Y86 Simulator Project Report

章凌豪

13307130225

2015 年 6 月 7 日

Contents

1 Overview

1.1 Development Environment

开发语言	JavaScript / HTML / CSS
浏览器环境	Chrome / Firefox / Safari
	jQuery
第三方库	Bootstrap
	FileSaver.js
	Chart.js

1.2 File Organization

本次 Project 的主要代码均在 **js** 目录下，说明如下：

js/constants.js	一些常量的定义
js/utils.js	一些辅助函数的实现
js/register.js	寄存器和流水线寄存器的实现
js/memory.js	内存的实现
js/cache.js	缓存的实现
js/kernel.js	ALU 和 CPU 的实现
js/assembler.js	Y86 汇编器的实现
js/disassembler.js	Y86 反汇编器的实现
js/controller.js	控制函数的实现

2 Design

本节将叙述对本次 Project 的需求分析和开发选择，并列出最终实现的所有功能。

2.1 Motivation

本次 Project 要求实现一个 Y86 流水线模拟器，这涉及到大量的逻辑和封装，以及对图形界面的需求。最后我使用了 Javascript，配合 HTML 和 CSS，开发了一个纯前端的模拟器。这么选择的原因如下：

首先，我能够熟练运用的语言有 C 和 Python，比较熟练的有 Java 和 JavaScript。其中 Python 和 JavaScript 这两种**面向对象的脚本语言**能够给逻辑部分的开发能带来很大便利。

其次，考虑到要为模拟器实现一个图形界面，而且要在界面上显示许多数值和状态。那么采用前后端分离的方式显然是不合适的，因为**前后端之间交互的数据量很大**，一方面影响性能，另一方面也会带来许多冗余的绑定代码。

基于这两点考虑，我可以采用纯前端 Web App 的形式，用 Javascript 实现所有的逻辑和界面；也可以采用 Desktop App 的形式，用 Python 配合一个 GUI 库进行开发。由于我没有 Desktop GUI App 的开发经验，加上 **Javascript 的数据表现能力明显更为出色**，所以我最后选择了用 JavaScript 来完成这次 Project。在开发过程中，我也充分利用了 JavaScript 的**闭包**和**匿名函数**等特性，避免了**命名空间重复和手动管理内存**等琐碎的问题。

2.2 Features

最终的版本实现了所有的基本功能：

- 实现了 Y86 指令集中的所有指令
- 实现了流水线控制逻辑
- 支持载入和解析.yo 文件，并能将每个周期内流水线寄存器的值作为输出保存到文件

在此基础上，我还添加了许多新功能：

- 支持步进、步退、自动运行、暂停等操作，并能以不同频率（最高 1000Hz）运行。
- 显示内存中的数据并指示当前栈的位置，也可以监视一个指定的内存地址。
- 实现了 **Y86 汇编器和反汇编器**，从而能够接受.ys 文件作为输入，也能保存汇编和反汇编的结果。
- 内存的实现采用了**分页技术**。

- 模拟实现了一个 **L1 缓存的简化版**。
- 程序运行结束后可以生成**性能分析**，包括对缓存命中和 CPI 的统计。
- 添加了一条新指令 iaddl，并**归纳了添加指令的步骤**。

具体的实现细节将在第三节中叙述，同时在第五节中会对部分功能进行进一步的讨论。

3 Details

本节将对模拟器的各个部件的具体实现细节进行详述，对于没有提及的细节请查看代码中的注释。

3.1 Kernel

在 **js/kernel.js** 中，我实现了 **window.VM** 对象，它包括**内存、缓存、两组寄存器**，以及整个模拟器的核心——**CPU**。

首先我用大量闭包实现和封装了一个 ALU，以供 CPU 调用。

在实现 CPU 时，我创建了两个对象 *input* 和 *output*，分别用于表示当前周期始流水线寄存器的状态以及当前周期末流水线寄存器的状态。在 $fetch$、$decode$、$execute$、$memory$ 和 $write\_back$ 五个阶段，主要的工作就是**实现流水线内部的数值传递以及读写内存**。而在 $pipeline\_control\_logic$ 阶段，我主要参考了课本提供的 HCL 代码，实现了 F_stall、D_stall、D_bubble、E_bubble 和 M_bubble 五种流水线控制机制，并对应地修改了流水线寄存的数值，最后将 output 作为新的 input 保存。

最后在 $CPU.tick()$ 中模拟了一个周期内 CPU 进行的操作。再加上一些用于获取内部信息的接口，CPU 对象也封装完毕了。

一些可能导致部分流水线寄存器的数值与其他实现不同的细节说明：

- 当一条指令没有用到某个流水线寄存器时，该寄存器的数值保持上一周期的状态不变。这不影响流水线逻辑。
- 当 E_icode 不是 JXX 时，M_Bch 的值不影响流水线逻辑，所以我选择对这种情况不加区分，仍然令 $M_Bch = alu.getCnd(input.E\_ifun)$。由于大部分指令 $ifun$ 都为 0，所以 M_Bch 大部分时间会等于 1。

3.2 Memory

内存使用 JavaScript 内建的 **Int8Array** 实现。在内存中数据以 **Little Endian** 格式存储；与外部交互时数据以 **Big Endian** 格式传输。

为了**有效管理不连续的内存**，我使用了**分页技术**。在具体实现中，每个 Block 为 $64K$，默认最多分配 1024 个 Block，即最多分配 $64M$ 内存。对于一个 32 位的地址 $addr$，令 $highAddr = addr >> 16$，$lowAddr = addr\&0xFFFF$，然后建立一个 $highAddr$ 与 Block 之间的映射 $mapping$，从而对 $addr$ 对应的内存进行操作就映射到了对 $Blocks[mapping[highAddr]][lowAddr]$ 进行操作。这样做可以有效处理栈指针减过 0 从而下溢出后带来的数组范围过大问题。

在分配一个新 Block 时，我选择将内存初始化成 $0xCC$，这样做的灵感来自于 **x86 指令用** $0xCC$ **来表示调试中断**，既方便了对非法指令地址的处理，也更直观地区分了未写过的内存和值为 0 的内存。

另外我还实现了一系列对内存进行读写的接口，以满足不同的需求。

3.3 Cache

为了模拟缓存机制，我在模拟器中实现了一个 **L1 Cache** 的简化版。对内存进行读写操作时，都要先访问缓存。若缓存未命中，则再访问内存，同时将读取或写入的值重新写入到缓存中。

缓存的参数设定为 $S = 2, E = 2, B = 64, m = 32$，即缓存中包含两个 Set，每个 Set 包含两条 Line，每条 Line 包含两个 Block，每个 Block 可以存储一个 32 位的数字。数据的读写都以一个 Word（32 位）为单位。这与现代处理器的 L1 Cache 相比是很大的简化，但就**说明缓存的概念和机制**而言是足够的。

每条 Line 有 $Valid$ 位表示是否写有数据，有 $Dirty$ 位表示数据在写入 Cache 之后是否被修改过，有 $countVisit$ 位记录被访问次数。

缓存策略如下：

- **选取策略：** 由于数据的读写以 32 位为单位，所以两个相邻地址之间差为 4，因此取 $index$ 为地址的低第 4 位，$offset$ 为地址的低第 3 位，这样可以较好地利用 **Spatial Locality**。
- **抛弃策略：** 当两条 Line 的 $Valid$ 位都为 1 时而又需要写入新数据时，选择访问次数少的一条 Line 写回内存。这样可以较好地利用 **Temporal Locality**。
- **写回策略：** 当一条 Line 的 $Dirty$ 位为 1，即数据被修改过时，才将其写回内存。

实现缓存之后，所有读写内存的接口函数都要改为访问缓存，只保留 $readWord$ 和 $writeWord$ 这两个直接读写内存的接口供处理 Cache Miss 时调用。

3.4 Assembler

为了方便测试用例的编写，我实现了一个 Y86 汇编器，在载入.ys 文件后能将其汇编为.yo 文件，再作为输入交由模拟器来运行。

具体的实现借鉴了一些**编译原理**中的做法，大致步骤如下：

1. 格式化输入文件，用正则表达式去除备注和多余空格。
2. 扫描并记录所有的 **symbol**，同时处理 **directives**，包括.pos、.align 和.long。其中对于.pos 和.align 要重新计算指令偏移量。

3. 再次扫描输入，对于每一条汇编指令数据，先根据定义好的语法格式提取它的参数并进行编码（必要时查阅 symbol table），再根据相应的格式拼接成目标码。

4. 拼接结果，或是处理 catch 到的异常。

为了降低耦合，我把 Y86 汇编代码的语法以及从汇编指令到目标码的转换函数分别封装在了 *insSyntax* 和 *insEncoder* 中，这样也方便添加对自定义指令的支持。

3.5 Disassembler

作为实现 Y86 汇编器的后继，我又尝试实现了一个不是特别完善的 Y86 反汇编器。它能将载入的.yo 文件反汇编成.ys 文件，但**不能区分指令和数据**。反汇编失败的部分会在结果中注明。

实现步骤与汇编器类似，如下：

1. 格式化输入文件。

2. 扫描输入并尝试转换每一行目标码为汇编指令。其中对于 JXX 和 CALL 这两类指令，要将其跳转到的地址加入 target table，并在反汇编结果相应的地方插入 target。

3. 拼接结果，处理异常。

类似地，目标码的语法和从目标码到汇编指令的转换函数分别封装在 *codeSyntax* 和 *insDecoder* 中，方便修改和添加。

3.6 Web UI

由于对 CPU、内存以及缓存做了很好的封装，实现界面时就轻松多了。我采用了比较传统的 Web App 实现思路，在 **index.html** 中确定了页面布局，并将按钮和输入框等交互元素与 **js/controller.js** 中实现的诸如开始运行、保存结果、更新显示等控制函数进行了绑定。而在控制函数中只要简单地调用 **js/kernel.js** 中实现的 **window.VM** 对象就可以通过各种接口获取所有需要展示出来的状态和数值。网页的样式则是使用了 **Bootstrap**，并在 **css/simulator.css** 中自定义了部分样式。

界面的实现中值得一提的有：

- 将文件拖拽页面中任一位置即可加载，通过 **HTML5** 的 **drag & drop** 实现
- 寄存器值发生变化时会触发**闪烁动画提示**，通过 **jQuery** 的 **animate** 实现
- 有两个指示%ebp 和%esp 的指针会随着对应寄存器值的改变而**滑动**，通过 **CSS** 实现
- 支持通过**键盘快捷键**控制运行状态，通过 **jQuery** 监听键盘事件实现

3.7 Performance Analysis

在程序运行期间，我对 Cache Hit 和 CPI 做了统计。

前者比较简单，在 Cache 的实现中添加一个计数即可。后者在计算时要注意增加计数值的位置。

count_mrmovl、*count_popl*、*count_cond_branch*、*count_ret* 的计数是在 CPU 的*execute* 阶段进行的，分别对应 E_icode = MRMOVL、POPL、JXX 和 RET 的情况。而 *count_load_use* 和 *count_mispredict* 则要在*pipleline_control_logic* 阶段进行，所对应的情况参考了课本上的描述。

最后根据以上数据生成两张对应的表格，并生成一张以周期为横坐标、以 CPI 值为纵坐标的图。

4 Evaluation

本节将叙述对模拟器的测试和性能评估。

4.1 Test Cases

在 **doc** 目录下收录有我所使用的测试用例，说明如下：

#	测试文件	描述	预期结果
1	asum.yo	测试样例，包括.long、循环和调用	%eax = 0xabcd
2	Halt.ys	测试模拟器是否能能正确结束运行	%eax = 0x1 %ebx = 0x0
3	Forward.ys	测试模拟器是否能正确 Forward 后修改的寄存器值	%eax = 0x3
4	Load_Use.ys	测试模拟器是否能处理 Load/Use Hazard	%eax = 0xd
5	Comb_A.ys	测试模拟器是否能处理 Combination A	%eax = 1
6	List_Sum.ys	对链表中的三个元素进行求和，测试内存取值相关操作	%eax = 0xcba
7	List_Sum_R.ys	同上，但用递归完成，测试调用和返回等操作	%eax = 0xcba

由于测试 1 包含了大部分情况，所以在模拟器通过测试 1 并且将结果与给出的 **asum.txt** 对比发现基本一致后，可以认为逻辑大体上没有问题。

再通过测试 2 到测试 5，可以基本肯定流水线逻辑（Stall / Bubble / Forward）没有问题。

最后的测试 6 和测试 7 是为了进一步验证模拟器的正确性，同时为性能分析提供更多例子。

此外还对几类常见的报错进行了简单的测试。由于时间原因，没有进行更多的测试。

4.2 Performance

在 Chrome 37.0 下测试，得到不限速时运行速度可达 $15Kcycle/sec$。

同时，在几个测试用例中，缓存命中率均在 25% 左右。

5 Discussion

本节将对设计和开发过程中遇到的一些问题、做出的一些取舍，以及目前的实现存在的一些局限性进行讨论。

5.1 Memory Issues

分页内存的实现主要是考虑到可能出现栈指针减过 0 从而下溢出的情形。因为我认为这种情况下简单地报错并不合适，所以对地址取了 Unsigned。但这又使得地址成为一个很大的值，从而导致数组范围过大，消耗大量内存甚至使浏览器进程崩溃。

实现了分页内存后，这个问题并没有迎刃而解，因为这个问题虽然在逻辑部分很容易解决，但在显示部分就没有一个太好的方案。假如在 0x100 附近和 0xFFFFFFF0 附近有一些内存被修改了，因为它们被分到不同的页，所以在显示时就有两种选择：一是分页显示，二是只显示有效的内存值。然而后者需要记录哪些内存是有效的，这显然不是一个具有扩展性的好方法；但前者不够直观，而且在页数变大时也面临过度消耗资源的问题。所以最后出于前端性能考虑，只显示前 1024 位内存。同时在测试中，我都手动将栈指针指定到 0x200 的位置，这样更便于展示栈指针的移动。

5.2 Disassembler Issues

在编写 Y86 反汇编器时遇到的一个大问题就是如何区分指令和数据。在尝试、查阅、讨论和权衡后，我决定不试图在反汇编器中实现对指令和数据的区分。在最终的实现中，反编译器会试图将每行目标码都当做指令去解析。假设一行目标码本应是数据，如果将其作为指令能解析成功，则继续；如果解析失败，则将其记录下来，并在反汇编的结果最后注明这可能是因为遇到了数据区域。

这么做的主要理由是，对于 Y86 这种**指令和数据可以任意混合**的汇编语言，不存在一个可行的办法对两者进行区分。因为不管是采用静态分析还是动态分析，要判断一行目标码是否属于一条指令，只能判断它是否能被执行到，而这就**与停机问题等价**。虽然在小规模的情况下可以通过暴力算法解决，但这样做相当于在程序中留下了一个隐患。

再就是对于反汇编来说，将数据误解析成指令是不可避免的，处理这种情况其实已经**超出了反汇编器的工作范围**。常见的反汇编工具（如 **objdump**），则是因为其处理的格式（如 **ELF** 格式）对指令区域和数据区域的编排是有规范的。

所以这个问题只能通过为 Y86 汇编语言制定类似的规范或者接受一个不保证能停止的算法来解决。基于以上考虑我决定不试图处理这种情况。

5.3 Limitation of Cache

在测试中，Cache Hit Rate 基本在 25% 左右。而在现实中 CPU 的 L1 Cache 往往有 95% 的 Cache Hit Rate。这之间的差异主要在于我实现的 L1 Cache 简化版只能容纳 $32B$ 数据，而现代 CPU 的 L1 Cache 往往是这个大小的上百倍，如 Pentium 4 处理器的 L1 Cache 就有 $8KB$。当然从另一个角度看也可以证明缓存机制的效用之高：**我只是根据缓存的相关概念和原理用极其简单的数据结构和策略进行了实现，就能达到可观的命中率。**

我这么选择主要是由于在当初实现时考虑到一个 Set 里有超过两条 Line 或者一条 Line 里有超过两个 Block 会带来一些代码上的复杂度，加上缓存策略的实现也颇为繁杂，容易出现难以调试的 bug。而在将缓存正确实现后，由于还要实现其他的功能，加上时间原因，没能再对这部分进行改动。如果将 Set 的数量改为 4 个并增加 Block 的数量，缓存命中率应该还能提高不少。

5.4 Limitation of Test Cases

如第四节中所言，本次 Project 中进行的测试所能覆盖的情况有限。如果有时间，可以多编写一些代码来进行测试。

不过由于在开发过程中**大量使用闭包、匿名函数、封装和解耦**，代码的可靠性还是很高的，在有可能导致程序崩溃的地方都做了限制，不容易 Crash。主要还是需要更复杂的 **Hazard Combination** 和 **Corner Cases** 来验证程序的逻辑和交互正确性。

5.5 Customized Instructions

在实现模拟器的大部分功能之后，我试着添加了一条新指令 IADDL，即将一个寄存器的值加上一个立即数。同时我将为模拟器添加新的自定义指令的步骤归纳如下：

1. 分析和列出要添加的新指令在每个阶段需要完成的操作。
2. 在 **js/constants.js** 中定义用于代表新指令的常量。
3. 在 **js/kernel.js** 中流水线的各个阶段做对应的修改：
4. *fetch*：将指令加入合法指令列表，并根据需求在抽取 $rA:rB$ 和 $valC$ 时将其添加到判断列表中。
5. *decode*：在设置 E_srcA 等一系列流水线寄存器的值时，将指令添加到对应情形的判断列表中。这里主要分为数据运算和移动以及调用栈这两类情形。
6. *execute*：同理，在设置 $ALU.inputA$ 和 $ALU.inputB$ 时，根据情形添加指令。如果指令涉及运算，不要忘记 $setCC$；如果指令可能成为 Hazard Cause，不要忘记增加对该指令的计数。

7. *memory*：判断指令是否要读写内存，并进行相应读写操作。

8. *write_back*：不需要做任何修改。

9. $pipeline_control_logic$：用与课本上类似的方法分析指令可能触发何种 Hazard，并将其添加到对应的处理列表中。同样不要忘记增加计数。

10. 在 **js/assembler.js** 中的几个常量数组中添加指令对应的条目；在 *insSyntax* 和 *insEncoder* 中分别添加指令的语法以及转换函数。由于其他函数与具体指令无关，所以不需要再做修改。以 IADDL 为例，有 $insSyntax['iaddl'] = ['V','rB']$ 和 $insEncoder['iaddl'] = 'c08' + this.rB + this.V$。

11. 在 **js/disassembler.js** 中做与上述类似的修改。以 IADDL 为例，有 $codeSyntax['iaddl'] = ['rA','rB','V']$ 和 $insDecoder['iaddl'] = this.ins + '\$0x' + this.V + ',' + this.rB$。

可以看出，在自定义新指令时，只要**将其每个阶段的操作分析出来**，修改流水线逻辑是很方便的。同时对汇编器和反汇编器的修改也由于**良好的封装和低耦合**而极为便捷。

由于时间原因，再加上这纯粹是重复性的机械劳动，我没有进一步添加更多的新指令。

6 Acknowledgements

在本次 Project 的设计和开发过程中，我从下列同学那里获得了许多帮助，在此向他们表示衷心的感谢：

- **丁　成：** 帮我解决了许多与前端界面相关的问题。
- **余一夫：** 在分页内存和 CPU 的实现过程中给我提供了宝贵的意见。
- **杨皓然：** 在反汇编器和自定义指令的实现过程中给我提供了宝贵的意见。

除此之外，关于缓存和汇编器的实现我参考了**维基百科**，同样在此表示诚挚的谢意。