EECS 370 - Lecture 6

Function Calls

Announcements

- Project 1a due tonight
- Project 1s+m due next Thursday
- Let us know about exam conflicts in the **next week**
  - Form on website

Reminder: Memory Alignment

- Most modern ISAs require that data be aligned
  - An N-byte variable must start at an address A, such that (A%N) == 0
- For example, starting address of a 32 bit **int** must be divisible by 4

int can start here

int CAN'T start here

0x1000	0x1001	0x1002	0x1003
0x12	0x34	0x56	0x78

- Starting address of a 16 bit **short** must be divisible by 2

short can start here

int CAN'T start here

0x1000	0x1001	0x1002	0x1003
0x12	0x34	0x56	0x78

Golden Rule of Alignment

- Every (primitive) object starts at an address divisible by its size
- "Padding" is placed in between objects if needed

```
char  c;
short s;
int   i;
```

0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007
[c]	[padding]	[s]		[i]			

- But what about non-primitive data types?
  - Arrays? Treat as independent objects
  - Structs? Trickier…

Problem with Structs

```
char c;

struct {
  char c;
  int i;
} s[2];
```

- If we align each element of a struct according to the Golden Rule, we can still run into issues
  - E.g.: An array of structs

Amount of padding is different across different instances

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	100A	100B	100C	100D	100E	100F
c	s[0].c	[pad]	[pad]	s[0].i				s[1].c	[pad]	[pad]	[pad]	s[1].i			

- Why is this bad?
- It makes "for" loops very difficult to write!
  - Offsets need to be different on each iteration

Structure Alignment

- Solution: in addition to laying out each field according to Golden Rule…
  - Identify largest (primitive) field
    - Starting address of overall struct is aligned based on the largest field
    - Padded in the back so total size is a multiple of the largest primitive

```
char c;

struct {
  char c;
  int i;
} s[2];
```

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	100A	100B	100C	100D	100E	100F
c	[pad]	[pad]	[pad]	s[0].c	[pad]	[pad]	[pad]	s[0].i				s[1.c]	[pad]	[pad]	[pad]

Guaranteed to lay out each instance identically

Structure Example

```
struct {
  char w;
  int x[3];
  char y;
  short z;
}
```

<u>Poll:</u> What boundary should this struct be aligned to?

a) 1 byte
b) 4 bytes
c) 12 bytes
d) 2 bytes
e) 19 bytes

- Assume struct starts at location 1000,
  - char w → 1000
  - x[0] → 1004-1007, x[1] → 1008 – 1011, x[2] → 1012 – 1015
  - char y → 1016
  - short z → 1018 – 1019

Total size = 20 bytes!

Calculating Load/Store Addresses for Variables

Datatype	size (bytes)
char	1
short	2
int	4
double	8

```
short a[100];
char b;
int c;
double d;
short e;
struct {
  char f;
  int g[1];
  char h;
} i;
```

- *Problem*: Assume data memory starts at address 100 and no reordering, calculate the total amount of memory needed

a = 200 bytes (100-299)

b = 1 byte (300-300)

c = 4 bytes (304-307)

d = 8 bytes (312-319)

e = 2 bytes (320-321)

struct: largest field is 4 bytes, start at 324

f = 1 byte (324-324)

g = 4 bytes (328-331)

h = 1 byte (332-332)

i = 12 bytes (324-335)

236 bytes total!! (compared to 221, originally)

Data Layout – Why?

- Does gcc (or another compiler) reorder variables in memory to avoid padding?

- Only outside structs
- C99 forbids reordering elements inside a struct

- The programmer (i.e., you) are expected to manage data layout of variables for your program and structs.

- Two optimal strategies:
  - Order fields in struct by datatype size, smallest first
  - Or by largest first

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

Agenda

- Using branches more generally
- **Function calls and the call stack**
- Assigning variables to memory locations
- Saving registers
- Caller/callee example

Implementing Functions

Poll: What's wrong with this?

- Does this assembly code do what we need?

```c
int mult_2(int x) {
 int temp = x*2;
 return temp;
}

int GLOBAL = 6;

int main() {
 int result = mult_2(GLOBAL+1);
 printf(result);
}
```

```
        LDURSW X1, [XZR, GLOBAL]
        ADD    X2, X1, #1          // Inc GLOBAL
        STURW  X2, [XZR, X]        // Pass arg
        B      MULT_2              // Execute func
RETURN:
        LDURSW X3, [XZR, TEMP]     // load result
        STURW  X3, [XZR, STRING]   // Pass arg
        B PRINTF                   // Execute func
…
MULT_2:
        LDURSW X1, [XZR, X]         // load arg
        ADD    X2, X1, X1          // mult by 2
        STURW  X2, [XZR, TEMP]     // return result
        B RETURN                   // return
```

Problem 1: Returning from Functions

- Branches so far have hard-coded destination

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

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

```
int func(int x) {
  printf(x * 10);
  return;
}
int helper() {
  func(7);
}
int main() {
  helper();
  func(13);
}
```

Should this return to "helper" or "main"?

- This is fine for if-statements, for-loops etc
- But functions can be called from multiple places
  - Meaning we'll return to different spots on each func call! Can't hardcode offset!

Solution: Indirect Jumps

- Indirect branches or "jumps" don't hardcode destination in instruction
- They index a register whose value holds destination

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

- Use "BL" to **call a function**
  - Destination is hardcoded
  - PC +4 (return address) stored in X30
- Use "BR" to **return from a function**
  - X30 is read for return address
  - Allows us to return to different places

Solution: Indirect Jumps

```
int mult_2(int x) {
  int temp = x*2;
  return temp;
}

int GLOBAL = 6;

int main() {
  int result = mult_2(GLOBAL+1);
  printf(result);
}
```

Also don't need "return" labels

```
        LDURSW X1, [XZR, GLOBAL]
        ADD    X2, X1, #1          // Inc GLOBAL
        STURW  X2, [XZR, X]        // Pass arg
        BL     MULT_2              // Execute func
~~RETURN:~~
        LDURSW X3, [XZR, TEMP]     // load result
        STURW  X3, [XZR, STRING]   // Pass arg
        BL PRINTF                  // Execute func
…
MULT_2:
        LDURSW X1, [XZR, X]        // load arg
        ADD    X2, X1, X1          // mult by 2
        STURW  X2, [XZR, TEMP]     // return result
        BR                         // return
```

Now MULT_2 can return to whatever function called it

Problem 2: Passing Parameters

For any recursive functions, global variables will be overwritten

```c
int mult_2(int x) {
  int temp = x*2;
  return temp;
}

int GLOBAL = 6;

int main() {
  int result = mult_2(GLOBAL+1);
  printf(result);
}
```

```
        LDURSW X1, [XZR, GLOBAL]
        ADD    X2, X1, #1          // Inc GLOBAL
        STURW  X2, [XZR, X]        // Pass arg
        BL     MULT_2              // Execute func
        LDURSW X3, [XZR, TEMP]     // load result
        STURW  X3, [XZR, STRING]   // Pass arg
        BL PRINTF                  // Execute func
…
MULT_2:
        LDURSW X1, [XZR, X]         // load arg
        ADD    X2, X1, X1          // mult by 2
        STURW  X2, [XZR, TEMP]     // return result
        BR                         // return
```

How to pass in parameters?

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();
}
```

inside foo

foo's stack frame

foo calls bar

foo's stack frame
bar's stack frame

bar calls printf

foo's stack frame
bar's stack frame
printf's stack frame

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

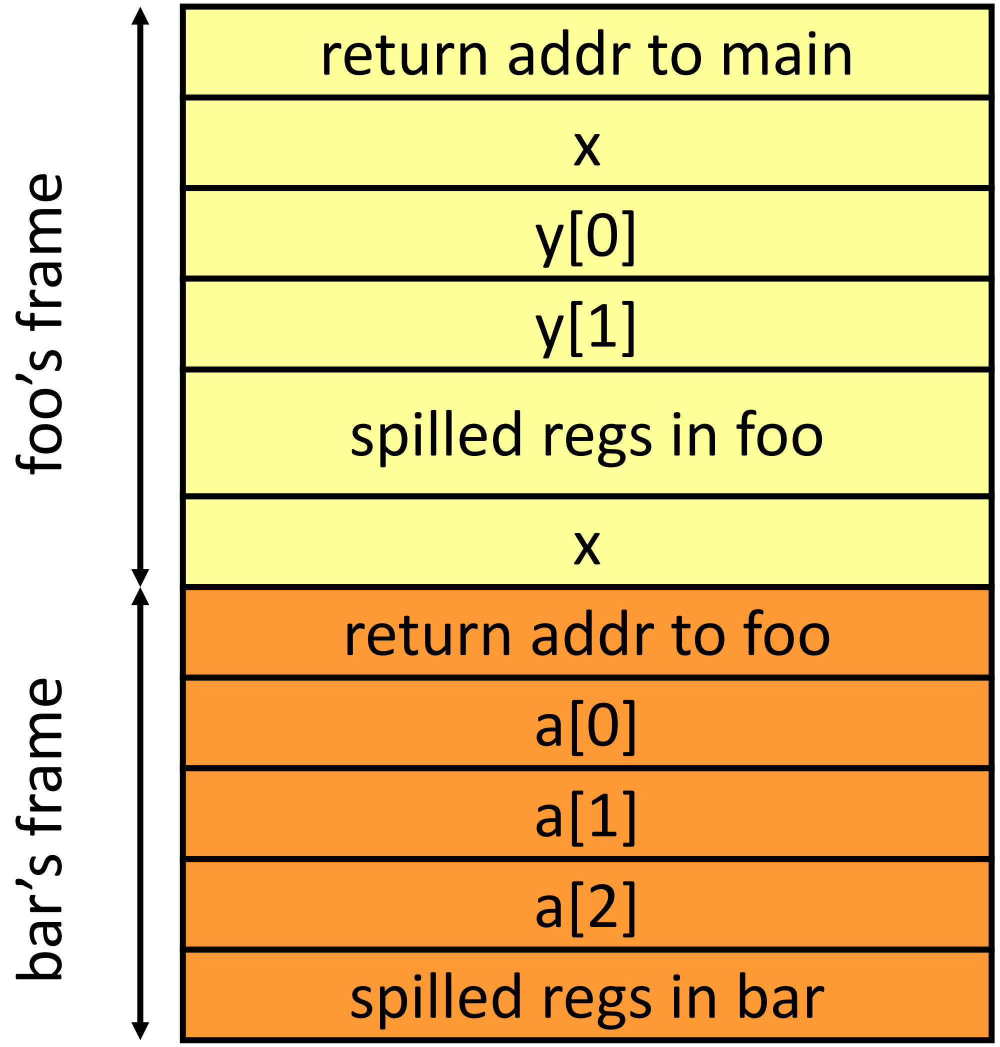
return addr to main
x
y[0]
y[1]
spilled registers in foo

Stack frame contents (2)


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

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

foo calls bar



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

Agenda

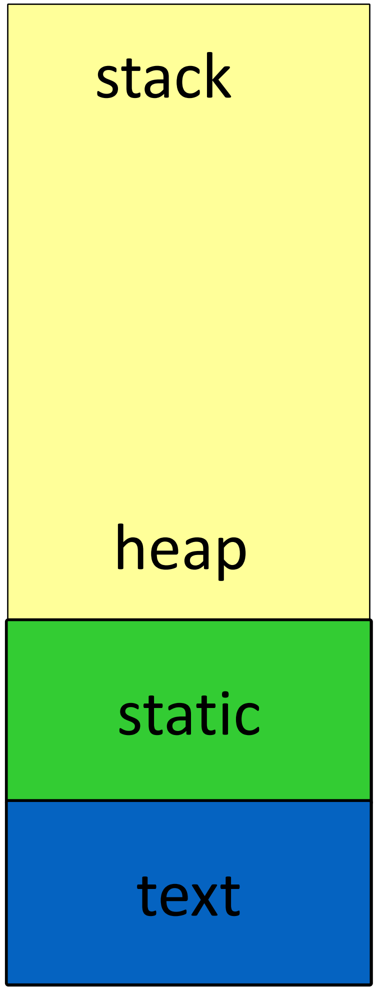
- Using branches more generally
- Function calls and the call stack
- **Assigning variables to memory locations**
- Saving registers
- Caller/callee example

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

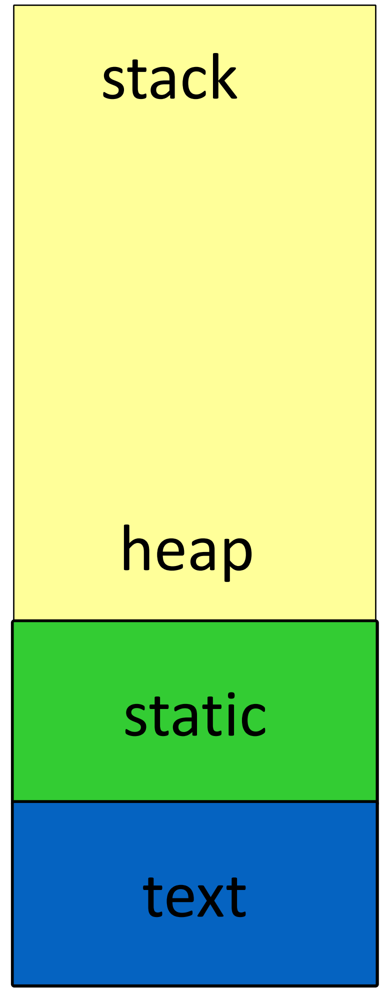
y goes in static, 1 copy of this!!

p goes on the stack

allocate 10 bytes on heap, ptr set to the address

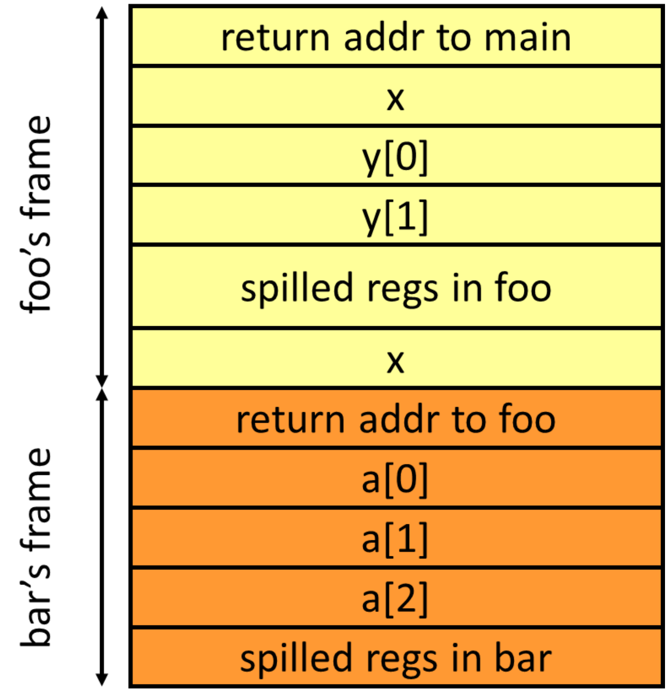
string literal "%s\n" goes in static, implicit pointer to string on stack, p goes on stack

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



Accessing Local Variables

- Stack pointer (SP):
  - register that keeps track of current top of stack
- Compiler (or assembly writer) knows relative offsets of objects in stack
- Can access using lw/sw offsets



Agenda

- Using branches more generally
- Function calls and the call stack
- Assigning variables to memory locations
- **Saving registers**
- Caller/callee example

Problem 3: Reusing 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!

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

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

```
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

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

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

```
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 r4
  restore r1
  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 r4
  restore r3
  restore r2
  restore r1
  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

- Callee-save
  - What if main() 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();
}
```

```
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();
}
```

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();
}
```

Agenda

- Using branches more generally
- Function calls and the call stack
- Assigning variables to memory locations
- Saving registers
- **Caller/callee example**

Caller versus Callee

- Which is better??
- Let's look at some examples…

- Simplifying 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

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

  c = 5; d = 6;
  a = 2; b = 3;
  foo();
  d = a+b+c+d;

}
```

```c
void foo(){
  int e,f;


  e = 2; f = 3;
  bar();
  e = e + f;

}
```

```c
void bar(){
  int g,h,i,j;

  g = 0; h = 1;
  i = 2; j = 3;
  final();
  j = g+h+i;

}
```

```c
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. How many registers need to be saved/restored if we use a **caller-save** convention?
2. How many registers need to be saved/restored if we use a **callee-save** convention?
3. How many registers need to be saved/restored if we use a mix of **caller-save** and **callee-save**?

Poll: How many ld/st pairs are needed?

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

Poll: How many ld/st pairs are needed?

```
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

Is one better?

- Caller-save works best when we don't have many live values across function call
- Callee-save works best when we don't use many registers overall
- We probably see functions of both kinds across an entire program
- Solution:
  - Use both!
  - E.g. if we have 6 registers, use some (say r0-r2) as caller-save and others (say r3-r5) as callee-save
  - Now each function can optimize for each situation to reduce saving/restoring
  - Not discussed further in this class

LEGv8 ABI- Application Binary Interface

- The ABI is an agreement about how to use the various registers
- Not enforced by hardware, just a convention by programmers / compilers
- If you won't your code to work with other functions / libraries, **follow these**
- Some register conventions in ARMv8
  - X30 is the **link register** – used to hold return address
  - X28 is **stack pointer** – holds address of top of stack
  - X19-X27 are **callee-saved** – function must save these before writing to them
  - X0-15 are **caller-saved** –function must save live values before call
  - X0-X7 used for **arguments** (memory used if more space is needed)
  - X0 used for **return value**

Next Time

- Finish Up Function Calls
- Talks about linking – the final puzzle piece of software