Poll: Who is responsible for erasing a whiteboard in a public space?

a) The person who is done using it
b) The person who is about to use it

EECS 370 - Lecture 6

Function Calls

Announcements

- P1
  - Part a due Thu @ 11:59 via Autograder
  - Project 1 s + m due Thu 1/26

- HW 2
  - Posted on website, due Mon 2/6 at 8 pm
- Get exam conflicts and SSD accommodations sent to us **ASAP**
  - Forms listed on the website

Instruction Set Architecture (ISA) Design Lectures

- Lecture 2: ISA - storage types, binary and addressing modes
- Lecture 3 : LC2K
- Lecture 4 : ARM
- Lecture 5 : Converting C to assembly – basic blocks
- **Lecture 6 : Converting C to assembly – functions**
- Lecture 7 : Translation software; libraries, memory layout

ARM/LEGv8 Sequencing Instructions

- Sequencing instructions change the flow of instructions that are executed
  - This is achieved by modifying the program counter (PC)
- Unconditional branches are the most straightforward
  they ALWAYS change the PC and thus "jump" to another instruction out of the usual sequence
- Conditional branches

If (condition_test) goto target_address

condition_test examines the four flags from the processor status word (SPSR)
target_address is a 19 bit signed <u>word</u> displacement on current PC

LEGv8 Conditional Instructions

- Two varieties of conditional branches
  1. One type compares a register to see if it is equal to zero.
  2. Another type checks the condition codes set in the status register.

Conditional branch	compare and branch on equal 0	`CBZ X1, 25`	`if (X1 == 0) go to PC + 100`	Equal 0 test; PC-relative branch
	compare and branch on not equal 0	`CBNZ X1, 25`	`if (X1 != 0) go to PC + 100`	Not equal 0 test; PC-relative branch
	branch conditionally	`B.cond 25`	`if (condition true) go to PC + 100`	Test condition codes; if true, branch

- Talked about CBZ and CBNZ last time
- Let's look at B.cond now

LEGv8 Conditional Instructions

- Motivation:
  - Some types of branches makes sense to check if a certain value is zero or not
    - while(a)
  - But not all:
    - if(a > b)
    - if(a == b)
  - Using an extra **program status register** to check for various conditions allows for a greater breadth of branching behavior

LEGv8 Conditional Instructions Using FLAGS

- FLAGS: NZVC record the results of (arithmetic) operations Negative, Zero, oVerflow, Carry—not present in LC2K
- We explicitly set them using the "set" modification to ADD/SUB etc.
- Example: ADDS causes the 4 flag bits to be set according as the outcome is negative, zero, overflows, or generates a carry

Category	InstructionExample		Meaning	Comments
Arithmetic	add	ADD X1, X2, X3	X1 = X2 + X3	Three register operands
	subtract	SUB X1, X2, X3	X1 = X2 – X3	Three register operands
	add immediate	ADDI X1, X2, 20	X1 = X2 + 20	Used to add constants
	subtract immediate	SUBI X1, X2, 20	X1 = X2 – 20	Used to subtract constants
	add and set flags	ADDS X1, X2, X3	X1 = X2 + X3	Add, set condition codes
	subtract and set flags	SUBS X1, X2, X3	X1 = X2 – X3	Subtract, set condition codes
	add immediate and set flags	ADDIS X1, X2, 20	X1 = X2 + 20	Add constant, set condition codes
	subtract immediate and set flags	SUBIS X1, X2, 20	X1 = X2 – 20	Subtract constant, set condition codes

ARM Condition Codes Determine Direction of Branch

- In LEGv8 only ADDS / SUBS / ADDIS / SUBIS / CMP /CMPI set the condition codes FLAGs or condition codes in PSR—the program status register
- Four primary condition codes evaluated:
  - N – set if the result is negative (i.e., bit 63 is non-zero)
  - Z – set if the result is zero (i.e., all 64 bits are zero)
  - ~~C – set if last addition/subtraction had a carry/borrow out of bit 63~~
  - ~~V – set if the last addition/subtraction produced an overflow (e.g., two negative numbers added together produce a positive result)~~
- Don't worry about the C and V for this class

ARM Condition Codes Determine Direction of Branch--continued

Encoding	Name (& alias)	Meaning (integer)	Flags
0000	EQ	Equal	Z==1
0001	NE	Not equal	Z==0
0010	HS (CS)	Unsigned higher or same (Carry set)	C==1
0011	LO (CC)	Unsigned lower (Carry clear)	C==0
0100	MI	Minus (negative)	N==1
0101	PL	Plus (positive or zero)	N==0
0110	VS	Overflow set	V==1
0111	VC	Overflow clear	V==0
1000	HI	Unsigned higher	C==1 && Z==0
1001	LS	Unsigned lower or same	!(C==1 && Z==0)
1010	GE	Signed greater than or equal	N==V
1011	LT	Signed less than	N!=V
1100	GT	Signed greater than	Z==0 && N==V
1101	LE	Signed less than or equal	!(Z==0 && N==V)
1110	AL	Always	Any
1111	NV†		

Need to know the 7 with the red arrows

Conditional Branches: How to use

- CMP instruction lets you compare two registers.
  - Could also use SUBS etc.
    - That could save you an instruction.
- B.cond lets you branch based on that comparison.

- Example:

```
CMP   X1, X2
B.GT Label1
```

  - Branches to Label1 if X1 is greater than X2.

Branch—Example

- Convert the following C code into LEGv8 assembly (assume x is in X1, y in X2):

```
int x, y;
if (x == y)
  x++;
else
  y++;
// …
```

Branch—Example

- Convert the following C code into LEGv8 assembly (assume x is in X1, y in X2):

```c
int x, y;
if (x == y)
  x++;
else
  y++;
// …
```

Using Labels

```
     CMP  X1, X2
     B.NE L1
     ADD  X1, X1, #1
     B    L2
L1:  ADD  X2, X2, #1
L2:  …
```

Note that conditions in assembly are often the inverse of the "if" condition. Why?

Without Labels

```
CMP     X1, X2
B.NE    3
ADD     X1, X1, #1
B       2
ADD     X2, X2, #1
```

Assemblers must deal with labels and assign displacements

Loop—Example

// assume all variables are long long integers (64 bits or 8 bytes)
// i is in X1, start of a is at address 100, sum is in X2

```
sum = 0;
for (i=0 ; i < 10 ; i++) {
  if (a[i] >= 0) {
    sum += a[i];
  }
}
```

```
           MOV     X1, XZR
           MOV     X2, XZR
Loop1:     CMPI    X1, #10
           B.EQ    endLoop
           LSL     X6, X1, #3
           LDUR    X5, [X6, #100]
           CMPI    X5, #0
           B.LT    endif
           ADD     X2, X2, X5
endif:     ADDI    X1, X1, #1
           B       Loop1
endLoop:
```

# of branch instructions
= 3*10 + 1= 31

a.k.a. while-do template

Extra Example: Do-while Loop

Look through this on your own

// assume all variables are long long integers (64 bits or 8 bytes)
// i is in X1, start of a is at address 100, sum is in X2

```
sum = 0;
for (i=0 ; i < 10 ; i++) {
  if (a[i] >= 0) {
    sum += a[i];
  }
}
```

```
          MOV    X1, XZR
          MOV    X2, XZR
Loop1:    LSL    X6, X1, #3
          LDUR   X5, [X6, #100]
          CMPI   X5, #0
          B.LT   endif
          ADD    X2, X2, X5
endIf:    ADDI   X1, X1, #1
          CMPI   X1, #10
          B.LT   Loop1
endLoop:
```

# of branch instructions
= 2*10 = 20

a.k.a. do-while template

Extra Problem – For Your Reference

- 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++;
```

Extra Problem

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

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

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

```
cmp r1, #0
        beq Endif
        ldursw  r2, [r1, #0]
        cmp r2, #0
        b.le Endif
        add r2, r2, #1
        str r2, [r1, #0]
Endif : ….
```

Branching far away

- Underlying philosophy of ISA design: **make the common case fast**
- Most branches target nearby instructions
  - Displacement of 19 bits is usually enough
- BUT what if we need to branch really far away (more than 2^{19} words)?

```
        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!
- In LC2K, we can do a similar thing by using JALR instead of BEQ

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?

Poll

Converting function calls to assembly code

C: factorial(5);

- Need to pass parameters to the called function—factorial
- Need to save return address of caller so we can get back
- Need to save register values (why?)
- Need to jump to factorial

Execute instructions for factorial()
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:
  - Both
    - Put the first few parameters in registers (if they fit) (X0 – X7)
    - Put the rest in memory on the call stack— important concept

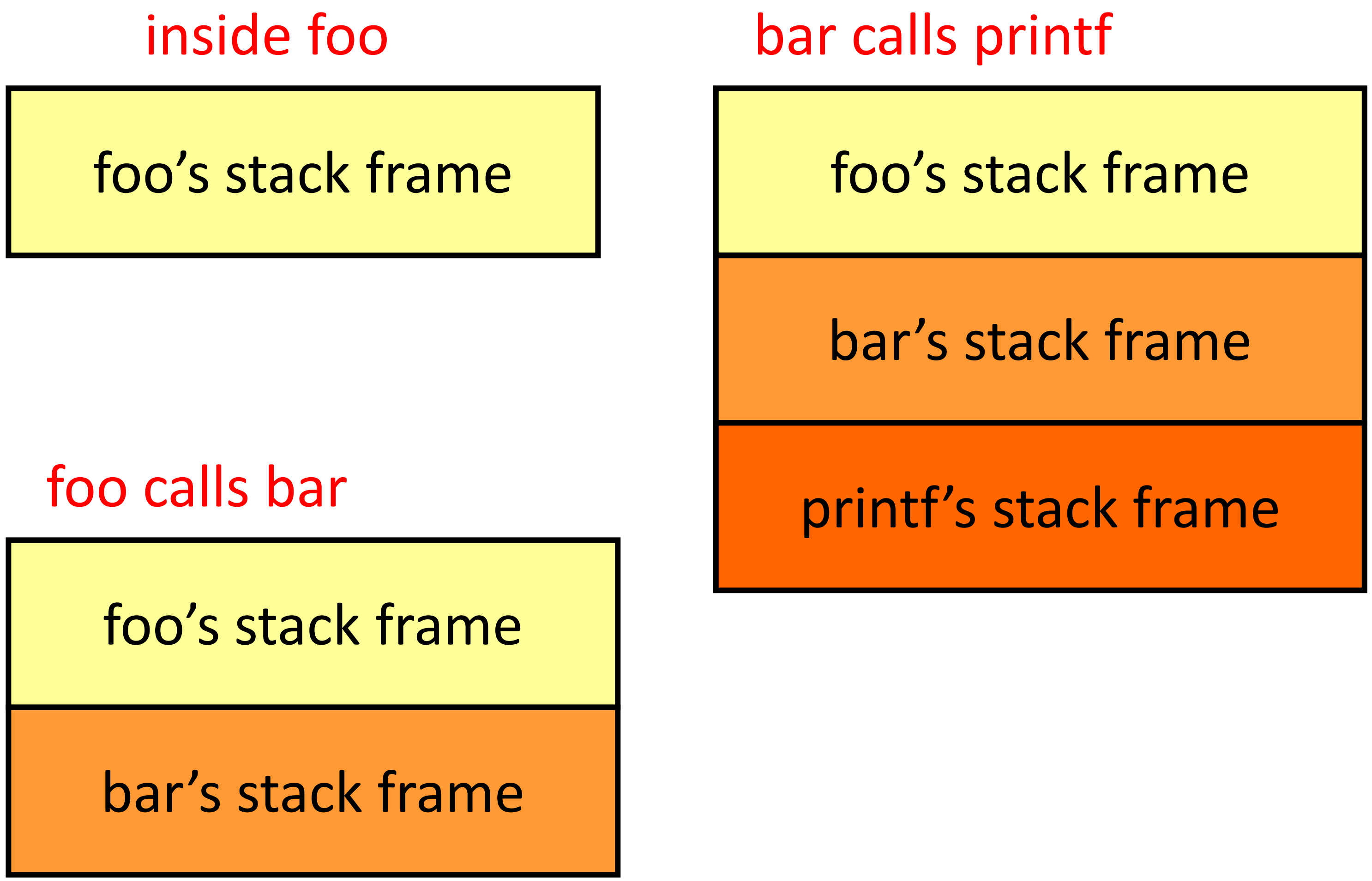
Call stack

- 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 [**stack frames**] are allocated when you make a function call, and de-allocated when you return from a function

The stack grows as functions are called

FUNCTION CALLS

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

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



The stack shrinks as functions return

FUNCTION CALLS

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

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

printf returns

foo's stack frame
bar's stack frame

bar returns

foo's stack frame

Stack frame contents

FUNCTION CALLS

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

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

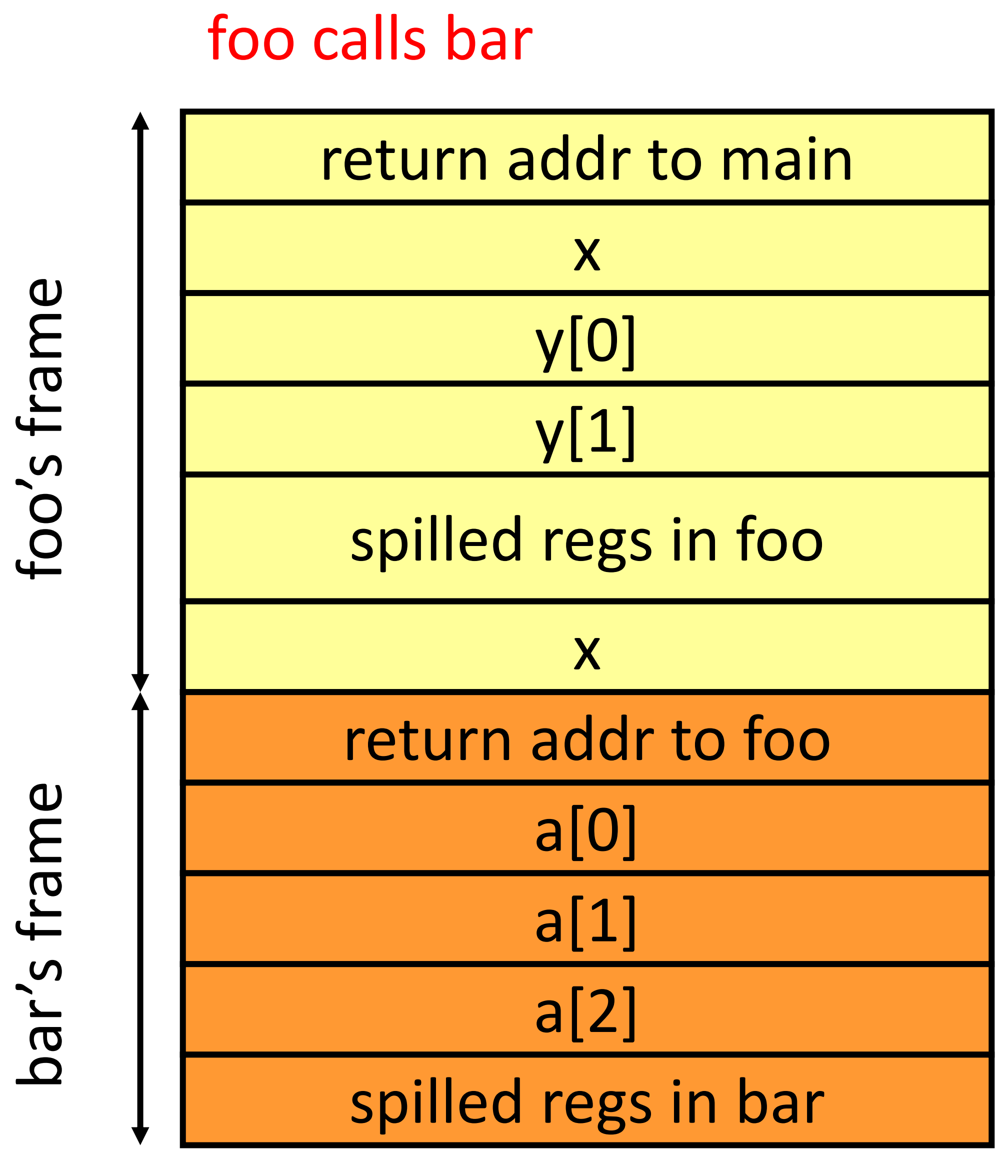
foo’s stack frame

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

Stack frame contents (2)

FUNCTION CALLS

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

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



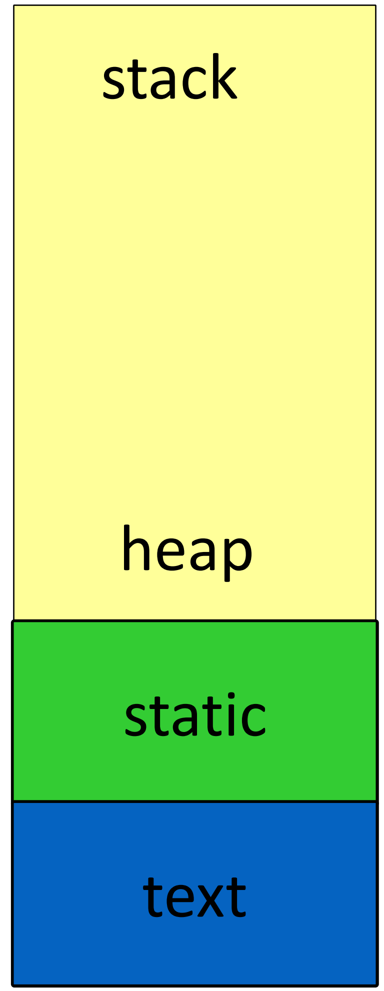
Spill data-–not enough room in x0-x7 for params and also caller and callee saves

Review: Where do the variables go?

Assigning variables to memory spaces

FUNCTION CALLS

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



Assigning variables to memory spaces

FUNCTION CALLS

```c
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

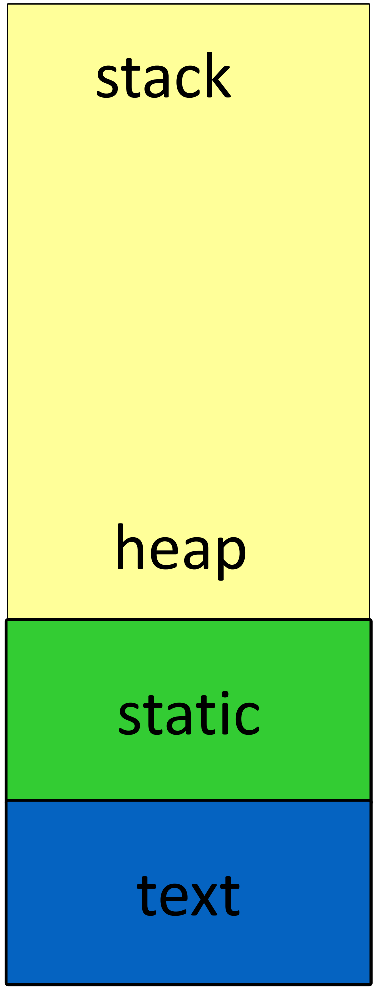
x goes on the stack, as it's a parameter

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 addresses of local variables will be different depending on where we are in the call stack

ld r13, r1, STACK
// Here, r13 is holding the offset of how far we are down the stack

What about registers?

- Higher level languages (like C/C++) provide many abstractions that don't exist at the assembly level
- E.g. in C, each function has its own local variables
  - Even if different function have local variables with the same name, they are independent and guaranteed not to interfere with each other!

Still prints "1"... these don't interfere

```
void foo(){
   int a=1;
   bar();
   printf(a);
}
```

```
void bar(){
   int a=2;
   return;
}
```

What about registers?

- But in assembly, all functions share a small set (e.g. 32) of registers
  - Called functions will overwrite registers needed by calling functions

```
main: movz X0, #1
      bl foo
      bl printf
```

foo() overwrites X0 if we don't do something!!

```
foo: movz X0, #2
     br X30
```

- "Someone" needs to save/restore values when a function is called to ensure this doesn't happen

Two Possible Solutions

- Either the **called** function saves register values before it overwrites them and restores them before the function returns (**callee** saved)…

```
main: movz X0, #1
      bl foo
      bl printf
```

```
foo:  stur X0, [stack]
      movz X0, #2
      ldur X0, [stack]
      br X30
```

- Or the **calling** function saves register values before the function call and restores them after the function call (**caller** saved)…

```
main: movz X0, #1
      stur X0, [stack]
      bl foo
      ldur X0, [stack]
      bl printf
```

```
foo:  movz X0, #2
      br X30
```

a->r1, b->r2, etc.

Another example

Original C Code

```
void foo(){
  int a,b,c,d;

  a = 5; b = 6;
  c = a+1; d=c-1;


  bar();


  d = a+d;
  return();
}
```

No need to save r2/r3. Why?

Additions for Caller-save

```
void foo(){
  int a,b,c,d;

  a = 5; b = 6;
  c = a+1; d=c-1;
  save r1 to stack
  save r4 to stack
  bar();
  restore r1
  restore r4
  d = a+d;
  return();
}
```

Assume bar() will overwrite registers holding a,d

Additions for Callee-save

```
void foo(){
  int a,b,c,d;
  save r1
  save r2
  save r3
  save r4
  a = 5; b = 6;
  c = a+1; d=c-1;
  bar();
  d = a+d;
  restore r1
  restore r2
  restore r3
  restore r4
  return();
}
```

bar() will save a,b, but now foo() must save main's variables

“caller-save” vs. “callee-save”

- Caller-save
  - What if bar() doesn't use r1/r4?
  - No harm done, but wasted work

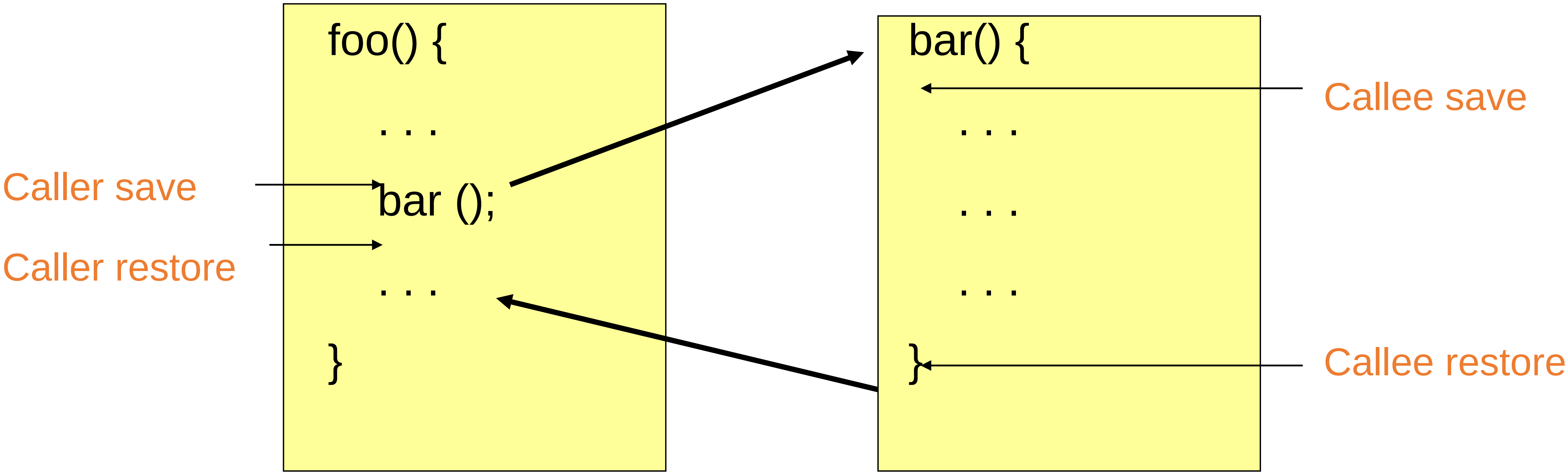
```
void foo(){
  int a,b,c,d;

  a = 5; b = 6;
  c = a+1; d=c-1;
  save r1 to stack
  save r4 to stack
  bar();
  restore r1
  restore r4
  d = a+d;
  return();
}
```

- Caller-save
  - What if main() doesn't use r1-r4?
  - No harm done, but wasted work

```
void foo(){
  int a,b,c,d;
  save r1
  save r2
  save r3
  save r4
  a = 5; b = 6;
  c = a+1; d=c-1;
  bar();
  d = a+d;
  restore r1
  restore r2
  restore r3
  restore r4
  return();
}
```

Another helpful visual

CALLER-CALLEE



Saving/Restoring Optimizations

CALLER-CALLEE

- Where can we avoid loads/stores?
- Caller-saved
  - Only needs saving if value is "live" across 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

a, d are live

b, c are NOT live

```
void foo(){
  int a,b,c,d;

  a = 5; b = 6;
  c = a+1; d=c-1;

  bar();

  d = a+d;
  return();
}
```

Saving/Restoring Optimizations

CALLER-CALLEE

- Where can we avoid loads/stores?
- Callee-saved
  - Only needs saving at beginning of function and restoring at end of function
  - Only save/restore it if function overwrites the register

Only use r1-r4

No need to save other registers

```
void foo(){
  int a,b,c,d;

  a = 5; b = 6;
  c = a+1; d=c-1;

  bar();

  d = a+d;
  return();
}
```

Caller versus Callee

- Which is better??
- Neither is obviously better – depends on specific code
- Common solution: designate some registers as caller-saved, some as callee-saved
- More next time

Next Time

- Finish Up Function Calls
- Talks about linking – the final puzzle piece of software
- Lingering questions / feedback? I'll include an anonymous form at the end of every lecture: https://bit.ly/3oXr4Ah