CO-P5-流水线CPU设计文档

#CPU #Verilog

---

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

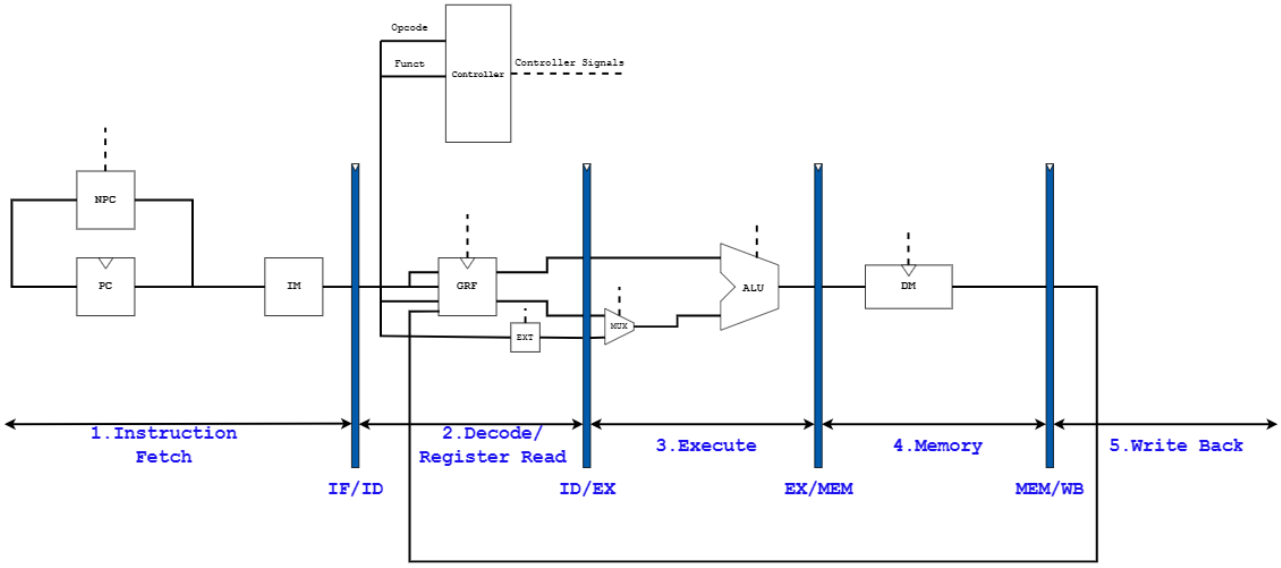
1. 处理器应支持如下指令集：{ `add, sub, ori, lw, sw, beq, lui, jal, jr, nop` }。
2. 处理器为五级流水线设计。
3. 流水线的设计以追求性能为第一目标，因此必须尽最大可能**支持转发**以解决数据冒险。这一点在本 project 的最终成绩中所占比重较大，课上测试时会通过测试程序所运行的**总周期数**进行判定，望大家慎重对待。
4.
    1. 对于 b 类和 j 类指令， 流水线设计必须**支持延迟槽**，因此设计需要注意使用 **PC@D + 8**或**PC@I + 4**。以及为了对拍MARS应该勾选**Delayed Branches**
5. 转发数据来源必须是**某级流水线寄存器**，**不允许**对功能部件的输出直接进行转发。
6. `IM` = 4096 *32 bits，4098 =* 2^{12} *;* `DM` *= 3072* 32 bits，2^{11} < 3072 < 2^{12}
7. PC 的初始地址为 **0x00003000**，和 Mars 中我们要求设置的代码初始地址相同
8. 最外层的 mips 模块的文件名必须为 mips.v ， 该文件中的 module 也必须命名为 **mips** 。

9. 修改GRF和DM输出，加入`$time`

---

3 Heirarchy Designation

3.1 Buses and Arrangements



3.2 WorkFlow

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

- IFU
- NPC

- D_REG
  - D_CU (CU)
  - GRFs
  - EXT
  - CMP
- E_REG
  - E_CU (CU)
  - ALU
- M_REG
  - M_CU (CU)
  - DM
- W_REG
  - W_CU (CU)
- CU
- MFC
- STALL

4.2 IFU

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
stall	I	冻结信号
npc[31:0]	I	下一个指令的地址

信号	方向	描述
pc[31:0]	O	当前指令地址
Instr[31:0]	O	当前指令

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

本质上就是一个寄存器 + 大寄存器数组
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. 冻结需要将上一周期Instr接受到本周期Instr

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
pc_D[31:0]	I	当前D级的pc值

信号	方向	描述
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	服务BEQ	满足输出1，不满足输出0
...	待扩展	待扩展

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

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

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

信号	方向	描述
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
E_outC[31:0]	I	ALU.outC@E
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_Tnew[3:0]	O	M级的Tnew

4.11 DM

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
MemWrite	I	写入/读取信号
MemAddr[31:0]	I	写入地址
MemData[31:0]	I	写入数据
pc[31:0]	*I*	*程序计数器，辅助输出评测*
ReadData[31:0]	O	输出数据

序号	功能	描述
1	载入	当MemWrite=1时且在时钟上升沿时，载入在MemAddr处载入MemData
2	读取	持续进行数据输出，按写入地址寻址

特别说明

1. 起始地址：**0x0000_0000**
2. 模块规格：12KiB（3072 × 32bit）→ 2^{12}
3. 地址数据范围：**0x0000_0000 ~ 0×0000_2FFF**
4. 如果需要写入数据：每个**时钟上升沿**到来时若**要写入数据**（即写使能信号为 1 且非 reset 时）则输出写入的位置及写入的值，格式（请注意空格）为：

```
$display("%d@%h: *%h <= %h", $time, pc,
addr, din);
```

符号映射关系：`pc`(pc)，`addr`(MemAddr)，`din`(MemData)

实现方法

注意：

1. **同步**复位
2. 利用`$display`输出，关注输出条件

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_ReadData[31:0]	I	读取内存值ReadData@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_ReadData[31:0]	O	ReadData@W
W_outC[31:0]	O	ALU.outC@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
MemWrite	O	DM 写入控制信号
Tnew[3:0]	O	E_Tnew值
Tuse_RS[3:0]	O	Tuse_RS值
Tuse_RT[3:0]	O	Tuse_RT值

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

序号	信号	功能
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

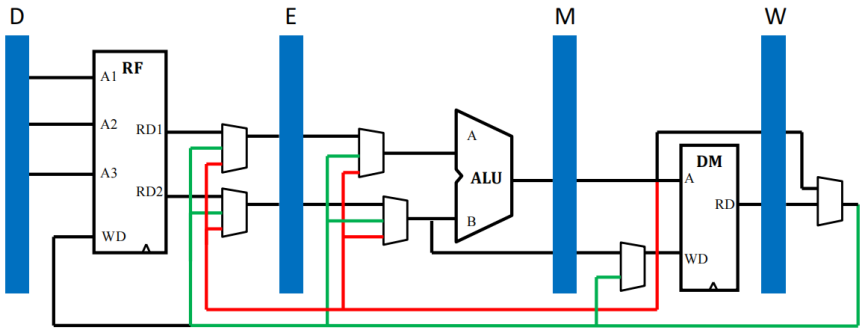
其中Tuse_RS与Tuse_RT会在D级流水中的CU使用
而Tnew会生成E级流水的T_new并且随着流水线逐级寄存器传递

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



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

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
```

端口信号/功能表

信号	方向	描述
rs_D[4:0]	I	
rt_D[4:0]	I	
rs_E[4:0]	I	
rt_E[4:0]	I	
rt_M[4:0]	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

信号	方向	描述
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
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

---

5 思考题

5.1

> 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。

对于指令序列

```
add $1, $1, $1;
beq $1, $2, label;
```

就算我们把分支预测提前到了D级，但是由于前一条指令涉及到了写寄存器1，而add指令处于E级，其结果尚未流入M级寄存器，无法转发；因此必须通过把beq指令阻塞到D级等一周期才能执行跳转，这样的话与beq在E级完成判断所需时钟周期数一样。

5.2

> 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?

因为由于延时槽的存在，我们的CPU会在执行跳转指令后执行PC+4位置的指令，而当跳转回去的时候，如果写回的是PC+4，则该位置的指令会执行两次，与我们对于程序执行的预期不符

5.3

> 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?

我认为这是考虑到了实际工程设计问题，对于流水线寄存器中的值一定在上升沿后为稳定的，但是组合电路存在传播延时，也就

是说正确的数据需要等待一段时间才能传递到对应位置

对比M级转发与E-ALU结果转发至D级：
M级转发delay = 转发延时
E-ALU转发delay = ALU计算延时 + 转发延时

显然功能部件转发延时 > 流水线寄存器转发延时

这会造成我们的流水线CPU单个时钟周期延长，影响性能

5.4

> 我们为什么要使用 GPR 内部转发？该如何实现？

很大的一点是将转发控制的复杂度降下来，

实现方法可以使用半写半读的机制，但是为了保证代码风格规范，我直接使用判断是否读写地址一致且可写且非0

```
assign RD1 = (RegAddr == A1 && RegWrite ==
1'b1 && A1 != 5'h00) ? RegData : mem[A1];

assign RD2 = (RegAddr == A2 && RegWrite ==
1'b1 && A2 != 5'h00) ? RegData : mem[A2];
```

5.5

> 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？

在功能部件部分MFSCU已经说明

5.6

指令分类：

> 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。

- 寄存器立即数计算：`addi, addiu, slti, sltiu, andi, ori, xori, sll, srl, sra`
- 寄存器寄存器计算：`add, addu, sub, subu, slt, sltu, and, or, nor, xor, sllv, srlv, srav`
- 根据寄存器分支：`beq, bne, bgez, bgtz, blez, bltz`
- 写内存：`sw, sh, sb`
- 读内存：`lw, lh, lhu, lb, lbu`
- 读乘除法寄存器：`mfhi, mflo`
- 写乘除法寄存器：`mthi, mtlo, mult, multu, div, divu`
- 跳转并链接：`jal, jalr`

- 跳转寄存器：`jr, jalr`
- 加载高位：`lui`
- 空指令：`nop`

主要的数据通路扩展问题其实来自于新增单元模块，而控制信号的扩展可能会更加丰富，基本按照工作流：

1. 确定数据通路
2. 确定控制信号矩阵
3. 扩展转发信号SRCMUX和DSTMUX
4. 计算Tuse和Tnew并加入到Controller.v中

即可完成一条指令扩展

5.7

> 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。

译码：分布式译码 + 控制信号驱动型

每级别流水实例化一个Controller元件，其巨大优势在于简化了我每个流水级寄存器的IO（因为现在我发现它已经相当庞大了），但是在真正工程设计时其实会消耗更多的晶体管元件，其成本更高

控制信号驱动是因为我发现使用指令驱动型时，由于指令数量和控制信号数量的增加，我的Switch块有些过于庞大以至于可读性

可维护性非常差，选择换一种代码风格，这个问题就是有的时候容易遗漏指令。

---

6 Test Strategy

> 测试完全基于开发的评测机和Toby学长魔改版Mars
> 该部分已经在CO讨论区开源说明

6.1 数据通路测试

先测试基本不会发生转发和阻塞的情况下，数据通路和控制信号是否存在问题

> 将narrow调高，让随机数在0-31之间随机选取寄存器，基本不会发生转发和阻塞相关问题

分为两轮：
第一轮测试仅仅包含算数和存取指令的测试

```
python main.py
--narrow 30
--mixed False
--suit 1
```

第二轮测试包含分支跳转

```
python main.py
--narrow 30
--mixed True
```

6.2 数据冒险测试

同样分为两轮，此时，调低narrow，限制寄存器范围，大大提高转发与阻塞的需求概率

第一轮测试算数+存取

```
python main.py
--narrow 3
--mixed False
--suit 1
```

第二轮测试全指令集

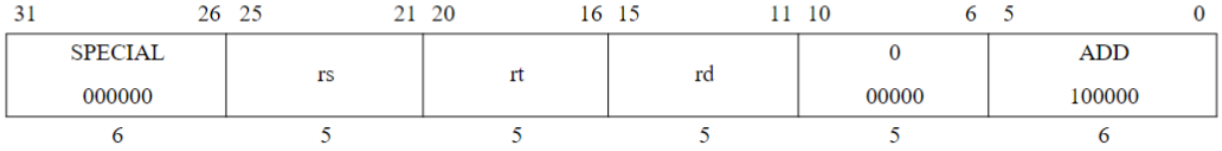
```
python main.py
--narrow 3
--mixed True
```

---

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

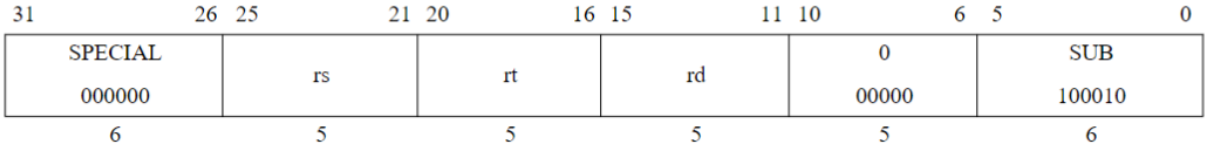
Programming Notes:

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

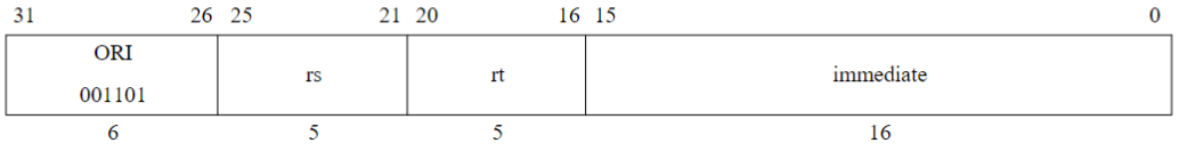
Programming Notes:

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)
```

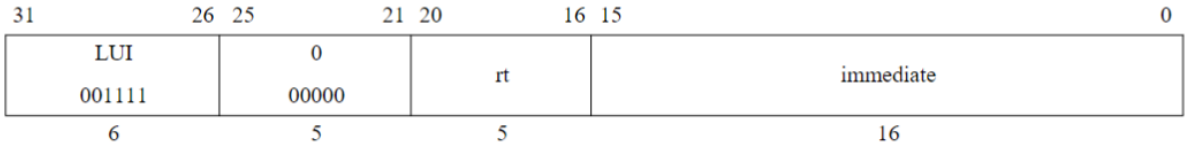
Exceptions:

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:

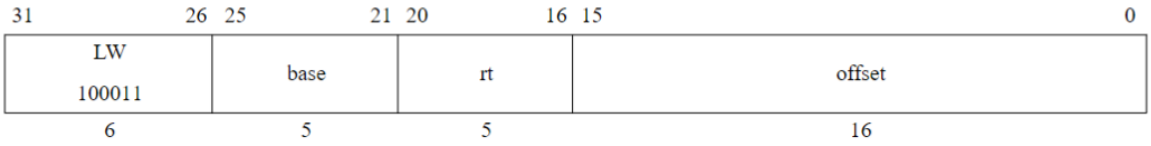
```
GPR[rt] ← immediate || 0^16
```

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:

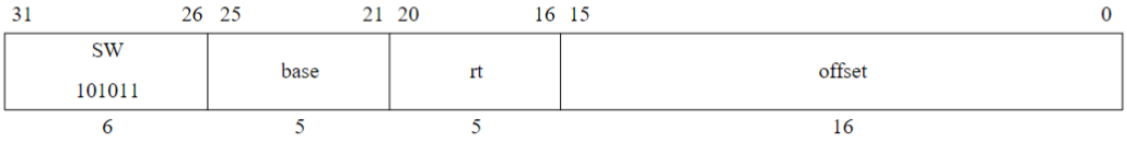
```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
memword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt]← memword
```

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:

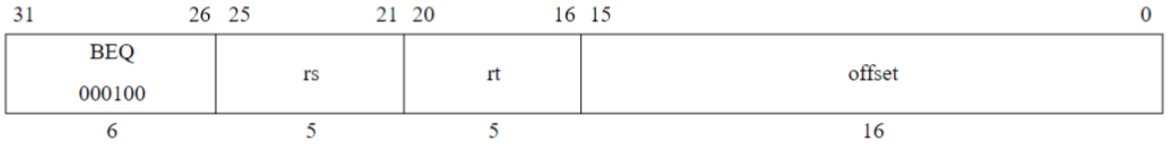
```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
dataword← GPR[rt]
StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
```

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
```

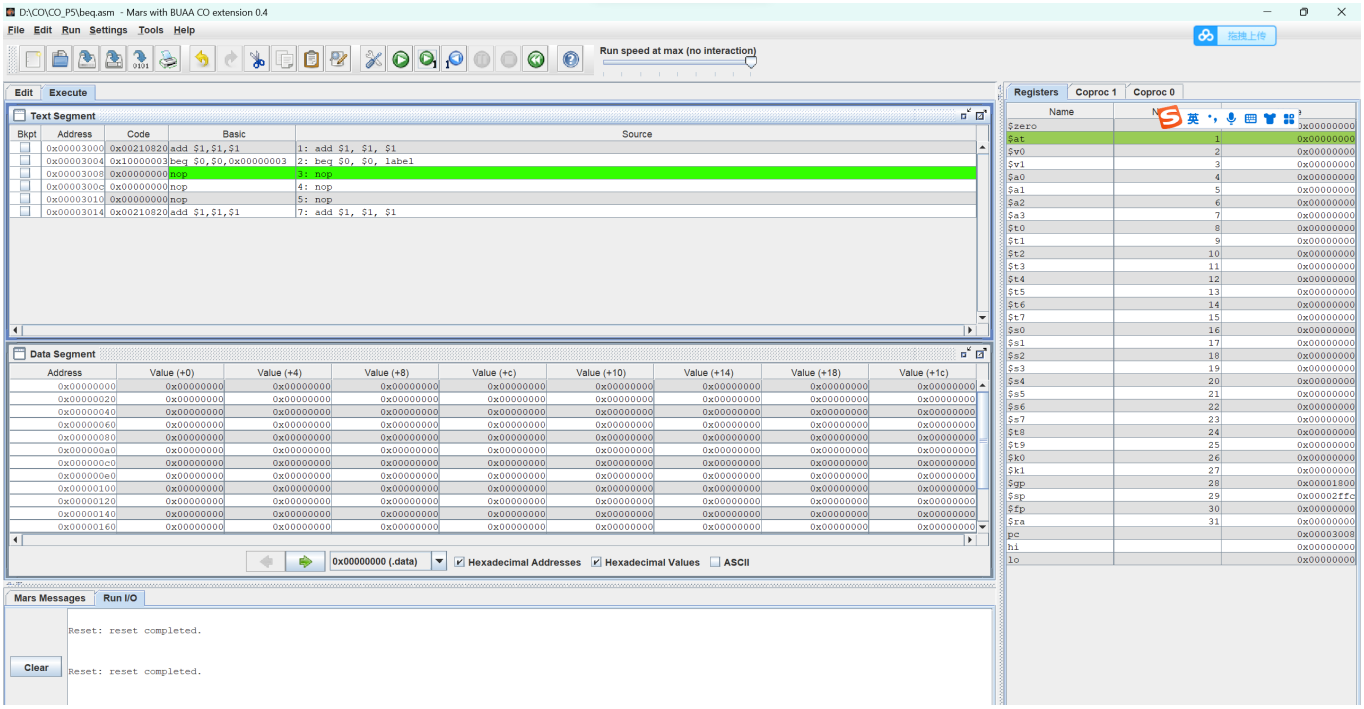
Exceptions:

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.



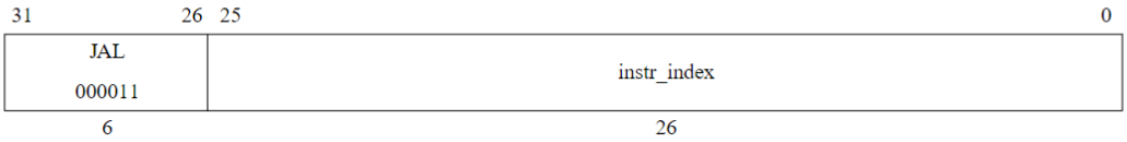
由于延时槽问题，beq下的一条指令会被执行，而跳转的仍然还是用D_pc + 4 + offset = F_PC + offset

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
```

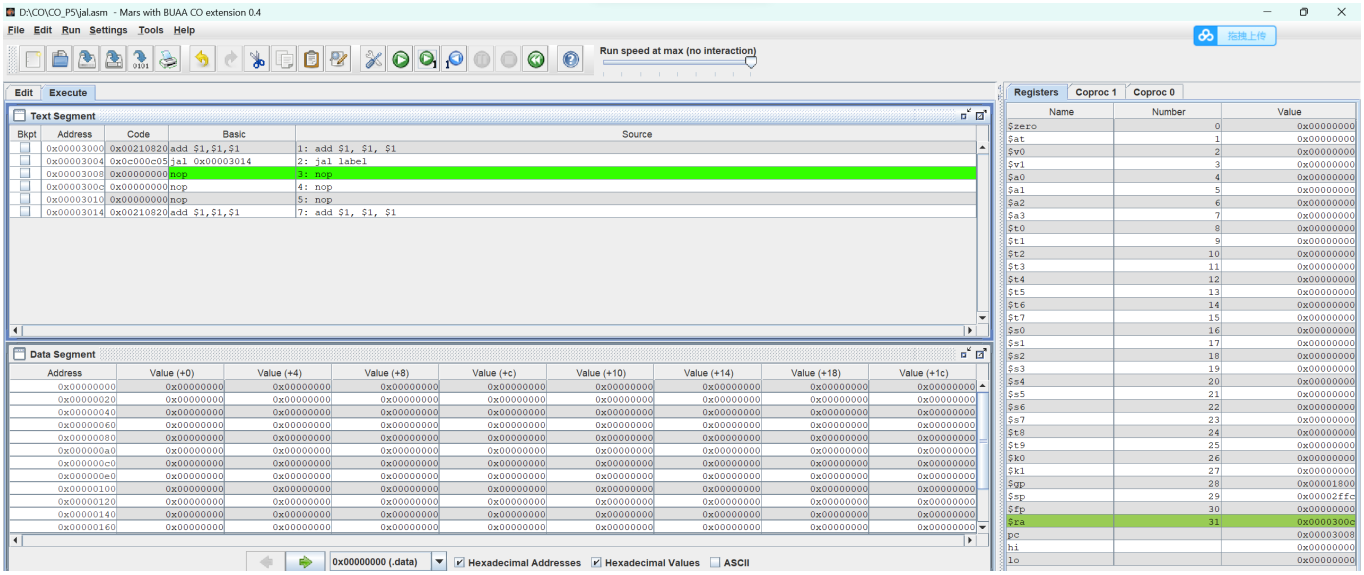
Exceptions:

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.

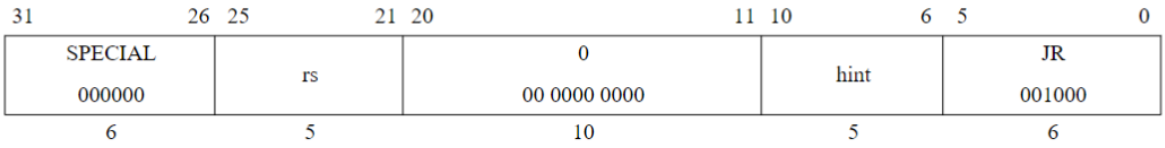


$ra 存储的为当前D_PC + 8 = F_PC + 4（NPC.pc4）
相当于jal在D级的时候，存储值为F级PC + 4

---

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
```

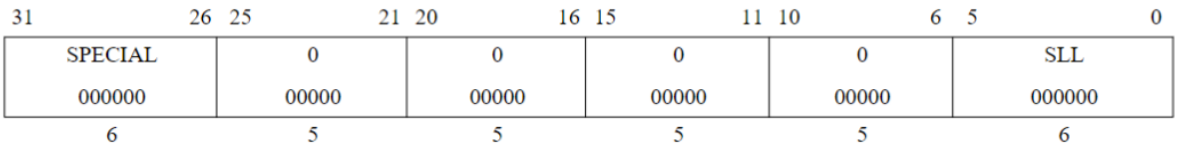
Exceptions:

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.