

Chapter-2

(How do we get a RISC-V instruction set from a high-level code?)

RISC-V → instruction set architecture.
Reduced Instruction Set Computer
an abstract interface between the hardware and the lowest level software.

RISC-V ISA has a modular design.
Base + Optional Extension
ISA

Unit	=> Double Word = 64 bits
Word	= 32 bits
Half Word	= 16 bits
Byte	= 8 bits

Registers:

⇒ Used for frequently accessed data. Registers are faster than regular memory.

⇒ RISC-V has 32, 64-bit registers.] 64 bit architecture ⇒ handling 64 bit data at a time.
Follows Design Principle 2 ⇒ Smaller is faster

↳ The clock period of the system could be limited by many different things, Register File is also one of those factors.

Hence, computer architects pick the size of the register file very carefully. 32×64 bit is a small enough register file that is not a limiting factor for our systems.

RISC-V Register Details:

Theoretically you could store any information in any register but architectural programmers have decided to follow a convention, in which they use certain registers for certain purposes. ↴ increases readability

Register Num.	Functionality
x0	Constant value 0 => holds constant value 0.
x1	Return address => to store the return address that you should return to after a function call.
x2	Stack pointer => stores an address in memory with the top of the stack.
x3	Global pointer
X4	Thread pointer
x5-7 ✓✓	Temporaries
x8	Saved register / Frame pointer ↴ => Temporary reg.
x9 ✓✓	Saved register => to store variables.
x10-11	Function arguments / return values] for values passed to a function
x12-17	Function arguments] and values returned from a function.
x18-27 ✓✓	Saved registers
x28-31 ✓✓	Temporaries => for holding temp. values

Operands: part of instruction that contains the data to act on

Based on the physical location

or, the memory location of the data in a register.

- Register operand
(small capacity + faster access time)
- Memory operand
(large capacity + longer access time)
- Constant / immediate
(physically in the instruction itself)

Register Operand: Arithmetic instructions use register operands.

$$f = (g+h) - (i+j);$$

↳ Only one operation is performed per RISC-V instruction.

Hence, compiler must break this statement down to several parts.

RISC-V code:
1 codeline divided into multiple instruction lines

add x_5, x_{20}, x_{21}	<small>g+h is a step towards our actual answer. one of Hence stored in the temp rega.</small>
add x_6, x_{22}, x_{23}	<small>same explanation as the prev. one</small>
sub x_{10}, x_5, x_6	

$$\begin{aligned} f &\Rightarrow x_{10} \checkmark \\ g &\Rightarrow x_{20} \\ h &\Rightarrow x_{21} \\ i &\Rightarrow x_{22} \\ j &\Rightarrow x_{23} \end{aligned}$$

Add/Sub instruction syntax:

Instruction name
add dest., source1, source2
 \Rightarrow dest. = source1 + source2

sub dest., source1, source2
 \Rightarrow dest. = source1 - source2

always 3 registers
operands

Memory Operand:

* We store the composite data in Main Memory.

* In order to perform arithmetic operations we must load data from memory to register.

* Memory is Byte addressed. (each slot contains 8 bits of data)

* No alignment restrictions.

words must start at addresses that are multiple of 4.
double words " " " " " " " " " " 8.

* RISC-V is little endian

unique
memory

address ↓

Each slot consists of 8 bits

#0000	0 000 0000	
#0001	0 000 0000	
#0002	0 000 0000	
#0003	0 000 0000	64
#0004	0 000 0000	bit
#0005	0 000 0000	
#0006	0 000 0000	
#0007	0 000 1010	
#0008	next data - 1	
#0009	u	
#0010	u	64
#0011	u	bit
#0012	u	
#0013	u	
#0014	u	
#0015	u	
#0016	next data - 2	
:	:	
:	64	
:	bit	
:		
:		
:		
:		
#n		

64 bit architecture

↳ represent each into 64 bits.

suppose we want to store $(10)_10$ in memory.

0 000 0000 0000 0000
0 000 0000 0000 0000
0 000 0000 0000 0000
0 000 0000 0000 0000
0 000 0000 0000 1010
LSB ↑

Hence, in order to retrieve a 64 bit data we need to choose 8 consecutive slots.

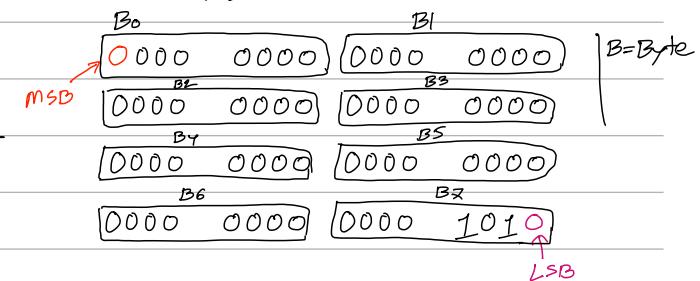
Endianness: The order in which computer memory stores data. (byte)

→ Big Endian

→ Little Endian (Followed by RISC-V)

↳ Bytes are ordered from right to left.

↳ LSB stored at lowest address.



#0000	B7 0000 1010	LSB
#0001	B6 0000 0000	
#0002	B5 0000 0000	
#0003	B4 0000 0000	64
#0004	B3 0000 0000	bit
#0005	B2 0000 0000	
#0006	B1 0000 0000	
#0007	B0 0000 0000	MSB

$A[\text{index}] = \text{Base Register} + \text{offset}$

word register that contains the Base Address

Actual address = [Base Add. + offset]

Let's assume that A is an array of 100 doublewords and that the compiler has associated the variables g and h with the registers x_{20} and x_{21} as before. Let's also assume that the starting address, or *base address*, of the array is in x_{22} . Compile this C assignment statement:

$g = h + A[8];$

Memory Operand

ld $x_5, 8 \times 8 [x_{22}]$
 ↓
 temp. reg.
 off.
 Base Reg.
 add x_{20}, x_{21}, x_5

$g = x_{20}$

$h = x_{21}$

Base of $A = x_{22}$

Assume variable h is associated with register x_{21} and the base address of the array A is in x_{22} . What is the RISC-V assembly code for the C assignment statement below?

$A[12] = h + A[8];$
 ↙ Store back to mem
 ↘ memory

ld $x_6, 64 [x_{22}]$
 add x_6, x_{21}, x_6
 sd $x_6, 96 [x_{22}]$

$h = x_{21}$

Base of $A = x_{22}$

Load / Store Syntax :

Load \Rightarrow loads data from memory to reg.

Store \Rightarrow stores data back to memory from reg.

ld $\underbrace{\text{reg}}, \underbrace{\text{memory}}$
 ↗ Data copy

sd $\underbrace{\text{reg}}, \underbrace{\text{memory}}$
 ↗ Data copy

ld \Rightarrow load double word	(64 bits)
lw \Rightarrow load word	(32 bits)
lh \Rightarrow load half word	(16 bits)
lb \Rightarrow load byte	(8 bits)

Immediate Operands : (avoids a load instruction)
 faster
 if constant data specified in an instruction, **No sub!**

Addi Instruction Syntax :

addi $\underbrace{\text{dest}}, \underbrace{\text{Source 1}}, \underbrace{\text{Source 2 (constant)}}$
 $\Rightarrow \text{dest.} = \text{source 1} + \text{source 2}$

If compiler can not add/sub two integers.

addi $x_{22}, x_{22}, 4$

immediate to sub 4 \Rightarrow make it -4
 addi $x_{22}, x_{22}, -4$

$$A - B = A + (-B)$$

	Range
Unsigned	0 to $2^m - 1$
2's Com.	-2^{m-1} to $(2^m - 1)$

Signed Negation

negate +2

$$+2 = 0000\ 0000 \dots 0010$$

$$\begin{array}{r} \text{pos} \\ \hline 1111\ 1111\ \dots\ 1101 \\ +1 \\ \hline -2 = 1111\ 1111\ \dots\ 1110 \end{array}$$

Sign Extension

representing a number with more bits.

Keep the value same.

↳ if signed :

Replicate the sign bit to the left.

else :

extend 0s to the left.

$+2 = 0000\ 0010$ [8 bit] \Rightarrow 16 bit	$-2 = 1111\ 1110$ [8 bit] \Rightarrow 16 bit	pos neg
		0000 0000 0000 0010 1111 1111 1111 1110

LB \Rightarrow Load Byte (Signed extension)

LBU \Rightarrow Load Byte (Unsigned extension)

Translating a RISC-V assembly instruction into a Machine

Instruction

Higher-level Language
Program

A[30] = h + A[30] + 1;

↓ compiler

Assembly Language
Program

```
ld x9, 240(x10) // Temporary reg x9 gets A[30]
add x9, x21, x9 // Temporary reg x9 gets h+A[30]
addi x9, x9, 1 // Temporary reg x9 gets h+A[30]+1
sd x9, 240(x10) // Stores h+A[30]+1 back into A[30]
```

↓ Assembler

Machine Language
Program

immediate	rs1	funct3	rd	opcode
000011110000	01010	011	01001	00000011
funct7	rs2	rs1	funct3	rd
0000000	01001	10101	000	0100011
immediate	rs1	funct3	rd	opcode
000000000001	01001	000	01001	00100011
immediate[11:5]	rs2	rs1	funct3	immediate[4:0] opcode
0000111	01001	01010	011	10000 0100011

Machine only understands high and low electronic signals.

In RISC-V instruction takes exactly 32 bits (one word)

The layout of the instruction is called Instruction Format.

And \Rightarrow Bit masking

and $\underbrace{x_9}_{P2d}, \underbrace{x_{10}}_{P2s1}, \underbrace{x_{11}}_{P2s2}$

$$\begin{array}{ccccccccc} x_{10} = & 0000 & \dots & 0000 & 1100 & 1100 & & \text{only these} \\ x_{11} = & 0000 & \dots & 0000 & 0000 & 1100 & & \text{two bits should} \\ & & & & & & & \text{remain as it} \\ & & & & & & & \text{is, rest 0.} \end{array}$$

$$x_9 = 0000 \dots 0000 \quad 0000 \quad 1100$$

OR \Rightarrow Include Bits

and x_9 , x_{10} , x_{11}

$X_{10} = 0000 \dots 0000 \quad | \downarrow 100 \quad 0000$

$X_{11} = 0000 \dots 0000 \quad 0011 \quad 0000$

$X_0 = 0000 \dots 0000 \quad 0011 \quad 0000$

you want to
set these bits
to 1 and rest
should remain as
it is.

$\times OR \Rightarrow$ Can work as Buffer / Not

x_0 , x_1 , x_2

A	B	<u>A ⊕ B</u>
0	0	0
1	0	1

xor with
0 = Buffer

0	1	1	xor with
1	1	0	1 = Not