Y86 Assembly IDE project report

周芯怡 17307130354
张作柏 17300240035

2018 年 12 月 2 日

目录

1 概览

1.1 开发环境

开发语言	JavaScript / HTML / CSS / Python
浏览器环境	Chrome / Firefox / Safari
第三方库	jQuery / Bootstrap

1.2 文件构成

本次Project的后端代码均在**backend**目录下，说明如下：

kernel.py	流水线控制逻辑	**others/cc_stat.py**	状态码与条件码类
stages/Fetch.py	流水线取指阶段	**others/constant.py**	常量的定义
stages/Decode.py	流水线译码阶段	**others/decoder.py**	预处理.yo文件
stages/Execute.py	流水线执行阶段	**others/Init.py**	处理.yo文件
stages/Memory.py	流水线访存阶段	**others/little_endian.py**	处理小端法
stages/WriteBack.py	流水线写回阶段	**others/Help.py**	Debug帮助文件
memory_sys/memory.py	内存的实现	**memory_sys/register.py**	系统寄存器的实现
memory_sys/piperegister.py	流水线寄存器的实现	**encoder/encoder.py**	汇编器

本次Project的前段代码均在**frontend**目录下，说明如下：

main.html	前端界面
arrow.png	前端素材
HappyKiller.ttf	前端字体
js/WriteCode.py	前后端交互代码部分
js/WriteDisplay.py	前后端交互显示框部分
js/WriteReg.py	前后端交互寄存器部分
js/WriteStack.py	前后端交互栈部分
js/WriteStage.py	前后端交互流水线寄存器部分
js/WriteStat.py	前后端交互流水线状态码部分

2 使用方法

详见Manual.pdf

3 设计

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。

3.1 动机

本次Project的基本要求实现一个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。图形界面的几个选择有各种语言自带的图形库、Web、Unity、uwp等等。最后我们选择了Python+Web的组合，主要原因如下：

1. 我们能够熟练运用的语言只有C++，略知一二的有Python。Python**易上手**，且未来的**使用需求较高**，所以这里我们选择了Python作为我们的后端语言。
2. 相比其他语言，Python有着**强大的库函数**，使用更便利，可以大大减少代码量。在不考虑效率的情况下，Python比C++更加友好，这尤其体现在后期代码量巨大的IDE设计上。
3. 由于C++和Python自带的图形界面并不算出彩，所以我们选择了**显示更美观**、**交互性更好**的Web作为前端。同时，Python支持的框架有很多，所以Python搭配Web是很好的选择。
4. Web是前端开发的基础，而Python更是当代程序员的必备技能。这两种工具的**学习资料较多**，且**未来用处更大**，不妨借此机会尝试一下。

在github上参考了许多学长学姐的project之后，我们开始思考如何才能做出自己的特点。一些使用Unity作为前端的学长的作品，有非常抓人眼球的**3D显示效果**，这是Web所做不到的。但与此同时，换来的也是使用上的不便捷。同时也有许多用Web作为前端的学长，他们的project往往风格十分朴素，前端**设计精简**，提供一些**基本的交互功能**，但仅此是不能做出我们自己的特色的。

再三考虑之后，我们打算做一个IDE。Web显示风格十分简洁，且不失美观，恰好符合作为调试工具的基本要求。而通过后端输入指令，更是符合程序员写代码的习惯。于是，我们打算开发一个简单的**书写Y86汇编代码的IDE**。

基于以上考虑，我们采用前端Web的形式，后端用Python实现所有的逻辑。

3.2 特点

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，并能在前端显示每个周期内流水线寄存器的数值。

在此基础上，我们还添加了许多新功能：

- 加入了**iaddq**操作。
- 加入了**加载转发**优化(见课本练习4.57)。
- 前端实时监视指定的栈、内存、寄存器中的值。
- 实现了**Y86汇编器**，从而能够接受.ys文件作为输入，也能保存汇编的结果。
- 支持动态输入汇编代码，实时执行的功能，**写代码**，**调试代码**二合一，实现简易的**IDE**。
- 模仿gdb，实现了步进、断点等**22条动态调试指令**，调试功能强大。

4 实现细节

本节将对IDE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。

4.1 流水线

在**backend/memory_sys**中，我们将内存、寄存器、系统寄存器封装成类，以便之后使用。封装和注释使合作完成代码的过程更加顺利，同时也使思路更清晰。

在**backend/stages**中，我们按照顺序实现了流水线的五个阶段。最开始的时候，没有加入转发逻辑与控制逻辑。每个stage完成后，独立调试，确保没有局部的细节错误。之后，在Decode.py中加入转发逻辑，在kernel.py中加入控制逻辑，五个阶段联调。

在**backend/others**中，添加读入的接口和后期所需要的类以及处理模块。因为这部分独立于整个流水线的设计，所以很适合分工来写，只需要声明好接口即可。

在实现流水线的过程中，我主要参考了书中的HCL代码。利用python的语法，可以写出与HCL相似的代码，降低了错误的概率，更加易于调试。五个阶段中，最简单的是$Memory$和$WriteBack$阶段，只用了1K的代码量就完成了。剩下的三个阶段比较繁琐：$Fetch$阶段需要考虑**指令的合法性**，**取出当前指令**，并**预测下一指令的地址**。$Decode$阶段需要处理**转发逻辑**，因为转发逻辑是后期加入的，且书上列出了HCL代码，所以难度并不大。$Execute$阶段实现**ALU**，设置**状态码**和**条件码**，加入**加载转发**优化。不过相比之下，控制逻辑的实现更加困难。

在**kernel.py**中，我们完成了CPU的**核心部分**，将五个阶段连接起来，并实现控制逻辑。五个阶段在执行的过程中，使用的均是上个周期中流水线寄存器的值，写入一个新的流水线寄存器中，当五个阶段都执行完后，才更新流水线寄存器。这样做的好处是，五个阶段**相互分离**，只需要使用上个周期的值，可以**假装并行**。但是由于流水线本身的设计问题，在加入转发逻辑后，$Decode$阶段是依赖于当前周期中$Execute$、$Memory$和$WriteBack$三个阶段的运算结果的，这导致当前的流水线设计并**不能**实现真正的并行。为了保证正确性，我们只好将五个阶段**倒序**执行，以处理结果依赖的问题。控制逻辑的实现全部在**kernel.py**中体现，在更新寄存器的时候分别检查是否出现了三种问题，并做相对应的处理。同时在其中实现更新PC值的相应操作。

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：

- 除去课本中规定的四种状态外，我们新增加了BUB和NON两种状态，分别表示*bubble*和在非指令集位置取指令。后者常会发生于代码结束，而没有遇到HLT，导致PC持续增加的情况下。
- 加入了**加载转发**优化，通过增加在$Execute$阶段的特判，减少*bubble*的产生，详见课本练习4.57。

4.2 汇编器

作为一个IDE的基本需求，自然要实现一个Y86汇编器以实现编译汇编代码的需求。在实现YAS之后，可以通过书写.ys文件来造样例，这是非常方便的。

比较严谨的实现可能要参考**编译原理**这门课中的做法，但是时间有限，Y86的汇编代码比较简单，所以凭自己的想象写了一个汇编器，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。由于我在读指令的时候是一行一行读的，所以暂时没办法处理跨行的注释。

2. 扫描并记录所有的**label**，同时记录相应的指令地址。处理**directives**，包括.pos、.align、.quad和.byte，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，根据相应的语法格式**提取参数**并编码。这一步中要进行大量的特判，以保证能够检验出所有的非法指令，通过及时报错，来避免程序中断的问题。

4. 拼接结果，或是处理检验到的异常。

整个YAS实现的细节都在**encoder/encoder.py**文件中，这是本PJ中代码量较大的一部分。

4.3 Debug功能

为了完善IDE，改善用户体验，我们加入了相当数量的调试指令。其中大部分功能均是参考gdb中支持的指令及其语法格式。

大部分指令的实现细节都在**manual.pdf**中提到，此处只大致概述。

1. *#Step*指令集　运行程序。把原来执行指令的循环改成每次只执行一步，通过设置步长和断点，把控结束的时间。其中比较难实现的是*next*和*finish*两个指令，因为要牵扯到判断在哪一个函数中。我们可以通过*call*和*ret*的数量，来判断当前的递归深度。需要注意的是，**不能**统计在取指阶段取出的*call*和*ret*的数量，因为流水线的控制逻辑可能会导致误取指令。而为了体现步进的特点，*step*和*continue*指令均是查看**取指阶段**中取出的指令，所以这里在使用的时候会有一些偏差。

2. *#Jump*指令集　跳转。最开始想通过向流水线中加入一条不存在的虚拟指令，以实现跳转操作。可是万恶的流水线控制逻辑有可能会将新加入的指令当做误取指令来处理！无奈，只好规定在跳的时候清空流水线寄存器中的信息，并**手动**执行*call*和*ret*的操作。

3. *#Breakpoints*指令集　设置断点。实现相对简单，只需要用一个*list*来记录断点，并在执行的时候判断是否抵达断点即可。因为要维护断点的多条信息，所以这里如果把断点封装成类可能会更好些。

4. *#Display*指令集　设置需要监视的值。和*#Breakpoints*的实现类似，用一个*list*记录*Display*的表达式，前后端交互时读取相应的值并发送给前端。实现最简单的一个指令集。

5. *#I\O*指令集　输入/输出代码。在实现了YAS之后，我们可以很容易地支持加载并输出.yo和.ys操作。除此之外，我们还实现了**动态输入**代码的功能，当输入一段指令时，使用YAS进行译码，并将新得到的指令码加入当前指令集中。

6. *#Others*指令集　其他功能。这里的几项功能逻辑都比较简单，不会遇到很大的问题。最麻烦的*help*指令，在参考了gdb的*help*文档后，改造完成，提升使用的**便捷性**。

4.4 Web界面

我们采用了HTML、CSS作为前端开发语言，同时借助了第三方库Bootstrap。利用这些强大的工具，我们得以比较轻松的实现较为美观的前端界面。具体的实现细节如下：

1. Stack显示区和Code显示区：利用Bootstrap提供的表格功能实现

2. 状态和寄存器显示区：原本想用Bootstrap提供的选项卡功能实现，但是因为没能搞懂链接操作失败了。最后利用radio表单，让被选中的单选框对应的内容展示，实现了不同选项卡的切换。但缺点是每一次换选项卡时页面都会重新刷新

3. 流水线寄存器和Display展示区：同样利用表格排版，同时用CSS设置style，把每一个显示区设置为卡片风格，并进行页面的总体布局和美化。

4.5 前后端交互

前后端交互主要利用了JavaScript。具体的实现是后端通过调用相应的WriteXXX函数把前端需要的数据整理成HTML格式，输入到.txt文件中。然后通过Load.js，每隔100ms把.txt文件中的内容加载到main.html文件对应的位置。这样就实现了前端的不断更新。

4.6 存储器结构

暂未完成。

5 测试

本节将叙述对模拟器的测试和性能评估。

5.1 测试样例

在**test**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
1	asum.yo/asumr.yo	测试样例，包括.quad、循环和调用
2	asumi.yo	测试iaddq操作
3	abs-asum-jmp/cmov.yo	测试条件码的设置
4	Load_Use.ys	测试模拟器是否能处理Load/Use Hazard
5	Comb_A.ys	测试模拟器是否能处理Combination A
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
7	List_Sum_R.ys	同上，但用递归完成，测试调用和返回等操作

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。

为了更好地检验Y86模拟器的正确性，我们构造了几组数据，收录与**test/new**目录下，说明如下：

5.2 性能分析

暂未完成。

#	测试文件	描述
1	ex32.yo	测试转发逻辑(见书4.32)
2	ex33.yo	测试转发逻辑(见书4.33)
3	load_forward.yo	测试加载转发优化(见书4.37)
4	overflow.yo	测试数据溢出
5	INS1~4.yo	测试非法指令

6 开发过程

本节将叙述Project的详细开发过程、遇到的问题和感想。

6.1 开发流程

从学长们那里得到的开发经验，如果两人明确分工一人做前端，一人做后端的话，工作量可能差异较大，并且两个人不能并行工作，效率很低。而且因为两个人都有必要熟悉流水线的逻辑，同时也需要掌握前端的开发技术，所以最后决定两人不做明确分工，在开发的过程中共同处理遇到的问题。

第一周(11.1~11.4) 开发过程中遇到的第一个问题是工具的选择。第一次做图形界面，面对许多陌生的开发工具，不知道其各自的优势与局限性在哪里，只好通过不断向学长和搜索引擎求助来得到比较好的答案。在决定组队之后，我们确定使用Python和Web作为开发工具，同时开始学习Python的使用和HTML、CSS、JS的基础知识。

第二周(11.5~11.11) 速成开发语言后，两人开始熟悉流水线细节，与此同时，实现简单的模块和类，以熟悉Python的使用。在第二周周五的时候，实现出流水线的第一版。自此开始分工，一人负责学习前后端交互的知识，另一人负责完成流水线的调试。

第三周(11.12~11.18) 有了做IDE的想法，发现需要在后端下比较大的功夫，决定尽早完成调试和前端界面的基本任务。开发过程中遇到的第二个问题是前后端交互方式的选择。尽管有许多现成的Web开发框架，但是在缺少网络相关知识的情况下，很难看懂相关的资料，并且调试的难度大大提高了。在学习了多种开发工具之后，最终决定放弃所有，使用最暴力的读写文件方式进行交互。完成流水线的调试，值后对前端页面进行初步布局，完成前后端的简单交互。某人周四晚上一时兴起实现了YAS。

第四周(11.19~11.25) 二人分工。一人负责美化前端，另一人负责参考学长们的Project以获取灵感(~~摸鱼~~)。周末基本完成前端设计，某人一时兴起实现了Debug功能。

第五周(11.26~12.2) 基本完工，摸鱼一周。期间处理过一些前端和后端的bug，周末进行联调，书写报告的工作。

新技能get

- Python

- HTML5/CSS3/JavaScript
- Flask/Django
- jQuery/Bootstrap
- JSON/ajax
- Github
- Latex

6.2 踩过的坑

- 课本中的溢出标志只考虑了正数的情况，所以不能直接引用。
- Python中的long类型是没有上界的，这导致整数相加时不会产生溢出，需要手动设置溢出。
- Python函数传参时，要注意对象是否可更改。
- Python的全局变量是指同一模块中的变量，对模块外的全局变量就没法用global声明了。
-

6.3 感想

6.3.1 张作柏

这是我的第一个Project，也是第一次和其他人合作做Project，的确是一次非常有趣的经历。

刚开始拿到PJ的时候，我是一头雾水的。虽然上课时老师讲的很清楚，但是在自己实现Y86的时候，却要考虑非常多的细节。但更让我头秃的是图形界面的设计，对于一个审美畸形的程序员来说，选择合适的工具，设计出一个美观的界面实在是太难了。幸好有几位非常nice的助教和学长指点，我才找到几条可以选择的路。说实话，本来我是没有组队的想法的，想自己干一票大的。但如果想要做出漂亮的PJ，要学的东西实在是太多了，自己一个人做太容易翻车。所以在我面前基本就只有两种选择，第一种是自己用C++和QT做，在熟练使用QT后，可能能做出一个质量还不错的PJ，第二种是找人组队，尝试一些完全没有接触过的技术，争取做出一个非常棒的PJ。因为分数的计算方式犹豫了一会，最后决定既然要做，那便做到最好，总要走出舒适区，尝试些全新的事物。之后嘛，就成功地抱到了队友的大腿。

确定了组队后，紧接而来的问题是如何分工。询问了之前学长们的经验，很多的队伍是一个人负责前端，另一个人负责后端，但是这就会导致两个人的并行效率非常差，而且一方出了问题另一方往往帮不上忙，那便失去了组队的意义。恰好我和队友都有学习Python和Web的需求，所以我们决定一起完成前端和后端。但确实分工是个大问题，这在后面写代码的时候也体现了出来，还好我的队友比较给力，解决了我代码里的各种问题，不过在以后更大工程的开发上，确实要仔细考虑分工的情况。

先说说各种工具的速成情况吧。在了解各种选择后，我便有了用Web做前端的想法，所以在第一个周末，就把HTML和CSS用一个晚上速成了一遍，大致了解这些工具的用途。HTML不同于我们以前写的C++，HTML的逻辑性不强，更多的是描述性，它更像是在描述网页中的每一个元素，这种语

言的学习对于熟悉了C++的我来说，是一个巨大的挑战。而Python的速成相对就简单很多，参考了一些简单的代码和学长的PJ后，Python的基本用法就算是掌握了。为了合作的效率，我还粗略地学习了一下GitHub的使用。不过最难学的还是前后端交互的框架，在不断碰壁之后，我还是选择采用最粗暴的文件读写来进行传值，在这个阶段浪费了大概一个周的时间。再之后就是各种第三方库的学习，不过在实际开发中我用到的也并不是很多。最后，也是最关键的，实验报告的书写。这是我第一次正式尝试用Latex写报告，排版确实美观，也学会了一些基本的操作。感觉学了这么多的东西，是一次相当棒的经历了!

再谈谈idea的部分。做IDE的想法是我最先提出的，之后我的队友给我了许多非常棒的建议。在看过很多学长的报告之后，我发现用Web做的Project套路都很单一，很难做出彩。我发现许多Project虽然外表华丽，但并不实用，所以在如何提高实用性上，我动了点脑筋。本想像Python支持一个动态执行代码的功能，相当酷炫，但因为流水线自身的原因和支持的指令集，并不支持这样做，只好退而求其次，加了个动态输入代码的功能。最后的Debug功能实在是突发奇想，一个IDE没有调试功能怎么行呢？于是就从gdb上扒了一些好实现的功能来做了。

本来应该是重头戏的写代码部分，我似乎没啥想说的。这个Project虽然代码量比之前写过的代码都要多一点，但是除流水线外，基本都是各种特判，流水线的逻辑在书上也写的很清楚了，这其中基本没有涉及任何算法，所以调试的难度反而不如一些代码量较小的算法题。当然，调试工作主要还是由我的队友来做的，所以我也没有太多的发言权。最大的一个bug是由我队友发现的Python不会溢出的问题，在此之后，就没有太多难题了。

最后谈谈感想吧。合作的时候能遇到一个好队友真的是很棒的事，省去了很多烦恼，而且有人陪着一起学习新东西比自己单干要更容易坚持下来。其实，对于做PJ而言，学到的东西远比做出来的东西重要得多。假如没有这次PJ，我可能压根没有机会接触到这些前端开发的内容。总之，遇到了一个好队友，学到了很多想学的东西，做出了自己还算满意的项目，~~希望这样的PJ还能再来几个~~!

6.3.2 周芯怡

7 致谢

- 感谢金城老师为我们解答流水线实现相关的问题。
- 感谢几位助教给出了技术支持和宝贵的建议。
- 感谢胡志峰学长提供他的PJ供我们参考。
- 感谢解润芃、王辰浩等几位同学提供了构造测试样例的思路。