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

Register:

=> 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 3 => 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 x 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 architects/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
x9 ✓	Saved register => to store variables.
x10-11	Function arguments / return values
x12-17	Function arguments
x18-27 ✓	Saved registers
x28-31 ✓✓	Temporaries => For holding temp. values

11 => Saved reg.

7 => Temporary reg.

] for values passed to a function and values returned from a function.

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

Based on the physical location

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

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

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:

```
add  x5, x20, x21   # g+h is a step towards our actual answer. one of Hence, stored in the temp regs.
add  x6, x22, x23   # same explanation as the prev. one
sub  x19, x5, x6
```

1 codeline divided into multi-ple instruction lines

$f \Rightarrow x_{19}$	
$g \Rightarrow x_{20}$	
$h \Rightarrow x_{21}$	
$i \Rightarrow x_{22}$	
$j \Rightarrow x_{23}$	

Add / Sub instruction syntax:

Instruction name

```
add  ____ , ____ , ____
     dest.  Source1  Source2
```
⇒ dest. = source1 + source2

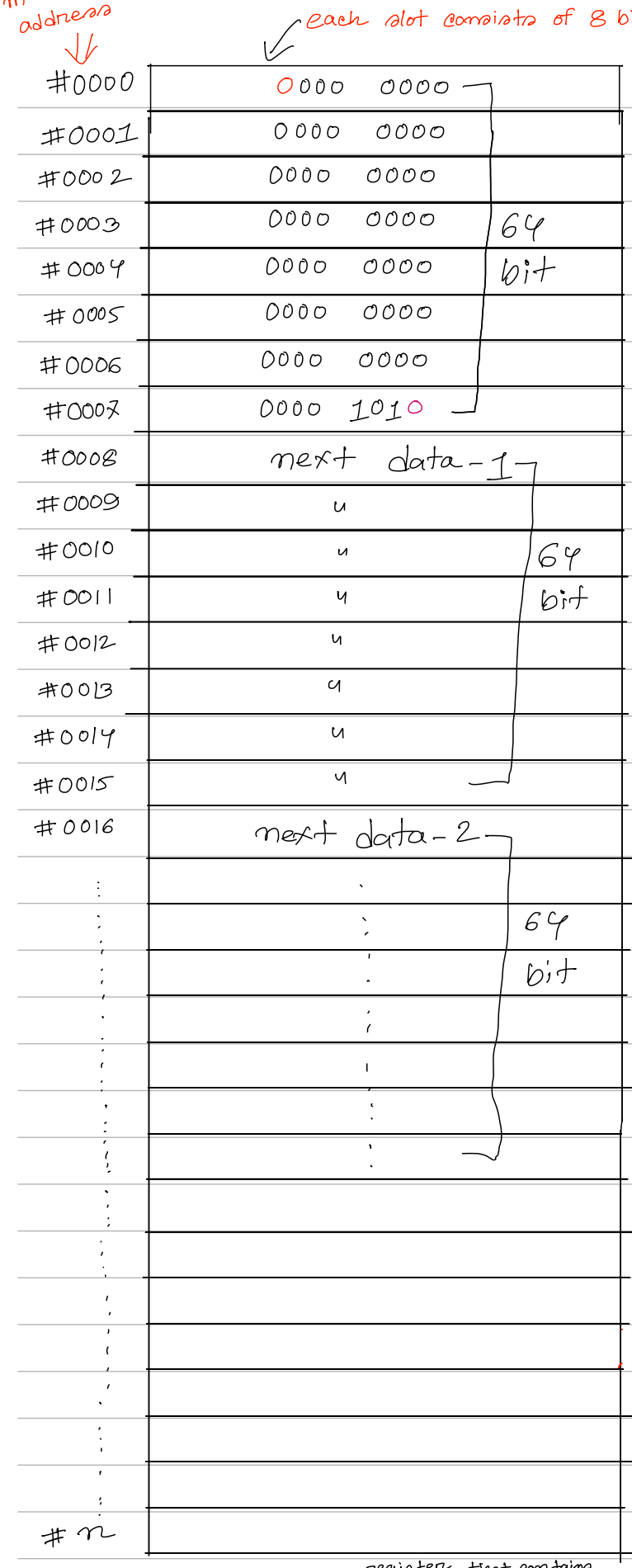
```
sub  ____ , ____ , ____
     dest.  Source1  Source2
```
⇒ dest. = source1 − source2

always 3 register operands

Memory Operand:

- We store the **composite data** (array) 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



64 bit architecture
↳ represent each into 64 bits.

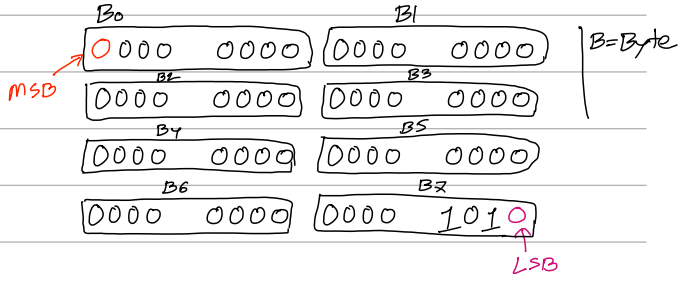
suppose we want to store $(10)_{10}$ in memory.

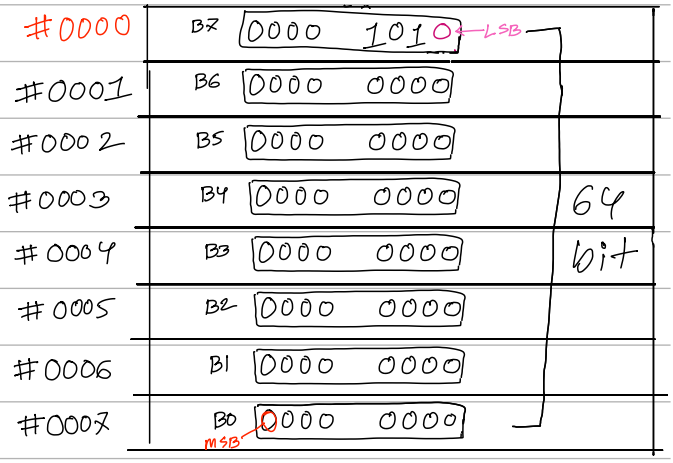
MSB
0000 0000 0000 0000
0000 0000 0000 0000
0000 0000 0000 0000
0000 0000 0000 1010
LSB

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.





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 `x20` and `x21` as before. Let's also assume that the starting address, or *base address*, of the array is in `x22`. Compile this C assignment statement:

```
g = h + A[8];
```

Memory Operand

```
ld  x5, 8x8[x22]
add x20, x21, x5
```

(x5: temp. reg. | 8x8: off. | x22: Base Reg.)

$g = X_{20}$
$h = X_{21}$
Base of $A = X_{22}$

Assume variable `h` is associated with register `x21` and the base address of the array `A` is in `x22`. What is the RISC-V assembly code for the C assignment statement below?

```
A[12] = h + A[8];
```

Store back to mem. | Memory

```
ld  x6, 64[x22]
add x6, x21, x6
sd  x6, 96[x22]
```

$h = X_{21}$
Base of $A = X_{22}$

Load / Store Syntax:

Load => Loads data from memory to reg.

Store => Stores data back to memory from reg.

```
ld  reg, memory
```

(Data copy: memory → reg)

```
sd  reg, memory
```

(Data copy: reg → memory)

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

Immediate Operands: (avoids a load instruction) faster

constant data specified in an instruction, No subi

Addi instruction syntax:

```
addi dest., Source1, Source2 (constant)
```

=> dest. = source1 + source2

addi x22, x22, 4

inorder to sub 4 => Make it -4

addi x22, x22, -4

A - B = A + (-B)

Compiler can not add/sub two integers.

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

Signed Negation

negate +2

+2 = 0000 0000 0010

-2 = 1111 1111 .... 1101 + 1 = 1111 1111 1110

Sign Extension

representing a number with more bits.

Keep the value same.

↳ if signed :

Replicate the sign bit to the left.

Ex: +2 = 0000 0010 (pos)] 8 bit => 16 bit 0000 0000 0000 0010

-2 = 1111 1110 (neg)] 8 bit => 16 bit 1111 1111 1111 1110

else :

extend 0s to the left.

Ex: 2 = 0000 0010] 8 bit => 16 bit 0000 0000 0000 0010 (extension)

LB => Load Byte (Signed extension)

LBU => 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	0000011

funct7	rs2	rs1	funct3	rd	opcode
0000000	01001	10101	000	01001	0110011

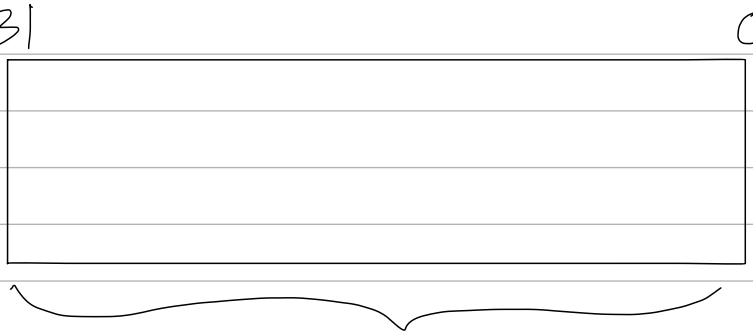
immediate	rs1	funct3	rd	opcode
000000000001	01001	000	01001	0010011

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
0000111	01001	01010	011	10000	0100011

Machine only understands high (1) and low (0) el .nic / nals.

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

The layout of the instruction is called Instruction Format.



Divide the 32 bits of an instruction into "fields"

↳ Conflict

Desire to keep all the instructions length same

V/s

Desire to have a single instruction format.

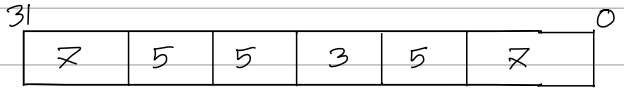
Design Principle 3: Good design demands good compromises.

⇒ RISC-V chooses to keep all the instruction length same; thereby requiring distinct instruction formats for different instructions.

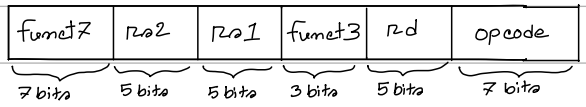
Instruction Formats:

- ↳ R type ⇒ instructions that use 3 registers. (Add, Sub, SLL, XOR, OR, AND, ...)
- ↳ I type ⇒ " " " immediate and 2 registers. (Addi, SLLI, SRLI, ORI, ANDI, Load)
- ↳ S type ⇒ Store instruction

R type instruction



* each field has unique name and size.



(Partially)

Opcode = Denotes the format of an instruction and instruction itself.

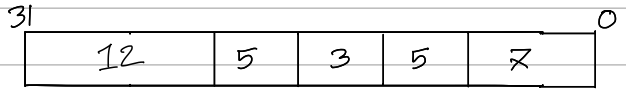
rd = Register destination operand

rs1 = " source 1 "

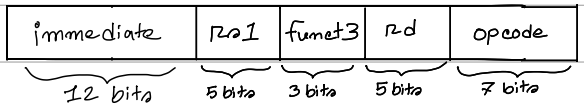
rs2 = " source 2 "

funct3 = } their combination tells
funct7 = } us which instruction
to perform.

I type instruction



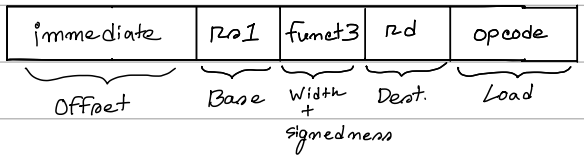
* each field has unique name and size.



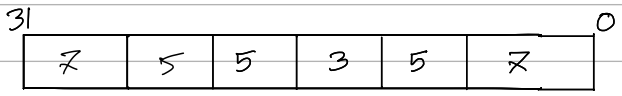
Immediate = Constant / offset [2's comp.]

rs1 = Source / Base Register number

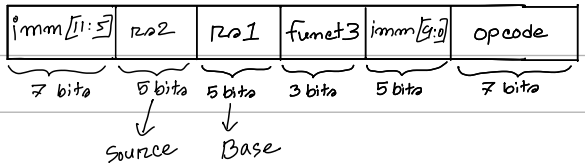
Load



S type instruction



* each field has unique name and size.



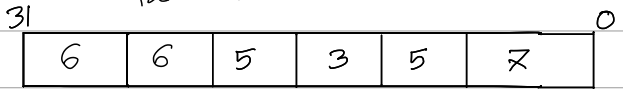
reason for imm. split is they want to keep rs1 and rs2 fields in the same place in all instruction formats.

Shift Operation

↳ follows I-type but modified

slli x11, x10, 4

rd rs imm



Why ?? ⇒ If you shift a 64 bit value more than 63 bits what happens?

Shift Left

↳ shift left and fill the positions with 0

⇒ We can perform multiplication by 2^i using slli.

slli x11, x10, 4

↓

the value that will be stored in x11 is basically, val in $x_{10} * 2^4$

Shift Right

↳ shift right and fill the positions with 0

⇒ We can perform division by 2^i using srli.

srli x11, x10, 4

↓

the value that will be stored in x11 is basically, val in $x_{10} / 2^4$

And ⇒ Bit Musking

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

(rd, rs1, rs2)

x_{10} = 0000 .... 0000 1100 1100 ← only these two bits should remain as it in, rest 0.

x_{11} = 0000 .... 0000 0000 1100

x_9 = 0000 .... 0000 0000 1100

OR ⇒ Include Bits

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

(rd, rs1, rs2)

x_{10} = 0000 .... 0000 1100 0000 ← you want to set these bits to 1 and rest should remain as it is.

x_{11} = 0000 .... 0000 0011 0000

x_9 = 0000 .... 0000 0011 0000

XOR ⇒ Can work as Buffer / Not

XOR x_9, x_{10}, x_{11}
(12d, 1251, 2882)

A	B	A ⊕ B
0	0	0
1	0	1
0	1	1
1	1	0

XOR with 0 = Buffer

XOR with 1 = Not