系统结构大作业——Tomasulo算法模拟器

王艺涵 计65 2016011365

实验完成内容

基础要求实现

1. 能够正确接受任意 NEL 汇编语言编写的指令序列作为输入
2. 能够输出每一条指令发射的时间周期，执行完成的时间周期，写回结果的时间周期
3. 能够输出各时间周期的寄存器数值
4. 能够输出各时间周期保留站状态、LoadBuffer 状态和寄存器结果状态

扩展要求实现

1. 分支预测实现
2. 丰富 NEL 语言

实验设计与实现

实验设计

基础实验要求设计

程序中定义了Simulator类来定义模拟器的状态和行为。

---

resultStatus 记录每个寄存器的状态和寄存器值

reserveStation 记录每个缓存站的状态

FU 记录每个功能单元的状态

loadBuffer 记录每个内存工作单元的状态

---

tick()

每个时钟周期运行一次

1. FU运行一个周期并检查是否有已经准备好写回结果的FU
2. 尝试issue下一条指令
3. 遍历所有保留站，检查是否有指令可以就绪或是放入FU中开始执行

readInst()

尝试issue每个指令时进行调用，返回是否issue成功

1. 检查是否有reservation station剩余
2. 检查需要的结果是否已经预备好，如果没有预备好，注册一个等待

3. 标记自己的result status

实验中定义了parser.py来处理对*.NEL文件的解析，将每行指令解析为一个列表作为simulator的输入

分支预测设计

Tomasulo算法的分支预测可以使用前瞻执行来实现，基于Tomasulo算法的前瞻执行只需要在除result status之外增加一个相同的字段buffer作为缓冲寄存器。

当第一次遇见jump时，标记模拟器进入前瞻执行模式，如果在前瞻执行模式中再次遇见jump，issue失败并进行等待。

在非前瞻执行状态中，写回寄存器写入result status。

进入前瞻执行状态时，将result status缓存在buffer，然后拷贝一份作为result status。

接下来的执行过程中，如果FU指向的Reg位于Buffer中，那么两边都写回（说明是在跳转之前issue的指令）。如果位于result status中，那么只写入到result status。

当跳转运算完成之后，如果不进行跳转，那么直接使用result status继续执行即可，并将指针拷贝为result status对应指针。退出前瞻执行模式。如果进行跳转，将result status置为Buffer，并从跳转位置开始issue执行。

如果进行跳转，当退出前瞻执行模式之后，在JUMP指令issue之后已经被前瞻执行但是还没有写回的指令仍然在执行，但是由于他指向的result status已经被丢弃，因此他们的写回也不会影响正常执行的结果。

上述设计实际上实现的是**对分支预测技术机制上的支持**（即支持了两个寄存器的同时执行，并支持在跳转结果最终确定时确定真实应该被执行得到的寄存器数值）

在此分支预测机制的支持下，可以实现不同的分支预测策略，代码中实现的版本为预测分支不进行跳转。而如果添加一个BHT表，即可很方便的实现不同位数下的BHT分支预测。

NEL指令扩展

对parser进行扩展就可以实现对更多NEL指令的支持，Tomasulo模拟器的实现可以很方便的进行指令集扩展，其中32号寄存器被指定为编译器用寄存器

我的实验实现中增加的指令支持有

ADDONE F1, F2 被解析为 LD F32, 0x1 ADD F1, F2, F32

ADDI F1, F2, I 被解析为 LD F32, I ADD F1, F2, F32

测试结果分析——以test0.NEL为例

不使用分支预测的结果

指令运行记录（与例子相同）

Inst	Issue	Exec Complete	Write Back
LD,F1,0x2	1	4	5
LD,F2,0x1	2	5	6
LD,F3,0xFFFFFFFF	3	7	8
SUB,F1,F1,F2	4	9	10
DIV,F4,F3,F1	5	14	15
JUMP,0x0,F1,0x2	6	11	12
JUMP,0xFFFFFFFF,F3,0xFFFFFFFD	12	13	14
MUL,F3,F1,F4	20	24	25

执行结束之后寄存器结果数值（与例子相同）

1	2	3	4	5	6	7	8
0	1	0	-1	0	0	0	0
9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0
17	18	19	20	21	22	23	24
0	0	0	0	0	0	0	0
25	26	27	28	29	30	31	32
0	0	0	0	0	0	0	0

使用分支预测的结果

指令运行记录，可以看到指令MUL,F3,F1,F4在第二条jump指令发射之后没有等待，直接发射进行前瞻执行。

Inst	Issue	Exec Complete	Write Back
LD,F1,0x2	1	4	5
LD,F2,0x1	2	5	6
LD,F3,0xFFFFFFFF	3	7	8
SUB,F1,F1,F2	4	9	10
DIV,F4,F3,F1	5	14	15
JUMP,0x0,F1,0x2	6	11	12
JUMP,0xFFFFFFFF,F3,0xFFFFFFFD	12	13	14
MUL,F3,F1,F4	13	19	20

运行结束之后寄存器数值如下图，可以看到虽然进行了前瞻执行，并且预测失败（实际上第二条JUMP指令进行了跳转）但是并没有影响程序运行的最终结果。

0	1	2	3	4	5	6	7
0	1	0	-1	0	0	0	0
8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0
16	17	18	19	20	21	22	23
0	0	0	0	0	0	0	0
24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0

NEL扩展测试

此处只使用一些单句指令对扩展的NEL语句进行测试

```
LD,F1,0x1
ADDONE,F2,F1
ADDI,F3,F2,0x3
```

1	2	3	4	5	6	7	8
1	2	5	0	0	0	0	0
9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0
17	18	19	20	21	22	23	24
0	0	0	0	0	0	0	0
25	26	27	28	29	30	31	32
0	0	0	0	0	0	0	3

可以看到和实际结果相同

实验代码组织和运行方式

代码位于src文件夹中，是一个由git维护的仓库。master分支为不支持分支预测的模拟器。branch_done分支支持分支预测。

程序入口位于main.py。直接`python3 main.py`即可运行。

```
sim = Simulator()
inst, inst_list= parse("test.NEL") # 指定读入指令文件名，进行解析，文件位于当前目录下
sim.runInstr(inst_list, inst) # 初始化模拟器

for i in range(30): # 指定运行周期
    sim.tick()

sim.printTable() # 输出运行指令记录
```

每一步运行都会输出当前reg，reserveStation状态和内容

额外的测试

test1.NEL运行寄存器结果

```
LD,F1,0x3
LD,F2,0x0
LD,F3,0xFFFFFFFF
ADD,F2,F1,F2
MUL,F4,F1,F3
DIV,F2,F3,F1
SUB,F4,F2,F1
JUMP,0x0,F1,0xFFFFFFFE
```

```
1       2       3       4       5       6       7       8
3       0       -1      -3      0       0       0       0
9       10      11      12      13      14      15      16
0       0       0       0       0       0       0       0
17      18      19      20      21      22      23      24
0       0       0       0       0       0       0       0
25      26      27      28      29      30      31      32
0       0       0       0       0       0       0       0
```

test2.NEL运行寄存器结果

```
LD,F1,0x3
LD,F2,0x0
LD,F3,0xFFFFFFFF
ADD,F2,F1,F2
MUL,F4,F1,F3
DIV,F2,F3,F1
SUB,F4,F2,F1
JUMP,0x0,F1,0xFFFFFFFE
```

1	2	3	4	5	6	7	8
11	-50065	-6	36	50101	0	0	0
9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0
17	18	19	20	21	22	23	24
0	1	0	0	0	0	0	0
25	26	27	28	29	30	31	32
0	0	0	0	0	0	0	0

均和预期结果相同

自定义测试

该程序的语义为循环计算1 + 2 + 3 + 4并存储在F4中

```
LD,F1,0x5
LD,F2,0x1
LD,F3,0xFFFFFFFF
SUB,F1,F1,F2
ADD,F4,F4,F1
JUMP,0x0,F1,0x2
JUMP,0xFFFFFFFF,F3,0xFFFFFFFD
MUL,F3,F1,F4

```

运行完成之后寄存器结果如下图，可以看到满足预期结果

```
1   2   3   4   5   6   7   8
0   1   0   10  0   0   0   0
9   10  11  12  13  14  15  16
0   0   0   0   0   0   0   0
17  18  19  20  21  22  23  24
0   0   0   0   0   0   0   0
25  26  27  28  29  30  31  32
0   0   0   0   0   0   0   0
```