Speculative Tomasulo 算法——体系结构大作业

21307077 凌国明

1. 实验任务

1.1 **任务描述.** 使用 python 搭建以下的仿真器，仿真器中有以下组成部分：CDB 通用数据总线，Instruction Queue 指令队列，Reservation Station 保留站，Address Unit 地址计算单元，Memory Unit 内存单元，FP Adder 浮点加法运算单元，FP Multiplier 浮点乘法运算单元 Reorder Buffer 重排序缓冲区，FP Registers 浮点寄存器组

通过以上组件，实现乱序执行，顺序提交的 Speculative Tomasulo 算法

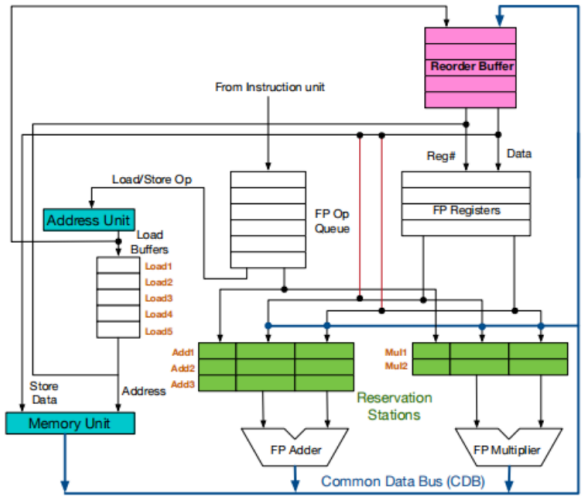

图 1. 仿真器架构图

1.2 **实验原理.** Tomasulo 算法是一种用于处理器中的动态调度技术，主要用于浮点运算单元。该算法的目的是通过硬件机制解决数据冲突和控制冲突，从而提高指令的并行执行效率。它引入了保留站的概念，用于存放即将执行的指令和操作数，同时使用了寄存器重命名技术来避免 WAR（写后读）和 WAW（写后写）冲突。通过这些机制，Tomasulo 算法可以实现更高的指令级并行性，并允许多个指令在同一时刻处于不同的执行阶段，有效提升了处理器的性能。

Speculative Tomasulo 算法在 Tomasulo 算法的基础上进一步发展，加入了对乱序执行和分支预测的支持。这种算法允许处理器在等待分支指令的最终结果时，预测分支的走向并继续执行后续指令。为了支持这种投机性执行，引入了重新排序缓冲区（Reorder Buffer, ROB），它负责记录指令的原始顺序，确保即使在乱序执行的情况下也能按照正确的顺序提交结果。同时，为了处理分支预测失败导致的异常情况，Speculative Tomasulo 算法还需能够快速撤销或回滚错误预测指令的结果。这种算法大大提高了处理器在面对复杂流程控制和高数据依赖性时的执行效率和吞吐量，是现代超标量处理器设计中的重要技术之一。

1.3 **基本假设.** 功能单元不是流水线式的。

功能单元之间没有转发；结果通过公共数据总线（CDB）进行通信。

执行阶段（EX）既执行有效地址计算，也执行加载和存储的内存访问。执行有效地址计算，加载和存储的内存访问各需要一个周期

指令发射（IS）和结果写回（WB）阶段各需要一个时钟周期。有三个加载缓冲区插槽和三个存储缓冲区插槽。

假设不等于零的分支（BNEZ）指令需要一个时钟周期

假设一个周期内可以 commit 多个已 write back 的指令

2. 实验过程

2.1 **整体架构.** 以下是作业初期写的每个模块的属性和功能，其中体现着面向对象的思想

CDB 通用数据总线传输数据，进行广播，只需将数据存好，让其他模块访问即可

Instruction Queue 指令队列：属性包括 n 条指令（读取 txt），保留站和 Reorder Buffer 的引用等等存储一系列的浮点指令，等待发射，每个周期向保留站顺序地发送一条指令（如果保留站有位置）这意味着指令发射需要 1 个周期将指令发送到保留站的同时，改写 Reorder Buffer：处理器会在 Reorder Buffer 中按顺序找到一个空行写入指令，置 Busy 位为 Yes，表示当前行含有指令信息；置 State 为 Issue，表示当前指令刚刚完成发射；并在 Dest 处标记目的寄存器编号。

Reservation Station 保留站：保留站每个条目有 name，busy，Op，Vj，Vk，Qj，Qk，Dest，A 这些字段每个条目存储指令类型 Op，目的地址 Dest，操作数（Vj，Vk，Qj，Qk），V 表示已准备好的数据，Q 表示有依赖的未准备好的数据，A 表示 Load/Store 指令的地址 FP Adder 有三个加载缓冲区插槽，FP Multiplier 有三个加载缓冲区插槽，Memory Unit 有三个存储缓冲区插槽若保留站中的某个指令的操作数都准备完毕了，且对应的功能单元不处于忙碌状态，则执行这条指令，执行完成后结果会在 CDB 中广播，保留站自己侦听，从而为保留站中的其他指令获取操作数

Address Unit 地址计算单元：有 busy，delay time，instruction，当前 clock，指令开始执行时的 clock 等等属性，这里 delay time 为 1 个周期，表示地址计算需要 1 个周期为了执行 Load/Store 指令，需要计算目的内存地址，计算需要 1 个周期，也就是说发送 Load 指令时，需要 1 个周期准备保留站中对应 Load 指令的 A 字段地址计算单元执行地址计算时，如果单元状态为空闲，则先将 instruction 和当前 clock 存下来，设置单元状态为忙碌。每次调用执行函数时，将 clock 加 1，如果当前 clock-指令开始执行时的 clock >= delay time，则执行完成，输出结果，置单元状态为空闲；如果当前 clock-指令开始执行时的 clock < delay time，则执行未完成，输出 None

Memory Unit 内存单元，FP Adder 浮点加法运算单元，FP Multiplier 浮点乘法运算单元：有 busy，delay time，instruction，当前 clock，指令开始执行时的 clock 等等属性，这里 delay time 为 1 个周期，表示访存需要 1 个周期，无论是写还是读都是 1 个周期不是流水线式的，同时只能执行一条指令，用 busy 标志单元是否空闲如果 busy 为 false 则保留站可以将发射的一条指令放入单元中执行，同时置 busy 为 true 内存单元执行访存时，如果单元状态为空闲，则先将 instruction 和当前 clock 存下来，设置单元状态为忙碌。每次调用执行函数时，将 clock 加 1，如果当前 clock-指令开始执行时的 clock >= delay time，则执行完成，输出结果，置单元状态为空闲；如果当前 clock-指令开始执行时的 clock < delay time，则执行未完成，输出 None

Reorder Buffer 重排序缓冲区:Reorder Buffer 有 head 和 tail，分别代表最老的指令和最新的指令对应的条目号假设 Reorder Buffer 有 9 个条目，刚好和保留站总条目数对应 Reorder Buffer 中的每个条目有 Entry，busy，Instruction，State，Dest，Value 字段 Busy 位指示某一行是否正保存有指令；State 位用来指示保存的指令当前的运行情况，ROB 就是通过 State 的信息来判断某条指令是否可以提交，当最老的指令还没到提交阶段时，ROB 中的所有指令都要等待，不能提交；Value 还有别的作用：在一条指令执行完毕但还不能提交时，后序指令有可能从 ROB 中读取 Value。Destination 指示指令的目的寄存器；Value 保存指令的结果，当指令可以提交，就直接提交 Value 到逻辑寄存器，这个过程不用通过 CDB，而是 Reorder Buffer 直接与 FP Registers 模块交流

FP Registers 浮点寄存器组：假设有 10 个寄存器（F0,F1,F2,F3,F4,F5,F6,F7,F8,F10）每个寄存器有 value，reorder，busy，分别表示当前值，对应的 reorder 的条目编号，是否在等待指令的结果

2.2 模块的实现.

2.2.1 通用数据总线. 这段代码定义了 CDB 类，它模拟了 Speculative Tomasulo 算法中的通用数据总线。该类负责数据的广播和检索，通过维护一个数据存储字典和一个忙碌状态标志，实现了数据的传递和共享，使不同组件能够获取所需的计算结果。

```python
class CDB:
    def __init__(self):
        # 数据存储区，用于存储广播的数据
        # 使用字典来存储，键是数据的目标寄存器，值是对应的数据
        self.data = {}
        self.busy = False

    def broadcast(self, target, value):
        """
        广播数据到公共数据总线。
        :param target: 目标寄存器或保留站的名称。
        :param value: 要广播的数据值。
        """
        self.data[target] = value
        self.busy = True
        # print("CDB: ", target, value)

    def get_data(self, target):
        """
        从公共数据总线获取数据。
        :param target: 请求数据的寄存器或保留站的名称。
        :return: 返回请求的数据值，如果没有则返回None。
        """
        return self.data.get(target)

    def clear(self):
        """
        清除公共数据总线上的所有数据。
        """
        self.data.clear()
        self.busy = False
```

Code Listing 1. 通用数据总线

2.2.2 寄存器组. 这段代码定义了一个名为 FPRegisters 的类，表示浮点寄存器组。该类包括 10 个浮点寄存器（F0 到 F10），每个寄存器具有三个属性：value（当前值）、reorder（对应的重排序缓冲区条目编号）、busy（是否在等待指令结果）。类中提供了方法来设置寄存器的值、设置寄存器为忙碌状态、获取寄存器的值以及检查寄存器是否忙碌。此外，还有一个 show 方法用于以表格形式打印出寄存器的状态，包括寄存器名称、重排序缓冲区条目编号和是否忙碌。

```python
class FPRegisters:
    def __init__(self):
        # 浮点寄存器组的属性
        # 假设有10个寄存器（F0,F1,F2,F3,F4,F5,F6,F7,F8,F10）
        # 每个寄存器包含value（当前值）、reorder（对应的重排序缓冲区条目编号）、busy（是否在等待指令结果）
        self.registers = {f"F{i}": {"value": 0, "reorder": None, "busy": False} for i in range(11)}

    def set_value(self, head, register, value):
        """
        设置寄存器的值。
        :param register: 寄存器名称。
        :param value: 设置的值。
        """
        if register in self.registers and head == self.registers[register]["reorder"]:
            self.registers[register]["value"] = value
            self.registers[register]["busy"] = False

    def set_busy(self, register, reorder):
        """
        设置寄存器为忙碌状态，并记录对应的重排序缓冲区条目。
        :param register: 寄存器名称。
        :param reorder: 对应的重排序缓冲区条目编号。
        """
        if register in self.registers:
            self.registers[register]["reorder"] = reorder
            self.registers[register]["busy"] = True

    def get_value(self, register):
        """
        获取寄存器的值。
        :param register: 寄存器名称。
        :return: 寄存器的值。
        """
        return self.registers[register]["value"] if register in self.registers else None

    def is_busy(self, register):
        """
        检查寄存器是否忙碌。
        :param register: 寄存器名称。
        :return: 寄存器是否忙碌。
        """
        return self.registers[register]["busy"] if register in self.registers else False

    def show(self):
```

Code Listing 2. 浮点寄存器组

2.2.3 执行单元. AddressUnit 类表示地址计算单元，MemoryUnit 类表示内存单元，FPAdder 类表示浮点加法运算单元，FPMultiplier 类表示浮点乘法运算单元。

这些类的共同点是它们都具有忙碌状态、周期数、当前执行的指令等属性，以及发送指令和执行操作的方法。每个类都会在执行完成后将结果广播到共享的数据总线 CDB 上。

这个执行单元的具体过程为：输入指令，记录输入的时钟 clock，每个 clock 到来时，仿真器（Simulator）会调用执行单元的 exec 函数，让执行单元度过一个周期，如果当前 clock-开始 clock>=延迟时间，则表示执行完成，将结果广播到 CDB 中。

以下代码仅展示浮点乘法运算单元，其他三个执行单元的结构和逻辑是类似的，都具有忙碌状态、周期数、记录指令等属性，以及发送指令和执行操

作的方法

```python
class FPMultiplier:
    def __init__(self, cdb: CDB):
        # 浮点乘法运算单元的基本属性
        self.busy = False                    # 标志单元是否忙碌
        self.delay_time = {'MULTD': 10, 'DIVD': 20}  # 乘法和除法的周期数
        self.instruction = None              # 当前正在执行的指令
        self.current_clock = 0               # 当前的时钟周期
        self.start_clock = 0                 # 指令开始执行的时钟周期
        self.cdb = cdb

    def issue_instruction(self, instruction, current_clock):
        """
        发送指令到浮点乘法运算单元执行。
        instruction: 输入指令
        """
        if not self.busy:
            self.instruction = instruction
            self.start_clock = current_clock
            self.current_clock = current_clock
            self.busy = True
            return True
        return False

    def execute(self):
        """
        执行浮点乘法运算。
        """
        if self.busy:
            self.current_clock += 1
            op = self.instruction["Instruction"].split()[0].upper()  # 获取指令中的操作类型
            delay = self.delay_time.get(op, 10)  # 默认乘法延迟
            # 判断运算是否完成，若完成且cdb空闲，则在cdb上广播
            if self.current_clock - self.start_clock >= delay and self.cdb.busy:
                self.busy = False
                self.cdb.broadcast(self.instruction["Entry"], 10)
                # 返回指令执行结果
                return 10  # 这里简化处理，直接返回10作为结果
        return None
```

Code Listing 3. 执行单元

2.2.4 **保留站.** 保留站的结构是一个包含三个类别（Add、Mult、Load）的保留站集合，每个类别包含多个条目，每个条目具有属性来表示该条目是否忙碌、操作类型、操作数的值或来源（Vj、Vk、Qj、Qk）、目标寄存器、以及操作数的地址。这些保留站条目用于暂时存储指令的信息，并等待操作数的就绪。保留站与其他计算单元（如浮点寄存器、浮点运算单元、内存单元等）以及重排序缓冲区和公共数据总线进行交互，实现指令的执行和结果的广播。

保留站的每个条目存储指令类型 Op，目的地址 Dest，操作数（Vj，Vk，Qj，Qk），V 表示已准备好的数据，Q 表示有依赖的未准备好的数据，A 表示 Load/Store 指令的地址

```python
class ReservationStation:
    def __init__(self, cdb, fp_registers, fp_adder, fp_multiplier, memory_unit, address_unit, reorder_buffer):
        # 初始化保留站的条目
        self.entries = {
            "Add": [{"name": f"Add{i}", "busy": False, "Op": None, "Vj": None, "Vk": None, "Qj": None, "Qk": None, "Dest": None, "A": None} for i in range(3)],
            "Mult": [{"name": f"Mult{i}", "busy": False, "Op": None, "Vj": None, "Vk": None, "Qj": None, "Qk": None, "Dest": None, "A": None} for i in range(3)],
            "Load": [{"name": f"Load{i}", "busy": False, "Op": None, "Vj": None, "Vk": None, "Qj": None, "Qk": None, "Dest": None, "A": None} for i in range(3)]
        }
        self.cdb = cdb
        self.fp_registers = fp_registers
        self.fp_adder = fp_adder
        self.fp_multiplier = fp_multiplier
        self.memory_unit = memory_unit
        self.address_unit = address_unit
        self.reorder_buffer = reorder_buffer
        self.clock = 0
```

Code Listing 4. 保留站的结构

这个函数用于向保留站发射指令。在执行过程中，它会根据指令的操作类型（如加法、乘法、加载等）将指令分配给相应类型的保留站条目。每个保留站条目具有各自的属性，包括操作类型、操作码、源操作数、目的寄存器等。该函数首先解析传入的指令，然后根据指令的操作类型将指令分配给合适的保留站条目。如果条目未被占用（busy 为 False），则将其标记为占用，并设置相应的属性，包括操作码、源操作数、目的寄存器等。这个过程确保指令按照操作类型被正确地分发到相应的保留站中，以供后续执行。

```python
def issue_instruction(self, instruction):
    """
    向保留站发射指令。
    :param instruction: 要发射的指令。
    """
    op, *operands = instruction.split()
    entry_type = "Add" if op in ["ADDD", "SUBD"] else "Mult" if op in ["MULTD", "DIVD"] else "Load"

    for entry in self.entries[entry_type]:
        if not entry["busy"]:
            entry["busy"] = True
            entry["Op"] = op
            # 检查源操作数对应的寄存器是否busy
            if op == "LD":
                entry["Vj"] = operands[2]  # 地址
                entry["Vk"] = None
                entry["Qj"] = None
                entry["Qk"] = None
                entry["A"] = operands[1]
            elif op == "SD":
                # 检查操作数对应的寄存器是否busy
                if self.fp_registers.is_busy(operands[0]):
                    entry["Qj"] = operands[0]
                else:
                    entry["Vj"] = operands[0]
                entry["Vk"] = operands[2]
                entry["Qk"] = None
                entry["A"] = operands[1]
            else:
```

```
                # 检查操作数对应的寄存器是否busy
                if self.fp_registers.is_busy(
                    operands[1]):
                    entry["Qj"] = operands[1]
                else:
                    entry["Vj"] = operands[1]
                if self.fp_registers.is_busy(
                    operands[2]):
                    entry["Qk"] = operands[2]
                else:
                    entry["Vk"] = operands[2]
                entry["Dest"] = str(self.
                    reorder_buffer.tail)  # 目的地址
                break
```

Code Listing 5. 向保留站发射指令

这个函数用于执行保留站中的指令。遍历各个保留站的条目，如果条目被标记为 busy，并且与该条目相关的重排序缓冲区条目处于”WriteBack”或”Commit” 状态，那么该条目将被释放（busy 标记为 False）。接着，它检查条目的操作类型，如果所有操作数都已准备就绪（Qj 和 Qk 均为 None），则可以执行指令。根据指令的操作类型，选择相应的执行单元（如浮点加法器、浮点乘法器、内存单元等），并将指令发送给执行单元执行。如果执行单元不忙，并且与该指令相关的重排序缓冲区条目处于”Executing” 状态，则执行单元会接受指令并执行。最后，函数调用各执行单元的 execute 方法来执行指令，使得执行单元度过一个周期。

这部分的关键在于“判断是否将指令输入到执行单元”的逻辑。具体逻辑为：当执行单元不忙且当前指令的所有操作数都已就绪时，向单元输入指令，将 ROB 对应条目的状态改为 Executing，这是通过其他的连接线来完成的，所以不用向 CDB 发送数据

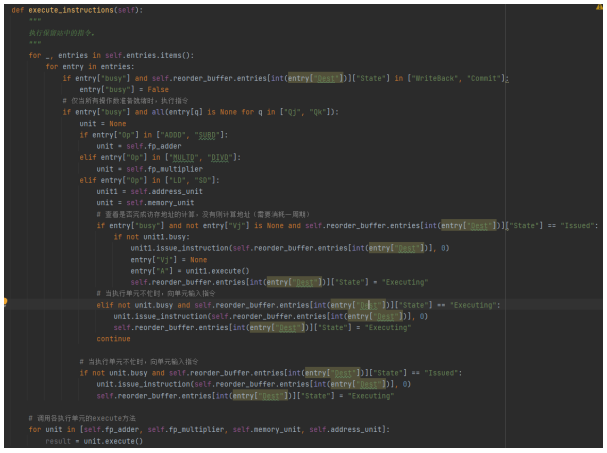

图 2. 执行保留站中的指令

```
def execute_instructions(self):
    """
    执行保留站中的指令。
    """
    for _, entries in self.entries.items():
        for entry in entries:
            if entry["busy"] and self.
                reorder_buffer.entries[int(entry[
                "Dest"])]["State"] in ["WriteBack
                ", "Commit"]:
                entry["busy"] = False
            # 仅当所有操作数准备就绪时，执行指令
            if entry["busy"] and all(entry[q] is
                None for q in ["Qj", "Qk"]):
                unit = None
                if entry["Op"] in ["ADDD", "SUBD"
                    ]:
                    unit = self.fp_adder
                elif entry["Op"] in ["MULTD", "
                    DIVD"]:
                    unit = self.fp_multiplier
                elif entry["Op"] in ["LD", "SD"]:
                    unit1 = self.address_unit
                    unit = self.memory_unit
                    # 查看是否完成访存地址的计
                        算，没有则计算地址（需要
                        消耗一周期）
                    if entry["busy"] and not
                        entry["Vj"] is None and
                        self.reorder_buffer.
                        entries[int(entry["Dest"
                        ])]["State"] == "Issued":
                        if not unit1.busy:
                            unit1.
                                issue_instruction
                                (self.
                                reorder_buffer.
                                entries[int(entry
                                ["Dest"])], 0)
                            entry["Vj"] = None
                            entry["A"] = unit1.
                                execute()
                            self.reorder_buffer.
                                entries[int(entry
                                ["Dest"])]["State
                                "] = "Executing"
                    # 当执行单元不忙时，向单元输
                        入指令
                    elif not unit.busy and self.
                        reorder_buffer.entries[
                        int(entry["Dest"])]["
                        State"] == "Executing":
                        unit.issue_instruction(
                            self.reorder_buffer.
                            entries[int(entry["
                            Dest"])], 0)
                        self.reorder_buffer.
                            entries[int(entry["
                            Dest"])]["State"] = "
                            Executing"
                    continue

                # 当执行单元不忙时，向单元输入指
                    令
                if not unit.busy and self.
                    reorder_buffer.entries[int(
                    entry["Dest"])]["State"] == "
                    Issued":
                    unit.issue_instruction(self.
                        reorder_buffer.entries[
                        int(entry["Dest"])], 0)
                    self.reorder_buffer.entries[
                        int(entry["Dest"])]["
                        State"] = "Executing"

    # 调用各执行单元的execute方法
    for unit in [self.fp_adder, self.
        fp_multiplier, self.memory_unit, self.
        address_unit]:
        result = unit.execute()
```

Code Listing 6. 执行保留站中的指令

这里因为单行代码有点长，导致展示效果不是很好，如果想仔细阅读这部分的代码，可以看上面的图片或者源代码

这个函数用于更新可用的数据源，从重排序缓冲区（ROB）中获取数据。在执行过程中，它会遍历各个保留站的条目，检查是否有条目处于占用状态（busy 为 True）。如果某条目的 Qj 字段不为 None，说明它依赖于另一指令的执行结果，此时会检查该依赖指令是否已经执行完成并将其结果保存在 ROB 中。如果依赖指令的结果已经就绪，那么将 Qj 字段转换为 Vj，表示该操作数已经可用。同样的逻辑也适用于 Qk 字段，用于处理第二个操作数。这个函数的目的是确保所有操作数都已准备就绪，以便执行指令。

图 3. 从 ROB 中获取新数据

```python
def update_from_rob(self):
    # 更新可用数据源(从rob中)
    for _, entries in self.entries.items():
        for entry in entries:
            if entry["busy"]:
                if not entry["Qj"] is None:
                    # 已经执行完成保存在buffer
                        中，可将Q转为V
                    if self.reorder_buffer.
                        entries[self.fp_registers
                        .registers[entry["Qj"]]["
                        reorder"]]["Value"] is
                        not None:
                        # print(self.
                            reorder_buffer.
                            entries[self.
                            fp_registers.
                            registers[entry["Qj
                            "]]["reorder"]]["
                            Value"])
                        entry["Vj"] = entry["Qj"]
                        entry["Qj"] = None

                if not entry["Qk"] is None:
                    if self.reorder_buffer.
                        entries[self.fp_registers
                        .registers[entry["Qk"]]["
                        reorder"]][
                            "Value"] is not None:
                        entry["Vk"] = entry["Qk"]
                        entry["Qk"] = None
```

Code Listing 7. 从 ROB 中获取新数据

小结：保留站结构在执行指令时维护了不同类型指令的条目，包括加法、乘法和内存访问指令。保留站的主要功能包括发射、执行和更新操作，它负责分配可用的执行单元来执行指令，并通过重排序缓冲区（ROB）来跟踪指令的执行状态，以确保操作数的准备就绪，从而实现高效的指令调度和执行。

2.2.5 **重排序缓冲区.** ReorderBuffer 是一个用于跟踪指令执行状态和结果的缓冲区，它包含了多个条目，每个条目代表一条指令的执行情况。它的主要功能是记录指令的状态（如”Issued”、”Executing”、”WriteBack”、”Commit” 等）、目的寄存器、执行结果以及是否繁忙等信息。通过不断更新头部和尾部的位置，它能够追踪最老和最新的指令，以支持乱序执行，并且能够与公共数据总线（CDB）和浮点寄存器配合，实现指令的写回和提交。

```python
class ReorderBuffer:
    def __init__(self, num_entries, fp_registers,
        cdb):
        self.entries = [{"Entry": i, "busy":
            False, "Instruction": None, "State":
            None, "Dest": None, "Value": None}
            for i in range(num_entries)]
        self.head = 0  # 最老的指令对应的条目号
        self.tail = 0  # 最新的指令对应的条目号
        self.cdb = cdb
        self.fp_registers = fp_registers
        self.to_write_pos = None
        self.to_write_data = None
```

Code Listing 8. ROB 结构

add_instruction 函数用于将一条指令添加到重排序缓冲区（Reorder Buffer）中。它会检查缓冲区中是否有可用的条目，如果有，就将指令和相关信息（如目的寄存器、状态等）添加到缓冲区中，并更新相应的寄存器的忙碌状态。这个函数支持乱序执行，确保指令按照发射的顺序进入缓冲区，并循环利用缓冲区的条目。

```python
def add_instruction(self, instruction, dest):
    """
    添加一条指令到重排序缓冲区。
    :param instruction: 要添加的指令。
    :param dest: 指令的目的寄存器。
    """
    if not self.entries[self.tail]["busy"]:
        self.entries[self.tail] = {
            "Entry": self.tail,
            "busy": True,
            "Instruction": instruction,
            "State": "Issued",
            "Dest": dest,
            "Value": None
        }
        # store指令不设置寄存器的busy
        if self.entries[self.tail]["Instruction"
            ].split()[0] != "SD":
            self.fp_registers.set_busy(
                instruction.split()[1], self.tail
                )
        self.tail = (self.tail + 1) % len(self.
            entries)
```

Code Listing 9. 向 ROB 中增加条目

check_cdb 函数用于检查 Common Data Bus (CDB) 上是否有数据可供缓冲区处理。首先，它会检查是否有等待写回的数据，如果有，就将该数据写回到缓冲区中，并更新相应的状态。然后，它检

查 CDB 是否忙碌，如果忙碌，它会遍历 CDB 上的数据，并找到等待执行的指令。如果找到需要写回的数据，它会记录下来，并将相应的指令状态设置为”Executed”。最后，它会清除 CDB 上的所有数据，以准备接收新的数据。这个函数用于维护指令的执行状态和数据的写回状态。

特别注意，每个周期在侦听到 CDB 的数据后，要延迟一周期再提交，这样才能符合“写回需要一个周期”的假设，因此每次侦听到数据，先用一个变量记录下来，下个周期写到相应的条目处。

```
def check_cdb(self):
    if self.to_write_pos is not None:
        self.entries[self.to_write_pos]["State"]
            = "WriteBack"
        self.entries[self.to_write_pos]["Value"]
            = self.to_write_data
        self.to_write_pos = None
        self.to_write_data = None
    if self.cdb.busy:
        for i in self.cdb.data:
            if self.entries[i]["State"] == "
                Executing":
                if self.entries[i]["Instruction"
                    ].split()[0] == "SD":
                    self.entries[i]["State"] = "
                        Commit"
                    break
                self.to_write_pos = i
                self.to_write_data = self.cdb.
                    data[i]
                self.entries[self.to_write_pos]["
                    State"] = "Executed"
                break
    self.cdb.clear()
```

Code Listing 10. ROB 侦听 CDB

commit_instructions 函数用于提交重排序缓冲区中已经执行完成的指令。它通过循环遍历缓冲区头部的指令，检查是否可以提交（即状态为”WriteBack” 或”Commit” 且不忙碌），如果可以提交，则将指令的结果写入目标寄存器，将指令状态设置为”Commit”，并标记为不忙碌。然后，它将头部指针移动到下一个条目，以继续检查下一条指令是否可以提交。这个函数用于维护指令的提交状态和结果写回到寄存器中。这里假设了一个周期可以 commit 多个已经 writeback 的指令

```
def commit_instructions(self):
    """
    提交重排序缓冲区中的指令。
    """
    while self.entries[self.head]["busy"] and
        self.entries[self.head]["State"] in ["
        WriteBack", "Commit"]:
        entry = self.entries[self.head]
        self.fp_registers.set_value(self.head,
            entry["Dest"], entry["Value"])
        entry["State"] = "Commit"
        entry["busy"] = False
        self.head = (self.head + 1) % len(self.
            entries)
```

Code Listing 11. 提交指令

2.2.6 **指令队列.** InstructionQueue 类用于管理指令队列，它的功能包括从文件读取指令、向保留站和重排序缓冲区发射指令。

在初始化时，它读取指令文件并存储在 instructions 列表中，同时接收保留站和重排序缓冲区作为参数。read_instructions 方法用于从文件中读取指令并返回指令列表。issue_instruction 方法用于发射指令，它首先检查当前处理的指令索引是否小于指令列表的长度且重排序缓冲区是否未满，如果满足条件，则获取下一条指令并将其发送到保留站，然后更新重排序缓冲区，最后将当前指令索引递增。这个类的主要作用是协调指令的发射和管理重排序缓冲区。

```
class InstructionQueue:
def __init__(self, file_path, reservation_station
    , reorder_buffer):
    self.instructions = self.read_instructions(
        file_path)
    self.reservation_station =
        reservation_station
    self.reorder_buffer = reorder_buffer
    self.current_index = 0  # 当前处理的指令索引

def read_instructions(self, file_path):
    """
    从文件读取指令。
    :param file_path: 指令文件的路径。
    :return: 指令列表。
    """
    instructions = []
    with open(file_path, 'r') as file:
        for line in file:
            instructions.append(line.strip())
    return instructions

def issue_instruction(self):
    """
    向保留站和重排序缓冲区发射指令。
    """
    if self.current_index < len(self.instructions
        ) and not self.reorder_buffer.is_full():
        instruction = self.instructions[self.
            current_index]
        parts = instruction.split()
        op = parts[0]
        dest = parts[1]

        # 发送指令到保留站
        self.reservation_station.
            issue_instruction(instruction)

        # 更新重排序缓冲区
        self.reorder_buffer.add_instruction(
            instruction, dest)
        self.current_index += 1
def has_instructions(self):
    """
    检查是否还有未处理的指令。
    :return: 布尔值，表示是否还有指令待处理。
    """
    return self.current_index < len(self.
        instructions)
```

Code Listing 12. 指令队列

指令队列每个周期只发送一条指令，这是因为“发送指令需要一个时钟周期”的假设。

指令队列的实现较为简单，核心的功能都在保留站和重排序缓冲区中。

2.2.6 **仿真器.** 这段代码定义了一个仿真器（Simulator）类，用于模拟处理器的运行过程。模拟器包括多个组件，如浮点寄存器（FPRegisters）、公共数据总线（CDB）、浮点加法器（FPAdder）、浮点乘法器（FPMultiplier）、内存单元（MemoryUnit）、地址计算单元（AddressUnit）、重排序缓冲区（ReorderBuffer）、保留站（ReservationStation）和指令队列（InstructionQueue）。模拟器的主要功能是在时钟周期内模拟处理器的操作，包括指令的发射、执行、提交等，不断循环执行这些操作，直到指令队列为空且重排序缓冲区为空，模拟器结束运行。

```
class Simulator:
def __init__(self, instruction_file_path):
    # 初始化组件
    self.fp_registers = FPRegisters()
    self.cdb = CDB()
    self.fp_adder = FPAdder(self.cdb)
    self.fp_multiplier = FPMultiplier(self.cdb)
    self.memory_unit = MemoryUnit(self.cdb)
    self.address_unit = AddressUnit(self.cdb)
    self.reorder_buffer = ReorderBuffer(9, self.
        fp_registers, self.cdb)
    self.reservation_station = ReservationStation
        (self.cdb, self.fp_registers, self.
        fp_adder, self.fp_multiplier, self.
        memory_unit, self.address_unit, self.
        reorder_buffer)
    self.instruction_queue = InstructionQueue(
        instruction_file_path, self.
        reservation_station, self.reorder_buffer)
    self.clock = 0

def run(self):
    # 运行模拟器
    while self.instruction_queue.has_instructions
        () or not self.reorder_buffer.is_empty():
        print(f"--------------------------Clock
            Cycle: {self.clock
            +1}--------------------------\n")

        self.cdb.clear()
        self.reorder_buffer.commit_instructions()
        self.reservation_station.
            execute_instructions()
        self.reorder_buffer.check_cdb()
        self.reservation_station.update_from_rob
            ()
        self.instruction_queue.issue_instruction
            ()

        self.clock += 1

        self.reorder_buffer.show()
        print()
        self.reservation_station.show()
        print()
        self.fp_registers.show()
        print("\n\n")

        # 防止无限循环
        if self.clock > 100:
            break
```

Code Listing 13. 仿真器

可见，各个部件运行的顺序与一条指令的执行顺序是大致相反的。这是因为如果先发射指令，再将指令发送到执行单元中，再从 CDB 中侦听数据，会导致这一整串操作在一个周期内完成（至少在我的代码里是这样）。为了实现各个部件的同步正确运行

3. 结果分析

3.1 Load **指令和“写后读”相关.** 第一个周期，发射第一条指令 LD1，LD1 的目的地址还未算出

```
--------------------------Clock Cycle: 1--------------------------

 Entry | busy | Instruction   | State  | Dest | Value
-------------------------------------------------------------
   0   | Yes  | LD F6 34+ R2  | Issued |  F6  |
   1   |  No  |               |        |      |
   2   |  No  |               |        |      |
   3   |  No  |               |        |      |
   4   |  No  |               |        |      |
   5   |  No  |               |        |      |
   6   |  No  |               |        |      |
   7   |  No  |               |        |      |
   8   |  No  |               |        |      |

  name | busy | Op | Vj | Vk | Qj | Qk | Dest |  A
-------------------------------------------------------------
 Add0  |  No  |    |    |    |    |    |      |
 Add1  |  No  |    |    |    |    |    |      |
 Add2  |  No  |    |    |    |    |    |      |
 Mult0 |  No  |    |    |    |    |    |      |
 Mult1 |  No  |    |    |    |    |    |      |
 Mult2 |  No  |    |    |    |    |    |      |
 Load0 | Yes  | LD | R2 |    |    |    |  0   | 34+
 Load1 |  No  |    |    |    |    |    |      |
 Load2 |  No  |    |    |    |    |    |      |

Field   | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | F10
Reorder |    |    |    |    |    |    |  0 |    |    |    |
Busy    | No | No | No | No | No | No | Yes | No | No | No | No
```

图 4. clock 1

第二个周期，发射第二条指令 LD2，LD1 的目的地址已被算出（A 字段），LD2 的目的地址未算出

```
--------------------------Clock Cycle: 2--------------------------

 Entry | busy | Instruction   |   State   | Dest | Value
-------------------------------------------------------------
   0   | Yes  | LD F6 34+ R2  | Executing |  F6  |
   1   | Yes  | LD F2 45+ R3  |   Issued  |  F2  |
   2   |  No  |               |           |      |
   3   |  No  |               |           |      |
   4   |  No  |               |           |      |
   5   |  No  |               |           |      |
   6   |  No  |               |           |      |
   7   |  No  |               |           |      |
   8   |  No  |               |           |      |

  name | busy | Op | Vj | Vk | Qj | Qk | Dest |  A
-------------------------------------------------------------
 Add0  |  No  |    |    |    |    |    |      |
 Add1  |  No  |    |    |    |    |    |      |
 Add2  |  No  |    |    |    |    |    |      |
 Mult0 |  No  |    |    |    |    |    |      |
 Mult1 |  No  |    |    |    |    |    |      |
 Mult2 |  No  |    |    |    |    |    |      |
 Load0 | Yes  | LD |    |    |    |    |  0   |  44
 Load1 | Yes  | LD | R3 |    |    |    |  1   | 45+
 Load2 |  No  |    |    |    |    |    |      |

Field   | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | F10
Reorder |    |    |  1 |    |    |    |  0 |    |    |    |
Busy    | No | No | Yes | No | No | No | Yes | No | No | No | No
```

图 5. clock 2

第三个周期，发射了第三条指令 MULTD，第一条指令 LD1 已完成，将结果广播到了 CDB 中

```
-------------------------Clock Cycle: 3-------------------------

  Entry  |  busy  |   Instruction    |    State    |  Dest  |  Value
-----------------------------------------------------------------------
    0    |  Yes   |  LD F6 34+ R2    |  Executed   |   F6   |
    1    |  Yes   |  LD F2 45+ R3    |  Executing  |   F2   |
    2    |  Yes   |  MULTD F0 F2 F4  |    Issued   |   F0   |
    3    |   No   |                  |             |        |
    4    |   No   |                  |             |        |
    5    |   No   |                  |             |        |
    6    |   No   |                  |             |        |
    7    |   No   |                  |             |        |
    8    |   No   |                  |             |        |

   name  |  busy  |    Op    |  Vj  |  Vk  |  Qj  |  Qk  |  Dest  |  A
-------------------------------------------------------------------------
   Add0  |   No   |          |      |      |      |      |        |
   Add1  |   No   |          |      |      |      |      |        |
   Add2  |   No   |          |      |      |      |      |        |
  Mult0  |  Yes   |  MULTD   |      |  F4  |  F2  |      |   2    |
  Mult1  |   No   |          |      |      |      |      |        |
  Mult2  |   No   |          |      |      |      |      |        |
  Load0  |  Yes   |    LD    |      |      |      |      |   0    |  44
  Load1  |  Yes   |    LD    |      |      |      |      |   1    |  55
  Load2  |   No   |          |      |      |      |      |        |

Field    |   F0 |  F1 |  F2 |  F3 |  F4 |  F5 |  F6 |  F7 |  F8 |  F9 | F10
Reorder  |   2  |     |  1  |     |     |     |  0  |     |     |     |
Busy     |  Yes |  No | Yes |  No |  No |  No | Yes |  No |  No |  No |  No
```

图 6. clock 3

第四周期，发射 SUBD，第一条指令 LD1 的结果已经写回，第二条指令 LD2 计算完成，将结果发到 CDB 中。因为第三条指令的操作数 F2 依赖于第二条指令 LD2 的结果，所以不能开始执行

```
-------------------------Clock Cycle: 4-------------------------

  Entry  |  busy  |   Instruction    |    State    |  Dest  |  Value
-----------------------------------------------------------------------
    0    |  Yes   |  LD F6 34+ R2    |  WriteBack  |   F6   |    10
    1    |  Yes   |  LD F2 45+ R3    |  Executed   |   F2   |
    2    |  Yes   |  MULTD F0 F2 F4  |    Issued   |   F0   |
    3    |  Yes   |  SUBD F8 F6 F2   |    Issued   |   F8   |
    4    |   No   |                  |             |        |
    5    |   No   |                  |             |        |
    6    |   No   |                  |             |        |
    7    |   No   |                  |             |        |
    8    |   No   |                  |             |        |

   name  |  busy  |    Op    |  Vj  |  Vk  |  Qj  |  Qk  |  Dest  |  A
-------------------------------------------------------------------------
   Add0  |  Yes   |   SUBD   |      |      |  F6  |  F2  |   3    |
   Add1  |   No   |          |      |      |      |      |        |
   Add2  |   No   |          |      |      |      |      |        |
  Mult0  |  Yes   |  MULTD   |      |  F4  |  F2  |      |   2    |
  Mult1  |   No   |          |      |      |      |      |        |
  Mult2  |   No   |          |      |      |      |      |        |
  Load0  |  Yes   |    LD    |      |      |      |      |   0    |  44
  Load1  |  Yes   |    LD    |      |      |      |      |   1    |  55
  Load2  |   No   |          |      |      |      |      |        |

Field    |   F0 |  F1 |  F2 |  F3 |  F4 |  F5 |  F6 |  F7 |  F8 |  F9 | F10
Reorder  |   2  |     |  1  |     |     |     |  0  |     |  3  |     |
Busy     |  Yes |  No | Yes |  No |  No |  No | Yes |  No | Yes |  No |  No
```

图 7. clock 4

第五个周期，发射 DIVD，提交第一条指令 LD1（因为是按序的），写回 LD2 的结果（写回要一周期）

```
-------------------------Clock Cycle: 5-------------------------

  Entry  |  busy  |   Instruction    |    State    |  Dest  |  Value
-----------------------------------------------------------------------
    0    |   No   |  LD F6 34+ R2    |   Commit    |   F6   |    10
    1    |  Yes   |  LD F2 45+ R3    |  WriteBack  |   F2   |    10
    2    |  Yes   |  MULTD F0 F2 F4  |    Issued   |   F0   |
    3    |  Yes   |  SUBD F8 F6 F2   |    Issued   |   F8   |
    4    |  Yes   |  DIVD F10 F0 F6  |    Issued   |  F10   |
    5    |   No   |                  |             |        |
    6    |   No   |                  |             |        |
    7    |   No   |                  |             |        |
    8    |   No   |                  |             |        |

   name  |  busy  |    Op    |  Vj  |  Vk  |  Qj  |  Qk  |  Dest  |  A
-------------------------------------------------------------------------
   Add0  |  Yes   |   SUBD   |  F6  |  F2  |      |      |   3    |
   Add1  |   No   |          |      |      |      |      |        |
   Add2  |   No   |          |      |      |      |      |        |
  Mult0  |  Yes   |  MULTD   |  F2  |  F4  |      |      |   2    |
  Mult1  |  Yes   |   DIVD   |      |  F6  |  F0  |      |   4    |
  Mult2  |   No   |          |      |      |      |      |        |
  Load0  |   No   |    LD    |      |      |      |      |   0    |  44
  Load1  |  Yes   |    LD    |      |      |      |      |   1    |  55
  Load2  |   No   |          |      |      |      |      |        |

Field    |   F0 |  F1 |  F2 |  F3 |  F4 |  F5 |  F6 |  F7 |  F8 |  F9 | F10
Reorder  |   2  |     |  1  |     |     |     |  0  |     |  3  |     |  4
Busy     |  Yes |  No | Yes |  No |  No |  No |  No |  No | Yes |  No | Yes
```

图 8. clock 5

第六个周期，发射 ADDD，提交第二条指令 LD2 的结果于 F2 中。因为 F2 的值已被计算，对 F2 有依赖的 MULTD 和 SUBD 也可以同时开始执行。

```
-------------------------Clock Cycle: 6-------------------------

  Entry  |  busy  |   Instruction    |    State    |  Dest  |  Value
-----------------------------------------------------------------------
    0    |   No   |  LD F6 34+ R2    |   Commit    |   F6   |    10
    1    |   No   |  LD F2 45+ R3    |   Commit    |   F2   |    10
    2    |  Yes   |  MULTD F0 F2 F4  |  Executing  |   F0   |
    3    |  Yes   |  SUBD F8 F6 F2   |  Executing  |   F8   |
    4    |  Yes   |  DIVD F10 F0 F6  |    Issued   |  F10   |
    5    |  Yes   |  ADDD F6 F8 F2   |    Issued   |   F6   |
    6    |   No   |                  |             |        |
    7    |   No   |                  |             |        |
    8    |   No   |                  |             |        |

   name  |  busy  |    Op    |  Vj  |  Vk  |  Qj  |  Qk  |  Dest  |  A
-------------------------------------------------------------------------
   Add0  |  Yes   |   SUBD   |  F6  |  F2  |      |      |   3    |
   Add1  |  Yes   |   ADDD   |      |  F2  |  F8  |      |   5    |
   Add2  |   No   |          |      |      |      |      |        |
  Mult0  |  Yes   |  MULTD   |  F2  |  F4  |      |      |   2    |
  Mult1  |  Yes   |   DIVD   |      |  F6  |  F0  |      |   4    |
  Mult2  |   No   |          |      |      |      |      |        |
  Load0  |   No   |    LD    |      |      |      |      |   0    |  44
  Load1  |   No   |    LD    |      |      |      |      |   1    |  55
  Load2  |   No   |          |      |      |      |      |        |

Field    |   F0 |  F1 |  F2 |  F3 |  F4 |  F5 |  F6 |  F7 |  F8 |  F9 | F10
Reorder  |   2  |     |  1  |     |     |     |  5  |     |  3  |     |  4
Busy     |  Yes |  No |  No |  No |  No |  No | Yes |  No | Yes |  No | Yes
```

图 9. clock 6

3.2 **乱序执行、顺序提交.** 第十四个周期，SUBD 的结果已经写回，但因为前一个指令 MULTD 还没有提交，所以 SUBD 不能提交

```
-------------------------Clock Cycle: 14-------------------------

  Entry  |  busy  |   Instruction    |    State    |  Dest  |  Value
-------------------------------------------------------------------
    0    |   No   |   LD F6 34+ R2   |    Commit   |   F6   |    10
    1    |   No   |   LD F2 45+ R3   |    Commit   |   F2   |    10
    2    |  Yes   |  MULTD F0 F2 F4  |  Executing  |   F0   |
    3    |  Yes   |  SUBD F8 F6 F2   |  WriteBack  |   F8   |    10
    4    |  Yes   |  DIVD F10 F0 F6  |    Issued   |  F10   |
    5    |  Yes   |  ADDD F6 F8 F2   |  WriteBack  |   F6   |    10
    6    |   No   |                  |             |        |
    7    |   No   |                  |             |        |
    8    |   No   |                  |             |        |

   name  |  busy  |    Op    |  Vj  |  Vk  |  Qj  |  Qk  |  Dest  |  A
-----------------------------------------------------------------------
   Add0  |   No   |   SUBD   |  F6  |  F2  |      |      |   3    |
   Add1  |   No   |   ADDD   |  F8  |  F2  |      |      |   5    |
   Add2  |   No   |          |      |      |      |      |        |
  Mult0  |  Yes   |  MULTD   |  F2  |  F4  |      |      |   2    |
  Mult1  |  Yes   |   DIVD   |      |  F6  |  F0  |      |   4    |
  Mult2  |   No   |          |      |      |      |      |        |
  Load0  |   No   |    LD    |      |      |      |      |   0    |  44
  Load1  |   No   |    LD    |      |      |      |      |   1    |  55
  Load2  |   No   |          |      |      |      |      |        |

Field   |  F0 |  F1 |  F2 |  F3 |  F4 |  F5 |  F6 |  F7 |  F8 |  F9 | F10
Reorder |  2  |     |  1  |     |     |     |  5  |     |  3  |     |  4
Busy    | Yes |  No |  No |  No |  No |  No | Yes |  No | Yes |  No | Yes
```

图 10. clock 14

第十五个周期，MULTD 执行完成，将执行结果传到 CDB 中

```
-------------------------Clock Cycle: 15-------------------------

  Entry  |  busy  |   Instruction    |    State    |  Dest  |  Value
-------------------------------------------------------------------
    0    |   No   |   LD F6 34+ R2   |    Commit   |   F6   |    10
    1    |   No   |   LD F2 45+ R3   |    Commit   |   F2   |    10
    2    |  Yes   |  MULTD F0 F2 F4  |  Executed   |   F0   |
    3    |  Yes   |  SUBD F8 F6 F2   |  WriteBack  |   F8   |    10
    4    |  Yes   |  DIVD F10 F0 F6  |    Issued   |  F10   |
    5    |  Yes   |  ADDD F6 F8 F2   |  WriteBack  |   F6   |    10
    6    |   No   |                  |             |        |
    7    |   No   |                  |             |        |
    8    |   No   |                  |             |        |

   name  |  busy  |    Op    |  Vj  |  Vk  |  Qj  |  Qk  |  Dest  |  A
-----------------------------------------------------------------------
   Add0  |   No   |   SUBD   |  F6  |  F2  |      |      |   3    |
   Add1  |   No   |   ADDD   |  F8  |  F2  |      |      |   5    |
   Add2  |   No   |          |      |      |      |      |        |
  Mult0  |  Yes   |  MULTD   |  F2  |  F4  |      |      |   2    |
  Mult1  |  Yes   |   DIVD   |      |  F6  |  F0  |      |   4    |
  Mult2  |   No   |          |      |      |      |      |        |
  Load0  |   No   |    LD    |      |      |      |      |   0    |  44
  Load1  |   No   |    LD    |      |      |      |      |   1    |  55
  Load2  |   No   |          |      |      |      |      |        |

Field   |  F0 |  F1 |  F2 |  F3 |  F4 |  F5 |  F6 |  F7 |  F8 |  F9 | F10
Reorder |  2  |     |  1  |     |     |     |  5  |     |  3  |     |  4
Busy    | Yes |  No |  No |  No |  No |  No | Yes |  No | Yes |  No | Yes
```

图 11. clock 15

第十六个周期,MULTD 的结果写回了,此时 MULTD 和 SUBD 都处于 WriteBack 状态

```
-------------------------Clock Cycle: 16-------------------------

  Entry  |  busy  |   Instruction    |    State    |  Dest  |  Value
-------------------------------------------------------------------
    0    |   No   |   LD F6 34+ R2   |    Commit   |   F6   |    10
    1    |   No   |   LD F2 45+ R3   |    Commit   |   F2   |    10
    2    |  Yes   |  MULTD F0 F2 F4  |  WriteBack  |   F0   |    10
    3    |  Yes   |  SUBD F8 F6 F2   |  WriteBack  |   F8   |    10
    4    |  Yes   |  DIVD F10 F0 F6  |    Issued   |  F10   |
    5    |  Yes   |  ADDD F6 F8 F2   |  WriteBack  |   F6   |    10
    6    |   No   |                  |             |        |
    7    |   No   |                  |             |        |
    8    |   No   |                  |             |        |

   name  |  busy  |    Op    |  Vj  |  Vk  |  Qj  |  Qk  |  Dest  |  A
-----------------------------------------------------------------------
   Add0  |   No   |   SUBD   |  F6  |  F2  |      |      |   3    |
   Add1  |   No   |   ADDD   |  F8  |  F2  |      |      |   5    |
   Add2  |   No   |          |      |      |      |      |        |
  Mult0  |  Yes   |  MULTD   |  F2  |  F4  |      |      |   2    |
  Mult1  |  Yes   |   DIVD   |  F0  |  F6  |      |      |   4    |
  Mult2  |   No   |          |      |      |      |      |        |
  Load0  |   No   |    LD    |      |      |      |      |   0    |  44
  Load1  |   No   |    LD    |      |      |      |      |   1    |  55
  Load2  |   No   |          |      |      |      |      |        |

Field   |  F0 |  F1 |  F2 |  F3 |  F4 |  F5 |  F6 |  F7 |  F8 |  F9 | F10
Reorder |  2  |     |  1  |     |     |     |  5  |     |  3  |     |  4
Busy    | Yes |  No |  No |  No |  No |  No | Yes |  No | Yes |  No | Yes
```

图 12. clock 16

第十七个周期，将 MULTD 和 SUBD 两条指令一起提交（这里用到了一周期可以 commit 多指令的假设）

```
-------------------------Clock Cycle: 17-------------------------

  Entry  |  busy  |   Instruction    |    State    |  Dest  |  Value
-------------------------------------------------------------------
    0    |   No   |   LD F6 34+ R2   |    Commit   |   F6   |    10
    1    |   No   |   LD F2 45+ R3   |    Commit   |   F2   |    10
    2    |   No   |  MULTD F0 F2 F4  |    Commit   |   F0   |    10
    3    |   No   |  SUBD F8 F6 F2   |    Commit   |   F8   |    10
    4    |  Yes   |  DIVD F10 F0 F6  |  Executing  |  F10   |
    5    |  Yes   |  ADDD F6 F8 F2   |  WriteBack  |   F6   |    10
    6    |   No   |                  |             |        |
    7    |   No   |                  |             |        |
    8    |   No   |                  |             |        |

   name  |  busy  |    Op    |  Vj  |  Vk  |  Qj  |  Qk  |  Dest  |  A
-----------------------------------------------------------------------
   Add0  |   No   |   SUBD   |  F6  |  F2  |      |      |   3    |
   Add1  |   No   |   ADDD   |  F8  |  F2  |      |      |   5    |
   Add2  |   No   |          |      |      |      |      |        |
  Mult0  |   No   |  MULTD   |  F2  |  F4  |      |      |   2    |
  Mult1  |  Yes   |   DIVD   |  F0  |  F6  |      |      |   4    |
  Mult2  |   No   |          |      |      |      |      |        |
  Load0  |   No   |    LD    |      |      |      |      |   0    |  44
  Load1  |   No   |    LD    |      |      |      |      |   1    |  55
  Load2  |   No   |          |      |      |      |      |        |

Field   |  F0 |  F1 |  F2 |  F3 |  F4 |  F5 |  F6 |  F7 |  F8 |  F9 | F10
Reorder |  2  |     |  1  |     |     |     |  5  |     |  3  |     |  4
Busy    |  No |  No |  No |  No |  No |  No | Yes |  No |  No |  No | Yes
```

图 13. clock 17

3.3 **最终执行结果.** 以下是 input1 的结果

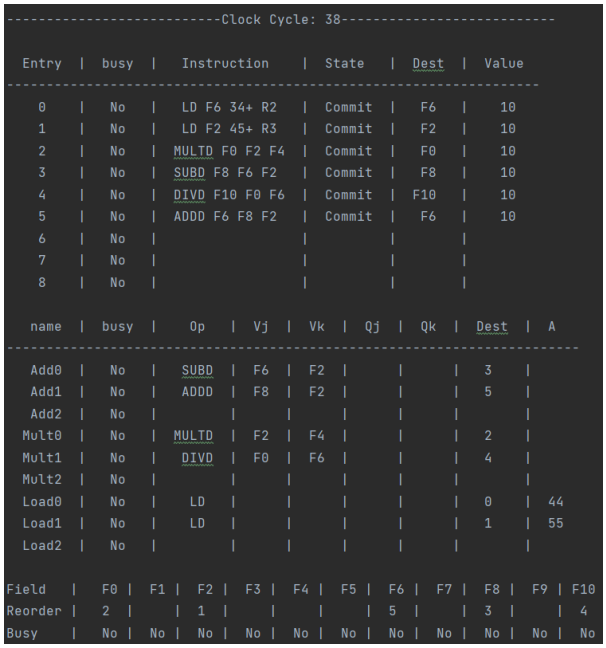

图 14. clock 38

指令	发射	完成	写回	提交
LD F6 34+ R2	1	3	4	5
LD F2 45+ R3	2	4	5	6
MULTD F0 F2 F4	3	15	16	17
SUBD F8 F6 F2	4	7	8	17
DIVD F10 F0 F6	5	36	37	38
ADDD F6 F8 F2	6	10	11	38

表 1. 指令周期表 output1

以下是 input2 的结果，分析部分省略

指令	发射	完成	写回	提交
LD F2 0 R2	1	3	4	5
LD F4 0 R3	2	4	5	6
DIVD F0 F4 F2	3	25	26	27
MULTD F6 F0 F2	4	36	37	38
ADDD F11 F4 F2	5	7	8	38
SD F6 0 R3	6	39	40	41
MULTD F13 F0 F2	7	36	37	41
SD F6 0 R1	8	40	41	42

表 2. 指令周期表 output2

4. 附加题

4.1 Tomasulo 算法的优缺点.

4.1.1 Tomasulo 算法相比 Scoreboard 算法的优点.

消除假数据冒险：Tomasulo 算法通过寄存器重命名的思想消除了假数据冒险(写后写和读后写冒险),从而提高了处理器的乱序执行性能。在 Scoreboard 算法中,假数据冒险会导致不必要的停滞,而 Tomasulo 算法通过动态地分配资源，避免了这种情况。

指令级并行度提高：保留站的引入允许多条指令在等待执行资源时并行地处于保留站中，这比 Scoreboard 算法中的单一配置通路存储更有效率。因此，Tomasulo 算法能更好地利用执行单元，提高并行度。

更有效的资源利用：在 Tomasulo 算法中，一旦指令的源操作数可用，它们就被拷贝到保留站中，减少了对寄存器的依赖。这种数据流动方式使得后续的指令不必等待前序指令完成，从而更有效地利用资源。

动态调度：Tomasulo 算法支持动态调度，它能根据运行时的情况调整指令的执行顺序，而 Scoreboard 算法则更多地依赖于静态调度。

4.1.2 Tomasulo 算法的缺点.

复杂性：Tomasulo 算法的实现比 Scoreboard 算法复杂。它需要更多的硬件支持，如保留站、寄存器结果状态表和公共数据总线（CDB），这些增加了 CPU 设计的复杂性。

选择执行指令的挑战：在同一时间，可能有多条指令准备好执行，但由于执行单元的限制，必须从中选择一条指令。这需要额外的策略来决定哪些指令优先执行。

写回冲突：Tomasulo 算法中，在 CDB 上可能发生多条指令同时完成并准备广播的情况，这需要处理冲突，可能导致性能下降。

不支持精确中断：Tomasulo 算法不支持精确中断，这对于处理中断、异常以及程序调试等方面带来了挑战。解决这一问题需要引入更复杂的机制，如重排序缓冲区。

4.2 **引入重排序改进 Tomasulo 的原理.**

4.2.1 ROB **引入的原因.** Tomasulo 算法虽然提高了处理器的乱序执行性能，但它带来了乱序提交的问题。在 Tomasulo 算法中，一旦指令执行完毕且可以写回，它就会立即写回，这导致了指令的提交顺序与程序原有的顺序不一致。这种乱序提交违背了冯诺依曼体系结构向程序员承诺的“指令按程序顺序执行”的原则，给程序调试和处理中断/异常带来了困难。为了解决这个问题，重排序缓存（ROB）被提出，它允许指令乱序执行，但保证了指令的顺序提交。

4.2.2 ROB **的核心原理.** ROB 的核心思想是记录指令在程序中的原始顺序，并且在这个缓冲区中暂存指令的执行结果。每条指令在发射时进入 ROB，并在执行完成后，其结果被存储在 ROB 中，而不是立即更新到寄存器或内存。ROB 以 FIFO（先进先出）的方式管理指令，确保最先发射的指令在其所有前置指令提交后才能提交。

ROB 的每个条目通常包含几个关键信息：指令类型、目标寄存器、执行结果、指令的状态（如是否已完成执行）等。在指令执行完毕后，ROB 会监控并等待所有在它之前的指令提交完成，然后才允许该指令按照原始程序顺序提交其结果。

4.2.3 ROB **的改进.** 顺序提交与精确中断：通过 ROB，尽管指令内部乱序执行，但最终的提交顺序与程序中的顺序一致，从而支持了精确中断。这使得处理器能够在任何特定指令之间准确地处理中断和异常，而不会受乱序执行的干扰。

分支预测的支持：ROB 为分支预测提供了方便。在分支预测失败的情况下，处理器可以轻松地丢弃或回滚 ROB 中的相关指令，恢复到正确的执行路径。

改善数据流动和保留站的利用：在 ROB 的帮助下，Tomasulo 算法中的保留站可以在指令执行后立即释放，从而提高了保留站的利用率和整体处理器的吞吐量。

4.2.4 **小结.** 重排序缓存的引入使得 Tomasulo 算法在维持乱序执行的同时，实现了指令的顺序提交，从而支持了精确中断和更加有效的分支预测处理。这一改进显著提升了乱序执行处理器的实用性，使得它更符合现代处理器设计的需求。通过这种方式，ROB 有效地补足了 Tomasulo 算法的一些核心缺陷，尤其是在程序的可控性和中断处理。

4.3 **重排序缓存的缺点.** 增加硬件复杂性和资源需求：ROB 的实现增加了处理器设计的复杂性。它需要额外的硬件资源来存储和管理指令的状态信息，包括指令类型、目标寄存器、执行结果等。这不仅增加了硬件成本，也可能导致更大的电路面积和更高的能耗。

读取数据的复杂性增加：在使用 ROB 的体系结构中，指令需要从多个来源（寄存器堆、CDB 和 ROB）获取数据，这增加了数据获取的复杂性。特别是在多端口 ROB 设计中，为了同时支持多个数据源的读取，可能需要更复杂的控制逻辑和增加的布线压力。

可能的性能瓶颈：在某些情况下，ROB 可能成为性能的瓶颈。例如，在高度并行的处理器中，ROB 可能需要支持多个读写端口，以便同时处理多个指令。这可能导致 ROB 的设计变得更加复杂和昂贵，同时可能增加处理器的关键路径长度。

对设计和调试的挑战：ROB 的加入使得处理器的整体设计更加复杂，这不仅对处理器的设计带来挑战，同时也可能增加调试和验证的难度。在发现和修复处理器中的错误时，设计人员可能需要考虑 ROB 的状态和行为，这可能使得调试过程变得更加困难。

与分支预测的复杂交互：尽管 ROB 支持更有效的分支预测，但它也使得处理分支预测失败的情况变得更复杂。在分支预测失败时，处理器需要正确地处理 ROB 中的指令，这可能涉及到复杂的逻辑来确定哪些指令需要被撤销或重执行。

总结来说，ROB 在提高乱序执行处理器性能方面发挥了关键作用，但同时也带来了硬件复杂性、性能瓶颈和设计挑战等缺点