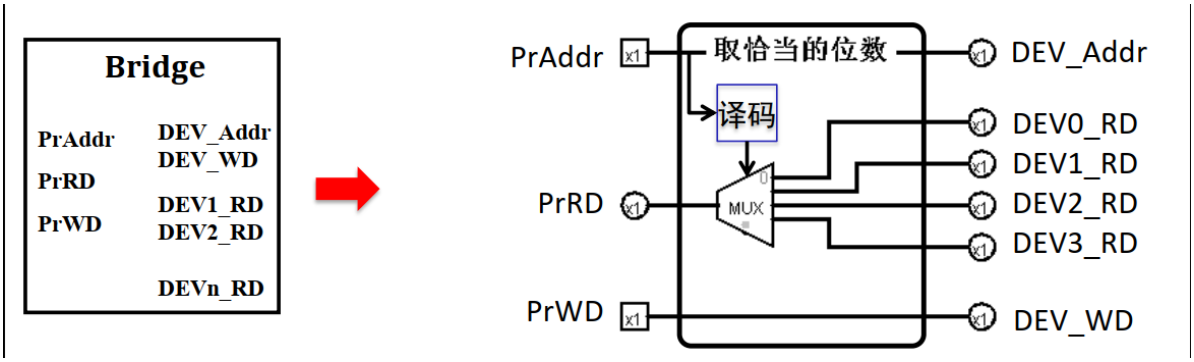
MIPS微体系结构

Bridge

方便CPU与多种不同的设备I/O，CPU侧一组接口，设备侧N组接口。其功能为完成地址、数据转换，控制信号的产生。

输入：PrAddr，PrRD，PrWD

输出：DEV_Addr，DEV_WD，DEV1_RD，DEV1_WE……



CP0

协处理器0，处理异常和中断。内部包含4个寄存器，分别为

- **SR**：用于对系统进行控制
  - 指令可读可写
  - IM7-2[15:10]：6个中断屏蔽位，分别对应6个外部中断
    - 1-允许中断，0-不允许中断
  - IE[0]：全局中断使能
    - 1-允许中断，0-不允许中断
  - EXL[1]：异常级
    - 1-进入异常，不允许再中断；0-允许中断
    - 重入（在中断程序中仍然允许再次进行中断
- **Cause**：指令读取，硬件控制写入
  - IP7-2[15:10]：对应外部6个中断源
    - 记录当前那些硬件中断正在有效
    - 1-有中断，0-无中断
  - ExcCode[6:2]：异常/中断类型编码值
    - 共32种，与课程有关的为
- **EPC**：用于保存异常/中断发生时的PC
  - 保存PC：硬件控制写入
  - 指令读取：中断服务程序
- **PRId**：处理器ID

中断异常同时发生时，中断优先级高于异常

异常码

ExcCode	助记符	指令与指令类型	描述
0	Int（外部中断）	所有指令	中断
4	AdEL（取指/数异常）	load类，PC处	地址出错
5	AdES（存数异常）	store类	地址出错
10	RI（未知指令）	-	不识别(非法)指令
12	Ov（溢出异常）	add,addi,sub	算数指令导致的异常(溢出等)

CPU

顶层设计

各元件所在层级

1. F级：IFU
2. D级：GRF，EXT，CMP，NPC
3. E级：ALU，HILO
4. M级：DM，BE
5. W级：无

各层级寄存器储存的值

1. D级：instr，pc，BD，ExcCode
2. E级：instr，pc，EXTout，RD1，RD2，BD，ExcCode
3. M级：instr，pc，EXTout，ALUout，RD2，HILOout，BD，ExcCode
4. W级：instr，pc，EXTout，ALUout，DMout，HILOout，CP0out

转发

转发的供给端为各层寄存器的输出。

只有rs和rt会需要转发，转发需求端为：

1. NPC的rs输入
2. CMP的rs和rt输入
3. ALU的rs和rt输入
4. HILO的rs和rt输入
5. DM的rt输入

转发路径：

1. E2D，M2D，W2D。EM通过直接转发完成，W通过grf内部转发完成，优先级为EMW。
2. M2E，W2E。均通过直接转发完成。优先级为MW。
3. W2E。通过直接转发完成。

转发策略：

1. 在寄存器输出端完成转发，即本阶段使用的所有相关值若存在转发则都用转发值。
2. 转发的判断标准是地址相同，写使能为1，从前到后依次转发。

阻塞

阻塞的唯一判别标准是同一个数据使用时间小于数据产生时间，也即$Tuse < Tnew$。

若阻塞则需要置零pc寄存器的写使能，D级寄存器的写使能以及需要清零E级寄存器。

支持的指令

```
add, sub, addu, subu, and, or, slt, sltu, lui
addi, addiu, andi, ori
lb, lh, lhu, lw, sb, sh, sw
mult, multu, div, divu, mfhi, mflo, mthi, mtlo
beq, bne, jal, jr
```

指令识别表

指令	opcode	funct	指令	opcode	funct
add	000000	100000	sub	000000	100010
addu	000000	100001	subu	000000	100011
and	000000	100100	or	000000	100101
slt	000000	101010	sltu	000000	101011
lui	001111		addi	001000	
andi	001100		ori	001101	
lb	100000		lh	100001	
lw	100011		sb	101000	

指令	opcode	funct	指令	opcode	funct
sh	101001		sw	101011	
mult	000000	011000	multu	000000	011001
div	000000	011010	divu	000000	011011
mfhi	000000	010000	mflo	000000	010010
mthi	000000	010001	mtlo	000000	010011
beq	000100		bne	000101	
jal	000011		jr	000000	001000
eret	010000	011000	syscall	000000	001100
addiu	001001		lhu	100101	
指令	**opcode**	**[25:21]**	**指令**	**opcode**	**[25:21]**
mfc0	010000	00000	mtc0	010000	00100

指令分类

```
calc_r:  add, sub, addu, subu, and, or, slt, sltu
calc_i:  addi, addiu, andi, ori
  load:  lb, lh, lhu, lw
 store:  sb, sh, sw
muldiv:  mult, multu, div, divu
    mf:  mfhi, mflo
    mt:  mthi, mtlo
branch:  beq, bne

lui, jal, jr
```

T值表

指令/指令类	D_Tuse_RS	D_Tuse_RT	D_Tnew	Tuse_EPC	Tnew_EPC	Dst
calc_r	1	1	2			rd
calc_i	1	x	2			rt
load	1	x	3			rt
store	1	2	x			x
muldiv	1	1	x			x
mf	x	x	2			rd
mt	1	x	x			x
branch	0	0	x			x
lui	x	x	2			rt
jal	x	x	0			ra
jr	0	x	x			x
eret	x	x	x	0	3	x
syscall	x	x	x			x
mtc0	x	2	x			x
mfc0	x	x	3			rt

控制信号表

```
 GRF:  RegWrite, A3Sel[2:0], WD3Sel[2:0]
  DM:  MemWrite, DMOp[2:0], BEOp[2:0]
 ALU:  ALUSrc, ALUOp[4:0]
 NPC:  NPCOp[3:0]
 CMP:  CMPOp[2:0]
 EXT:  EXTOp[2:0]
HILO:  HILOOp[4:0]
```

控制真值表

一般指令类行为:

	RegWrite	A3Sel	WD3Sel	MemWrite	DMOp	BEOp
calc_r	1	rd(1)	ALUout(0)	0	x	x
calc_i	1	rt(0)	ALUout(0)	0	x	x
load	1	rt(0)	DMout(1)	0	___	___
store	0	x	x	1	___	x
muldiv	0	x	x	0	x	x
mf	1	rd(1)	HILOout(3)	0	x	x

	RegWrite	A3Sel	WD3Sel	MemWrite	DMOp	BEOp
mt	0	x	x	0	x	x
branch	0	x	x	0	x	x

	ALUSrc	ALUOp	NPCOp	CMPOp	EXTOp	HILOOp
calc_r	WD2(0)	___	ADD4(0)	x	x	x
calc_i	EXTout(1)	___	ADD4(0)	x	___	x
load	EXTout(1)	ADD(0)	ADD4(0)	x	SEXT(1)	x
store	EXTout(1)	ADD(0)	ADD4(0)	x	SEXT(1)	x
muldiv	x	x	ADD4(0)	x	x	___
mf	x	x	ADD4(0)	x	x	___
mt	x	x	ADD4(0)	x	x	___
branch	x	x	BRCH(1)	___	x	x

一般指令行为:

calc_r:

	add	sub	addu	subu	and	or	slt	sltu
ALUOp	ADD(0)	SUB(1)	ADDU(7)	SUBU(8)	AND(2)	OR(3)	SLT(5)	SLTU(6)

calc_i:

	addi	addiu	andi	ori
ALUOp	ADD(0)	ADDU(7)	AND(2)	OR(3)
EXTOp	**SEXT(1)**	**SEXT(1)**	**ZEXT(0)**	**ZEXT(0)**

load:

	lb	lh	lhu	lw
DMOp	LB(6)	LH(5)	LH(5)	LW(4)
BEOp	BS(2)	HS(4)	HU(3)	NO(0)

store:

	sb	sh	sw
DMOp	SB(3)	SH(2)	SW(1)

muldiv:

	mult	multu	div	divu

HILOOp	MULT(1) **mult**	MULTU(2) **multu**	DIV(3) **div**	DIVU(4) **divu**

mf:

	mfhi	**mflo**
HILOOp	MFHI(5)	MFLO(6)

mt:

	mthi	**mtlo**
HILOOp	MTHI(7)	MTLO(8)

branch:

	beq	**bne**
CMPOp	EQ(0)	NE(1)

特殊指令行为:

	lui	**jal**	**jr**	**eret**	**syscall**	**mtc0**	**mfc0**
RegWrite!!!	1	1	0	0	0	0	1
A3Sel	rt(0)	ra(2)	x	x	x	x	rt(0)
WD3Sel[2:0]	ALUout(0)	PC8(2)	x	x	x	x	CP0(4)
MemWrite!!!	0	0	0	0	0	0	0
DMOp[2:0]	x	x	x	x	x	x	x
BEOp[2:0]	x	x	x	x	x	x	x
ALUSrc	EXTout(1)	x	x	x	x	x	x
ALUOp[4:0]	LUI(4)	x	x	x	x	x	x
NPCOp[3:0]	ADD4(0)	JAL(2)	JR(3)	ERET(4)	x	ADD4(0)	ADD4(0)
CMPOp[2:0]	x	x	x	x	x	x	x
ExtOp[2:0]	ZEXT(0)	x	x	x	x	x	x
HILOOp[4:0]	x	x	x	x	x	x	x
CP0Write!!!	0	0	0	0	0	1	0

模块

PC

包含一个32位PC寄存器，保存当前程序指令位置。

信号名	方向	描述
clock	I	时钟信号
reset	I	同步复位信号 1：将PC寄存器复位为0x0000_3000 0：无效
stall	I	是否中断 1：中断，$PC \leftarrow PC$ 0：不中断，$PC \leftarrow NPC$
npc[31:0]	I	下一个PC值，在时钟上升沿更新
pc[31:0]	O	当前PC值

CMP

信号名	方向	描述
A[31:0]	I	第一个输入
B[31:0]	I	第二个输入
CMPOp[2:0]	I	CMP功能选择信号 EQ(0)：$isBrch \leftarrow A = B$ BE(1)：$isBrch \leftarrow A \neq B$
isBrch	O	判断结果，是否跳转

NPC

计算下一个PC值

信号名	方向	描述
Req	I	中断信号
EPC[31:0]	I	EPC值
F_pc[31:0]	I	F级的pc值
D_pc[31:0]	I	D级的pc值
imm16[15:0]	I	16位立即数
imm26[25:0]	I	26位立即数
NPCOp[3:0]	I	选择如何更新PC ADD4(0)：$NPC \leftarrow F\_pc + 4$ BRCH(1)： $NPC \leftarrow \begin{cases} D\_pc + 4 + sign\_ext\{imm16 \| \| 0^2\} & isBrch = 1 \\ F\_pc + 4 & isBrch = 0 \end{cases}$ JAL(2)：$NPC \leftarrow D\_pc_{31..28} \| \| imm26 \| \| 0^2$ JR(3)：$NPC \leftarrow RD1$ ERET(4)：$NPC \leftarrow EPC + 4$
RD1	I	获取GPR[rs]
isBrch	I	CMP决定是否跳转
NPC[31:0]	O	下一个PC值

GRF

寄存器文件，包含32个32位寄存器，对应0-31号寄存器，0号寄存器结果恒为0。

信号名	方向	描述
clock	I	时钟信号
reset	I	同步复位信号 1：将所有寄存器复位为0 0：无效
WE3	I	写使能信号 1：可向GRF中写入数据 0：不可向GRF中写入数据
A1[4:0]	I	5位地址输入信号
A2[4:0]	I	5位地址输入信号
A3[4:0]	I	5位地址输入信号
WD3[31:0]	I	GRF[A3]←WD if WE = 1
pc[31:0]	I	获取当前pc
RD1[31:0]	O	RD1←GRF[A1]
RD2[31:0]	O	RD2←GRF[A2]

ALU

算数逻辑单元，根据ALUop进行运算。

信号名	方向	描述
A[31:0]	I	第一个输入
B[31:0]	I	第二个输入
ALUOp[4:0]	I	ALU功能选择信号 ADD(0)：$C \leftarrow A+B$ SUB(1)：$C \leftarrow A-B$ AND(2)：$C \leftarrow A\&B$ OR(3)：$C \leftarrow A\|B$ LUI(4)：$C \leftarrow B<<16$ SLT(5)：$C \leftarrow signed(A<B)$ SLTU(6)：$C \leftarrow A<B$
C[31:0]	O	输出结果
Ov	O	是否溢出

HILO

乘除逻辑单元，根据HILOOp执行不同操作，内含state寄存器存储计算剩余时间周期。

信号名	方向	描述
clk	I	时钟信号
rst	I	同步复位信号
A[31:0]	I	第一个输入
B[31:0]	I	第二个输入
HILOOp[4:0]	I	HILO功能选择信号 MULT(0)：$(hi,lo) \leftarrow A \times B$ MULTU(1)：$(hi,lo) \leftarrow unsign(A \times B)$ DIV(2)：$lo \leftarrow A/B$，$hi \leftarrow A \bmod B$ DIVU(3)：$lo \leftarrow unsign(A/B)$，$hi \leftarrow unsign(A \bmod B)$ MFLO(4)：$HILOout \leftarrow lo$ MFHI(5)：$HILOout \leftarrow hi$ MTLO(6)：$lo \leftarrow A$ MTHI(7)：$hi \leftarrow A$
HILOout[31:0]	O	输出结果
HILObusy	O	是否占用

DM

数据存储器信号转换器。

信号名	方向	描述
MemWrite	I	写使能信号
DMOp[2:0]	I	DM功能选择信号 LOAD(0)，SW(1)，SH(2)，SB(3)
m_data_addr[31:0]	O	待写入/读出的数据存储器相应地址
m_data_wdata[31:0]	O	待写入数据存储器相应数据
m_data_byteen[3:0]	O	四位字节使能

BE

字节扩展器，扩展数据存储器中读出的字节。

信号名	方向	描述
A[1:0]	I	最低两位的地址
Din[31:0]	I	输入的 32 位数据
BEOp[2:0]	I	数据扩展控制码 NO(0)：无扩展 BU(1)：无符号字节数据扩展 BS(2)：符号字节数据扩展 HU(3)：无符号半字数据扩展 HS(4)：符号半字数据扩展
Dout[31:0]	O	扩展后的 32 位数据

EXT

输入一个16位数，将其符号扩展为32位。

信号名	方向	描述
Input[15:0]	I	16位输入
EXTOp[2:0]	I	EXT功能选择信号 0：0扩展，$O \leftarrow \{0_{16}\|I\}$ 1：符号扩展，$O \leftarrow \{I[15]_{16}\|I\}$ 2：加载到高位，低位补零，$O \leftarrow \{I\|0_{16}\}$
Output[31:0]	O	32位输出

测试相关

随机测试使用generator生成mips代码，后通过mars dump to hex code，转存到code.txt中，对比输出序列检测正确性。

generator使用roife学长编写的cpp，**mars**使用Toby-Shi学长改编的mars-co。

使用数据：

- https://github.com/wzk1015/Computer-Organization
- https://github.com/refkxh/BUAA_CO_2019Fall
- https://github.com/PotassiumWings/BUAA-CO-2019
- https://github.com/rfhits/Computer-Organization-BUAA-2020
- https://github.com/BUAADreamer/BUAA-CO-2020
- 优 https://github.com/flyinglandlord/BUAA-CO-2021
- 优 https://github.com/SgtPepperr/BUAA_CO_2020
- 优 https://github.com/saltyfishyjk/BUAA-CO
- 优 https://github.com/NormalLLer/BUAA-CO-2021

思考题

P7

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？

   鼠标和键盘的输入信号是通过中断的方式被 CPU 知晓的。

   当外部设备（如鼠标或键盘）准备好传递信息给 CPU 时，它会向 CPU 发出中断请求。处理器收到中断请求后，它会暂停当前的工作，跳转到执行中断处理程序的地址。中断处理程序负责处理来自外部设备的信息，然后将控制权返回给处理器。

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）

   如果允许用户自定义入口地址，会出现以下问题：

   - 安全风险：如果用户自定义的中断处理程序存在安全漏洞，那么可能会导致数据泄露、恶意代码执行等安全问题。
   - 系统稳定性：如果用户自定义的中断处理程序不稳定，可能会导致系统崩溃或者其他故障。
   - 可维护性：如果允许用户自定义中断处理程序，系统维护人员就无法确定该系统中有哪些中断处理程序，也无法保证这些程序的质量和稳定性。

3. 为何与外设通信需要 Bridge？

   使用 Bridge 连接 CPU 和外设可以提高计算机系统的性能和灵活性，因为它允许 CPU 和外设使用不同的总线进行通信，并且可以灵活地添加或删除外设。

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。

   - Mode 0：产生定时中断。计时一次并于结束时产生中断信号，直到信号被响应。
   - Mode 1：产生周期中断。循环计时，结束时产生一周期的中断信号并进入新的计时。

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？

   写入的 EPC 和 BD 信号均为空，中断返回的位置会出现问题。

   需要保留其下一条指令的 EPC 和 BD。

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

因为需要保证在相同条件下多次执行同一个跳转指令，指令行为不变。

若相等，在`jalr`的延迟槽指令处产生了中断或异常，`jalr`的写入已经完成，返回时回到`jalr`，会导致跳转至错误的地址，即`jalr`写入的地址，也就是会无限循环。

P6

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？

   乘除法耗时较大，若整合进ALU则关键路径会变得很长，影响效率。使乘除法部件完全独立，不影响整体效率。

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。

   真实流水线CPU中乘除法被分为了若干个较小的过程，每周期计算特定的几位，之后一起结合成正确结果，会在几个周期后得到结果。

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？

   当busy为1且下一条指令为hilo相关时阻塞。

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）

   字节使能能使控制路径变短，加快处理速度，且统一表示每个byte，便于理解与控制。

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？

   不是一字节。在字节存取操作较多的时候，按字存取需要额外的处理，会增长关键路径，此时按字节存取效率高。

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？

   对指令进行合理的抽象与归类，同一类的控制信号与T值具有较大的相似性，仅有少量不同，可以利用这一点简化控制电路的搭建。

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？

   P6相对于P5只增加了乘除法相关的冲突，增加busy信号并扩展控制电路。

   ```
   mult $s0, $s1
   mfhi $s2
   mflo $s3
   mthi $t0
   mfhi $t0
   ```

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。

   每次考虑新增指令与老指令之间可能存在的冲突，并构造相应测试样例；对指令进行合理分类，对分类指令进行交叉测试。

P5

1. 我们使用提前分支判断的方法尽早产生结果来减少因不确定而带来的开销，但实际上这种方法并非总能提高效率，请从流水线冒险的角度思考其原因并给出一个指令序列的例子。

   跳转指令所需数据可能产生较晚，导致产生阻塞，提前分支判断也就失去了意义。

   ```
   add $1, $2, $3
   beq $1, $2, lable
   ```

2. 因为延迟槽的存在，对于 jal 等需要将指令地址写入寄存器的指令，要写回 PC + 8，请思考为什么这样设计？

   因为延迟槽会在跳转的时候执行下一条指令，所以跳回时需要执行下面的第二条指令。

3. 我们要求大家所有转发数据都来源于流水寄存器而不能是功能部件（如 DM 、 ALU ），请思考为什么？

   来源于流水线寄存器更方便管理与控制，其结构更加清晰，同时关键路径也更短。

4. 我们为什么要使用 GPR 内部转发？该如何实现？

   为了解决同时读写的结构冒险，实现可以为

   ```
   assign RD1 = (WE3 && A3 && A3 == A1) ? WD3 : regs[A1];
   ```

5. 我们转发时数据的需求者和供给者可能来源于哪些位置？共有哪些转发数据通路？

   见顶层设计中内转发的内容。

6. 在课上测试时，我们需要你现场实现新的指令，对于这些新的指令，你可能需要在原有的数据通路上做哪些扩展或修改？提示：你可以对指令进行分类，思考每一类指令可能修改或扩展哪些位置。

   在ALU、CMP、NPC中增加新的数据通路，在CTRL中增加新的判断通路，在中断控制中增加新的中断条件。

7. 确定你的译码方式，简要描述你的译码器架构，并思考该架构的优势以及不足。

   我的译码方式为首先识别命令，后根据命令通过一组或门来得到控制信号。

   优势为：编写简单，运行速度快。

   不足为：查错较为困难。

8. [P5 选做] 在冒险的解决中，我们引入了 AT 法，如果你有其他的解决方案，请简述你的思路，并给出一段指令序列，简单说明你是如何做到尽力转发的。

   无其他解决方案。

9. [P5 选做] 请详细描述你的测试方案及测试数据构造策略。

   见上。

10. [P5、P6 选做] 请评估我们给出的覆盖率分析模型的合理性，如有更好的方案，可一并提出。

P4

1. 阅读下面给出的 DM 的输入示例中（示例 DM 容量为 4KB，即 32bit × 1024字），根据你的理解回答，这个 addr 信号又是从哪里来的？地址信号 addr 位数为什么是 [11:2] 而不是 [9:0] ？

   addr信号来自ALU的输出，为了与输入的bit对齐。

2. 思考上述两种控制器设计的译码方式，给出代码示例，并尝试对比各方式的优劣。

指令对应的控制信号如何取值：在verilog代码中表现为每个控制信号若干行三目运算符，可读性较强，但是由于使用MUX执行速度较慢。

控制信号每种取值所对应的指令：在verilog代码中表现为每个控制信号的每个bit都是一组或门运算，可读性较差，但是只利用了一层或门，执行速度较快。

3. 在相应的部件中，复位信号的设计都是**同步复位**，这与 P3 中的设计要求不同。请对比**同步复位**与**异步复位**这两种方式的 reset 信号与 clk 信号优先级的关系。

   同步复位中clk优先，**异步复位**中reset优先。

4. C 语言是一种弱类型程序设计语言。C 语言中不对计算结果溢出进行处理，这意味着 C 语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持 C 语言，MIPS 指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi 与 addiu 是等价的，add 与 addu 是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中相关指令的 Operation 部分（详见文档 page 34、page 35）。

   由指令集可知，addi和addiu，add和addu的区别仅仅在于有符号时多了一个if条件语句判断是否抛异常，所以在忽略溢出的情况下这两对指令是等价的。

P3

1. 请大家指出单周期 CPU 所用到的模块中，哪些发挥状态存储功能，哪些发挥状态转移功能。
   - 状态存储：pc，ifu，grf，dm
   - 状态转移：alu，ext，controller
2. 现在我们的模块中 IM 使用 ROM，DM 使用 RAM，GRF 使用 Register，这种做法合理吗？ 请给出分析，若有改进意见也请一并给出。
   - IM使用ROM：指令在执行过程中只需取，不会改变，可以使用ROM
   - DM使用RAM：数据在执行过程中需要存取，故用RAM
   - GRF使用Register：寄存器速度快，且mips架构中不能同时对DM进行存取操作，需要寄存器中转
3. 在上述提示的模块之外，你是否在实际实现时设计了其他的模块？如果是的话，请给出介绍和设计的思路。

   暂无。

4. 事实上，实现 `nop` 空指令，我们并不需要将它加入控制信号真值表，为什么？

   nop的opcode为000000，与add/sub相同，但其操作的是0号寄存器，不会造成任何影响。

5. 阅读 Pre 的 “MIPS 指令集及汇编语言” 一节中给出的测试样例，评价其强度（可从各个指令的覆盖情况，单一指令各种行为的覆盖情况等方面分析），并指出具体的不足之处。
   - add/sub：边界情况未覆盖
     - beq：无向前跳转

附录

P5指令控制表：

funct	**100000**	**100010**	**100001**	**100011**	**n/a**
opcode	000000	000000	000000	000000	001101
	add	sub	addu	subu	ori
ExtOp[2:0]	x	x	x	x	ZEXT(0)
WD3Sel[2:0]	ALUout(0)	ALUout(0)	ALUout(0)	ALUout(0)	ALUout(0)
MemWrite!!!	0	0	0	0	0
ALUOp[4:0]	ADD(0)	SUB(1)	ADD(0)	SUB(1)	OR(3)
ALUSrc	WD2(0)	WD2(0)	WD2(0)	WD2(0)	EXTout(1)
A3Sel	rd(1)	rd(1)	rd(1)	rd(1)	rt(0)
RegWrite!!!	1	1	1	1	1
DMOp[2:0]	x	x	x	x	x
NPCOp[3:0]	ADD4(0)	ADD4(0)	ADD4(0)	ADD4(0)	ADD4(0)
CMPOp[2:0]	x	x	x	x	x

funct	**n/a**	**n/a**	**n/a**	**n/a**	**n/a**
opcode	100011	101011	000100	001111	000011
	lw	sw	beq	lui	jal
ExtOp[2:0]	SEXT(1)	SEXT(1)	SEXT(1)	ZEXT(0)	x
WD3Sel[2:0]	DMout(1)	x	x	ALUout(0)	PC4(2)
MemWrite!!!	0	1	0	0	0
ALUOp[4:0]	ADD(0)	ADD(0)	x	LUI(4)	x
ALUSrc	EXTout(1)	EXTout(1)	WD2(0)	EXTout(1)	x
A3Sel	rt(0)	x	x	rt(0)	ra(2)
RegWrite!!!	1	0	0	1	1
DMop[2:0]	LW_SW(0)	LW_SW(0)	x	x	x
NPCOp[3:0]	ADD4(0)	ADD4(0)	BRCH(1)	ADD4(0)	JAL(2)
CMPOp[2:0]	x	x	EQ(0)	x	x

funct	**001000**				
opcode	000000				
	jr				
ExtOp[2:0]	x				
WD3Sel[2:0]	x				
MemWrite!!!	0				
ALUOp[4:0]	x				
ALUSrc	x				
A3Sel	x				
RegWrite!!!	0				
DMop[2:0]	x				
NPCOp[3:0]	JR(3)				
CMPOp[2:0]	x				

学长博客

- https://github.com/DouyaBula/BUAA-CO-2022/tree/main
-