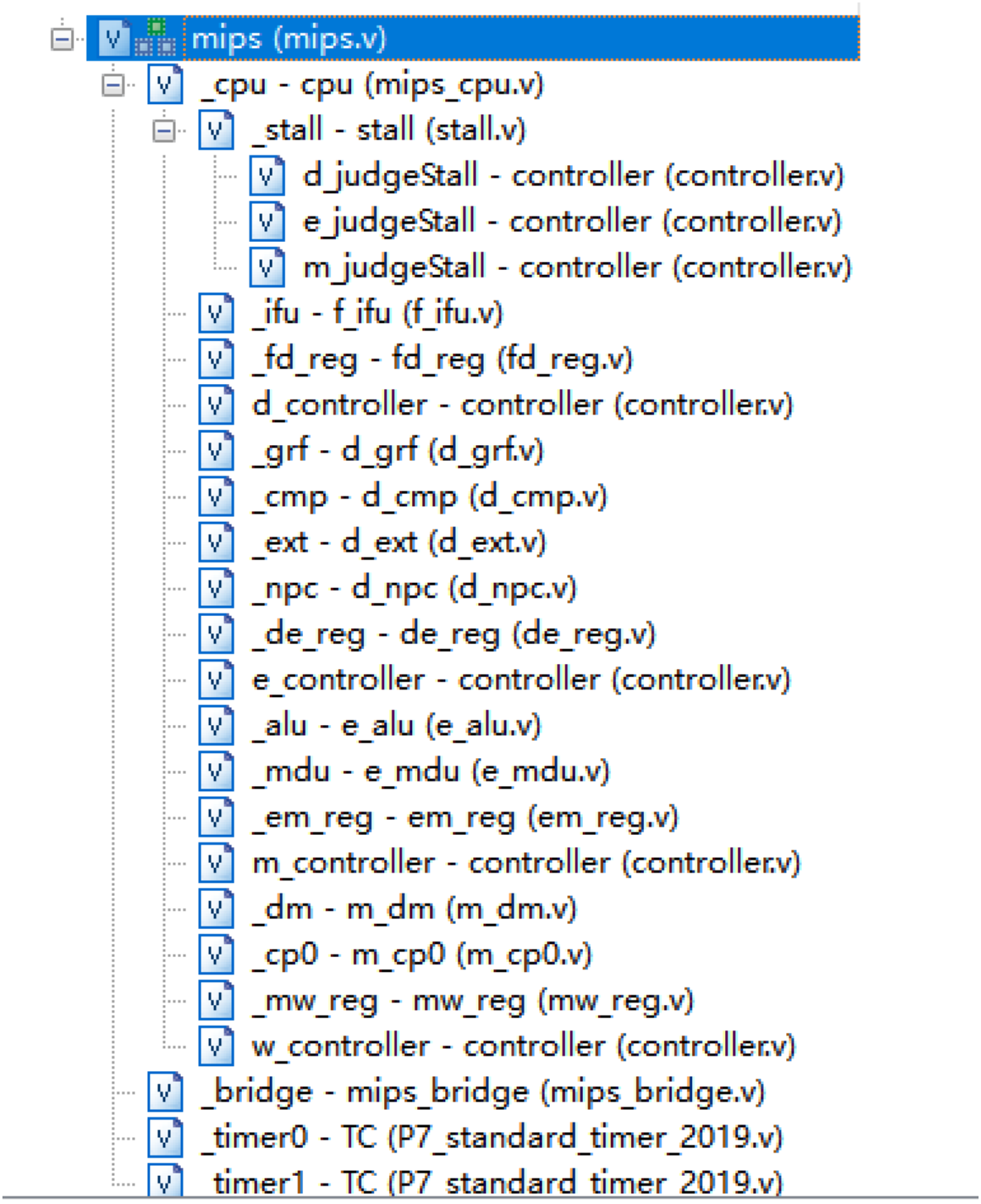
P7设计文档

命名规则：

1. 主模块为mips.v，命名为mips
2. 主程序中的模块为mips_模块名，其它和P6保持一致（如下）
3. 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”
4. 流水线寄存器有四个，命名为“流水线连续层级_reg“
5. 其余模块命名为”_模块名“

整体架构



相对P6的调整

1. 增加m_cp0模块，用于统筹中断和异常
2. 增加异常判断数据通路，用以收集指令异常信息
3. 将原有mips程序封装为mips_cpu，以实现cpu与系统桥的交互，从而达到"高内聚低耦合"的目标
4. 增加mips_bridge，以将原有的dm接口分发到dm、timer0、timer1三个不同的区域

5. 增加两个mips_timer，定期产生中断信号

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	4
pc4	pcImm16	pcImm26	pcReg	epc

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

---

cp0相关信号

	mfc0	mtc0
cp0Write	0	1
cp0AddrIn	[15:11]	
cp0AddrOut		[15:11]
cp0Wd		grf_rt

cp0Write

决定是否对cp0进行写操作

cp0AddrIn

从cp0中读数据的地址

cp0AddrOut

向cp0中写数据的地址

cp0Addr	register
12	sr
13	cause
14	epc

cp0Wd

向cp0中写入的数据

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、请查阅相关资料，说明鼠标和键盘的输入信号是如何被 CPU 知晓的？

键盘和鼠标这类的低速设备是通过中断请求的方式进行IO操作的。当键盘上按下一个按键的时候，键盘会发出一个中断信号，中断信号经过中断控制器传到CPU，然后CPU根据不同的中断号执行不同的中断响应程序，然后进行相应的IO操作，把按下的按键编码读到寄存器（或者鼠标的操作），最后放入内存中

这个过程可以被简化为以下几个步骤：

1. 用户通过键盘或鼠标进行操作。
2. 键盘或鼠标生成中断信号。
3. 中断信号被送到中断控制器。
4. 中断控制器将中断信号发送到CPU。
5. CPU根据中断信号的类型，执行相应的中断处理程序。
6. 中断处理程序读取键盘或鼠标的输入，并将其存储在寄存器中。
7. 操作系统或应用程序使用这些信息完成相应操作。

2、请思考为什么我们的 CPU 处理中断异常必须是已经指定好的地址？如果你的 CPU 支持用户自定义入口地址，即处理中断异常的程序由用户提供，其还能提供我们所希望的功能吗？如果可以，请说明这样可能会出现什么问题？否则举例说明。（假设用户提供的中断处理程序合法）

异常处理程序的地址实际上是该程序的"入口"，通过访问它可以让操作系统对中断异常进行响应，所以必须固定一个不随用户程序更改的地址，以在程序刚开始执行时就被加载到程序的特定位置；

如果CPU支持用户自定义入口地址，即处理中断异常的程序由用户提供，理论上是可以的。但是，这样做可能会带来一些问题：

1. **安全性问题**：用户提供的中断处理程序可能会包含恶意代码，这可能会导致系统崩溃或者数据泄露。
2. **稳定性问题**：用户提供的中断处理程序可能存在bug，这可能会导致系统崩溃或者不稳定。
3. **兼容性问题**：不同的用户可能会提供不同的中断处理程序，这可能会导致系统在不同的环境下表现不一致。

3、为何与外设通信需要 Bridge？

作为程序员，内存对我们来说是透明的，所以我们将内存当作一段连续的地址空间来进行各种内存操作。而实际上，内存空间可能是由多种多样的程序由多种多样的结构组合而成的，这时就需要一个bridge将整体的地址分发到不同的程序中。这也符合"高内聚，低耦合"的原则。

4、请阅读官方提供的定时器源代码，阐述两种中断模式的异同，并分别针对每一种模式绘制状态移图。

计时器和中断发生器是两种不同的中断模式。以下是它们的异同以及各自的状态转移图：

计时器：

- 计时器是一种外部设备，其主要功能是根据设定的时间来定时产生中断信号。
- 当计时器到达预设的时间，它会产生一个中断信号并发送给 CPU。
- CPU 会响应这个中断信号，并执行相应的中断处理程序。

中断发生器：

- 中断发生器是一种可以在任何时刻产生中断信号的设备。
- 当中断发生器产生一个中断信号，它会持续置高，直到微系统做出响应，才变回低位。
- 对中断发生器的响应是通过系统桥来实现的，通过 store 类指令访问地址 0x7F20，就可以达到响应中断的目的。

计时器状态转移图：

```
1 开始 -> 设置时间 -> 计时开始 -> 时间到达 -> 产生中断 -> CPU 响应中断 -> 结束
```

中断发生器状态转移图：

```
1 开始 -> 产生中断 -> 中断信号置高 -> 系统桥响应 -> 中断信号置低 -> 结束
```

5、倘若中断信号流入的时候，在检测宏观 PC 的一级如果是一条空泡（你的 CPU 该级所有信息均为空）指令，此时会发生什么问题？在此例基础上请思考：在 P7 中，清空流水线产生的空泡指令应该保留原指令的哪些信息？

这会导致存入epc寄存器的地址为0，最终使得程序无法正常返回正确中断地址；

空泡应保持pc信息，以保证eret正确

6、为什么 `jalr` 指令为什么不能写成 `jalr $31, $31`？

当执行 `jalr $31, $31` 时，由于指令的执行是顺序的，CPU 首先会将 PC+4 （返回地址）保存到 `$31`，然后再跳转到 `$31` 中的地址。但是，由于 `$31` 已经被修改，所以跳转的地址已经不再是原来 `$31` 中的地址，而是新保存的返回地址。这就导致了跳转地址和返回地址相同，形成了一个无限循环，程序无法继续执行