# EECS 370 - Lecture 6

## Function Calls

# Announcements

- Project 1
  - P1a due Thursday **1/29**
  - P1s due Thursday **2/5**
  - P1m due Thursday **2/5**
- HW1
  - Posted, due Monday **2/2**
- Lab
  - **Lab quiz due TODAY 1/22, 11:55pm**
  - Lab meets this upcoming Friday/Monday/Tuesday
- Midterm:
  - Thursday March 12th, 7-9pm
  - *Exam conflict form is due by January 23rd via the Exam Conflict form* (ed post #17)

**Lecture 3 covered what you need.**

**Each problem is labelled by what lecture you need.**

# Review of last lecture

- Memory alignment
  - **Aligning Structs**
- Control flow instructions
  - C-code examples

# 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

# Agenda

- Memory alignment
  - Aligning Structs
- **Control flow instructions**
  - C-code examples

# 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—the number of instructions. Recall instructions are 32 bits of 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?

# 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

```
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.

# Agenda

- Memory alignment
  - Aligning Structs
- Control flow instructions
  - **C-code examples**
- Extra Problems

# 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

Note: Could further optimize by counting down (9 to 0) and using CNBZ in place of the CMPI and B.LT at the end!

# 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

# 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

- **Branching far away**
- Function calls and the call stack
- Assigning variables to memory locations
- Saving registers
- Caller/callee example

# 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

# Agenda

- Branching far away
- **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, RETURN]   // 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
```

- 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!

```
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 register
- 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, RETURN]    // 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, RETURN]   // 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

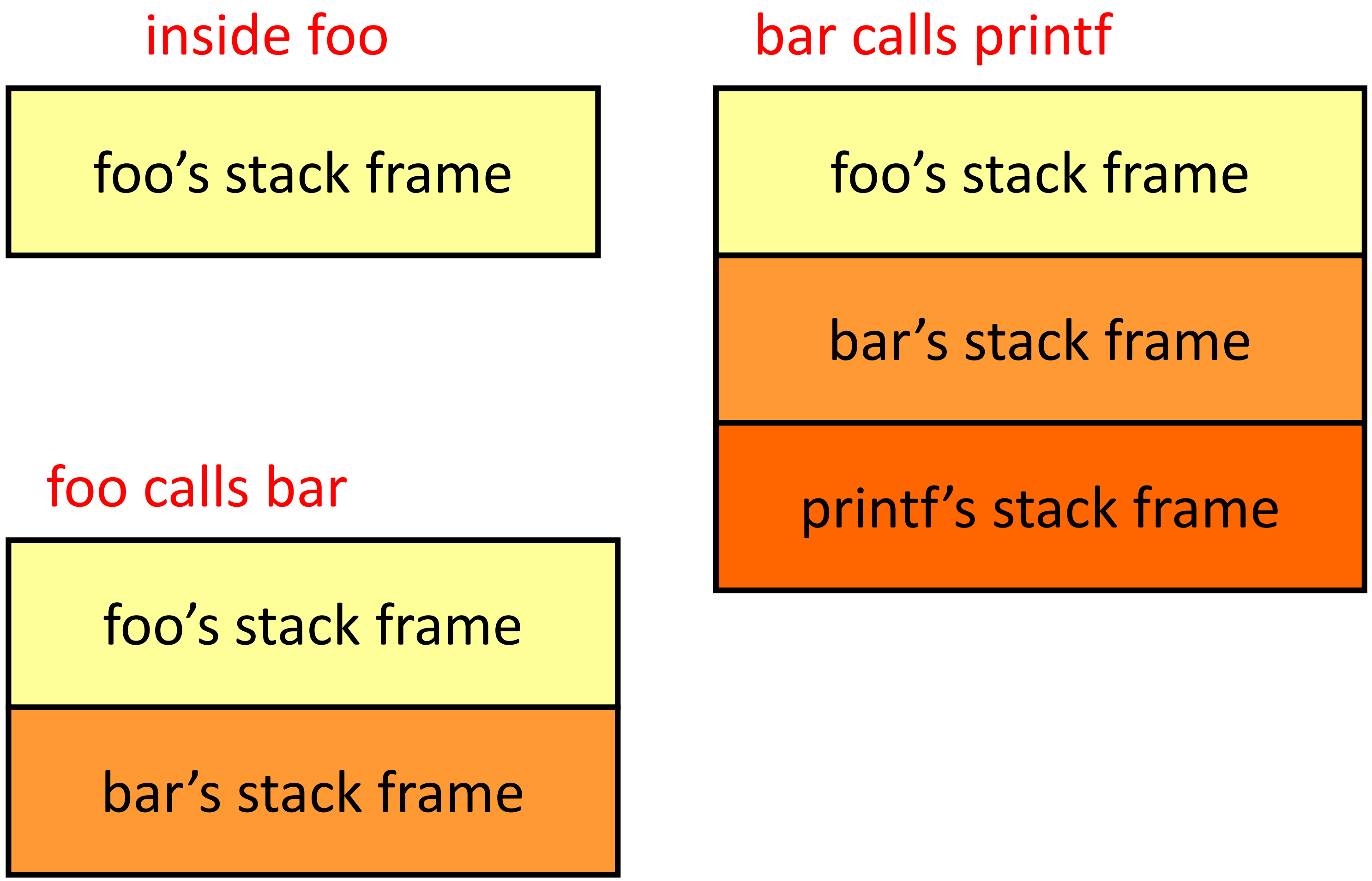
- ARM conventions (and most other processors) allocate a region of memory for the “call” stack
  - This memory is used to manage all the storage requirements to simulate function call semantics
    - Parameters (that were not passed through registers)
    - Local variables
    - Temporary storage (when 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

```c
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



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

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

foo's stack frame

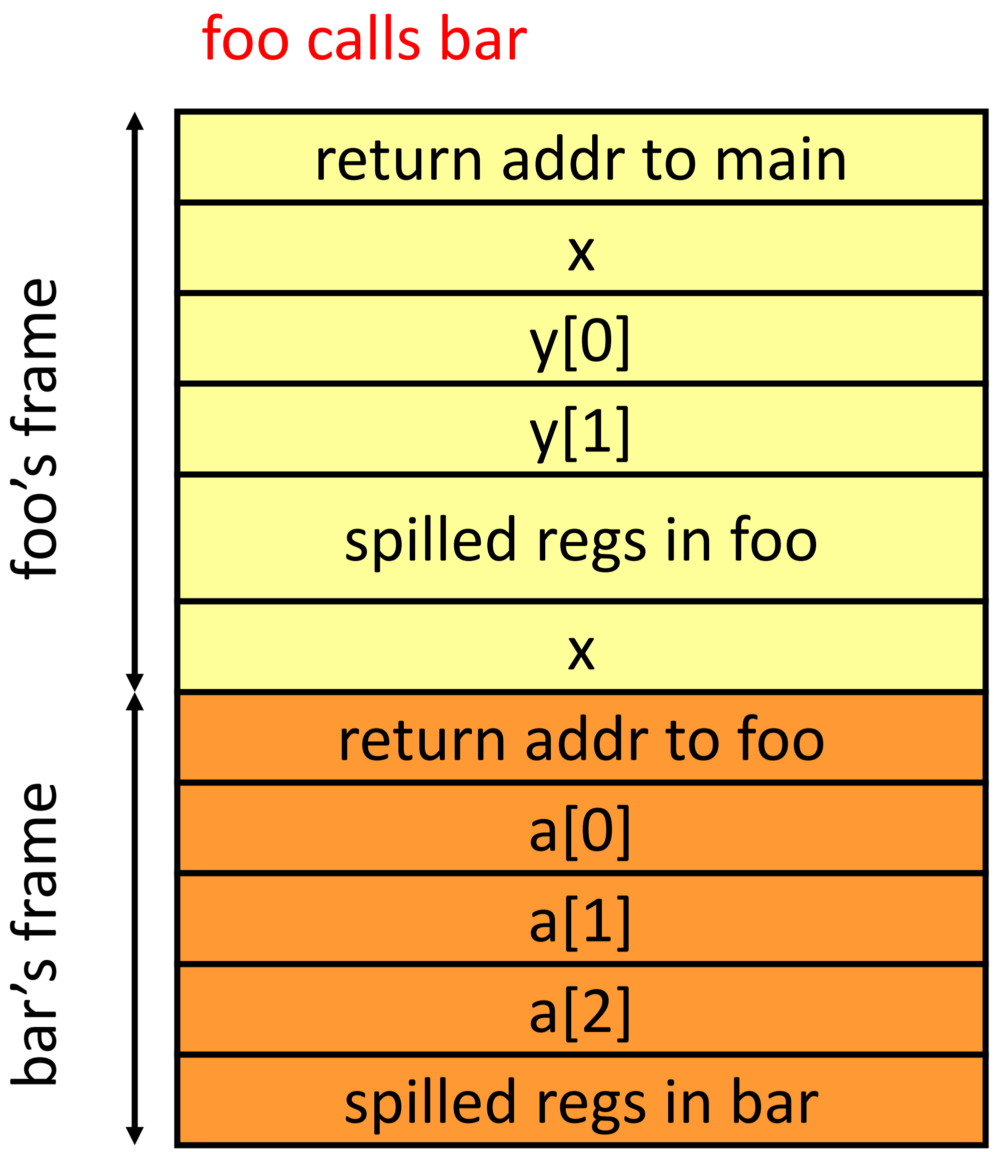
| |
|---|
| return addr to main |
| x |
| y[0] |
| y[1] |
| spilled registers in foo |

# 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

# Agenda

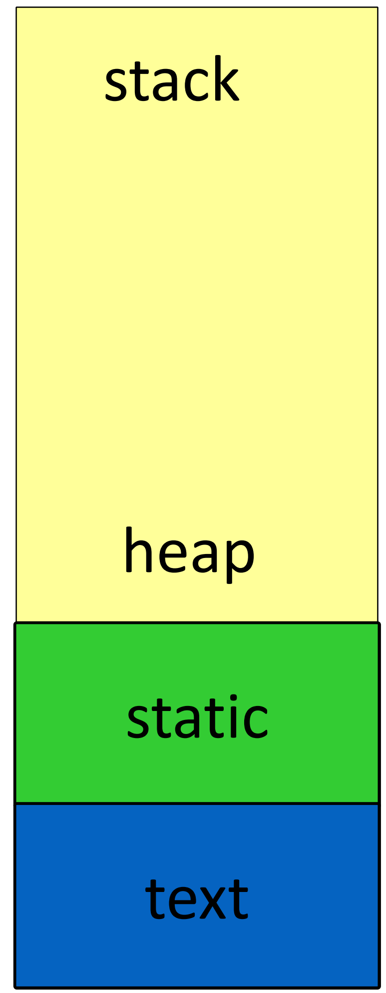
- Branching far away
- Function calls and the call stack
- **Assigning variables to memory locations**
- Saving registers
- Caller/callee example

**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

```
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

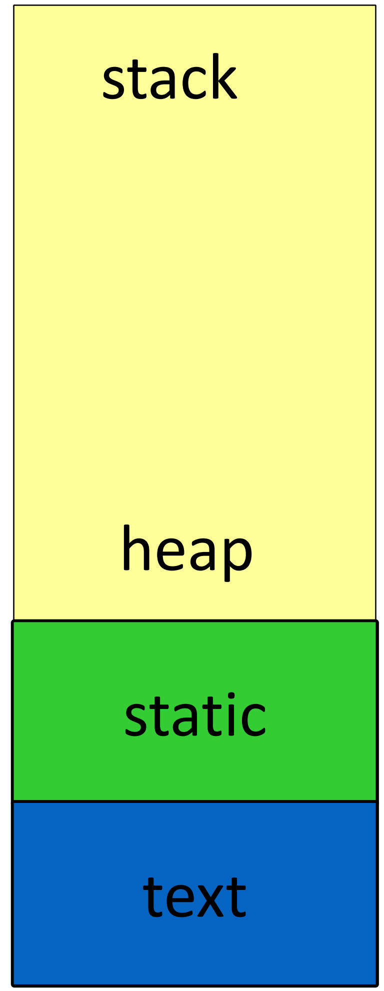
x goes on the stack, as it's a parameter

y goes in static, 1 copy of this!!

p goes on the stack

allocate 10 bytes on heap, ptr set to the address

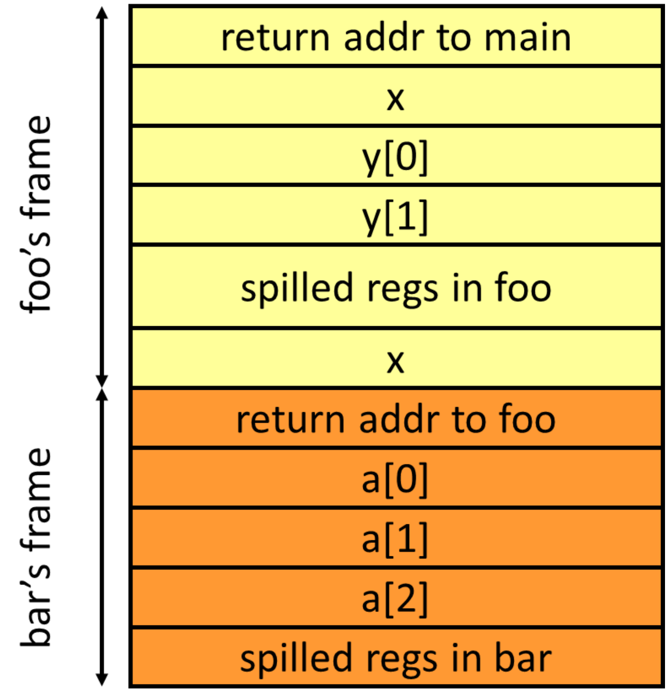
string literal "%s\n" goes in static, implicit pointer to string on stack, p goes on stack




**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

- Branching far away
- 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

```
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 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

```
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();
}
```

- Callee-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();
}
```

# 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

- Branching far away
- 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

```
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. 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 want 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