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

预计采用历史预测的策略。

### 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]，[1:0]永远=0。利用PC[7:2]去寻址BHT，可见BHT的表项有64个，且每个表项记录的BHR为6bit，即记录某一条分支指令过去六次的分支情况。最后用BHR+ PC[5:2]这样一个10bit的数去寻址PHT，得到PHT中两位饱和计数器的值，完成预测。

在四decode-width的超标量处理器中，每次要实现对四条指令的PHT查询，所以对BHR以及PHT都要使用4BANK的堆叠设计，由于访问PHT的地址是BHR+ PC[5:2]，访问BHR的地址是PC[7:2]，四条相邻指令的最后2bit必然是不一样的，可以完美地实现交叠访问。

局部历史预测所需的BHT：2^6\*6bit（其中2^6分为2^4\*4），局部历史预测所需的PHT：2^10\*2bit（其中2^10分为2^8\*4）

### 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，每次取出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，那我们即时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

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。

## 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。

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，采取的策略是《超标量处理器设计》page184中图6.4的方法，暂停解码之前的流水线，等待两次解码完成。

指令解码要做的事有两件：分析出源寄存器以及目标寄存器提供给重命名单元；保留指令中其他的重要信息，如执行什么操作，立即数是多少等等。

## 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。

关于物理寄存器的个数，整数寄存器和浮点寄存器暂定为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，ROB中按照指令原有次序记录了指令重命名后的信息，如果一条分支指令发生了错误，那么我们就知道，ROB中其后面的指令，如果他有目标寄存器，就说明他错误地从free-list中要了物理寄存器，我们可以据此再将这些寄存器写回到free-list中。这个过程可能需要时间。我们要考虑的问题是，状态恢复时是不是必须立刻将free-list进行恢复，可能并不是，因为free-list一般还有一定的余量，足以支撑流水线继续进行，只要其他的环节恢复完成，待加入free-list的信息都在ROB中，可以后续慢慢进行。

## 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。