Tomasulo 调度算法实验报告

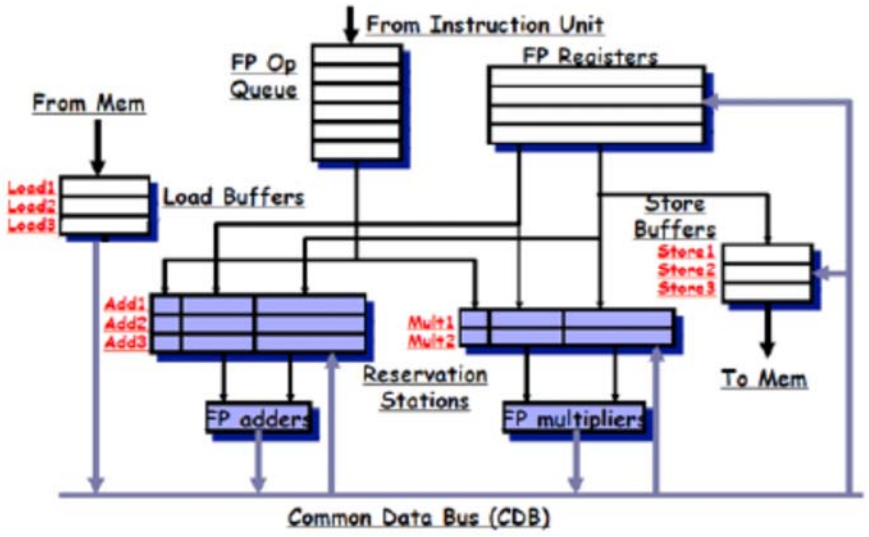
计 44 冯瑜林 2014011365
计 44 王奥丞 2014011367
计 44 武伟轩 2014011375

一、实验原理

Tomasulo 算法以硬件方式实现了寄存器重命名，允许指令乱序执行，以此来提高流水线的吞吐率和效率。

这里浮点处理的部件结构如下：点处理部件从取指单元接收指令，存入浮点操作队列。浮点操作队列每拍最多发射 1 条指令给浮点加法器或浮点乘除法器。浮点处理部件包含一个浮点加法器和一个浮点乘除法器。浮点加法器为两段流水线，输入端有三个保留站 add0、add1、add2，浮点乘除法器为六段流水线，输入端有两个保留站 mult0，mult1。当任意一个保留站中的两个源操作数到齐后，如果对应的操作部件空闲，可以把两个操作数立即送到浮点操作部件中执行。Load Buffer 和 Store Buffer 各缓存三条访存操作，可以将其也看作保留站，设定的编号分别为 ld0、ld1、ld2 与 st0、st1、st2。

其具体结构如下图所示：



二、算法实现

算法部分基于 python 实现，具体的实现思路大致如下：

首先创建三个类：Mem、Reg 与 Station，分别用来表示内存、寄存器与保留站

其中 Mem 类对象存储所有的内存值(共 4096 个 32 位 double 值，可以更改大小)，并创建一个全局的 Mem 类实例 mem，用来代表整个内存。

Reg 类对象存储所有的寄存器(共 11 个，可以更改大小)，同时创建一个全局的 Reg 类实例 reg，用来代表整个寄存器堆。

Station 为保留站类，这里总共有四类保留站：load、store、adder、mult，每一类保留站都是一个单独的 Station 对象。

1. Mem 类构造

在 Mem 类中封装了一些接口函数，可以设定与获得所有内存的值，该类比较简单，要完成的功能也很少，代码如下：

```
# coding: utf-8

# memory.py

class Mem(object):
    """docstring for Memory"""
    def __init__(self, size=4096):
        super(Mem, self).__init__()

        # memory size
        self.size = size

        # init memory as zero
        self.arr  = [0.0 for x in range(size)]

    def set(self, i, con):
        self.arr[i] = con

    def setAll(self, data):
        for x in range(self.size):
            self.arr[x] = data[x]

    def get(self, i):
```

```
        return self.arr[i]

    def getAll(self):
        return self.arr[:]

# global memory variable
mem = Mem()
```

2. Reg 类构造

在 Reg 类中同样封装了一些接口函数，可以设定与获得所有寄存器的值，不过不同之处在于一个寄存器可能保存一个 double 值，也可能指向一个保留站，这里由于 python 是弱类型语言，并且两者必居其一，故每个寄存器都只用一个变量来表示，该变量要么为某个保留站的名字(str 类型，如"ld0")，要么为某个具体的值(double 类型，如 1.4). 整个寄存器为一个由上述变量组成的列表。

这里在修改寄存器的值时，定义了两个接口函数：identify 与 update，前者用来将一个寄存器标记为某个保留站的指针，后者用来设定寄存器的 value。

代码如下：

```
# coding: utf-8

class Reg(object):
    """Register class
    the register format:
        | content/station identifier |
    """
    def __init__(self, size=11):
        super(Reg, self).__init__()
        self.size = size

        # init registers
        self.arr  = [0.0 for x in range(self.size)]

    def identify(self, i, name):
```

```
        # print("identify register %d as %s"%(i, name))
        self.arr[i] = name

    def update(self, i, con):
        # print("update register %d as %f"%(i, con))
        self.arr[i] = con

    def get(self, i):
        return self.arr[i]

    def getAll(self):
        return self.arr[:]

    def print(self):
        print("All registers:")
        for x in range(self.size):
            print("reg F%d:"%(x), self.arr[x])
        print()

# global register variable
reg = Reg()
```

这里 Reg 类的代码比较简单，不再赘述。

3. Station 类构造

下面介绍 Station 类，该类将四类保留站的公共部分提取出来，从而使得可以直接生成四个 Station 类的实例，每个实例分别代表一类保留站，根据保留站的不同来实现相应的逻辑。

一个 Station 中有多个保留站，其所需要完成的工作大致有以下几部分：

添加一条指令(add)、
选择一条指令(choose)、
总线广播后检查所有保留站并更新(update)

另外还有一些需要的接口：

检查站中指令是否都执行完(empty)
检查是否有冲突的内存(checkMem，之后会介绍该部分)
重置某个保留站(reset)
得到当前所有保留站的信息(getAll)

下面介绍 Station 类的具体实现：

Station 有三个成员变量：name、size 与 entry

name 是该类保留站的名称，4 类保留站的 name 分别为 ld、st、add、mult

size 是该类保留站中保留站的数目，Load 与 Store 的都为 3，Adder 的 size 为 3，Mult 的 size 为 2

entry 为一个 dict，其中的每一项代表一个保留站，key 为保留站的名字，value 为一个列表，代表保留站的内容。列表的长度即为 size 的大小。每个保留站都是一个 python 列表(list)，其长度为 5，格式如下：

[-1/dest, source1, source2, add/sub(True/False), inst_num]

含义如下(从左到右)：

目的寄存器(如果该保留站没有指令则该处为-1)
源寄存器 1 (值或者保留站标识)
源寄存器 2 (同上)
指令为 add 异或 sub (mult 异或 div)，若是加/乘则该值为 True，若为减/除则该值为 False
该指令的编号

可以看到，对于 LD 与 ST 指令，entry 中的一些地方是不会用到的，不过为了方便，对所有类型的保留站采用相同的格式，而解析时则根据保留站的类型来解析。

下面分别就每个具体的步骤分析代码：

初始化：

在创建一个 Station 对象时，在构造函数中会初始化所有的保留站，将其首位设置为 -1，表示该保留站可用(没有指令)。

代码如下：

```
def __init__(self, name, size):
    super(Station, self).__init__()
    self.name = name
    self.size = size
    self.entry = {"%s%d"%(self.name, x):[-1, -1, -1, True,
-1] for x in range(self.size)}
```

添加指令：

在 add 一条指令时，先查找是否有可用的保留站，如果不能找到返回错误，找到后根据参数，判断当前保留站是哪种类型，然后相应地设置保留站中的指令。

这里只需要注意：

load、adder、mult 保留站在添加指令时，需要将目的寄存器标识为该保留站的名称

store、adder、mult 保留站在添加指令时，需要将源寄存器(1 个或 2 个) 的内容取出放到保留站中，无论该内容是数值还是保留站指针。

adder、mult 保留站还需要注意判断指令是加还是减、是乘还是除，并据此设置保留站的第 4 位

代码如下：

```
def add(self, inst_num, arg1, arg2, arg3, types=True):
    """add an instruction to the reservation station. if
the station is full, return False"""

    choose = None
    for item in self.entry:
        if self.entry[item][0] == -1:
            choose = item
            break

    if choose == None:
        return False

    if arg3 == "load":
        # LD
        self.entry[choose][0] = arg1
```

```
            reg.identify(arg1, choose)
            self.entry[choose][1] = arg2
        elif arg3 == "store":
            # ST
            self.entry[choose][0] = reg.get(arg1)
            self.entry[choose][1] = arg2
        else:
            # ADDD | SUBD | MULT | DIVD
            self.entry[choose][0] = arg1
            reg.identify(arg1, choose)
            self.entry[choose][1] = reg.get(arg2)
            self.entry[choose][2] = reg.get(arg3)
            self.entry[choose][3] = types

        self.entry[choose][4] = inst_num
        return True
```

选择指令：

在 choose 函数中选择一条指令时，需要找到一条含有指令且指令中寄存器都准备好的保留站。判断指令是否准备好，只需要判断是否指令中使用到的所有寄存器是否都不是 str 类型(不是某个保留站的标识)。

另外，需要注意的一个地方在于应该在所有可用的指令中找到一个编号(inst_num)最小的指令，这样做主要是为了避免出现对同一块内存的读写发生冲突。下面会对解决内存冲突的方法进行详细描述。这里不赘述。

代码如下：

```
def choose(self):
    """choose an usable instruction"""

    ret = (False, None)
    min_inst_num = 0xffffffff
    for (name, inst) in self.entry.items():
        if inst[0] != -1:
```

```
            flag = True
            for addr in inst:
                if type(addr) == str:
                    flag = False
                    break

            if flag and inst[4] < min_inst_num:
                ret = (name, inst[:])
                min_inst_num = inst[4]
    return ret
```

更新保留站标识：

在 update 函数中传入某个保留站的标识(名字) name 及其中指令执行的结果 value，然后在该 Station 中，检查所有寄存器，如果其值为 name 则将更新为 value。

更新函数实现很简单，代码如下：

```
def update(self, name, value):
    """update the item identifiers"""

    for item in self.entry:
        for x in range(3):
            if self.entry[item][x] == name:
                    self.entry[item][x] = value
```

检查内存冲突：

所谓内存冲突指的是多条 LD 或者 ST 指令都要修改同一块内存，由于是乱序执行，可能会出现后面的指令先执行、前面的指令后执行的情况，这样可能会得到错误的结果。例如下面几条指令：

```
ST F1 0x9
ST F3 0x9
LD F7 0x9
```

如果第二条指令先于第一条指令执行，则最终 0x9 处的内存得到错误的结果，LD 指令也得到错误的结果；如果第三条指令先于前两条指令执行，则 LD 指令会得到未 store 的错误的结果。

要解决这个问题，需要满足两点：

1. 如果同时有多条可执行的 ST 指令，先执行编号最小的指令
2. 挑选出一条 LD 或者 ST 指令后，需要检查所有的 load 与 store 保留站，如果有指令内存地址与该指令相同且编号小于该指令，则推迟执行该指令。

上面的 choose 函数中已经满足了第一个条件，下面实现一个名为 checkMem 的函数，检查第二个条件是否满足，代码如下：

```
def checkMem(self, mem_addr, inst_num):
    """check the store/load station mem_addr and judge
whether an LD/ST instruction can be executed now or not"""

    for (name, inst) in self.entry.items():
        if inst[0] != -1 and inst[1] == mem_addr and
inst[4] < inst_num:
            return False

        return True
```

每次在挑选出一条 LD 或者 ST 指令后，都需要在 load 与 store 保留站中调用 checkSum 函数检查内存是否冲突，冲突则暂缓，否则执行。

其余函数：

其余的一些函数 (如 reset、getAll 等) 不是很重要，这里不再赘述。

4. 整体算法实现

这里已经介绍了 Mem、Reg、Station 几个类的实现，下面介绍整个的算法调度流程：

代码中将当前内存、寄存器、保留站及执行的状态都设置为全局变量，并定义了若干个函数接口供前端调用，这些函数会调用前面所介绍的一些函数，来修改当前的一些状态。

主要定义了以下函数接口：

init：初始化所有状态，并设置指令队列

step：执行一个周期

setAllMem：设置所有的内存

getStates：获取当前每条指令执行的情况(第几个周期发射、第几个周期执行、第几个周期写回)

getAllReg：获取所有的寄存器值或者保留站标识

getLoadQueue：获取 load 保留站中的内容

getStoreQueue：获取 store 保留站中的内容

getReservation：获取 adder 及 mult 保留站中的内容

这里的重点在于 step 函数，下面着重介绍这个函数：

step 函数的作用在于执行一个周期，其中定义了几个内部函数(由于未在外部使用，为了代码清晰易读，写成了内部函数)：update、reset、execute 与 exeDone。下面会分别介绍

在每个周期开始时，会先尝试将当前的指令发射出去，如果可以发射(保留站中有空余位置)则将当前指令从指令队列中剔除(当前的 inst_num 加一)。

之后检查每类保留站，是否当前正在执行某条指令，若正在执行，判断现在是否执行完及是否该写回结果，若执行完，则判断是否是 ST 指令，不是的话设定下一个周期写回结果，若是则 reset 保留站；若该写回结果，则 update 寄存器的值并 reset 保留站。

若当前并不在执行指令，则 choose 一条指令，如果能找到可执行的指令，则放入运算器中执行。

这里大概的逻辑如上，具体每类保留站中执行指令的状态是用一个名为 stations 的全局变量来存储的，其中存储了每个保留站执行的指令及当前的状态。

执行状态是用一个 num 变量表示，每条 MULT 指令发射时 num 设置为 10、DIVD 指令发射时 num 设置为 40、其余指令发射时设置为 2。每个周期 num 值减一，减到 1 时说明该指令执行完毕，此时调用 execute 内部函数，减到 0 时说明该指令应该被写回，此时写回寄存器并 reset 保留站。

大致的实现如上面所述，详细的细节不再赘述(细节地方需要考虑的比较多，但逻辑大概如上)。

谢谢助教的阅读，代码中关键处都做了注释，上述解析不十分清楚的地方请参看注释。谢谢！