Nested / Non-leaf Procedures

* Procedures that call another procedure.
  - a new procedure.
  - it self.

* Callee pushes the return address

```
def max(a, b):
    if (a > b):
        return a
    else:
        return b
```

Procedure calling another procedure

* caller must push
  - arg. registers (x_{10}-x_{17})
  - temp. registers (x_5-x_7, x_{28}-x_{31})
  - return address (x_1) before the call.
  - restore from stack after the call.

```
def calc(a, b, c):
    d = max(a, b)        # x10
    return d+c
```

jal x_1, max → return address to procedure calc.

```
print(calc(8, 7, 10))
```

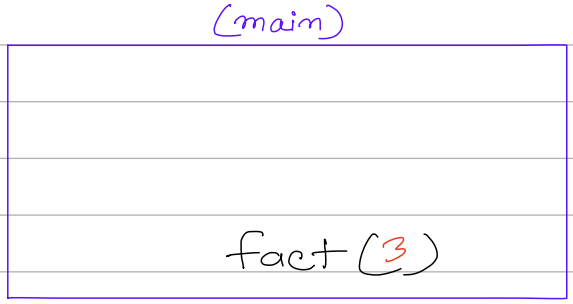
x_{10}

jal x_1, calc → return address to main prog.

Conflict arises
- x_{10} val. changed
- x_1 – return val. changed.

Example Non-Leaf :-

```
def fact(n):
    if (n < 1):
        return 1
    else:
        return (n * fact(n-1))
```



```
Addi x10, x0, 3      [ Storing value in the arg register]
Jal  x1,  fact       [ transfering control to callee ]
```

```
Fact:
Addi  SP,  SP,  -16
SD    x1,  8(SP)
SD    x10, 0(SP)
```
Storing the return address & value of n in stack

```
Addi  x5,  x0,  1    [ Storing 1 in temp reg. x1]
BGE   x10, x5,  Else [ n >= 1 ⇒ else ]
Addi  x10, x0,  1    [ return 1]
Addi  SP,  SP,  16   [x1 and x10 do not change when n<1]
Jalr  x0,  0(x1)     [return control to caller]
```

```
Else:
Addi  x10, x10, -1   [ n = n-1 & argument stored in x10 ]
Jal   x1,  Fact
LD    x6,  0(SP)
LD    x1,  8(SP)
Addi  SP,  SP,  16
Mul   x10, x10, x6
Jalr  x0,  0(x1)
```
Loading the previously stored values into stack.

"Memory Layout"

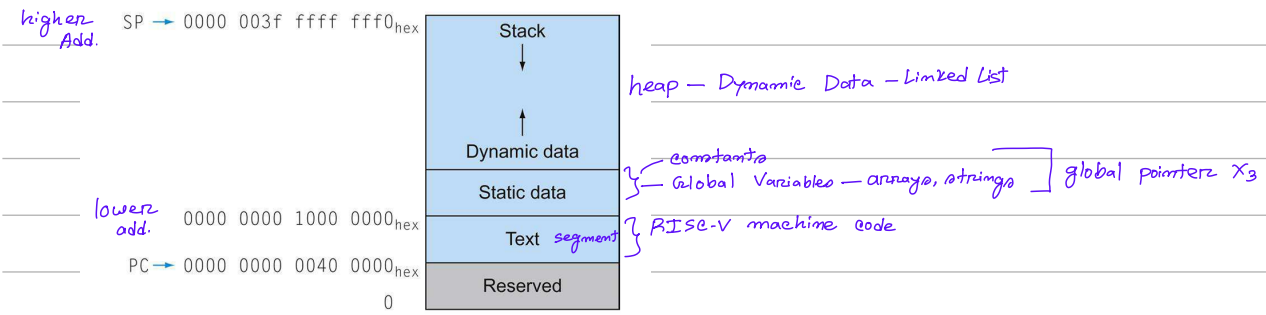


FIGURE 2.13 The RISC-V memory allocation for program and data.

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

C

C ⇒ malloc() → allocates space on heap and returns a pointer

Free() → releases space on to the heap

uses these functions explicitly for allocating and releasing space on the heap.

↓

Leads to bugs.

such as → memory leak — forgets to free space — OS crash

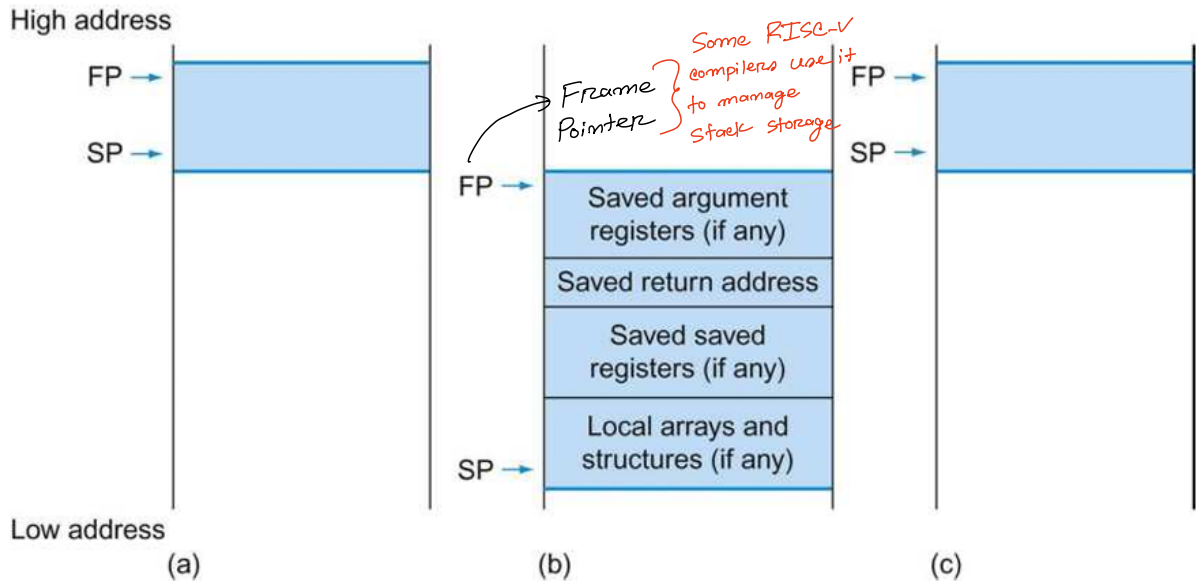
→ dangling pointers.

└ freeing space too early

Java

uses automatic memory allocation and garbage collection to avoid these bugs.

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)`

Str Copy Example in C

```
void strcpy (char x[], char y[])
{ size_t i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

8 bit 8 bit

=> size of the value stored in each index is 8 bits

Base of x = X_{10}

" " y = X_{11}

I = X_{19}

```
Addi  SP, SP, -8     ] Storing saved
Sd    X19, 0(SP)     ] register data
Addi  X19, X0, 0     ] X19 = i = 0
Loop:
Add   X5, X10, X11   ] physical add. calculation. | char = 8 bits data in C
                       for y[i]
lbu   X6, 0(X5)      ] load
Add   X7, X10, X10   ] physical add. calculation
                       for x[i]
sb    X6, 0(X7)      ] store
```

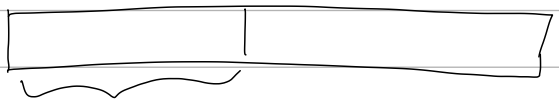
```
Beq   X6, X0, terminateLoop ]  if whatever you stored in y[i] == 0 terminate loop.
Addi  X10, X10, 1           ]  i += 1
Jal   X0, Loop              ]

terminateLoop:
ld    X10, 0(SP)    ]  restore the value in X10
Addi  SP, SP, 8     ]  and increase the stack pointer
jalr  X0, 0(X1)     ]  return control to caller.
```

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

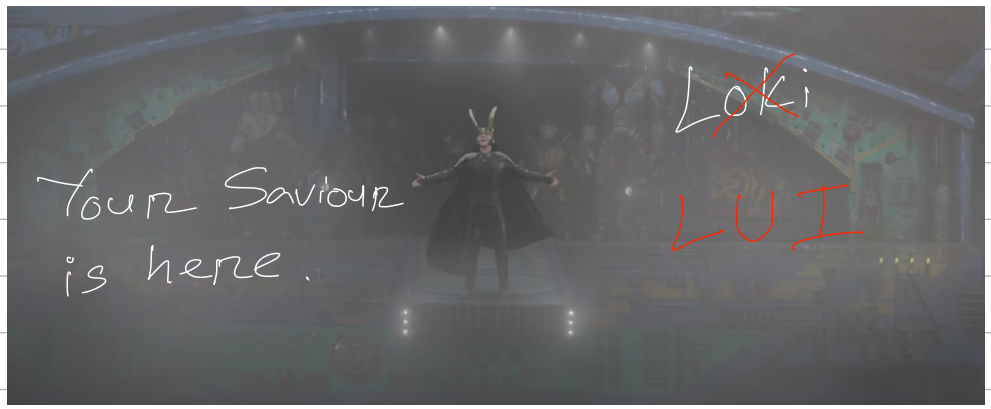
Addi X10, X10, 65520

I type =



imm
= 12 bits ??

then how do we do this?

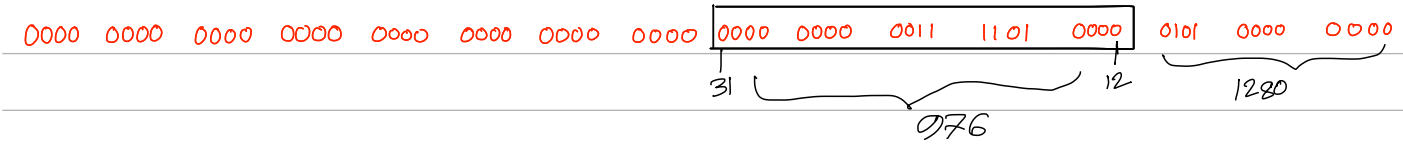


LUI = Load Upper Immediate — use it to form 32 bit immediates

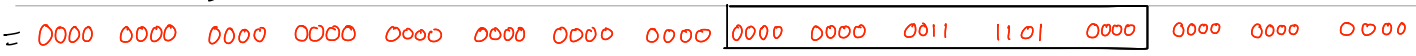
Syntax = LUI rd, constant → 20 bits

- 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} ⇒



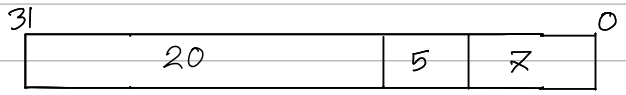
lui x_{10}, 976

x_{10} = 

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

x_{10} = 

U type instruction → LUI



* each field has unique name and size.