实验报告

郭家宝 2010011267 计 02

周越 2009011471 自 94

总体目标

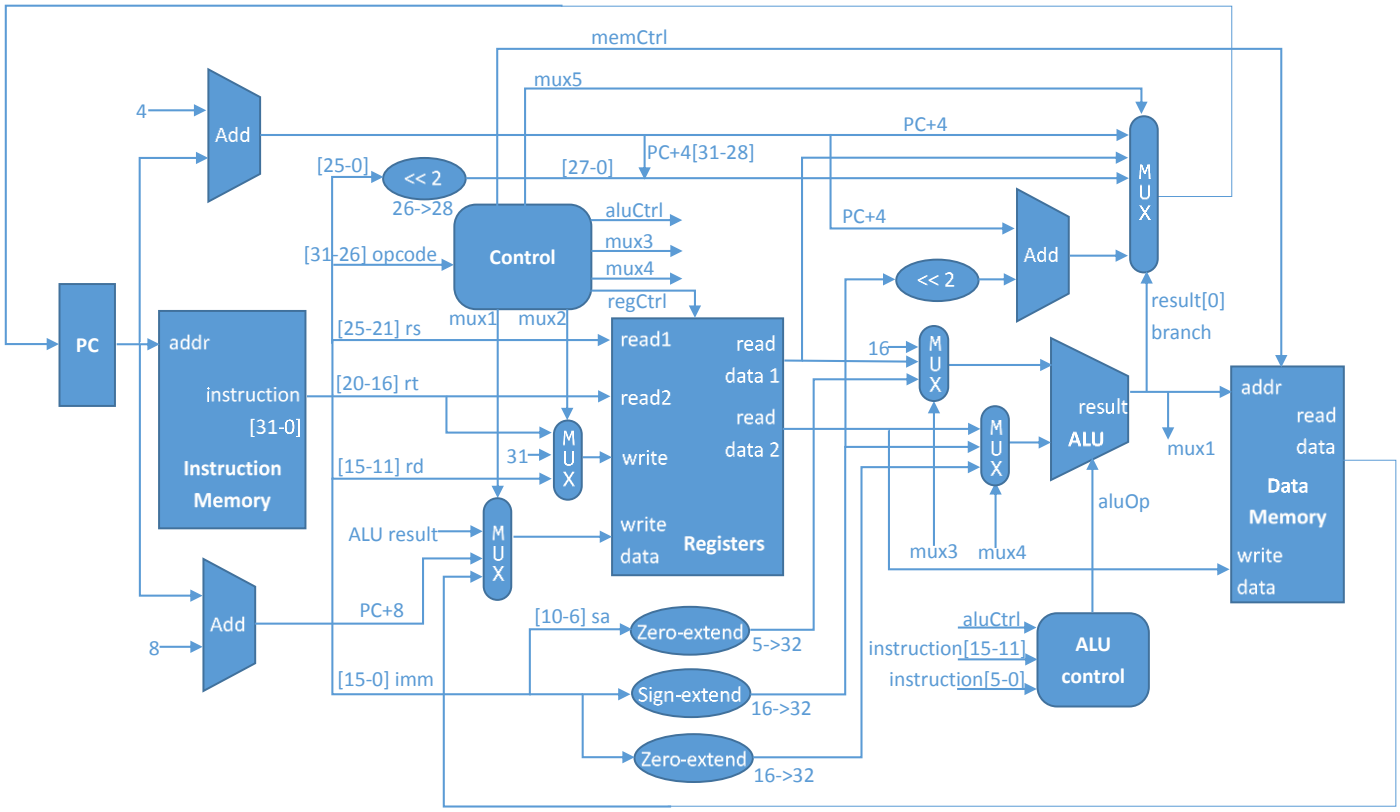
实现以 MIPS32（精简）指令集 CPU 为核心的一套相对完整的计算机系统。

CPU 为 32 位 MIPS32 指令集，省略一些不常用的指令；支持中断、MMU（TLB）以支持操作系统的运行。

基本目标：能跑自制的汇编程序；**进阶目标**：能运行操作系统。

模块设计

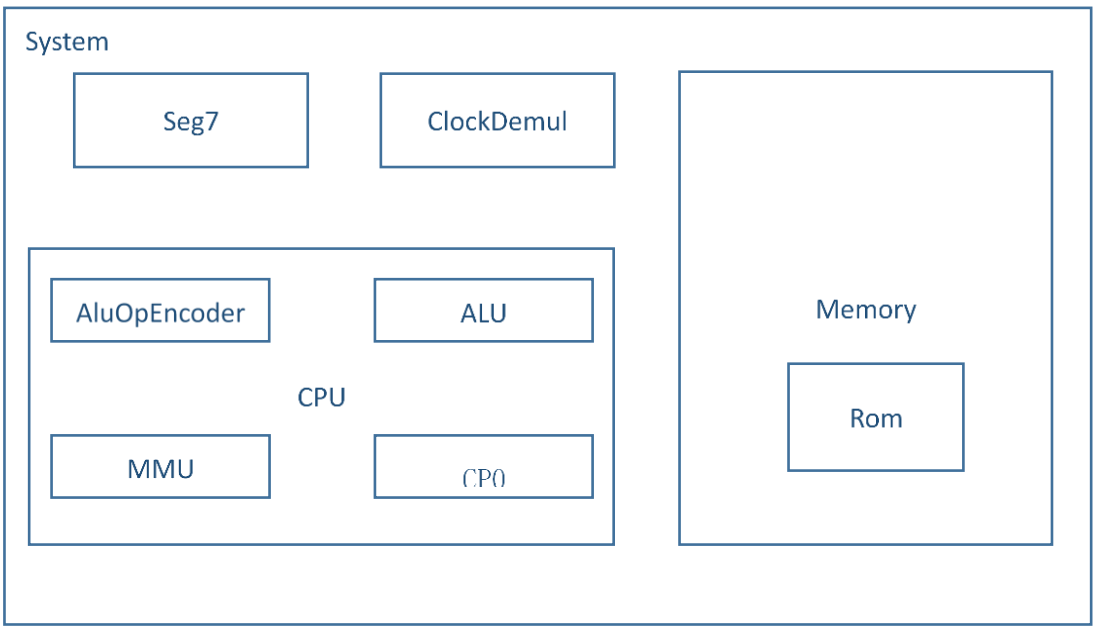
基本 CPU（不含扩展功能）的数据通路图如下：



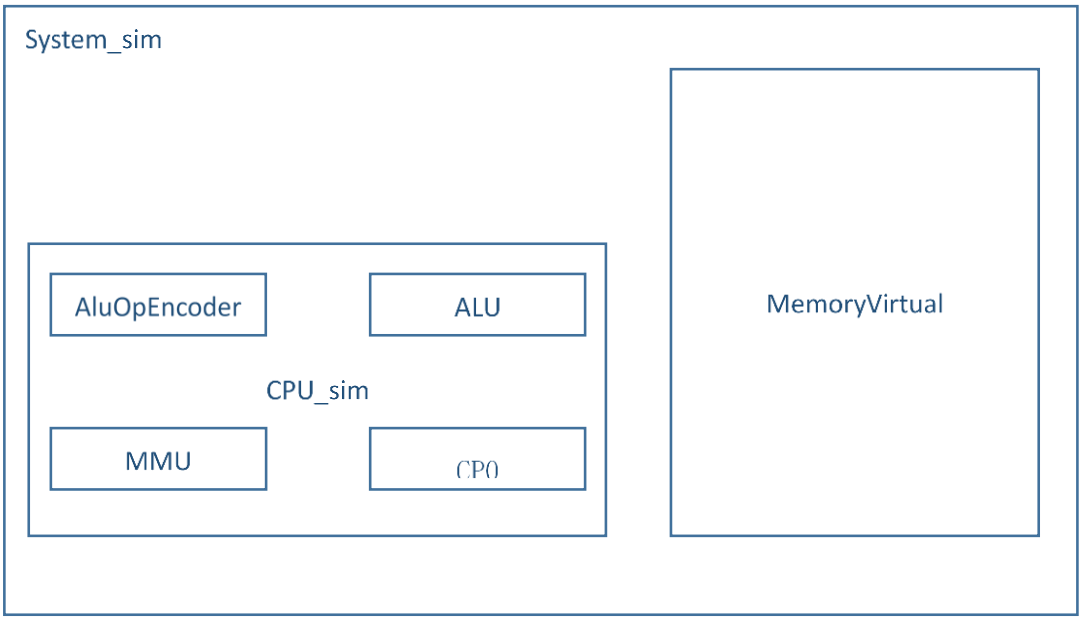
整个系统的模块如下：

- System 整个系统，顶层模块，用于配置和连接时钟、CPU、存储和其他外设。
- System_sim 功能同 System，但用于软件仿真。
- CPU 整个 CPU，包含了 CPU 的全部逻辑
- CPU_sim 与 CPU 模块共用代码，每次仿真时由脚本自动生成，包含一些仅用于仿真的代码
- Memory 存储设备的封装，包含 RAM、ROM、Flash 和串口等。
- MemoryVirtual 同 Memory，但用于软件仿真，可以从文件加载初始内存镜像，可以把写入串口的数据输出到标准输出。
- Rom 片上 ROM，包含 ROM 中的程序。
- RegisterFile 寄存器堆
- ClockDemul 分频器
- ALU 运算逻辑单元，用于各种运算。
- AluOpEncoder 用于把指令中的信息编码为 ALU 的操作。
- Seg7 数码管
- Common 各程序模块共用的“头文件”

用于**实际综合**的模块关系图：



用于**软件仿真**的模块关系图：



实现细节

Memory 部分（包括外设）

我们实现的存储器和外设有：

- 静态随机存储器（SRAM）
- 只读存储器（ROM）
- 串口（COM）
- 闪存（Flash）

物理地址划分

设备	物理地址范围
RAM	00000000-000FFFFF
ROM	1FC00000-1FC00FFF
Flash	1E000000-1EFFFFFF
COM	1FD003F8-1FD003FC

虚拟地址划分

名称	虚拟地址范围	权限	MMU	对应物理地址
kuseg	00000000-7FFFFFFF	用户	是	由 TLB 转换
kseg0	80000000-9FFFFFFF	内核	否	0x00000000-0x1FFFFFFF
kseg1	A0000000-BFFFFFFF	内核	否	0x00000000-0x1FFFFFFF
kseg2	C0000000-FFFFFFFF	内核	是	由 TLB 转换

内存控制器实现

内存控制器作为一个独立的单元被抽象出来，CPU 控制器可以直接通过控制信号访问。以下是接口描述：

信号名称	方向	作用
clk	输入	时钟信号
rst	输入	复位信号
en	输入	使能
rw	输入	读/写状态
length	输入	数据长度(8 位、16 位、32 位)
addr	输入	地址
data_in	输入	数据输入
data_out	输出	数据输出
completed	输出	就绪信号
int_com	输出	串口中断信号

内存控制器是**异步**的，其时钟信号可以与 CPU 时钟不同步，CPU 通过 completed 信号来判断访存完成。当使能端 en 关闭时，completed 信号自动清除。rw 用于控制内存控制器位读状态还是写状态。length 的可选参数位 Lword、Lhalf 和 Lbyte，分别是访问 32 位数据、16 位数据和 8 位数据。当串口数据到达时，int_com 信号会被置 1，由 CPU 控制器来处理中断。

SRAM 访问与数据对齐

SRAM 由两块 512KB 的内存芯片组成，总空间为共 1MB。由于 SRAM 的数据总线只有 16 位，我们将 32 位的数据按照高低位拆分为两部分分别存储到不同的内存芯片。

访问 32 位数据(lw, sw 指令)时要求地址按照 32 位对齐，即地址的最低 2 位必须是 0。对于一个 32 位的数据，其布局为：

RAM2		RAM1	
高 16 位		低 16 位	
高 8 位	低 8 位	高 8 位	低 8 位

若访问 16 位数据(lh, sh 指令)，则要求 16 位对齐，其最低位必须是 0，倒数第二位用于判断从 RAM1 还是从 RAM2 读取数据，高 16 位全部填充 0。

访问 8 位数据(lb, sb 指令)不需要对齐，倒数第二位用于判断从 RAM1 还是从 RAM2 读取数据，最后一位用于判断输出获取到数据的高 8 位还是低 8 位，高 24 位全部填充 0。

8 位写入的实现

由于内存芯片只能按 16 位访问，不能只读写 8 位数据，我们在遇到写 8 位数据的请求(sb 指令)前会先将对应的 16 位数据全部读出，然后在将其不需要修改的 8 位补全以后写回内存芯片，其所需的时钟周期是其他访存指令的两倍。

串口读写

串口的物理地址是 1FD003F8，每次只能读写 8 位数据，读写前需要先由软件判断状态位 1FD003FC。状态位的最低位表示串口是否可写，倒数第二位表示是否可读。从串口读取数据后，状态位倒数第二位和中断信号会自动被清除。

串口模块的主要部分位于 CPLD 中，波特率为 115200，数据校验位奇校验。在 FPGA 端，对串口的控制通过 data_ready、tbre、tsre、rdn、wrn 进行。当 data_ready 为 1 时，串口数据就绪，可以读出。当 tbre 和 tsre 都为 1 时，表示可以向串口

写入。将 rdn 置 0 且数据线写高阻，即可从数据线读出串口数据，将数据写入数据线且将 wrn 置 0，即可向串口发出数据。

串口与 RAM1 共用数据线，在读写串口时必须将 RAM1 关闭。

片上 ROM 的生成

片上 ROM 的功能是存储引导程序，因此只能按照 32 位访问。我们实现了 genrom.py 脚本，用于将汇编器产生的二进制文件直接转换为 VHDL 代码，通过 VHDL 的逻辑综合器生成片上 ROM。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.Common.all;
entity Program is
  port (
    addr: in Int10;
    data: out Int32
  );
end Program;
architecture Behavioral of Program is
  constant NUM_ROM_CELLS: integer := 1024;
  type ProgramType is array(0 to NUM_ROM_CELLS - 1) of Int32;
  signal rom: ProgramType;
begin
  data <= rom(to_integer(unsigned(addr)));
  rom(0) <= x"3c0dbfd0";
  rom(1) <= x"35ad03f8";
  rom(2) <= x"3c04be00";
  rom(3) <= x"241d00ff";
```

```
  rom(4) <= x"241c0040";
  rom(5) <= x"241b0070";
  rom(6) <= x"241a0020";
  rom(7) <= x"241900d0";
  rom(8) <= x"24010001";
  rom(9) <= x"24020002";
  ……
end architecture;
```

Flash 读写

Flash 的读写方式与 SRAM 相同，擦除由软件来控制。由于 Flash 数据线只有 16 位，因而读出的数据高 16 位总是 0，写入的数据高 16 位会被忽略。

虚拟存储器（MemoryVirtual）

为了方便使用 ModelSim 调试，我们还实现了虚拟存储器，完全仿真真正存储器的行为，用于软件调试。

CP0 的实现

CP0 包含的一个系统所必需的一些功能，包括系统状态的设置与存储、TLB、异常与中断等等功能。MIPS 为了保持 CPU 基本 ISA 的简洁，单独将这些功能放入 CP0 中。

简单的说，CP0 包含了一系列有特殊意义的寄存器。我们实现了的寄存器包括：

- 最基本的状态寄存器 SR
- TLB 相关的 Index、EntryHi、EntryLo0、EntryLo1 寄存器
- 时钟中断相关的 Count、Compare 寄存器
- 异常、中断相关的 Cause、EPC、EBase、BadVAddr 寄存器

CPU 通过对这些寄存器进行读和写，就能完成所有这些功能。所以只需要 MTC0（Move To Cp0）和 MFC0（Move From Cp0）两条指令在通用寄存器和 CP0 寄存

器直接进行数据交换即可，大大简化了 ISA。

异常与中断

硬件的中断算是异常的一种，其它异常包括 TLB 异常、访存异常、Syscall 等等。

对于软件异常，只要全局的异常开关（EXL）打开，同时相应指令执行的过程中满足特定的条件，即可触发异常。

对于硬件中断异常，要求全局异常开关（EXL）打开，全局中断开关（IE）打开，每个独立中断的 Mask（IM）开启，同时对应的外部连线有中断信号（IP），才能触发。

触发了异常之后，原因写入 Cause 寄存器，关闭 EXL，当前 PC 保存到 EPC，然后 PC 置为异常入口地址开始执行异常处理程序。

异常返回指令 eret 执行时，EXL 打开，PC 恢复为 EPC 即可。

EBase 寄存器保存的地址值（处理后）加上 0x180 即为异常入口地址。

MMU/TLB

MMU 即内存管理单元，操作系统和 MMU 配合实现虚拟地址到物理地址的转换。

MIPS 中有两段地址直接映射到物理地址，而另外两段地址使用页表记录虚页号（VPN）到物理页号（PFN）的映射，通常操作系统会把页表存储在内存（甚至硬盘）上，而 MMU 中使用 TLB 作为页表项的缓存。

我们的 TLB 设计为 16 项的全相联（Fully Associative）的 cache。

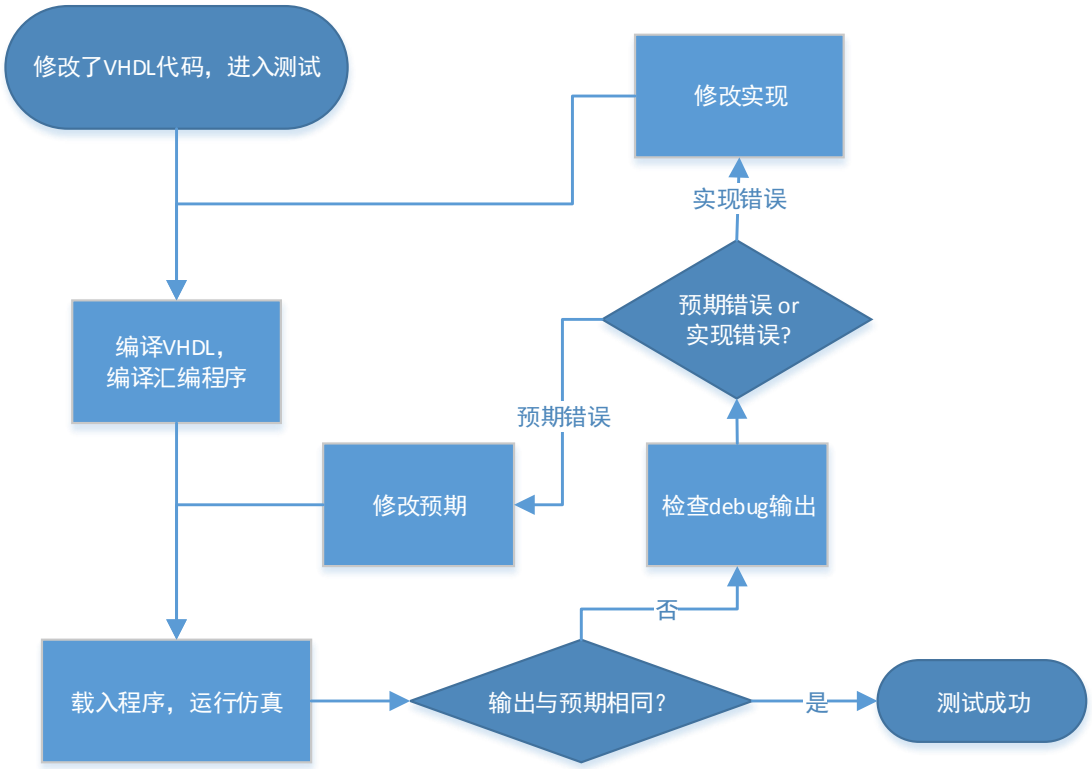
其它

MIPS 的 ISA 中有延时槽（Delay Slot），我们为了让 CPU 与 MIPS 完全兼容，PC 的增加考虑到延时槽，但是为了简化设计，不会执行延时槽中的指令，所以在手写汇编程序时应该手动在延时槽处填入 NOP；若使用编译器时， 则记得加上保留 NOP 的命令行开关。

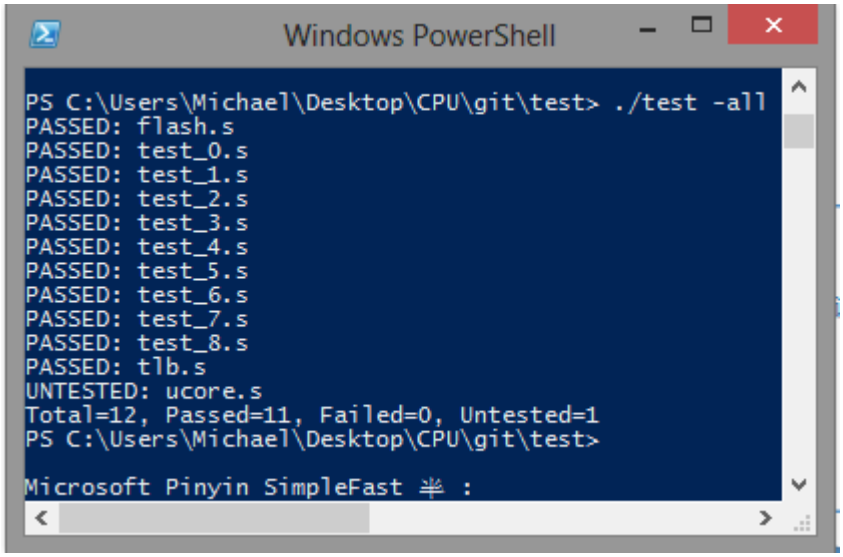
仿真验证

我们利用 ModelSim 的仿真功能，搭建了一整套方便的仿真测试工具，能够在每次修改 VHDL 代码后，现在软件上快速验证设计，提前发现错误，能够大大节省调试时间。

测试流程：



运行全部测试：



我们在 VHDL 里面使用 std.textio 输入调试信息。以下列程序为例：

```
#Load a number and send it to COM
start:
  .text 0
  la $4, 0x1234
  la $5, 0xBFD003F8 # r5 := COM_ADDR
  sw $4, 0($5) # mem[r5] := r4
  break
```

注：我们在仿真过程中，使用“break”指令用于停机（即终止仿真运行）

此程序返汇编结果：

```
test_0.out:     file format elf32-tradlittlemips
Disassembly of section .text:
80000000 <_ftext> 24041234   li a0,4660
80000004 <_ftext+0x4> 3c05bfd0  luia1,0xbfd0
80000008 <_ftext+0x8> 34a503f8  oria1,a1,0x3f8
8000000c <_ftext+0xc> aca40000  sw a0,0(a1)
80000010 <_ftext+0x10> 0000000d    break
```

使用我们的测试工作仿真的输出为：

```
0x34
```

详细的调试输出为：

```
booting


fetch instr @ 0x80000000 (10000000000000000000000000000000)
MMU[0x80000000] :  0x00000000
Mem[0x00000000] :  0x24041234 (00100100000001000001001000110100)
```

```
001001 | 00000 | 00100 | 0001001000110100
ADDIU    rs: 0   rt: 4   imm:0x1234
R[ 0] :  0x00000000 (00000000000000000000000000000000)
alu_a <= 0x00000000 (00000000000000000000000000000000)
alu_b <= 0x00001234 (00000000000000000001001000110100)
alu_r :  0x00001234 (00000000000000000001001000110100)
R[ 4] <= 0x00001234 (00000000000000000001001000110100)

fetch instr @ 0x80000004 (10000000000000000000000000000100)
MMU[0x80000004] :  0x00000004
Mem[0x00000004] :  0x3c05bfd0 (00111100000001011011111111010000)
001111 | 00000 | 00101 | 1011111111010000
LUI      rs: 0   rt: 5   imm:0xbfd0
R[ 5] <= 0xbfd00000 (10111111110100000000000000000000)

fetch instr @ 0x80000008 (10000000000000000000000000001000)
MMU[0x80000008] :  0x00000008
Mem[0x00000008] :  0x34a503f8 (00110100101001010000001111111000)
001101 | 00101 | 00101 | 0000001111111000
ORI      rs: 5   rt: 5   imm:0x03f8
R[ 5] :  0xbfd00000 (10111111110100000000000000000000)
alu_a <= 0xbfd00000 (10111111110100000000000000000000)
alu_b <= 0x000003f8 (00000000000000000000001111111000)
alu_r :  0xbfd003f8 (10111111110100000000001111111000)
R[ 5] <= 0xbfd003f8 (10111111110100000000001111111000)

fetch instr @ 0x8000000c (10000000000000000000000000001100)
MMU[0x8000000c] :  0x0000000c
```

```
Mem[0x0000000c] :  0xaca40000 (10101100101001000000000000000000)
101011 | 00101 | 00100 | 0000000000000000
SW      rs: 5   rt: 4   imm:0x0000
R[ 5] :  0xbfd003f8 (10111111110100000000001111111000)
R[ 4] :  0x00001234 (00000000000000000001001000110100)
alu_a <= 0xbfd003f8 (10111111110100000000001111111000)
alu_b <= 0x00000000 (00000000000000000000000000000000)
alu_r :  0xbfd003f8 (10111111110100000000001111111000)
MMU[0xbfd003f8] :  0x1fd003f8
Mem[0x1fd003f8] <= 0x34 (00110100)

fetch instr @ 0x80000010 (10000000000000000000000000010000)
MMU[0x80000010] :  0x00000010
Mem[0x00000010] :  0x0000000d (00000000000000000000000000001101)
000000 | 00000 | 00000 | 00000 | 00000 | 001101
SPECIAL  rs: 0   rt: 0   rd: 0   sa: 0   BREAK
halt
```

实验成果

50 条指令，7 种异常（含硬件中断异常，实现了时钟、串口两种硬件中断），MMU（TLB），特权极。

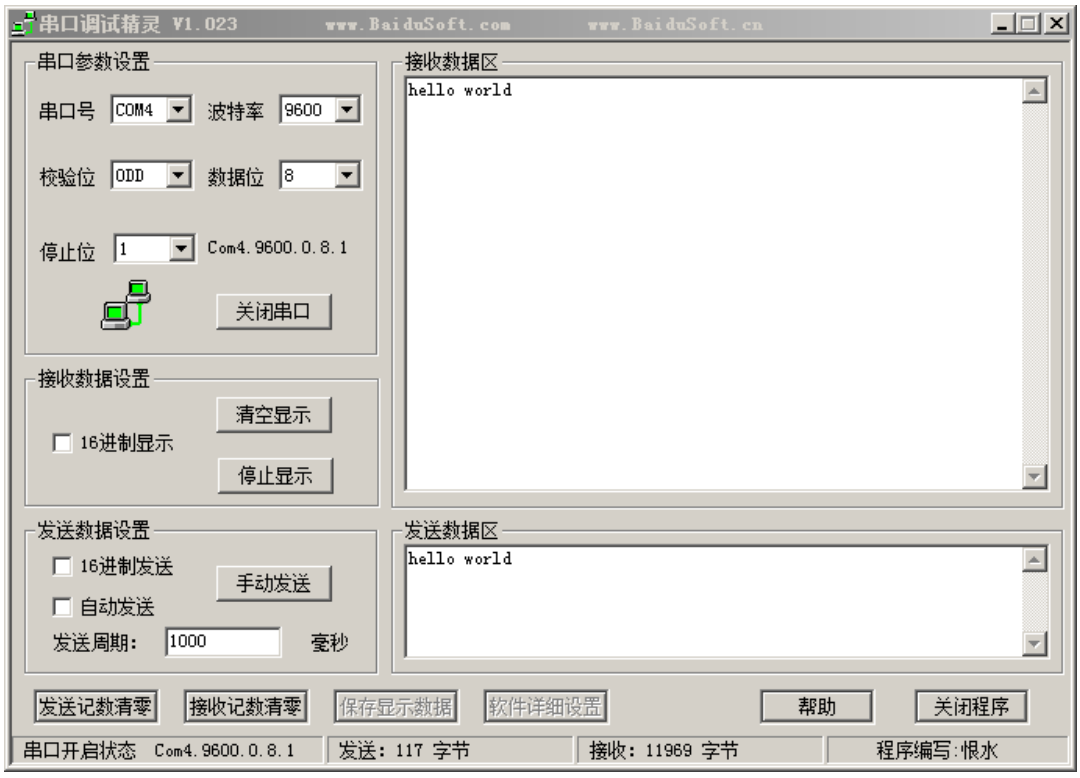
CPU 稳定时钟 25MHz，内存与 CPU 可以使用异步时钟。

实现的设备包括：片上 ROM，SRAM，串口，FLASH，LED 和数码管。

运行情况：汇编程序全部正确运行，包括使用了中断和 TLB 的程序。操作系统运行至分配 page 后的 assert 出错。

实验截图

允许自己的程序测试串口中断：



没有实现 TLB 之前，运行操作系统截图：（kernel panic）

心得和体会

- 使用软件仿真进行验证能大大节省时间

  在软件仿真阶段发现的错误，可以马上修改，省去综合布线和在硬件上调试的时间。仿真的时候还可以输出丰富的调试信息，很快能找到错误所在。实际上，本组两个人中，一人写 CPU 逻辑，一人在硬件上调 Memory。通过了仿真验证的 CPU，在修正了一个由综合器引起的时序问题之后，即可与 Memory 和其他外设很好的配合起来，以 25MHz 的频率稳定的跑在硬件上。后来在调整了 Flash 的读取时序后，同样的 CPU 代码，能以 50MHz 的频率稳定运行。可见软件仿真，让 CPU 的执行逻辑几乎不需要在硬件上调试。

- 使用版本管理软件能够提高团队协作效率

  我们使用 git 进行版本管理。不需要像一些组一样用很多很多不同名字的文件夹来保存各个阶段、各个版本的源代码。我们使用 git 不但能够记录源文件的修改历史，还可以任意回滚到任意版本进行调试，并可以把修改再合并到主分枝。使用 git 的另一个意义在于我们可以两个人分开在各自宿舍同时工作（两人宿舍不在一起），我们可以修改相同的文件，然后由 git 检测是否有冲突。大大提高的两人的协作效率。