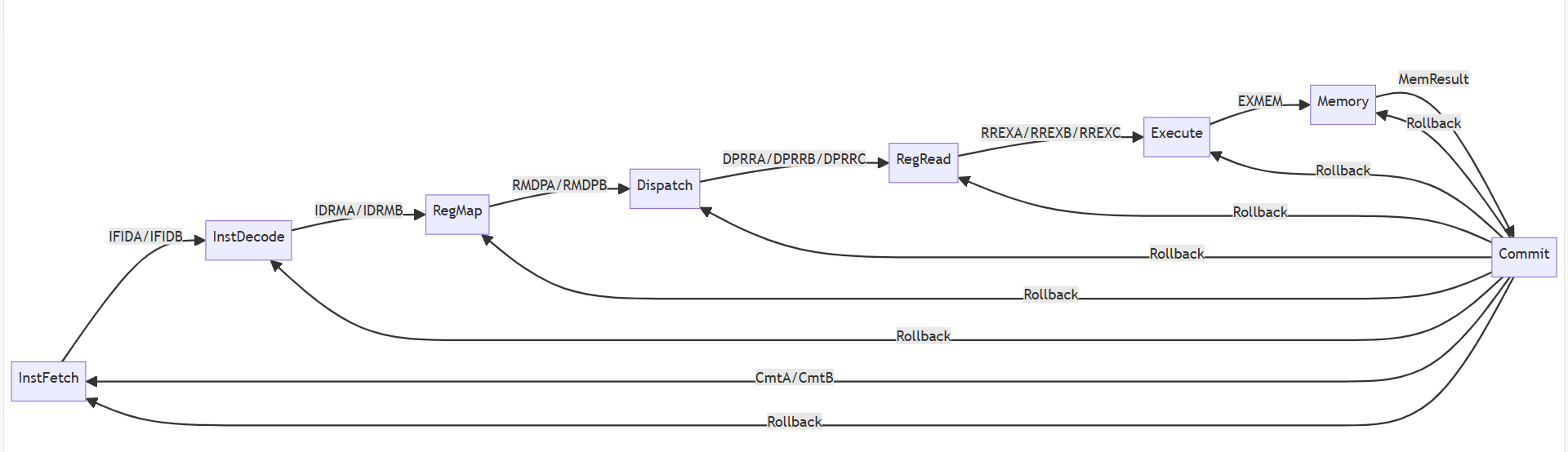
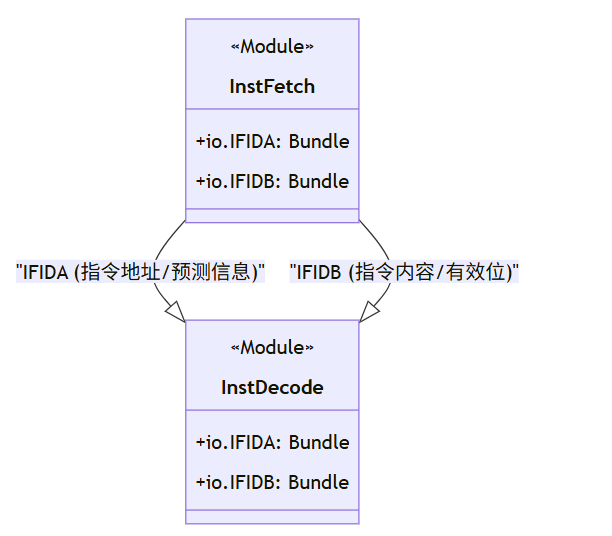
一：顶层核心架构

**1：核心架构图**

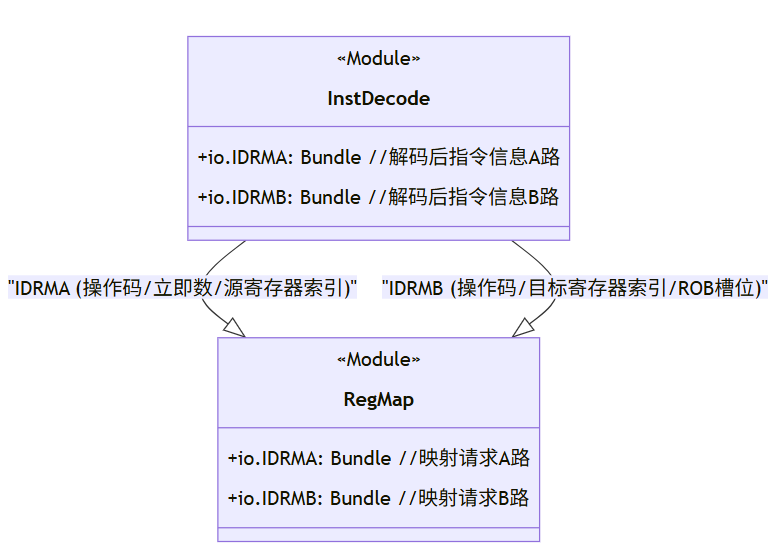
1. **：整体架构**

****

1. **：InstFetch <->InstDecode**

****

**3）: InstDecode ↔ RegMap**

****

**2：外接端口**

|  |  |  |
| --- | --- | --- |
| 名称 | 数据类型 | 描述 |
| InstRam | RAMHelperIO | 连接指令内存 |
| DataRam | RAMHelperIO | 连接数据内存 |
| Uart | UARTIO | 串行接口，用于输入输出 |

**3：内部模块**

指令获取模块：InstFetch

指令解码模块：InstDecode

寄存器映射模：RegMap

指令分发模块：Dispatch

寄存器读取模块：RegRead

执行模块：Execute

内存访问模块：Memory

指令提交模块：Commit

**4**: **流水线连接关系：**

**1）流水线主数据通路连接：**

取指与译码：双发射架构，每个周期从取值阶段向解码阶段传递两条指令。

InstFetch.io.IFIDA <> InstDecode.io.IFIDA

InstFetch.io.IFIDB <> InstDecode.io.IFIDB

解码与重命名：将解码后的两条指令传递给寄存器映射模块进行寄存器重命名。

InstDecode.io.IDRMA <> RegMap.io.IDRMA

InstDecode.io.IDRMB <> RegMap.io.IDRMB

重命名与分发：重命名后的指令传递给分发阶段，为执行做准备。

RegMap.io.RMDPA <> Dispatch.io.RMDPA

RegMap.io.RMDPB <> Dispatch.io.RMDPB

分发与寄存器读取：分发模块向寄存器读取模块传递三条指令(方便后续完成信号传递）

Dispatch.io.DPRRA <> RegRead.io.DPRRA

Dispatch.io.DPRRB <> RegRead.io.DPRRB

Dispatch.io.DPRRC <> RegRead.io.DPRRC

读取与执行：读取阶段将三/二条已读操作数的指令传递给执行阶段

RegRead.io.RREXA <> Execute.io.RREXA

RegRead.io.RREXB <> Execute.io.RREXB

RegRead.io.RREXC <> Execute.io.RREXC

执行与内存：将执行结果（主要是内存访问指令）传递给内存阶段

Execute.io.EXMEM <> Memory.io.EXMEM

**2）重排序与提交机制：**

将 RegMap 发射（Dispatch）阶段的物理寄存器分配信息 RMDPA, RMDPB 顺着整个程序最后传递到 Commit，用于提交时释放物理寄存器资源。

Commit.io.EnA/EnB <> RegMap.io.RMDPA/RMDPB

Commit.io.ReOrderNumA/B <> RegMap.io.ReOrderNumA/B

此处传递回去的时候用于释放物理寄存器资源

将 RegMap 分配给两个指令的重排序缓冲区编号传递给 Commit，便于 Commit 确定哪些指令可以提交、提交顺序等。（因为我们是要顺序提交，而重排序缓冲区编号便是指令的提交顺序）

Commit.io.ReOrderNumA <> RegMap.io.ReOrderNumA

Commit.io.ReOrderNumB <> RegMap.io.ReOrderNumB

从各个阶段阶段获取指令完成信息，用于追踪指令是否已经准备好可以提交。

Commit.io.CmtA/B <> 各模块.io.CmtA/B

将“已提交信息“这个信号广播到需要的模块。

Commit.io.CmtA/B <> 各模块.io.CmtA/B

**3）寄存器状态与前馈连接：**

分发阶段需要物理寄存器状态信息来进行依赖的分析和调度；

Dispatch.io.PhyRegStates <> RegMap.io.PhyRegStates

实现结果前馈，使后续指令能够立即使用最新的计算结果，无需等待提交。

RegMap.io.FinA/B/C/D/E <> 各阶段.io.FinA/B/C/D/E

RegRead.io.FinC/D/E <> 各阶段.io.FinC/D/E

**4）流水线控制信号连接：**

当分发资源不足或提交需要时，组织新的指令进入流水线。

val FetchBlock = Dispatch.io.FetchBlock || Commit.io.FetchBlock

FetchBlock <> 各模块.io.FetchBlock

当发生异常或者分支预测错误时，向所有阶段发送回滚（冲刷）信号。

Commit.io.Rollback <> 各模块.io.Rollback

**5）内存前馈连接：**

实现内存访问指令间的数据前馈，正确处理存储-加载依赖。

Commit.io.ForwardLoad/Store <> Memory.io.ForwardLoad/Store

**6）外接端口连接：**

连接处理器核心与外部指令内存和数据内存

io.InstRam <> InstFetch.io.InstRam

io.DataRam <> Memory.io.DataRam

**5：数据块自定义**

这种数据块本质都是一个数据的集体传递，从而体现了chisel相对于verilog的优势

**1）：JumpIssue Bundle：**

|  |  |  |
| --- | --- | --- |
| 名称 | 数据类型 | 描述 |
| Valid | Bool() | 当前跳转信息是否有效 |
| proTarget | UInt(32.W) | 分支预测给出的跳转地址 |
| actTarget | UInt(32.W) | 实际跳转地址 |
| link | UInt(32.W) | 返回地址（如 JAL 指令） |

**作用：**记录跳转指令（如JAL/JALR）的预测目标地址、实际目标地址和返回地址，用于验证预测正确性和恢复执行流。

**2）：BranchIssue Bundle：**

|  |  |  |
| --- | --- | --- |
| 名称 | 数据类型 | 描述 |
| Valid | Bool() | 当前分支信息是否有效 |
| proTaken | Bool() | 分支预测是否认为会跳转 |
| actTaken | Bool() | 实际是否发生跳转 |
| target | UInt(32.W) | 分支目标地址 |

**作用：**存储分支指令（如BEQ/BNE）的预测方向（Taken/Not Taken）和实际结果，支持分支预测失败时的流水线冲刷和重定向。

**3）：LoadIssue Bundle：**

|  |  |  |
| --- | --- | --- |
| 名称 | 数据类型 | 描述 |
| Valid | Bool() | 当前加载操作是否有效 |
| addr | UInt(32.W) | 加载操作的目标地址 |
| data | UInt(32.W) | 加载得到的数据 |
| Ready | Bool() | 加载是否完成、数据是否可用 |

**作用：**管理加载指令（如LW/LD）的内存地址、加载数据和就绪状态，协调数据缓存或内存系统的访问。

**4）：StoreIssue Bundle（存储指令信息）：**

|  |  |  |
| --- | --- | --- |
| 名称 | 数据类型 | 描述 |
| Valid | Bool() | 当前存储操作是否有效 |
| addr | UInt(32.W) | 存储操作的目标地址 |
| mask | UInt(32.W) | 写掩码，指示哪些字节有效 |
| data | Uint(32.W) | 需要写入的数据 |
| Ready | Bool() | 存储操作是否准备就绪 |

作用：控制存储指令（如SW/SD）的地址、数据、写入掩码和就绪状态，确保数据按指定字节宽度写入内存。

**5）：InsCtrlBlock Bundle:（指令数据块信息传递）**

|  |  |  |
| --- | --- | --- |
| **名称** | **数据类型** | **描述** |
| **基本信息字段** |  |  |
| Valid | Bool | 指令是否有效 |
| Inst | UInt(32.W) | 32位原始指令 |
| Pc | UInt(32.W) | 程序计数器 |
| Isa | ISA | 指令类型译码信息 |
| **乱序执行相关字段** |  |  |
| finish | Bool | 指令是否完成执行 |
| ReOrderNum | Unit(6.W) | 重排序缓冲区编号，用于保证按序提交 |
| **寄存器映射字段** |  |  |
| Regdes | Unit(5.W) | 目标寄存器 |
| Regsrc1 | Unit(5.W) | 源寄存器1 |
| Regsrc2 | Unit(5.W) | 源寄存器2 |
| Pregsrc1 | Unit(7.W) | 重命名后源寄存器1 |
| Pregsrc2 | Unit(7.W) | 重命名后源寄存器2 |
| Pregdes | Unit(7.W) | 重命名后目标寄存器 |
| Cmtdes | Unit(7.W) | 提交阶段使用的目标物理寄存器 |
| **执行阶段数据** |  |  |
| Src1 | UInt(32.W) | 源操作数1 |
| Src2 | UInt(32.W) | 源操作数2 |
| Imm | IMM | 立即数（支持多种格式） |
| Wbdata | UInt(32.W) | 写回数据 |
| **功能单元接口** |  |  |
| Jump | JumpIssue | 跳转指令处理 |
| branch | BranchIssue | 分支指令处理 |
| Load | LoadIssue | 加载指令处理 |
| Store | StoreIssue | 存储指令处理 |

**作用：**作为处理器流水线的核心数据通路，封装了指令的基本信息（PC、指令编码）、寄存器映射（逻辑/物理寄存器）、执行数据（操作数、立即数、写回值）以及分支、跳转、访存等子模块，用于在流水线各级间传递和控制指令的执行流程。

1. ：**ISA Bundle（这是一个指令集类型）（包含所有的指令集类型）**
2. **：imm bundle（这是一个立即数mode）**

|  |  |  |
| --- | --- | --- |
| 名称 | 数据类型 | 描述 |
| I | UInt(32.W) | I-type 指令的立即数（如 ADDI） |
| B | UInt(32.W) | B-type 分支指令的立即数（如 BEQ） |
| S | UInt(32.W) | S-type 存储指令的立即数（如 SW） |
| U | UInt(32.W) | U-type 指令的立即数（如 LUI、AUIPC） |
| J | UInt(32.W) | J-type 跳转指令的立即数（如 JAL） |
| Z | UInt(32.W) | 特殊用途立即数（如 CSR 中的 zimm 字段） |

根据指令类型（I/B/S/U/J/Z）解码和扩展不同格式的立即数，为ALU或地址计算提供操作数。

二：instFetch

1. 概述

IF 模块是五级流水线的第一个阶段，负责从指令存储器中读取指令，并根据分支预测模块（BP）的预测结果决定下一条指令的地址。其主要任务包括：

**指令预取**：根据当前 PC（Program Counter）值从指令存储器中读取指令。

**分支预测支持**：结合 BP 模块的预测结果（predTaken、predTarget），决定下一条指令的地址（顺序执行或跳转）。

**流水线控制**：处理来自 ID 阶段的预测错误纠正（pcRedirect）和流水线暂停（stall）信号，确保指令流的正确执行。

IF 模块与 BP 模块紧密配合，通过提前预测分支行为，减少因分支指令导致的流水线停顿（Control Hazard），从而提高处理器的执行效率。

2. 接口信号

输入信号

|  |  |  |
| --- | --- | --- |
| 信号名称 | 位宽 | 描述 |
| clk | 1 | 时钟信号 |
| reset | 1 | 异步复位信号 |
| stall | 1 | 流水线暂停信号 |
| branchTaken | 1 | 来自ID阶段的分支实际执行结果 |
| branchTarget | 32 | 来自ID阶段的分支目标地址 |
| predTaken | 1 | 来自BP模块的预测跳转信号 |
| predTarget | 32 | 来自BP模块的预测目标地址 |
| pcRedirect | 1 | 来自ID阶段的PC重定向信号 |

输出信号

|  |  |  |
| --- | --- | --- |
| 信号名称 | 位宽 | 描述 |
| pc | 32 | 当前取指地址 |
| instruction | 32 | 取出的指令 |
| pcPlus4 | 32 | PC+4的值 |
| predlndex | N | 传递给BP模块的预测索引 |

3. 功能描述

在时钟上升沿，根据当前状态更新PC值：

复位时，PC被设置为初始地址（如0x00000000）

当流水线暂停时，PC保持不变

当有分支预测错误时（pcRedirect=1），使用ID阶段提供的branchTarget

当分支预测为跳转时（predTaken=1），使用predTarget

其他情况下，PC递增4（顺序执行）

使用PC值从指令存储器中读取指令

计算PC+4的值并输出

将当前PC值传递给分支预测模块进行下一周期的预测

三：InstDecode

1. 概述

ID 模块是五级流水线的第二个阶段，负责对指令进行译码、读取寄存器文件，并处理分支指令。其主要任务包括：

**指令译码**：解析指令的操作码（opcode）和功能码（funct），生成控制信号（如 aluOp、memWrite 等）。

**寄存器访问**：根据指令中的寄存器编号（rs、rt、rd）读取寄存器文件。

**立即数处理**：对指令中的立即数进行符号扩展或零扩展。

**分支验证**：计算分支目标地址，并验证 BP 模块的预测结果（predTaken）。若预测错误，则产生 pcRedirect 信号，通知 IF 模块修正 PC 值。

ID 模块不仅负责译码，还承担了分支预测的验证和纠正功能，确保流水线的正确执行。

2. 接口信号

输入信号

|  |  |  |
| --- | --- | --- |
| 信号名称 | 位宽 | 描述 |
| clk | 1 | 时钟信号 |
| reset | 1 | 异步复位信号 |
| stall | 1 | 流水线暂停信号 |
| instruction | 32 | 来自IF阶段的指令 |
| pc | 32 | 来自IF阶段的PC值 |
| pcPlus4 | 32 | 来自IF阶段的PC+4值 |
| predTaken | 1 | 来自IF阶段的预测跳转信号 |
| regWriteData | 32 | 来自WB阶段的寄存器写回数据 |
| regWriteAddr | 5 | 来自WB阶段的寄存器写地址 |
| regWriteEn | 1 | 来自WB阶段的寄存器写使能 |

输出信号

|  |  |  |
| --- | --- | --- |
| 信号名称 | 位宽 | 描述 |
| branchlnfo | - | 分支信息结构体，包含以下字段： target: 分支目标地址 branch: 是否为分支指令 jump: 是否为跳转指令 pc: 当前指令PC值 |
| readData1 | 32 | 从寄存器文件读取的数据1 |
| readData2 | 32 | 从寄存器文件读取的数据2 |
| imm | 32 | 符号扩展后的立即数 |
| aluOp | 4 | ALU操作码 |
| regDst | 1 | 寄存器目标选择信号 |
| memRead | 1 | 存储器读使能 |
| memWrite | 1 | 存储器写使能 |
| regWrite | 1 | 寄存器写使能 |
| memToReg | 1 | 存储器到寄存器选择信号 |
| branchTaken | 1 | 分支实际执行结果 |
| branchTarget | 32 | 分支实际目标地址 |
| pcRedirect | 1 | PC重定向信号（当分支预测错误时置1） |

3. 功能描述

指令译码：解析指令的操作码和功能码，生成控制信号

寄存器读取：根据指令中的寄存器地址读取寄存器文件

立即数处理：对指令中的立即数进行符号扩展或零扩展

分支处理：

计算分支目标地址

根据寄存器值和分支类型判断分支是否跳转

比较实际分支结果与预测结果（predTaken）

当预测错误时，产生pcRedirect信号并输出正确的目标地址

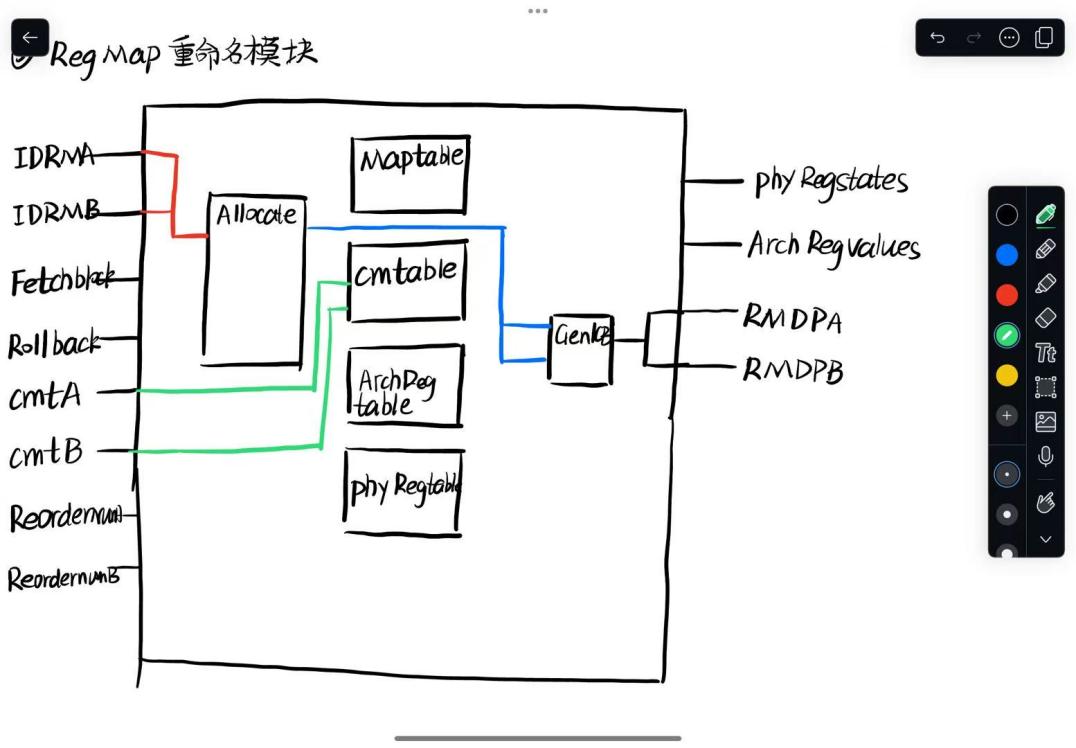
转发控制：检测数据冒险并生成转发控制信号（可在单独的转发单元中实现）

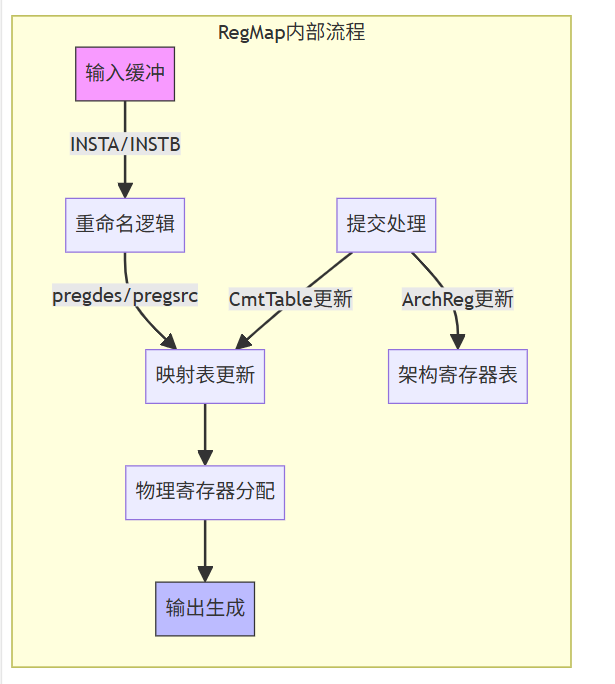
流水线寄存器：将译码后的信号传递到EX阶段

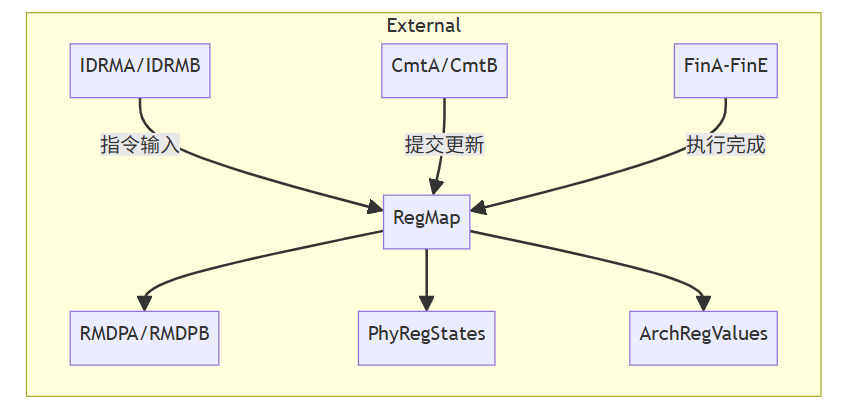
这两个模块与分支预测模块协同工作，IF模块使用预测结果进行指令预取，ID模块验证预测结果并在预测错误时纠正流水线状态。

四：寄存器重命名模块（RegMap）

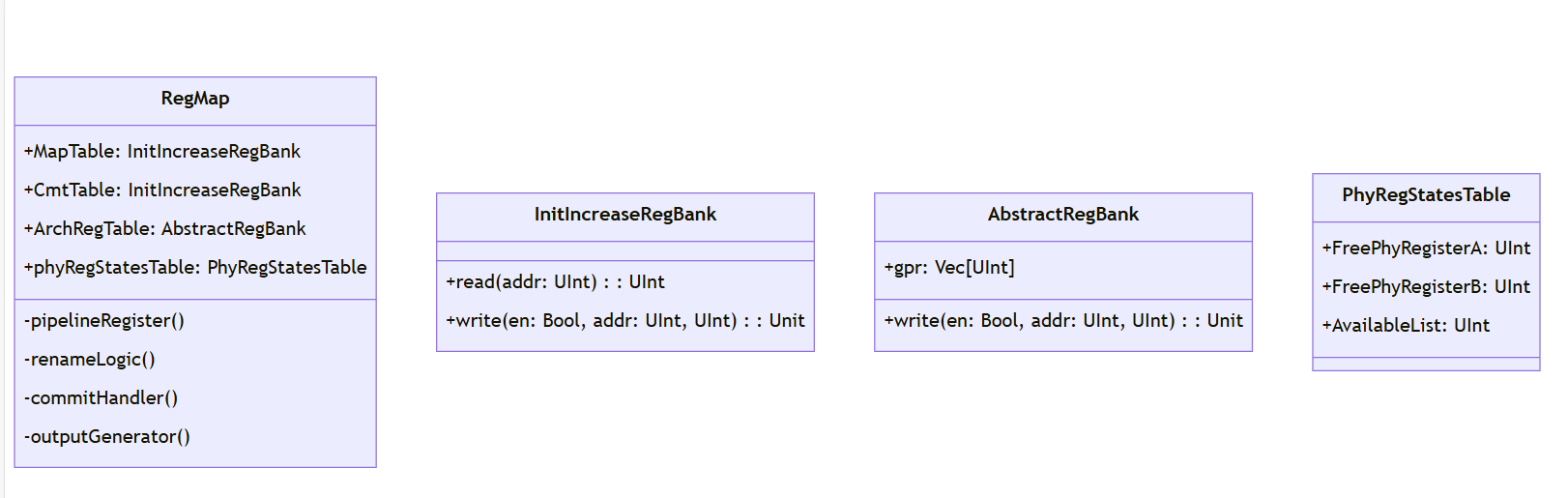
1:流程图







子模块



2：端口信息

|  |  |  |  |
| --- | --- | --- | --- |
| **端口名** | **类型&位宽** | **来源模块** | **描述** |
| IDRMA | InstCtrlBlock | 解码 | 来自解码阶段的A指令的数据块（包括32riscv指令 |
| IDRMB | InstCtrlBlock | 解码 | 来自解码阶段的B指令的数据块 |
| Fetchblock | bool | 分发 | 阻塞信号，暂停新指令进入重命名阶段 |
| Rollback | bool | 异常处理单元 | 回滚信号，清空流水线并且将寄存器映射表对应关系恢复到最近的一次存档点 |
| CmtA | InstCtrlBlock | 提交 | 第一条已提交指令，用于释放物理寄存器和更新寄存器值和相应的寄存器状态 |
| CmtB | InstCtrlBlock | 提交 | 第二条已提交指令，用于释放物理寄存器和更新寄存器值和相应的寄存器状态 |
| reordernumA | UInt（6.w） | 执行 | 提交指令A在全局的索引号（年龄） |
| reordernumB | UInt（6.w） | ROB | 提交指令B在全局的索引号（年龄） |
| FinA - FinE | InstCtrlBlock\*5 | ROB | 五个执行单元的指令，用于更新物理寄存器的就绪状态 |

|  |  |  |  |
| --- | --- | --- | --- |
| **端口名** | **类型&位宽** | **目标模块** | **描述** |
| RMDPA | InstCtrlBlock | 分发 | 重命名后第一条指令的数据块 |
| RMDPB | InstCtrlBlock | 分发 | 重命名后第二条指令的数据块 |
| phyregstates | UInt（128.w) | 分发 | 物理寄存器的状态表（1代表就绪，0代表未就绪） |
| archregvalues | vec(32,UInt（128.w)) | 分发 | 32个架构寄存器存储的值 |

3：核心步骤&功能

（1）重命名映射表建立

首先初始化映射表

val MapTable = new InitIncreaseRegBank(32, 7)//记录的是寄存器之间的对应关系，（如x1->p5）

从而消除WAW和WAR

然后每次分配完寄存器之后要更新映射表的存档关系

1. 物理寄存器的动态分配和释放

这里面要用到一个从空闲列表里面返回最低索引的物理寄存器和次低索引的物理寄存器

于是我们首先定义个FreePhyRegisterA和FreePhyRegisterB函数用于返回索引。

下一步便可以用到分配逻辑

val pregdesA = Mux(INSTA.regdes === 0.U, 0.U, phyRegStatesTable.FreePhyRegisterA)

val pregdesB = Mux(INSTB.regdes === 0.U, 0.U, phyRegStatesTable.FreePhyRegisterB)

这里的逻辑是如果目标寄存器是x0（即满足INSTX.regdes === 0.U），那么便直接返回0，否则我们调用（这个查找函数在后面有定义）

分配完之后便需要更新上面的重命名映射表

1. 提交阶段处理
2. 回滚存档点处理

实际上是类似于游戏存档一样的效果，而这个存档我们便记录在的是cmtTable里面，这里面一定是我们已经提交上的所有安全的映射表

1）：MapTable.table(i) := Mux(i.U === io.CmtA.regdes, io.CmtA.pregdes, CmtTable.table(i))

让所有的寄存器从当前提交备份表里面恢复原始的映射关系

2）：phyRegStatesTable.Rollback(io.CmtA.cmtdes, io.CmtA.pregdes)

关于具体的rollback方面后面会有定义，具体的操作是

1. 当分支预测错误或者异常后，我们需要将未提交指令分配的寄存器状态回滚
2. 但是需要保留跳转的寄存器写入

确保已经提交指令的寄存器状态不被破坏

def Rollback(free: UInt, assign: UInt): Unit = {

for(i <- 1 to 127){ // 遍历物理寄存器1-127（跳过x0对应的P0）

when(i.U === free){

PhyRegStates(i) := 0.U // 情况1：释放旧寄存器

}.elsewhen(i.U === assign){

PhyRegStates(i) := 3.U // 情况2：保留新寄存器

}.otherwise{

PhyRegStates(i) := Mux(PhyRegStates(i) === 3.U, 3.U, 0.U) // 情况3：其他寄存器处理

}

}

}这便完成了寄存器状态的回滚

（5）状态更新和信息导出

1）：这里涉及到了提交相应的指令之后需要把完成的指令信息储存在相应的架构里面方便后续的存档读取。

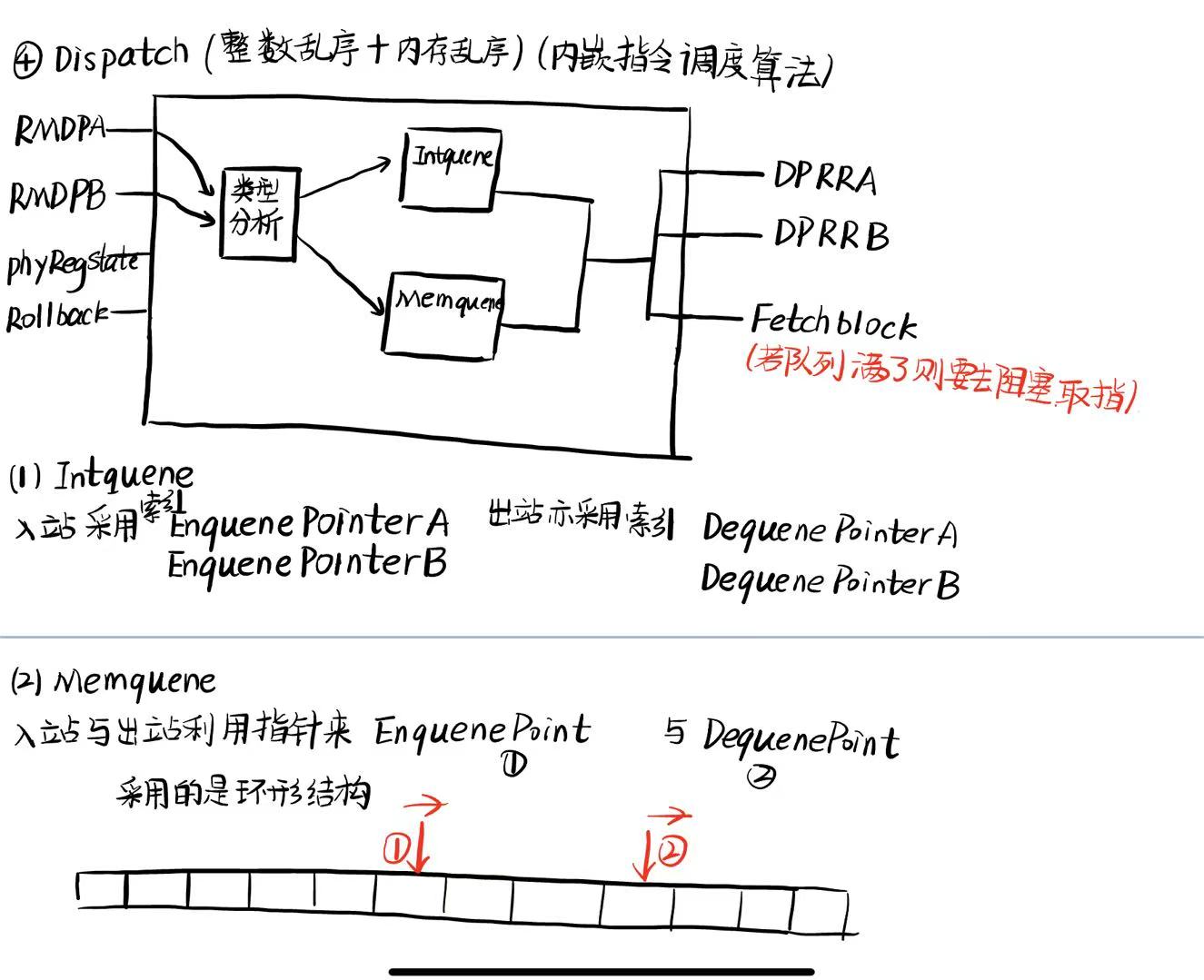
2）：更新提交表cmtTable，记录映射关系

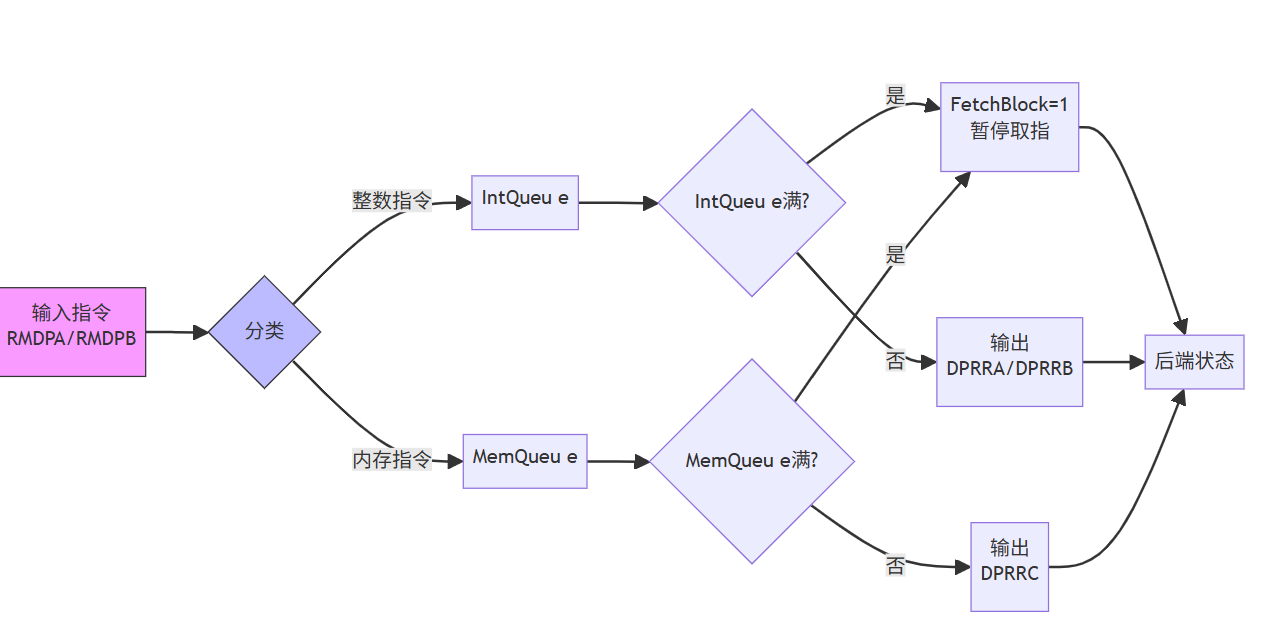
3）：释放旧的物理寄存器，并且更新相应的物理寄存器的状态

数据体的基本信息

五：分发（dispitch）

1：流程图





2：端口定义

1）外包整体模块端口

|  |  |  |  |
| --- | --- | --- | --- |
| **端口名** | **类型&位宽** | **来源模块** | **描述** |
| RMDPA | InstCtrlBlock | 重命名 | 重命名后第一条指令的数据块 |
| RMDPB | InstCtrlBlock | 重命名 | 重命名后第二条指令的数据块 |
| PhyRegStates | UInt（128.w) | 重命名 | 物理寄存器的状态表（1代表就绪，0代表未就绪） |
| Rollback | bool() | 异常处理单元 | 用于流水线冲刷的信号 |

|  |  |  |  |
| --- | --- | --- | --- |
| **端口名** | **类型&位宽** | **目标模块** | **描述** |
| DPRRA | InstCtrlBlock | regread | 分发到执行的指令A数据块 |
| DPRRB | InstCtrlBlock | regread | 分发到执行的指令B数据块 |
| DPRRC | InstCtrlBlock | regread | 分发到执行的指令C数据块 |
| Fetchblock | boll（） | 取指 译码 重命名 | 流水线阻塞信号，当整数块或者访存块队列满了旧要去阻塞 |

2）：访存quene模块

| **信号名称** | **类型/位宽** | **功能描述** |
| --- | --- | --- |
| EnQueueA | InstCtrlBlock | 第1条待入队的访存指令 |
| EnQueueB | InstCtrlBlock | 第2条待入队的访存指令 |
| PhyRegStates | UInt(128.W) | 物理寄存器就绪状态位图，bit[i]=1表示寄存器i数据就绪 |
| Rollback | Bool() | 高电平有效，触发时清空队列并恢复检查点 |

| **信号名称** | **类型/位宽** | **功能描述** |
| --- | --- | --- |
| DeQueue | InstCtrlBlock | 出队的访存指令,顺序执行 |
| QueueFull | Bool() | 高电平表示队列已满，停止接收新指令 |

3）：整形quene模块

| **信号名称** | **类型/位宽** | **功能描述** |
| --- | --- | --- |
| EnQueueA | InstCtrlBlock | 第1条待入队的访存指令 |
| EnQueueB | InstCtrlBlock | 第2条待入队的访存指令 |
| PhyRegStates | UInt(128.W) | 物理寄存器就绪状态位图，bit[i]=1表示寄存器i数据就绪 |
| Rollback | Bool() | 高电平有效，触发时清空队列并恢复检查点 |

| **信号名称** | **类型/位宽** | **功能描述** |
| --- | --- | --- |
| DeQueueA | InstCtrlBlock | 出队的访存指令，乱序执行 |
| DeQueueB | InstCtrlBlock | 出队的访存指令，乱序执行 |
| QueueFull | Bool() | 高电平表示队列已满，停止接收新指令 |

3：核心主要功能

1. ：完成指令类型的分类

首先是用于判断是否是整形指令类型还是访存类型然后利用多路选择器来进行数据块的赋值

1. ：完成分类之后我们要把相应的指令丢入寄存器队列里面

这里面涉及到空闲槽位的管理（1代表空闲，0代表占用，最后生成的是一长串01码）

val FreeList = GenFreeList() // 生成空闲槽位掩码（1=空闲，0=占用）

val NewFreeList = FreeList - lowbit(FreeList) // 移除最低位空闲槽位

val EnQueuePointerA = Log2(lowbit(FreeList)) // 最低空闲槽位索引

val EnQueuePointerB = Log2(lowbit(NewFreeList)) // 次低空闲槽位索引

io.QueueFull := EnQueuePointerA === 0.U || EnQueuePointerB === 0.U // 队列满条件

EnQueuePointerA便是用于指向当前最低空闲的槽位

EnQueuePointerB：指向次低空闲槽位（用于双写入）。

GenFreeList()：一个自定义的方法1代表空闲，0代表占用，最后生成的是一长串01码

管理好了空闲槽位的状态判别，下一步要判别指令的状态就绪

val ReadyList = GenReadyList() // 生成就绪指令掩码（1=就绪）

val NewReadyList = ReadyList - lowbit(ReadyList)

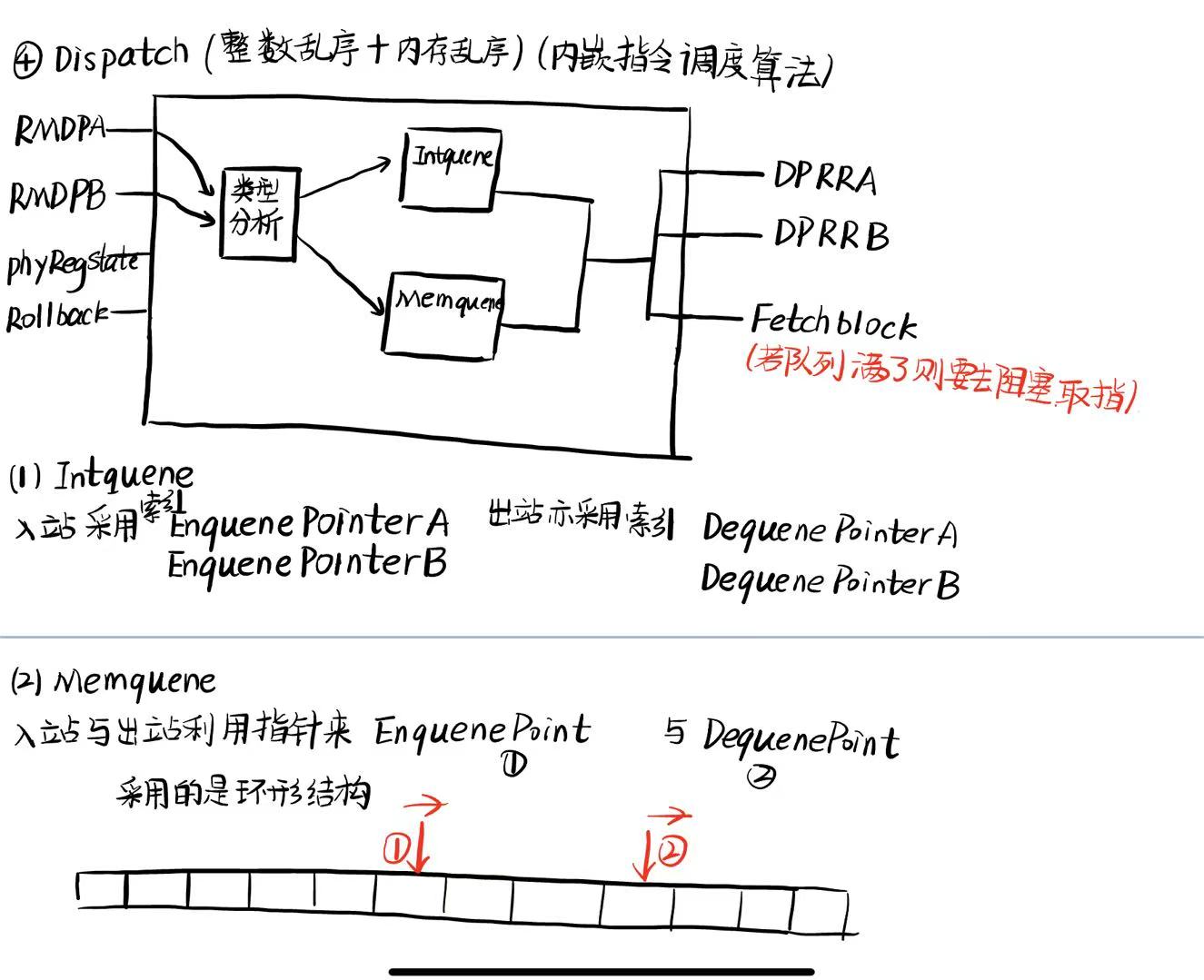
val DeQueuePointerA = Log2(lowbit(ReadyList)) // 最低就绪指令索引

val DeQueuePointerB = Log2(lowbit(NewReadyList)) // 次低就绪指令索引

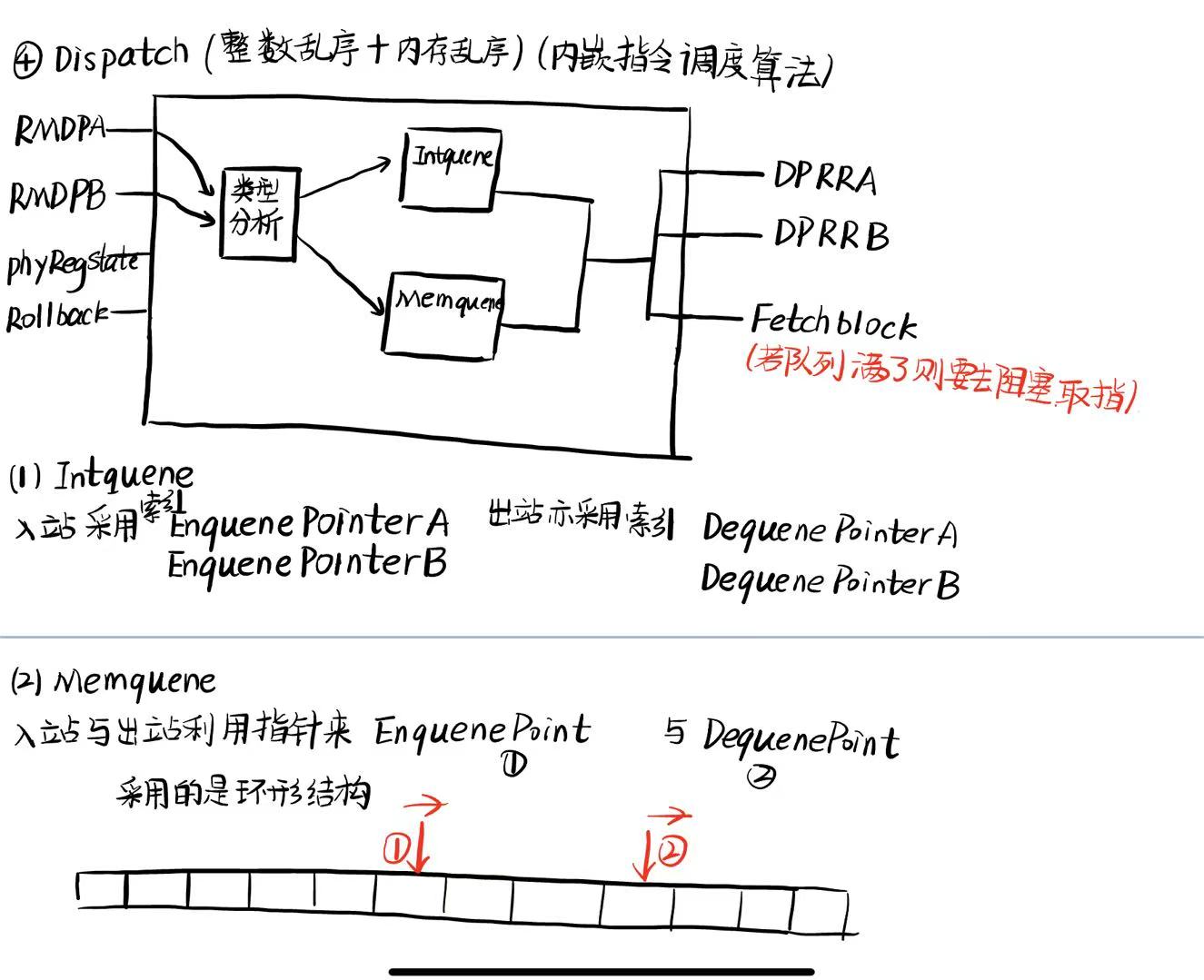
GenReadyList()：一个自定义的方法1代表空闲，0代表占用，最后生成的是一长串01码

1. ：寄存器队列内部要规范出列逻辑

整形模块乱序执行出队暂时不考虑年龄的话，不会对正确性有影响，但是会对性能有影响。



而访存模块顺序执行，我们便采用环形指针逻辑



1. ：回滚逻辑

直接清除掉所有的未提交的指令

for(i <- 0 to 15) {

Bank(i) := WireInit(0.U.asTypeOf(new InstCtrlBlock()))

}遍历掉16个槽位，将16个槽位指令重置为无效的状态

io.DeQueueA := WireInit(0.U.asTypeOf(new InstCtrlBlock()))

io.DeQueueB := WireInit(0.U.asTypeOf(new InstCtrlBlock()))

并且清空掉所有的输出端口，防止作废的指令影响后续的执行阶段

六：regread

1：流程图

2：端口信息

### 端口及其功能

|  |  |  |  |
| --- | --- | --- | --- |
| **端口名** | **类型&位宽** | **来源模块** | **描述** |
| DPRRA | InstCtrlBlock | Dispatch | 指令A的解码后信息，包含操作码、寄存器索引、立即数等控制信息 |
| DPRRB | InstCtrlBlock | Dispatch | 指令B的解码后信息，包含操作码、寄存器索引、立即数等控制信息 |
| DPRRC | InstCtrlBlock | Dispatch | 指令C的解码后信息，包含操作码、寄存器索引、立即数等控制信息 |
| FinC | InstCtrlBlock | Execute | 外部完成端口C的写回信息，包含目标寄存器索引和写回数据 |
| FinD | InstCtrlBlock | Execute | 外部完成端口D的写回信息，包含目标寄存器索引和写回数据 |
| FinE | InstCtrlBlock | Memory | 外部完成端口E的写回信息，包含目标寄存器索引和写回数据 |
| Rollback | Bool(1-bit) | Commit | 流水线回滚信号，为真时清空所有输出端口 |

|  |  |  |  |
| --- | --- | --- | --- |
| **端口名** | **类型&位宽** | **去向模块** | **描述** |
| RREXA | InstCtrlBlock | Execute | 指令A的执行控制信息，包含操作数、操作码和寄存器信息 |
| RREXB | InstCtrlBlock | Execute | 指令B的执行控制信息，包含操作数、操作码和寄存器信息 |
| RREXC | InstCtrlBlock | Execute | 指令C的执行控制信息，包含操作数、操作码和寄存器信息 |
| FinA | InstCtrlBlock | Commit RegMap | 指令A的完成信息，包含写回数据、目标寄存器和完成标志 |
| FinB | InstCtrlBlock | Commit RegMap | 指令B的完成信息，包含写回数据、目标寄存器和完成标志 |

## 3：主要功能

1）：处理三条并行指令的寄存器读取、写回操作。

2）：处理前两条指令的分支跳转操作，相关信息从FinA、FinB输出到其他模块。

七：excute

**Execute：**

**1）：输入端口：**

|  |  |  |
| --- | --- | --- |
| 名称 | 数据类型 | 描述 |
| PREXA | InsCtrlBlock | 第一条指令输入 |
| PREXB | InsCtrlBlock | 第二条指令输入 |
| PREXC | InsCtrlBlock | 第三条指令输入 |
| RollBack | Bool | 流水线回滚信号 |

**输出端口：**

|  |  |  |
| --- | --- | --- |
| 名称 | 数据类型 | 描述 |
| EXMEM | InsCtrlBlock | 输出到内存阶段 |
| FinC | InsCtrlBlock | 完成的指令C |
| FinD | InsCtrlBlock | 完成的指令D |

**2）：内部单元：**

1. **算数逻辑单元ALU（ArithmeticLogicalUnit）:**

需要两个ALU实例，处理并行的计算指令。

**输入端口：**

|  |  |  |
| --- | --- | --- |
| 名称 | 数据类型 | 描述 |
| isa | ISA | 指令类型译码信息 |
| pc | Unit(32.W) | 程序计数器 |
| src1 | Unit(32.W) | 源操作数1 |
| src2 | Unit(32.W) | 源操作数2 |
| imm | IMM | 立即数 |

**输出端口：**

|  |  |  |
| --- | --- | --- |
| 名称 | 数据类型 | 描述 |
| result | Unit(64.W) | 计算结果 |

**运算逻辑：**

算术运算：ADD/ADDI/SUB/LUI 等

逻辑运算：AND/OR/XOR 及其立即数版本

比较运算：SLT/SLTU 及其立即数版本

移位运算：SLL/SRL/SRA 及其立即数和字版本

链接和地址运算：JAL/JALR/AUIPC

1. **地址生成单元AGU（AddressGenerationUnit）：**

专门处理内存访问指令；为加载指令生成访问地址；为存储指令生成地址、数据和掩码；支持多种数据宽度。

**输入端口：**

|  |  |  |
| --- | --- | --- |
| 名称 | 数据类型 | 描述 |
| isa | ISA | 指令类型译码信息 |
| pc | Unit(32.W) | 程序计数器 |
| src1 | Unit(32.W) | 源操作数1 |
| src2 | Unit(32.W) | 源操作数2 |
| imm | IMM | 立即数 |

**输出端口：**

|  |  |  |
| --- | --- | --- |
| 名称 | 数据类型 | 描述 |
| load | LoadIssue | 加载指令处理 |
| store | StoreIssue | 存储指令处理 |

**3）：执行逻辑：**

1. 接收三条指令：RREXA、RREXB、RREXC

2. ALU1 处理第一条指令 (INSTA)

ALU2 处理第二条指令 (INSTB)

AGU1 处理第一条指令 (INSTC，主要为内存访问指令)

AGU2 处理第二条指令 (INSTC，主要为内存访问指令)

3. 根据回滚信号决定输出：

若 Rollback=true，所有输出清零; 否则输出正常计算结果

八：memory

**1：流程图**

## 2：端口及其功能

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 类型&位宽 | 来源模块 | 描述 |
| EXMEM | InstCtrlBlock | 执行  excute | 来自执行阶段的完整指令控制信息 |
| CmtA | InstCtrlBlock | 提交  commmit | 已提交指令的控制信息(主要用于存储操作) |
| ForwardStore | StoreIssue | 提交  commmit | 来自ROB的存储数据转发信息(用于解决load-store冲突) |
| Rollback | Bool | 异常处理单元 | 流水线回滚信号，用于分支预测错误的复位 |
| DataRam | RAMHelperO | 控制信号 | 数据RAM接口 |

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 类型&位宽 | 目标模块 | 描述 |
| FinE | InstCtrlBlock | 提交commit  重命名regmap | 送往执行完成阶段的指令控制信息 |
| ForwardLoad | LoadIssue | 提交commmit | 加载转发输出 |
| DataRam | RAMHelperO | 控制块 | 数据RAM接口 |

## 3：主要功能

1：加载处理指令

从数据RAM读取数据，即相当于从内存里面读取相应的数据

为了解决处理和未提交指令的转发问题

我们采取从转发通道里面获取未提交的存储指令的数据和掩码

2：处理存储指令

在提交阶段将数据写入RAM

3：处理异常分支情况

传递指令信息到下一阶段

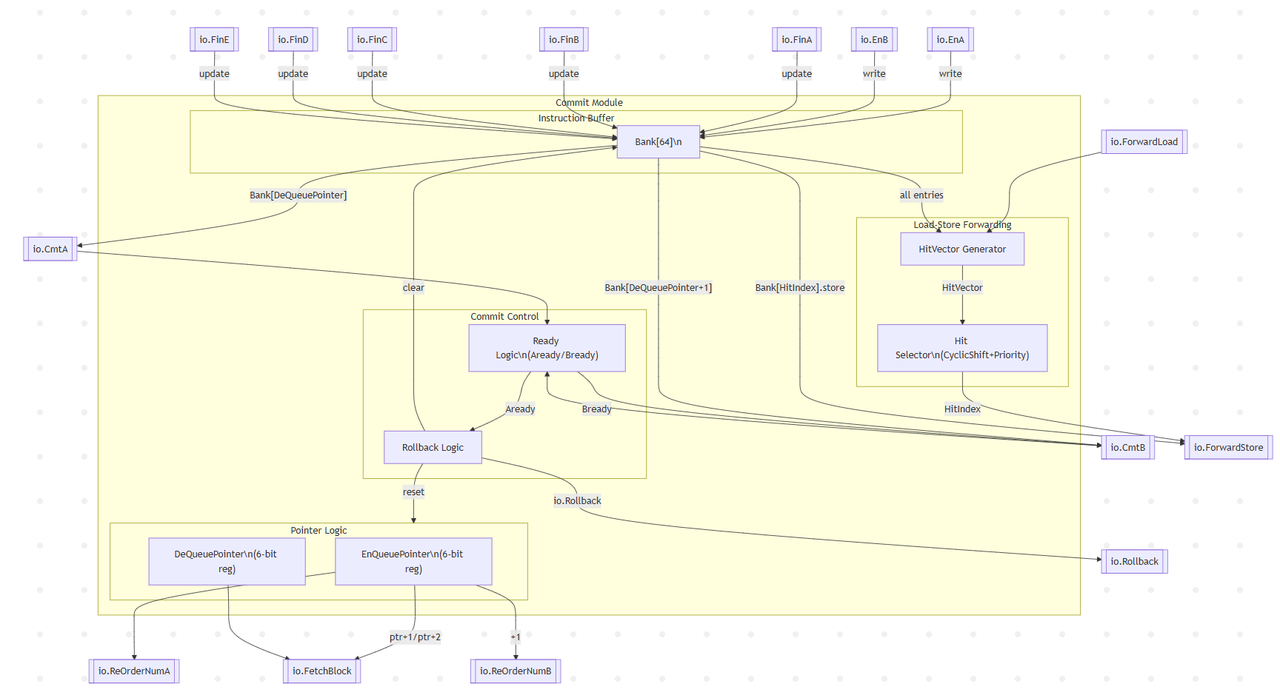
当出现了分支预测错误需要，会根据rollback信号，处理回滚情况

4：提供加载/存储转发通道

相当于提供了通路从而让数据能够前馈完成

九：commit

**1：流程图**



## 2：端口及其功能：

|  |  |  |  |
| --- | --- | --- | --- |
| **端口名** | **类型&位宽** | **来源模块** | **描述** |
| EnA | InstCtrlBlock | Dispatch | 第一条待进入重排序缓冲区(ROB)的指令，包含指令控制信息和执行结果预测 |
| EnB | InstCtrlBlock | Dispatch | 第二条待进入ROB的指令（双发射情况） |
| ForwardLoad | LoadIssue | Memory | 需要数据前递的加载请求，包含： • Valid: 请求有效 • addr: 加载地址 |
| FinA - FinE | InstCtrlBlock\*5 | RegRead  Execute  Memory | 五个执行单元的指令，用于更新物理寄存器的就绪状态 |

|  |  |  |  |
| --- | --- | --- | --- |
| **端口名** | **类型&位宽** | **去向模块** | **描述** |
| CmtA | InstCtrlBlock | InstFetch  RegMap  Memory | 第一条已提交指令，包含： 有效位和完成标志 指令内容和PC值 物理寄存器编号和回写数据 |
| CmtB | InstCtrlBlock | RegMap | 第二条提交指令 |
| ReOrderNumA | UInt(6.W) | RegMap | 分配给第一条新指令的ROB编号（0-63） |
| ReOrderNumB | UInt(6.W) | RegMap | 分配给第二条新指令的ROB编号（连续分配） |
| ForwardStore | StoreIssue | Memory | 存储前递数据，包含： • Valid: 前递有效 • addr/data: 存储地址和数据 |
| FetchBlock | Bool | InstFetch InstDecode RegMap | 阻塞信号（当ROB剩余空间≤2时拉高） |
| Rollback | Bool | 全流水线 | 流水线回滚信号（分支预测失败时触发） |

3：**主要功能**

1）：主要作为ROB（reorder buffer），确保指令以正确的顺序被提交。用一个环形队列来进行指令的缓存，每个周期根据执行阶段的指令信息更新缓存中的指令序列，完成新指令的入队和队首指令的提交

2）：阻塞控制，当缓存中的指令数目小于等于1时阻塞取值，防止缓存中未提交的指令信息被覆盖

3）：异常处理，当前端分支预测出现错误时，产生rollback信号，对全流水线进行恢复

4）：实现存储转发，解决加载-存储依赖

存储转发的主要目的是：当一条加载指令（Load）要读取的内存地址与队列中尚未提交的存储指令（Store）的目标地址相同时，直接从存储指令中获取数据，而不是从内存中读取。这样可以避免加载指令等待存储指令提交到内存，提高性能。