RISCV_指令译码表建表格

阅读37条RISCV指令

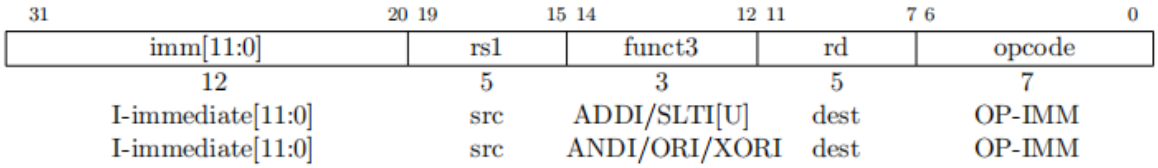
LUI、AUIPC、JAL、JALR、BEQ、BNE、BLT、BGE、BLTU、BGEU、LB、LH、LW、LBU、LHU、SB、SH、SW、ADDI、SLTI、SLTI、SLTIU、XORI、ORI、ANDI、SLLI、SRLI、SRAI、ADD、SUB、SLL、SLT、SLTU、XOR、SRL、SRA、OR、AND

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、I-type、S-type、B-type、U-type、J-type
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 是否需要写寄存器堆，waddr写地址是什么，wdata写数据是什么?
```

完成37条指令填充

1-6 ADDI / SLTI / SLTIU / ANDI / ORI / XORI



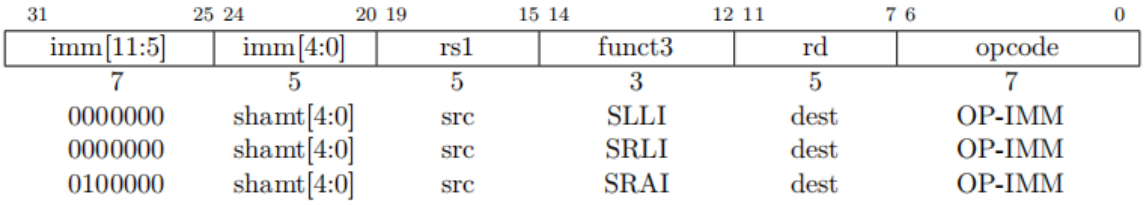
31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

```
ADDI 将 有符号扩展后的 12 位立即数，与 寄存器 rs1 中的内容相加。 算术运算的溢出将被忽略，结果只是最终计算结果的最低 XLEN 位。 ADDI rd, rs1, 0 被用来实现 MV rd, rs1 的汇编器伪指令。
SLTI(set less than immediate) 将 rd 寄存器中的值 赋值为 1 ，如果寄存器 rs1 中的数小于 符号位扩展后的 立即数，这两者都被看作为有符号数， 否则 rd 寄存器中 赋值为 0.
SLTIU 是类似的操作，但是将这两个数视为无符号数(i.e., 立即数首先 有符号扩展为 XLEN bits， 然后将其视作无符号数).注意到 SLTIU rd,rs1,1 将 rd 设为 1，如果 rs1 等于 0， 否则，将 rd 设为 0(汇编伪指令 SEQZ rd, rs).
ANDI,ORI,XORI 是逻辑运算，对于寄存器 rs1 中的数据和 符号位扩展后的 12 位立即数进行按位的 AND,OR,XOR，并且将结果放在 rd 寄存器中。注意到, XORI rd, rs1, -1 将 寄存器 rs1 中的数进行逻辑的翻转操作(汇编伪指令 NOT rd,rs).
```

```
1.指令格式:
Instruction[31 : 20] = immediate;
Instruction[19 : 15] = rs1;
Instruction[14 : 12] = func;
Instruction[11 : 7] = rd;
Instruction[6 : 0] = Opcode;
ADDI:   | immediate(12) |  rs1(5)| func3(000) | rd(5) | Opcode7(0010011) |
SLTI:   | immediate(12) |  rs1(5)| func3(010) | rd(5) | Opcode7(0010011) |
SLTIU:  | immediate(12) |  rs1(5)| func3(011) | rd(5) | Opcode7(0010011) |
ANDI:   | immediate(12) |  rs1(5)| func3(111) | rd(5) | Opcode7(0010011) |
```

```
ORI:    | immediate(12) |  rs1(5)| func3(110) | rd(5) | Opcode7(0010011) |
XORI:   | immediate(12) |  rs1(5)| func3(100) | rd(5) | Opcode7(0010011) |
2. RTF 语言:
ADDI :  R[rd] <- R[rs1] + Sign_extend(immediate)
SLTI:   R[rd] <- bool(R[rs1] < Sign_extend(immediate)) // 有符号数
SLTIU:  R[rd] <- bool(R[rs1] < Sign_extend(immediate)) // 有符号扩展的无符号数
ANDI :  R[rd] <- R[rs1] & Sign_extend(immediate)
ORI :   R[rd] <- R[rs1] | Sign_extend(immediate)
XORI:   R[rd] <- R[rs1] ^ Sign_extend(immediate)
3. 类型
I-Type
4. 寄存器堆读
raddr1 = rs1(Instruction[19 : 15])  raddr2(无)
5. ALU
A: Read_data1 = R[rs1] = R[Instruction[19 : 15]]
B: Sign_extend(immediate) Sign_extend(Instruction[31 : 20])
ALUop:
ADDI : 加法
SLTI : 有符号数的 slt 比较
SLTIU: 无符号数的 slt 比较
ANDI : 按位与
ORI :  按位或
XORI : 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  需要写寄存器堆
waddr = rd = Instruction[11 : 7]
wdata = R[Instruction[19 : 15]] Op Sign_extend(Instruction[31 : 20])
```

7-9 SLLI \ SRLI \ SRAI



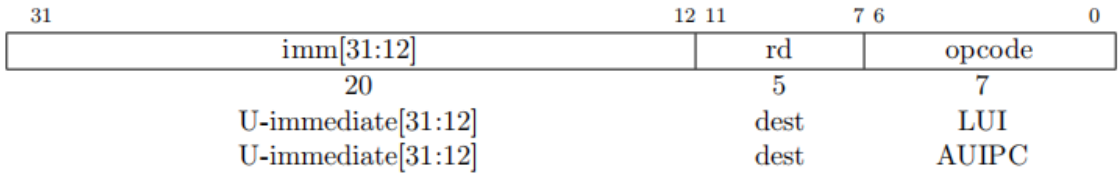
31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

将数据 移位 一个常量的大小被编码为特殊的 I-type 类型。需要被移位的操作数 位于 rs1 中，移位的总量位于 I_immediate 的 低 5 位。 相应的移位类型被编码在 I_immediate 的高位。SLLI 是逻辑左移（ 0 被移动到数据的低位); SRLI 是逻辑右移 (0 被移动到高位);并且 SRAI 是一个算术右移(原始的符号位被赋值并添加到高位)

```
1. 指令格式
Instruction[31 : 25] = 移位类型的操作数，或者写成 Immediate[11 : 5]
Instruction[24 : 20] = 移位总量， 或者写成 Immediate[4 : 0]
Instruction[19 : 15] = rs1;
Instruction[14 : 12] = func;
Instruction[11 : 7] = rd;
Instruction[6 : 0] = Opcode;
SLLI: | 0000000(7) | shamt(5)| rs1(5)| 001(func3) | rd(5) | 0010011(7)
SRLI: | 0000000(7) | shamt(5)| rs1(5)| 101(func3) | rd(5) | 0010011(7)
SRAI: | 0100000(7) | shamt(5)| rs1(5)| 101(func3) | rd(5) | 0010011(7)
```

```
2. RTF 语言
SLLI : R[rd] <- R[rs1] << shamt(Instruction[24 : 20])
SRLI : R[rd] <- R[rs1] >>(u) shamt(Instruction[24 : 20]) // 补 0
SRAI : R[rd] <- R[rs1] >>(s) shamt(Instruction[24 : 20]) // 补符号位
3. 类型
I-Type
4. 寄存器堆读
raddr1 = R[rs1] = R[Instruction[19 : 15]]
raddr2 (无)
5. ALU(不使用ALU)、使用 shifter
A: R[rs1] = R[Instruction[19 : 15]]
B: shamt = Instruction[24 : 20]
ShifterOp:
SLLI : 逻辑左移
SRLI: 逻辑右移
SRAI : 算术右移
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[11 : 7]];
wdata = R[Instruction[19 : 15]] << / >>(u/s) Instruction[24 : 20]
```

10-11 LUI / AUIPC




```
LUI(load upper immediate) 被用来构造 32-位的常量，并且使用 U-type 类型。LUI 将
U_immediate 的值放入 目标寄存器 rd 的高 20 位，将低12 位补 0
AUIPC(add upper immediate to PC) 被用来构造 PC-相关的地址，并且使用 U-type 的格式。
AUIPC 从 20-bit 的 U_immediate 构造出一个 32-bit 的偏移，将低12 位补充 0，并且将这个偏移
加到 PC 上，然后将值放入 rd 寄存器
```

```
1.指令格式
Instruction[31 : 12] = U_immediate
Instruction[11 : 7] = rd
Instruction[6 : 0] = opcode
LUI:        | immediate(20) | rd(5) | 0110111(7)|
AUIPC:      | immediate(20) | rd(5) | 0010111(7)|
2. RTF 语言
LUI:   R[rd] <- (U_immediate[31 : 12] << 12)
AUIPC: R[rd] <- PC + (U_immediate[31 : 12] << 12)
3. 类型
U-type
4. 寄存器堆读
不需要读寄存器堆
5. ALU:
LUI: 不需要 ALU
AUIPC : A: PC
```

```
B: (U_immediate[31 : 12] << 12) = {Instruction[31 : 12] , {12{1'b0}}}
ALUop: 加法
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[11 : 7]]
LUI : wdata = (U_immediate[31 : 12] << 12) = (Instruction[31 : 12] << 12)
AUIPC:wdata =PC + (U_immediate[31 : 12] << 12) = PC + (Instruction[31 : 12] <<
12)
```

12-21 ADD / SLT / SLTU / AND / OR / XOR / SLL / SRL / SUB / SRA

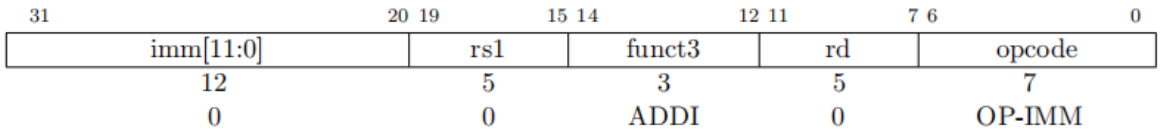
31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

ADD 和 SUB 分别执行加减运算。忽略溢出，并且低 XLEN 位被写到目标寄存器。 SLT 和 SLTU 将分别进行 有符号和无符号的比较， 如果 rs1 < rs2，则 将 1 写入到 rd 中,否则 写 0. 注意到,SLTU rd,x0,rs2 将 rd 设为 1，如果 rs2 不等于0，否则将 rd 设为 0(汇编伪指令 SNEZ rd,rs).
AND,OR,和 XOR 做按位逻辑运算
SLL,SRL和SRA 分别做逻辑左移，逻辑右移，算术右移，对于寄存器 rs1 的值 通过寄存器 rs2 的低5 位进行移位操作。

```
1.指令格式
Instruction[31 : 25] = func7;
Instruction[24 : 20] = rs2;
Instruction[19 : 15] = rs1;
Instruction[14 : 12] = func3;
Instruction[11 : 7] = rd;
Instruction[6 : 0] = opcode;
ADD : | 0000000(7) | rs2(5) | rs1(5) | 000(3) | rd(5) | 0110011(7) |
SLT:  | 0000000(7) | rs2(5) | rs1(5) | 010(3) | rd(5) | 0110011(7) |
SLTU: | 0000000(7) | rs2(5) | rs1(5) | 011(3) | rd(5) | 0110011(7) |
AND:  | 0000000(7) | rs2(5) | rs1(5) | 111(3) | rd(5) | 0110011(7) |
OR:   | 0000000(7) | rs2(5) | rs1(5) | 110(3) | rd(5) | 0110011(7) |
XOR:  | 0000000(7) | rs2(5) | rs1(5) | 100(3) | rd(5) | 0110011(7) |
SLL:  | 0000000(7) | rs2(5) | rs1(5) | 001(3) | rd(5) | 0110011(7) |
SRL:  | 0000000(7) | rs2(5) | rs1(5) | 101(3) | rd(5) | 0110011(7) |
SUB:  | 0100000(7) | rs2(5) | rs1(5) | 000(3) | rd(5) | 0110011(7) |
SRA:  | 0100000(7) | rs2(5) | rs1(5) | 101(3) | rd(5) | 0110011(7) |
2. RTF 语言
ADD : R[rd] <- R[rs1] + R[rs2]
SLT : R[rd] <- bool( R[rs1] < (s) R[rs2])
SLTU: R[rd] <- bool( R[rs1] < (u) R[rs2])
AND:  R[rd] <- R[rs1] & R[rs2]
OR:   R[rd] <- R[rs1] | R[rs2]
XOR:  R[rd] <- R[rs1] ^ R[rs2]
SLL:  R[rd] <- R[rs1] << R[rs2][5 : 0]
SRL : R[rd] <- R[rs1] >>(u) R[rs2][5 : 0]
```

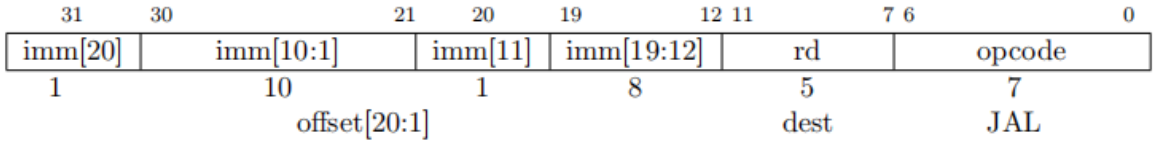
```
SUB : R[rd] <- R[rs1] - R[rs2]
SRA : R[rd] <- R[rs1] >> (s) R[rs2][5 : 0]
3.类型
R-Type
4.寄存器堆读
raddr1 = R[rs1] = R[Instruction[19 : 15]]
raddr2 = R[rs2] = R[Instruction[24 : 20]]
5.ALU:
A : R[Instruction[19 : 15]]
B : R[Instruction[24 : 20]]
ALUop: 加法 / 有符号slt / 无符号 slt / 按位与 / 按位或
按位异或 / 逻辑左移 / 逻辑右移 / 减法 / 算术右移
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[11 : 7]]
wdata = R[rs1] op R[rs2]
```

21.5 NOP指令




```
Nop指令不需要进行任何 可见的使用状态的变化，只需要 PC + 4. NOP 编码为 ADDI x0,x0,0.
| immediate(12'b0) | rs1(5'b0) | func3(000) | rd(5'b0) | opcode(7) 0010011 |
```

22.JAL 指令

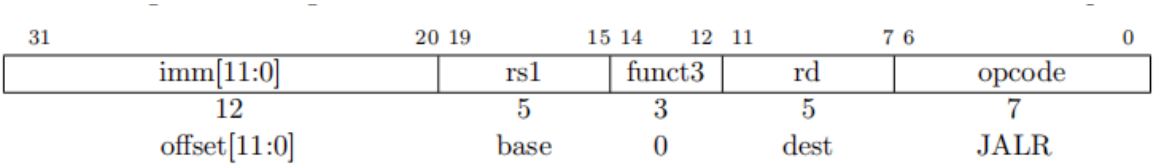



```
jump and link(JAL) 跳转并链接
指令使用了 J-Type 类型的格式，J-immediate 以两个字节的倍数编码一个有符号的偏移量。 偏移量被符号位扩展之后，添加到 PC 来完成 jump跳转的目标地址。Jumps 可以完成 +_1 MiB 范围内的地址跳转。
JAL 存储jump 指令 (pc + 4) 之后的地址到寄存器 rd 中。 标准的软件调用约定使用 x1 作为返回地址的寄存器， x5 作为备用的链接寄存器.
无条件跳转(汇编伪指令 J) 被编码为 一个 JAL 且 rd = x0
```

```
1. 指令格式
| imm[20] | imm[10 : 1] | imm[11] | imm[19 : 12] | rd(5) | opcode(7) 1101111
| imm[20]imm[10 : 1]imm[11]imm[19 : 12] (20) | rd(5) | 1101111 |
2. RTF 语言
R[rd] = PC + 4
PC = PC + Sign_extend(immediate)
3.类型
J-Type
4.寄存器堆读
```

不需要读寄存器堆
5.ALU
不需要ALU
6.内存访问:
无内存访问
7.跳转:
需要跳转，跳转地址是 PC + Sign_extend(immediate) = PC + Sign_extend({12{Instruction[31]},Instruction[31],Instruction[19 : 12],Instruction[20],Instruction[30 : 21]});
8.寄存器堆写:
wen = 1 需要写 rd
waddr = R[rd] = R[Instruction[11 : 7]];
wdata = PC + 4

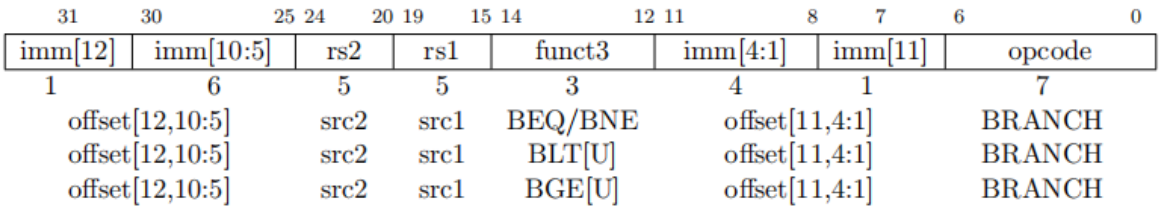
23.JALR 指令



非直接跳转指令JALR(jump and link register)使用 I-type 编码。目标地址根据符号位扩展后的 12-bit I-immediate 加上寄存器堆 rs1 中的数据得到，设置结果的最低有效位为 0.将原来的 PC + 4 的地址写入寄存器rd。.如果不需要该结果，则可以将寄存器x0用作目的地。

1. 指令格式
| immediate(12) | rs1(5) | func(3) 000 | rd(5) | opcode(7) 1100111
2.RTF 语言
R[rd] <- PC + 4   PC = (R[rs1] + Sign_extend(immediate)) & {31'b1,1'b0}.
3.类型
I-Type
4.寄存器堆读
raddr1 = R[rs1] = R[Instruction[19 : 15]].
raddr2 (无)
5.ALU
不需要ALU
6.内存访问
无内存访问
7.跳转:
需要跳转，跳转地址是:
PC = (R[Instruction[19 : 15]] + Sign_extend(Instruction[31 : 20])) ^{31'b1,1'b0}.
8.寄存器堆写:
wen = 1 需要写 rd
waddr = R[rd] = R[Instruction[11 : 7]].
wdata = PC(原来的，不是经过加法运算之后的) + 4

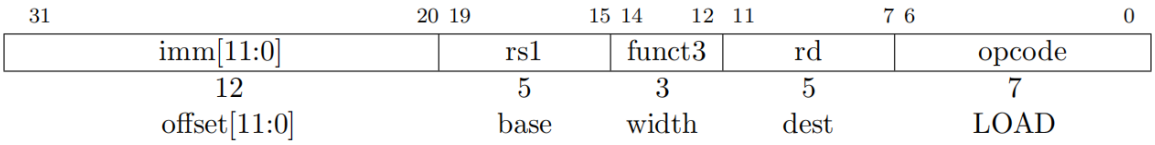
24-29 BEQ / BNE / BLT / BLTU / BGE / BGEU



Branch 指令比较两个寄存器之间的内容. BEQ 和 BNE 选择分支跳转，如果两个寄存器 rs1 和 rs2 中的内容相等或者不相等. BLT 和 BLTU 选择分支跳转，如果 rs1 中的内容小于 rs2 中的内容，使用有符号的比较和无符号的比较。 BGE 和 BGEU 采用 branch 指令，如果寄存器 rs1 的内容 大于等于 rs2 中的内容，分别采用有符号和无符号的比较。注意到，BGT,BGTU,BLE,BLEU 可以被综合使用，通过将 BLT, BLTU,BGE,BGEU 运用取反操作得到。

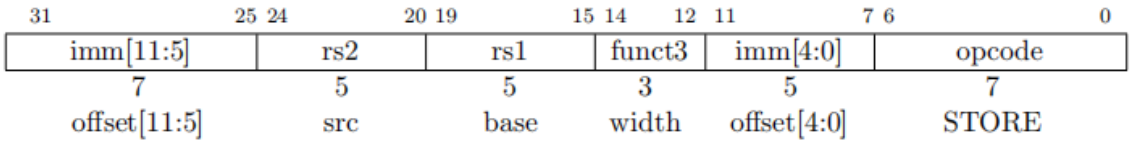
1.指令格式
| imm[12]|imm[10 : 5]| rs2(5) | rs1(5) | func3(3) | imm[4 : 1]|
imm[11]|Opcode(7)|
BEQ: |imm[12]imm[10:5] | rs2(5) | rs1(5)| 000 | imm[4 : 1]imm[11] | 1100011
BNE: |imm[12]imm[10:5] | rs2(5) | rs1(5)| 001 | imm[4 : 1]imm[11] | 1100011
BLT: |imm[12]imm[10:5] | rs2(5) | rs1(5)| 100 | imm[4 : 1]imm[11] | 1100011
BLTU:|imm[12]imm[10:5] | rs2(5) | rs1(5)| 110 | imm[4 : 1]imm[11] | 1100011
BGE: |imm[12]imm[10:5] | rs2(5) | rs1(5)| 101 | imm[4 : 1]imm[11] | 1100011
BGEU:|imm[12]imm[10:5] | rs2(5) | rs1(5)| 111 | imm[4 : 1]imm[11] | 1100011
2. RTF 语言
BEQ: R[rs1]==R[rs2] --> PC = PC + Sign_extend(imm[12]imm[11]imm[10:5]imm[4:1])
BNE: R[rs1]!=R[rs2] --> PC = PC + Sign_extend(immediate)
BLT: R[rs1] <(s)R[rs2] --> PC = PC + Sign_extend(immediate)
BLTU: R[rs1] <(u)R[rs2] --> PC = PC + Sign_extend(immediate)
BGE: R[rs1] >=(s)R[rs2] --> PC = PC + Sign_extend(immediate)
BGEU: R[rs1] >=(u)R[rs2] --> PC = PC + Sign_extend(immediate)
3. 类型
B-Type
4. 寄存器堆读
raddr1 = R[rs1] = R[Instruction[19 : 15]]
raddr2 = R[rs2] = R[Instruction[24 : 20]]
5. ALU
A : R[Instruction[19 : 15]]
B : R[Instruction[24 : 20]]
ALUop:
BEQ : 减法 Zero 标志位 == 1
BNE: 减法 Zero 标志位 == 0
BLT: 减法 运算的符号位Result[31] == 1 , Overflow == 0 或 Result[31] == 0, Overflow == 1 即 Result[31] ^ Overflow == 1
且 Zero != 1
BLTU: 减法 运算的符号位 Result[31] == 1 且 Zero != 1
BGE: 减法 运算的符号位 Result[31] == 0, Overflow == 0 或 Result[31] == 1, Overflow == 1 即 Result[31] ^ Overflow == 0
BGEU: 减法 Result[31] == 0
6.内存访问
无内存访问
7.跳转
需要跳转，判断条件成立时跳转
跳转地址为: PC = PC + Sign_extend(immediate)
8.寄存器堆写:
不需要写寄存器堆

30-34 lb / lh / lw / lbu / lhu



```
1.指令格式
Instruction[31 : 20] = offset[11 : 0];
Instruction[19 : 15] = rs1;
Instruction[14 : 12] = func3;
Instruction[11 : 7] = rd;
Instruction[6 : 0] = opcode;
LB: | offset(12) | rs1(5) | func3(3) 000 | rd(5) | opcode(7) 0000011
LH: | offset(12) | rs1(5) | func3(3) 001 | rd(5) | opcode(7) 0000011
LW: | offset(12) | rs1(5) | func3(3) 010 | rd(5) | opcode(7) 0000011
LBU:| offset(12) | rs1(5) | func3(3) 100 | rd(5) | opcode(7) 0000011
LHU:| offset(12) | rs1(5) | func3(3) 101 | rd(5) | opcode(7) 0000011
2.RTF 语言:
LB : R[rd] <- Sign_extend{Mem[R[rs1] + Sign_extend(offset)][7 : 0]}
LH:  R[rd] <- Sign_extend{Mem[R[rs1]] + Sign_extend(offset)] [15 : 0]}
LW:  R[rd] <- {Mem[R[rs1]] + Sign_extend(offset)] [31 : 0]}
LBU: R[rd] <- Zero_extend{Mem[R[rs1] + Sign_extend(offset)][7 : 0]}
LHU: R[rd] <- Zero_extend{Mem[R[rs1] + Sign_extend(offset)][15 : 0]}
3.类型:
I-Type
4.寄存器堆读
raddr1 = R(Instruction[19 : 15])  raddr2(无)
5.ALU
不需要ALU
6.内存访问
Address : R[rs1] + Sign_extend(offset)
MemRead : 1
Read需要在内部赋值:
LB: 0001
LH: 0011
LW: 1111
LBU: 0001
LHU: 0011
不需要写内存
7.跳转
不需要跳转
8.寄存器堆写:
wen = 1
waddr = R[rd] = R[Instruction[11 : 7]]
wdata = Sign_extend(Mem[R[rs1] + Sign_extend(offset)){Read_data}
```

35-37 SB / SH / SW




```
1.指令格式:
Instruction[31 : 25] = offset[11 : 5]
Instruction[24 : 20] = rs2;
Instruction[19 : 15] = rs1;
Instruction[14 : 12] = func3;
Instruction[11 : 7] = offset[4 : 0];
Instruction[6 : 0] = opcode;
SB: | offset[11:5]| rs2(5)| rs1(5)| func(000) | offset[4 : 0] | Opcode(7)
0100011|
```

SH: | offset[11:5]| rs2(5)| rs1(5)| func(001) | offset[4 : 0] | Opcode(7)
0100011|
SW: | offset[11:5]| rs2(5)| rs1(5)| func(010) | offset[4 : 0] | Opcode(7)
0100011|
2.RTF 语言:
SB: Mem[R[rs1] + Sign_extend(offset)] <- R[rs2][7 : 0]
SH: Mem[R[rs1] + Sign_extend(offset)] <- R[rs2][15 : 0]
SW: Mem[R[rs1] + Sign_extend(offset)] <- R[rs2][31 : 0]
3.类型
S-Type
4.寄存器堆读
raddr1 = R[Instruction[19 : 15]] = R[rs1];
raddr2 = R[Instruction[24 : 20]] = R[rs2];
5.ALU
不需要ALU
6.内存访问
Address : R[rs1] + Sign_extend(offset)
MemWrite : 1
需要写内存
Write_Data: R[rs2] = R[Instruction[24 : 20]]
Write_Strb :
SB: 0001
SH: 0011
SW: 1111
7.跳转
不需要跳转
8.寄存器堆写:
不需要写寄存器堆