计算机组成原理综合实验实验报告

清华大学
计 14 贾开 2011011275
计 13 周昕宇 2011011252
计 14 李铁峥 2011011268

Contents

1 导论 **2**

2 CPU 核心 **2**
2.1 整体设计 . . . 2
2.2 冲突处理 . . . 3
2.3 异常处理 . . . 3
2.4 内存管理 . . . 4
2.5 实现及开发技巧 . . . 4

3 外设 **5**
3.1 串口与通信 . . . 5
3.2 VGA . . . 6
3.3 ps/2 键盘 . . . 6

4 ucore **6**
4.1 编译 . . . 6
4.2 使用 qemu 模拟 . . . 6
4.3 抓取远程文件 . . . 6
4.4 VGA 输出 . . . 7

5 decaf 实验方案 **7**
5.1 汇编指令生成 . . . 7
5.2 库函数调用及 calling convention . . . 7
5.3 程序入口及退出 . . . 7

6 实验成果 **7**
6.1 展示效果 . . . 7
6.2 开发板测试套件 . . . 8

7 未解之谜 **9**

8 参考文献 **9**

1 导论

实验目标如下：

1. 使用老师提供的开发板，在 FPGA 上编程实现一个基于标准 32 位 MIPS 指令集的子集的流水 CPU，支持异常、中断、TLB 等。
2. 在该 CPU 上运行 ucore 操作系统，进入用户态及 shell 环境，正常执行 shell 命令。
3. 修改 ucore，实现简单的远程文件执行功能，即通过串口从 PC 上获取 ELF 文件，并在本地执行。
4. 修改编译原理课程中的 decaf 编译器，并结合 GNU Binutils，编译出可在 ucore 上执行的 ELF 文件。
5. 可选实现对 VGA、ps/2 keyboard 等其它外设的支持。

实验硬件环境为老师提供的一块开发板，主要核心部件包含 Xilinx Spartan6 xc6slx100 FPGA，8MB 32-bit 字长的 RAM，8MB 16-bit 字长的 flash，另有一块与 FPGA 相连的 CPLD 用于一些外设 I/O 操作。

项目使用 git 管理，并托管在https://git.net9.org/armcpu-devteam/armcpu上。目录结构如图 1所示。注意，项目中很多地方使用了软链接，并且有大量的脚本，因此只能在 linux 等类 unix 系统下使用。

```
/
├── archive ..... 经测试可用的 FPGA, CPLD 及 ucore 二进制映象
├── armcpu
│   ├── ipcore_dir ..... xilinx ipcore 相关，包括乘法器和 VGA 缓存
│   ├── prog ..... 写入 ROM 的引导程序的汇编代码
│   ├── simulate ..... 用于逻辑仿真的代码及期望输出
│   ├── src ..... CPU 核心部分源代码
│   │   └── gencode ..... 用于代码生成的脚本
│   └── synthesis ..... 用于逻辑综合的代码及 ucf 文件
├── decaf ..... 修改过的 decaf 编译器
├── lib ..... 一些通用的 ucf 及 verilog 文件，提供常见设备的驱动
├── misc_fpga_proj ..... 开发初期进行实验及硬件测试的 xilinx 项目
│   └── cpld ..... 需要烧入 CPLD 的程序，用于转发串口及处理 ps/2 键盘扫描码
├── report ..... 实验报告的源代码
├── screenshots
├── slides ..... 展示时所用的幻灯片的相关资源
├── ucore ..... 修改过的 ucore 源代码
│   └── ours ..... 运行于 ucore 上的独立程序
└── utils ..... 各种工具脚本
```

图 1: 项目目录结构

2 CPU 核心

2.1 整体设计

CPU 的整体设计如图 2所示。其核心部分有 instruction fetch(IF), instruction decoding(ID), execution(EX), memory access(MEM)四个 stage，并有独立的 multiplication(MULT), forwarding(FWD), memory management unit(MMU) 三个额外的辅助单元。

CPU 的总体设计参考了[1]和[2] 中的相关章节。其基本思路是在不同的 stage 间插入锁存器，其中有后续 stage 所需的全部信息，stage 内部是组合逻辑，这样每经过一个时钟周期，所以指令便会向后“移动”一个 stage，从而增加 throughput 实现流水 CPU。

寄存器堆(register file)直接放在了 ID 中，这样便于在指令译码时访问寄存器得到相应的寄存器值。对 CP0 和乘法器的直接读写都放在了 MEM 中进行，这样可以在每个周期中向访问内存一样对其进行访问，使得整体设计更为统一。另外，将 CP0 放在 MEM 中也有利于对异常和中断的处理。

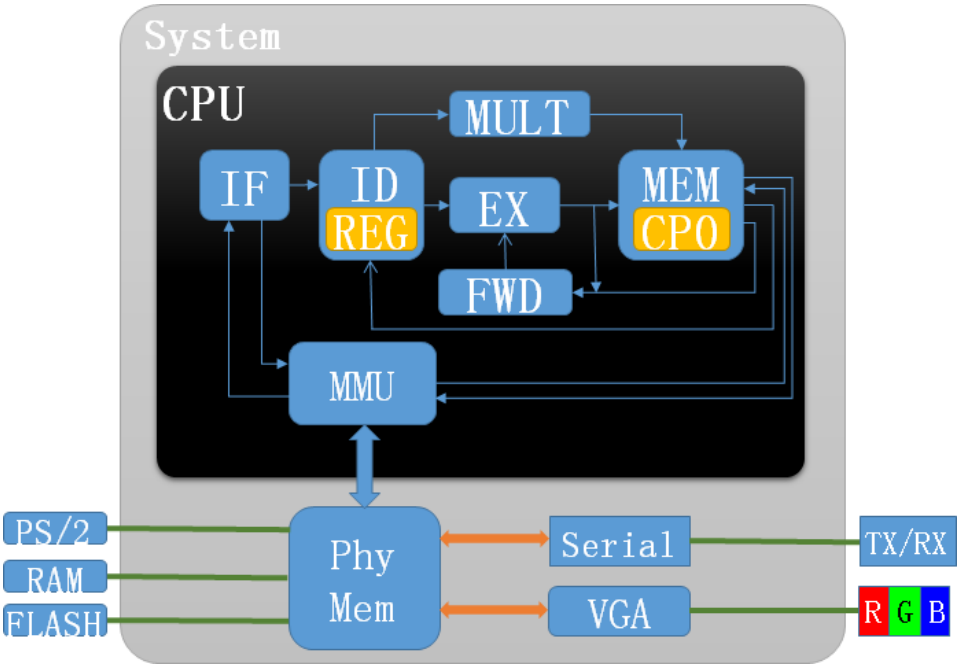

图 2: CPU 总体构架

2.2 冲突处理

流水线主要有结构冲突、数据冲突、控制冲突三种冲突类型。

由于指令和数据共用内存，而且没有实现 cache，结构冲突是不可避免的，目前我们采取了最简单的处理方式，即访存时就设置 stall 标志位，暂停除 MEM 外所有 stage 的活动。

对于数据冲突，通过精心设计的 forwarding 单元已可完全解决，无论前后指令间的数据依赖如何(甚至包括 lw+jr 等组合)，均不需要暂停处理器。其具体实现需要所有 stage 的共同协作和全局的设计考虑，不便于文字描述，详见代码；需要注意的是，要充分利用时钟的正负边沿。

对于控制冲突，通过 MIPS 中的延迟槽即可解决，也无需暂停处理器，但编译器可能需要插入不必要的 NOP。

2.3 异常处理

MIPS 要求精确异常处理(precise exception handling)，即要求流水 CPU 处理异常时对外所表现出的行为与单周期、多周期 CPU 一样。为此，在相邻 stage 间加入`exc_code`, `exc_epc`, `exc_badvaddr`这三个锁存器，同时把 CP0 放在最后一个 stage(即 MEM)中。在某个 stage 中检查到异常时，只将其到下一个 stage 的异常相关锁存器赋值，而最后在 MEM 中才真正执行异常操作，即设置一些标志变量使得下一次 IF 从

异常处理或中断返回的位置取代码。这样一来，异常发生前的所有指令都会被执行，而异常之后的指令都不会被执行，从而实现了精确异常处理。

对于中断，MEM 中的 CP0 判断中断条件是否成立，如果成立的话就设置相应标志变量，使得下个周期中 IF 产生一个虚拟的中断异常，待该异常传递到 MEM 后才进行实际的中断处理。

2.4 内存管理

虚拟内存方面，MMU 单元负责进行指令和数据访存的切换，TLB 的维护及查找，以及产生内存相关异常。MMU 接收指令地址和数据地址两个输入，如果没有数据操作，则表现得像组合逻辑，在半周期内向 IF 返回指令地址对应的内容或者访问其造成的异常；否则进行数据读写，并在操作过程中设置输出的 busy 标志位，待 busy 重新被置零后 MEM 就可以处理异常及数据。另外也可以向 MMU 发送一个 TLB 写入指令，使得其写入一个 TLB 条目。

我们将 CPU 的边界定义在 MMU。也就是说，对外而言，CPU 在每个周期内只会做以下几件事之一：

- CPU 给出物理内存地址，要求在半个周期内得到相应数据。
- CPU 给出物理内存地址及数据，启动物理内存写操作。
- CPU 等待物理内存写操作的完成。

对于所有外设，均通过物理内存映射进行 I/O。具体映射方案如图 3所示。 如果访问到了未映射的物

物理地址	对应设备	附加说明
[0x00000000, 0x007FFFFF]	RAM	共 8MB
[0x10000000, 0x10000FFF]	引导 ROM	共 4KB
[0x1E000000, 0x1EFFFFFF]	flash	地址空间共 16MB，但对于每个 32 位的字，只有低 16 位可用
[0x1A000000, 0x1A096000]	VGA	详见3.2节
0x1FD003F8	串口数据	可读写，读入则清除中断
0x1FD003FC	串口状态	第 0 位为是否可写，第 1 位为是否可读
0x1FD00400	数码管	在 2 位数码管上显示所写入数据的最低有效字节
0x0F000000	键盘码	读入则清除中断，详见3.3节

图 3: 物理内存映射

理地址，对于读操作始终返回 0，对所有写操作忽略。

2.5 实现及开发技巧

- **开发语言**
  使用 verilog 而非 VHDL 作为开发语言，相比而言前者语法更为紧凑简洁，更具表达力，最后 CPU 的核心代码不过一千多行，而且也更贴近 C 的习惯，极大地减少了开发和维护的工作量。

- **元编程**
  由于 xilinx 相关工具不支持 SystemVerilog，而后者具有结构体等很有用的语言特性，因此部分 verilog 代码使用 python 脚本生成以模拟类似 C 的 struct 结构；这部分代码主要用于跨 stage 的锁存器。另外，使用 xilinx ipcore 生成 ROM 的流程较为复杂也不易维护，而且实际 ROM 中的数据很少，因此 ROM 也采取 python 脚本生成，直接作为一个 case 语句放在 verilog 源代码里。

- **逻辑仿真**
  在 CPU 源代码中，使用 verilog 的`$display`，`$monitor`等 system task 输出了大量调试信息，再加上对串口、RAM、flash 等设备的 HDL 模拟，并使用 bash 脚本作为入口点，最终实现了自动化的

仿真工具链，只需给出汇编代码，就可以自动编译并将二进制载入模拟的 RAM 执行，同时 dump 出所有信号的波形文件，极大地方便了开发和调试。图 4展示了仿真及波形显示的界面，其中波形显示使用了 gtkwave。

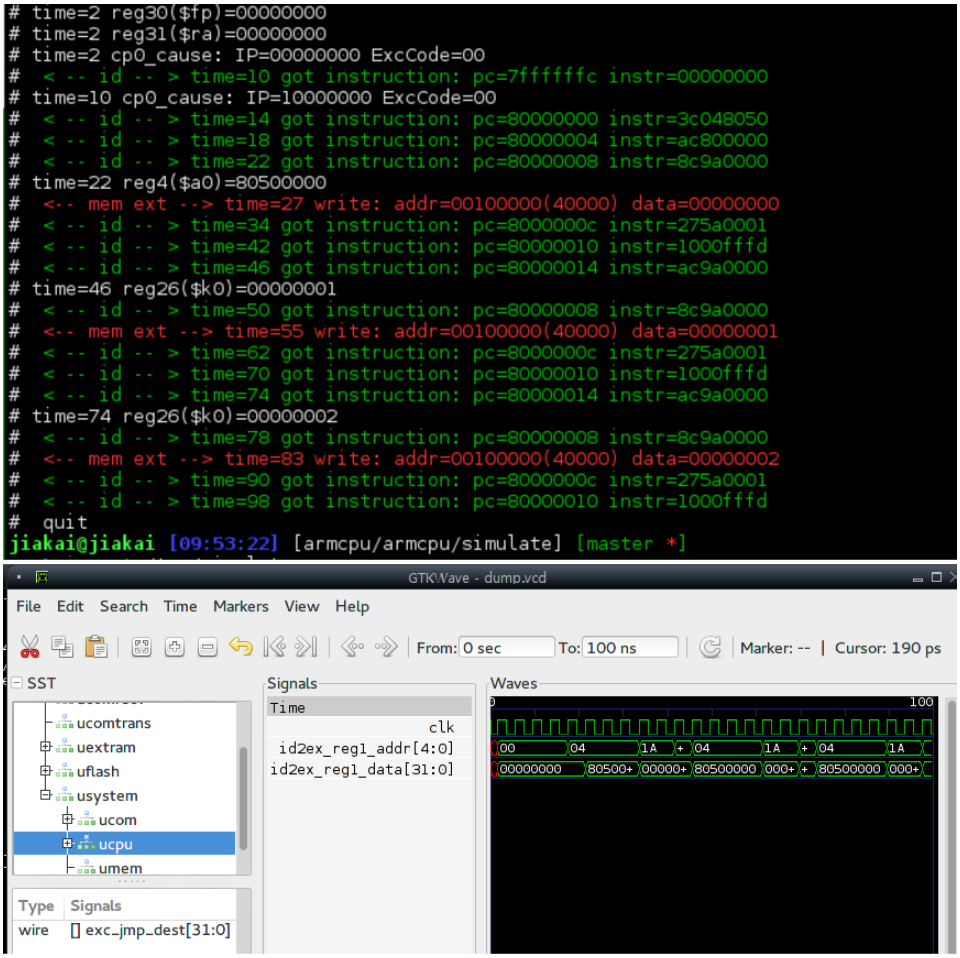

图 4: 逻辑仿真的相关界面

3 外设

3.1 串口与通信

开发板使用串口和 PC 机通信。老师提供了一套串口通信的 VHDL 代码，但经过测试发现在 115200 波特率下其连续接收 PC 端发送的数据时并容易丢包。后来采取了http://www.fpga4fun.com/SerialInterface.html上的代码，要稳定许多。其核心思想在于，接收时使用高频时钟在每个信号周期内多次采样，并通过滤波得到稳定的数据信号。对串口的解码和编码也被放在了 FPGA 中进行，这样 CPLD 仅负责对 TX/RX 端口的转发，有两个好处：(1)串口数据不需要占用内存数据线，也就无需处理冲突，简化了设计；(2)占用更少的 FPGA/CPLD 互连线，为后面 ps/2 键盘的实现提供了方便。

通信方面，我们设计了一套简单的通信协议，可以对 flash 和 RAM 进行读写。其核心流程是，由 PC 端发起操作，先传给开发板一个操作码，表示接下来的操作是读还是写，对象是 RAM 还是 flash，或者是擦除 flash；随后 PC 端向开发板传送操作的起、止地址，接下来就是数据，通信结束后开发板向 PC 返回接收到的所有字节的异或值作为校验码。经测试，实际传输速率约 10k/s。

基于这样的通信协议，在 PC 端用 python 实现了通信库，可以对大段数据进行分段传输和校验。在开发板上，分别用 verilog 及汇编在硬件、软件层面实现了接收端，方便数据的传输及后期的调试。其中。verilog 接收端主要用于早期开发；在 CPU 功能相对完善后，就将汇编的接收端写入了引导 ROM，可以通过拨码开关选择引导 ROM 的内容是接收端还是 ELF 加载器，方便对操作系统的调试。

3.2 VGA

为了充分利用 50MHz 的时钟，我们选择了 800 × 600@72Hz 的 VGA 模式，具体时序可参考http://tinyvga.com/vga-timing/800x600@72Hz。我们使用了 xilinx ipcore 在 FPGA 内生成了一块 RAM 作为 VGA 的缓存，每个像素使用一个 byte，为 256 色。考虑到 CPU 速度有限，VGA 的实际分辨率为 400×300，在显示时由硬件把一个像素扩展为四个像素。为了加速寻址，VGA 缓存的实际大小为 512×300，这样程序中就可以通过一次位移和一次加法计算出任意像素的偏移量。而为了把这段缓存映射到物理内存中，每个字节被补成了 4 字节的字，因此 VGA 缓存占用的物理地址空间是 $512 \times 300 \times 4 = 614400$。

3.3 ps/2 键盘

我们还加入了对 ps/2 键盘的简单支持。CPLD 负责对原始的信号进行解码及校验，将扫描码分为高 4 位/低 4 位两部分依次传送到 FPGA 中，这样做是为了减少对互连线的占用。FPGA 处理按下、释放事件以及维护 shift 状态，当键被按下时产生一个中断，并将传给操作系统的键盘码置为 shift 状态 +ascii 码低 7 位。当时为了简化实现，采取了这样一种简陋的方式。当然一个更好的方案是直接把原始扫描码传给操作系统，并且维护一个未读取的扫描码缓存，以防止操作系统的处理速度跟不上按键速度。

4 ucore

4.1 编译

如果使用较新的 gcc(4.7.3)编译，ucore 中的一个 bug 会暴露出来：编译`kern/trap/vector.S`时，编译器会自动在跳转语句后插入 NOP，导致所有的偏移量出错，解决方法是在文件头部加入`.set noreorder`。然而，即使修复了这个 bug，编译出的代码仍然不能在我们的 CPU 上正常运行。最终我们使用了与最初移植 ucore 到 mips 的学长所用的相同的版本的 gcc[1]，编译出的代码可正常运行。

另外，我们尝试了打开 gcc 的-O2 优化，发现系统运行速度明显提高，但系统引导时会触发`fs/sfs/sfs_fs.c`中的`panic("unused_blocks not equal!\n")`。我们未能找到其原因，后来注释掉这条 panic 发现系统可以正常工作，于是均使用-O2 优化了。

4.2 使用 qemu 模拟

修改 ucore 时，如果能在真实硬件上运行前先用模拟器模拟，将能极大提高开发效率。好在移植 ucore 的学长已经修改了 qemu 模拟器，可于https://github.com/chyh1990/qemu-thumips下载，并使用`./configure --target-list=mipsel-softmmu`编译；根据系统的不同，可能需要手动修复一些链接错误。

4.3 抓取远程文件

为了实现从 PC 上抓取文件并在本地执行，需要实现一个抓取远程文件的系统调用。为此，首先修改了控制台驱动，当输出一个字符时，会先往串口写入一个 MAGIC0 字符，再写入实际输出的字符；另外增加

[1] https://sourcery.mentor.com/GNUToolchain/release2189

了抓取文件的系统调用，它被调用时会先向串口写入 MAGIC1 字符，接下来进入抓取文件相关的通信协议。同时需要修改 PC 上的终端程序，让它遇到 MAGIC0 时就显示接下来的字符，遇到 MAGIC1 时就进入文件传输模式。另外由于 ucore 的 VFS 不支持新建文件，因此只能先在 ramdisk 中预留好一个足够大的空文件，把抓取到的文件内容写入这个其中。

4.4 VGA 输出

为了实现控制台内容同步输出到 VGA，首先需要字体文件。我们通过一个脚本渲染了某字体的可见 ascii 字符，并将得到的位图写入了一个 C 数组中。在 ucore 中加入了简单的字体渲染模块，需要处理换行、滚屏等，同时修改控制台驱动，每输出一个字符，在向串口输出的同时，也需要将该字符传递给字体渲染模块。

另外，为了能让用户程序进行 VGA 绘图后正常返回控制台，还提供了重绘全屏幕的系统调用。由于 ucore 没有提供物理内存映射的相关调用，同时也为了简化设计，CPU 中没有处理访问特权内存的异常，这样用户程序可以通过直接访问 VGA 缓存来操作屏幕内容。

5 decaf 实验方案

5.1 汇编指令生成

由于简化的 CPU 中并未实现 add、sub 指令，需要把 decaf 的 MIPS 后端里生成 add、sub 指令的部分改成 addu、subu，区别仅在于溢出时后者不会产生异常。

另外，CPU 中也未实现除法指令，不过由于所用测试程序中没有除法运算，因此也未进行相关修改；如果需要除法，可以用其它指令手动实现除法函数，并把除法翻译成函数调用。

当然，一个更好的方法应该是修改 ucore 系统，在异常处理中捕捉非法指令异常，并软件模拟未实现的指令。这样，对于编译器而言，目标机器就是一个标准的 MIPS32 CPU 了。

5.2 库函数调用及 calling convention

标准 MIPS32 使用 O32 ABI，函数调用的前四个参数通过$a0-$a3 四个寄存器传输；但 decaf 编译出的程序的参数全都在栈上传递。当然，无论什么 calling convention，只要能自恰，程序本身就应该能正常运行，所以需要解决的问题只有用户程序与 C 实现的库函数及操作系统交互的部分。

一种常规解决方案是修改 decaf 编译器，使得其遵循 O32 ABI，直接调用相应的函数。但这需要对后端进行较大的改动，也会造成与现有 decaf 编译出的二进制代码的不兼容。

在这里，如果把我们的 MIPS 系统看作一个要移植到的目标平台，并追求对 decaf 尽量少的改动，可以采用一种逆向的思路：在 decaf 和库函数之间增加一个适配器层，将 decaf 的调用约定翻译成 O32 ABI 再调用库函数。我们的实验中采取了这种方案，用汇编实现了这样的中间层，转发对库函数的调用。

5.3 程序入口及退出

我们直接使用了 ucore 里的 linker script(`user.ld`)以及用户静态函数库`libuser.a`，在该环境下系统会设置好一些全局变量，然后跳转到 main 执行。我们修改了 decaf 编译器，将其输出的`main`重命名为`decaf_main`，然后汇编实现了一个新的 main 函数。由于 decaf 的 main 是 void 类型，我们便默认其都执行成功返回 0，于是在`decaf_main`返回后直接调用 exit(0)。

6 实验成果

6.1 展示效果

除了基本实验要求外，还实现了两个用户态的程序：贪吃蛇和幻灯片放映。运行效果如图 5所示。

其中贪吃蛇还需要产生随机数，熵的来源是 CP0 中 count 寄存器的值。对于幻灯片放映，利用了 VGA 内存映射区是行优先存储的特性，实现了流畅的纵向换页的动画效果。具体而言幻灯片的原始数据以行优先连续存储在 flash 里，而静态显示或者播放动画时只需从某一行开始把整个屏幕的内容从 flash 拷贝到 VGA 的内存映射区域即可。另外，为了在 256 色的 VGA 输出下实现较好的显示效果，对图像预先使用了 Floyd-Steinberg dithering 算法进行量化，相关细节请自行查阅文献。

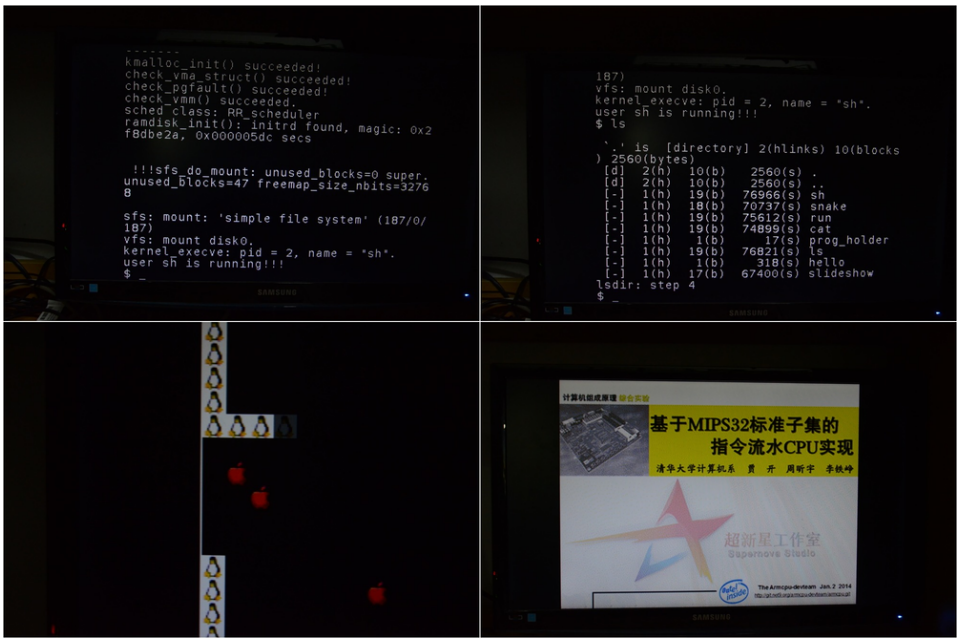

图 5: ucore 运行效果，左上、右上、左下、右下依次为刚进入 shell 时的画面、执行 ls 后的输出、贪吃蛇游戏界面、幻灯片播放界面

6.2 开发板测试套件

本实验的一个附加产品是开发板基本功能的测试套件。

1. 整体测试
   对于一个新的开发板，可以先尝试在 CPLD 中烧入`archive/cpld.jed`，再在 FPGA 上烧入`archive/armcpu.bit`。将拨码开关最右一个拨到 1，表示使用 12.5M 的时钟频率；其它全部置零。

   在 PC 上连接好串口线后，`cd utils/memtrans`，运行`./run_all.sh`，如果没有出错，说明串口、flash 和 RAM 均工作正常。随后运行

   ```
   ./controller.py flash write ../../archive/ucore-kernel-initrd
   ```

   烧入 ucore 的系统镜像。然后接上 VGA，插入 ps/2 键盘，运行`../terminal.py`进入终端。最后把最左的拨码开关拨到 1，表示从 flash 引导，这时应该能看到显示器和串口终端都打印出 ucore 的引导信息，PC 的键盘或 ps/2 键盘均可用于输入。此时开发板上的 LED 如图 6所示，其中数码管应显示 23，内存 LED 指示灯快速闪烁，数码管旁边的 LED 最右几个亮起。

2. LED 测试
   烧入`misc_fpga_proj/ledtest`里的工程，数码管旁的 LED 应从右向左扫描。

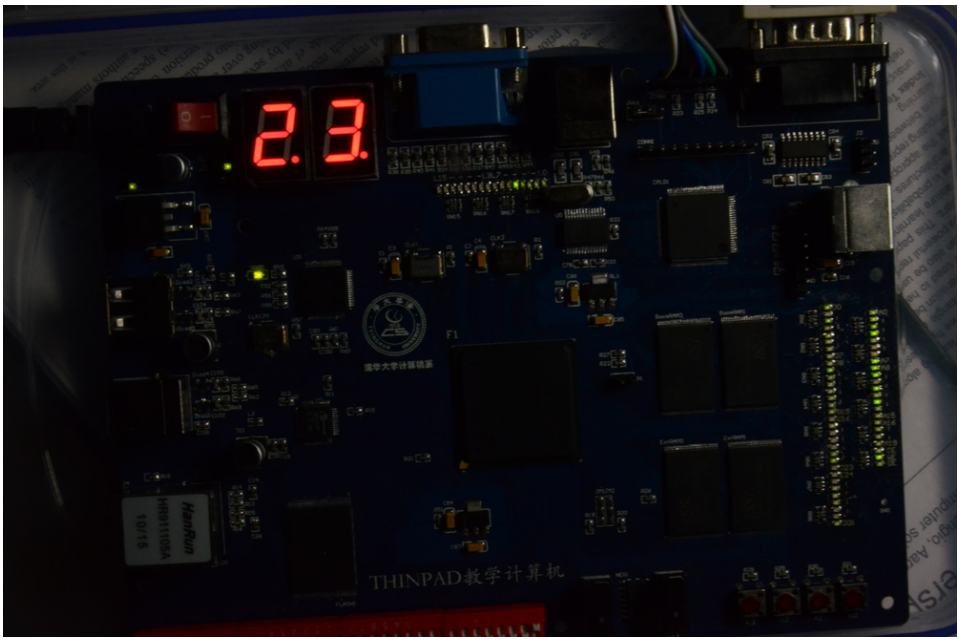

图 6: ucore 运行时的 LED 情况

7 未解之谜

在本项目的开发过程中，遇到了不计其数的问题，其中不少都调试出来并正确解决了，但也有些未解之谜，最终通过各种手段得以绕过。现列举如下：

1. 时序问题
   系统通过了逻辑仿真，在逻辑综合并烧入 FPGA 后，却发现时钟频率达到 50MB 后无法正常工作。后来用拨码开关调节 CPU 时钟，发现如果从 ROM 中读取程序，可以工作在 25MHz；从 RAM 读取的话只能达到 12.5MHz。分析 xilinx 逻辑综合产生的报告，发现部分数据通路的线路和逻辑延时都太大，而且在设计上很难优化。花了很多时间想解决这个问题，也尝试了几种不同的设计，但都未能成功，最终 CPU 运行在 12.5MHz。不过由于是流水 CPU，加上编译器优化，实际程序的运行速度还是不错的。

2. 诡异 bug
   最诡异的一个 bug，可以参考 git commit 8d5509e44b7ce5de286a7792c1a27b5bf95389c5。当时发现 VGA 的红、绿、蓝分量接反了，但将其修改正确后数码管却不能稳定工作了。但唯一的修改仅仅是交换了两条线的顺序。估计可能是 xilinx 全局布线优化时造成了时序的问题。

8 参考文献

[1] Randal E Bryant and David R O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice-Hall, 2008.

[2] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, 2008.