Project 2: Tomasulo Simulation

Due：2020.5.19 TUE 00:00 (UTC +8)

（一）实验简介

本实验要求同学们使用 C/C++编写一个 Tomasulo 算法模拟器，使用软件模拟课堂上介绍的 Tomasulo 算法的流程。该模拟器的输入为一种被称为 NEL（Naïve Assemble Language）的简单汇编语言，输出为各条指令的发射、执行完成和写回结果对应的周期，并支持对于任意周期结束时各保留栈、LoadBuffer 以及寄存器状态的查询。

（二）实验要求

输入：

NEL 汇编语言测试样例（在 TestCase 文件夹下，关于 NEL 语言的严格定义和测试样例详情参见附录，只需测试 Basic 测试用例，Extend 和 Performance 测试用例用于测试扩展性）

要求根据 NEL 的语法，自己编写设计至少一个 NEL 测试用例，并给出模拟器在该用例上的表现，如果测试用例较为典型，可获得一定加分。

输出：

（1）**记录各条指令的发射周期，执行完成周期，写回结果周期**，并在模拟器运行完毕之后统一输出到 Log 文件中（按照“%d %d %d\n”的格式输出，第 i 行有三个整数，分别为第 i 条指令的发射周期，执行完成周期和写回结果周期），规定 Log 文件应当以“学号_测试用例.log”的方法命名（例如 20170xxxxx_0.basic.log）统一放在一个命名为 Log 的文件夹下。

（2）**能够支持查询模拟器执行过程中任意一个 Cycle 的瞬时状态**，该状态包括，LoadLoader 状态（Busy，Address），保留站的状态（Busy，Op，Vj，Vk，Qj，Qk），寄存器状态（正在等待的保留站名称或数值）。

扩展加分项（二选一，按照完成质量可获得 0-20 分的加分，总分不超过 100 分）：

（1）**实现 Jump 指令**。在实现 Jump 指令时可能会有很多潜在的问题，建议同学要认真阅读 Tips 部分和样例，熟悉相关细节，**并可尝试在该基础上，进行分支预测**，汇报分支预测成功的次数，以及对于整体周期的削减情况。

（2）**进行性能优化**，设计更为高效数据结构和模拟算法，并汇报在 Performance 类测试用例上所花费的时间（Performance 用例详情参见附录）

（三）实验步骤建议（建议仅做参考，达到实验要求即可）

各位可以依次执行如下各个步骤

（1）读入并解析 NEL 汇编语言，解析方法可直接使用字符串切割算法，按照”,”切割，明确下一条要执行的语句所需要使用的保留站，功能部件，所需要读取的寄存器等。

依次模拟各个周期的动作：

（2）根据保留站是否空闲，Jump 指令的目标地址（如果做了 Jump 扩展），分支预测结果（如果做了分支预测扩展）等情况，确定是否在当前周期是否有指令发射。

（3）根据该指令执行所需要读取的寄存器是否已经就位，功能部件是否空闲，确定在该周期是否有指令就绪并开始执行。

（4）判断当前周期是否有指令已经执行完毕，查看该指令的寄存器是否状态已经修改从而决定是否写回。

（三）检查说明

请各位提交压缩包中包括：

1、**实验源代码及编译方法**（如给出 Makefile 等）

2、**Log：**要求必须给出 Basic 文件的 Log，如果做了 Jump 扩展，要求给出 Extend 类文件的 Log。对于 Performance 类型文件的 Log 不要求输出，但要给出运行时间

3、**实验报告：**要求说明模拟器的设计思路，并简要分析 Tomasulo 算法的特点，讨论该算法和记分牌算法的差异性（鼓励通过设计新的 NEL 测例来辅助讨论）。如果完成了扩展，请说明扩展的的设计思路。如果调研了不同的分支预测技术，请分析不同分支预测方法的特点以及在相应 NEL 测试用例上的表现（如果给出的测试用例不足以体现出不同分支预测方法的差异，鼓励设计新的典型的 NEL 测例。（**请认真完成实验报告，过于简略将可能影响分数**）

当面检查：

在实验完成之后，我们可能会安排一次当面检查，当面检查以线上的方式开展，需要同学们现场给助教演示模拟器功能，并回答助教的问题。

（四）Tips

1. Tomasulo 算法涉及的细节很多，为了方便于同学们理解教程，更好地完成实验，专门为同学们准备了一个样例（样例对应于 example.nel 文件）来说明各类细节的处理方法，该样例仅供参考，如果同学们认为存在疏漏或者不合理的地方欢迎随时指出。

2. 因为存在循环，同一条指令可能会被多次执行，你只需要输出第一次执行该指令时的发射周期，执行完成周期和写回结果周期即可。
3. 可以参考开源模拟器的实现方案，也可以使用一些开源框架，但请在引用文献中说明。如果开源框架功能过于强大可能会影响到相应部分的实验评分。
4. 同学们需要注意的是在 Tomasulo 算法中，某条指令在第 k 个周期就绪（也就是它所有的源寄存器都写回了结果）他在第 k+1 个周期才开始执行，假设它需要执行 m 个周期，那么它将在 k+m 周期的末尾执行完成，并在 k+m+1 周期末尾写回，将就绪和开始执行混为一谈是错误的。
5. 只有在寄存器状态的目标寄存器和当前的目标寄存器一致的时候才可以更新寄存器数值，否则将会导致 WAW 之后，结果不一致的现象。

附录 1：NEL 汇编语言简介

NEL 是我们自定义的简单汇编语言，其文法定义如下：

1. 词法定义（正则表达式定义）

DIGIT = ([0-9])

HEX_DIGIT=([0-9A-F])

DEC_INTEGER=({DIGIT}+)

HEX_INTEGER=(0x{HEX_DIGIT}+)

INTEGER={HEX_INTEGER}

REGISTER=(F{DEC_INTEGER})

保留字：

“ADD”，“MUL”，“SUB”，“DIV”，“LD”，“JUMP”

2.语法定义（CFG 语法定义）

Program := InstList

InstList := Inst

InstList := Inst ‘\n’ InstList

Inst := OPR ‘,’ REGISTER ‘,’ REGISTER ‘,’ REGISTER

Inst := “LD” ‘,’ REGISTER ‘,’ INTEGER

Inst := “JUMP” ‘,’ INTEGER ‘,’ REGISTER ‘,’ INTEGER

OPR := “ADD” | “MUL” | “SUB” | “DIV”

3.说明

1、NEL 中的整数统一按照带符号 32 位补码的形式存储，表示一律使用十六进制表示（0x 开头，字母位大写）

2、NEL 是一种由连续三地址码指令组成的汇编语言，它只支持三类指令，分别是运算指令（ADD，SUB，MUL，DIV）、装载指令（LD）和控制指令（Jump）。它没有内存模型，所有的运算都首先通过使用 Load 将数字读入寄存器，经过寄存器之间的各类运算获得结果，此外还可以通过 Jump 来实现控制。

3、基本运算指令语义：NEL 的运算指令都可以表示为 OPR1 ‘,’ REGISTER ‘,’ REGISTER ‘,’ REGISTER （例如 ADD,R1,R2,R3）的形式，上述例子的语义为针对 R2 和 R3 寄存器中的数值进行 ADD 运算，将结果存入 R1 中，其他指令的含义以此类推，（除法指令均表示整除，特别的，如果除数位置寄存器数值为零，则将被除数存入目标寄存器，相当于除 1；在本模拟器中默认除法指令如果发现除数为 0，那么它就只需要执行 1 个周期；此外除法取整按照 C++取整标准）

4、装载指令语义：NEL 的装载指令都可以表示为“LD” ‘,’ REGISTER ‘,’ INTEGER 的形式 (如 LD,R1,0xFFFF)，上述例子的语义为将 0x0000FFFF 装载到 R1 中。

5、跳转指令语义：NEL 的跳转指令都可以表示为“JUMP” ‘,’ INTEGER ‘,’ REGISTER ‘,’ INTEGER 的形式（如 JUMP,0,R1,0xFFFFFFFD），上述例子的语义为如果 R1 寄存器中的数值为 0 的话（严格等于零），那么将向前从该指令开始向前数 3 句执行（比如当前指令编号为 3，那么将从编号为 0 的指令开始执行），默认 JUMP 指令和 ADD，SUB 一样使用 Add1 等加减法其完成，使用 Ars1 等加减法保留站来记录结果，JUMP 发射之后，执行需要一个周期，如果不做任何分支预测，只有在写回的周期时才可以发射下一条目标指令。

6、寄存器栈模型：NEL 支持无限个寄存器，每一个寄存器都是 32 位的，寄存器编号按十进制编码，分别为 R0,R1,R2….，但是在测试用例中只需要考虑 32 个寄存器即可（R0-R31），默认所有寄存器的初始数值均为 0。

对应的各个 NEL 测试样例及说明如下：

测例名称	行数	包含指令	种类
0.basic.nel	6	LD, ADD, MUL, SUB, DIV	Basic
1.basic.nel	34	LD, SUB, ADD	Basic
2.basic.nel	27	LD, ADD, MUL, SUB, DIV	Basic
3.basic.nel	28	LD, ADD, MUL, SUB, DIV	Basic
4.basic.nel	24	LD, ADD, MUL, SUB, DIV	Basic
Fact.nel	10	LD, ADD, MUL, SUB, JUMP	Extend
Fabo.nel	14	LD, ADD, SUB, JUMP	Extend
Example.nel	8	LD, SUB, MUL, JUMP	Extend
Gcd.nel	21	LD, ADD, MUL, SUB, DIV, JUMP	Extend
Mul.nel	1018	LD, MUL	Performance
Big_test.nel	1000000	LD	Performance

附录 2：假设硬件平台情况：

我们假设有一个硬件平台包含如下功能部件

3 个加减法器（Add1, Add2, Add3）

2 个乘除法器（Mult1, Mult2）

2 个 Load 部件（Load）

同时我们假设有如下保留站：

6 个加减法保留站（Ars1, Ars2, …）

3 个乘除法保留站（Mrs1, Mrs2, …）

3 个 LoadBuffer（LB1, LB2, …）

注意：

在本硬件平台中，保留站的数量要多于功能部件的数量，因此保留站中就绪的指令将构成一个先进先出队列，每当功能部件出现空闲时，取最先就绪的指令进入功能部件执行，如果有两条指令同时就绪，那么就按照它们编号（即是指令序列中的第几条指令）大小排序，编号比较小的指令默认排在更靠前的位置。

附录 3：各类指令的假定运行时间：

LD 指令需要 3 个周期完成；JUMP 指令需要 1 个周期完成；ADD 和 SUB 指令需要 3 个周期完成；MUL 指令需要 4 个周期完成；DIV 指令需要 4 个周期完成，如果遇到除 0 的情况，则只需要 1 个周期；