超标量处理器设计SPEC

目录

[1. 项目背景 3](#_Toc186547024)

[2. FPGA选择 3](#_Toc186547025)

[3. 整体架构 3](#_Toc186547026)

[4. 设计细节 4](#_Toc186547027)

[4.1 TLB与cache 4](#_Toc186547028)

[4.2分支预测 4](#_Toc186547029)

[4.2.1局部历史预测 4](#_Toc186547030)

[4.2.2 BTB与RAS结构设计 5](#_Toc186547031)

[4.3指令解码 5](#_Toc186547032)

[4.4寄存器重命名 5](#_Toc186547033)

[4.5发射队列 6](#_Toc186547034)

[4.5.1发射队列基本信息 6](#_Toc186547035)

[4.5.2指令唤醒 7](#_Toc186547036)

[4.6指令执行 7](#_Toc186547037)

[4.6.1旁路网络的设计 7](#_Toc186547038)

[4.6.2 cluster结构的划分 8](#_Toc186547039)

[4.7指令提交 8](#_Toc186547040)

# 项目背景

为了回顾巩固《超标量处理器设计》这本书，准备学习编写一个乱序执行的超标量处理器，同时这也是为了提升自己的水平。

长远目标是要在FPGA上跑通该项目，并成功运行Linux微系统，这有一定的难度。中期目标则是要完成这整个项目的代码编写，并尝试做一些验证。

如果能借此契机参加一下龙芯杯就更好了。下列链接是关于龙芯杯的一些信息：

[【CQUT】从零开始龙芯杯-初章 - 哔哩哔哩](https://www.bilibili.com/opus/993344116458782775)

# FPGA选择

初步准备选择Xilinx AC7010系列-XC7Z010，关键参数有：512MB的DDR3和256KB的BRAM。更详细的细节参见：

[ALINX FPGA SOM AC7010: with AMD Zynq7000 SoC FPGA module Industrial Grade](https://china.xilinx.com/products/boards-and-kits/1-t5lo2d.html)

# 整体架构

本次设计主要参考《超标量处理器设计》中page346的流水线架构图，一些细节的内容有待进一步商榷、修改。



系统整体采用32位，支持MIPS32的一些基础指令，可以参加D:\个人\代码积累\Verilog\超标量处理器设计 目录下《MIPS\_Vol2》。 又或者，可以考虑参考书籍第五章中“指令集体系”的介绍中关于MIPS指令的介绍部分。

# 设计细节

## 4.1 TLB与cache参数

支持TLB，采用32位的虚拟地址，32位的物理地址。因此理论上来说DRAM可以支持到1GB。采用4KB的小页形式，也就是地址的[11:0]是虚拟地址与物理地址共有的。为了便于内存空间占用，TLB缺失时虚拟地址到物理地址的转换设计为三级结构：第一级用ASID，第二级用虚拟地址[31:22]，第三级用虚拟地址[21:12]，访问内存三次方可获得相应物理地址。

TLB设计分为I-TLB和D-TLB，每个TLB都由32级全相联的结构构成，TLB的缺失设计为硬件自动完成访问内存，如果在内存里再次发生缺失将引发page-fault。

cache支持虚拟存储器，并选用virtual-index+pysical-tag的形式，L1-cache两级流水线访存使用虚拟地址，第一级流水取出cache内容以及TLB的物理地址，第二级流水完成相应的tag比对。

cache的容量设计上，L1-cache分为I-cache和D-cache，为了方便virtual-index，用地址的[11:0]来进行索引，这样只能实现4KB的容量，在此基础上实现4路组相联，这样可以使得L1-cache总容量达到16KB+16KB共计32KB。L2-cache使用纯物理地址，预设容量为128KB，暂时不准备采用组相联的结构。关于cacheline的大小，选择64byte即512bit。

### 4.1.1 cache时序细节

将采用串行访问cache的思路，并流水化处理，第一步访问tag-SRAM得到tag的结果，同时也会并行访问TLB获取相应的pysical-address，第二拍根据tag和pysical-address的比较结果获取hit/miss信号，同时访问对应的data-SRAM(如果hit了)

### 4.1.2 L1中I-cache与D-cache的细节

综合考虑64byte的cacheline大小，以及每次取出4条指令的设计细节，我们要明确L1中I-cache的读取细节，一条MIPS32指令只有4B，所以可以将I-cache16B的cacheline分为4个4B的小BANK（当然，4B内部还可以再细分小BANK），这样其实每次取指令只会激活其中一个BANK，这里涉及到取指令的地址不是4字对齐的问题，如果取指令地址不是4字节对齐的，那就只取其后侧的指令作为有效指令。

I-cache中，每条cacheline的内容包括：1bit vld信号位，20bit tag数据位，还有512bit即64byte的data数据位。

D-cache的设计可以与I-cache一致，也将4B作为一个BANK去使用，好在数据访问的时候是自动字节对齐的，这样也就无需考虑一个32bit的数前半部分在一个BANK，后半部分在另一个BANK的情况。但是比起I-cache数据位，D-cache多出1bit的dirty信号位。每条cacheline的内容包括：1bit vld信号位，1bit dirty信号位，20bit tag数据位，还有512bit即64byte的data数据位。

L2cache的基本构成与D-cache类似。在进行设计的时候，将vld位dirty位与tag数据一起放在tag-sram，data数据可以单独放置在data-sram。

## 4.2分支预测

结合取指方面的设计，每次取指的地址边界默认都是四字对齐的，但是由于跳转指令的存在，可能会使得某次PC不是四字对齐的，但是取指令的时候只会取出本次四字对齐中的相关指令。所以本次超标量处理中的分支预测就是基于这四字对齐的进行一次预测：我们预设4字中最多只有一条分支指令，这已经可以cover绝大多数的情况，正如上述所言，有可能此次PC不是四字对齐的，所以需要表明跳转指令在四条指令中的位置。

整体的预测格局是：预设4字中最多只有一条分支指令，对其进行分支预测，PC给出后根据历史预测得知是否跳转，根据BTB得知预测地址，然后将PC进行相应的变化。BTB的资源很宝贵，只有分支指令才会在其retire的时候将目标地址写入到BTB中，因此设计的一处细节是：如果分支预测是发生跳转，但BTB中没有找到相应的目标地址，那么系统将按照不发生跳转继续执行。

现在我们需要细致考虑分支预测的执行过程：在获取了分支预测的地址之后，这个地址要和PC一样随着pipeline流动，分支预测的指令需要将预测的地址结果和计算的地址结果进行比较，通过比较才能知道预测的是否正确，是否需要进行状态恢复。这也体现成分支预测指令要单独设立一个发射队列的必要性，其他类型的指令没有存储这32bit的必要。接下来要细致考虑我们的前提条件：四字对齐的四条指令中只有一条指令是分支指令，这只是我们假想的前提条件，我们分为三个方面来考虑这个问题：

第一种情况：四字对齐的四条指令中确实只有一条指令是分支指令，那么很简单，预测的地址直接与这条分支指令配对即可，也不用考虑别的情况。

第二种情况：四字对齐的四条指令中不止有一条指令是分支指令，但是第一条分支指令预测跳转，这种情况其实本质上还是第一种情况，虽然有多条分支指令，但是第一条已经预测跳转，decode的时候他后面的指令就不会是vld状态，更不会被执行，预测跳转的地址直接和第一条指令配对即可。在写入到IQ和ROB的时候也只用写入这一条指令。

第三种情况：四字对齐的四条指令中不止有一条指令是分支指令，但是第一条分支指令预测不跳转，这种情况下，四字对齐的四条指令其实全部都已经被预测不跳转，因此在将这些分支指令写入到IQ和ROB的时候，与他们配对的预测地址都标记为PC+size of (fetch)。

### 4.2.1局部历史预测

局部历史预测：参考《超标量处理器设计》page111图4.17，不过放弃hash处理。对于PC[31:0]来说，PC[1:0]恒为0，且我们是基于四字对齐来进行分支预测，所以利用PC[9:4]去寻址BHT，可见BHT的表项有64个，且每个表项记录的BHR为4bit，即记录某一条分支指令过去四次的分支情况。最后用BHR+ PC[4:2]这样一个7bit的数去寻址PHT，得到PHT中两位饱和计数器的值，完成预测。很显然，系统可能会出现4字对齐中不止一条指令，此时预测的准确度极低。

局部历史预测所需的BHT：2^6\*4bit，局部历史预测所需的PHT：2^7\*2bit。BHT以及PHT的更新选定在指令retire的时候，由于系统在预测的时候选择了只预测四字对齐中的第一条分支指令，所以在retire修改的时候，也只将4字对齐中的第一条指令信息写回到BHT及PHT中。

### 4.2.2 BTB与RAS结构设计

BTB的结构与cache类型，参考P124图4.32，但是有一处细节需要明确，基于SRAM的cache其时序是：第一拍给出访问信号，第二拍给出结果。这在CPU的流水线中无法满足要求，PC寄存器要在每一拍都完成相应的变化。因此BTB结构的底层器件应该基于寄存器堆/特制的SRAM。

为了应对超标量处理器的设计，预设的取指模式是在4字对齐的边界，所以我们使用PC[31:4]去访问BTB。这里有两个重要的前提，第一是BTB里只记录了对应4条指令中一条分支指令的跳转地址，所以我们预设4字中最多只有一条分支指令；第二是由于取值总是在4字对齐的边界，有时候我们可能无法一次取出四条指令：如四字对齐的四条指令为0123，如果给的地址指向0那我们就将0123都取出，但有可能给的地址指向2，那我们即使0123都取出也只能使用23。这很重要。

BTB的结构预设在128个表项，使用PC[10:4]，tag项为PC[31:11]，除此之外BTB的内容还包含一个vld项、2bit表明分支指令类型、4bit表明分支指令在该组指令中的位置以及32bit的目标地址项。预计容量所需：2^7\*(1+2+21+4+32) bit≈8Kb≈1KB。

针对BTB的写入，需要明确的是：BTB中存储的并不是下一拍PC的地址，而是分支指令预测要跳转过去的地址！用不用BTB中的结果地址还要根据前方局部历史预测的结果。BTB主要用于直接给出offest的跳转以及call指令相关的跳转。对于前者，只要在解码阶段解出了相应的offest，就可以根据PC将PC+offest写入到BTB中去。对于后者，只要指令被执行计算出目标地址，就可以进行相应的写入，之所以不在retire的时候写入是因为BTB中存储的并不是下一拍PC的地址，而是分支指令预测要跳转过去的地址！他不需要指令真得执行，这里要想清楚。所以BTB无需考虑分支预测失败状态恢复的问题。

RAS选择栈结构，先进后出，他的时序需要和BTB同样，也就是能在一拍内给出结果，预设容量在2^6\*32bit=2Kb，也就是最多可以存放64个最近的return指令的目标地址。RAS还有一处细节是如果RAS已经被填满了，则按照次序继续填充。

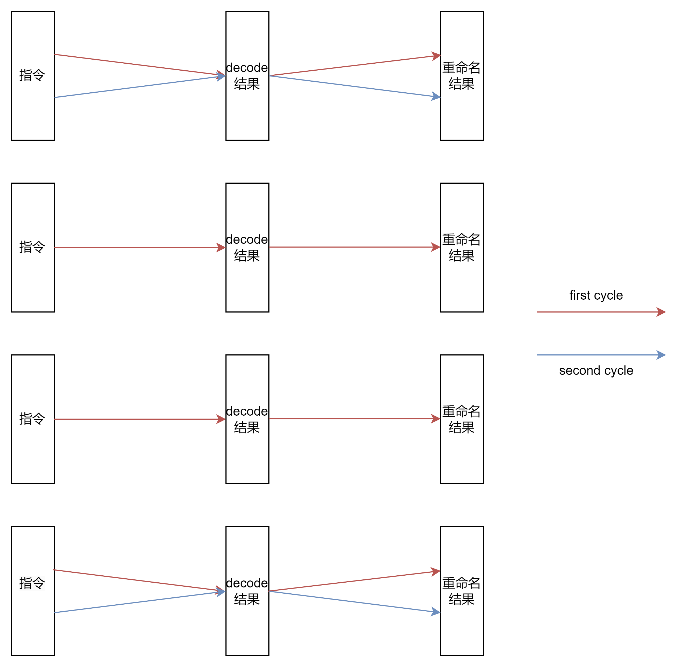
关于RAS的写入与弹出，在访问BTB的结果给出：此次分支指令是call指令的时候，就需要把PC+4写入到RAS中，同理在BTB的结果给出此次分支指令是return指令的时候，RAS的栈顶要弹出一个数据，RAS要考虑状态恢复的问题，可以参考RAT的状态恢复：在retire处设立一个栈顶影子指针，每当有call指令retire，就将影子指针+1，每当有return指令retire，就将影子指针-1，这样在遇到分支预测失败的时候，直接将影子指针赋给栈顶指针即可。

## 4.3指令解码

解码宽度设计在4条指令，也就是4-way超标量处理器。上一个取指令环节，我们约定取值地址默认是四字对齐的，如果因为跳转指令或者其他指令使得PC的数值不再四字对齐，而是落在了四字中，那么取出的指令只有PC及其后方的指令有效。

指令解码环节做的事有四项：第一分析出这条指令是什么类型的指令，方便后续完成重命名后写入到发射队列中。第二分析运算指令进行的是哪种运算，方便向ALU提供操作码。第三指示出指令运算的源寄存器，目标寄存器，是否为立即数等。第四，对于直接跳转的分支指令，用于检查分支预测的地址是否正确，如果预测不正确的话就要立刻flush掉前方的错误环节，并从正确的地址开始取指令继续运行。

在解码单元前没有设计Instruction-buff，取出四条指令直接送往解码单元，对于乘法以及乘加这样的有两个目标寄存器的特殊指令，采取了一种特殊的策略，将整体重命名分为两个时序周期来完成，有些类似书中page184介绍的方法一，只不过将缓存的位置由一个新的结构转换为重命名结果寄存器，也就是拓宽了重命名结果寄存器。正常情况下，重命名的结果会有四个寄存器idx，两个源寄存器，目标寄存器的old\_idx和new\_idx，乘法以及乘加指令会使得多出一个目标寄存器，可以拓宽一下承载重命名结果的数据，拓宽了2x6bit=12bit，这12bit只有乘法指令、除法指令、乘加减指令可以使用。这里的拓宽还有一个好处是可以直接将两条decode出的指令直接作为一条指令写入到乘除法IQ中。除此之外，还需要将重命名结果写入到ROB中，这里也有一定的特殊处理。下文将会介绍。



如果四条指令中包括了乘法指令、除法指令和乘加减法指令（即使只有一条），那么就需要两拍来完成整个解码工作，第一拍上述指令解码出前半段，第二拍解码出下半段，在第一拍的时候，需要解码阶段之前的流水线保持静止hold住。这样做的好处是，四条指令中不论有几条这样有两个目标寄存器的特殊指令，整体都只需要两个时钟周期就可完成重命名。

但实际上，完成了重命名还要考虑重命名写入到ROB的情况，ROB的空间较为宝贵，给ROB扩容来承担这多出的目标寄存器不值得，所以采用了延时来完成写入ROB的操作。上图的例子在hold一个cycle的基础上刚好可以完成写入ROB：first cycle结束到second cycle开始的上升沿可以用两个写端口完成NO.2 NO.3指令的ROB写入，second cycle结束时可以用四个写端口完成NO.1 NO.4指令的ROB写入。这种方法可以cover多数的情况。

但当四条指令中包括3/4条多目标寄存器指令的时候，就需要再延时一个cycle来完成写入（因为只有ROB写入端口只有四个）。最坏的情况即四条指令都是多目标寄存器指令，完成从decode到dispatch需要hold住2个cycle，共计3cycle来完成整体操作。

## 4.4寄存器重命名

寄存器的重命名采用统一的PRF寄存器重命名。

逻辑寄存器中，R0恒为0，不参与重命名，除此之外，Ho以及Li也要参与重命名，共计33个整数寄存器以及32个浮点数寄存器。R0恒为0不参与重命名这里，要细致考虑一下，是否需要为了这里的特殊情况在流水线寄存器中加入一些控制信号，因为他的流程和其他通用寄存器格格不入，其他操作都按照规划好的流水线，进入重命名-取操作数-执行。他既不需要重命名，也不需要到PRF中取操作数，最后执行的时候也不要写入。整体的思路设定为巧用decode过程。如果decode时发现目标寄存器是R0，那就把这条指令标记为NOP指令，这样他后续就不会再执行。若decode时发现有源操作数为R0，那么可以指示此处的操作数为立即数0，这样下一个环节就不会再去访问使用立即数的值。

关于物理寄存器的个数，整数寄存器和浮点寄存器暂定为各64个。

整数寄存器和浮点寄存器的重命名相互独立，最极端的情况为：四条指令都为整数指令/四条指令都是浮点指令。在解码时也会给出是否为整数/浮点运算指令，重命名环节可以根据控制信号分配指令的重命名。

使用统一的PRF寄存器进行重命名，重命名映射表最极端的情况下要准备12个读端口和4个写端口，与此同时4个指令之间的相关性也要考虑，这里的细节很多，参考书上page215，逻辑要写清楚。

从寄存器重命名环节开始，要着重考虑分支预测失败以及出现异常等情况的状态回复问题。

这里采用基于ROB的状态恢复方法，没有使用checkpoints。对于重命名环节，状态恢复要考虑的就是重命名映射表(RAT)和free-list。

对于重命名映射表，在指令退休处将指令退休结果的映射关系写入a-RAT，这里a-RAT是绝对正确的，进行状态恢复时将a-RAT的内容复制给前方的RAT。对于free-list的恢复，需要根据ROB中后续指令中包含目标寄存器的个数直接调整读指针，这里的原理是无误的，细节需要想明白，但是这里有一个小细节是ROB中存储的指令并不一定都有目标寄存器，可以设立一个计数器在rename环节后，对多少个目标寄存器rename了，就+多少，同时retire环节retire了多少带有目标寄存器的。等待分支预测失败进行状态恢复的时候，

## 4.5发射队列

### 4.5.1发射队列基本信息

对于整体CPU的发射队列来说，采用分布式、非压缩式、非数据捕捉的形式。

发射队列的具体内容细节参考page259中图8.39。

为了简化设计逻辑，设计的思路是一个发射队列对应一个FU，这样乱序执行的逻辑只需要考虑1 of N即可。分为六个IQ，分别是：加减法IQ、逻辑运算与移位IQ、乘除法IQ、访存IQ、分支预测判断指令IQ、浮点运算指令IQ。乘除法为何要一个IQ呢？原因在于乘除法中有一些像乘加的指令，他们有四个源寄存器和两个目标寄存器，需要特殊的处理。分为六个发射队列，对应的仲裁电路和唤醒电路自然翻倍，为了防止布局布线资源紧张，各个IQ的entry容量考虑设计的小一些。

这里的细节设计在于，访存IQ与分支预测判断指令IQ都设计为普通的FIFO结构，他们没有被设计为可以乱序执行。

预设entry数目设计为：加减法IQ分配8entry；逻辑运算与移位IQ分配8entry；乘除法IQ分配4entry；访存IQ分配4entry；分支预测判断指令IQ分配8entry；浮点运算指令IQ分配4entry。

非压缩式的IQ需要能够在队列中找到已经可以发射给FU的指令，这里可以借助寻找第一个1的算法，将各个vld信号拼凑成一个序列，利用第一个1算法去找到相应的待发射指令。

采用非数据捕捉形式，则需要在后方准备寄存器堆，由于需要访存的端口众多，准备对寄存器堆采用cluster结构，像page302中图9.39一样，关于FU的分配，加减法FU+逻辑运算与移位FU+访存FU用一个寄存器堆，而乘除法FU+分支预测判断指令FU用另一个寄存器堆。

### 4.5.2指令唤醒

现在我们要着重考虑多周期指令延迟唤醒的问题，准备采用延迟唤醒的思路，这需要指令在解码的时候准备好指令执行所需要的时间，并写入到流水线寄存器中，最后写入到发射队列的表项中。在发射队列中某一条指令被选中的同时，目的寄存器编号和延时信息都会给到相应的广播总线。最终完成相应的唤醒。这种方法是对于执行周期明确的指令而言的。

对于load这种事先不清楚要多少个cycle的指令，采取另一种思路。

较为先进的方法应为对load指令做推测唤醒，推测唤醒涉及到推测错误时指令重写回的问题，还有已唤醒的寄存器再次置为无效的问题，这些问题都比较复杂。为了简化设计难度，对load指令采用最原始的设计方法：访存IQ为传统FIFO结构，不设计为乱序执行，同时对于load指令的目的寄存器的唤醒，只有当相应数据在D-cache种成功找到才会进行相应的广播。

对于指令的延迟唤醒形式，上面说采用的延迟唤醒的思路，对于一般的指令只需要数个周期就可以完成运算，延迟信息也只需要几个bit，但是对于整数除法来说，FU的设计中采用了状态机循环减法的方式实现，具体执行多少个cycle是不定的，因此可以对整数除法采用和load指令相同的操作，待结果运算完成再进行相应的广播。

## 4.6指令执行

### 4.6.1旁路网络的设计

在指令执行的时候，有一个很重要的点要考虑，就是同一个FU中不同操作的指令如果需要不同的周期，可能会导致出错，举例来说，上一个操作要3个周期，下一个操作要2个周期，这两个运算会同时抢占FU的出口。好在本次设计中将指令运算FU分的很细致，根据4.5.1中分为六个IQ：加减法IQ、逻辑运算与移位IQ、乘除法IQ、访存IQ、分支预测判断指令IQ、浮点运算指令IQ，我们可以使用对应的六个FU：加减法FU、逻辑运算与移位FU、乘除法FU、访存FU、分支预测判断指令FU、浮点运算指令FU。在这些FU中，只有整数乘除法可能遇到上述的问题，这里是这样解决的：FU此时如果是在进行乘法，则后续只能吞吐乘法进入，只有FU中没有运算可以吞吐除法进入，且除法在运算的过程中不再吞入运算。（这里是结合状态机的方法实现32位整数除法，运算时长不定）

对于旁路网络的实现，采用了cluster结构设计，参考书上page304中图9.40，本次设计中将FU拆分的较为细致，如果使用平常的bypass电路，那么跨FU的bypass电路压力将会非常大，加减法FU、逻辑运算与移位FU、乘除法FU中的乘法都要提供bypass的入口选项。使用cluster结构的bypass电路之后，可以取出source-drive和result-drive电路，给execute结构省去两级流水，同时bypass电路只在每一个FU内部存在，别的FU如果想要使用运算结果就需要访问寄存器，对于寄存器访问设置为支持同时读写。读寄存器时如果本周起恰好有写该寄存器的需求，那么读出的数据将会是要写入的数据，同时完成写入。

基于load指令唤醒寄存器的设计中，采用了最原始的得到数据再唤醒对应的寄存器（除法也是同样的操作），所以他们同样不必进行相应的bypass电路结构入口选项提供。

### cluster结构的划分

在4.5.1中已经提及了对寄存器堆进行cluster结构，加减法FU+逻辑运算与移位FU+访存FU用一个寄存器堆，而乘除法FU+分支预测判断指令FU用另一个寄存器堆。在此基础上，每个FU的bypass电路也是bypass的，即只给自己bypass，其他FU如果想使用对应的结果，只能从寄存器堆中访问。

## 4.7指令提交

指令提交最重要的环节就是ROB，ROB的设计细节参考page323中图10.2，ROB中要记录分支预测指令是不是是否预测正确，这部分内容准备加入到图10.2中指令类型的那部分，通过对这部分标记，完成记录是不是分支预测失败的功能。与dispatch环节保持一致，每一拍ROB可退休4条指令。

在寄存器重命名的时候选择了使用统一的PRF的方式进行重命名，这种方式需要结合free-list与busy-table使用，busy-table是辅助表明寄存器状态的有助于free-list的补充，本次设计中在ROB中记录old-physical-register，在指令retire的时候，将old-physical-register释放写回到free-list。在这个基础上，正常的处理器为了帮助外界查看寄存器状态，会有一个辅助结构来存放寄存器的状态是architecture还是physical，这里为了简化也没有加入此功能。

在指令提交的时候又可能遇到分支预测失败/异常/中断，这时候需要等待相应的指令成为最old的指令，然后进行相应的状态恢复，对于RAT的状态恢复最为简单，可以直接将a-RAT复制给RAT，对于free-list的恢复，需要根据ROB中后续指令的个数直接调整读指针，这里的原理是无误的，细节需要想明白。

对于store指令，当其退休时就被写入到store-buffer，store-buffer中的指令会按照次序依次写入到cache中，所以load指令在搜索cache的同时也要搜索store-buffer的内容。最后一点小细节是，store-buffer的规格制定，store-buffer预计设计为8格FIFO结构，且每次只能写入一个store指令，如果待退休的指令有超过一条store指令就mask住，不让他退休。