Decision Making

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

RISC-V includes two decision making instructions.

(if statement with a go to)

→ `beq rs1, rs2, L1`

- `beq` ↑ Branch if equal
- `rs1, rs2` ↑ label

Explanation: Go to the statement labeled "L1"; if the values in rs1 = rs2

`bne rs1, rs2, L1`

- `bne` ↑ Branch If not equal
- `rs1, rs2` ↑ label

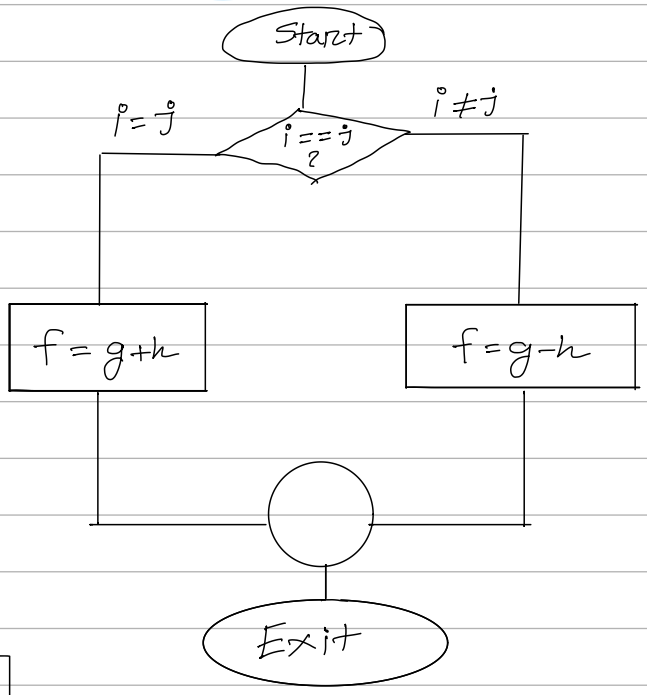
Explanation: Go to the statement labeled "L1"; if the values in rs1 != rs2

Conditional Branches: testing a value, based on the test result allows for a transfer of control to a new address in the program

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

Else:
    sub  x10, x20, x21

Exit:
```

(unconditional Branch)

f = x19
g = x20
h = x21
i = x22
j = x23

Conditional Jumps:

	Instruction	Syntax	operation
=	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

PC (Program Counter) => is the register that holds the address of the current instruction being executed.

Jal x_1, procedureLabel
↑
(PC+4) is stored.

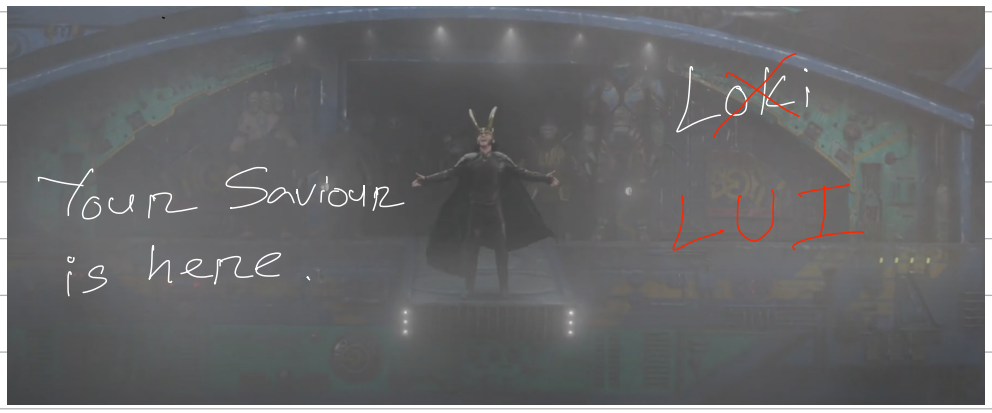
Jal x_0, Label } unconditional branch within a procedure.
↑
0 is hardwired;
Discard the return Address

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 (976) — 1280

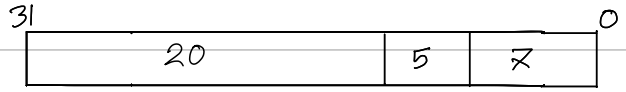
lui x_{10}, 976

x_{10} = 0000 0000 0000 0000 0000 0000 0000 0000 **0000 0000 0011 1101 0000** 0000 0000 0000

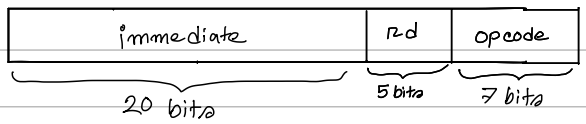
addi x_{10}, x_{10}, 1280

x_{10} = 0000 0000 0000 0000 0000 0000 0000 0000 **0000 0000 0011 1101 0000** 0101 0000 0000

U type instruction → LUI

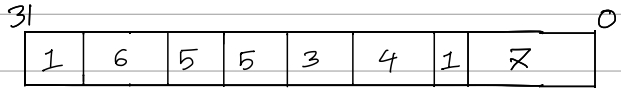


* each field has unique name and size.

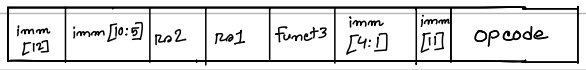


Branching Instructions

SB type instruction — beq, bne, blt, bge



each field has unique name and size.



represents branch addresses in multiples of 2.

forward & backward moving.

the unusual encoding of imm. simplifies datapath design.

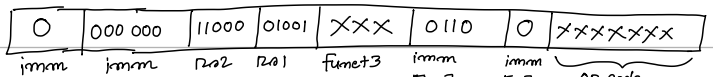
```
        Loop:
#80000  slli  X10, X22, 3      ——— line 1
#80004  add   X10, X10, X25    ——— line 2
#80008  ld    X9,  0(X10)      ——— line 3
#80012  bne   X9,  X24, Exit   ——— line 4
                rs1   rs2
#80016  addi  X22, X22, 1      ——— line 5
#80020  beq   X0,  X0,  loop   ——— line 6
        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

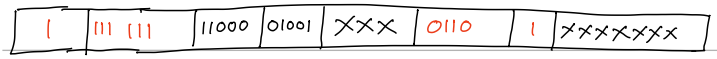
+ 1

1 1111 1110 1100 ⇒ −20 in 2s com.

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

PC relative addressing

= PC + immediate × 2



0

(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

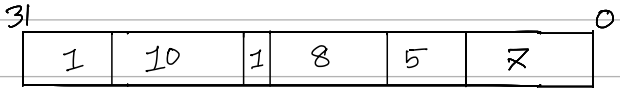
0000 0000 1010 ⇒ 10

(iii) PC ± imm × 2

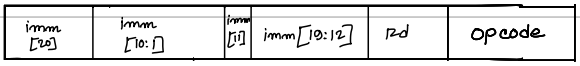
imm × 2 → offset

± → Based on the sign bit

UJ type instruction – Jal

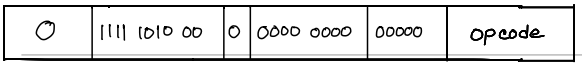


* each field has unique name and size.



* 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.

Jal X0, 2000



2000 = 0000 0000 0111 1101 0000

= 0 0000 0000 0111 1101 0000

20 | 19:12 | 11 | 10:1