EECS 370 - Lecture 6

Function Calls

Announcements

- Project 1a due tonight
- Project 1s+m due next Thursday
- HW 1 Posted
  - Due Monday 9/22
- Let us know about exam conflicts in the **next week**
  - Form on Ed

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

Reminder: Memory Alignment

```
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 = 2 bytes * 100 = 200

b = 1 byte

c = 4 bytes

d = 8 bytes

e = 2 bytes

i = 1 + 4 + 1 = 6 bytes

total = 221, right or wrong?

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

Reminder: Golden Rule of Alignment

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

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

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…

Structure Alignment

- In addition, for structs…
  - Identify largest (potentially nested) **primitive** component
    - Starting address of overall struct is aligned based on the largest component
    - Padded in the back so total size is a multiple of the largest component

```
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 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

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

- Let's look at the first type: CBZ and CBNZ
  - CBZ: Conditional Branch if Zero
  - CBNZ: Conditional Branch if Not Zero

LEGv8 Conditional Instructions

- CBZ/CBNZ: test a register against zero and branch to a PC relative address
  - The relative address is a 19 bit signed integer representing the number of instructions (not bytes) to branch. Recall instructions are 4 bytes

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

- Example: CBNZ X3, Again
  - If X3 doesn't equal 0, then branch to label "Again"
  - "Again" is an offset from the PC of the current instruction (CBNZ)
  - Why does "25" in the above table result in PC + 100?1

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	Instruction	Example	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--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†	Always	Any

Need to know the 7 with the red arrows

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

For this example, we branch if X1 is >= to X2

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

- Here's how we can convert an if/else statement to ARM (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 800, 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, #800]
          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

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 approach?

- 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);
 other(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 OTHER                      // 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
```

- 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!

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

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

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);
  other(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 OTHER                    // 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 recursive functions, global variables could 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);
  other(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 OTHER                     // 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?
    - Where to put it? And, memory is 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

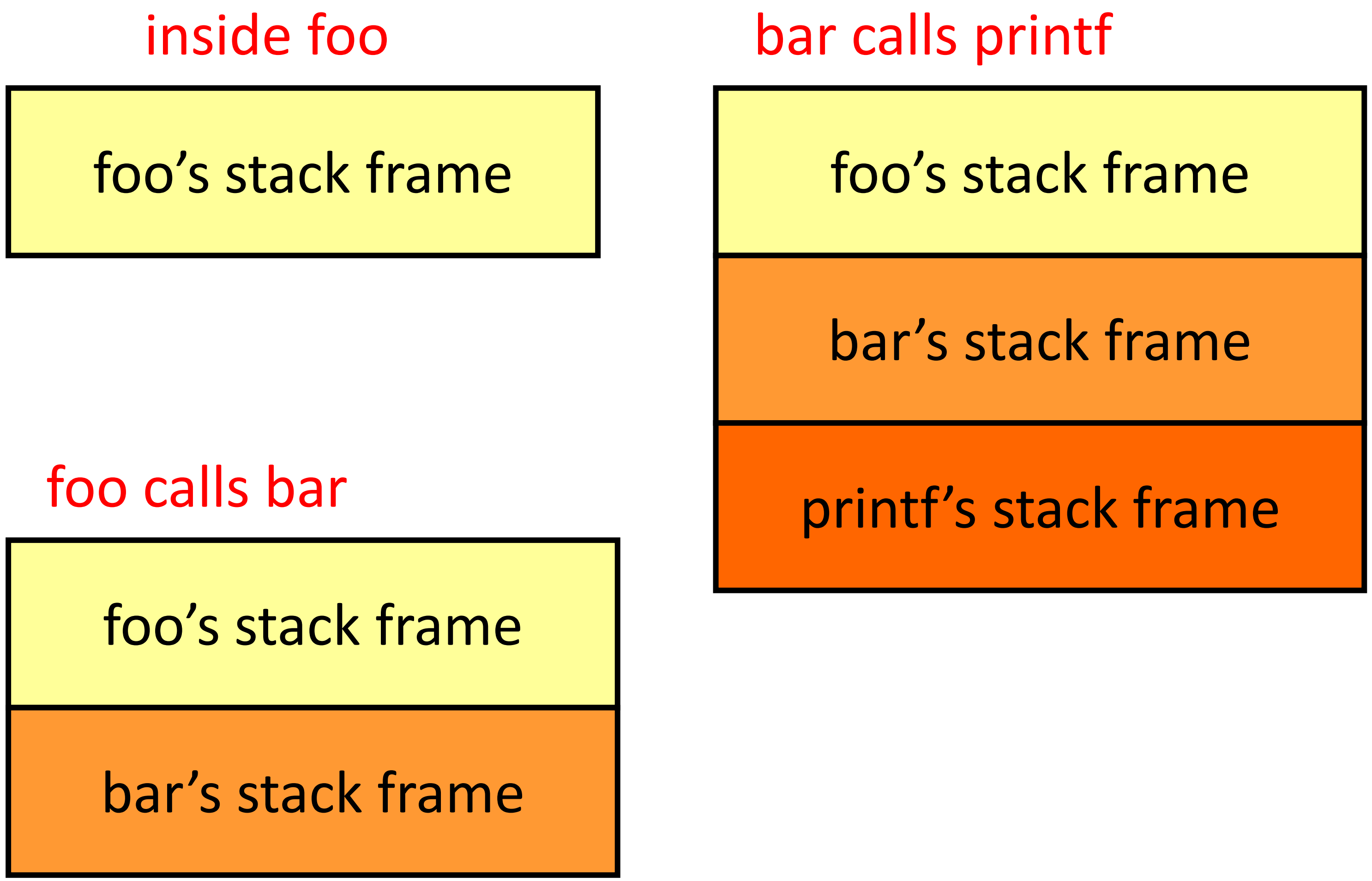
- ARM conventions (and most other architectures) allocate a region of memory for the “call” stack
  - This memory is used to manage all the storage requirements to simulate function call semantics
    - Parameters (that were not passed through registers)
    - Local variables
    - Temporary storage (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
- Grows “down” to lower addresses, usually (but not for LC2K projects)

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

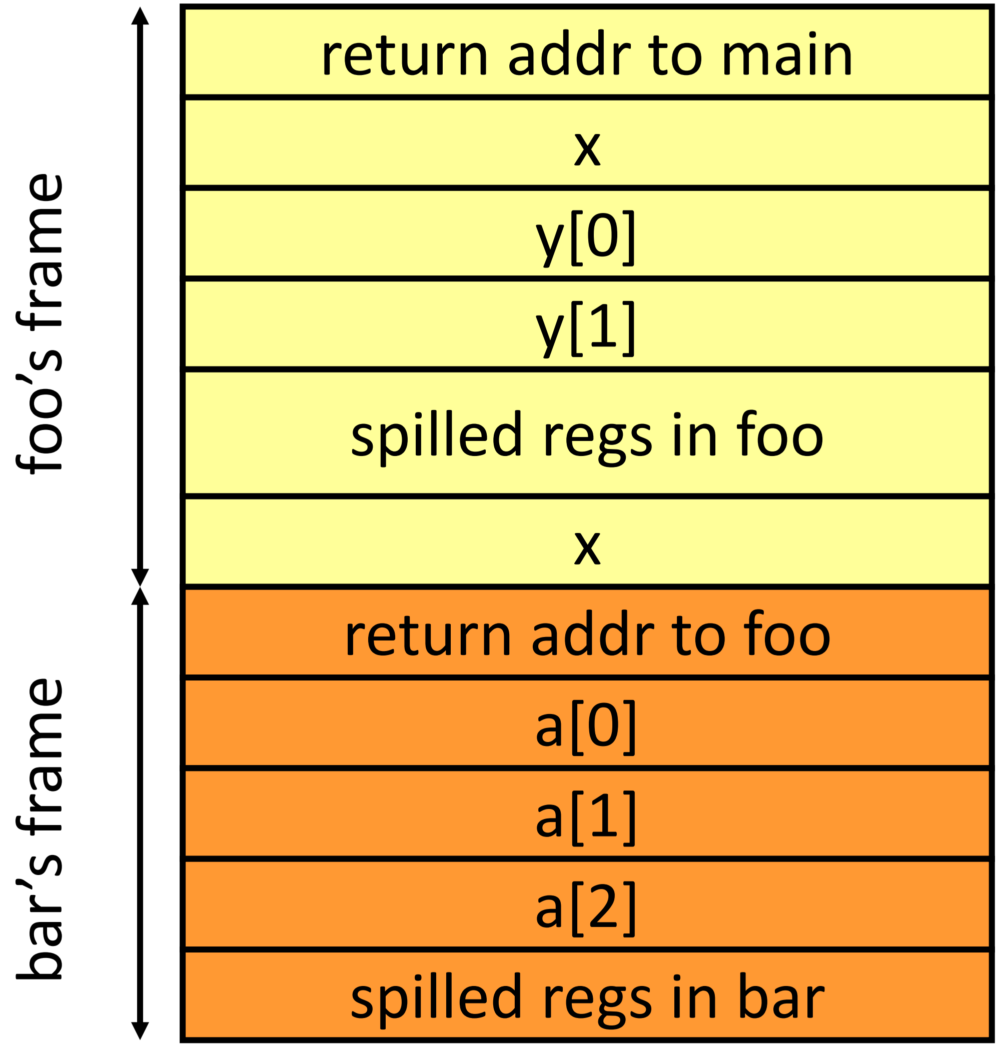
Stack frame contents (2)

FUNCTION CALLS

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

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

foo calls bar



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

Agenda

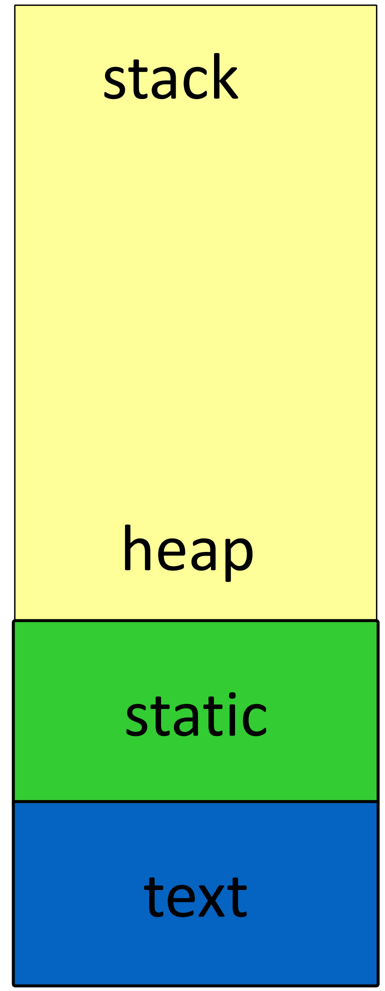
- Using branches more generally
- Function calls and the call stack
- **Assigning variables to memory locations**
- Saving registers
- Caller/callee example

Review: Where do the variables go?

Assigning variables to memory spaces

FUNCTION CALLS

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



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);
  free(p);
}
```

w goes in static, as it's a global

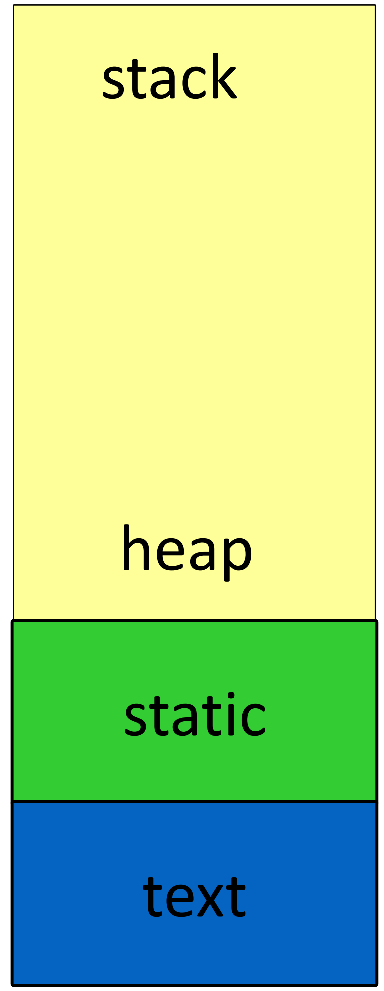
x goes on the stack, as it's a parameter

y goes in static, 1 copy of this!!

p goes on the stack

allocate 10 bytes on heap, ptr set to the address

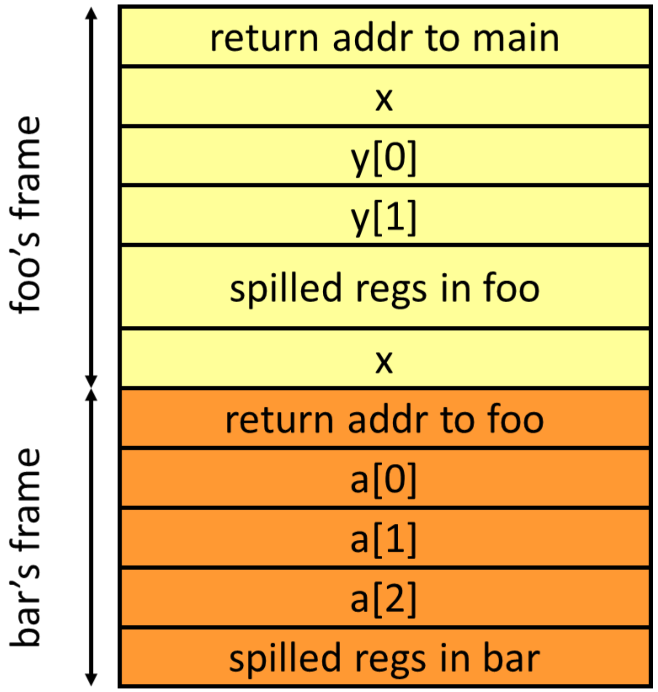
string literal "%s\n" goes in static, implicit pointer to string on stack, p goes on stack



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

Accessing Local Variables

- Stack pointer (SP):
  - register intended keeps track of current top of stack
  - Aliases to “X28” in LEGv8
- Compiler (or assembly writer) knows relative offsets of objects in stack
- Can modify SP or use load/store offsets
- **<u>DOESN’T USE LABELS!</u>**




```
sub     sp, sp, #16          // make room on stack

mov     x0, #42              // put 42 in x0
stur    x0, [sp, 0]          // store local var on stack

ldur    x1, [sp, 0]          // load back into x1
```

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
```

Next Time

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

Extra problems

Extra Example: Do-while Loop

// 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 Example: Do-while Loop

// 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];
  }
}
```

# 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:

```
// 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 : ….
```

Extra Class Problem

- How much memory is required for the following data, assuming that the data starts at address 200 and is a 32 bit address space?

```
int a;
struct {double b, char c, int d} e;
char* f;
short g[20];
```

<u>Poll:</u> How much memory?

a) x < 40 bytes
b) 40 < x < 50 bytes
c) 50 < x < 60 bytes
d) 60 < x bytes