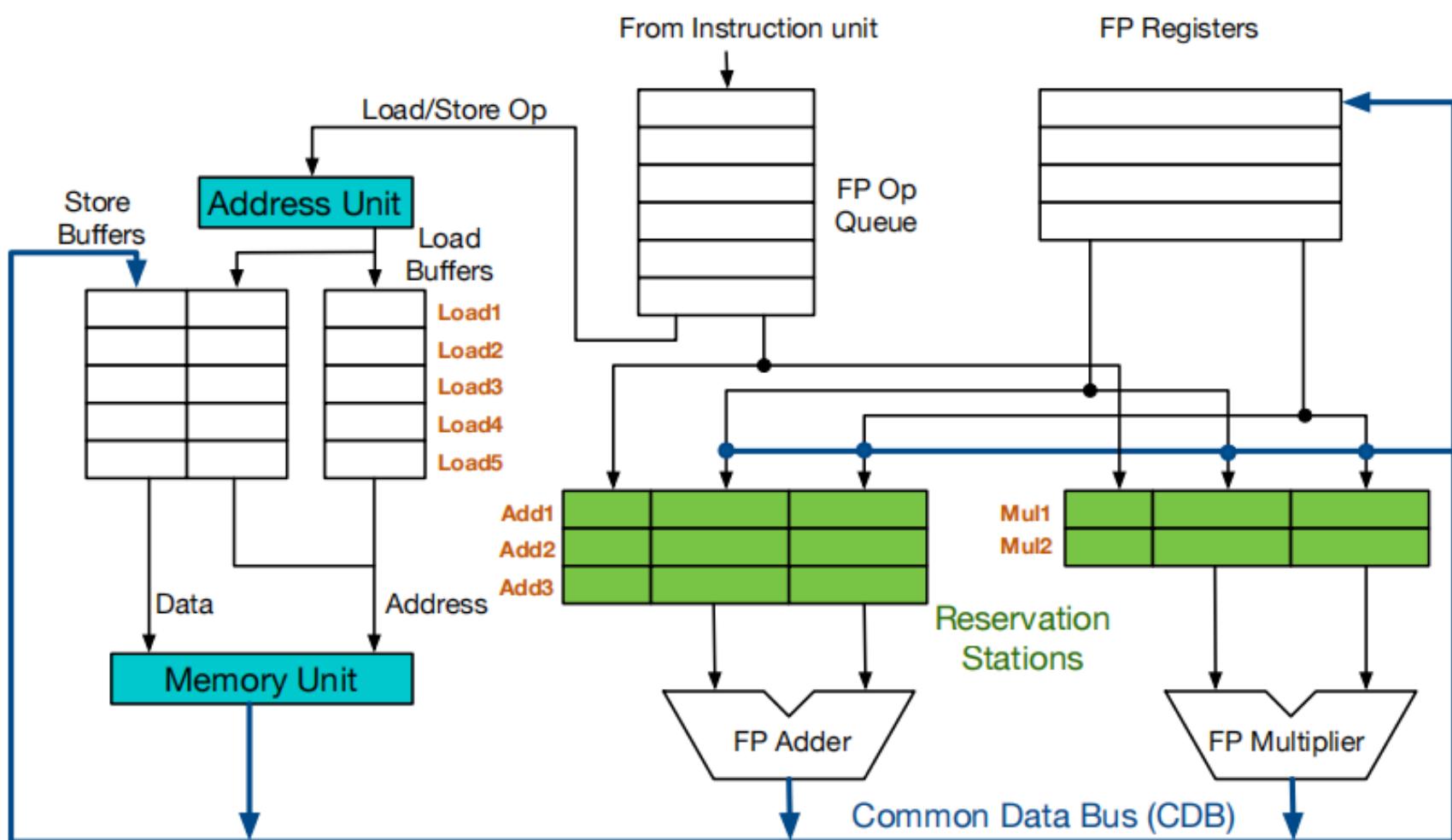


# Speculative Tomasulo 算法实现

21307035 邓栩瀛

## Speculative Tomasulo 算法介绍

Tomasulo是一种计算机硬件架构的算法，用于动态调度指令的指令，允许乱序执行以及更有效率的使用多个执行单元。Tomasulo最大的特点就是通过借助重命名的思想消除了假数据冒险，从而提高了机器的乱序性能。Tomasulo算法的实现结构：



- FP OP Queue, 浮点指令队列，指令在这里等待发射；
- 青绿色模块是加法单元和乘法单元的保留站（保留站保留已经发射的指令的信息和缓冲下来的数据）；
- 蓝绿色的Address Unit是地址计算单元，在这个算法中存储指令在执行前会先计算好存储地址；
- Memory Unit存储单元；
- CDB是数据广播总线，可以直达寄存器堆（用来更新通用寄存器）、加法乘法存储单元的保留站

(输送保留站中指令需要的数据)。

## 保留站和寄存器结果状态表

保留站是Tomasulo算法提出的新结构，有点类似记分牌中每一个配置通路前面的译码信息流水段寄存器，但是记分牌中每一条配置通路只能存放一条指令，而Tomasulo算法则为每一条通路配置了一组缓冲，就像上图中的绿色模块，其中浮点加法单元拥有能够缓冲三条指令的保留站。保留站存储的信息和记分牌有点类似，如下图所示

### • Reservation Station

- **Busy:** Indicates reservation station is busy
- **Op:** operation to perform in the unit
- $V_j, V_k$ : Value of Source operands
- $Q_j, Q_k$ : Reservation stations producing source registers
  - $Q_j, Q_k = 0 \Rightarrow$  operand ready (in  $V_j$  or  $V_k$ )
- **A:** Address for Load or Store

Time	Name	Busy	Op	$V_j$	$V_k$	$Q_j$	$Q_k$	$A$
	Load1	No						
	Load2	Yes	Load					44+Regs[R3]
	Add1	Yes	SUB	Mem[32+Regs[R2]]	Load2			
	Add2	Yes	ADD			Add1	Load2	
	Add3	No						
	Mult1	Yes	MUL		Reg[F4]	Load2		
	Mult2	Yes	DIV	Mem[32+Reg[R2]]		Mult1		

记分牌和保留站相同的地方是都记录了 $Q_j$ 和 $Q_k$ ，即一旦需要的数据被算出来，就通过 $Q_j$ 和 $Q_k$ 捕捉广播数据，这样的做法其实就是重命名，即用保留站的编号而不是寄存器编号来标记数据源。

### • Register result status:

$Q_i$ : indicates which functional unit (FU) will write each register

	F0	F2	F4	F6	F8	F10	F12	...	F30
Cycle 0	$Q_i$			Mult1					
	Data	0.0	2.0	4.0	6.0	8.0	10.0	12.0	30.0

除了保留站数据结构之外，Tomasulo同样要记录寄存器结果状态，和记分牌一样，Tomasulo也会记录寄存器将被哪条指令更新，这个信息在指令寻找源数据时被使用。

## 调度流程

Tomasulo算法的调度分为三个步骤：发射、执行、写回。相比记分牌少了读数这个环节，因为在Tomasulo中指令在发射时就会读数。

- 发射：Tomasulo算法是顺序发射的，即指令按照程序中的顺序一条接一条被发射到保留站。判断能否发射的唯一标准是指令对应通路的保留站是否有空余位置，只要保留站有空余，就可以把指令发射到保留站中。周期结束时会更新保留站和寄存器结果状态表，如果指令有可以读取的数据，就会立刻拷贝到保留站中；寄存器结果状态表中总是存有最新的值，即如果后序指令的目的寄存器和前序指令的目的寄存器重合，那就只保留后序指令的写信息。
- 执行：指令通过拷贝数据和监听CDB获得源数据，然后开始执行，执行可能是多周期的，在执行过程中不改变处理器状态。
- 写回：指令在写回阶段通过CDB总线将数据直通到寄存器堆和各个保留站；周期结束时，根据寄存器结果状态表来更新寄存器堆，并且清除保留站和寄存器结果状态表的信息。

## 案例解读

以 `input1.txt` 为例

- Issue LD, imm $\rightarrow$ A,  $\text{Regs}[R2] \rightarrow V_k$
- Instruction status

Op	dest	j	k	Issue	Exec Comp	Write Result
LD	F6,	34	(R2)	✓1		
LD	F2,	45	(R3)			
MULD	F0,	F2,	F4			
SUBD	F8,	F2,	F6			
DIVD	F10,	F0,	F6			
ADDD	F6,	F8,	F2			

- Reservation Station

Time	Name	Busy	Op	$V_j$	$V_k$	$Q_j$	$Q_k$	A
	Load1	Yes	LD		Regs[R2]=200			34
	Load2	No						
	Add1	No						
	Add2	No						
	Mult1	No						
	Mult2	No						

- Register result status

Cycle	1	$Q_i$	Load1							...	F30
			F0	F2	F4	F6	F8	F10	F12	...	F30
			0.0	2.0	4.0	6.0	8.0	10.0	12.0	30.0	

第一个周期，LD指令发射到存储通路的保留站第一行。在周期结束时，保留站第一行Busy位置1，表明第一行现在存有指令， $V_k$ 直接从寄存器堆中拷贝数据，A是地址偏移立即数。寄存器结果状态中F6被标记，表示F6将被存储通路保留站第一行的指令改写。

- Issue second LD, first LD calculate address

- Instruction status

Op	dest	j	k	Issue	Exec Comp	Write Result
LD	F6, 34 (R2)			✓1		
LD	F2, 45 (R3)			✓2		
MULD	F0, F2, F4					
SUBD	F8, F2, F6					
DIVD	F10, F0, F6					
ADDD	F6, F8, F2					

- Reservation Station

Time	Name	Busy	Op	$V_j$	$V_k$	$Q_j$	$Q_k$	A
	Load1	Yes	LD					234
	Load2	Yes	LD	Regs[R3]=300				45
	Add1	No						
	Add2	No						
	Mult1	No						
	Mult2	No						

- Register result status

	F0	F2	F4	F6	F8	F10	F12	...	F30
Cycle 2	$Q_i$	Load2		Load1					
	Data	0.0	2.0	4.0	6.0	8.0	10.0	12.0	30.0

第二个周期，第一条指令计算存储地址，还没有从存储器中取数，所以没有执行完，因此没有到 Exec Comp段；第二条指令顺利发射，发射到存储通路的保留站第二行。

周期结束时，第一条指令算得存储地址并更新保留站，地址为234；第二条指令从寄存器堆中拷贝源数据，并更新寄存器结果状态表，表示F2将被存储通路的保留站第二行的指令更新。

- Issue MULD, first LD complete, second LD calculate address

- Instruction status

Op	dest	j	k	Issue	Exec Comp	Write Result
LD	F6, 34 (R2)			✓1	✓3	
LD	F2, 45 (R3)			✓2		
MULD	F0, F2, F4			✓3		
SUBD	F8, F2, F6					
DIVD	F10, F0, F6					
ADDD	F6, F8, F2					

- Reservation Station

Time	Name	Busy	Op	$V_j$	$V_k$	$Q_j$	$Q_k$	A
	Load1	Yes	LD					234
	Load2	Yes	LD					345
	Add1	No						
	Add2	No						
	Mult1	Yes	MULTD	Regs[F4]=4.0	Load2			
	Mult2	No						

- Register result status

	F0	F2	F4	F6	F8	F10	F12	...	F30
Cycle 3	$Q_i$	Mult1	Load2	Load1					
	Data	0.0	2.0	4.0	6.0	8.0	10.0	12.0	30.0

第三个周期，第一条指令顺利读数，渡过Exec Comp段；第二条指令计算存储地址；第三条指令顺利发射。

周期结束时，第一条指令不更新保留站和寄存器结果状态表；第二条指令更新保留站，把计算得到的地址更新到A；第三条指令拷贝源数据，并标记 $Q_j$ 位Load2，表示第一个源数据现在不在寄存器堆中，而由存储通路保留站第二行的指令算出，此时Load2还没有到写回阶段，所以Mult1只能等待，另外第三条指令也会更新寄存器结果状态表。

- Issue SUBD, first LD write results, second LD complete

- Instruction status**

Op	dest	j	k	Issue	Exec Comp	Write Result
LD	F6, 34 (R2)			✓1	✓3	✓4
LD	F2, 45 (R3)			✓2	✓4	
MULD	F0, F2, F4				✓3	
SUBD	F8, F2, F6				✓4	
DIVD	F10, F0, F6					
ADDD	F6, F8, F2					

- Reservation Station**

Time	Name	Busy	Op	$V_j$	$V_k$	$Q_j$	$Q_k$	A
	Load1	No						
	Load2	Yes	LD					345
	Add1	Yes	SUBD		Regs[F6]=234.0	Load2		
	Add2	No						
	Mult1	Yes	MULTD		4.0	Load2		
	Mult2	No						

- Register result status**

	F0	F2	F4	F6	F8	F10	F12	...	F30
Cycle 4	$Q_i$	Mult1	Load2			Add1			
	Data	0.0	2.0	4.0	MEM[234]=234.0	8.0	10.0	12.0	30.0

第四个周期，第一条指令渡过写回阶段，通过CDB广播自己取到的数据；第二条指令在计算完地址之后进入存储通路，开始取数；第三条指令还在等待Load2，只能停留在保留站中；第四条指令发射。

周期结束时，第一条指令清除自己在保留站中的痕迹，Busy位置0，表示这一行不存有指令信息，并消除寄存器结果状态表中的记录，表明F6现在是最新值；第二条指令不更新状态；第三条指令等到数据，不更新状态；第四条指令拷贝数据，标记Load2，等待Load2的执行结果，并更新寄存器结果状态表。

- Issue DIVD, second LD write result (broadcast!)

- Instruction status**

Op	dest	j	k	Issue	Exec Comp	Write Result
LD	F6, 34 (R2)			✓1	✓3	✓4
LD	F2, 45 (R3)			✓2	✓4	✓5
MULD	F0, F2, F4				✓3	
SUBD	F8, F2, F6				✓4	
DIVD	F10, F0, F6				✓5	
ADDD	F6, F8, F2					

- Reservation Station**

Time	Name	Busy	Op	$V_j$	$V_k$	$Q_j$	$Q_k$	A
	Load1	No						
	Load2	No						
2	Add1	Yes	SUBD	345.0	234.0			
	Add2	No						
10	Mult1	Yes	MULTD	345.0	4.0			
	Mult2	Yes	DIVD		Regs[F6]=234.0	Mult1		

- Register result status**

	F0	F2	F4	F6	F8	F10	F12	...	F30
Cycle 5	$Q_i$	Mult1			Add1	Mult2			
	Data	0.0	MEM[345]=345.0	4.0	234.0	8.0	10.0	12.0	30.0

第五个周期，第二条指令到写回阶段，通过CDB总线广播数据；第三条指令通过CDB总线抓取到了源数据，下一个周期就会开始执行，图中Mult1前面的数字10表示这条指令接下来将用十个周期完成执行；同理第四条指令通过CDB总线抓取到了源数据，下一个周期就会开始执行；第五条指令因为乘法单元的保留站有空余，所以可以发射。

周期结束时，第二条指令清除保留站信息，清除寄存器结果状态表；第三条指令和第四条指令更新保留站，将CDB总线上的数据拷贝到保留站中，至此，这两条指令数据准备完毕，马上就可以执行了；第五条指令更新保留站和寄存器结果状态表。Q<sub>j</sub>显示该指令需要等到乘法保留站第一行的指令的结果。

- Issue ADDD, MULD and SUBD execution

- Instruction status

Op	dest	j	k	Issue	Exec Comp	Write Result
LD	F6,	34	(R2)	✓1	✓3	✓4
LD	F2,	45	(R3)	✓2	✓4	✓5
MULD	F0,	F2,	F4	✓3		
SUBD	F8,	F2,	F6	✓4		
DIVD	F10,	F0,	F6	✓5		
ADDD	F6,	F8,	F2	✓6		

- Reservation Station

Time	Name	Busy	Op	V <sub>j</sub>	V <sub>k</sub>	Q <sub>j</sub>	Q <sub>k</sub>	A
	Load1	No						
	Load2	No						
1	Add1	Yes	SUBD	345.0	234.0			
	Add2	Yes	ADDD		Regs[F2]=345.0	Add1		
9	Mult1	Yes	MULTD	345.0	4.0			
	Mult2	Yes	DIVD		234.0		Mult1	

- Register result status

		F0	F2	F4	F6	F8	F10	F12	...	F30
Cycle 6	Q <sub>i</sub>	Mult1		Add2	Add1	Mult2				
	Data	0.0	345.0	4.0	234.0	8.0	10.0	12.0		30.0

第六个周期，第三第四条指令开始执行，但是远没有执行完毕，因此显示在Exec Comp段上；第五条指令在等到Mult1，因此停留在保留站中；第六条指令因为加法保留站还有空闲，所以可以发射。

周期结束时，第三第四第五条指令不更新状态，第六条指令拷贝数据，更新寄存器结果状态表。

后续周期同理，不再赘述。

## Speculative Tomasulo 算法实现

### 输出格式

## Reorder Buffer

Reorder buffer					
Entry	Busy	Instruction	State	Destination	Value
1	No	f1d	f6,32(x2)	Commit	f6
2	No	f1d	f2,44(x3)	Commit	f2
3	Yes	fmul.d	f0,f2,f4	Write result	f0
4	Yes	fsub.d	f8,f2,f6	Write result	f8
5	Yes	fdiv.d	f0,f0,f6	Execute	f0
6	Yes	fadd.d	f6,f8,f2	Write result	f6

Reservation stations							
Name	Busy	Op	Vj	Vk	Qj	Qk	Dest A
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	

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

```

cycle_n;
entry1 : No ,fld f6 32(x2),Commit,f6,Mem[32+Regs[x2]];
entry2 : No ,fld f2 44(x3),Commit,f2,Mem[44+Regs[x3]];
entry3 : Yes,fmul.d f0,f2,f4,Write result,f0,#2 × Regs[f4];
entry4 : Yes,fsub.d f8,f2,f6,Write result,f8,#2-#1;
entry5 : Yes,fdiv.d f0,f0,f6,Execute,f0,;
entry6 : Yes,fadd.d f6,f8,f2,Write result,f6,#4+#2;
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;
Reorder:f0: 3;F1:;.. .;F10: 5;
Busy:f0: Yes;F1:No;.. .;F10: Yes;

```

## 代码解读

具体代码详见 [Tomasulo.py](#) 文件

### ReservationStation 类

用于模拟处理器的保留站（Reservation Station）的行为，保留站用来执行指令，每个保留站与特定的执行部件相关联。

#### `__init__` 方法:

为Load1、Load2、Add1、Add2、Add3、Mult1、Mult2创建 [Station](#) 实例

#### `insert` 方法:

- 向保留站中插入指令
- 根据指令的类型 ([inst\\_type](#)) 确定需要插入的执行部件
- 遍历执行部件，找到第一个可用的执行部件并插入指令，返回1表示插入成功，返回-1表示插入失败

#### `update` 方法:

- 更新保留站中的状态。
- 遍历所有执行部件，分别进行写更新 ([write\\_update](#)) 和读更新 ([read\\_update](#)) 操作

#### `show` 方法:

- 显示保留站的状态

## Station 类

用于表示处理器中的执行部件的状态和操作

### `__init__` 方法:

- 创建 `buffer`，用于存储执行部件的状态信息
- 将 `buffer[0]` 初始化为0，表示该执行部件当前不可用
- 初始化 `time` 为0，用于跟踪执行部件的剩余执行时间

### `is_available` 方法:

- 判断执行部件是否可用，根据 `buffer[0]` 的值为0或非0来判断。

### `insert` 方法:

- 插入指令到执行部件中
- 根据指令的类型和操作数，设置 `buffer` 中的相应字段
- 根据指令的类型和操作数，更新相关寄存器（`rrs`）的状态
- 返回1表示插入成功

### `write_update` 方法:

- 在写结果阶段更新执行部件和寄存器的状态

### `read_update` 方法:

- 在读取结果阶段更新执行部件和寄存器的状态。

### `show` 方法:

- 显示执行部件的状态
- 如果执行部件不可用，则返回"No,,,,,\n"
- 如果执行部件可用，则返回"Yes"和 `buffer` 中的状态信息

## ReorderBuffer 类

用于模拟处理器中的重排序缓冲区（Reorder Buffer），其中，重排序缓冲区用来处理乱序指令执行，包含多个 `Entry` 实例

### `__init__` 方法:

- 初始化重排序缓冲区的属性，包括 `index`（当前索引位置）、`length`（缓冲区长度）和 `entry`（包含多个 `Entry` 实例的列表）

- 循环创建指定长度的 `Entry` 实例，并存储在 `entry` 列表中

`show` 方法：

- 显示重排序缓冲区的状态

`insert` 方法：

- 向重排序缓冲区中插入指令
- 将指令插入到当前索引位置对应的 `Entry` 实例中，并更新当前索引位置

`check` 方法：

- 检查重排序缓冲区中是否所有的 `Entry` 都已完成
- 遍历缓冲区中的每个 `Entry`，如果发现有一个实例的 `buffer[0]` 为1（即未完成），则返回 `False`，否则返回 `True`

`show_result` 方法：

- 显示已完成指令的执行结果
- 遍历缓冲区中的每个 `Entry` 实例，如果发现某个实例的 `times[0]` 为0（表示已完成），则停止遍历并返回已完成指令的结果字符串

## `Entry` 类

表示重排序缓冲区中的一个条目，其中，每个 `Entry` 实例包含了用于存储指令和执行状态的属性

`__init__` 方法：

- 每个 `Entry` 实例设置了初始索引（`index`）、`buffer` 列表和 `times` 列表
- `buffer[0]` 初始化为0，表示该条目未被占用
- `times` 列表用于存储不同阶段的时钟周期

`insert` 方法：

- 向条目中插入指令。
- 更新相关寄存器状态，并记录插入时的时钟周期。

`show` 方法：

- 显示条目的状态。

`show_result` 方法：

- 显示已完成指令的执行结果

### `check` 方法:

- 更新指定阶段的时钟周期
- 根据传入的 `index` 参数，更新 `times` 列表中对应位置的时钟周期

### `ReorderReservationStations` 类

表示重排序保留站，重排序保留站用于存储浮点操作数（F0到F10）的状态信息，包括是否被占用、状态码和当前数值

#### `__init__` 方法:

- 设置了初始时钟周期数（`cycle` 为0）、存储浮点寄存器的占用状态（`busy`）、状态码（`status`）和当前数值（`value`）

#### `is_available` 方法:

- 检查指定浮点寄存器是否可用
- 如果该浮点寄存器未被占用，或者被占用但对应的重排序缓冲区条目已经写入结果，则返回 `True`；否则返回 `False`

#### `show` 方法:

- 显示重排序保留站的状态
- 生成包含重排序站状态、是否 `busy` 以及对应浮点寄存器的信息并返回

## 部分输出样例

完整结果请查看 `output1.txt` 和 `output2.txt`

### `output1.txt`

```
cycle_1;
entry1 : Yes,LD F6,34+,R2,Issue,F6,;
entry2 : No,....;
entry3 : No,....;
entry4 : No,....;
entry5 : No,....;
entry6 : No,....;
Load1 : Yes,LD,,Mem[34+Regs[R2]],,,F6;
Load2 : No,....;
Add1 : No,....;
Add2 : No,....;
Add3 : No,....;
Mult1 : No,....;
Mult2 : No,....;
Reorder:F0:;F1:;F2:;F3:;F4:;F5:;F6: 1;F7:;F8:;F9:;F10:
Busy:F0: No;F1: No;F2: No;F3: No;F4: No;F5: No;F6: Yes;F7: No;F8: No;F9: No;F10: No
                                         LD F6,34+,R2:1,3,5,6;
                                         LD F2,45+,R3:3,4,6,7;
                                         MULTD F0,F2,F4:4,7,17,18;
                                         SUBD F8,F6,F2:5,7,9,19;
                                         DIVD F10,F0,F6:6,18,38,39;
                                         ADDD F6,F8,F2:7,10,12,40;
```

### output2.txt

```
cycle_1;
entry1 : Yes,LD F2,0,R2,Issue,F2,;
entry2 : No,....;
entry3 : No,....;
entry4 : No,....;
entry5 : No,....;
entry6 : No,....;
entry7 : No,....;
entry8 : No,....;
Load1 : Yes,LD,,Mem[Regs[R2]],,,F2;
Load2 : No,....;
Add1 : No,....;
Add2 : No,....;
Add3 : No,....;
Mult1 : No,....;
Mult2 : No,....;
Reorder:F0:;F1:;F2: 1;F3:;F4:;F5:;F6:;F7:;F8:;F9:;F10:
Busy:F0: No;F1: No;F2: Yes;F3: No;F4: No;F5: No;F6: No;F7: No;F8: No;F9: No;F10: No
                                         LD F2,0,R2:2,3,5,6;
                                         LD F4,0,R3:3,4,6,7;
                                         DIVD F0,F4,F2:4,7,27,28;
                                         MULTD F6,F0,F2:5,28,38,39;
                                         ADDD F0,F4,F2:6,7,9,40;
                                         SD F6,0,R3:7,8,10,41;
                                         MULTD F6,F0,F2:8,10,20,41;
                                         SD F6,0,R1:9,10,12,42;
```

## 附加问题

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

相对于Scoreboard算法，Tomasulo算法具有以下优点：

- l. 避免结构冲突：Tomasulo算法通过使用重命名寄存器和功能单元的方式，可以避免结构冲突。它允许多个指令同时使用同一个功能单元，而无需等待其他指令完成。
- l. 避免数据冲突：Tomasulo算法通过使用公共数据总线和重命名寄存器的方式，可以避免数据冲突。指令可以读取最新的结果，而不需要等待之前的指令。
- l. 高度并行：Tomasulo算法支持乱序执行指令，能够充分利用处理器中的资源，提高指令级并行性。它可以同时执行多条指令，从而加快程序的执行速度。
- l. 动态调度：Tomasulo算法采用动态调度，根据操作数的就绪状态和功能单元的可用性，动态选择指令执行，提高了程序的执行效率。

缺点：

- l. 硬件开销：Tomasulo算法需要额外的硬件支持，如重命名寄存器和状态表，以及公共数据总线等。这些硬件开销会增加处理器的成本，并且需要更多的面积和功耗。
- l. 额外的复杂性：Tomasulo算法相对于Scoreboard算法而言更为复杂，实现和调试的难度更大。需要更多的控制逻辑和状态维护，增加了设计和验证的复杂性。
- l. 无法解决控制冲突：Tomasulo算法主要解决的是数据冲突和结构冲突，而对于控制冲突（如分支指令）的处理并没有特别的优势。在存在大量控制冲突的情况下，Tomasulo算法的性能可能会下降。

## (2) 简要介绍引入重排序改进 **Tomasulo** 的原理。

引入重排序是为了改进Tomasulo算法的性能，特别是在处理控制冲突（如分支指令）时。重排序的目标是通过改变指令执行的顺序，尽量减少控制冲突对程序性能的影响。

原始的Tomasulo算法是按照指令的顺序依次执行的，因此在遇到分支指令时，需要等待分支指令的结果才能确定执行路径，这将导致一定的延迟和性能损失。

重排序的原理是在编译器或者硬件层面，对指令序列进行转换和调整，使得原本相互依赖的指令可以并行执行，从而减少控制冲突的影响。通过重排序，指令的执行顺序可以重新调整，使得程序可以更好地利用处理器资源，减少控制冲突的影响，提高程序的执行效率和性能。

重排序的步骤：

- l. 提前执行非依赖指令：如果某个指令不依赖于前面的指令的结果，并且没有数据和结构冲突，那么可以提前执行它，而不必等待前面的指令完成。
- l. 延迟执行依赖指令：对于依赖于前面指令结果的指令，可以将它们延迟到不会产生冲突的时候执行。通过重命名寄存器和功能单元的方式，可以避免数据和结构冲突，并允许指令乱序执行。
- l. 预测执行路径：在遇到分支指令时，可以使用分支预测技术来预测分支的执行路径。根据预测结果，可以提前执行预测路径上的指令，减少分支带来的延迟。

### (3) 请分析重排序缓存的缺点。

- 1. 重排序缓存需要额外的硬件资源来实现指令的重排和结果的重组。这包括存储单元、状态寄存器和控制逻辑等。因此，引入重排序缓存会增加处理器的成本、面积和功耗。
- 2. 重排序缓存的容量是有限的，它需要存储指令的操作数、结果和状态信息。如果重排序缓存的容量不足，可能会导致指令无法进入缓存，从而降低指令的乱序执行能力。
- 3. 指令在重排序缓存中进行重排和重组，可能会引入额外的延迟和冲突。当有多条指令竞争同一个缓存位置时，需要进行冲突检测和解决，这可能会导致延迟增加，并且需要更复杂的控制逻辑。
- 4. 重排序缓存的引入可能增加异常处理的复杂性。由于指令的执行顺序被重排，当发生异常时，可能需要回滚缓存中的指令并保持指令的顺序一致性，这会增加异常处理的开销和难度。
- 5. 重排序缓存对于内存一致性的处理是一个挑战。由于指令的乱序执行，可能会导致对内存的读写顺序发生变化，需要确保程序的内存访问满足一致性要求。