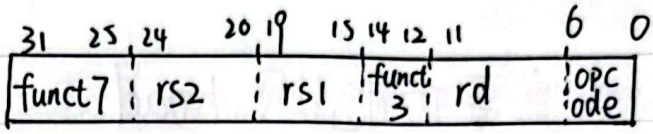
No.

Date . .

CA Review: RISC-V

Assembly Instructions: 4个核心 type (RISU) + B/J
不同type有不同format，但 rs1, rs2, rd 在相同位置

R-type: Register-register operation
接受 rs1, rs2，输出至 rd (S: source; d: destination)
不可访问 main memory
$a = b \pm c$



$a \Leftrightarrow rd \quad b \Leftrightarrow rs1 \quad c \Leftrightarrow rs2$
还可以是位运算：AND/OR/XOR；还可比较：SLT/SLTU：~ rd, rs1, rs2
rd=1 if rs1 < rs2 else 0；且把 number 当作 signed/unsigned with slt/sltu
(SLT(U): store if less than, U: unsigned)

slt(u) 可各自检测 加减法在 signed/unsigned 数域内是否 overflow:
unsigned: 如 add x_2, x_1, x_1 若 $x_3=0$, $x_2<x_1$, 则溢;
sltu x_3, x_2, x_1 否则 $x_2=x_1+x_1>x_1$

signed:
add t_0, t_1, t_2 $t_3=1$ if $t_2<0$ $t_0=t_1+t_2$.
slti $t_3, t_2, 0$ $t_4=1$ if $t_0<t_1$
slt t_4, t_0, t_1 若 $t_2<0$, $t_0>t_1$ 矛盾? 溢出!
bne t_3, t_4, overflow 若 $t_2>0$, $t_0<t_1$. 矛盾? 溢出!

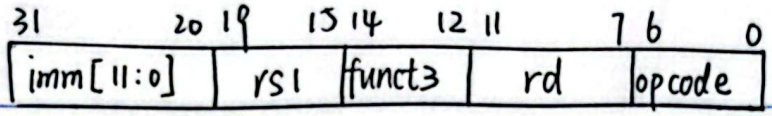
还可是 shift 操作: shift left/right (arithmetic) sll/srl/sra rd, rs1, rs2
rs1 左/右移，多少位呢? rs2 中低5位所代表的十进制数 ($2^3=32$)
而 arithmetic: sign-extended ※指的是右移后前面空出的位
用1/0填充，rs1原为负(符号位为1)则填1; 反之填0
而: srl 总用0填充; sll, 空出的低5位必用0填

※: RISC-V中，所有Imm均是sign-extended

KOKUYO

No.

Date . .



I-type：有两个operands：一个从reg中拿(rs1)，一个是立即数(sign-extended)

常用命令：addi x1, x0, 10 / -10 (有符号)

slli/srli/srai 按imm低5位十进制移；如何从机器码分辨是srai？

用imm高7位中1位 (imm[11:0]). (or funct7 field).

andi/ori/xori/slti/sltui 都是与signed-imm操作：

slti: rs1中数→有符号 ↘ sltui: rs1中数→unsigned.

还有一类重要功能：Load

lw: lw rd, imm(rs1) : load word at addr. to rd.

而 addr.: rs1中数 + imm 注意 word是4byte 32位

附：Little Endian: ADDR: 3 2 1 0 (Big Endian反之)

Byte value: 3 2 1 0

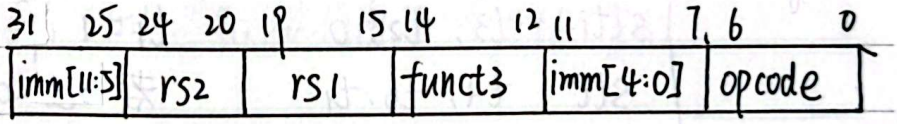
lb/lbu : load signed/unsigned byte at addr. to rd.

☆：拿一个byte来，用1/0填至32位(填高24位)；8位中sign-bit为1，则1而lbu总用0填充

类似地：lh/lhu : halfword: 2 bytes

而能load，就能store：也就是 S-type：

S-type:



sw rs2, imm(rs1) : store word (32bit) at rs2 to addr.

addr. = (number in rs1) + imm.

由于rs2中数本身32bit，故也就无"填充之说"；而sh, sb呢？

无shu, sbu !! 直接取rs2中数低16/8位，就sh/sb了。

Recap addi x11, x0, 0x4F6 ⇒ 0x85F6.

Exercise. sw x11, 0(x5) (假设已超过分配指令)

lb x12, 1(x5)

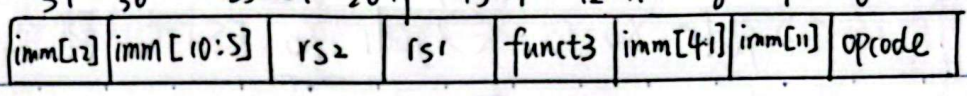
Campus x12中：0x4. ↙0x FFFFFF85.

(lb: signed ⇒ padding)

No.

Date

B-type(又称SB)



conditional branch (unconditional: J-type).

bne/beq rs1, rs2, L (imm/Label). going to L if rs1 ≠/= rs2

blt/bltu/bge/bgeu rs1, rs2, L

< ≥ → 比较时：视为unsigned (无符号).

△ Otherwise: go to next statement

当前命令地址

L若是Label，则跳至label；若是imm，则跳至 (PC+imm).

介绍了type后，介绍RISC-V中如何实现类似"call function"

正常情况，PC一条条运行指令，但跳至函数后，不像Label，要能返回跳时的下一指令，i.e., PC+4

可见应有reg保管PC信息 ⇒ ra (x1).

传参统一用 a_0-a_7 表示与保存 (a_0-a_1 用于返回)；

s_1, s_2-s_{11} 用于作saving reg；t_3-t_6 用作临时变量

sp: stack pointer, sp↓开辟空间，↑则恢复原状

☆ Calling convention: callee reg 通通存起来！

sp. s_1, s_2-s_{11}, s_0 (fp) (不要求了解).

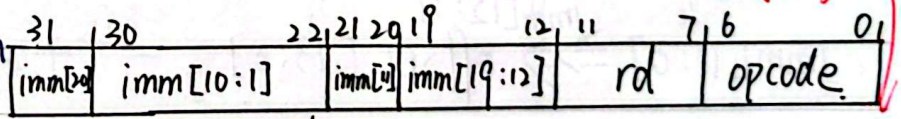
良好使用 calling convention，可以支持 nested call，因为在进入函数后，原来callee内容存至memory，那么function就能使用它们了；只需ret前lw拿回来即可

⚠⚠它是J-type！放在此介绍仅因其与function call有关

最后的J-type，就与函数调用有关：

J-type:

jalr rd rs1 label



rs1+imm 最低位清零 (2 byte对齐)

jump to label and return PC+4 to rd; if imm, jump to (rs1+imm)&~1

而：jal offset ⇔ jal x1, offset (x1=ra) | jal rd, offset, 跳至PC+offset并PC+4给rd.

j offset ⇔ jal x0, offset (相当于不存PC+4) (常用于Loop)

区别.

KOKUYO

No. Δ: 指令 machine code 中都是拿 imm 的；若传 label，则会计算 imm(offset)！

Date

jr rs ⇔ jalr x0, rs ⇔ jalr x0, rs, 0，跳至 rs，并不保存 PC+4（x0 永远为 0）

Δ: nested call 中，sp 移动且 sw callee reg 且 setup argument reg 如 add a0, s0, x0），这一堆操作称为 Prologue
而最后 lw 恢复 callee 且 sp 移回，这一堆操作称为 Epilogue

最后简单介绍 instruction → machine code 的 format:
rd, rs1, rs2 用 5 位 unsigned bits 表示 reg. 的编号 ($2^5=32$)
opcode: 7位 & funct3 (3位) & funct7 (7位) 共同决定是什么操作

但不是任何指令都有 funct3 & 7；也在 I type 中，funct3 & 7 together 足以判定指令

Encoding in: I-type: imm 12 bit, signed, ∈[-2048, 2047] ⇐ rule: 2 的补码
① imm 在指令用它操作前，要先 sign-extended !! ②但若用于位移，则直接取[4:0]，不必 sign-extend
③而 load 中，funct3 中第一位 indicates signed(0)/unsigned(1)
而后两位 indicates word(10)/h(01)/b(00)

(I type 3类任务):
① ⇒ arithmetic ② ⇒ shift ③ ⇒ load

S-type: funct3 前一位总0，后两位：word(10)/h(01)/b(00)

B-type: 如何表示 label*? 相对于该命令地址的相对地址!
use imm field as a two's complement offset to PC
可见，address 相对范围为 $\pm 2^n$。但是一个指令四 byte，怎么说也要 2-byte alignment，则 offset 最后两位无用，则：

offset[11:0] ⇒ offset[13:2]，一下子 offset 范围扩至 $\pm 2^{13}$

但若至少 2-byte，则是 1 位无用，范围：$\pm 2^{12}$ (from PC)
相当于：$\pm 2^{10}$ 32 bit instructions [signed]

Campus * 机器码中，imm(即 offset)的 [12:1] 被 encode

R-type: rd. rs1. rs2. 5 bit unsigned ⇒ No. reg

funct3 & 7 共同定 operation type

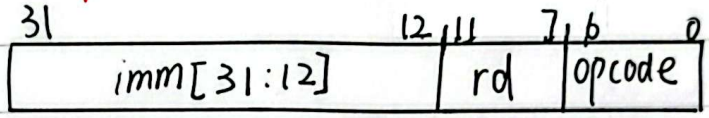
J-type: jal rd, label , store PC+4 to rd, jump to label

label = PC + offset (imm). , 故 label 转为 offset , ± 2^{18} 32-bit instructions (因为支持的 imm: [20:1])

I-type 补 jalr: store PC+4 to rd and jump to label = rs+imm

imm ∈ [-2048, 2047], 不再相对于 PC. 而相对于 rs reg 中的地址。注意: imm [11:0] 前10位全部保留, 而非最后一位清零, 因为 rs 中地址可能不是对齐的

U-type: Lui & auipc



lui rd, imm , rd = imm <<12

先前 I-type 中 imm 范围为 [-2048, 2047], 但和 lui 结合, 给 reg 使的数的范围就能 32-bit 了: [11:0]

i.e., li x5 imm ⇒ lui x5 imm[31:12] + addi x5, x5, imm

(附: 若 li rd imm 的 imm 小于等于12位, 则 addi 即可)

△: Given imm, signed, 若 imm 第12位为1呢? 则 低12位照取, 而高20位最后一位要加1

auipc: rd, imm: rd = PC + (imm . <<12).

这是因为 S type imm 12位, 但理应支持32位的 offset

则 Store/Load with PC相对32位 offset / 32位绝对地址:

auipc x5, <high20> / lui x5, <high20>

sw/lw rd, (<low12>) x5

↳ store/load ↳ 相对offset / 绝对address.

最后: 拿过来 32bit instruction, 如何看出指令?

step1: 查 opcode / funct3 / funct7 ⇒ type / operation.

step2: 找 rs1 / rs2 / rd / imm values (若存在)

KOKUYO