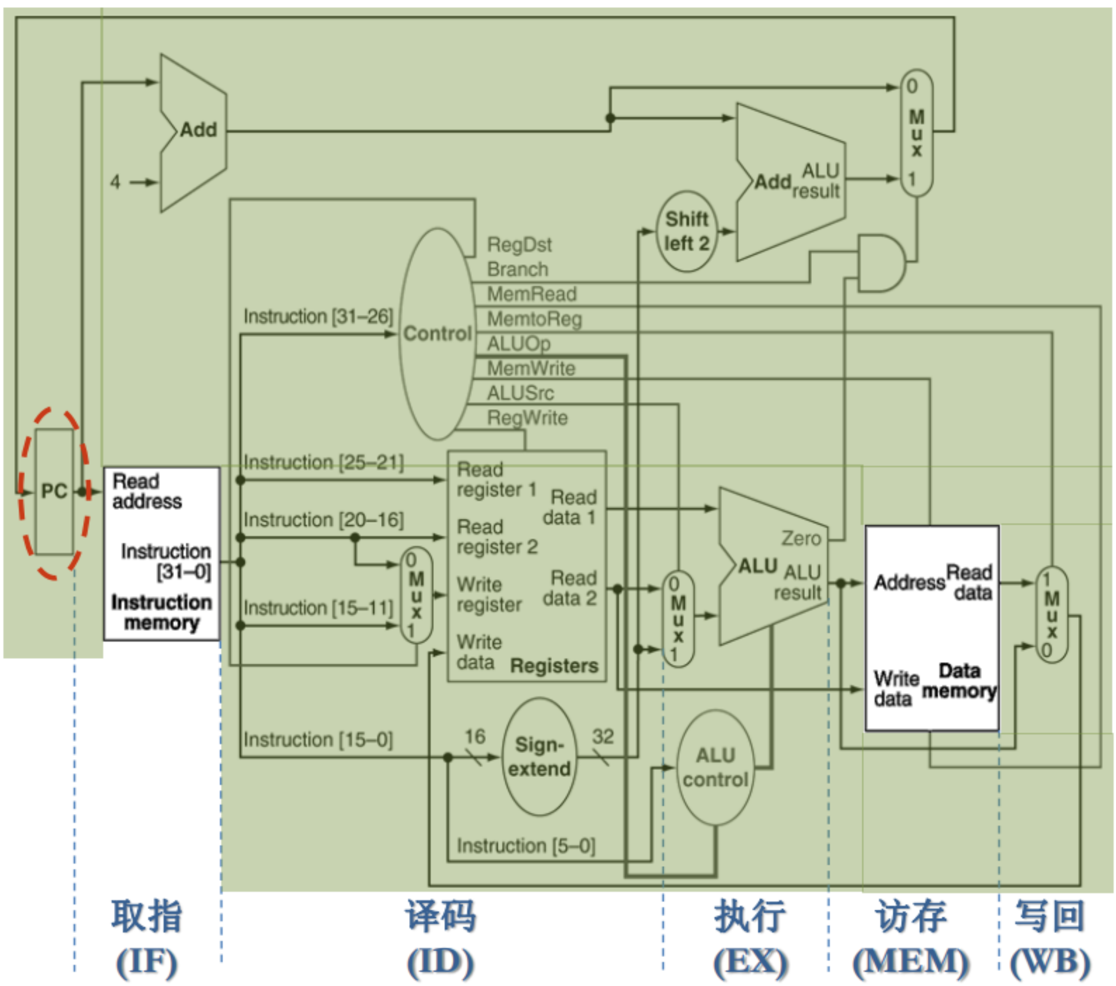
单周期CPU实验设计流程

阅读45条MIPS指令

指令分类	需完成的指令
运算类指令(14条)	addiu,addu,subu,and,andi,nor,or,ori,xor,xori,slt,slti,sltu,sltiu
移位指令（6条）	sll,slv,sra,srav,srl,srlv
跳转类指令(10条)	bne,beq,bgez,bgtz,blez,bltz,j,jal,jr,jalr
访存类指令（12条）	lb,lh,lw,lbu,lhu,lwl,lwr,sb,sh,sw,swl,swr
数据移动及立即数指令（3条）	movn,movz,lui

处理器各阶段需要的控制信号



下面的任务是，对于以上列出的所有指令，完成以下操作：

```
1.31-0指令格式(截图)
2.RTF语言
3.属于哪一类型: R-Type,REGIMM,J-Type,I-Type(分支), I-Type(运算), I-Type(访存)
4.寄存器堆读(raddr1,raddr2)
ALU A是什么? B是什么? ALUop是什么? (这里用中文写出操作名称)
内存访问: Address, 内存访问的地址是什么? MemRead,需要读内存吗? MemWrite, 需要写内存
吗?,Write_Data,需要写的数据是什么?
Write_Strb,需要写哪几个字节
跳转: 跳转地址, 地址更新
寄存器堆写:wen 是否要写寄存器堆,waddr写地址是什么,wdata写数据是什么?
```

完成45条指令填充

运算类

1.addiu(无符号立即数-字加法)

Add Immediate Unsigned Word **ADDIU**

31–26	25–21	20–16	15–0
ADDIU 001001	rs	rt	immediate
6	5	5	16

Format: `ADDIU rt, rs, immediate` **MIPS32 (MIPS I)**

Purpose:

To add a constant to a 32-bit integer

Description: `rt ← rs + immediate`

The 16-bit signed *immediate* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rt*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

```
temp ← GPR[rs] + sign_extend(immediate)
GPR[rt]← temp
```

Exceptions:

None

Programming Notes:

The term "unsigned" in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

```
1. 指令格式:
Instruction[31:26] = 6'b001 001;
Instruction[25:21] = rs;
Instruction[20:16] = rt;
Instruction[15:0] = immediate;
|001 001| rs(5) | rt(5) | immediate(16)|
2.RTF语言:
R[rt] <- R[rs] + immediate
3. 类型
I-Type(运算)
4, 寄存器堆读
```

```
raddr1 = rs(Instruction[25:21])  raddr2(无)
5. ALU
A: Readdata1 = R[rs]=R[Instruction[25:21]]
B: Sign_extend(immediate)  Sign_extend(Instruction[15:0])
ALUop: 无符号数加法
6.内存访问
不涉及内存访问（均设置为0）
7.跳转
不涉及跳转
8.寄存器堆写:
wen = 1      需要写寄存器
waddr = rt = Instruction[20:16]
wdata = R[Instruction[25:21]] + Sign_extend(Instruction[15:0]) = R[rs] + imme;
```

2.addu(无符号数加法)

Add Unsigned Word **ADDU**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	ADDU 100001
6	5	5	5	5	6

Format: `ADDU rd, rs, rt` **MIPS32 (MIPS I)**

Purpose:

To add 32-bit integers

Description: `rd ← rs + rt`

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

```
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← temp
```

Exceptions:

None

Programming Notes:

The term "unsigned" in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

```
1.指令格式
| 000 000 | rs(5) | rt(5) | rd(5) | 00000 (5) | 100 001(addu)|
Instruction[31:26] = 6'b0;
Instruction[25:21] = rs;
Instruction[20:16] = rt;
Instruction[15:11] = rd;
Instruction[10:6] = 5'b0;
Instruction[5:0] = 6'b 100 001;
2.RTF语言
R[rd] <- R[rs] + R[rt]
3.类型
R-Type
4.寄存器堆读
raddr1 = R[rs] = R[Instruction[25:21]];
```

```
raddr2 = R[rt] = R[Instruction[20:16]];
5.ALU
A: R[rs]= R[Instruction[25:21]];
B: R[rt] =  R[Instruction[20:16]];
ALUop: 加法
func: 100 001
6.内存访问
无内存访问，均设为0
7.跳转
无跳转
8.寄存器堆写:
wen = 1 需要写 rd
waddr = R[rd] = R[Instruction[15:11]];
wdata = R[Instruction[25:21]] + R[Instruction[20:16]] = R[rs] + R[rt]
```

3.subu(无符号 字的减法)

Subtract Unsigned Word **SUBU**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	SUBU 100011
6	5	5	5	5	6

Format: `SUBU rd, rs, rt` **MIPS32 (MIPS I)**

Purpose:

To subtract 32-bit integers

Description: `rd ← rs - rt`

The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* and the 32-bit arithmetic result is and placed into GPR *rd*.

No integer overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

```
temp    ← GPR[rs] - GPR[rt]
GPR[rd]← temp
```

Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. It is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

```
1.指令格式
| 000 000 | rs(5) | rt(5) | rd(5) | 00 000 | 100 011|
2.RTF语言
R[rd] = R[rs] - R[rt]
3.R-Type
4.寄存器堆读
raddr1 = R[rs] = R[Instruction[25 : 21]];
raddr2 = R[rt] = R[Instruction[20 : 16]];
5.ALU
A : R[Instruction[25:21]]
B : R[Instruction[20:16]]
ALUop:减法
```

```
func: 100 011
6.内存访问:
无内存访问
7.跳转:
无跳转
8.寄存器堆写:
wen = 1 需要写 rd
waddr = R[rd] = R[Instrucion[15:11]]
wdata = R[rs] - R[rt] = R[Instruction[25:21]] - R[Instrucion[20:16]];
```

4.and(按位与)

And **AND**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	AND 100100
6	5	5	5	5	6

Format: `AND rd, rs, rt` **MIPS32 (MIPS I)**

Purpose:

To do a bitwise logical AND

Description: `rd ← rs AND rt`

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical AND operation. The result is placed into GPR *rd*.

Restrictions:

None

Operation:

```
GPR[rd] ← GPR[rs] and GPR[rt]
```

Exceptions:

None

```
1.指令格式
| 000 000 | rs(5) | rt(5) | rd(5) | 00 000 | 100 100(AND)|
2.RTF语言
R[rd] = R[rs] & R[rt]
3.R-Type
4.寄存器堆读
raddr1 = R[rs] = R[Instruction[25 : 21]];
raddr2 = R[rt] = R[Instruction[20 : 16]];
5.ALU
A : R[Instruction[25:21]]
B : R[Instruction[20:16]]
ALUop:按位与
func: 100 100
6.内存访问:
无内存访问
7.跳转:
无跳转
8.寄存器堆写:
wen = 1 需要写 rd
waddr = R[rd] = R[Instrucion[15:11]]
wdata = R[rs] & R[rt] = R[Instruction[25:21]] & R[Instrucion[20:16]];
```

5.andi（立即数按位与）

And Immediate **ANDI**

31–26	25–21	20–16	15–0
ANDI 001100	rs	rt	immediate
6	5	5	16

Format: `ANDI rt, rs, immediate` **MIPS32 (MIPS I)**

Purpose:

To do a bitwise logical AND with a constant

Description: `rt ← rs AND immediate`

The 16-bit *immediate* is zero-extended to the left and combined with the contents of GPR *rs* in a bitwise logical AND operation. The result is placed into GPR *rt*.

Restrictions:

None

Operation:

```
GPR[rt] ← GPR[rs] and zero_extend(immediate)
```

Exceptions:

None

```
1. 指令格式：
Instruction[31:26] = 6'b001 100;
Instruction[25:21] = rs;
Instruction[20:16] = rt;
Instruction[15:0] = immediate;
|001 100| rs(5) | rt(5) | immediate(16)|
2.RTF语言：
R[rt] <- R[rs] & zero_extend(immediate)
3. 类型
I-Type(运算)
4，寄存器堆读
raddr1 = rs(Instruction[25:21])  raddr2(无)
5. ALU
A: Readdata1 = R[rs]=R[Instruction[25:21]]
B: zero_extend(immediate)  zero_extend(Instruction[15:0])
ALUop: R[rs] 和 0-延拓后的Immediate按位与
6.内存访问
不涉及内存访问（均设置为0）
7.跳转
不涉及跳转
8.寄存器堆写：
wen = 1     需要写寄存器
waddr = rt = Instruction[20:16]
wdata = R[Instruction[25:21]] & zero_extend(Instruction[15:0]) = R[rs] &
zero(imme) ;
```

6.nor(或非门)

Not Or **NOR**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	NOR 100111
6	5	5	5	5	6

Format: `NOR rd, rs, rt` **MIPS32 (MIPS I)**

Purpose:

To do a bitwise logical NOT OR

Description: `rd ← rs NOR rt`

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical NOR operation. The result is placed into GPR *rd*.

Restrictions:

None

Operation:

```
GPR[rd] ← GPR[rs] nor GPR[rt]
```

Exceptions:

None

```
1.指令格式
| 000 000 | rs(5) | rt(5) | rd(5) | 00 000 | 100 111(NOR)|
2.RTF语言
R[rd] = ~(R[rs] | R[rt])
3.R-Type
4.寄存器堆读
raddr1 = R[rs] = R[Instruction[25 : 21]];
raddr2 = R[rt] = R[Instruction[20 : 16]];
5.ALU
A : R[Instruction[25:21]]
B : R[Instruction[20:16]]
ALUop:按位或非
func: 100 111
6.内存访问:
无内存访问
7.跳转:
无跳转
8.寄存器堆写:
wen = 1 需要写 rd
waddr = R[rd] = R[Instrucion[15:11]]
wdata = R[rs] & R[rt] = ~(R[Instruction[25:21]] | R[Instrucion[20:16]]);
```

7.or(按位或)

Or **OR**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	OR 100101
6	5	5	5	5	6

Format: `OR rd, rs, rt` **MIPS32 (MIPS I)**

Purpose:

To do a bitwise logical OR

Description: `rd ← rs or rt`

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical OR operation. The result is placed into GPR *rd*.

Restrictions:

None

Operation:

```
GPR[rd] ← GPR[rs] or GPR[rt]
```

Exceptions:

None

```
1.指令格式
| 000 000 | rs(5) | rt(5) | rd(5) | 00 000 | 100 101(OR)|
2.RTF语言
R[rd] = (R[rs] | R[rt])
3.R-Type
4.寄存器堆读
raddr1 = R[rs] = R[Instruction[25 : 21]];
raddr2 = R[rt] = R[Instruction[20 : 16]];
5.ALU
A : R[Instruction[25:21]]
B : R[Instruction[20:16]]
ALUop:按位或
func: 100 101
6.内存访问:
无内存访问
7.跳转:
无跳转
8.寄存器堆写:
wen = 1 需要写 rd
waddr = R[rd] = R[Instrucion[15:11]]
wdata = R[rs] & R[rt] = (R[Instruction[25:21]] | R[Instrucion[20:16]]);
```

8.ori(立即数或操作)

Or Immediate **ORI**

31 … 26	25 … 21	20 … 16	15 … 0
ORI 001101	rs	rt	immediate
6	5	5	16

Format: `ORI rt, rs, immediate` **MIPS32 (MIPS I)**

Purpose:

To do a bitwise logical OR with a constant

Description: `rt ← 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

```
1. 指令格式:
Instruction[31:26] = 6'b001 101;
Instruction[25:21] = rs;
Instruction[20:16] = rt;
Instruction[15:0] = immediate;
|001 100| rs(5) | rt(5) | immediate(16)|
2.RTF语言:
R[rt] <- R[rs] | zero_extend(immediate)
3. 类型
I-Type(运算)
4, 寄存器堆读
raddr1 = rs(Instruction[25:21])  raddr2(无)
5. ALU
A: Readdata1 = R[rs]=R[Instruction[25:21]]
B: zero_extend(immediate)  zero_extend(Instruction[15:0])
ALUop: R[rs] 和 0-延拓后的Immediate按位或
6.内存访问
不涉及内存访问（均设置为0）
7.跳转
不涉及跳转
8.寄存器堆写:
wen = 1      需要写寄存器
waddr = rt = Instruction[20:16]
wdata = R[Instruction[25:21]] | zero_extend(Instruction[15:0]) = R[rs] |
zero(imme);
```

9.xor（按位异或）

Exclusive OR	**XOR**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	XOR 100110
6	5	5	5	5	6

Format: `XOR rd, rs, rt` **MIPS32 (MIPS I)**

Purpose:

To do a bitwise logical Exclusive OR

Description: `rd ← rs XOR rt`

Combine the contents of GPR *rs* and GPR *rt* in a bitwise logical Exclusive OR operation and place the result into GPR *rd*.

Restrictions:

None

Operation:

```
GPR[rd] ← GPR[rs] xor GPR[rt]
```

Exceptions:

None

```
1.指令格式
| 000 000 | rs(5) | rt(5) | rd(5) | 00 000 | 100 110(OR)|
2.RTF语言
R[rd] = (R[rs] ^ R[rt])
3.R-Type
4.寄存器堆读
raddr1 = R[rs] = R[Instruction[25 : 21]];
raddr2 = R[rt] = R[Instruction[20 : 16]];
5.ALU
A : R[Instruction[25:21]]
B : R[Instruction[20:16]]
ALUop:按位异或
func: 100 110
6.内存访问:
无内存访问
7.跳转:
无跳转
8.寄存器堆写:
wen = 1 需要写 rd
waddr = R[rd] = R[Instrucion[15:11]]
wdata = R[rs] ^ R[rt] = (R[Instruction[25:21]] ^ R[Instrucion[20:16]]);
```

10.xori(立即数异或)

Exclusive OR Immediate **XORI**

31–26	25–21	20–16	15–0
XORI 001110	rs	rt	immediate
6	5	5	16

Format: `XORI rt, rs, immediate` **MIPS32 (MIPS I)**

Purpose:

To do a bitwise logical Exclusive OR with a constant

Description: `rt ← rs XOR immediate`

Combine the contents of GPR *rs* and the 16-bit zero-extended *immediate* in a bitwise logical Exclusive OR operation and place the result into GPR *rt*.

Restrictions:

None

Operation:

```
GPR[rt] ← GPR[rs] xor zero_extend(immediate)
```

Exceptions:

None

```
1. 指令格式:
Instruction[31:26] = 6'b001 110;
Instruction[25:21] = rs;
Instruction[20:16] = rt;
Instruction[15:0] = immediate;
|001 110| rs(5) | rt(5) | immediate(16)|
2.RTF语言:
R[rt] <- R[rs] ^ zero_extend(immediate)
3. 类型
I-Type(运算)
4, 寄存器堆读
raddr1 = rs(Instruction[25:21])  raddr2(无)
5. ALU
A: Readdata1 = R[rs]=R[Instruction[25:21]]
B: zero_extend(immediate)  zero_extend(Instruction[15:0])
ALUop: R[rs] 和 0-延拓后的Immediate按位异或
6.内存访问
不涉及内存访问（均设置为0）
7.跳转
不涉及跳转
8.寄存器堆写:
wen = 1     需要写寄存器
waddr = rt = Instruction[20:16]
wdata = R[Instruction[25:21]]^ zero_extend(Instruction[15:0]) = R[rs] ^
zero(imme);
```

11.slt(有符号数比较)

> 从slt部分的4条指令，需要用到ALU中定义的相关output

Set on Less Than **SLT**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	SLT 101010
6	5	5	5	5	6

Format: `SLT rd, rs, rt` **MIPS32 (MIPS I)**

Purpose:

To record the result of a less-than comparison

Description: `rd ← (rs < rt)`

Compare the contents of GPR *rs* and GPR *rt* as signed integers and record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

```
if GPR[rs] < GPR[rt] then
    GPR[rd] ← 0^(GPRLEN-1) || 1
else
    GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None

```
1.指令格式
| 000 000 | rs(5) | rt(5) | rd(5) | 00 000 | 101 010(slt)|
2.RTF语言
R[rd] = bool((R[rs] < R[rt]))
3.R-Type
4.寄存器堆读
raddr1 = R[rs] = R[Instruction[25 : 21]];
raddr2 = R[rt] = R[Instruction[20 : 16]];
5.ALU
A : R[Instruction[25:21]]
B : R[Instruction[20:16]]
ALUop:减法
func: 101 010
6.内存访问:
无内存访问
7.跳转:
无跳转
8.寄存器堆写:
wen = 1 需要写 rd
waddr = R[rd] = R[Instrucion[15:11]]
wdata = 31'b0 || (ALUop中减法运算的最高符号位 ^ 溢出标志)
= 31'b0 || (Result_prev[31] ^ overflow);
/*  R[rs] - R[rt] < 0
第一种情况: 符号为负(1), 并且未溢出overflow(0)
第二种情况: 符号为正(0), 但溢出了overflow(1) */
```

12.slti(有符号立即数比较)

Set on Less Than Immediate **SLTI**

31 26	25 21	20 16	15 0
SLTI 001010	rs	rt	immediate
6	5	5	16

Format: `SLTI rt, rs, immediate` **MIPS32 (MIPS I)**

Purpose:

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

Description: `rt ← (rs < immediate)`

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

```
if GPR[rs] < sign_extend(immediate) then
   GPR[rd] ← 0^(GPRLEN-1)|| 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None

```
1. 指令格式:
Instruction[31:26] = 6'b001 010;
Instruction[25:21] = rs;
Instruction[20:16] = rt;
Instruction[15:0] = immediate;
|001 010| rs(5) | rt(5) | immediate(16)|
2.RTF语言:
R[rt] <- bool(R[rs] < sign_extend(immediate))
3. 类型
I-Type(运算)
4, 寄存器堆读
raddr1 = R(Instruction[25:21])  raddr2(无)
5. ALU
A: Readdata1 = R[rs]=R[Instruction[25:21]]
B: sign_extend(immediate)  sign_extend(Instruction[15:0])
ALUop: R[rs] 和 延拓后的Immediate相减
opcode: 001 010
6.内存访问
不涉及内存访问（均设置为0）
7.跳转
不涉及跳转
8.寄存器堆写:
wen = 1      需要写寄存器
waddr = rt = Instruction[20:16]
wdata = R[Instruction[25:21]]- sign_extend(Instruction[15:0]) = R[rs] -
sign(imme);
wdata = 31'b0 || (ALUop中减法运算的最高符号位 ^ 溢出标志)
= 31'b0 || (Result_prev[31] ^ Overflow);
/*  R[rs] - Sign(immediate) < 0
```

```
第一种情况：符号为负（1），并且未溢出overflow(0)
第二种情况：符号为正（0），但溢出了overflow(1) */
```

13.sltu(无符号数比较)

Set on Less Than Unsigned **SLTU**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	SLTU 101011
6	5	5	5	5	6

Format: `SLTU rd, rs, rt` **MIPS32 (MIPS I)**

Purpose:

To record the result of an unsigned less-than comparison

Description: `rd ← (rs < rt)`

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers and record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

```
if (0 || GPR[rs]) < (0 || GPR[rt]) then
    GPR[rd] ← 0^(GPRLEN-1) || 1
else
    GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None

```
1.指令格式
| 000 000 | rs(5) | rt(5) | rd(5) | 00 000 | 101 011(sltu)|
2.RTF语言
R[rd] = bool((R[rs] < R[rt]))
3.R-Type
4.寄存器堆读
raddr1 = R[rs] = R[Instruction[25 : 21]];
raddr2 = R[rt] = R[Instruction[20 : 16]];
5.ALU
A : R[Instruction[25:21]]
B : R[Instruction[20:16]]
ALUop:减法
func: 101 011
6.内存访问:
无内存访问
7.跳转:
无跳转
8.寄存器堆写:
wen = 1 需要写 rd
waddr = R[rd] = R[Instrucion[15:11]]
wdata = 31'b0 || (ALUop中减法运算的最高符号位)
= 31'b0 || (Result_prev[31]);
/*看符号位*/
```

14.sltui(无符号立即数比较)

Set on Less Than Immediate Unsigned **SLTIU**

31–26	25–21	20–16	15–0
SLTIU 001011	rs	rt	immediate
6	5	5	16

Format: `SLTIU rt, rs, immediate` **MIPS32 (MIPS I)**

Purpose:

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

Description: `rt ← (rs < immediate)`

Compare the contents of GPR *rs* and the sign-extended 16-bit *immediate* as unsigned integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

Because the 16-bit *immediate* is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

```
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
    GPR[rd] ← 0^(GPRLEN-1) || 1
else
    GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None

```
1. 指令格式:
Instruction[31:26] = 6'b001 011;
Instruction[25:21] = rs;
Instruction[20:16] = rt;
Instruction[15:0] = immediate;
|001 011| rs(5) | rt(5) | immediate(16)|
2.RTF语言:
R[rt] <- bool(R[rs] < sign_extend(immediate))
3. 类型
I-Type(运算)
4, 寄存器堆读
raddr1 = R(Instruction[25:21])  raddr2(无)
5. ALU
A: Readdata1 = R[rs]=R[Instruction[25:21]]
B: sign_extend(immediate)  sign_extend(Instruction[15:0])
ALUop: R[rs] 和 无符号延拓后的Immediate相减
opcode: 001 011
6.内存访问
不涉及内存访问（均设置为0）
7.跳转
不涉及跳转
8.寄存器堆写:
wen = 1     需要写寄存器
waddr = rt = Instruction[20:16]
wdata = R[Instruction[25:21]]- sign_extend(Instruction[15:0]) = R[rs] -
sign(imme);
wdata = 31'b0 || (ALUop中减法运算的最高符号位 )
```

```
= 31'b0 || (Result_prev[31]);
```

移位类

15.sll(shamt左移)

Shift Word Left Logical **SLL**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	0 00000	rt	rd	sa	SLL 000000
6	5	5	5	5	6

Format: `SLL rd, rt, sa` **MIPS32 (MIPS I)**

Purpose:

To left-shift a word by a fixed number of bits

Description: `rd ← rt << sa`

The contents of the low-order 32-bit word of GPR *rt* are shifted left, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.

Restrictions:

None

Operation:

$$
\begin{aligned}
&s \quad\quad\ \leftarrow sa\\
&temp \quad \leftarrow GPR[rt]_{(31-s)..0} \ || \ 0^{s}\\
&GPR[rd] \leftarrow temp
\end{aligned}
$$

Exceptions:

None

Programming Notes:

SLL r0, r0, 0, expressed as NOP, is the assembly idiom used to denote no operation.

SLL r0, r0, 1, expressed as SSNOP, is the assembly idiom used to denote no operation that causes an issue break on superscalar processors.

```
1.指令格式
| 000 000 | 000 00 | rt(5) | rd(5) | sa(5) | 000 000(sll)|
2.RTF语言
R[rd] = R[rt] << sa
3.R-Type
4.寄存器堆读
raddr1 = 无;
raddr2 = R[rt] = R[Instruction[20 : 16]];
5.ALU
不需要ALU
6.内存访问:
无内存访问
7.跳转:
无跳转
8.寄存器堆写:
wen = 1 需要写 rd
waddr = R[rd] = R[Instruction[15:11]]
wdata = R[rt] << sa = R[Instruction[20:16]] << Instruction[10:6];
// 实际上, wdata = shifter(R[Instruction[20:16]],Instruction[10:6],左移)
9.移位器
A [31:0] = R[rt] = R[Instruction[20:16]];
B [4 : 0] = Instruction[10 : 6]; // B 表示移动多少
Shiftop: 左移, 后面添 0
输出是 : Result
```

```
需要移位器
```

16.sllv（字符寄存器数据左移）

Shift Word Left Logical Variable **SLLV**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	SLLV 000100
6	5	5	5	5	6

Format: `SLLV rd, rt, rs` **MIPS32 (MIPS I)**

Purpose: To left-shift a word by a variable number of bits

Description: `rd ← rt << rs`

The contents of the low-order 32-bit word of GPR *rt* are shifted left, inserting zeros into the emptied bits; the result word is placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions: None

Operation:

$s \leftarrow GPR[rs]_{4..0}$

$temp \leftarrow GPR[rt]_{(31-s)..0} \,||\, 0^s$

$GPR[rd] \leftarrow temp$

Exceptions: None

Programming Notes:

None

```
1.指令格式
| 000 000 | rs(5) | rt(5) | rd(5) | 00 000 | 000 100(sllv)|
2.RTF语言
R[rd] = R[rt] << R[rs][5:0]
3.R-Type
4.寄存器堆读
raddr1 = R[rs][5:0] = R[Instruction[25:21]][5:0];
raddr2 = R[rt] = R[Instruction[20 : 16]];
5.ALU
不需要ALU
6.内存访问:
无内存访问
7.跳转:
无跳转
8.寄存器堆写:
wen = 1 需要写 rd
waddr = R[rd] = R[Instruction[15:11]]
wdata = R[rt] << R[rs][5:0] = R[Instruction[20:16]] << R[Instruction[25:21]]
[5:0];
// 实际上, wdata = shifter(R[Instruction[20:16]],Instruction[10:6],左移)
9.移位器
A [31:0] = R[rt] = R[Instruction[20:16]];
B [4 : 0] = R[Instruction[25 : 21]][5:0]; // B 表示移动多少
// A 对应Readdata2,B对应Readdata1[5:0]
Shiftop: 左移, 后面添 0
输出是 : Result
需要移位器
```

17.sra(shamt算术右移)

Shift Word Right Arithmetic **SRA**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	0 00000	rt	rd	sa	SRA 000011
6	5	5	5	5	6

Format: `SRA rd, rt, sa` **MIPS32 (MIPS I)**

Purpose:

To execute an arithmetic right-shift of a word by a fixed number of bits

Description: `rd ← rt >> sa (arithmetic)`

The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.

Restrictions:

None

Operation:

$$
\begin{aligned}
&s \leftarrow sa \\
&temp \leftarrow (GPR[rt]_{31})^{s} \ || \ GPR[rt]_{31..s} \\
&GPR[rd] \leftarrow temp
\end{aligned}
$$

Exceptions: None

```
1.指令格式
| 000 000 | 000 00 | rt(5) | rd(5) | sa(5) | 000 011(sra)|
2.RTF语言
R[rd] = R[rt] >> sa
3.R-Type
4.寄存器堆读
raddr1 = 无;
raddr2 = R[rt] = R[Instruction[20 : 16]];
5.ALU
不需要ALU
6.内存访问:
无内存访问
7.跳转:
无跳转
8.寄存器堆写:
wen = 1 需要写 rd
waddr = R[rd] = R[Instruction[15:11]]
wdata = R[rt] >> sa = R[Instruction[20:16]] << Instruction[10:6];
// 实际上,wdata = shifter(R[Instruction[20:16]],Instruction[10:6],右移)
9.移位器
A [31:0] = R[rt] = R[Instruction[20:16]];
B [4 : 0] = Instruction[10 : 6]; // B 表示移动多少
Shiftop: 算术右移2'b11, 前面添 R[Instruction[20:16]][20]
输出是 : Result
需要移位器
`define DATA_WIDTH 32
module shifter(
    input [`DATA_WIDTH - 1 : 0] A,
    input [ 4 : 0] B,
    input [ 1 : 0] Shiftop,
    output [`DATA_WIDTH - 1 : 0] Result);

    assign Result = ( {Shiftop == 2'b00 }  & (A[31-B:0] || {B{0}} ) ) |
```

```
        ({Shiftop == 2'b11} & ({B{A[31]}}  || A[31:B]));
endmodule
```

18.srav（算术右移（寄存器））

Shift Word Right Arithmetic Variable **SRAV**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	SRAV 000111
6	5	5	5	5	6

Format: `SRAV rd, rt, rs` **MIPS32 (MIPS I)**

Purpose:

To execute an arithmetic right-shift of a word by a variable number of bits

Description: `rd ← rt >> rs (arithmetic)`

The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions:

None

Operation:

$$
\begin{aligned}
&s \leftarrow \mathrm{GPR}[rs]_{4..0}\\
&temp \leftarrow (\mathrm{GPR}[rt]_{31})^{s} \parallel \mathrm{GPR}[rt]_{31..s}\\
&\mathrm{GPR}[rd] \leftarrow temp
\end{aligned}
$$

Exceptions:

None

```
1.指令格式
| 000 000 | rs(5) | rt(5) | rd(5) | 00 000 | 000 111(srav)|
2.RTF语言
R[rd] = R[rt] >> R[rs][5:0]
3.R-Type
4.寄存器堆读
raddr1 = R[rs][5:0] = R[Instruction[25:21]][5:0];
raddr2 = R[rt] = R[Instruction[20 : 16]];
5.ALU
不需要ALU
6.内存访问:
无内存访问
7.跳转:
无跳转
8.寄存器堆写:
wen = 1 需要写 rd
waddr = R[rd] = R[Instruction[15:11]]
wdata = R[rt] >> R[rs][5:0] = R[Instruction[20:16]] >> R[Instruction[25:21]]
[5:0];
// 实际上，wdata = shifter(R[Instruction[20:16]],Instruction[10:6],左移)
9.移位器
A [31:0] = R[rt] = R[Instruction[20:16]];
B [4 : 0] = R[Instruction[25 : 21]][5:0]; // B 表示移动多少
// A 对应Readdata2,B对应Readdata1[5:0]
Shiftop: 右移，前面添 R[rt][31] = R[Instruction[20:16]][31];
输出是 : Result
需要移位器
```

19.srl(逻辑右移)

Shift Word Right Logical **SRL**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SRL 000010
6	5	5	5	5	6

Format: `SRL rd, rt, sa` **MIPS32 (MIPS I)**

Purpose:

To execute a logical right-shift of a word by a fixed number of bits

Description: `rd ← rt >> sa (logical)`

The contents of the low-order 32-bit word of GPR *rt* are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.

Restrictions:

None

Operation:

```
s        ← sa
temp     ← 0^s || GPR[rt]31..s
GPR[rd]← temp
```

Exceptions:

None

```
1.指令格式
| 000 000 | 000 00 | rt(5) | rd(5) | sa(5) | 000 010(srl)|
2.RTF语言
R[rd] = R[rt] >> sa(logical)
3.R-Type
4.寄存器堆读
raddr1 = 无;
raddr2 = R[rt] = R[Instruction[20 : 16]];
5.ALU
不需要ALU
6.内存访问:
无内存访问
7.跳转:
无跳转
8.寄存器堆写:
wen = 1 需要写 rd
waddr = R[rd] = R[Instruction[15:11]]
wdata = R[rt] >> sa = R[Instruction[20:16]] >> Instruction[10:6];
// 实际上，wdata = shifter(R[Instruction[20:16]],Instruction[10:6],逻辑右移)
9.移位器
A [31:0] = R[rt] = R[Instruction[20:16]];
B [4 : 0] = Instruction[10 : 6]; // B 表示移动多少
Shiftop: 逻右移2'b10, 前面添 Instruction[10:6]个 0
输出是 : Result
需要移位器
`define DATA_WIDTH 32
module shifter(
    input [`DATA_WIDTH - 1 : 0] A,
    input [ 4 : 0] B,
    input [ 1 : 0] Shiftop,
    output [`DATA_WIDTH - 1 : 0] Result);
```

```
    assign Result = ( {Shiftop == 2'b00 }  & (A[31-B:0] || {B{0}} ) ) |
        ({Shiftop == 2'b11} & ({B{A[31]}}  || A[31:B])) |
        ({Shiftop == 2'b10} & ({B{0}} || A[31:B] ));
endmodule
```

20.srlv(寄存器逻辑右移)

Shift Word Right Logical Variable **SRLV**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	SRLV 000110
6	5	5	5	5	6

Format: `SRLV rd, rt, rs` **MIPS32 (MIPS I)**

Purpose:

To execute a logical right-shift of a word by a variable number of bits

Description: `rd ← rt >> rs (logical)`

The contents of the low-order 32-bit word of GPR *rt* are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions:

None

Operation:

$$
\begin{aligned}
&s \leftarrow \mathrm{GPR}[rs]_{4..0} \\
&temp \leftarrow 0^{s} \mid\mid \mathrm{GPR}[rt]_{31..s} \\
&\mathrm{GPR}[rd] \leftarrow temp
\end{aligned}
$$

Exceptions:

None

```
1.指令格式
| 000 000 | rs(5) | rt(5) | rd(5) | 00 000 | 000 110(srlv)|
2.RTF语言
R[rd] = R[rt] >> R[rs][5:0]
3.R-Type
4.寄存器堆读
raddr1 = R[rs][5:0] = R[Instruction[25:21]][5:0];
raddr2 = R[rt] = R[Instruction[20 : 16]];
5.ALU
不需要ALU
6.内存访问:
无内存访问
7.跳转:
无跳转
8.寄存器堆写:
wen = 1 需要写 rd
waddr = R[rd] = R[Instruction[15:11]]
wdata = R[rt] >> R[rs][5:0] = R[Instruction[20:16]] >> R[Instruction[25:21]]
[5:0];
// 实际上, wdata = shifter(R[Instruction[20:16]],Instruction[10:6],左移)
9.移位器
A [31:0] = R[rt] = R[Instruction[20:16]];
B [4 : 0] = R[Instruction[25 : 21]][5:0]; // B 表示移动多少
// A 对应Readdata2,B对应Readdata1[5:0]
Shiftop: 逻辑右移, 前面添 R[Instruction[25 : 21]] 个 {0}
输出是 : Result
需要移位器
```

跳转类

21.bne（branch on not equal)

Branch on Not Equal **BNE**

31–26	25–21	20–16	15–0
BNE 000101	rs	rt	offset
6	5	5	16

Format: `BNE rs, rt, offset` **MIPS32 (MIPS I)**

Purpose:

To compare GPRs then do a PC-relative conditional branch

Description: `if rs ≠ 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 not 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.

```
1. 指令格式:
Instruction[31:26] = 6'b000 101;
Instruction[25:21] = rs;
Instruction[20:16] = rt;
Instruction[15:0] = offset;
|000 101| rs(5) | rt(5) | offset(16)|
2.RTF语言:
if R[rs] != R[rt] then PC <- PC + sign_extend(offset || {2{0}})
3. 类型
I-Type(分支)
4, 寄存器堆读
raddr1 = R(Instruction[25:21]) = R[rs]
raddr2 = R(Instruction[20:16]) = R[rt]
5. ALU
A: Readdata1 = R[rs]=R[Instruction[25:21]]
B: Readdata2 = R[rt] = R[Instruction[20:16]]
ALUop: R[rs] - R[rt]
opcode: 000 101
6.内存访问
不涉及内存访问（均设置为0）
7.跳转
```

```
跳转地址：PC + sign_extend(offset || {2{0}})
= PC + sign_extend(Instruction[15:0] || {2{0}})
// ~Zero & branch == 1
地址更新条件：Zero标志位为 0,两寄存器内容不相等
8.寄存器堆写：
不需要写寄存器
```

22.beq (branch on equal)

Branch on Equal **BEQ**

31–26	25–21	20–16	15–0
BEQ 000100	rs	rt	offset
6	5	5	16

Format: `BEQ rs, rt, offset` **MIPS32 (MIPS I)**

Purpose:

To compare GPRs then do a PC-relative conditional branch

Description: `if rs = 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.

```
1. 指令格式：
Instruction[31:26] = 6'b000 100;
Instruction[25:21] = rs;
Instruction[20:16] = rt;
Instruction[15:0] = offset;
|000 100| rs(5) | rt(5) | offset(16)|
2.RTF语言：
if R[rs] == R[rt] then PC <- PC + sign_extend(offset || {2{0}})
3. 类型
I-Type(分支)
4, 寄存器堆读
raddr1 = R(Instruction[25:21]) = R[rs]
raddr2 = R(Instruction[20:16]) = R[rt]
5. ALU
A: Readdata1 = R[rs]=R[Instruction[25:21]]
B: Readdata2 = R[rt] = R[Instruction[20:16]]
```

```
ALUop: R[rs] - R[rt]
opcode: 000 100
6.内存访问
不涉及内存访问（均设置为0）
7.跳转
跳转地址：PC + sign_extend(offset || {2{0}})
= PC + sign_extend(Instruction[15:0] || {2{0}})
// zero & branch == 1
地址更新条件：zero标志位为 1,两寄存器内容相等
8.寄存器堆写：
不需要写寄存器
```

23.bgez　(branch on rs >= 0)

Branch on Greater Than or Equal to Zero **BGEZ**

31–26	25–21	20–16	15–0
REGIMM 000001	rs	BGEZ 00001	offset
6	5	5	16

Format: `BGEZ rs, offset` **MIPS32 (MIPS I)**

Purpose:

To test a GPR then do a PC-relative conditional branch

Description: `if rs ≥ 0 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* are greater than or equal to zero (sign bit is 0), 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] ≥ 0^GPRLEN
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.

```
1. 指令格式：
Instruction[31:26] = 6'b000 001(REGIMM);
Instruction[25:21] = rs;
Instruction[20:16] = 000 01(bgez);
Instruction[15:0] = offset;
|000 001| rs(5) | 000 01(bgez) | offset(16)|
2.RTF语言：
if R[rs] >= 0 then PC <- PC + sign_extend(offset || {2{0}})
3. 类型
REGIMM
4, 寄存器堆读
raddr1 = R(Instruction[25:21]) = R[rs]
```

```
// 和 0 号寄存器中的内容比较
raddr2 = R(5{0}) = R[0]
5. ALU
A: Readdata1 = R[rs]=R[Instruction[25:21]]
B: Readdata2 = R[0] = 5'b0
ALUop: R[rs] - R[0]
opcode: 000 001
6.内存访问
不涉及内存访问（均设置为0）
7.跳转
跳转地址: PC + sign_extend(offset || {2{0}})
= PC + sign_extend(Instruction[15:0] || {2{0}})

地址更新条件: R[rs] 中的内容 >= 0
一种可行的方案是借助 ALU,判断最终得到 - 0 的最高符号位 Result_prev[31]是否为 0
if (Result_prev[31] == 0) then branch

8.寄存器堆写:
不需要写寄存器
```

24.bgtz(branch on rs > 0)

Branch on Greater Than Zero **BGTZ**

31–26	25–21	20–16	15–0
BGTZ 000111	rs	0 00000	offset
6	5	5	16

Format: `BGTZ rs, offset` **MIPS32 (MIPS I)**

Purpose:

To test a GPR then do a PC-relative conditional branch

Description: `if rs > 0 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* are greater than zero (sign bit is 0 but value not zero), 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] > 0^GPRLEN
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.

```
1. 指令格式:
Instruction[31:26] = 6'b000 111(BGTZ);
Instruction[25:21] = rs;
```

```
Instruction[20:16] = 000 00;
Instruction[15:0] = offset;
|000 111(bgtz)| rs(5) | 000 00 | offset(16)|
2.RTF语言:
if R[rs] > 0 then PC <- PC + sign_extend(offset || {2{0}})
3. 类型
    REGIMM / I-type(分支)
4, 寄存器堆读
    raddr1 = R(Instruction[25:21]) = R[rs]
    // 和 0 号寄存器中的内容比较
    raddr2 = R(Instruction[20:16]) = R[0]
5. ALU
    A: Readdata1 = R[rs]=R[Instruction[25:21]]
    B: Readdata2 = R[Instruction[20:16]] = 5'b0
    ALUop: R[rs] - R[0]
    opcode: 000 111
6.内存访问
不涉及内存访问（均设置为0）
7.跳转
跳转地址: PC + sign_extend(offset || {2{0}})
= PC + sign_extend(Instruction[15:0] || {2{0}})

地址更新条件: R[rs] 中的内容 > 0
    一种可行的方案是借助 ALU,判断最终得到 - 0 的最高符号位 Result_prev[31]是否为 0,
    并且需要通过溢出进行判断:
    1.首先Zero不能等于1, 即两者不能相等 2.其次判断符号
    // 符号位结果为 0, 表明为正数
    if ( ~ Zero && ~(Result[31]))
        then branch
8.寄存器堆写:
不需要写寄存器
```

25.blez（branch on <= 0)

Branch on Less Than or Equal to Zero	BLEZ

31 - 26	25 - 21	20 - 16	15 - 0
BLEZ 000110	rs	0 00000	offset
6	5	5	16

Format: `BLEZ rs, offset` **MIPS32 (MIPS I)**

Purpose:

To test a GPR then do a PC-relative conditional branch

Description: `if rs ≤ 0 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* are less than or equal to zero (sign bit is 1 or value is zero), 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] ≤ 0^GPRLEN
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.

```
1. 指令格式：
Instruction[31:26] = 6'b000 110(BLEZ);
Instruction[25:21] = rs;
Instruction[20:16] = 000 00;
Instruction[15:0] = offset;
|000 110(blez)| rs(5) | 000 00 | offset(16)|
2.RTF语言：
if R[rs] <= 0 then PC <- PC + sign_extend(offset || {2{0}})
3. 类型
    REGIMM/I-type(分支)
4, 寄存器堆读
raddr1 = R(Instruction[25:21]) = R[rs]
// 和 0 号寄存器中的内容比较
    raddr2 = R(Instruction[20:16]) = R[0]
5. ALU
A: Readdata1 = R[rs]=R[Instruction[25:21]]
    B: Readdata2 = R(Instruction[20:16]) = R[0]
ALUop: R[rs] - R[0]
opcode: 000 110
6.内存访问
不涉及内存访问（均设置为0）
7.跳转
跳转地址：PC + sign_extend(offset || {2{0}})
= PC + sign_extend(Instruction[15:0] || {2{0}})

    地址更新条件：R[rs] 中的内容 <= 0
```

```
一种可行的方案是借助 ALU,判断最终得到 - 0 的最高符号位 Result_prev[31]是否为 0
    if (Result_prev[31] == 1 || Zero) then branch

8.寄存器堆写:
不需要写寄存器
```

26.bltz(branch on < 0)

Branch on Less Than Zero **BLTZ**

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BLTZ 00000	offset
6	5	5	16

Format: `BLTZ rs, offset` **MIPS32 (MIPS I)**

Purpose:

To test a GPR then do a PC-relative conditional branch

Description: `if rs < 0 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* are less than zero (sign bit is 1), 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] < 0^GPRLEN
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 and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.

```
1. 指令格式:
Instruction[31:26] = 6'b000 001(REGIMM);
Instruction[25:21] = rs;
Instruction[20:16] = 000 00(BLTZ);
Instruction[15:0] = offset;
|000 001| rs(5) | 000 00 | offset(16)|
2.RTF语言:
if R[rs] < 0 then PC <- PC + sign_extend(offset || {2{0}})
3. 类型
    REGIMM / I-type(分支)
4, 寄存器堆读
    raddr1 = R(Instruction[25:21]) = R[rs]
    // 和 0 号寄存器中的内容比较
    raddr2 = R(Instruction[20:16]) = R[0]
5. ALU
    A: Readdata1 = R[rs]=R[Instruction[25:21]]
    B: Readdata2 = R[Instruction[20:16]] = 5'b0
    ALUop: R[rs] - R[0]
```

```
    opcode: 000 001
6.内存访问
不涉及内存访问（均设置为0）
7.跳转
跳转地址：PC + sign_extend(offset || {2{0}})
= PC + sign_extend(Instruction[15:0] || {2{0}})

    地址更新条件：R[rs] 中的内容 < 0
    一种可行的方案是借助 ALU,判断最终得到 - 0 的最高符号位 Result_prev[31]是否为 1
    1.首先Zero不能等于1，即两者不能相等 2.其次判断符号
    // 符号位结果为 1，表明为负数
    if ( ~ Zero && (Result[31]))
        then branch
8.寄存器堆写：
不需要写寄存器
```

27.J（跳转）

Jump **J**

31 26	25 0
J 000010	instr_index
6	26

Format: `J target` **MIPS32 (MIPS I)**

Purpose:

To branch within the current 256 MB-aligned region

Description:

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:

$$\begin{aligned}&\mathtt{I:}\\&\mathtt{I+1:PC} \leftarrow \mathrm{PC}_{\mathrm{GPRLEN..28}} \;||\; \mathtt{instr\_index} \;||\; 0^2\end{aligned}$$

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 jump 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.

```
1. 指令格式：
|000 010(J)|  instr_index(26) |
2.RTF语言：
PC <- PC[31:28] || instr_index || {2{0}}
3. 类型
    J-Type
4, 寄存器堆读
```

```
    不需要读寄存器堆
5. ALU
    不需要这里的ALU
6.内存访问
不涉及内存访问（均设置为0）
7.跳转
跳转地址：PC[31:28] || Instruction[25:0] || {2{0}})

    地址更新条件：无条件
    需要地址加法器
    1.PC已经变成 PC + 4
    2. PC[31:28] 取左移两位后的 instr_index 连接

8.寄存器堆写：
不需要写寄存器
```

28.Jal（跳转并记录过程调用返回）

Jump and Link **JAL**

31 26	25 0
JAL 000011	instr_index
6	26

Format: `JAL target` **MIPS32 (MIPS I)**

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..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.

```
1. 指令格式：
|000 011(J)|  instr_index(26) |
2.RTF语言：
R[31] <- PC + 8
PC <- PC[31:28] || instr_index || {2{0}}
```

```
3. 类型
    J-Type
4，寄存器堆读
    不需要读寄存器堆
5. ALU
    不需要这里的ALU
6.内存访问
不涉及内存访问（均设置为0）
7.跳转
跳转地址：PC[31:28] || Instruction[25:0] || {2{0}})

    地址更新条件：无条件
    需要地址加法器
    1.PC已经变成 PC + 4
    2. PC[31:28] 取左移两位后的 instr_index 连接

8.寄存器堆写：
    wen = 1
    waddr = 31
    wdata = PC + 8
```

29.jr(跳转到寄存器内容处的PC)

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 (MIPS I)**

Purpose:

To execute a branch to an instruction address in a register

Description: `PC ← 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 MIPS16 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 MIPS16 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 MIPS16 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.

At this time the only defined hint field value is 0, which sets default handling of JR. Future versions of the architecture may define additional hint values.

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 Config1_CA = 0 then
        PC ← temp
    else
        PC ← temp_GPRLEN-1..1 || 0
        ISAMode ← temp_0
    endif
```

Exceptions:

None

```
1. 指令格式:
| 000 000 | rs(5) | 0 (10) | hint(5) | 001 000 (JR)|
2.RTF语言:
PC <- R[rs]
3. 类型
    R-Type
4, 寄存器堆读
raddr1 = R[Instruction[25:21]];
raddr2 = 无;
5. ALU
    不需要这里的ALU
6.内存访问
不涉及内存访问(均设置为0)
7.跳转
跳转地址: R[rs]
地址更新条件:  无条件
8.寄存器堆写:
不需要写寄存器堆
```

30.jalr（跳转并保存返回地址）

Jump and Link Register **JALR**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	0 00000	rd	hint	JALR 001001
6	5	5	5	5	6

Format: `JALR rs (rd = 31 implied)` **MIPS32 (MIPS I)**
`JALR rd, rs` **MIPS32 (MIPS I)**

Purpose:

To execute a procedure call to an instruction address in a register

Description: `rd ← return_addr, PC ← rs`

Place the return address link in GPR *rd*. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

For processors that do not implement the MIPS16 ASE:

- Jump to the effective target address in GPR *rs*. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

For processors that do implement the MIPS16 ASE:

- Jump to the effective target address in GPR *rs*. 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

At this time the only defined hint field value is 0, which sets default handling of JALR. Future versions of the architecture may define additional hint values.

Restrictions:

Register specifiers *rs* and *rd* must not be equal, because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

The effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16 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 MIPS16 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.

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

```
1. 指令格式:
| 000 000 | rs(5) | 0 (5) | rd(5) | hint(5) | 001 001 (jalr) |
2.RTF语言:
```

```
PC <- R[rs]
R[rd] <- PC + 8
3. 类型
    R-Type
4，寄存器堆读
raddr1 = R[Instruction[25:21]];
raddr2 = 无;
5. ALU
    需要计算 PC + 8
6.内存访问
不涉及内存访问（均设置为0）
7.跳转
跳转地址: R[rs]
地址更新条件:  无条件
8.寄存器堆写:
wen = 1
waddr = R[rd] = R[Instruction[15:11]]
wdata = PC + 8
```

访存类

31.lb（加载 8-bit 字节）

Load Byte **LB**

31–26	25–21	20–16	15–0
LB 100000	base	rt	offset
6	5	5	16

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

Purpose:

To load a byte from memory as a signed value

Description: `rt ← memory[base+offset]`

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, sign-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

Operation:

$$
\begin{aligned}
&\texttt{vAddr} \leftarrow \texttt{sign\_extend(offset) + GPR[base]}\\
&\texttt{(pAddr, CCA)} \leftarrow \texttt{AddressTranslation (vAddr, DATA, LOAD)}\\
&\texttt{pAddr} \leftarrow \texttt{pAddr}_{\texttt{PSIZE-1..2}} \texttt{ || (pAddr}_{\texttt{1..0}} \texttt{ xor ReverseEndian}^2\texttt{)}\\
&\texttt{memword} \leftarrow \texttt{LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)}\\
&\texttt{byte} \leftarrow \texttt{vAddr}_{\texttt{1..0}} \texttt{ xor BigEndianCPU}^2\\
&\texttt{GPR[rt]} \leftarrow \texttt{sign\_extend(memword}_{\texttt{7+8*byte..8*byte}}\texttt{)}
\end{aligned}
$$

Exceptions:

TLB Refill, TLB Invalid, Address Error

```
1. 指令格式:
| 100 000 | base(5) | rt(5) | offset(16) |
2.RTF语言:
R[rt] <- mem[base + offset]
3. 类型
I-Type(访存)
4，寄存器堆读
raddr1 = R[Instruction[25:21]] = R[base] // 可以认为是 R[rs]
```

```
raddr2 = 无
5. ALU
A: R[Instruction[25:21]] = R[base]
B: sign_extend(Instruction[15:0])
ALUop : 有符号数加法
6.内存访问
Address : ALU Result = R[base] + Sign_extend(Instruction[15:0])
MemRead : 1
Read需要在内部赋值 Read_strb 这里只能有{1000，0100，0010，0001}四种之一
MemWrite : 0
Write_Data : 不需要写
Write_Strb : 0000
7.跳转
不需要跳转
8.寄存器堆写:
wen = 1
waddr = R[rt] = R[Instruction[20:16]]
wdata =sign_extend(mem[R[Instruction[25:21]] + sign_extend(Instruction[15:0])])
即 sign_extend(memword[7 + 8*byte,8*byte])
// 需要对取得的 8 -bit 进行符号位扩展后放入R[rt]
```

32.lh(加载半字 16-bit)

Load Halfword **LH**

31–26	25–21	20–16	15–0
LH 100001	base	rt	offset
6	5	5	16

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

Purpose:

To load a halfword from memory as a signed value

Description: `rt ← memory[base+offset]`

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, sign-extended, 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 the least-significant bit of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr_0 ≠ 0 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor (ReverseEndian || 0))
memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte   ← vAddr_1..0 xor (BigEndianCPU || 0)
GPR[rt] ← sign_extend(memword_15+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

```
1. 指令格式:
| 100 001 | base(5) | rt(5) | offset(16) |
2.RTF语言:
R[rt] <- mem[base + offset]
3. 类型
I-Type(访存)
```

```
4, 寄存器堆读
raddr1 = R[Instruction[25:21]] = R[base] // 可以认为是 R[rs]
raddr2 = 无
5. ALU
A: R[Instruction[25:21]] = R[base]
B: sign_extend(Instruction[15:0])
ALUop : 有符号数加法
6.内存访问
Address : ALU Result = R[base] + Sign_extend(Instruction[15:0])
MemRead : 1
Read需要在内部赋值 Read_strb 这里只能有{1100，0011}两种之一
MemWrite : 0
Write_Data : 不需要写
Write_Strb : 0000
7.跳转
不需要跳转
8.寄存器堆写:
wen = 1
waddr = R[rt] = R[Instruction[20:16]]
wdata =sign_extend(mem[R[Instruction[25:21]] + sign_extend(Instruction[15:0])])
即 sign_extend(memword[15 + 8*byte,8*byte])
// 需要对取得的 16 -bit 进行符号位扩展后放入R[rt]
```

33.lw(加载字 32-bit)

Load Word **LW**

31 — 26	25 — 21	20 — 16	15 — 0
LW 100011	base	rt	offset
6	5	5	16

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

Purpose:

To load a word from memory as a signed value

Description: `rt ← memory[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 vAddr_{1..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

```
1. 指令格式:
| 100 011 | base(5) | rt(5) | offset(16) |
2.RTF语言:
R[rt] <- mem[base + offset]
3. 类型
```

```
I-Type(访存)
4, 寄存器堆读
raddr1 = R[Instruction[25:21]] = R[base] // 可以认为是 R[rs]
raddr2 = 无
5. ALU
A: R[Instruction[25:21]] = R[base]
B: sign_extend(Instruction[15:0])
ALUop : 有符号数加法
6.内存访问
Address : ALU Result = R[base] + Sign_extend(Instruction[15:0])
MemRead : 1
Read需要在内部赋值 Read_strb 这里为 1111
MemWrite : 0
Write_Data : 不需要写
Write_Strb : 0000
7.跳转
不需要跳转
8.寄存器堆写:
wen = 1
waddr = R[rt] = R[Instruction[20:16]]
wdata =mem[R[Instruction[25:21]] + sign_extend(Instruction[15:0])]
// 需要对取得的 32 -bit 放入R[rt]
```

34.lbu（加载无符号字节）

Load Byte Unsigned **LBU**

31–26	25–21	20–16	15–0
LBU 100100	base	rt	offset
6	5	5	16

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

Purpose:

To load a byte from memory as an unsigned value

Description: `rt ← memory[base+offset]`

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, zero-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
memword← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr_1..0 xor BigEndianCPU^2
GPR[rt]← zero_extend(memword_7+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Address Error

```
1. 指令格式:
| 100 100 | base(5) | rt(5) | offset(16) |
2.RTF语言:
R[rt] <- mem[base + offset]
3. 类型
I-Type(访存)
4, 寄存器堆读
```

```
raddr1 = R[Instruction[25:21]] = R[base] // 可以认为是 R[rs]
raddr2 = 无
5. ALU
A: R[Instruction[25:21]] = R[base]
B: sign_extend(Instruction[15:0])
ALUop : 有符号数加法
6.内存访问
Address : ALU Result = R[base] + sign_extend(Instruction[15:0])
MemRead : 1
Read需要在内部赋值 Read_strb 这里只能有{1000，0100，0010，0001}四种之一
MemWrite : 0
Write_Data : 不需要写
Write_Strb : 0000
7.跳转
不需要跳转
8.寄存器堆写：
wen = 1
waddr = R[rt] = R[Instruction[20:16]]
wdata =zero_extend(mem[R[Instruction[25:21]] + sign_extend(Instruction[15:0])])
即 zero_extend(memword[7 + 8*byte,8*byte])
// 需要对取得的 8 -bit 进行零延拓后放入R[rt]
```

35.lhu（加载无符号半字）

Load Halfword Unsigned **LHU**

31–26	25–21	20–16	15–0
LHU 100101	base	rt	offset
6	5	5	16

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

Purpose:

To load a halfword from memory as an unsigned value

Description: `rt ← memory[base+offset]`

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, zero-extended, 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 the least-significant bit of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddrPSIZE-1..2 || (pAddr1..0 xor (ReverseEndian || 0))
memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte   ← vAddr1..0 xor (BigEndianCPU || 0)
GPR[rt] ← zero_extend(memword15+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Address Error

```
1. 指令格式：
| 100 101 | base(5) | rt(5) | offset(16) |
2.RTF语言：
R[rt] <- mem[base + offset]
```

```
3. 类型
I-Type(访存)
4, 寄存器堆读
raddr1 = R[Instruction[25:21]] = R[base] // 可以认为是 R[rs]
raddr2 = 无
5. ALU
A: R[Instruction[25:21]] = R[base]
B: sign_extend(Instruction[15:0])
ALUop : 有符号数加法
6.内存访问
Address : ALU Result = R[base] + sign_extend(Instruction[15:0])
MemRead : 1
Read需要在内部赋值 Read_strb 这里只能有{1100, 0011}两种之一
MemWrite : 0
Write_Data : 不需要写
Write_Strb : 0000
7.跳转
不需要跳转
8.寄存器堆写:
wen = 1
waddr = R[rt] = R[Instruction[20:16]]
wdata =zero_extend(mem[R[Instruction[25:21]] + sign_extend(Instruction[15:0])])
即 zero_extend(memword[15 + 8*byte,8*byte])
// 需要对取得的 16 -bit 进行零延拓后放入R[rt]
```

36.lwl(加载未对齐字段，左边部分)

Load Word Left **LWL**

31 26	25 21	20 16	15 0
LWL 100010	base	rt	offset
6	5	5	16

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

Purpose:

To load the most-significant part of a word as a signed value from an unaligned memory address

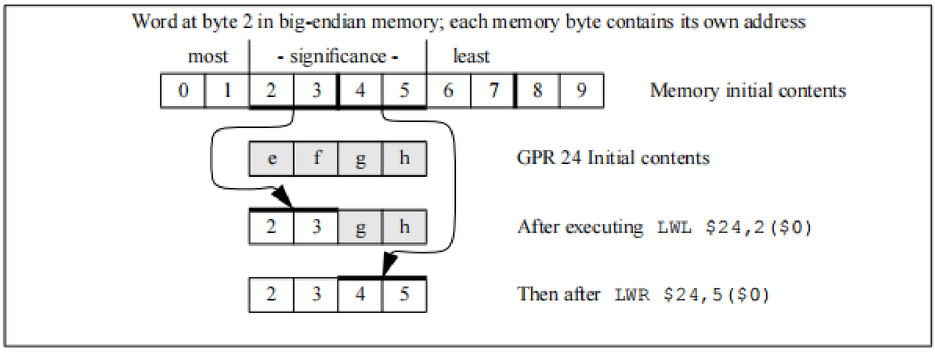
Description: `rt ← rt MERGE memory[base+offset]`

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr)*. *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word *(W)* in memory starting at an arbitrary byte boundary.

The most-significant 1 to 4 bytes of *W* is in the aligned word containing the *EffAddr*. This part of *W* is loaded into the most-significant (left) part of the word in GPR *rt*. The remaining least-significant part of the word in GPR *rt* is unchanged.

The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the most-significant byte at 2. First, LWL loads these 2 bytes into the left part of the destination register word and leaves the right part of the destination word unchanged. Next, the complementary LWR loads the remainder of the unaligned word

Figure 3-2 Unaligned Word Load Using LWL and LWR




```
1. 指令格式:
| 100 010(lwl) | base(5) | rt(5) | offset(16) |
2.RTF语言:
R[rt] <-  mem[base + offset][31:16] || R[rt][15:0]
3. 类型
I-Type(访存)
4, 寄存器堆读
raddr1 = R[Instruction[25:21]] = R[base] // 可以认为是 R[rs]
raddr2 = 无
5. ALU
A: R[Instruction[25:21]] = R[base]
B: sign_extend(Instruction[15:0])
ALUop : 有符号数加法
6.内存访问
Address : ALU Result = R[base] + sign_extend(Instruction[15:0])
MemRead : 1
Read需要在内部赋值 Read_strb = 4'b0011;
MemWrite : 0
Write_Data : 不需要写
Write_Strb : 0000
7.跳转
不需要跳转
8.寄存器堆写:
wen = 1
```

```
waddr = R[rt] = R[Instruction[20:16]]
wdata = (R[Instruction[25:21]] + sign_extend(Instruction[15:0]))[31:16]
设定寄存器堆写有效或者写无效
寄存器堆的写 RF_write_strb = 4'b1100;
```

37.lwr(加载未对齐右半字)

Load Word Right **LWR**

31 26	25 21	20 16	15 0
LWR 100110	base	rt	offset
6	5	5	16

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

Purpose:

To load the least-significant part of a word from an unaligned memory address as a signed value

Description: `rt ← rt MERGE memory[base+offset]`

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word (*W*) in memory starting at an arbitrary byte boundary.

A part of *W*, the least-significant 1 to 4 bytes, is in the aligned word containing *EffAddr*. This part of *W* is loaded into the least-significant (right) part of the word in GPR *rt*. The remaining most-significant part of the word in GPR *rt* is unchanged.

Executing both LWR and LWL, in either order, delivers a sign-extended word value in the destination register.

The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the least-significant byte at 5. First, LWR loads these 2 bytes into the right part of the destination register. Next, the complementary LWL loads the remainder of the unaligned word.

```
1. 指令格式:
| 100 110(lwr) | base(5) | rt(5) | offset(16) |
2.RTF语言:
R[rt] <- R[rt][31:16] || mem[base + offset][15:0]
3. 类型
I-Type(访存)
4, 寄存器堆读
raddr1 = R[Instruction[25:21]] = R[base] // 可以认为是 R[rs]
raddr2 = 无
5. ALU
A: R[Instruction[25:21]] = R[base]
B: sign_extend(Instruction[15:0])
ALUop : 有符号数加法
6.内存访问
Address : ALU Result = R[base] + sign_extend(Instruction[15:0])
MemRead : 1
Read需要在内部赋值 Read_strb = 4'b1100;
MemWrite : 0
Write_Data : 不需要写
Write_Strb : 0000
7.跳转
不需要跳转
8.寄存器堆写:
wen = 1
waddr = R[rt] = R[Instruction[20:16]]
wdata = (R[Instruction[25:21]] + sign_extend(Instruction[15:0]))[15:0]
设定寄存器堆写有效或者写无效
```

```
寄存器堆的写 RF_write_strb = 4'b0011;
```

38.sb(存储字节)

Store Byte **SB**

31 26	25 21	20 16	15 0
SB 101000	base	rt	offset
6	5	5	16

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

Purpose:

To store a byte to memory

Description: `memory[base+offset] ← rt`

The least-significant 8-bit byte of GPR *rt* is stored in memory at the location specified by the effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

Operation:

```
vAddr      ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr      ← pAddr_{PSIZE-1..2} || (pAddr_{1..0} xor ReverseEndian^2)
bytesel    ← vAddr_{1..0} xor BigEndianCPU^2
dataword   ← GPR[rt]_{31-8*bytesel..0} || 0^{8*bytesel}
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error

```
1. 指令格式:
| 101 000 | base(5) | rt(5) | offset(16) |
2.RTF语言:
mem[base + offset] <- R[rt]
3. 类型
I-Type(访存)
4, 寄存器堆读
raddr1 = R[Instruction[25:21]] = R[base] // 可以认为是 R[rs]
raddr2 = 无
5. ALU
A: R[Instruction[25:21]] = R[base]
B: sign_extend(Instruction[15:0])
ALUop : 有符号数加法
6.内存访问
Address : ALU Result = R[base] + sign_extend(Instruction[15:0])
MemRead : 0
MemWrite : 1
需要写内存
Write_Data : R[rt] = R[Instruction[20:16]]
Write_Strb : 1000(大端序) 0001 (小端序)
7.跳转
不需要跳转
8.寄存器堆写:
不需要写寄存器堆
```

39.sh(存储半字)

Store Halfword **SH**

31–26	25–21	20–16	15–0
SH 101001	base	rt	offset
6	5	5	16

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

Purpose:

To store a halfword to memory

Description: `memory[base+offset] ← rt`

The least-significant 16-bit halfword of register *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 the least-significant bit of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr_0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_PSIZE-1..2 || (pAddr1_1..0 xor (ReverseEndian || 0))
bytesel← vAddr1_1..0 xor (BigEndianCPU || 0)
dataword← GPR[rt]_31-8*bytesel..0 || 0^8*bytesel
StoreMemory (CCA, HALFWORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error

```
1. 指令格式:
| 101 001 | base(5) | rt(5) | offset(16) |
2.RTF语言:
mem[base + offset] <- R[rt]
3. 类型
I-Type(访存)
4, 寄存器堆读
raddr1 = R[Instruction[25:21]] = R[base] // 可以认为是 R[rs]
raddr2 = 无
5. ALU
A: R[Instruction[25:21]] = R[base]
B: sign_extend(Instruction[15:0])
ALUop : 有符号数加法
6.内存访问
Address : ALU Result = R[base] + sign_extend(Instruction[15:0])
MemRead : 0
MemWrite : 1
需要写内存
Write_Data : R[rt] = R[Instruction[20:16]]
Write_Strb : 1100(大端序) 0011（小端序）
7.跳转
不需要跳转
8.寄存器堆写:
不需要写寄存器堆
```

40.sw(存储字)

Store Word **SW**

31 26	25 21	20 16	15 0
SW 101011	base	rt	offset
6	5	5	16

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

Purpose:

To store a word to memory

Description: `memory[base+offset] ← rt`

The least-significant 32-bit word of register *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

```
1. 指令格式：
| 101 011 | base(5) | rt(5) | offset(16) |
2.RTF语言：
mem[base + offset] <- R[rt]
3. 类型
I-Type(访存)
4，寄存器堆读
raddr1 = R[Instruction[25:21]] = R[base] // 可以认为是 R[rs]
raddr2 = 无
5. ALU
A: R[Instruction[25:21]] = R[base]
B: sign_extend(Instruction[15:0])
ALUop : 有符号数加法
6.内存访问
Address : ALU Result = R[base] + sign_extend(Instruction[15:0])
MemRead : 0
MemWrite : 1
需要写内存
Write_Data : R[rt] = R[Instruction[20:16]]
Write_Strb : 1111
7.跳转
不需要跳转
8.寄存器堆写：
不需要写寄存器堆
```

41.swl(存储左边半字)

Store Word Left **SWL**

31 ... 26	25 ... 21	20 ... 16	15 ... 0
SWL 101010	base	rt	offset
6	5	5	16

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

Purpose:

To store the most-significant part of a word to an unaligned memory address

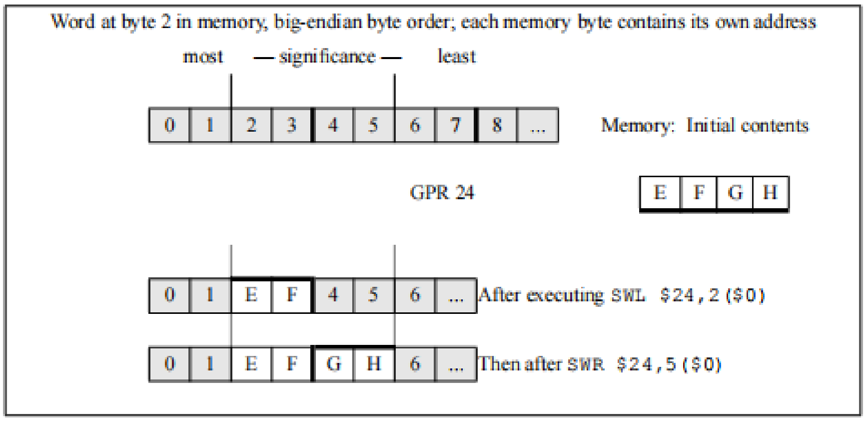
Description: `memory[base+offset] ← rt`

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr)*. *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word *(W)* in memory starting at an arbitrary byte boundary.

A part of *W*, the most-significant 1 to 4 bytes, is in the aligned word containing *EffAddr*. The same number of the most-significant (left) bytes from the word in GPR *rt* are stored into these bytes of *W*.

The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is located in the aligned word containing the most-significant byte at 2. First, SWL stores the most-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWR stores the remainder of the unaligned word.

Figure 3-6 Unaligned Word Store Using SWL and SWR



The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address (*vAddr1..0*)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte ordering.

```
1. 指令格式:
| 101 010(lwl) | base(5) | rt(5) | offset(16) |
2.RTF语言:
mem[base + offset] <- R[rt]
3. 类型
I-Type(访存)
4, 寄存器堆读
raddr1 = R[Instruction[25:21]] = R[base] // 可以认为是 R[rs]
raddr2 = 无
5. ALU
A: R[Instruction[25:21]] = R[base]
B: sign_extend(Instruction[15:0])
ALUop : 有符号数加法
6.内存访问
Address : ALU Result = R[base] + sign_extend(Instruction[15:0])
MemRead : 0
```

```
MemWrite : 1
Write_Data : R[rt][31:16]
Write_Strb : 0011
7.跳转
不需要跳转
8.寄存器堆写：
不需要写寄存器堆
```

42.swr（存储右边半字）

Store Word Right **SWR**

31–26	25–21	20–16	15–0
SWR 101110	base	rt	offset
6	5	5	16

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

Purpose:

To store the least-significant part of a word to an unaligned memory address

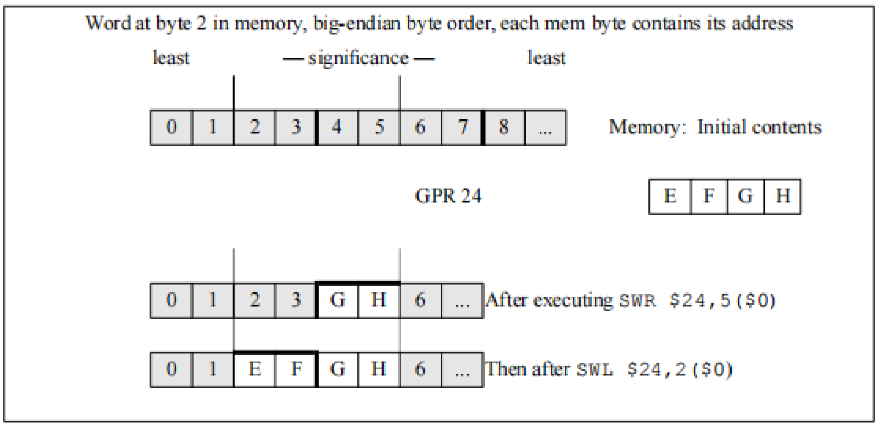
Description: `memory[base+offset] ← rt`

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr)*. *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word *(W)* in memory starting at an arbitrary byte boundary.

A part of *W*, the least-significant 1 to 4 bytes, is in the aligned word containing *EffAddr*. The same number of the least-significant (right) bytes from the word in GPR *rt* are stored into these bytes of *W*.

The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is contained in the aligned word containing the least-significant byte at 5. First, SWR stores the least-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWL stores the remainder of the unaligned word.

Figure 3-8 Unaligned Word Store Using SWR and SWL




```
1. 指令格式：
| 101 110(swr) | base(5) | rt(5) | offset(16) |
2.RTF语言：
mem[base + offset] <- R[rt]
3. 类型
I-Type(访存)
4，寄存器堆读
raddr1 = R[Instruction[25:21]] = R[base] // 可以认为是 R[rs]
raddr2 = 无
5. ALU
A: R[Instruction[25:21]] = R[base]
B: sign_extend(Instruction[15:0])
```

```
ALUop : 有符号数加法
6.内存访问
Address : ALU Result = R[base] + sign_extend(Instruction[15:0])
MemRead : 0
MemWrite : 1
Write_Data : R[rt][15:00]
Write_Strb : 1100
7.跳转
不需要跳转
8.寄存器堆写:
不需要写寄存器堆
```

立即数

43.movn(非零时移动)

Move Conditional on Not Zero **MOVN**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	MOVN 001011
6	5	5	5	5	6

Format: `MOVN rd, rs, rt` **MIPS32 (MIPS IV)**

Purpose:

To conditionally move a GPR after testing a GPR value

Description: `if rt ≠ 0 then rd ← rs`

If the value in GPR *rt* is not equal to zero, then the contents of GPR *rs* are placed into GPR *rd*.

Restrictions:

None

Operation:

```
if GPR[rt] ≠ 0 then
    GPR[rd] ← GPR[rs]
endif
```

Exceptions:

None

Programming Notes:

The non-zero value tested here is the *condition true* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.

```
1. 指令格式:
| 000 000 | rs(5) | rt(5) | rd(5) | 00 000 | 001 011(movn)|
2.RTF语言:
if R[rt] != 0 then R[rd] <- R[rs]
3. 类型
R-Type
4, 寄存器堆读
    raddr1 = R[Instruction[25:21]] = R[rs]
    raddr2 = R[Instruction[20:16]] = R[rt]
5. ALU
    A: R[Instruction[20:16]] = R[rt]
    B: 0
ALUop : 有符号数减法
6.内存访问
```

```
不需要内存访问
7.跳转
不需要跳转
8.寄存器堆写:
    if(Zero == 0)  then wen = 1
    waddr = rd = Instruction[15:11]
    wdata = R[rs] = R[Instruction[25:21]]
```

44.movz（零时移动）

Move Conditional on Zero **MOVZ**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	MOVZ 001010
6	5	5	5	5	6

Format: `MOVZ rd, rs, rt` **MIPS32 (MIPS IV**

Purpose:

To conditionally move a GPR after testing a GPR value

Description: `if rt = 0 then rd ← rs`

If the value in GPR *rt* is equal to zero, then the contents of GPR *rs* are placed into GPR *rd*.

Restrictions:

None

Operation:

```
if GPR[rt] = 0 then
    GPR[rd] ← GPR[rs]
endif
```

Exceptions:

None

Programming Notes:

The zero value tested here is the *condition false* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.

```
1. 指令格式:
| 000 000 | rs(5) | rt(5) | rd(5) | 00 000 | 001 010(movz)|
2.RTF语言:
if R[rt] == 0 then R[rd] <- R[rs]
3. 类型
R-Type
4, 寄存器堆读
    raddr1 = R[Instruction[25:21]] = R[rs]
    raddr2 = R[Instruction[20:16]] = R[rt]
5. ALU
    A: R[Instruction[20:16]] = R[rt]
    B: 0
ALUop : 有符号数减法
6.内存访问
不需要内存访问
7.跳转
不需要跳转
8.寄存器堆写:
    if(Zero == 1)  then wen = 1
    waddr = rd = Instruction[15:11]
    wdata = R[rs] = R[Instruction[25:21]]
```

45.lui(存储无符号半字)

Load Upper Immediate **LUI**

31–26	25–21	20–16	15–0
LUI 001111	0 00000	rt	immediate
6	5	5	16

Format: `LUI rt, immediate` **MIPS32 (MIPS I)**

Purpose:

To load a constant into the upper half of a word

Description: `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

```
1. 指令格式:
| 001 111(lhu) | 000 00 | rt(5) | immediate(16) |
2.RTF语言:
R[rt] <- immediate || {16{0}}
3. 类型
I-Type(运算)
4, 寄存器堆读
不需要读
5. ALU
不需要ALU
6.内存访问
不需要内存访问
7.跳转
不需要跳转
8.寄存器堆写:
wen = 1
waddr = rt = Instruction[20:16]
wdata = Instruction[15 : 0] || {16{0}}
```