中 CTS、RTS 用于硬件流控，可以不用处理，只需要将 CTS 置低即可，如图 11.3 所示。

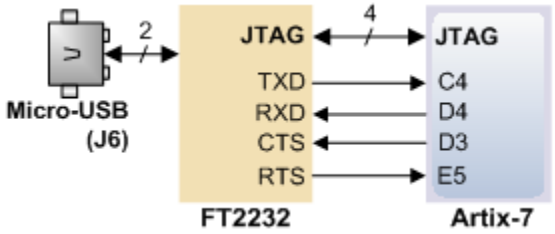

图 11.3 Nexys A7-100T 串口连接

实验板直接通过 USB 端口连接到电脑，调试时只需要安装支持 UART 软件即可，可以自行安装串口调试工具，例如应用商店里提供的 Serial Debug Assistant。在连接 USB 后，会看到 USB 转换的串口（一般是 COM4），通过该串口即可与 Nexys A7-100T 实验室板进行通信，如图 11.4 所示。Nexys A7-100T 实验板在 USB 接口附近有 2 个黄色的 LED，分别指示当前串口是否在收发。当串口调试工具打开串口，并开始通信时，收发 LED 会随每次收发闪烁，可以帮助确定收发是否正常。通过串口调试工具，可以向 FPGA 收发文本命令（注意，串口缺省是不回显的，即发出去的命令不一定会显示在接收窗口），或者直接发送内存初始化文件.hex 来直接更新指令或数据存储器。

图 11.4　串口调试工具界面

为简化实验，实验提供了基于 Nexys A7-100T 串口发送模块、串口接收模块以及相关的调试基础功能。感兴趣的同学可以在示例代码上增加自己需要的功能。。

串口发送的基本原理是将数据一位一位的按给定波特率发送到对端的 RXD 端口上去。串口发送端模块 serial_tx.v 的参考代码如下，其中输入时钟 clk 是 100MHz，用于产生 115200Hz 的发送时钟。八比特的数据输入 data 通过串行方式由送至 tx_out，最终该 tx_out 信号会连接到顶层模块中的 UART_RXD_OUT 信号上去。除此之外，模块还提供了输出 tx_ready 和输入 tx_enable 两个握手信号，用于和上层模块通信。

```
module serial_tx(
    input clk,                  //输入时钟 clk 是 100MHz，用于产生 115200Hz 的发送时钟
    input [7:0] data,       //数据输入 data 通过串行方式由送至 tx_out
    input tx_enable,        //握手信号，用于和上层模块通信
    output reg tx_out,     //连接到顶层模块中的 UART_RXD_OUT 信号
    output tx_ready         //握手信号，用于和上层模块通信
 );
    wire tx_clk;                                        // serial clock
    reg   tx_busy;
    reg [10:0] tx_buf;
    reg [3:0]  tx_count;
    initial
        begin
            tx_count= 4'd0;
            tx_busy = 1'b0;
            tx_out = 1'b1;
        end
```

```
    clkgen #(115200) mytxclk(clk,1'b0,1'b1,tx_clk); //always enable tx_clock

    assign tx_ready = ~ tx_busy;
    always@(posedge tx_clk)                     //state change only at the tx_clk edge
        begin
            if(tx_busy)                         //busy, ignore tx_enable and send
            begin
                tx_out<=tx_buf[0];
                tx_buf[9:0]<=tx_buf[10:1];   //shift tx_buf
                if(tx_count>=4'd10)                 //finished
                begin
                    tx_busy = 1'b0;
                    tx_count<=4'd15;
                end
                else
                begin
                    tx_busy = 1'b1;
                    tx_count<=tx_count+4'd1;
                end
            end
            else
            begin
                tx_out=1'd1;
                if(tx_enable)                   //load the buffer and start next cycle
                begin
                    tx_buf[0]<=1'b0;
                    tx_buf[8:1]<=data[7:0];
                    tx_buf[9]<=~(^data[7:0]);
                    tx_buf[10]<=1'b1;
                    tx_busy<=1'd1;
                    tx_count<=4'd0;
                end
                else //no data
                begin
                    tx_buf<=tx_buf;
                    tx_busy<=1'd0;
                    tx_count<=4'd0;
                end
            end
        end
endmodule
```

发送端的逻辑相对简单，在上层模块需要发送时，将数据 data 准备好，然后将 tx_enbale 置 1，此时发送模块将起始位、数据、校验位和停止位准备好，然后发送端变为忙 tx_busy，开始按位发送。发送完毕后将 tx_ready 置 1，准备下一次发送。

串口接收端类似 PS/2 接口，先对信号进行同步，寻找信号的沿，然后产生和发送端一样的时钟，并在时钟周期的中间来读取串口数据。PS/2 不同的是串口只有一根数据线，没有时钟，时钟需要接收端自己生成。接收端的接口包括了 100MHz 的时钟输入 clk 用于产生串口时钟。此处时钟注意要在收到第一个开始位的下降沿时 reset，保持与发送端同步。串口先通过 txd_in 信号接入，该信号最终会连接上顶层模块的 UART_TXD_IN 端口。串口接收到的数据通过 8 比特的 rx_data 送给上层模块。接收端与上层模块通过 rx_finish 和 rx_ready 两个信号来进行握手。顶层模块串口的另外两根信号线 UART_CTS 和 UART_RTS 在实际使用中可以缺省处理，及对于输出的 UART_CTS 可以直接置 0，对于输入的 UART_RTS 可以不进行判断。串口接收模块 serial_rx.v 参考代码如下：

```
module serial_rx(
    input clk,
    input txd_in,
    output reg [7:0] rx_data,
    output reg rx_finish,
    input rx_ready
);

  wire rx_clk;
  reg [2:0] clk_sync;
  reg [2:0] start_sync;
  reg [3:0] rx_count;
  reg [10:0] rx_buf;
  wire sampling;
  wire start_bit;
  reg rx_busy;
  reg rx_clk_rst;

  initial
  begin
    rx_finish =1'b0;
    rx_busy =1'b0;
    rx_clk_rst =1'b0;
  end
  clkgen #(115200) mytxclk(clk,rx_clk_rst,rx_busy,rx_clk); //rst clk at negedge
  always @ (posedge clk)
  begin
    clk_sync <=  {clk_sync[1:0],rx_clk};
```

```
end
assign sampling = ~clk_sync[2] & clk_sync[1]; //sampling of receive clock
// posedge because we start the clock at begining of the bit
always @ (posedge clk)
begin
    start_sync <=  {start_sync[1:0],txd_in};
end
assign start_bit = start_sync[2] & ~start_sync[1]; //find negedge of start bit
always @ (posedge clk) //triger the clock when find negedge
begin
    if(rx_busy) //receiving
    begin
        rx_clk_rst<=1'b0; //reset only last one clock
        if(rx_finish)
        begin
            rx_busy<=1'b0;
        end
        else
        begin
            rx_busy<=rx_busy;
            if(sampling)
            begin
                if(rx_count>=4'd10)
                begin
                    if( (rx_buf[0]==0)&&(^rx_buf[9:1]) )
                    begin
                        rx_data[7:0]<=rx_buf[8:1];
                        rx_finish<=1'b1;
                    end
                end
                else
                begin
                    rx_buf[rx_count]<= txd_in;
                    rx_count <= rx_count + 4'd1;
                end
            end
        end
    end
    else//not busy
    begin
        if(start_bit & (~rx_finish)) //start receiving when find start bit
        begin
```

```
                rx_busy<=1'b1;
                rx_clk_rst<=1'b1;
                rx_count<=4'd0;
                rx_data<=8'd0;
                rx_finish<=1'b0;
            end
            else
            begin
                rx_busy<=1'b0;
                rx_clk_rst<=1'b0;
            end
            if(rx_ready)
            begin
                rx_finish<=1'b0;
            end
        end
    end
    endmodule
```

UART 调试模块通过调用串口发送模块及接收模块来进行串口通信，根据串口的输入确定需要执行的调试动作，并对 CPU 进行控制。示例调试程序提供了基本的 CPU reset、指令存储器加载和数据存储器加载功能。同学们可以根据自己的需求来对调试模块进行改造，增加更多功能。例如，单步调试、显示 PC、显示指令和数据存储器中的数据、显示寄存器内容等等。

UART 调试模块 uart_test 的端口参考定义如下：

```
module uart_test(
    input    clk,
    input    clk100m,                    //收发模块的时钟 clk100m
    output rxd_out,                      //串口的发送数据线
    input    txd_in,                     //串口的接收数据线
    output reg [31:0] dbgdata,   //32 位调试信息，例如 PC、寄存器堆、存储器内容等等
    input    rst,
    output [31:0] instaddr,        //指令存储器地址
    output reg[31:0] instdata,   //指令存储器数据
    output reg iwe,                     //指令存储器写使能
    output [31:0] dataaddr,        //数据存储器地址
    output [31:0] mdata,             //数据存储器数据
    output reg       dwe,                  //数据存储器写使能信号
    output reg       cpuhalt,            //CPU 暂停
    output reg       cpureset            //CPU 重置
);
```

其中包括调试模块自身时钟 clk（建议与 CPU 时钟保持一致）及提供给收发模块的时钟 clk100m。串口的收发数据线 rxd_out 及 txd_in。串口调试工具对外提供的 32 位调试信息 dbgdata，用于显示调试结果，可以直接连在七段数码管上，例如 PC、寄存器堆、存储器内容等等。为了刷新指令存储器和数据存储器，调试工具还输出了指令存储器地址 instaddr，指令数据 instdata 和指令存储器写使能 iwe，以及数据存储器对应的地址、数据和写使能信号。为了实现对 CPU 的控制，调试模块还输出了 CPU 暂停 cpuhalt 和 CPU 重置 cpureset 信号。

调试模块对 CPU 的控制：需要对 CPU 输入端口进行简单修改实现对 CPU 的控制。

```
 //main CPU
SingleCycleCPU    mycpu(
                        .clock(clk&~uart_halt),
                        .reset(BTNC|uart_rst),
                        .InstrMemaddr(iaddr), .InstrMemdataout(idataout), .InstrMemclk(iclk),
                        .DataMemaddr(daddr), .DataMemdataout(ddataout), .DataMemdatain(ddatain),
                        .DataMemrdclk(drdclk),.DataMemwrclk(dwrclk), .DataMemop(dop),
                        .DataMemwe(dwe), .dbgdata(cpudbgdata)
                        );
```

这里将 uart_halt 信号取反后与 CPU 时钟进行了与操作。这样，当 uart_halt 是高电平时，CPU 时钟始终是为 0 的，即 CPU 不再动作。因此， 串口调试模块可以通过 uart_halt 来暂停 CPU 的动作，观察 CPU 各项信息，然后再将 uart_halt 置 0，让 CPU 继续执行。同样，CPU reset 的输入也从仅支持硬件按钮 BTNC 重启，修改为在 uart_rst 信号有效时也可以重启 CPU。

调试模块对存储器的控制同样的，调试模块也可以对指令存储器和控制存储器进行控制。例如，我们可以将指令存储器的接口进行如下改写：

```
InstrMem myinstrmem(
        .addra(uart_halt?uart_iaddr[17:2]:iaddr[17:2]),
        .clka(uart_halt?~uart_clk:iclk),
        .dina(uart_idata),
        .douta(idataout),
        .ena(1'b1),
        .wea(uart_iwe&uart_halt)
        );
```

在 uart_halt 信号有效时，CPU 暂停工作。同时，指令存储器的地址、时钟、输入数据和写使能均由 UART 调试模块来进行控制。这样，UART 调试模块可以在 CPU 停下时对指令存储器进行任意的读写操

作。这里需要注意的是存储器的时序问题，提供给存储器的时钟可能会需要与 UART 自身时钟错开半个周期，以保证在数据有效时（UART 时钟周期中间点）进行存储器的数据读写。UART 调试器对数据存储器的操作也类似，这里不再详述。

调试模块的串口收发功能：调试模块的串口收发功能通过两个先入先出的队列，tx_buf 和 rx_buf，来缓存串口通信数据。对于接收队列，串口收发时仅负责向队尾添加数据，递增 rx_tail 尾指针。而对于发送队列，该部分代码负责在队列中有数据时取出 tx_head 进行数据发送。具体对接收数据的处理及控制发送数据是由 UART 调试器的状态机来负责实现的。由于 UART 调试器的时钟远高于串口的 115200Hz 时钟，所以在程序设计中没有考虑收发缓冲溢出的问题。

```
 // send and receive logic
always@(posedge clk)
begin
    if(rst)
    begin
        tx_head<=4'd0;rx_tail<=4'd0;
        tx_send<=1'b0; rx_gotdata<=1'b0;
    end
    else
    begin
        if(rx_ready)
        begin
            if(rx_gotdata==1'b0)
            begin
                rx_buf[rx_tail]<=rx_data;
                rx_tail<=rx_tail+4'd1;
                rx_gotdata<=1'b1;
            end
        end
        else
        begin
            rx_gotdata<=1'b0;
        end
        if(tx_head!=tx_tail)            //has data to send
        begin
            if(~tx_send&tx_ready) //tx free
            begin
                tx_data <= tx_buf[tx_head];
                tx_head<= tx_head+4'd1;
                tx_send<=1'b1;
            end
            else
```

```
                begin
                    if(~tx_ready) tx_send<=1'b0;
                end
            end
            else
            begin
                if(~tx_ready)
                begin
                    tx_send<=1'b0;
                end
            end
        end
    end
```

注意 Verilog 中的缓冲区不是 C 语言中的数组，建议不要在同一个时钟周期内对同一个缓冲区的不同位置进行多次读写操作，这样有可能对存储的读写时序要求太高，在综合时无法通过 RAM 实现，最终造成无法综合或出错。例如，当要通过串口输出多个字符时，不可在一个周期将多个字符同时写入缓存区，而是要分多个周期每次写入一个字符。由于这样实现比较复杂，建议使用实验板上的七段数码管来实现调试信息的输出，而非使用串口直接输出。

调试模块的状态机：UART 调试模块的具体调试功能通过状态机来实现。在初始状态下，调试器等待用户输入的单字符命令，例如，

i：表示加载指令存储器；

d：表示加载数据存储器；

r：表示重启 CPU 等等。

当串口输入数据能够匹配对应指令，调试器将转移到具体的调试状态。例如，在匹配“r”之后，调试器将输出 cpureset 置 1，同时进入重启延迟状态，即维持 reset 信号一段时间后，再清除 reset 信号，并返回初始状态。对于加载指令和数据存储器，在进入加载状态后，调试器将首先暂停 CPU 的执行，同时，等待串口传输的.hex 文件。前面介绍过存储初始化 hex 文件的格式，其中以’@’开头的行，后续为十六进制的地址，因此可以用状态机来解析 hex 文件中指示的加载地址，并将 UART 调试器的 instaddr 设置成对应的地址。hex 中数据部分每行表示存储器一个单元的数据，也是以十六进制表示，状态机可以将其转换后，通过控制存储器的写入数据端口和写使能端口，每收到一个数据就写入对应存储器，并将地址加一。当传输完毕后，用户可以输入“q”来结束加载，同时复位 CPU 并开始用新的指令或数据进行执行。

参考上述设计，可以实现其他调试功能：例如，通过 cpuhalt 信号来控制 CPU 只执行一个周期即停止等待，实现单步执行；通过将 CPU 的指令地址送给 UART 调试模块，实现设置断点功能；通过输入指定地址，实现查看指令和数据存储器中的内容；通过改造 CPU 和 UART 调试器的接口，可以查看 CPU 中寄存器内容。

4、Debug 工具

Verilator 仿真工具能帮助大家使用 C++语言对代码进行调试。

Verilator 能够将 verilog 代码编译成可执行的 C++类，在 C 程序中实例化这个类即可通过软件赋值来模拟数字电路的执行过程。

环境准备：在 ubuntu 下安装 Verilator。

$ sudo apt-get install verilator

$ git clone https://github.com/addrices/RISCV-TESTFRAMEWORK.git

各文件说明：

/src

1）、cpu_shell.v 文件

```
module cpu_tst(
    input clk, //时钟
    input reset, //复位高有效
    input [8*8:1] testcase, //输入要运行的 TestCase 的字符串
    output [31:0] dbg_pc, //dbg_pc 输出当前周期已经完成的指令的 PC 值
    output [31:0] dbgregs_0, //直接接到 RegFile 中去
    output [31:0] dbgregs_1,
    output [31:0] dbgregs_2,
    output [31:0] dbgregs_3,
    ……,
    output done, //当读取到全 0 的 Instr 的时候完成拉高，保持 PC 不变
    output wb //当前周期是否有指令完成
);
```

这个文件会变成最后的测试顶层文件，每当 reset 置高时，会将 testcase 表示的 8 位字符（8 个 char）读入，并读取对应的.hex 文件到 instructions Ram 中去。然后复位 reset，开始这个测试用例的执行测试。如果当前周期完成了某条指令(数据已经写回寄存器堆)，把 wb 拉高，并输出完成指令的 pc（dbg_pc，测试框架中默认的初始 pc 是 0），完成指令后的寄存器堆中的值。

2）、mycpu.v

```
module rv32is(
    input clock,
    input reset,
    output [31:0] imemaddr, //imem 的地址
    input [31:0] imemdataout, //imem 读取到的数据
    output imemclk, //imem 的时钟
    output [31:0] dmemaddr, //dmem 的地址
    input [31:0] dmemdataout, //dmem 读取到的数据
    output [31:0] dmemdatain, //需要写入 dmem 的数据
```

```
        output dmemrdclk, //dmem 读口时钟
        output dmemwrclk, //dmem 写口时钟
        output [2:0] dmemop, //3'b000:sb 3'b001:sh 3'b010://sw
        output dmemwe, //dmem 写有效
        output [31:0] dbg_pc, //当前完成的指令的 PC
        output done, //读取到 Instr 为 0 时认为程序结束
        output wb //当前周期是否有指令完成
        );
        //add your code here
    endmodule
```

在 mycpu 中添加缺省的设计代码。

/emu

3）、SingleRiscv*等文件都是由一个单周期 CPU 编译而来的。

main.cpp 中

```
//SingleRiscv运行一个周期
void next_cycle_r(std::shared_ptr<SingleRiscv> dut){
    dut->eval();
    dut->clk = 0;
    dut->eval();
    dut->clk = 1;
    dut->eval();
}
//mycpu运行一个周期
void next_cycle(std::shared_ptr<emu> mycpu){
    mycpu->eval();
    mycpu->clk = 0;
    mycpu->eval();
    mycpu->clk = 1;
    mycpu->eval();
}
```

SingleRiscv 和 emu 均为硬件转化为的 C++类，eval()函数是更新当前类对应的硬件的电路，通过上下拨动 clk，即可达到经过了一个时钟的效果。

cycle_check 函数用于检查当前指令的 pc 和 regfile 中的值是否和 SingleRiscv 一致。

```
for (int i = 0; i < 1000; i++) {
    next_cycle(mycpu);
    if(mycpu->wb == 1){
        next_cycle_r(dut);
        if(cycle_check(dut ,mycpu) == false){
            return 0;
            }
```

```
        }
    if(dut->done == 1 && mycpu->done == 1){
        printf("Pass Test\n");
        break;
        }
    }
```

初始化完成后，会将 mycpu 的时钟不断拨动，待到 mycpu 中完成一条指令，同样拨动一下 SingleCpu 的时钟（SingleCpu 是单周期，每个周期都会写回一下）。

testcase 中包含着许多测试用例，其 Pc 起始地址为 0，将其文件名写入 Makefile 第 12 行中 TEST，运行 make run-emu 即可调用测试。

提示：在理解框架后可以自行修改框架，添加自己的外设，编译文件等等。

当然也可以使用 RARS、Riples 或其他熟悉仿真测试工具来验证代码。

四、实验内容

本实验需要进行上板验收，通过连接键盘和显示器演示完整的计算机系统；支持单步执行模式，用按钮作时钟，在七段数码管上显示当前 PC。建议实现的功能包括：

1) Xterm命令：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。

   输入 hello，显示 Hello World!

   输入 clear，清屏。

   输入 fib n，显示斐波那契数列。

   输入 sort，通过键盘输入未排序数据，然后把排好序的数据显示在显示器上。

2) 实现简单的C语言库函数，移植其他课程实现程序或游戏代码，例如拔尖班PA中的AM代码（https://nju-projectn.github.io/ics-pa-gitbook/ics2021/2.3.html）或操作系统课程实验（选做）。

3) 使用AXI Lite总线来连接各类外设，将外设模块化，增加DDR、串口，SD卡等类型的外设（选做）。

4) 自我创新设计（选做）。

根据上述要求，实现一个计算机系统并进行演示。

1) 使用 Vivado 创建一个新工程 lab11。
2) 添加设计源码文件。
3) 添加测试文件。
4) 添加约束文件。
5) 根据实验要求，完成源码文件的设计。

6) 对工程进行仿真测试。

7) 仿真通过后，进行综合、实现并生成比特流文件。

8) 生成位流文件后，加载到实验开发板，进行调试验证，并记录验证过程。

五、思考题

1、编译运行流行的 benchmark，对 CPU 进行性能评估。

2、如何在系统中使用高速缓冲存储器。

3、如何通过使用 DDR，移植一个简单操作系统。