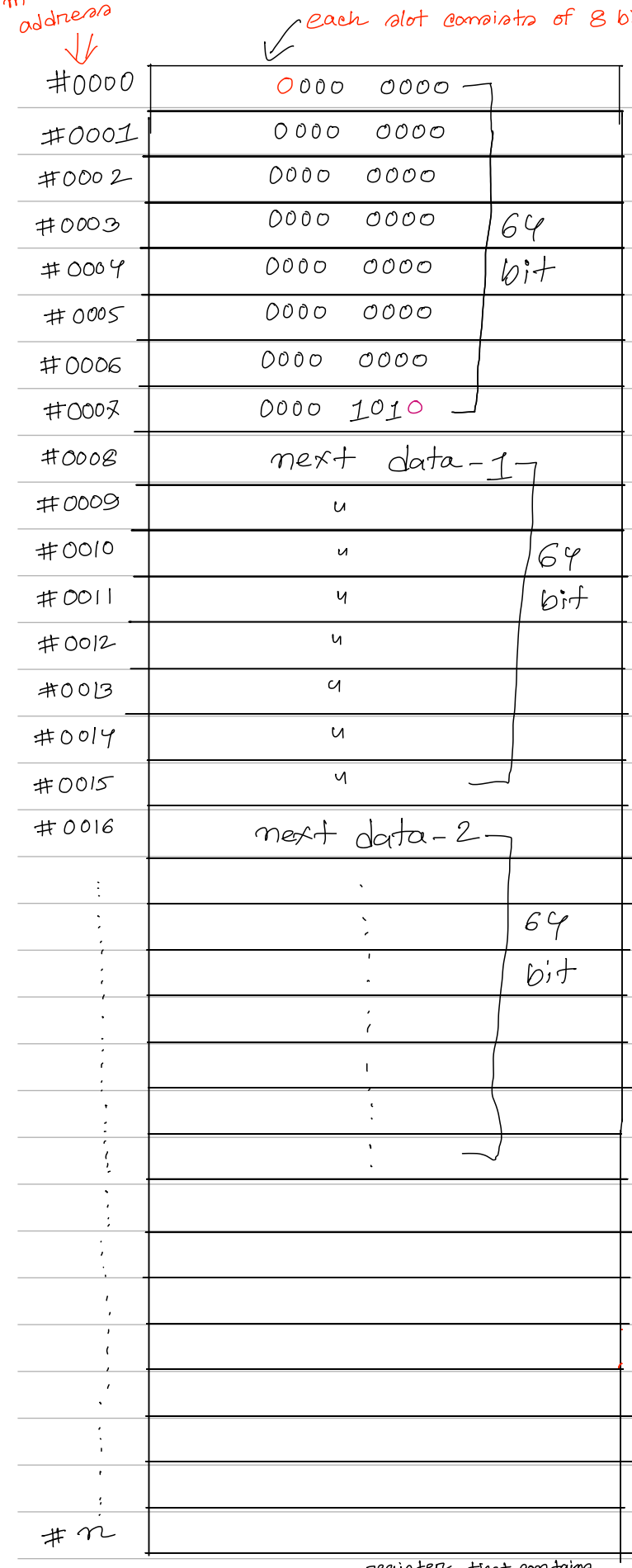
Memory Operand:

- 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.

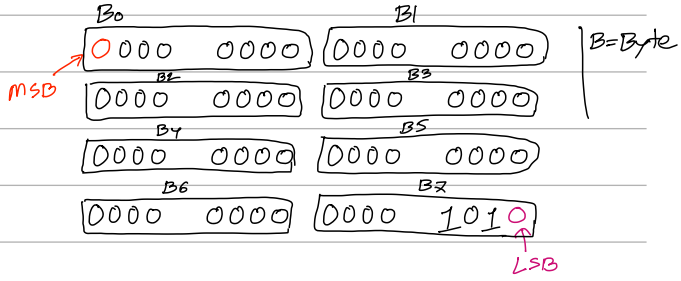
suppose we want to store $(10)_{10}$ in memory.

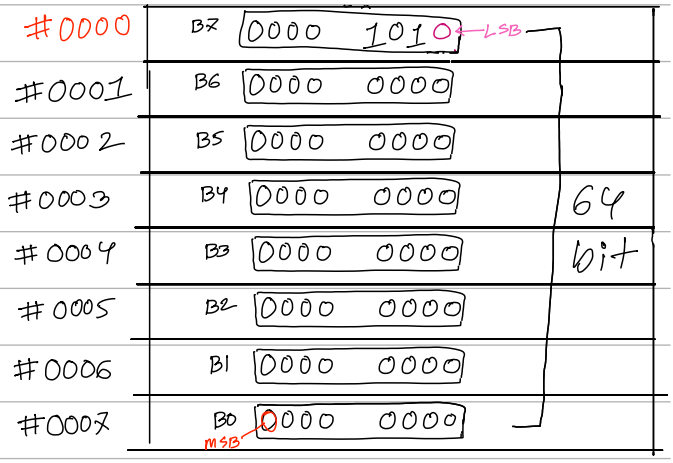
MSB
0000 0000 0000 0000
0000 0000 0000 0000
0000 0000 0000 0000
0000 0000 0000 1010
LSB

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] 8 bit => 16 bit 0000 0000 0000 0010 (pos)

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

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

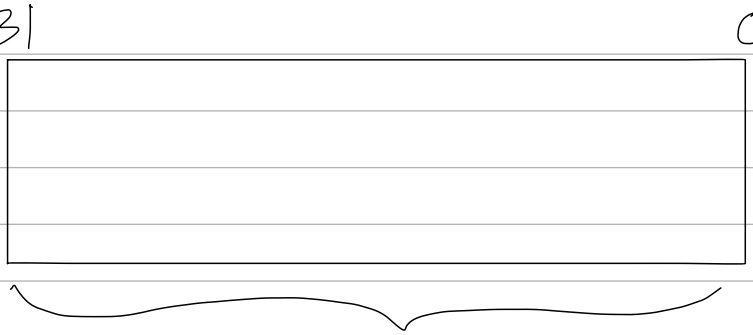
immediate	rs1	funct3	rd	opcode
000011110000	01010	011	01001	0000011

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) electronic signals.
* In RISC-V instruction takes exactly 32 bits
* 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.

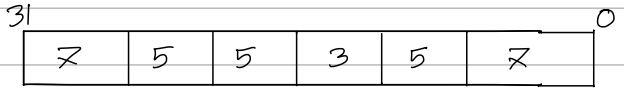
Design Principle 3: Good design demands good compromises.

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

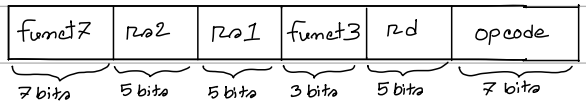
Instruction Formats:

- ↳ 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.

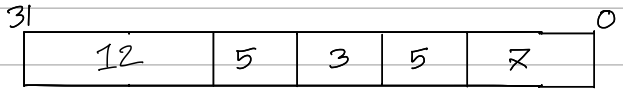
rd = Register destination operand

rs1 = " source 1 "

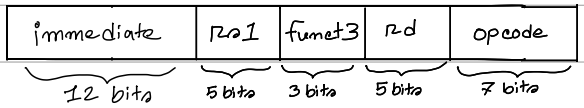
rs2 = " source 2 "

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

I type instruction



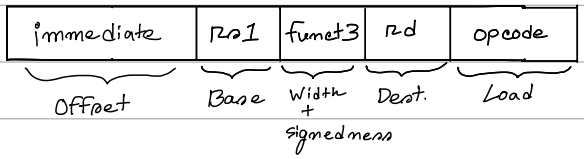
* each field has unique name and size.



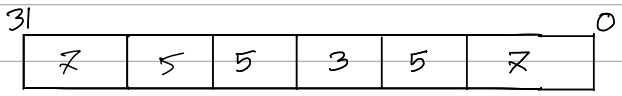
Immediate = Constant / Offset [2's comp.]

rs1 = Source / Base Register number

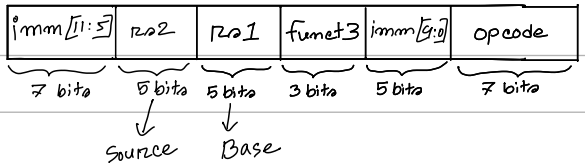
Load



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.