Contents

1 chapter 2 ISA and risc-v instruction

1.1 ISA ?

risc-v : ISA (instruction set architecture).
ISA , risc-v , Arm .
ISA , , 32bit64 bit. , . , . , 64 bit 32 bit .
,

1.1.1

1.

2.

3. . .

4. .

1.1.2

1.2 risc-v ISA

1.2.1

op dst, src1, src2
, dst(destination), src. . . C . imm .

1.2.2

32, x0–x31. RV32I RV64I , risc-v , . dark side
[1]. load save . x0 , 0. RV32I 32 , RV64I 64
, ... , .

x0	zero	0
x1	ra	return address
x2	sp	stack pointer
x3	gp	global pointer
x4	tp	thread pointer
x5-x7	t0-t2	temporary register
x8	s0/fp	save register/frame pointer
x9	s1	save register
x10-x11	a0-a1	return value
x12-x17	a2-a7	function argument
x18-x27	s2-s11	save register
x28-x31	t3-t6	temporary register

1.2.3

Save Load, . .

Byte, Byte 8 bit. , Byte . 0xFF. word 32 bit, 4 Byte. double word (dword) word, 8 Byte. half word, word, 16 bit. save load .

1.3 Instructions introduction

1.3.1 primitive instructions

- add
- sub
- subi

[1] x86 , . add .
add x1, x1, Imm(x2)
(, x1 x86), Imm + x2 , x1 x1 .

- imm
- mul
- div
- or and
- sll sla
- save load
- shamt
- bne beq
- jalr jal

1.3.2 pseudo instruction

, mv, sw, lw (sw, lw ,). , la li .
`Label rd` . `Label` PC , `delta`, 32 .

1.4 How to write a function using assembly [0/3]

☐ part 1

☐ part 2

☐ part 3

1.4.1 How to write a loop

, , :
for (int i = 0 ; i < 2 ; i++) { ans = ans + i; }

1. Label
2. ans = ans + i;
3. i++
4.

: add t0, x0, x0 add t1, x0, x0 li t2, 2 Loop: add t1, t1, t0 addi t0, t0, 1 bne t0, t2, Loop
, t0 i; t1 ans; t2 2. , bne bge . Loop.

1.4.2 the concept of basic block

basic block. , , .

1.4.3 Function and stack

stack, stack . , , , stack . bing :

factorial: addi sp, sp, -16 # Allocate space on stack sw ra, 12(sp) # Save return address sw a0, 8(sp) # Save argument n addi a1, x0, 1 # Initialize result to 1 beq a0, x0, end # If n == 0, jump to end loop: mul a1, a1, a0 # result *= n addi a0, a0, -1 # n– bne a0, x0, loop # If n != 0, jump to loop end: mv a0, a1 # Return result in a0 lw ra, 12(sp) # Restore return address addi sp, sp, 16 # Deallocate space on stack ret # Return from function

1.5 The expression of an instruction

1.5.1 the field of an instruction code

instruction code 32. 56field, field. , field. add .

field	funct7	rst2	rst1	funct3	rd	opcode
bits	7	5	5	3	5	7

R-type .

1.5.2 the type of the instruction code

, R, I, J, S, SB, U .

2 imm

i, . . , , . e.g. `addi x1, x1, 5`
, , `addi`, `subi`. `f = g - 10` `addi x3, x4, -10` %EndOfParagraph

3 mul

mulh . ppt %EndOfParagraph

4 div

ppt. %EndOfParagraph

`sw`	
`sd`	
`sh`	
`sb`	byte

Figure 1:

5 or and and

, risc-v . , .

6 shift

, , , , . .
, . : , , , , . , , 1, 0, .

7 save load

. .
, , `save load` . risc-v . x86 . : `memop reg, offset(bAddrReg)` ppt. `memop` (`s l`), `reg` , `offset(bAddrReg)`, `offset` ; `bAddrReg` , . `offset + bAddrReg`. [2]
, ~1 %EndOfParagraph

8 sign extension

`load` , , `w`, `d`, `h`, `b`. , 32/64/16/8 , 32 64 . , . , , 32 64.
`0x80`, `10000000`. 1, , 32 `0xFF80`.
, , , , , `u`.
e.g. . ppt .

9 add

add add rd, rst1, rst2
rd is register of destination

[2] x86 , ,

The command tell computer to do the computation—`rd = rst1 + rst2`. Note that rst1 rst2 are treated as signed number.

Additionally, `addi` tells computer to do `rd = rst1 + imm`, where `rst2` is replaced by an immediate number.

10 sub

{
`sub, rd, rst1, rst2`} is the form of the instruction, telling that `rd = rst1 - rst2`.

It is worth noting that there is no such thing as `subi`, cause `addi` can do the same thing.

10.0.1 mul

11 div rem

12 xor

Logical operations can do thing likes . To achieve this we can use `and` and `0xFFFFFFFF`. Let us assume that the length of register is 32. A number `and 0xFFFFFFFF` is the number itself. But when it `and 0x0FFFFFFF`, the number loose it first four bits.

13 shift

{
`s + l/r + a/l + [i]`} is the decompostion of an instruction, where `s` for shift, `l,r` for **left** or **right**, `a,l` for **arithmetic** or **logical**. `[i]` is optional, which stands for `imm`.

About the arithmetic shift, you check 01.pdf out.

14 shamt

Indeed, for a 64-bits data store in a register. It would be of no use to shift of 64-bits, which resulting that in `slai` or other shifting command ending with `i`, only the lowest 6-bits of immediate number are useful. Other bits are abandoned. The remaining part is called shamt.

15 shift left arithmetic

There is not such thing as `sla[i]`. We already know when the shift cause ailment. Exactly when the number is starting with `10` or `01` the result of `sla` is not what we want.

However, you may check that when there is no ailment, `sla` works just like `sll`. So `sla` become less needed.

15.0.1 s, l

{
`s/l r, offset(AddR)`} where `AddR` is a register. The command tells computer to load data from address `offset + AddR` to `r`, or to save the data in `r` to address `offset + AddR`.

15.0.2 address and word and byte

A **word** in risc-v has **32** bits. There arouses an interesting question: how to load 32-bit data to a 64-bit register?

No, what I am saying is that you need to care for whether the data is unsigned type or not. You need to expand a number when it is treated as a negative number.

15.0.3 slt

slt for set less than. `slt` is an instruction to compare the value of some data.

{
`slt rd, rst1, rst2`} means that `rd = rst1 and rst2`

16 bne beq blt bltu

1. b for break
2. eq for equal.
3. ne for not equal.
4. lt for less then.
5. ge for greater or equal.

Use this set of command to jump which is used to achieve if-else structure. Note that for blt and bge, there exists unsigned type of commands.

17 jalr jal

jal rd, Label

. PC+4 rd . PC `Label`.

`Label` , `Exit Loop` . .

The actual operation it takes is PC = PC + `offSet`, where `offSet` is translated from `Label`.

jalr rd, offSet(AddR)

`jalr` . , `rd` = PC + 4. PC `offset + AddR`.