CASS Exercise session 3

Functions and the stack

Functions recap: C

sum.c

```c
#include <stdio.h>

int sum(int a, int b)
{
    return a + b;
}

int main()
{
    int n;
    printf("Enter a number:\n");
    scanf("%d", &n);
    printf("Result: n+2=%d\n", sum(n, 2));
    return 0;
}
```

Console

```
$ - gcc sum.c -o sum
$ - ./sum
Enter a number:
5
Result: n+2=7
$ -
```

Functions recap: C

sum.c

```c
#include <stdio.h>

int sum(int a, int b)
{
    return a + b;
}

int main()
{
    int n;
    printf("Enter a number:\n");
    scanf("%d", &n);
    printf("Result: n+2=%d\n", sum(n, 2));
    return 0;
}
```

C functions allow for **abstraction** and **modularization** (via *parameters* and *return values*)

Console

```
-o sum
$ - ./sum
Enter a number:
5
Result: n+2=7
$ -
```

Functions recap: C

sum.c

```c
#include <stdio.h>

int sum(int a, int b)
{
    return a + b;
}

int main()
{
    int n;
    printf("Enter a number:\n");
    scanf("%d", &n);
    printf("Result: n+2=%d\n", sum(n, 2));
    return 0;
}
```

C comes with a convenient collection of **library functions**

Console

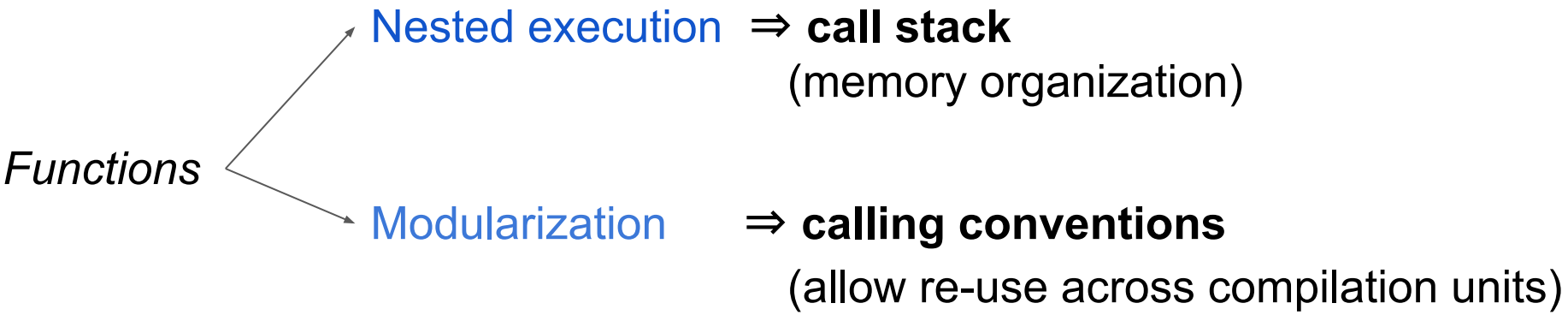
```
$ - gcc sum.c -o sum
$ - ./sum
Enter a number:
5
```

Function abstractions essential for higher-level programming languages → what about **asm**?

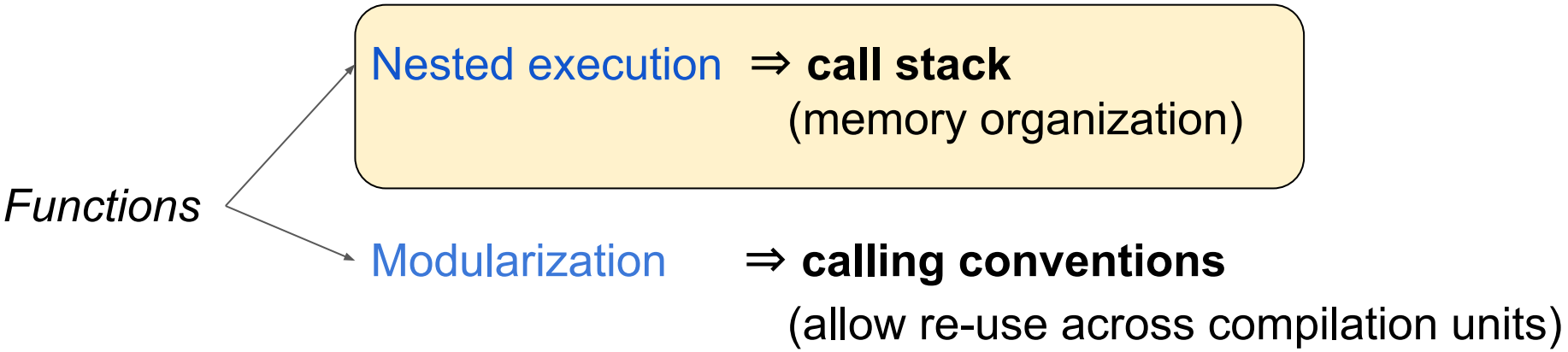
Functions: Challenges in asm

- (Almost) no concept of “function” or “procedure” in **asm**
- Implement functions using **common low-level primitives**:
  - labels
  - jumps
  - registers
- And that’s what this session is about!

Functions: Challenges in asm

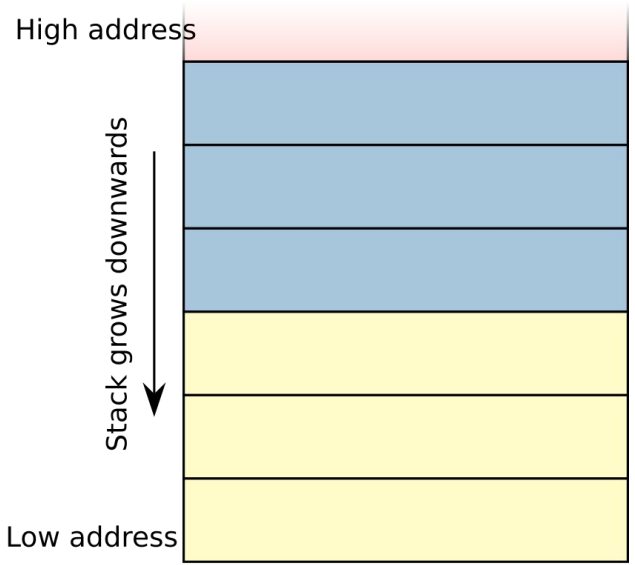


Functions: Challenges in asm



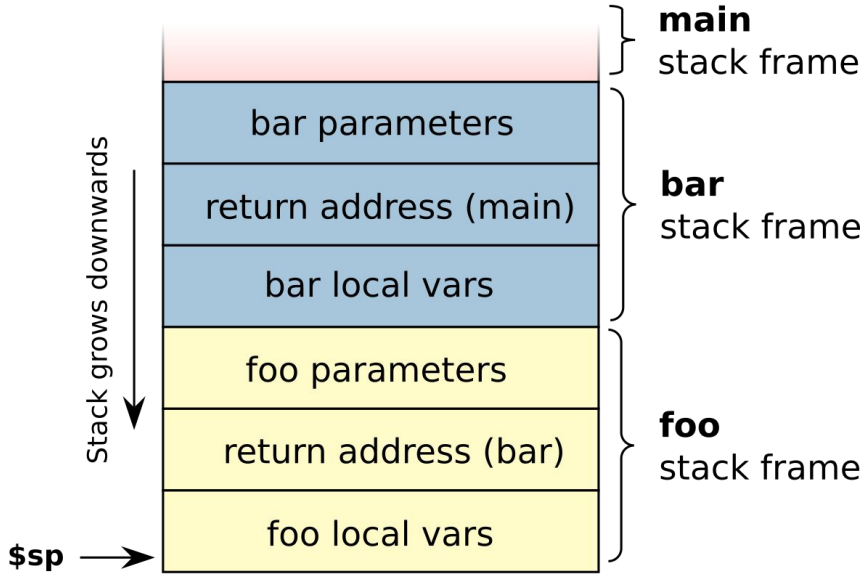
Call stack: Organizing memory for function calls

- **Dynamic** data structure: LIFO
  - grows from high to low addresses
  - push/pop primitives



Call stack: Organizing memory for function calls

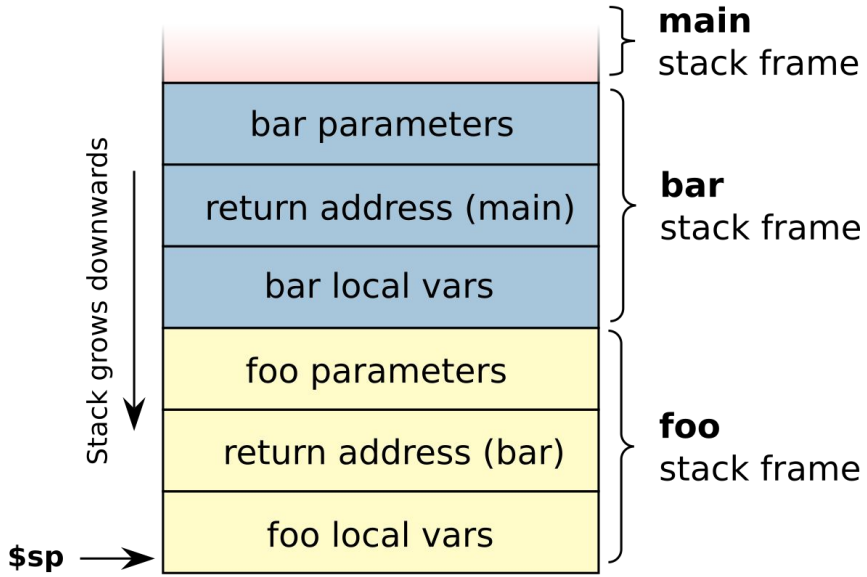
- **Stack frames** store *local exec context*:
  - parameters
  - return address
  - local vars



main() → bar() → foo()

Call stack: Organizing memory for function calls

- Dedicated **asm registers**:
  - sp: points to top of stack
  - push: `addi sp,sp,-4`
  - pop: `addi sp,sp,4`



main() → bar() → foo()

Stack example

stack-example.c

```
int x = 1;
int y = 2;
int z;

int foo(int a, int b)
{
  return a + b;
}

int bar(int c, int d)
{
  return foo(c, d);
}

int main()
{
  z = bar(x, y);
}
```

stack-example.asm

```
.data
 x: .word 1
 y: .word 2
 z: .space 4
.globl main

.text

foo:
  add a0, a0, a1
  ret

bar:
  addi sp, sp, -4
  sw   ra, 0(sp)
  jal  foo
  lw   ra, 0(sp)
  addi sp, sp, 4
  ret

main:
  lw  a0, x
  lw  a1, y
  jal bar
  sw  a0, z, t0
```

stack-example.asm

```
.data
 x: .word 1
 y: .word 2
 z: .space 4
.globl main

.text

# foo(a, b): a in a0, b in a1, return in a0
foo:
  add a0, a0, a1
  ret

# bar(c, d): c in a0, d in a1, return in a0
bar:
  addi sp, sp, -4
  sw   ra, 0(sp)
  jal  foo
  lw   ra, 0(sp)
  addi sp, sp, 4
  ret

main:
  lw  a0, x
  lw  a1, y
  jal bar
  sw  a0, z, t0
```

Main Memory + registers

a0	???
a1	???
ra	???
sp	???

...	...
0x7fffeff0	???
0x7fffeff4	???
0x7fffeff8	???
0x7fffeffc	???
...	...

stack-example.asm

```
.data
 x: .word 1
 y: .word 2
 z: .space 4
.globl main

.text

# foo(a, b): a in a0, b in a1, return in a0
foo:
  add a0, a0, a1
  ret

# bar(c, d): c in a0, d in a1, return in a0
bar:
  addi sp, sp, -4
  sw   ra, 0(sp)
  jal  foo
  lw   ra, 0(sp)
  addi sp, sp, 4
  ret

main:
  lw  a0, x
  lw  a1, y
  jal bar
  sw  a0, z, t0
```

At the start of the program execution, $sp points to a current stack top

Main Memory + registers

a0	???
a1	???
ra	???
sp	0x7fffeffc

...	...
0x7fffeff0	???
0x7fffeff4	???
0x7fffeff8	???
0x7fffeffc	???
...	...

stack-example.asm

```
.data
 x: .word 1
 y: .word 2
 z: .space 4
.globl main

.text

# foo(a, b): a in a0, b in a1, return in a0
foo:
  add a0, a0, a1
  ret

# bar(c, d): c in a0, d in a1, return in a0
bar:
  addi sp, sp, -4
  sw   ra, 0(sp)
  jal  foo
  lw   ra, 0(sp)
  addi sp, sp, 4
  ret

main:
  lw  a0, x
  lw  a1, y
  jal bar
  sw  a0, z, t0
```

Preparing arguments: placing them in registers

Main Memory + registers

a0	0x00000001
a1	0x00000002
ra	???
sp	0x7fffeffc

...	...
0x7fffeff0	???
0x7fffeff4	???
0x7fffeff8	???
0x7fffeffc	???
...	...

stack-example.asm

```
.data
 x: .word 1
 y: .word 2
 z: .space 4
.globl main

.text

# foo(a, b): a in a0, b in a1, return in a0
foo:
  add a0, a0, a1
  ret

# bar(c, d): c in a0, d in a1, return in a0
bar:
  addi sp, sp, -4
  sw   ra, 0(sp)
  jal  foo
  lw   ra, 0(sp)
  addi sp, sp, 4
  ret

main:
  lw  a0, x
  lw  a1, y
  jal bar
  sw  a0, z, t0
```

$ra = Address of instruction after jal (0x00400044)

Main Memory + registers

a0	0x00000001
a1	0x00000002
ra	0x00400044
sp	0x7fffeffc

…	…
0x7fffeff0	???
0x7fffeff4	???
0x7fffeff8	???
0x7fffeffc	???
…	…

stack-example.asm

```
.data
 x: .word 1
 y: .word 2
 z: .space 4
.globl main

.text

# foo(a, b): a in a0, b in a1, return in a0
foo:
  add a0, a0, a1
  ret

# bar(c, d): c in a0, d in a1, return in a0
bar:
  addi sp, sp, -4
  sw   ra, 0(sp)
  jal  foo
  lw   ra, 0(sp)
  addi sp, sp, 4
  ret

main:
  lw  a0, x
  lw  a1, y
  jal bar
  sw  a0, z, t0
```

Push return address on the stack:
(1) allocate space

Main Memory + registers

a0	0x00000001
a1	0x00000002
ra	0x00400044
sp	0x7fffeff8

...	...
0x7fffeff0	???
0x7fffeff4	???
0x7fffeff8	???
0x7fffeffc	???
...	...

stack-example.asm

```
.data
 x: .word 1
 y: .word 2
 z: .space 4
.globl main

.text

# foo(a, b): a in a0, b in a1, return in a0
foo:
  add a0, a0, a1
  ret

# bar(c, d): c in a0, d in a1, return in a0
bar:
  addi sp, sp, -4
  sw   ra, 0(sp)
  jal  foo
  lw   ra, 0(sp)
  addi sp, sp, 4
  ret

main:
  lw  a0, x
  lw  a1, y
  jal bar
  sw  a0, z, t0
```

Push return address on the stack:
(2) store the value

Main Memory + registers

a0	0x00000001
a1	0x00000002
ra	0x00400044
sp	0x7fffeff8

...	...
0x7fffeff0	???
0x7fffeff4	???
0x7fffeff8	0x00400044
0x7fffeffc	???
...	...

stack-example.asm

```
.data
 x: .word 1
 y: .word 2
 z: .space 4
.globl main

.text

# foo(a, b): a in a0, b in a1, return in a0
foo:
  add a0, a0, a1
  ret

# bar(c, d): c in a0, d in a1, return in a0
bar:
  addi sp, sp, -4
  sw   ra, 0(sp)
  jal  foo
  lw   ra, 0(sp)
  addi sp, sp, 4
  ret

main:
  lw  a0, x
  lw  a1, y
  jal bar
  sw  a0, z, t0
```

$ra = Address of instruction after jal (0x0040001c)

Main Memory + registers

a0	0x00000001
a1	0x00000002
ra	0x0040001c
sp	0x7fffeff8

...	...
0x7fffeff0	???
0x7fffeff4	???
0x7fffeff8	0x00400044
0x7fffeffc	???
...	...

stack-example.asm

```
.data
 x: .word 1
 y: .word 2
 z: .space 4
.globl main

.text

# foo(a, b): a in a0, b in a1, return in a0
foo:
  add a0, a0, a1
  ret

# bar(c, d): c in a0, d in a1, return in a0
bar:
  addi sp, sp, -4
  sw   ra, 0(sp)
  jal  foo
  lw   ra, 0(sp)
  addi sp, sp, 4
  ret

main:
  lw  a0, x
  lw  a1, y
  jal bar
  sw  a0, z, t0
```

ret: jump to code address in ra register

Main Memory + registers

a0	0x00000001
a1	0x00000002
ra	**0x0040001c**
sp	0x7fffeff8

...	...
0x7fffeff0	???
0x7fffeff4	???
0x7fffeff8	0x00400044
0x7fffeffc	???
...	...

stack-example.asm

```
.data
 x: .word 1
 y: .word 2
 z: .space 4
.globl main

.text

# foo(a, b): a in a0, b in a1, return in a0
foo:
  add a0, a0, a1
  ret

# bar(c, d): c in a0, d in a1, return in a0
bar:
  addi sp, sp, -4
  sw   ra, 0(sp)
  jal  foo
  lw   ra, 0(sp)
  addi sp, sp, 4
  ret

main:
  lw  a0, x
  lw  a1, y
  jal bar
  sw  a0, z, t0
```

Pop return address from the stack:
(1) load the value

Main Memory + registers

a0	0x00000001
a1	0x00000002
ra	0x00400044
sp	0x7fffeff8

...	...
0x7fffeff0	???
0x7fffeff4	???
0x7fffeff8	0x00400044
0x7fffeffc	???
...	...

stack-example.asm

```
.data
 x: .word 1
 y: .word 2
 z: .space 4
.globl main

.text

# foo(a, b): a in a0, b in a1, return in a0
foo:
  add a0, a0, a1
  ret

# bar(c, d): c in a0, d in a1, return in a0
bar:
  addi sp, sp, -4
  sw   ra, 0(sp)
  jal  foo
  lw   ra, 0(sp)
  addi sp, sp, 4
  ret

main:
  lw  a0, x
  lw  a1, y
  jal bar
  sw  a0, z, t0
```

Pop return address from the stack:
(2) deallocate the space

Main Memory + registers

a0	0x00000001
a1	0x00000002
ra	0x00400044
sp	0x7fffeffc

...	...
0x7fffeff0	???
0x7fffeff4	???
0x7fffeff8	0x00400044
0x7fffeffc	???
...	...

stack-example.asm

```
.data
 x: .word 1
 y: .word 2
 z: .space 4
.globl main

.text

# foo(a, b): a in a0, b in a1, return in a0
foo:
  add a0, a0, a1
  ret

# bar(c, d): c in a0, d in a1, return in a0
bar:
  addi sp, sp, -4
  sw   ra, 0(sp)
  jal  foo
  lw   ra, 0(sp)
  addi sp, sp, 4
  ret

main:
  lw  a0, x
  lw  a1, y
  jal bar
  sw  a0, z, t0
```

ret: jump to code address in ra register

Main Memory + registers

a0	0x00000001
a1	0x00000002
ra	**0x00400044**
sp	0x7fffeffc

...	...
0x7fffeff0	???
0x7fffeff4	???
0x7fffeff8	???
0x7fffeffc	???
...	...

stack-example.asm

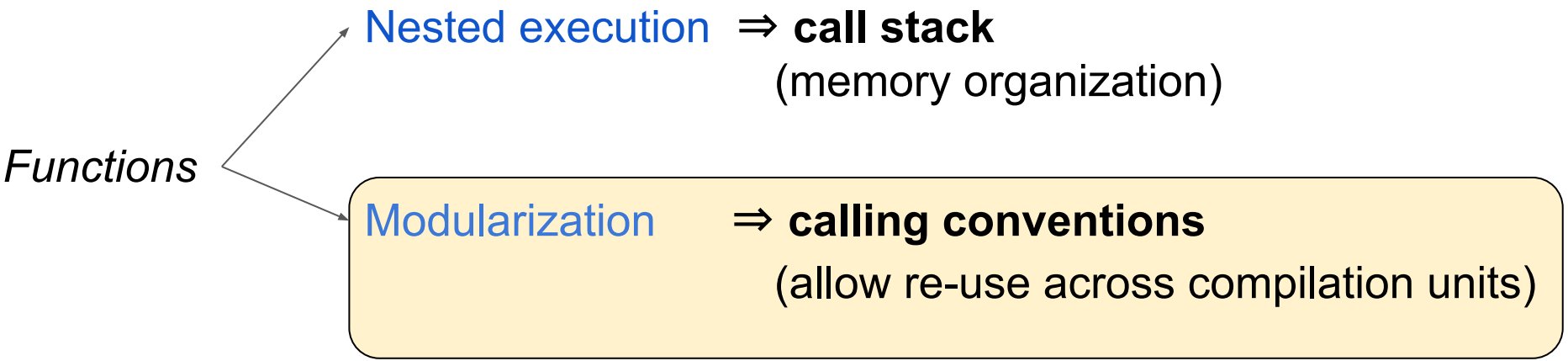
```
.data
 x: .word 1
 y: .word 2
 z: .space 4
.globl main

.text

# foo(a, b): a in a0, b in a1, return in a0
foo:
  add a0, a0, a1
  ret

# bar(c, d): c in a0, d in a1, return in a0
bar:
  addi sp, sp, -4
  sw   ra, 0(sp)
  jal  foo
  lw   ra, 0(sp)
  addi sp, sp, 4
  ret

main:
  lw  a0, x
  lw  a1, y
  jal bar
  sw  a0, z, t0
```

Main Memory + registers

a0	0x00000001
a1	0x00000002
ra	0x00400044
sp	0x7fffeffc

...	...
0x7fffeff0	???
0x7fffeff4	???
0x7fffeff8	???
0x7fffeffc	???
...	...

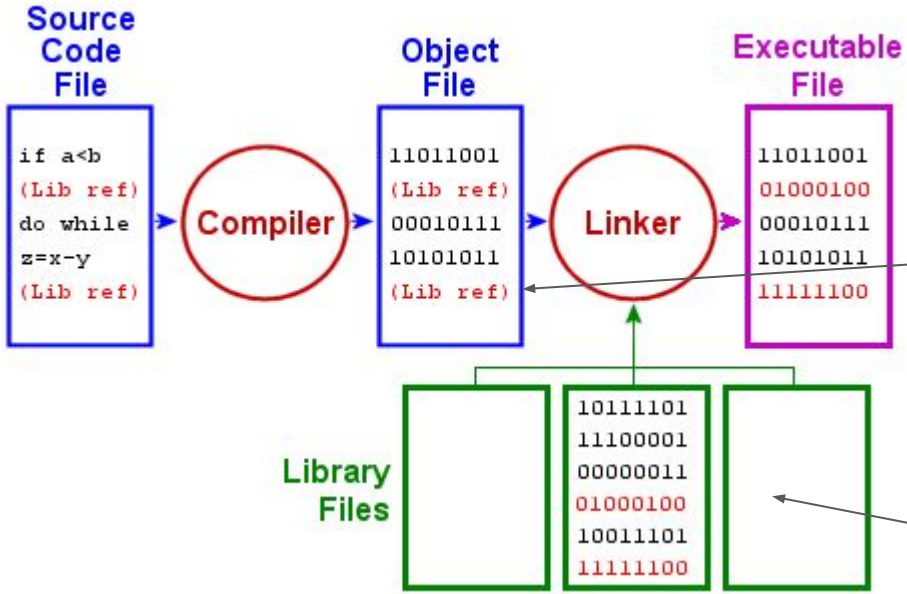
Functions: Challenges in asm



Calling conventions

Problem: asm functions expect *arguments* and *return values* in certain *registers*

→ e.g., library function call



Caller

Which registers to pass **args**?
Which register to expect **rv**?

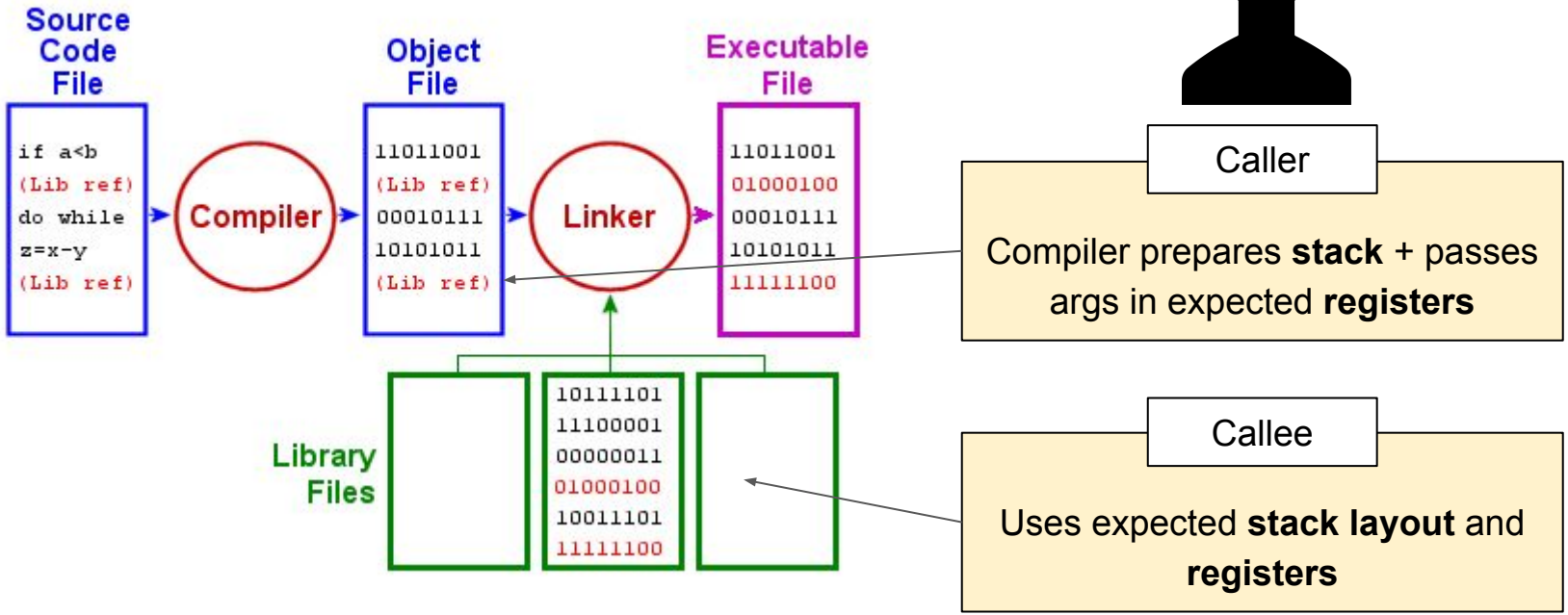
Callee

Which registers to expect **args**?
Which register to pass **rv**?

Calling conventions

Solution: fixed compiler *conventions* on stack layout and register usage

→ e.g., library function call



Calling conventions

- Just a convention, yet **necessary** for smooth execution and cooperation between code

Calling conventions: Risc-V

- https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf
- Arguments in registers a0 - a7
  - Big data types (>1 register) split into lower and upper bits
    - void fun(long double x); -> 16 byte arg!
    - pass in a0, a1, a2, a3
  - Not enough registers? Use Stack!
- Return value(s) in a0 - a1
  - Data type > 2 registers? Pass return by pointer
    - long double fun(int x); -> 16 byte retval!
    - Store in memory and pass address

Calling conventions: Risc-V

- https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf
  - **Read this!**
- Predefined usage for registers
- Fixed sizes defined for C data types
- Conventions on function calls
- A version will be available on the exam
  - Don’t study by heart
  - Always use as a reference

Calling conventions: Risc-V function calls

- Arguments in registers a0 - a7
  - Data types with 2x register size: split into lower and upper bits
    - void fun(double x); -> 8 bytes arg
  - Data types with > 2x register size: pass by reference
    - void fun(long double x); -> 16 byte arg
    - Store in memory, pass address in a0
  - Not enough registers? Use Stack!
- Return value(s) in a0 - a1
  - Data type > 2 registers? Pass by reference
    - long double fun(int x); -> 16 byte retval!
    - Store in memory and pass address in a0

Calling conventions: Stack alignment

- Risc-V conventions: sp must be 16-byte aligned
- CASS conventions: sp must be 4-byte aligned
  - You can choose to 16-byte align
    - Be consistent in your choice! No mixing
  - Solutions and slides will be 4-byte aligned
- More info?
  - https://github.com/riscv/riscv-elf-psabi-doc/issues/21
- This is the only convention that we break!
  - More practical to manually write 4-byte aligned code

Caller-save vs. callee-save registers

- **Caller-save** == not guaranteed to be preserved across call

- **Callee-save** == guaranteed to be preserved across call

Caller-save vs. callee-save registers

- **Caller-save** == not guaranteed to be preserved across call

  → *stored **before** function call and restored **after** return*

- **Callee-save** == guaranteed to be preserved across call

  → *stored **after** function entry if to be used, and restored **before** return*

Calling conventions: Risc-V caller-save vs. callee-save

- **Caller-save** == not guaranteed to be preserved across call
  - Temporary registers: `t0 - t6`
  - Argument registers `a0 - a7`
  - Return address register `ra`
- **Callee-save** == guaranteed to be preserved across call
  - The saved registers: `s0 - s11`
  - The stack pointer: `sp`
  - The frame pointer: `fp (=s0)`

Recursion

“...a method where the solution to a problem depends on solutions to smaller instances of the same problem...”

- Ubiquitous concept in mathematics and CS
- A function is recursive if it calls itself in its body

```
int factorial( int n )
{
    if ( n >= 1 )
        return n * factorial( n-1);
    else
        return 1;
}
```

Recursion

"...a method where the solution
solutions to smaller instances o

- Ubiquitous concept in mathematics and CS
- A function is recursive if it calls itself in its body

```
int factorial( int n )
{
    if ( n >= 1 )
        return n * factorial( n-1);
    else
        return 1;
}
```

With correct stack usage and register handling, recursion is **just like any other function call**

Extra Checklist: Calling a function?

Caller

1. Save ra on stack
2. Save other caller-saved registers *if you need the value after the function call*
3. Call function
4. Restore caller-saved registers and ra from stack

Callee

1. Save callee-saved registers that you overwrite on stack first
2. Restore callee-saved registers from stack before returning