体系结构实习——**UniCore2** 模拟器
实习报告

张番栋 00848180
刘澜涛 00848200
王　沛 00848205
CS08

December 23, 2010

目录

第 1 章　实习内容

1.1　目标

实习的主要内容是用 C 语言编写一个支持 UniCore2 精简指令系统的模拟器，并对其进行测试、验证。在之后的报告中，均称实习所完成的模拟器为 MinSim。MiniSim 的首要目标是进行正确的功能模拟，在此基础上增加对 CPU 流水线以及 Cache 的结构模拟，另外还有对程序动态运行情况的统计。MiniSim 采用五级流水结构，和真实的 UniCore2 处理器并不相同。模拟的 Cache 则具有较灵活的定制性。

1.2　具体实现功能

MiniSim 支持的 UniCore2 指令子集中包含五类指令:

- 数据处理指令
- 乘法和乘加指令
- 跳转切换指令
- 单数据传输指令
- 条件转移指令和带链接条件转移指令。

从中可以看出，MiniSim 并未对处理器特权状态的相关功能进行模拟，进而无法完全支持基于现代操作系统下的大部分程序。针对这个问题，我们做了一些工作，使得 MiniSim 在接受标准 ELF 文件作为输入的情况下，仍然能够完成大部分应用级别的功能模拟。

本次实习是与编译实习联合进行，因此我们在完成实习的过程中做了一些编译器和模拟器之间的协调工作。

最后，为了方便使用和调试，我们为 MiniSim 编写了一个简单的控制台模块，使得 MiniSim 在运行时可以设置断点，单步执行、查看寄存器、内存和流水线的状态。

第 2 章　模拟器的存储体系

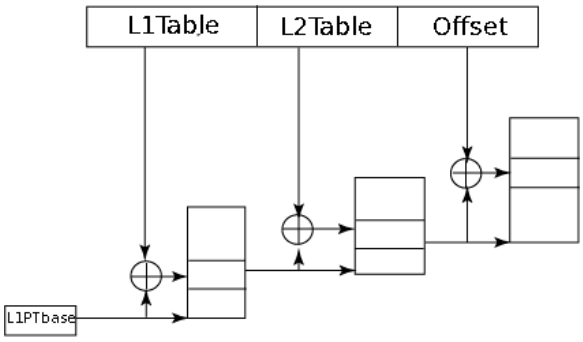
2.1　寄存器堆

MiniSim 以 UniCore2 处理器为目标进行模拟，因此寄存器堆的结构基本与之相同。有所区别的是，由于 MiniSim 无法处理异常（Exception），也不区分处理器状态，所以不需要 SMSR 标志寄存器。因此 MiniSim 共有 32 个通用寄存器和一个 CMSR 标志寄存器。

2.2　内存管理

MiniSim 在运行时需要在自己的地址空间内同时维护目标程序的地址空间，因此需要建立一套面向目标程序的虚拟内存机制。在这个问题上，MiniSim 采用的是页式内存管理机制。对于目标程序的 32 位地址空间，MiniSim 进行了如下划分

- 31-25 位: 一级页表偏移
- 24-15 位: 二级页表偏移
- 14-0 位: 页内偏移



这样的结构表明页的大小为 32KB。一级页表常驻内存，共有 128 个页表项，每个页表项大小为 4B，即指向二级页表的指针。每个二级页表含有 1024 个页表项，每个页表项 8B，存储的信息包括其所指向的页的起始地址和该页的读、写、执行属性。

这是一个很经典的虚拟内存管理机制，尤其在管理目标程序的栈时可以提供较好的局部性。在这种机制下，只要模拟器的存储管理模块要提供读、写内存的操作接口，真正执行目标程序的模块就可以减少很多工作量。同时，页式结构在一定程度上阻止非法的内存访问。

2.3 Cahce

MiniSim 的 Cache 采取可定制结构，即用户可通过提供不同参数来获得不同的 Cahce 组织方式和 Cache 维护策略。我们共提供 3 种 Cache 更新策略（LRU，随机，轮转）和 2 种 Cache 写回策略（直写和回写）。

第 3 章　模拟器运行环境的建立

MiniSim 接受标准 ELF 文件作为输入。在真正模拟运行目标机程序之前，需要做一些初始化工作。换言之，MiniSim 需要完成一部分装载器和操作系统的工作。主要内容有 ELF 文件解析内存环境的建立、代码段和数据段的载入、寄存器堆和流水线的初始化等等。

3.1　对系统相关指令、功能的处理

由于 MiniSim 不支持操作系统级别的指令，同时也不需要支持复杂的程序运行环境，所以模拟区间仅限于 main 函数。也就是说，MiniSim 会跳过 main 函数之前的代码段，待 main 函数结束后退出。另外，为了便于检查，MiniSim 还需要支持整型数据的非格式化输出。输出是与系统调用有关的功能，因此需要做一些约定：在给予 MiniSim 的输入中，输出功能要被封装在一个特定的函数中。当模拟进行到这个函数时，模拟器会以宿主机的输出调用替代之，然后结束该函数，从返回地址继续执行。为了达到这个目的，需要在 ELF 文件装载阶段进行一些工作。

3.2　ELF 文件解析

需要做的工作主要有两点，具体的流程与 ELF 文件结构关系密切，这里只进行简单讨论。（ELF 文件结构在 elf.h 头文件中定义。）

3.2.1　关键函数的入口地址

关键函数指的是 main 函数和输出封装函数。前者的入口是模拟程序的入口，后者的入口则决定了模拟器调用宿主输出机制的时机。大致流程如下：

1. 找到.shstrtab 节区。
2. 遍历所有节区, 找到类型为 SHT_SYMTAB 的两个节区:.symtab 和.strtab。
3. 遍历.symtab 节区, 找到需要的函数符号。
4. 返回其虚拟地址。

3.2.2　装载程序段

获得 ELF 文件中代码段和数据段的位置和大小，以备建立模拟内存环境之用。

1. 获取程序段表 (Program Header Table) 基址
2. 遍历程序段表, 找到所有类型为 PT_LOAD 的段。一般只有 2 个，分别存放代码、只读数据和可读写数据。

3. 根据段的大小和段需要的起始虚拟地址开辟新页，将段的数据从文件装入内存。

基于效率的考虑，MiniSim 没有建立动态装载的机制，即在模拟开始前将可能需要的代码和数据一次性装入。如果要进行动态装载，那么在每次访存过程中都需要判断是否要转入新页。由于取值操作每个周期都要进行，这样的判断会带来较大的性能损失。

3.2.3 寄存器堆的初始化

寄存器堆中有一些重要的寄存器需要在执行程序前进行必要的初始化。需要初始化的寄存器主要有 4 个：

1. PC。在运行程序前，需要在 PC 中装入之前解析 ELF 文件所得到的 main 函数入口地址。

2. LR。为了让 MiniSim 能够检测到 main 函数的结束，需要给 LR 一个特殊的值（一般是指向保留的地址空间，MiniSim 在实现时采用了 0 这个特殊值）。这样当 PC 等于这个特殊值是，MiniSim 就可以确定 main 函数已经结束，从而结束对目标程序的模拟执行。

3. SP。该寄存器的初始值决定了目标程序的栈的起始位置。在 MiniSim 的视线中，SP 被初始化为 0xf0000000。

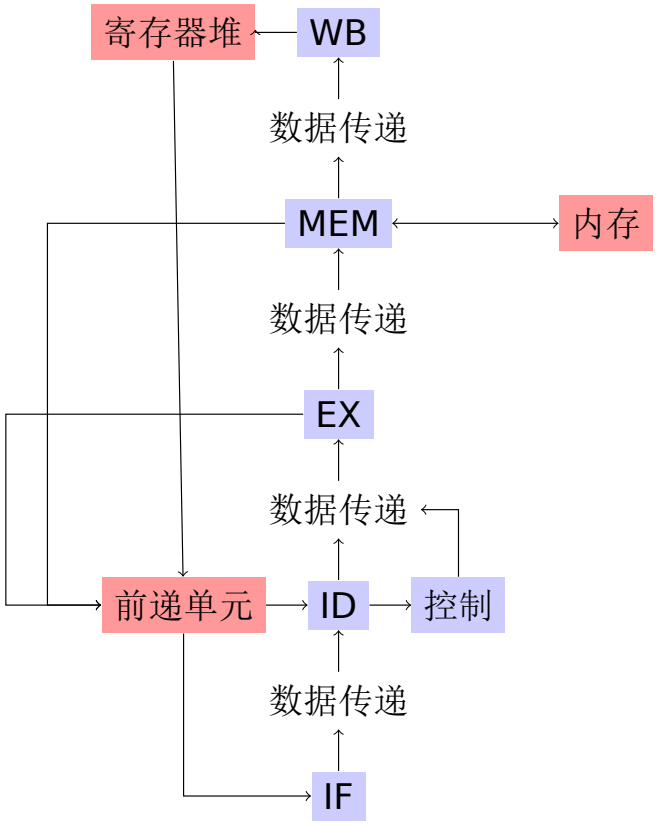
4. FP。该寄存器的初始值决定了 main 函数栈帧的基址，因此需要给予与 SP 相同的初始值。

第 4 章　流水线设计

4.1　整体结构

4.1.1　设计描述

MiniSim 在完成指令功能模拟的同时，以软件的形式模拟了处理器的流水线结构。MiniSim 的流水线分为取指（IF），译码（ID），执行（EX），访存（MEM），写回（WB）五级。由于软件不可能实现真正的并发，所以 MiniSim 的流水线实际上是以从 WB 到 IF 的顺序分步执行的。虽是如此，在真实流水线存在的与并发有关的问题在 MiniSim 中还是有所体现，比如数据和控制冒险。解决这些问题将是流水线设计和实现的重点。下图为流水线的基本结构：



4.1.2　程序实现

下面是 MiniSim 实现流水线的主要数据结构：

```
typedef struct
{
    FwdData ex_fwd[2];
    FwdData mem_fwd;
```

```
5
6       /* data transfered between pipeline stages */
7       ID_input id_in;
8       EX_input ex_in;
9       MEM_input mem_in;
10      WB_input wb_in;
11  } PipeState;
12
13  typedef struct
14  {
15      uint32_t reg[32];
16      PSW CMSR;
17      uint32_t usr_trap_no;
18  } RegFile;
```

PipeState 中存储了每一个流水级所需要的数据及控制信息。

1. ID_input，主要是需要译码的指令以及从 IF 阶段上传的流水级信息。
2. EX_input，包括 ID 阶段获取的操作数、需要执行的 ALU 或乘法器操作类型、以及在 ID 阶段产生、需要上传个 MEM 和 WB 的控制信号。
3. MEM_input，包括从前一流水级上传的控制信号，主要有读写控制、地址选择（基址还是变址）、访存数据宽度、是否进行对 load 数据进行符号扩展等。
4. WB_input，包括需要回写的目标寄存器号，需要回写的数据本身，以及回写目标选择（回写存储器输出，回写 EX 阶段输出，或者都回写，或者都不回写）。

4.2 数据冒险相关

由于数据冒险和控制冒险问题影响着整个流水线的运行，所以在描述各个流水级之前，先对 MiiSim 解决数据冒险的机制和策略做一介绍。

4.2.1 两种数据冒险

在 MiniSim 中，影响流水线运行的数据冒险实际上只有 RAW 一种。根据解决方式的不同，将之划分为两种:

1. 第一种 RAW 数据冒险是由数据处理指令产生的。在这种情况下，可以对数据处理指定的结果进行前递来满足后续指令的数据需求。
2. 第二种 RAW 数据貌相是由加载指令产生的。因为加载指令的结果在 MEM 阶段才会产生，所以无法通过前递满足紧随其后的一条指令的需求。如果此时发生数据相关，只能通过暂停流水线来解决。我们称这种情况为加载互锁。

4.2.2 数据冒险的解决

1. 数据前递。数据前递的实现方法是在流水线状态信息（数据结构为 PipeState）的 ID_input 域中加入了三个前递数据槽。其中两个接受 EX 流水级前递的数据，一个接受 MEM 流水级前递的数据。为 EX 设立两个数据槽的原因是，UniCore2 指令集中的访存指令对基址寄存器可以进行回写。因此相邻的两条指令中可能产生多达 4 次的回写。同时 Unicore32 指令最多可以有 3 个源寄存器。基于以上事实可知，仅仅两个数据槽不能满足所有的前递要求。因

此需要为 EX 阶段增加一个数据槽。MEM 阶段因为距离 WB 阶段只有 1 个周期，因此只需要有一个数据槽即可。

2. 加载互锁。加载互锁的实现比较简单，只需在 ID 阶段判断是否前一个流水级是否为 load 指令，该 load 指令与当前 ID 阶段的指令是否有数据相关即可。发生相关的条件是要读取的寄存器号与需要回写的寄存器号相同。由于流水线和模拟器流程执行顺序的关系，ID 阶段要判断的回写信息存储在流水线状态中的 mem_in 一级。当然，如果 ID 之后的流水级中插入了气泡，那么判断无需暂停流水线。

因此对于数据冒险问题的解决可以概括如下：

在数据前递方面，是否进行前递基于对回写信号的解析。为 EX 设立的两个数据槽形成一个队列，第一个数据槽永远保存最新一次的前递信息。

读寄存器堆方面，首先判断是否需要进行加载互锁，如果是，暂停流水线，否则要扫描前递槽。需要注意的是 EX 数据前递槽的第二个槽要最后扫描，如果命中则取数据。未命中，则直接从寄存器堆取数据。

4.3 取值（IF）阶段

IF 阶段完成的工作是取值，判断关键入口并进行处理，或者判断模拟是否已经结束。

4.3.1 取指令

IF 阶段中会根据寄存器堆中 PC 的值访问存储器（借口由存储管理模块提供），并在取值结束后对 PC 进行更新（加 4 操作）。

值得注意的是，由于 UniCore2 指令系统中的 PC 对汇编程序是可见的，所以程序员可以将 PC 作为数据处理指令的目的寄存器使用。由于每一次 PC 的非累加性更改都相当与一次跳转，所以我们希望对与 PC 的修改可以尽早被探测到。因此，需要考虑 PC 的前递问题。

这也就是说，在取指令时不能依赖当前的 PC 值，而是需要先检查前递单元中是否有指向 PC 的前递。如果有，则要根据前递值去取指，并根据前递来源来清空流水线。如果前递来自 EX 阶段，则需要清空从 ID 到 MEM 的所有流水级。需要清空 MEM 的原因是，由于流水线倒序执行，到 IF 得到前递值时，制造前递的指令实际上已经从 EX 流水级进入了 MEM。同理，如果前递来自较早的 EX 阶段或者 MEM 阶段，需要将从 ID 到 WB 的所有流水级排空。

可以看到，这样对 PC 的操作会对流水线产生非常不利的影响。因此我们约定在编译器中绝对不会通过直接修改 PC 来实现跳转。

4.3.2 对关键函数入口地址的处理

最初已经提到，我们的模拟器无法处理与处理器特权态有关的指令，加之编译器方面的限制，只能支持一种与系统调用有关的行为，即输出一个整数。在模拟器执行指令前的 ELF 文件载入工作中已经获得了封装输出子例程的函数入口。IF 阶段要做的就是判断当前的 PC 是否为这个关键入口，如果是，则 IF 向 ID 送出一条伪指令，编码为 0xffffffff。这条指令在 ID 阶段会被进行特殊处理。

4.3.3 模拟结束的时机

对结束模拟执行时机的判断也在 IF 阶段进行。同时将 lr 寄存器的值设置为 0。这样一旦在 IF 阶段发现 PC 为全 0，就可以认为模拟即将结束了。此时 IF 阶段会

在产生 4 个汽泡后（带其他流水级中的指令执行完毕），产生结束模拟的信息。

4.4 指令译码（ID）阶段

ID 阶段是五个流水级中最复杂的阶段，需要完成的工作较多。对指令类型的判断、重要控制信号以及 EX 阶段所需操作数的产生、跳转的实现等都在 ID 阶段完成。

4.4.1 指令译码

对指令进行译码，根据指令信息产生相关的控制信号（在程序中就是为与控制有关的变量赋相应的值）。控制信号由专门的控制模块产生。它独立于 ID 模块。

4.4.2 产生操作数

EX 阶段所需要的两个操作数也在 ID 阶段产生，所以需要完成读寄存器堆、对寄存器中的值进行移位等工作。这之中伴随着数据冒险的检测和解决。如果检测到需要加载互锁，ID 阶段函数会返回一个特殊的值交由流水线判断，流水线控制逻辑会根据 ID 的返回值决定是否进行互锁。一般真实流水线中取前递值的操作应该是在 EX 阶段的最初时刻进行。在 MiniSim，基于简化软件设计的考虑，我们将数据冒险的检测和解决封装到了读寄存器堆操作中，也就是说读取前递值这项任务在 ID 阶段就已经完成。

EX 阶段需要的第二操作数一般并不是某个寄存器中的数据本身，而是需要对其进行某种移位操作。在这过程中还需要根据移位情况对 CMSR 的 C 位进行设置。具体的移位类型有：

1. 寄存器的立即数位移。
2. 寄存器的寄存器位移。
3. 立即数的立即数循环右移。

移位的具体实现被封装在一个函数中，该函数会根据基址，移位值和移位类型自动产生结果并设置标志位。

4.4.3 跳转

第 5 章　模拟器控制台

第 6 章　测试与验证