PART ONE 设计草稿

实现指令

```
RRCal Type:
    ADD, SUB, AND, OR, SLT, SLTU
RICal Type:
    ADDI, ANDI, ORI, LUI(AUI)
LM Type:
    LB, LH, LW
SM Type:
    SB, SH, SW
MD Type:
    MULT, MULTU, DIV, DIVU, MFHI, MFLO, MTHI, MTLO
B Type:
    BEQ, BNE
J Type:
    JAL, JR
NOP:
    NOP
```

通用寄存器冒险解决思路

对于我的CPU结构，根据以下四条原则，可以写出所有的通用寄存器冒险处理逻辑。

1. 只有D级和E级需要转发。
   只有D级需要延迟。
2. D级需要的冲突数据只会来自于E级，M级，W级（内部转发）。
   E级需要的冲突数据只会来自于M级，W级。
3. 对于不写入寄存器(RegWrite==0)的指令，我们将它的TnewD设为0(不产生)，将它的WriteReg也设为0。

   对于不读取寄存器的指令，我们将它的TuseD设为3(不使用)。
   这样直接使用AT法时就不会出现错误的延迟和转发。
4. 使用AT法
   如果Tnew(E/M/W)==0且寄存器(D/E)==WriteReg(E/M/W)，则需要转发。
   如果TuseD<TnewD，则需要延迟。
   其中，根据每一条指令的RegDataSrc，Tnew何时为0是确定的。参见下图。

乘除寄存器冒险解决思路

对于我的CPU结构，根据以下两条原则，可以写出乘除寄存器的冒险处理逻辑。

1. 只有处于D级的读取HILO型指令需要延迟。

   只有处于D级的写入HILO型指令需要覆盖。

2. 读取HILO型指令在延迟后可以直接读取HILO，无需再次转发。

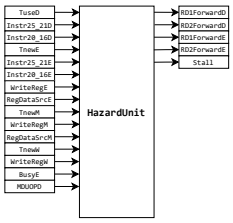
   写入HILO型指令直接覆盖原进程。

顶层设计图

绿色代表模块外置。

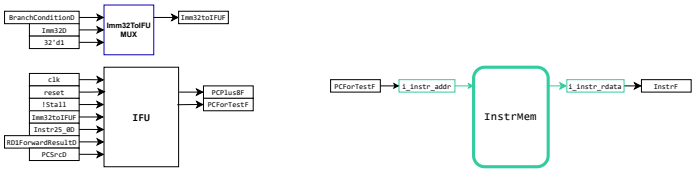
Hazard Unit

Handling Hazards



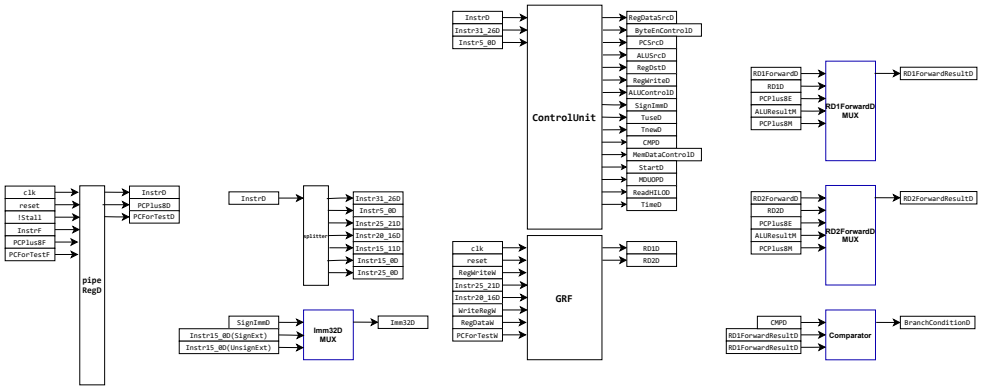
Level F

Instruction Fetch



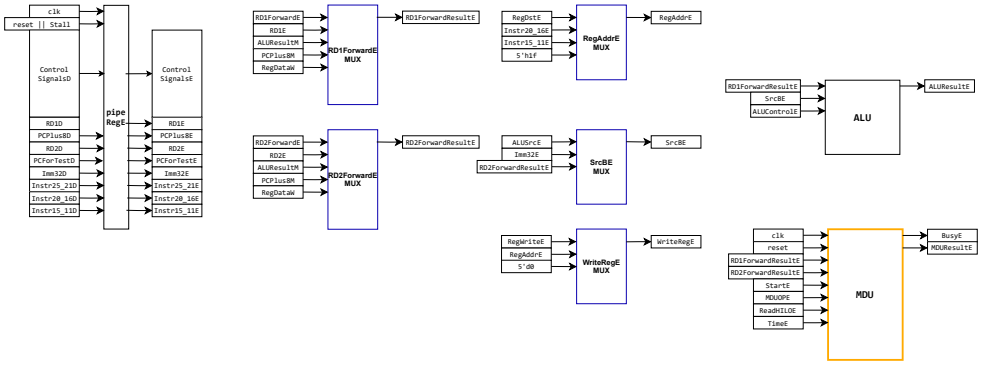
Level D

Instruction Decode



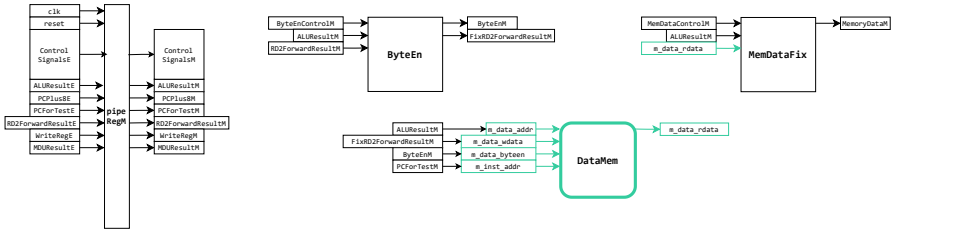
Level E

Excute



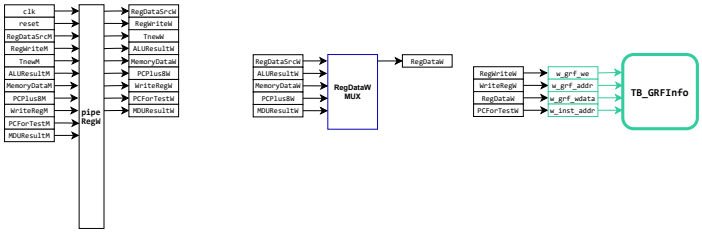
Level M

Memory Access



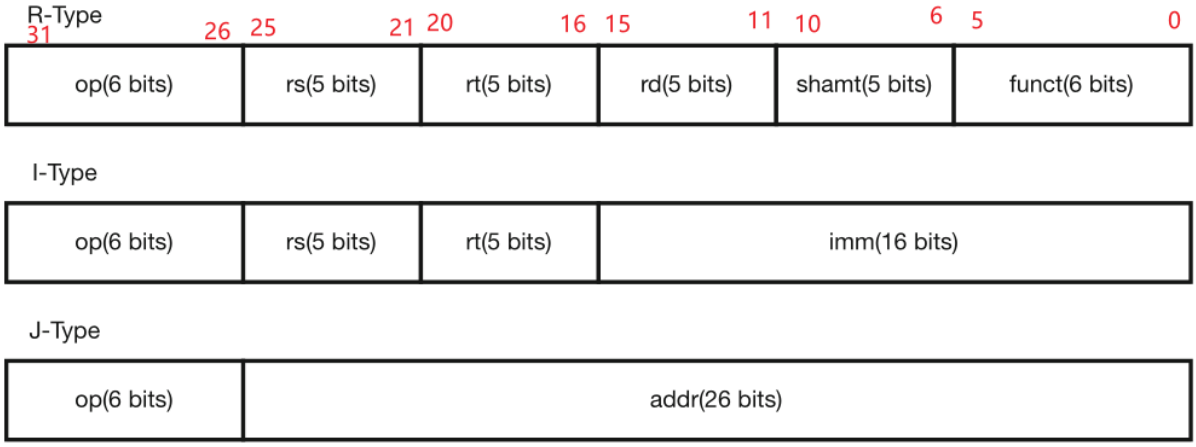
Level W

Register Write Back



编码表格

三类指令结构



TypeDecoder

RRCal Type	Opcode/Funct	RICal Type	Opcode/Funct	LM Type	Opcode/Funct
ADD	000000/100000	ADDI	001000	LB	100000
SUB	000000/100010	ANDI	001100	LH	100001
AND	000000/100100	ORI	001101	LW	100011
OR	000000/100101	LUI(AUI)	001111		
SLT	000000/101010				
SLTU	000000/101011				
SM Type	Opcode/Funct	MD Type	Opcode/Funct	B Type	Opcode/Funct
SB	101000	MULT	000000/011000	BEQ	000100
SH	101001	MULTU	000000/011001	BNE	000101
SW	101011	DIV	000000/011010		
		DIVU	000000/011011		
		MFHI	000000/010000		
		MFLO	000000/010010		
		MTHI	000000/010001		
		MTLO	000000/010011		
J Type	Opcode/Funct	NOP	0x00000000		
JAL	000011	NOP	0x00000000		
JR	000000/001000				

Hazard Unit

	TnewD	TnewE	TnewM	TnewW
写入ALUResult类	2	1	0	0
写入PCPlus8类	0	0	0	0
写入MemoryData类	3	2	1	0
写入MDUResult类	2	1	0	0

SignalDecoder_PCsrcDecoder

Instr	PCSrc		Instr	CMP
B Type			B Type	
BEQ	001		BEQ	000
BNE	001		BNE	001
J Type			Others	xxx
JAL	010			
JR	011			
Others	000			

SignalDecoder_ImmEXTDecoder

Instr	SignImm
RICal Type	
ADDI	1
ANDI	0
ORI	0
LUI(AUI)	1
LM Type	
LB	1
LH	1
LW	1
SM Type	
SB	1
SH	1
SW	1
B Type	
BEQ	1
BNE	1
Others	0

SignalDecoder_MemDecoder

Instr	ByteEnControl/Meaning		Instr	MemDataControl/Meaning
SM Type			LM Type	
SB	001/addr[1:0]		LB	001/addr[1:0]
SH	010/addr[1]		LH	010/addr[1]
SW	011/1111		LW	011/1111
Others	000/0000		Others	000/0000

SignalDecoder_RegDecoder

Instr	RegWrite*	RegDataSrc*	RegDst*
RRCal Type			
ADD	1	000	001
SUB	1	000	001
AND	1	000	001
OR	1	000	001
SLT	1	000	001
SLTU	1	000	001
RICal Type			
ADDI	1	000	000
ANDI	1	000	000
ORI	1	000	000
LUI(AUI)	1	000	000
LM Type			
LB	1	001	000
LH	1	001	000
LW	1	001	000
MD Type			
MULT			
MULTU			
DIV			
DIVU			
MFHI	1	010	001
MFLO	1	010	001
MTHI			
MTLO			
J Type			
JAL	1	011	010
JR			
Others	0	X(111)	X

SignalDecoder_TimeDecoder

Instr	Tuse	TnewD
RRCal Type		
ADD	1	2
SUB	1	2
AND	1	2
OR	1	2
SLT	1	2
SLTU	1	2
RICal Type		
ADDI	1	2
ANDI	1	2
ORI	1	2
LUI(AUI)	1	2
LM Type		
LB	1	3
LH	1	3
LW	1	3
SM Type		
SB	1	0(不产生)
SH	1	0(不产生)
SW	1	0(不产生)
MD Type		
MULT	1	0(不产生)
MULTU	1	0(不产生)
DIV	1	0(不产生)
DIVU	1	0(不产生)
MFHI	3(不使用)	2
MFLO	3(不使用)	2
MTHI	1	0(不产生)
MTLO	1	0(不产生)
B Type		

Instr	Tuse	TnewD
BEQ	0	0(不产生)
BNE	0	0(不产生)
J Type		
JAL	3(不使用)	0(已经产生)
JR	0	0(不产生)
NOP		
NOP	3(不使用)	0(不产生)

SignalDecoder_ALUDecoder

Instr	ALUControl/Meaning	ALUSrc
RRCal Type		
ADD	0000/Add	0
SUB	0001/Sub	0
AND	0010/And	0
OR	0011/Or	0
SLT	0100/Less Than?	0
SLTU	0101/Less Than U?	0
RICal Type		
ADDI	0000/Add	1
ANDI	0010/And	1
ORI	0011/Or	1
LUI(AUI)	0110/Add Upper Imm	1
LM Type		
LB	0000/Add	1
LH	0000/Add	1
LW	0000/Add	1
SM Type		
SB	0000/Add	1
SH	0000/Add	1
SW	0000/Add	1
Others	X	X

SignalDecoder_MDUDecoder

Instr	Start	MDUOP	ReadHILO	Time
MD Type				
MULT	1	0001	00	5
MULTU	1	0010	00	5
DIV	1	0011	00	10
DIVU	1	0100	00	10
MFHI	0	1111	10	0
MFLO	0	1111	01	0
MTHI	0	0101	00	0
MTLO	0	0110	00	0
Others	0	0000	00	0

PART TWO 测试方案

使用工具：魔改版Mars，Winmerge。

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

PART THREE 思考题

> 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？

答：因为乘除法部件有时延。可能HI、LO独立出来方便设计。

> 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。

答：二进制的**乘法器**（英语：**multiplier**）是数字电路的一种元件，它可以将两个二进制数相乘。乘法器是由更基本的加法器组成的。乘法器作为基本的功能单元电路被广泛的应用于各种的信号处理和变换电路中。可以使用一系列计算机算数技术来实现数字乘法器。大多数的技术涉及了对部分积（partial product）的计算（其过程和使用竖式手工计算多位十进制数乘法十分类似），然后将这些部分积相加起来。这一过程与小学生进行多位十进制数乘法的过程类似，不过在这里根据二进制的情况进行了修改。 *（来源Wikipedia）*

> 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？

答：

代码实现：

```
assign Stall = StallReg || (BusyE && MDUOPD == 4'b1111);
```

> 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）

答：采用字节使能信号处理写指令更清晰，更统一。

> 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？

答：不是，是按字读。在按字节寻址的情况下。

> 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？

答：根据功能分类指令、遵照低内聚高耦合原则设计模块。特点是清晰统一，帮助是实现逻辑简单。

> 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？

答：相比于P5，我在P6中遇到了MUL和MTHI、MFHI一类的冲突。解决方法是遇到MFHI时进行暂停，遇到MTHI进行覆盖。测试样例：

```
ori $t1, $zero, 0x114
ori $t2, $zero, 0x514
mul $t1, $t2
mfhi $s1
mul $t1, $t2
mthi $s1
```

> 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。

答：手动构造。首先对每条指令单独构造一系列数据，确保指令功能正确。然后按照构造一系列会产生冒险和转发的数据，确保转发和暂停正确。最后白嫖github上的数据。