

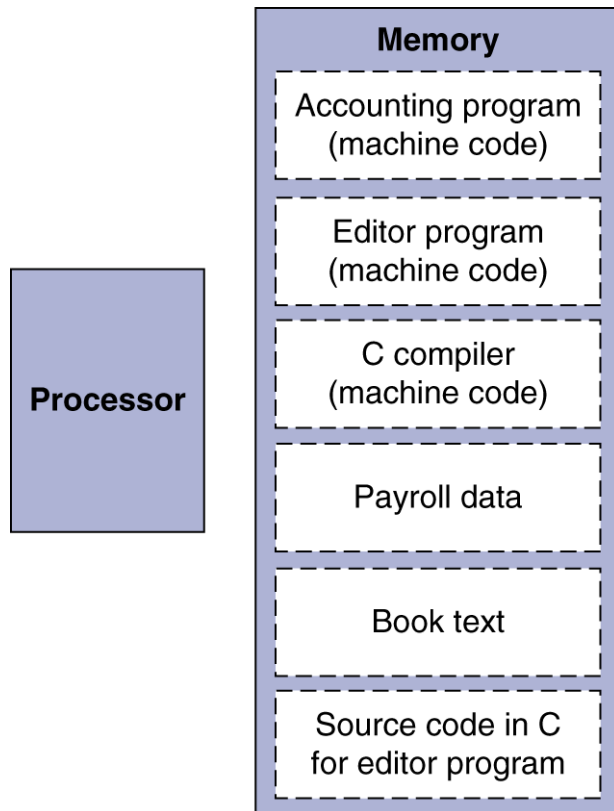


Topic 3

Assembly Programming - Function (Procedure) Call

Stored Program

The BIG Picture

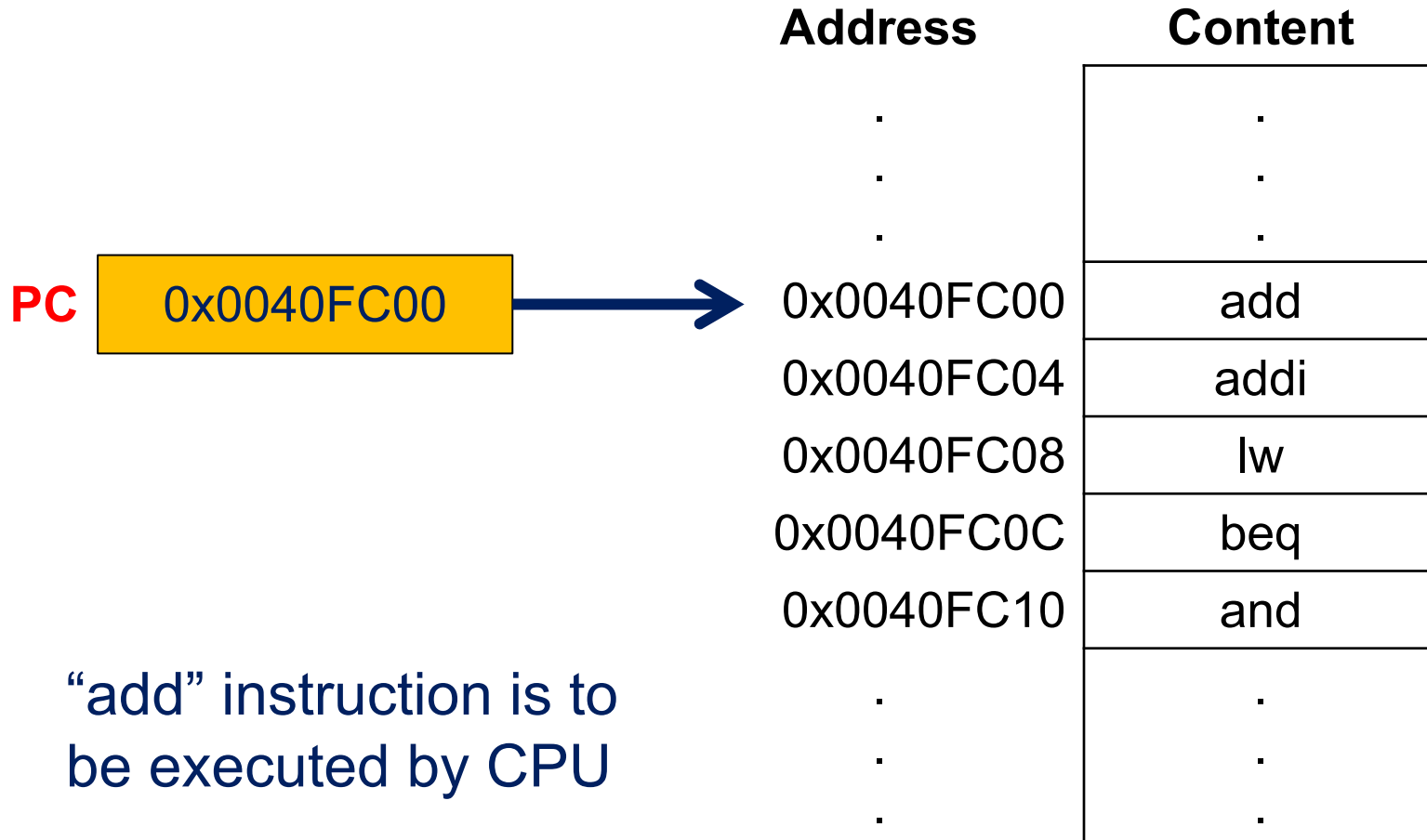


- Instructions are represented in binary, just like data
- Instructions and data are both stored in memory – **stored program**

Program Counter (PC)

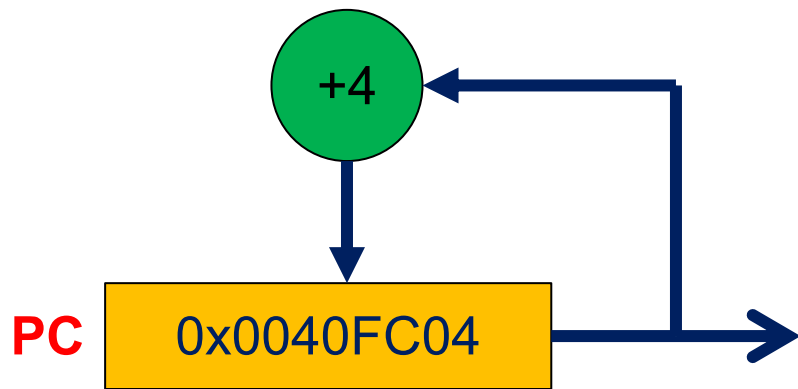
- Each instruction is stored as a 32-bit word in program memory
 - has an address
 - when labeled, the label is equal to the address
- PC holds address of an instruction to be executed
 - 32 bits register
 - Increased by 4 for RV32
- PC is a special register in CPU
 - Different from the registers in register file

Program Counter (PC)



Program stored in memory

Program Counter (PC)

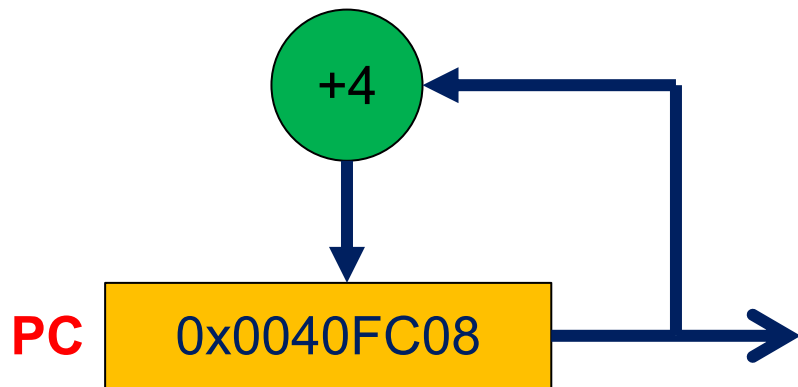


“addi” instruction is to be executed by CPU

Address	Content
.	.
.	.
.	.
0x0040FC00	add
0x0040FC04	addi
0x0040FC08	lw
0x0040FC0C	beq
0x0040FC10	and
.	.
.	.
.	.

Program stored in memory

Program Counter (PC)



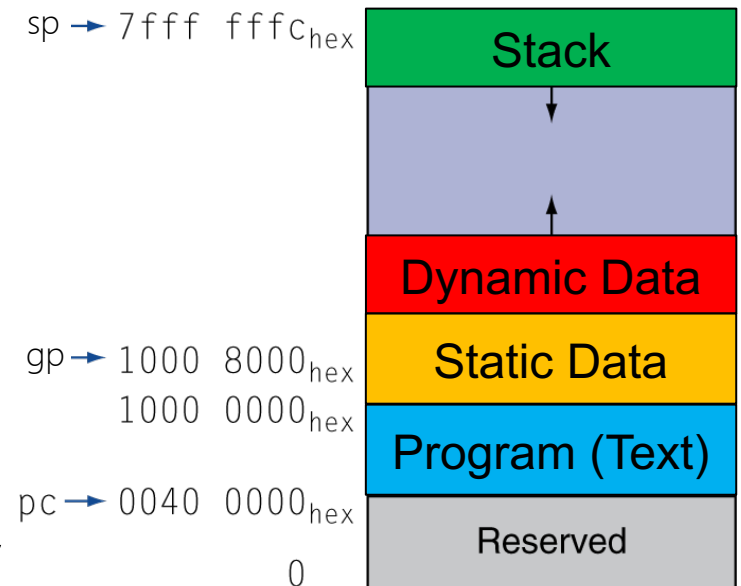
“lw” instruction is to be executed by CPU

Address	Content
.	.
.	.
.	.
0x0040FC00	add
0x0040FC04	addi
0x0040FC08	lw
0x0040FC0C	beq
0x0040FC10	and
.	.
.	.
.	.

Program stored in memory

Memory Layout

- Text: program code
- Static data: global/static variables
 - x3 (global pointer) initialized to the middle of this segment, 0x10008000 allowing \pm offset
- Dynamic data: heap
 - E.g., malloc in C, new in Java
- Stack: storage for temporary variable in functions
 - x2 (sp, stack pointer) initialized to 0x7ffffffc, growing towards low address

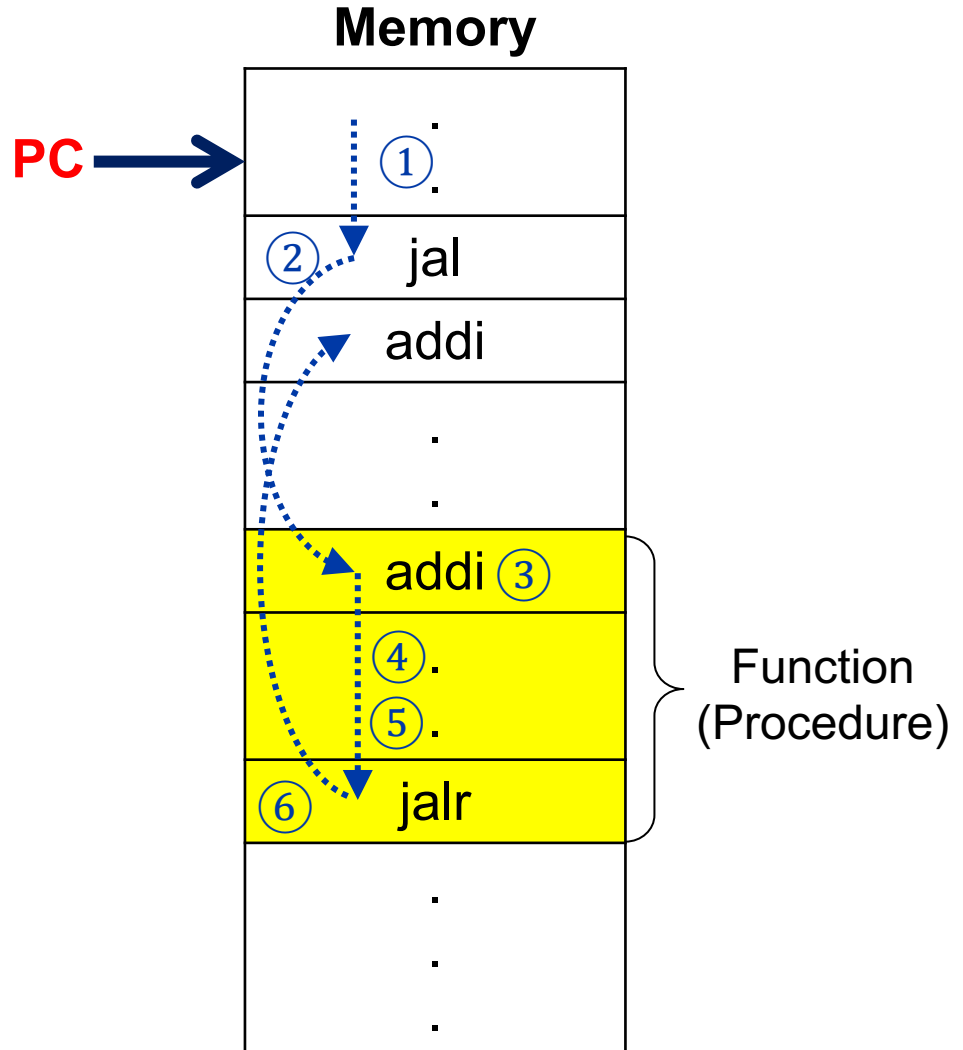


Function Calling

- Used to improve reusability and manageability
- Steps for function calling operation
 - ① Place parameters in registers x10 to x17
 - ② Transfer control to procedure
 - ③ Acquire storage on stack for procedure
 - ④ Perform procedure's operations
 - ⑤ Place result in register x10 and x11 for caller
 - ⑥ Return to place of call (address in x1)

Function Calling

- ① Place parameters in registers x10 to x17
- ② Transfer control to procedure
- ③ Acquire storage on stack for procedure
- ④ Perform procedure's operations
- ⑤ Place result in register x10 and x11 for caller
- ⑥ Return to place of call (address in x1)



Function Call Instructions

- Function call: jump and link

`jal x1, ProcedureLabel`

- $x1 \leq PC + 4$, $x1$ is called return address reg.
- $PC \leq \text{ProcedureLabel}$

- Function return: jump and link register

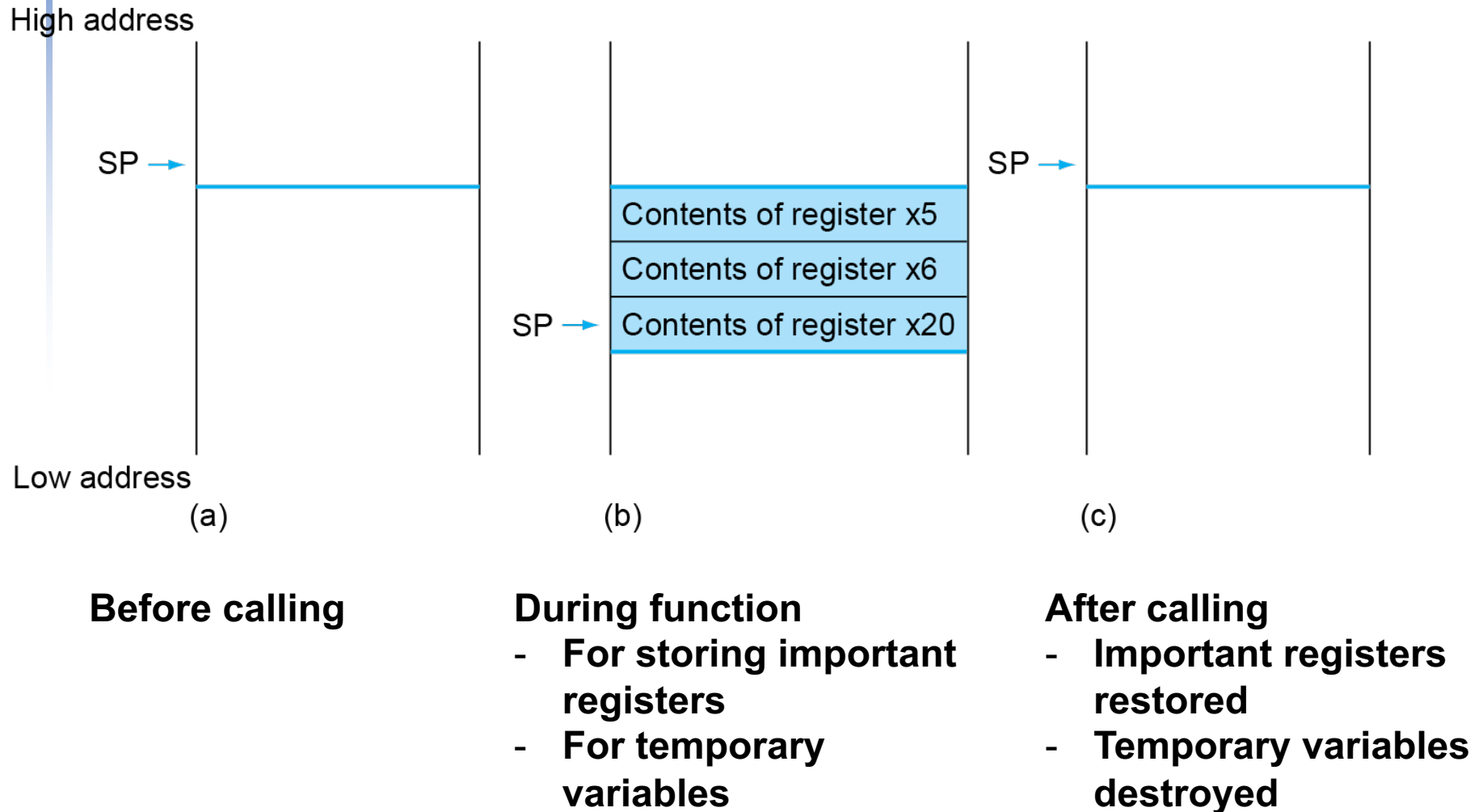
`jalr x0, offset(x1)`

- $x0 \leq PC + 4$ ($x0 \equiv 0$, nothing happens)
- $PC \leq \text{offset} + \text{return address stored in } x1$,
offset usually is 0 for function return
- Can also be used for computed jumps

Register Usage

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

Uses of Stack in Function Call



Register Usage

- x5 – x7, x28 – x31: temporary registers
 - Not preserved by the callee
- x8 – x9, x18 – x27: saved registers
 - If used, the callee saves and restores them

Leaf Function

- Functions that don't call other functions

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;}
```

- Assumptions:

- Arguments g, ..., j in x10, ..., x13
- f in x20
- temporaries x5, x6
- If we decide to save x5, x6, x20 all on stack

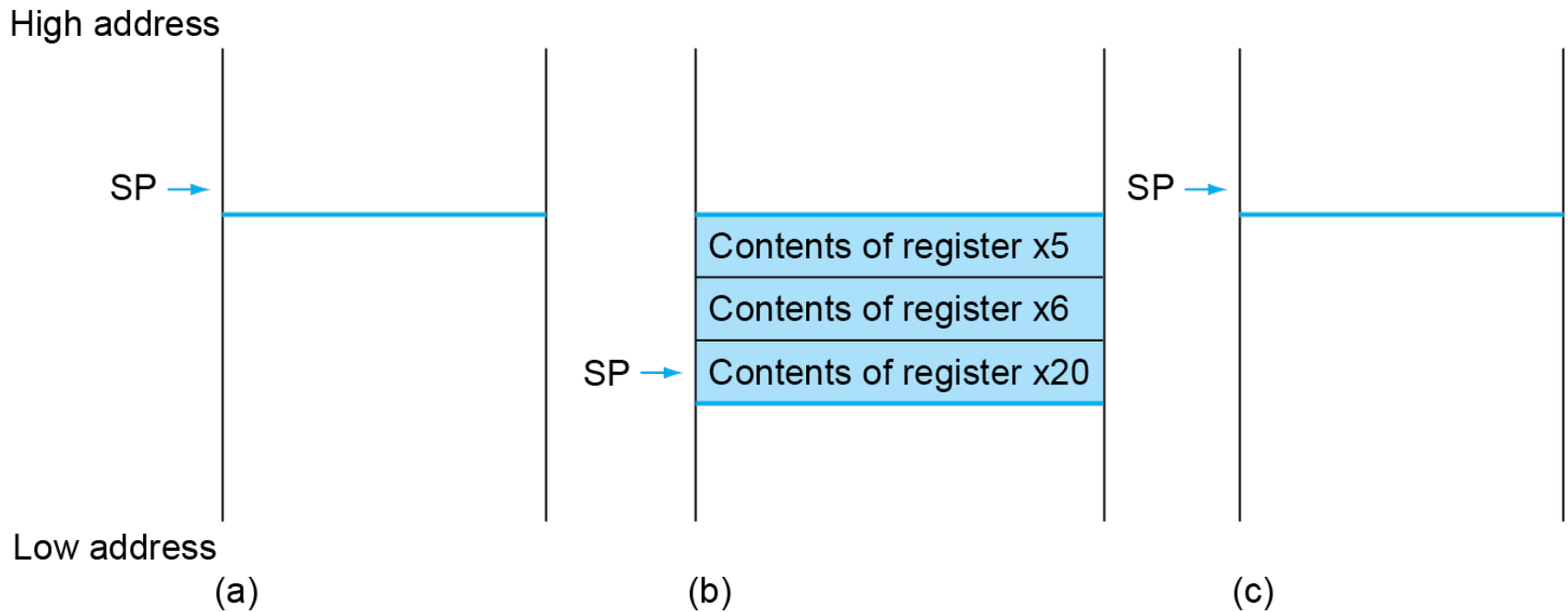
Leaf Function Example

■ RISC-V code:

leaf_example:

```
addi sp,sp,-12    #create spaces on stack ③
sw    x5,8(sp)    #Save x5, x6, x20 on stack
sw    x6,4(sp)
sw    x20,0(sp)
add    x5,x10,x11  #x5 = g + h
add    x6,x12,x1   #x6 = i + j ④
sub    x20,x5,x6   #f = x5 - x6
addi   x10,x20,0   #copy f to return register ⑤
lw     x20,0(sp)   #Resore x5, x6, x20 from stack
lw     x6,4(sp)
lw     x5,8(sp)
addi   sp,sp,12    #release space on stack
jalr   x0,0(x1)    #return to caller ⑥
```

Local Data on the Stack



Leaf Function Example

■ RISC-V code:

leaf_example:

addi sp,sp,-12

sw x5,8(sp)

sw x6,4(sp)

sw x20,0(sp)

add x5,x10,x11

add x6,x12,x1

sub x20,x5,x6

addi x10,x20,0

lw x20,0(sp)

lw x6,4(sp)

lw x5,8(sp)

addi sp,sp,12

jalr x0,0(x1)

#create spaces on stack

#Save x5, x6, x20 on stack

#only need to store saved regs

#x5 = g + h

#x6 = i + j

#f = x5 - x6

#copy f to return register

#Restore x5, x6, x20 from stack

#release space on stack

#return to caller

Unnecessary, because they are temporary registers, no need to save them by the callee

String Copy Example

- C code:

- Null-terminated string

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

String Copy Example

■ RISC-V code:

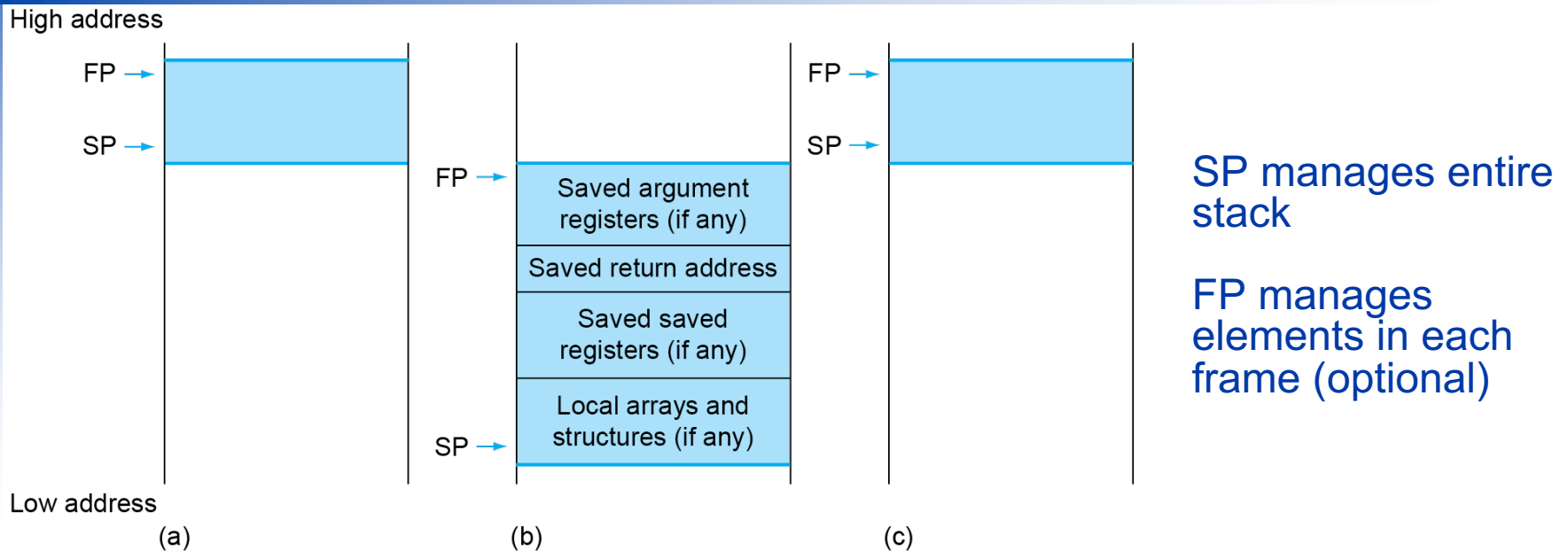
strcpy:

addi	sp,sp,-4	# adjust stack for 1 word
sw	x19,0(sp)	# push x19 - a saved register
add	x19,x0,x0	# i=0
L1:	add x5,x19,x10	# x5 = addr of y[i]
lbu	x6,0(x5)	# x6 = y[i]
add	x7,x19,x11	# x7 = addr of x[i]
sb	x6,0(x7)	# x[i] = y[i]
beq	x6,x0,L2	# if y[i] == 0 then exit
addi	x19,x19,1	# i = i + 1
jal	x0,L1	# next iteration of loop
L2:	lw x19,0(sp)	# restore saved x19
addi	sp,sp,4	# pop 1 word from stack
jalr	x0,0(x1)	# and return

Non-Leaf Functions

- Functions that call other functions
- For nested call, caller needs to save on the stack before calling another function:
 - Its return address
 - Any argument registers
 - Temporary registers needed after the call
- Restore from the stack after the call

Local Data on the Stack



- A frame (activation record) is temporary memory created for a function, it should always save:
 - Saved registers (x8, x9, x18-x27)
 - Local arrays and structures (if any)
- When it's a non-leaf function (caller) calling another function, it should also save:
 - Return address
 - Argument registers (if any)
 - Temporary registers (x5-x7, x28-x31) needed after the function call

Non-Leaf Function Example

- C code:

```
int fact (int n)
{
    if (n < 1) return f;
    else return n * fact(n - 1);
}
```

- Argument n in x10
- Result in x10

Non-Leaf Procedure Example

■ RISC-V code:

fact:

addi sp, sp, -8

Save return address and n on stack

sw x1, 4(sp)

sw x10, 0(sp)

addi x5, x10, -1

$x5 = n - 1$

bge x5, x0, L1

if $n \geq 1$, go to L1

addi x10, x0, 1

Else, set return value to 1

addi sp, sp, 8

Pop stack, don't bother restoring values

jalr x0, 0(x1)

Return

L1: addi x10, x10, -1

$n = n - 1$

jal x1, fact

call fact($n-1$)

addi x6, x10, 0

move result of fact($n - 1$) to x6

lw x10, 0(sp)

Restore caller's n

lw x1, 4(sp)

Restore caller's return address

addi sp, sp, 8

Pop stack

mul x10, x10, x6

return $n * \text{fact}(n-1)$

jalr x0, 0(x1)

return

More Examples: C Sort

- Illustrates use of assembly instructions for a C bubble sort function

- Swap procedure (leaf)

```
void swap(int v[],
          int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- v in x10, k in x11, temp in x5

The Procedure Swap

swap:

```
slli x6,x11,2      # reg x6 = k * 4
add  x6,x10,x6      # reg x6 = v + (k * 4)
                        # (address of v[k])
lw   x5,0(x6)       # reg x5 (temp) = v[k]
lw   x7,4(x6)       # reg x7 = v[k + 1]
sw   x7,0(x6)       # v[k] = reg x7 (v[k+1])
sw   x5,4(x6)       # v[k+1] = reg x5 (temp)
jalr x0,0(x1)       # return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1;
             j >= 0 && v[j] > v[j + 1];
             j -= 1) {
            swap(v, j);
        }
    }
}
```

- v in x10, n in x11, i in x19, j in x20

The Outer Loop

- Skeleton of outer loop:

- for ($i = 0; i < n; i += 1$) {

```
li    x19,0          # i = 0, pseudo instruction
```

```
for1tst:
```

```
bge   x19,x11,exit1  # go to exit1 if  $x19 \geq x11$  ( $i \geq n$ )
```

```
(body of outer for-loop)
```

```
addi  x19,x19,1      # i += 1
```

```
j     for1tst        # branch to test of outer loop
```

```
# pseudo instruction
```

```
exit1:
```

The Inner Loop

- Skeleton of inner loop:

- for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
 addi x20,x19,-1 # j = i - 1

for2tst:

```
blt  x20,x0,exit2  # go to exit2 if x20 < 0 (j < 0)
slli  x5,x20,2      # reg x5 = j * 4
add   x5,x10,x5     # reg x5 = v + (j * 4)
lw    x6,0(x5)      # reg x6 = v[j]
lw    x7,4(x5)      # reg x7 = v[j + 1]
ble   x6,x7,exit2   # go to exit2 if x6 ≤ x7, pseudo
mv     x21, x10      # copy parameter x10 into x21, pseudo
mv     x22, x11      # copy parameter x11 into x22
mv     x10, x21      # first swap parameter is v
mv     x11, x20      # second swap parameter is j
jal    x1,swap       # call swap
addi   x20,x20,-1    # j -= 1
j      for2tst       # branch to test of inner loop, pseudo
```

exit2: