Speculative Tomasulo 仿真算法

一：项目简介

该项目基于作者本人选修的**计算机体系结构课程的期末实验项目**，实现了Speculative Tomasulo 的仿真算法。

Speculative Tomasulo算法是一种用于动态调度指令执行的技术，通常应用于超标量处理器中。该算法通过指令发射、操作数就绪、执行、结果写回等步骤实现指令的乱序执行和并行处理。它利用Reservation Station跟踪指令的操作数可用性，并使用重命名表来处理异常和维护执行状态。通过分支预测和乱序执行，Speculative Tomasulo算法能够最大程度地提高处理器的性能和效率。

本项目按照Speculative Tomasulo算法的基本原理进行设计，并参照课程教学中的合理假设，最后设计了多个测试输入指令序列并验证算法的正确性。

二：第三节的附加问题

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

Tomasulo算法相较于Scoreboard算法的优点:

- Tomasulo引入保留站之后每条通路可以缓冲下多条指令，这样的做法平缓了指令发射的速度；
- 通过寄存器重命名技术有效地解决了写后写数据冒险问题。在Tomasulo算法中，发生写后写冒险时总是把最新的值写进寄存器，旧值不写进寄存器，但是广播；
- 通过寄存器重命名技术有效地解决了读后写数据冒险问题。Tomasulo算法里不会出现读后写冒险，因为指令一旦发射，指令就会把能读取的数据保存到保留站，源寄存器是否被改写就与该指令无关。

Tomasulo存在的缺点：

- Tomasulo算法没办法处理中断，精确中断是指在指令和指令之间如果出现了中断/异常，那么处理器要确保中断/异常之前的所有指令都执行完毕，而中断/异常之后的所有指令都没有执行。Tomasulo并不支持指令按序提交，如果指令没办法按序提交，那就很难处理分支指令，如果有分支预测且分支预测失败的话，很难恢复处理器状态。
- Tomasulo算法如果在一个周期内同时多条指令就绪或者多条指令写回，受制于总线以及执行单元数量，无法实现完全并行。并且处理多发射指令时，需要增加寄存器读写口，增加控制逻辑。
- Tomasulo算法在处理存储指令的冲突时，需要额外加入控制逻辑以及硬件支持。

2. 简要介绍引入重排序改进 Tomasulo 的原理。

在经典的Tomasulo算法中，指令的提交是乱序的，这可能导致不符合程序员期望的执行顺序，分支指令难以处理。为了解决这个问题，引入了ROB，作为指令执行的缓冲区。ROB保持指令的执行顺序，并在指令执行完毕后等待有序提交。

ROB是一个类似FIFO队列的结构，其中包含了每条指令的信息，如编号、Busy位、State位、Destination和Value等字段。指令在发射时被分配ROB编号，等到执行完毕后再提交。ROB的结构允许指令在执行完毕后保持在队列中，而不立即修改逻辑寄存器。
指令在ROB中按照它们在程序中的顺序等待提交。当一条指令成为ROB中最老的指令时，即ROB的头指针指向该指令，就可以有序提交。这确保了指令在逻辑寄存器中的修改按照程序顺序完成。与此同时，ROB也能代替RS的作用实现寄存器重命名，从而保证指令的效率。

ROB的有序提交机制实现了精确中断，确保了在中断发生时能够保持指令的有序提交，而不会导致程序执行错误。ROB的引入允许在指令执行时清除保留站，从而提高了后续指令的发射效率。通过引入ROB，改进的Tomasulo算法克服了原始Tomasulo算法中的一些限制，使得乱序执行更符合程序员的预期，并提高了整体的性能和灵活性。

3. 请分析重排序缓存的缺点。

- 重排序缓存需要更多存储空间：在基于ROB的Tomasulo算法中，一个逻辑寄存器的结果被拷贝到多个地方，数据可能存在逻辑寄存器中，也可能存在保留站中，也可能存在ROB中，即一个数据需要三倍于数据长度的存储空间，从而需要更多资源。
- 重排序缓存需要更多的硬件资源：指令的源数据可以从寄存器堆、CDB总线和ROB中取得。为了支持指令读取数据，需要在 ROB 中配置读口，增加了处理器内部逻辑和数据传递的布线压力。可能导致关键路径的延长，从而影响整体性能。此外，为了处理 ROB 中的数据，需要在读取数据的线路末尾增加选择器，进一步增加了设计的复杂性。
- 重排序缓存可能引入延迟：尽管 ROB 可以提高指令级并行性，但在某些情况下，指令需要等待 ROB 中的其他指令完成执行才能提交。这可能引入一定的延迟。
- 重排序缓存在多发射处理器中具有很大难度。为了满足多个指令同时读取 ROB 中的状态，ROB 需要支持多个读端口。在四发射的机器中，如果每个指令都需要两个读端口，ROB 就需要支持八个读端口。这种多端口读的实现增加了硬件结构的复杂性，可能导致更大的芯片面积和更高的功耗，同时也增加了设计和验证的难度。

三：项目结构

```
Speculative Tomasulo
├─ input                 // 输入样例
|  ├─ input1.txt
|  └─ input2.txt
├─ output                // 输出结果
|  ├─ output1.txt
|  └─ output2.txt
└─ src                   // 源代码
   ├─ main.py            // 主程序
   └─ cpu_component.py           // CPU中的功能单元
```

请在运行前配置好环境依赖，并使得项目结构如上所示，运行方式：

```
cd src
python main.py
```

或者在IDE中运行main.py

四：新增假设

针对作业要求中未说明的细节，在本次实验中采用以下假设，假设原理符合课件“Chap03-ILP-Part3-Speculation-v2”内容。

1. 作业要求中假设有三个加载缓冲区插槽和三个存储缓冲区插槽，因为仿真器架构中并无存储缓冲区、课件与书本内容中均将存储缓冲区与ROB结合，因此在本次实验中假设有三个加载缓冲区插槽，存储缓冲区插槽集成在ROB中实现。

2. 假设Load指令经历IF/EX/WB/COMMIT四个阶段，指令发射与提交各需一个时钟周期，EX执行地址计算与加载内存访问(共需两个时钟周期)；**Store 指令经历IF/EX/COMMIT三个阶段，在EX阶段执行地址计算并等待需要写入数据，因为Store指令无需广播，因此直接连接COMMIT阶段**，所以在无需等待数据以及满足Commit条件时，Store指令共需三个时钟周期。
3. 假设ROB使用循环队列的方式，队列长度为6。
4. 假设RS保留站的数量分别为：Load为2个、Add为3个、Mult为2个；**并且实际的可并行的功能单元数量与RS保留站数量相等**，即同一功能单元不同保留站中的指令可以同时执行。
5. 在结果输出部分中，为了能够能明显的突出实现结果，进行以下约定：ROB采用循环队列的模式，因此展示最近的六条ROB记录；RS记录中，若vj、qj中数据是直接从寄存器组中获得，则输出类似于`Reg[F1]`；若是从总线广播或者ROB与寄存器组连接数据线中获得，则输出类似为`#1`，具体数值可以参照对应ROB中的`value`属性(本次实验中，总线参照真实模型传输数据，只是在输出时进行了统一简化)；vk、qk中数据输出类似于`#x`，表示等待对应ROB x 中结果。此外类型为`Load`的RS保留站只负责输出`Load`类型缓存区，`Store`缓冲区内容集成到ROB中进行输出。其余输出内容均按照假设进行。
6. 最终指令最终执行情况表中，输出格式“(Instruction):(Issue cycle),(Exec comp cycle),(Write result cycle),(Commit cycle)；”。例如`LD F6 34 R2: 1,3,4,5`，代表该指令在时钟 1 进行发射并完成，在时钟 3 结束时执行完毕，在时钟 4 进行Write result并完成，在时钟 5 进行Commit并完成。
7. 能够实现的指令仅包含两个输入样例中出现过的指令，指令格式参考输入样例中格式。

五：仿真器实现思路与具体实现方法

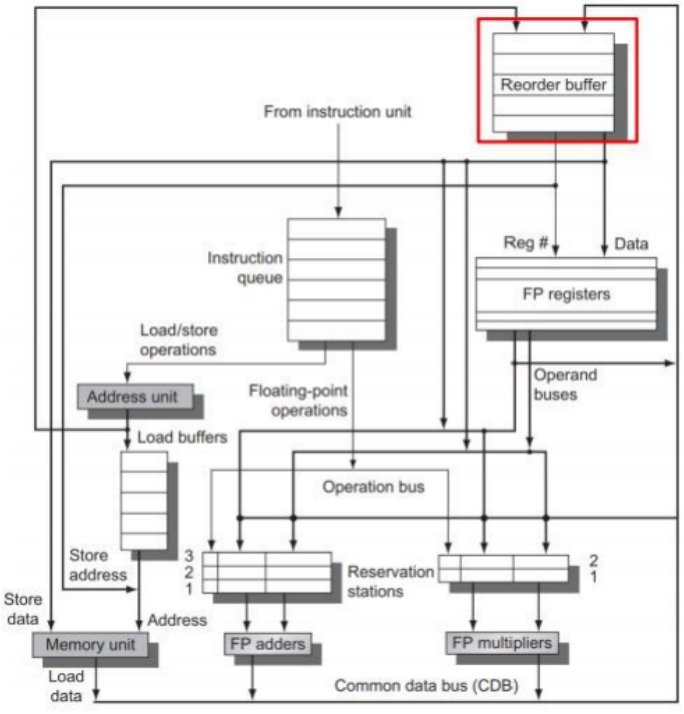

图 2. 仿真器架构

依照仿真器架构进行设计，设计各个CPU中的功能部件，例如：ROB、RS保留站、FP计算部件、总线、寄存器组、内存等等；最后通过一个CPU类将各个功能部件组合在一起，同时由CPU提供控制逻辑，最大程度还原CPU中Speculative Tomasulo算法工作原理。

源代码文件中完整代码以及更加详细、可读性高的注释，下面展示部分略去部分注释以及一些较简单的代码，旨在突出本次实验仿真器实现思路与的具体实现方法，突出关键代码。

CPU 类

`CPU` 类位于 `main.py` 中，仿真CPU的工作过程，将在cpu_component中实现的各个CPU中功能单元按照所要求的的仿真器架构搭建起来，总控各个功能单元并在类函数中实现模拟运行、按要求保存运行结果。
`CPU`类的私有属性为各个功能单元的实例化，时钟周期以及指令队列

```
class CPU:
    def __init__(self, num_registers, memory_size, num_load_buffers,
num_rob_entries,
                 instruction_queue):
        self.bus = Bus()    # 创建总线
        self.rob_bus = Bus()    # 创建rob使用的数据bus
        self.register_group = RegisterGroup(num_registers, rob_bus=self.rob_bus)
   # 创建寄存器组
        self.memory = Memory(memory_size, bus=self.bus,
num_load_buffers=num_load_buffers)  # 创建内存
        self.fp_add = FPUnit(unit_type="Add", num_reservation_stations=3,
execution_cycles={
            "ADDD": 2, "SUBD": 2
        }, bus=self.bus)    # 创建浮点数执行单元
        self.fp_multd = FPUnit(unit_type="Mult", num_reservation_stations=2,
execution_cycles={
            "MULTD": 10, "DIVD": 20
        }, bus=self.bus)
        self.reorder_buffer = ReorderBuffer(num_rob_entries, bus=self.bus,
rob_bus=self.rob_bus)
        self.clock_cycles = 0    # 初始化时钟周期计数
        self.instruction_queue = instruction_queue  # 设置初始指令队列
```

`CPU` 类的 `run_simulation` 方法模拟整个过程开始，管理CPU时钟。该函数负责调用类中的其他函数，进行模拟指令发射、运行、写回过程：

1. 调用 `issue_instruction` 方法发射指令；
2. 调用`update_component` 方法，执行一个时钟周期，模拟真实环境中各个功能单元根据时钟执行操作更新状态；
3. 调用寄存器组和数据总线的 `update` 方法，模拟真实环境中寄存器组以及数据总线中根据时钟的数据写回过程；
4. 按要求记录本周期中ROB、RS保留站以及寄存器组状态。
   按照输出要求将每阶段的输出写入output文件中，并根据所有功能单元都工作结束后结束模拟。

```
def run_simulation(self):
    with open(output_file, 'w') as output:
        # 用于处理前后两个周期输出相同状态
        pre_state = ""
        same_counter = 0
        while True:
            self.clock_cycles += 1  # 模拟时钟周期开始
            print(f"Clock Cycle: {self.clock_cycles}")
            self.issue_instructions()    # 阶段 1：发射指令
            self.update_components()     # 阶段 2：更新各个组件
            self.register_group.update()     # 阶段 3：模拟写回
```

```
            self.bus.update()
            self.rob_bus.update()
            new_state = self.record_component_state(self.clock_cycles)    # 记录组
件状态到文件，包含处理重复输出操作
            ... # 检查新状态是否与前一状态不同
            if self.are_all_components_idle():  # 检查是否所有组件都处于空闲状态
                ...# 如果是，则模拟结束.按要求添加每条指令四个阶段代表周期
```

`CPU` 类的 `issue_instruction` 方法负责按序发射指令：

1. 先检查是否有未发射指令；
2. 若有则继续检查是否有空闲的ROB可以发射；
3. 根据指令的不同判断是否有空闲的RS可以发射，并且判断指令的操作数是否就绪；
4. 若指令会更新寄存器则调用寄存器组更新函数，进行记录；

```
    def issue_instructions(self):
        if self.instruction_queue:  # 检查指令队列是否非空
            instruction = self.instruction_queue[0]
            # 检查指令是否可以发射
            sd_vj, sd_qj = self.register_group.read(instruction.destination)
            rob_index = self.reorder_buffer.issue_instruction(instruction,
self.clock_cycles, sd_vj, sd_qj)

            if not rob_index:   # 没有空闲ROB时，发射失败
                return False
            # 根据指令类型调用相应的功能单元
            if instruction.opcode in {"ADDD", "SUBD"}:
                vj, qj = self.register_group.read(instruction.src1)
                vk, qk = self.register_group.read(instruction.src2)
                if self.fp_add.issue_instruction(instruction, vj, vk, qj, qk,
rob_index):
                    self.instruction_queue.pop(0)
                else:
                    self.reorder_buffer.clear_rob()
            elif instruction.opcode in {"MULTD", "DIVD"}:
                ... #类似
            elif instruction.opcode == "LD":
                ... #类似
            elif instruction.opcode == "SD":
                self.instruction_queue.pop(0)
                return
            else:
                raise ValueError(f"Error Instruction!")
            # 更新目标寄存器的值
            self.register_group.write(instruction.destination, rob_index)
```

`CPU` 类中还有三个方法，但不是本次实验重点部分。 `update_components` 方法负责调用各个功能部件的更新函数，实现按照时钟的更新(只是简单调用)； `record_component_state` 方法负责将每个周期的ROB、RS保留站、寄存器组状态转化为符合要求格式的字符串返回； `are_all_components_idle` 方法负责调用各个功能部件的 `finish` 函数，用于判断是否整个仿真过程结束；(详细代码与注释可以参考源代码模块)

`ReorderBuffer` 类

`ReorderBuffer` 类中采用循环队列的方式实现了多条rob条目，通过 `bus` 总线进行数据传输，通过 `rob_bus` 与存储器进行数据传输。

```
class ReorderBuffer:
    def __init__(self, size, bus, rob_bus):
        self.size = size + 1  # 多一个位置实现循环队列
        self.bus = bus
        self.rob_bus = rob_bus
        ...
```

`ReorderBuffer` 类的 `issue_instruction` 方法实现指令发射到ROB当中。尝试在ROB缓冲区中创建新的ROB条目，将其加入缓冲区，并更新ROB头尾指针。如果ROB缓冲区未满，成功发射的ROB条目将被标记为"Issue"状态，其发射周期将被记录。对于SD指令，目标寄存器将被置为None，而vj和qj将被记录在SD数据字段中。如果ROB缓冲区已满，则发射失败。

```
    def issue_instruction(self, instruction, clock_cycle, vj, qj):
        self.rob_index_counter += 1  # 创建一个新的ROB条目
        rob_entry = ReorderBufferEntry(self.rob_index_counter, instruction)

        next_tail = (self.tail + 1) % self.size  # 尝试将ROB条目加入ROB缓冲区
        if next_tail != self.head:  # 如果缓冲区未满，加入ROB条目
            ...# 处理普通条目与SD条目
            return rob_entry.rob_index
        else:   # 如果缓冲区已满，发射指令失败
            self.rob_index_counter -= 1
            return None
```

`ReorderBuffer` 类的 `update` 方法负责更新ROB组中条目。首先检查总线上是否有新数据，然后遍历ROB缓冲区中的每个条目，根据其状态和总线上的数据更新条目状态。同时，将满足条件的条目移动到Commit状态。如果总线上有新数据，且ROB中有对应的条目，则使用总线上的数据更新条目的值，并将其状态设为"Write result"，同时尝试将结果写回寄存器。最后，更新ROB的头指针。

```
    def update(self, clock_cycle):
        label, data = self.bus.read()
        exec_list = self.bus.exec  # 检查总线上是否有新数据
        index = self.head
        # 根据不同条件，更新ROB的四个状态
        while index != self.tail:
            entry = self.entries[index]
            if entry.state == "Issue" and entry.rob_index in exec_list:
                entry.state = "Exec"
            ... # 类似
        self.head = self.new_head
```

因为将存储缓冲区集成到ROB当中，所以 `ReorderBuffer` 类还有一个特殊的 `update_sd` 方法：该函数根据SD指令的特殊逻辑更新ROB中的条目状态，包括处理等待的操作数和转移到下一个状态。(与 `update` 方法并无太大区别，因此不再展示)

`Bus` 类

总线类，用于在执行单元之间传递数据和标签。属性包含当前总线上的标签、数据以及新写入的标签、数据。

```
class Bus:
    def __init__(self):
        self.label = ""
        self.value = ""
        self.new_label = ""
        self.new_value = ""
        self.exec = []
```

`Bus`类中还有`read`、`write`以及`update`方法，这些方法负责简单的读写操作，`update`方法检查总线是否有新的数据，如果有，则更新当前总线的标签和数据，否则清空总线上的标签和数据，并清空执行列表。

```
def read(self):
    return (self.label, self.value)
def write(self, label, data):
    if not self.new_label:
        self.new_value = data
        self.new_label = label
        return True
    else:
        return False
def update(self):
    if self.new_label:
        self.value = self.new_value
        self.label = self.new_label
        self.new_value = ""
        self.new_label = ""
    else:
        self.value = ""
        self.label = ""
    self.exec = []
```

说明：真实CPU中总线不会传递当前正在执行指令，因为项目输出中ROB中需要及时更改每个条目状态，因此在借助总线额外传递了一些信息(真实中ROB也不会在执行中更改，为了符合输出要求进行了一些合理修改)。

`FPUnit` 类

`FPUnit`类，模拟浮点运算单元的抽象，用于执行浮点数运算。该类包含多个保留站（ReservationStation）以协调浮点数指令的执行。初始化浮点数运算单元时，需要指定执行单元类型、保留站数量、操作的执行周期，以及总线对象用于与总线通信。(同时也有一个简单的`ReservationStation`用以模拟RS保留站，包含RS保留站中各种属性)

`FPUnit` 类的 `issue_instruction` 方法负责将指令发射到保留站中。函数主要是判断是否有可用的保留站，如果成功发射，返回 True；否则返回 False。保留站中参数由`CPU`控制模块传入，该方法只需填入即可。

```
def issue_instruction(self, instruction, vj, vk, qj, qk, rob_index):
```

```
    for rs in self.reservation_stations:
        if not rs.busy():
            # 如果 Reservation Station 可用，发射指令，即在Reservation Station中加入对
应属性
            rs.busy = True
            rs.op = instruction.opcode
            rs.vj = vj
            rs.vk = vk
            rs.qj = qj
            rs.qk = qk
            rs.dest = instruction.destination
            rs.rob_index = rob_index
            rs.remain_time = self.execution_cycles.get(instruction.opcode, 1)
            rs.issue_this_cycle = True
            return True
    # 如果没有可用的 Reservation Station，指令发射失败
    return False
```

`FPUnit` 类的 `update` 方法负责执行一个周期。它遍历每个保留站，如果该保留站的 `busy` 字段为 `True`，则通过 `counter` 判断指令处于哪个阶段并作相应操作；该函数会返回在该周期执行完毕的指令的 PC，通过总线传递给ROB，实现ROB状态的更新操作

```
def update(self):
    label, data = self.bus.read()   # 从总线中读取写入的数据

    for rs in self.reservation_stations:
        if rs.busy:
            if rs.qj or rs.qk:  # 操作数未就绪，判断总线中广播数据是否需要
                if data and rs.qj == label:
                    rs.vj = f"#{label}"
                    rs.qj = None
                elif data and rs.qk == label:
                    rs.vk = f"#{label}"
                    rs.qk = None
            elif rs.remain_time > 0:  # 操作数已经就绪，执行
                if rs.issue_this_cycle:  # 因为发射指令需要一个周期，因此需要跳过新发射的
指令
                    rs.issue_this_cycle = False
                    continue
                self.bus.exec.append(rs.rob_index)  # 将正在执行EX阶段的指令传递给
ROB，用以修改ROB状态
                # 执行阶段
                rs.remain_time -= 1
                if rs.remain_time == 0:  # 若执行完成，根据要求输出格式记录结果
                    if isinstance(rs.vj, str):
                        vj_result = rs.vj
                    else:
                        vj_result = f"Reg[F{rs.vj}]"
                    if isinstance(rs.vk, str):
                        vk_result = rs.vk
                    else:
                        vk_result = f"Reg[F{rs.vk}]"
                    if rs.op == "ADDD":
                        result = f"{vj_result} + {vk_result}"
                    elif rs.op == "SUBD":
```

```
                        result = f"{vj_result} - {vk_result}"
                    elif rs.op == "MULTD":
                        result = f"{vj_result} * {vk_result}"
                    elif rs.op == "DIVD":
                        result = f"{vj_result} / {vk_result}"
                    else:
                        raise ValueError(f"Error operation!")
                    if not self.bus.write(rs.rob_index, result):  # 若当前总线有写入
阶段，则需要下个周期再次尝试写入
                        rs.remain_time += 1
            else:
                rs.busy = False
            rs.issue_this_cycle = False
```

`FPUnit` 类的 `finish` 方法简单的检查是否所有的RS保留站均已经释放，即处于busy=False的状态。若全都空闲，则返回完成；否则返回未完成。(详细代码见源代码模块)

`Memory` 类

`Memory` 类，模拟计算机中的内存单元，用于进行数据的存储读取操作。另外，该类包含 `Load` 缓冲区(Load Buffer)，使用与上文中 `FPUnit` 中一样的数据结构RS（ReservationStation）以模拟 `Load` 指令加载内存的执行。初始化浮点数运算单元时，需要指定 `Load Buffer` 数量、内存大小，以及总线对象用于与总线通信。

在 `Memory` 类中主要实现的方法为 `issue_instruction` 以及 `update`，分别用于发射指令与根据总线结果、每个时钟周期正确更新缓冲区状态。实现方式与上文所示 `FPUnit` 中实现方式一致，因此不再赘述。同时也在类中实现了 `finishh`、`read`、`write` 等简单辅助函数，用以更加真实模拟内存读写情况与辅助 `CPU` 调用判断是否完成所有指令。

`RegisterGroup` 类

`RegisterGroup` 类是一个用于管理通用寄存器和基址寄存器的实用工具。通过指定寄存器数量和与重排序缓冲区（ROB）通信的总线对象，创建了一组寄存器，并提供了**读取和写入数据的方法**。在时钟周期中，通过与ROB通信，可以**更新寄存器的状态**。实现方法较简单，具体参照源代码模块。(同时有一个简单的 `Register` 类用以模拟寄存器，包含寄存器各种状态属性)

辅助函数与`main`函数

最后还有一些辅助函数，例如 `parse_instruction`、`trans`、`rs_state` 等等用于：辅助处理指令，进行适当转换，记录RS保留站状态。(因为实现简单，均不再赘述)。

主函数模块，负责实例化 `CPU` 类，读取指令序列并处理，最后运行模拟过程。

六：优化项目点

1. 实现了一个周期内总线上只有一次合法的写事务，从而避免了同一周期对总线多次写入造成错误结果。
2. **实现了对于ROB中的条目的清除**，用于模拟分支条件判断语句错误时，对应的ROB操作。(更加突出Speculative Tomasulo算法特点)
3. **实现了避免存储器冒险的操作**，推测可以消除存储器中的 WAW 和 WAR 冒险；通过额外限制条件解决存储器中的 RAW 冒险：若一条存储指令占用的活动ROB项目的“目的地”字段与一条载入指令的A字段取值匹配，则不允许该载入指令开始执行第二步骤。

4. 操作数的来源可以是ROB、寄存器组、总线，本项目中合理考虑了多条数据线并发读写的情况。

以上优化功能均在代码中进行了注释。

七：实验结果与分析

实现仿真器的关键组件与思路见上一部分，在完成依赖环境配置之后，运行程序。在主函数中，读取标准输入文件转化为指令队列，设置输出文件，随后实例化`CPU`类(提供必要参数，本次实验参数参照假设进行设置)并调用类中模拟运行的方法即可。模拟运行会在所有指令运行结束之后，自动停止。

本次实验完整输出结果见`output1.txt`和`output2.txt`，这里举例样例 1 中的前几个时钟周期来说明本程序能够正确仿真CPU中 Tomasulo 动态调度算法，对输入指令进行动态调度。**输出参照标准规定输出格式并在上文中对输出含义以及正确性进行了解释**

1. 第 1 个时钟周期，发射第一条指令到ROB以及RS保留站中，RS保留站可以中可以直接读取寄存器组中数据，寄存器组中F6变为busy且对应于ROB条目1，程序输出如下：

```
cycle_1;
entry1 : Yes, fld F6 34(R2), Issue, F6, None;
entry2 :No,,,,;
entry3 :No,,,,;
entry4 :No,,,,;
entry5 :No,,,,;
entry6 :No,,,,;
Load1 : Yes, LD, Regs[R2], , , , #1;
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;
```

2. 第 2 个时钟周期，第一条指令开始执行，ROB中状态变为Exec，同时发射第二条指令到ROB以及RS保留站中，寄存器组中也进行相应修改，程序输出如下：

```
cycle_2;
entry1 : Yes, fld F6 34(R2), Exec, F6, None;
entry2 : Yes, fld F2 45(R3), Issue, F2, None;
entry3 :No,,,,;
entry4 :No,,,,;
entry5 :No,,,,;
entry6 :No,,,,;
Load1 : Yes, LD, Regs[R2], , , , #1;
Load2 : Yes, LD, Regs[R3], , , , #2;
Add1 : NO,,,,,,;
Add2 : NO,,,,,,;
Add3 : NO,,,,,,;
Mult1 : NO,,,,,,;
Mult2 : NO,,,,,,;
Reorder:F0:;F1:;F2: 2;F3:;F4:;F5:;F6: 1;F7:;F8:;F9:;F10:;
Busy:F0:No;F1:No;F2:Yes;F3:No;F4:No;F5:No;F6:Yes;F7:No;F8:No;F9:No;F10:No;
```

3. 第 3 个时钟周期，第一、二条指令处于执行阶段，发射第三条指令到ROB以及RS保留站中(但是该指令所需的操作数 `F2` 依赖于第二条指令，尚未就绪，因此相应设置vj与qj)，同时更新寄存器组中状态，程序输出如下：

```
cycle_3;
entry1 : Yes, fld F6 34(R2), Exec, F6, None;
entry2 : Yes, fld F2 45(R3), Exec, F2, None;
entry3 : Yes, fmul.d F0,F2,F4, Issue, F0, None;
entry4 :No,,,,;
entry5 :No,,,,;
entry6 :No,,,,;
Load1 : Yes, LD, Regs[R2], , , , #1;
Load2 : Yes, LD, Regs[R3], , , , #2;
Add1 : NO,,,,,,;
Add2 : NO,,,,,,;
Add3 : NO,,,,,,;
Mult1 : Yes, MULTD, , Regs[F4], #2, , #3;
Mult2 : NO,,,,,,;
Reorder:F0: 3;F1:;F2: 2;F3:;F4:;F5:;F6: 1;F7:;F8:;F9:;F10:;
Busy:F0:Yes;F1:No;F2:Yes;F3:No;F4:No;F5:No;F6:Yes;F7:No;F8:No;F9:No;F10:No;
```

4. 第 4 个时钟周期，第一条指令执行完毕处于Write Result阶段，结果 `Mem(34+Regs[R2]])`，进行广播；第二条指令处于执行阶段；第三条指令因为操作数未就绪，仍然处于发射阶段；发射第四条指令，接受到广播的操作数 `F6`，将数据填入vj中（但是该指令所需的操作数 `F2` 依赖于第二条指令，尚未就绪），程序输出如下：

```
cycle_4;
entry1 : Yes, fld F6 34(R2), Write result, F6, Mem[34+Regs[R2]];
entry2 : Yes, fld F2 45(R3), Exec, F2, None;
entry3 : Yes, fmul.d F0,F2,F4, Issue, F0, None;
entry4 : Yes, fsub.d F8,F6,F2, Issue, F8, None;
entry5 :No,,,,;
entry6 :No,,,,;
Load1 : NO,,,,,,;
Load2 : Yes, LD, Regs[R3], , , , #2;
Add1 : Yes, SUBD, #1, , , #2, #4;
Add2 : NO,,,,,,;
Add3 : NO,,,,,,;
Mult1 : Yes, MULTD, , Regs[F4], #2, , #3;
Mult2 : NO,,,,,,;
Reorder:F0: 3;F1:;F2: 2;F3:;F4:;F5:;F6: 1;F7:;F8: 4;F9:;F10:;
Busy:F0:Yes;F1:No;F2:Yes;F3:No;F4:No;F5:No;F6:Yes;F7:No;F8:Yes;F9:No;F10:No;
```

5. 第 5 个时钟周期，第一条判断能够Commit，因此修改自身状态并将结果值写入寄存器 `F6` 中，寄存器组进行相应修改；第二条指令执行完毕，将结果 `Mem[45+Regs[R3]]` 通过总线进行广播，同时第三、四条指令在总线中将需要的数据 `F2` 进行读取，转为就绪状态，发射第五条指令（但是该指令所需的操作数 `F0` 依赖于第三条指令，尚未就绪），程序输出如下：

```
cycle_5;
entry1 : No, fld F6 34(R2), Commit, F6, Mem[34+Regs[R2]];
entry2 : Yes, fld F2 45(R3), Write result, F2, Mem[45+Regs[R3]];
entry3 : Yes, fmul.d F0,F2,F4, Issue, F0, None;
entry4 : Yes, fsub.d F8,F6,F2, Issue, F8, None;
```

```
entry5 : Yes, fdid.d F10,F0,F6, Issue, F10, None;
entry6 :No,,,,;
Load1 : NO,,,,,,;
Load2 : NO,,,,,,;
Add1 : Yes, SUBD, #1, #2, , , #4;
Add2 : NO,,,,,,;
Add3 : NO,,,,,,;
Mult1 : Yes, MULTD, #2, Regs[F4], , , #3;
Mult2 : Yes, DIVD, , #1, #3, , #5;
Reorder:F0: 3;F1:;F2: 2;F3:;F4:;F5:;F6:;F7:;F8: 4;F9:;F10: 5;
Busy:F0:Yes;F1:No;F2:Yes;F3:No;F4:No;F5:No;F6:No;F7:No;F8:Yes;F9:No;F10:Yes;
```

6. 第 6 个时钟周期，第二条指令判断能够Commit，将数据写入寄存器`F2`中，相应修改寄存器组状态；第三、四条在上一个周期操作数就绪，因此该周期中进入执行Exec状态；发射第六条指令到ROB以及RS保留站中（但是该指令所需的操作数`F8`依赖于第四条指令，尚未就绪）。第 7 个时钟周期，状态相同，因此合并输出：

```
cycle_6-7;
entry1 : No, fld F6 34(R2), Commit, F6, Mem[34+Regs[R2]];
entry2 : No, fld F2 45(R3), Commit, F2, Mem[45+Regs[R3]];
entry3 : Yes, fmul.d F0,F2,F4, Exec, F0, None;
entry4 : Yes, fsub.d F8,F6,F2, Exec, F8, None;
entry5 : Yes, fdid.d F10,F0,F6, Issue, F10, None;
entry6 : Yes, fadd.d F6,F8,F2, Issue, F6, None;
Load1 : NO,,,,,,;
Load2 : NO,,,,,,;
Add1 : Yes, SUBD, #1, #2, , , #4;
Add2 : Yes, ADDD, , #2, #4, , #6;
Add3 : NO,,,,,,;
Mult1 : Yes, MULTD, #2, Regs[F4], , , #3;
Mult2 : Yes, DIVD, , #1, #3, , #5;
Reorder:F0: 3;F1:;F2:;F3:;F4:;F5:;F6: 6;F7:;F8: 4;F9:;F10: 5;
Busy:F0:Yes;F1:No;F2:No;F3:No;F4:No;F5:No;F6:Yes;F7:No;F8:Yes;F9:No;F10:Yes;
```

执行完毕全部六条指令共需 39 个时钟周期，详细输出结果见`output1.txt`文件，并且程序运行过程中也加入了一些`print`帮助理解指令运行状况。最终每条指令发射、执行结束、写回的时钟周期如下：

```
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: 4,7,8,18
DIVD F10 F0 F6: 5,36,37,38
ADDD F6 F8 F2: 6,10,11,39
```

八：实验总结

Speculative Tomasulo算法的乱序执行和分支预测能够显著提高处理器的性能和效率。通过充分利用可用的执行单元，并在执行过程中处理异常情况，算法能够实现更高的指令级并行性，从而加速指令的执行。

算法的Reservation Station和重命名表在处理器设计中起着关键作用。Reservation Station跟踪指令的操作数可用性，允许指令按需发射和执行，而重命名表则用于处理异常和维护执行状态，确保指令的正确执行和结果的正确写回。

基于作者时间、知识与精力有限，本项目只是一个仿真版的Speculative Tomasulo算法实现。后续还有大量的工作可以进行完善，本项目全部代码与报告展示已经开源于：

项目链接

如有错漏之处 敬请指正🫶