CSE340: Computer Architecture

Handout_Chapter - 2: Instructions: Language of the Computer

Inspiring Excellence

Prepared by: Partha Bhoumik (PBK)
Course Coordinator, CSE340

The RISC-V Instruction Set

To command a computer's hardware, you must speak its language. The words of a computer's language are called instructions and its vocabulary is called an instruction set. The chosen instruction set is RISC-V, originally developed at UC Berkeley starting in 2010.

RISC stands for Reduced Instruction Set Computer.

To understand more about ISA. You can read this article
RISC-V has a modular design. Base ISA + Optional Extension.

Different Units

Unit	Size (bits)
Double Word (D)	64
Word (W)	32
Half Word (H)	16
Byte (B)	8

Note: The size of a word varies in different architectures.

Registers in RISC-V

Used for frequently accessed data. Registers are faster than memory. Data must be in registers to perform any arithmetic operations.

RISC-V has a 32 X 64-bit register file. This means there are 32 registers each can store 64 bits of data.

What do you understand by a 64-bit architecture?

RISC-V register details:

Theoretically, you can store any information in any register but architects or programmers decided to follow a convention, in which they use certain registers for certain purposes. It increases the readability of the code.

Register Number	Functionality
x0	Holds constant value 0. (Zero register)
x1	Stores the return address where you should return to after a function call is executed (Return Address register)
x2	Stores an address in the memory with the top of the stack. (Stack Pointer register)
x3	Global Pointer register.
x4	Thread Pointer register.
x5 to x7 and x28 to x31	Temporary register.
x8	Frame Pointer register.
x9, x18 to x27	Saved Register.
x10 to x11	Function arguments/ Return Values
x12 to x17	Function arguments

Operands

The part of an instruction that contains the data to be acted on or the memory location of the data.

Based on the physical location of data, there are three types of operands in RISC-V:

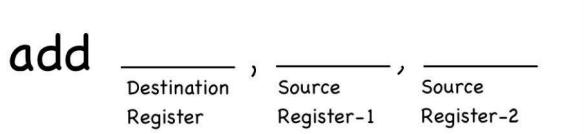
i. Register Operand. (Small capacity + Faster access time)
ii. Memory Operand. (Large capacity + Longer access time)
iii. Constant/Immediate. (Data is present in the instruction itself)

Register Operand

A register operand refers to a situation where the data used in an instruction is stored in a register.

Add Instruction:

This is an arithmetic instruction that performs integer addition.

Syntax: 

Example:
add x18, x19, x20
Explanation:

Operands:
x19: Contains the first operand, which is 12.
x20: Contains the second operand, which is 10.

Destination:
x18: The result of the addition will be stored in this register.

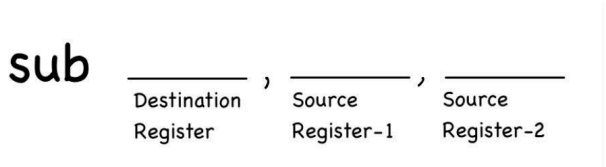
Execution:
i. The processor reads the values stored in x19 (12) and x20 (10).
ii. It performs the addition: 12 + 10 = 22. (***val of source reg1 + val of source reg2***)
iii. The result, 22, is written into the destination register x18.

Outcome:
After execution, the contents of the registers will be:
x18 = 22
x19 = 12 (unchanged)
x20 = 10 (unchanged)

Sub Instruction:

This is an arithmetic instruction that performs integer subtraction.

Syntax:



Example:
sub x18, x19, x20
Explanation:

Operands:
x19: Contains the first operand, which is 12.
x20: Contains the second operand, which is 10.

Destination:
x18: The result of the addition will be stored in this register.

Execution:
i. The processor reads the values stored in x19 (12) and x20 (10).
ii. It performs the addition: 12 - 10 = 2. (***val of source reg1 - val of source reg2***)
iii. The result, 2, is written into the destination register x18.

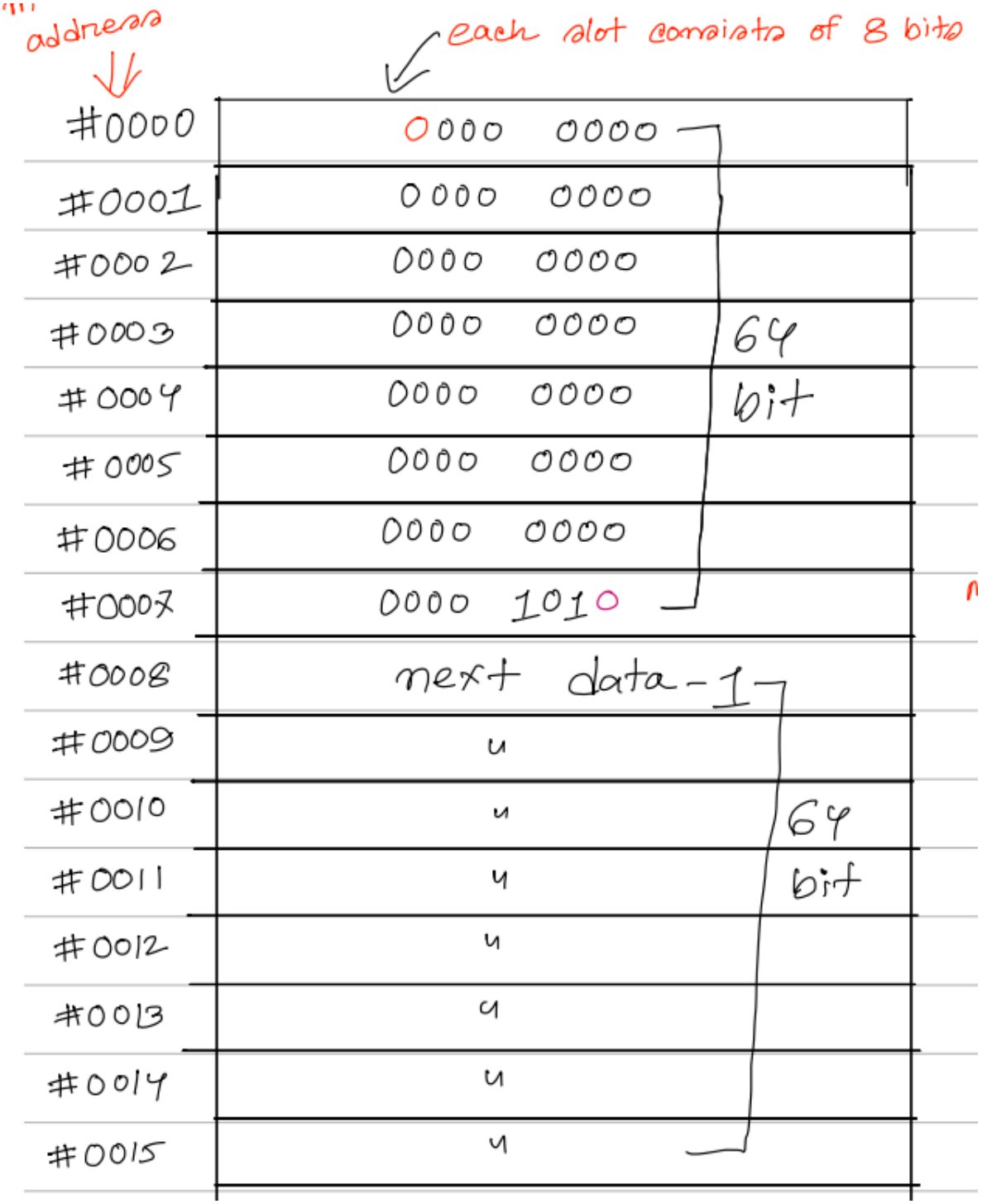
Outcome:
After execution, the contents of the registers will be:
x18 = 2
x19 = 12 (unchanged)
x20 = 10 (unchanged)

Memory Operand

Composite data, such as an array, is stored in memory. To perform arithmetic operations on this data, it must be loaded into a register first.

Memory features:

i. Memory is Byte addressed:
Each slot in memory can store 8 bits of data.



ii. No alignment restrictions:

Alignment means that a word must start at an address that is a multiple of 4, and a double word must start at an address that is a multiple of 8. However, RISC-V allows data to be stored in both aligned and unaligned ways, providing flexibility in memory storage.

iii. RISC-V is little endian:

Endianness refers to the order in which a computer stores the bytes of multi-byte data in memory.

Two approaches:

1. Little Endian -

Least Significant Byte is stored at the lowest address.

Example: Storing the following 64-bit data in memory in little-endian approach.

0000 0000 1010 0000 0000 0000 0000 1010 0000 0000 0000 0000 0000 0000 **0000 1010**

Assuming the memory address starts from 0.

Address (Decimal)	Data (Binary)	
0	**0000 1010**	Byte-7
1	0000 0000	Byte-6
2	0000 0000	Byte-5
3	0000 0000	Byte-4
4	0000 1010	Byte-3
5	0000 0000	Byte-2
6	1010 0000	Byte-1
7	0000 0000	Byte-0

2. Big Endian -

Most Significant Byte is stored at the lowest address.

Example: Storing the following 64-bit data in memory in big big-endian approach.

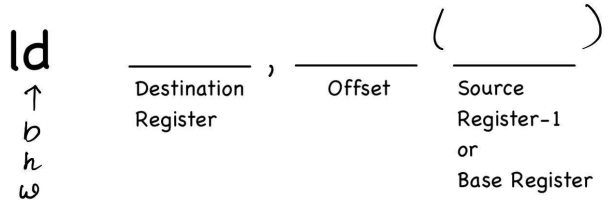
0000 0000 1010 0000 0000 0000 0000 1010 0000 0000 0000 0000 0000 0000 0000 1010
Assuming the memory address starts from 0.

Address (Decimal)	Data (Binary)	
0	**0000 0000**	Byte-0
1	1010 0000	Byte-1
2	0000 0000	Byte-2
3	0000 1010	Byte-3
4	0000 1010	Byte-4
5	0000 0000	Byte-5
6	1010 0000	Byte-6
7	0000 1010	Byte-7

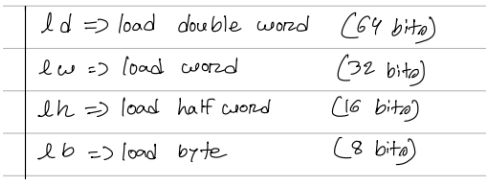
Load Instruction:

Loads data from memory to register.

Syntax:



Example:
ld x18, 10(x20)



Explanation:
ld stands for load double word (64 bits). This means 64 bits of data will be loaded from memory into the destination register.
Operands:
x20: Contains the Base address.
10 is the offset.

Destination:
x18: The data fetched from memory will be loaded into the x18 register.

Let's assume,
x20 has a value 10 inside it.

Data memory:

Location	Value
20	50
21	
22	
23	
24	
25	
26	
27	

Execution:
[for simplicity of calculation we are assuming all the values are in decimal]

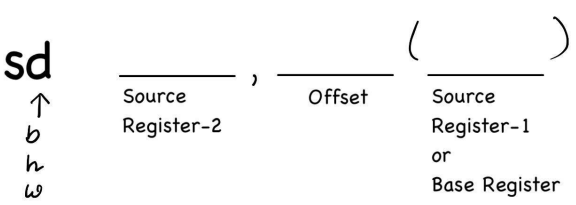
i. The processor calculates the effective memory address by adding the base address in x20 (10) and the offset (10), resulting in an effective address of 20.
ii. The processor then fetches the 64-bit data from the calculated address in the data memory, which is 20 in this case.
iii. In memory location 20, the stored data is 50.
iv. The fetched data 50 is then loaded into the destination register x18.

Outcome:
After execution, the contents of the registers will be:
x18 = 50 (loaded from memory).
x20 = 10 (unchanged).

Store Instruction:

Stores data from the register to memory.

Syntax:



Example:

sd x18, 10(x20)

Explanation:

sd stands for store double word (64 bits). This instruction stores 64 bits of data from a register into memory.

Operands:

x20: Contains the Base address.

x18: The source register containing the 64-bit data to be stored in memory.

10 is the offset.

Destination:

The memory location is calculated by adding the value of the base register and offset.

Let's assume,

x20 has a value 10 inside it.

Data memory:

Location	Value
20	50
21	
22	
23	
24	
25	
26	
27	

Execution:
[for simplicity of calculation we are assuming the values are stored in decimal]

i. The processor calculates the effective memory address by adding the base address in x20 and the offset (10):
Effective address = x20 + 10 = 10 + 10 = 20.
ii. It then stores the 64-bit value from **x18** into memory at the calculated address **20**.

Outcome:
After execution, both x18 and x20 will remain unchanged.

Example of compiling a statement when an operand is in Memory:

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];
```

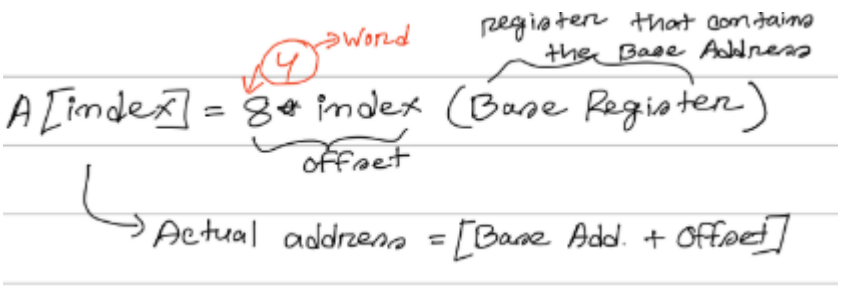
Memory Operand

```
ld  x5, 8x8(x22)
add x20, x21, x5
```

temp. reg. · off. · Base Reg.

g = x20
h = x21
Base of A = x22

Note:
i. A is an array of 100 doublewords meaning A has 100 indices and each size of doubleword (64 bits).
ii. The base address of the array is in x22 means the location of the starting index, A[0] is stored in register x22.



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)	h = x21
add x6, x21, x6	Base of A = x22
sd x6, 96(x22)	

Immediate Operand

Constant data specified in the instruction itself.

Addi Instruction:

This is an arithmetic instruction that performs integer addition.

Syntax: addi _____ , _____ _____

Destination Register | Source Register-1 | Constant or Immediate

Example:
addi x18, x19, 10
Explanation:

Operands:
x19: Contains the first operand, which is 12.
10: is the immediate or constant.

Destination:
x18: The result of the addition will be stored in this register.

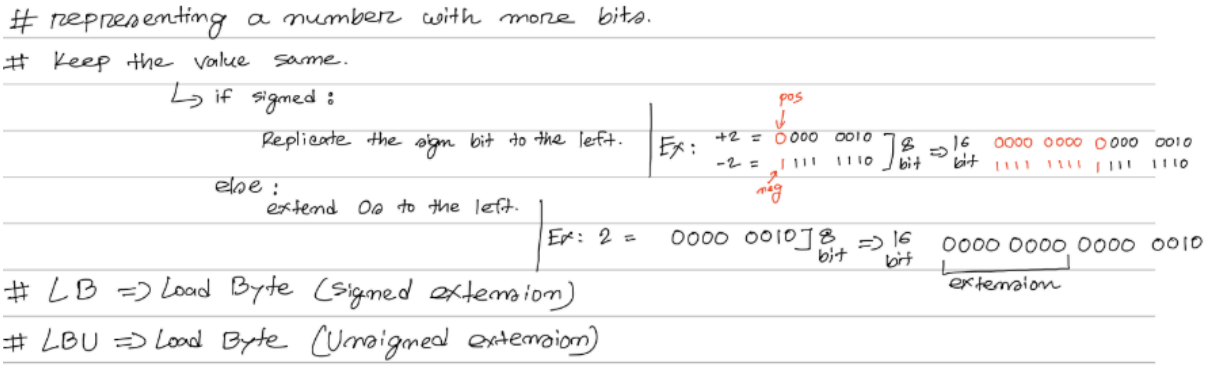
Execution:
i. The processor reads the value stored in x19 (12).

ii. It performs the addition: 12 + 10 = 22. (***val of source reg1 + constant***)
iii. The result, 22, is written into the destination register x18.

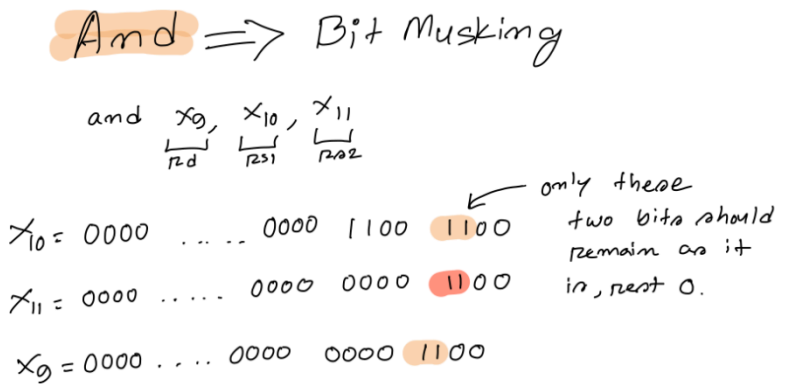
Outcome:
After execution, the contents of the registers will be:
x18 = 22
x19 = 12 (unchanged)

Note:
There is **no Subi** instruction.
Let's say, you want to subtract 4 from the value inside x22 register and store the result again in x22.
Addi x22, x22, -4

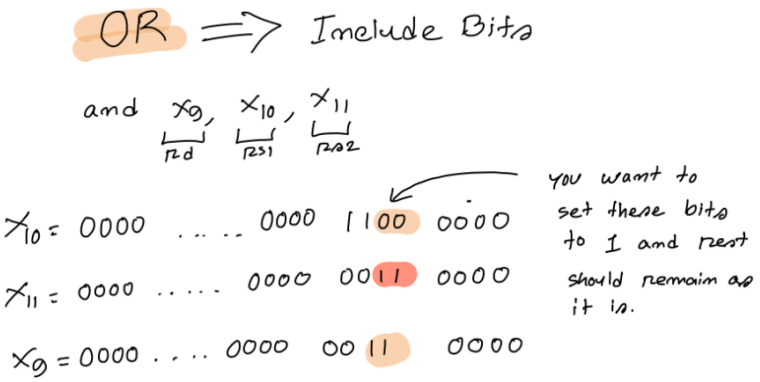
Sign Extension



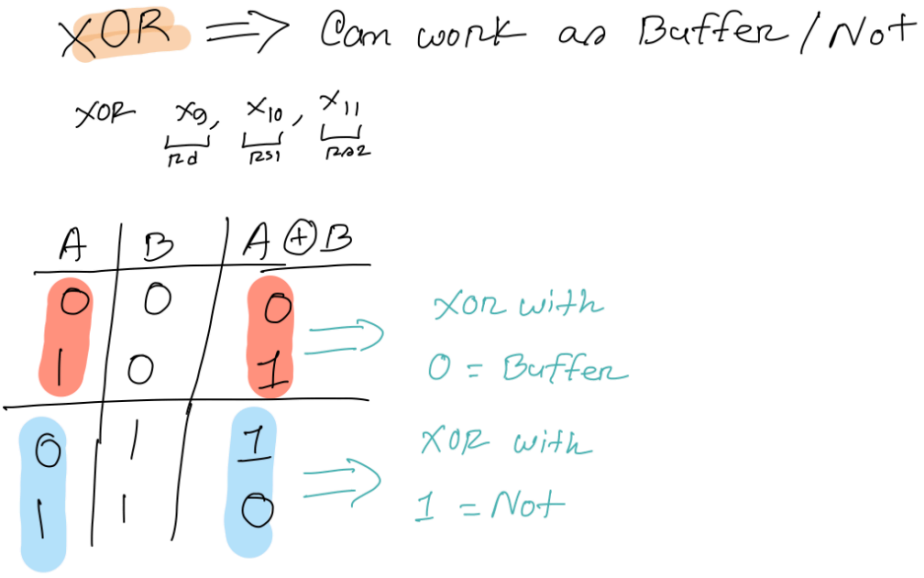
And Instruction



Or Instruction



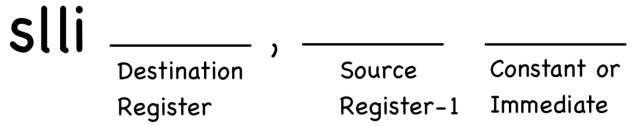
XOR Instruction



SLLi Instruction

Slli stands for Shift Left Logical Immediate. This instruction shifts the bits of the value in the source register to the left by a specified number of positions (immediate value), filling the vacant lower bits with zeros.

Syntax:



Example:
slli x18, x19, 2
Explanation:

Operands:
x19: Contains the first operand, which is 12.
2: Specifies the number of positions each bit will be shifted to the left.

Destination:
x18: The result of the shifting will be stored in this register.

Execution:
i. The processor reads the values stored in x19 (12) and immediate is 2.
ii. 12 in 64 bits = 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 1100
Shifting each bit to the left by 2 positions,
00 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0011 0000
The first two **00 will be lost;** which will create a vacancy in the last two bits. That vacancy will be filled up by two 0s.
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0011 0000

iii. The result,
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0011 0000
(48 in decimal) is written into the destination register x18.

Outcome:
After execution, the contents of the registers will be:
x18 = 48
x19 = 12 (unchanged)

If you shift a value that is stored in a register by more than 63 bits what happens?

We can also perform multiplication by 2^i using slli instruction.

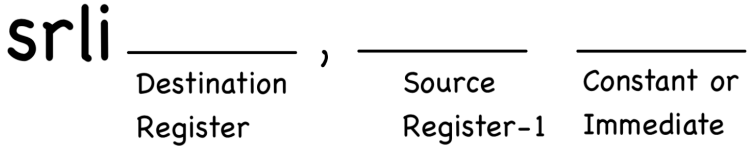
Slli x11, x19, 4

After executing this instruction, x11 will have, value inside x19 * 2^4

SRLi Instruction

Srli stands for Shift Right Logical Immediate. This instruction shifts the bits of the value in the source register to the right by a specified number of positions (immediate value), filling the vacant lower bits with zeros.

Syntax:



Example:
srli x18, x19, 2

Explanation:
Operands:
x19: Contains the first operand, which is 12.
2: Specifies the number of positions each bit will be shifted to the right.

Destination:
x18: The result of the shifting will be stored in this register.

Execution:
i. The processor reads the values stored in x19 (12) and immediate is 2.
ii. 12 in 64 bits = 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 1100
Shifting each bit to the right by 2 positions,
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 001100
The last two **0s will be lost;** which will create a vacancy in the first two bits. That vacancy will be filled up by two 0s.
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0011

iii. The result,
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0011
(3 in decimal) is written into the destination register x18.

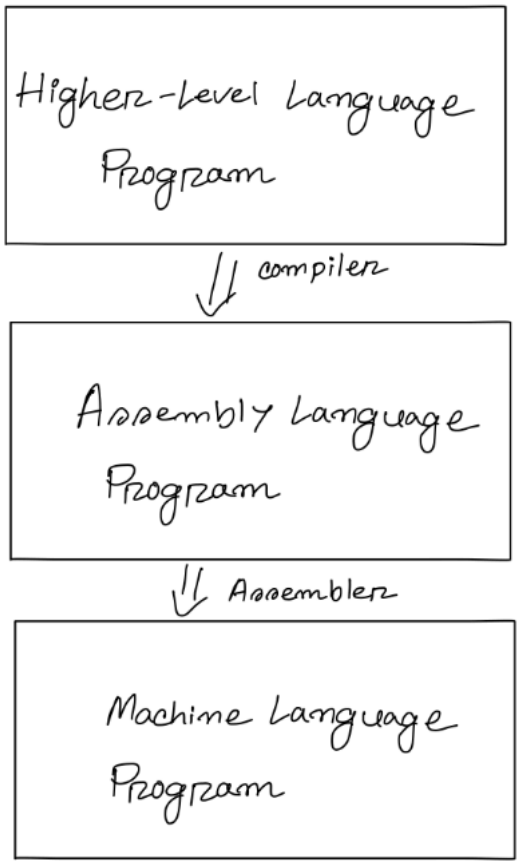
Outcome:
After execution, the contents of the registers will be:
x18 = 3
x19 = 12 (unchanged)

If you shift a value that is stored in a register by more than 63 bits what happens?

We can also perform division by 2^i using slli instruction.

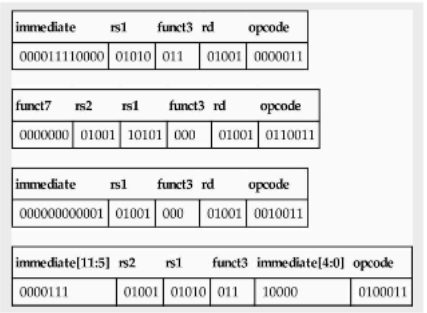
Srli x11, x19, 4
After executing this instruction, x11 will have, value inside x19 / 2^4

Translating RISC-V assembly instructions to machine code




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

```
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]
```



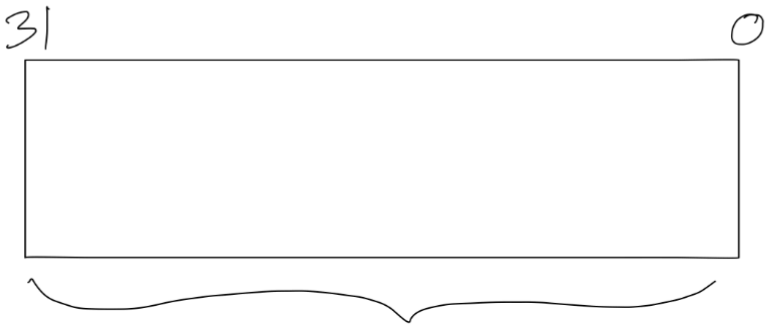
immediate	rs1	funct3	rd	opcode
000011110000	01010	011	01001	0000011

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 (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.

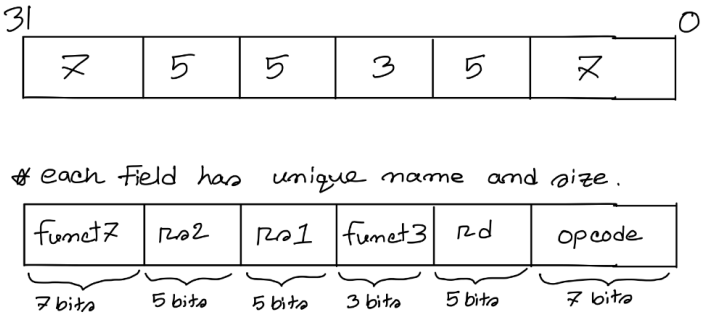
Design Principle 3: Good design demands good compromises.

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

Instruction Formats

1. R type: Instructions that use 3 registers. (Add, Sub, Sll, Srl, And, Or, Xor,.....)
2. I type: Instructions that use 2 registers (1 source, 1 destination) and 1 immediate. (Addi, Load, Slli, Srli, Andi, Ori, Xori,.....)
3. S type: Instructions that use 2 registers (2 sources) and 1 immediate. (Store)
4. U type: Instructions that use 1 register (1 destination) and 1 immediate (LUI)
5. SB type: Conditional jump instructions fall under this format
6. UJ type: Unconditional jump instructions fall under this format

R-type Instruction Format



Field	Bit Length	Description
opcode	7 bits	It determines if the instruction is of the R-type or not.
rs1	5 bits	Source Register 1 – Number of the source register-1.
rs2	5 bits	Source Register 2 – Number of the source register-2.
funct3	3 bits	both of these two fields are used to specify the exact operation (add, sub,...) within the R-type format.
funct7	7 bits	
rd	5 bits	Destination Register – Number of the destination register.

Example:
Add x21, x22, x23; convert it to its corresponding machine code where, Opcode = 0110011, Funct3 = 000 and Funct7 = 0000000. Show your final answer in hex format.
Solution:

Rs1 = 22 = 10110
Rs2 = 23 = 10111
Rd = 21 = 10101
Opcode = 0110011
Funct3 = 000
Funct7 = 0000000

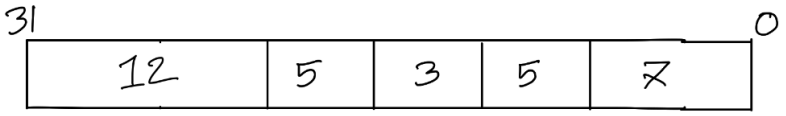
Forming the 32-bit machine code now,

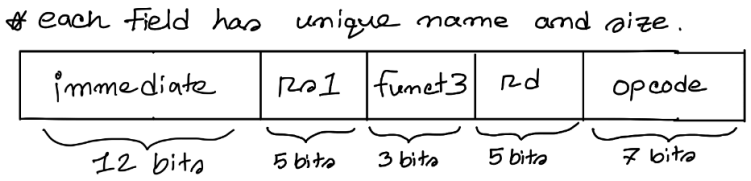
0000000	10111	10110	000	10101	0110011
funct7	rs2	rs1	funct3	rd	opcode

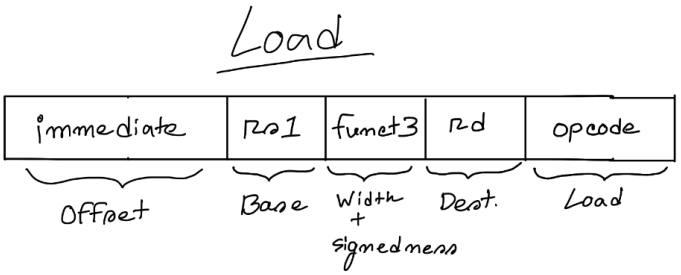
$(00000001011110110000101010110011)_2 = 17B0AB3_{16}$

Now, given a machine code think, how can you decode and find the corresponding assembly code?

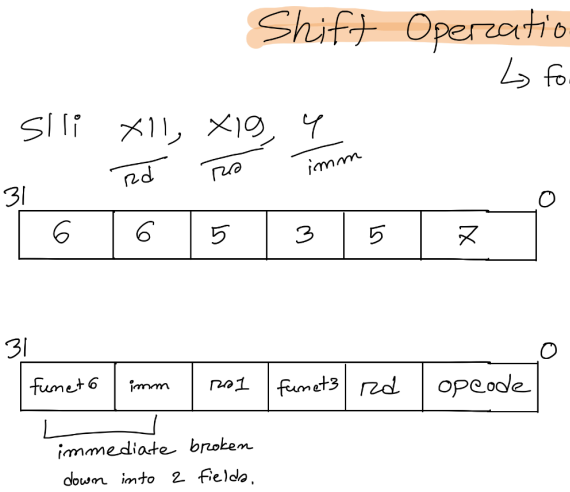
I-type Instruction Format







Field	Bit Length	Description
opcode	7 bits	It determines both if the instruction is of the I-type or not and the specific instruction (addi, load,...) itself.
rs1	5 bits	Source Register 1 – Number of the source register-1.
funct3	3 bits	In the case of load instruction specifies the size and signedness of the data being loaded.
immediate	12 bits	the constant itself in 2s complement mode
rd	5 bits	Destination Register – Number of the destination register.



To answer why shift follows a modified version of I type, remember what would happen if you shift a 64-bit value more than 63 times?
Now, check how many bits are necessary to represent 63 in binary.

Example1:
Addi x21, x22, -4; convert it to its corresponding machine code where, Opcode = 0010011, Funct3 = 000. Show your final answer in hex format.
Solution:

Rs1 = 22 = 10110
Rd = 21 = 10101
Opcode = 0010011
Funct3 = 000
Immediate = -4
4 = 100
+4 = 0100
= 0000 0000 0100
-4 = 1111 1111 1100 (2s complement)

Forming the 32-bit machine code now,

1111 1111 1100	10110	000	10101	0010011
immediate	rs1	funct3	rd	opcode

$(11111111110010110000101010010011)_2 = FFCB0A93_{16}$

Now, try to do the reverse yourself.

Example2:
Ld x21, 10(x22); convert it to its corresponding machine code where, Opcode = 0000011, Funct3 = 010. Show your final answer in hex format.
Solution:

Rs1 = 22 = 10110
Rd = 21 = 10101
Opcode = 0000011
Funct3 = 010
Immediate = +10
10 = 1010
+10 = 01010
= 0000 0000 1010 (2s complement)

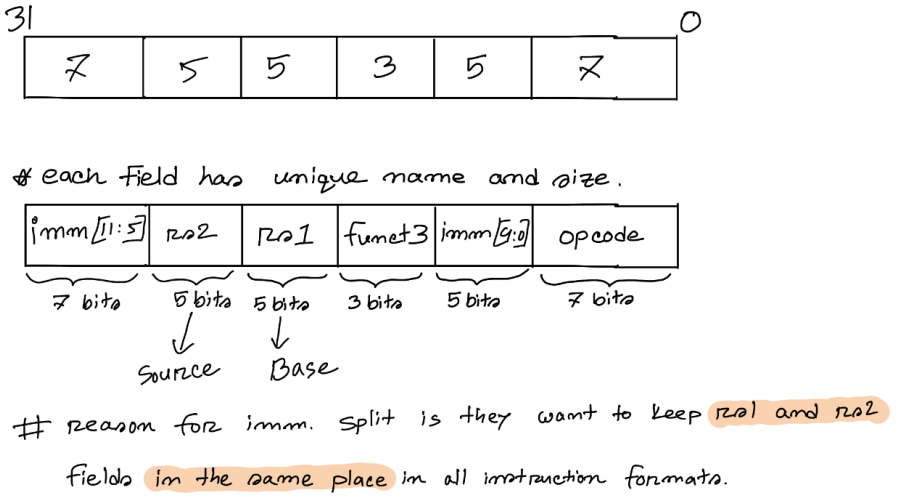
Forming the 32-bit machine code now,

0000 0000 1010	10110	010	10101	0000011
immediate	rs1	funct3	rd	opcode

$(00000000101010110010101010000011)_2 = 00AB2A833_{16}$

Now, try to do the reverse yourself.

S-type Instruction Format



Field	Bit Length	Description
opcode	7 bits	It determines both if the instruction is of the S-type or not and the specific instruction (addi, load,...) itself.
rs1	5 bits	Source Register 1 – Number of the source register-1.
rs2	5 bits	Source Register 2 – Number of the source register-2.
funct3	3 bits	In the case of load instruction specifies the size and signedness of the data being loaded.
immediate	12 bits	the constant itself in 2s complement mode

Example:

Sd x21, 10(x22); convert it to its corresponding machine code where, Opcode = 0100011, Funct3 = 010. Show your final answer in hex format.

Solution:

Rs1 = 22 = 10110

Rs2 = 21 = 10101

Opcode = 0100011

Funct3 = 010

Immediate = +10

10 = 1010

+10 = 01010

= 0000000 01010 (2s complement)

Forming the 32-bit machine code now,

0000000	10101	10110	010	01010	0100011
imm[11:5]	rs2	rs1	funct3	immd[4:0]	opcode

$(00000001010110110010010100100011)_2 = 015B2523_{16}$

Now, try to do the reverse yourself.

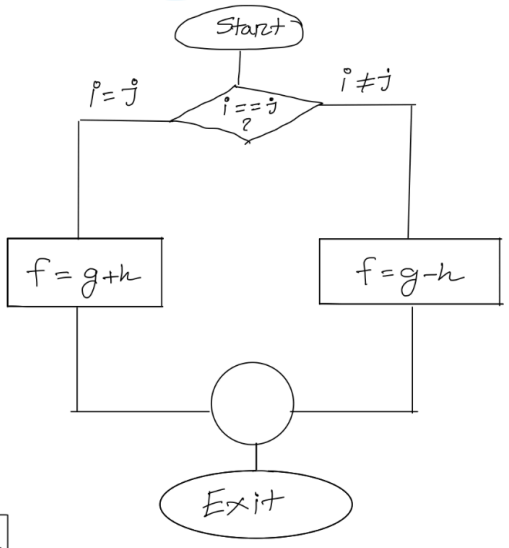
Decision Making

* It is commonly represented in programming languages using the
  (i) If statement
  (ii) goto statements (label)

Given Code

```
If (i == j):
    f = g+h
else:
    f = g-h
```

Flow Chart:



RISC-V assembly code:

```
    bne  X22, X23, Else
    add  X10, X20, X21
    beq  X0, X0, Exit      (Unconditional Branch)

Else:
    sub  X10, X20, X21
Exit:
```

f = X19
g = X20
h = X21
i = X22
j = X23

Conditional Jumps:

	Instruction	Syntax	operation
=	beq	beq reg1, reg2, L1	reg1 == reg2
!=	bne	bne reg1, reg2, L2	reg1 != reg2
<	blt	blt reg1, reg2, L3	reg1 < reg2
>=	bge	bge reg1, reg2, L4	reg1 >= reg2

Loop

```
while (save[i] == k)
    i = i+1
```

Assume that i and k correspond to registers x22 and x24 and the base of the word array save is in x25. What is the RISC-V assembly code corresponding to this
C code?

Solution:

Loop:	// We need to add the label **Loop** to it so that we can branch back to that instruction at the end of the loop:
slli x10, x22, 2	// x10 = i * 4
add x10, x10, x25	To get the address of save[i], we need to add x10 and the base of save in x25: // x10 = address of save[i]
lw x9, 0(x10)	Now we can use that address to load save[i] into a temporary register: // x9 = save[i]
bne x9, x24, Exit	The next instruction performs the loop test, exiting if save[i] ≠ k: // go to Exit if save[i] ≠ k
addi x22, x22, 1	// i = i + 1
beq x0, x0, Loop	// go to Loop
Exit:	

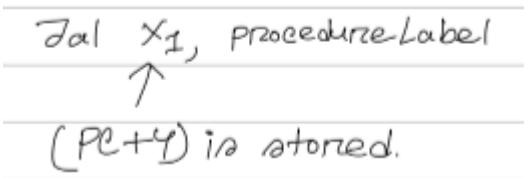
Try the same for a **for** loop yourself.

Program Counter

PC or Program Counter is the register that holds the address of the current instruction being executed.

Unconditional Jump

Jal instruction:



// jump to ProcedureAddress and write return address to x1

Jal stands for jump and link;
It does not check any condition before jumping like beq, bne, bge, blt.
The link portion means that an address or link is formed that points to the calling site to allow the procedure to return to the proper address.
This "link," stored in register x1, is called the return address.

Jalr instruction:

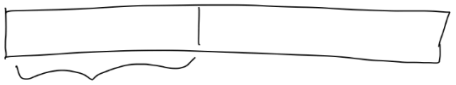
Jalr x0, 0(x1)

Jalr stands for jump and link register;
The jalr instruction branches to the address stored in register x1

LUI instruction

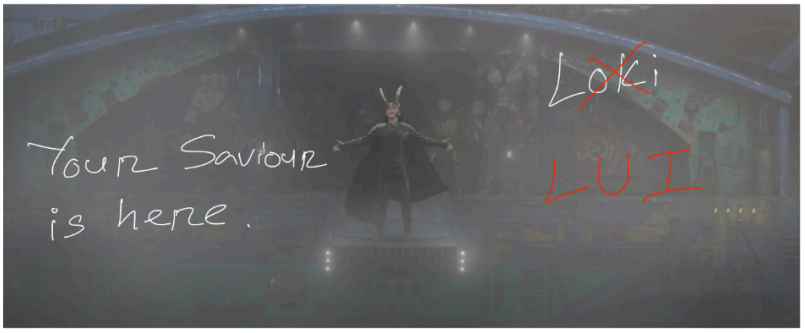
Try this ⇒ X_{10} = 1111 1111 1111 0000

Addi X_{10}, X_{10}, 65520

I type = 

imm = 12 bits ??

then how do we do this?



LUI = Load Upper Immediate — use it to form 32 bit immediates

Syntax = LUI rd, constant → 20 bits

- copies the 20 bit data into rd's [31:12]
- copy whatever you have in bit 31 to bits [63:32]
- copy 0s in [11:0] of rd

load this 64 bit value into X_{10} ⇒

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0011 1101 0000 0101 0000 0000

31 12 1280

976

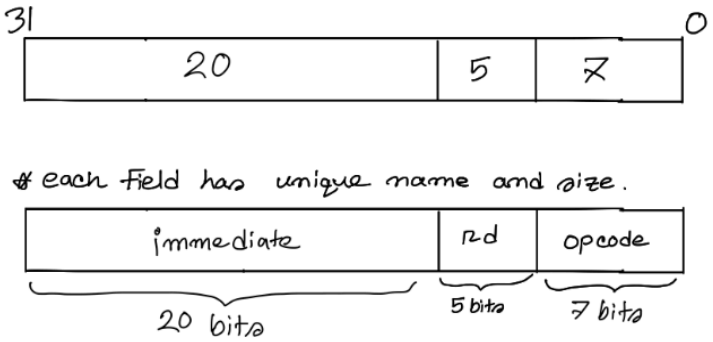
lui X_{10}, 976

X_{10} = 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0011 1101 0000 0000 0000 0000

addi X_{10}, X_{10}, 1280

X_{10} = 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0011 1101 0000 0101 0000 0000

U-type Instruction Format



Field	Bit Length	Description
opcode	7 bits	It determines both if the instruction is of the U-type or not and the specific instruction (LUI) itself.
rd	5 bits	Destination Register – Number of the destination register.
immediate	20 bits	the constant itself in 2s complement mode

Example:
LUI x21, 10; convert it to its corresponding machine code where Opcode = 0110111. Show your final answer in hex format.

Solution:

Rd = 21 = 10101
Opcode = 0100011
Immediate = +10
10 = 1010
+10 = 0 1010
= 0000 0000 0000
0000 1010

Forming the 32-bit machine code now,

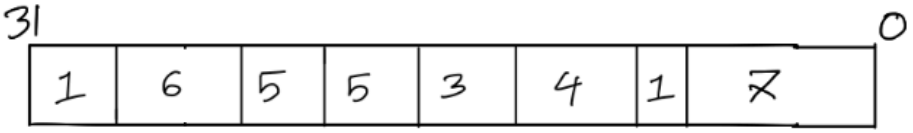
0000 0000 0000 0000 1010	10101	0110111
imm[19:0]	rd	opcode

(2s complement)

$(00000000000000001010101010110111)_2 = 0000AAB7_{16}$

Now, try to do the reverse yourself.

SB-type Instruction Format



* each field has unique name and size.

imm [12]	imm [10:5]	rs2	rs1	funct3	imm [4:1]	imm [11]	opcode

* represents branch addresses in multiples of 2.
* forward & backward moving.
* the unusual encoding of imm. simplifies datapath design.

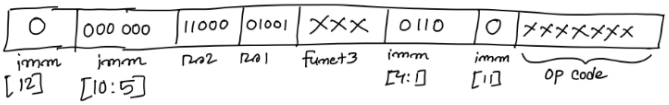
```
Loop:
#80000  slli  X10, X22, 3      ---- line 1
#80004  add   X10, X10, X25    ---- line 2
#80008  ld    X9,  0(X10)      ---- line 3
#80012  bne   X9,  X24, Exit   ---- line 4
#80016  addi  X22, X22, 1      ---- line 5
#80020  beq   X0,  X0,  loop   ---- line 6
Exit:
#80024                         ---- line 7
```

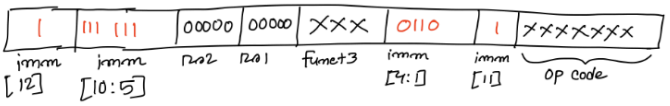
3 × 4 = 12

0000 0000 1100

0 0000 0000 1100 — Discard it.

12 | 11 | 10:5 | 4:1





5 × 4 = 20 but its going upward so −20.

= 0000 0001 0100

= 0 0000 0001 0100

= 1 1111 1110 1011

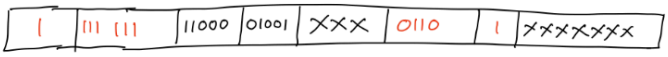
+ 1

1 1111 1110 1100 ⇒ −20 in 2s comp.

12 | 11 | 10:5 | 4:1 | 0

PC relative addressing

= PC + immediate × 2



(i) Form the 12 bit number

11 111 111 0110

(ii) Detect if positive or negative number.

if neg perform 2s comp. again,

1111 1111 0110

0000 0000 1001

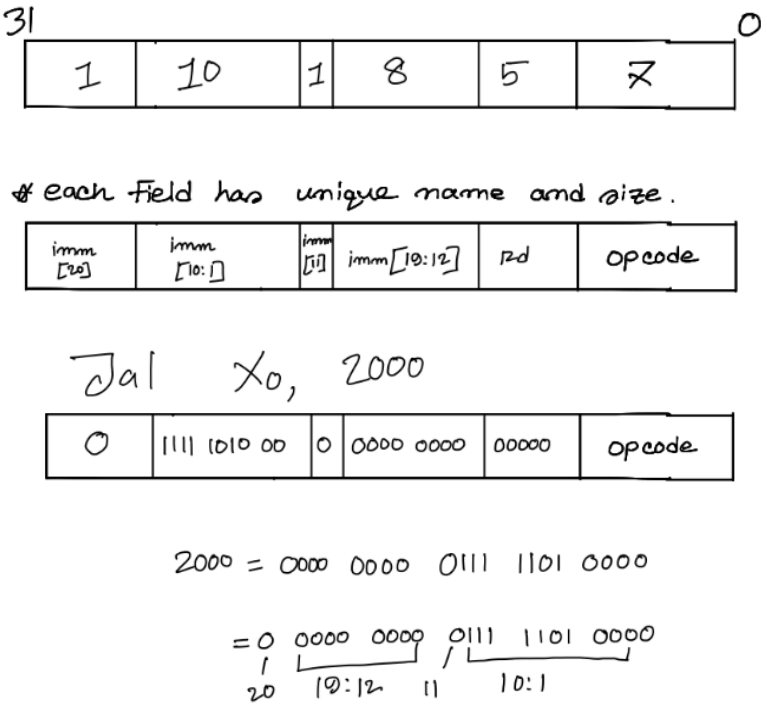
+ 1

0000 0000 1010 ⇒ 10

(iii) PC (±) imm × 2 → offset

Based on the sign bit

UJ-type Instruction Format



* uses 20 bit immediates for larger jumps.
* For more large jump, (32 bit)
  lui: load address [31:12] to a temp reg.
  jalr: add address [11:0] and jump to target.

PC relative addressing
= PC + immediate x2

Given a branch on register `x10` being equal to zero,

```
beq x10, x0, L1
```

replace it by a pair of instructions that offers a much greater branching distance.

Answer

These instructions replace the short-address conditional branch:

```
 bne x10, x0, L2
 jal x0, L1
L2:
```