Chapter-2

(How do we get a RISC-V instruction set from a high-level code?)

RISC-V → instruction set architecture.

Reduced Instruction Set Computer

an abstract interface between the hardware and the lowest level software.

RISC-V ISA has a modular design.

Base + Optional Extension

ISA

Unit	
Double Word	= 64 bits
Word	= 32 bits
Half Word	= 16 bits
Byte	= 8 bits

Register:

⇒ Used for frequently accessed data. Registers are faster than regular memory.

⇒ RISC-V has 32, 64-bit registers.] 64 bit architecture ⇒ handling 64 bit data at a time.

↓

Follows Design Principle 2 ⇒ Smaller is faster

↳ The clock period of the system could be limited by many different things, Register File is also one of those factors.

Hence, computer architects pick the size of the register file very carefully. 32 × 64 bit is a small enough register file that is not a limiting factor for our systems.

RISC-V Register Details:

Theoretically you could store any information in any register but architects/programmers have decided to follow a convention, in which they use certain registers for certain purposes.

↓

increases readability

Register Num.	Functionality
x0	Constant value 0 ⇒ holds constant value 0.
x1	Return address ⇒ to store the return address that you should return to after a function call.
x2	Stack pointer ⇒ stores an address in memory with the top of the stack.
x3	Global pointer
X4	Thread pointer
x5-7 ✓✓	Temporaries
x8	~~Saved register~~ / Frame pointer
x9 ✓	Saved register ⇒ to store variables.
x10-11	Function arguments / return values] for values passed to a function and values returned from a function.
x12-17	Function arguments
x18-27 ✓	Saved registers
x28-31 ✓✓	Temporaries ⇒ For holding temp. values

|| ⇒ Saved reg.

7 ⇒ Temporary reg.

Operands: part of instruction that contains the data to acted on

Based on the physical location

- Register operand (small capacity + faster access time)
- Memory operand (large capacity + longer access time)
- Constant / immediates (physically in the instruction itself)

or, the memory location of the data in a register.

Register Operand: Arithmetic instructions use register operands.

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

↳ Only one operation is performed per RISC-V instruction.

Hence, compiler must break this statement down to several parts.

RISC-V code:

g+h is a step towards our actual answer. Hence, stored in the temp regs. one of

```
add  x5, x20, x21
add  x6, x22, x23
sub  x19, x5, x6
```

same explanation as the prev. one

1 codeline divided into multi-ple instruction lines

$f \Rightarrow x_{19}$ ✓
$g \Rightarrow x_{20}$
$h \Rightarrow x_{21}$
$i \Rightarrow x_{22}$
$j \Rightarrow x_{23}$

Add / Sub instruction syntax:

Instruction name

```
add ____ , ____ , ____
    dest.  Source1  Source2
```

⇒ dest. = source1 + source2

```
sub ____ , ____ , ____
    dest.  Source1  Source2
```

⇒ dest. = source1 − source2

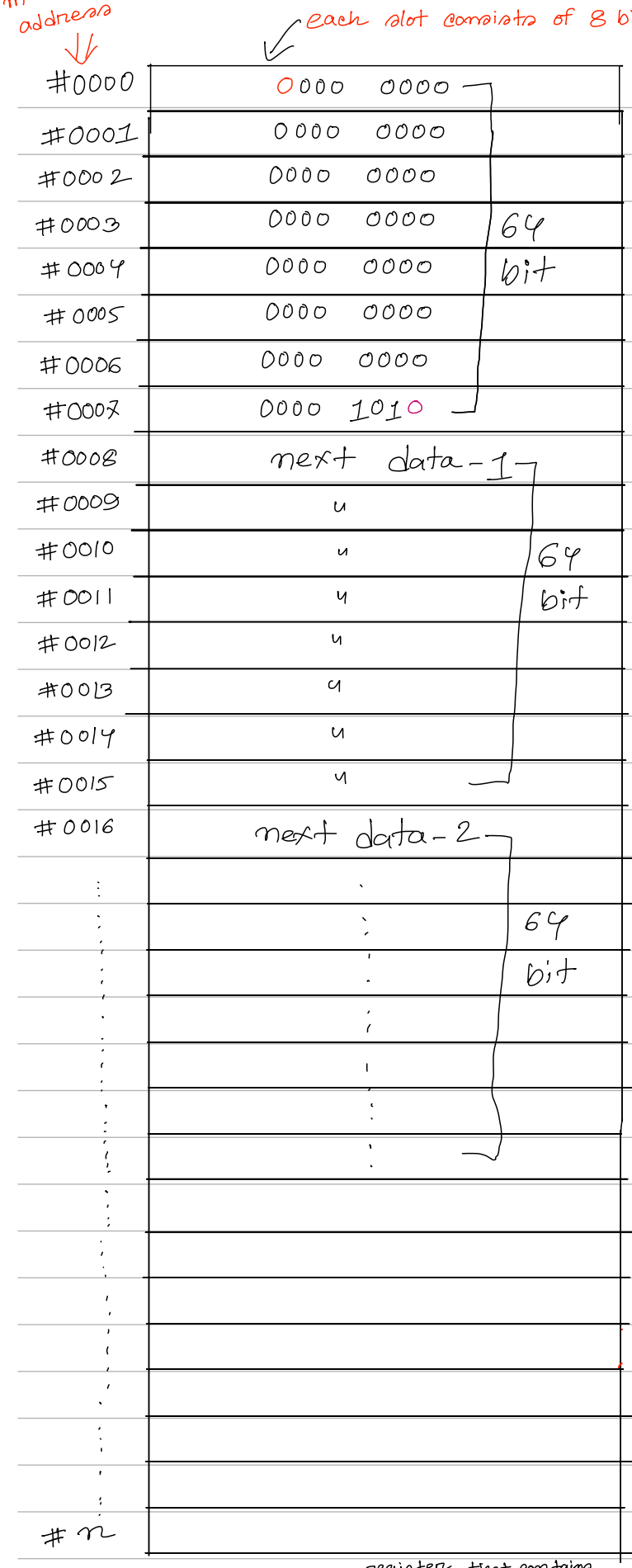
always 3 register operands

unique memory

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.

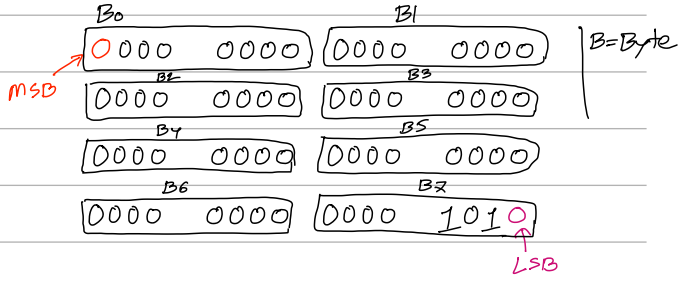
suppose we want to store $(10)_{10}$ in memory.

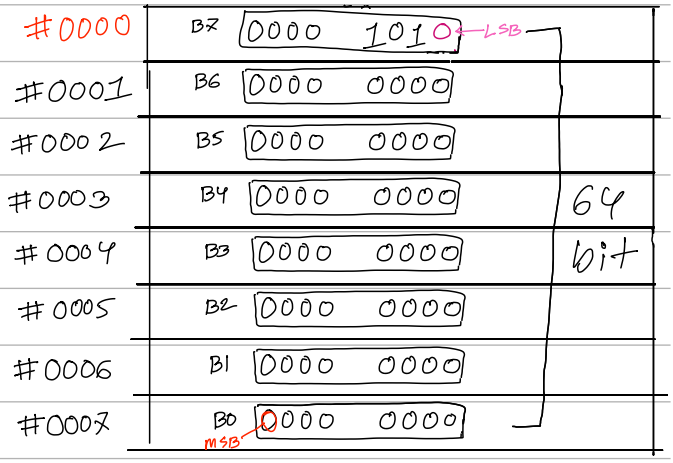
MSB
0000 0000 0000 0000
0000 0000 0000 0000
0000 0000 0000 0000
0000 0000 0000 1010
LSB

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

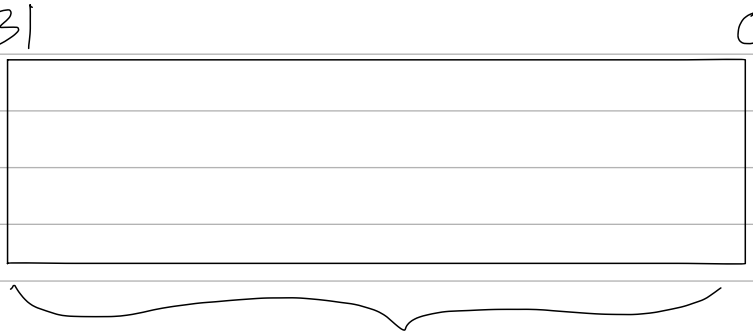
immediate	rs1	funct3	rd	opcode
000011110000	01010	011	01001	0000011

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.

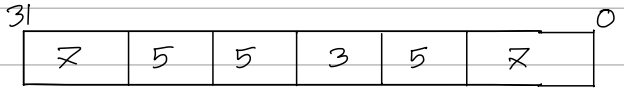
Design Principle 3: Good design demands good compromises.

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

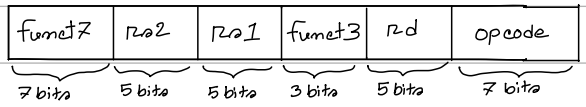
Instruction Formats:

- ↳ 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.

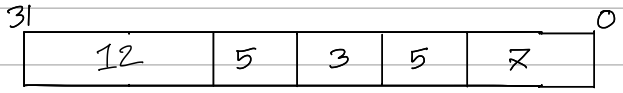
rd = Register destination operand

rs1 = " source 1 "

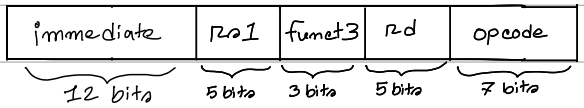
rs2 = " source 2 "

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

I type instruction



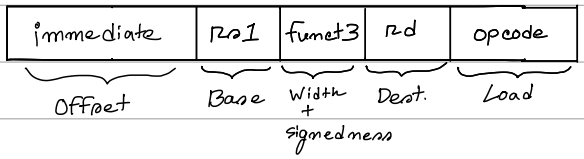
* each field has unique name and size.



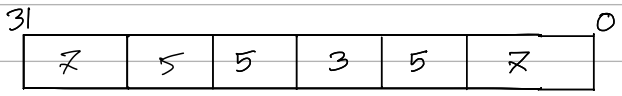
Immediate = Constant / Offset [2's comp.]

rs1 = Source / Base Register number

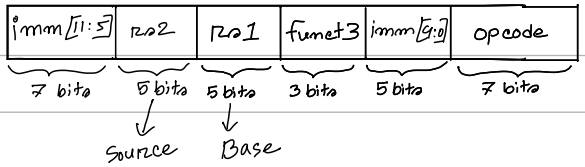
Load



S type instruction



* each field has unique name and size.

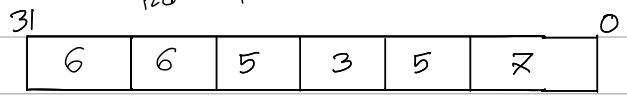


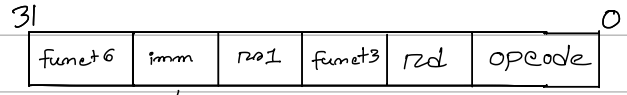
reason for imm. split is they want to keep rs1 and rs2 fields in the same place in all instruction formats.

Shift Operation

↳ follows I-type but modified

```
slli  x11, x19, 4
      rd   rs   imm
```





immediate broken down into 2 fields.

Why ?? ⇒ If you shift a 64 bit value more than 63 bits what happens?

Shift Left

↳ shift left and fill the positions with 0

⇒ We can perform multiplication by 2^i using slli.

```
slli  x11, x19, 4
```

↓ the value that will be stored in x11 is basically, val in x19 * 2^4

Shift Right

↳ shift right and fill the positions with 0

⇒ We can perform division by 2^i using srli.

```
srli  x11, x19, 4
```

↓ the value that will be stored in x11 is basically, val in x19 / 2^4

And ⇒ Bit Musking

and x_9, x_{10}, x_{11}
(rd, rs1, rs2)

x_{10} = 0000 .... 0000 1100 1100 ← only these two bits should remain as it is, rest 0.

x_{11} = 0000 .... 0000 0000 1100

x_9 = 0000 .... 0000 0000 1100

OR ⇒ Include Bits

and x_9, x_{10}, x_{11}
(rd, rs1, rs2)

x_{10} = 0000 .... 0000 1100 0000 ← you want to set these bits to 1 and rest should remain as it is.

x_{11} = 0000 .... 0000 0011 0000

x_9 = 0000 .... 0000 1111 0000

XOR ⇒ Can work as Buffer / Not

XOR x_9, x_{10}, x_{11}
(rd, rs1, rs2)

A	B	A ⊕ B	
0	0	0	XOR with 0 = Buffer
1	0	1	
0	1	1	XOR with 1 = Not
1	1	0	

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

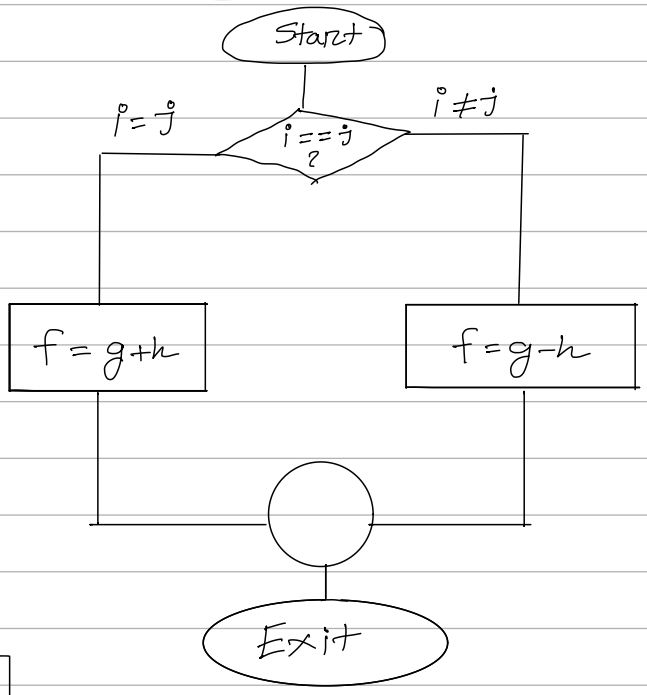
Conditional Branches

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

Loop

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

$i = X_{22}$
$k = X_{24}$
$a = X_{23}$
Save, base $= X_{25}$

```
Loop:
ld   X7, 0[X25]      X Static Wrong
BNE  X7, X24, Exit
Addi X22, X22, 1   }
Addi X23, X23, 1   }  Repeat ?
Beq  X0, X0, loop
Exit:
```

```
Loop:
SLLi X8, X22, 3
Add  X8, X25, X8
LD   X7, 0[X8]       ✓ Correct
BNE  X7, X24, Exit
Addi X22, X22, 1   }
Addi X23, X23, 1   }  Repeat ?
Beq  X0, X0, loop
Exit:
```

"Procedure Calling"

* Let's assume procedure like a spy.

Execution process of spy

(i) leaves with a secret plan.

(ii) acquires resources.

(iii) performs the task.

(iv) covers traces.

(v) returns to the point of origin with desired answers.

Execution of a procedure

Step 1: Put parameters in a place where the procedure can access them. → ($X_{10} - X_{17}$)

Step 2: Transfer control to the procedure.

Step 3: Acquire storage resources needed for the proc.

Step 4: Perform the desired task.

Step 5: Put the result value in a place where the calling program can access it.

Step 6: Return control to the point of origin. → X_1

[A proc. can be called from several points in a program]

JAL => Jump and Link instruction

Syntax: JAL X_1, procedureLabel

(X_1: Fixed; procedureLabel: Jump Destination)

Explination: Jump to procedureLabel and write return Address to X1.

JALR => Jump and Link Register

Syntax: JALR X_0, $0[X_1]$

($0[X_1]$: Jump Destination)

* The calling program (caller) puts the parameter values in $X_{10} - X_{17}$.
* Uses Jal X1, abc to branch to procedure label abc (callee)
* Callee performs the calculations, places the results in the same parameter registers
* Returns control to the caller using jalr X_0, $0(X_1)$

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

What if compiler needs more registers for a procedure than the 8 arg. registers?

⇒ If the callee require any register that is already in use by the caller, callee must restore the values that were contained before the procedure was invoked.

↳ First store the values in stack

You want to store 3 register datas.

⇒ each register size = 64 bits
= 8 locations

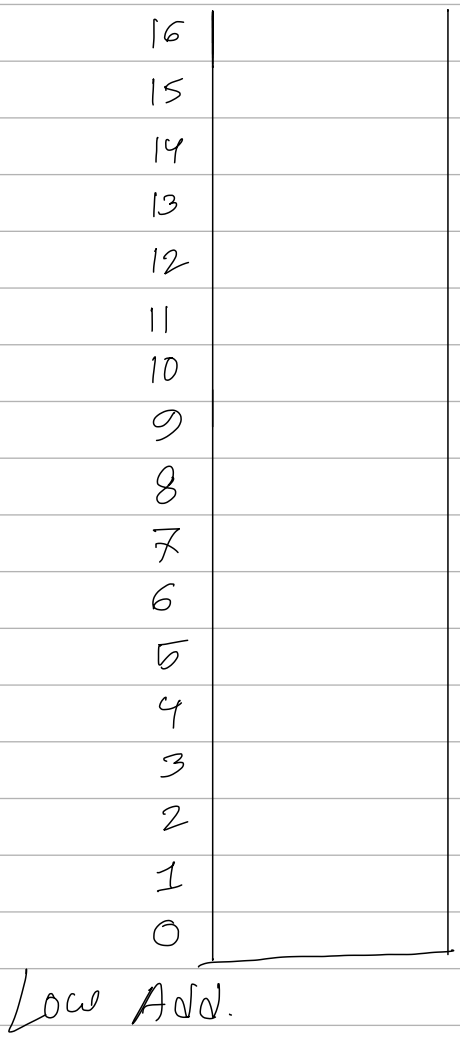
High Add.

SP ⇒ 24
23
22
21
20
19
18
17

Stack

⇒ to store data of 3 registers
= (3 * 8) = 24 locations are needed

(LIFO)



(i) Decrease the SP by the #locations you need

(ii) Then store the values in stack

```
def leaf_example(g, h, i, j):
    f = (g+h) - (i+j)
    return f
```

	$g = x_{10}$
leaf_procedure:	$h = x_{11}$
addi sp, sp, -24	$i = x_{12}$
sd x5, 16(sp) } Stored into	$j = x_{13}$
sd x6, 8(sp) } stack	$f = x_{20}$ } Save these
sd x20, 0(sp) }	temp = x_5, x_6 } 3 in the stack first.
Add x_5, x_{10}, x_{11}	
Add x_6, x_{12}, x_{13}	
Sub x_{20}, x_5, x_6	
Addi $x_{10}, x_{20}, 0$ } Store the return value	

```
ld   x20,  0(sp)   ┐
ld   x6,   8(sp)   ├ Stored into stack
ld   x5,  16(sp)   ┘
addi sp, sp, 24
jalr x0, 0[x1]
```

To avoid saving and restoring a register whose value is never used,

$X_5 - X_7$ and $X_{28} - X_{31}$ ⇒ temp. register values are not preserved by the callee.

$X_8 - X_9$ and $X_{18} - X_{27}$ ⇒ saved register values must be preserved by the callee.

Hence, from the above example we do not need to restore the values of X_5, X_6

"Memory Layout"

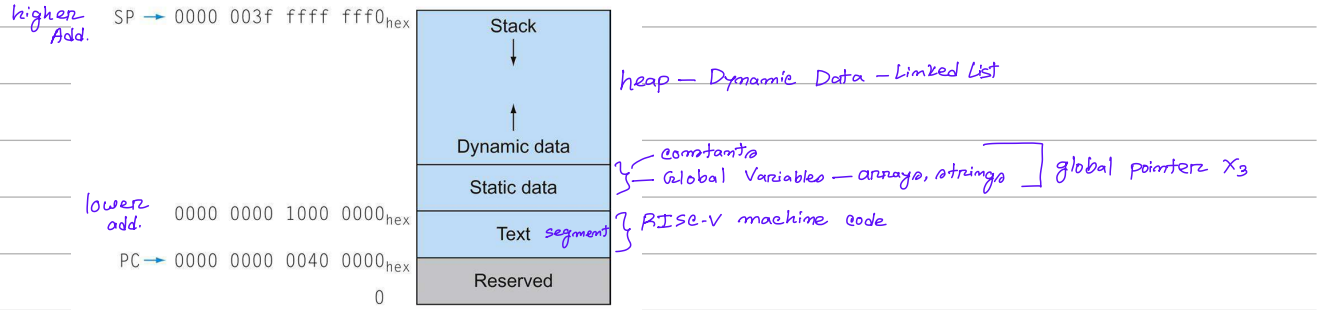
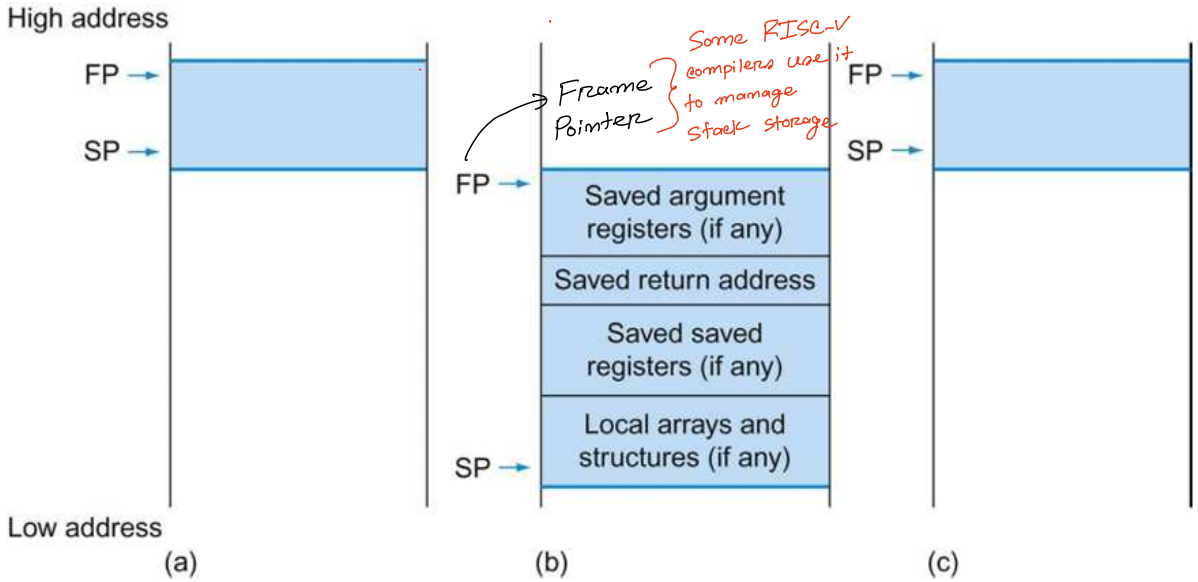


FIGURE 2.13 The RISC-V memory allocation for program and data.

⇒ RISC-V convention for allocation of memory when running the Linux OS.

Local Data on Stack



The segment of the stack that contains a procedure's saved registers, and local variables called Procedure Frame / Activation Period.

- saved reg.
- arg. reg.
- return add. reg.

What if there are more than 8 parameters?

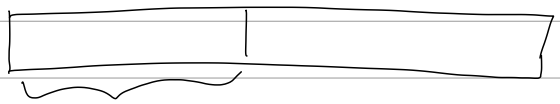
- place the extra parameters on the stack just above the frame pointer.
- Procedure will expect first eight params to be in (x_{10}-x_{17}) & rest in memory, addressable via the frame pointer.

- RISC-V byte/halfword/word load/store
  - Load byte/halfword/word: Sign extend to 64 bits in rd
    - `lb rd, offset(rs1)`
    - `lh rd, offset(rs1)`
    - `lw rd, offset(rs1)`
  - Load byte/halfword/word unsigned: Zero extend to 64 bits in rd
    - `lbu rd, offset(rs1)`
    - `lhu rd, offset(rs1)`
    - `lwu rd, offset(rs1)`
  - Store byte/halfword/word: Store rightmost 8/16/32 bits
    - `sb rs2, offset(rs1)`
    - `sh rs2, offset(rs1)`
    - `sw rs2, offset(rs1)`

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

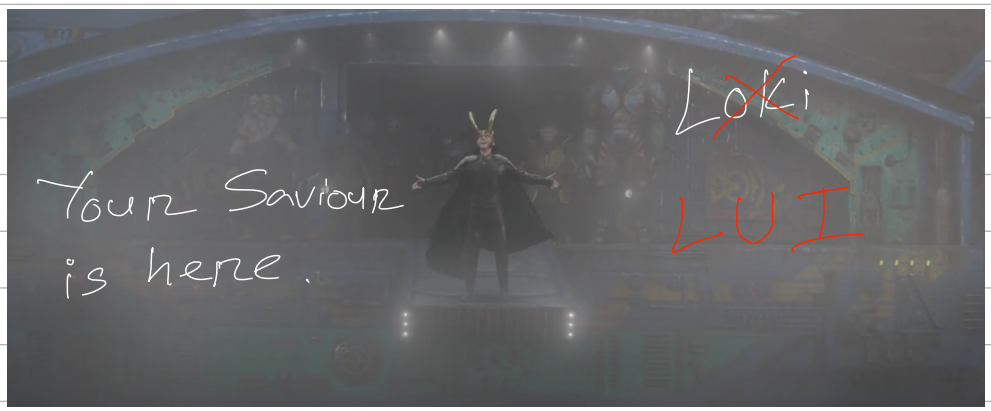
Addi X_{10}, X_{10}, 65520

I type =



imm
= 12 bits ??

then how do we do this?

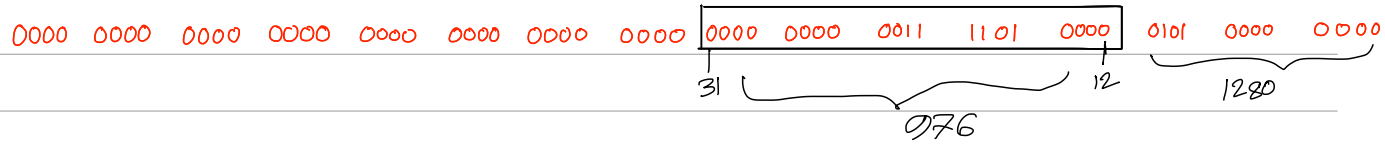


LUI = Load Upper Immediate — use it to form 32 bit immediates

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} ⇒



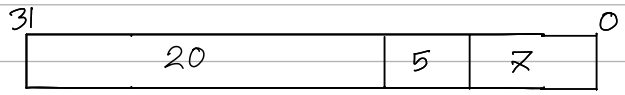
lui x_{10}, 976

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

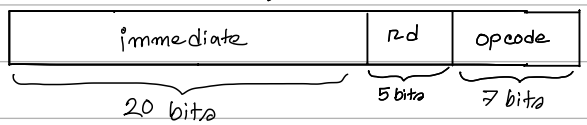
addi x_{10}, x_{10}, 1280

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

U type instruction → LUI

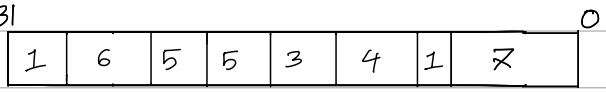


* each field has unique name and size.

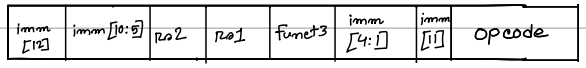


Branching Instructions

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.

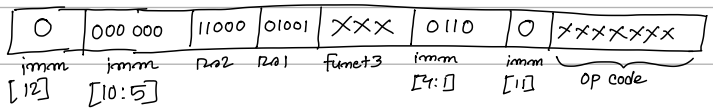
	Loop:	
#80000	slli X_{10}, X_{22}, 3	line 1
#80004	add X_{10}, X_{10}, X_{25}	line 2
#80008	ld X_9, 0(X_{10})	line 3
#80012	bne X_9, X_{24}, Exit	line 4
#80016	addi X_{22}, X_{22}, 1	line 5
#80020	beq X_0, X_0, loop	line 6
	Exit:	
#80024		line 7

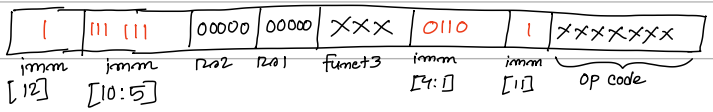
3 × 4 = 12

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

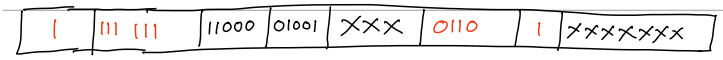
+ 1

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

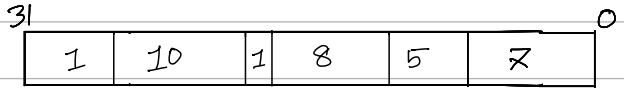
+ 1

0000 0000 1010 ⇒ 10

(iii) PC ± imm × 2 → offset

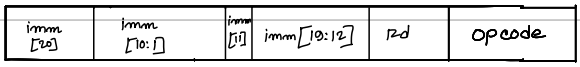
Based on the sign bit

UJ type instruction – Jal



* uses 20 bit immediates for larger jumps.

* each field has unique name and size.

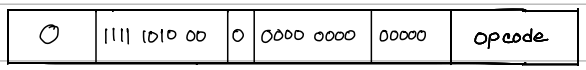


* For more large jump, (32 bit)

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

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:
```