

### Memory Operands

array

\* 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

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.

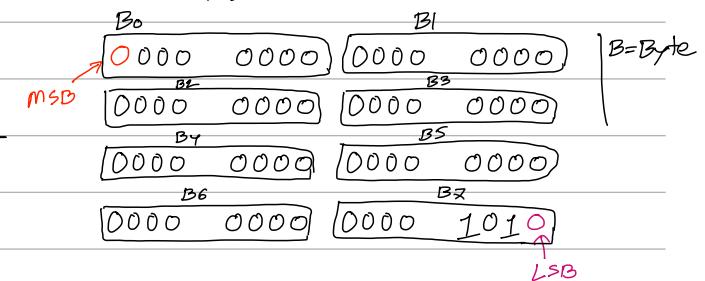
**Endianness:** The order in which computer memory stores data. (byte)

↳ Big Endian

↳ Little Endian (Followed by RISC-V)

↳ Bytes are ordered from right to left.

↳ LSB stored at lowest address.



|       |    |            |     |
|-------|----|------------|-----|
| #0000 | B7 | 0 000 1010 | LSB |
| #0001 | B6 | 0 000 0000 |     |
| #0002 | B5 | 0 000 0000 |     |
| #0003 | B4 | 0 000 0000 | 64  |
| #0004 | B3 | 0 000 0000 | bit |
| #0005 | B2 | 0 000 0000 |     |
| #0006 | B1 | 0 000 0000 |     |
| #0007 | B0 | 0 000 0000 | MSB |

A[index] = 8 \* index (Base Register)  
word  
register that contains the Base Address  
offset

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

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

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

If compiler can not add/sub two integers.

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

## Signed Negation

# negate +2

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

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

## 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 |
| <b>pos</b><br>↓                                | <b>neg</b><br>↓                                |

# 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

$$\checkmark A[30] = h + A[30] + 1;$$

↓ 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    | 01001                 |
| 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 01000011        |

\* Machine only understands high and low electronic signals.

\* In RISC-V instruction takes exactly 32 bits

\* The layout of the instruction is called Instruction Format.

31

O



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**.

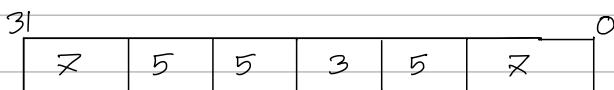
Design Principle 3: Good design demands good compromises.

⇒ 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 ⇒    u    u    immediate and 2 registers. (Addi, SLLI, SR LI, ORI, ANDI, Load)
- ↳ S type ⇒ Store instruction

**R type instruction**



\* each field has unique name and size.

| funct7 | r2     | r2     | r2     | funct3 | r2d    | opcode |
|--------|--------|--------|--------|--------|--------|--------|
| 7 bits | 5 bits | 5 bits | 3 bits | 5 bits | 7 bits |        |

(Partially)

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

r2d = Register destination operand

r2l =    u    source1        u

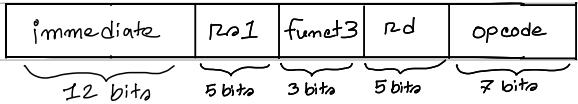
r2r =    u    source2        u

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

## I type instruction

|    |    |   |   |   |   |   |
|----|----|---|---|---|---|---|
| 31 | 12 | 5 | 3 | 5 | 2 | 0 |
|----|----|---|---|---|---|---|

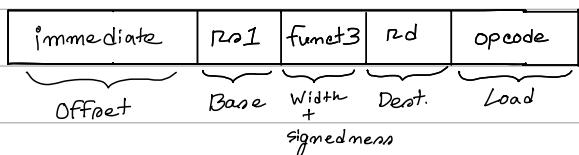
# each field has unique name and size.



Immediate = Constant / offset [2s comp.]

r201 = Source / Base Register number

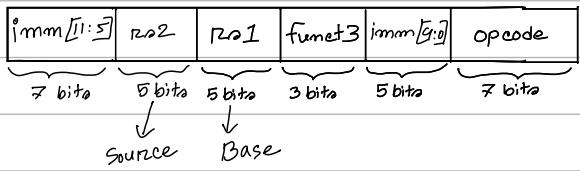
## Load



## S type instruction

|    |   |   |   |   |   |   |   |
|----|---|---|---|---|---|---|---|
| 31 | 2 | 5 | 5 | 3 | 5 | 2 | 0 |
|----|---|---|---|---|---|---|---|

# each field has unique name and size.



# reason for imm. split is they want to keep r201 and r202

fields in the same place in all instruction formats.