# **复旦大学数字逻辑基础荣誉课程实验报告**

**课程设计名称**

图片包含 户外, 标牌

已生成极高可信度的说明

|  |  |
| --- | --- |
| 院 系： | 微电子学院 |
| 专 业： | 微电子科学与工程（本研贯通） |
| 姓 名： | 秦振航 杨远达 庄集 |
| 学 号： | 21307140085；21307140079；21307140047 |
| 完 成 日 期： | 2022 年 1 月 7 日 |

# 概述

## 设计目的

### 理解RISC-V的基本指令（包括I型、R型、S型、B型、J型、U指令），掌握指令的编码原理。

### 熟练Verilog HDL语言基本语法及模块化设计方法，并实现简单的RISC-V的五级流水线设计

### 培养动手能力及小组协作能力，为今后的学习生活打下良好基础。

## 功能描述

### 采用五级流水线结构，时钟频率工作在50Mhz

### 实现基本六种类型指令，并进行了乘除法指令集扩展

### 外设C语言自动编译为机器码，采用VGA显示

### 小组成员学习GitHub远程协作完成

## 小组分工

CPU核心部分由小组成员共同编写完成，其中取指、译码主要由庄集完成，执行由秦振航完成，访存、写回由杨远达完成，数据冒险和控制冒险由三人商议合作完成。此外，秦振航同学负责乘除法的扩展以及指令的仿真；杨远达负责外设架构、vga显示；庄集负责撰写实验报告。

# CPU设计

## 结构框图

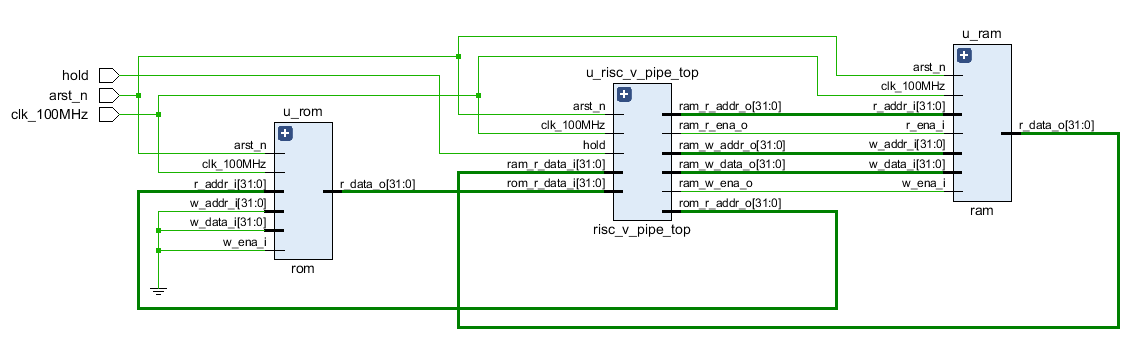
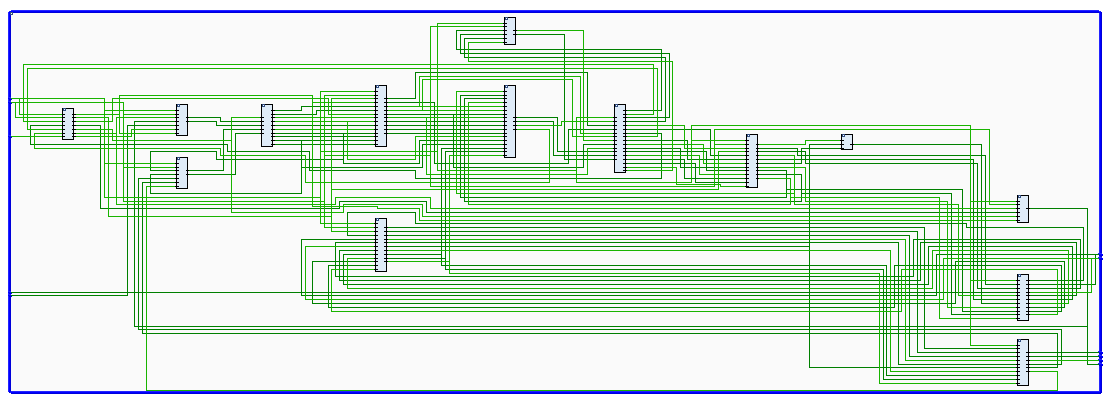


图1：CPU总体结构

图2：CPU内部模块

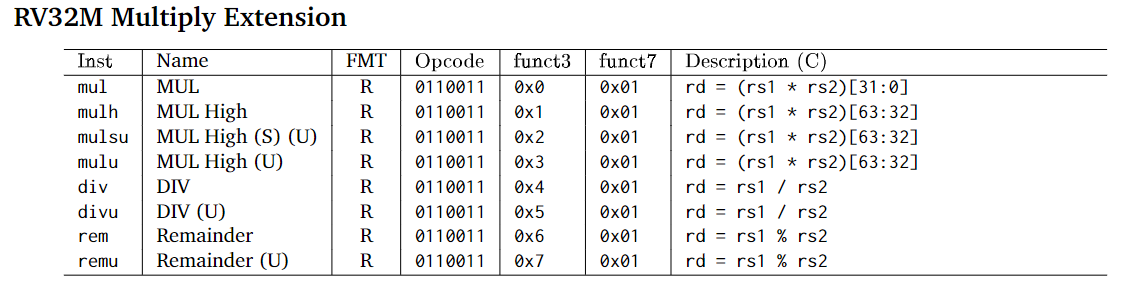
## 指令构架

### RV32I 指令集

与大多数指令集相比, RISC-V 指令集可以自由地用于任何目的, 允许任何人设计、制造和销售 RISC-V 芯片和软件. 虽然这不是第一个开源指令集, 但它具有重要意义, 因为其设计使其适用于现代计算设备（如仓库规模云计算机、高端移动电话和微小嵌入式系统）。设计者考虑到了这些用途中的性能与功率效率. 该指令集还具有众多支持的软件, 这解决了新指令集通常的弱点。

### 指令集索引

图3：RV32I指令集索引

图4：RV32M指令集索引

## CPU模块

CPU内部模块多且复杂，报告中仅展示rom和ram的输入输出端口，详细端口及分析见[readme.md](README.md)文档。

### 数据存储ram.v

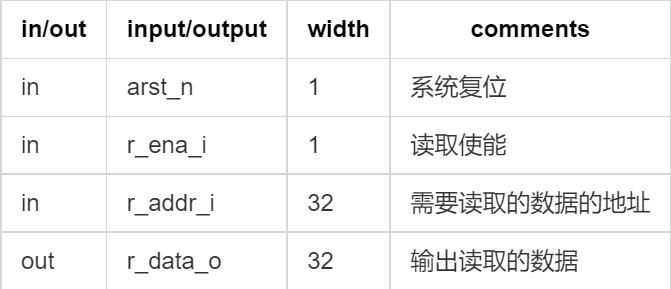


图5：ram.v端口

ram.v作为外设之一，其作用为存储读出的数据.

### 指令存储rom.v

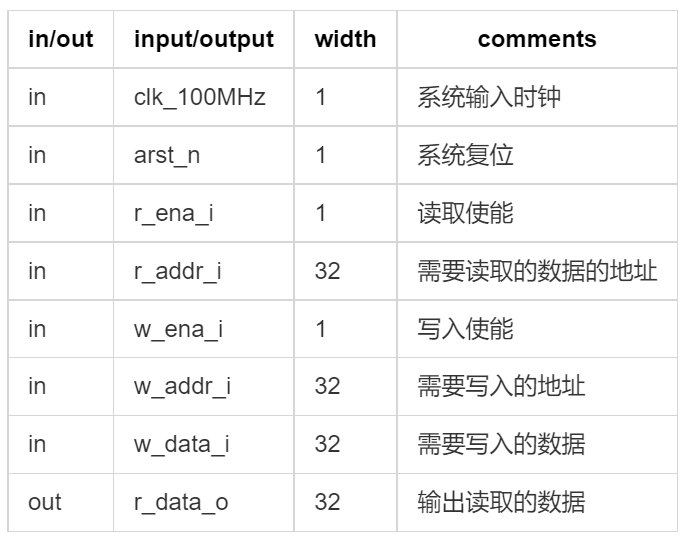


图6：rom.v端口

## ·代码规范

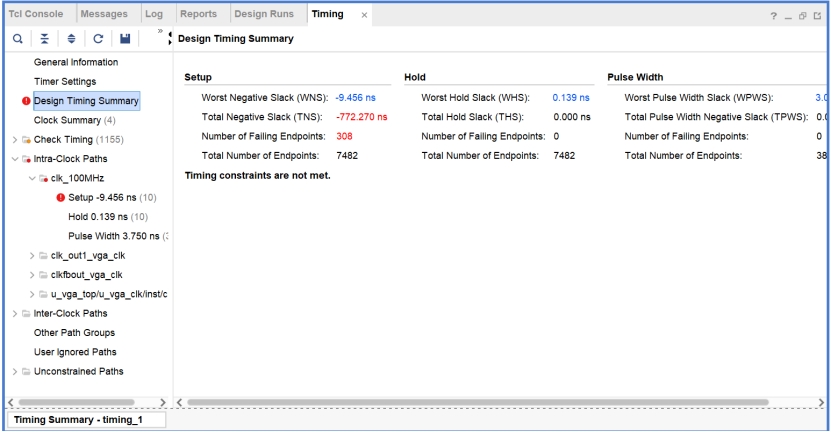
由于篇幅较长，详细请见<代码规范.md>

# .仿真结果

由于篇幅较大，详细仿真结果见<仿真.docx>

# 时序分析

## 100Mhz时序分析

当时序约束中设置时钟频率为100MHz时，从时序报告中可以看出此时的时序不满足。在不满足的路径中可以看到，不满足的路径主要发生在通用寄存器reg中。这是因为使用了共32个32位的通用寄存器，寄存器的读写逻辑稍微复杂。同时该cpu还实现了乘除法的功能，也限制了时钟的频率。具体的时序报告在timing\_report\_100中。

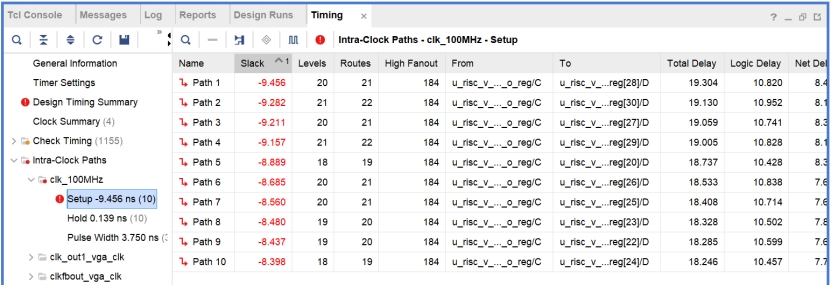
图7：100Mhz时序报告1

图8：100Mhz时序报告2

## 50Mhz时序分析

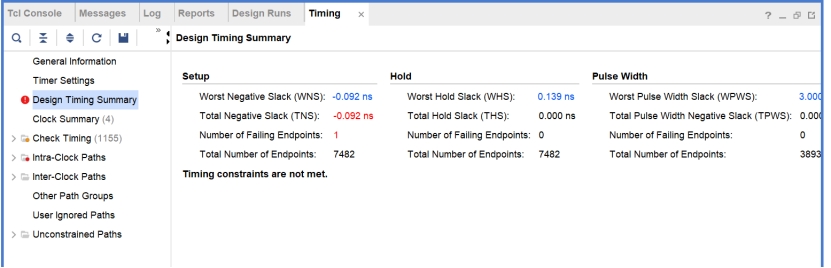
修改时钟约束为50MHz，得到的时序报告如下（见timing\_report\_50）。可以看见此时的时序几乎恰好是满足的。因此该cpu的最高频率约为50MHz。

图9：50Mhz时序报告

# 外设及功能演示

## c语言程序编译

该cpu支持通过c语言编写代码，进行编译后通过vivado的bram将生成的机器码输入到开发板上，以执行相应的功能。

### 编译操作流程

1. 将待编译的c语言程序放在RISC\_V\_PIPE\compile\compiler\_workspace目录下。

2. 运行该目录下的riscv\_compiler.bat脚本。输入需要编译的c程序的名称（不需要后缀）。可以看到在该目录下出现了新建的build文件夹，在该文件夹中可以看到生成的汇编文件（.s)、可执行文件(.o)以及机器码文件(.data)

3. 运行`RISC\_V\_PIPE\compile\compiler\_workspace`目录下的coe\_cond.exe文件。输入`build/(需要编译的c程序名称，不需要后缀)`，在build文件夹中就可以看到最终需要的机器码文件（.coe)

### 编译流程说明

下面通过一个简单的c语言代码说明编译的过程。该代码实现从1到50的求和。

int main(void)

{

int sum =0 ;

int a = 50;

for(int i=1;i<=a;i++)

sum += i;

return 0;

}

通过开源的riscv-none-embed-gcc工具，可以实现从c语言到riscv汇编，进而生成相应的机器码的过程。从c语言代码生成汇编，是通过riscv\_compiler.bat中的一行语句来实现：

riscv-none-embed-gcc -mabi=ilp32 -march=rv32i -S %var%%var\_c% -o %var\_build%%var%%var\_s%

这段脚本把当前目录下的sum.c文件编译成build文件夹下的汇编文件sum.s。生成的汇编代码sum.s如下

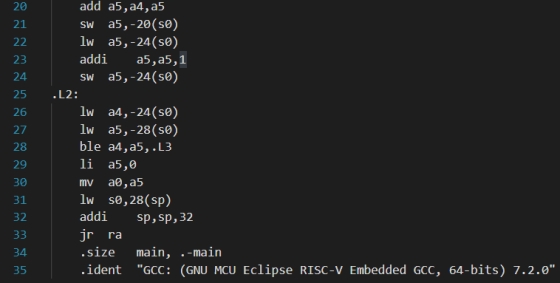
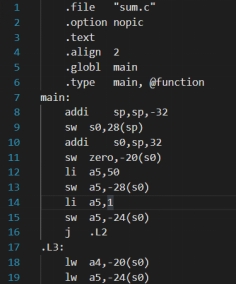


图10:生成的汇编代码

从汇编代码中也可以看出，变量sum的存储位置是-20。由于该cpu的外设ram的容量为64，因此最终sum的值存储在第59个字。

从汇编文件sum.s到机器码，是通过riscv\_compiler.bat中的脚本语句：

riscv-none-embed-as -mabi=ilp32 -march=rv32i %var\_build%%var%%var\_s% -o %var\_build%%var%%var\_o%

riscv-none-embed-objcopy -I elf32-littleriscv %var\_build%%var%%var\_o% -O verilog %var\_build%%var%%var\_data%

第一句将sum.s编译成对象文件sum.o。但是cpu无法识别该对象文件，因此再通过第二条语句生成机器码。生成的机器码如下图所示：

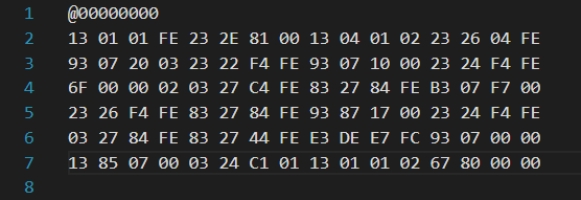


图11：机器码1

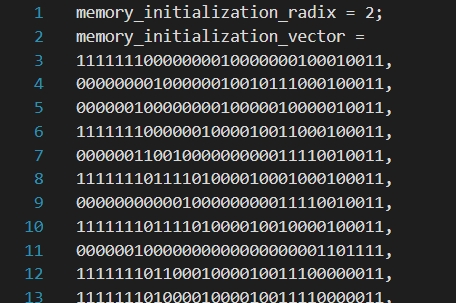
这里的机器码有很大的问题：第一，里面的指令是倒序的，第二，其格式不符合coe文件的要求。因此必须经过处理之后才能使用。通过使用c++写的小程序coe\_cond.exe，生成最后的可以直接导入到开发板上的机器码文件sum.coe，如下图所示。coe\_cond.exe的源码在\RISC\_V\_PIPE\compile\compiler\_tool\coe\_cond\coe\_cond.cpp中。

图12：机器码 2

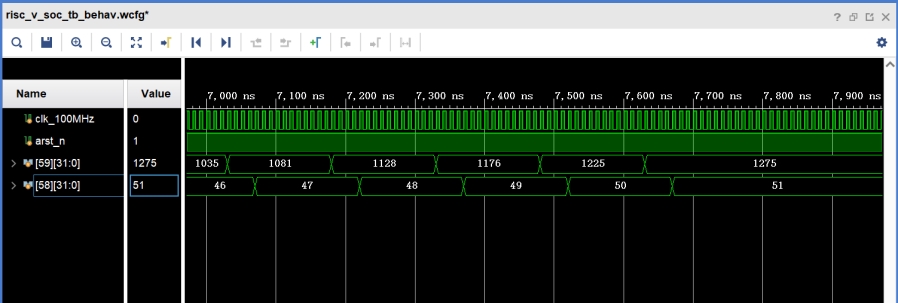
下面是该代码的仿真结果：

图13：仿真结果

可以看到ram的第59个字存储sum的值，第58个字存储当前计算到的数字。结果1275正是1到50的求和，仿真结果正确。

## 功能实现

下面，使用cpu来实现1到50的求和的功能并通过vga显示。

### c语言的编译。

首先编写1到50求和的c语言程序，并放在RISC\_V\_PIPE\compile\compiler\_workspace目录下。

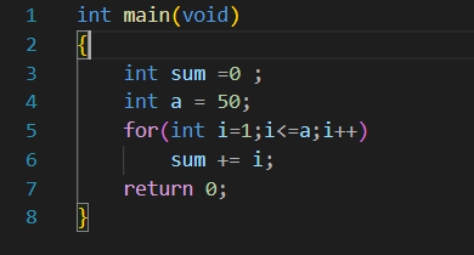


图14：c语言程序

然后运行目录下的riscv\_compiler.bat脚本，在脚本界面中输入“sum”。在该目录下新建的build文件夹中可以看到生成的汇编文件（sum.s)、可执行文件(sum.o)以及机器码文件(sum.data)。继续运行coe\_cond.exe可执行文件，输入“build/sum”，在build文件夹中得到可以导入bram中的sum.coe文件。

### 将代码数据导入开发板

修改项目中ip核code的数据来源为sum.coe。重新综合、实现，并导出数据流，下载到开发板上。

### 显示输出

通电前，开发板上的复位应处于低电平有效状态，暂停信号hold应处于高电平有效状态。通电后，拉上复位开关。这一瞬间，代码数据就被导入到cpu的rom模块中。

最后拨下暂停信号的开关，可以看到屏幕上显示了输出：10011111011。这就是十进制下的1275，也就是从1到50求和的结果。因此可以说明整个流程的功能都是能够正常运行

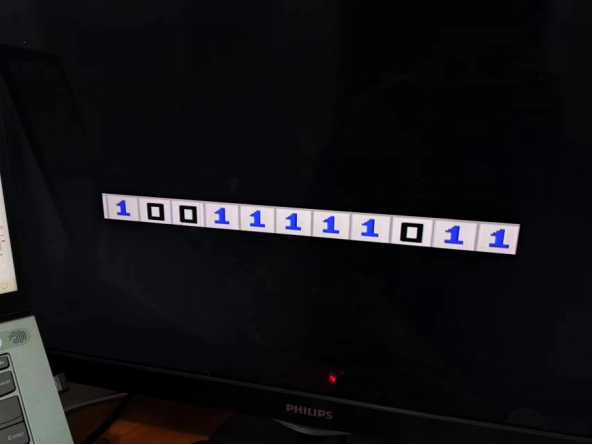


图14：输出结果

# 问题与解决

## 数据冒险

### 情况1

执行以下两条指令

addi x1,x0,1

addi x2,x1,1

当第二条指令处在译码阶段的时候，需要读取寄存器X1的值，但是此时第一条指令处于执行阶段，按照正常的流水线进度，需要等到第五个周期之后才能将X1的结果写回。因此这个时候读出来的X1的值是旧值，这是我们不希望出现的情况。

要想解决此问题，最容易的方法就是停顿流水线，等待上一条指令的结果写回寄存器之后，再对下一条指令译码。但是这样做会大大降低流水线执行指令的效率。

第二种方法是避免这种情况的出现，比如调整指令的顺序，在两条指令中间插入其他不相关的指令，就像上一篇文章中进行测试的指令一样，刻意地避开这种情况。但是这种方法指标不治本。

第三种方法，通过添加硬件来解决。

虽然第一条指令需要等第五个时钟周期之后才能写回寄存器，但是这条指令的执行结果在第三个时钟周期结束的时候就已经出来了。同样的，虽然第二条指令在第三个时钟周期就要读取X1的值，但是真正使用X1的值是在第四个时钟周期开始的时候。所以，我们可以打破常规，在第一条指令执行结束的时候（第三个周期末）就将待写回的X1的结果传递到第二条指令的执行阶段（第四个周期始）。这样就可以避免停顿并且不需要额外的调整指令的顺序。

### 情况2

执行以下三条指令

addi x1,x0,1

addi x2,x0,1

addi x3,x1,2

这种情况下，第一条和第二条指令能够正常运行，但是第三条指令出现了数据冒险。当第三条指令处于译码阶段的时候，需要读取X1的值，但是此时第一条指令处于访存的阶段，同样不能读取到最新的X1的值。

按照上面的分析方法，就需要在第一条指令访存阶段结束的时候（第四个周期末），将待写回的X1的结果传递到第三条指令的执行阶段开始的时候（第五个周期始）。

### 情况3

执行以下四条指令

addi x1,x0,1

addi x2,x0,1

addi x3,x0,2

addi x4,x1,3

这种情况下，前三条指令都能正常执行。当第四条指令处于译码阶段的时候，需要读取寄存器X1的值，但是此时第一条指令处但是并没有完全将寄存器X1的结果写回到寄存器，因为写回寄存器是时序的。

为此，有两种解决方法，一是采用下降沿进行写回，这样在第四条指令进入执行阶段前便可以将正确的数据读出来；二是在寄存器内部采用数据前递的思想，当写入数据地址与读数据地址相同时，将要写入数据的值直接赋给读数据的值。具体代码如下

always @(\*) begin

if (reg1\_r\_addr\_i == `ZERO\_REG) begin

reg1\_r\_data\_o = `ZERO\_WORD;

end

else if (reg1\_r\_addr\_i == w\_addr\_i && w\_ena\_i == `WRITE\_ENABLE) begin

reg1\_r\_data\_o = w\_data\_i;

end else begin

reg1\_r\_data\_o = regs[reg1\_r\_addr\_i];//把读寄存器的地址所对应的data取出来

end

end

### 情况4

执行以下指令

lw x1,0,x0

addi x2,x1,1

addi x3,x1,2

addi x4,x1,3

上面讨论的三种冒险的情况有一个共同点，就是在执行阶段结束的时候，带写回X1的结果就已经知道了，所以我们可以通过将数据前递解决冒险。

对于访存指令，访存指令的结果要在访存结束后才能知道。对第二条指令来说，第二条指令处在译码阶段的时候，第一条指令处在执行阶段，第二条指令在第三个时钟周期开始的时候（执行阶段）就需要使用最新的X1的值，而此时第一条指令才刚刚进入访存阶段，还没有来的及读出待写回X1的值。

所以这种情况下的数据前递是无效的，只能通过等待一个时钟周期，等第一条指令访存阶段结束后，再进行数据前递。对第三条和第四条指令，则不需要额外修改，之前的数据前递的方法仍然有效。

### 情况5

执行以下指令

lw x1,0,x0

sw x0,0,x1

lw x2,0,x0

sw x2,1,x0

由5.1.4得出结论，在发生加载——使用型冒险时，需要停顿一个时钟周期，而现在的情况又有所不同，因为load指令后面跟的是store指令，store指令的两个源寄存器rs1和rs2是有区别的，rs1寄存器里的数用来计算地址，在EX阶段使用，rs2寄存器里面的数据用来作为写入数据存储器的数，在MEM阶段使用。

因此当store指令的rs2与load指令的目标寄存器相同时，是不需要停顿的，只需要将数据前递即可。

## 控制冒险

控制冒险即在遇到条件分支指令时，由于跳转条件是否成立在执行阶段才知道，因此流水线就需要停顿或者冲刷指令才能正确运行。

针对控制冒险，解决方案可以分为三种

1.最简单粗暴的方法，每次遇到跳转指令就将流水线停顿，等目标地址计算出来之后，再取新的指令。这无疑会大大降低流水线的运行速度。

2.总是假设不跳转，就像上面分析的，遇到分支指令之后，先正常的去取指令，如果分支不发生，则流水线正常运行，如果发生跳转，则冲刷掉多取的两条指令。这便是本篇文章要解决的方案。

3.仍然假设不跳转，但是将分支地址的计算前递到译码阶段，这样在分支目标地址结果出来之前，仅仅会多余的取出一条指令，那么如果发生跳转，则只需要冲刷掉多取的这一条指令。

经讨论我们采用第二种方式，即正常取指，发生分支时冲刷流水线。从代码上看即在执行模块输出冲刷流水线信号给控制模块，控制模块分配给需要冲刷的模块。

# 实验心得

## 小组完成可以分三步走，第一步是所有人一同学习相关知识并讨论，第二部是明确分工后的自我独立探索，第三步是设计的整合及扩展讨论。

## 成员的讨论是必要的，但不能过于频繁，一个晚上的叨叨常不如一个人独立思考问题，讨论要侧重于信息的交互，对自己部分问题的思考尽量做到独立完成。

## 只完成自己的工作是不够的，完成一项大作业不仅是个人能力的成长，也是对团队协作的认识与实践，懂得如何充分发挥团队的积极性，运用团队的力量同样重要。