Tomasulo算法实现

（因为页数的10页限制 删去了很多内容 代码部分的说明就不放在实验报告里了 可以看源代码文件的注释）

Tomasulo算法是一种通过动态指令调度、寄存器重命名和**保留站**机制解决指令间数据依赖问题并实现 **乱序执**行的技术，从而有效提升指**令流**水**线的效率**和处理器的整体性能。在此基础上，带有**Reorder Buffer（ROB）的Tomasulo算法引**入**了指令顺序提交机制**，以确保指令执行结果按照程序顺序正确更新寄存器和内存。**双发射超标量Tomasulo算法**进一步扩展了这一框架，通过**每个时钟周期同时发射两条指令**，显著提高了指令的**并行度**，增强了处理器的吞吐量和执行效率，同时仍维持指令提交的顺序 一致性，确保程序的正确执行。

任务1 单发射处理器的模拟

考虑在一个单发射处理器上执行指令片段，请分别在不支持推测执行和支持推测执行的情况下实现Tomasulo算法。假设功能单元有延迟：**浮点数加法为2个时钟周期，浮点数乘法为6个时钟周期，浮点数除法为12个时钟周期。**

```
fld f6,32(x2)
fld f2,44(x3)
fmul.d f0,f2,f4
fsub.d f8,f2,f6
fdiv.d f0,f0,f6
fadd.d f6,f8,f2
```

这里定义的程序指令因为没有分支指令 所以先不考虑推测执行和不支持推测执行的情况 放在任务2考虑

实验过程（预先定义的数据结构）

1.规定指令所需时钟周期数

首先按照题目要求定义好每个类型的指令所需的时钟周期数 这里没有对存数取数的时钟周期数做规定，所以设置为2

```
EX_Cycles = {"fld": 2, "fsd": 2,"fsub.d": 2, "fadd.d": 2, "fmul.d": 6, "fdiv.d": 12}
```

2.指令读取 push 以及定义指令类

观察给出的指令格式，可知可以把 **,** 与 **(** 都统一为空格处理 然后用空格作为分隔符

然后定义Instruction_queue的列表用来**模拟FIFO队列**存指令，然后按照定义的指令类所需的参数instruction, op, j, k, dest来初始化每条指令，加入Instruction_queue列表，这里每条指令初始构造都让time为None

3.指令执行 pop方法

借用指针cur模拟出队列的过程

4.定义ReorderBuffer类 用于保证指令顺序提交机制

ReorderBuffer类最后存储结构如图

只有指令提交commit了 才会把这个指令条目从这个表中删去

Reorder buffer						
Entry	Busy	Instruction		State	Destination	Value
1	No	fld	f6,32(x2)	Commit	f6	Mem[32 + Regs[x2]]
2	No	fld	f2,44(x3)	Commit	f2	Mem[44 + Regs[x3]]
3	Yes	fmul.d	f0,f2,f4	Write result	f0	#2 × Regs[f4]
4	Yes	fsub.d	f8,f2,f6	Write result	f8	#2 – #1
5	Yes	fdiv.d	f0,f0,f6	Execute	f0	
6	Yes	fadd.d	f6,f8,f2	Write result	f6	#4 + #2

5.定义保留站ReservationStation 用于解决数据冒险

为解决多指令流水线执行方式而导致的**数据冒险**而引入的一种数据结构,用于保存等待执行的指令的相关信息

Reservation stations							
Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
Load1	No						
Load2	No						
Add1	No						
Add2	No						
Add3	No						
Mult1	No	fmul.d	Mem[44 + Regs[x3]]	Regs[f4]			#3
Mult2	Yes	fdiv.d		Mem[32 + Regs[x2]]	#3		#5

6.定义寄存器状态

寄存器数据结构如图 这里定义了11个寄存器以供使用

FP register status										
Field	f0	f1	f2	f3	f4	f5	f6	f7	f8	f10
Reorder #	3						6		4	5
Busy	Yes	No	No	No	No	No	Yes	…	Yes	Yes

7.主函数控制时钟

注意这里调用各阶段方法的顺序和指令执行的阶段顺序正好相反，用于避免不同阶段的指令相互干扰

```
update = update | self.Commit()
update = update | self.Write()
update = update | self.Execute()
update = update | self.Issue()
update = update | self.Decode()
```

在主函数中一直循环clock+1 ,writeprocess

实验核心 Tomasulo算法的实现

这里将一条指令的执行阶段**分为Decode ->Issue->Execute->Wirte->Commit五个阶段**

Decode（译码阶段）：此阶段主要负责从指令队列（`instruction_queue`）中取出指令，并将其放置到重排序缓冲区（`reorderbufferentry`）的合适位置，同时进行一些相关属性的初始化设置。

Issue（发射阶段）：该阶段的核心任务是从已经译码完成（在重排序缓冲区中时间标记为`"Decode"`）的指令里，挑选合适的指令发射到对应的保留站（`reservationstation`）中，前提是相应类型的保留站有可用资源，并且要处理好操作数的准备情况以及相关寄存器状态的更新等问题。

Execute（执行阶段）：负责在保留站中执行已经准备好（`ready == 1` 且 `busy == "Yes"`）的指令，按时钟周期推进指令的执行进度，记录指令执行完成的时钟周期等相关信息。

Write（写回阶段）：当指令在保留站中执行完毕（执行周期耗尽且准备好等条件满足）后，此阶段把执行结果写回到重排序缓冲区，并处理相关的依赖关系，比如如果有其他保留站中的指令在等待该结果，要将结果传递过去并更新相应的准备状态等。

Commit（提交阶段）：主要负责将已经完成写回结果且按顺序轮到的指令进行最终的提交操作，释放相关资源，比如重排序缓冲区、保留站以及寄存器等资源，完成整个指令处理流程的收尾工作。

任务2：双发射处理器的模拟

```
1 Loop:    ld x2,0(x1)        //x2 = 数组元素
2          addi x2,x2,1       //递增 x2
3          sd x2,0(x1)        //存储结果
4          addi x1,x1,8       //递增指针
5          bne x2,x3,Loop     //如果不是最后一个元素，则跳转
```

不支持推测执行的情况下，跟在bne指令后面的ld指令不能提前开始执行，必须等待分支结果 的判断。在支持推测执行的情况下，跟在bne指令后面的ld指令可以提前开始执行。

代码展示

主要展示与单发射代码的不同之处 也主要在五个阶段有所不同

1.Decode

因为是双发射，所以需要用两次循环执行decode过程 但每次执行也需要检查是否有nullbuffer可用

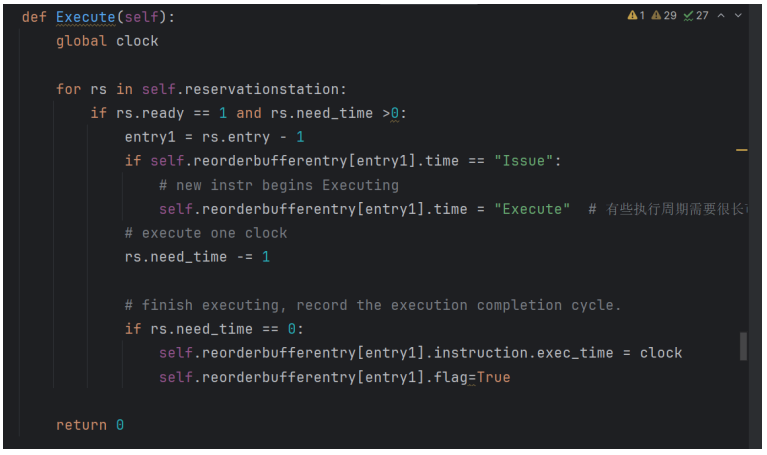
```
1 个用法
def Decode(self):
    # 把instruction_queue里的指令放入nullbuffer中
    nullbuffer = [buffer.entry for buffer in self.reorderbufferentry if buffer.busy == "No"]
    if not nullbuffer or success():
        return 0
    for i in range(2):
        if nullbuffer and not success():
            # put next instruction into ready entry in the entry buffer
            instr = pop()
            readyentry = nullbuffer[0] - 1
            nullbuffer.pop(0)  # 剩余buffer减一个
            self.reorderbufferentry[readyentry].busy = "Yes"
            self.reorderbufferentry[readyentry].instruction = instr
            self.reorderbufferentry[readyentry].time = "Decode"
            self.reorderbufferentry[readyentry].dest = instr.dest
            self.reorderbufferentry[readyentry].value = None
            self.commitorder.append(readyentry + 1)
    return 1
```

2.Issue

同理 “按commitorder顺序看reorderbuffer中有哪些指令是①属于decode阶段且②有对应的保留站条目空闲可用 才可以进行到issue阶段” 这个步骤执行两次 但不一定一定会有两个指令变为issue状态，因为还要看保留站是否有条目空闲可用

3.Execute

因为本身执行阶段就是遍历reservation看是否有ready的条目就立刻执行，所以不需要单独的for i in range(2)控制双发射

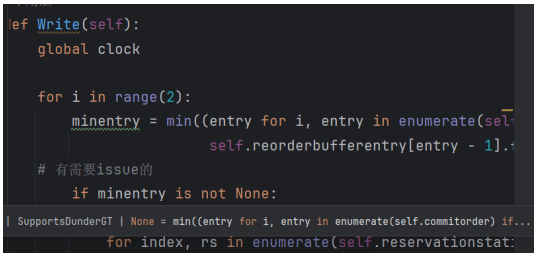


4.Write

这里在原来的单发射逻辑前面加上了这段代码：

```
minentry = min((entry for i, entry in enumerate(self.commitorder) if
                            self.reorderbufferentry[entry - 1].flag==True),
default=None)
```

因为之前是单指令流水线式执行 所以不可能出现两条指令恰好在一个时钟周期完成执行进入到write阶段 所以简化了代码编写 直接遍历reservation station 而现在双指令指令之间重合的部分很大 并行度很高，所以需要严格控制循环两次 且是按commitorder的顺序转换指令所处阶段为write



5.commit

这里是分为处于commit阶段的开始还是末尾讨论 如果处于末尾则没有任何限制 直接清空对应的reorderbuffer 但如果处于开始 就是write result和commit的交界点 就要受到总线的限制 即只能双发射两条指令进行到下一阶段 所以后面部分代码用了for i in range(2)修饰

```
def Commit(self):
    global clock
    update = 0
    for i in range(len(self.reorderbufferentry)):
        rf = self.reorderbufferentry[i]
        if rf.time == "Commit":
            rf.busy, rf.instruction, rf.time, rf.dest, rf.value = "No", None, None, None, None

    # for instr that wrote result, commit in order
    for i in range(2):
        for i, rf in enumerate(self.reorderbufferentry):
            if len(self.commitorder) > 0 and i + 1 == self.commitorder[0] and rf.time == "Write result"
                self.commitorder.pop(0)
                rf.instruction.commit_time = clock
                rf.time = "Commit"

            # release the resource from reservation station
                rs = [rs for rs in self.reservationstation if rs.entry == i + 1][0]
                rs.busy, rs.op, rs.vj, rs.vk, rs.qj, rs.qk, rs.dest = "No", None, None, None, None, Nor
                rs.need_time = None
                rs.ready = 0
```

6.bne 指令处理

且这里多了一种指令bne，所以这里暂且规定执行所需要的时钟周期为5

并且为bne指令设置了2个保留站条目，以便于不支持推测执行需要延迟的情况处理

```
 self.reservationstation = [ReservationStation(name) for name in
                                    ["Load1", "Load2", "Add1", "Add2", "Add3",
"Mult1", "Mult2","bne1","bne2"]]
```

7.相关约定

为简化标签判断 这里直接令loop循环3次，即读入的指令是这样的

```
1  ld x2,0(x1)
2  addi x2,x2,1
3  sd x2,0(x1)
4  addi x1,x1,8
5  bne x3,x1,Loop
6  ld x2,0(x1)
7  addi x2,x2,1
8  sd x2,0(x1)
9  addi x1,x1,8
10 bne x3,x1,Loop
11 ld x2,0(x1)
12 addi x2,x2,1
13 sd x2,0(x1)
14 addi x1,x1,8
15 bne x3,x1,Loop
16
```

支持推测执行的处理

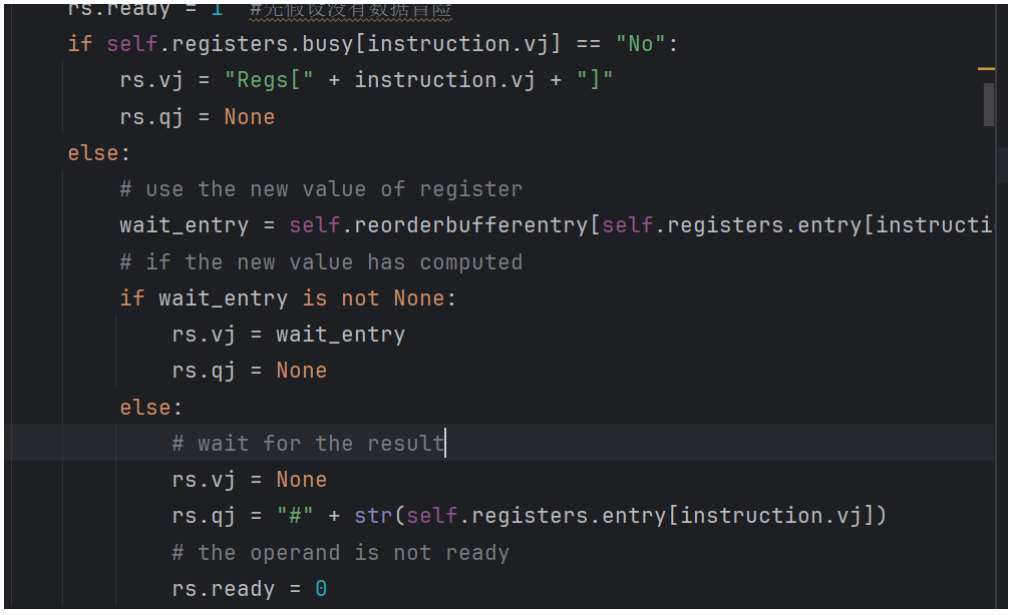
如果支持推测执行，则即使发生了分支冒险 ，也可以不用等待分支条件的判断，可以先预测下一条指令的地址，这里假设预测的结果都是跳转到标签loop处。**所以不需要看判断的操作数是否在之前的指令的目的操作数出现过，即不需要阻塞在Issue阶段。**

在issue阶段对bne指令的处理逻辑是直接把ready标志赋值为1，即可以进入exec阶段 把后续判断是否发生数据冒险的部分直接注释掉

最后的结果可以看到bne的issue下个时钟周期就进入到了execuate阶段。这里之所以exec_time和issue_time相差了5个时钟周期是因为我们约定 exec_time是指指令执行结束的时钟周期，所以三个bne指令都是exec_time和issue_time相差了5个时钟周期，说明issue后不会阻塞立即进入exec

不支持推测执行

即需要等待分支条件的判断结果才能确定下一条指令的地址，所以如果判断结果依赖于之前指令的目的寄存器的值，则需要阻塞在issue阶段 观察可知bne指令只有x1需要依赖之前的指令执行结果 所以这里简化代码 只需要判断vj这个源操作数 对应的寄存器是否繁忙



实验要求的结果展示

ReorderBuffer状态（这里相关的资源以第6个时钟周期为例）

可以看到只有执行到write result阶段才会把结果写回

```
clock6 :
reorderbuffer entry----------------------------

entry 1: Yes    , fld f6,32(x2)     , Commit    , f6    ,Mem[32+Regs[x2]]
entry 2: Yes    , fld f2,44(x3)     , Write result  , f2    ,Mem[44+Regs[x3]]
entry 3: Yes    , fmul.d f0,f2,f4   , Execute   , f0    ,None
entry 4: Yes    , fsub.d f8,f2,f6   , Execute   , f8    ,None
entry 5: Yes    , fdiv.d f0,f0,f6   , Issue     , f0    ,None
entry 6: Yes    , fadd.d f6,f8,f2   , Decode    , f6    ,None
```

每个时钟周期的保留站状态

相关分析：

①因为第一跳的指令 为commit 所以可以看到他已经被从reorder buffer 中删去了

②而第二个指令 fld f2,44(x3) 的两个操作数 Regs[x3]和44都已经被正确读取，放在了保留站内 最后一个标识#2意思是这条指令的结果暂时先不写入对应寄存器 以免有数据冲突的情况发生

③我们再看最后一条mult2中的这个指令 这里发生了数据冒险 f6因为第一条指令已经从内存当中读入了数放进寄存器f6 所以没有发生数据冲突 而另一个操作数 f0会有数据冒险 所以不直接把f0放进源操作数而是放指令的标识号entry#3 表示数据冒险依赖于#3这条指令的执行结果

```
reservation station----------------------------

Load1   : No ,  ,   ,   ,   ,   ,   ;
Load2   : Yes , fld , 44    , Regs[x3]  ,   ,   , #2    ;
Add1    : Yes , fsub.d  , Mem[44+Regs[x3]]  , Mem[32+Regs[x2]]  ,   ,   , #4
Add2    : No ,  ,   ,   ,   ,   ,   ;
Add3    : No ,  ,   ,   ,   ,   ,   ;
Mult1   : Yes , fmul.d  , Mem[44+Regs[x3]]  , Regs[f4]  ,   ,   , #3    ;
Mult2   : Yes , fdiv.d  ,   , Regs[f6]  , #3    ,   , #5    ;
```

寄存器组状态

这里的第一行标识是哪一个reorder buffer中的指令条目占用了这个寄存器 第二行标识寄存器的状态 是否busy

```
registers----------------------------

reorderbuffer_entry: f0:5   ; f1:   ; f2:2   ; f3:   ; f4:   ; f5:   ; f6:   ; f7:   ; f8:4   ; f9:
Busy    : f0:Yes ; f1:No ; f2:Yes ; f3:No ; f4:No ; f5:No ; f6:No ; f7:No ; f8:Yes ; f9:No ; f10:No ;
--------------------------------------------------------------------------------
```

结合你的代码，说明支持推测执行的双发射模拟器中各个部件的数据结构和指令处理关键逻辑的设计。

各个部件的数据结构和指令处理的代码展示和说明主要见上面的“任务1 单发射处理器的模拟”部分，支持推测执行的双发射模拟器的特定部分见“任务2：双发射处理器的模拟”部分

基于任务1和任务2各自的两组实验结果，分别创建表格以显示指令的执行情况

注意这里规定展示的时钟周期都是这个阶段的时间末尾而不是时间开头

1.任务1

（这里没有分支指令 所以没办法体现支持推测执行和不推测执行 只能在第二段代码体现 所以这里只展示单发射和双发射）

```
1 fld f6,32(x2)
2 fld f2,44(x3)
3 fmul.d f0,f2,f4
4 fsub.d f8,f2,f6
5 fdiv.d f0,f0,f6
6 fadd.d f6,f8,f2
```

单发射指令的执行情况

```
指令       , 发射指令的时钟周期   , 执行指令的时钟周期    , 访问存储器的时钟周期   , 写CDB的时钟周期
fld f6,32(x2)   : 2     , 4     , 5     , 6
fld f2,44(x3)   : 3     , 5     , 6     , 7
fmul.d f0,f2,f4     : 4     , 12    , 13    , 14
fsub.d f8,f2,f6     : 5     , 8     , 9     , 15
fdiv.d f0,f0,f6     : 6     , 25    , 26    , 27
fadd.d f6,f8,f2     : 7     , 11    , 12    , 28
```

双发射指令的执行情况

```
instruction       ,issue_time    , exec_time     , write_time    , commit_time
fld f6,32(x2)   : 2     , 4     , 5     , 6
fld f2,44(x3)   : 2     , 4     , 5     , 6
fmul.d f0,f2,f4     : 3     , 11    , 12    , 13
fsub.d f8,f2,f6     : 3     , 7     , 8     , 13
fdiv.d f0,f0,f6     : 4     , 24    , 25    , 26
fadd.d f6,f8,f2     : 4     , 10    , 11    , 26
```

2.任务2

支持推测执行的的双发射模拟器指令的执行情况

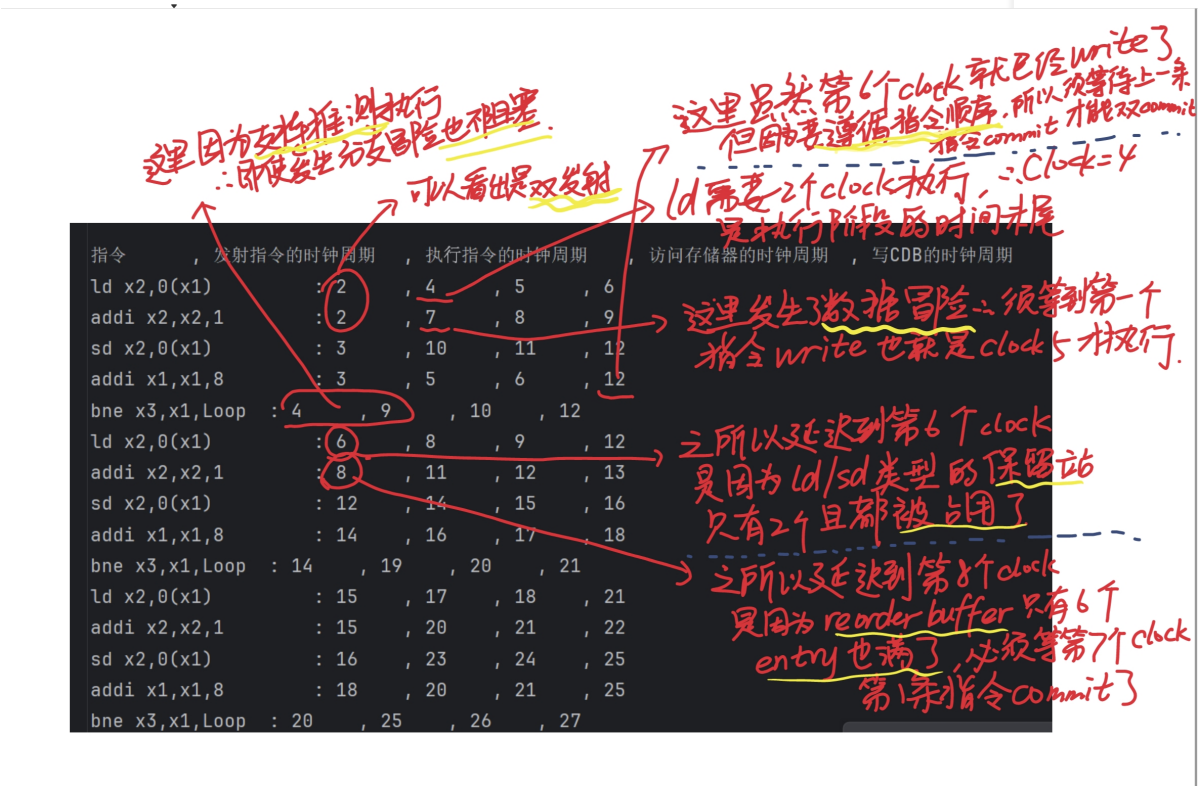
```
指令       , 发射指令的时钟周期   , 执行指令的时钟周期    , 访问存储器的时钟周期   , 写CDB的时钟周期
ld x2,0(x1)          : 2     , 4     , 5     , 6
addi x2,x2,1         : 2     , 7     , 8     , 9
sd x2,0(x1)          : 3     , 10    , 11    , 12
addi x1,x1,8         : 3     , 5     , 6     , 12
bne x3,x1,Loop  : 4     , 9     , 10    , 12
ld x2,0(x1)          : 6     , 8     , 9     , 12
addi x2,x2,1         : 8     , 11    , 12    , 13
sd x2,0(x1)          : 12    , 14    , 15    , 16
addi x1,x1,8         : 14    , 16    , 17    , 18
bne x3,x1,Loop  : 14    , 19    , 20    , 21
ld x2,0(x1)          : 15    , 17    , 18    , 21
addi x2,x2,1         : 15    , 20    , 21    , 22
sd x2,0(x1)          : 16    , 23    , 24    , 25
addi x1,x1,8         : 18    , 20    , 21    , 25
bne x3,x1,Loop  : 20    , 25    , 26    , 27
```

不支持推测执行的的双发射模拟器指令的执行情况

```
指令       , 发射指令的时钟周期   , 执行指令的时钟周期    , 访问存储器的时钟周期   , 写CDB的时钟周期
ld x2,0(x1)          : 2     , 4     , 5     , 6
addi x2,x2,1         : 2     , 7     , 8     , 9
sd x2,0(x1)          : 3     , 10    , 11    , 12
addi x1,x1,8         : 3     , 5     , 6     , 12
bne x3,x1,Loop  : 4     , 11    , 12    , 13
ld x2,0(x1)          : 6     , 8     , 9     , 13
addi x2,x2,1         : 8     , 11    , 12    , 13
sd x2,0(x1)          : 12    , 14    , 15    , 16
addi x1,x1,8         : 14    , 16    , 17    , 18
bne x3,x1,Loop  : 14    , 22    , 23    , 24
ld x2,0(x1)          : 15    , 17    , 18    , 24
addi x2,x2,1         : 15    , 20    , 21    , 24
sd x2,0(x1)          : 16    , 23    , 24    , 25
addi x1,x1,8         : 18    , 20    , 21    , 25
bne x3,x1,Loop  : 20    , 26    , 27    , 28
```

结合你的实验结果，说明推测执行改进Tomasulo算法的原理和实际效果

指令执行结果的一个分析



可以看到推测执行改进Tomasulo算法可以**提升性能。**对于具有循环结构和条件分支的程序，推测执行可以显著提高程序的执行速度。例如，在循环中，能够提前预测循环体中的指令执行，避免每次循环都等待分支条件的判断。也可以**减少流水线停顿****，Tomasulo 算法本身已经减少了部分数据冒险导致的流水线停顿。推测执行在此基础上进一步减少了由于分支指令导致的流水线停顿，使流水线能够更流畅地运行，提高了整个处理器的吞吐率。

分别计算4组实验的IPC，并结合你的实验结果说明多发射超标量的优势和挑战。

IPC

计算公式为：$IPC = \frac{\text{执行的指令总数}}{\text{时钟周期总数}}$。

这里主要是为了说明多发射超标量的优势，所以要用相同指令的程序进行比较，任务一的单发射IPC=6/28=21.4% 任务一的双发射IPC=6/26=23%

第三组（也就是支持推测执行的的双发射模拟器）的IPC=15/27=55.5% 第四组IPC=53%

多发射超标量

多发射超标量是一种处理器架构技术，它允许处理器在一个时钟周期内发射并执行多条指令，通过并行处理指令来提高处理器的性能。

优势:（一）提高指令级并行度（ILP）（二）提高处理器的吞吐率

挑战:（一）指令调度复杂性，多条指令并行执行时，指令间的数据依赖关系变得更加复杂。（二）资源冲突解决 多发射超标量处理器内部的功能单元资源是有限的。所以在我们的程序也可以看出双发射比单发射的IPC并没有高出多少，**主要是受reorder buffer，reservation的资源的限制和数据冒险的阻塞**