# 模块说明

**总体架构**：

典型的单发射五段流水CPU，在IF段取指令，在ID段译码，并读取通用寄存器、HILO寄存器和cp0寄存器；在EX段进行ALU的运算；在MEM段读写数据存储器，并进行异常处理；在WB段进行通用寄存器、HILO寄存器和cp0寄存器的写回。

用数据重定向进行大部分数据冲突（ReadAfterWrite读后写）的处理，用插入一个气泡的方式处理load-use数据冲突，数据冲突判断的是ID段的读和EX、MEM段的写之间的关系。分支跳转在ID段进行，由于有延迟槽指令，所以跳转时不用清空任何一段。所有的异常都送到MEM段处理，发生异常跳转时，MEM段之前的段都要清空。

## decoder

将IF段取出的32位指令，送到ID段，截取出各字段（op、rs、rt、rd、shamt、func、imm16、imm26），就是简单的赋值，甚至都不需要做成一个单独的模块，直接在数据通路里面就可以赋值实现。

## controller

控制器。先根据操作码op、功能码func等，识别指令。比如addi指令op的值为001000，那么就用assign addi = ~op6 & ~op5 & op4 & ~op3 & ~op2 & ~op1;识别出该指令。有些指令用op字段还不够，则需要更多的与非门。识别指令还可以用assign addi = (op==001000)，这样相当于一个比较器，而不是前一种方式的与或非操作。在组成原理课程实验中，老师说过不建议用比较器，这样开销比较大。具体开销有多大，我们没有测试过，而且也不确定vivado是否会自动对这种比较语句进行优化。不过不管哪种写法的开销大，都不至于成为系统的瓶颈。

有些指令存在特殊情况，比如beq，特殊情况为b，一定跳转，它就是在beq的基础上使某个寄存器的值为0，在控制器中不用专门识别b指令，因为它已经涵盖在一般情况beq中。不过我们的控制器有个反例，识别了nop指令，它是sll的特殊情况，这是因为功能仿真时有个bug无法解决，迫不得已才利用了nop指令。

## regs

通用寄存器组，ID段。5位地址控制32个32位寄存器，两路读，一路写，0号永远为0。上升沿写入数据。读数据则为组合逻辑。在组成原理课程设计中，老师建议在下降沿写回数据到寄存器，这里为了统一，没有使用下降沿，而是在上升沿写入的，不过这样可能会使得要写回的数据不能及时被ID段读到，为了解决这一问题，在每个上升沿到来之前，我们就把要写入的数据读出了。等于说，数据还没有被写入到寄存器，就越过了寄存器被直接读出了，可以称作“读穿”，这样能保证WB段写的数据能够及时被ID段读出。

## hilo\_reg

HI LO寄存器，位于ID段。一开始使用了清华大学的HILO寄存器，后来发现不好用，就自己改了，所以这一部分完全是自定义的。

同样是上升沿写入，组合逻辑读出。能够修改HILO寄存器的指令仅限乘除法、mtlo和mthi，所以我们在控制器中根据这几条指令生成了一个读写hilo寄存器的模式信号，2位，11表示乘除法，将ALU的运算结果写入；01表示mthi，10表示mtlo，将通用寄存器的值分别写到HI和LO。这两位信号用来控制hilo寄存器的写入。

## reg\_read\_select

对通用寄存器读的地址的选择。有些指令用rs字段作为读寄存器编号，有些则用rt，需要先选择好再送到寄存器，选择信号由控制器生成。

## reg\_write\_select

对通用寄存器写地址的选择，同理，有些指令用rt作为作为写地址，有些用rd，需要选择。选择信号由控制器生成。

## reg\_din\_select

通用寄存器写回段要写入的数据的选择。有6路可能的写入数据：ALU的运算结果、PC的值、HI寄存器的值、LO寄存器的值、CP0寄存器的值、访存的结果。选择信号由控制器生成。

## extend

位扩展器。ALU的操作数是32位的，而I型指令的立即数是16位的，所以需要扩展。具体是符号扩展还是无符号扩展，要遵守MIPS规范，这一模块就是对16位立即数进行扩展。

后来又发现，移位指令sll等，有shamt字段，表示位移量，只有5位，最终也需要接到ALU作为操作数，所以也需要扩展，就把这一路也加进来了。不过现在发现这是多此一举，因为shamt扩展成32位以后，送到ALU，ALU最终也只用了低5位进行运算，也就是高位都没有用，扩展白做了，还不如不扩展直接送ALU，这样虽然vivado会警告位宽不匹配，但是不影响功能。这种涉及到数据选择、位扩展等的操作，都不是核心部件，放在哪一段选择，哪一段扩展，都是可以的，具体看自己的设计。有些选择甚至不需要写成单独的模块，直接在定成模块里用一条语句就可以实现。

## alu\_select

ALU有两个操作数，两个操作数都可能有多路来源，通过选择信号进行选择，选择的结果送ALU的输入端。可能的数据来源有，通用寄存器的两个输出值、移位指令的5位shamt字段、I型指令的16位立即数的有符号或无符号扩展，还有lui指令的操作数为常数16。其中，shamt、16位立即数都已经在extend模块进行了扩展，统一成32位输出了，所以实际上是从通用寄存器的两个输出、位扩展结果、常数16中进行选择，选择信号由控制器生成。

Extend模块和alu\_select模块，严格来说可以合并。我们设计不够合理，先在ID段进行了位扩展，送到EX段，然后再把位扩展的结果用alu\_select和其他输入一起选择，这相当于进行了两次选择，可以缩减为1次，只是增加选择信号的位宽就可以了。

## ALU

运算逻辑单元。ALU的移位运算、逻辑运算都没有什么好说的，重点是乘除法。

一开始进行功能测试，不求效率，只求功能正确，可以用Verilog语言中的‘\*’、‘/’来实现。不过，这两个符号进行的是无符号运算，如果需要有符号乘法，应该用$signed(X)\*$signed(Y)，除法同理。

到了性能测试阶段，这一用法就根本行不通了，主频可能30MHz都达不到，乘除法严重限制了系统性能。乘法的解决方法是，把32位扩展成64位，再进行运算，如果是无符号乘，就用0扩展，有符号乘则用符号扩展。扩展后的操作数用乘号‘\*’进行乘法操作，结果是正确的。‘\*’在vivado综合时会使用板子上的DSP slice，可以单周期完成乘法。

除法用位扩展的方式行不通，我们目前能想到的办法就是拆分成多个周期。组成原理课程中学过多个周期实现的除法，大概原理是不停地移位相加减，32位就需要32次。在网上搜索除法的Verilog代码，基本上都不满意，当时又不想自己写，所以就用了清华大学去年参赛用的除法代码，清华也是引用了别人的开源代码。除法过程需要32个周期，这个过程需要停机等待，发送一个正在运算的信号，迫使流水段停顿。由于清华的ALU设计与我们不一样，所以除法的调用、停机信号的生成都不一样，不能直接移植，我们在这一部分花了很多时间才调通，所以建议不要使用清华的除法模块不太好用，自己根据组成原理的知识写个多周期的除法比较好，这样对整个模块都很熟悉，调试起来也方便。

另外，在跟清华的学长交流中了解到，他们一开始做的是30多个周期的除法，但是后来又改成了2周期，性能也不差。这就相当于，除法需要进行32次运算，你可以把这32次运算均分到2、4、8、16或者32个周期。分到32个周期固然是没有问题的，但是浪费的周期就太多了，很有可能，分成4个周期来做，每个周期也不是太长，不会影响主频。具体做成几个周期，需要做测试才知道。

## Branch\_Jump\_ID

生成分支、跳转目标地址的模块。分支（branch）指令根据两个通用寄存器的值对比，判断是否跳转，目标地址是在当前地址上加上16位立即数的符号扩展。跳转（jump）指令一定跳转，目标地址由26位立即数和当前地址合成。JR指令也一定跳转，但目标地址是通用寄存器的值，不是立即数。

这一模块不负责跳转，只负责生成跳转地址和跳转信号，送到PC模块，由PC模块来跳转。如果跳转信号为1，则下一条指令的地址是分支目标地址，否则不跳。

## Conflict

冲突检测模块。大部分的数据冲突都通过重定向解决了，只有load-use冲突需要插入气泡。这一模块就是用来检测ID段的use和EX、MEM段的load，生成气泡信号。还有个special\_pop，请忽略，这是因为我们的CPU放到龙芯提供的功能测试框架下，有点bug，只好插个气泡。

## redirect\_reg\_id

通用寄存器重定向。检测ID段读寄存器与EX、MEM写的寄存器编号是否冲突，将EX段写通用寄存器的值、MEM段写通用寄存器的值到ID段，这样就保证ID段读到的寄存器值都是正确的。EX段和MEM段要向通用寄存器写什么值，就重定向什么值，而写到通用寄存器的值有很多，包括ALU的运算结果、HILO寄存器的值、CP0寄存器的值、PC的值等，所以需要选择信号，与reg\_din\_select模块的选择信号相同。理论上可以把reg\_din\_select模块的输出直接作为重定向的值送到ID段，但在这样存在一个问题，reg\_din\_select的输入中有一路是访存的结果，涉及到访存的情况，我们是不用重定向的，需要插气泡（就是load-use冲突），也就是说这一路输入在重定向逻辑中没有用到，但是它仍然存在于电路中，会被vivado检测为最长路径，最长路径的时延太大，会降低主频。根据我们使用的经验，vivado虽然可以检测最长路径，但是它还不能检测这条路径有没有用，就好比，在两个点之间连了3跟线，它能够把最长的那根指出来，但它不知道最长的那根线永远都不会使用，这时候我们应该把最长的线路剪断，也就是从多路选择器中去除这条用不到的路径。

## redirect\_hilo\_id

HILO寄存器重定向。HILO寄存器与通用寄存器一样，在ID段读，在WB段写，所以也存在数据冲突问题，不过情况没有通用寄存器复杂。要读HILO寄存器的指令只有mflo和mfhi，要写HILO的指令只有乘除法、mthi、mtlo。如果EX或MEM段是乘除法，则把ALU的运算结果定向到ID段，若为mthi或mtlo，则把通用寄存器的值定向到ID段。

## IF\_ID、ID\_EX、EX\_MEM、MEM\_WB

流水接口部件。四个寄存器，用于信号传输，标准的寄存器写法。

## Pc

程序计数器。正常情况下每次PC加4；跳转信号为1时，跳转到分支目标地址；异常信号为1，跳转到异常处理地址。

PC还需要检测地址的合法性，非法地址需要报告地址异常，通过流水段传输到异常模块。合法的条件：低2位为0。这一功能也可以从PC模块中单列出来，独立成一个模块。

## illegal\_addr

用于判断数据地址是否合法。程序地址是否合法的判断集成到了PC模块中，数据地址的合法判断则用了一个单独的模块。合法判断条件与程序地址有所不同。对lb、sb、lbu指令，任何地址都合法；对lh、sh、lhu，低位必须为0；对sw、lw指令，低2位必须为0。

## dram\_mode

用于生成数据RAM访问需要的4位信号。4位信号分别对应32位数据的4个字节，哪个为1就表示哪个有效。sw指令为全1，sb、sh则要根据地址低位进行判断生成。

## dm\_in\_select

写入数据RAM的数据选择模块。送到数据RAM的数据一般来说是通用寄存器的值，不能直接把该值作为数据RAM的输入，比如sb指令，它只写32位中的一个字节，要把需要写的数据向左移位到相应的字节位置，再接到RAM上。这个是MIPS标准规定的，具体可以参照sb、sh等指令的说明，32位数据需要和dram\_mode生成的4位写使能信号配合使用才能将数据成功写入。

## DMout\_select\_extend

数据RAM读数据选择模块。读数据和写数据同理，有可能只读1个字节或者半字，需要移位和扩展，由这个模块实现。

## Datapath

数据通路，调用所有子模块。信号的命名基本上是按照五段进行的，绝大多数信号都带有if、id、ex、mem、wb后缀。所有的模块中宏的定义，我们都写在了模块的开头，其实最好的方法是使用头文件统一进行宏定义。

# 功能测试

功能测试阶段主要追求功能正确，没有太考虑代码的效率。一开始就确定一个比较合理的架构当然是很重要的，但是在具体实现时，不用太纠结代码怎么写，功能正确就行了，比如乘除法的性能问题，暂且不管，只要是有关性能的事都可以丢到后面性能测试阶段去解决。有些种类的冲突，如果数据冲突处理模块没有设计好，重定向存在bug的话，就直接插入气泡，这样可以掩盖bug，暂时通过功能测试。等到性能测试发现插气泡太浪费周期了，再回来完善。

# 性能测试

性能测试要使运行测试程序的时间越短越好，CPU时间=每个周期长度×周期数。每个周期的长度由主频决定，周期数则由气泡数、访存停机等决定，性能优化也就是想方设法提高主频、减少周期。

## 减少周期数

### 减少插入的气泡

对冲突的处理要考虑清楚，不要插入没有用处的气泡，按理说需要插入气泡的只有load-use冲突，这种情况不是太多。

### 减少访存时间

大概又可以分成两个方面：减少访问次数、减少每次访存的周期数。

AXI总线自行实现，一次访问需要经过握手确认等多个阶段，需要将这个过程的周期数降到最小。为了减少一次访问的周期数，可以在一个周期的上升沿和下降沿分别做不同的事情，但是这样也有坏处，会使得主频降低，需要在二者之间权衡。

总线的有些周期是没法省去的，也就不要在这方面下功夫了，把burst突发传输实现了就有比较高的效率了。

在一次总线访存的时延无法继续减少的情况下，我们要做的就是减少总线访问的次数，使cache更高效。我们的cache命中率在99%左右，已经算是比较好了。Cache使用哪种相联方式最好，一行多大，几路相联，都需要进行测试。我们进行了多种参数下的cache的测试，当指令cache太小（4KB）时，有一项性能测试的分数极低，提升到8KB后，性能明显好了很多，但是继续增大到16KB，性能则没有多少变化，说明在我们的设计方案下，指令cache8KB就够用了，用16KB固然可以，但是它会占用很多资源，导致vivado布线更加困难，走线延迟可能变得很大，主频上不去。

我们在cache命中率上已经基本上做到极致，下一步需要继续优化，可能需要做指令的预取，这个是我们没有做的，可以尝试。

## 提升主频

在vivado下进行综合（Synthesis），布线（Implementation），然后打开Implementation的结果，选择Report Timing Summary，查看时序报告，里面会报告最长的路径。一般来说最长路径与访存、重定向有关，因为访存要经过cache，cache里面逻辑复杂，而重定向把多个段串了起来，也会很长。以我们优化过程的两点工作为例进行说明。

1. 最开始vivado报告的最长路径是从ALU出来再重定向到ID段的。ALU的大部分运算，比如加减法、逻辑运算，都是耗时很短的，最耗时的是乘法，所以乘法是最长路径。

重定向的目的是处理读写通用寄存器的冲突，所以只对需要写通用寄存器的指令有效，乘法的结果是写到HILO寄存器的，不需要写通用寄存器，所以刚好可以把乘法的结果从ALU的运算结果中剔除掉，专门生成一个不带乘法的ALU运算结果，用于重定向。一开始做功能测试的时候，没有考虑这么多，所以一股脑把ALU的结果直接送到了重定向模块，最后在性能测试的时候才发现不行。也就是说，在我们的重定向路径中，有一部分路径永远都不会被选择到，而它刚好就是最长路径，所以需要想办法去除这部分最长路径，vivado只会检测这条路径的延迟，而无法得知这条路径是不是要被用到。因为一条用不到的路径限制了主频是很不划算的。

在西北工业大学2017年的比赛中就有提到“小概率前递”这一名词，他们学校可能把重定向叫做前递，他们在进行性能优化时，把一部分很长、但是很少发生的重定向路径用插入气泡的方式解决，增加了一部分时钟周期，但是获得了更高的主频。而我们的情况就更好办了，我们根本就不需要插入气泡，因为我们的这条最长路径根本就没用。

除法为什么不是最长路径？除法本来也和乘法一样，很长，但是它被拆分成了32个周期，所以每个周期都不长，开发板上有专门的硬件用于乘法运算，而除法只能自己实现。

1. 扇出优化。这部分工作主要是林力韬做的，大概的意思是，一根线接到几百上千个位置，那么这根线的扇出太大，带不动，需要先把这根线分成两根，再分成四根，多次用二分法，增加级数，较小每一级的扇出。
2. 以上两点是我们提升主频的过程中做了的，还是不够的，只把主频提到了69MHz，有些学校达到了100MHz，肯定还有很多办法，比如，把五段流水线拆分成七段流水线。

# 建议

1. 使用mipsel-linux-objdump -d命令可以把操作系统内核反汇编，再自己写个C语言程序获取汇编代码中用到的MIPS指令，现在最新的Linux内核大概80多条指令，建议全部实现，初赛实现50多条，决赛再去加指令是比较麻烦的，一次性实现全部的更好。