流水线 CPU 设计文档

一、CPU 设计方案综述

（一）总体设计概述

本 CPU 为 Logisim 实现的流水线 MIPS - CPU，支持的指令集包含{addu, subu, ori, lui, lw, sw, beq, j, jal, jr , nop}。

Add Unsigned Word **ADDU**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	ADDU 100001
6	5	5	5	5	6

Format: `ADDU rd, rs, rt` **MIPS32**

Purpose:

To add 32-bit integers

Description: `GPR[rd] ← GPR[rs] + GPR[rt]`

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

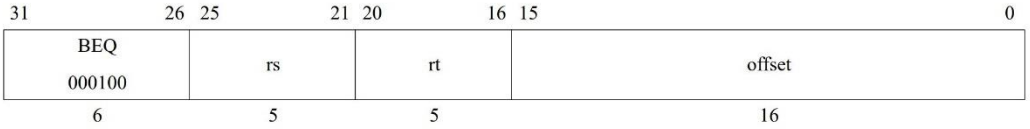
Restrictions:

None

Operation:

```
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← temp
```

Branch on Equal — BEQ



Format: `BEQ rs, rt, offset` **MIPS32**

Purpose:

To compare GPRs then do a PC-relative conditional branch

Description: `if GPR[rs] = GPR[rt] then branch`

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are equal, branch to the effective target address after the instruction in the delay slot is executed.

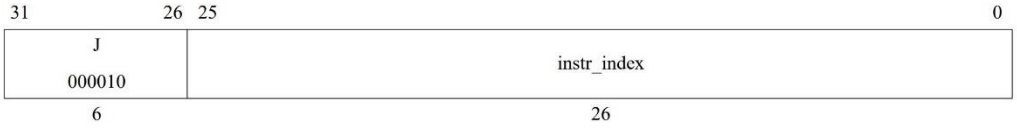
Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:     target_offset ← sign_extend(offset || 0^2)
          condition ← (GPR[rs] = GPR[rt])
I+1:   if condition then
          PC ← PC + target_offset
       endif
```

Jump — J



Format: `J target` **MIPS32**

Purpose:

To branch within the current 256 MB-aligned region

Description:

This is a PC-region branch (not PC-relative); the effective target address is in the “current” 256 MB-aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

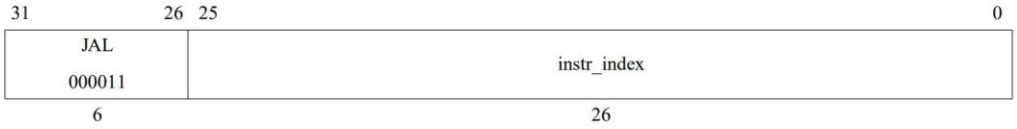
Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:
I+1:PC ← PC_GPRLEN-1..28 || instr_index || 0^2
```

Jump and Link — JAL



Format: `JAL target` **MIPS32**

Purpose:

To execute a procedure call within the current 256 MB-aligned region

Description:

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, at which location execution continues after a procedure call.

This is a PC-region branch (not PC-relative); the effective target address is in the "current" 256 MB-aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

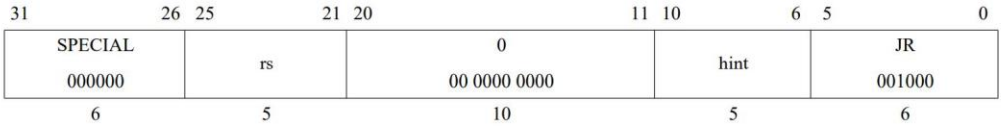
Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:   GPR[31]← PC + 8
I+1:PC      ← PC_GPRLEN-1..28 || instr_index || 0^2
```

Jump Register — JR



Format: `JR rs` **MIPS32**

Purpose:

To execute a branch to an instruction address in a register

Description: `PC ← GPR[rs]`

Jump to the effective target address in GPR *rs*. Execute the instruction following the jump, in the branch delay slot, before jumping.

For processors that implement the MIPS16e ASE, set the *ISA Mode* bit to the value in GPR *rs* bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one

Restrictions:

The effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16e ASE, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction. For processors that do implement the MIPS16e ASE, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

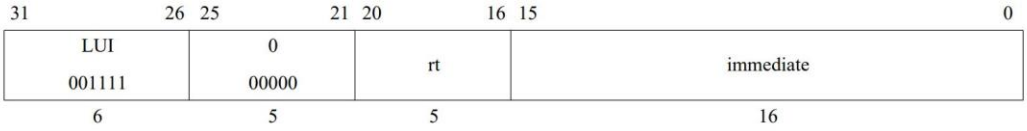
In release 1 of the architecture, the only defined hint field value is 0, which sets default handling of JR. In Release 2 of the architecture, bit 10 of the hint field is used to encode an instruction hazard barrier. See the JR.HB instruction description for additional information.

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:   temp ← GPR[rs]
I+1:if Config1CA = 0 then
        PC ← temp
    else
        PC ← tempGPRLEN-1..1 || 0
        ISAMode ← temp0
    endif
```

Load Upper Immediate — LUI



Format: `LUI rt, immediate` **MIPS32**

Purpose:

To load a constant into the upper half of a word

Description: `GPR[rt] ← immediate || 0^16`

The 16-bit *immediate* is shifted left 16 bits and concatenated with 16 bits of low-order zeros. The 32-bit result is placed into GPR *rt*.

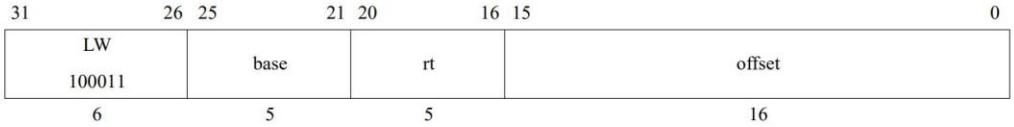
Restrictions:

None

Operation:

```
GPR[rt] ← immediate || 0^16
```

Load Word — LW



Format: `LW rt, offset(base)` **MIPS32**

Purpose:

To load a word from memory as a signed value

Description: `GPR[rt] ← memory[GPR[base] + offset]`

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

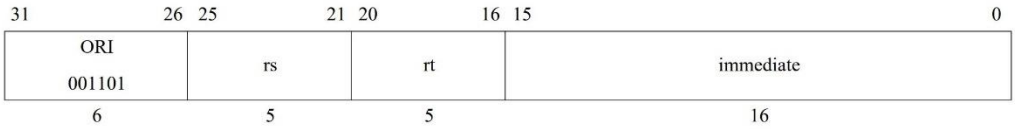
Restrictions:

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
memword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt]← memword
```

Or Immediate — ORI



Format: `ORI rt, rs, immediate` **MIPS32**

Purpose:

To do a bitwise logical OR with a constant

Description: `GPR[rt] ← GPR[rs] or immediate`

The 16-bit *immediate* is zero-extended to the left and combined with the contents of GPR *rs* in a bitwise logical OR operation. The result is placed into GPR *rt*.

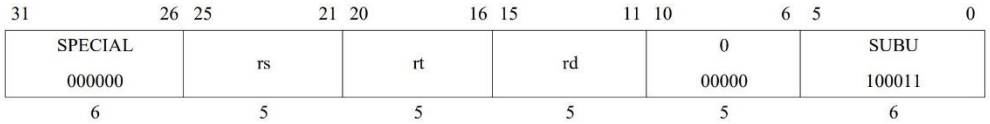
Restrictions:

None

Operation:

```
GPR[rt] ← GPR[rs] or zero_extend(immediate)
```

Subtract Unsigned Word **SUBU**



31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	SUBU 100011
6	5	5	5	5	6

Format: `SUBU rd, rs, rt` **MIPS32**

Purpose:

To subtract 32-bit integers

Description: `GPR[rd] ← GPR[rs] - GPR[rt]`

The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* and the 32-bit arithmetic result is and placed into GPR *rd*.

No integer overflow exception occurs under any circumstances.

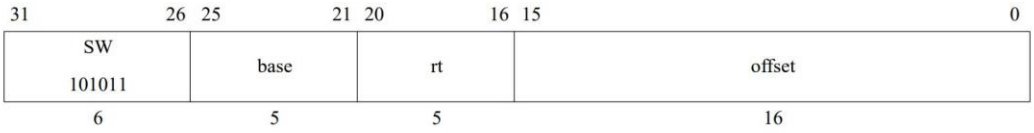
Restrictions:

None

Operation:

```
temp   ← GPR[rs] - GPR[rt]
GPR[rd]← temp
```

Store Word **SW**



31–26	25–21	20–16	15–0
SW 101011	base	rt	offset
6	5	5	16

Format: `SW rt, offset(base)` **MIPS32**

Purpose:

To store a word to memory

Description: `memory[GPR[base] + offset] ← GPR[rt]`

The least-significant 32-bit word of GPR *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
dataword← GPR[rt]
StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
```

为了实现这些功能，CPU 主要包含了 IM、GRF、ALU、DM、PC 等，这些模块按照流水线的顶层设计逐级展开。此处省略了一些小模块，比如左移两位，和一些信号的加减等。

流水线 CPU 执行指令分为五个阶段：取指令 Fetch，译码 Decode，执行 Execute，存储器 Memory，写回 Writeback。为此，流水线 CPU 需要增加 4 层

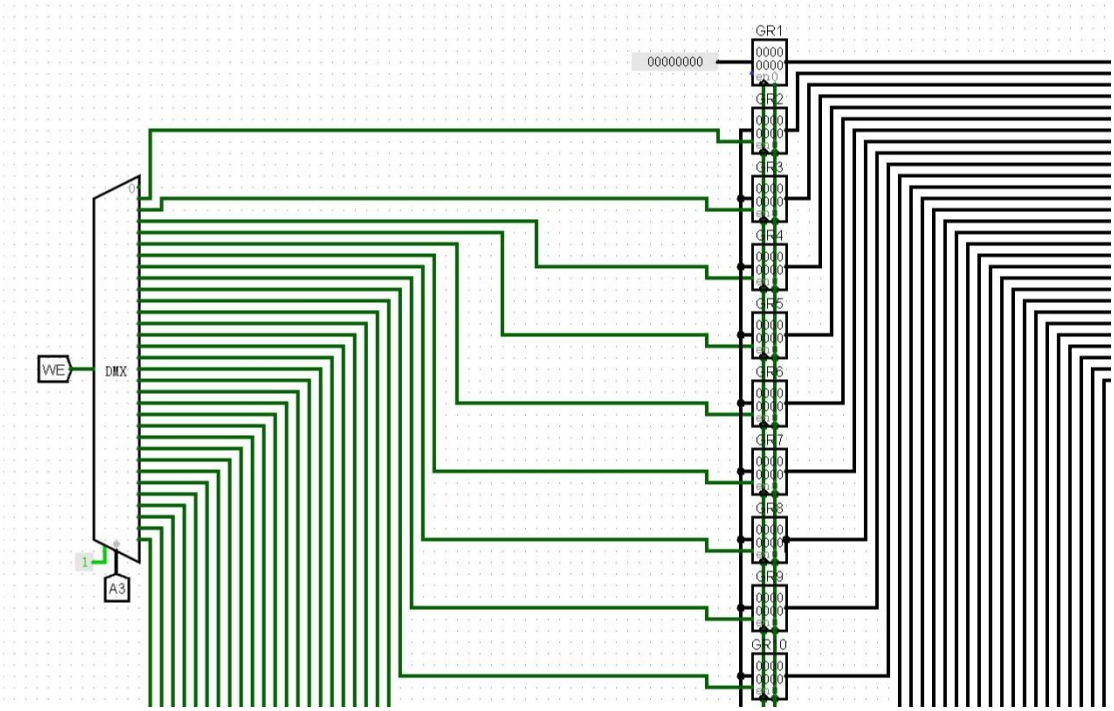
寄存器，分别为 D 级，E 级，M 级和 W 级。

（二）关键模块定义

1. GRF

功能描述	32 个 32 位寄存器，具有异步复位功能，0 号寄存器接地；读出时将 A1 和 A2 对应的寄存器值通过 RD1 和 RD2 输出；写入时如果 Wr 有效，则将 WD 输入写入 A3 寄存器。第 32 个寄存器初始化为 0x00003000.	
信号名	方向	功能
A1[4:0]	I	读出数据的第一个寄存器编号
A2[4:0]	I	读出数据的第二个寄存器编号
A3[4:0]	I	写入寄存器的编号
RD1[31:0]	O	读出的第一个寄存器值
RD2[31:0]	O	读出的第二个寄存器值
WD3	I	写入寄存器的值
RegWrite	I	写入使能
Clk（内置）	I	CPU 时钟
Reset	I	异步复位信号

部分结构如下（截不全）



2. DM

RAM 调为 separate 模式。DM 起始地址为 0x00000000.

功能描述	读写存储器	
信号名	**方向**	**功能**
A[31:0]	I	要写入或读出的地址（取 5 位即可）
WD[31:0]	I	写入的 32 位数据
RD[31:0]	O	读出的 32 位数据
Wr	I	写入使能
Clk（内置）	I	CPU 时钟
reset	I	异步复位信号

3. ALU

ALU 内部部件有 ALU 控制器，加法器，反相器，或运算器，外部的 bit-extend 等。

3.1. ALU 功能分析

功能	MIPS 指令	addu	subu	ori	lw	sw	beq
	计算需求	**加法**	**减法**	**立即数**	**加法**	**加法**	**相等比**

分析				**或运算**			**较**
	加法器 B 端的选择	原	反相	-	原	原	反相
	加法器进位	0	1	-	0	0	1
	扩展方式	-	-	无符号	符号	符号	-

3.2. ALU 输入输出信号

功能描述	实现加、减、或、比较大小等功能	
信号名	方向	功能
ALUOp[1:0]	I	控制信号，00-加，01-减，10 保留，11-或
Zero	O	输出 beq 的比较结果，1 为等于，0 为不等
SrcA[31:0]	I	第一个运算数
SrcB[31:0]	I	第二个运算数
ALUOut[31:0]	O	ALU 计算结果

3.3. ALU 控制器

ALUOp[1:0]	M1Sel	Cin	M2Sel
00	0	0	0
01	1	1	0
10	-	-	--
11	-	-	1

4. PC

功能描述	32 位可异步复位的寄存器，起始地址为 0x00003000	
信号名	方向	功能
Clk（内置）	I	CPU 时钟
Reset	I	异步复位信号，将 PC 值复为 0x00003000
Di[31:0]	I	32 位输入，输入下一 PC 地址
Do[31:0]	O	32 位输出，输出当前 PC 地址

5. IM

功能描述	ROM 实现，大小为 32*32bit，根据输入地址，输出指令数据	
信号名	方向	功能
A[31:0]	I	32 位指令输入
D[31:0]	O	32 位指令输出

6. EXT

选择三种扩展方式，无符号、有符号、lui 扩展。

功能描述	实现三种扩展方式	
信号名	**方向**	**功能**
EXTOp[1:0]	I	控制信号，00 为无符号扩展；01 为符号扩展；10 为 lui 扩展。
Imm[15:0]	I	带扩展的 16 位立即数
Ext[31:0]	O	输出 32 位扩展的数字

7. D 级流水线寄存器

传递的信号有 PC[31:0]、IM[31:0]、PCPlus4F[31:0]，其中 PC 的寄存器初始化为 0x00003000.为了暂停和转发，除了这些寄存器之外，还添加了外部的 reset 信号和用于冻结寄存器的使能信号 EnD。另外，内置有 CPU 时钟。

8. E 级流水线寄存器

由于我按照书上的结构，采用了集中译码的方式，因此传递的信号比较多。传递的信号有 PC[31:0]、PCPlus4F[31:0]、JalD、RegWriteD、MemtoRegD、MemWriteD、BranchD、ALUControlD、ALUSrcD、RegDstD、Instr[31:0]、RD1[31:0]、reE[4:0]、rdE[4:0]、Signext[31:0].除此之外，为了暂停和转发，满足清除 E 级寄存器的要求，我为 E 级寄存器添加了 FlushE 信号用于清零。

9. M 级流水线寄存器

传递的信号有 JalE、RegWriteE、MemtoRegE、MemWriteE、BranchE、

Instr[31:0]、Zero、ALUOut[31:0]、WriteDataE[31:0]、WriteRegE[4:0]、PCBranchE[31:0]、PC[31:0]。

10. W 级流水线寄存器

传递的信号有 JalM、RegWriteM、MemtoRegM、Instr[31:0]、ALUOut[31:0]、WriteDataE[31:0]、WriteRegE[4:0]、PC[31:0]。

（三）重要机制实现方法

0. 流水线数据通路及控制信号

数据通路：

部件	PC	IM	GRF				ALU		DM		EXT
信号	PC’	A	A1	A2	A3	WD3	SrcAE	SrcBE	A	WD	Imm
addu	PCPlus4F	-	InstrD[25:21]	InstrD[20:16]	Rd	ALUOutW	RD1	RD2	-	-	-
subu	PCPlus4F	-	InstrD[25:21]	InstrD[20:16]	Rd	ALUOutW	RD1	RD2	-	-	-
ori	PCPlus4F	-	InstrD[25:21]	-	Rt	ALUOutW	RD1	ImmE	-	-	Instr[15:0]
lui	PCPlus4F	-	InstrD[25:21]	-	Rt	ALUOutW	RD1	ImmE	-	-	Instr[15:0]
lw	PCPlus4F	-	InstrD[25:21]	-	Rt	ReadDataW	RD1	ImmE	ALUOutM	-	Instr[15:0]
sw	PCPlus4F	-	InstrD[25:21]	InstrD[20:16]	-	-	RD1	ImmE	ALUOutM	WD	Instr[15:0]
beq	PCBranchM	-	InstrD[25:21]	InstrD[20:16]	-	-	RD1	RD2	-	--	Instr[15:0]
j	JumpAddr	-	-	-	-	-	-	-	-	-	-
jal	JumpAddr	-	-	-	31	PCPlus4D	-	-	-	-	-
jr	ResultW	-	InstrD[25:21]	-	-	-	-	-	-	-	-

控制信号矩阵：

指令	RegWriteD	MemtoRegD	MemWriteD	BranchD	ALUControlD	ALUSrcD	RegDstD	ExtOpD	JumpD	JalD
addu	1	0	0	0	00	0	01	00	00	0
subu	1	0	0	0	01	0	01	00	00	0
ori	1	0	0	0	11	1	00	00	00	0

lui	1	0	0	0	00	1	00	10	00	0
lw	1	1	0	0	00	1	00	01	00	0
sw	0	0	1	0	00	1	00	01	00	0
beq	0	0	0	1	01	0	00	01	00	0
j	0	0	0	0	00	0	00	00	01	0
jal	1	0	0	0	00	0	10	00	01	1
jr	0	0	0	0	00	0	0	00	10	0

由控制信号矩阵，以及 p4 采用过的和逻辑、或逻辑的方法，即可构建出流水线控制器。

1. 跳转

对于跳转，我采用了延迟槽方法。这种方法要求跳转类指令需要在第二个周期（也就是 D 级）就必须完成 PC 寄存器的赋值，而仅仅允许一条指令进入延迟槽。因此，我将 beq 指令的比较运算提前，并添加了许多控制信号，修正了数据通路，用以实现这个方法。

首先，beq 指令需要在 D 级内安排一个 32 位数的比较器，比较后使用这个结果和 EXT 模块的输出结果联合控制 PC 的输入信号。

对于 j 指令，我将 26 位立即数传至 PC 更新值前，然后和原 PC 值的前 4 位组合，构成了新的 PC 值。

jal 指令则比较特别，需要在第二个周期完成跳转，而在第五个周期完成 31 号寄存器的赋值！跳转部分类似 j 指令，而传值部分直接将其交给后面的寄存器即可，让 jal 需要赋的值与 0 号寄存器做加法，最后传回 D 级的 GRF 模块。

对于 jr 指令，我新建了一条从 GRF 的 RD1 到 PC 的数据通路，用于将 jr 中寄存器的值直接取回给 PC 完成跳转。

综上所述，我对四条跳转指令完成了延迟槽的设计。

2. 转发（还未实现）

采用标记法来实现转发。由于转发只会因为寄存器的值冲突而产生，因此只需要接收 E 级的两个寄存器的值，以及存储器和写回阶段制定的目的寄存器。当存储器和写回的需求地址等于 E 级寄存器转发过来的地址时，将输出更新为转发过来的值；否则不更新，直接传递原来的需求位点读入的值。

3. 暂停（正在实现）

采用需求时间-供给时间策略，来分析是否需要暂停和转发。需求时间 Tuse 定义为这条指令位于 D 级的时候，还需要多少个周期就必须要获得相应的数据。

指令	beq	addu	subu	ori	lw/sw_rs	sw_rt	jr
Tuse	0	1	1	1	1	2	0

供给时间 Tnew 定义为这条指令位于某流水级的时候，还需要多少个周期才能算出结果并放回流水级寄存器中。

指令	addu	subu	ori	lui	lw	jal
Tnew_D	2	2	2	2(1)	3	1(2)
Tnew_E	1	1	1	1(0)	2	0(1)
Tnew_M	0	0	0	0	1	0
Tnew_W	0	0	0	0	0	0

Tnew>Tuse 时，需要暂停。得到

D 级当前指令			E 级			M 级
指令类型	源寄存器	Tuse	addu/subu/ori	lw	jal	lw
beq/jr	rs/rt	0	暂停	暂停		暂停
addu/subu	rs/rt	1		暂停		
ori/lui	rt	1		暂停		
lw/sw	rs	1		暂停		
sw	rt	2				

因此，暂停的条件为：

$$
\begin{aligned}
stall\_b \;=\; & InstrD.op == beq \;\&\{[InstrE.op == addu/subu \;\&\; ((InstrD.rs == \\
& == InstrE.rd)|(InstrD.rt == InstrE.rd))] \mid [InstrE.op == \\
& == ori/lui \;\&\; ((InstrD.rs == InstrE.\mathrm{rt})|(InstrD.rt == \\
& == InstrE.\mathrm{rt}))] \mid [InstrE.op == lw \;\&\; ((InstrD.rs == \\
& == InstrE.\mathrm{rt})|(InstrD.rt == InstrE.\mathrm{rt}))] \mid [InstrM.op == \\
& == lw \;\&\; ((InstrD.rs == Instr\mathrm{M}.\mathrm{rt})|(InstrD.rt == Instr\mathrm{M}.\mathrm{rt}))]\}
\end{aligned}
$$

$$stall\_jr = InstrD.op == jr \&\{[InstrE.op == addu/subu \& (InstrD.rs == InstrE.rd)] | [InstrE.op == ori/lui \& (InstrD.rs == InstrE.rt)] | [InstrE.op == lw \& (InstrD.rs == InstrE.rt)] | [InstrM.op == lw \& (InstrD.rs == InstrM.rt)]\}$$

$$stall\_calr = InstrD.op == addu/subu/ori \& InstrE.op == lw \& [InstrD.rs == InstrE.rt | InstrD.rt == InstrE.rt]$$

$$stall\_ls = InstrD.op == lw/sw \& InstrE.op == lw \& [InstrD.rs == InstrE.rt | InstrD.rt == InstrE.rt]$$

stall = stall_B+stall_jr+stall_calr+stall_ls.

暂停要执行的操作有：

- 冻结 PC
- 清零 E 级寄存器
- 冻结 D 级寄存器

二、测试方案

（一）典型测试样例

1. ALU 功能测试

测试了以下几条运算指令：

addu	$31, $30, $29	(0x03DDF821)
addu	$31, $31, $29	(0x03FDF821)
subu	$31, $31, $29	(0x03DDF823)
subu	$18, $17, $16	(0x02508823)
lui	$30, 0x0000ffff	(0x3C1EFFFF)
ori	$30, $30, -2	(0x37DEFFFE)
ori	$29, $29, 1	(0x37BD0001)

2. DM 功能测试

这部分主要沿袭了 p4 的部件和测试方法。

3. 指令测试

指令编码：

addu 000000 100001

subu 000000 100011

ori 001101

lw 100011

sw 101011

beq 000100

lui 001111

nop 000000 000000

jal 000011

j 000010

jr 000000 001000

4.暂停测试

正在完成

（二）自动测试工具

1. 综合测试

使用平台上给的弱测程序，并加以了一定的修改，保证覆盖了所有跳转的可能性。

三、思考题

（一）为何只有 GRF 和 PC 的状态改变会引发冲突？DM 的状态也会改变，也会读取改变后的值，为何不会产生冲突？（提示：从对于某一存储模块，如 GRF，DM，分析流水线中哪一级写入，哪一级读取）

因为可能对 DM 产生读写操作的指令只有 lw 和 sw，而这些指令都是在流

水线 M 级才会产生对存储器的操作。这样其实确保了不会出现“写后读”而且读的周期在写之前这样的事情发生了。

（二）上文的描述的标记法中，有哪些细节 bug？比如当地址为 0 号寄存器时，应该如何处理？GRF 是否有内部转发导致的需求位点一共有那些？请同学们思考，并在自己 CPU 的具体实现中理解分析并应用。

当地址为 0 号寄存器时，可能会出现 E 级无关指令的该域也为 0 的情况，造成错误的转发。因此当地址为 0 时需要特判，多加一个控制信号来封堵这个转发的可能，转而直接输入一个 0 值。