流水线处理器设计文档

一、 模块规格

1. Instruction Fetch Unit（取指令单元）

1) 端口定义

表 1 IFU 的端口定义

名称	方向	端口规格	功能描述
GO_HANDLER	input	[0:0]	传入当前是否跳转进入 handler 的决定信号
ERET	input	[0:0]	传入当前（IF/ID）是否为 eret 指令
EPC	input	[31:0]	传入跳回主程序的 EPC 地址
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
PCXcp	output	[0:0]	当前 PC 取指令异常信号

2) 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。

当 GO_HANDLER 有效时，下一时刻 PC 计数器改为 0x00004180。

当 ERET 有效时，下一时刻 PC 计数器改为 EPC 的值。

当 PC 计数器不是 4 的倍数，或者不在 IM 范围内时，PCXcp 输出 1。

2. General Register File（通用寄存器堆）

1) 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

2) 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）。

3. Arithmetic Logical Unit（算术逻辑单元）

1) 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
Overflow	output	[0:0]	加减法溢出信号（不一定最后导致异常）

2) 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）。

由于 beq 指令前移，故不再需要 Zero 信号。

当按照 MIPS 规则判断当前指令存在溢出时，Overflow 信号为 1。注意，OverflowXcp = Overflow & OverflowDect。其中 OverflowDect 为：当前指令是否需要检测溢出。

4. Data Memory（数据内存）

1) 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
MemCode	input	[3:0]	内存操作指令的类型
AddressXcp	output	[0:0]	内存/设备 操作异常信号

2) 功能说明

DM 由 4096 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[13:2]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 的写入操作（mem[Addr] = WD）当且仅当 clk 为上升沿且 WE=1 时，且无异常时生效。

当由 MemCode 所知的内存操作指令与操作的内存地址不符时（或地址范围不在内存、设备中时），抛出 AddressXcp 异常。

如：WORD 指令要求地址是 4 的倍数，HALF 指令要求地址是 2 的倍数等。

5. Immediate Calculator（立即数计算器）

1) 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

2) 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。

6. Controller（控制器）

1) 端口定义

表 6 Controller 的端口定义

名称	方向	端口规格	功能描述
Instr	input	[31:0]	32bit 指令
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。
OverflowDect	output	[0:0]	当前指令是否需要检测溢出
CP0Write	output	[0:0]	CP0 写使能

ERET	output	[0:0]	当前指令是否是 ERET
OpXcp	output	[0:0]	是否为非法 / 未识别指令

2) 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。

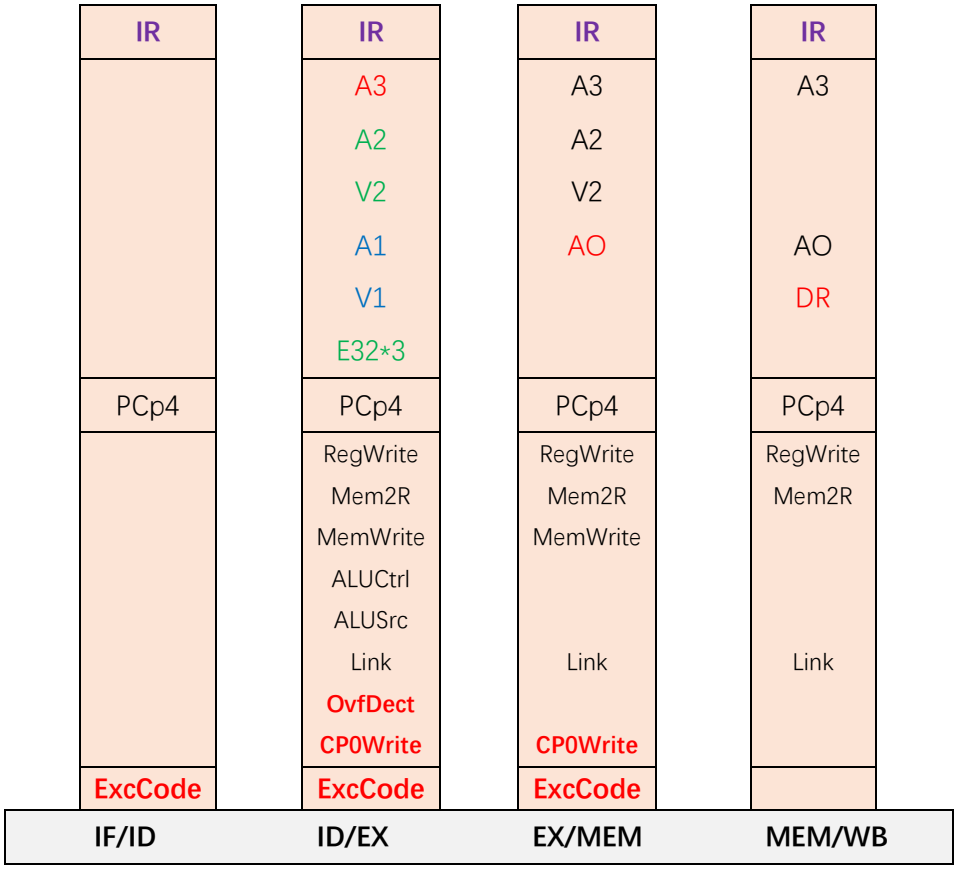
具体地，如何根据某条指令的机器码和其意义设计控制信号，见下文“控制器设计”中阐述。

二、 数据通路设计

1. Pipeline 各级概览

表 7 流水线各级组成

	IF	ID	EX	MEM	WB
PC	IFU	rRF EXT CMP CTRL	ALU	DM **DEV** **CP0**	wRF



2. 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、**DR@MEM/WB** 存在不同的连线方式，故需要建立五个普通 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	
M_DR_MEMWB	DMread	DEVread	CP0read	

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，**六个转发 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*	
MF_EPC	**EPC**	***V2@EXMEM***	***MF_ALU_B***

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。

3. Data Hazard 分析

表 10 T_{use}-T_{new} 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
		1	2	0	0	1	0	0	0	0

T_{use} @IF/ID	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 T_{use} 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 T_{new} 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。

MFC0：看成 lw。

MTC0：看成 sw。

ERET：看成 JR。

三、 控制器设计

1. 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	001111	100011	101011	000000
	addu	**subu**	**ori**	**lui**	**lw**	**sw**	**nop**
MemtoReg	0	0	0	0	1	x	0
MemWrite	0	0	0	0	0	1	0
Branch	0	0	0	0	0	0	x
ALUControl	010	110	001	010	010	010	xxx
ALUSrc	00	00	10	11	01	01	xx
RegDst	1	1	0	0	0	x	x
RegWrite	1	1	1	1	1	0	x
Jump	0						

Link	0
JR	0
OvflDetect	0
CP0Write	0
ERET	0

funct / **[25:21]**	/	/	001000	/	00000	00100	011000
opcode	000011	000010	000000	000100	010000		
	jal	**j**	**jr**	**beq**	**MFC0**	**MTC0**	**ERET**
MemtoReg	0	0	0	x	1	0	x
MemWrite	0	0	0	0	0	0	0
Branch	0	0	0	1	0	0	0
ALUControl	000	000	010	110	xxx	xxx	xxx
ALUSrc	00	00	00	00	xx	xx	xx
RegDst	0	0	0	x	0	x	x
RegWrite	1	0	0	0	1	0	0
Jump	1	1	0	0	0	0	0
Link	1	0	0	0	0	0	0
JR	0	0	1	0	0	0	0
OvflDetect	0						
CP0Write	0				0	1	0
ERET	0				0	0	1

funct / **[25:21]**	100000	100010
opcode	000000	000000
	add	**sub**
MemtoReg	0	0
MemWrite	0	0
Branch	0	0

ALUControl	010	110
ALUSrc	00	00
RegDst	1	1
RegWrite	1	1
Jump	0	0
Link	0	0
JR	0	0
OvflDetect	1	
CP0Write	0	
ERET	0	

2. 控制信号设计规则

1) MemtoReg

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

2) MemWrite

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

3) Branch

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

4) 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。

5) 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。

6) 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。

7) 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。

8) 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。

9) 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。

10) 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。

3. 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。

1) **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。

2) **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 |…;
```

四、 测试 CPU

1. 主程序

```
.text                Main.v
  # Interrupt TEST
  ori $t0, $0, 0x0000FC01 # Allow All Interrupt
  mtc0 $t0, $12
  addu $s0, $0, 0x00007F00 # Timer 1
  addu $s1, $0, 0x00007F10 # Timer 2
  ori $t0, $0, 7
  sw $t0, 4($s0)     # init = 7
  sw $t0, 4($s1)     # init = 7
  ori $t0, $0, 9
  sw $t0, 0($s0)     # start mode 0
  sw $t0, 0($s1)     # start mode 0

  # Long Operations Without Expection
  jal loadimm
   lui $s1, 233
   save:
   sw $t0, 0($s0)
   sw $t1, 4($s0)
   sw $t2, 8($s0)
   sw $t3, 12($s0)
   lui $s3, 3
   lui $s4, 4
   lw $t3, 0($s0)
```

```
lw $t2, 4($s0)
lw $t1, 8($s0)
lw $t0, 12($s0)
subu $t4, $t1, $t2
lui $s5, 5
lui $s6, 6
beq $t4, $t3, save
lui $s1, 556
lui $s2, 666
j next_test
lui $s1, 445
loadimm:
   ori $t0, $0, 7    # t0 = 7
   lui $t3, 1        # t3 = 65536
   ori $s0, $0, 4    # s0 = 4
   ori $t1, $t0, 50  # t1 = t0 or 50 = 55
   lui $s1, 0
   nop
   addu $t2, $t0, $t1   # t2 = t0 or t1 = 62
   lui $s2, 2
   nop
   subu $t4, $t3, $t2   # t4 = t3 or t3 = 65474

   jr $ra
   lui $s1, 334
next_test:
# Operation TEST
and $1, $0, $0          # EXP 10
lbu $t0, 200($0)        # EXP 10
```

```
# Memory TEST
ori $s0, $0, 1
lw $s1, 3($s0)          # noEXP
lw $s1, 1($s0)          # EXP 4
sw $31, 2($s0)          # EXP 5
sw $30, 7($s0)          # noEXP

# Overflow TEST
ori $t1, $0, 1          # $1 = 1
lui $t2, 0x7fff
ori $t2, $t2, 0xffff # $2 = INT_MAX
add $t3, $t1, $t2       # $2 =  1+INT_MAX (EXP 12)
sub $t3, $0, $t1        # $3 = -1
sub $t3, $t3, $t2       # $3 = -1-INTMAX (noEXP)
sub $t3, $t3, $t1       # $3 = -1-INTMAX-1 (EXP 12)
```

2. Handler1 及其预期结果

```
.ktext 0x00004180      # Handler 1: pass + reset
  mfc0 $t7, $14        # pass
  addu $t7, $t7, 4
  mtc0 $t7, $14

  addu $s2, $0, 0x00007F00 # Timer 1
  addu $s3, $0, 0x00007F10 # Timer 2
  ori $t0, $0, 9
  sw $0, 0($s2)  # CTRL: 0,0
  sw $0, 0($s3)
  sw $t0, 0($s2) # CTRL: 1,1
  sw $t0, 0($s3)
```

```
eret
ori $t0, $0, 0x0000FFFF
```

先发生两个硬件中断，之后发生异常：10、10、4、5、12、12（见注释）。每次发生中断/异常，Timer 重启，跳过 EPC，eret 后的 ori 不执行（而是执行硬件级 nop）。

3. Handler2 及其预期结果

```
.ktext 0x00004180       # Handler 2: forward & stall
  addu $t1, $0, 0x00003000#t1=3000
  ori $t2, $0, 4
  addu $t2, $t1, $t2        #t2=3004
  mtc0 $t2, $14             #epc = 3004
  sw $t2, 0($0)
  lw $t1, 0($0)
  mtc0 $t1, $14             #epc = 3004
  eret
```

完成 addu → mtc0 的转发，完成 lw → mtc0 → eret 的转发和暂停。

五、 思考题

- 什么是“硬件/软件接口”？

  传输硬件数据，并将硬件映射到内存地址的 Bridge、片选控制器和总线等。

- DM 在现代计算机中的位置？

  不在 CPU 内部，而是通过 NorthBridge 与 CPU 沟通。

  相当于一种高速设备。

- PrBE：

  作用是决定写入设备对于内存数据的哪些字节。

  对于存储的数据都是以 WORD 为单位的设备，无需 PrBE。

- 主程序及 Handler：

```
.text 0x00003000          # Main.v
  ori $t0, $0, 0x0000FC01 # SR: Allow All Interrupt
  mtc0 $t0, $12
  addu $s0, $0, 0x00007F00 # Timer 1
  addu $s1, $0, 0x00007F10 # Timer 2
  ori $t0, $0, 7
  sw $t0, 4($s0)          # init = 7
  sw $t0, 4($s1)          # init = 7
  ori $t0, $0, 9
  sw $t0, 0($s0)          # start mode 0
  sw $t0, 0($s1)          # start mode 0
  ……
  ……
  (while(True): pass)

.ktext 0x00004180      # Handler.v
  addu $s2, $0, 0x00007F00 # Timer 1
  addu $s3, $0, 0x00007F10 # Timer 2
  ori $t0, $0, 9
  sw $0, 0($s2)     # CTRL: 0,0
  sw $0, 0($s3)
  sw $t0, 0($s2)    # CTRL: 1,1
  sw $t0, 0($s3)
  eret
```

- 键盘鼠标的输入信号：

  低速设备通过南桥和北桥与 CPU 沟通。用中断信号让 CPU 响应。

  当键盘/鼠标触发一个动作时，设备发出一个中断信号。CPU 接收中断

信号后，根据不同的中断信号执行不同的中断处理程序，从而特异性的响应外设动作：将设备存入的寄存器 / 设备内存中的数据保存。