

Functions



Caiwen Ding

Department of Computer Science and Engineering
University of Connecticut

CSE3666: Introduction to Computer Architecture

Outline

- Function basics
 - Write a function
 - Call a function
 - Calling convention
- Using stack to save/restore data

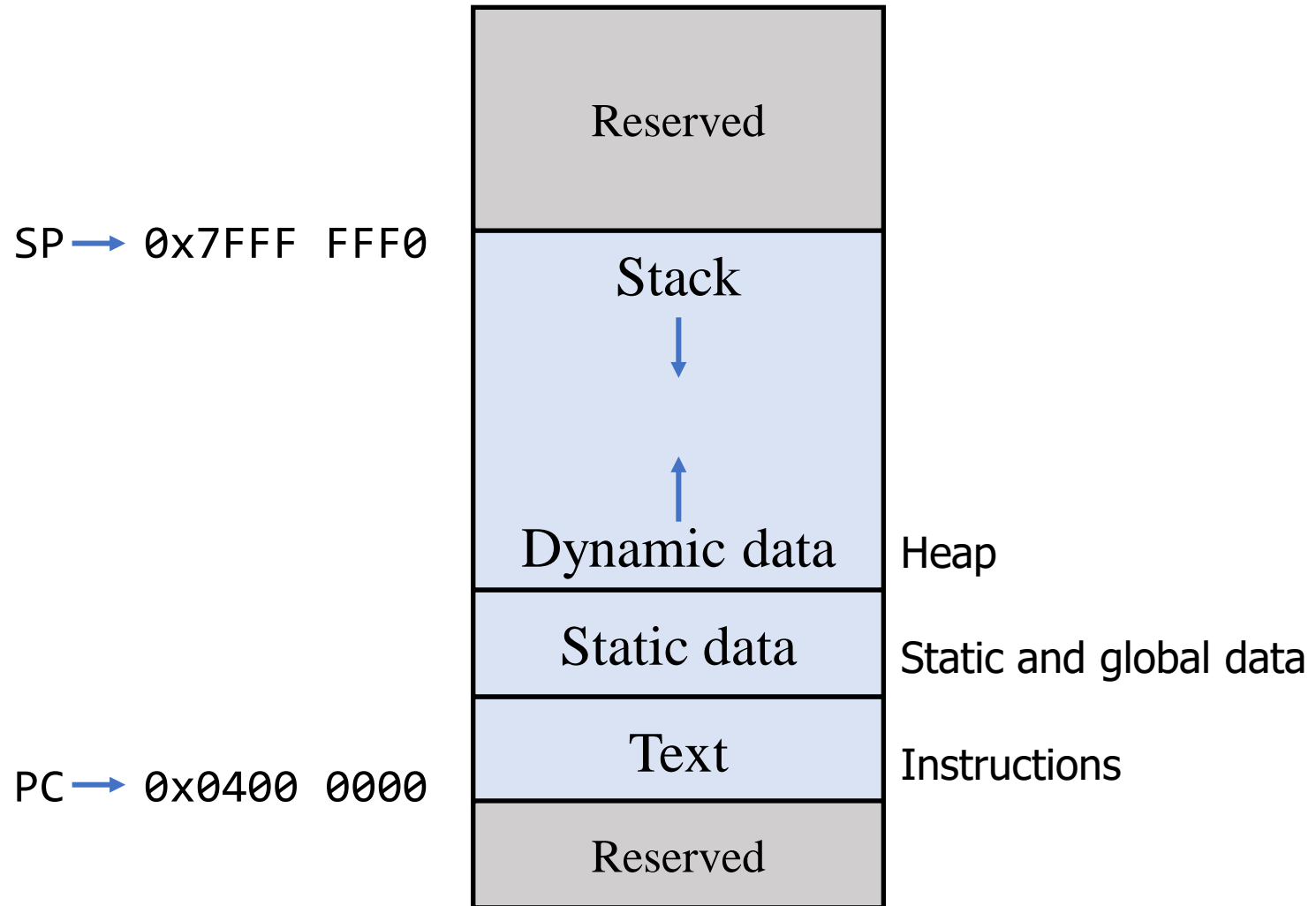
Reading: Sections 2.8

References: Reference card in the book

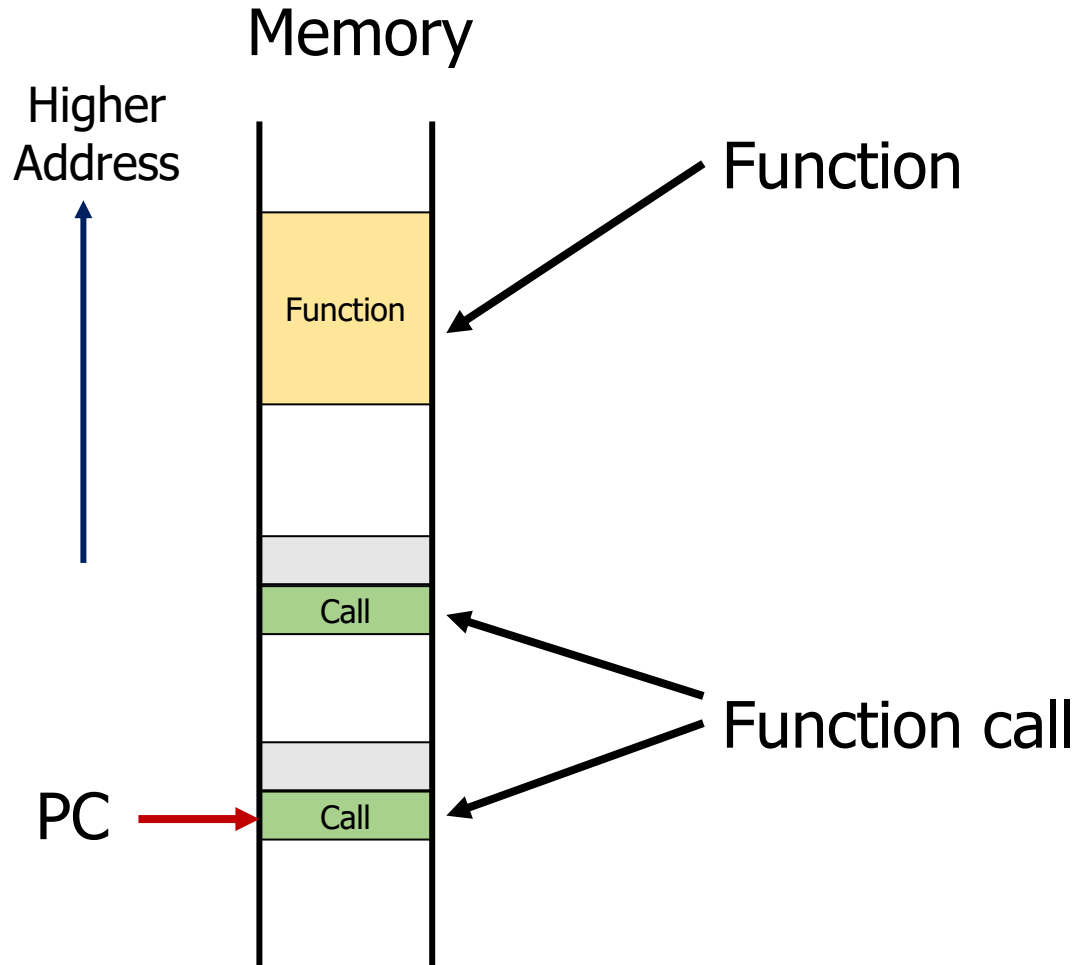
Functions

- Make code manageable
 - Hierarchical design
- Reduce code size
 - You do not have to write the same code again and again
- Easy to maintain
 - No need to find all the copies of the code to fix a bug
 - New versions of functions are called if the library is dynamically linked
- ...

Memory Layout

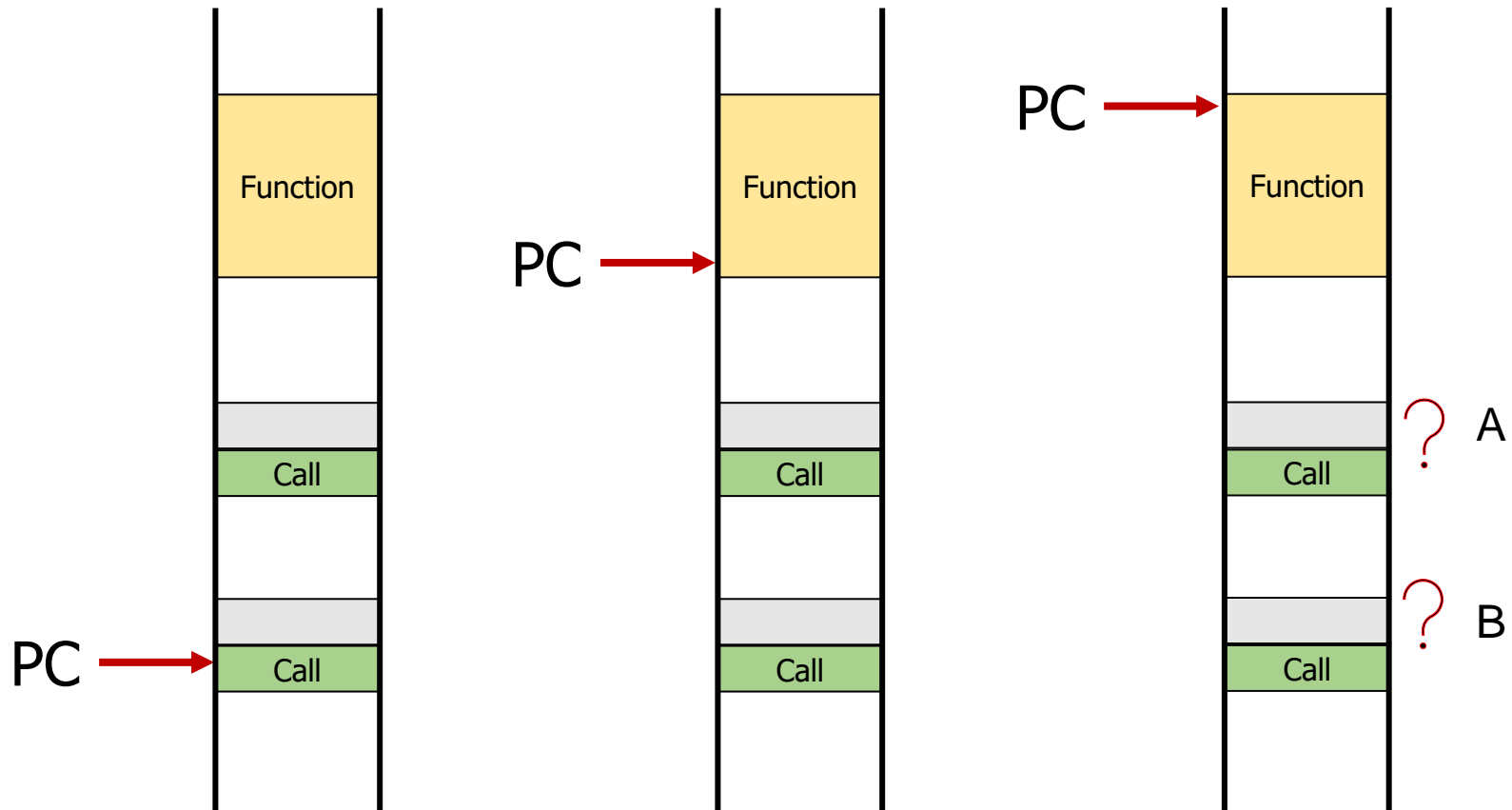


Flow with functions



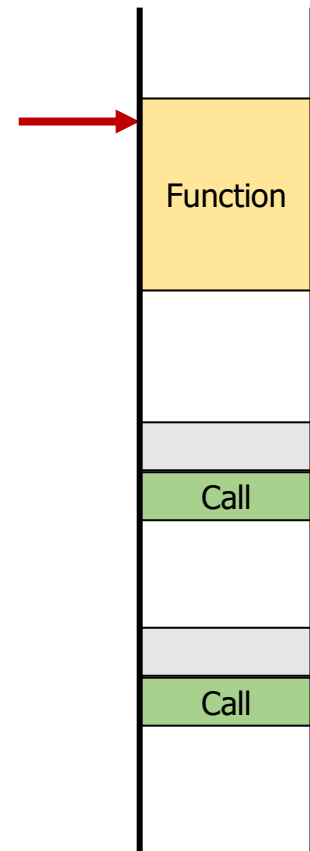
Flow with functions

- Upon function call, processor starts to execute instructions in functions
- When function returns, **where should the processor get the next instruction?**



Flow with functions

- Processor **needs to know the return address** to resume flow
 - Branches won't work!
- Processor should resume from the instruction that follows the function call



RISC-V Procedure Call Instructions - JAL

`jal rd, function_name`

Function name is just a label,
indicating a memory address



- Procedure call: **jump and link**
 - Go to the label
 - And save **the address of the following instruction (PC+4)** in **rd**
- **rd** is the return address
 - If the return address is not needed, use **x0** as **rd**
- **Pseudoinstruction J** is a JAL writing to **x0**
 - There is no separate jump instruction in RISC-V

Save return address in ra (or x1) in function calls.

How do we return?

- Now the return address is in a register, e.g., ra. How can we actually go there?
- Return means "jump to the return address saved in a register". JAL instruction does not have a source register.
- We need another instruction that can compute the jump target using the address in a register

JALR

JALR

`jalr rd, offset(rs1)` # supported in RARS and gcc

- Procedure call or return: **jump and link to address in a register**
 - Go to an address stored in rs1 plus an offset
 - Saves **the address of the following instruction (PC+4)** in rd
- The return address is saved in rd
 - If the return address is not needed, use x0 as rd

Although JAL and JALR can save PC+4 in any register,
Save return address in ra (or x1) in function calls.

Another way to write the instruction: “jalr rd, rs1, offset”

Use JALR

- We can use JALR to return from a function

```
jalr    x0, 0(ra)
```

```
# the following are pseudoinstructions
```

```
jr      ra
```

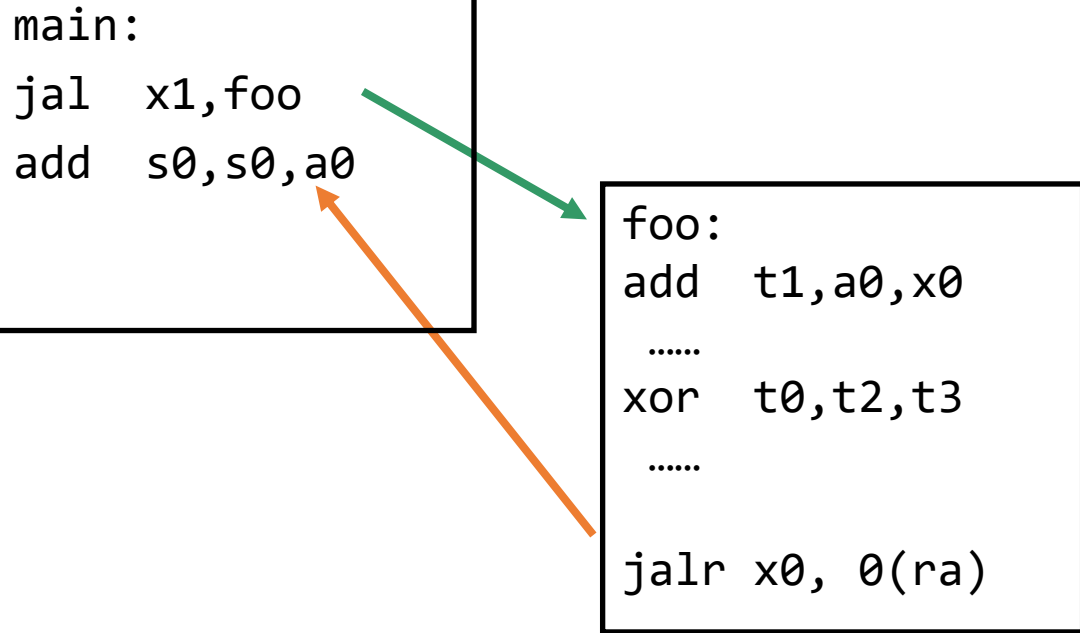
```
ret
```

- We can call a function that is far away
 - Place the function's address in a register
 - JAL cannot jump too far
- We can also change the function we calls
 - Function pointers, and virtual functions

Function Call Examples

main:

```
jal x1,foo  
add s0,s0,a0
```



foo:
add t1,a0,x0
.....
xor t0,t2,t3
.....
jalr x0, 0(ra)

ra = main + 4

main() is the caller. foo() is the callee().

foo() is a **leaf function/procedure**, which does not call other functions

Example

- Write a function to calculate the absolute value of a word

```
int    abs (int    n)
{
    int    rv = n;
    if (n < 0)
        rv = -n;
    return rv;
}
```

How does the function get the argument?

How does the function pass the return value to the caller?

Passing parameters and returning values

- First eight parameters/arguments are passed to the function in

a0, a1, a2, a3, a4, a5, a6, a7

- Two values can be returned in

a0, a1

Do not have to use all.

A function that returns only one value just needs a0.

Now, we can continue

Label the entry point



```
# RISC-V
abs:
```

```
int abs (int n)
{
    int rv = n;
    if (n < 0)
        rv = -n;
    return rv;
}
```

Translate the first statement:

n is in a0

Which register do we assign to rv?

abs function

```
abs:  # when the function is called, a0 is n
      bge    a0, x0, exit      # n < 0?
      sub    a0, x0, a0        # rv = -n
exit:
      jalr   x0, 0(ra)         # return
```

```
# calling abs(-2)
addi   a0, x0, -2             # set a0
jal    ra, abs                # call abs
# a0 should be 2 here
```


Nested Function Call Examples

main:

```
jal x1,foo  
add s0,s0,a0
```

ra = main + 4

foo:

```
add t1,a0,x0
```

...

```
jal x1,bar
```

```
xor t0,a0,t1
```

...

```
jalr x0, 0(ra)
```

ra is addr of XOR

bar:

...

```
add t1,t0,s0
```

...

```
jalr x0, 0(ra)
```



Do you see any problems?

What registers can be used by callee?

- Suppose F calls G. F is the caller and G is the callee
- What registers can G use?
 - F wants some registers to be preserved through function call. G should not change them
 - G needs registers to do its job

Callee does not need to preserve t0-t6 and a0-a7

Callee needs to preserve other registers (how?)

Preserved	Not preserved
Saved registers: x8-x9, x18-x27	Temporary registers: x5-x7, x28-x31
Stack pointer register: x2(sp)	Argument/result registers: x10-x17
Frame pointer: x8(fp)	
Return address: x1(ra)	
Stack above the stack pointer	Stack below the stack pointer

Stack

- Stack grows from higher address toward lower address
 - `sp` is the address of the word at the top of the stack
- Two operations:
 - push adds data to the top of the stack
 - pop removes data from the top of the stack
- What can stack be used for?
 - Saving registers
 - Keeping local variables (used by a function)
 - Passing Arguments
 - Returning values

Stack operations

#push x1

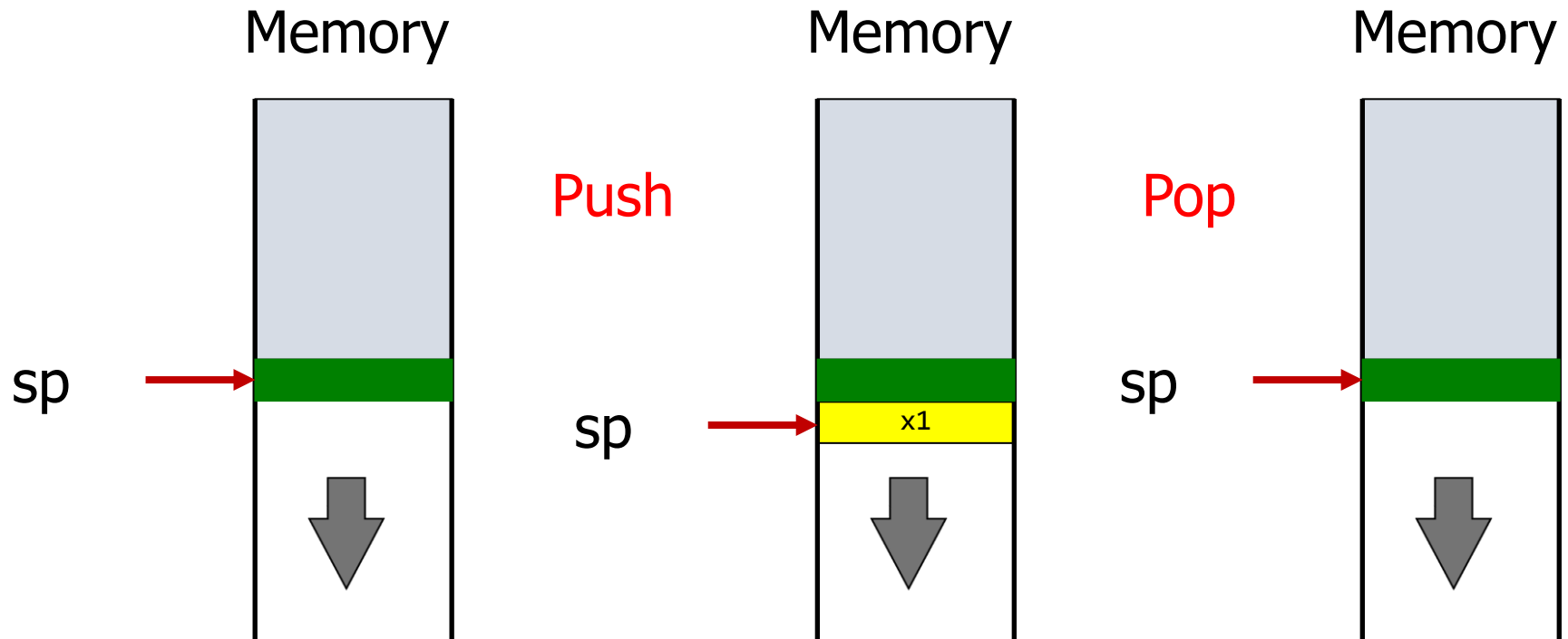
addi sp, sp, -4

sw x1, 0(sp)

pop x1

lw x1, 0(sp)

addi sp, sp, 4



Exercises

- Write RISC-V instructions for the following operations

Push ra onto the stack

Pop ra from the stack

Push s1, s2, and s3 onto the stack

How many instructions do you need for each of them?

Example: push multiple words

- Write RISC-V instructions to push s1, s2, and s3 onto the stack

Instead of 3 ADDI's, we can adjust sp once for three words

reserve space for 3 words

addi sp, sp, -12

sw s1, 8(sp)

sw s2, 4(sp)

sw s3, 0(sp)

Before push
sp →

sp →

Address	Value
0x7FFF 901C	
0x7FFF 9018	
0x7FFF 9014	
0x7FFF 9010	s1
0x7FFF 900C	s2
0x7FFF 9008	s3
0x7FFF 9004	
0x7FFF 9000	

Push and pop pairs

push s1, s2, and s3

addi sp, sp, -12

sw s1, 8(sp)

sw s2, 4(sp)

sw s3, 0(sp)

pop s1, s2, and s3

lw s1, 8(sp)

lw s2, 4(sp)

lw s3, 0(sp)

addi sp, sp, 12

restore sp after loads

sp →

Address	Value
0x7FFF 901C	
0x7FFF 9018	
0x7FFF 9014	
0x7FFF 9010	s1
0x7FFF 900C	s2
0x7FFF 9008	s3
0x7FFF 9004	
0x7FFF 9000	

Nested Function Call Examples

main:

```
jal x1,foo  
add s0,s0,a0
```

ra = main + 4

foo:

```
addi t1,a0,10
```

...

```
jal x1,bar
```

```
xor t0,a0,t1
```

...

```
jalr x0, 0(ra)
```

ra is addr of xor

bar:

...

```
add t1,t0,s0
```

...

```
jalr x0, 0(ra)
```



To fix the problem, we need to save ra and t1 in foo()

Example: fact

- C code:

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

Argument n in a0

Result in a0

Fact – Will this work?

fact:

```
# if (n < 1) return 1
addi t0, zero, 1
bge a0, t0, ELSE    # go to else branch if n >= 1
```

```
# if branch
addi a0, zero, 1    # set return value 1
beq x0, x0, EXIT    # go to exit
```

```
ELSE:    # else n * fact(n - 1)
addi a0, a0, -1    # compute fact(n-1)
jal ra, fact
mul a0, a0, a0    # n * (n - 1)!
```

```
EXIT:
jalr x0, 0(ra)    # Use only one exit !
```

RISC-V code for fact

```
fact:  # One entrance and one exit
      addi sp, sp, -8      # Save return address and n
      sw   ra, 4(sp)
      sw   a0, 0(sp)
      addi t0, zero, 1
      bge  a0, t0, ELSE    # if n >= 1, go to else branch
      addi a0, zero, 1     # return 1
      beq  x0, x0, EXIT    # goto exit

ELSE:
      addi a0, a0, -1      # recursive call with n - 1
      jal  ra, fact
      lw   t0, 0(sp)       # restore original n
      mul  a0, a0, t0      # n * (n - 1)!
EXIT:  lw   ra, 4(sp)       # restore return address
      addi sp, sp, 8       # restore sp
      jalr x0, 0(ra)       # and return
```

Local storage

- Local storage is allocated on the stack

```
int foo()
{
    // keep both a and b on stack
    int a[2], b;
    ...
}
```

```
foo:
# reserve space for 4 words
addi    sp, sp, -16
sw      ra, 12(sp)
```

```
# sp is 0x7FFF_9008 after addi
# locate a and b on stack?
# where is ra stored?
```

Address	Value
0x7FFF 901C	
0x7FFF 9018	
0x7FFF 9014	
0x7FFF 9010	
0x7FFF 900C	
0x7FFF 9008	
0x7FFF 9004	
0x7FFF 9000	

Passing arrays or strings to a function

- The starting address of the array is passed to functions
 - Changes to the array are preserved after the function returns
 - An ASCII string is an array of bytes, ending with a null character

```
int  array_max (int arr[], int length);  
unsigned int strlen (char s[]);
```

The caller puts the starting address of arr or s in a0.

Example of passing a string to a function

```
unsigned int strlen (char s[]);
```

The caller puts the correct address in a0.

For example,

```
strlen(a_string)
```

```
la    a0, a_string
# a0 = 0x00FE9000
jal   ra, strlen
```

a_string

Address	Value
0x00FE 9007	0
0x00FE 9006	54
0x00FE 9005	54
0x00FE 9004	54
0x00FE 9003	51
0x00FE 9002	69
0x00FE 9001	83
0x00FE 9000	67

Example: strlen

- C function strlen() returns length of a (null-terminated) string

```
unsigned int strlen (char s[])
```

```
{  
    unsigned int i;  
    i = 0;  
    while (s[i] != 0)  
        i += 1;  
    return i;  
}
```

strlen() does not call other functions.
It is a **leaf procedure**.

- When implementing the function with RISC-V,
 - Where do we find the address of the string?
 - Where do we keep i? a, t, or s registers?

RISC-V strlen

```
strlen:      # a0 is the starting address
            add    t1, 0, 0                # i = 0
loop:
            add    t0, t1, a0              # addr of s[i]
            lb     t0, 0(t0)               # load a byte
            beq    t0, x0, exit
            addi   t1, t1, 1                # i += 1
            beq    x0, x0, loop             # goto loop
exit:
            addi   a0, t1, 0                # set return value
            jalr   x0, 0(ra)               # return
```


Question

- Which register is used to pass parameters to a function?
- A. a0
- B. t0
- C. s0
- D. sp

Question

- If a function returns a value, where can the caller find it?
- A. `a0`
- B. `t0`
- C. `s0`
- D. `sp`

Question

- Which register should a function save before changing it?
- A. a0
- B. t0
- C. s0
- D. sp

Question

- When implementing the following function with RISC-V assembly language, in which register is argument c stored at the entry of the function?

```
int  foo(int  a[], int b, char c[], int d);
```

- A. a0
- B. a1
- C. a2
- D. a3

Question

- Which register is better to keep i?

A. t0

B. s0

```
int    foo()
{
    int  i;

    // some instructions

    for (i = 0; i < 10; i ++)
        bar(i);

    // more instructions
}
```

RISC-V Calling Convention

- Parameter passing
 - The first eight arguments are in a0, a1, a2, a3, a4, a5, a6, and a7
 - More arguments are placed on stack
- Register preservation and restore
- Return values
 - Return values are placed in a0 and a1

Assume argument registers (like a0 and a1) and temporary registers (like t0 and t1) are changed during any function call although they are not in some cases

RISC-V Calling Convention

[riscv-elf-psabi-doc/riscv-cc.adoc at master · riscv/riscv-elf-psabi-doc \(github.com\)](https://github.com/riscv/riscv-elf-psabi-doc)

Study the remaining slides yourself

Alignment on stack

- RISC-V calling convention requires the stack pointer is always 16-byte aligned.
 - Even if you need space for one word, compiler reserves 16 bytes
- In this course, we keep the stack pointer aligned at a word address
 - 4-byte aligned

Study the sort example in textbook

- Study the code and answer the following questions
 - Which registers are used to store loop control variables i and j? Is it better to assign temporary registers (like t0 and t1) to i and j?
 - What registers are saved and restored in sort function?
- Try to write the sort function without looking at the assembly code provided in the textbook

RISC-V register conventions

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5 - x7	5–7	Temporaries	no
x8 - x9	8–9	Saved	yes
x10 - x17	10–17	Arguments/results	no
x18 - x27	18–27	Saved	yes
x28 - x31	28–31	Temporaries	no

General Procedure Calling Steps

Caller

1. Save (t, a, and ra) registers, if needed
2. Set parameters
3. Transfer control to function (JAL, JALR)

Callee

1. Acquire storage for local variables/register saving
2. Save registers, if necessary
3. Do the work
4. Set return value
5. Restore registers and stack
6. Return (JALR)

4. Restore registers and stack

Tips

- Write function body first, and then check what registers need to be saved
- Save registers at the beginning of a function
- Restore registers and stack just before returning
 - Use one exit from the function. **Only one return!**

```
add    a0, s1, s2    # return s1 + s2
beq    x0, x0, exit  # goto exit
...
li     a0, -1        # return -1
```

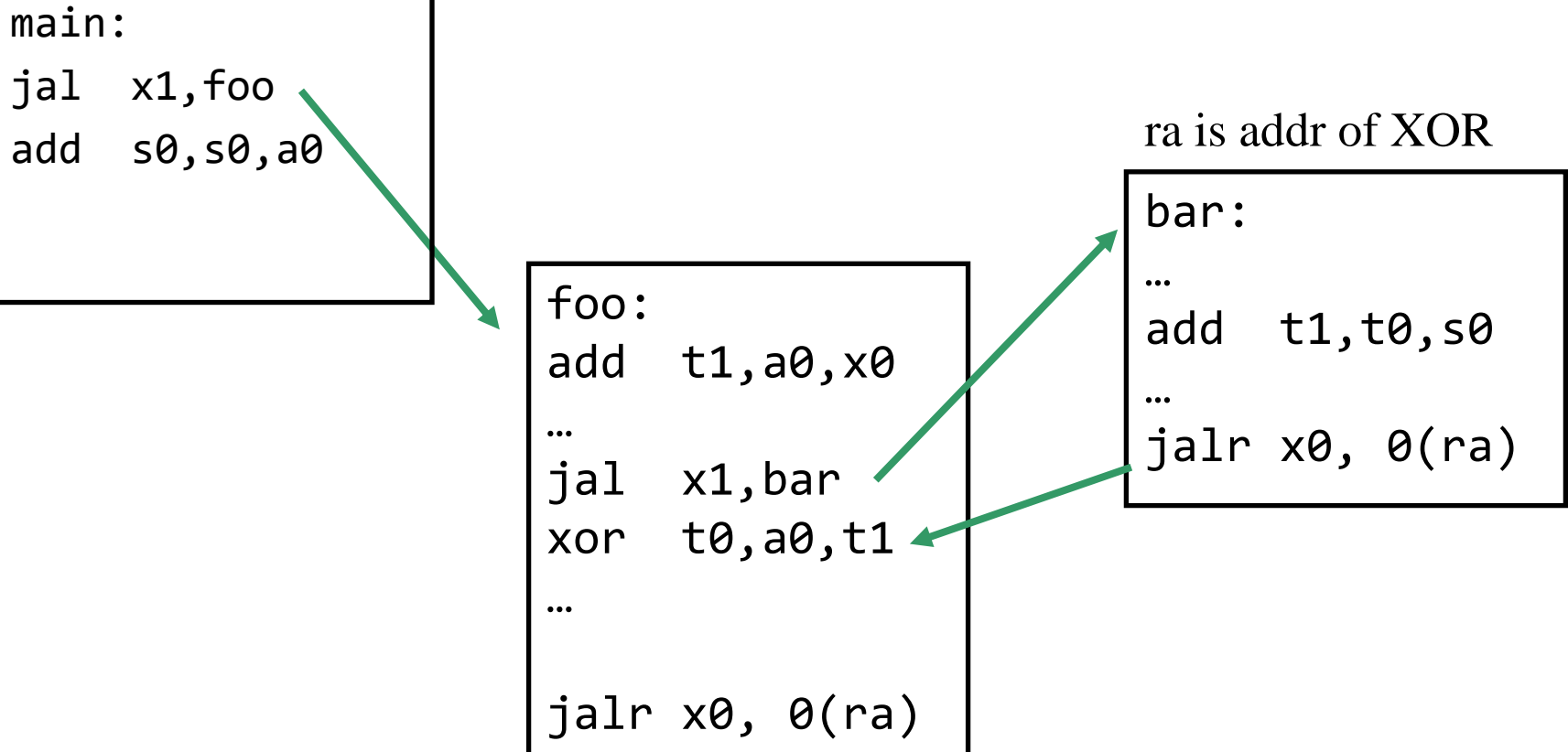
exit:

```
# restore registers and then return
```

Temporary registers in nested Function Call

main:

```
jal  x1,foo  
add  s0,s0,a0
```



foo:

```
add  t1,a0,x0  
...  
jal  x1,bar  
xor  t0,a0,t1  
...  
jalr x0, 0(ra)
```

ra is addr of XOR

bar:

```
...  
add  t1,t0,s0  
...  
jalr x0, 0(ra)
```

foo needs to save ra

foo may need to save t0, t1, ... before it calls bar

Pitfalls

- Forget calling conventions
 - Some values are not preserved in function calls
 - Get arguments from wrong registers
 - Place return value in wrong registers
 - Forget to save ra in non-leaf functions
 - Forget to save temporary registers in non-leaf functions
 - Forget to restore sp
- Too many exits from a function
 - Easy to forget to fix bugs at all exits
- Mix up code from different functions

Pitfalls

- Using pointers to variables that are no longer on the stack
 - For example, returning a pointer to a local variable in functions

Wrong order ! Load first

addi sp, sp, 4 # space already released!

lw ra, -4(sp)

Source code

Function names are just labels in the source code!

Source code

```
main:  
    jal  bar  
    nop  
    .....
```

```
bar:  
    addi t1,...  
    .....
```

```
foo:  
    xor  t0,...  
    .....
```

Functions do not overlap

Exercise: max3

```
int max3 (int a, int b, int c)
{
    int i = a;

    if (i < b)
        i = b;
    if (i < c)
        i = c;
    return i;
}
```

Exercise

- Write code to call `max3()`
`m = max3(10, 20, -20)`
- Write examples in the textbook
 - `sort`
 - `strcpy`
- Write more code
 - For example, a function that finds the median