

# **6. Instruction Set Architecture – from C to assembly – Functions**

---

**EECS 370 – Introduction to Computer Organization – Winter 2023**

**EECS Department  
University of Michigan in Ann Arbor, USA**

# Announcements—Reminders

---

- ❑ **Project 1.a due Thursday!**
- ❑ **Project 1.s and 1.m due Thursday 2/2**
- ❑ **Homework 3 due Monday 2/6**
  - Group and individual (turned in separately)
  - For group part, if you don't have a team, consider using the "search for teammates" message on Piazza.
    - Or talk to someone in this room.
- ❑ **All due dates on calendar on web page.**

# Instruction Set Architecture (ISA) Design Lectures

---

- ❑ Lecture 2: Storage types
- ❑ Lecture 3 : Addressing modes and LC2K
- ❑ Lecture 4 : ARM Assembly
- ❑ Lecture 5 : C to Assembly
- ❑ **Lecture 6 : Function calls**
- ❑ Lecture 7: Linker and Floating Point

# Class Problem

---

Write the ARM assembly code to implement the following C code:

```
// assume ptr is in X1
// struct {int val; struct node *next;} node;
// struct node *ptr;
```

```
if ((ptr != NULL) && (ptr->val > 0))
    ptr->val++;
```

```
cmpi X1, #0
b.eq Endif
ldur X2, [X1, #0]
cmpi X2, #0
b.le Endif
add X2, X2, #1
str X2, [X1, #0]
Endif : ....
```

# Branching far away

- ❑ The underlying philosophy of ISA design and microarchitecture in general is to **make the common case fast**
- ❑ In the case of branches, you are commonly going to branch to other instructions nearby.
  - In ARMv8, the encoding for the displacement of conditional branches is 19 bits.
  - Having a displacement of 19 bits is usually enough
- ❑ BUT what if we need to a target (Label) that we cannot get to with a 19 bit displacement from the current PC?
 

```
CBZ      X15, FarLabel
```
- ❑ The assembler is smart enough to replace that with
 

```
CBNZ     X15, L1
B        FarLabel

L1:      .....
```
- ❑ The simple branch instruction (B) has a 26 bit offset which spans about 64 million instructions!

# Unconditional Branching Instructions

Unconditional branch	branch	B      2500	go to PC + 10000	Branch to target address; PC-relative
	branch to register	BR      X30	go to X30	For switch, procedure return
	branch with link	BL      2500	X30 = PC + 4; PC + 10000	For procedure call PC-relative

- ❑ There are three types of unconditional branches in the LEGv8 ISA.
  - The first (**B**) is the PC relative branch with the 26 bit offset from the last slide.
  - The second (**BR**) jumps to the address contained in a register (X30 above)
  - The third (**BL**) is like our PC relative branch but it does something else.
    - It sets X30 (always) to be the current PC+4 before it branches.

- ❑ Why is BL storing PC+4 into a register?

## Branch with Link (BL)

---

- ❑ Branch with Link is the branch instruction used to call functions
  - Functions need to know where they were called from so they can return.
    - In particular they will need to return to right after the function call
    - Can use “BR X30”
  
- ❑ Say that we execute the instruction BL #200 when at PC 1000.
  - What address will be branched to?
  - What value is stored in X30?
  - How is that value in X30 useful?

# 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 so we can get back
- 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

- ❑ 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 slow
- ❑ ARMv8 solution—and the usual answer:
  - Registers and memory
    - Put the first few parameters in registers (if they fit) (X0 – X7)
    - Put the rest in memory on the call stack— **important concept**
- ❑ Comment: Make sure you understand the general idea behind a stack data structure—ubiquitous in computing
  - As basic concept it is a list in that you can only access at one end by **pushing** a data item into the top of the stack and **popping** an item off of the stack—real stacks are a little more complex

# 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
  - the stack frame is a fixed template of memory locations

# An Older ARM (Linux) Memory Map

**Stack:** starts at 0x0000 007F FFFF FFFC and grows down to lower addresses. Bottom of the stack resides in the SP register

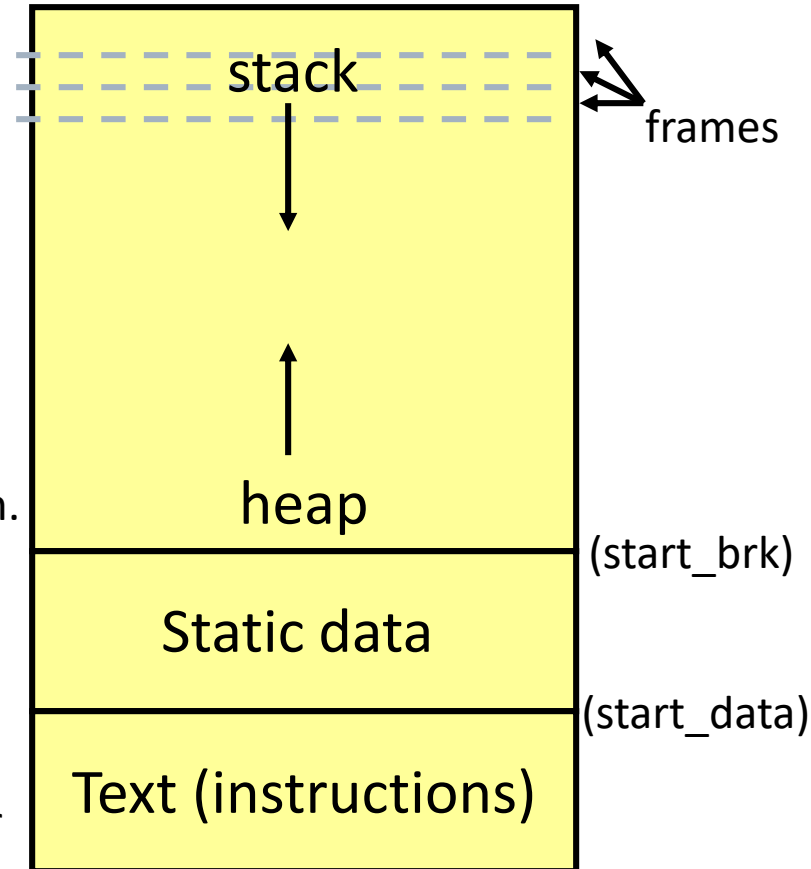
**Heap:** starts above static data and grows up to higher addresses. Allocation done explicitly with malloc(). Deallocation with free(). Runtime error if no free memory before running into SP address. NB not same as data structure heap—just uninitialized mem.

**Static:** starts above text.

Holds all global variables and those locals explicitly declared as “static”.

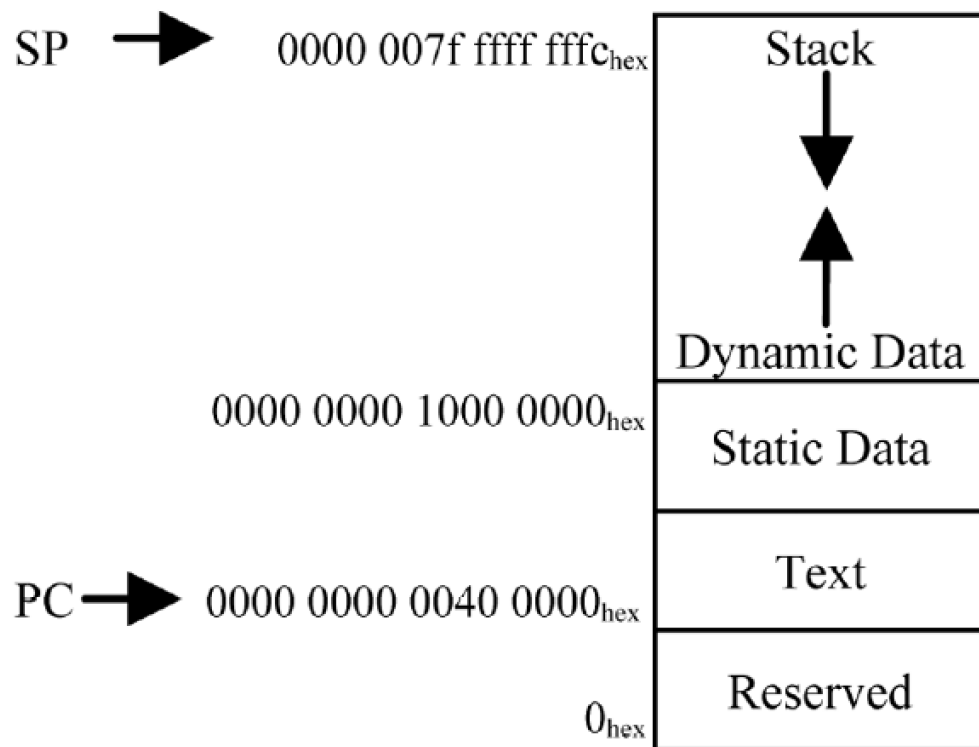
**Text:** starts at 0x0000 0000 0004 0000.

Holds all instructions in the program (except for dynamically linked library routines DLLs)



# Memory Map—details

- ❑ The map at left shows the starting points of the various memory regions
- ❑ Because all instructions are in the text region, the PC will always point into that region
- ❑ The stack pointer SP points to the TOS (top-of-stack)
- ❑ This layout is purely a convention



# 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

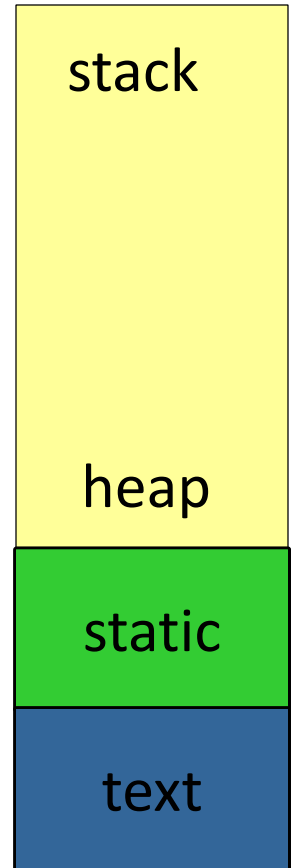
x goes on the stack, as it's a parameter

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

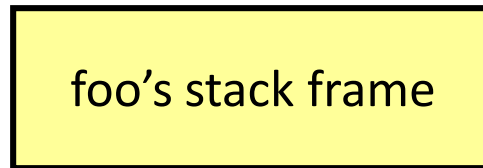


# The stack grows as functions are called

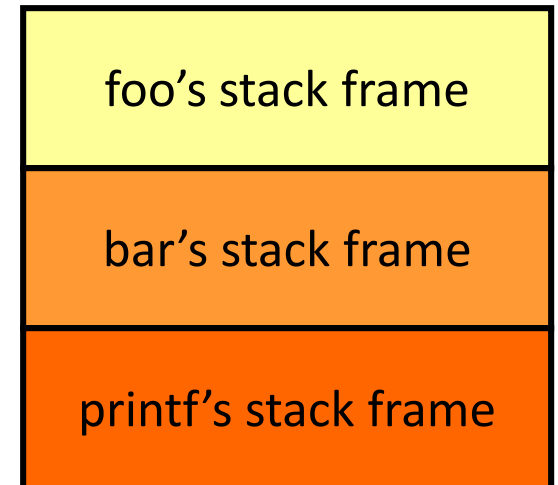
```
void foo()
```

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

inside foo



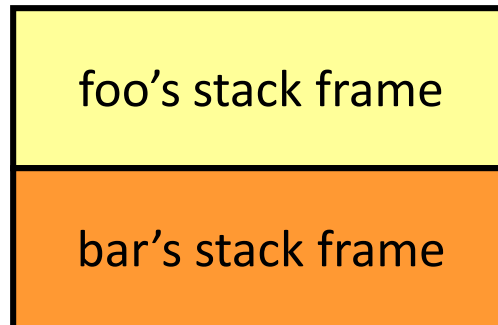
bar calls printf



```
void bar(int x)
```

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

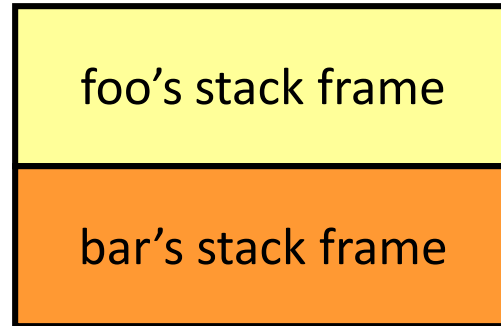
foo calls bar



# The stack shrinks as functions return

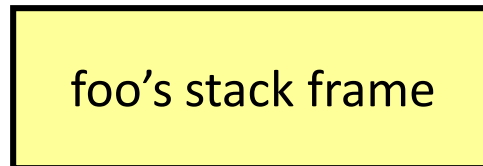
```
void foo()  
{  
    int x, y[2];  
    bar(x);  
}
```

printf returns



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

bar returns



## 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 registers in foo

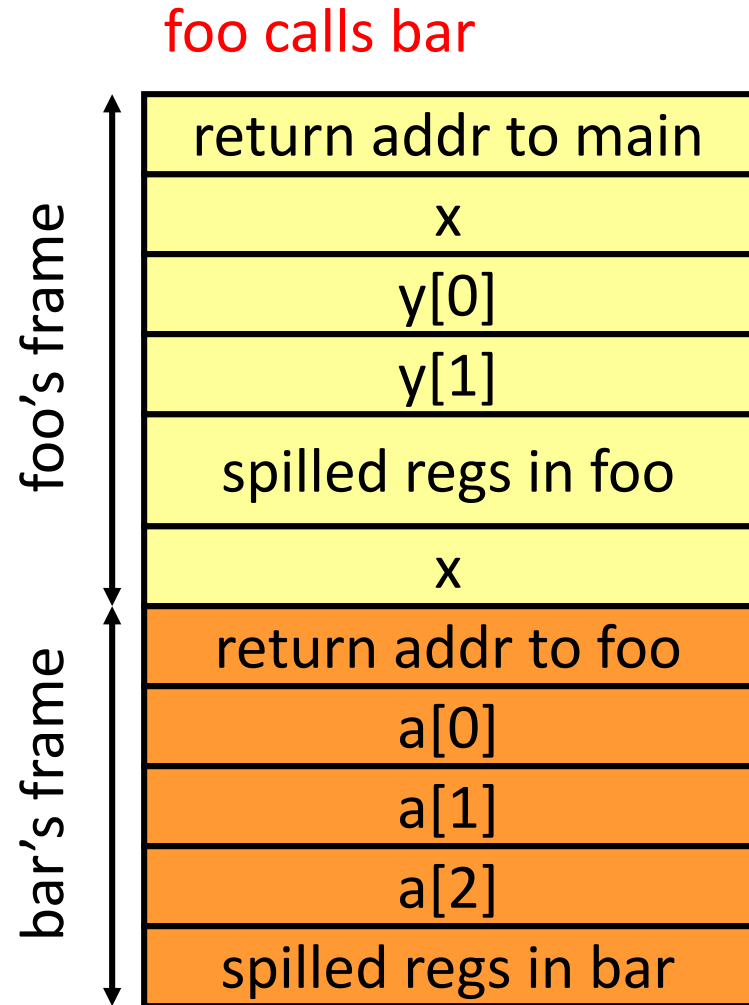


## Stack frame contents (2)

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

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

**Spill data**—not enough room in x0-x7 for  
params and also caller and callee saves



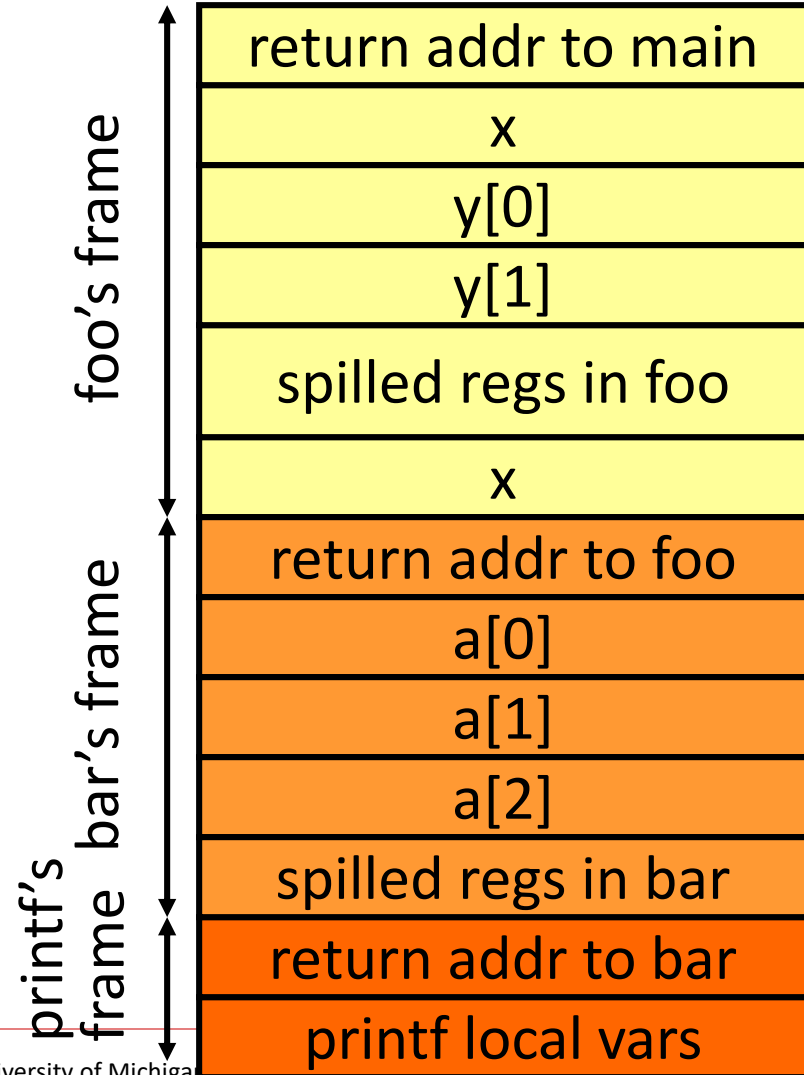
## Stack frame contents (3)

FUNCTION CALLS

bar calls printf

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

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



# Recursive function example

FUNCTION CALLS

```
main()
```

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

main calls foo

```
void foo(int a)
```

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

foo calls foo

foo calls foo

return addr to ...

2

return addr to main

x, y[0], y[1]

spills in foo

1

return addr to foo

x, y[0], y[1]

spills in foo

0

return addr to foo

x, y[0], y[1]

spills in foo

# What about values in registers?

---

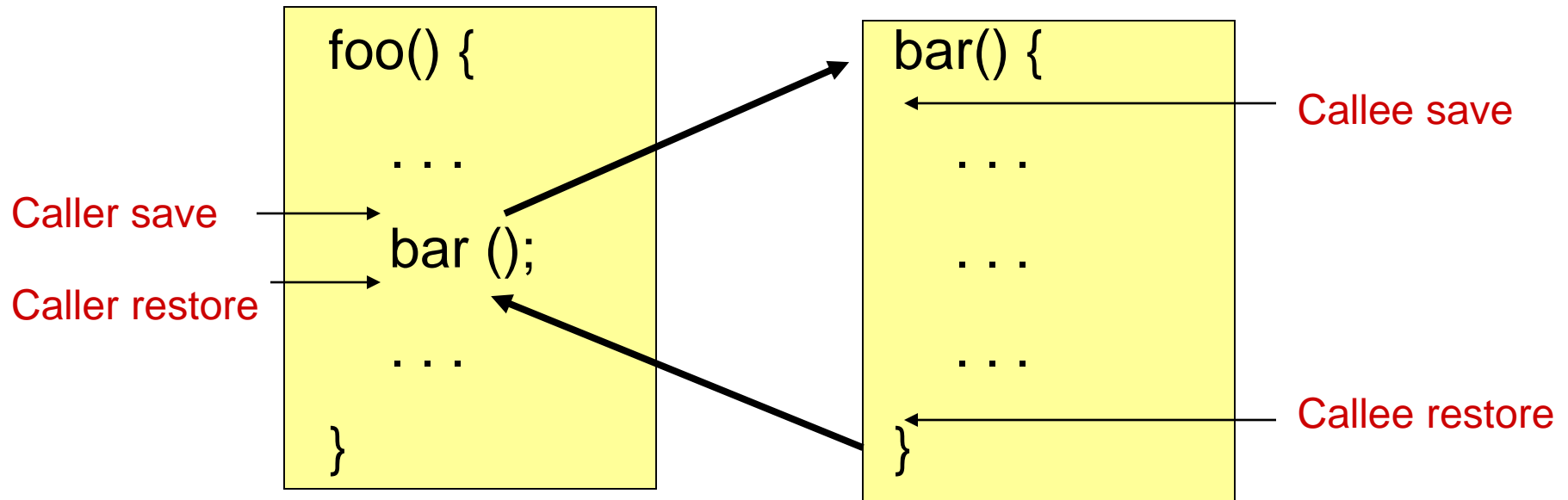
- ❑ When function “foo” calls function “bar”, function “bar” is, like all assembly code, going to store some values in registers.
- ❑ But function “foo” might have some values stored in registers that it wants to use after the call.
  - How can “foo” be sure “bar” won’t overwrite those values?
  - Answer: “foo” needs to save those values to memory (on the stack) before it calls “bar”.
    - Now “bar” can freely use registers
  - And “foo” will have to copy the values back from memory once “bar” returns.
- ❑ In this case the “caller” (foo) is saving the registers to the stack.
  - One could imagine the “callee” (bar) saving the registers.

## “caller-save” vs. “callee-save”

---

- ❑ So we have two basic options:
  - You can save your registers **before** you make the function call and restore the registers when you return (**caller-save**).
    - What if the function you are calling doesn't use that register? No harm done, but wasted work!!!
  - You can save your registers **after** you make the function call and restore the registers before you return (**callee-save**).
    - What if the caller function doesn't use that register? No harm done, but wasted work!!!
- ❑ Most common scheme is to have some of the registers be the responsibility of the caller, and others be the responsibility of the callee.

# Caller-Callee save/restore



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

**Callee save registers:** Must be the same value as when called. May do this by either not changing the value **or** by inserting saves at the start of the function and restores at the end

# Saving/Restoring Optimizations

---

## ❑ 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 (a function that calls no other function), caller saves can be used without saving/restoring

## ❑ Callee-saved

- Only needs saving at beginning of function and restoring at end of function
- Only save/restore it if function overwrites the register

## ❑ Each has its advantages. Neither is always better.

# 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?



# 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

---

- ❑ 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

# 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 e,f;  
    .  
    .  
    e = 2; f = 3;  
    bar();  
    e = e + f;  
    .  
    .  
    .  
}
```

```
void bar(){  
    int g,h,i,j;  
    .  
    .  
    g = 0; h = 1;  
    i = 2; j = 3;  
    final();  
    j = g+h+i;  
    .  
    .  
    .  
}
```

```
void final(){  
    int y,z;  
    .  
    .  
    y = 2; z = 3;  
    .  
    z = y+z;  
    .  
    .  
    .  
}
```

Note: assume main does not have to save any callee registers

## Caller-saved vs. callee saved – Multiple function case

---

### ❑ Questions:

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

## Question 1: Caller-save

---

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

```
void foo(){  
    int e,f;  
    .  
    .  
    e = 2; f = 3;  
    [2 STUR]  
    bar();  
    [2 LDUR]  
    e = e + f;  
    .  
    .  
}
```

```
void bar(){  
    int g,h,i,j;  
    .  
    .  
    g = 0; h = 1;  
    i = 2; j = 3;  
    [3 STUR]  
    final();  
    [3 LDUR]  
    j = g+h+i;  
    .  
    .  
    .  
}
```

```
void final(){  
    int y,z;  
    .  
    .  
    y = 2; z = 3;  
    .  
    z = y+z;  
    .  
    .  
    .  
}
```

Total: 9 STUR / 9 LDUR

## Question 2: Callee-save

---

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

```
void foo(){
    [2 STUR]
    int e,f;
    .
    .
    e = 2; f = 3;
    bar();
    e = e + f;
    .
    [2 LDUR]
}
```

```
void bar(){
    [4 STUR]
    int g,h,i,j;
    .
    g = 0; h = 1;
    i = 2; j = 3;
    final();
    j = g+h+i;
    .
    [4 LDUR]
}
```

```
void final(){
    [2 STUR]
    int y,z;
    .
    .
    y = 2; z = 3;
    .
    z = y+z;
    .
    [2 LDUR]
}
```

Total: 8 STUR / 8 LDUR

## Caller-save and callee-save registers

---

- ❑ Again, what we really tend to do is have some of the registers be the responsibility of the caller and others the responsibility of the callee.
  - So if you have six registers, maybe r0-r2 are caller-save registers and r3-r5 are callee save registers.
  
- ❑ How does that help?

## Question 3: Mixed 3 caller / 3 callee

For main, we'd like to put all the variables into callee save registers. But we only have 3 callee save registers (r3-r5), so one variable needs to end up in a caller-save register.

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

    [1 STUR]
    foo();
    [1 LDUR]

    d = a+b+c+d;
}
```

1 caller r.  
3 callee r.

```
void foo() {
    [2 STUR]
    int e,f;
    .
    .
    e = 2; f = 3;
    bar();
    e = e + f;
    .
    .
    [2 LDUR]
}
```

2 callee r.

```
void bar() {
    [3 STUR]
    int g,h,i,j;
    .
    .
    g = 0; h = 1;
    i = 2; j = 3;
    final();
    j = g+h+i;
    .
    .
    .
    [3 LDUR]
}
```

1 caller r.  
3 callee r.

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

3 caller r.

**Total: 6 STUR / 6 LDUR** For foo it doesn't really matter what registers we use.  
Either way we will have to save and restore 2 values



## It can get quite a bit more complex than this

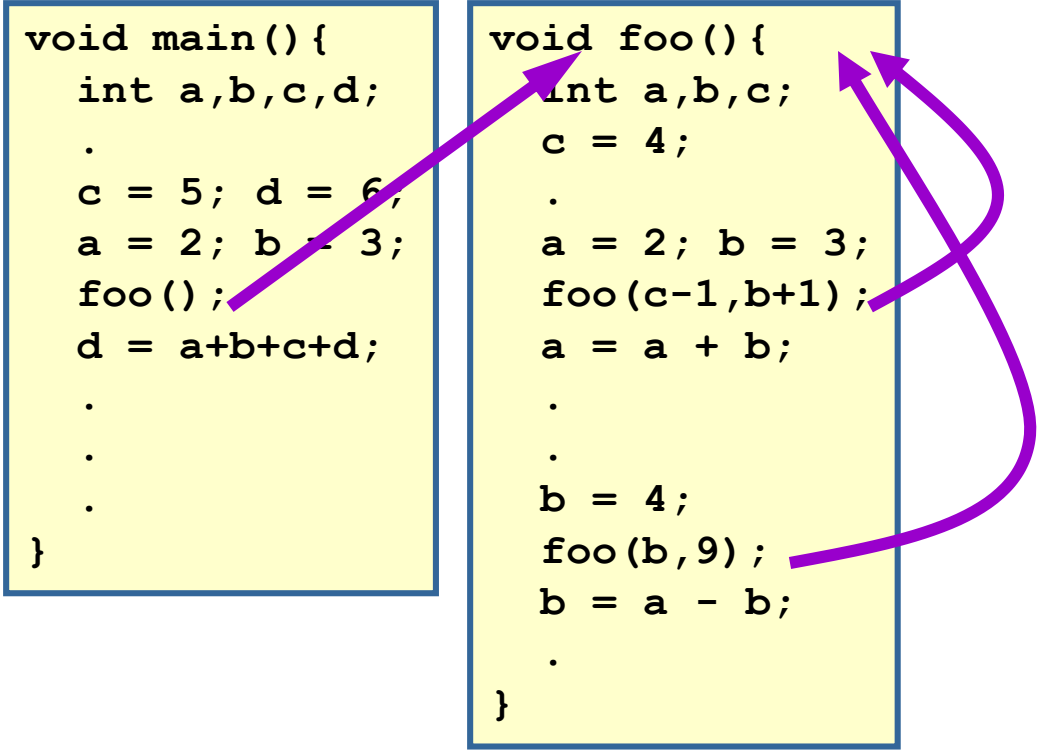
---

- ❑ Multiple function calls in a given function will make things more complex
  - As will loops
  
- ❑ The video review for caller/callee save found on the class website is quite useful
  - You'll also get some examples in discussion

## Does Recursion Change Caller/Callee?

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



- ❑ No! Treat foo() just like any normal function and assume you are calling an unknown function.

# LEGv8 ABI—Application Binary Interface

---

- ❑ The ABI is an agreement about how to use the various registers. These can be broken into three groups
  - Some registers are reserved for special use Register usage definitions
    - (X31) zero register, (X30) link register, (X29) frame pointer, (x28) stack pointer, (X16-18) reserved for other uses.
- ❑ Callee save: X19-X27
- ❑ Caller save: X0-X15
  - In addition, we pass arguments using registers X0-X7 (and memory if there are more arguments)
  - Return value goes into X0

# LEGv8 Stack Frame

- ❑ Must be aligned on 16 byte boundaries
- ❑ FP (Frame pointer, found in X29) provides a fixed address from which to access items in the current stack frame
- ❑ Stack frames are connected via a linked list of FPs
- ❑ We can do without frame pointers if we carefully track the stack pointer (SP), **but** FP makes life easier...

Previous frame

