5. 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 next Thursday!**
- Project 1.s and 1.m due Thursday 2/2
- Homework 1 due Monday 1/23
  - 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

DATA LAYOUT

Converting C to assembly – Example

Write ARM assembly code for the following C expression:

```
struct { int a; unsigned char b, c; } y;
y.a  = y.b + y.c;
```

Assume that a pointer to y is in X1.

```
LDURB   X2,  [X1, #4]   // load y.b
LDURB   X3,  [X1, #5]   // load y.c
ADD     X4,  X2,  X3    // calculate y.b+y.c
STURW   X4,  [X1, #0]   // store y.a
```

How do you determine the offsets for the struct sub-fields?

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

Calculating Load/Store Addresses for Variables

```
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, 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?

Memory layout of variables

- Most modern ISAs require that data be aligned.
- What do we mean by alignment in this context?
  - An N-byte variable must start at an address A, such that (A % N) == 0
- "Golden" rule – Address of a variable is aligned based on the size of the variable
  - **char** is byte aligned (any address is fine)
  - **short** is half-word (H) aligned (LSBit of address must be 0)
  - **int** is word aligned (W) (2 LSBit's of addr must be 0)

Structure alignment

- Each field is laid out in the order it is declared using the Golden Rule for alignment

- Identify largest field
  - Starting address of overall struct is aligned based on the largest field
  - Size of overall struct is a multiple of the largest field
  - Reason for this is so we can have an array of structs

Structure Example

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

The largest field is **int** (4 bytes), hence:

➡ struct size is multiple of 4

➡ struct's starting addr is word aligned

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!

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

Earlier Example – 2nd Try

```
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, 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

Class Problem

- How much memory is required for the following data, assuming that the data starts at address 200?
  - Note: pointers are *generally*the same size as ints.

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

Data Layout – Why?

- Does gcc (or another compiler) reorder variables in memory to avoid padding?
- No, C99 forbids this
  - Memory is laid out in order of declaration for structs
- 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

BRANCH

ARM/LEGv8 Sequencing Instructions

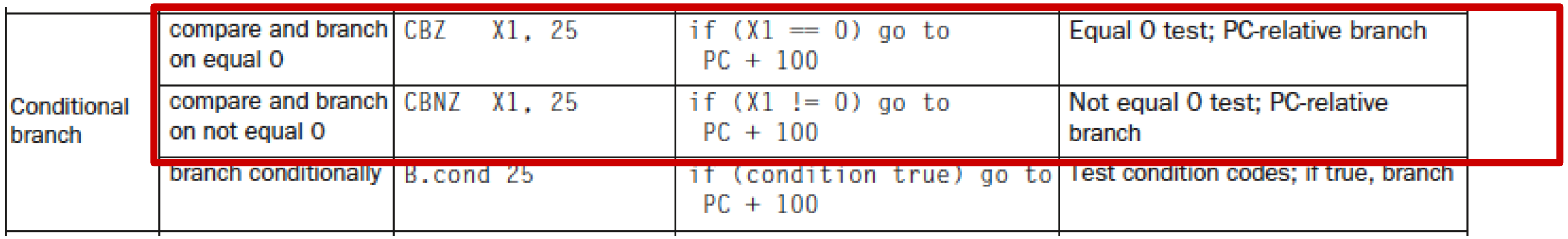
- Sequencing instructions change the flow of instructions that are executed
  - This is achieved by modifying the program counter (PC)
- Unconditional branches are the most straightforward they ALWAYS change the PC and thus “jump” to another instruction out of the usual sequence
- Conditional branches

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 word 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.



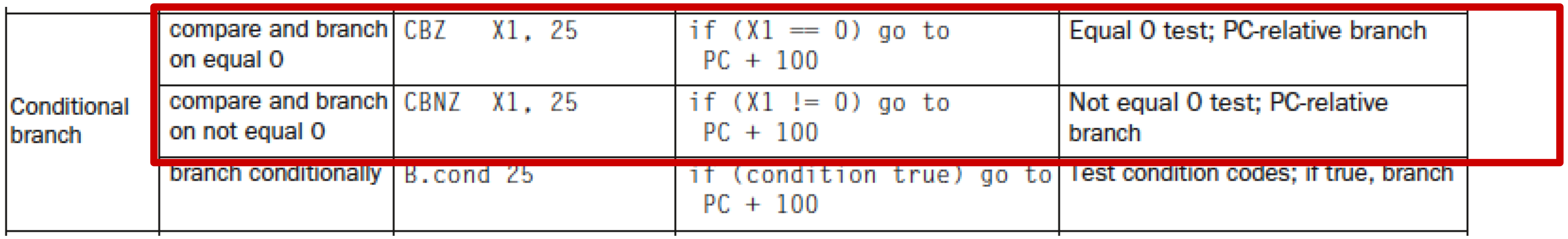
Conditional branch	compare and branch on equal 0	`CBZ X1, 25`	`if (X1 == 0) go to PC + 100`	Equal 0 test; PC-relative branch
	compare and branch on not equal 0	`CBNZ X1, 25`	`if (X1 != 0) go to PC + 100`	Not equal 0 test; PC-relative branch
	branch conditionally	`B.cond 25`	`if (condition true) go to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

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

BRANCH

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?

BRANCH

LEGv8 Conditional Instructions

- Example: What would the offset or displacement be if there were two instructions between ADDI and CBNZ?

```
Again:          ADDI      X3, X3, #-1
               --------------
                --------------
                CBNZ      X3, Again
```

- Answer = -3
  - The offset field is 19 bits signed so the bit pattern would be 111 1111 1111 1111 1111 1101
  - Two zeroes are appended to the above 19 bits and then the result would be sign-extended (with one's) to 64 bits and added to the value of PC at CBNZ
  - Why the two zeroes?

BRANCH

LEGv8 Conditional Instructions

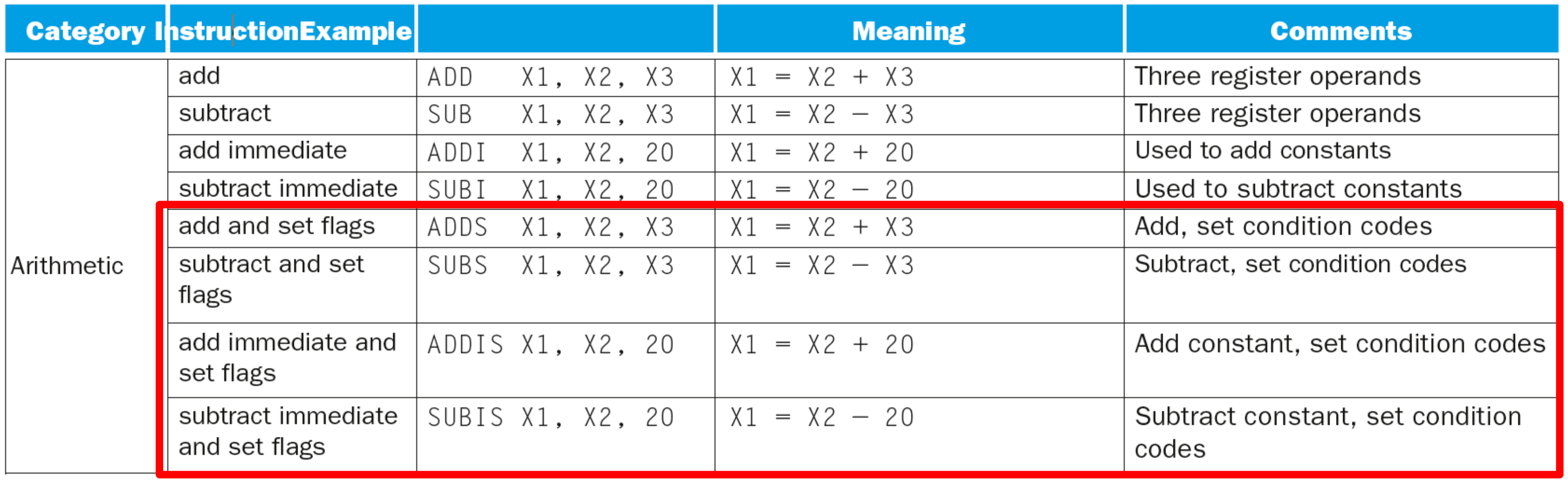
```
Again:    ADDI    X3, X3, #-1
          --------------
          --------------
          CBNZ    X3, Again
```

- The assembler figures out that the offset is -3
  - Less error prone than writing CBNZ X3, #-3
  - If we remove the instruction in red perhaps because we modified the program, the assembler will correct the offset to -2

ARM ISA

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

ARM ISA

- 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 bits for this class.

ARM Condition Codes Determine Direction of Branch--continued

ARM ISA

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 ADDS 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

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++;
(L1) else
         y++;
(L2) ...
```

Using Labels

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

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

BRANCH

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
        MOVZ    X4, #10
Loop1:  CMP     X1, X4
        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

BRANCH

Same Loop, Different Assembly

// assume all variables are long long integers (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
          MOVZ    X4, #10
Loop1:    LSL     X6, X1, #3
          LDUR    X5, [X6, #100]
          CMPI    X5, #0
          B.LT    endif
          ADD     X2, X2, X5
endIf:    ADDI    X1, X1, #1
          CMP     X1, X4
          B.LT    Loop1
endLoop:
```

# of branch instructions
= 2*10 = 20

a.k.a. do-while template

BRANCH

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++;
```

BRANCH

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++;
```

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

Branching far away

- The underlying philosophy of ISA design and microachitecture 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!

BRANCH

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

$\searrow$

Execute instructions for printf()
Jump to return address

$\swarrow$

- 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

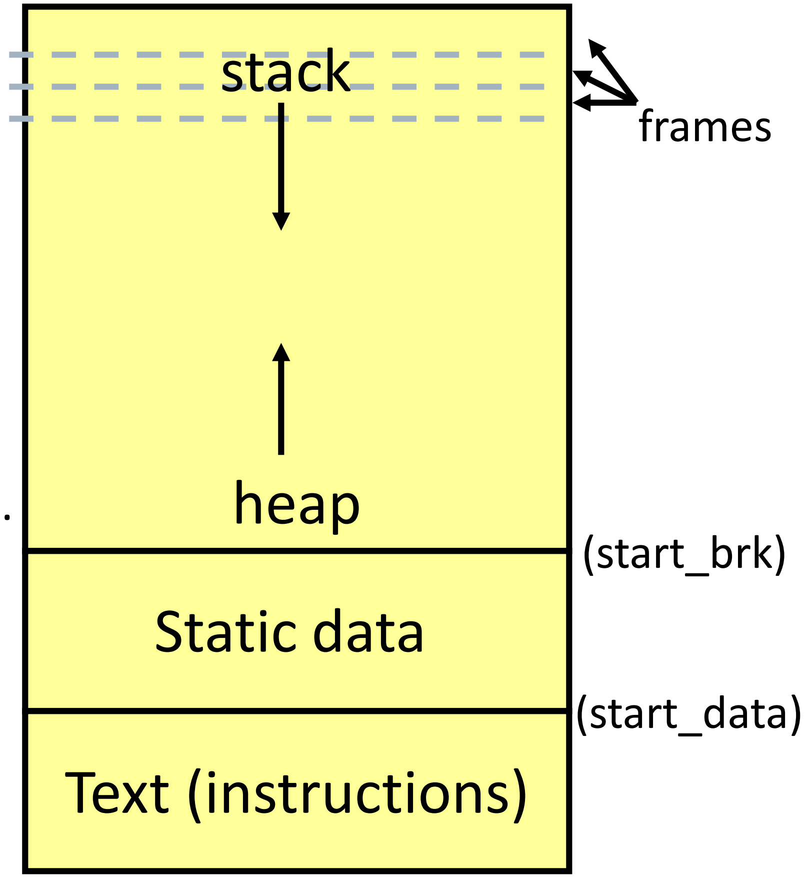
An Older ARM (Linux) Memory Map

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.

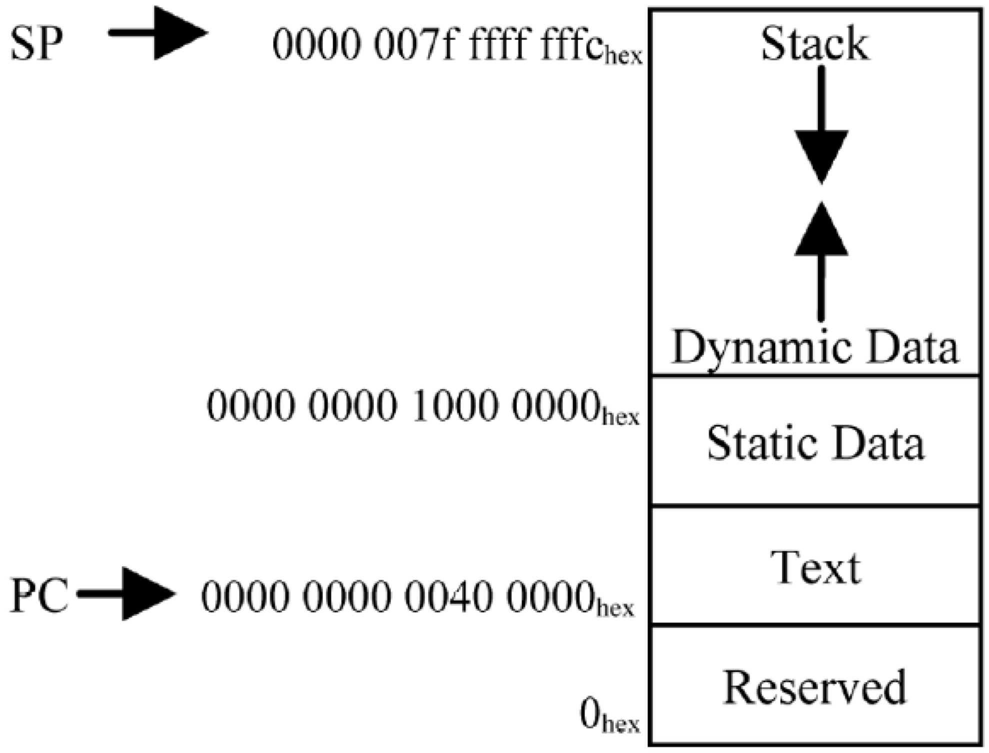
Static: starts above text. Holds all global variables and those locals explicitly declared as "static".

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


SP
0000 007f ffff fffc$_{hex}$
Stack
Dynamic Data
0000 0000 1000 0000$_{hex}$
Static Data
PC
0000 0000 0040 0000$_{hex}$
Text
0$_{hex}$
Reserved

FUNCTION CALLS

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

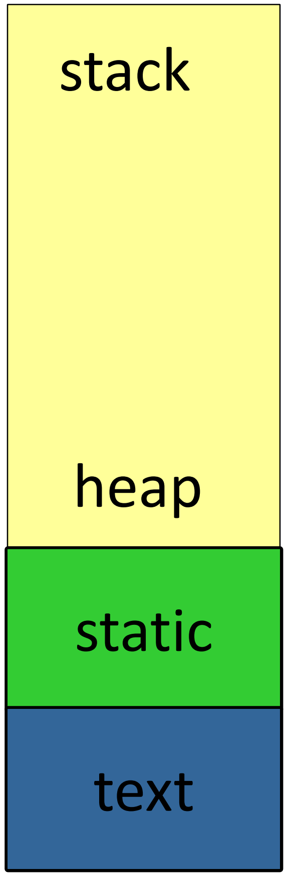
x goes on the stack, as it's a parameter

y goes in static, 1 copy of this!!

p goes on the stack

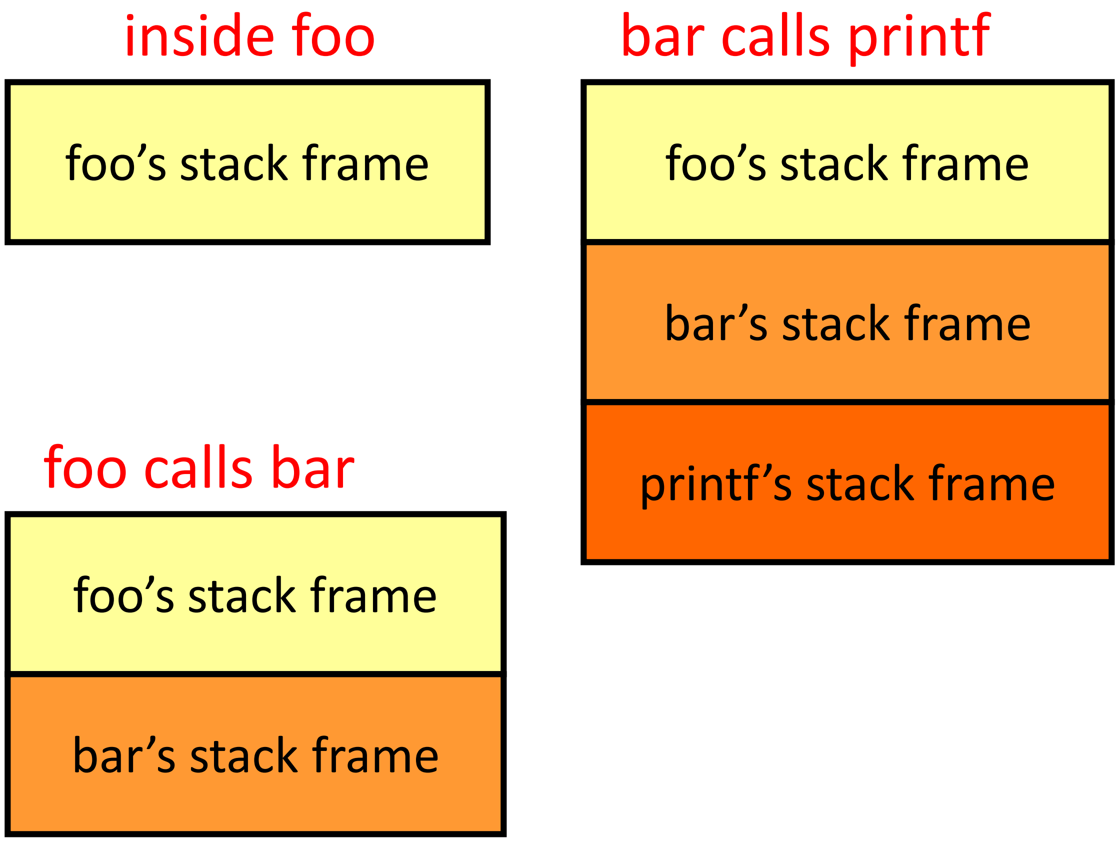
allocate 10 bytes on heap, ptr set to the address

string goes in static, pointer to string on stack, p goes on stack



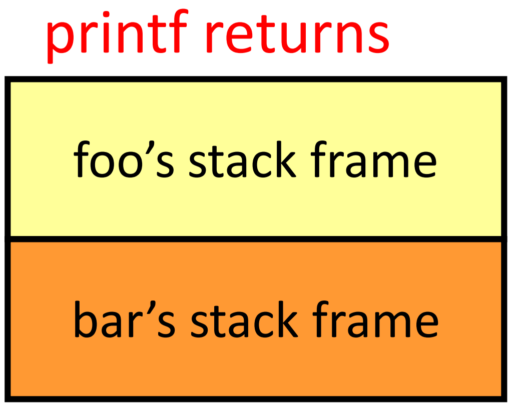
The stack grows as functions are called

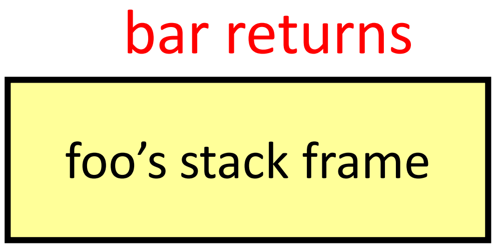
```
void foo()
{
    int x, y[2];
    bar(x);
}

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



The stack shrinks as functions return

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

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

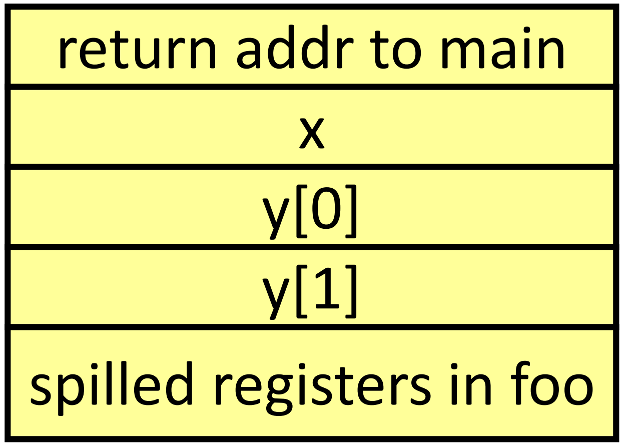




Stack frame contents

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

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

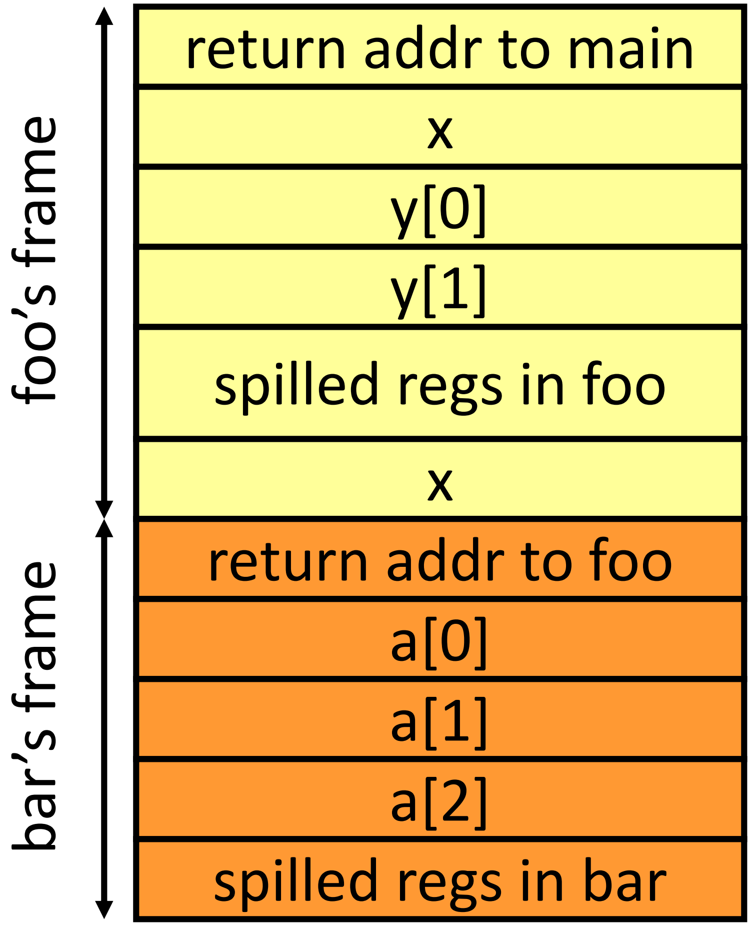
foo's stack frame



Stack frame contents (2)

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

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

foo calls bar



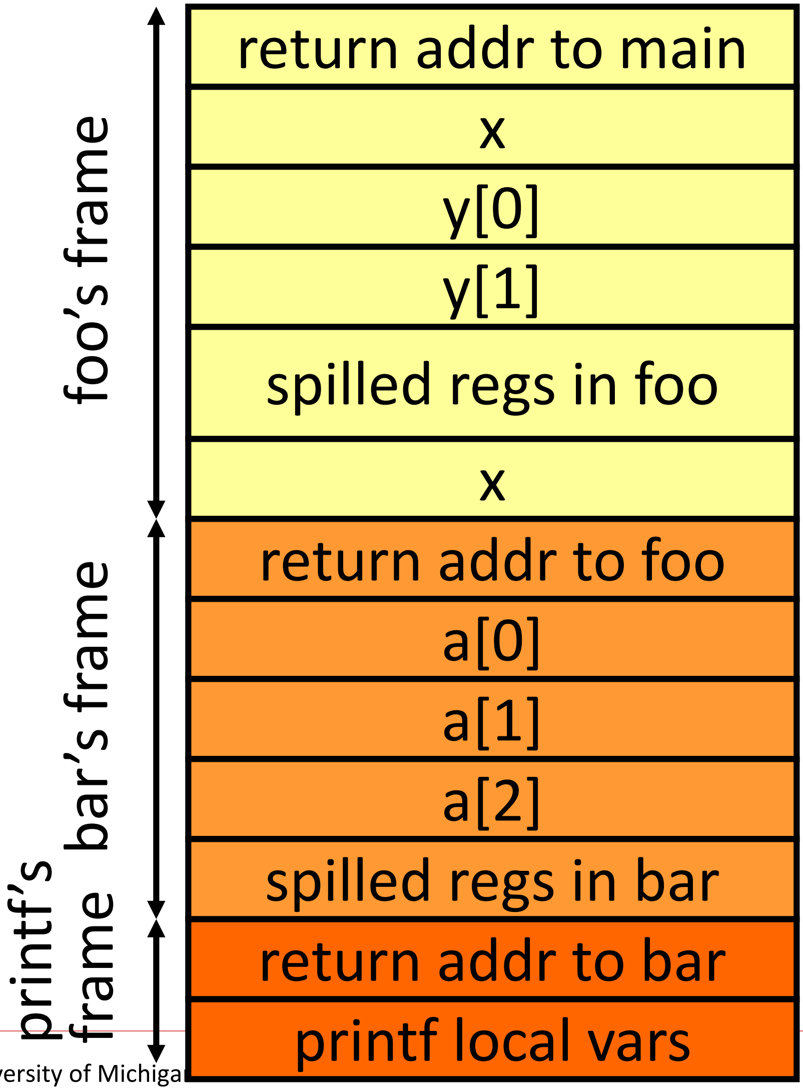
Spill data—not enough room in x0-x7 for params and also caller and callee saves

FUNCTION CALLS

Stack frame contents (3)

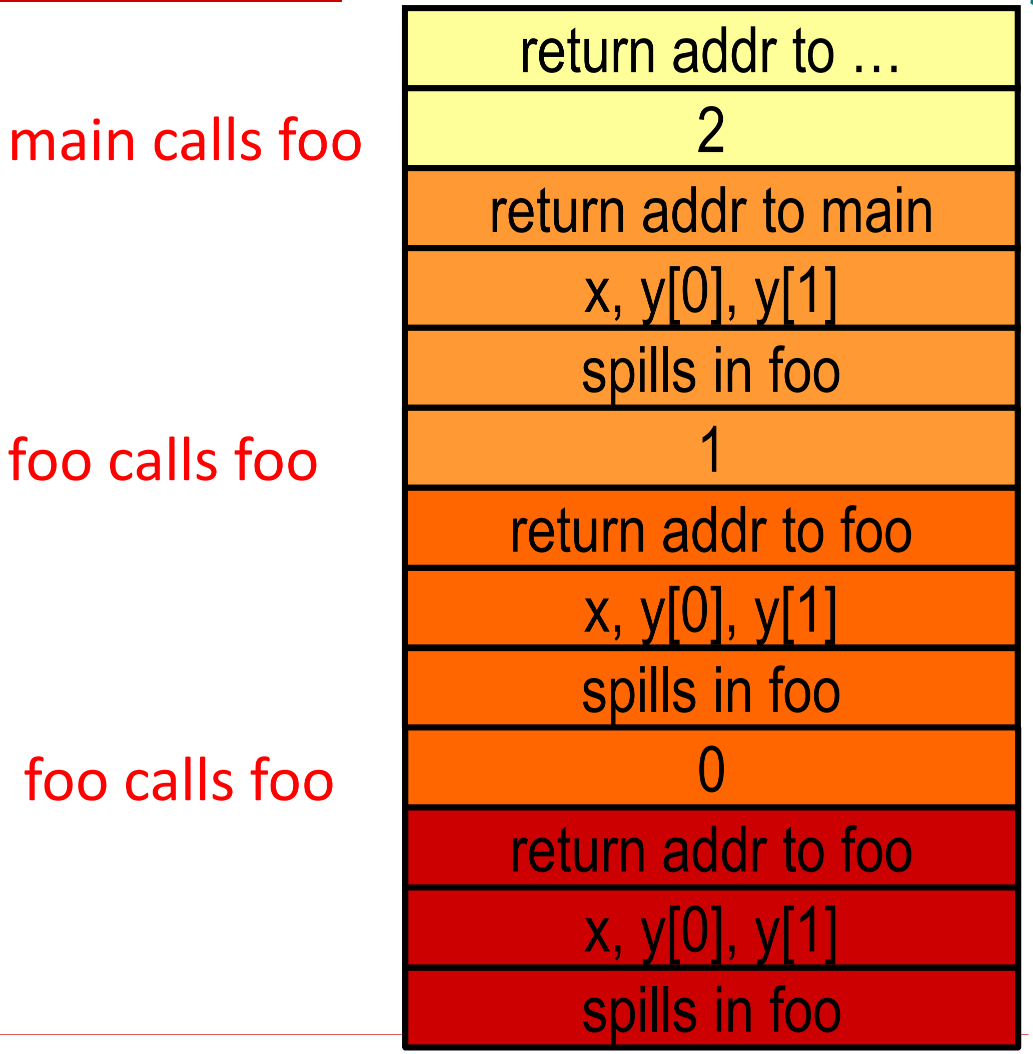
bar calls printf

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

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



Recursive function example

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

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



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”

CALLER-CALLEE

- 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.