Unicore32子集模拟器结题报告

李春奇　　彭焯　　华连盛　　王衍

December 30, 2010

Contents

1 实验目的和要求

本项目实现了一个unicore32体系结构的模拟器，模拟器实现了CPU的五级流水、分立的i-cache和d-cache、内存管理、动态指令统计等模块，实现的指令集是unicore32体系结构指令集的一个子集，与同组的四个人所做的编译器实现的c语言子集相对应。最终实现的目标是能够使得实验室提供的unicore-gcc编译得到的ELF可执行文件能够正确在模拟器上运行，同组的编译器能够正确的在模拟器上运行。

2 实验小组成员及分工

本项目由于和编译器同时展开，所以同组共有四名同学。在编译器和模拟器的分工上，有着一定的偏重，其中李春奇、彭焯同学偏重于模拟器的实现，华连盛、王衍同学偏重于编译器的实现。但是在设计阶段和后期编译目标代码生成、编译优化、模拟器综合验证部分是由四名同学一起来完成的，即李春奇、彭焯同学同时也参与了编译器方面的工作，华连盛、王衍同学也同时参与了模拟器的验证工作。

在模拟器方面，彭焯同学负责cache模块、反汇编模块的实现，李春奇同学负责其他模块的实现，彭焯、华连盛、王衍同学负责模拟器的验证工作。

在编译器方面，李春奇、彭焯同学在语法分析、词法分析阶段与另外两名同学共同完成，之后的阶段李春奇同学和彭焯同学开始偏重于模拟器的实现。在目标代码生成和优化阶段由于模拟器已经基本完成，所以李春奇、彭焯同学又参与到编译器和编译器、模拟器联合调试的部分中。

3 实验环境

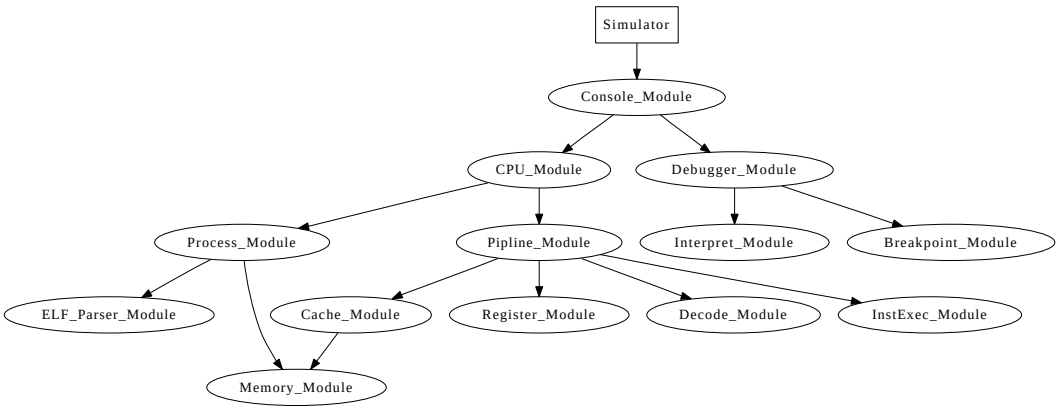
项目最终实现的模拟器能够在任何内核版本高于2.6.32的linux环境以及版本高于4.4.3的gcc环境下正确运行，需要外部libelf库，请在项目svn的lib文件夹下解压安装。

项目svn地址: https://minic.googlecode.com/svn/trunk/simulator

4 模块说明

4.1 模块整体视图

模块的视图如下:



通过模块视图可以看到，处于第零层的模块是Console模块，即所有模块最终被封装到了一个控制台上，这样Simulator解析命令行参数之后只需要通过调用控制台即可完成下面所有的调用工作。处于第一层的模块有CPU模块和Debugger模块，他们分别被Console Module调用，完成各自的功能。CPU模块是模拟CPU的模块，其子模块分为Process模块和Pipline模块，分别是模拟进程和流水线的模块。Process模块负责ELF文件的解析和对进程内存的初始化工作，所以其子模块是ELF Parser模块和Memory模块。Pipline模块则是模拟了CPU的五级流水，在五级流水中包括Cache模块、Register模块、Decode模块和指令执行模块。第一层的模块中另外一部分是Debugger模块，这一部分主要负责调试的工作，包含的子模块有Interpret模块（负责将机器码反汇编成汇编语言指令）和Breakpoint模块（设置和维护断点）。

以上就是项目的模块试图，每个模块对应项目中的一个文件，整体结构清晰明了，模块内部紧凑，模块之间相互独立，模块功能实现正交，便于实现和维护。下面就针对一些重要的模块（组）来进行说明，虽然在实现的过程中为了便于测试我们是采用自底向上的方式来实现，但是在说明的时候我们采用自顶向下的方式来说明，以便更清晰的说明层次关系。

项目在开题时的设计文档请参考“Simulator整体设计文档（草案）.pdf”，在设计过程中我们又对项目进行了一些细微的修改和增删，最终的项目设计以本文档为准。而在实现过程中整体逻辑并没有大的变化，所以《草案》作为开题时的设计文档还是有一定的参考价值的。

4.2 CPU模块

CPU模块主要包括以下两个子模块:

4.3 Process子模块

该模块负责ELF文件的载入以及进程对应内存模块的装载。

ELF文件的解析是从装载者的角度来处理，主要载入几个需要载入的segments，并且需要载入符号表以及每个表项在字符串表中对应的字符串，以便在后面反汇编时应用。此外还需要解析main函数的入口地址，这个可以简单的在符号表中查找即可。ELF文件解析模块应用到了libelf库解析ELF文件。

Process子模块再解析完ELF文件之后需要建立该进程对应的内存，这里采用的是把整个进程都调入内存的方式，即没有设计页式内存管理机制，对于cache来说内存是平坦的。每个进程第一个segment为该进程的栈，栈底地址为0x10000000，栈向低地址方向生长，栈大小可以在编译前配置。从编号为1的segment开始为ELF文件中需要载入的段以此存储。在Memory模块中需要特别说明的是实现了内存映射的机制，即对于任何进程（simulator中的进程）来说，是看不到物理机的实际内存地址的，对于任何虚拟地址的访问，都会通过一个中间层来处理，中间层首先检查内存的合法性，在内存合法的前提下进行映射转换，得到对应的数据。这样的设计保证了模拟器上运行的进程对内存地址访问的合法性检查，如果在模拟器运行阶段发现跑在上面的进程访问了不正确的内存，即报告模拟器的错误Invalid address，如果是产生了系统的segment fault，那么则是模拟器的实现有问题。这样可以有效的区分模拟器的错误和编译器的错误，在二者联合调试的时候这一点设计为我们节约和很多调试的时间。

进程的PCB数据结构如下:

```
1  typedef struct{
2      int status;// Process status
3      uint32_t entry;// Process entry addr
4      PROC_STACK* stack;// Process stack
5      PROC_MEM* mem;// Process memory, include stack
6      PROC_SYMTBL symtbl;// Process symbal table
7      BP_LIST* bp_list;// Breakpoint list
8      uint32_t list_cur_addr;// Breakpoint current address
9      int step_mode;//If process is step mode
10     INST_STATISTIC inst_statistic;// Instructions statistics
11 }PROCESS;
```

4.4 Pipline子模块

Pipline模块是模拟器的核心，该模块模拟了CPU的五级流水，并且在流水中涉及到了对cache和register的操作。在说明五级流水之前，首先说明一下Cache子模块和Register子模块。

4.4.1 Cache子模块

模拟器采用的cache是单片cache、I-cache和D-cache分离的Havard结构。对于CPU来说是不能够直接看到进程的内存的（在这里是被虚拟化和映

射过的内存），CPU只能直接对cache操作，而cache负责处理与内存间的数据一致性等问题。cache预设的未命中等待的CPU周期为8个周期，该数值可以编译前在头文件中定义。cache采用的回写策略是换出时写回策略。另外由于debugger模块有修改内存的需要，所以cache模块设计了同步函数，负责同步cache与写脏的内存的一致性。

Cache的数据结构如下:

```
typedef struct{
    int block_num;
    int sign_bits_num;
    PROC_MEM* mem;
    int valid[CACHE_SIZE/CACHE_BLOCK_SIZE];
    uint8_t data[CACHE_SIZE/CACHE_BLOCK_SIZE][CACHE_BLOCK_SIZE];
    uint32_t mark[CACHE_SIZE/CACHE_BLOCK_SIZE];
}CACHE;
```

4.4.2 Register子模块

Register子模块是所有模块中设计最简单的一个模块。寄存器堆包括32个通用寄存器，期中31号寄存器为PC，30号寄存器为SP，其余的均由编译器自行定义。另外还有32位CMSR寄存器，并且实现了对N、Z、C、V四个标志位的读写操作。

Register的数据结构如下:

```
typedef struct{
    int32_t r[32];
    int32_t flag;
}REGISTERS;
```

4.5 五级流水的设计

五级流水的设计是Pipline模块的核心，同样也是整个模拟器的核心，所以在此单独作为一个模块说明。这里对五级流水的设计进行整体设计上和功能上做详细的描述。

4.5.1 流水线设计总体描述

五级流水的第一级是IF级，即取指令。这里CPU直接面向i-cache取指，并返回指令及cache命中情况。第二级是指令译码级，在这一级对指令进行译码，获得不同指令的各个域位的值和分析指令的类型。第三级是指令执行级，负责ALU指令的执行、跳转指令目的地址以及访存指令的存储地址。第四级是访存级，负责访存指令对内存的读写。第五级是写回级，负责目的寄存器的写回。

流水线的设计中需要说明的是互锁和前递。流水线的前递是在第三级（ALU指令）和第四级（访存指令）进行的，在模拟器中只是简单的将得到的结果写入到目的寄存器中。流水线的互锁是在第三级执行LOAD类指令时，对标记变量进行标记，标记是哪个寄存器作为LOAD的目的寄存器。在流水线下一次执行到第三级时检查标记变量，如果标记变量值有效，则阻塞前两级流水线并且在第三级插入一个气泡。

流水线作为CPU的核心，面对的是cache和寄存器堆。流水线是不能直接操作内存的，这种很清晰的逻辑结构能够保证对内存读写的正确性和有效性，也能够保证了调试的准确性和系统性。

4.5.2 流水线需要解决的冒险

1. 结构冒险：通过指令cache和数据cache分离的Havard结构解决

2. 数据冒险：

   数据转发机制解决一部分冒险问题

   但是“加载-使用型数据冒险”需要加一个气泡

3. 控制冒险：

   分支控制导致的冒险：用分支预测的方法来解决：

   分支预测的策略：a.B指令一定预测跳转，b.其他条件跳转指令预测不跳转

4.5.3 各级流水的设计和接口规范

第一级：IF（Instruction Fetch）
输入：指令地址
输出：struct
实现：直接对cache进行访问，分为命中和未命中两种情况

第二级：ID（Instruction Decode）
输入：IF的输出
输出：struct
实现：对指令进行译码，获得对应寄存器的值，寄存器的编号，操作类型、操作数、移位立即数的值等等并保存在结构体中

第三级：Ex（Execuation）
输入：ID的输出
输出：struct

实现：根据指令的类型进行相应的运算，对于R型指令，写回在此处进行（即在此处进行数据转发）；对于分支指令，分支预测在此处进行处理。

第四级：Mem（Memory）
输入：Ex的输出
输出：struct
实现：lw指令的写回在此处实现，其他指令到此处均已执行完毕

第五级：WB（Write Back）
所有指令均已经执行完毕，此模块只做一些初始化工作。

4.5.4 各模块调用次序及相关结构的定义

各模块在逻辑上是并行的，在实现上是从后向前调用，遇到需要插入气泡的情况直接阻塞前边的各级流水
上文所述的结构：

```
typedef struct{
    uint32_t inst_addr;
    uint32_t inst_code;
    int inst_type;
    int opcodes;
    uint32_t Rn, Rd, Rs, Rm;
    int imm;
    int shift;
    int rotate;
    int cond;
    int high_offset, low_offset;
    int S, A, P, U, B, W, L, H;
    uint32_t cur_inst_PC;
    uint32_t addr;
}PIPLINE_DATA;
```

流水线的数据结构如下：

```
typedef struct{
    int block;//1 means pipline block, 0 mean the opposite
    int block_reg;
    PIPLINE_DATA* pipline_data[PIPLINE_LEVEL];
    char pipline_info[PIPLINE_LEVEL][200];
    PROC_STACK* stack;
    REGISTERS* regs;
    CACHE *i_cache, *d_cache;
    PROCESS* proc;
    int drain_pipline;
    int pc_src;
```

```
12      int ex_begin;
13  }PIPLINE;
```

4.6 内存模块

内存模块是模拟器设计的另外一个重点。在这个模块的设计中，实现了虚拟内存到物理内存的映射、内存访问有效性检查、内存越界检查（段错误）等操作系统完成的一些任务。由于模拟器主要模拟的是CPU的行为，所以在这里并不模拟页式内存管理，而是把整个进程内存作为一个页整体全部载入到内存中来统一操作。内存面对的上一级是I-cache和D-cache，CPU经由cache完成对内存的访问。

4.6.1 虚拟内存到物理内存映射机制

虚拟内存到物理内存的映射是实现内存访问有效性检查和内存越界检查的机制基础，也是我们模拟器设计的亮点之一。我们将物理内存分段映射到物理内存中，通过记录每个段的虚拟偏移量和段大小来判断越界和检查有效性。在载入器的角度，ELF文件有几个在运行时需要被载入的Segment，每个Segment作为一个段被载入到内存中并做映射。ELF文件中的段载入编号从1开始，这是因为每个进程的进程栈统一作为第0号段初始化到内存中。栈和Segment统一管理统一编址并且统一做有效性检查，保证了进程的执行完全是在一个模拟器搭建的软件平台上运行。有效避免了由于所运行的程序错误导致模拟器崩溃的情况。这样的保证能够保证模拟器的有效性和安全性，但是代价是内存读写的代价变大。即每次内存读写都要先进行一次有效性检查再通过内存复制将内存中的内容复制到目标地址，这种调用系统底层的操作显然效率要低于直接对内存的操作。但是在这里，如果访问的内存是一个块（例如cache未命中时在内存中读入一个块），按块的内存复制的效率则会更高。由上文说明可知，CPU只能间接通过cache访问内存，所以内存复制的代价在这里就被权衡掉了。

4.6.2 栈作为段统一管理机制

上文中介绍栈作为段在内存中统一管理，这一机制保证了内存管理的一致性。代价则是对栈的操作的复杂性增大以及栈空间不能够动态增长。我们只能在编译阶段为栈指定一个最大空间，并且在进程运行的时候只能且一定为栈分配同样大的空间。这样使得模拟器可能在空间上有较大的开销。进一步的想法是，可以为段在内存中的保存设置成可动态扩充的机制，同时设定一个标记位标记哪些段可以被扩充，这样能够使得模拟器在空间复杂度上有一定的提高。

4.6.3 内存模块的数据结构

进程内存的数据结构如下:

```
typedef struct{
    unsigned int seg_num;
    PROC_SEGMENT * segments;
}PROC_MEM;
```

对于进程内存中的每个Segment，结构定义如下:

```
typedef struct{
    unsigned int vaddr_offset;
    unsigned int size;
    uint8_t *base;
    int flag;
}PROC_SEGMENT;
```

另外，栈也作为一个Segment统一进行内存管理，定义如下:

```
typedef PROC_SEGMENT PROC_STACK;
```

4.7 调试模块

调试模块是模拟器中另外一个重要的模块，这个模块对模拟器的底层状态进行了封装，并且实现对模拟器状态的修改。这个模块中实现了反汇编（即Interpret子模块）和设置断点的子模块，使得模拟器的调试功能变得非常强大。使用模拟器验证编译器的时候，就如同使用gdb来调试c语言程序一样，能够轻松的跟踪程序状态，使得我们在编译器调试的效率得到了很大的提高。

下面就对调试模块本身和它的两个子模块（反汇编子模块和断点子模块）进行一些说明。

4.7.1 调试模块实现的功能描述

调试模块包括四个部分: 输出cpu和指令统计信息的、寄存器堆的读写、内存的读写、栈的读写。

cpu和指令统计信息在流水线阶段进行统计，调试模块负责输出; 寄存器堆写是不包括对CMSR的写，寄存器堆读包括对所有寄存器的读; 内存读可以读取任何一块有效内存，但是寄存器写只能写可写内存，对于代码段等所在的内存是不可写的（因为这种内存改动是不必要的），这样可以保证了模拟器的安全性; 对于栈的读写则是以栈底到sp寄存器标记的栈顶一段部分的读写。

4.7.2 反汇编子模块

反汇编子模块是模拟器设计的一个亮点。这个模块的实现并不在开题报告的计划中，实现这个模块的动机来源于在用模拟器调试编译器的过程中，每次都要对ELF文件进行反汇编。为了调试方便，对机器码的反汇编模块就应运而生了。这个模块的编写在指令译码模块之后，在编写这个模块的时候吸取了指令译码模块的一些经验教训，把尽可能多类型的指令的反汇编放在一起来处理。这个模块主要是对机器码的解析，虽然技术含量不高，但是却是非常的实用。

4.7.3 断点子模块

断点子模块是调试模块的一个重要的组成部分。通过反汇编子模块得到程序的汇编码，再根据汇编码在我们需要的地方设置断点，可以更加有效的调试程序。由于断点的数量没有上限也没有下限，所以在实现这个模块的时候采用了动态增长的方式。开始为断点的最大数量设置一个初始值，若断点的总量达到了容量，则将其容量扩大为原来的二倍。这种方式与STL库中的Vector相同，实现了对该模块的动态管理。

4.8 模拟器的三种模式

为了使得模拟器适应多种不同的应用场景，模拟器设计了三种应用模式：普通模式、验证模式、自动模式。

工作在普通模式下的模拟器会输出最全面的调试信息，也能够执行最全面的调试命令，该模式主要用来做指令级的调试，验证每条指令的执行结果是否正确的写回了寄存器堆、是否正确的设置的CMSR，是否正确的写回了内存等等。该模式中可以设置安静模式（输入命令set m s），该模式只输出最终统计信息而不输出每个cpu周期流水线状态，便于获得较大程序的统计信息。

验证模式用于验证模拟器的正确性，在模拟器验证的过程中起到了很大的作用。该模式下只输出每条指令执行后的CMSR和最终的寄存器堆状态，而不输出任何其他调试信息。该模式主要用来验证每条指令是否正确的写回了寄存器堆和正确的设置的CMSR标志位。

自动模式是为了批量验证设计的。该模式下不会输出任何调试信息，并模拟器开始运行时就运行程序，输出所有程序输出。对该模式下的模拟器输出重定位到一个文件，就可以用diff命令与标准答案对比。在这里我们也实现了两个脚本batch_simulator和batch_simulator2来对批量的测试用例进行验证。

5 模拟器的验证

在模拟器的设计工作完成后，为了保证用模拟器来验证编译器的结果是否正确，首先要保证模拟器本身的正确性。虽然模拟器在设计逻辑上已经能够保证了对于一部分BUG能够区分出是模拟器的问题还是编译器的问题，但是对于非程序BUG（如模拟器本身对指令的实现）并不能由设计逻辑来保证，所以这里需要对模拟器进行整体的验证。

对模拟器的验证主要分为两个方面:

5.1 函数级验证

函数级验证主要负责验证Simulator的函数调用是否正确，着重验证栈和流水线的执行状态是否正确。重点验证递归函数的调用和主函数的返回、控制流是否正确（分支循环）。

验证内容和代码列表如下:

1. 简单的main函数验证，验证代码: 00simple.c
2. 函数调用验证，验证代码: 01function.c
3. 递归函数调用验证，验证代码: 02recursive_func.c
4. if分支语句验证，验证代码: 03if.c
5. while循环语句验证，验证代码: 04while.c
6. for循环语句验证，验证代码: 05for.c

5.2 指令级验证

指令级验证主要负责验证Simulator实现的各条指令是否正确，指令验证分为三个部分: 数据处理指令、分支跳转指令、访存指令，其中数据处理指令主要验证数据正确性和标志寄存器是否被正确写入，分支跳转指令主要验证跳转位置的正确性，访存指令主要验证存储器读写的正确性。

验证的主要策略是按代码中不同的域进行验证。

验证内容和代码列表如下:

a、数据处理指令的验证（D_Imm_shift、D_Reg_shift、D_Immediate）

所有数据处理指令验证的代码参见verification/data_process_inst文件夹

1. opcodes域的验证，验证代码: opcode.s
2. shift域（前两类）和Rotate域（第三类）的验证，验证代码: shift.s

3. S域的验证（标志寄存器设置），验证代码：cmsr_logic.s，cmsr_arith.s

4. PC作为目的寄存器的验证，验证代码:pc_dst.s

5. 乘法和乘加指令的验证，验证代码：mul.s

b、分支跳转指令的验证（主要是条件转移指令的验证）
分支跳转指令验证的代码参见verification文件夹

对cond域的验证和跳转地址正确性的验证，验证代码：30b_cond.c

c、访存指令的验证
访存类指令的验证代码参见verification文件夹

1. 单数据传输指令的验证:

shift域的验证（L_S_R_Offset），验证代码：40mem_single_shift.s

立即数访存的验证（L_S_I_Offset），验证代码：41mem_single_imm.s

U域、B域的验证，验证代码：42mem_single_U.s

2. 半字和有符号字传输指令的验证:

S、H域的验证，验证代码：50mem_Hw_S_H.s

6 与编译器的联合调试

由于模拟器部分是与编译器部分联合进行实验的，所以模拟器与编译器联合进行调试是模拟器工作后期的重点。虽然在前期的时候模拟器和编译器两边的人员分工各有偏重，但是在模拟器验证、编译器目标代码生成和优化这一段时间里，模拟器和编译器最终在联合调试的过程中走到了一起，完善了二者的功能。在这一部分中，模拟器的作用主要表现在三个方面

6.1 对编译器目标代码生成正确性的验证

编译器目标代码生成是一项复杂的工作，具体的工作由我们小组的四个人协作完成。对编译器生成的目标代码进行正确性验证的工作则是由模拟器来完成。由于我们的模拟器设计的非常成功，所以在对编译器进行调试的过程中节约了大量的时间，尤其是上文中提到的反汇编模块和对寄存器堆、内存的显示和修改使得模拟器可以逐个指令的检查编译器生成的目标代码是不是我们期望得到的目标代码。这一段时间内模拟器主要是通过逐条指令的执行和人工的验证来对编译器的正确性进行验证的。

6.2 对编译优化的验证

编译优化的首要一条是要保证优化后的代码的正确性。所以模拟器对编译优化的验证也成为了编译优化这个环节的基础。由于我们做了大量的编译优化，对于每一种优化模拟器都需要对我们构造的测试集和老师为我们提供的十个测试样例进行验证。这个过程是一个非常机械化的过程，我们对于这个过程的实现采用了自动化的方式。首先我们修改了模拟器，增加了自动运行模式；然后我们写了一个脚本把编译器封装起来，实现了前端和后端的衔接；最后我们用batch_simulator脚本将他们联合起来。我们将所有的测试样例放在一个文件夹中并且构造了测试集文件，对于每个测试文件我们将他们的标准答案放在扩展名为ans的同名文件中，batch_simulator脚本在测试集文件中读取测试文件的文件名，依次将他们用我们的编译器编译成为可执行文件，再用模拟器的自动运行模式得到最终结果，并重定位到结果文件中与标准答案做diff操作，即可判断编译器对于该测试样例的编译结果是否正确。

6.3 对编译优化效果的衡量

对编译优化效果的衡量主要应用的是模拟器的统计模块，主要统计cpu的总周期数和空闲周期数，以此来衡量编译优化后和优化前的性能。

7 总结

7.1 实验感受

这次编译和体系联合实验是我们在大学两年半做过的最完整的实习，两门紧密结合在一起的实习课程让我们连贯的体会了编译和体系密切的关系。在体系课程上我们知道，对于一个好的CPU团队，一定要有一个好的编译器团队为他们做支持，通过这个实习我们对这一点有了更深的体会。对于CPU的不同设计，需要不同的编译优化策略才能使得CPU发挥最佳的效率。在实验中团队的协作、讨论也让我们在合作中逐渐完善自己，在最终我们编译器编译出来的程序能够正确快速的在我们实现的模拟器上运行的时候，那种满足感也是无以言表的。这两门实习，虽然是我们大学生活中最难的、也是最耗时的两门实习，但是确是对我们来说意义最为重要的实习之一，相信我们学到的能够对得起我们所付出的。

7.2 一些建议

做完了这两个实习，我们或多或少也有了一些对这两门实习的一些想法和建议，在这里我们针对模拟器这一部分说一说我们的一些建议。

1. 由于体系实习是和体系课程一起进行的，在实现模拟器的时候还有很多的课程没有学到。这对我们来说是一个挑战，这也导致了我们在学过了某部分内容之后对涉及到该部分的项目代码进行了重构。希望我们这门课程经过几个学期的积累，能够形成一个更为完善的体系，在实现之前就对于一些关键的部分提前说明，避免走弯路。

2. Unicore32的指令手册中标明的汇编指令和真正汇编器实现的汇编指令有着较大的差别，希望在以后的课程中能够提供更为完善的文档。

3. 我们在两个实习中实现了编译器和模拟器，但是在这两部分之间仍然有这一道鸿沟，就是汇编器和链接器。尤其是链接器部分，并没有什么课程对此进行详细的讲授。如果可能可以考虑我们的实习课程实现一些简单的汇编器和链接器的功能，让编译器和模拟器真正的连接在一起，形成一个更为完整的体系。

最后，感谢刘锋老师和刘先华老师一个学期以来对我们的悉心指导，谢谢你们。

附录

A Simulator配置说明

Simulator的编译需要版本高于2.6.32的linux内核环境以及版本高于4.4.3的gcc环境下运行，另外需要libelf库的支持。

A.1 libelf库的配置

进入模拟器根目录，在终端输入如下指令:

```
1 cd lib
2 tar xvf libelf-0.8.13.tar
3 cd libelf-0.8.13/
4 ./configure --enable-gnu-names --prefix=/usr/local
5 make
6 sudo make install
```

即可将libelf库装入系统中。

A.2 模拟器的编译

模拟器的编译很简单，只需要进入模拟器根目录，运行make命令即可生成可执行文件simulator。

B 模拟器使用说明

我们的模拟器能够支持多种模式和多种命令，下面对每种命令进行详细的说明:

B.1 三种模式的使用

模拟器支持三种模式: 普通模式、验证模式（-v参数）、自动模式（-r参数）。对于三种模式的说明已经在上文中说明过，在此不再赘述。

三种模式需要在模拟器启动的时候指定，普通模式不带任何参数，验证模式带-v参数运行模拟器，自动模式带-r参数运行模拟器。

例: ./simulator qsort -r

B.2 运行指令

模拟器启动后，出现命令行提示符“>”，此时可以输入命令。

调试程序前需要将程序运行起来。有两个命令可以将程序运行起来，分别是“r“命令和“rt“命令。其中“r“命令会直接让程序执行到下一个断点或者程序结尾，而“rt“命令使程序进入单步模式，此时程序运行起来并且执行了第一条指令。

B.3 单步运行指令和运行到下一个断点指令

程序进入单步模式后，输入“s“命令使得程序执行下一步操作，输入“n“指令执行到下一个断点或者程序结尾。

B.4 list指令

list指令可以指定列出某部分程序的汇编代码。

list指令有三种格式:

1. list后不带任何参数，输出当前位置后10行代码。
2. list后可以带c参数或者f参数，前者表示在当前位置列出代码，后者表示列出某函数入口处的代码。

   如list c 5 表示列出当前位置后5行代码

   如list f qsort 7 表示列出函数qsort从入口如开始7行代码

   上述两种用法均可以省略最后一个参数，默认列出10行代码

3. list后跟0x开头的地址，表示列出地址对应位置的代码。其后的代码行数也是可选的。

   如list 0x020002a8 9 表示列出地址0x020002a8处开始9行代码

在线帮助文档可以在命令行提示符中输入help list。

B.5 断点类指令

断点类指令主要包括设置断点、删除断点、显示断点三个功能。

1. 设置断点: b <address> 或者 b [a/add] <address>
2. 删除断点: b [del/delete] <breakpoin_id>
3. 显示断点: b [l/list]

在线帮助文档可以在命令行提示符中输入help breakpoint。

B.6 显示信息类指令

显示信息类指令包括显示CPU信息（info cpu）、显示寄存器信息（info registers）、显示栈信息（info stack）、显示标志寄存器信息（info CMSR）、显示动态指令数统计信息（info inst）。

在线帮助文档可以在命令行中输入help info。

B.7 显示寄存器信息、内存信息

显示单个寄存器信息（print $rn，n为寄存器号），显示内存信息（x <address>）。

在线帮助文档可以在命令行中输入help print和help x。

B.8 修改寄存器、内存信息

修改寄存器信息（modify rn <value>，n为寄存器号，value为要修改的值），修改内存信息（modify <address> <value>，address为需要修改的内存地址，value为要修改的值）

B.9 退出程序

使用命令q或者quit可以退出程序。