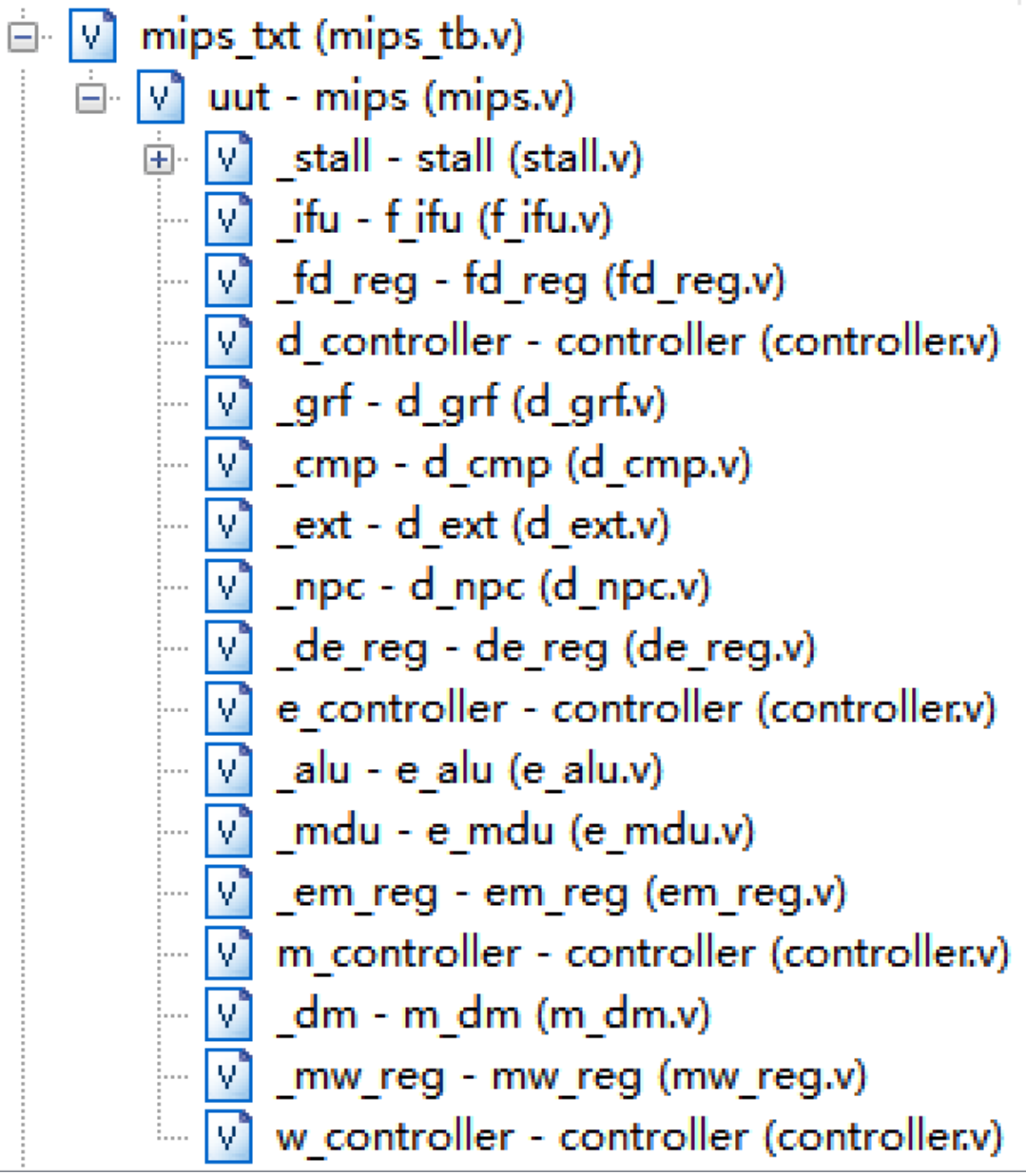
# P6设计文档

## 命名规则：

1. 主模块为mips.v，命名为mips
2. 特殊模块为stall和controller，其中stall命名为_stall，controller在stall中命名为“流水线层级_judgeStall”，在mips中命名为“流水线层级_controller”
3. 流水线寄存器有四个，命名为“流水线连续层级_reg“
4. 其余模块命名为”_模块名“

## 整体架构



## H2 相对P5的调整

1. 修改controller译码方式

   原本译码方式为用每一个指令得到各自的信号

   经过修改，改为首先为每一个指令对应到各自的类别中，在根据不同的类别得到信号

   类别共有11类

| 类别 | 信号 |
| --- | --- |
| load | lw lh lb |
| save | sw sh sb |
| alu_r | add sub and or slt sltu sll(nop) |
| alu_i | addi subi ori |
| md | mult multu div divu |
| mt | mthi mtlo |
| mf | mfhi mflo |
| j_b | beq bne |
| jal | jal |
| jr | jr |
| lui | lui |

2. 增加乘除槽

   具体修改信号在下文进行介绍

3. 修改ifu和dm端口连接方式

   具体修改信号在下文进行介绍

## Controller

采用了分布式译码的方式，最大化减少代码重复，增加效率

下面对每一个信号进行阐述

---

### judgeStall

| | load | save | alu_r | alu_i | md | mt | mf | j_b | jal | jr | lui |
|---|---|---|---|---|---|---|---|---|---|---|---|
| tuse_rs | 1 | 1 | 1 | 1 | 1 | 1 | \ | 0 | \ | 0 | \ |
| tuse_rt | \ | 2 | 1 | \ | 1 | \ | \ | 0 | \ | \ | \ |
| tnew | 3 | 0 | 2 | 2 | 0 | 0 | \ | 0 | 1 | 0 | 1 |
| mdu_busy | | | | | | | | | | | |
| mdu_start | | | | | | | | | | | |

#### tuse_rs

在几个时钟周期到来后会使用grf_rs

\为高阻态

#### tuse_rt

在几个时钟周期到来后会使用grf_rt

\为高阻态

#### tnew

在几个时钟周期到来后能够将写入写入寄存器的内容写入流水寄存器

只有当tuse>=tnew时，电路才能正常流水，否则就要阻塞

#### mdu_busy mdu_start

判断乘除槽当前是否被占用

若被占用，且当前d级指令为乘除槽相关指令，则暴力阻塞

---

### d_controller

| | load | save | alu_r | alu_i | md | mt | mf | j_b | jal | jr | lui |
|---|---|---|---|---|---|---|---|---|---|---|---|
| cmpOp | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 |
| nPcOp | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |
| extOp | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | 4 | 0 | 3 |

| | beq | bne |
|---|---|---|
| cmpOp | 1 | 2 |

#### cmpOp

当分支判断语句到来时，控制cmp模块中进行运算的类型，输出branch

| 0 | 1 | 2 |
|---|---|---|
| nope | == | != |

#### nPcOp

控制nPc模块跳转pc的类型

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| pc4 | pcImm16 | pcImm26 | pcReg |

#### extOp

控制ext模块中拓展立即数的类型，输出extImm

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| nope | signext | signext00 | sll16 | ext00 | zeroext |

注意：

1. signex00为16位offset的拓展，结果已经加上了pc+4
2. ex00为26为立即数的拓展，结果已经在前面复制了四位pc

### e_controller

| | load | save | alu_r | alu_i | md | mt | mf | j_b | jal | jr | lui |
|---|---|---|---|---|---|---|---|---|---|---|---|
| srcASel | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| srcBSel | 2 | 2 | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| aluOp | 1 | 1 | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| | add | sub | and | or | slt | sltu | addi | andi | ori | slll |
|---|---|---|---|---|---|---|---|---|---|---|
| aluOp | 1 | 2 | 5 | 3 | 6 | 6 | 1 | 5 | 3 | 4 |

| | load | save | alu_r | alu_i | md | mt | mf | j_b | jal | jr | lui |
|---|---|---|---|---|---|---|---|---|---|---|---|
| d1Sel | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| d2Sel | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| mduOp | 0 | 0 | 0 | 0 | | | 0 | 0 | 0 | 0 | 0 |

| | mult | multu | div | divu | mthi | mtlo |
|---|---|---|---|---|---|---|
| mduOp | 1 | 2 | 3 | 4 | 5 | 6 |

#### srcASel

控制alu中srcA的类型

| 0 | 1 |
|---|---|
| nope | grf_rs |

#### srcBSel

控制alu中srcB的类型

| 0 | 1 | 2 |
|---|---|---|
| nope | grf_rt | extImm |

#### aluOp

控制alu中进行运算的类型

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| nope | + | - | \| | << | & | < |

#### d1Sel

控制被乘数，被除数和mthi，mtlo来源

| 0 | 1 |
|---|---|
| nope | grf_rs |

#### d2Sel

控制乘数和除数来源

| 0 | 1 |
|---|---|
| nope | grf_rt |

#### mudOp

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| nope | mult | multu | div | divu | mthi | mtlo |

---

### m_controller

| | load | save | alu_r | alu_i | md | mt | mf | j_b | j_jal | j_jr | lui |
|---|---|---|---|---|---|---|---|---|---|---|---|
| memWrite | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| memOp | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| | lw | lh | lb | sw | sh | sb |
|---|---|---|---|---|---|---|
| memOp | 1 | 2 | 3 | 1 | 2 | 3 |

#### memWrite

dm写使能信号，1为写，0为不写

#### memOp

控制当前对dm进行读写操作的数据类型

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| nope | w | h | b |

---

### 转发相关信号

| | load | save | alu_r | alu_i | md | mt | mf | j_b | jal | jr | lui |
|---|---|---|---|---|---|---|---|---|---|---|---|
| regDstSel | 2 | 0 | 1 | 2 | 0 | 0 | 1 | 0 | 3 | 0 | 2 |
| regWdSel | 2 | 0 | 1 | 1 | 0 | 0 | | 0 | 4 | 0 | 3 |

| | mfhi | mhlo |
|---|---|---|
| regWdSel | 5 | 6 |

#### regDstSel

选择写入寄存器的位置来源

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0 | [15:11] | [20:16] | 31 |

注意：

1. 在controller中最后会直接输出regDst，而非选择信号
2. 当写入位置为0时，视为regWrite为0

#### regWdSel

选择写入寄存器的内容来源

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| nope | aluResult | memRd | extImm | pc4 | hi | lo |

唯一需要注意的是pc4在转发和写入时其实为pc+8，这是由延迟槽的性质决定的

## 测试思路

1. 随机出现指令，指令为[要求支持的指令集]中的一种
2. 使用c++编写单周期程序，保证测试程序合理性
3. 不断生成，以得到更到强度的程序
4. 对于剩余没有覆盖到的点，自己手动构造数据进行测试

自己跑出的一个点

```
standard pipeline-cycle: 879
slow pipeline-cycle: 1695
accepted cycle range: [715, 1450]
```

在自己构造数据测试时，我发现了cpu对beq在阻塞中的行为在一些特殊情况下有异常，并成功通过改变阻塞的位置解决的问题

## 思考题

#### 1、为什么需要有单独的乘除法部件而不是整合进 ALU？为何需要有独立的 HI、LO 寄存器？

这是由乘除法所需的特殊的时间周期决定的：乘除法需要消耗大量的时间，而且阻塞方式和普通的alu指令不同，所以将mud独立可以减少对其它指令的阻塞；因为HI、LO寄存器只在乘除槽中起作用，将其独立一方面形成了保护，避免被程序员随意修改，另一方面减少了不必要的阻塞，提高了cpu运行效率

#### 2、真实的流水线 CPU 是如何使用实现乘除法的？请查阅相关资料进行简单说明。

1. **乘法**：在流水线 CPU 中，乘法通常通过乘累加和乘累减指令来实现。这些指令在流水线执行阶段占用多个时钟周期，因此需要暂停流水线，以等待这些多周期指令执行完毕。
2. **除法**：除法运算通常采用"试商法"实现，对于32位的除法，至少需要32个时钟周期才能得到除法结果。
3. **流水线暂停**：由于乘法和除法指令在流水线执行阶段占用多个时钟周期，因此需要暂停流水线，以等待这些多周期指令执行完毕。

#### 3、请结合自己的实现分析，你是如何处理 Busy 信号带来的周期阻塞的？

这一点在前文已经有过简单的说明

具体来说，我增加了e_stall_mdu信号，它与e_stall_rs等信号发挥同样的作用

```
wire e_stall_mdu = (d_md || d_mt || d_mf) &&
(e_mdu_busy | e_mdu_start);
```

一旦e_stall_mdu为真，立刻阻塞流水线

#### 4、请问采用字节使能信号的方式处理写指令有什么好处？（提示：从清晰性、统一性等角度考虑）

1. 清晰性：采用字节使能减少了多种多样不同的写入写出实现方式带来的混乱，增加了程序的可读性、清晰性，一旦问题出现，可以方便地找到错误语句
2. 统一性：采用字节使能使得w、h、b和其他可能的内存写入写出信号有了统一的表达方式，其最根本的思想是将默认可读写的最小长度为字节

#### 5、请思考，我们在按字节读和按字节写时，实际从 DM 获得的数据和向 DM 写入的数据是否是一字节？在什么情况下我们按字节读和按字节写的效率会高于按字读和按字写呢？

不是，在我们想要写入字节时，实际上是先读出dm中该单元存储的一个字的数据，然后将我们需要修改的部分通过组合逻辑修改上去，最后再将一个字整体写回；当我们需要单次修改多个字节的内容，如sw和sh时，将字整体写回将大大提高系统效率

#### 6、为了对抗复杂性你采取了哪些抽象和规范手段？这些手段在译码和处理数据冲突的时候有什么样的特点与帮助？

见“命名规则”板块和“controller”板块

我的整体思想是：可修改性>时间效率>可读性

#### 7、在本实验中你遇到了哪些不同指令类型组合产生的冲突？你又是如何解决的？相应的测试样例是什么样的？

见“测试思路”板块

#### 8、如果你是手动构造的样例，请说明构造策略，说明你的测试程序如何保证覆盖了所有需要测试的情况；如果你是完全随机生成的测试样例，请思考完全随机的测试程序有何不足之处；如果你在生成测试样例时采用了特殊的策略，比如构造连续数据冒险序列，请你描述一下你使用的策略如何结合了随机性达到强测的效果。

见“测试思路”板块