
2023 年秋季计算机体系结构——大作业

Speculative Tomasulo 算法实现

21307099 李英骏

liyj323@mail2.sysu.edu.cn

Abstract

Tomasulo 算法的原始结构如下图 (1) 所示

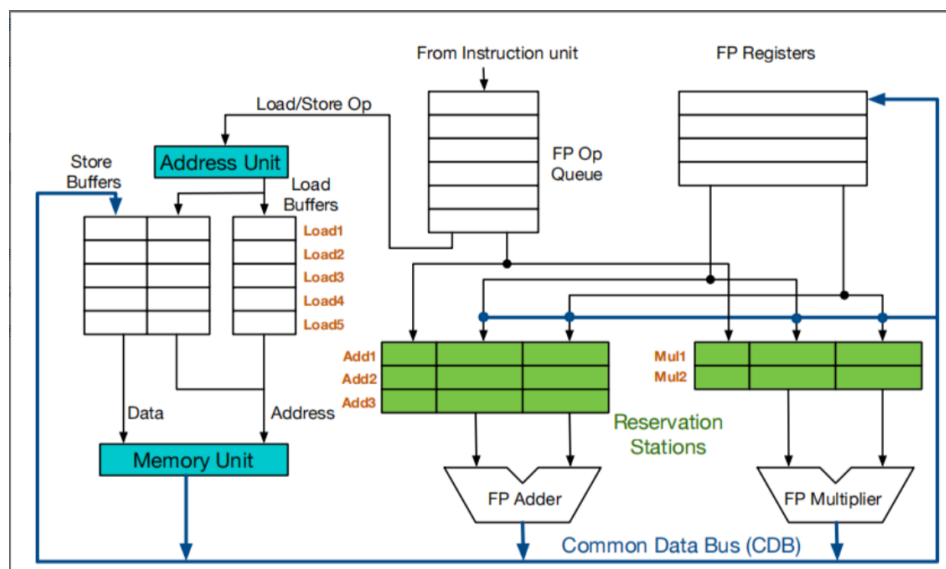


图 1: 原始 Tomasulo

其缺点在于:

- 一个周期只能写回一条指令
- 在多发射情形下, 存储指令之间可以有数据冲突, 需要额外的逻辑进行调度
- 不能保证按序提交, 因此难以处理分支指令, 也无法实现精确中断。
- ...

因此设计人员提出了重排序缓冲的概念, 利用这一概念, 可以改进 Tomasulo 算法 (如图2), 从而实现处理器的按序发射-乱序执行-按序提交。

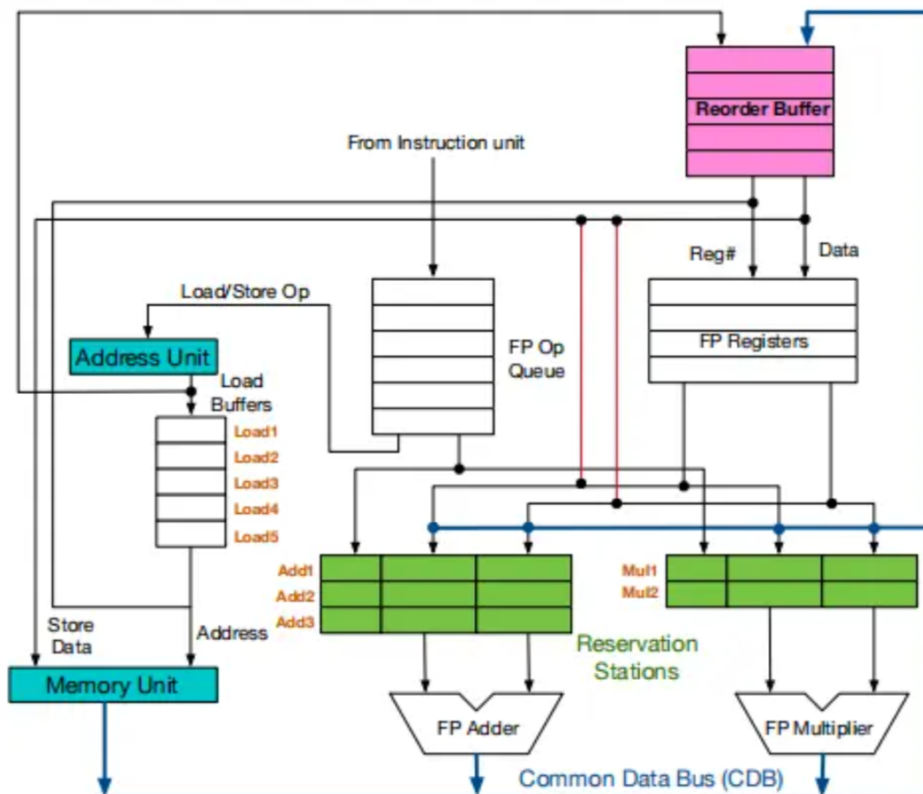


图 2: Speculative Tomasulo

和原始的 Tomasulo 结构 (如图1) 相比, 结合 ROB 的 Tomasulo 主要有三点改动:

- 增加了 Reorder Buffer (即 ROB);
- CDB 总线不再直通逻辑寄存器堆, 而是直通 ROB;
- 指令需要从 ROB 读取数据。

重排序缓冲区 (ROB) 以类似 FIFO 结构的方式使 Tomasulo 算法实现了指令的顺序提交, 指令在被发射时加入队列, 在提交时出队。这不仅方便了处理器处理中断异常, 也简化了分支预测的处理过程。ROB 还可作为寄存器重命名中的物理寄存器使用, 从而使 Tomasulo 算法中的保留站 (此前作为物理寄存器使用) 得以释放, 进而提高了处理器的发射效率。

1 架构说明

根据作业要求，架构图如下：

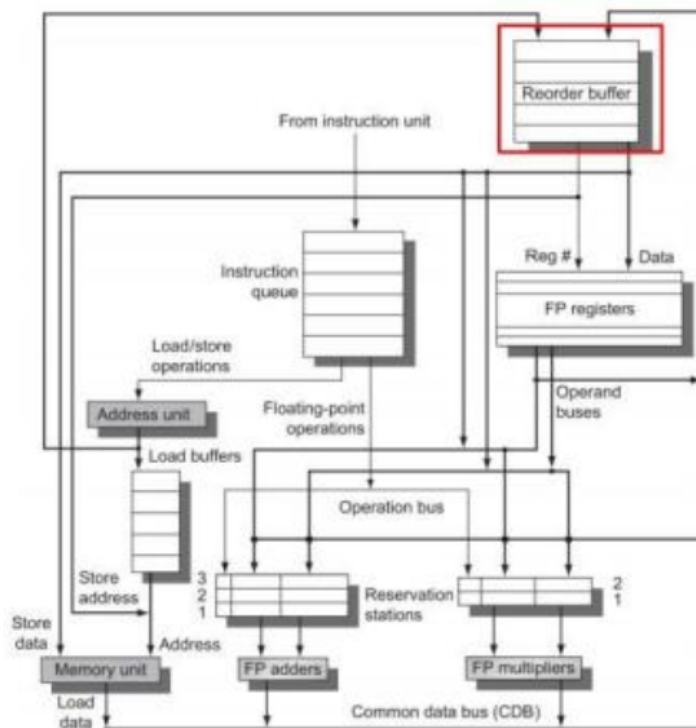


图 3: Speculative Tomasulo 仿真器架构图

除了 ROB，我们还可以看到以下部件：

1. 指令队列（Instruction queue）：存储待处理的指令。
2. 浮点寄存器（FP registers）：存储浮点数值的寄存器。
3. 操作数总线（Operand bus）：用于传送操作数至不同的功能单元。
4. 浮点加法器（FP adders）和浮点乘法器（FP multipliers）：执行浮点加法和乘法操作的功能单元。
5. 保留站（Reservation stations）：暂存那些已经分派但尚未执行的指令。
6. 公共数据总线（CDB, Common Data Bus）：功能单元用来将计算结果发送回重排序缓冲区或寄存器的总线。
7. 地址单元和加载/存储缓冲区（Load/Store buffers）：用于处理内存访问操作。
8. 内存单元（Memory Unit）和地址单元（Address Unit）

2 代码实现

我们按顺序实现以下部件：

1. 重排序缓冲区（ROB）
2. 公共数据总线（CDB, Common Data Bus）
3. 保留站（Reservation stations）
4. 地址单元和加载/存储缓冲区（Load/Store buffers）
5. 浮点加法器（FP adders）和浮点乘法器（FP multipliers）

2.1 额外假设

我们假设指令加入 buffer 和从 buffer 删除也消耗一个周期

2.2 ROB

Reorder Buffer（ROB）有以下表项：

条目	繁忙	指令	状态	目的地	值
----	----	----	----	-----	---

因此我们定义如下的 ROB 类：

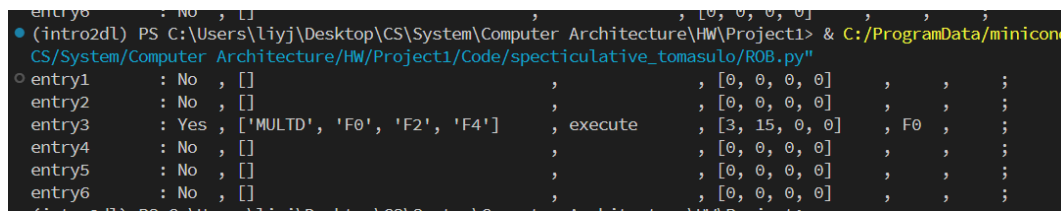
```
1 class Reorder_Buffer:
2     class Buffer_Unit:
3         def __init__(self, entry=0) -> None:
4             self.entry = entry # 条目号
5             self.busy = False # 繁忙位
6             self.instruction = [] # 指令
7             self.states_name = None # 状态
8             self.states_num = [0, 0, 0, 0] # 状态结束的周期
9             self.destination = '' # 目的地
10            self.value = '' # 值
11
12        def display(self): # 显示自身状态
13            print(f'entry{self.entry:<7}', end = ' : ')
14            print(f'{"Yes" if self.busy else "No":<4}', end=', ')
15            print(f'{str(self.instruction):<32}', end=', ')
16            if self.states_name != None:
17                print(f'{self.states_name:<13}', end=', ')
```

```

18         else:
19             print(' ' * 13, end=', ')
20             print(f'{str(self.states_num):<17}', end=', ')
21             print(f'{self.destination:<4}', end=', ')
22             print(f'{str(self.value):<4}' + ';;')
23
24     def __init__(self, size=6):
25         self.buffers = [self.Buffer_Unit(entry=i+1) for i in range(
26             size)]
27     def display(self):
28         for unit in self.buffers:
29             unit.display()

```

打印的位数是用最长的可能位数 +1 计算的，数字按最多 2 位计。打印测试如下图：



```

entry0 : No , [] , [0, 0, 0, 0] , , ;
entry1 : No , [] , [0, 0, 0, 0] , , ;
entry2 : No , [] , [0, 0, 0, 0] , , ;
entry3 : Yes , ['MULTD', 'F0', 'F2', 'F4'] , execute , [3, 15, 0, 0] , F0 , ;
entry4 : No , [] , [0, 0, 0, 0] , , ;
entry5 : No , [] , [0, 0, 0, 0] , , ;
entry6 : No , [] , [0, 0, 0, 0] , , ;

```

图 4: ROG Display Test

2.3 公共数据总线（CDB）

```

1 class CDB:
2     def __init__(self):
3         self.data = 0
4         self.source_entry = -1

```

模拟公共数据总线的类。

`self.data`: 这个属性用来存储被传输的数据。在这个简化的模型中，`data` 被初始化为 0，表示最初没有数据在总线上。

`self.source_entry`: 这个属性用于标识产生 `data` 的源头（即数据来源的保留站的编号）。它被初始化为 -1，表示最初没有任何保留站向 CDB 发送数据。

2.4 保留站

```
1 class Reservation_Station:
2     def __init__(self, type="", id=0, CDB=None, ROB=None):
3         self.type = type
4         self.id = id
5         self.busy = False
6         self.Op = ''
7         self.Vj = '' # 拷贝可读取的数据
8         self.Vk = ''
9         self.Qj = '' # 记录尚不能读取的数据将由哪条指令算出
10        self.Qk = ''
11        self.Dest = 0 # 目的ROB编号
12        self.cdb = CDB
13        self.rob = ROB
14
15    def write_in(self):
16        if self.busy == 'No':
17            return
18        if self.Qj == self.cdb.source_entry:
19            self.Qj = ''
20            self.Vj = copy.copy(self.cdb.data)
21        if self.Qk == self.cdb.source_entry:
22            self.Qk = ''
23            self.Vk = copy.copy(self.cdb.data)
24
25    def display(self):
26        print(f'{self.type:<4}{self.id} : ', end= ' ')
27        print(f'{"Yes" if self.busy else "No":<4}', end=', ')
28        print(f'{self.Op:<8}', end=', ')
29        print(f'{str(self.Vj):<30}', end=', ')
30        print(f'{str(self.Vk):<30}', end=', ')
31        print(f'{str(self.Qj):<3}', end=', ')
32        print(f'{str(self.Qk):<3}', end=', ')
33        print(f'{str(self.Dest):<4};')
```

根据 PPT 上的表项配置变量。

self.Qj 表示第一个源操作数所依赖的那个尚未完成的操作的保留站编号。如果 self.Qj 等于 self.cdb.source_entry，这意味着所依赖的操作已经完成，并且其结果已经在公共

数据总线上可用。self.Qj 被清空，表示不再依赖任何未完成的操作。
self.Vj 被设置为 self.cdb.data 的副本，这是因为 self.cdb.data 包含了所依赖操作的结果。这里使用 copy.copy 是为了确保 self.Vj 获得的是数据的副本，避免直接修改 self.cdb.data。

类似地，self.Qk 表示第二个源操作数所依赖的那个尚未完成的操作的保留站编号。

2.5 地址单元和加载/存储缓冲区 (Load/Store buffers)

加法器和乘法器的实现基本类似，只需注意 delay 的区别（此处可以用一个 flag 位在每次执行时翻转，产生 1 的 delay。但 delay 非 1 时不能这样做）

```
1  class Load_Buffer:
2  class Load_Buffer_Unit(Reservation_Station):
3      def __init__(self,type='load',id=0, CDB=None, ROB=None,
4          register_states=None):
5          super().__init__(type=type,id=id,CDB=CDB,ROB=ROB)
6          self.address = ''
7          self.flag = 0
8          self.register_states = register_states_
9
10     def execute(self,cycle):
11         if not self.busy:
12             return
13         rob_entry = self.rob.buffers[self.Dest]
14         rob_entry.update_state(StateName.EXECUTE, 1)
15         print(self.Qj,self.Qk)
16         print(self.Qj,self.Qk)
17         print(self.Qj, self.Qk)
18         print(self.Qj, self.Qk)
19         if self.Qj == '' and self.Qk == '' :
20             self.address = self.Vj+self.Vk
21             if self.Op == "LD" or (self.Op == "SD" and self.
22                 register_states[f'F{self.Dest+1}']== "No"):
23                 self.flag = 1
24
25     def __init__(self, CDB=None, ROB=None, register_states=None, memory=
26         None, registers=None):
27         self.buffers = [self.Load_Buffer_Unit(id=i, CDB=CDB,ROB=ROB,
28             register_states_=register_states) for i in range(1,3)]
```

```

26     self.head = 0
27     self.memory = memory
28     self.ROB = ROB
29     self.CDB = CDB
30     self.register_states = register_states
31     self.registers = registers
32     self.count=0
33 def write_in(self):
34     for i in range(2): self.buffers[i].write_in()
35
36 def execute(self,cycle):
37     # print('cycle',cycle)
38     if self.count==6: self.head = 1-self.head
39     running_buffer = self.buffers[self.head]
40     # flag标志位为1（表示地址计算已完成）
41     #if running_buffer.Op == 'SD' and cycle >30 : running_buffer.
42         flag = 1
43
44     if running_buffer.flag:
45
46         if running_buffer.Op == 'LD':
47             #for i in range(CDB.source_entry):
48                 if running_buffer.Dest > self.CDB.source_entry:
49                     '''
50                     如果头部加载单元是一个加载指令 并且该指令的目的地小于通
51                     用数据总线（CDB）上当前的源指令索引，那么进行以下操
52                     作：
53
54                     从内存地址中读取数据并将其写入CDB
55                     设置CDB的数据来源为当前加载单元的目的地
56                     更新执行阶段结束的周期数
57                     重新初始化当前加载单元并更新头部指针
58                     '''
59                     self.CDB.data = self.memory[running_buffer.address]
60                     self.CDB.source_entry = running_buffer.Dest
61
62                 elif running_buffer.Op == 'SD':
63                     temp = self.ROB.buffers[running_buffer.Dest].instruction
64                         [1]

```



```

61         if type(temp) != int:
62             temp = self.registers[temp]
63         self.memory[running_buffer.address] = temp
64         self.ROB.buffers[running_buffer.Dest].value = 0
65
66         self.ROB.buffers[running_buffer.Dest].states_num[1] = cycle
67         running_buffer.__init__(id=running_buffer.id, CDB=self.CDB, ROB
        =self.ROB, register_states=self.register_states)
68
69         self.head = 1 - self.head
70         self.count+=1
71         self.buffers[0].execute(cycle)
72         self.buffers[1].execute(cycle)
73
74     def display(self):
75         self.buffers[0].display()
76         self.buffers[1].display()
77
78     def find_no_busy(self):
79         for i in range(2):
80             if not self.buffers[i].busy:
81                 return i
82         return -1

```

2.6 主函数

```

1
2     while top != bottom:
3         # 逆序执行
4         # 删除标记
5         top, delete_sign, insert_sign = handle_delete_sign(
6             rob, top, bottom, delete_sign, insert_sign)
7         if top == None: break
8
9         # commit
10        delete_sign = commit(rob, top, registers, registers_name,
11                               registers_state, result, delete_sign, cycle)
12
13

```

```

14         # 写结果
15         write_result(cdb, rob, loads, fp_adders, fp_multipliers, cycle)
16         #执行
17         loads.execute(cycle)
18         fp_adders.execute(cycle)
19         fp_multipliers.execute(cycle)
20
21         # 发射
22         top = issue_instructions(rob, top, bottom, loads, fp_adders,
23                                 fp_multipliers, registers_name,
24                                 registers_state, registers, cycle)
25
26         line, bottom, insert_sign = handle_insert_sign(
27             line, bottom, instructions, rob, insert_sign)
28
29         display_status(cycle, rob, loads, fp_adders,
30                        fp_multipliers, registers_name, registers_state)
31         cycle += 1
32
33         display_result(result=result)

```

处理删除标记 (delete_sign): 当 delete_sign 为真时, 重置位于 top 索引的重排序缓冲区 (ROB) 条目。top 索引增加, 以指向下一个条目。重置 delete_sign 并设置 insert_sign, 允许新的条目插入。如果 top 和 bottom 相等, 表示 ROB 为空, 循环终止。

处理提交 (commit): 如果 rob.buffers[top % 6].value 不为空, 表示该指令已经完成, 可以提交。更新寄存器值 (除非是 SD 指令), 释放相应资源。将结果添加到 result 列表中, 并准备删除该条目。

写结果 (write_result): 如果 cdb.source_entry 不为 -1, 表示 CDB 有数据可供写入。更新相应的 ROB 条目, 以及所有正在等待这个数据的保留站。

执行功能单元操作 (execute_units): 执行 loads, fp_adders, fp_multipliers 的 execute 方法。处理发射指令 (issue_instructions):

遍历 ROB, 寻找可以发射的指令。如果找到, 更新相关状态, 并将指令分配给相应的功能单元。

处理插入标记 (insert_sign): 如果 insert_sign 为真且还有未处理的指令, 将新指令添加到 ROB 中。

3 结果展示

```

915
916 cycle : 39
917 | busy instruction | states_name | dest | value |
918 | entry1 : No , | , issue , , ; |
919 | entry2 : No , | , issue , , ; |
920 | entry3 : No , | , issue , , ; |
921 | entry4 : No , | , issue , , ; |
922 | entry5 : No , | , issue , , ; |
923 | entry6 : Yes , ADDD, F6, F8, F2 | , commit , F6 , 46 ; |
924 | busy op Vj | Vk | Qj | Qk | dest |
925 | load1 : No , | , , , 0 ; |
926 | load2 : No , | , , , 0 ; |
927 | add 1 : No , | , , , 0 ; |
928 | add 2 : No , | , , , 0 ; |
929 | add 3 : No , | , , , 0 ; |
930 | mult1 : No , | , , , 0 ; |
931 | mult2 : No , | , , , 0 ; |
932 registers_name : F0: , F1: , F2: , F3: , F4: , F5: , F6: , F7: , F8: , F9: , F10: ,
933 registers_state : F0:No , F1:No , F2:No , F3:No , F4:No , F5:No , F6:No , F7:No , F8:No , F9:No , F10:No ,
934
935 Finally result :
936 ['LD', 'F6', 34, 'R2'] : [1, 3, 4, 5] 46
937 ['LD', 'F2', 45, 'R3'] : [2, 4, 5, 6] 45
938 ['MULTD', 'F0', 'F2', 'F4'] : [3, 15, 16, 17] 720
939 ['SUBD', 'F8', 'F6', 'F2'] : [4, 7, 8, 18] 1
940 ['DIVD', 'F10', 'F0', 'F6'] : [5, 36, 37, 38] 15.652173913043478
941 ['ADDD', 'F6', 'F8', 'F2'] : [6, 10, 11, 39] 46
942

```

图 5: Output1

```

1308 cycle : 42
1309 | busy instruction | states_name | dest | value |
1310 | entry1 : No , | , issue , , ; |
1311 | entry2 : Yes , SD, F6, 0, R1 | , commit , F6 , 0 ; |
1312 | entry3 : No , | , issue , , ; |
1313 | entry4 : No , | , issue , , ; |
1314 | entry5 : No , | , issue , , ; |
1315 | entry6 : No , | , issue , , ; |
1316 | busy op Vj | Vk | Qj | Qk | dest |
1317 | load1 : No , | , , , 0 ; |
1318 | load2 : No , | , , , 0 ; |
1319 | add 1 : No , | , , , 0 ; |
1320 | add 2 : No , | , , , 0 ; |
1321 | add 3 : No , | , , , 0 ; |
1322 | mult1 : No , | , , , 0 ; |
1323 | mult2 : No , | , , , 0 ; |
1324 registers_name : F0: , F1: , F2: , F3: , F4: , F5: , F6: , F7: , F8: , F9: , F10: ,
1325 registers_state : F0:No , F1:No , F2:No , F3:No , F4:No , F5:No , F6:No , F7:No , F8:No , F9:No , F10:No ,
1326
1327 Finally result :
1328 ['LD', 'F2', 0, 'R2'] : [1, 3, 4, 5] 46
1329 ['LD', 'F4', 0, 'R3'] : [2, 4, 5, 6] 46
1330 ['DIVD', 'F0', 'F4', 'F2'] : [3, 25, 26, 27] 1.0
1331 ['MULTD', 'F6', 'F0', 'F2'] : [4, 36, 37, 38] 46.0
1332 ['ADDD', 'F0', 'F4', 'F2'] : [5, 8, 9, 39] 92
1333 ['SD', 'F6', 0, 'R3'] : [6, 39, 0, 40] 0
1334 ['MULTD', 'F6', 'F0', 'F2'] : [26, 37, 38, 41] 4232
1335 ['SD', 'F6', 0, 'R1'] : [27, 40, 0, 42] 0
1336

```

图 6: Output2

4 附加问题

4.1 Tomasulo 算法

- Tomasulo 算法的优点：
 - 动态调度：可以在运行时动态地重新排序指令，减少数据冲突和结构冲突。
 - 寄存器重命名：“写后写”和“读后写”冒险不是真冒险，没必要为他们阻塞指令的流动。通过重命名寄存器来减少 WAR 和 WAW 冲突。
 - 更高的资源利用率：而且记分牌为了乱序执行指令，在碰到写后写、读后写这两个冒险的时候也会暂停流水线，而这其实是不必要的。由于 Tomasulo 动态调度和寄存器重命名，更有效地利用硬件资源。
 - 适用于超标量架构：适合在一个周期内发射多条指令的处理器设计。
- Tomasulo 算法的缺点：
 - 硬件复杂性：实现 Tomasulo 算法需要更复杂的硬件支持。
 - 功耗和成本：由于硬件复杂性，功耗和成本相比 Scoreboarding 更高。
 - 每一个周期只能写回一条指令，还要增加额外的控制逻辑使得多条指令变得有序
 - 存储指令之间也会发生数据冲突
 - Tomasulo 算法没办法实现精确中断

4.2 引入重排序缓冲区改进 Tomasulo 算法的原理

重排序缓存的目的是让乱序执行的指令被顺序地提交。

乱序提交是 Tomasulo 最大的缺点。因为冯诺依曼结构向程序员承诺了处理器会按照程序的顺序来执行指令，因此程序员在调试程序的时候会希望当他在某行代码停下来的时候，代码前面的指令全部执行完，而代码后面的指令一条都没有执行，但是很显然，一个乱序执行、乱序提交的机器是没办法达到程序员的预期的。另外，因为控制指令、程序异常和外部中断也会截断指令流，所以通过顺序提交指令来实现精确中断（这里的中断是指中断指令流）至关重要。

ROB 的核心原理是记录下指令在程序中的顺序，一条指令在执行完毕之后不会立马提交（这里的提交指的是修改处理器状态，如修改逻辑寄存器堆），而是先在 Buffer 中等待，等到前面的所有指令都提交完毕，才可以提交结果到逻辑寄存器堆。

4.3 重排序缓存的缺点

- 资源消耗：ROB 是一个大型的硬件结构，需要大量的寄存器和逻辑电路。
- 复杂的控制逻辑：管理 ROB 的逻辑非常复杂，特别是在处理分支预测失败和异常时。
- 限制了缓冲区大小：ROB 的大小是有限的，限制了可以乱序执行的指令数量。

- 潜在的性能瓶颈：如果 ROB 满了，新的指令无法被发射，可能导致性能下降。