Chapter 2

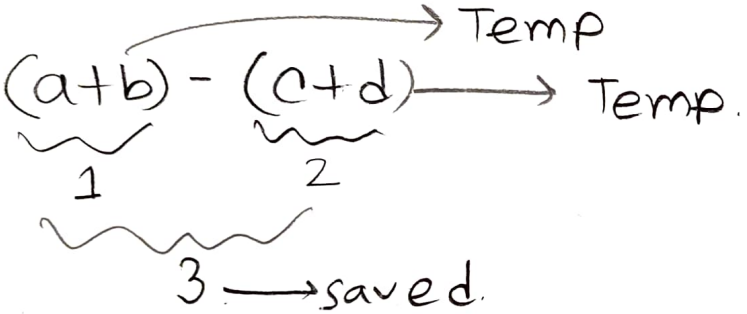
Register → units of memory immidiate use of CPU

~~32 bits~~ 32 X 64 bit.

smallen is faster.

x_0 → holds 0. Canbe used to copy things.

x_1 → Return addren.



x_{10-11} → Function arguments / Return

x_{12-17} → " "

x_{5-7} → Temp.

x_9 → Saved

x_{18-27} → "

x_{28-31} → Temp.

operand

- Register operands
- memory operands
- Constant / immediates

↳ addi x_{28}, x_{29}, 5
↙
Hard code করে দিচ্ছি.

Register: Arithmatic instruction use register operands.

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

$f \rightarrow x_{19}$

$j \Rightarrow x_{23}$.

add x_5, x_{20}, x_{21}
add x_6, x_{22}, x_{23}
sub x_{19}, x_5, x_6.

Memory operand

Risc-V is little endian.

memory is byte addressed.

8 x index = offset

∴ offset + base.

A[index] = 8 * index

↑ Actual Address = [Base Add + offset]

A[3]

⇒ 8 x 3

= 24 + 0000

= 24.

Memory operand.

words must start at address multiple of 4

double " " " " " " " " 8.

32 bit ⟶ #0004 ✓

#0003 ✗

64 bit ⟶ #0008 ✓

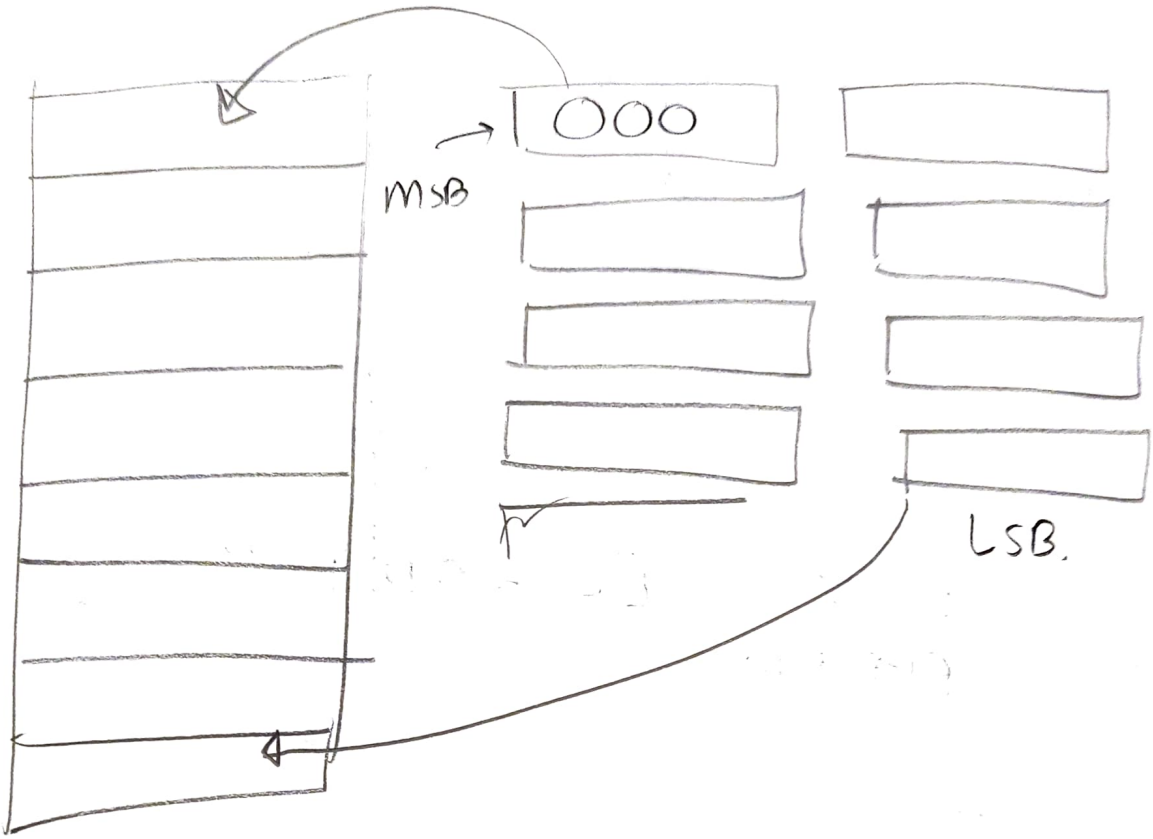
#0009 ✗

Risc-V → No alignment restriction

Risc-V is little endian

The order in which memory stores data.

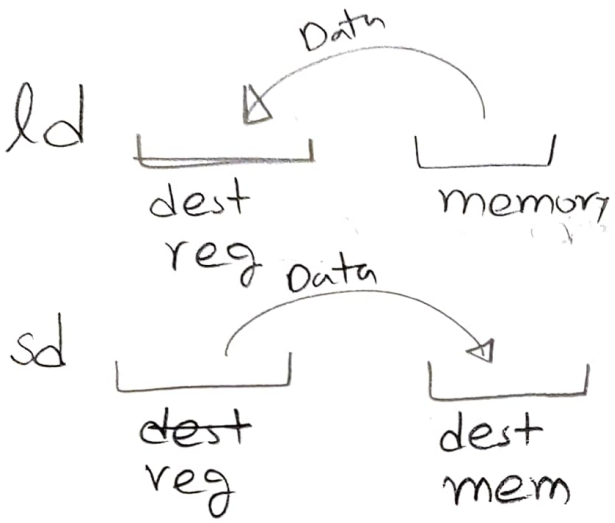
Big endian



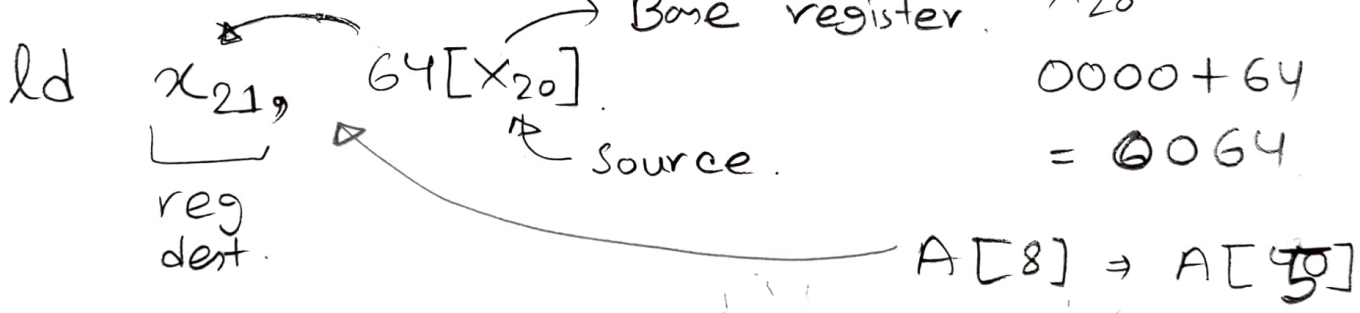
Why little endian

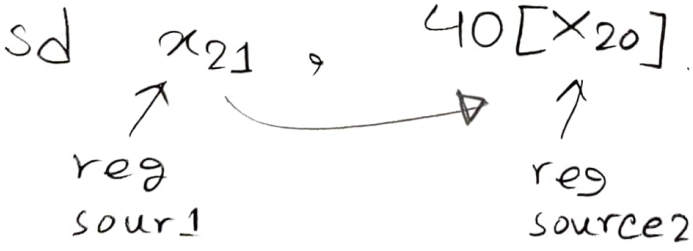
load, store Instruction syntax.

load → load data from mem to reg
store → store data from ~~me~~ to mem from reg.



ld ⇒ load double (64)
lw → load word (32)
lh → load half (16)
lb → load byte (8)





5:50

Array → 100 double words

→ Temp. ← off ← Base reg

ld X_5, 64[X_{22}]

add X_{20}, X_{21}, X_5

$g = X_{20}$

$h = X_{21}$

Base of A = X_{22}

g = h + A[8]

memory location

offset = 8 × 8 = 64

Assume var h is associated with X_{21}

base address of array A is X_{22}.

RISC-V assembly code for,

A[12] = h + A[8]

12 × 8 = 96.

ld X_5, 64[X_{22}]

add X_5, X_{21}, X_5

sd X_5, 96[X_{22}]

Immediate Operation.

↳ Avoids a load instruction (faster)

Addi $X_{22}, X_{22}, 1$.

Compiler cannot add/sub 2 integ.

Addi $X_{22}, X_{21}, 2$ ⊗.

$X_{22} = X_{22} - 5$.

↳ Addi $X_{22}, X_{22}, -5$.

Immediate ope. 2.

↳ avoids load instructions.

addi $x_{22}, x_{21}, 5$.

Sign extension

LB → sign extension.

LBU → load byte (unsigned extension)

if signed:
replicate the sign bits to the left

else:
extend 0s to the left (←)

$(2)_{10} = (10)_2$

$(2)_{10}$ = in unsigned 8 bit.

$(0000\ 0010)_2$.

0000 0010

LBU X_{22}, 10[X_{21}]

24 zero. (0000 0010)

$(+2)_{10} = (010)_2$.

⇒ 0000 0010 (copied sign bit)

$(-2)_{10}$ ⇒ (2's compliment of +2.

+2 ⇒ 0000 0010.

1's c - 1111 1101

+1 (2's c)

$(-2)_{10} = (1111\ 1110)_2$

Unsigned = 0 to $2^n - 1$

signed = -2^{n-1} to $+2^{n-1} - 1$ [2's comp]

Desire to keep All instructions "length same" vs "format same"

RiscV → Machine code

Higher
↓ Compiler
Assembly
↓ Assembler
Machine Language

RiscV → instructions take ex. 32 bit.

The layout of ins is called "Instruction format"

divide 32 bits into fields.

length.
vs
format

	rd	rs1	rs2
Add	X_{20},	X_{20},	X_{21}
Addi	X_{20},	X_{20},	5 (imm)
ld	X_{20}, (rd)	16 (off)	$[X_{21}]$ (rs)
sd	X_{20}, (rs)	16 (off. imm)	$[X_{21}]$ (rs.)

→ Same Format.

RiscV chooses to keep the length same. But different format.

755 357

R-type instructions

1 → rdest
2 → rsour

Instruction formats exactly

- → R ⇒ Ins that use 3 registers (Add, sub, sll, XOR, OR, AND)
- → I ⇒
- → S

R-type

funct7	rs2	rs1	func3	rd	opcode	
7b	5b	5b	3b	5b	7b	= 32 bits

opcode = Denote the format of an instruction

opcode, func3, func7 ⇒ কোনো ধরনের কাজ কোন type আর কোন instruction (Rtype ADD)

rd = destination

rs1 = source 1

rs2 = source 2

funct7	rs2	rs1	fun3	rd	OP
0000 000	10101 $(21)_2$	10100 $(20)_2$	000	10100 $(20)_2$	Given 1101010

Add X20, X20, X21

X20 = rd, X20 = rs1, X21 = rs2

শুধু reg number not content

Convert to Hex (Boundary ones)

0	1	5	A
0000	0001	0101	1010
0000	1010	0110	1010
0	A	6	A

Hex: 015A0A6A

I-type

→ 1 immidiate value (offset)

→ must be two reg involved (rd, rs)

imm.	rs1	fun3	rd	opcode
12b	5b	3b	5b	7b

Immediate = constant/offset (2s comp)

rs1 = ~~load~~ source / Base Reg number

I-type { Addi X_{22}, X_{22}, 5 (rs1, imm(con))
ld X_{22}, 16[X_{21}] (rs1, 16 → immed. (offset))

Why is rs/rd = 5 bit?

→ Because we just put the register numbers. we have 32 registers. So, we need 5 bits (2^5)

[~~rs~~ load এর ক্ষেত্রে func3 → width ✗ signedness
কত সাইজের ডেটা
sign status (lb/lbu)]

Addi X_{22}, X_{22}, 5

16 8 4 2 1
1 0 1 1 0

0000 0000 0101	10110	101	10110	___
	rs		rd.	ope

(suppose)

→ Hex.

S-type ins. 755357.

How to distinguish.

→ store instruction (sd)

SB → byte
SH → Half
SW → word
SD → double

কত সাইজের Data? Signed নাকি unsigned?

imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode
7b	5b	5b	3b	5b	7b.
	source	Base.			

Why is imm val. divided in 7 and 5?

To make the instruction formats more common. Like with R-type and I-type. This is totally match with R-type.

ex LD X_5, 16[X_{20}]
rd mem Base rs offset

SD X_5, 16[X_{20}]
rs2(1) rs2(2) immediate
2 source.

SD X_5, -16[X_{20}]
rs2 rs1 imm.

16	8	4	2	1
1	0	1	0	0
1	0	0	0	0

11 10 9 8 7 6 5 4 3 2 1 0
0000 000 10000

0000 000	00101	10100	xxx	10000	xxxx xxx
7 (im 11:5)	5 rs2	5 rs1	3 func	5 (im 4-0)	7 (op)

Shift operation.

lost ← 1010 1111

1 বার left shift → 0101 1110 → 1 time left shift

2 time left shift. → 101 11100 → 2 time left shift on prev number.

101 11100 → Right shift

0101 1110.

Rig	left shift
SRR	SLL → left shift logical
SRRI	SLLi

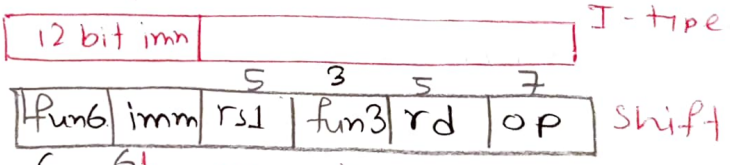
I-type instruction ← SLLi $X_{19}, X_{20}, 4$ (rd rs1 immediate)

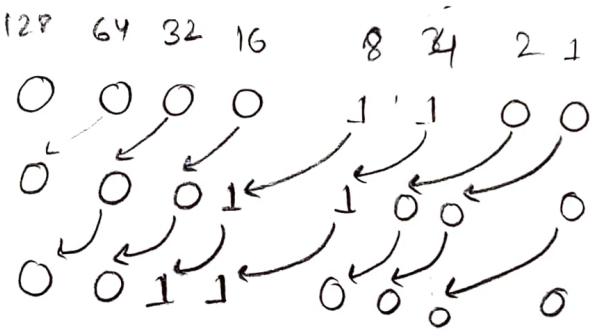
↳ X_{20} এর value কে 4 বার left shift করে X_{19} এ

what'll happen if we shift 64 bits? বলবে.

→ সম্পূর্ণ 0 হয়ে যাবে। so, meaningless.

so cap is **1 - 63**





Right shift

2^i দিয়ে ভাগ.

128 64 32 16 8 4 2 1

0 0 1 1 0 0 0 0 ⇒ 48

0 0 0 1 1 0 0 0 ⇒ 24. srri 1

0 0 0 0 1 1 0 0 ⇒ 12. srri 2

if we right shift odd number, we take the floor value. $5 // 2^1 \Rightarrow 2$

SLLi performs multiplication using 2^i → যে কয়বার shift হচ্ছে

what about (12×5)

$(12 \times 2^2) + 12$

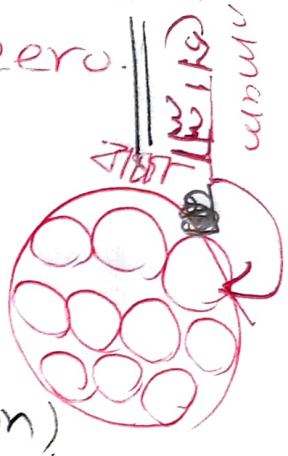
12×7

$(12 \times 2^3) - 12$

~~Right shift~~

AND → Bit Masking → select some bits and clear others to zero.

something and 0 ⇒ 0.
something and 1 ⇒ something.



and x_9, x_{10}, x_{11} (R-type instruction)
rd rs1 rs2.

যে পজিশনে bit gula intact রাখতে চাই সেখানে 1 দিয়ে and করব

		(7-4)	(2-1)
		7 6 5 4	3 2 1 0
x_{10} =	000 0 0000	1100	1100
x_{11} =	0000 0000	1111	0110 → Mask
x_9 =	0000 0000	1100	0100

bge rs1, rs2, L1

bge rs1, rs2, L1

OR & XOR

OR → Include bit.

or X_9, X_{10}, X_{11}.

A	B	A+B
0	0	0
1	0	1
0	1	1
1	1	1

11 থাকলে some or 0 = som
some or 1 = 1

X_{10} = 0000 ..... 0000 1100 0000

X_{11} = 0000 ..... 0000 0011 0000

X_9 = 0000 ..... 0000 1111 0000

XOR ⇒ Can work as Buffer / Not

in	out
0	0
1	1

in	out
1	0
0	1

XOR X_9, X_{10}, X_{11}.

A	B	A⊕B
0	0	0
1	0	1
0	1	1
1	1	0

something XOR 0 → Copy (Buffer)

something XOR 1 → Not

Practice

A[3] = A[5] + 1

~~addi 24[x_{21}], 40[x_{21}], 1~~

Base → x_{21}

A[0] = x_{21}

A is double word array

```
ld x5, 40[x21]
addi x5, x5, 1
sd x5, 24[x21]
```

if A[g] → variable is associated with x_{25}

```
slli x5, x25, 3
```

① index multiplied with num of memory cells.

$x_{25} \times 2^3$

```
ld x6, x5[x21]
```

x wrong → now offset

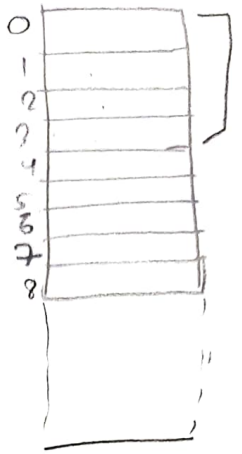
```
add x6, x21, x5   (Base + offset)
ld x7, 0[x6]      x6 is the actual address
```

Temp → 5, 6, 7

28, 29, 30, 31

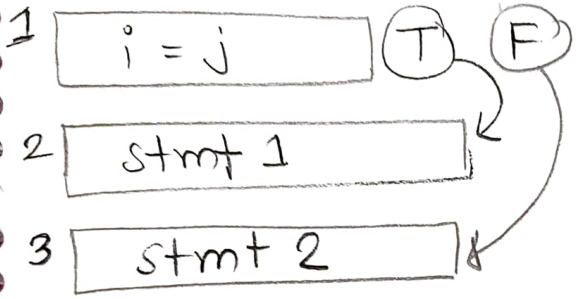
Decision making (if-else)

```
if (i == j):
        st - 1
else
        st - 1
```



8 × 4 = 32 bit.

assembler will always run sequentially



Risc V includes two de. making

① beq → branch if equal.

Cond. branching. beq, rs1, rs2, L1 ← Level 1 এ jump করবে

equal হলে

② bne (bran. not equal)

bne rs1, rs2, L1 (same as other)

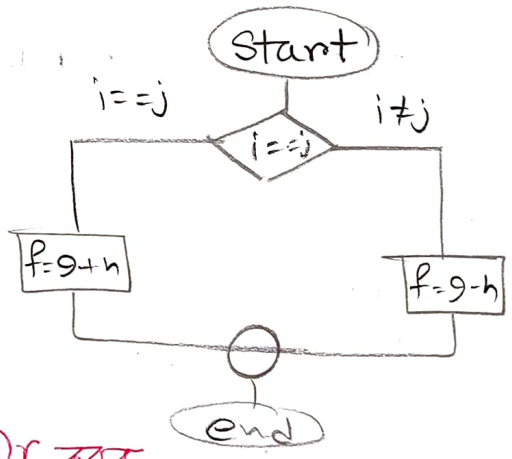
Cond. jumps

	Inst.	Syntax	Operations
=	beq	beq, rs1, rs2, L1	rs1 == rs2
!=	bne	bne, rs1, rs2, L2	rs1 != rs2
<	blt	blt, rs1, rs2, L3	rs1 < rs2
>=	bge	bge, rs1, rs2, L4	rs1 >= rs2

blt = branch less than
bge = branch greater than equal to

example

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



সবসময় bne করার চেষ্টা করব.

```
bne  X22, X23, Else
add  X19, X20, X21.
beq  X0,  X0,  exit
Else:
  sub X19, X20, X21
Exit:
```

> Constructed a unconditional Jump.

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

my bne?

if (i==j):
    f=gth.

```
beq  X22, X23, IF
beq  X0,  X0,  Exit
IF:
add X19, X20, X21.
Exit:
```

```
bne  X22, X23, Exit
add  X19, X20, X21

Exit:
:
```

Procedure calling

let's assume procedure like a spy.

Execution procen of spy:

(I) leaves with a secret plan.
(II) acquires resources (night vision)
(III) performs the task. (news নিয়ে আসবে)
(IV) covers traces. (খুন খারাবি)
(V) return to the point of origin with desired answer.

```
def calc (a,b,c):
```
Parameter

```
d = calc (1, 2, 3)
```
Arg

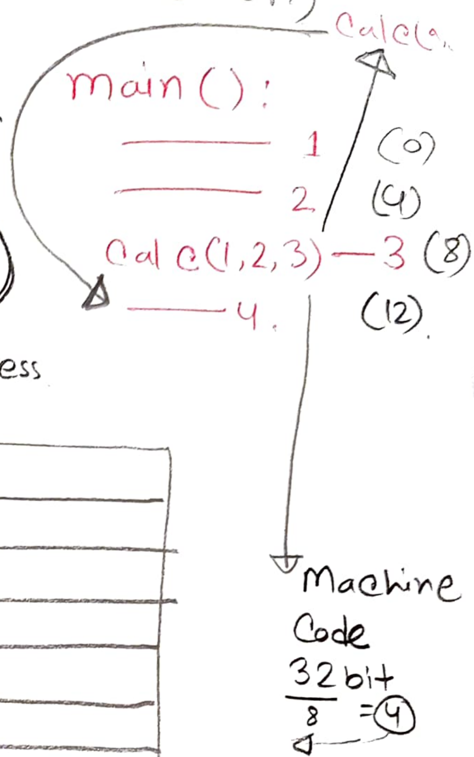
Ex of procedure

(I) Put param. where procedure can accen. (X_{10} - X_{17})
(II) Transfer cont to procedure
(III) resource for proc.
(IV) Perf the desired task
(V) result value in a place so that the call program can accen
(VI) Return control to the point of origin.

② JAL (Jump and Link instruction)

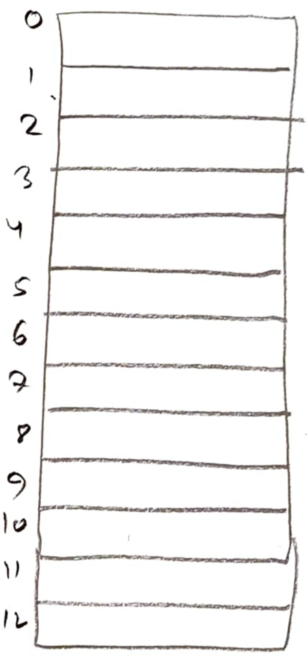
JAL, x_1, procedure label.

↑ Fixed. (কোথায় back করব সেটা থাকে এখানে) return address



PC (Program counter)
↳ current instruction এর memory location.

PC = 0. ← First instruction running.
PC = 4 ← second "
PC = 8 ← third "
PC = 12 ← fourth "



JAL (X1) Label.
PC = #0004. ← when executing
→ PC+4 = 0008 (কারণ 0004 executed, পরের line এ চলো)

JALR = Jump and link register. → function to main method.

JALR, x_0, 0[x_1]. → আমার বাসা থেকে আমার বাসার distance
(Hard) ↑ PC+4

why x_0 → function call থেকে main জায়গায় যাচ্ছি. so কোন লাইন এ PC+4, এর দরকার নাই

Caller ⇒ main() method

Callee ⇒ অন্য function (calc).

JAL X_0, Label ⟵ unconditional Jump.

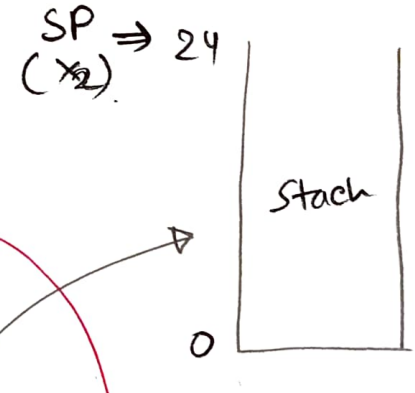
main(): ⇒ X_9, X_{26}, → 10 store করছে main.

Calc(): ⇒ X_9 চাচ্ছে, এখন কি হবে?

$X_9 = 10$ 0.

main method যেসব reg use করছে, এখন কোনো reg যদি callee use করে, তাহলে callee কে সেই register এর value আবার ফিরিয়ে আনার জন্য কোথাও save করে দেওয়া লাগবে.

↳ store করব stack এ.



each reg 64 bit.
= 8 location.

∴ 3 reg = 24 location.

X_2 = stack pointer.

we want to store 3 register

① decereare the stack pointer by the # locations you need.

eg. addi X_2, X_2, -24

⑪ Then store the values in the stack.

Forming a 32 bit immediate using LUI

16 bits.

X_{19} = 1111 1111 1111 000. ⇒ convert to decimal.

Addi X_{19}, X_0, 65520.

↑ itype instruction. → immediate = 12 bit.

Big ekhane 16 bit immediate thakle

what do we do?

Our savior LUI → Load upper immediate.

- Copies 20 bit data into rd's [31:12]. ①
- Copy whatever you have in bit 31 to bits [63:32]
- Copy 0s from 11th to 2eroth bit

LUI rd, Constant → 20 bit

load this 64 bit value into X_{19}

0000 0000 0000 0000 0000 0000 0000 0000

0000 0000 0011 1101 0000 0101 0000 0000

31 — 976 — 12 ; 1280

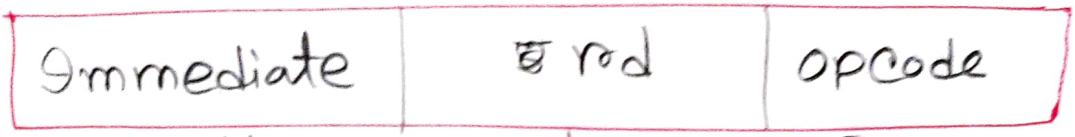
lui X_{19}, 976.

X_{19} = 0 … 0 | Copy | 0 … 0

63 … 32 | 31 … 12 | 11 … 0

addi X_{19}, X_{19}, 1280

LUI → U type instruction

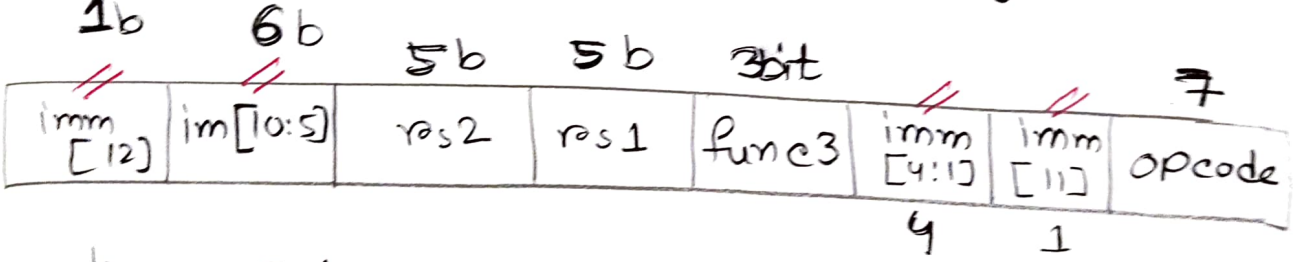


SB-type

SB type → beq, bne, blt, bge

↳ Conditional jump

if a==j, if a != b



beq rs1, rs2, label.

*

* Rep. branch address in multiples of 2.

* forward and backward moving

* unusual mapping of imm. design

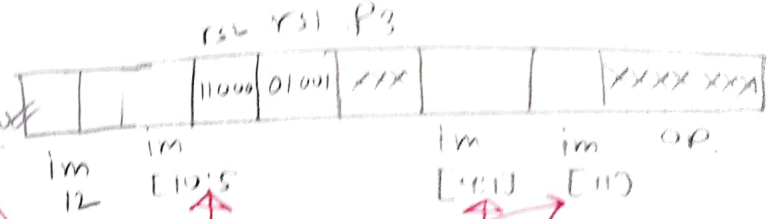
bne x_9, x_{24}, exit

exit

Ex:

8000	1	Slli x_{10}, x_{22}, 3
8004	2	add x_{10}, x_{10}, x_{25}
8008	3	ld x_9, 0[x_{10}]
8012	4	bne x_9, x_{24}, Exit
8016	5	addi x_{22}, x_{22}, -1
8020	6	beq x_0, x_0, loop
8024	7	Exit:

line 4 to line 7



PC relative addressing → Base থেকে কতদূরে jump করতে

4 থেকে exit e jump দিতে

so, line 4 to line 7 (3টা line)

3 × 4 = 12

↳ তিন ins. করলে 4 টা slot

0000 0000 1100.

↓ Positive (নিচের দিকে)

0 0000 0000 1100 → Discard to make it multiple of 2.

12 11 10:5 4:1

যদি নিচে থেকে উপরে যায়

loop.

5 × 4 = 20 → নিচে (তাই upore so, -20

sign bit.

0000 0001 0100

0 0000 0001 0100 = +20

1 1111 1110 1011 1's.

+1

1 1111 1110 1100 2's compliment (-20)

12 11 10:5 4:1 0

↳ box এর immediate.

(i) Form the 12 bit number.

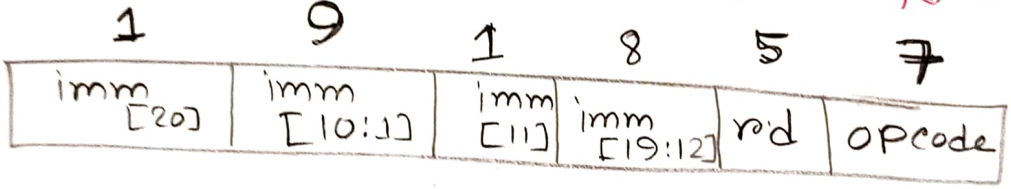
(12) (11) (10:5) (4:1)

(ii) If negative do 2's compliment again.

(iii) PC ± (imm × 2)

UJ type instruction. → JAL.

→ কোথায় যাবে jump.



Jal x0, 2000 → 20 bit immediate.