"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

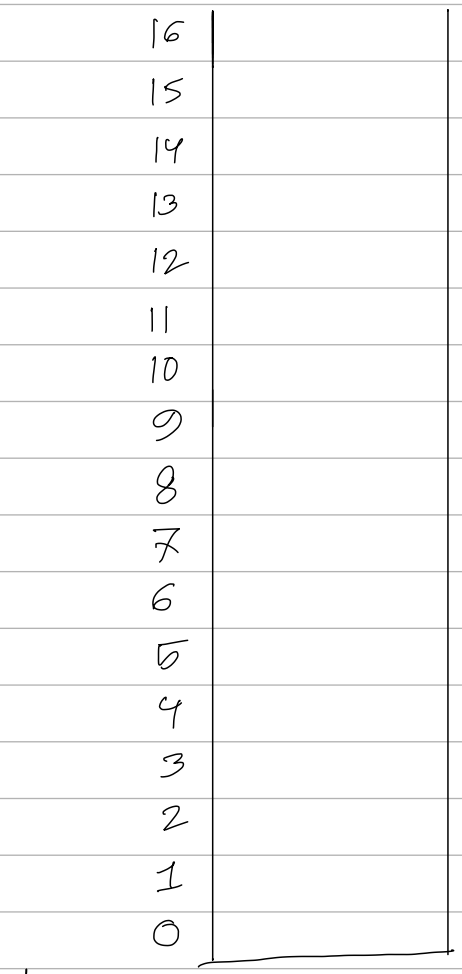
High Add.

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"

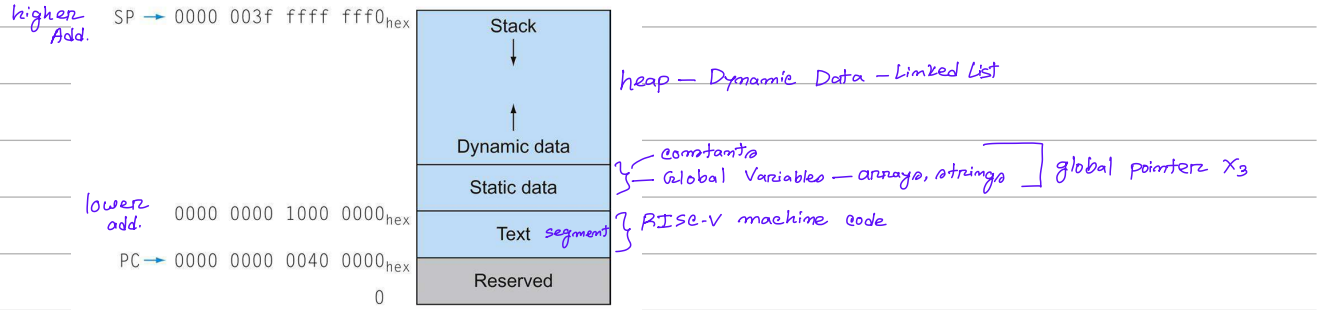
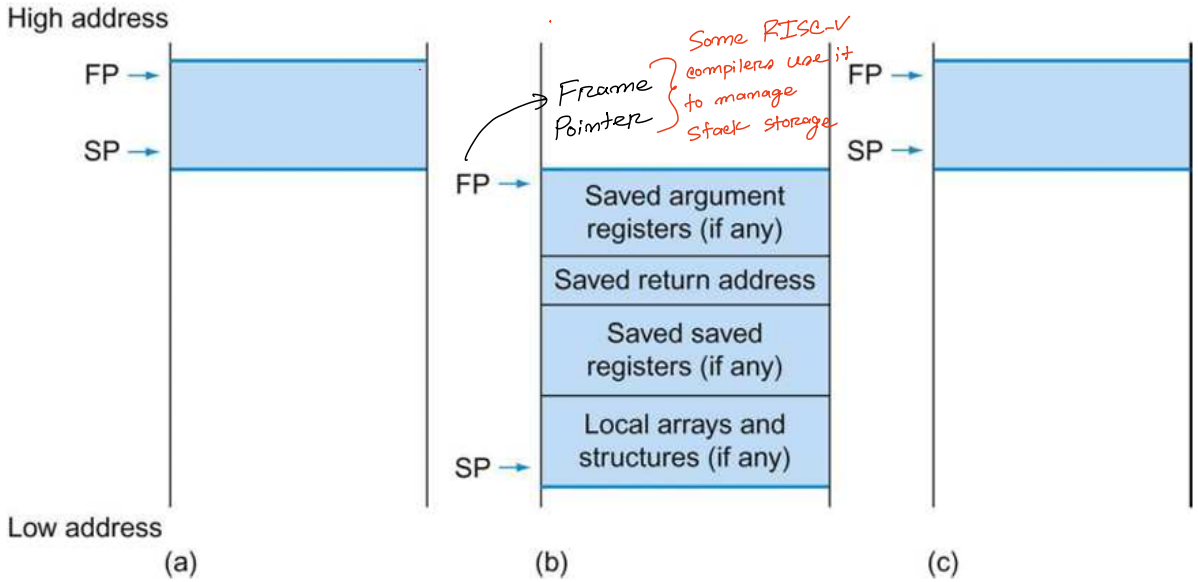


FIGURE 2.13 The RISC-V memory allocation for program and data.

⇒ 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?

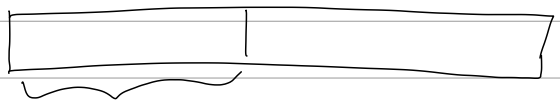
- place the extra parameters on the stack just above the frame pointer.
- Procedure will expect first eight params to be in (x_{10}-x_{17}) & rest in memory, addressable via the frame pointer.

- 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

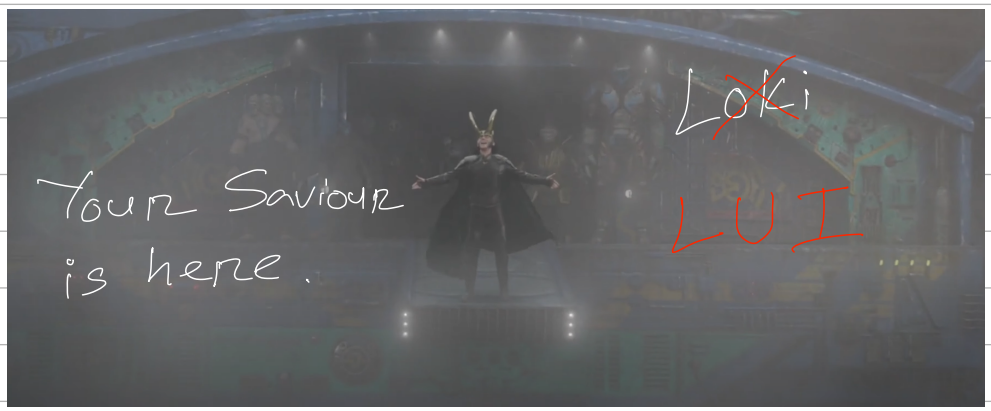
Addi X_{10}, X_{10}, 65520

I type =



imm
= 12 bits ??

then how do we do this?

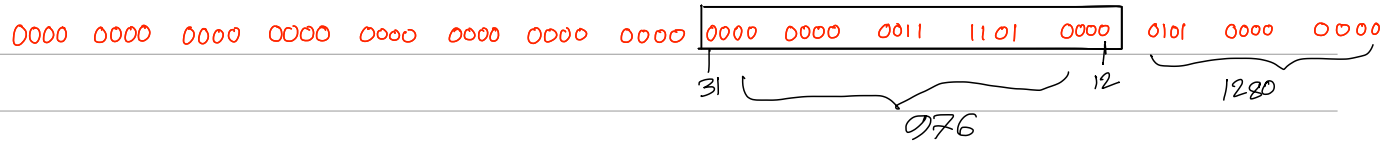


LUI = Load Upper Immediate — use it to form 32 bit immediates

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} ⇒



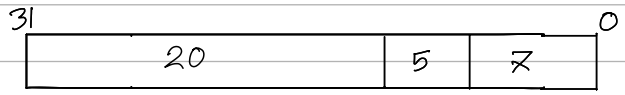
lui x_{10}, 976

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

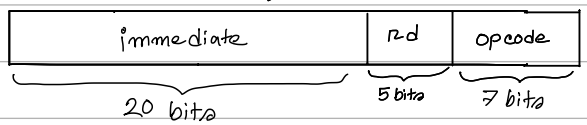
addi x_{10}, x_{10}, 1280

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

U type instruction → LUI

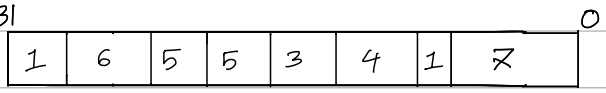


* each field has unique name and size.

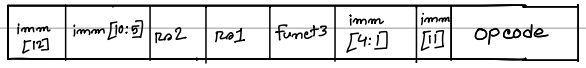


Branching Instructions

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.

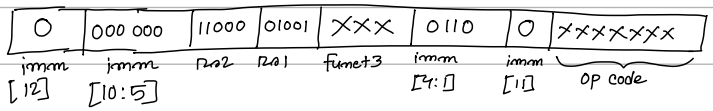
	Loop:	
#80000	slli X_{10}, X_{22}, 3	line 1
#80004	add X_{10}, X_{10}, X_{25}	line 2
#80008	ld X_9, 0(X_{10})	line 3
#80012	bne X_9, X_{24}, Exit	line 4
#80016	addi X_{22}, X_{22}, 1	line 5
#80020	beq X_0, X_0, loop	line 6
	Exit:	
#80024		line 7

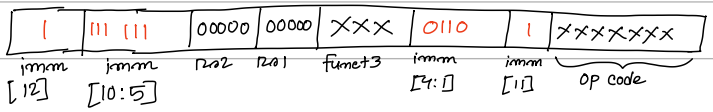
3 × 4 = 12

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

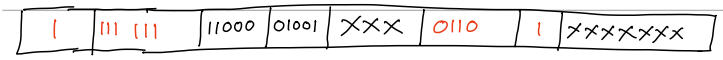
+ 1

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

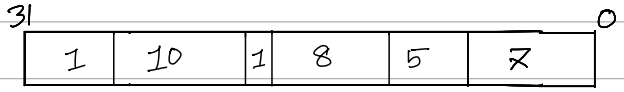
+ 1

0000 0000 1010 ⇒ 10

(iii) PC ± imm × 2 → offset

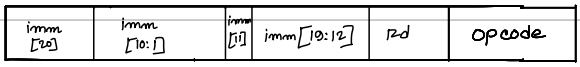
Based on the sign bit

UJ type instruction – Jal



* uses 20 bit immediates for larger jumps.

* each field has unique name and size.

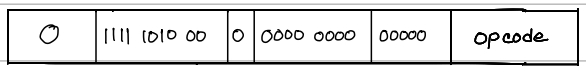


* For more large jump, (32 bit)

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:
```