

Tomasulo 算法实现

Tomasulo 算法实现

运行方式

输入输出

输入通过文件

输出格式示例

结果展示

input1.txt 的运行结果

input2.txt 的运行结果

input2.txt 的运行过程分析

普通指令的四阶段

SD 指令

ROB实现：循环的定长队列

组织逻辑

FP Op Queue

Reorder Buffer

Reservation Status

Register Result Status

Speculative Tomasulo

运算时间

Stord 指令

流程逻辑

发射阶段

执行阶段

写回阶段

提交阶段

附加题

1 Tomasulo 算法相对于 Scoreboard 算法的优点？同时简述Tomasulo 存在的缺点。

2 简要介绍引入重排序改进 Tomasulo 的原理。

寄存器重命名逻辑实现






重排序改进 Tomasulo 的原理

3 请分析重排序缓存的缺点。

(拿Markdown文档看说不定会好看一点)

运行方式

本程序的实现语言是 python，唯一主动引用的库是简单的 `queue`。文件组织如下：

 FPOPqueue.py	最终版
 RegisterResultStatus.py	最终版
 ReorderBuffer.py	最终版
 ReservationStations.py	最终版
 main.py	最终版

强烈建议运行一遍程序，因为我终端输出调的很好看！！但是输出到 txt 或打印成pdf就不好看了。

1. 解压文件夹/src
2. 可以的话用 pycharm 打开该文件夹
3. 在 main.py 中运行 main() 函数

输入输出

输入通过文件

首先将输入规定为以下格式：

```
# 例一
LD F6 34 R2
LD F2 45 R3
MULTD 0 F2 F4
SUBD F8 F6 F2
DIVD F10 F0 F6
ADDD F6 F8 F2
```

```
# 例二
LD F2 0 R2
LD F4 0 R3
DIVD F0 F4 F2
MULTD F6 F0 F2
ADDD F0 F4 F2
SD F6 0 R3
MULTD F6 F0 F2
SD F6 0 R1
```

输出格式示例

```
=====
Cycles 0
Reorder Buffer(Empty): Head=1 Tear=1
Entry  Busy  Instruction      State  Dest  Value
  1     No
  2     No
  3     No
  4     No
  5     No
  6     No

Reservation Stations
Time  Name  Busy  Op    Vj          Vk          Qj  Qk  Dest  Addr
    Add1   No
    Add2   No
    Add3   No
    Mult1  No
    Mult2  No
    Load1 No
    Load2 No

Register Result Status
          F0  F2  F4  F6  F8  F10
Reorder#
Busy      No  No  No  No  No  No
=====
```

注：我采用的保留站各类 Entry 数量、如上所示。但是这些在我的实现中都只是超参数！可以人为设置！！

例如下面这张图是重排序缓存大小为3，可以发现这的确影响了最终执行效果（请观察最后一指令的 Issue时间）。

```
=====+=====
Cycles 36
Reorder Buffer(Empty): Head=1 Tear=1
Entry  Busy  Instruction      State  Dest  Value
  1     No  SUBD F8 F6 F2  Commit  F8  F6-F2
  2     No  DIVD F10 F0 F6  Commit  F10  #3/F6
  3     No  ADDD F6 F8 F2  Commit  F6  F8+F2

Reservation Stations
Time  Name  Busy  Op    Vj          Vk          Qj  Qk  Dest  Addr
    Add1   No
    Add2   No
    Add3   No
    Mult1  No
    Mult2  No
    Load1 No
    Load2 No

Register Result Status
          F0  F2  F4  F6  F8  F10
Reorder#
Busy      No  No  No  No  No  No
=====

Instructions Cycle Table
          Issue  Exec  Write  Commit
LD F6 34 R2      1    2    4    5
LD F2 45 R3      2    3    5    6
MULTD F0 F2 F4   3    5   15   16
SUBD F8 F6 F2    5    6    8   16
DIVD F10 F0 F6   6   15   35   36
ADDD F6 F8 F2   16   17   19   36
=====
```

结果展示

最有说服力的还是运行结果。下面两张图分别是 `input1.txt` 和 `input2.txt` 的运行结果：

input1.txt 的运行结果

```
=====
Cycles 36
Reorder Buffer(Empty): Head=1 Tail=1
Entry  Busy  Instruction      State  Dest  Value
  1    No   LD F6 34 R2      Commit F6   Mem[Load1]
  2    No   LD F2 45 R3      Commit F2   Mem[Load2]
  3    No   MULTD F0 F2 F4   Commit F0   #2*F4
  4    No   SUBD F8 F6 F2     Commit F8   F6-#2
  5    No   DIVD F10 F0 F6    Commit F10  #3/F6
  6    No   ADDD F6 F8 F2     Commit F6   #4+F2

Reservation Stations
Time  Name  Busy  Op      Vj          Vk          Qj  Qk  Dest  Addr
    Add1   No
    Add2   No
    Add3   No
    Mult1   No
    Mult2   No
    Load1  No
    Load2  No

Register Result Status
                F0  F2  F4  F6  F8  F10
Reorder#
Busy          No  No  No  No  No  No
=====
Instructions Cycle Table
                Issue  Exec  Write  Commit
LD F6 34 R2      1    2    4    5
LD F2 45 R3      2    3    5    6
MULTD F0 F2 F4   3    5    15   16
SUBD F8 F6 F2    4    5    7    16
DIVD F10 F0 F6   5    15   35   36
ADDD F6 F8 F2    6    7    9    36

注： 这张 Cycle Table 展示的是指令进入某一状态的时间。
本程序支持：
    1. Write 阶段的结果旁路到 Exec 阶段
    2. Commit 阶段的写寄存器旁路到 Issue 阶段，使能 RSEntry.ready，以解决 RAW 冲突
    3. 单周期只能发射至多一条指令（多发射也不难实现）
    4. 单周期允许提交至少一条指令（这才叫超标量处理器嘛）
    5. 互斥访问运算单元与内存（这是容易遗漏的）
=====

进程已结束，退出代码为 0
```

从上图中您能够看出：

- 1. 各指令的运算所用 cycle 数
- 2. ROB 消除了假相关数据冲突
- 3. RAW 冲突存在时的阻塞
- 4. Write 状态与 Commit 状态的旁路
- 5. 按序地发射
- 6. 按序地提交，且单周期允许多提交（实际上会对CDB提出更大要求）

由于 RAW 依赖的缘故，上述指令中占用相同运算单元的指令都是串行执行的，看不出我所实现的**运算单元互斥访问**。但别急，下面的指令流展示了这一点。

input2.txt 的运行结果

```
=====
Cycles 46
Reorder Buffer(Empty): Head=3 Tear=3
Entry  Busy  Instruction  State  Dest  Value
1      No    MULTD F6 F0 F2  Commit  F6    F0*F2
2      No    SD F6 0 R1      Commit
3      No    DIVD F0 F4 F2  Commit  F0    #2/#1
4      No    MULTD F6 F0 F2  Commit  F6    #3*F2
5      No    ADDD F0 F4 F2  Commit  F0    F4+F2
6      No    SD F6 0 R3      Commit

Reservation Stations
Time  Name  Busy  Op    Vj          Vk          Qj  Qk  Dest  Addr
      Add1   No
      Add2   No
      Add3   No
      Mult1  No
      Mult2  No
      Load1  No
      Load2  No

Register Result Status
              F0  F2  F4  F6  F8  F10
Reorder#
Busy          No  No  No  No  No  No
=====
Instructions Cycle Table
              Issue  Exec  Write  Commit
LD F2 0 R2    1     2     4     5
LD F4 0 R3    2     3     5     6
DIVD F0 F4 F2 3     5    25    26
MULTD F6 F0 F2 4    25    35    36
ADDD F0 F4 F2 5     6     8     36
SD F6 0 R3    6     7    35    36
MULTD F6 F0 F2 25    35    45    46
SD F6 0 R1    26    27    45    46
注： 这张 Cycle Table 展示的是指令进入某一状态的时间。
本程序支持：
1. Write 阶段的结果旁路到 Exec 阶段
2. Commit 阶段的写寄存器旁路到 Issue 阶段，使能 RSEntry.ready，以解决 RAW 冲突
3. 单周期只能发射至多一条指令（多发射也不难实现）
4. 单周期允许提交至少一条指令（这才叫超标量处理器嘛）
5. 互斥访问运算单元与内存（这是容易遗漏的）
=====
进程已结束，退出代码为 0
```

从input2.txt 的运行结果中您能够看出（之前提过的不再赘述）：

- 1. ROB实现：循环的定长队列
- 2. 第7条指令 `MULTD F6 F0 F2` 在第25 cycle 就满足两个源操作数了，但是直到第35 cycle它才执行，这是因为**乘法运算单元的互斥访问**。
- 3. SD指令：
 - 1. 在 Exec 阶段进行了地址计算
 - 2. 但是（两条SD指令都是）一直等待所 RAW 依赖的源浮点寄存器通过旁路取得后（checkRAW），并且进行获取**访问 Mem 的互斥锁**的判断。两种条件都满足后才能进入 Write 阶段。

内存作为临界区，读操作（LD）需要获取共享锁S才能进入，写操作（SD）需要获取互斥锁X才能进入。

锁相容性矩阵:

	S	X
S	true	false
X	false	false

3. 在 Write 阶段进行内存的 dirty write。
4. commit: 检查是否为最旧的指令。dirty data 真正地提交内存。

input2.txt 的运行过程分析

普通指令的四阶段

LD指令没有源操作数寄存器。

```
=====
Cycles 3
Reorder Buffer: Head=1 Tear=3
Entry  Busy  Instruction      State  Dest  Value
  1     Yes   LD F2 0 R2        Exec   F2
  2     Yes   LD F4 0 R3        Issue  F4
  3     No
  4     No
  5     No
  6     No

Reservation Stations
Time  Name   Busy  Op      Vj          Vk          Qj  Qk  Dest  Addr
    Add1    No
    Add2    No
    Add3    No
    Mult1   No
    Mult2   No
  1  Load1  Yes   LD                #1   0+Regs[R2]
    Load2  Yes   LD                #2   0+Regs[R3]

Register Result Status
          F0  F2  F4  F6  F8  F10
Reorder#          #1  #2
Busy           No  Yes Yes No  No  No
=====
```

=====

Cycles 4

Reorder Buffer: Head=1 Tear=4

Entry	Busy	Instruction	State	Dest	Value
1	Yes	LD F2 0 R2	Exec	F2	
2	Yes	LD F4 0 R3	Exec	F4	
3	Yes	DIVD F0 F4 F2	Issue	F0	
4	No				
5	No				
6	No				

Reservation Stations

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	Addr
	Add1	No							
	Add2	No							
	Add3	No							
	Mult1	Yes	DIVD			#2	#1	#3	None
	Mult2	No							
0	Load1	Yes	LD					#1	0+Regs[R2]
1	Load2	Yes	LD					#2	0+Regs[R3]

Register Result Status

	F0	F2	F4	F6	F8	F10
Reorder#	#3	#1	#2			
Busy	Yes	Yes	Yes	No	No	No

=====

=====

Cycles 5

Reorder Buffer: Head=1 Tear=5

Entry	Busy	Instruction	State	Dest	Value
1	Yes	LD F2 0 R2	Write	F2	Mem[Load1]
2	Yes	LD F4 0 R3	Exec	F4	
3	Yes	DIVD F0 F4 F2	Issue	F0	
4	Yes	MULTD F6 F0 F2	Issue	F6	
5	No				
6	No				

Reservation Stations

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	Addr
	Add1	No							
	Add2	No							
	Add3	No							
	Mult1	Yes	DIVD		Mem[Load1]	#2		#3	None
	Mult2	Yes	MULTD		Regs[F2]	#3		#4	None
	Load1	No							
0	Load2	Yes	LD					#2	0+Regs[R3]

Register Result Status

	F0	F2	F4	F6	F8	F10
Reorder#	#3	#1	#2	#4		
Busy	Yes	Yes	Yes	Yes	No	No

=====

=====

Cycles 6

Reorder Buffer: Head=2 Tear=6

Entry	Busy	Instruction	State	Dest	Value
1	No	LD F2 0 R2	Commit	F2	Mem[Load1]
2	Yes	LD F4 0 R3	Write	F4	Mem[Load2]
3	Yes	DIVD F0 F4 F2	Exec	F0	
4	Yes	MULTD F6 F0 F2	Issue	F6	
5	Yes	ADDD F0 F4 F2	Issue	F0	
6	No				

Reservation Stations

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	Addr
	Add1	Yes	ADDD	Regs[F4]	Regs[F2]			#5	None
	Add2	No							
	Add3	No							
19	Mult1	Yes	DIVD	Mem[Load2]	Mem[Load1]			#3	None
	Mult2	Yes	MULTD		Regs[F2]	#3		#4	None
	Load1	No							
	Load2	No							

Register Result Status

	F0	F2	F4	F6	F8	F10
Reorder#	#5		#2	#4		
Busy	Yes	No	Yes	Yes	No	No

=====

SD 指令

=====

Cycles 7

Reorder Buffer: Head=3 Tear=1

Entry	Busy	Instruction	State	Dest	Value
1	No	LD F2 0 R2	Commit	F2	Mem[Load1]
2	No	LD F4 0 R3	Commit	F4	Mem[Load2]
3	Yes	DIVD F0 F4 F2	Exec	F0	
4	Yes	MULTD F6 F0 F2	Issue	F6	
5	Yes	ADDD F0 F4 F2	Exec	F0	
6	Yes	SD F6 0 R3	Issue		

Reservation Stations

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	Addr
1	Add1	Yes	ADDD	Regs[F4]	Regs[F2]			#5	None
	Add2	No							
	Add3	No							
18	Mult1	Yes	DIVD	Mem[Load2]	Mem[Load1]			#3	None
	Mult2	Yes	MULTD		Regs[F2]	#3		#4	None
	Load1	Yes	SD			#4			0+Regs[R3]
	Load2	No							

Register Result Status

	F0	F2	F4	F6	F8	F10
Reorder#	#5			#4		
Busy	Yes	No	No	Yes	No	No

=====

SD没有目的寄存器，但是有源操作数

=====

ROB or RS is full, function 'Issue MULTD F6 F0 F2' failed.

=====

Cycles 8

Reorder Buffer: Head=3 Tear=1

Entry	Busy	Instruction	State	Dest	Value
1	No	LD F2 0 R2	Commit	F2	Mem[Load1]
2	No	LD F4 0 R3	Commit	F4	Mem[Load2]
3	Yes	DIVD F0 F4 F2	Exec	F0	
4	Yes	MULTD F6 F0 F2	Issue	F6	
5	Yes	ADDD F0 F4 F2	Exec	F0	
6	Yes	SD F6 0 R3	Exec		

Reservation Stations

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	Addr
0	Add1	Yes	ADDD	Regs[F4]	Regs[F2]			#5	None
	Add2	No							
	Add3	No							
17	Mult1	Yes	DIVD	Mem[Load2]	Mem[Load1]			#3	None
	Mult2	Yes	MULTD		Regs[F2]	#3		#4	None
0	Load1	Yes	SD			#4			0+Regs[R3]
	Load2	No							

Register Result Status

	F0	F2	F4	F6	F8	F10
Reorder#	#5			#4		
Busy	Yes	No	No	Yes	No	No

=====

=====

ROB or RS is full, function 'Issue MULTD F6 F0 F2' failed.

=====

Cycles 9

Reorder Buffer: Head=3 Tear=1

Entry	Busy	Instruction	State	Dest	Value
1	No	LD F2 0 R2	Commit	F2	Mem[Load1]
2	No	LD F4 0 R3	Commit	F4	Mem[Load2]
3	Yes	DIVD F0 F4 F2	Exec	F0	
4	Yes	MULTD F6 F0 F2	Issue	F6	
5	Yes	ADDD F0 F4 F2	Write	F0	F4+F2
6	Yes	SD F6 0 R3	Exec		

Reservation Stations

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	Addr
	Add1	No							
	Add2	No							
	Add3	No							
16	Mult1	Yes	DIVD	Mem[Load2]	Mem[Load1]			#3	None
	Mult2	Yes	MULTD		Regs[F2]	#3		#4	None
	Load1	Yes	SD			#4			0+Regs[R3]
	Load2	No							

Register Result Status

	F0	F2	F4	F6	F8	F10
Reorder#	#5			#4		
Busy	Yes	No	No	Yes	No	No

=====

=====

ROB or RS is full, function 'Issue MULTD F6 F0 F2' failed.

=====

Cycles 25

Reorder Buffer: Head=3 Tear=1

Entry	Busy	Instruction	State	Dest	Value
1	No	LD F2 0 R2	Commit	F2	Mem[Load1]
2	No	LD F4 0 R3	Commit	F4	Mem[Load2]
3	Yes	DIVD F0 F4 F2	Exec	F0	
4	Yes	MULTD F6 F0 F2	Issue	F6	
5	Yes	ADDD F0 F4 F2	Write	F0	F4+F2
6	Yes	SD F6 0 R3	Exec		

Reservation Stations

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	Addr
	Add1	No							
	Add2	No							
	Add3	No							
0	Mult1	Yes	DIVD	Mem[Load2]	Mem[Load1]			#3	None
	Mult2	Yes	MULTD		Regs[F2]	#3		#4	None
	Load1	Yes	SD			#4			0+Regs[R3]
	Load2	No							

Register Result Status

	F0	F2	F4	F6	F8	F10
Reorder#	#5			#4		
Busy	Yes	No	No	Yes	No	No

=====

=====

=====

Cycles 26

Reorder Buffer: Head=3 Tear=2

Entry	Busy	Instruction	State	Dest	Value
1	Yes	MULTD F6 F0 F2	Issue	F6	
2	No	LD F4 0 R3	Commit	F4	Mem[Load2]
3	Yes	DIVD F0 F4 F2	Write	F0	#2/#1
4	Yes	MULTD F6 F0 F2	Exec	F6	
5	Yes	ADDD F0 F4 F2	Write	F0	F4+F2
6	Yes	SD F6 0 R3	Exec		

Reservation Stations

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	Addr
	Add1	No							
	Add2	No							
	Add3	No							
	Mult1	Yes	MULTD	Regs[F0]	Regs[F2]			#1	None
9	Mult2	Yes	MULTD	#2/#1	Regs[F2]			#4	None
	Load1	Yes	SD			#4			0+Regs[R3]
	Load2	No							

Register Result Status

	F0	F2	F4	F6	F8	F10
Reorder#	#5			#1		
Busy	Yes	No	No	Yes	No	No

=====

=====

ROB实现：循环的定长队列

Cycles 26

Reorder Buffer: Head=3 Tear=2

Entry	Busy	Instruction	State	Dest	Value
1	Yes	MULTD F6 F0 F2	Issue	F6	
2	No	LD F4 0 R3	Commit	F4	Mem[Load2]
3	Yes	DIVD F0 F4 F2	Write	F0	#2/#1
4	Yes	MULTD F6 F0 F2	Exec	F6	
5	Yes	ADDD F0 F4 F2	Write	F0	F4+F2
6	Yes	SD F6 0 R3	Exec		

Reservation Stations

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	Addr
	Add1	No							
	Add2	No							
	Add3	No							
	Mult1	Yes	MULTD	Regs[F0]	Regs[F2]			#1	None
9	Mult2	Yes	MULTD	#2/#1	Regs[F2]			#4	None
	Load1	Yes	SD			#4			0+Regs[R3]
	Load2	No							

Register Result Status

	F0	F2	F4	F6	F8	F10
Reorder#	#5			#1		
Busy	Yes	No	No	Yes	No	No

Cycles 27

Reorder Buffer: Head=4 Tear=3

Entry	Busy	Instruction	State	Dest	Value
1	Yes	MULTD F6 F0 F2	Issue	F6	
2	Yes	SD F6 0 R1	Issue		
3	No	DIVD F0 F4 F2	Commit	F0	#2/#1
4	Yes	MULTD F6 F0 F2	Exec	F6	
5	Yes	ADDD F0 F4 F2	Write	F0	F4+F2
6	Yes	SD F6 0 R3	Exec		

Reservation Stations

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	Addr
	Add1	No							
	Add2	No							
	Add3	No							
	Mult1	Yes	MULTD	Regs[F0]	Regs[F2]			#1	None
8	Mult2	Yes	MULTD	#2/#1	Regs[F2]			#4	None
	Load1	Yes	SD			#4			0+Regs[R3]
	Load2	Yes	SD			#1			0+Regs[R1]

Register Result Status

	F0	F2	F4	F6	F8	F10
Reorder#	#5			#1		
Busy	Yes	No	No	Yes	No	No

组织逻辑

通过实例化管理各个单元：

```
from FPOPqueue import FQ_OP_QUEUE
from RegisterResultStatus import REGISTER_RESULT_STATUS
from ReorderBuffer import REORDER_BUFFER
from ReservationStations import RESERVATION_STATIONS

class SpeculativeTomasulo: # 算法模拟器（老大）
    def __init__(self):
        self.Units = { # EX阶段的运算单元
            "Add": False,
            "Mult": False,
            "Mem": {'LD': 0, 'SD': False} # 值为'LD'或'SD'。LD共享锁，SD互斥锁
        }
        self.RunningCycle = RunningCycleTable() # 管理各个单元（字典嵌套字典{指令唯一标识: {指令, 指令当前状态, 指令对应ROBEntry, 指令对应REntry}}）
        self.cycle = 0
```

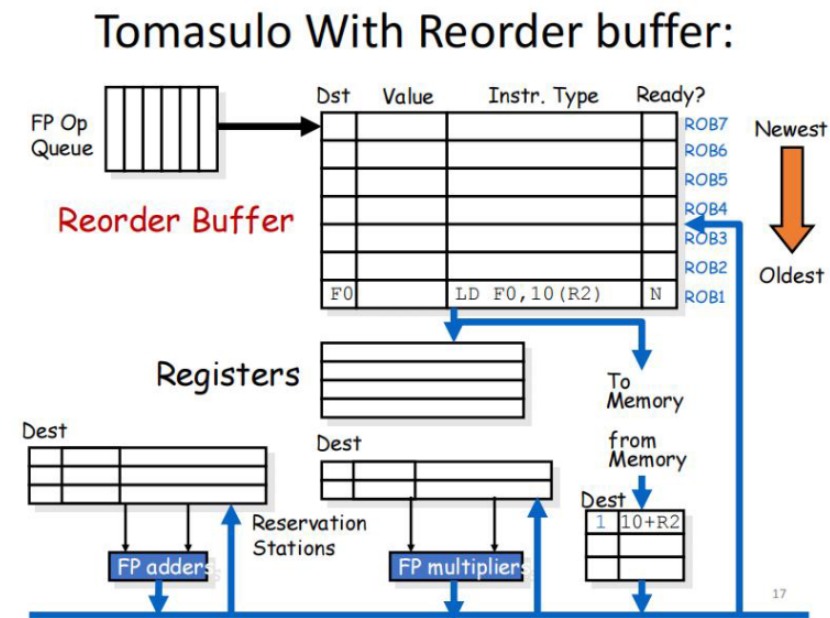


图 1 带有重新排序缓存的 Tomasulo

根据上面这个仿真器的结构图，可以得到如下结构：

FP Op Queue

使用 python 自带的 queue 实现。对其封装即可

```
from queue import Queue

class FPOPqueue:
    def __init__(self):
        self.queue = Queue()
        self.size = 0
```

Reorder Buffer

Cycles 29						
Reorder Buffer: Head=4 Tear=3						
Entry	Busy	Instruction	State	Dest	Value	
1	Yes	MULTD F6 F0 F2	Issue	F6		
2	Yes	SD F6 0 R1	Exec			
3	No	DIVD F0 F4 F2	Commit	F0	#2/#1	
4	Yes	MULTD F6 F0 F2	Exec	F6		
5	Yes	ADDD F0 F4 F2	Write	F0	F4+F2	
6	Yes	SD F6 0 R3	Exec			

Reorder Buffer 和 Reservation Status 是最复杂的，涉及大量旁路、RAW检查、操作。

```
class ReorderBufferEntry:
    def __init__(self, number, busy=False, instruction=None, state=None,
destination=None, value=None, ready=False):
        self.number = number
        self.busy = busy
        self.instruction = instruction # 操作类型，如ADDD F6 F8 F2
        self.state = state # 结果是否就绪
        self.destination = destination # 目标寄存器
        self.value = value # 计算结果或加载的值

class ReorderBuffer:
    def __init__(self):
        self.queue = []
        self.maxlen = ROB_MAX_SIZE
        self.head = 0 # 定长循环队列实现Reorder Buffer
        self.tear = 0
        self.size = 0
        for i in range(self.maxlen):
            self.queue.append(ReorderBufferEntry(i+1)) #number是ROBEntry的唯一标识
```

Reservation Status

Reservation Stations									
Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	Addr
	Add1	No							
	Add2	No							
	Add3	No							
4	Mult1	Yes	DIVD	Mem[Load2]	Mem[Load1]			#3	None
	Mult2	Yes	MULTD		Regs[F2]	#3		#4	None
	Load1	Yes	SD			#4			0+Regs[R3]

Reservation Station 是输入到执行单元的前一站。

```
class ReservationStationsEntry:
    def __init__(self, name, time=-1, busy=False, operation=None,
                 value1=None, value2=None, source1=None, source2=None,
                 destination=None, value=None, address=None, ready=False):
        self.name = name      # 初始化后不修改
        self.timer = time     # 只在Ex阶段使用
        self.busy = busy      # 该项是否使用中
        self.operation = operation # 操作类型, 如ADD, Mult
        self.value1 = value1   #Vj
        self.value2 = value2   #Vk
        self.source1 = source1 #Qj。计算outcome时会用到
        self.source2 = source2 #Qk
        self.destination = destination # 目标寄存器
        self.address = address #Mem指令的地址
        self.ready = ready     # 结果是否就绪。Qj和Qk都None时, ready!

class Station:
    def __init__(self, maxlen, name):
        self.station = []
        self.maxlen = maxlen
        for i in range(maxlen):
            self.station.append(ReservationStationsEntry(name + str(i + 1)))

class ReservationStations:
    def __init__(self):
        self.AddStation = Station(ADDD_MAX_SIZE, 'Add')
        self.MultStation = Station(MULT_MAX_SIZE, 'Mult')
        self.LoadStation = Station(LD_MAX_SIZE, 'Load')
```

Register Result Status

Register Result Status						
	F0	F2	F4	F6	F8	F10
Reorder#	#5			#4		
Busy	Yes	No	No	Yes	No	No

```

class RegisterResultStatus:
    def __init__(self, ):
        self.reg_map = {
            "F0": 0,
            "F2": 1,
            "F4": 2,
            "F6": 3,
            "F8": 4,
            "F10": 5
        }
        self.reorder_number = [None, None, None, None, None, None]
        self.busy_status = [False, False, False, False, False, False,]

```

Speculative Tomasulo

Instructions Cycle Table

	Issue	Exec	Write	Commit
LD F2 0 R2	1	2	4	5
LD F4 0 R3	2	3	5	6
DIVD F0 F4 F2	3	5	25	26
MULTD F6 F0 F2	4	25	35	36
ADDD F0 F4 F2	5	6	8	36
SD F6 0 R3	6	7	35	36
MULTD F6 F0 F2	25	35	45	46
SD F6 0 R1	26	27	45	46

注：这张 Cycle Table 展示的是指令进入某一状态的时间。

本程序支持：

1. Write 阶段的结果旁路到 Exec 阶段
2. Commit 阶段的写寄存器旁路到 Issue 阶段，使能 RSEntry.ready，以解决 RAW 冲突
3. 单周期只能发射至多一条指令（多发射也不难实现）
4. 单周期允许提交至少一条指令（这才叫超标量处理器嘛）
5. 互斥访问运算单元与内存（这是容易遗漏的）

```

class RunningCycleTable:
    def __init__(self):
        self.instructionInfo = {} #存储已发送的指令的状态,ROBEntry,RSEntry
        self.instructionCycle = {} #存储已发送指令进入各个状态的cycle，在程序的最后打印
        的表格

        def append(self, instructionNumber, instruction, cycle, ROBEntryNumber,
        RSEntryName):
            # instructionNumber是指令进入算法的序号号，也是指令的唯一标识。在字典中作为key是再
            适合不过了
            self.instructionInfo[instructionNumber] = {'Instruction': instruction,
            'State': 'Issue', 'ROBEntry': ROBEntryNumber, 'RSEntry': RSEntryName} #方便算法获
            取指令在各个元件中的存放位置
            self.instructionCycle[instructionNumber] = {'Issue': cycle, 'Exec':
            None, 'Write': None, 'Commit': None} #记录最后的输出

class SpeculativeTomasulo:
    def __init__(self):
        self.Units = { # EX阶段的运算单元
            "Add": False, #互斥锁
            "Mult": False,

```

```

        "Mem": {'LD': 0, 'SD': False} #值为'LD'或'SD'。LD共享锁，SD互斥锁
    }
    self.RunningCycle = RunningCycleTable()
    self.cycle = 0

```

每种运算执行单元只有一个，不会出现两个同类型执行单元并行的情况，执行单位内部不允许流水线操作。内存的互斥访问也是重要的部分，在运行展示中已有提及。

运算时间

FP instruction	EX Cycles
fadd	2
fsub	2
fmult	10
fdvi	20

LS instruction	EX Cycles
load	2
store	2

Stord 指令

store指令较为特殊，计算地址1 cycle，访问内存 1 cycle

- Issue：若有 ROBEntry 和 RSEntry 空闲，则直接发射
- Exec：执行 1 cycle 的地址计算
- write：checkRAW，检查源寄存器的RAW数据依赖情况。SD 互斥锁获取判断。进行内存的 dirty write。
- commit：检查是否为最旧的指令。dirty data 真正地提交内存

流程逻辑

Speculative Tomasulo 中的run()相当于主函数，一个while结构的每一次迭代就是一个 cycle。在每个 cycle中，指令的执行是并行的，但程序的实现却只能是串行的，但是我通过合理的执行顺序解决了这件事：指令之间的影响只有两种：RAW 和 运算单元/内存的互斥访问。这两种影响又都是旧指令优先于新指令。同时我们的旧指令需要在Write阶段和Commit阶段进行旁路，将指令结果送至新指令。因此在每个 cycle 中，我从旧到新地遍历指令，嵌套中从Commit到Exec阶段地判断指令的状态改变，在遍历完指令之后在执行一次Issue。这样的程序可以保证所有旁路的实现。

```

def run(self, file_path):
    self.cycle = 0
    while True:
        self.cycle += 1
        #对每条指令状态进行判断顺序:提交->写回->执行。RAW只会影响更新的指令，因此从旧往新
        更新即可。
        for i, (instructionNumber, instructionInfo) in
            enumerate(self.RunningCycle.instructionInfo.items()):

```

```

state = instructionInfo['State']
if state == 'Commit':
    continue #提交的指令已经完成
elif state == 'Write':
    self.commit(instructionNumber, instructionInfo) #最旧指令提交,
更新寄存器
elif state == 'Exec':
    self.write(instructionNumber, instructionInfo) #可能是对指令
Timer--,也可能是将指令write, 旁路, 更新依赖项ready
elif state == 'Issue':
    if instructionNumber==5 and self.cycle == 27:
        print()
    self.exec(instructionNumber, instructionInfo) #检查ready, 为真
则将指令状态置为Exec, 设置计时器
self.issue() #这是每个周期都要做的事情
self.updateRRS() # 显式更新RRSmap
#终止判断
if (FQ_OP_QUEUE.isempty()
    and REORDER_BUFFER.isempty()
    and RESERVATION_STATIONS.isempty()
    and REGISTER_RESULT_STATUS.isempty()):
    break

```

发射阶段

```

def issue(self):
    # FP OP Queue 发送 ins 给 ROB 和 RS 。需要ROB和RS都有空位 （每周只发一条）。
    # 从 FP queue 获取一条指令试试水
    if FQ_OP_QUEUE.isempty():
        return
    instruction = FQ_OP_QUEUE.queue.queue[0]
    op = instruction.split()[0]
    if not REORDER_BUFFER.isfull() and not RESERVATION_STATIONS.isfull(op):
        instructionNumber = FQ_OP_QUEUE.pop() #从 FP queue 中取出
        ROBEntryNumber = REORDER_BUFFER.issue(instruction) #发射到ROB。
        ROBEntry状态初始化为issue。
        RSEntryName = RESERVATION_STATIONS.issue(instruction,
        ROBEntryNumber, REORDER_BUFFER) #发射到RS。RS会有状态更新（需要checkRAW）。
        destination = instruction.split()[1]
        self.issueSetValue(ROBEntryNumber, RSEntryName) #ROB利用RS表项, 提前提前
        提前计算出value的表达式
        self.RunningCycle.append(instructionNumber, instruction, self.cycle,
        ROBEntryNumber, RSEntryName) #加入到列表中
    else:
        print(f"ROB or RS is full, function \'Issue {instruction}\' failed.
        ")

def issueSetValue(self, ROBEntryNumber, RSEntryName): #ROB利用RS表项, 提前提前
        提前计算出value的表达式
        ROBEntry = REORDER_BUFFER.getROBEntry(ROBEntryNumber)
        RSEntry = RESERVATION_STATIONS.getRSEntry(RSEntryName)
        outcome = RSEntry.getROBValue()
        ROBEntry.value = outcome

```

执行阶段

```
def exec(self, instructionNumber, instructionInfo): #为单条指令检查ready and 功能单元锁，为真则将指令状态置为Exec，设置计时器
    RSEntry = RESERVATION_STATIONS.getRSEntry(instructionInfo['RSEntry'])
    op = RSEntry.operation
    if (op=='SD' #SD不需要等待执行ready，直接计算地址
        or RSEntry.ready #检查ready，为真则将指令状态置为Exec，设置计时器，更新
CycleTable
        and (((op=='ADD' or op=='SUB') and not self.Units['Add']) # 运算单元的监听
            or ((op=='MULT' or op=='DIV') and not self.Units['Mult'])
            or (op=='LD' and not self.Units['Mem']['SD']))):
    if op=='ADD' or op=='SUB': #运算单元的获取
        self.Units['Add'] = True
    elif op=='MULT' or op=='DIV':
        self.Units['Mult'] = True
    elif op=='LD':
        self.Units['Mem']['LD'] += 1
    ROEntry = REORDER_BUFFER.getROEntry(instructionInfo['ROEntry'])
    ROEntry.state = 'Exec' #将指令状态置为Exec
    op = RSEntry.operation
    RSEntry.timer = EXEC_SPEND_CYCLE[op]-1 #设置计时器
    self.RunningCycle.setExec(instructionNumber, self.cycle) #更新
CycleTable
```

写回阶段

```
def write(self, instructionNumber, instructionInfo):
    #若执行计数器未执行完，更新计数器，保持state=Exec
    RSEntry = RESERVATION_STATIONS.getRSEntry(instructionInfo['RSEntry'])
    if RSEntry.timer >= 0:
        RSEntry.timer -= 1
    # 否则清除RS、修改ROB状态、执行旁路
    if RSEntry.timer < 0:
        op = RSEntry.operation
        ROEntryNumber = instructionInfo['ROEntry']
        ROEntry = REORDER_BUFFER.getROEntry(ROEntryNumber)
        if op == 'SD': #SD不写寄存器。判断写内存条件。不清空RSEntry（commit才清空）
            if RSEntry.ready and self.Units['Mem']['LD']==0 and not
self.Units['Mem']['SD']: # MEM的互斥访问
                RSEntry.timer = EXEC_SPEND_CYCLE[op] - 1 #访问 Mem 也要 1
cycle
                self.Units['Mem']['SD'] = True
                self.RunningCycle.setwrite(instructionNumber, self.cycle) #更新
新CycleTable
                ROEntry.state = 'Write' #修改ROB状态为Write。SD在Write阶段写内存（脏写，可旁路（未实现）。commit才正式写入内存）
            else: #除了SD之外的指令
                if op == 'ADD' or op == 'SUB': # 运算单元的释放
                    self.Units['Add'] = False
                elif op == 'MULT' or op == 'DIV':
                    self.Units['Mult'] = False
                elif op == 'LD':
```

```

        self.Units['Mem']['LD'] -= 1
    REntry.clear()
    ROEntry.state = 'Write' #修改ROB状态
    self.RunningCycle.setWrite(instructionNumber, self.cycle) #更新
CycleTable

    #旁路: 从ROEntry.value旁路到RS.v==None的项: 遍历RS, 找到RS.v==None的
    REntry, 根据RS.q找到目标ROEntry.value
    RESERVATION_STATIONS.forwarding(ROEntryNumber, ROEntry.value)

    #执行旁路, 主动更新RAW的依赖寄存器

```

提交阶段

```

def commit(self, instructionNumber, instructionInfo):
    instruction = instructionInfo['Instruction']
    op = instruction.split()[0]
    destination = instruction.split()[1]
    if op == 'SD': #将执行结果写入Memory (do nothing)
        self.Units['Mem']['SD'] = False # MEM的释放
        REntry =
    RESERVATION_STATIONS.getREntry(instructionInfo['REntry'])
        REntry.clear() #清空REntry
        #所有指令。清除ROB (修改state)
        if REORDER_BUFFER.commit(instructionInfo['ROEntry']): # 检查是否为最旧的指令。清除ROB (修改state、busy)
            ROEntry = REORDER_BUFFER.getROEntry(instructionInfo['ROEntry'])
            ROEntry.state = 'Commit'
            self.RunningCycle.setCommit(instructionNumber, self.cycle) #更新
CycleTable

```

事实上对每条指令在每个阶段进行判断的思想很有 FSM 的味道！

更多详情请翻阅或执行源码（反正python写的，简单易懂！！）

附加题

1 Tomasulo 算法相对于 Scoreboard 算法的优点？同时简述 Tomasulo 存在的缺点。

首先阐述 Tomasulo 算法相对于 Scoreboard 算法的优点：

数据相关的解决方式上：

- 发射：记分牌算法发射一条指令时，不仅要判断对应功能单元是否被占用，还要判断对应目标寄存器是否正要被其他指令写（保证WAW）。记分牌算法解决这两个问题的方法是阻塞，但是后一种情况可以由 Tomasulo 的寄存器重命名机制避免，从而减少阻塞。
- 写回：记分牌算法写回一条指令的执行结果到目的寄存器时，要判断对应目标寄存器是否需要被其他指令读（保证WAR）。而后者的阻塞情况可以由 Tomasulo 的寄存器重命名机制避免。

记分牌算法是顺序发射、乱序执行、乱序写回的。Tomasulo 算法是乱序发射、乱序执行、顺序写回的。

- 顺序发射：记分牌算法采用顺序发射的发射方式，一旦发射阶段时指令被记分牌单元（ScoreBoard Unit）判断为不能发射，之后的指令都不能发射，导致流水线阻塞。然而

Tomasulo 算法没有顺序发射的要求，在发射阶段中如果一个指令被判断为不能发射，作为代替，下一条指令将会被发射。

- 乱序写回：记分牌算法采用顺序发射的发射方式，不利于处理器处理中断、异常等情况，不利于程序员调试程序。而顺序写回的 Tomasulo 算法则不会有这种问题。

接下来简述Tomasulo 存在的缺点：

Tomasulo 纯净版本是没有ROB的，是乱序提交的，这导致：

- 没办法实现精确中断没办法分支错误后回退

但是这些问题都随着 Reorder Buffer 的引入而解决了。

2 简要介绍引入重排序改进 Tomasulo 的原理。

寄存器重命名逻辑实现

视厂商不同，实现寄存器重命名的方式也不同，主要是一下三种：

1. 将逻辑寄存器(Architecture Register File,ARF)扩展来实现寄存器重命名
2. 使用统一的物理寄存器(Physical Register File,PRF)来实现寄存器重命名（MIPS）
3. 使用 ROB来实现寄存器重命名（多见于intel）

最简单的寄存器重命名实现方法就是第3种方法借助ROB，ROB的每一个表项都会存储指令的执行结果（相当于物理寄存器），同时该表项在ROB中的编号作为物理寄存器的编号。这个过程相当于为每一条指令的目的寄存器都进行了寄存器重命名！

第2种方法的逻辑与第3种方法完全一致，只是将PRF独立出来了。

第1种方法做到了将ROB与PRF真正地解耦开，大大提高了PRF的资源利用率。该方法用三个表格管理PRF：已提交（commit or retire）指令目的寄存器的重命名映射表、实际上最新的指令目的寄存器的重命名映射表、尚未使用的PR的空闲列表。前两者的区别在于，前者对外体现处理器状态（透明化乱序过程，方便程序员调试程序），后者对内服务于乱序机制。

重排序改进 Tomasulo 的原理

Tomasulo 采用的是第3种重排序方法，即ROB实现。

```
class ROB_Entry:
    def __inti__(self):
        Busy: Bool = False
        Instruction: String = ''
        State: String = ''
        Dest: int = None
        value: int = None
```

Tomasulo 算法因为控制指令、程序异常和外部中断会截断指令流而无法顺序提交指令，而乱序提交无法满足处理器按照程序顺序来执行指令的需求。

使用重排序改进后的 Tomasulo 算法称为 Speculative Tomasulo 算法，使用重排序缓存 ROB，让乱序执行的指令被顺序地提交。

3 请分析重排序缓存的缺点。

ROB的一个表项由四部分组成：指令类型、目的寄存器号、执行结果值、是否完成标志 (completed flag) (在模拟器上一般使用State字段)

重排序缓存可以代替物理寄存器组 (Physical Register File, PRF) 进行重命名时, 此时缺点主要有两个:

1. ROB机制下, 每一条指令都要占用一个表项, 但并不是每条指令都拥有目的寄存器 (重命名机制通常作用于目的寄存器而非源寄存器), 因此不需要重命名, 然而通过ROB实现寄存器重命名相当于每个ROB表项都有一个物理寄存器。这种情况就导致了PRF资源的浪费。
2. ROB机制下, 对于一条指令来说, 它既可能在ROB中其它表项的目的寄存器中读取源操作数, 也可能在逻辑寄存器组 (Architecture Register File, ARF) 中读取源操作数, 这带来了额外的路选。对于例如 4-way 的超标量处理器, 最坏情况下每个周期同时四条指令读取源操作数, 每条指令2个源操作数, 因此一共是 $2 \times 4 = 8$ 个读端口 (每个读端口都是一个ROB与ARF的路选)。坏处体现在芯片面积、延迟与复杂性方面。