CO-P6-CPU设计文档

#Verilog #CPU

---

1 Notation

> 命名规则参考:
> CO教程要求
> Digital Design and Computer Architecture

1.1 DataBus

- Instr: 32 位指令信号，将被分为
  - opcode
  - rs
  - rt
  - rd
  - shamt
  - funct
  - imm16
  - imm26
- pc: 32 位程序计数器信号
- npc: 32位下一时钟刻程序计数器信号
- pc4：始终为PC + 4

- ExtImm：EXT输出完成扩展后的立即数
- RegAddr: GRF 5 位写入地址
- RegData: GRF 32 位写入数据
- srcA：ALU写入数据源A
- srcB：ALU写入数据源B
- outC：ALU输出数据源C
- ALUsig：ALU特殊计算标志位，处理Zero或其余标志信号
- MemAddr: DM 32 位写入地址
- MemData: DM 32 位写入数据
- ReadData：DM 32位输出数据
- **流水级相关**：使用@F @D @E @M @W表示某一阶段的数据，例如某级的指令编码为Instr@M

1.2 ControlBus

- RegWrite: GRF 写入控制信号
- MemWrite: DM 写入控制信号
- ALUOp：ALU操作信号
- NPCOp：NPC操作信号
- EXTOp：EXT操作信号
- RegDst：GRF.A3选择数据源
- RegSrc：GRF选择数据源
- ALUSrc：ALU选择数据源

1.3 MoveForward

规定所有经过转发的RD1/RD2数据源使用T作为前缀，标注为经历转发的真实的数据，包括：

- T_RD1_D：D级经过转发后的真实GRF[rs]
- T_RD2_D：D级经过转发后的真实GRF[rt]
- T_RD1_E：E级经过转发后的真实GRF[rs]
- T_RD2_E：E级经过转发后的真实GRF[rt]
- T_RD2_M：M级经过转发后的真实GRF[rt]

---

2 Requirements

- 支持指令集

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

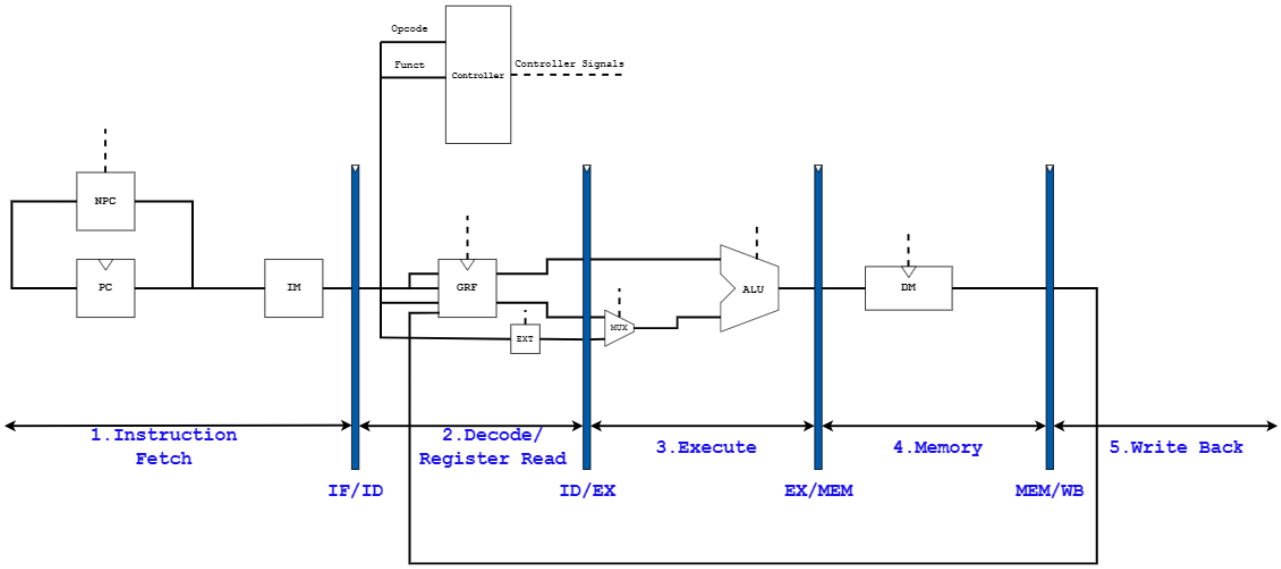
- 直接通过顶层模块 mips.v 的 output 端口传出相应信号，**不允许出现 `display` 语句**，具体要求见“在线测试相关说明”小节。
- 要求存储器外置，即将 IM 和 DM 放置在 CPU 之外。P6 的 IM 和 DM 两个模块被内置于官方评测的 testbench 中（官方使用的 tb 已在“存储器外置”小节公开），**不再需要**大家自行实现 IM 和 DM 模块。
- 需有单独的**乘除法模块**和**数据扩展模块**

- 必须严格按照模块的端口定义，顶层文件模块名为 mips，需要考虑延迟槽。
- 在部件中，reset 信号的优先级高于其它控制信号，且设计为**同步复位**。保证自动评测在 CPU 运行前进行复位。
- 可使用官方 tb 在本地测试（下载链接），**Verilog 源代码中任何位置不允许出现 `$display` 语句。**
- 对于使用 ISE 的同学，请**谨慎使用** 64 位除法运算，ISE 的实现可能存在 bug，63 位及以下均正确。
- Verilog 文件打包要求：在**不超过**最大压缩包提交限制的情况下，全部 .v 文件放在同一文件夹下压缩成 zip 格式后提交。

---

3 Heirarchy Designation

3.1 Buses and Arrangements



3.2 WorkFlow

自底向上工程：

1. 实现MultDiv BE DP模块
2. 实例化并修改数据通路
3. 修改mips.v顶层模块接线
4. 修改IFU和DM符合规格
5. 实现控制器指令分类与控制信号生成
6. 调整转发阻塞处理器

确定数据通路:

- 根据指令进行数据通路确定，这里可能涉及到 (修改元件 / 增加元件 / 增加通路)

生成控制信号:

- 根据指令填表生成控制信号矩阵，可能涉及到 (增加新的控制信号 / 控制信号意义扩展)注意加入`head.v`

确定**转发数据通路**:

- 看最后W阶段，RegData的**写回数据源**有哪些（写寄存器数据有**哪些源-RegSrc**）
- 根据写回数据源，往回找每一级流水寄存器包含哪些写回数据（**E级M级W级写回数据**）
- 确定每一级流水SRCMUX的数据源，通过CU传出信号，搞定发送者MUX

- 在`MFSCU.v`中加入数据转发信号

计算**Tuse和Tnew**填表:

- Tuse：指令在D级时，还要多少个cycle才会用到
- Tnew：只用考虑**Tuse_E**，当前指令在E级时，还有多少个Cycle后，需要**新鲜的寄存器写回数据**会进入某一级流水线寄存器，按生成元件可分为（NPC-0，ALU-1，DM-2）
- 在`Controller.v`中加入Tnew和Tuse_E

代码模块接线注意事项：
对于所有input的数据，只连接有**T标记的数据源**（经过转发）

---

4 Module Designation

4.1 Module List

> 采取分布式译码

- mips
- PC_REG
- NPC
- D_REG
  - D_CU (CU)
  - GRFs
  - EXT
  - CMP

- E_REG
  - E_CU (CU)
  - ALU
  - MultDiv
- M_REG
  - M_CU (CU)
  - BE
  - DP
- W_REG
  - W_CU (CU)
- CU
- MFC
- STALL

4.2 mips

端口信号/功能表

信号名	方向	描述
clk	I	时钟信号
reset	I	同步复位信号
i_inst_rdata [31:0]	I	i_inst_addr 对应的 32 位指令
m_data_rdata [31:0]	I	m_data_addr 对应的 32 位数据

信号名	方向	描述
i_inst_addr [31:0]	O	需要进行取指操作的流水级 PC（一般为 F 级）
m_data_addr [31:0]	O	数据存储器待写入地址
m_data_wdata [31:0]	O	数据存储器待写入数据
m_data_byteen [3:0]	O	字节使能信号
m_inst_addr [31:0]	O	M 级 PC
w_grf_we	O	GRF 写使能信号
w_grf_addr [4:0]	O	GRF 中待写入寄存器编号
w_grf_wdata [31:0]	O	GRF 中待写入数据
w_inst_addr [31:0]	O	W 级 PC

必须在 Verilog HDL 设计中建立这个**顶层模块**，不允许修改模块名称、端口各信号以及变量的名称 / 类型，**代码中任何地方不允许包含 `display` 语句。**

4.2 PC_REG

端口信号/功能表

信号	方向	描述
clk	I	时钟信号

信号	方向	描述
reset	I	**同步**复位
stall	I	冻结信号
npc[31:0]	I	下一个指令的地址
pc[31:0]	O	当前指令地址

特别说明

要求数据范围约束：

1. pc，npc 为 **0x0000_3000 ~ 0×0000_6FFF**
2. IM规格：16KiB（4096 × 32bit）→ 2^{12}
3. 初始&复位地址：**0x0000_3000**
4. 指令按照**字为单位**进行fetch，取出addr[11:2]
5. 要求：“ROM 内部的起始地址是从 0 开始的，即 ROM 的 0 位置存储的是 PC 为 0x00003000 的指令，每条指令是一个 32bit 常数。”
6. 要求自行实现指令读入：要求使用`$readmemh`读入`code.txt`(说明见系统任务)

实现思路

本质上就是一个PC寄存器，不再需要自行设计IM
依旧实现地址映射：进入的-0x0000_3000，对输出的+0x0000_3000
注意一下：

1. 同步复位的复位值；

2. 取出地址的逻辑；
3. 需要一个`initial`块来执行`$readmemh`

4.3 D_REG

端口信号/功能表

信号	方向	描述
clk	I	时钟信号
reset	I	**同步**复位
F_Instr[31:0]	I	新指令
F_pc[31:0]	I	新指令pc值
Cond	I	条件跳转信号
D_Instr[31:0]	O	当前指令Instr
D_pc[31:0]	O	当前指令pc值
D_Imm16[15:0]	O	Instr分线Imm16@D
D_Imm26[25:0]	O	Instr分线Imm26@D

4.4 NPC

端口信号/功能表

信号	方向	描述
NPCOp[1:0]	I	操作控制码
pc_F[31:0]	I	当前F级的pc值

信号	方向	描述
pc_D[31:0]	I	当前D级的pc值
Imm16[15:0]	I	16位立即数
Imm26[25:0]	I	j指令26位伪地址
Addr[31:0]	I	寄存器相关直接寻址
npc[31:0]	O	下一个pc
pc4[31:0]	O	常态输出PC+4的值

NPCOp	功能	描述
2'b00	PC+4	npc = 顺序执行F_pc + 4
2'b01	PC相对寻址	npc = D_pc + 4 + SignExt(Imm16 | 00)
2'b10	伪直接寻址	npc = D_pc[31:28] | Imm26 | 00
2'b11	寄存器寻址	npc = Addr

> 似乎应该是取D_pc的高位？？
> 好像无所谓吧，因为最高四位都一样；还是需要确定下

实现方法

需要根据Controller给出的控制信号，作为一个MUX选择四路中一路进行输出

4.5 GRF

端口信号/功能表

信号	方向	描述
clk	I	时钟信号
reset	I	**同步**复位信号
RegWrite	I	写入使能信号
A1[4:0]	I	Src of RD1
A2[4:0]	I	Src of RD2
RegAddr[4:0]	I	写入寄存器地址
RegData[31:0]	I	写入寄存器堆的数据
pc[31:0]	*I*	*当前程序计数器值(辅助评测)*
RD1[31:0]	O	输出1
RD2[31:0]	O	输出2

序号	功能	描述
1	写入	当**RegWrite为1**且时钟**上升沿**时，更新GRF[A3]的数据为WD
2	输出	持续输出本上升沿记录的A1/A2数据，从RD1/RD2输出

特别说明

1. 0号寄存器无法被更改值，始终为0
2. 对于GRF而言，如果成功写入了数据（posedge clk，reset = 0，RegWrite = 1)

需要按照格式（请注意空格）输出：

```
$display("%d@%h: $%d <= %h", $time, WPC,
Waddr, WData);
```

`WPC`为指令存储地址(pc)，`Waddr`为写入寄存器地址(RegWrite)，`WData`为写入寄存器值(RegData)

实现思路

1. 使用`always @(posedge clk)`进行时序逻辑建模即可
2. **同步复位模式**
3. 注意下复合逻辑关系 + 辅助评测输出
4. 关注寄存器堆的定义：`reg [31:0] GRF[0:31]`

4.6 EXT

端口信号/功能表

信号	方向	描述
EXTOp[1:0]	I	2'b00则为ZeroEXT，2'b01则为SignEXT，2'10为LUI
Imm16[15:0]	I	16位立即数
ExtImm[31:0]	O	扩展后的Imm32

实现方法

先做两种扩展，之后直接使用assign语句配合三目运算符即可

4.7 CMP

端口信号/功能表

信号	方向	描述
C1[31:0]	I	比较数据源1
C2[31:0]	I	比较数据源2
CMPOp[2:0]	I	选择比较类型
Cond	O	比较结果

CMPOp	功能	描述
3'b000	BEQ	(C1 == C2) ? 1'b1 : 1'b0
3'b001	BNE	(C1 != C2) ? 1'b1 : 1'b0

实现方法

直接assign判断即可

4.8 E_REG

端口信号/功能表

信号	方向	描述
clk	I	时钟信号
reset	I	**同步**复位
D_Instr[31:0]	I	新指令
D_pc[31:0]	I	新指令pc值
RD1[31:0]	I	GRF.RD1
RD2[31:0]	I	GRF.RD2
pc4[31:0]	I	jal中F_pc+4
ExtImm[31:0]	I	EXT.ExtImm
E_Instr[31:0]	O	当前指令Instr
E_pc[31:0]	O	当前指令pc值
E_pc4[31:0]	O	jal中F_pc+4
E_RD1[31:0]	O	GRF.RD1@E
E_RD2[31:0]	O	GRF.RD2@E
E_ExtImm[31:0]	O	EXT.ExtImm@E

4.9 ALU

端口信号/功能表

信号	方向	描述
srcA[31:0]	I	数据源SrcA

信号	方向	描述
srcB[31:0]	I	数据源SrcB
ALUOp[4:0]	I	选取ALU操作
outC[31:0]	O	计算结果

ALUOp	功能	描述
5'h00	ADD加法	C = A + B
5'h01	SUB减法	C = A - B
5'h02	OR或	C = A | B
5'h03	LUI高位	C = B，由EXT执行扩展，仅仅借用数据通路
5'h04	AND与	C = A & B
5'h05	SLT	C = (A < B) ? 1'b1 : 1'b0 处理为有符号数
5'h06	SLTU	C = (A < B) ? 1'b1 : 1'b0 处理为有符号数

实现思路

没什么太需要注意的，翻译即可
ALUOp设置成5-bit的为之后扩展指令做准备

MultDiv

端口信号/功能表

信号	方向	描述
Start	I	乘除法启动信号
MDOp[2:0]	I	操作信号
MDWrite	I	MD元件写使能
Req	I	全局异常信号，出现时不启动乘除运算
MDSel	I	输出选择信号,1'b0为lo,1'b1为hi
MD1[31:0]	I	乘法源1/被除数
MD2[31:0]	I	乘法源2/除数
Busy	O	模拟乘除法运算周期，置为高位说明正在工作
MDout[31:0]	O	乘法高位/除法余数

MDOp	功能	描述
3'b000	mtlo	移动MD1(reg[rs])至寄存器lo
3'b001	mthi	移动MD1(reg[rs])至寄存器hi
3'b010	mult	执行乘法操作 (5周期)
3'b011	multu	执行无符号乘法操作 (5周期)
3'b100	div	执行除法操作 (10周期)
3'b101	divu	执行无符号除法操作 (10周期)

MDSel	功能	描述
0	mflo	移动lo值到reg[rd]
1	mfhi	移动hi值到reg[rd]

乘除模块行为约定如下：

- 自 Start 信号有效后的第 1 个 clock 上升沿开始，乘除法部件开始执行运算，同时将 Busy 置位为 1。
- 在运算结果保存到 HI 寄存器和 LO 寄存器后，Busy 位清除为 0。
- 当 Busy 信号或 Start 信号为 1 时，`mult, multu, div, divu, mfhi, mflo, mthi, mtlo` 等乘除法相关的指令均被阻塞在 D 流水级。
- 数据写入 HI 寄存器或 LO 寄存器，均只需 1 个时钟周期。

4.10 M_REG

端口信号/功能表

信号	方向	描述
clk	I	时钟信号
reset	I	**同步**复位
E_Instr[31:0]	I	新指令
E_pc[31:0]	I	新指令pc值
E_pc4[31:0]	I	pc + 4数值@E
E_RD2[31:0]	I	GRF.RD2@E

信号	方向	描述
E_outC[31:0]	I	ALU.outC@E
E_MDout[31:0]	I	MultDiv元件输出
E_Tnew[3:0]	I	E级的Tnew
M_Instr[31:0]	O	当前指令Instr
M_pc[31:0]	O	当前指令pc值
M_pc4[31:0]	O	pc + 4数值@M
M_RD2[31:0]	O	GRF.RD2@M
M_outC[31:0]	O	ALU.outC@M
M_MDout[31:0]	O	MultDiv元件输出流水
M_Tnew[3:0]	O	M级的Tnew

BE (Bit-Enable)

端口信号/功能表

信号	方向	描述
MemWrite	I	DM写使能信号
MemAddr[31:0]	I	32位DM地址
MemData[31:0]	I	DM中取出的数据
BEOp[1:0]	I	当前哪种类型存储sw/sh/sb
StoreData[31:0]	O	要写入DM的32位数据
ByteWrite[3:0]	O	4-bit，1代表对应位置写使能

BEOP	功能	描述
2'b00	SW	4'b1111 & {4{MemWrite}}
2'b01	SH	看Addr[1]，决定到底是4'b1100 & {4{MemWrite}} or 4'b1100 & {4{MemWrite}}
2'b10	SB	看Addr[1:0]，确定ByteWrite输出

DP (Data-Processor)

端口信号/功能表

信号	方向	描述
DPOp[2:0]	I	选择数据扩展方式
ReadData[31:0]	I	DM的32位输出数据
MemAddr[31:0]	I	只关心MemAddr[1:0]
LoadData[31:0]	O	向寄存器中load的数据值

信号	功能	描述
3'b000	不扩展	
3'b001	符号扩展	对8位数据进行符号扩展
3'b010	符号扩展	对16位数据进行符号扩展

4.12 W_REG

端口信号/功能表

信号	方向	描述
clk	I	时钟信号
reset	I	**同步**复位
M_Instr[31:0]	I	新指令
M_pc[31:0]	I	新指令pc值
M_pc4[31:0]	I	pc + 4数值@M
M_outC[31:0]	I	ALU.outC@M
M_MDout[31:0]	I	MultDiv.MDout@M
M_LoadData[31:0]	I	DP.LoadData@M
M_Tnew[3:0]	I	M级的Tnew
W_Instr[31:0]	O	当前指令Instr
W_pc[31:0]	O	当前指令pc值
W_pc4[31:0]	O	pc + 4数值@W
W_LoadData[31:0]	O	DP.LoadData@M
W_outC[31:0]	O	ALU.outC@W
W_MDout[31:0]	O	MultDiv.MDout@W
W_Tnew[3:0]	O	W级的Tnew

4.13 Controller

端口信号/功能表

信号	方向	描述
Instr[31:0]	I	需要译码的指令
Cond	I	条件跳转，判断是否条件成立
NPCOp[1:0]	O	NPC操作信号
EXTOp[1:0]	O	EXT操作信号
CMPOp[2:0]	O	CMP控制信号
RegWrite	O	GRF写入控制信号
RegDst[1:0]	O	GRF选择写入寄存器
RegSrc[1:0]	O	GRF选择RegData数据源
ALUSrc	O	ALU选择数据源
ALUOp[2:0]	O	ALU操作信号
MDOp[2:0]	O	MultDiv元件操作
MemWrite	O	DM 写入控制信号
BEOp[1:0]	O	BE存储指令类型
DPOp[2:0]	O	DP对内存值处理类型
Tnew[3:0]	O	E_Tnew值
Tuse_RS[3:0]	O	Tuse_RS值
Tuse_RT[3:0]	O	Tuse_RT值
Start	O	开始进行乘除法运算

序号	信号	功能
1	RegWrite	0禁止，1允许写入
2	MemWrite	0禁止，1允许写入
3	ALUOp[2:0]	ALU操作信号
4	NPCOp[1:0]	NPC操作信号
5	EXTOp[1:0]	00零扩展，01符号扩展，10LUI
6	CMPOp[2:0]	CMP控制信号，3'b000代表beq
7	RegDst[1:0]	2'b00选择rd，2'b01选择rt，2'b10选择0x1f($ra)
8	RegSrc[1:0]	2'b00选择ALU.C，2'b01选择DM.ReadData，2'b10选择PC+4
9	ALUSrc	0选择RD2，1选择ExtImm

其中Tuse_RS与Tuse_RT会在D级流水中的CU使用
而Tnew会生成E级流水的T_new并且随着流水线逐级寄存器传递

内部实现基于指令分类（对应**相同数据通路**，因此同类指令可能会出现Op不同，但是**MUX的选择信号一定相同**）

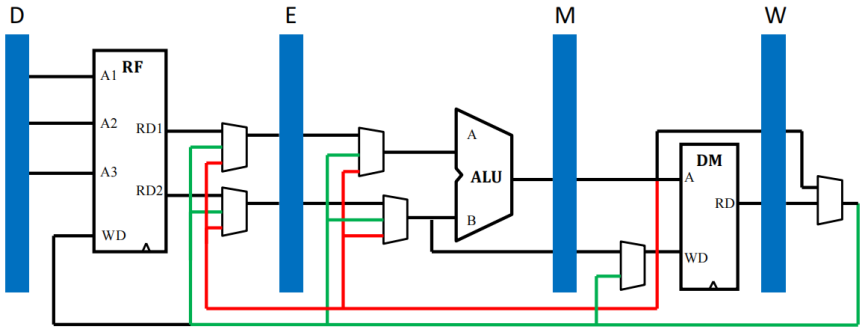
```
// Cal_reg_reg
add, sub, and, or, slt, sltu,

// Cal_Imm
lui, addi, andi, ori

// Load
lb, lh, lw

// Store
sb, sh, sw

// MultDiv
mult, multu, div, divu, mfhi, mflo, mthi,
mtlo

// B-type
beq, bne

// J_Link
jal

// J_reg
jr
```

4.14 MFSCU

转发

- 最新结果：可能出现在M和W
- 使用结果：D、E、M各级可能的位置
  - D级：RS寄存器值、RT寄存器值
  - E级：ALU的A、B；RT寄存器值
  - M级：DM的WD

	M级ALU计算结果	W级回写结果
D级被转发	RS寄存器值 RT寄存器值	RS寄存器值 RT寄存器值
E级被转发	ALU的A ALU的B RT寄存器值	ALU的A ALU的B RT寄存器值
M级被转发		DM的WD



67

遍历所有转发情况：

转发源	接收者
E_REG	D_GRF(RD1/RD2)
M_REG	D_GRF(RD1/RD2)
M_REG	E_ALU(srcA/srcB)
W_REG	D_GRF(RD1/RD2)
W_REG	E_ALU(srcA/srcB)
W_REG	M_DM(WD)

其中，W_REG向D_GRF的转发可以依靠GRF内部转发实现，降低转发模块复杂度

> 为什么引入E to D的转发?
> 保证全速流水，对于LUI指令，E级就已经算出来了LUI结果

因此，我们的转发模块需要实现的功能为：

- E to D
- M to D
- M to E
- W to E
- W to M

转发条件：

1. 转发源RegWrite信号有效
2. 转发源RegAddr与rs/rt一致
3. RegAddr非0号寄存器

转发设计构思：
暴力转发法，先假设所有数据冒险情况都可以通过转发解决，先设计转发数据通路，并且不考虑是否Tnew = 0。

> 这样的方法是合理的，因为错误的转发值在执行前一定会被正确的转发值覆盖
> 就算加入阻塞后，由于其实插入的值为nop，转发对冒险并不存在影响

转发层次：

1. 从源出发，选择需要转发什么(SrcMUX)
2. 到接受者，确定需要接受什么(DstMUX)
3. 考虑数据优先级(MFCU)

转发源SRC_MUX

考虑接受者真正需要的是什么，应该转发什么
例如对于M级：流水寄存器有ALU.outC@M，pc4@M，需要根据指令类型，确定我需要转发重写什么。

这里选择转发源的操作其实与本级指令CU译码出来的控制信号一致

我们不关心SRC_MUX的输出到底是哪一种信号，
我们只知道：其输出应该是**一个“新鲜”的rs/rt的数据**

接受源DST_MUX

考虑从当前接受的多个信号源中，用真正的新鲜的数据替代老旧的数据，这个控制信号需要依赖MFCU生成

MFSCU

流水级越低，数据越“新鲜”，优先使用低流水级数据

寄存器阻塞

使用A-T分析法易知，当且仅当出现Tuse < Tnew时才需要阻塞
列表分析指令的Tuse和Tnew，

并基于此给出策略矩阵

并且需要满足前提条件：

- 指令对应寄存器相等
- 需要进行写寄存器
- 寄存器非0

```
96      wire STALL_RS_E = (Tuse_RS < Tnew_E) && (RegAddr_E != 5'd0) && (RegAddr_E == rs_D) && (RegWrite_E);
97      wire STALL_RS_M = (Tuse_RS < Tnew_M) && (RegAddr_M != 5'd0) && (RegAddr_M == rs_D) && (RegWrite_M);
98      wire STALL_RS_W = (Tuse_RS < Tnew_W) && (RegAddr_W != 5'd0) && (RegAddr_W == rs_D) && (RegWrite_W);
99
100     wire STALL_RT_E = (Tuse_RT < Tnew_E) && (RegAddr_E != 5'd0) && (RegAddr_E == rt_D) && (RegWrite_E);
101     wire STALL_RT_M = (Tuse_RT < Tnew_M) && (RegAddr_M != 5'd0) && (RegAddr_M == rt_D) && (RegWrite_M);
102     wire STALL_RT_W = (Tuse_RT < Tnew_W) && (RegAddr_W != 5'd0) && (RegAddr_W == rt_D) && (RegWrite_W);
103
104     wire STALL_RS = STALL_RS_E | STALL_RS_M | STALL_RS_W;
105     wire STALL_RT = STALL_RT_E | STALL_RT_M | STALL_RT_W;
106
107     assign stall = STALL_RS | STALL_RT;
108
109 endmodule
110
```

乘除单元阻塞

约定： 当 Busy 信号或 Start 信号为 1 时，`mult, multu, div, divu, mfhi, mflo, mthi, mtlo` 等乘除法相关的指令均被阻塞在 D 流水级。

端口信号/功能表

信号	方向	描述
Start	I	乘除单元启动信号
Busy	I	乘除单元工作信号
Instr_D[31:0]	I	
Instr_E[31:0]	I	
Instr_M[31:0]	I	
RegAddr_E[4:0]	I	当前E级指令回写寄存器地址/无需回写则置0
RegWrite_E	I	E级指令写寄存器信号
RegAddr_M[4:0]	I	当前M级指令回写寄存器地址/无需回写则置0
RegWrite_M	I	M级指令写寄存器信号
RegAddr_W[4:0]	I	当前W级指令回写寄存器地址/无需回写则置0
RegWrite_W	I	W级指令写寄存器信号
Tuse_RS[3:0]	I	D级Tuse_RS
Tuse_RT[3:0]	I	D级Tuse_RT
Tnew_E[3:0]	I	E级流水的Tnew
Tnew_M[3:0]	I	M级流水的Tnew
Tnew_W[3:0]	I	W级流水的Tnew
D_RS_SEL[1:0]	O	D级RS的接受者MUX选择信号
D_RT_SEL[1:0]	O	D级RT的接受者MUX选择信号
E_RS_SEL[1:0]	O	E级RS的接受者MUX选择信号

信号	方向	描述
E_RT_SEL[1:0]	O	E级RT的接受者MUX选择信号
M_RT_SEL[1:0]	O	M级RT的接受者MUX选择信号
stall	O	阻塞信号

---

6 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？

> 因为乘除法需要的时钟周期需要5/10个时钟周期，而ALU运算仅需要一个

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。

> 乘法器使用几个小位数的乘法器来做，然后每个周期对几位来做最后加起来
> 除法会使用“试商法”，在一定周期内试探商的值最后形成结果

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？

> 在阻塞转发控制器中单独考虑Busy信号以及对D级指令进行解码；
> 在MD乘法器中使用一个count来作为计数器，进行周期的计数

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）

> 对于lw、lh、ls指令可以使用统一的形式进行编写，同时独热码的特性使其共享信号含义，以字节为单位非常清晰知道字节写入情况

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，而是整个字，这是为了统一化存储，同时如果不按字节使能势必要增长数据通路。

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？

> 在代码中使用头文件`head.v`并大量编写宏
> 以控制信号为驱动的控制处理，这样极大的简化
> 指令分类生成控制信号

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？

> 主要是加入乘法器相关操作的冲突

```
mult $3,$4
div $3,$2
mfhi $11
mflo $12
mthi $3
mtlo $2
mfhi $11
mflo $12
```

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。

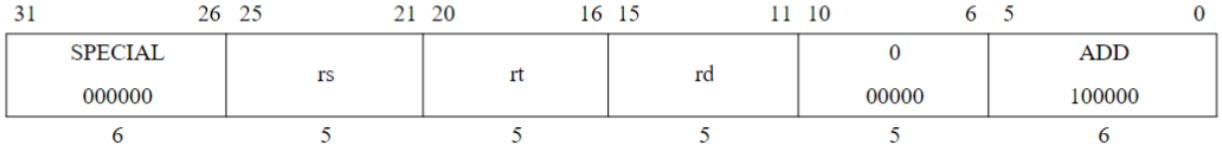
使用自己搭建的评测机，已在讨论区开源。

---

7 Reference

add

Add Word **ADD**



Format: `ADD rd, rs, rt` MIPS32

Purpose:

To add 32-bit integers. If an overflow occurs, then trap.

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* to produce a 32-bit result.

- If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

None

Operation:

```
temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp
endif
```

Exceptions:

Integer Overflow

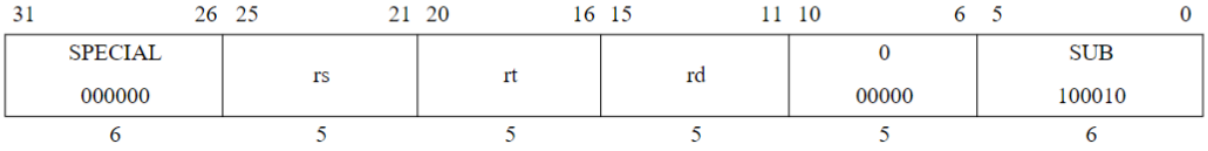
Programming Notes:

ADDU performs the same arithmetic operation but does not trap on overflow.

---

sub

Subtract Word SUB



Format: `SUB rd, rs, rt` MIPS32

Purpose:

To subtract 32-bit integers. If overflow occurs, then trap

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* to produce a 32-bit result. If the subtraction results in 32-bit 2's complement arithmetic overflow, then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

None

Operation:

```
temp ← (GPR[rs]31||GPR[rs]31..0) - (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp31..0
endif
```

Exceptions:

Integer Overflow

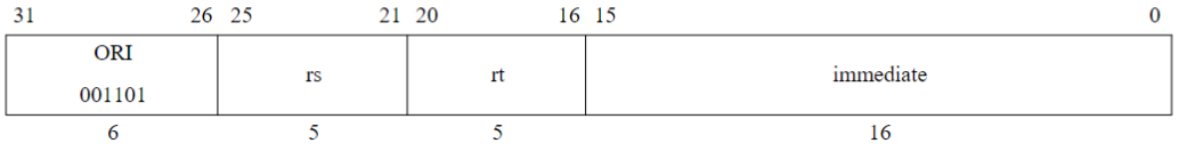
Programming Notes:

SUBU performs the same arithmetic operation but does not trap on overflow.

---

ori

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)
```

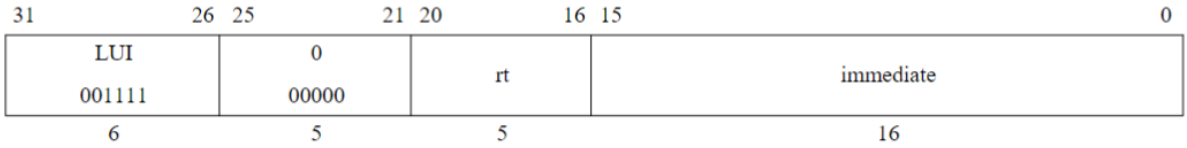
Exceptions:

None

---

lui

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:

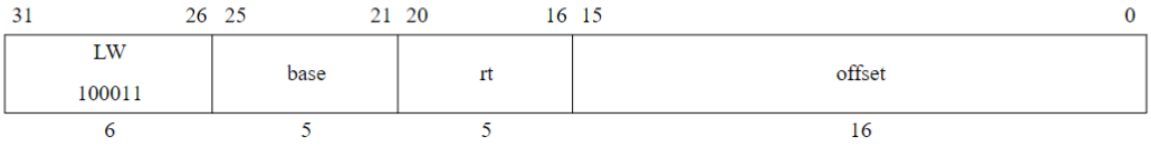
```
GPR[rt] ← immediate || 0^16
```

Exceptions:

None

lw

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:

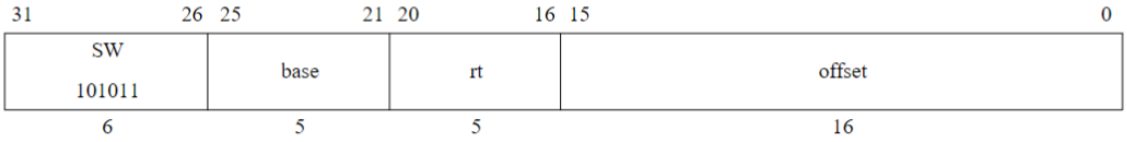
```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
memword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt]← memword
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error, Watch

sw

Store Word SW



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:

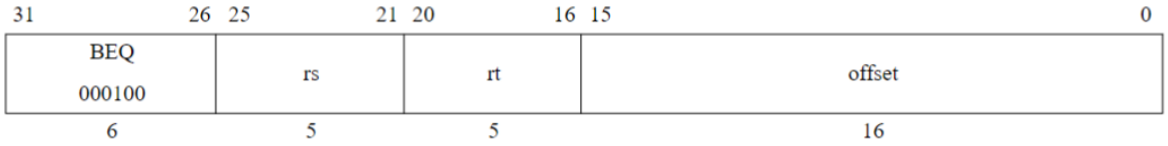
```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
dataword← GPR[rt]
StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

---

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
```

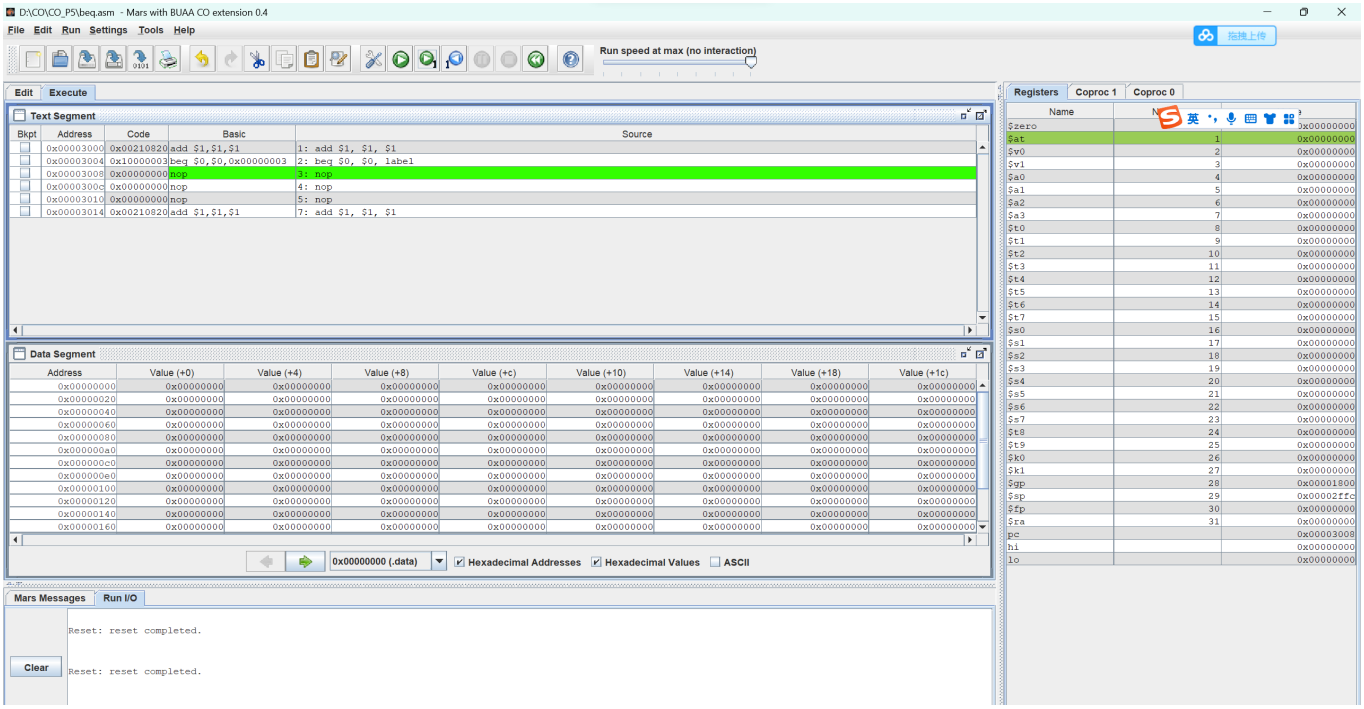
Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 Kbytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

BEQ r0, r0 offset, expressed as B offset, is the assembly idiom used to denote an unconditional branch.



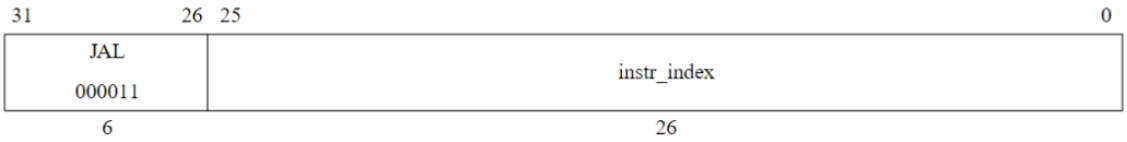
由于延时槽问题，beq下的一条指令会被执行，而跳转的仍然还是用D_pc + 4 + offset = F_PC + offset

Operation:

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

---

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
```

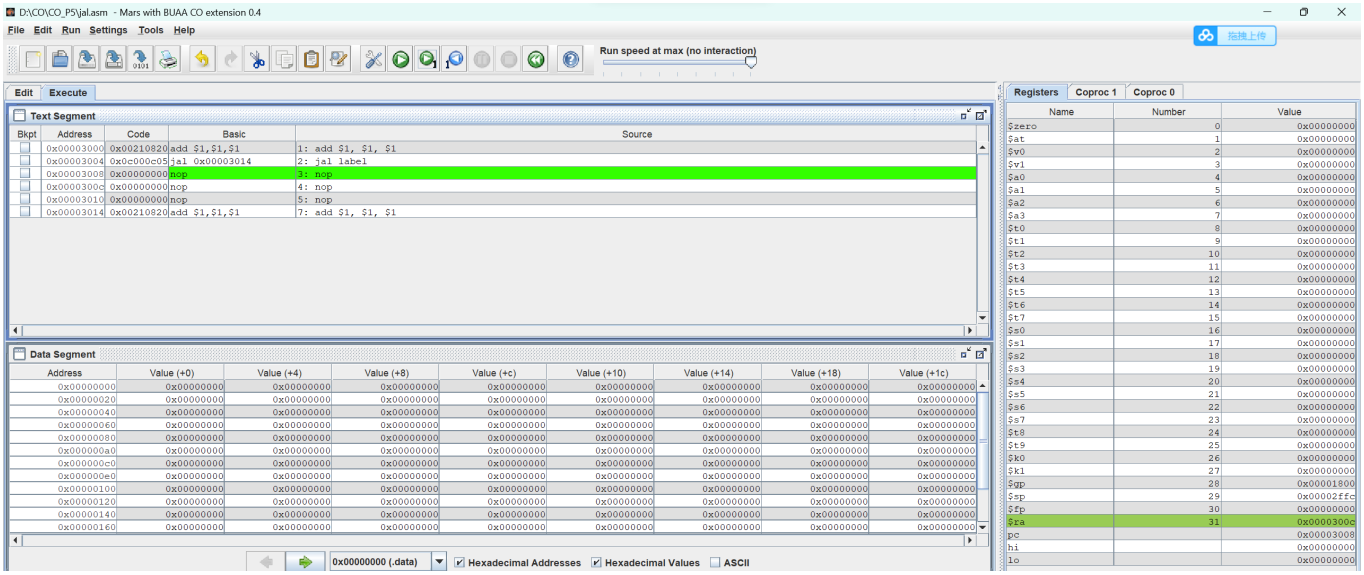
Exceptions:

None

Programming Notes:

Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

This definition creates the following boundary case: When the branch instruction is in the last word of a 256 MB region, it can branch only to the following 256 MB region containing the branch delay slot.

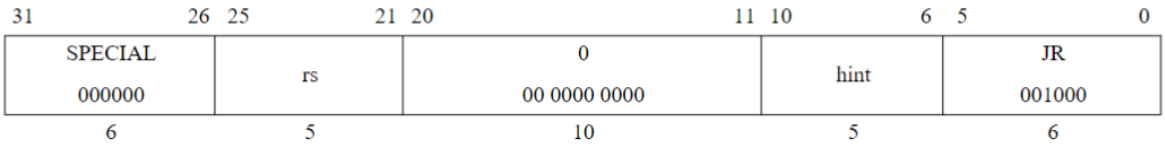


$ra 存储的为当前D_PC + 8 = F_PC + 4（NPC.pc4）
相当于jal在D级的时候，存储值为F级PC + 4

---

jr

Jump Register JR



31–26	25–21	20–11	10–6	5–0
SPECIAL 000000	rs	0 00 0000 0000	hint	JR 001000
6	5	10	5	6

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
```

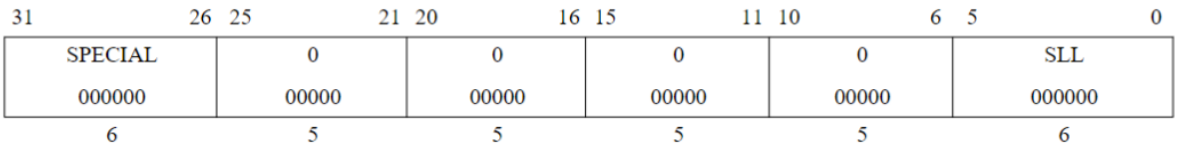
Exceptions:

None

---

nop

No Operation — NOP



Format: `NOP` — Assembly Idiom

Purpose:

To perform no operation.

Description:

NOP is the assembly idiom used to denote no operation. The actual instruction is interpreted by the hardware as SLL r0, r0, 0.

Restrictions:

None

Operation:

None

Exceptions:

None

Programming Notes:

The zero instruction word, which represents SLL, r0, r0, 0, is the preferred NOP for software to use to fill branch and jump delay slots and to pad out alignment sequences.