

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

Registers:

⇒ 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 ↴ => Temporary reg.
x9 ✓✓	Saved register => to store variables.
x10-11	Function arguments / return values] for values passed to a function
x12-17	Function arguments] and values returned from a function.
x18-27 ✓✓	Saved registers
x28-31 ✓✓	Temporaries => for holding temp. values

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

Based on the physical location

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

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

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:
1 codeline divided into multiple instruction lines

add x_5, x_{20}, x_{21}	$g+h$ is a step towards our actual answer. One of them stored in the temp reg. same explanation as the prev. one
add x_6, x_{22}, x_{23}	
sub x_{10}, x_5, x_6	

$$f \Rightarrow x_{10} \checkmark$$

$$g \Rightarrow x_{20}$$

$$h \Rightarrow x_{21}$$

$$i \Rightarrow x_{22}$$

$$j \Rightarrow x_{23}$$

Add/Sub instruction syntax:

Instruction name
add dest., source1, source2
 \Rightarrow dest. = source1 + source2

sub dest., source1, source2
 \Rightarrow dest. = source1 - source2

always 3 registers
operands

unique
memory

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.

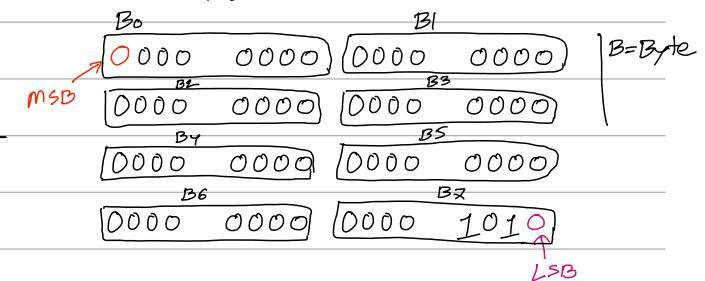
Endianness: The order in which computer memory stores data. (byte)

→ 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

constant data specified in an instruction, **No sub!**

addi $x_{22}, x_{22}, 4$

immediate to sub 4 \Rightarrow

addi $x_{22}, x_{22}, -4$

Make it -4

Addi Instruction Syntax :

addi $\underbrace{\text{dest}}, \underbrace{\text{Source 1}}, \underbrace{\text{Source 2 (constant)}}$

$\Rightarrow \text{dest.} = \text{source 1} + \text{source 2}$

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

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
pos ↓	neg ↓

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

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

↓ 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 0100011

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**.

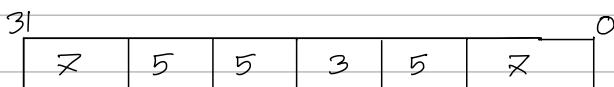
Design Principle 3: Good design demands good compromises.

⇒ 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	r2s2	r2s1	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

r2s1 = u source1 u

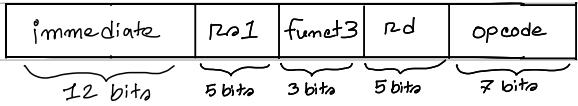
r2s2 = u source2 u

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

I type instruction

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

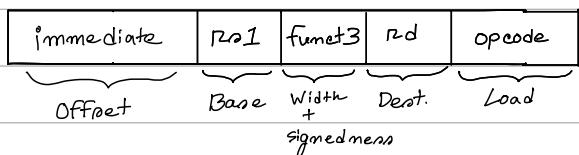
each field has unique name and size.



Immediate = Constant / offset [2s comp.]

r201 = Source / Base Register number

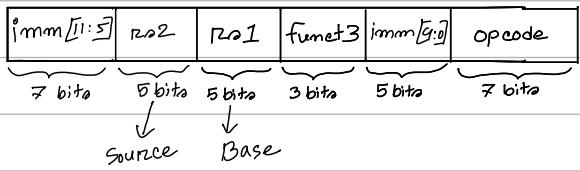
Load



S type instruction

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

each field has unique name and size.

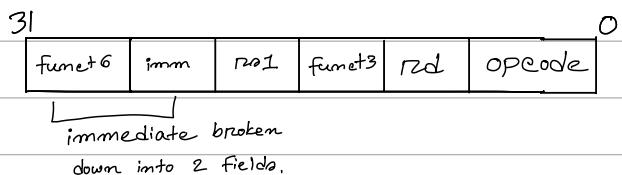
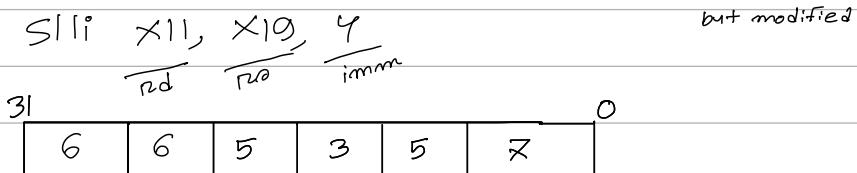


reason for imm. split is they want to keep r201 and r202

fields in the same place in all instruction formats.

Shift Operation

↳ follows I-type



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

Shift Left

↳ Shift left and fill the positions with 0

\Rightarrow We can perform multiplication

by 2^i using Slli.

Slli $x_{11}, x_{10}, 4$

↓
the value that will be

stored in x_{11} is basically,

val in $x_{10} \times 2^4$

Shift Right

↳ Shift right and fill the positions with 0

\Rightarrow We can perform division.

by 2^i using srli.

Srli $x_{11}, x_{10}, 4$

↓
the value that will be

stored in x_{11} is basically,

val in $x_{10} / 2^4$

And \Rightarrow Bit Masking

and x_0, x_{10}, x_{11}

$$x_0 = 0000 \dots 0000 1100 \quad \text{only these two bits should remain as it}$$

$$x_{11} = 0000 \dots 0000 0000 1100 \quad \text{in, rest 0.}$$

$$x_0 = 0000 \dots 0000 0000 1100$$

OR \Rightarrow Include Bits

and x_0, x_{10}, x_{11}

$$x_0 = 0000 \dots 0000 1100 0000 \quad \text{you want to set these bits to 1 and rest}$$

$$x_{11} = 0000 \dots 0000 0011 0000 \quad \text{should remain as it is.}$$

$$x_0 = 0000 \dots 0000 1111 0000$$

XOR \Rightarrow Can work as Buffer / Not

XOR x_0, x_{10}, x_{11}

A	B	$A \oplus B$
0	0	0
1	0	1
0	1	1
1	1	0

XOR with
0 = Buffer

XOR with
1 = Not

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.

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

Conditional Branches

(if statement with a go to)

$beg\ r21, r22, L1$

Branch If equal

label

Explanation: Go to the statement labeled "L1"; if the values in $r21 = r22$

$bne\ r21, r22, L1$

Branch If not equal

label

Explanation: Go to the statement labeled "L1"; if the values in $r21 \neq r22$

Given Code

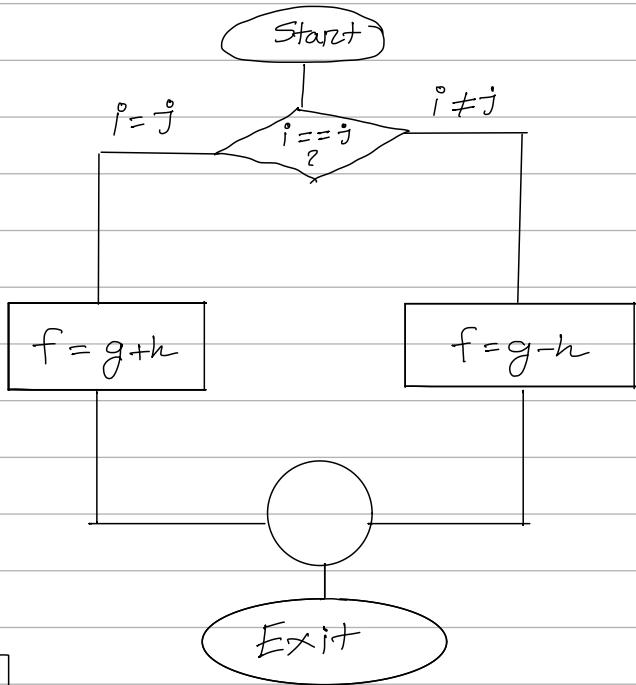
If ($i == j$) :

$f = g + h$

else :

$f = g - h$

Flow Chart:



RISC-V assembly code:

$bne\ x_{22}, x_{23}, \text{Else}$ $\text{add}\ x_{10}, x_{20}, x_{21}$ $\text{(unconditional } \rightarrow \text{Branch})$ $\text{beg}\ x_0, x_0, \text{Exit}$ $\text{Else}:$ $\text{Sub}\ x_{10}, x_{20}, x_{21}$ $\text{Exit}:$
--

$f = x_{10}$
 $g = x_{20}$
 $h = x_{21}$
 $i = x_{22}$
 $j = x_{23}$

Conditional Jumps:

	Instruction	Syntax	operation
$=$	beg	$beg\ r21, r22, L1$	$r21 == r22$
\neq	bne	$bne\ r21, r22, L2$	$r21 \neq r22$
$<$	blt	$blt\ r21, r22, L3$	$r21 < r22$
\geq	bge	$bge\ r21, r22, L4$	$r21 \geq r22$

Loop

while ($\text{Save}[i] == k$)
 $i = i + 1$
 $a = a + 1$

$i = x_{22}$
 $k = x_{24}$
 $a = x_{23}$
 $\text{Save}, \text{base} = x_{25}$

Loop:

$ld\ x_7, 0[x_{25}]$ X Static Wrong
BNE x_7, x_{24} , Exit

Addi $x_{22}, x_{22}, 1$ } Repeat?
Addi $x_{23}, x_{23}, 1$ }

Beg x_0, x_0, loop

Exit:

Loop:
SLLi $x_8, x_{22}, 3$
Add x_8, x_{25}, x_8
LD $x_7, 0[x_8]$

✓ Correct

BNE x_7, x_{24} , Exit

Addi $x_{22}, x_{22}, 1$ } Repeat?
Addi $x_{23}, x_{23}, 1$ }

Beg x_0, x_0, loop

Exit:

"Procedure Calling"

* Let's assume procedure like a SPJ.

Execution process of SPJ

(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. $\rightarrow X_1$

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

JAL \Rightarrow Jump and Link instruction

Syntax: **JAL** X_1 , procedureLabel
 \uparrow Fixed
Jump Destination

Explanation: Jump to procedureLabel and write return Address to X_1 .

JALR \Rightarrow Jump and Link Register

Syntax: **JALR** $X_0, 0[X_1]$
 \downarrow Jump Destination

* The calling program (caller) puts the parameters values in $X_{10} - X_{17}$.

* Uses **Jal** X_1, 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) \Rightarrow 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 \Rightarrow 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?

\Rightarrow 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.

\Rightarrow each register size = 64 bits

High Add.

= 8 locations

SP \Rightarrow 24
23
22
21
20
19
18
17

Stack

\Rightarrow to store data of 3 registers
 $= (3 \times 8) = 24$ locations

are needed

(LIFO)

16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0

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

(ii) Then store the values in stack

Low Add.

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

leaf_procedure:

addi SP, SP, -24

sd x5, 16(SP)

sd x6, 8(SP)

sd x20, 0(SP)

} stored into stack

$g = x_{10}$

$h = x_{11}$

$i = x_{12}$

$j = x_{13}$

$f = x_{20}$

$\text{temp} = x_5, x_6$

} save these 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

$| d \quad x_{20}, \quad 0(sp) \quad \} \quad$
 $| d \quad x_6, \quad 8(sp) \quad \} \quad$
 $| d \quad x_5, \quad 16(sp) \quad \}$
Stored into
stack

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 $x_{28} - x_{31} \Rightarrow$ temp. Register values are not preserved by the callee.

$x_8 - x_9$ and $x_{18} - x_{27} \Rightarrow$ 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"

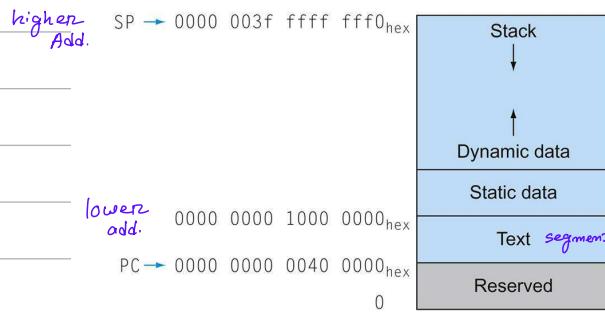


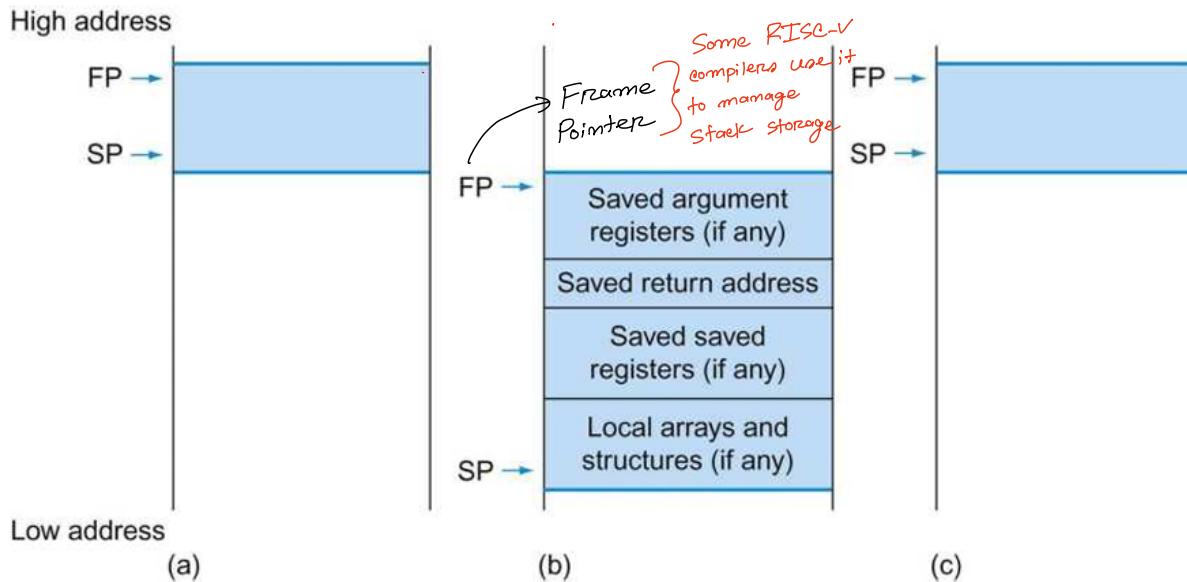
FIGURE 2.13 The RISC-V memory allocation for program and data.

heap - Dynamic Data - Linked List

constants
Global Variables - arrays, strings] global pointers x_3
RISC-V machine code

\Rightarrow 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.**

What if there are more than 8 parameters?

→ saved reg.
→ arg reg.
→ return add. reg.

→ place the extra parameters on the stack just above the frame pointer.

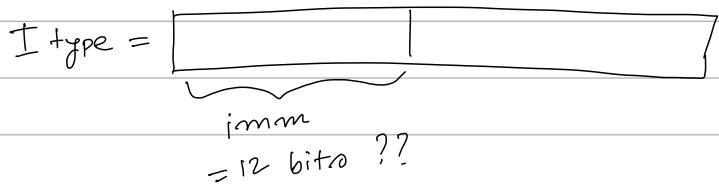
→ Procedure will expect first eight params to be in $(x_{10} - x_{17})$ & rest in memory, addressable via the frame pointer.

RISC-V byte/halfword/word load/store

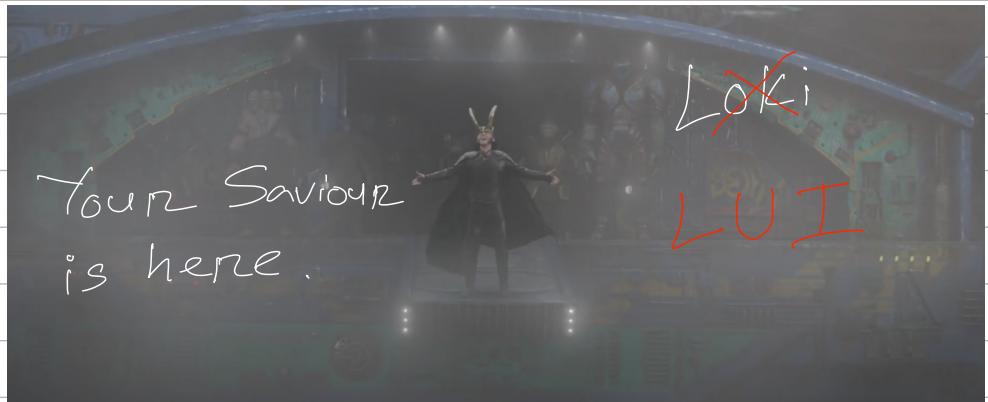
- Load byte/halfword/word: Sign extend to 64 bits in rd
 - `lb rd, offset(rs1)`
 - `lh rd, offset(rs1)`
 - `lw rd, offset(rs1)`
- Load byte/halfword/word unsigned: Zero extend to 64 bits in rd
 - `lbu rd, offset(rs1)`
 - `lhu rd, offset(rs1)`
 - `lwu rd, offset(rs1)`
- Store byte/halfword/word: Store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

Try this $\Rightarrow X_{10} = 1111\ 1111\ 1111\ 0000$

Add: X_{10} , X_{10} , 65520



then how do we do this?

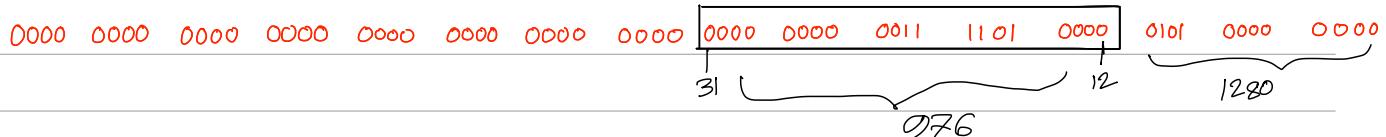


LUI = Load Upper Immediate — use it to form 32 bit
immediates

Syntax = LUI rd, constant

- * 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 $X_{10} \Rightarrow$



$lui X_{10}, 076$

$$X_{10} = 0000 0000 0000 0000 0000 0000 0000 \boxed{0000 0000 0011 1101 0000} 0000 0000 0000$$

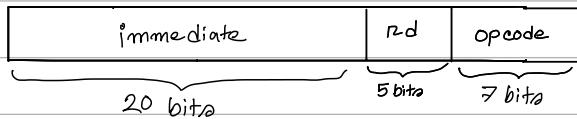
$addi X_{10}, X_{10}, 1280$

$$X_{10} = 0000 0000 0000 0000 0000 0000 0000 \boxed{0000 0000 0011 1101 0000} 0101 0000 0000$$

U type instruction $\rightarrow LUI$



* each field has unique name and size.



Branching Instructions

SB type instruction — beq, bne, blt, bge

31

1	6	5	5	3	4	1	2
---	---	---	---	---	---	---	---

0

represents branch addresses
in multiples of 2.

each field has unique name and size.

imm [12]	imm [10:5]	r22	r21	funct3	imm [4:1]	imm [1]	opcode
----------	------------	-----	-----	--------	-----------	---------	--------

forwarded & 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 x9, x9, loop — line 6 ✓

Exit:

- #80024 — line 7 ✓

$$3 \times 4 = 12$$

0000 0000 1100

0 0000 0000 1100

Discard it.

12 11 10:5 4:1

0	000 000	11000	01001	XXX	0110	0	xxxxxxx
[12]	[10:5]				[4:1]	[1]	

1	111 111	00000	00000	XXX	0110	1	xxxxxxx
[12]	[10:5]				[4:1]	[1]	

$5 \times 4 = 20$ but its going upward so -20.

= 0000 0001 0100

= 0 0000 0001 0100

= 1 111 1110 1011

+ 1

$\overline{1 \ 111 \ 1110 \ 1100} \Rightarrow -20$ in 2's comp.

1	111 111	11000	01001	XXX	0110	1	xxxxxxx
---	---------	-------	-------	-----	------	---	---------

0

(i) Form the 12 bit number

11 111 111 0110

(ii) Detect if positive or negative number.

if neg perform 2's comp again,

1111 1111 0110

0000 0000 1001

$+ 1$
 $\overline{0000 \ 0000 \ 1010} \Rightarrow 10$

(iii) PC \oplus imm $\times 2$ offset

Based on the sign bit

UJ type instruction - Jal

31

1	10	1	8	5	x
---	----	---	---	---	---

0

uses 20 bit

immediates for
larger jumps.

each field has unique name and size.

imm _{20:8}	imm _{10:1}	imm ₁	imm _[10:12]	rd	opcode
---------------------	---------------------	------------------	------------------------	----	--------

For more large jump, (32 bit)

lui: load address [31:12]
to a temp reg.

Jal x0, 2000

0	1111 0100 00	0	0000 0000	00000	opcode
---	--------------	---	-----------	-------	--------

jalr: add address [11:0] and
jump to target.

$$2000 = 0000 \ 0000 \ 0111 \ 1101 \ 0000$$

$$= 0 \underbrace{0000 \ 0000}_{20} \ / \underbrace{1111 \ 1101 \ 0000}_{10:1}$$

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:
```