Pipeline 代码部分开发文档

文档：徐栋

南京大学，计算机科学技术系

邮箱：**dc.swind@gmail.com**

时间：**2014.6.24**

1 文档介绍

此工程为南京大学计算机系计算机体系结构实验大作业设计。在本工程中我们设计了 **MIPS** 流水线，在完成最基本功能的基础上，加入了冒险模块、转发模块、**Cache** 模块、并进行了静态分支预测。使其可以正确的执行约 **50** 条 **MIPS** 指令。在实验中，我们引入了之前实验没有使用的指令，例如 **bne**、**beq**、**sb**、**sh**、**lb**、**lbu**、**lh**、**lhu** 等。因为电路 (图纸) 的设计相对比较完善，所以在 **controller** 中还是可以加入新的指令。这份文档主要是徐栋同学在实验中完成的部分，包括工程文件的构成、部分代码实现细节及 **API**、**Cache** 的设计、代码中的命名规则、控制信号的设计及介绍等。

2 工程文件构成

pipeline		介绍	
	ALU.v		ALU 模块，包含 ALU 控制器，实现 ALU 功能
	Condition_Check.v	*	根据 ALU 产生的信号等进行条件判断写使能，跳转等
	Controller.v	*	控制器模块，进行指令译码，控制信号赋值
	DataMemory.v		数据存储器模块，下降沿写入，读为组合逻辑
	EX_MEM_Seg.v		第三段寄存器，信号的输入及输出，在时钟上升沿工作
	Forward.v	*	转发单元，处理数据同步问题
	HazardControl.v	*	冒险控制单元，控制冒险问题
	ID_EX_Seg.v		第二段寄存器，信号的输入输出，在时钟上升沿工作
	IF_ID_Seg.v		第一段寄存器，信号的输入输出，在时钟上升沿工作
	insruMemory.v		指令寄存器单元，设置仿真指令，模块在下降沿提供写
	MEM_WB_Seg.v		第四段寄存器，信号的输入输出，在时钟上升沿工作
	Memory_Shift.v	*	lwl、swr 等指令的移位部分
	MIPS_Register.v		寄存器组模块，寄存器组初值设置
	NumExpansion.v		立即数扩展单元
	PC.v		PC 控制单元，PC 的初始化，在下降沿 PC 的写入
	pipeline.v	*	流水线主单元，将各个模块连线
	Shifter.v		移位单元，桶形移位器
	Cache.v	*	Cache 模块，Datamemory 的 cache 设计

3 各信号命名规则

因为在其他模块中的变量的个数较少，且命名时已经隐含了其意义，所以这里主要说明主模块及段寄存器中的命名规则，在主模块中，主要考虑到许多在不同段的同名信号的取值是不同的，所以要在命名上加以区分。在主模块中，几乎所有的信号都是 **wire** 线型。在定义变量名时，遵循以下规则：

1. 所有段之后产生的变量，均在表意名称后加段名表示，例如第一段的输出加 **__IF** 表示；第二段加 **__ID**；第三段加 **__EX**；第四段加 **__MEM.**

2. 所有模块的变量的定义分模块进行，且变量的定义在信号输出的地方定义。例如第一段的输出信号在第一段定义。

3. 所有段寄存器模块内部变量均按照设计图的变量顺序从上至下定义，输入输出一一对应，输出加 **reg** 型存储变量值。

4. 仿真时主模块输出信号命名：为简化命名，将各段信号定义注释掉，在 **output** 处定义。

4 控制信号，指令译码

4.1 控制信号表

	1	4	5	7	11	13	12	15	22	16	24	2	3	14	8	17	9	10	6	18/22	19	20	21/23
控制器译码表	add	sub	subu	xor	srav	sltu	rotr	j	begz	bgezal	beq	addi	addiu	slti	xori	lui	clo	clz	seb	lwrl	lw	sw	swrl
RegDt0(0 5'b0,1 Rt)				1				1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
ID_RsRead(Rs Read 1)	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1
ID_RtRead(Rt Read 1)				1				0	0	0	1	0	0	0	0	0	0	0	1	0	0	1	1
Ex_top[1:0](00ex,01uex,10luiex)	xx	xx	00	xx	xx	xx	xx	00	00	00	00	00	00	00	01	10	xx	xx	xx	00	00	00	00
BranchSel				0				0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
OverflowEn	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
Condition[2:0]				xxx				xxx	011	011	001	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	x	x	x	x
Branch				0				0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
PC_Write (Op低三位)																							
Mem_write_byte_en[3:0]				0000				0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	1111	1111
Rd_write_byte_en[3:0]				1111				0000	0000	1111	0000	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	0000	0000
MemWBSrc(0 一般,1 load)				0				0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
Jump				0				1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ALUShift_sel(0 shift_out,1 ALUout)	1	1	1	1	0	1	0	x	x	x	x	1	1	1	1	1	1	1	1	1	1	1	1
MemDateSrc(0 32'b0,1 Rt,2 PC+8)				xxx				xxx	xxx	010	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	001	001
ALUSrcA (Rs_out,32'b0)				0				x	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
ALUSrcB (Rt_out,Immediate32)				0				x	0	0	0	1	1	1	1	0	0	1	1	1	1	1	1
ALU_op[3:0]	1110	1111	0001	1001	x	0111	x	x	0001	0001	0001	1110	0000	0101	1001	0000	0011	0010	1010	0000	0000	0000	0000
RegDst[1:0](00 Rd,01 Rt,10 31)				00				xx	xx	10	xx	01	01	01	01	01	00	00	00	01	01	xx	xx
Shift_amountSrc(0 shamt 1 Rs)	x	x	x	x	1	x	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Shift_op[1:0]	xx	xx	xx	xx	10	xx	11	xx	xx	xx	xx	xx	xx	xx	xx	x	xx	xx	xx	x	x	xx	xx

	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
控制器译码表	sb	sh	lb	lbu	lh	lhu	blez	bgtz	bltz	bne	beql	jal	or	ori	and	andi	addu	slt	sltiu	sra	srlv	nop	nor
RegDt0(0 5'b0,1 Rt)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
ID_RsRead(Rs Read 1)	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	0	1
ID_RtRead(Rt Read 1)	1	1	0	0	0	0	1	1	1	1	1	0	1	0	1	0	1	1	0	1	1	0	1
Ex_top[1:0](00ex,01uex,10luiex)	00	00	00	00	00	00	00	00	00	00	00	00	xx	01	xx	01	xx	xx	00	xx	xx	00	xx
BranchSel	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
OverflowEn	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Condition[2:0]	xxx	xxx	xxx	xxx	xxx	xxx	101	100	110	010	001	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	000	xxx
Branch	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
PC_Write (Op低三位)																							
Mem_write_byte_en[3:0]	1111	1111	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
Rd_write_byte_en[3:0]	0000	0000	1111	0001	1111	0011	0000	0000	0000	0000	0000	0000	1111	1111	1111	1111	1111	1111	1111	1111	1111	0000	1111
MemWBSrc(0 一般,1 load)	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Jump	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
ALUShift_sel(0 shift_out,1 ALUout)	1	1	1	1	1	1	x	x	x	x	x	x	1	1	1	1	1	1	1	0	0	0	1
MemDateSrc(0 32'b0,1 Rt,2 PC+8)	001	001	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	000	xxx
ALUSrcA (Rs_out,32'b0)	0	0	0	0	0	0	0	0	0	0	0	x	0	0	0	0	0	0	0	0	0	0	0
ALUSrcB (Rt_out,Immediate32)	1	1	1	1	1	1	0	0	0	0	0	x	0	1	0	1	0	0	1	0	0	0	0
ALU_op[3:0]	0000	0000	0000	0000	0000	0000	0001	0001	0001	0001	0001	x	0110	0110	0100	0100	0000	0101	0111	xxxx	xxxx	0000	1000
RegDst[1:0](00 Rd,01 Rt,10 31)	xx	xx	01	01	01	01	xx	xx	xx	xx	xx	xx	00	01	00	01	00	00	01	00	00	00	00
Shift_amountSrc(0 shamt 1 Rs)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	1	0	x
Shift_op[1:0]	xx	xx	xx	xx	xx		xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	10	01	00	xx

4.2 控制信号介绍

信号名称	介绍
RegDt0	条件跳转指令时，Rt 输入的二选一控制信号，0-5'b00000，1-Rt。
ID_RsRead	冒险处理信号，即是否读 Rs 寄存器的值，1-读，0-不读。
ID_RtRead	冒险处理信号，即是否读 Rt 寄存器的值，1-读，0-不读。
Ex_top[1:0]	立即数扩展控制信号，00-符号扩展，01-无符号扩展，10-lui 使用。
BranchSel	bgezal 时选则 PC+8 压入 31
OverflowEn	写入是否关心 Overflow，1-Overflow 有效时不写入，0-不关心 Overflow。
Conditon[2:0]	条件转移指令的条件。011-bgezal 和 bgez。
Branch	是否为条件转移指令，1-是，2-否。
PC_Write[2:0]	Op 低三位，处理 load、store 的移位问题。
Mem_write_byte_en[3:0]	存储器写入使能，只有 store 指令为 4'b1111。
Rd_write_byte_en[3:0]	寄存器写入使能，是否向寄存器 Rd 写入。同时与 Overflow 有关。
MemWBSrc	load、store 的 writeback 选择信号，1-load 指令，0-其他指令。
Jump	1-jump 指令，0-非 jump。
ALUShift_sel	1-选择 ALU 结果，0-选择移位结果。
MemDataSrc[2:0]	选择器控制信号，选择 Store 的输入及 branch 的 PC+8 的压入。
ALUSrcA	0-Rs 值，1-0。
ALUSrcB	1-jump 指令，0-非 jump。
ALU_op[3:0]	ALU_op
RegDst[1:0]	写入寄存器选择信号。0-Rd，1-Rt，2-R31。
Shift_amountSrc	移位数选择，0-shamt 立即数，1-Rs。
Shift_op[1:0]	移位控制信号。

4.3 指令集介绍

正如在控制信号表中所列出的，我们实现了共计约 **50** 条 **MIPS** 指令。为了方便读者理解与阅读本工程、查询指令的含义及 **op**、**func** 字段的值，我们在此文档后的附录 **1** 中给出了指令集的简介。这部分简介均取自 **MIPS** 指令手册。

4.4 控制器指令译码

控制器模块的设计位于 **controller.v** 中，因为考虑到可扩充性，所以在具体实现中使用了 **case** 语句，根据 **op** 及 **func** 字段唯一确定一条指令，对应控制信号表进行控制信号赋值。为了使得仿真结果更加直观、更容易判别错误，我在最后将控制器中所有的不确定信号赋值为 **x**，而不是不赋值，这样避免沿用前面指令的信号导致一些难以发现的错误。为了简化电路，我将其中的部分相似的指令，如 **R** 型指令的相同的控制信号，在 **case(op)** 处赋值，而在 **case(func)** 处只赋值指令各不相同的控制信号，这样就大大简化了电路的复杂程度。

控制器模块的输入为 **IR**，输出为所有译码出的控制信号。

5 Cache 设计及实现

```
//assume Memory block size is 32bit rather than 8bit ,addr low 2 bit is 00
//31      7 6     3 2      2 1     0
//+--------+-------+--------+------+
//|  tag   | block | offset |  00  |
//+--------+-------+--------+------+
//    25       4       1        2
```

Figure 1: 直接映射 Cache 的地址分割

```
1  assign writeback = (hit == 0 && valid[block] == 1 && dirty[block] ===1);
2  always @(posedge Clk) begin
3      up1=!up1;
4      hit=((valid[block]==1)&&(tag[block]==addr[(ADDR_WIDTH-1):Block_Size_log
           +2+Cache_Size_log]));
5      writeramn = (tag[block][2:0]<<5)+(block<<1)+0;
6      if(writeback) begin
7          ram[writeramn] <= cache[block][31:0];
8          ram[writeramn+1] <= cache[block][63:32];
9      end
10 end
```

在 DataMemory 的设计中，分别设计了没有 Cache 的 DataMemory(DataMemory.v) 及带有 Cache 的 DataMemory(Cache.v)，其功能仿真的结果是一致的，因为带 Cache 的 DataMemory 逻辑较为复杂，所以并未对带 Cache 的 DataMemory 进行时序仿真。这里对带有 Cache 的 DataMemory 进行简要说明.

由于 Cache 是在完成前面基本功能的基础上添加的，所以在设计 Cache 的时候为了不影响其他部分，将 Cache 与 DataMemory 设计在同一模块并且让其在同一周期内完成，设计时使用了时钟的上升沿和下降沿。在上升沿，判断是否命中，是否需要写回 (writeback)，并完成写回（见上图第一段代码）。在下降沿，则是对 Cache 值的修改，为了减少逻辑上的延时，使用 case 语句将 store 指令的 write 和 miss 的修改同步完成。

Cache 的设计如 Figure1 所示，是直接映射。其中 block 共 16 个，所以占 4bit，而 offset 为 2，占 1bit，即每个 offset 为 64bit。同时 32bit 块内 offset 均为 00。在得到 addr 后，首先将其进行分割，得到 tag、block 号和 offset。根据比较 tag 及 valid 来确定是否 hit(见代码中 hit 的表达式)，如果 tag 相等且 valid==1 则 hit。如果 miss 且 valid==1 且 dirty 位为 1，则需要写回 (dirty 记录 cache 中的值是否被修改。)。当 writeback 信号有效时，则进行写回操作。

模块的输入为数据、地址、使能信号、时钟，输出为数据输出及一些冗余测试信号，如 hit 等。

6 其它各模块实现介绍及 API

6.1 Forward.v

```
assign Rs_EX_Forward = ((Rs_From_ID_EX == Rd_From_EX_MEM) && (
    RegWrite_From_EX_MEM != 4'b0000))?(2'b01):((Rs_From_ID_EX ==
    Rd_From_MEM_WB) && (RegWrite_From_MEM_WB != 4'b0000)?2'b10:2'b00);
assign Rt_EX_Forward = ((RegDstRt == 2'b00)&&((Rt_From_ID_EX ==
    Rd_From_EX_MEM) && (RegWrite_From_EX_MEM != 4'b0000)))?(2'b01):((RegDstRt
    == 2'b00) && (Rt_From_ID_EX == Rd_From_MEM_WB) && (RegWrite_From_MEM_WB
    != 4'b0000)?2'b10:2'b00);
assign Rs_LoudUse_Forward = (RegRead[0] & Rs_From_IF_ID == Rd_From_MEM_WB
    && RegWrite_From_MEM_WB != 4'b0)?1'b1:1'b0;
assign Rt_LoudUse_Forward = (RegRead[1] & Rt_From_IF_ID == Rd_From_MEM_WB
    && RegWrite_From_MEM_WB != 4'b0)?1'b1:1'b0;
```

主要处理当前使用的 **Rs**、**Rt** 的值在前面指令中更改及 **load-use** 的转发问题。模块通过输入各个段的 **Rs**、**Rt** 等的使用情况来进行判断是否需要转发。例如 **ID** 段的 **Rs** 和 **EX** 段的 **Rd** 地址相同且 **EX** 段对 **Rd** 寄存器的值进行了修改，则需要进行转发。上面给出的代码是转发信号的表达式，分别确定是否需要转发 **EX** 段的 **Rs**、**Rt** 以及 **Load-Use**。

6.2 HazardControl.v

```
assign IF_ID_stall = (LoadUse)?1'b1:1'b0;
assign ID_EX_stall = 1'b0;
assign EX_MEM_stall = 1'b0;
assign MEM_WB_stall = 1'b0;
assign IF_ID_flush = 1'b0;
assign ID_EX_flush =((Jump===1'b1)|(EX_MEM_Branch===1'b1)|LoadUse)?1'b1:1'b0;
assign EX_MEM_flush = (EX_MEM_Branch == 1'b1)?1'b1:1'b0;
assign MEM_WB_flush = 1'b0;
```

冒险检测模块主要处理跳转冒险及 **Load-Use** 冒险，考虑到毕竟跳转指令在代码中所占比例较少，所以在设计中，我们使用了静态分支预测始终预测不跳转，所以在 **j** 和 **branch** 条件满足的情况下需要冲刷 **(flush)** 段寄存器。而在产生 **Load-Use** 冒险时，需要进行阻塞 **(stall)**。

模块的输入为跳转信号及 **Rs**、**Rt** 信号，输出为控制各段寄存器的 **flush** 和 **stall** 信号。

6.3 Condition_Check.v

```
//110-right-01    010-left-00    other-10
assign Right = (PC_Write == 3'b110 ?2'b01:(PC_Write == 3'b010)?2'b00:2'b10);
```

```
assign Load = (MemWBSrc == 1'b1)?1'b1:1'b0;
assign Store = (Mem_Byte_Write == 4'b1111);
MUX8_1_SL sl1({Right,addr[1:0]},Rd_Write_Byte_en,4'b1111,4'b1110,4'b1100,4'b1000
    ,4'b0001,4'b0011,4'b0111,4'b1111,LoadOut);
MUX8_1_SL sl2({Right,addr[1:0]},Mem_Byte_Write,4'b1111,4'b0111,4'b0011,4'b0001,4
    'b1000,4'b1100,4'b1110,4'b1111,StoreOut);
wire condition_out;
MUX8_1_Icontrol MUX_Con(Condition,1'b0,Zero,!Zero,!Less,!(Less^Zero),Less^
    Zero,Less,1'b1,condition_out);
//生成控制信号
assign BranchValid = condition_out & Branch;
assign RdWriteEn = (Load === 1'b1)?LoadOut:((OverflowEn == 0)?(
    Rd_Write_Byte_en):((Overflow == 1'b0)?(4'b1111):(4'b0000)));
assign MemWriteEn = (Store === 1'b1) ? StoreOut:4'b0000;
```

```
//load/store ---shift
assign out = (Sel[3])?Write_Byte_En:(   //not lwl or lwr or swr or swl
    (Sel[2])?(Sel[1]?(Sel[0]?S7:S6):(Sel[0]?S5:S4)):(
     Sel[1]?(Sel[0]?S3:S2):(Sel[0]?S1:S0)));
```

根据 ALU 产生 Less、Overflow 等信号，主要生成 Rd_Write_Byte_en、BranchValid、MemWriteEn 信号。其中 Rd_Write_Byte_en 根据是否为 load 指令、OverflowEn 及 Overflow 信号判断，如果是 load 指令，则根据 load 指令的类型 (PC_Write) 确定。否则若 OverflowEn 有效，即写使能取决于 Overflow，此时如果 Overflow 则写使能无效，否则有效。而 BranchValid 则是根据是否 Branch 且条件是否满足判断。然后 MemWriteEn 则是根据 Mem_Byte_Write 及 store 指令类型确定。上面代码一给出各个信号的表达式。

如果是 load、store 指令，上面第一段代码通过 Op 判断是否为 lwl、lwr、swr、swl，生成一个 Right 中间信号，判断出左移 (2'b00) 还是右移 (2'b01)，如果不是这四条指令而是 sb 等指令，则将 Right 赋值为 2'b10，继而通过第二段代码，可以通过选择器将使能信号选择出来。

模块的输入为 Condition，PC_write，地址低两位，使能信号，Overflow、Less 等 ALU 信号，输出为条件转移信号及 Mem 和 Reg 的写使能信号。

6.4 Memory_Shift.v

```
assign MEM_data_shift_ctr[2] = (IR[31])&(!IR[30])&(!IR[29])&(((!IR[28])&(IR
    [27])) | ((IR[27])&(!IR[26])) );
assign MEM_data_shift_ctr[1] = (IR[31])&(!IR[30])&(!IR[29])&(((!IR[27])&(IR
    [26])) | ((IR[28])&(IR[27])&(!IR[26])));
assign MEM_data_shift_ctr[0] = (IR[31])&(!IR[30])&(!IR[29])&(((IR[28])&(!IR
    [27])) | ((!IR[28])&(IR[27])&(!IR[26])));
```

逻辑相对简单，首先根据 IR 进行移位控制信号的确定，然后通过选择器选择输出的结果。上面代码为控制器表达式，模块输入指令 Op、Mem 输出的数据、及地址低两位，输出为移位后的数据。

6.5 pipeline.v

是流水线的主模块，通过各种 wire 型的线将所有模块连接起来，在这里设计的时候需要注意两个问题，一个是 wire 型在何处定义，二是如何命名。在何处定义是为了防止在 input 和 output 处重复定义，如何命名是为了防止不同段的同一信号具有相同的名字，引起冲突。所以在模块的实现中，我规定在 output 处进行 wire 型的定义来解决问题一，同时如前面命名规则中提到的，在信号后加段名来进行区分。除了这两个问题还要注意 input、output 的顺序、个数问题。因为牵扯到大量的中间 wire，所以一定要细致检查。

在测试的过程中，因为需要输出部分信号检测正确性，所以会动态的变换 wire 的定义位置，但仍保持分块定义增加代码的可读性。

6.6 段寄存器

段寄存器设计相对简单，在输入稳定之后将其存入段寄存器，且不依赖于时钟在任何时候都可以读段寄存器。因为是一组寄存器，所以输入为需要存入寄存器的信号，输出为 wire 连线输出段寄存器的各个信号值。

6.7 ALU.v、PC.v、MIPS_Register.v 等

这部分模块的设计在之前的单时钟周期及多时钟周期的 CPU 设计中已设计过且进行了详尽说明，这里不再赘述。

7 遇到困难、问题解决及收获

在整个代码的实现过程中，经历了许多困难，一部分是对 verilog 不够熟悉造成的，比如说 « 符号优先级等，因为潜意识中有着 C 语言的惯性思维，所以导致这部分错误十分不容易发现。其次是一些连线及控制信号的细节问题，这部分问题主要是源于细心程度及团队交流中的一些失误造成的，好在经过团队的协同努力，很快、很好的解决了这部分问题。最后是在 Cache 的设计中，因为是在实现好流水线的基础上完成的，所以遇到了一些限制，只能不够十分规范的设计出 Cache。因为 Cache 的逻辑相对复杂，所以未能完成 Cache 的时序仿真。

经过这次实验，我收获了很多。一是代码能力上的收获，经过大量的代码实现，我对 verilog 语言的认识更深了一步，从语言的实现细节即语言生成的电路是如何的硬件层面上对 verilog 有了清晰的认识。这也是经过一个学期的组原实验训练的结果。其次是增进了团队协作能力，在团队协作的过程中，发现了一些在团队协作中容易出现的问题，比如说信息不同步等。在解决这些问题的过程中获得了宝贵的经验。当然更重要的收获是在实验的过程中，对于流水线有了深刻的认识，同时在搜集资料的过程中对当前计算机的体系结构有了更多的了解，像协处理器、指令体系、Cache 原理等。

8 感谢

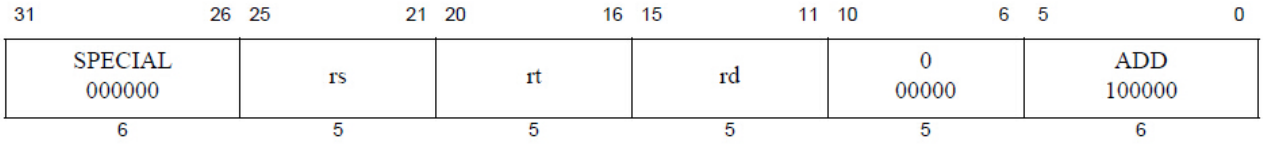
首先感谢在整个实验过程中张泽生老师给予的引导、指导，以及蔡晓燕老师及助教对一些问题的细致解答。同时感谢在整个学期内相互交流经验的同学们。最后感谢组员张瑞祎、李乾科同学，正是小组成员共同的努力才能够完成这个实验。

参考

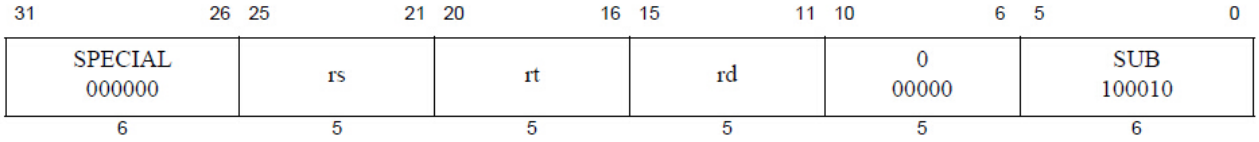
Verilog 语言手册
王帅老师体系结构 Cache 部分课件

附录 1

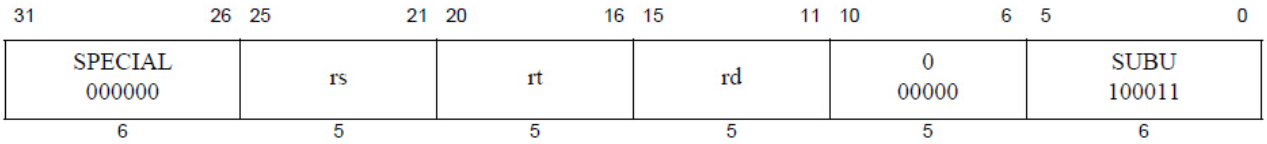
实现指令简介。



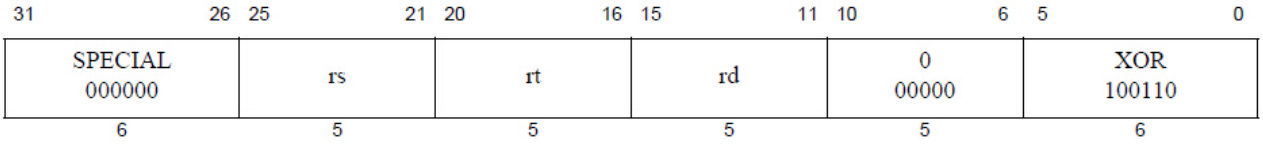
Format: `ADD rd, rs, rt` **MIPS32**



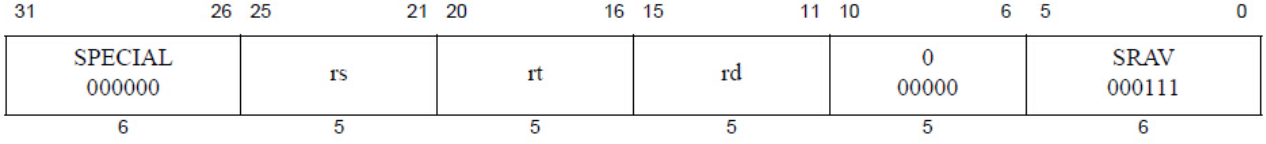
Format: `SUB rd, rs, rt` **MIPS32**



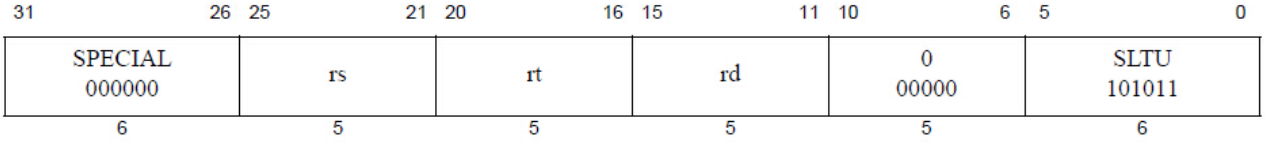
Format: `SUBU rd, rs, rt` **MIPS32**



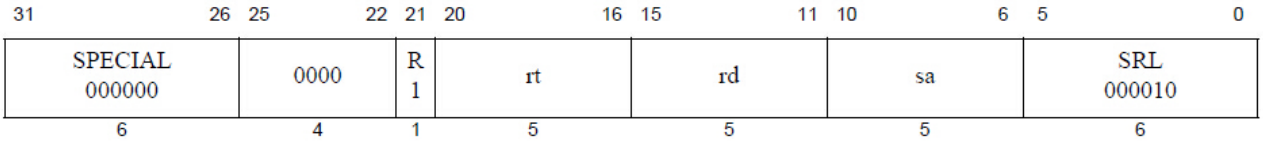
Format: `XOR rd, rs, rt` **MIPS32**



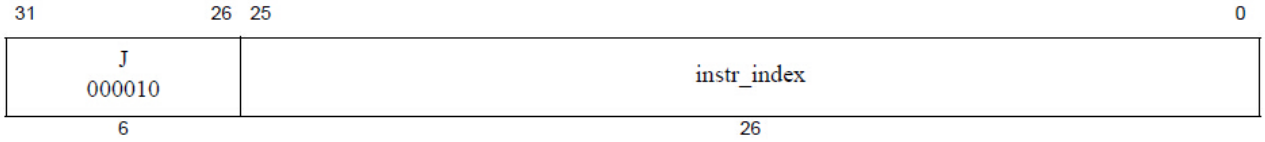
Format: `SRAV rd, rt, rs` **MIPS32**



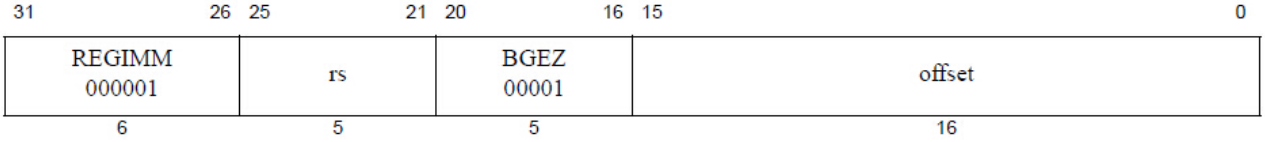
Format: `SLTU rd, rs, rt` **MIPS32**



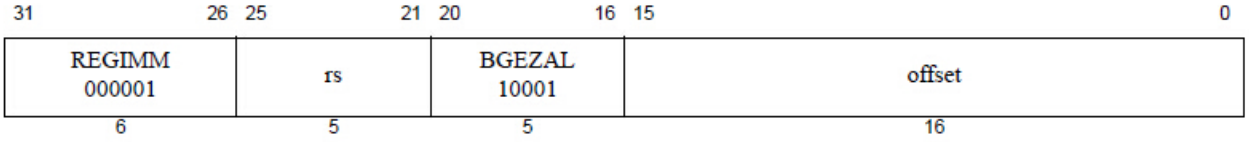
Format: `ROTR rd, rt, sa` **SmartMIPS Crypto, MIPS32 Release 2**



Format: `J target` **MIPS32**



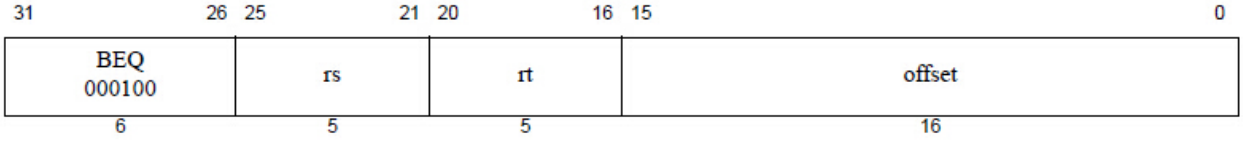
Format: `BGEZ rs, offset` **MIPS32**



Format: `BGEZAL rs, offset` **MIPS32**

Purpose: Branch on Greater Than or Equal to Zero and Link

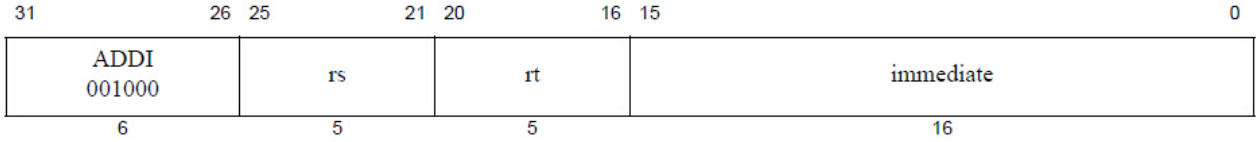
To test a GPR then do a PC-relative conditional procedure call



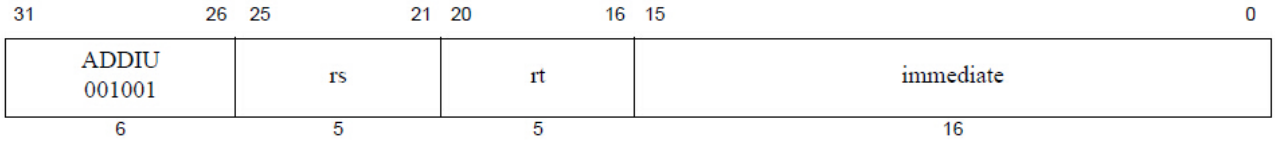
Format: `BEQ rs, rt, offset` **MIPS32**

Purpose: Branch on Equal

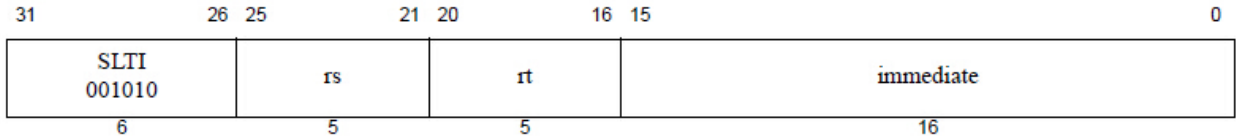
To compare GPRs then do a PC-relative conditional branch



Format: `ADDI rt, rs, immediate` **MIPS32**



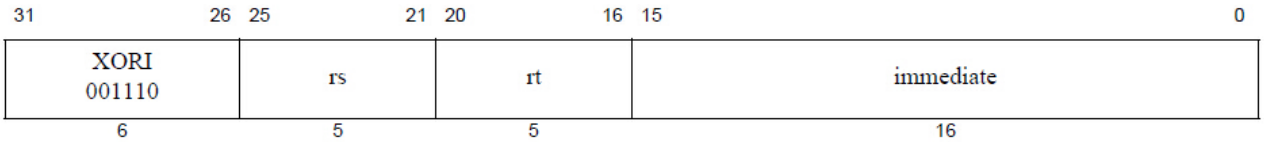
Format: `ADDIU rt, rs, immediate` **MIPS32**



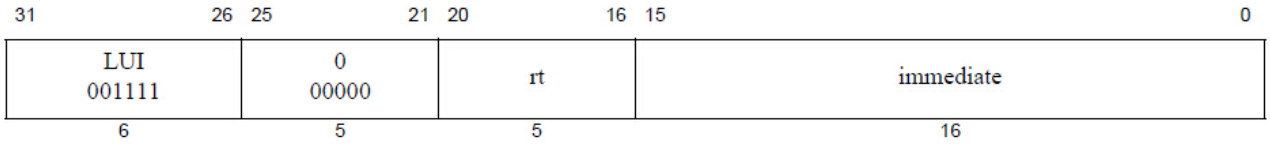
Format: `SLTI rt, rs, immediate` **MIPS32**

Purpose: Set on Less Than Immediate

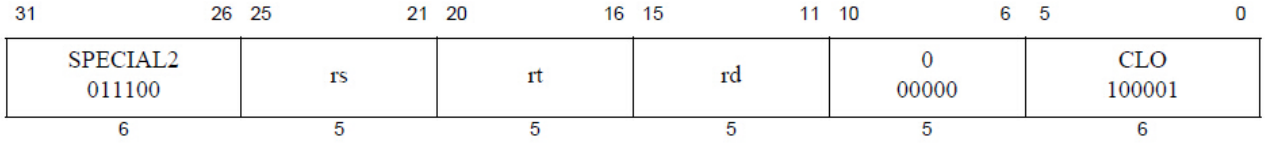
To record the result of a less-than comparison with a constant



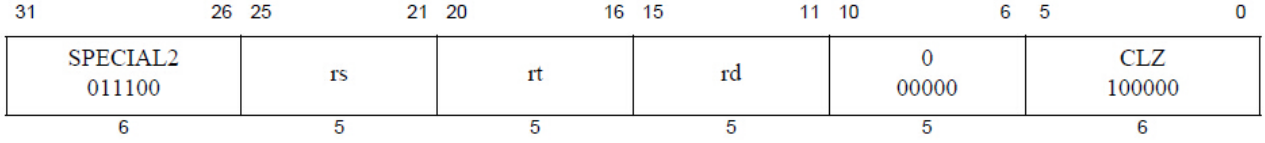
Format: `XORI rt, rs, immediate` **MIPS32**



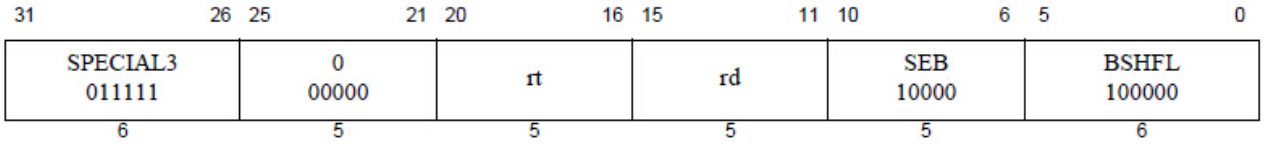
Format: `LUI rt, immediate` **MIPS32**



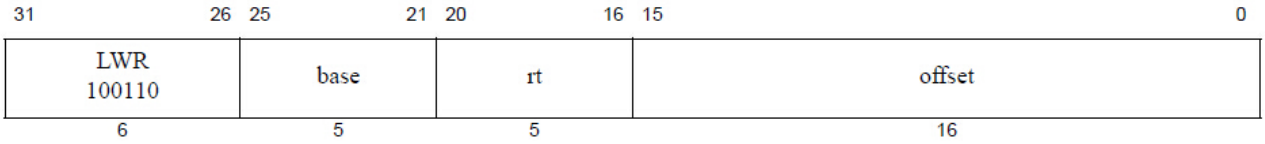
Format: `CLO rd, rs` **MIPS32**



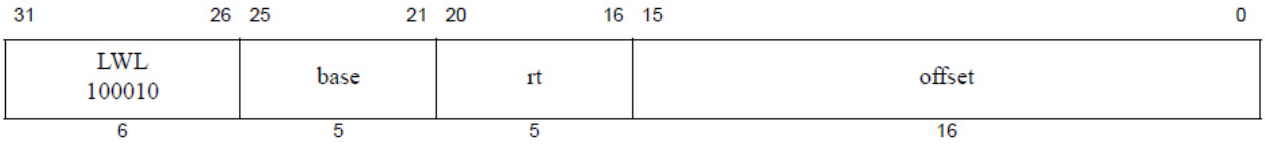
Format: `CLZ rd, rs` **MIPS32**



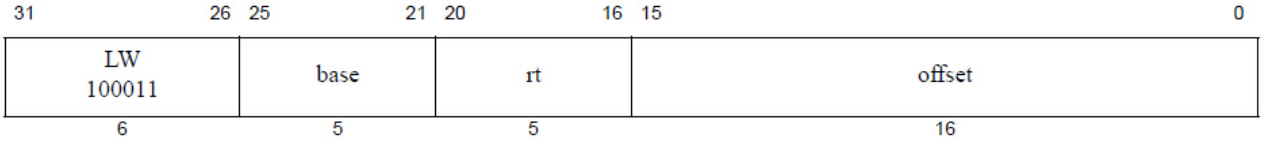
Format: `SEB rd, rt` **MIPS32 Release 2**



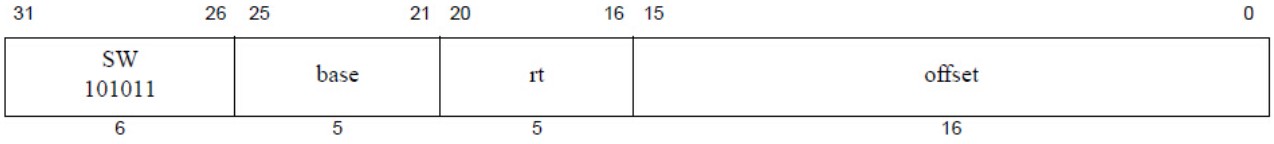
Format: `LWR rt, offset(base)` **MIPS32**



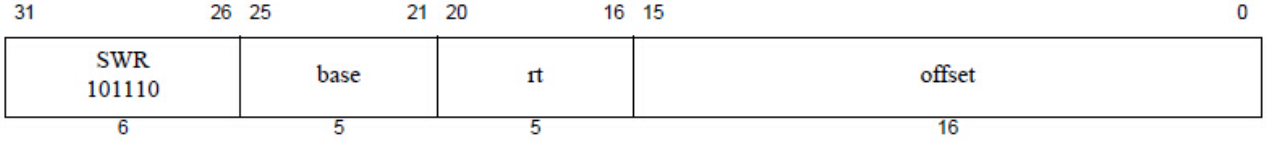
Format: `LWL rt, offset(base)` **MIPS32**



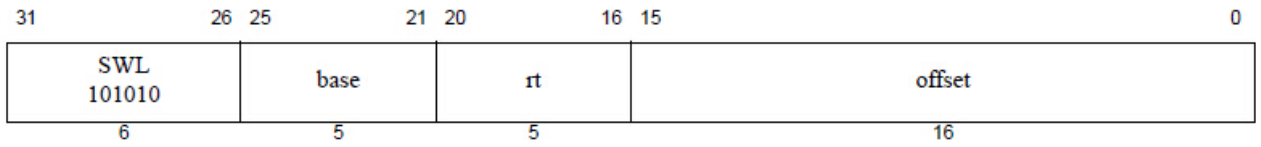
Format: `LW rt, offset(base)` **MIPS32**



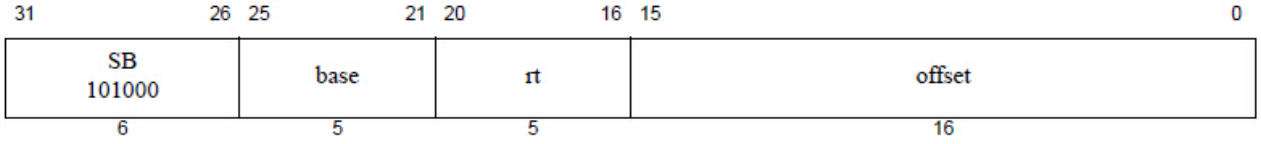
Format: `SW rt, offset(base)` **MIPS32**



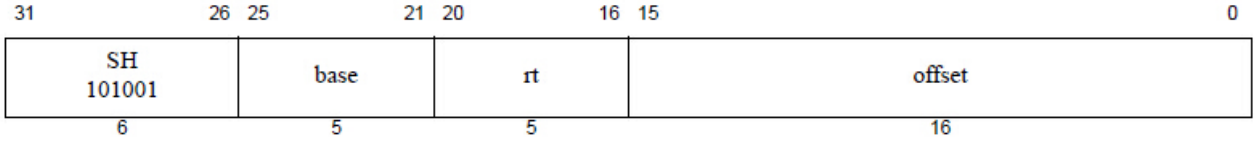
Format: `SWR rt, offset(base)` **MIPS32**



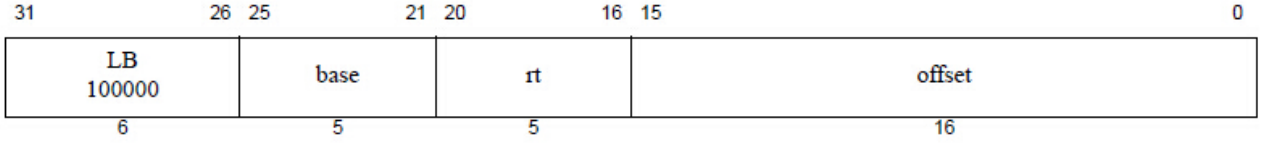
Format: `SWL rt, offset(base)` **MIPS32**



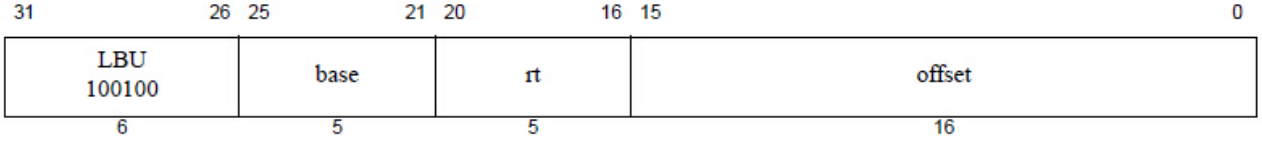
Format: `SB rt, offset(base)` **MIPS32**



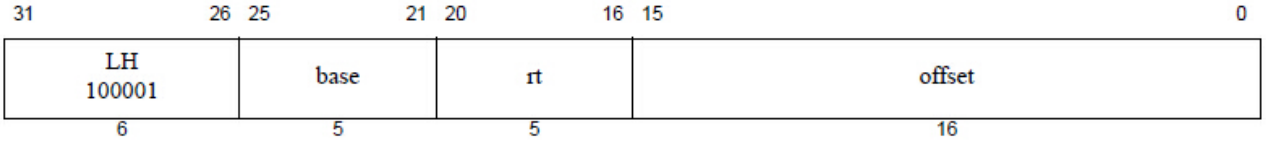
Format: `SH rt, offset(base)` **MIPS32**



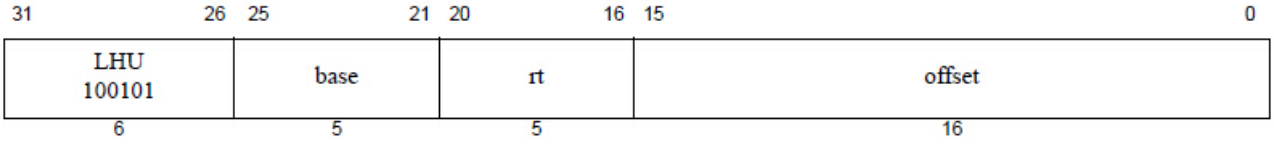
Format: `LB rt, offset(base)` **MIPS32**



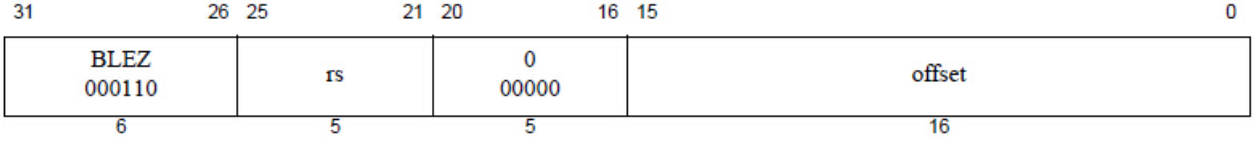
Format: `LBU rt, offset(base)` **MIPS32**



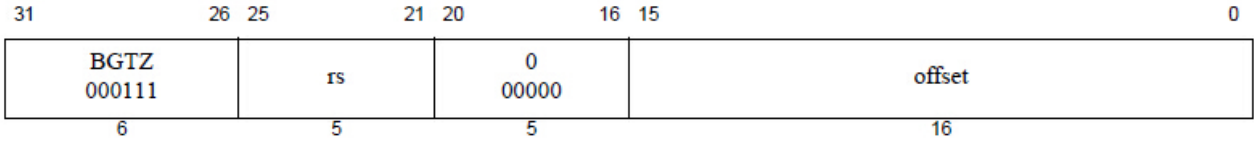
Format: `LH rt, offset(base)` **MIPS32**



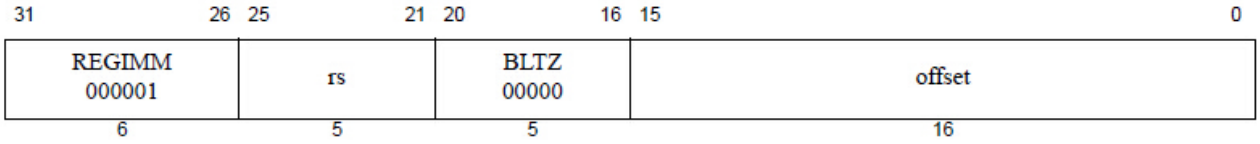
Format: `LHU rt, offset(base)` **MIPS32**



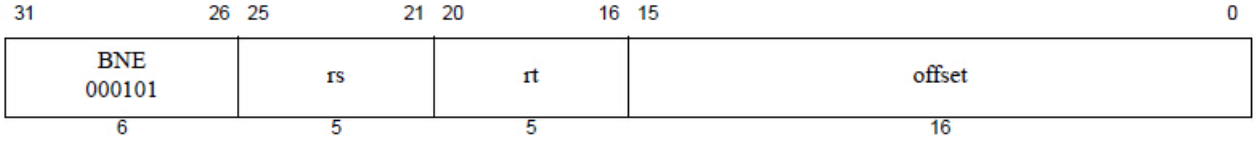
Format: `BLEZ rs, offset` **MIPS32**



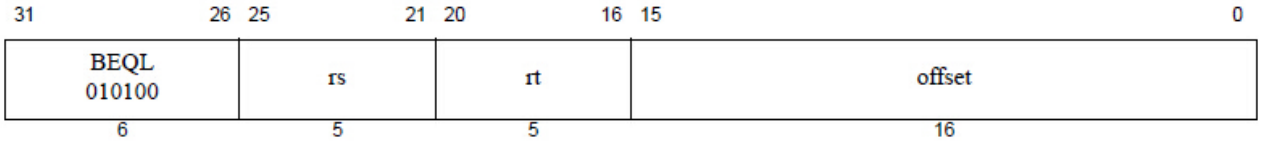
Format: `BGTZ rs, offset` **MIPS32**



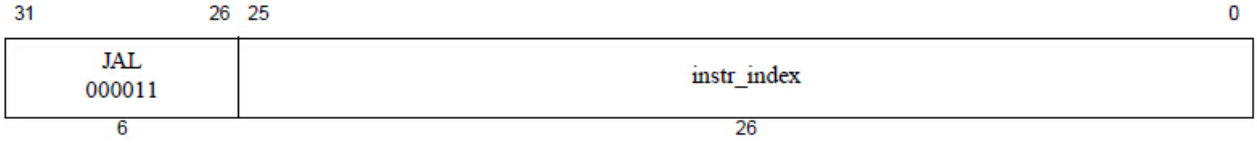
Format: `BLTZ rs, offset` **MIPS32**



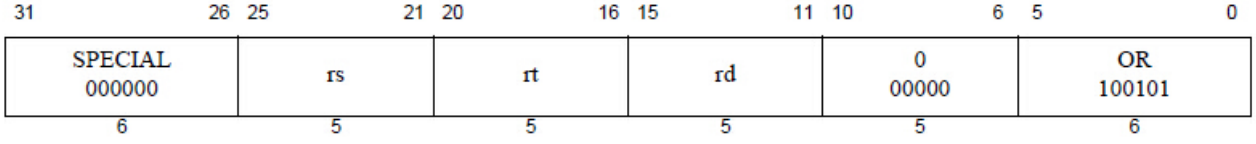
Format: `BNE rs, rt, offset` **MIPS32**



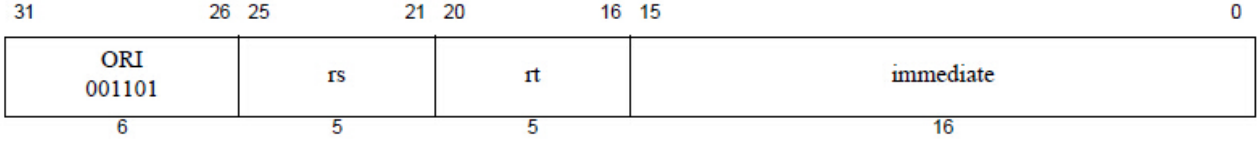
Format: `BEQL rs, rt, offset` **MIPS32**



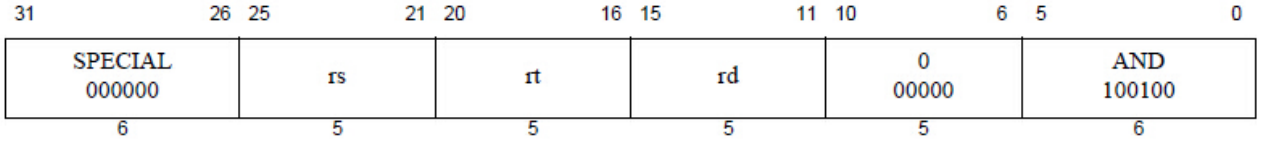
Format: `JAL target` **MIPS32**



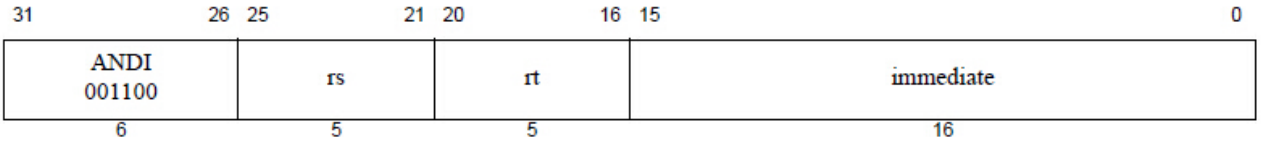
Format: `OR rd, rs, rt` **MIPS32**



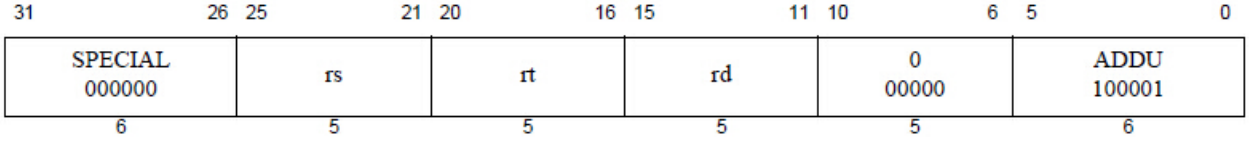
Format: `ORI rt, rs, immediate` **MIPS32**



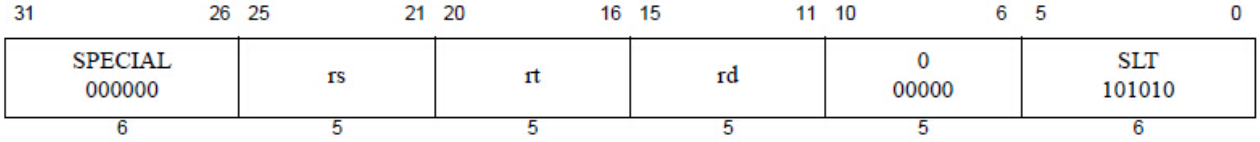
Format: `AND rd, rs, rt` **MIPS32**



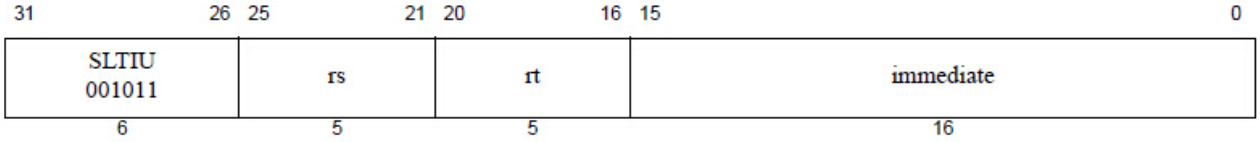
Format: `ANDI rt, rs, immediate` **MIPS32**



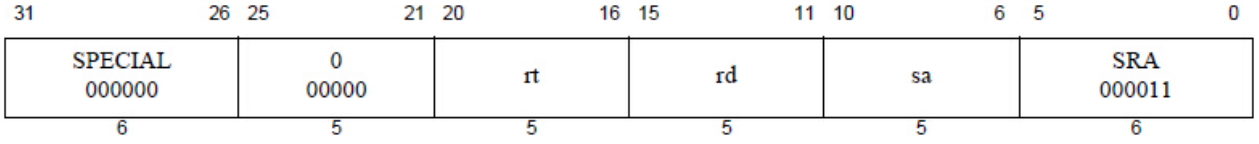
Format: `ADDU rd, rs, rt` **MIPS32**



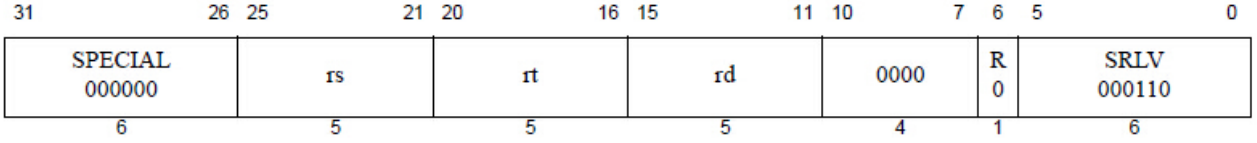
Format: `SLT rd, rs, rt` **MIPS32**



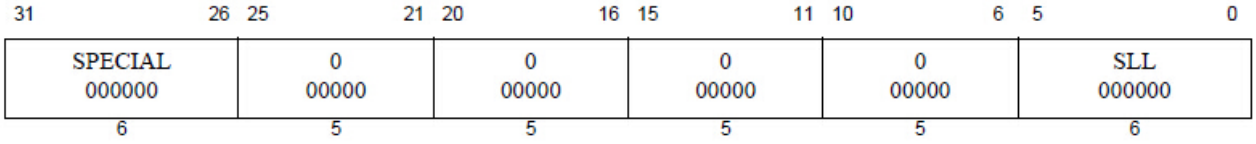
Format: `SLTIU rt, rs, immediate` **MIPS32**



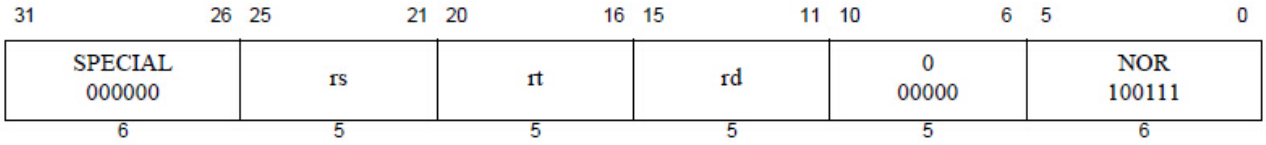
Format: `SRA rd, rt, sa` **MIPS32**



Format: `SRLV rd, rt, rs` **MIPS32**



Format: `NOP` **Assembly Idiom**



Format: `NOR rd, rs, rt` **MIPS32**