# REPEAT

## Introduction

**REPEAT $counter, $insn\_num**

功能：将REPEAT之后的$insn\_num条指令循环执行$counter次

Repeat循环指令中不可以再包含其他分支或者repeat指令，但是repeat的执行可以被中断。

## Algorithm



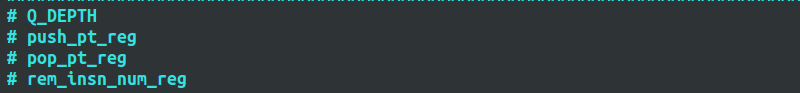
Repeat\_sequnce: REPEAT之后 $insn\_num条指令组成的顺序的指令序列



$counter = 5, $insn\_num=6, repeat\_sequnce = A-F, 所以REPEAT 5, 6的功能就是将REPEAT后的6条指令序列A-F循环执行5遍。

## Implementation

### 基于队列的派遣前提



* 我们的实现前提是所有的指令都是从指令队列里被派遣，指令队列的深度是Q\_DEPTH。指令队列负责根据队列里剩余的指令数目来发起顺序取指请求，并缓存取来的指令。
* Push\_pt\_reg: 队列的写指针
* Pop\_pt\_reg：队列的读指针
* Rem\_insn\_num\_reg：队列内剩余指令数目，就是可供派遣的有效指令数目



### REPEAT指令的派遣条件

* $counter和$insn\_num的值是确定的
* 不存在可能的流水线回退需要重发REPEAT指令

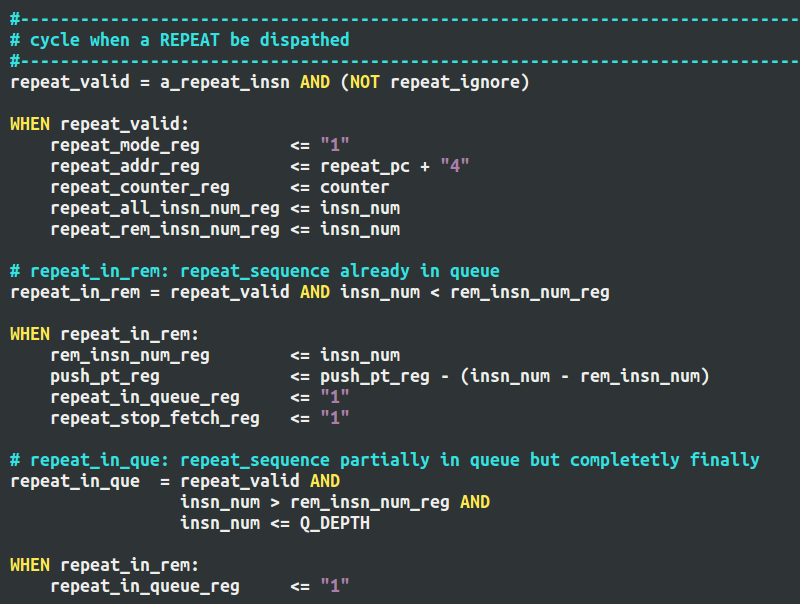
### REPEAT指令派遣逻辑

REPEAT派遣就是解析REPEAT指令本身，来决定不同的REPEAT处理方式，分为以下几种：

* **Repeat\_ignore: REPEAT指令的硬件循环操作可以被忽略**
* $counter = 0: repeat\_sequence执行0轮，即直接跳过repeat\_sequence，所以REPEAT指令的实际是一个jump操作
* $counter = 1: repeat\_sequnence 只执行一轮，那么REPEAT实际是一个nop操作
* $insn\_num = 0: repeat\_sequnce内不含有任何指令，没有需要REPEAT的指令，所以REPEAT实际仍然是一个nop操作
* **Repeat\_valid: 硬件要真正执行REPEAT循环操作**
* Repeat\_in\_rem: repeat\_sequence的全部指令都已经在queue里面了，可以直接切断取指令通路，直到repeat最后一轮
* Repeat\_in\_que: repeat\_sequnce的全部指令可以在queue里面放下，但是当前queue里只放了一部分，还要发送取值请求直到repeat\_sequence的全部指令取来，然后关闭取指通路，直到repeat最后一轮
* Repea\_out\_que：repeat\_sequnce的全部指令超出queue的容量，所以要不停的循环取指



* Repeat\_ignore: REPEAT指令被忽略的情况。



* Repeat\_valid：当REPEAT指令的操作不可以被忽略的时候才是有效的REPEAT操作。此时我们需要硬件记录下来repeat\_sequnce 循环执行所需的必要信息。
  + Repeat\_mode\_Reg: 标志硬件正在处理REPEAT操作，当repeat操作彻底结束，置无效
  + Repeat\_addr\_Reg: 每一轮repeat\_sequnce派完都需要回到repeat\_sequence的第一条指令重新派遣，repeat\_addr\_reg就是repeat\_sequence第一条指令的PC
  + Repeat\_counter\_reg: 标志repeat\_sequnence正在执行轮的counter，初始值为$counter
  + Repeat\_all\_insn\_num\_reg：repeat\_sequence的指令数目，就是$insn\_num的值
  + Repeat\_rem\_insn\_num\_reg：每一轮repeat\_sequnce的执行剩余的指令数目，每发完一轮，Repeat\_counter\_reg --， repeat\_rem\_insn\_num\_reg都要被repeat\_all\_insn\_num\_reg重置。
* Repeat\_in\_rem: REPEAT指令发出的时候，发现repeat\_sequence的所有指令序列都已经在队列里面了。这样repeat\_sequence在循环执行的过程中就不需要再向外发取值请求了，每发完一轮只需要调整pop\_pt\_reg就可以开始新一轮的执行。当然rem\_insn\_num\_reg也可能大于$insn\_num，这个时候还要调整push指针到正确的位置。
  + rem\_insn\_num\_reg: 因为repeat\_sequence的所有指令都已经在queue里面了，所以sequence之外的指令要被切出去，用$insn\_num给rem\_insn\_num\_reg赋值
  + push\_pt\_reg：相应调整push指针
  + repeat\_in\_queue\_reg: 置有效，标志repeat\_sequence的一轮指令发完后，直接回退指针就可以开始新一轮
  + repeat\_stop\_fetch: 置有效，repeat\_sequence的指令已经全部取来，不再需要取指操作
* repeat\_in\_que: repeat\_sequnce可以全部放入queue里，但现在queue里只放了一部分，所以还需要取指，但当一轮指令发完后repeat\_sequence的全部指令肯定已经在queue里了，直接回退指针就可以开始新一轮。
  + repeat\_in\_queue\_reg：置有效，等repeat\_sequnce的指令全部取来后再把repeat\_stop\_fetch\_reg置有效

### repeat\_sequence 指令的派遣

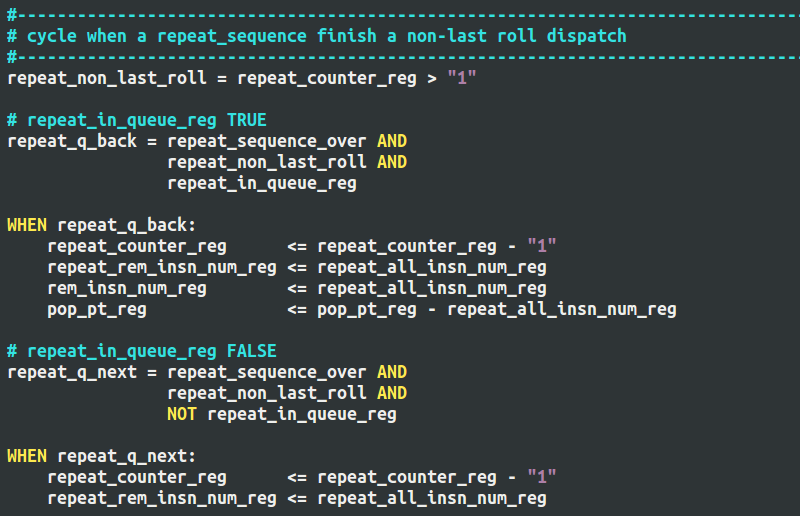
基本原则：不同轮的指令不可以混合派遣，原因：

* 会增加硬件设计复杂度
* 软件无法预测指令派遣规律

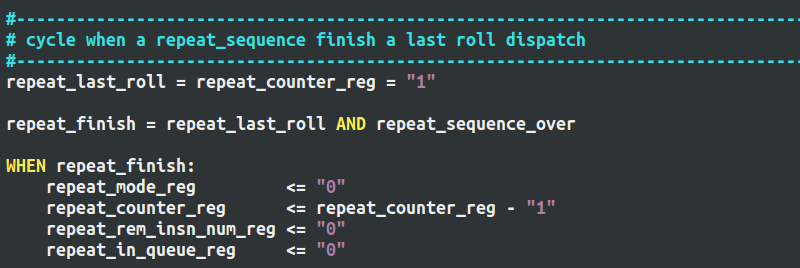
处理方式：rem\_insn\_num\_reg决定可派遣指令数，repeat\_rem\_insn\_num\_reg决定本轮可派遣指令数。



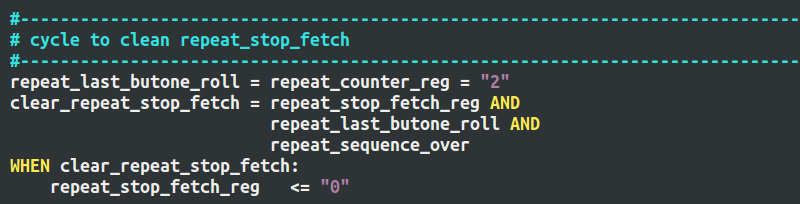
* Repeat\_sequence\_over: repeat\_sequence一轮派遣结束



* Repeat\_non\_last\_roll: 当repeat\_counter\_reg的值大于1的时候，说明还没到最后一轮
* Repeat\_q\_back：一轮指令派完了，且还没到最后一轮，且repeat\_sequence指令都在queue里，就需要指针回退，开始新一轮的指令派遣
  + Repeat\_counter\_reg: 轮数减1
  + Repeat\_rem\_insn\_num\_reg: 要被repeat\_all\_insn\_num\_reg重置
  + Rem\_insn\_num\_reg：新一轮开始，queue里的有效指令就是repeat\_sequence的全部指令
  + Pop\_pt\_reg：调整为repeat\_sequence第一条指令所在queue的地址
* Repeat\_q\_next：一轮指令派完了，且还没到最后一轮，且repeat\_sequence的指令在queue里放不下，所以需要重取新一轮的指令，新取来的指令已经顺序存入queue。
  + Repeat\_counter\_reg: 轮数减1
  + Repeat\_rem\_insn\_num\_reg: 要被repeat\_all\_insn\_num\_reg重置

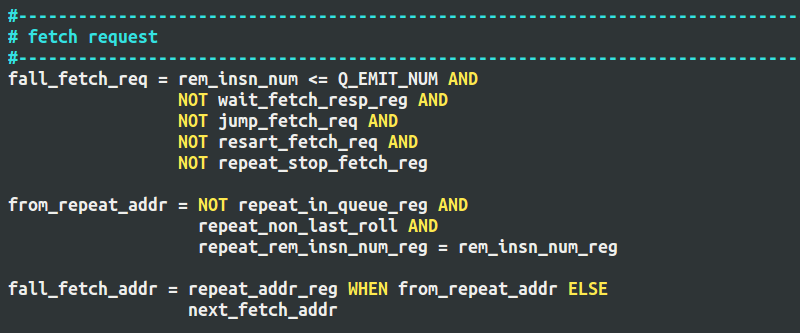


* Repeat\_last\_roll：repeat\_counter = 1， 表示repeat\_sequence派遣的最后一轮
* Repeat\_finish：最后一轮的repeat\_sequence指令也全部派遣完了。需要对repeat的一些信息进行清零处理。
  + Repeat\_mode\_reg：置0
  + Repeat\_counter\_reg：置0
  + Repeat\_rem\_insn\_num\_reg: 置0
  + Repeat\_in\_queue\_reg: 置0

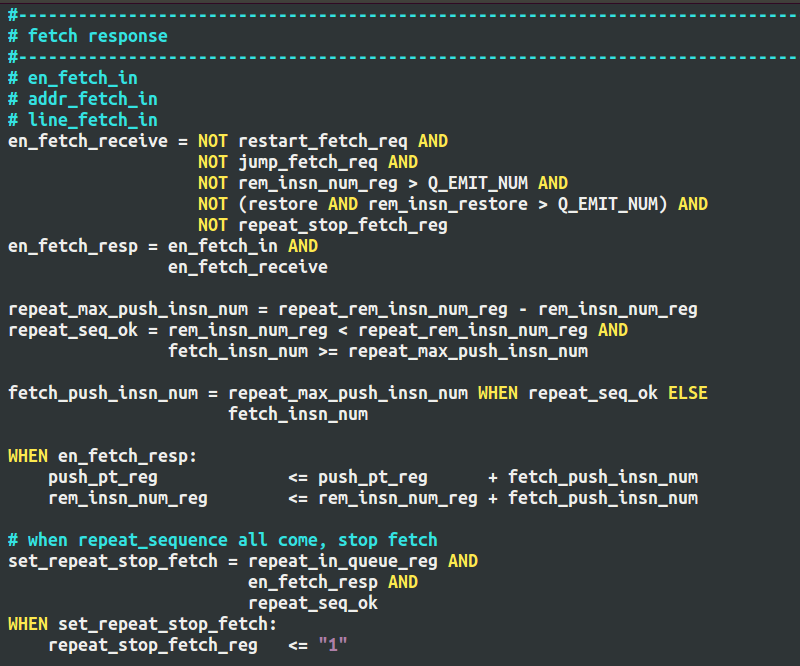


* Repeat\_last\_butone\_roll: repeat\_sequence派遣的倒数第二轮
* Clear\_repeat\_stop\_fetch：当repeat\_counter\_reg = 1的时候，即最后一轮的时候，即可以准备顺序往后的新指令了。所以当倒数第二轮派完的时候，把repeat\_stop\_fetch\_reg清零。(还有可能存在指令回退导致counter回退的场景，新取来的指令可能导致部分repeat\_sequence丢失，需特殊处理)
  + Repeat\_stop\_fetch\_reg：置0

### 加入REPEAT后的取指逻辑



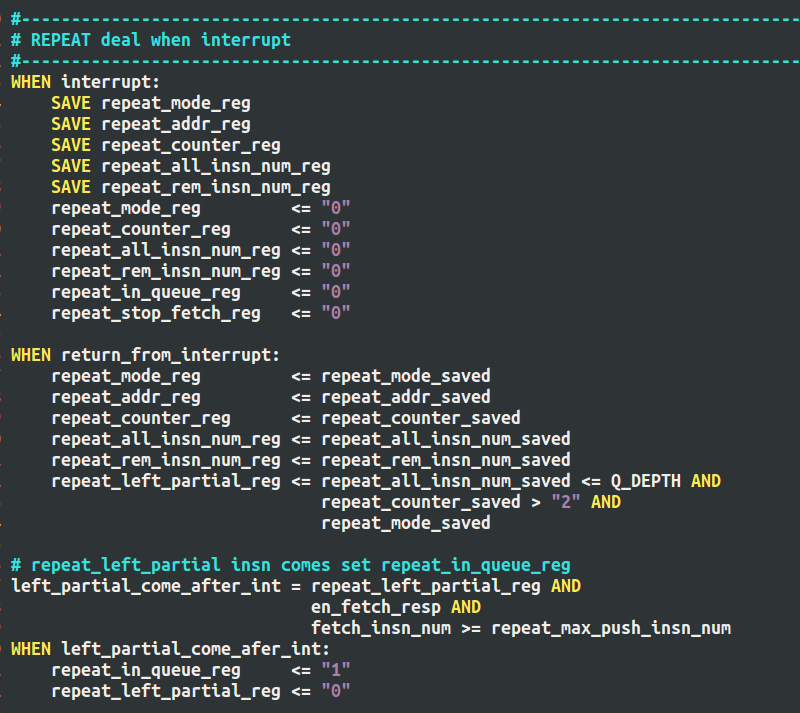
* Fall\_fetch\_req: queue发起的顺序取指请求，这里也包含了对repeat\_in\_queue\_reg条件不满足时到repeat\_addr\_reg的取指请求。正常fall\_fetch\_req发出的条件是queue里有足够的空闲项可以放一个新的cache line，且没有已经发出去的取指令请求，且当前没有更高级别的jump\_fetch\_req和restart\_fetch\_req，在这里加入新的条件：没有repeat\_stop\_fetch\_reg停止取指的行为
* From\_repeat\_addr: fall取指的地址都是下一个cache line的地址，但是这里加入了repeat\_sequence在queue里放不下时到repeat\_addr\_reg的重复取指。相当于把不同轮的repeat\_sequence打平依次填入queue。
* Fall\_fetch\_addr: 根据from\_repeat\_addr信号选择取指地址的来源



* En\_fetch\_receive: 取来的指令可以被接受的条件。没有更高优先级(jump\_fetch\_req/restart\_fetch\_req)的取指请求发出，如果有，旧的取指响应应该被抛弃；queue的空闲项数足够放取来的指令，如果rem\_insn\_num\_reg > Q\_EMIT\_NUM, 说明在取指请求到取指响应中间发生了queue内指令的回退覆盖；当拍没有指令回退且回退点会被新到指令覆盖的场景；额外加入了没有repeat\_stop\_fetch\_reg的场景
* En\_fetch\_resp: 经过各种条件过滤后真正有效的取指响应
* Repeat\_max\_push\_insn\_num: repeat\_sequence中还剩的不在queue内的指令数目
* Repeat\_seq\_ok: 接受了新取来的指令后，repeat\_sequence本轮需要的指令就全了
* Fetch\_push\_insn\_num: 通常是取来的指令全部都接收，但是如果指令超过了repeat\_sequence的边界，则只接收到repeat\_sequence的最后一条指令。
* Push\_pt\_reg: 一笔有效的取指响应要根据实际接收的指令数目对push指针进行更新
* Rem\_insn\_num\_reg：一笔有效的取指响应要根据实际接收的指令数目对rem\_insn\_num\_reg进行更新

### REPEAT被中断时的处理

Interrupt应该可以打断REPEAT的执行，因为如果软件给了$counter一个巨大的值，或者repeat\_sequence中混入了不符合规则的指令，我们要确保这些状况能用外部干涉来调整处理器状态。



* 当中断发生的时候，要把所有repeat相关信息保存起来。同时把repeat相关执行状态信息全部清零
* 当中断返回的时候，则要把保存的repeat相关信息又拿回来。如果依然回退到repeat\_mode且repeat\_sequence可以放在queue里，需要一个过程将整个repeat\_sequence装入queue。Repeat\_stop\_fetch\_reg被置零，这样要重新取repeat\_sequence的指令，repeat\_in\_queue\_reg清零，这样