流水线处理器设计文档

# 模块规格

## Instruction Fetch Unit（取指令单元）

### 端口定义

表1 IFU的端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 方向 | 端口规格 | 功能描述 |
| sign\_imm32 | input | [31:0] | 传入符号扩展后的offset，以适应beq指令 |
| addr26 | input | [25:0] | 传入j型指令的26位addr，以适应j型指令 |
| ra32 | input | [31:0] | 传入寄存器读出的32位addr，以适应jr指令 |
| PC\_choice | input | [1:0] | 传入PC的四种转移规则选择信号 |
| clk | input | [0:0] | 时钟信号 |
| reset | input | [0:0] | 同步复位信号，驱动PC同步复位成0x3000 |
| FREEZE | input | [0:0] | **PC计数器冻结信号**，为1时PC不可变。 |
| IR | output | [31:0] | 当前指令，即将放进IR/ID流水线寄存器中 |
| PC+4 | output | [31:0] | 下一条指令的地址，以适应jal指令 |

### 功能说明

IF中由PC计数器、PC转移器、指令存储器等组成。

PC的初始值为0x00003000。当时钟上升沿来临时，将其更新为PC转移器所计算出的下一个PC值。当reset信号有效且上升沿时，将PC计数器同步复位为0x00003000值。PC计数器的32bit输出经过截取后传给指令存储器，成为读取指令的10bit地址。

当**FREEZE**有效时，PC计数器不更新，IR信号不更新，便于实现STALL（暂停）。

PC转移器根据当前的PC状态、若干不同种类的立即数分别计算出顺序执行、分支执行、跳转执行、跳转寄存器执行的下一个PC值，并由PC\_choice信号选择其中一路成为PC的下一个状态：

PC\_choice = 0时，对应顺序执行：PC’ = PC + 4；

PC\_choice = 1时，对应beq成立：PC’ = PC + 4 + sign\_imm32\*4；

PC\_choice = 2时，对应j型指令：PC’ = PC[31:28]||addr26||00；

PC\_choice = 3时，对应jr指令： PC’ = ra32。

指令存储器从PC计数器截取10bit指令地址，从1024\*32bit的ROM中读取32位指令并输出。

总体来说，取指令单元由时钟信号和复位信号控制，在每一个上升沿输出待执行的机器码指令。指令的执行顺序由PC计数器控制，PC的次态由当前态、立即数、指令的类型和冻结信号FREEZE决定。

## General Register File（通用寄存器堆）

### 端口定义

表2 GRF的端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 方向 | 端口规格 | 功能描述 |
| A1 | input | [4:0] | 读取地址为A1 |
| A2 | input | [4:0] | 读取地址为A2 |
| A3 | input | [4:0] | 写入32bit数据的目的地址A3 |
| WD | input | [31:0] | 写入到A3的32bit数据 |
| clk | input | [0:0] | 时钟信号 |
| reset | input | [0:0] | 同步复位信号 |
| WE | input | [0:0] | 写使能信号。为1时可写，为0时忽略WD |
| RD1 | output | [31:0] | 从A1读取的32bit数据 |
| RD2 | output | [31:0] | 从A2读取的32bit数据 |

### 功能说明

GRF由具有写使能的32个寄存器组成，每个寄存器均为32bit，均初始化为0，其中0号寄存器（地址为5bit 00000）始终为常数0，忽略对其的任何修改。

其余寄存器由5位地址编码，从00000到11111。在每时每刻，A1传入的地址对应的寄存器的32bit值输出到RD1中，A2传入的地址对应的寄存器的32bit值输出到RD2中。

在时钟clk处于上升沿且写使能WE=1时，寄存器将当前时刻地址为A3的寄存器中写入32bit数据WD。

当同步复位信号reset=1时，所有寄存器在clk上升沿复位成0值。

总体来说，GRF的两个读取操作（RD1 = reg[A1], RD2 = reg[A2]）在每时每刻都生效，GRF的一个写入操作（reg[A3] = WD）当且仅当clk为上升沿且WE=1时生效。

GRF中存在**内部转发机制**：当A3 ==A1(A2)时且此时WE=1，则直接将即将写入的WD转发至输出RD1(RD2)端（除$0外）。

## Arithmetic Logical Unit（算术逻辑单元）

### 端口定义

表3 ALU的端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 方向 | 端口规格 | 功能描述 |
| A | input | [31:0] | 运算对象1 |
| B | input | [31:0] | 运算对象2 |
| ALUControl | input | [2:0] | ALU运算类型控制标记，支持5种运算 |
| C | output | [31:0] | 由对象1和2得到的运算结果 |

### 功能说明

该模块为纯组合逻辑，与时钟无关。

本ALU实现暂不检测溢出。

当ALUControl = 010时，C = A + B（简单二进制加法，不考虑溢出）。

当ALUControl = 110时，C = A - B（简单二进制减法，不考虑溢出）。

当ALUControl = 000时，C = A & B（按位与）。

当ALUControl = 001时，C = A | B（按位或）。

当ALUControl = 111时，C = A < B（小于置位，若A小于B，则C置为1；否则C置为0）。

当ALUControl = 110时，C = B，便于**与ori一致地实现lui**指令。此时B即为Extender输出的lui\_extend信号，直接输出到C端即可。

由于**beq指令前移**，故不再需要Zero信号。

## Data Memory（数据内存）

### 端口定义

表4 DM的端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 方向 | 端口规格 | 功能描述 |
| Addr | input | [31:0] | 读取/写入的目标地址，按字节寻址 |
| WD | input | [31:0] | 待写入的32bit数据 |
| clk | input | [0:0] | 时钟信号 |
| reset | input | [0:0] | 同步复位信号，驱动清零内存 |
| WE | input | [0:0] | 写使能信号。为1时可写，为0时忽略WD |
| RD | output | [31:0] | 从地址Addr读取出的32bit数据 |

### 功能说明

DM由1024个字段组成，每个字段均为32bit，均初始化为0。

DM在同一时刻只能有意义地支持读取和写入其中一种操作。

当WE = 0时，截取Addr的[11:2]位得到按字寻址的地址，读取该地址，从RD输出。此为读取。

当WE = 1时，若clk处于上升沿，则更新Addr对应的字段数据为WD中获取的数据。下一时刻clk上升沿过去，RD立即读取Addr的数据。此为写入。

当同步复位信号reset=1时，所有字段在clk上升沿复位成0值，回到初始状态。

总体来说，DM的读取操作（RD1= mem[Addr]）在WE=0时生效，DM的写入操作（mem[Addr] = WD）当且仅当clk为上升沿且WE=1时生效。

## Immediate Calculator（立即数计算器）

### 端口定义

表5 ImmCalc的端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 方向 | 端口规格 | 功能描述 |
| imm16 | input | [15:0] | I指令中的16位立即数 |
| sign\_imm32 | output | [31:0] | 符号扩展后的立即数 |
| zero\_imm32 | output | [31:0] | 零扩展后的立即数 |
| lui\_imm32 | output | [31:0] | 将imm16加载至高16位后的立即数 |

### 功能说明

I指令的16位立即数可能作为内存操作指令lw/sw的带符号偏移量、ori的无符号参数或是lui的参数等。它们分别需要对立即数进行符号扩展、零扩展、加载至高位的操作。

立即数计算器ImmCalc获取一个16bit立即数imm16，并将其符号扩展、零扩展、加载至高位的结果sign\_imm32、zero\_imm32、lui\_imm32分别输出。

## Controller（控制器）

### 端口定义

表6 Controller的端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 方向 | 端口规格 | 功能描述 |
| opcode | input | [5:0] | 32bit指令中的opcode段 |
| funct | input | [5:0] | 32bit指令中的funct段 |
| MemtoReg | output | [0:0] | 为0时，GRF接收ALU运算结果；  为1时，GRF接收DM读取结果。 |
| MemWrite | output | [0:0] | DM的写使能信号 |
| Branch | output | [0:0] | 优先级低于跳转指令相关信号。非跳转指令时，Branch为0时，PC的变换规则为PC+4；Branch为1时，若beq的两个源寄存器数值相等，则PC的变换规则为PC+4+Imm\*4，否则为PC+4。 |
| ALUControl | output | [2:0] | ALU的运算控制信号 |
| ALUSrc | output | [1:0] | ALU的运算对象2的来源。 |
| RegDst | output | [0:0] | GRF的写入对象。为0时为rt；为1时为rd |
| RegWrite | output | [0:0] | GRF的写使能信号 |
| Jump | output | [0:0] | 当JR=0时，Jump为0时，PC转移由Branch决定；Jump为1时，按照26位立即数的拼接法转移PC。 |
| Link | output | [0:0] | 为0时，无特殊操作；  为1时，将PC+4存储到$ra中。 |
| JR | output | [0:0] | 为0时，无特殊操作；  为1时，令PC’ = ra32。 |

### 功能说明

Controller特异性地确定各个控制信号。

MemtoReg控制GRF的写入数据是ALU的运算结果C，或是DM的读取结果RD。

MemWrite控制DM是否写使能。

Branch决定当前命令除跳转指令外，是不是分支指令。

ALUControl控制ALU的运算。

ALUSrc控制ALU的第二个运算对象的来源。即ALU的B可能接收GRF的读出，也可能接收立即数的符号扩展、零扩展、加载高位。

RegDst控制GRF写入的目的地址A3。可能向rt寄存器写入，也可能向rd寄存器写入。

RegWrite控制GRF是否写使能。

Jump、Link、JR共同决定了j、jal、jr的PC变化和存储特性，为PC的转移增加了两条规则。

具体地，如何根据某条指令的机器码和其意义设计控制信号，见下文“控制器设计”中阐述。

# 数据通路设计

## Pipeline各级概览

表7 流水线各级组成

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **IF** |  | **ID** |  | **EX** |  | **MEM** |  | **WB** |
| PC | IFU |  | rRF |  | ALU |  | DM |  | wRF |
|  |  |  | EXT |  |  |  |  |  |  |
|  |  |  | CMP |  |  |  |  |  |  |
|  |  |  | CTRL |  |  |  |  |  |  |
|  |  | **IR** |  | **IR** |  | **IR** |  | **IR** |  |
|  |  |  |  | A3 |  | A3 |  | A3 |  |
|  |  |  |  | A2 |  | A2 |  |  |  |
|  |  |  |  | V2 |  | V2 |  |  |  |
|  |  |  |  | A1 |  | AO |  | AO |  |
|  |  |  |  | V1 |  |  |  | DR |  |
|  |  |  |  | E32\*3 |  |  |  |  |  |
|  |  | PCp4 |  | PCp4 |  | PCp4 |  | PCp4 |  |
|  |  |  |  | RegW |  | RegW |  | RegW |  |
|  |  |  |  | Mem2R |  | Mem2R |  | Mem2R |  |
|  |  |  |  | MemW |  | MemW |  | Link |  |
|  |  |  |  | ALUCtrl |  | Link |  |  |  |
|  |  |  |  | ALUSrc |  |  |  |  |  |
|  |  |  |  | Link |  |  |  |  |  |
|  |  | **IF/ID** |  | **ID/EX** |  | **EX/MEM** | | **MEM/WB** | |

## 数据通路设计表格

将上述Pipeline各级概览中的每一个元件作为每一行，将每一条指令作为每一列，得到Datapath设计表格。

详见 ./数据通路数据表格.xlsx 。

其中看到，A3@ID/EX、B@ALU、PC@IFU、regWD@wRF存在不同的连线方式，故需要建立**四个普通MUX**：

表8 普通多选器

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| MUX名称 | 0 | 1 | 2 | 3 |
| M\_PC | PC+4 | NPC.branch | NPC.jump | *MF\_PC\_JRra* |
| M\_IDEX\_A3 | IR[rt] | IR[rd] | 31 |  |
| M\_ALU\_B | *MF\_ALU\_B* | E32 |  |  |
| M\_wRF\_WD | AO | DR | PCp4 |  |

其中需要JRra、V2@ID/EX的值的地方，均可能存在转发接受点，故使用转发MUX代替。

由转发分析可知，**五个转发MUX**如下：

表9 转发多选器

|  |  |  |
| --- | --- | --- |
| MF名称 | 0 | 1~6 |
| *MF\_PC\_JRra* | rRF.RD1 | *转发值* |
| *MF\_CMP\_A* | rRF.RD1 | *转发值* |
| *MF\_CMP\_B* | rRF.RD2 | *转发值* |
| *MF\_ALU\_A* | V1@ID/EX | *转发值* |
| *MF\_ALU\_B* | V2@ID/EX | *转发值* |

其中转发值有6个来源：

表9续 转发多选器

|  |  |
| --- | --- |
| 1 | DR@MEMWB |
| 2 | AO@MEMWB |
| 3 | AO@EXMEM |
| 4 | PCp4\_MEM/WB +4 |
| 5 | PCp4\_EX/MEM +4 |
| 6 | PCp4\_ID/EX +4 |

其中**转发优先级**由高到低为：ID/EX > EX/MEM > MEM/WB。

## Data Hazard分析

表10 Tuse-Tnew表格

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | IR@ID/EX | | | IR@EX/MEM | | | IR@MEM/WB | | |
|  |  | ALU型 | DM型 | PC型 | ALU型 | DM型 | PC型 | ALU型 | DM型 | PC型 |
| Tuse  @IF/ID |  | 1 | 2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | S | S | F | F | S | F | F | F | F |
| 1 |  | S | F | F |  | F | F | F | F |
| 2 |  |  |  |  |  |  |  |  |  |

其中Tuse表格如下：

表11 Tuse 表格

|  |  |  |
| --- | --- | --- |
|  | rs | rt |
| beq | 0 | 0 |
| jr | 0 |  |
| cal\_R | 1 | 1 |
| cal\_I – lui | 1 |  |
| lui |  |  |
| lw | 1 |  |
| sw | 1 | 2 |
| jal |  |  |

Tnew表格如下：

表12 Tnew 表格

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ID/EX | EX/MEM | MEM/WB | 类型 |
| beq | / | / | / | NW |
| jr | / | / | / | NW |
| cal\_R | 1 | 0 | 0 | ALU |
| cal\_I – lui | 1 | 0 | 0 | ALU |
| lui | 1 | 0 | 0 | ALU |
| lw | 2 | 1 | 0 | DM |
| sw | / | / | / | NW |
| jal | 0 | 0 | 0 | PC |

由Tuse – Tnew表格可以看出**转发（F）**和**暂停（S）**的策略。

策略由IF/ID处的IR（和操作寄存器），和后面各级流水寄存器的IR（和操作寄存器）共同决定。

注意，不转发尝试写$zero的值。

当**转发**时，将转发多选器的选择信号设置为对应来源的信号。

当**暂停**时，冻结PC计数器、冻结IF/ID流水寄存器、清零ID/EX流水寄存器。

# 控制器设计

## 真值表设计

此处的真值表指的是对于每一个(opcode, funct)的组合，都有一组确定的控制信号的组合。不同指令的不同控制信号组合并起来就成为了一张真值表。

支持指令集{addu, subu, ori, lw, sw, beq, lui, nop, j, jal, jr}的一张可能的真值表如下：

表13 指令-控制信号真值表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct | 100001 | 100011 | / | / | / | / | 000000 |
| opcode | 000000 | 000000 | 001101 | 100011 | 101011 | 001111 | 000000 |
|  | **addu** | **subu** | **ori** | **lw** | **sw** | **lui** | **nop** |
| **MemtoReg** | 0 | 0 | 0 | 1 | x | 0 | 0 |
| **MemWrite** | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| **Branch** | 0 | 0 | 0 | 0 | 0 | 0 | x |
| **ALUControl** | 010 | 110 | 001 | 010 | 010 | 010 | xxx |
| **ALUSrc** | 00 | 00 | 10 | 01 | 01 | 11 | xx |
| **RegDst** | 1 | 1 | 0 | 0 | x | 0 | x |
| **RegWrite** | 1 | 1 | 1 | 1 | 0 | 1 | x |
| **Jump** | 0  0  0 | | | | | | |
| **Link** |
| **JR** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| funct | / | / | 001000 | / |
| opcode | 000011 | 000010 | 000000 | 000100 |
|  | **jal** | **j** | **jr** | **beq** |
| **MemtoReg** | 0 | 0 | 0 | x |
| **MemWrite** | 0 | 0 | 0 | 0 |
| **Branch** | 0 | 0 | 0 | 1 |
| **ALUControl** | 000 | 000 | 010 | 110 |
| **ALUSrc** | 00 | 00 | 00 | 00 |
| **RegDst** | 0 | 0 | 0 | x |
| **RegWrite** | 1 | 0 | 0 | 0 |
| **Jump** | 1 | 1 | 0 | 0 |
| **Link** | 1 | 0 | 0 | 0 |
| **JR** | 0 | 0 | 1 | 0 |

## 控制信号设计规则

### MemtoReg

MemtoReg当且仅当指令为lw/lb/lh时，才使MemtoReg=1。

### MemWrite

MemWrite当且仅当指令为sw/sb/sh时，才使MemWrite=1。

### Branch

所有可能按照PC’ = PC+4+Imm\*4规则跳转的指令，Branch都应为1。

### ALUControl

ALUControl对于运算型指令，直接设为其指令意义对应的运算标志。

对于需要计算地址的lw/sw指令，由于有addr+offset的计算需求，故设为“+”运算对应的标志。

对于lui指令，由于rs读出永远为$zero = 32’b0，则可以设置为or/add。

### ALUSrc

考察其指令的具体意义，可以判断其立即数需要符号扩展、零扩展还是加载到高位，由此设置ALUSrc。

### RegDst

RegDst表示存入寄存器的地址为rt还是rd。对于R型指令，目的寄存器为rd；对于I型指令，目的寄存器为rt。

### RegWrite

RegWrite表示是否GRF写使能。对于j、beq、jr、sw、nop等不操作寄存器的指令，RegWrite=0。

### Jump

Jump直接表示了是否为跳转指令。因此所有应该按照PC’ = addr32规则跳转的指令，Jump都应为1。其余指令的Jump均应该为0。

### Link

Link直接决定了GRF参与储存的地址和数据是正常结果，还是$31和PC+4。当且仅当jal指令Link信号为1。

### JR

JR直接决定了PC’的第四种转移规则，故当且仅当jr指令JR信号为1。

## 控制器设计：从机器码到控制信号

对于某一条指令，其意义和控制信号被其opcode（和funct）唯一确定。

故控制器可由指令的识别和信号的生成两部分构成。

### 指令的识别

以lw指令的opcode ”100011”为例，采用以下的与门进行特异性识别指令：

一个lw识别器应当其opcode=100011时输出1，其他时候输出0。因此构造以下结构：

assign lw\_Detector = op[5] & !op[4] & !op[3] & !op[2] & op[1] & op[0];

对于R指令，其识别器应针对其funct码类似构造。

### 信号的生成

由真值表我们知道，lw对应的信号可合并为1000100101。因此可构造如下信号生成器：

assign lw\_Signal = SignExtend(lw\_Detector) & 10’b1000100101;

这样，对于每一个指令就有一个10位Signal信号。它们其中有且只有一个为目的信号，其余的均为10位0。这样，构造一个或门即可得到最终信号：

assign Signal = lw\_Signal | sw\_Signal | beq\_Signal |…;

# 测试CPU

## 测试代码

测试指令集：{addu, subu, ori, lw, sw, beq, lui, nop, j, jal, jr}

测试MIPS代码：

按照以下表格构造测试用例：（箭头前为时间先，箭头后为时间后）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | R | I | load | jal |
| R | R -> R | I -> R | lw -> R | jal -> R |
| I | R -> I | I -> I | lw -> I | jal -> I |
| beq | R -> beq | I -> beq | lw -> beq | jal -> beq |
| jr | … | … | … | … |
| lw/sw | … | … | … | … |

其中每一个方框中有四（二）个测试用例，分别为两条指令紧挨、两条指令间插入无关指令，和对上述两个测试用例的冲突寄存器rs / rt互换。

ori $2, $0, 1

ori $3, $0, 2

sw $2, 0($0)

sw $3, 4($0)

jal\_exec:

jr $ra

ori $ra, $0, 0

**#################### R after \***

**# R - R**

subu $1, $2, $3

addu $4, $1, $2

*（省略互换）*

*（省略插入无关指令）*

*（省略插入无关指令互换）*

**# I - R**

ori $1, $2, 1000

addu $4, $2, $1

*（省略互换）*

*（省略插入无关指令）*

*（省略插入无关指令互换）*

**# load - R**

lw $1, 0($0)

addu $2, $1, $3

*（省略互换）*

*（省略插入无关指令）*

*（省略插入无关指令互换）*

**# jal - R**

jal jal\_exec

addu $2, $ra, $3

*（省略互换）*

*（省略插入无关指令）*

*（省略插入无关指令互换）*

**#################### I after \***

**# R - I**

subu $1, $2, $3

ori $4, $1, 2

*（省略插入无关指令）*

**# I - I**

ori $1, $2, 1000

ori $4, $1, 2

*（省略插入无关指令）*

**# load - I**

lw $1, 0($0)

ori $1, $3, 2

*（省略插入无关指令）*

**# jal - I**

jal jal\_exec

ori $1, $ra, 2

*（省略插入无关指令）*

**#################### beq after \***

lui $1, 55

ori $2, $0, 1

ori $3, $0, 0

sw $2, 0($0)

t1: **# R - beq**

subu $1, $2, $3

beq $1, $2, beq\_act1

lui $1, 55

t2:

*（省略互换）*

t3:

*（省略插入无关指令）*

t4:

*（省略插入无关指令互换）*

t5: **# I - beq**

ori $1, $2, 0

beq $1, $2, beq\_act5

lui $1, 55

t6:

*（省略互换）*

t7:

*（省略插入无关指令）*

t8:

*（省略插入无关指令互换）*

t9: **# load - beq**

lw $1, 0($0)

beq $2, $1, beq\_act9

lui $1, 55

t10:

*（省略互换）*

t11:

*（省略插入无关指令）*

t12:

*（省略插入无关指令互换）*

t13: **# jal - beq**

jal jal\_exec

beq $2, $ra, beq\_act13

t14:

*（省略互换）*

t15:

*（省略插入无关指令）*

t16:

*（省略插入无关指令互换）*

beq\_act1:

j t2

lui $31, 2333

*（省略beq\_act2 ~ beq\_act15）*

beq\_act16:

j end

lui $31, 2333

end:

**#################### jr after \***

*（类似，省略）*

**#################### lw/sw after \***

*（类似，省略）*

## 预期结果

与MARS行为完全一致，计算型指令均计算正确，16个beq都成功执行，每一次使用$ra都拿到重新link后的值。

# 思考题

## 单独测试Stall

先列出暂停的所有情况：

IF/ID处的指令为beq / jr，ID/EX处的指令为calcR / calcI / lw。

IF/ID处的指令为beq / jr，EX/MEM处的指令为lw。

IF/ID处的指令为calcR / calcI / lw / sw，ID/EX处的指令为lw。

然后在Stall Detector的Test Bench中分别将IR@IF/ID、IR@ID/EX、IR@EX/MEM设为全部可能的指令{beq, jr, addu, subu, ori, lui, lw, sw, jr}（同时保证他们的读写寄存器有的冲突有的不冲突），之后观察输出波形的值，与上述是否对应。

表14 暂停检测器的Test Bench设计

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ID/EX | lw | lw | lw | lw | lw | lw | lw | lw |
| IF/ID | beq\*2 | 冲突  c\_R\*2\*2 | 不冲突  c\_R\*2 | 冲突  ori | 不冲突  lui | lw | sw | jr |
| Stall | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| ID/EX | c\_R\*2 | c\_R\*2 | c\_R\*2 | c\_R\*2 | c\_R\*2 | c\_R\*2 | c\_R\*2 | c\_R\*2 |
| IF/ID | beq\*2 | 冲突c\_R\*2\*2 | 不冲突c\_R\*2 | 冲突  ori | 不冲突  lui | lw | sw | jr |
| Stall | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| ID/EX | c\_I\*2 | c\_I\*2 | c\_I\*2 | c\_I\*2 | c\_I\*2 | c\_I\*2 | c\_I\*2 | c\_I\*2 |
| IF/ID | beq\*2 | 冲突c\_R\*2\*2 | 不冲突c\_R\*2 | 冲突  ori | 不冲突  lui | lw | sw | jr |
| Stall | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| EX/ME | lw | lw | lw | lw | lw | lw | lw | lw |
| IF/ID | beq\*2 | 冲突c\_R\*2\*2 | 不冲突c\_R\*2 | 冲突  ori | 不冲突  lui | lw | sw | jr |
| Stall | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

## 测试Forwarding

在完成Stall检测器的测试后，完成转发检测及转发控制。

之后按照“测试CPU”一节中的方式一起枚举、测试冲突。此时若全部通过则说明Forwarding的实现也是正确的。