南京大学

I4-NOOP-MIPS 结构说明

MIPS32r1 规范

陈嘉鹏

2017-11-1

目录

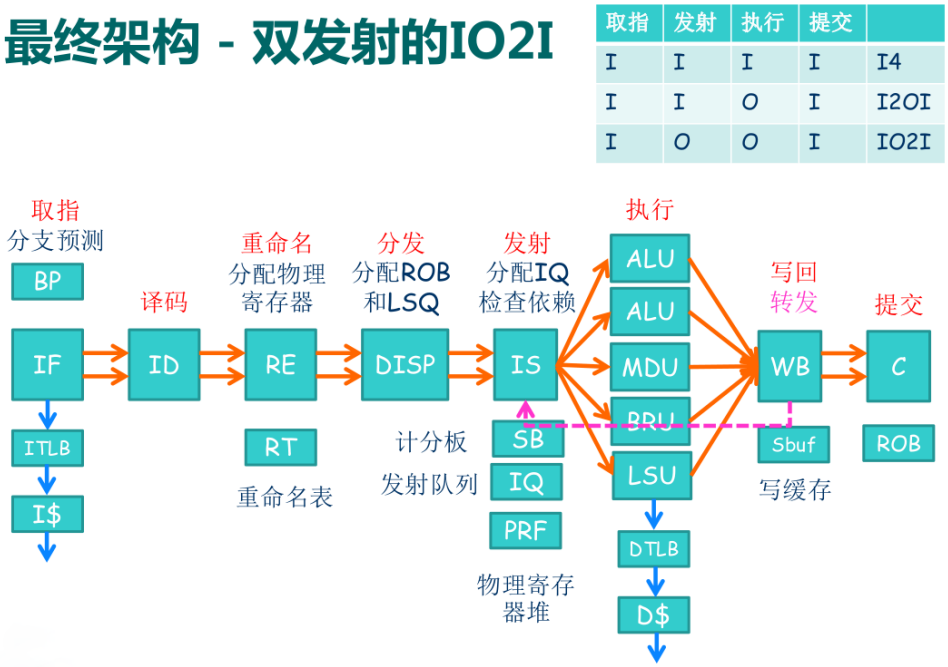
1、背景

NOOP(Nju Out-of-Order Processor)是以乱序二发射处理器为目标而设计的，我们曾经在计算机组成与结构的实验课实现过一个流水线处理器，但是鉴于当时经验不足，缺乏设计，导致实际实现的时候很混乱，扩展性不好。于是我们打算重新设计，在指导老师的的帮助下，NOOP 由此诞生。但是乱序二发射并不是一个简单的目标，具体实践过程中会遇到很多难题，于是我们决定制定一些阶段性的目标，一步一步达到目标。我们先做一个单周期，然后扩展成流水线，最后再考虑乱序。当然前面的实现需要考虑后面的扩展，这个分解的过程在后文会讲解。目前此 CPU 还处于流水线阶段。

2、NOOP-Core 设计

2.1、NOOP-core 框架设计

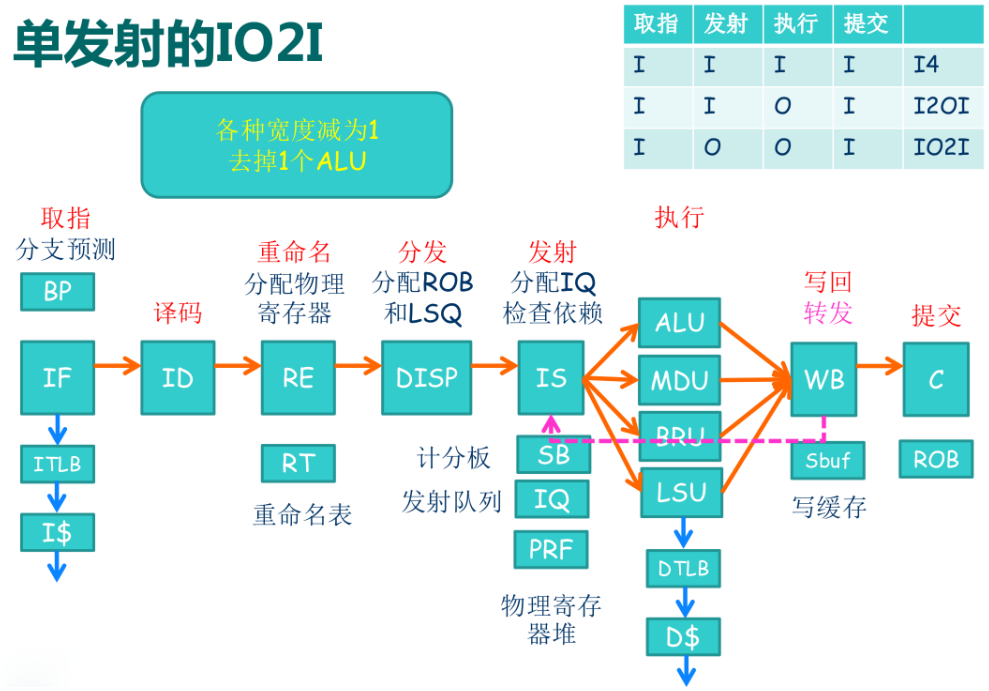
NOOP 的最终目标是实现一个乱序二发射处理器，借鉴一些经典的设计思路，我们得到了这样一个框架结构。



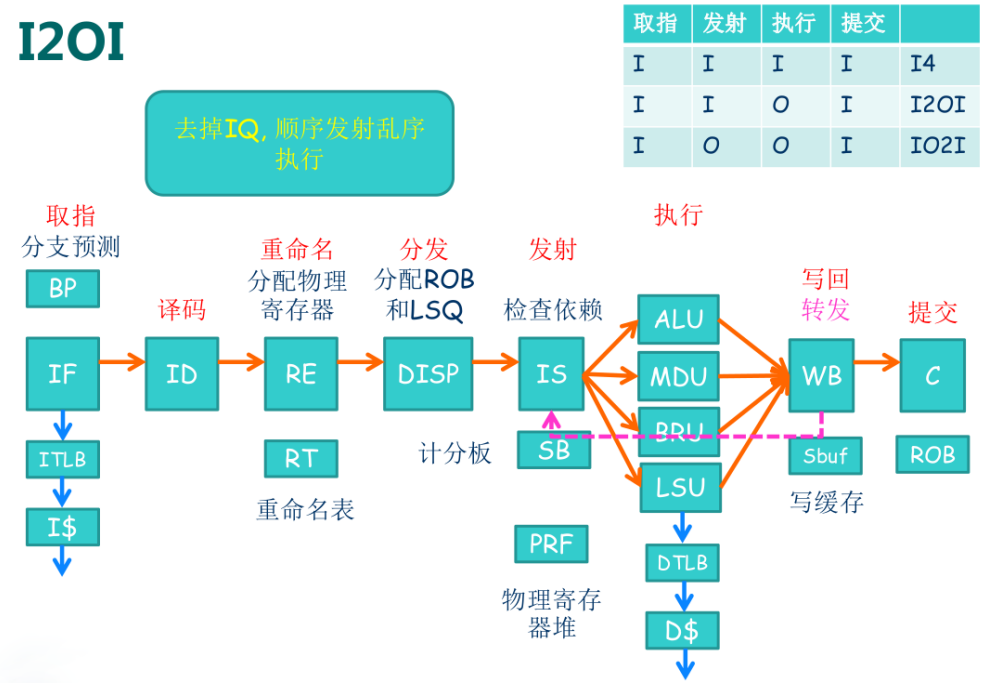
取指	发射	执行	提交	
I	I	I	I	I4
I	I	O	I	I2OI
I	O	O	I	IO2I

但是，要实现这样一个目标，如果一开始就要去考虑所有模块，实现起来根本无法入手，因此，需要将其逐步分解为一个一个可行的目标，在前一步实现正确的基础上再去考虑下一步的工作。因此，我们制定了一些阶段性的目标。

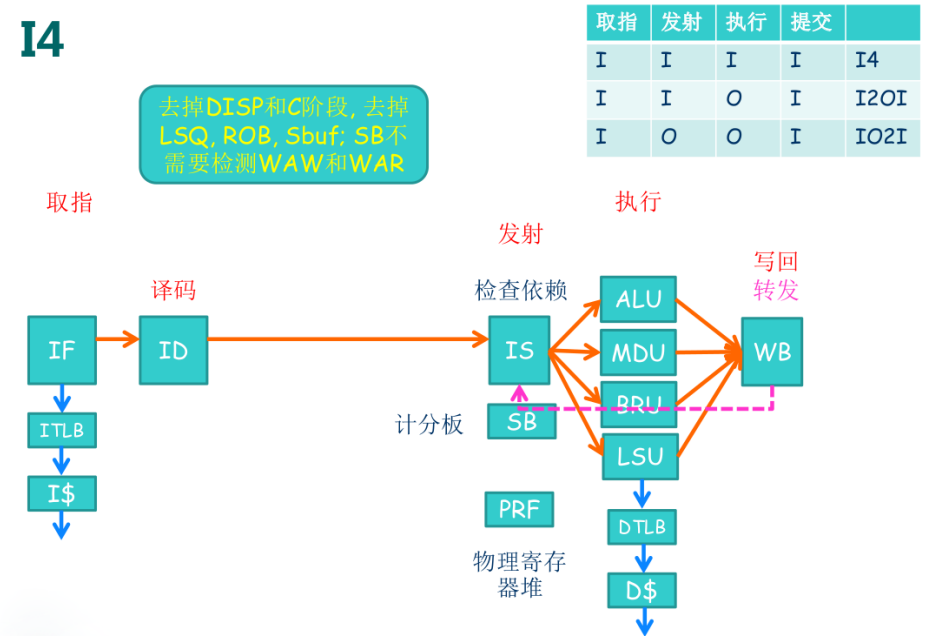
要实现一个双发射的 IO2I，首先要先实现一个单发射的 IO2I，只要将各种宽度减 1 并去掉一个 ALU，于是得到了这样一个框架：



再去掉发射队列，我们得到了一个顺序发射乱序执行的处理器：

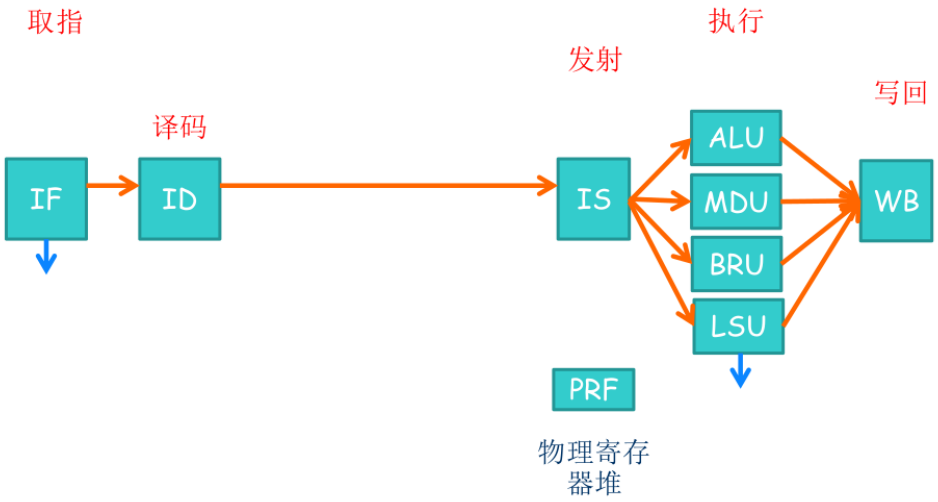


然后再分别去掉重命名机制，分支预测，LSQ/Sbuf，DISP，Commit 阶段，就得到了 I4：



取指	发射	执行	提交	
I	I	I	I	I4
I	I	O	I	I2OI
I	O	O	I	IO2I

再去掉转发，去掉 SB，我们得到了最基本的单周期：



这样，我们得到了一个最基本单周期的框架，我们的第一个目标就是实现这样的一个纯单周期的 MIPS 处理器。然后我们实现的时候逆着这个过程逐步添加一些模块，来达到我们的最终目标。

目前该 CPU 处于 I4 阶段。

2.2、模块功能规范

经典的 MIPS 五段流水线的五个阶段是 IF，ID，EX，MEM，WB，我们的五段流水线为 IF，ID，IS，EX，WB，而 EX 分为 ALU,MDU,BRU,LSU。我们加入的 ISSUE 阶段是考虑到后面的乱序发射的实现，而去掉的 MEM 阶段是因为很多指令不需要访存，我们单独把访存指令集中到一个执行单元 LSU。
各模块的功能规范如下：

- IFU：取指单元，根据当前 PC 通过取指通道拿到指令，并更新 PC
- IDU：译码单元，完成指令译码，生成各种控制信号
- ISU：发射单元，根据译码信号取出对应操作数，并将其分配到对应的执行单元中去
- EXU：执行单元，分为以下四个
  - ALU：算术逻辑运算单元，移位指令也在此实现，指令操作简单，一周期就可以完成
  - MDU：乘除法单元，利用乘除法 IP 核实现，目前乘法无延迟出数，除法 50 周期左右
  - BRU：分支跳转单元，该单元的指令可能会使 PC 发生跳转
  - LSU：访存单元，load/store 类指令执行单元，通过数据通道访问存储设备
- WBU：写回单元，将执行单元的结果写回到寄存器中，异常和中断处理

2.3、模块接口与握手机制

所有功能部件的接口都由自定义信息方式进行定义，格式如下：（定义：信息 = 一个或多个信号组合而成）

```
Unit -> Unit 信息名 #注释
信号名 信号长度       ------+
信号名 信号长度             |
.....  .......             +---->  一个信息
.....  .......             |
信号名 信号长度       ------+
```

将所有信号整个打包成一个信息传递到下一个模块中，下一个模块在这个信息相应的偏移处将信号取出，这样得以在两个不同的模块之间传递信号。具体的使用如下所示：

```
# 这段话定义了一个从IFU传递到IDU的信息，叫instr
# 而这个信息由以下一些阈值（信号）进行组成
IFU -> IDU instr
    PC 32  # propagate PC to BRU to calculate branch target
    instr 32
    is_delayslot 1
    branch_id 6
    ETW 32 # exception type word
```

这些定义的信号通过工程中相应的 python 脚本转化为 parameter 定义，再通过使用 msg_if.vh 中定义的宏将这些信息切割成不同的信号进行使用。而这些信息的传递，则使用了一个简单的握手信号进行处理。即 valid 信号与 ready 信号，当 valid 信号有效时说明本 unit 的信号是有效的，可以正常使用;而 ready 信号有效时说明本 unit 可以接受上一级的信号传输，当 valid 与 ready 同时为 1 的时候即为握手成功。

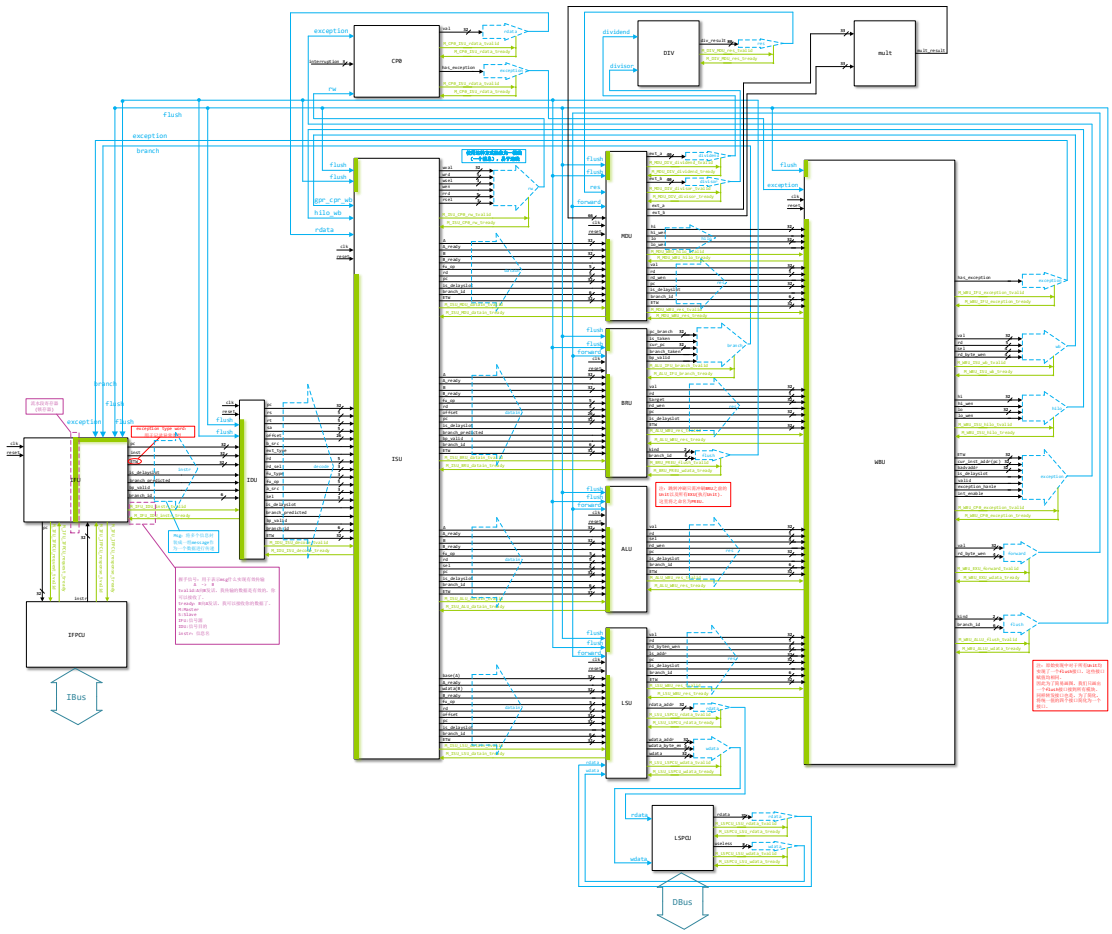
由这个握手信号可以衍生出各种微结构，即从单周期到流水线结构均可使用这个握手进行实现，而实际上，我们也是这样过来的，即从单周期逐步加到流水线，便是逐步微调这些握手信号。

单周期时各消息通道 valid 恒为 1，无视 ready，认为下游模块总能接收到消息。多周期时 IFU 收到 WBU 完成信号再取下一条指令。流水线时 valid 告诉下游模块是否值有效，ready 告诉上游模块是否能接收消息。

单周期如上所述实现了；流水线为了减少延时，当 ready 有效时即接收上游模块数据同时将 valid 一并存储下来，不管是否有效，之后再通过存储下来的 valid 来判断存储下来的数据是否有效，有效则正常赋值，无效则全部赋值为 0，于是流水线通过这样错开的方式完成了握手。

3、NOOP-Core 实现

3.1、具体结构图示意



注：此图可进行放大细看。

对外接口：clk, reset, IBUS(Instruction Bus), DBUS(Data Bus), Interruption。

对外接口图中只进行了简略的列出，未进行详细连线。

缩写说明：

- IFU - Instruction Fetch Unit
- IDU - Instruction Decode Unit
- ISU – Issue Unit
- ALU - Arithmetic and Logic Unit
- BRU – Branch Unit
- MDU – Multiplication and Division Unit
- LSU – Load Store Unit
- WBU – Write Back Unit
- IFPCU - Instruction Fetch Protocol Converter Unit
- LSPCU – Load Store Protocol Converter Unit

3.2、 模块控制逻辑

在我们的定义中，一个模块由五种状态，三种行为组成，两者结合控制着模块的活动。

具体来说，五种状态分别是 nop/flushed/passed/finished/working，这些状态共同实现了一个模块的控制，即 valid 信号与 ready 信号的赋值。状态的含义如下所述：

- nop　　这个模块的数据是无效的(valid 为 0)
- flushed　这个模块的数据被冲刷掉了
- passed　这个模块的数据传递到下一个模块了
- finished　这个模块的工作完成了（始终为 1 cycle 完成的模块不需要此状态）
- working：　这个模块正在工作中（默认状态）

模块的三种行为分别是：接收上一个模块的数据/正常工作/将工作后得到的数据传递给下一个模块。五种状态对应着三种行为的使能，其对应关系如下所述：

- nop　　可以接受上一个模块数据，不正常工作，不向下传递数据
- flushed　可以接受上一个模块数据，不正常工作，不向下传递数据
- passed　可以接受上一个模块数据，不正常工作，不向下传递数据
- finished　可以接受上一个模块数据，不正常工作，可以向下传递数据
- working　不接受上一个模块数据，正常工作，不向下传递数据

这样主要由五种状态控制，产生三种行为，控制着模块的正常活动情况。

3.3、 模块功能逻辑

3.3.1、IFU 模块

取指单元，主要负责三件事：包括分支预测的下地址逻辑、产生供后序阶段模块使用的标签、取指令。

下地址逻辑，我们使用了 64 个 2 位饱和计数器进行分支预测，同时与 BRU 送达的分支地址、异常处理地址、当前 PC+4 进行多路选择。

产生标签，产生供后序判断分支预测结果的信息、标记冲刷信息并向后传递。

取指令，我们在 IFU 中实现了两个自循环 PC 模块，一个专用于取指令，同时下地址逻辑仅作用在该 PC 上；另一个用于与取回的指令一并向下传递，满足与取回的指令是相对应的（而不是加 4 的关系），同时也用于分支预测。这样两个自循环 PC 模块实现了接连不断地取指令的功能。不会因为 BRAM 1 cycle 延迟而导致流水线无法填满。

3.3.2、IDU 模块

译码单元，主要工作是将指令译码成对应的数据和控制信号。译码生成的信号以及其含义如下：

```
fu_type      : 功能单元类型，可以是ALU类型，MDU类型，BRU类型，LSU类型中的一种
fu_op        : 某个功能单元内部操作，该信号和fu_type一起标识一条指令的行为
b_src        : 第二个源操作数类型，是寄存器还是立即数(即R型指令还是I型指令)
ext_type     : 第二个源操作数是立即数的时候它的拓展方式，0拓展还是符号扩展
rd_sel       : 目标操作数的类型，可以是RT,RD,HI(MTHI),LO(MTLO),HILO(MULT,DIV)中的一种
a_src        : 第一个源操作数的类型，可以是RS字段，SA字段，HI寄存器，LO寄存器，CP0寄存器，EPC(eret指令)中的一种
de_special   : 用于标识异常相关指令
mask         : 无效指令检查用到的掩码
value        : 某些指令要求某些字段中必须为特定的值，我们用指令和掩码按位与的结果应该和value相等，不相等就是无效指令
```

考虑到该部分的代码可以用一个查找表的结构来实现，如果手写代码的话，工作量很大，而且容易出错，后期维护也不方便。于是，我们将这些信号用表格的形式组织起来，word 不是文本文件，不利于维护，于是我们用 csv 文件格式（可以用 excel 打开编辑）保存译码结果。并用 python 脚本(gen_decoder.py 文件)将表格翻译成 verilog 代码，这样维护起来更加方便。

3.3.3、ISU 模块

发射单元，取操作数，判断数据依赖关系，控制分发数据和控制信号到对应的执行单元中去的时机。

在非单周期的实现中，发射操作需要考虑数据依赖关系，我们用 scoreboard 来记录指令间的依赖关系，先将所有寄存器文件重定义编号到统一一个空间中，在这个编号空间中如果发现了 RAW 冒险，则需要从后面执行单元或写回单元中转发数据，或是等待后续单元执行完毕并写回寄存器文件中，此时再将对应指令发射到执行单元。

在 I4 阶段我们选择将数据转发到执行单元中，由 scoreboard 判断原始数据是否有效，无效则使用转发过来的数据，同时进行字节写使能的掩码操作。I4 阶段只需以上功能即可。

到之后的阶段中，由于执行阶段可能存在多个正在执行的指令，不知道此次转发真正的目的是哪个指令，此时则需要在 scoreboard 中加入依赖指令编号的记录，并在转发时将指令编号也转发过去，在原来的基础上，再判断依赖指令编号与转发的指令编号是否相同，这样可以实现更通用的转发功能，即类似 Common Data Bus 的实现。

3.3.4、ALU 模块

算术逻辑运算单元，在此单元中执行的指令包括常用的算术逻辑指令(加减法，按位运算等)，移位指令，MOVN/MOVZ，MFC0/MTC0。这些指令操作简单，都能在一周期内完成。

3.3.5、MDU 模块

乘除法单元，乘除法运算我们采用了 Xillix IP 核用于实现，配置上，乘法无延迟出数，而除法大概需要 50 周期。

3.3.6、BRU 模块

分支跳转指令单元，branch 类指令和 j 类指令执行单元，得到分支跳转目标以及是否跳转的判断，这样可以知道分支预测是否正确。相应处理有：分支预测训练、分支预测错误弥补与冲刷信息分发。

若分支预测正确，则将信息发送到 IFU 模块，进行分支预测器的训练。

若分支预测错误，则在训练的同时，需要冲刷预测错误的指令，这些指令不能执行，因此在 BRU 模块向前面的模块以及执行单元发送冲刷的命令，将无关指令刷掉，弥补掉分支预测错误的不正确执行状况。

3.3.6、LSU 模块

load/store 指令单元，该单元是访存指令执行单元，通过数据通道访问外部资源(内存和设备)，数据通道部分的详细说明在 NOOP-uncore 部分讲解。

由于通用性冲刷的原因，同时为了避免产生关键路径的产生，我们设定冲刷只能在下一周期才能真正到达模块中，于是导致了 load/store 必须等待一周期才开始执行，这样使得目前 load/store 相关测试相对较差。这里是一个需要改进的地方。

3.3.6、WBU 模块

写回单元，各个执行单元的结果都会集中到该单元，因此在该单元对一系列事务进行统一的处理，分别有：寄存器写回、中断异常处理、冲刷信息分发。

寄存器写回，在没有产生异常中断以及冲刷的情况下进行写回，以修改程序员可见状态。

中断异常处理，前面的 IDU、ALU、LSU、BRU 可能会产生异常，但是我们并没有就地处理异常，而是将发生的异常记录在一个 32 位标签中，并随流水线往下流，直到 WBU 中进行处理，这么做实现了精确异常。

由于存在中断异常，中断异常出现的指令执行是错误，因此之后的指令也不能够进一步执行写回，这样需要对其前面的指令进行一个冲刷处理，所以 WBU 同样可以向前面的模块发送冲刷命令，将流水线中所有指令冲刷干净。

3.3.7、IFPCU 模块

取指协议转换单元，由于外部接口协议种类很多，例如：AXI4、AXI4-Lite、SRAM、自定义 SRAM、AHB、APB 等等，存在需求改变的时候切换接口是很麻烦的，但是要使用别人定义的部件，我们需要使用标准的接口。因此为了更加方便的接入，我们在内部定义了自己的接口，之后使用转换器从自己的接口转换到别人的接口上，这样就进行了一个耦合性更低的衔接。

因此，这个单元接收内部 CPU 的指令请求信号，图上有标出，并通过我们自己定义的转换器，转换到外部统一的接口上。

3.3.8、LSPCU 模块

Load Store 协议转换单元，同 IFPCU 一样，都是将内部接口转换到外部统一接口，原因一致，方式一致。

3.4、 冲刷控制

在已经设计好的 CPU 框架中，为了使得通用性更高，在进行冲刷处理时我们这样考虑。

首先，冲刷存在两种方式，一种是出现异常或者 eret 时流水线中所有指令都得冲刷，我们命名为无条件冲刷。另一种是出现跳转指令时，流水线需要冲刷除去分支跳转对应的延迟槽指令之外的其他指令，我们命名成有条件冲刷。

其次，我们假设取指令周期数不确定，流水线中可能存在多条延迟槽指令，也可能带来冲刷的跳转指令对应的延迟槽指令还未取出，这样冲刷对应需要保留的延迟槽指令在流水线中的位置是不确定的。

综合考虑这两种情况，我们意识到总控式冲刷是不可能完成这个任务的，除非将流水线中所有的指令进行记录，在冲刷时查记录表，但是这种方式控制复杂且容易产生关键路径。因此我们将冲刷实现为分布式控制冲刷方式，即由信息产生部件发出冲刷信息，其余各部件根据冲刷信息自行决定是否进行冲刷。

```
// flush kind
`define  no_flush        2'b00
`define  flush_no_cond   2'b01
`define  flush_cond      2'b10

// 分别代表:
//        no_flush:        忽略该冲刷信息
//        flush_no_cond:   立即进行冲刷
//        flush_cond:      检测条件满足时进行冲刷
```

这样分种类可以满足不同冲刷方式的需求。而对于需要检测条件的冲刷，如延迟槽保留的冲刷方式就需要一些检测条件了，为了满足延迟槽位置不确定的需求，我们加入了两个条件变量：

```
wire      is_delayslot; // 该指令是否为延迟槽指令，标签1
wire[5:0] branch_id;    // 该指令的编号，标签2
```

标签 1 用于条件判断该指令是否为延迟槽指令，否定则进行冲刷；标签 2 用于判定分支指令与延迟槽指令的对应关系，每个延迟槽与相应的分支指令有唯一的编号，这样可以在多个延迟槽指令中判断唯一保留的延迟槽。

更具体的，我们在 IFU 产生上述两个条件变量，并随着流水线向下流。

在 WBU 进行发送冲刷信号的判断，同时从流水线中取出冲刷信息进行发送。

所有的模块将冲刷信号锁存下来（避免直连线判断导致的关键路径），在下一个 cycle 中进行判断是否满足冲刷条件，若满足，则这条指令被冲刷掉，填写 flushed 状态。若不满足则该指令被保留，继续向下执行。

3.5、 中断与异常

3.5.1、检测

首先考虑架构布置，需要实现的异常可以分为两类，即同步与异步，同步指与指令流同步，异步指任意时候都可能发生，不同步于指令流。目前所需实现的异常除中断外全部为同步异常，中断为异步异常。因此同步异常通过在不同模块内检测之后随指令流向下流，在最后阶段进行解决即可。异步异常直接在 cp0 中进行检测，同同步异常一起判优先级，得到优先级最高的异常再继续处理即可。

3.5.2、处理

发生异常之后的处理有三个行为：写 cp0 寄存器，冲刷指令流，写 PC。第一个行为，写 cp0 寄存器，主要由同步异步、延迟槽、异常种类进行区别，具体看来，同步异步方面，由于异步只存在中断，分析来看两者行为是一致的，因此不必细分。只需要以延迟槽与异常种类进行处理区分即可。第二个行为，冲刷指令流，在前面所述中，冲刷种类分为三类，此处选择无条件冲刷即可。第三个行为，写 PC，使用与分支跳转指令相同的数据通路即可。

3.5.3、特殊情况

考虑到 fifo 类访存可能出现异常，同步异常时 load/store 指令使能信号会被关闭不会对外设进行访问；异步异常时，由于数据已经被 load/store，因此当出现 LSU 向后传的数据有效时会关闭异步异常的使能信号，这样取出的数据不会被冲刷掉，同样也不会反复存入。当 load/store 操作与异步异常同时进行时，load/store 指令使能信号同样被关闭。这样达到了避免 fifo 类访存失效的情况发生。

3.6、 CPU 执行流程说明

注：配合图食用更佳。

3.6.1 取指令

IFU 向 IFPCU 送出 pc 进行取指令，IFPCU 将内部协议转换为外部统一的协议，并将请求送入总线中，得到指令，之后将指令 instr，送到 IFU 中，完成了取指令操作。

IFU 根据内部记录以及该指令的内容判断这条指令是否是延迟槽(is_delayslot)，是否进行了分支预测(bp_valid)，分支预测是跳还是不跳(branch_predicted)，处于编号为多少的分支指令之后(branch_id)。连同 pc、instr、ETW(初始化为 0，用于之后的流水段收集异常信息)送入 IDU。

注：is_delayslot、branch_id 用于冲刷时留下延迟槽。
branch_predicted、bp_valid 用于 BRU 判断分支预测正误，并进行分支预测器的训练。

3.6.2、译码

IDU 通过 IFU 送来的信息进行译码操作，得到

- rs/rt/rd
- sa: shift amount
- offset: j 指令 offset 部分、以及包含的 imm 立即数部分
- b_src: 第二个源操作数类型，是寄存器还是立即数(即 R 型指令还是 I 型指令)
- ext_type: 第二个源操作数是立即数的时候它的拓展方式，0 拓展还是符号扩展
- rd_sel: 目标操作数的类型，可以是 RT, RD, HI(MTHI), LO(MTLO), HILO(MULT, DIV)中的一种
- fu_type: 功能单元类型，可以是 ALU 类型，MDU 类型，BRU 类型，LSU 类型中的一种
- fu_op: 某个功能单元内部操作，该信号和 fu_type 一起标识一条指令的行为
- a_src: 第一个源操作数的类型，可以是 RS 字段，SA 字段，HI 寄存器，LO 寄存器，CP0 寄存器，EPC(eret 指令)中的一种
- sel: CP0 寄存器中选择哪一组（CP0 中有很多组寄存器组，每组 32 个寄存器）

于此同时，IDU 由于可能有异常情况，如无效指令、syscall 指令、break 指令，因此判断之后在 ETW 中某个位进行填写最终的结果，0 为没有这个异常，1 为有这个异常。

得到所有这些信号后 IDU 连同之前的信号一起向后传递到 ISU 模块。

3.6.3、读写寄存器与发射

ISU 根据 IDU 译码出来的信息，读取寄存器值，并判断操作数 A 与操作数 B 应该是取什么值，从众多来源中选择该指令需要的操作数。

同时在 scoreboard 中得到 A 与 B 是否依赖于 EXU 或者 WBU 的指令，若依赖（即选择出来的操作数值不是最新的）则 A_ready 或 B_ready 置为 0，待 EXU 中进行转发解决。

由于代码中通用寄存器放置在 ISU 模块内，因此，ISU 模块需要执行 WBU 发送的写回操作(gpr_cpr_wb、hilo_wb)，由此统一程序员可见寄存器读写统一放置于 ISU 模块中。

在此 MIPS CPU 中，可见寄存器有 lo、hi（乘法高位低位、除法商和余数）、通用寄存器、CP0 寄存器。前三个代码放置在 ISU 模块内部，因此不需要向外发送信号，直接读写即可。而 CP0 寄存器，放置于外部，因此需要送出信号：

- wval 写入的值
- wrd 写入的寄存器编号
- wsel 写入的寄存器组编号
- wen 写使能

- ✧ rrd 读寄存器编号
- ✧ rsel 读寄存器组编号

CP0 直接完成写操作，将读数据通过 rdata 返回到 ISU 模块中。

完成以上数据准备后，ISU 模块通过由 IDU 送来的 fu_type 字段，判断该送到哪一个执行单元(ALU/BRU/MDU/LSU)，并将执行所需的信息送到该单元中。这些值在之前都说过，此处不再重复。

3.6.4、执行

3.6.4.1、ALU 执行

通过从 ISU 阶段来的源操作数 ready 信号，可以判断是否需要使用 WBU 送来的转发(forward)信息，以得到准确的源操作数，通过 fu_op 判断执行操作种类，并使用得到的准确操作数进行计算。

得到 val（计算结果）、rd_wen（结果写回使能信号）。

同时在计算中可能出现 Overflow 异常，若出现则在 ETW 对应位中记录，最终在 WBU 阶段进行处理。

这样，将得到的数据与之前传下来的后序需要的信号继续向下传递。

3.6.4.2、BRU 执行

同样，使用 ISU 来的信号，得到最新源操作数，并判断该指令是否会跳转，以及这个跳转结果与 branch_predicted 记录是否一样，不一样则将需要跳转的信息：

- ✧ pc_branch： pc 该跳转到的地址
- ✧ is_taken：是否需要使 pc 修改到 pc_branch，即预测错误情况
- ✧ cur_pc：该跳转指令的地址
- ✧ branch_taken：该分支是否跳转
- ✧ bp_valid：该指令是否使用了分支预测

前两个用于分支预测错误弥补，重新跳回正确的分支上去。后三个用于分支预测器训练。

同时，由于预测错误需要进行冲刷，ISU 向前面的模块以及 EXU 发送冲刷信号。

控制逻辑如上所述。

由于有一部分分支指令会对寄存器进行写入，如写入 31 号寄存器，因此 BRU 依旧向 WBU 发送写回信号，val/rd/rd_wen。

在 BRU 中存在 AdEL 异常，因此也会在 ETW 上写入相应的值，target 用于最后的错误信息记录。

3.6.4.3、MDU 执行

以同样的方式取得最新的源操作数，由于乘法器与除法器放置在模块外，需要向这两个模块递交操作数，因此在进行一定的处理后 ext_a 与 ext_b 送入乘法器模块或者除法器模块，进行相应的计算，最后将 hi/lo 结果与 hilo 写回使能 hi_wen、lo_wen 送入 WBU 进行写回操作。

MDU 同时也从 IDU 的 lo、hi 寄存器中读源操作数，送入通用寄存器中，因此存在相应的通用寄存器的写回通道(val/rd/rd_wen)，送入 WBU。

同样也可以从 IDU 的通用寄存器中读源操作数，送入 hi/lo 中。

输入输出显而易见。

3.6.4.4、LSU 执行

同样的方式得到最新的源操作数，计算出访存所需的条件：

- ✧ rdata_addr load 操作地址
- ✧ wdata_addr store 操作地址
- ✧ wdata_byte_en store 操作字节写使能
- ✧ wdata store 存入存储器的数据

之后通过 fu_op 知道访存的操作类型，对 LSPCU 进行访问，使其从本地读写接口转换为外部统一的接口，进行读写数据。结束后写数据通过 M_LSPCU_LSU_wdata_tvalid 通知 LSU 是否写入成功，读数据通过 rdata 返回给 LSU 模块，进行相应的处理后，将最终的读出的值送到 WBU 进行寄存器的写回。

LSU 也存在异常，ls_addr 用于后序异常的信息记录操作。

3.6.5、写回与处理中断异常

WBU 接收来自各个处理单元的信息，但是由于只是顺序执行，只有一个执行单元给的值是有效的。我们通过由前面单元传来的 M_EXU_WBU_res_tvalid 握手信号来看哪个执行给的值有效。(EXU：ALU/BRU/MDU/LSU)

得到有效值后，我们将所需要写回的值进行写回：

通用寄存器与 CP0：

- ✧ val：所需写回的值
- ✧ rd：所需写回的寄存器编号
- ✧ sel：最高位记录是通用寄存器还是 CP0 寄存器，后三位记录 CP0 寄存器组组号
- ✧ rd_byte_wen：写入寄存器的字节写使能

hilo：

- ✧ hi：hi 寄存器写回的值
- ✧ hi_wen：hi 寄存器写回使能
- ✧ lo：lo 寄存器写回的值
- ✧ lo_wen：lo 寄存器的写回使能

除此之外，我们在每个模块中收集了的异常信息(ETW 以及记录信息如 ls_addr 等)，需要在这里进行异常处理。

首先将 CP0 寄存器记录需要的异常信息(如 ls_addr 记录到 badvaddr 寄存器中，记录 delayslot 等)，同时通知异常中断是否能被处理，接口如下：

- ✧ ETW：之前收集的异常种类，用于判断是什么异常
- ✧ cur_inst_addr：CP0 异常信息的记录
- ✧ badvaddr：CP0 异常信息的记录
- ✧ is_delayslot：CP0 异常信息的记录
- ✧ valid：这些值是否有效
- ✧ exception_handle：这些异常是否能被处理，目前一直为 1
- ✧ int_enable：中断使能信号，如果是 load/store 指令由于可能存在 fifo 部件的原因不能被中断。

通知 pc 跳到异常处理地址，准备开始处理异常，这里直接一个 has_exception 通知即可。

冲刷后来的所有指令，使其不能执行。(目前 kind 有些多余，因为 BRU 只有一种冲刷即 flush_cond，WBU 也只有一种即 flush_no_cond)

WBU 由于可能在流水段寄存器中存在目的寄存器的最新值，因此需要转发到各个执行单元中：

- ✧ val：转发的值
- ✧ rd_byte_wen：字节写使能

4、附录

4.1、Core 中使用到的 Xillix IP 核

- Multiplier v12.0 （pg108-mult-gen）
- Divider Generator v5.1 （pg151-div-gen）

4.2、NOOP-uncore 内容

详见 NOOP-uncore 文档。