Functions and Stacks

Sparsh Mittal

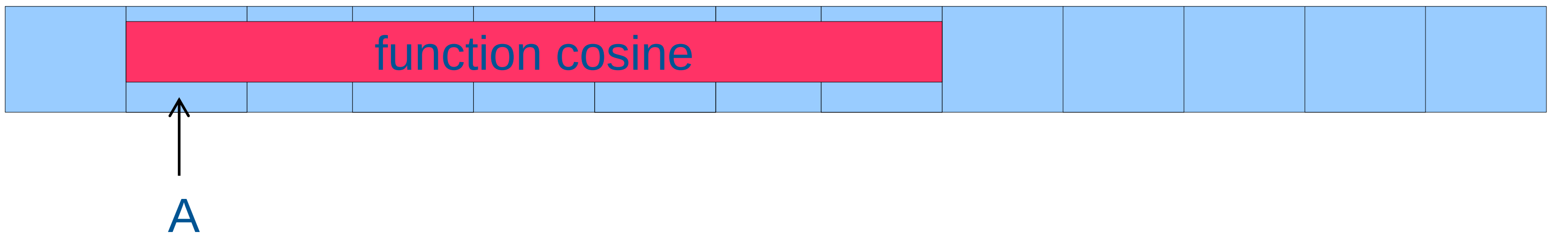
Courtesy for some slides: Smruti Ranjan Sarangi

Terms

- **caller**
- The program that instigates a procedure and provides the necessary parameter values.
- **callee**
- A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

Implementing Functions

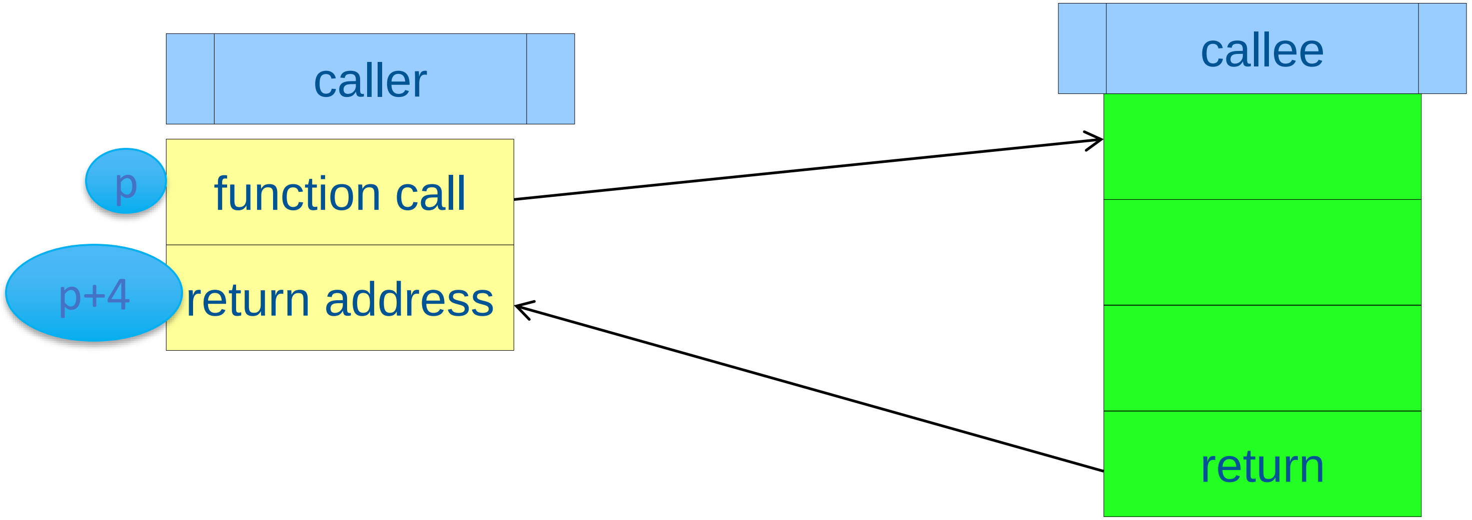
- Functions are blocks of assembly instructions that can be repeatedly invoked to perform a certain action
- Every function has a starting address in memory
- E.g., in below figure, **cosine** has a starting address A



Implementing Functions - II

- To call a function, we need to set :
  - pc $\leftarrow$ A
- We also need to store the location of the pc that we need to come to after the function returns
- This is known as the return address
- We can thus call any function, execute its instructions, and then return to the saved return address

Notion of the Return Address



- PC of the call instruction → p
- PC of the return address → p + 4

because, every instruction takes 4 bytes

Control flow instructions

Example instruction	Instruction name	Meaning
`jal x1,offset`	Jump and link	Regs[x1]←PC+4; PC←PC + (offset<<1)
`jalr x1,x2,offset`	Jump and link register	Regs[x1]←PC+4; PC←Regs[x2]+offset

Pseudo instructions

`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, rs, 0`	Jump register
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`ret`	`jalr x0, x1, 0`	Return from subroutine

RISC-V instructions for procedure call and return

- Procedure call: jump and link

```
jal x1, ProcedureLabel
```

1. Address of the following instruction is saved in x1 (also called ra or return address register)
2. Jumps to the target address `ProcedureLabel`

- Procedure return:

```
ret
```

  - Jumps to address in x1
  - ret is same as “jalr x0 x1 0”

Consider a C-function

```
int main() {
float a=2;        //Address= 1020
x=cosine (a);  //Address= 1024
cout<<x;        //Address= 1028
}
float cosine(float x) {
  some instruction  //PC=5008
}
```

Solved example

- We want to jump to a function that has a label `COSINE' and is stored at address 5008 in memory.
- At PC=1024, the following instruction appears:
- jal x1, COSINE
- On executing this instruction, what will be value of
- (i) PC and (ii) x1
- Answer: PC will store the address of COSINE (5008).
- x1 will store 1028.

How to pass arguments/ return values

- Solution : use registers

```
.func:
    add a0, a0, a1
    ret
.main:
    li a0, 3
    li a1, 5
    jal x1, .func
    addi a2, a0, 10
```

Before calling a function, arguments are copied to registers a0 and a1.
a0 is same as x10
a1 is same as x11

Return value is stored in a0 itself

Argument registers

x10 to x17 are used to pass arguments to a function.

If more than 8 arguments need to be passed, we use the stack.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Solved example

C code

```
int foo () {
return 2;
}

void main () {
int x = 3;
int y = x + foo ();
}
```

RISC-V code

```
.foo: # callee
li a0 ,  2              # a0 = 2
jalr zero , 0(ra)   # return inst .

. main :
li s0, 3                 # s0 = 3
jal ra , .foo           # jump to .foo
add s1 , s0 , a0     # y = x + foo ()
# s1 contains the result
```

Program to compute x^n

- x is in a1, n in a2, result in a0

```
.power :
        addi a0 , zero , 1 # a0 will contain the result
        add t1 , zero , a2 # t1 = n
        beq t1 , zero , .end # check (n == 0)
.loop :
        mul a0 , a0 , a1 # result *= x
        addi t1 , t1 , -1 # decrement n
        bne t1 , zero , .loop
        jalr zero , 0(ra) # return
.main :
        addi a1 , zero , 7 # x = 7
        addi a2 , zero , 3 # n = 3
        jal ra , .power # call the power function
```

Limitations with use of registers for argument passing or returning results
AND
How to address them

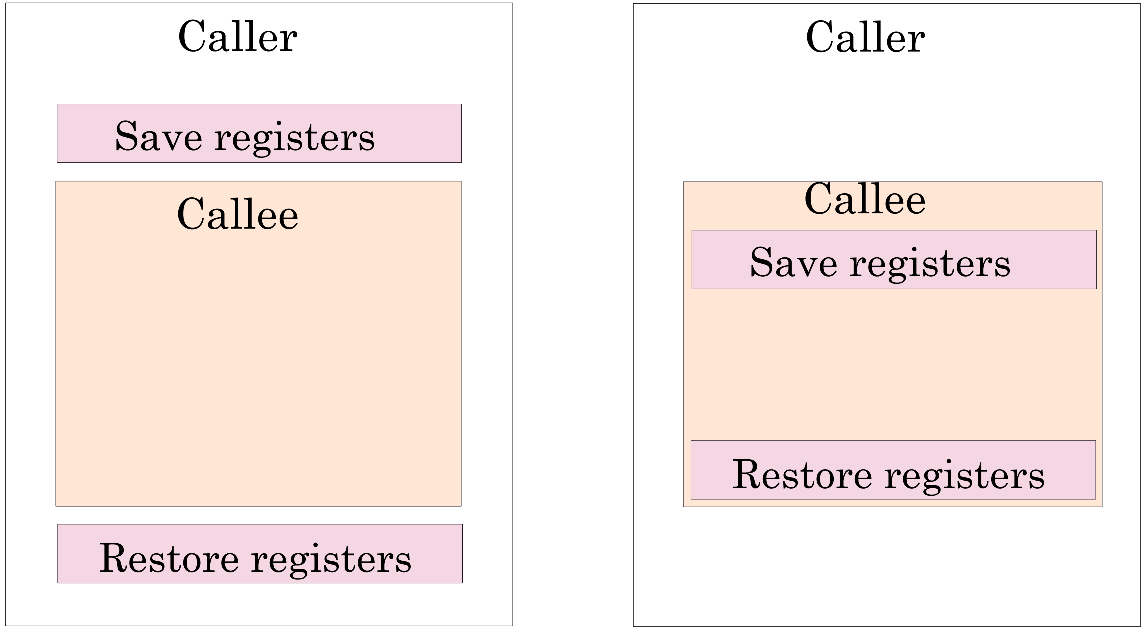
Limitations with use of registers for argument passing or returning results

* Space Problem
  * We have a limited number of registers
  * We cannot pass more than certain number of arguments
  * Solution : Use memory also
* Overwrite Problem
  * What if a function calls itself ? (recursive call)
  * The callee can overwrite the registers of the caller
  * Solution : Spilling

Register Spilling

* caller saved scheme
  * The caller can save the set of registers its needs
  * Call the function
  * And then restore the set of registers after the function returns
  * Known as the caller saved scheme
* callee saved scheme
  * The callee saves the registers, and later restores them

caller or callee-saver conventions



(a) Caller saved (b) Callee saved

Saver is "caller": means that a function caller must save that register somewhere before calling

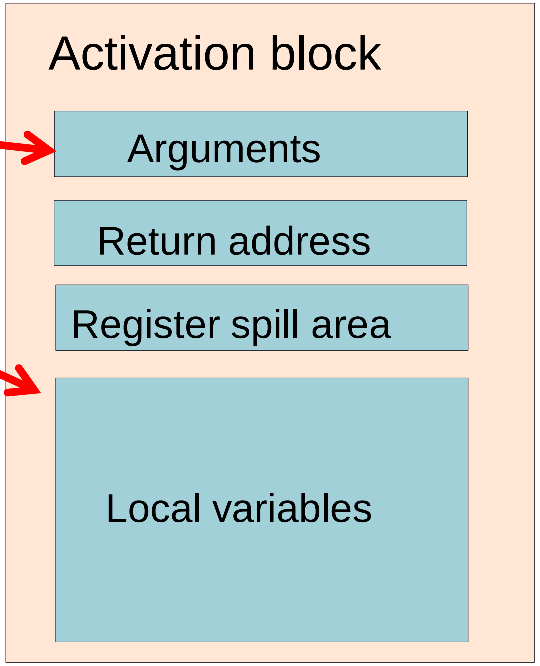
Saver is "callee": means that if a function wants to use that register, it must first save it somewhere, and restore it before returning

Limitations with our approach

- Using memory, and spilling solves both the space problem and overwrite problem
- However, there needs to be :
  - a strict agreement between the caller and the callee regarding the set of memory locations that need to be used
  - Secondly, after a function has finished execution, all the space that it uses needs to be reclaimed

Activation Block

```
int foo(int arg1) {
        int a, b, c;
        a = 3, b = 4;
        c = a + b + arg1;
        return c;
}
```



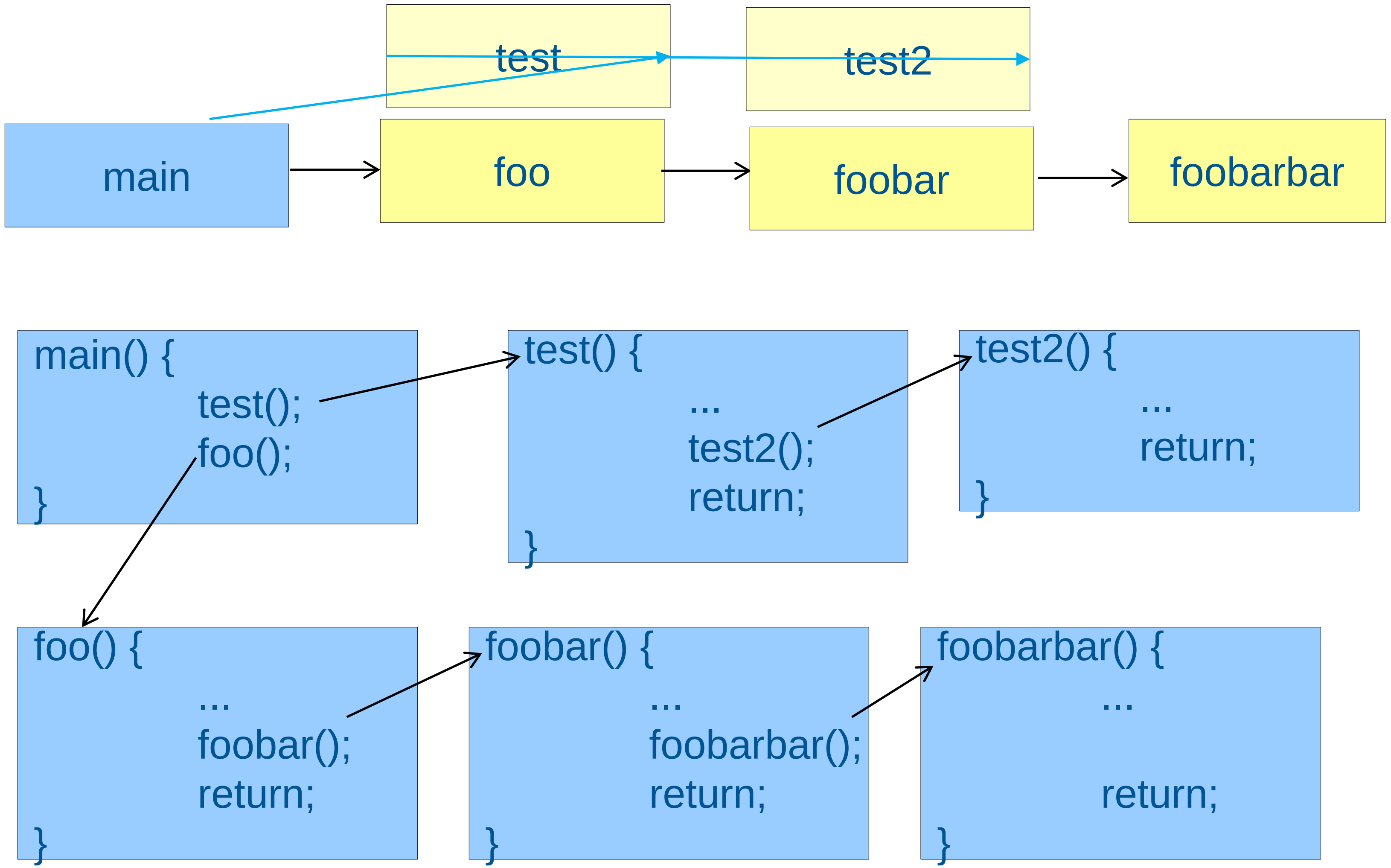
- Activation block → memory map of a function

  arguments, register spill area, local variables

Organising Activation Blocks

- All the information of an executing function is stored in its activation block
- These blocks need to be dynamically created and destroyed – millions of times
- What is the correct way of managing them, and ensuring their fast creation and deletion ?
- Is there a pattern ?

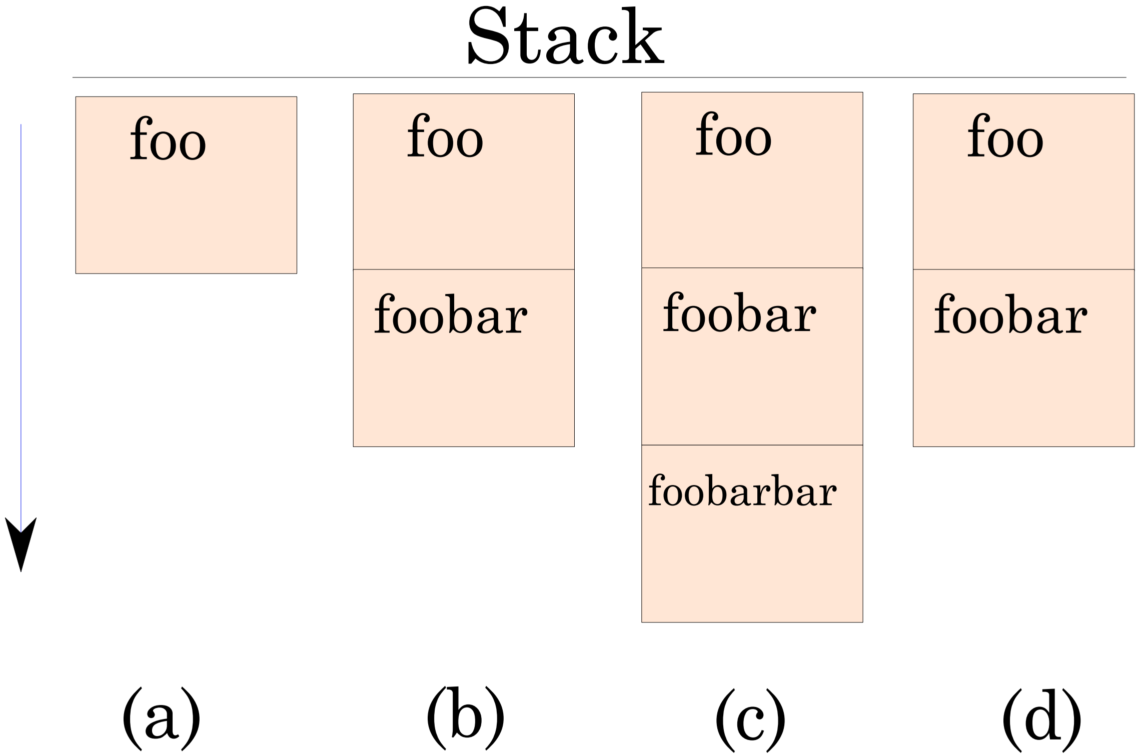
Pattern of Function Calls



Pattern of Function Calls

- Last in First Out

Use a stack to store activation blocks



Issues solved by stack

- Space problem
  - Pass as many parameters as required in the activation block
- Overwrite problem
  - Solved by activation blocks
- Management of activation blocks
  - Solved by the notion of the stack

Working with the Stack

- Allocate a part of the memory to save the stack
- Traditionally stacks are downward growing.
  - The first activation block starts at the highest address
  - Subsequent activation blocks are allocated lower addresses
- The stack pointer register (sp) points to the beginning of an activation block
- Allocating an activation block : sp $\leftarrow$ sp - <constant>
- De-allocating an activation block: sp $\leftarrow$ sp + <constant>

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Saving variable in stack (pushing and popping)

```
myFunction:
    addi sp,sp,-24
    sd   x5,16(sp)
    sd   x6,8(sp)
    sd   x20,0(sp)
    add  x5,x10,x11
    add  x6,x12,x13
    sub  x20,x5,x6
    addi x10,x20,0
    ld   x20,0(sp)
    ld   x6,8(sp)
    ld   x5,16(sp)
    addi sp,sp,24
    ret
```

Save x5, x6, x20 on stack (called pushing)

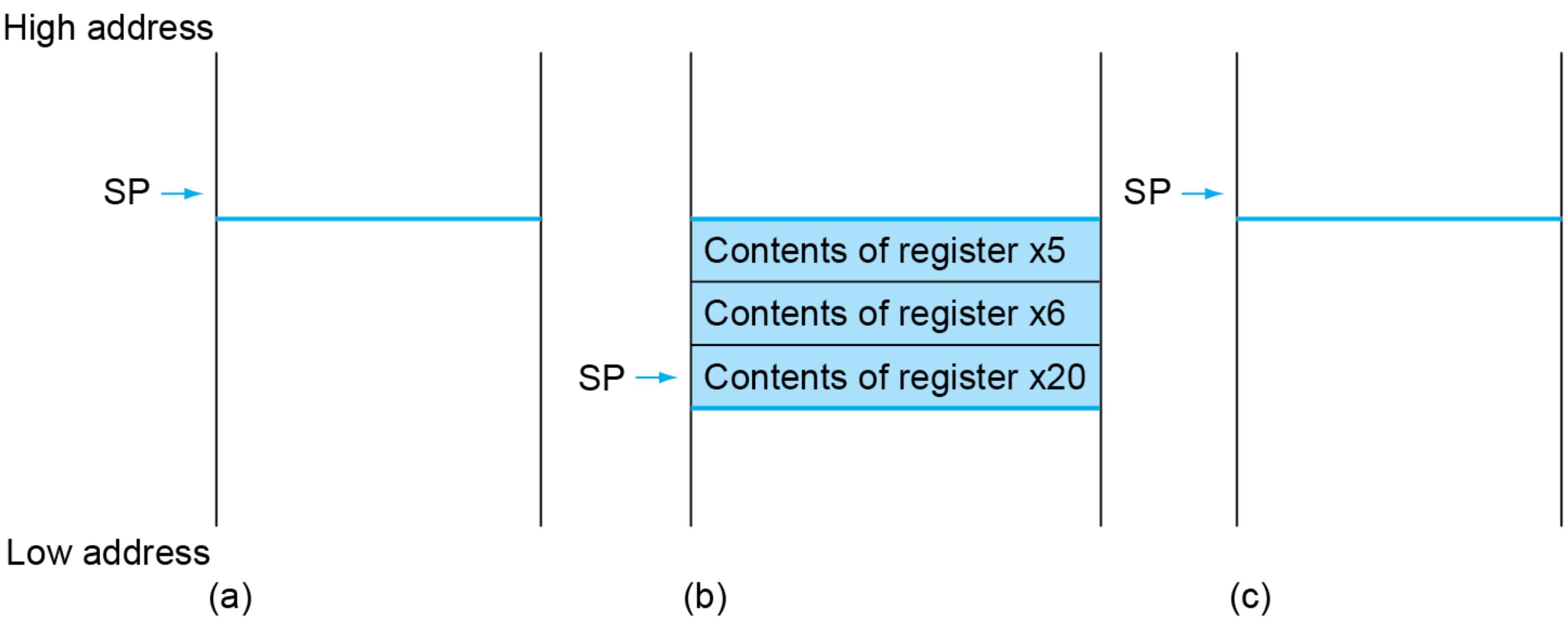
Do some processing of a function

Resore x5, x6, x20 from stack (called popping)

Return to caller

This is an example of callee saved scheme

How Stack Functions



Stack grows downwards

A Question on Stack

Consider 32b version of RISC-V and see below code.

```
addi sp,sp,-12
sw   x5,8(sp)
sw   x6,4(sp)
sw   x20,0(sp)
```

Initially, we have value of
x5 as 0x12345678,
x6 as 0xABCDEF09
x20 as 0xACE02468.
sp is 1024.

In this figure, write the values shown by ?

Byte addresses	
1011	
1012	
1013	
1014	?
1015	?
1016	?
1017	?
1018	?
1019	?
1020	?
1021	?
1022	
1023	
1024	
1025	
1026	

Solution

RISC-V is little endian
Hence, (a) is correct for RISC-V

Byte addresses	
1011	
1012	68
1013	24
1014	**E0**
1015	**AC**
1016	**09**
1017	**EF**
1018	**CD**
1019	**AB**
1020	**78**
1021	**56**
1022	34
1023	12
1024	
1025	
1026	

(a) Little Endian

Byte addresses	
1011	
1012	AC
1013	E0
1014	**24**
1015	**68**
1016	**AB**
1017	**CD**
1018	**EF**
1019	**09**
1020	**12**
1021	**34**
1022	56
1023	78
1024	
1025	
1026	

(b) Big Endian

(b) is correct for any Big-endian ISA

Convert C-code to RISC-V

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

- Arguments g, h, i, j in x10, x11, x12, x13
- f in x20
- temporaries x5, x6
- Need to save x5, x6, x20 on stack

Converted code in RISC-V code

```
myFunction:
    addi sp,sp,-24
    sd   x5,16(sp)
    sd   x6,8(sp)
    sd   x20,0(sp)
    add  x5,x10,x11
    add  x6,x12,x13
    sub  x20,x5,x6
    addi x10,x20,0
    ld   x20,0(sp)
    ld   x6,8(sp)
    ld   x5,16(sp)
    addi sp,sp,24
    ret
```

Save x5, x6, x20 on stack

x5 = g + h

x6 = i + j

f = x5 – x6

copy f to return register

Resore x5, x6, x20 from stack

Return to caller