5. Instruction Set Architecture – from C to assembly – Functions

EECS 370 – Introduction to Computer Organization – Winter 2015

Robert Dick, Andrew Lukefahr, and Satish Narayanasamy

EECS Department
University of Michigan in Ann Arbor, USA

Announcements

- HW 1 Due today by 6pm (Submit through CTools)
- HW 2 posted by the end of today
- Project 1 first part due Wed 1/28 (electronically)

- Declare midterm and final exam conflicts, and special accommodation requests by 1/29

Recap - Translating C to assembly

- Golden rules of data layout in memory
  - Simple variables
  - Struct
- Branches can translate many C/C++ control structures
  - if/then/else
  - for

Instruction Set Architecture (ISA) Design Lectures

- Lecture 2: Storage types and addressing modes
- Lecture 3 : LC-2K and MIPS architecture
- Lecture 4 : Converting C to assembly – basic blocks
- **Lecture 5 : Converting C to assembly – functions**
- Lecture 6 : Translation software; libraries, memory layout

Converting function calls to assembly code

C: printf("hello world\n");

- Need to pass parameters to the called function (printf)
- Need to save return address of caller
- Need to save register values
- Need to jump to printf

Execute instructions for printf()
Jump to return address

- Need to get return value (if used)
- Restore register values

Task 1: Passing parameters

FUNCTION CALLS

- Where should you put all of the parameters?
  - Registers?
    - Fast access but few in number and wrong size for some objects
  - Memory?
    - Good general solution but where?

- ARM answer:
  - Registers and memory
    - Put the first few parameters in registers (if they fit) (r0 – r3)
    - Put the rest in memory on the call stack

  - Example:
    mov   r0, #1000 // put address of char array “hello world” in r0

Call stack

- ARM conventions (and most other processors) allocate a region of memory for the call stack
  - This memory is used to manage all the storage requirements to simulate function call semantics
    - Parameters (that were not passed through registers)
    - Local variables
    - Temporary storage (when you run out of registers and need somewhere to save a value)
    - Return address
    - Etc.

- Sections of memory on the call stack [a.k.a. **stack frames**] are allocated when you make a function call, and de-allocated when you return from a function.

ARM (Linux) Memory Map

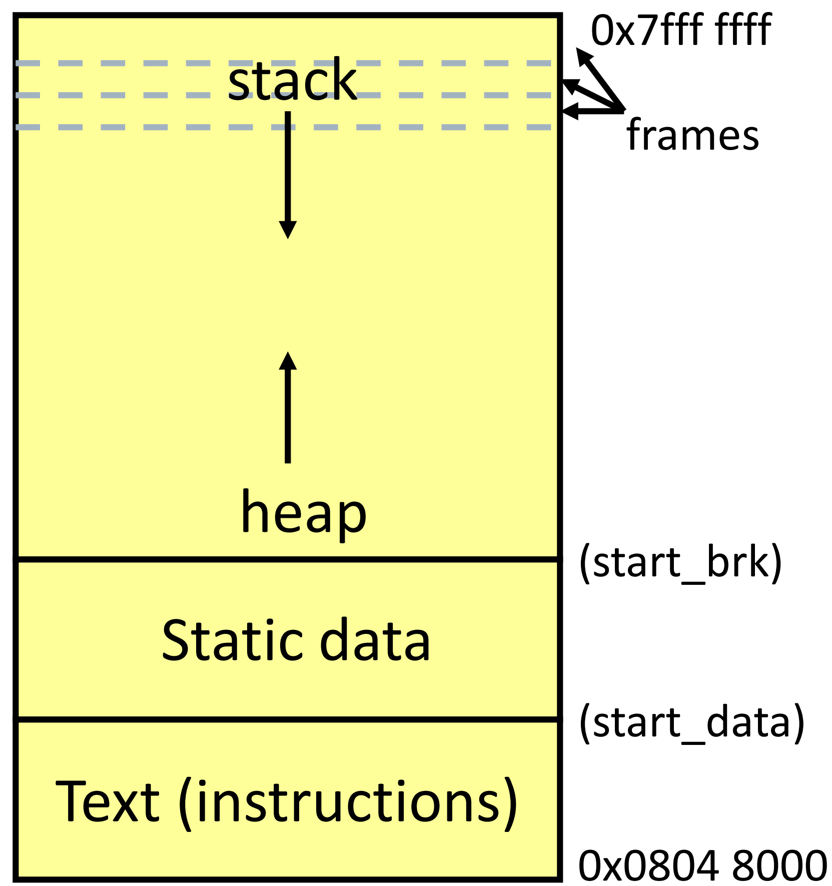
FUNCTION CALLS

Stack: starts at 0x7fff ffff and grows down to lower addresses. Bottom of the stack resides in the SP register

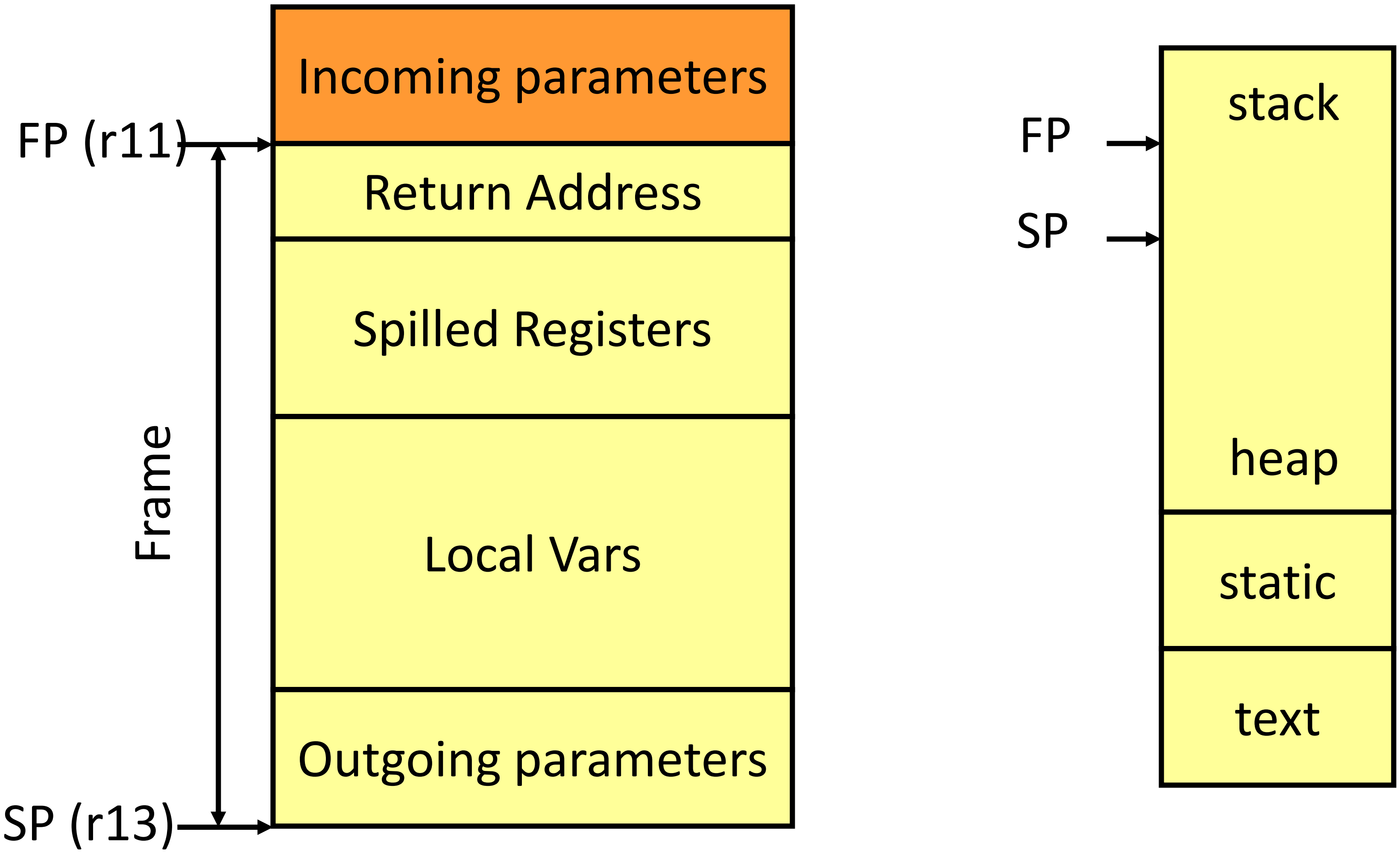
Heap: starts above static (page aligned) and grows up to higher addresses. Allocation done explicitly with malloc(). Deallocation with free(). Runtime error if no free memory before running into SP address.

Static: starts above text (page aligned). Holds all global variables and those locals explicitly declared as "static".

Text: starts at 0x08048000. Holds all instructions in the program (except for Dynamically linked library routines DLLs)



The ARM Stack Frame (typical organization)



Allocating space to local variables

- Local variables (by default) are created when you enter a function, and disappear when you leave
  - Technical terminology: local variables are placed in the automatic storage class (as opposed to the static storage class used for globals).

- Automatics are allocated on the call stack
  - How?
    by incrementing (or decrementing) the pointer to the top of the call stack

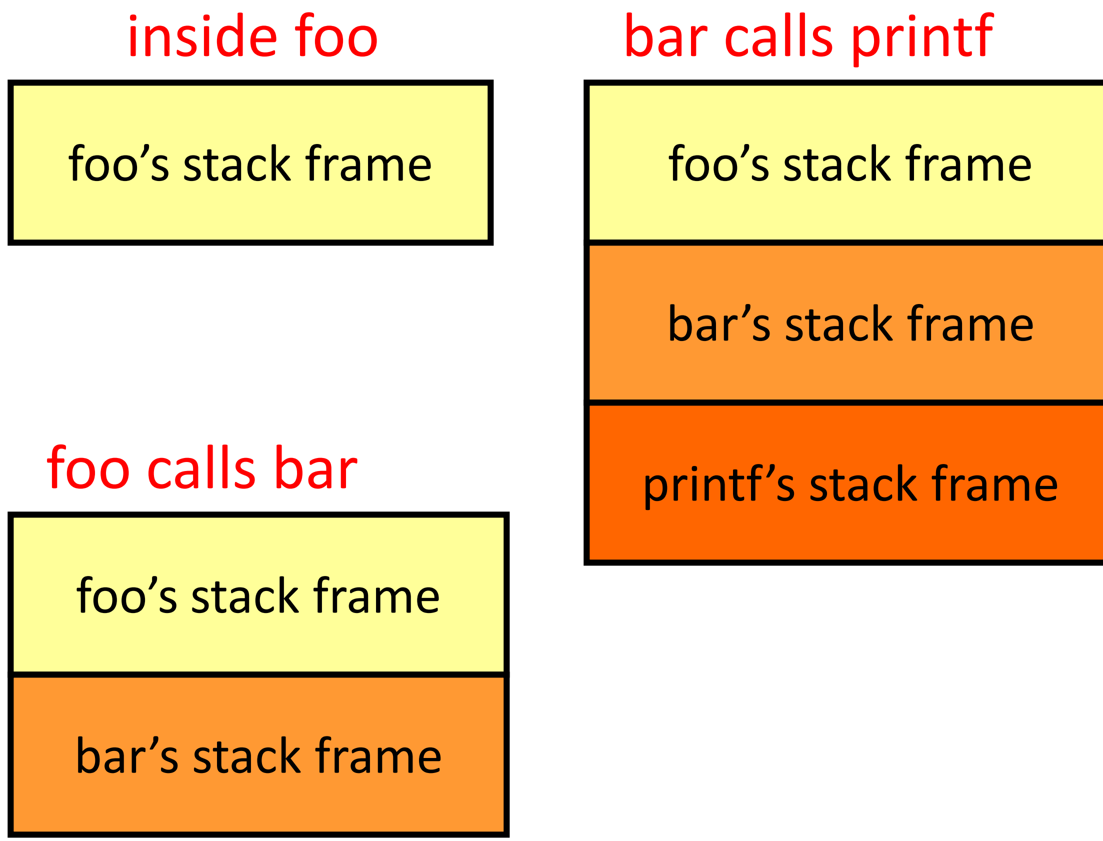
  - sub r13, r13, #12 // SP = SP – 12, allocate space for 3 integer locals
    add r13, r13, #12 // SP = SP + 12, de-allocate space for locals

FUNCTION CALLS

The stack grows as functions are called

```
void foo()
{
   int x, y[2];
   bar(x);
}

void bar(int x)
{
   int a[3];
   printf();
}
```

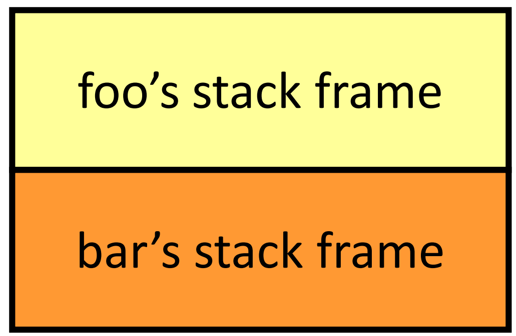


The stack shrinks as functions return

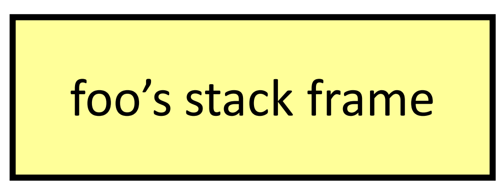
```
void foo()
{
   int x, y[2];
   bar(x);
}

void bar(int
x)
{
   int a[3];
   printf();
}
```

printf returns



bar returns



FUNCTION CALLS

Stack frame contents

```
void foo()
{
   int x, y[2];
   bar(x);
}

void bar(int
x)
{
   int a[3];
   printf();
}
```

foo’s stack frame

return addr to main
x
y[0]
y[1]
spilled regs in foo

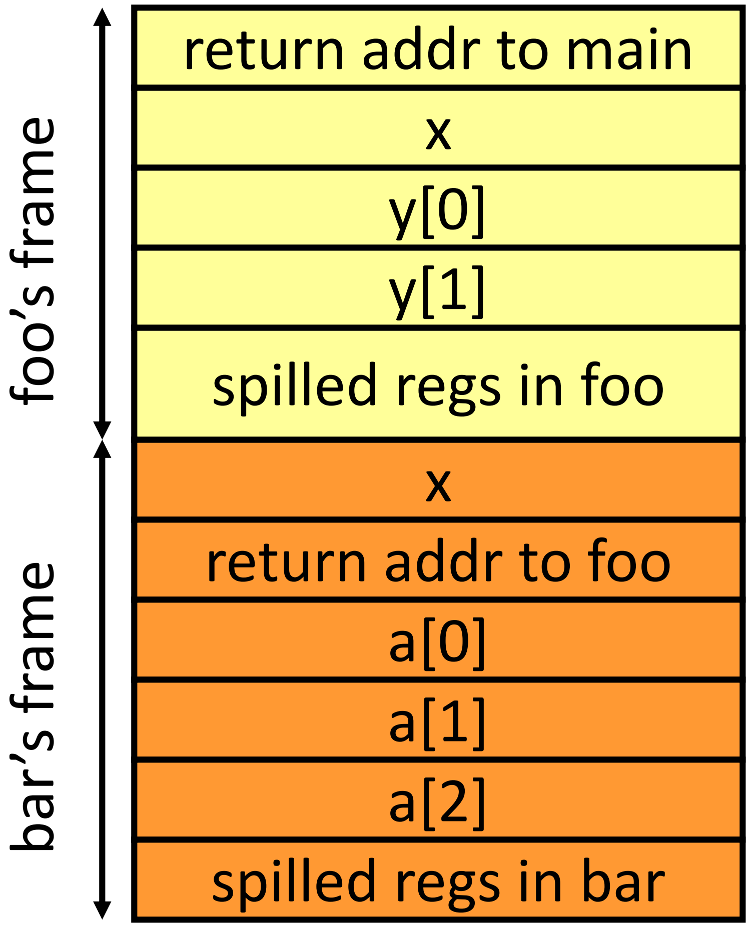
FUNCTION CALLS

Stack frame contents (2)

```
void foo()
{
   int x, y[2];
   bar(x);
}

void bar(int
x)
{
   int a[3];
   printf();
}
```

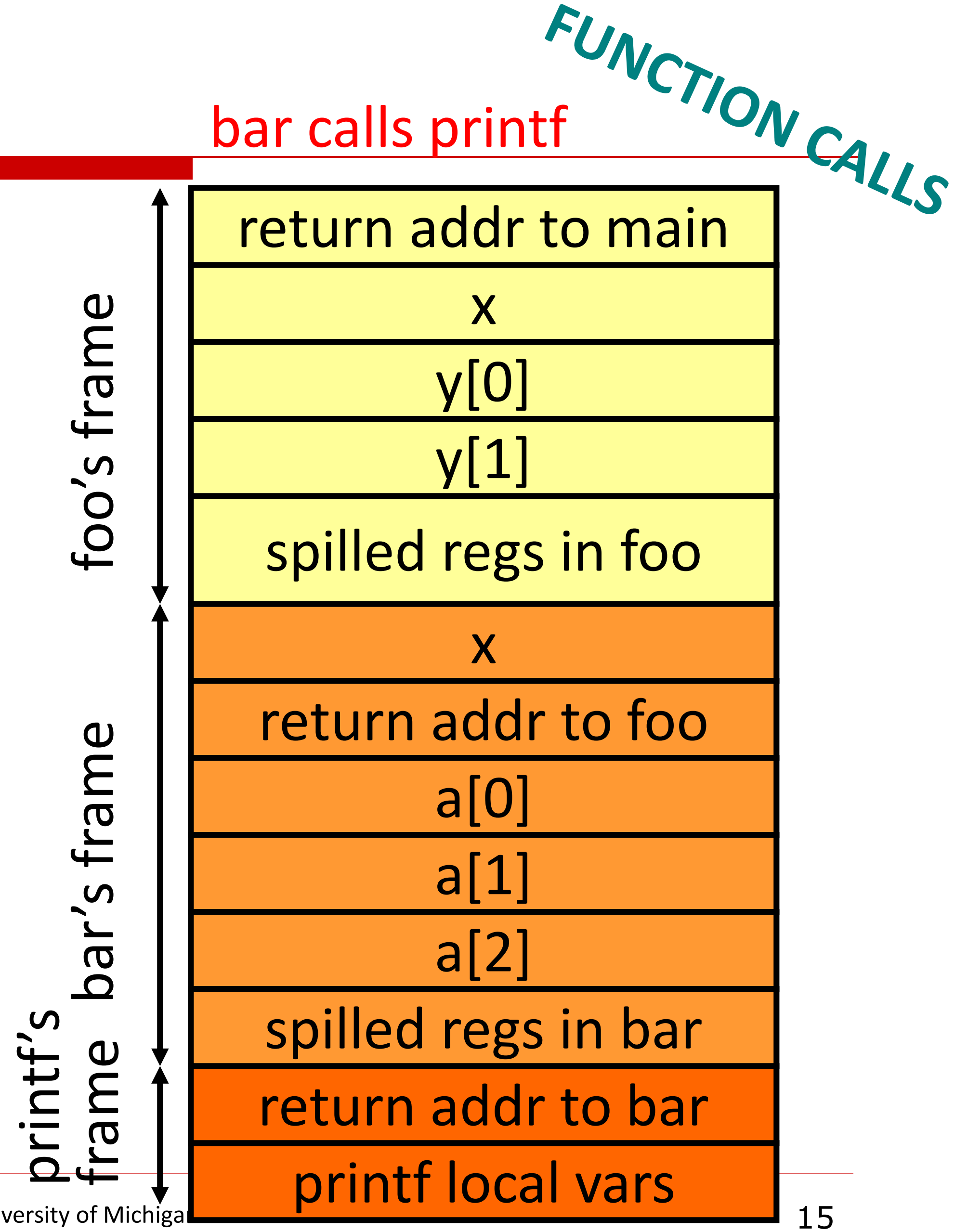
foo calls bar



Stack frame contents (3)

```
void foo()
{
    int x, y[2];
    bar(x);
}

void bar(int x)
{
    int a[3];
    printf();
}
```



Recursive function example

FUNCTION CALLS

```
main()
{
   foo(2);
}

void foo(int a)
{
   int x, y[2];
   if (a > 0)
      foo(a-1);
}
```

	return addr to …
main calls foo	2
	return addr to main
	x, y[0], y[1]
	spills in foo
foo calls foo	1
	return addr to foo
	x, y[0], y[1]
	spills in foo
foo calls foo	0
	return addr to foo
	x, y[0], y[1]
	spills in foo

FUNCTION CALLS

Assigning variables to memory spaces

```
int w;
void foo(int x)
{
    static int y[4];
    char *p;
    p = malloc(10);
    …
    printf("%s\n", p);
}
```

w goes in static, as it's a global

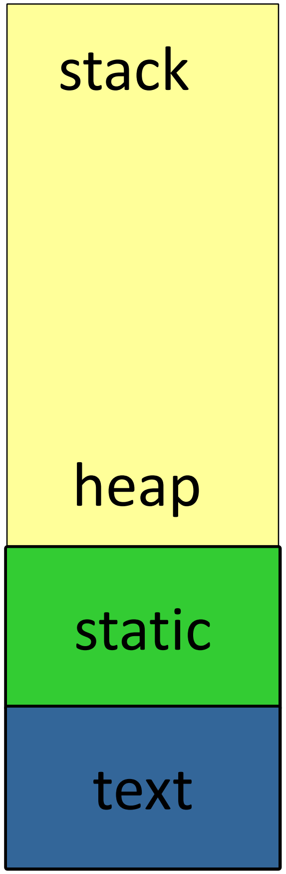
x goes on the stack, as it's a parameter

y goes in static, 1 copy of this!!

p goes on the stack

allocate 10 bytes on heap, ptr set to the address

string goes in static, pointer to string on stack, p goes on stack



Need for Saving registers during a call

CALLER-CALLEE

- What happens to the values we have in registers when we make a function call? Assume variables x in foo() and y in bar() happen to be allocated to the same register r1.

```
void foo()
{
    int x = 1;
    bar(x);
    x = x + 1;
}

void bar(int k)
{
    int y = 2;
    y++
 }
```

```
void foo()
{
    r1 = 1;
    bar(r1);
    r1 = r1 + 1;
}

void bar(int k)
{
    r1 = 2;
    r1 = r1 + 1
 }
```

Saving registers during a call

CALLER-CALLEE

- What happens to the values we have in registers when we make a function call?

a = b * c + sqrt (d);

Options:

1. You can save your registers **before** you make the function call and restore the registers when you return (**caller-save register**). Where?

   The stack frame is used to store anything required to support function calls

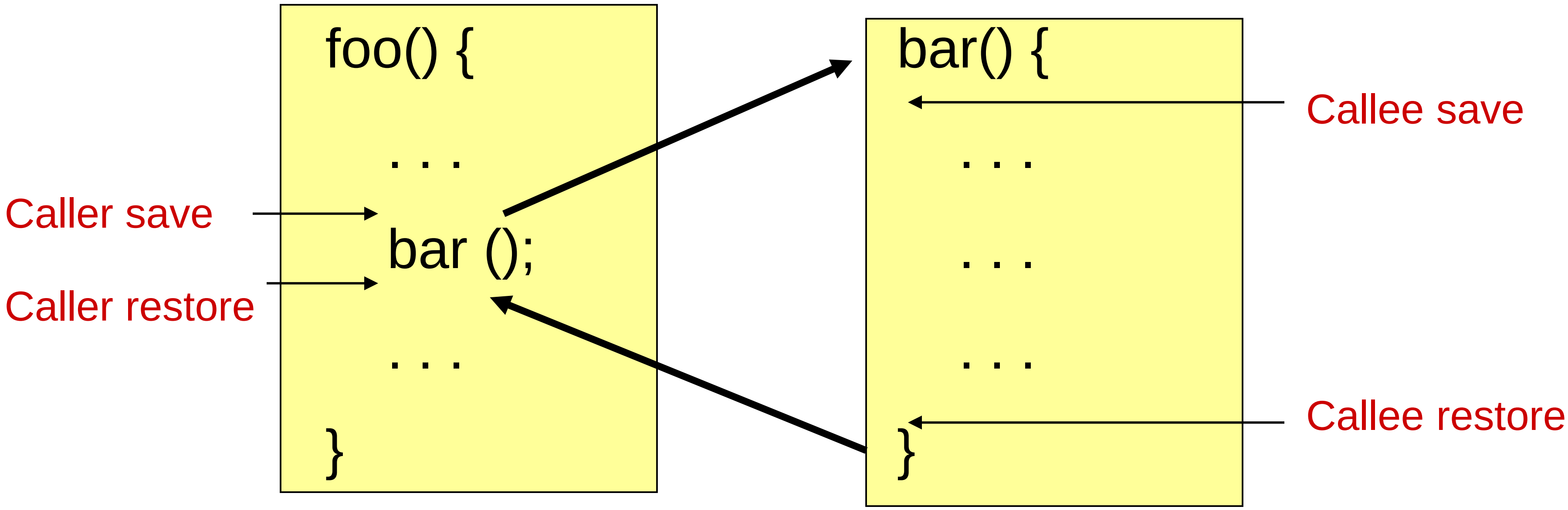
   What if the function you are calling doesn't use that register? No harm done, but wasted work!!!

2. You can save your registers **after** you make the function call and restore the registers before you return (**callee-save register**). Where?

   What if the caller function doesn't use that register? No harm done, but wasted work!!!

Caller-Callee save/restore

CALLER-CALLEE



Caller save: Callee may change, so caller responsible for saving immediately before call and restoring immediately after call

Callee save: Callee may not change, so callee (called function) must leave these unchanged. Can be ensured by inserting saves at the start of the function and restores at the end

Caller/callee example

CALLER-CALLEE

foo()
{
    r0 = 5;
    r4 = -1;
    bar();
    r3 = r0 + r4;
}

bar()
{
    r0 = 10;
    r4 = 5;
}

If r0 is caller-save and r4 is callee-save

foo()
{
    r0 = 5;
    r4 = -1;
    save r0; *i.e.,* str r0, [r13, #20]
    bar();
    restore r0; *i.e.,* ldr r0, [r13, #20]
    r3 = r0 + r4;
}

bar()
{
    save r4; *i.e.,* str r4, [r13, #8]
    r0 = 10;
    r4 = 5;
    restore r4; *i.e.,* ldr r4, r13, #8]
}

Saving/Restoring Optimizations

CALLER-CALLEE

- Caller-saved
  - Only needs saving if it is "live" across a function call
  - Live = contains a useful value: Assign value before function call, use that value after the function call
  - In a leaf function, caller saves can be used without saving/restoring
- Callee-saved
  - Only needs saving at beginning of function (generally infrequent as outside of loops) and restoring at end of function
  - Only save/restore it if function overwrites the register
- Each has its advantages. Neither is always better.

Provide Both Caller/Callee Regs

CALLER-CALLEE

- Have some registers that are caller-saved, some that are callee-saved
  - Example: ARM
    - 5 caller saved
    - 10 callee saved
- **Choose registers for variables so to minimize the number of dynamic saves/restores**

ARM register conventions

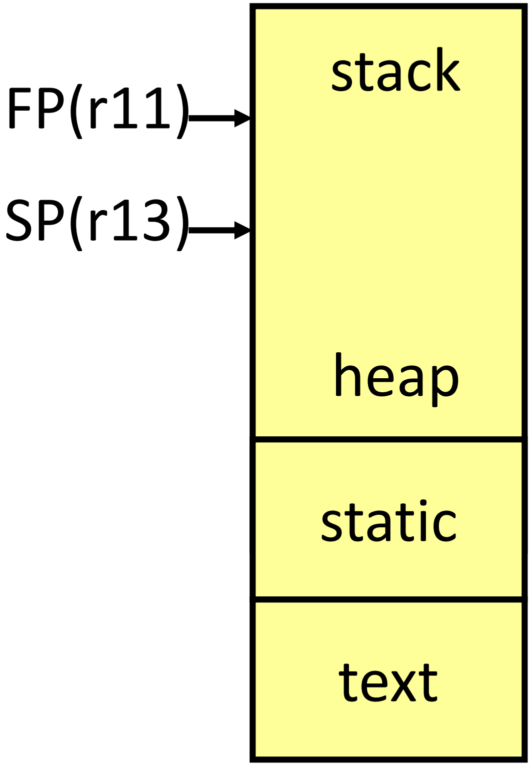
r0	parameter, return value, caller saved
r1-r3	parameters, caller saved
r4-r10	callee saved
r11	frame pointer, callee saved
r12	caller saved
r13	stack pointer, callee saved
r14	link register, callee saved
r15	program counter, not saved

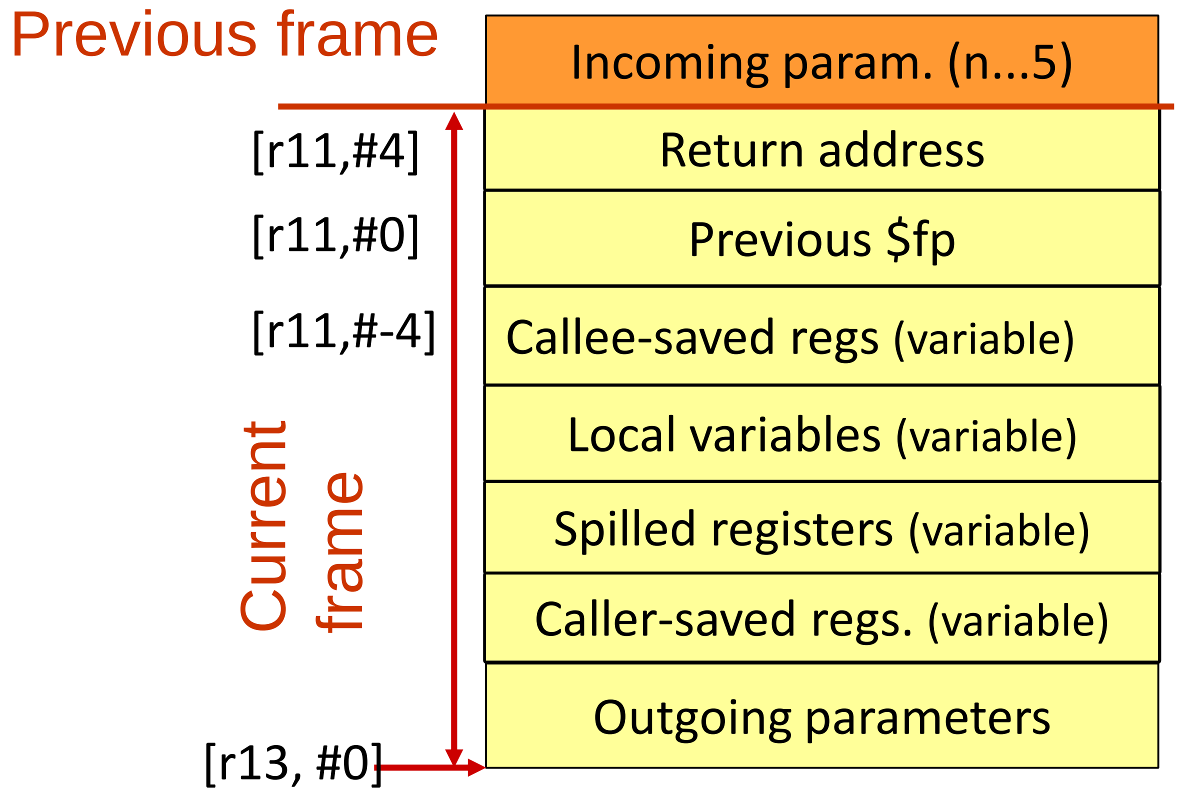
Calling convention

- This is a **convention**: calling convention
  - There is no difference in H/W between caller and callee save registers
- Passing parameters in registers is also a convention
- Allows assembly code written by different people to work together
  - Need conventions about who saves regs and where args are passed.
- These conventions collectively make up the ABI or “application binary interface”
- Why are these conventions important?
  - What happens if a programmer/compiler violates them?

ARM Stack Frame (typical organization)

CALLER-CALLEE





Note 1: in ARM, the first 4 parameters are passed via registers. Other ISAs have ≠ conventions

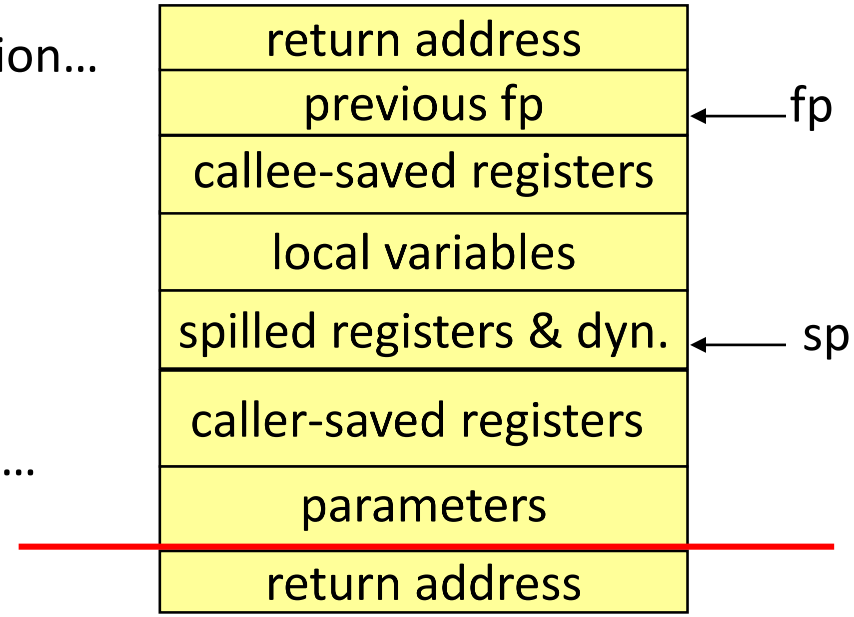
Note 2: why is the last parameter first on the stack ?

The stack during program execution…

CALLER-CALLEE

foo() is in execution…

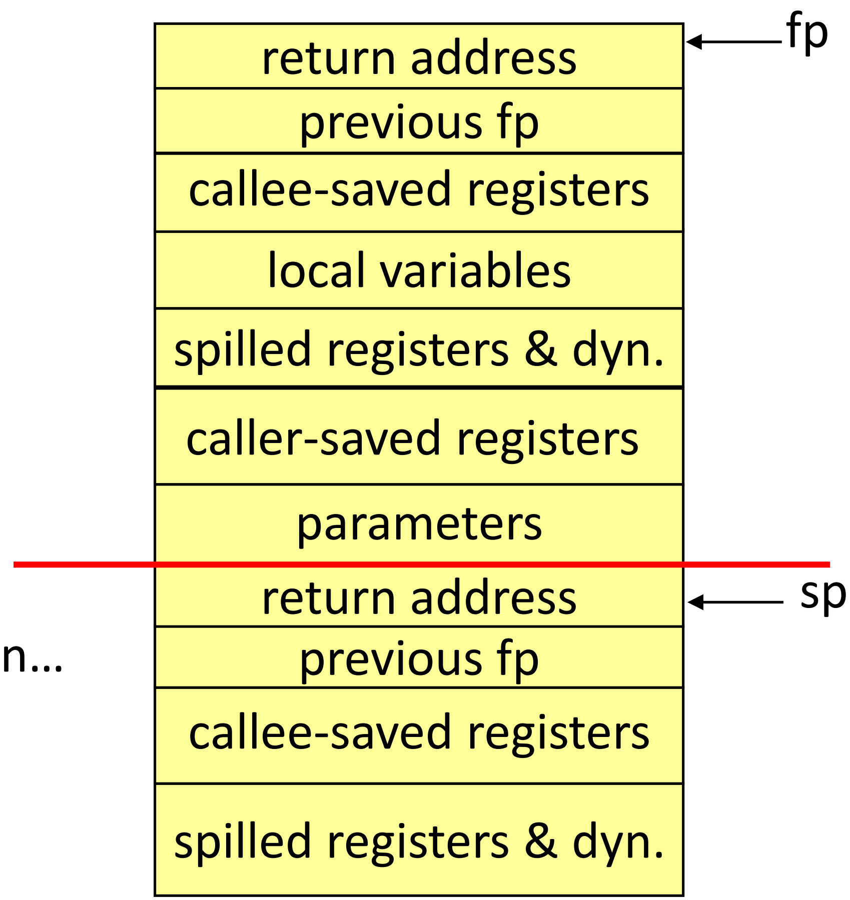
…foo() calls bar…



CALLER-CALLEE

The stack

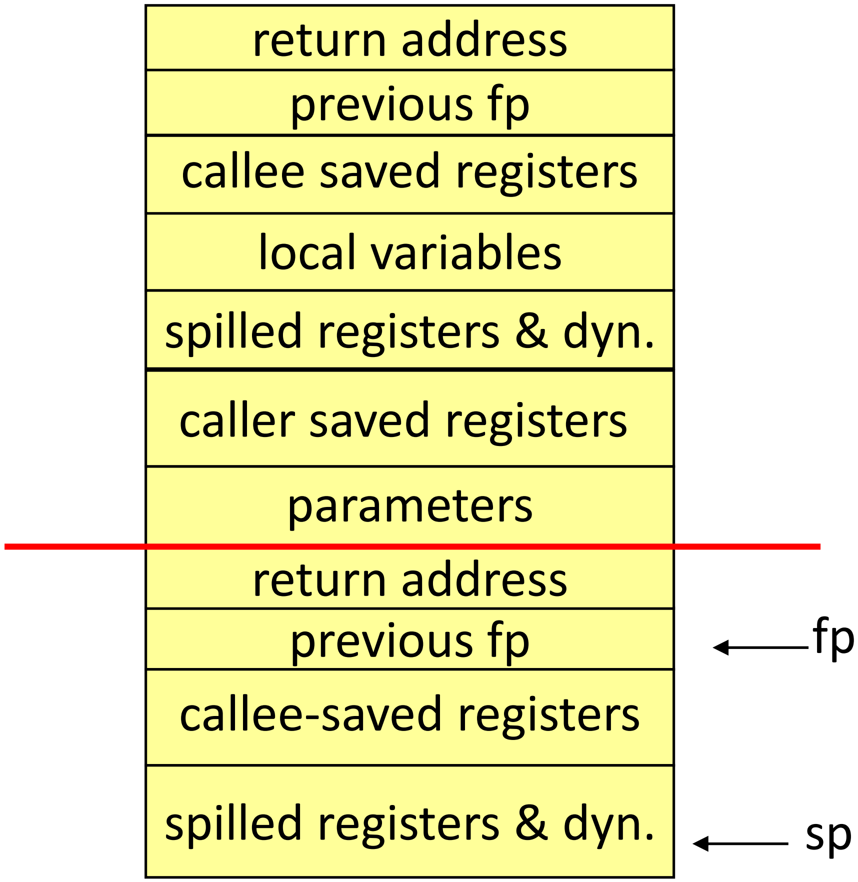
bar() in execution…



The stack

CALLER-CALLEE

foo() continues…



bar() returns…

Putting it all together (using activation records)

```
mov     r0, #1000        //  param "hello world\n"
str     r12, [r13, #24]  //  caller save r12
bl      _printf          //  call printf()
                         // r0 holds return value (ignored)
ldr     r12, [r13, #24]  // restore caller-saved r12 value
```

```
sub     r13, r13, #16    // allocate space for 2 locals +
str     r4, [r13, #8]    // callee save r4
str     r14, [r13, #12]  // save return address
…                        // function body
mov     r0, #0           // return value 0
ldr     r4, [r13,#8]     // restore callee-saved r4
ldr     r14, [r13, #12]  // restore return address
add     r13, r13, #16    // deallocate call frame
mov     r15, r14         // return to calling function
```

Calculating Caller/Callee Costs

Consider the cost of placing each variable **v** from function **f** in a callee register and a caller register:

Cost = number of store/load instructions required to accomplish the required saving/restoring

Callee_cost → save at the start of the function, restore at end
= 2 * number of invocations of f

Caller_cost → potentially save/restore across each funct. call in f
Caller cost = 0
For each function call in f, $call_i$
if (v is live) caller_cost += 2 * number of times $call_i$ is executed

Caller/Callee Selection

- Select assignment of variables to registers such that the sum of caller/callee costs is minimized
  - Execute fewest save/restores
- Each function greedily picks its own assignment ignoring the assignments in other functions
  - Calling convention assures all necessary registers will be saved
- 2 types of problems
  1. Given a single function → Assume it is called 1 time
  2. Set of functions or program → Compute number of times each function is called if it is obvious (i.e., loops with known trip counts or you are told)

Assumptions

CALLER-CALLEE

- A function can be invoked by many different call sites in different functions.
- Assume no inter-procedural analysis (hard problem)
  - A function has no knowledge about which registers are used in either its caller or callee
  - Assume main() is not invoked by another function
- Implication
  - Any register allocation optimization is done using function local information

Class Problem

```
foo() {
  a = …
  b = …
  bar();
  … = a;
  … = b;
  for (1 to 15) {
    c =  …
    d =  …
              … = c;
    printf();
              … = d;
  }
}
```

Assume that you have 2 caller and 2 callee save registers. Pick the best assignment for a, b, c, d. Assume each requires its own register.

Caller-saved vs. callee saved – Multiple function case

```
void main(){
  int a,b,c,d;
  .
  c = 5; d = 6;
  a = 2; b = 3;
  foo();
  d = a+b+c+d;
  .
  .
  .
}
```

```
void foo(){
  int a,b;
  .
  .
  a = 2; b = 3;
  bar();
  a = a + b;
  .
  .
  .
}
```

```
void bar(){
  int a,b,c,d;
  .
  c = 0; d = 1;
  a = 2; b = 3;
  final();
  a = a+b+c+d;
  .
  .
  .
}
```

```
void final(){
  int a,b,c;
  .
  .
  a = 2; b = 3;
  .
  c = a+b;
  .
  .
  .
}
```

Note: assume main does not have to save any callee reg. (that is really the case for start)

Caller-saved vs. callee saved – Multiple function case

- Questions:
  1. In assembly code, how many regs. need to be stored/loaded in total if we use a **caller-save** convention ?
  2. In assembly code, how many regs. need to be stored/loaded in total if we use a **callee-save** convention ?
  3. In assembly code, how many regs. need to be stored/loaded in total if we use a mixed **caller/callee**-save convention with 3 callee-s. and 3 caller-s. registers ?

  4. Assume bar() is in a loop inside foo() and the loop is iterated 10 times ? When the program is executed, how many regs. need to be stored/loaded in total for each of the above three scenarios?

Question 1: Caller-save

```
void main(){
   .
   .
   .
   [4 str]
   foo();
   [4 ldr]
   .
   .
   .
}
```

```
void foo(){
   .
   .
   .
   [2 str]
   bar();
   [2 ldr]
   .
   .
   .
}
```

```
void bar(){
   .
   .
   .
   [4 str]
   final();
   [4 ldr]
   .
   .
   .
}
```

```
void final(){
   .
   .
   .
   .
   .
   .
   .
   .
   .
}
```

Total: 10 str / 10 ldr

Question 2: Callee-save

```
void main(){
   .
   .
   .
   .
   foo();
   .
   .
   .
   .
}
```

```
void foo(){
   [2 str]
   .
   .
   .
   bar();
   .
   .
   .
   [2 ldr]
}
```

```
void bar(){
   [4 str]
   .
   .
   .
   final();
   .
   .
   .
   [4 ldr]
}
```

```
void final(){
   [3 str]
   .
   .
   .
   .
   .
   .
   .
   [3 ldr]
}
```

Total: 9 str / 9 ldr

Question 3: Mixed 3 caller / 3 callee

```
void main(){
   .
   .
   .
   [1 str]
   foo();
   [1 ldr]
   .
   .
   .
}
```

```
void foo(){
   [2 str]
   .
   .
   .
   bar();
   .
   .
   .
   [2 ldr]
}
```

```
void bar(){
   [4 str]
   .
   .
   .
   final();
   .
   .
   .
   [4 ldr]
}
```

```
void final(){
   .
   .
   .
   .
   .
   .
   .
   .
   .
}
```

1 caller r.
3 callee r.

3 caller r.

Total: 7 str / 7 ldr

Caller-saved vs. callee saved – Question 4

- Mixed 3 caller / 3 callee

```
void main(){
   .
   .
   .
   [1 str]
   foo();
   [1 ldr]
   .
   .
   .
}
```

```
void foo(){
   [2 str]
   .
   .
   .
   bar();
   .
   .
   .
   [2 ldr]
}
```

loop x10

```
void bar(){
   [4 str]
   .
   .
   .
   final();
   .
   .
   .
   [4 ldr]
}
```

```
void final(){
   .
   .
   .
   .
   .
   .
   .
   .
   .
}
```

1 caller r.
3 callee r.

2 callee r.

x10

3 caller r.

x10

Total: 43 str / 43 ldr *Pure caller: (4+20+40+0) str / ldr - Pure callee (0+2+40+30) str / ldr*

Caller-saved vs. callee saved – A more interesting case

```
void main(){
  int a,b,c,d;
  .
  c = 5; d = 6;
  a = 2; b = 3;
  foo();
  d = a+b+c+d;
  .
  .
  .
}
```

```
void foo(){
  int a,b,c,d,e;
  .
  c = 1; d = 1;
  e = 1;

  a = 2; b = 3;
  foo(e-1,b+1);
  a = a + b;
  .
  .
  a = 5, b =4;
  foo(b,9);
  b = a - b;
  .
  c++; d++; e++;
}
```

Caller-saved vs. callee saved – the interesting case

- Assume the function foo() is called recursively 15 times in total
- When the program is executed, how many regs need to be stored/loaded in total for the following scenarios:
  - Use a **caller-save** convention ?
  - Use a **callee-save** convention ?
  - Use a mixed **caller/callee**-save convention with 3 callee-s. and 3 caller-s. registers ?