

Intermediate Representation II

Storage Allocation and Management

EECS 483 – Lecture 18

University of Michigan

Wednesday, November 8, 2006

2 Classes of Storage in Processor

- ❖ Registers
 - » Fast access, but only a few of them
 - » Address space not visible to programmer
 - Doesn't support pointer access!
- ❖ Memory
 - » Slow access, but large
 - » Supports pointers
- ❖ Storage class for each variable generally determined when map HIR to LIR

Storage Class Selection

❖ Standard (simple) approach

- » Globals/statics – memory
- » Locals
 - Composite types (structs, arrays, etc.) – memory
 - Scalars
 - ◆ Accessed via ‘&’ operator? – memory
 - ◆ Rest – Virtual register, later we will map virtual registers to true machine registers. Note, as a result, some local scalars may be “spilled to memory”

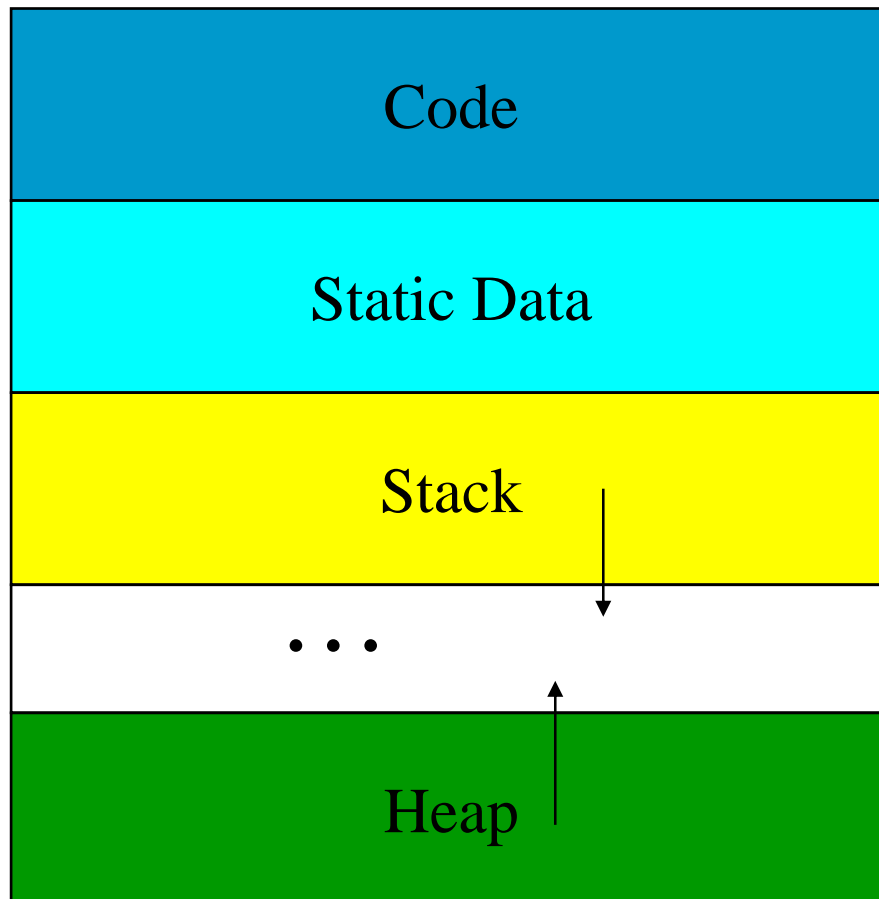
❖ All memory approach

- » Put all variables into memory
- » Register allocation relocates some mem vars to registers

4 Distinct Regions of Memory

- ❖ **Code space** – Instructions to be executed
 - » Best if read-only
- ❖ **Static (or Global)** – Variables that retain their value over the lifetime of the program
- ❖ **Stack** – Variables that is only as long as the block within which they are defined (local)
- ❖ **Heap** – Variables that are defined by calls to the system storage allocator (malloc, new)

Memory Organization



Code and static data sizes determined by the compiler

Stack and heap sizes vary at run-time

Stack grows downward

Heap grows upward

Some ABI's have stack/heap switched

Class Problem

Specify whether each variable is stored in register or memory.
For memory which area of the memory?

```
int a;
void foo(int b, double c)
{
    int d;
    struct { int e; char f;} g;
    int h[10];
    char i = 5;
    float j;
}
```

Variable Binding

❖ Definitions:

- » **Environment** – A function that maps a name to a storage location
- » **State** – A function that maps a storage location to a value

❖ When an environment associates a storage location S with a name N , we say **N is bound to S**

- » If N is a composite type, then N might be bound to a set of locations (usually contiguous, though not required)

Static Allocation

- ❖ Static storage has fixed allocation that is unchanged during program execution
- ❖ Used for:
 - » Global variables
 - » Constants
 - » All static variables in C have this (hence a global lifetime!)

```
int count (int n) {  
    static int sum = 0;  
    sum += n;  
}
```

sum has local visibility
coupled with a global
lifetime → lots of bugs!

Heap Allocation

- ❖ Parcels out pieces of continuous storage
- ❖ Pieces may be deallocated in any order
 - » Over time, heap will consist of alternate areas of free and in-use sections of memory
- ❖ Heap is global
 - » Items exist until explicitly freed
 - » Or if support garbage collection, until no-one points to the piece of memory
 - » Heap management is for the OS people to worry about! (e.g., first-fit, best-fit, ...)

Accessing Static/Heap Variables

❖ Static

- » Addresses are known to compiler
 - Assigned by linker
- » Compiler backend uses symbolic names (labels)
 - Same for branch addresses

❖ Heap

- » Are unnamed locations
- » Can be accessed only by dereferencing variables which hold their addresses

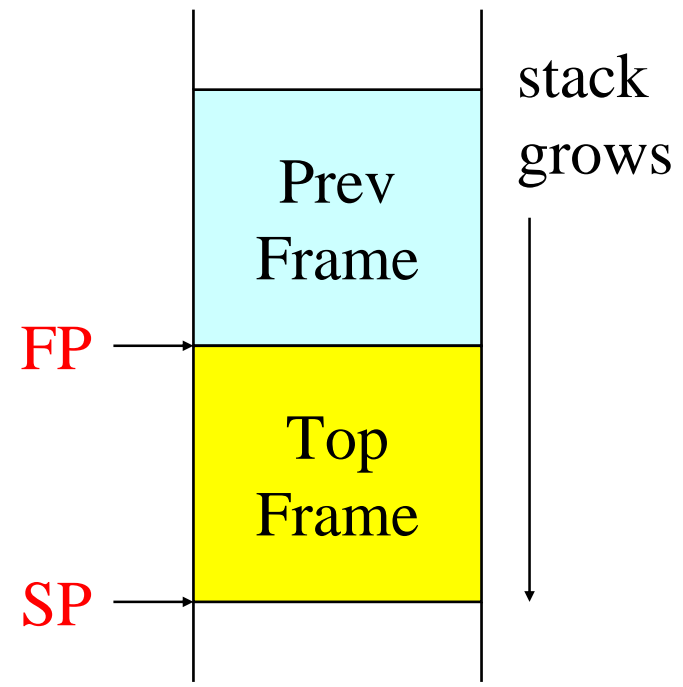
Run-Time Stack

- ❖ A frame (or activation record) for each function execution
 - » Represents execution environment of the function
 - Per invocation! Recursive function – each dynamic call has its own frame
 - » Includes: local variables, parameters, return value, temporary storage (register spill)
- ❖ Run-time stack of frames
 - » Push frame of *f* on stack when program calls *f*
 - » Pop stack frame when *f* returns
 - » Top frame = fame of currently executing function

Stack Pointers

- ❖ Assuming stack grows downwards
 - » Address of top of stack increases
- ❖ Values of current frame accessed using 2 pointers

- » **Stack pointer (SP)**:
points to frame top
- » **Frame pointer (FP)**:
points to frame base
- » Variable access: use
offset from FP (SP)

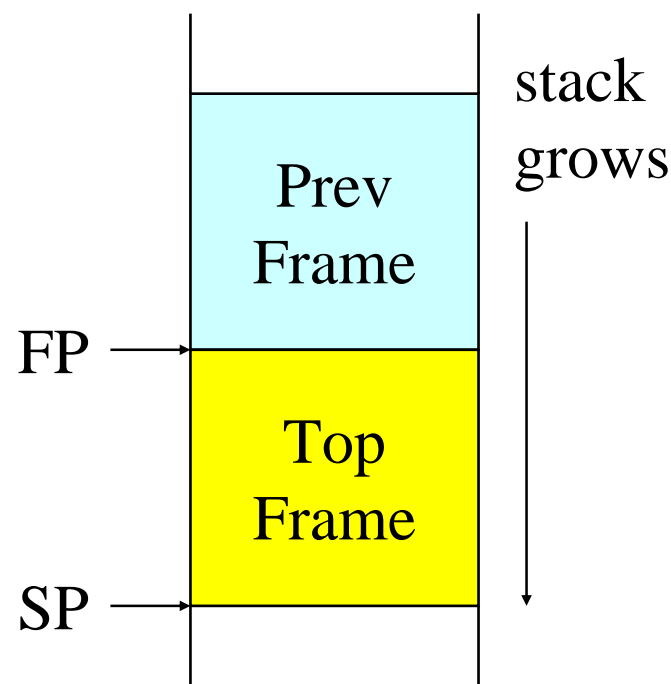


Why 2 Stack Pointers?

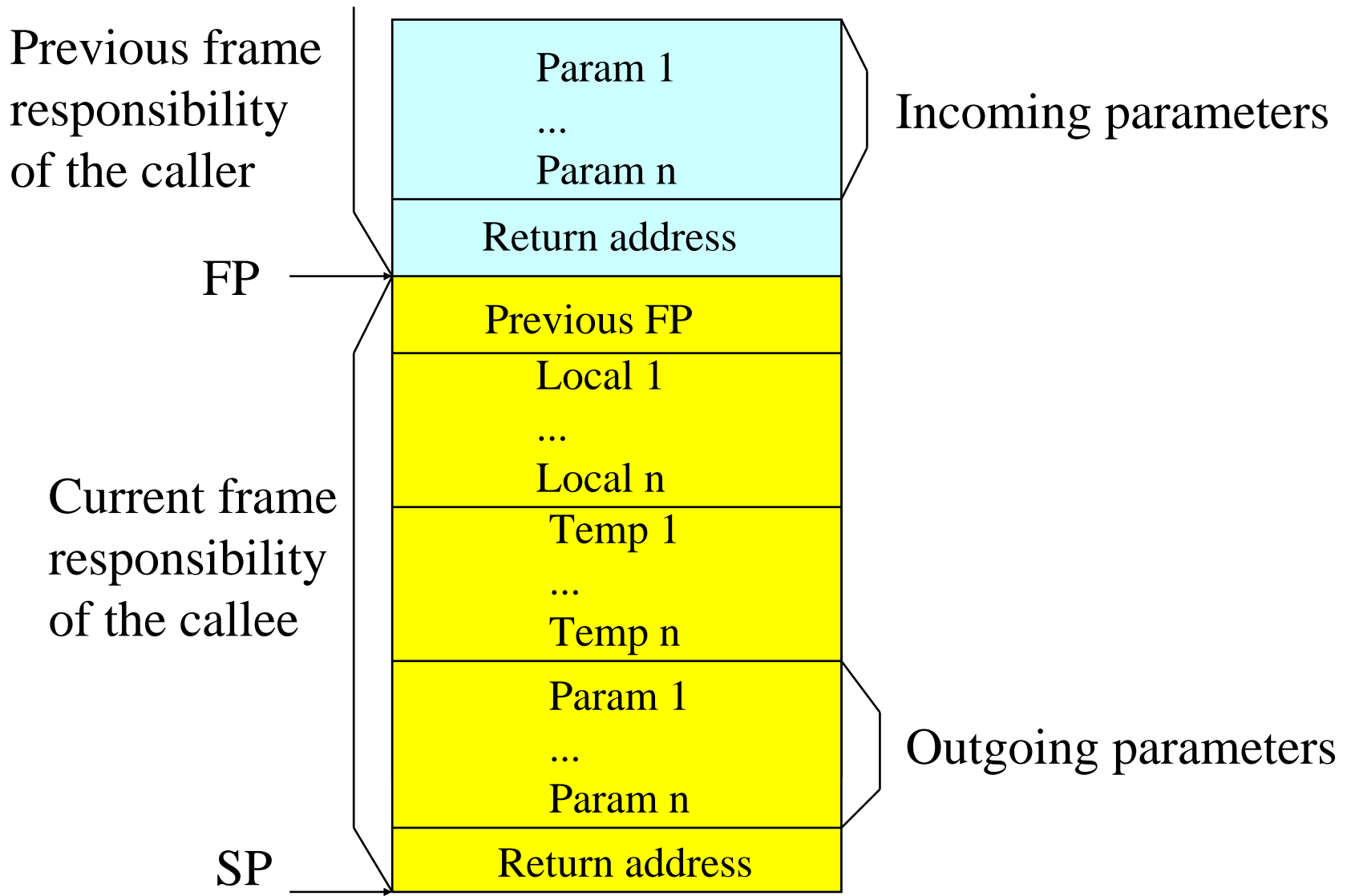
- ❖ For fun – not quite!
- ❖ Keep small offsets
 - » Instruction encoding limits sizes of literals that can be encoded

- ❖ **Real reason**

- » **Stack frame size not always known at compile time**
- » **Example: alloca (dynamic allocation on stack)**



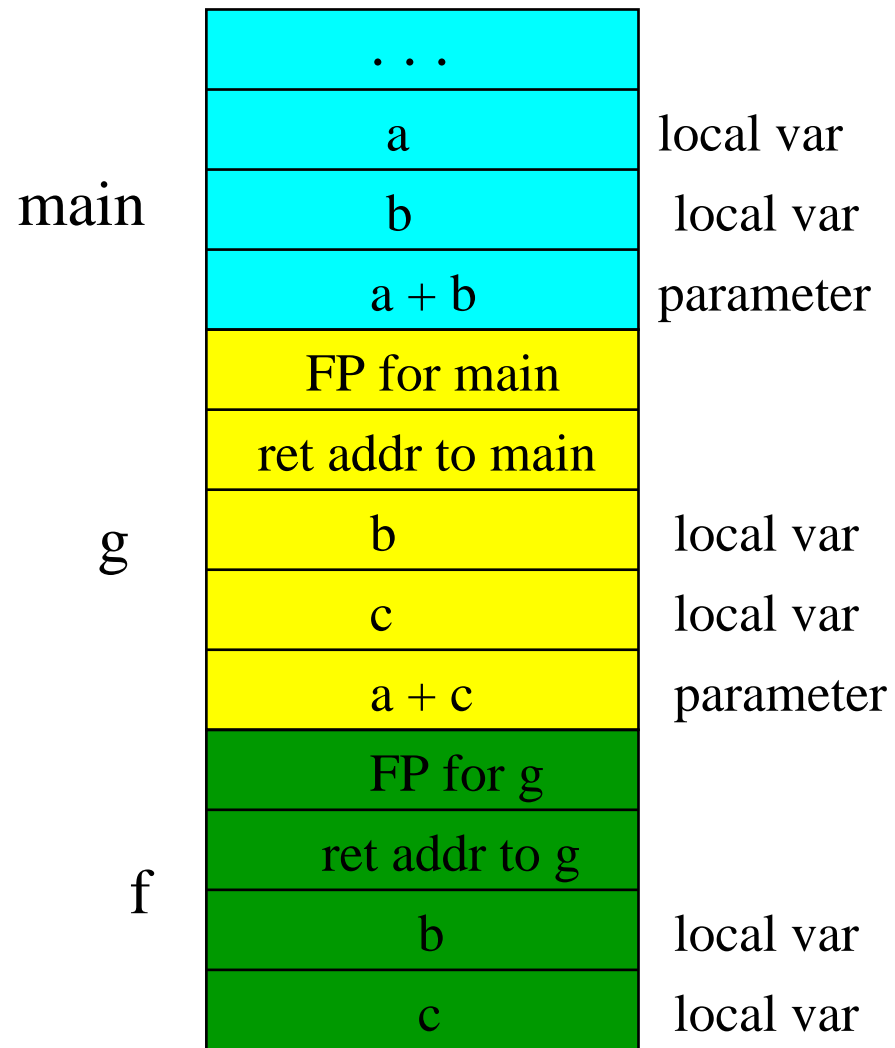
Anatomy of a Stack Frame



Stack Frame Construction Example

```
int f(int a) {
    int b, c;
}
void g(int a) {
    int b, c;
    ...
    b = f(a+c);
    ...
}
main() {
    int a, b;
    ...
    g(a+b);
    ...
}
```

Note: I have left out the temp part of the stack frame



Class Problem

For the following program:

```
int foo(int a) {  
    int x;  
    if (a = 1) return 1;  
    x = foo(a-1) + foo(a-2);  
    return (x);  
}
```

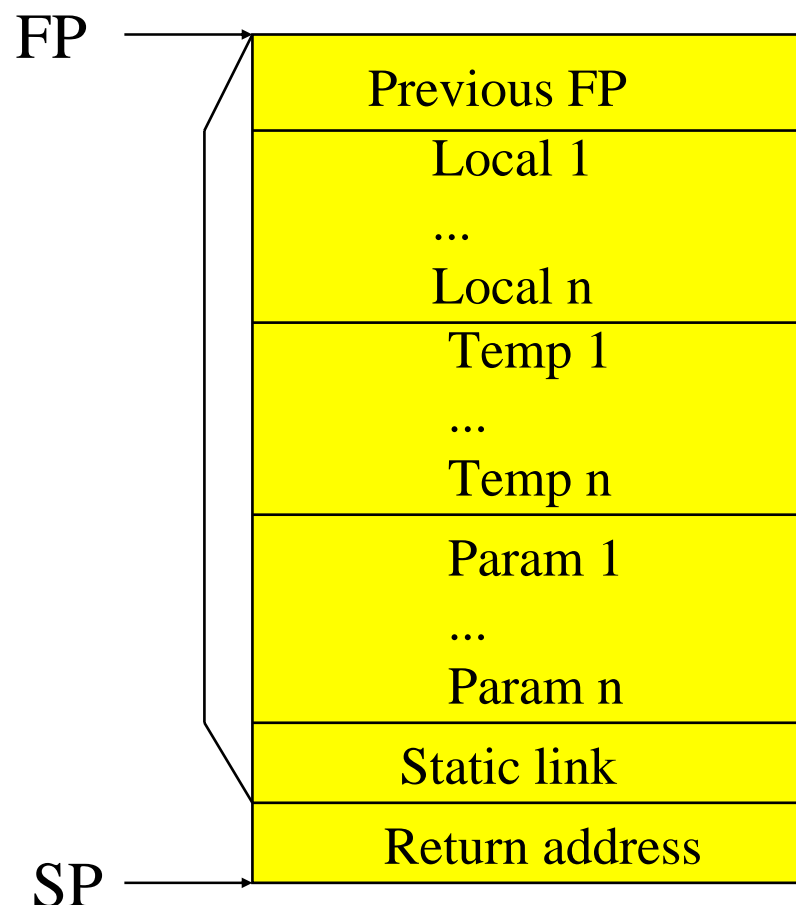
```
main() {  
    int y, z = 10;  
    y = foo(z);  
}
```

1. Show the first 3 stack frames created when this program is executed (starting with main).

2. Whats the maximum number of frames the stack grows to during the execution of this program?

Static Links

- ❖ Problem for languages with nested functions (Pascal, ML): How do we access local variables from other frames?
- ❖ Need a static link: a pointer to the frame of the enclosing function
 - » Defined away in C as C has only a 2 level binding
- ❖ Previous FP = dynamic link, ie, pointer to the previous frame in the current execution



Saving Registers

- ❖ Problem: Execution of invoked function may overwrite useful values in registers
- ❖ Generated code must:
 - » Save registers when function is invoked
 - » Restore registers when function returns
- ❖ Possibilities
 - » Callee saves/restores registers
 - » Caller saves/restores registers
 - » Split up the job, ie both do part of it

Calling Sequences

- ❖ How to generate the code that builds the frames?
- ❖ Generate code which pushes values on stack:
 - » Before call instructions (caller responsibilities)
 - » At function entry (callee responsibilities)
- ❖ Generate code which pops values off stack:
 - » After call instructions (caller responsibilities)
 - » At return instructions (callee responsibilities)

Push Values on Stack

- ❖ Code before call instruction
 - » Push each actual parameter
 - » Push caller-saved registers
 - » Push static link (if necessary)
 - » Push return address (current PC) and jump to caller code

- ❖ Prologue = code at function entry
 - » Push dynamic link (ie FP)
 - » Old stack pointer becomes new frame pointer
 - » Push callee-saved registers
 - » Push local variables

Pop Values from Stack

- ❖ Epilogue = code at return instruction
 - » Pop (restore) callee-saved registers
 - » Store return value at appropriate place
 - » Restore old stack pointer (pop callee frame)
 - » Pop old frame pointer
 - » Pop return address and jump to that address
- ❖ Code after call
 - » Pop (restore) caller-saved registers
 - » Use return value

Example Call

- ❖ Consider call `foo(3,5)`, assume machine has 2 registers `r1`, `r2` that are both callee save
- ❖ Code before call instruction
 - » `push arg1: [sp] = 3`
 - » `pushd arg2: [sp] = 5`
 - » `make room for return address and 2 args: sp = sp+12`
 - » `call foo`
- ❖ Prologue
 - » `push old frame pointer: [sp] = fp`
 - » `compute new fp: fp = sp`
 - » `push r1, r2: [sp+4] = r1, [sp+8] = r2`
 - » `create frame with 3 local (int) variables, sp = sp+24`

Example Call, continued

❖ Epilogue

- » pop r1, r2: $r1 = [sp-16]$, $r2 = [sp-20]$
- » restore old fp: $fp = [sp-4]$, $sp=sp-4$
- » pop frame: $sp = sp-24$
- » pop return address and execute return: rts

❖ Code after call

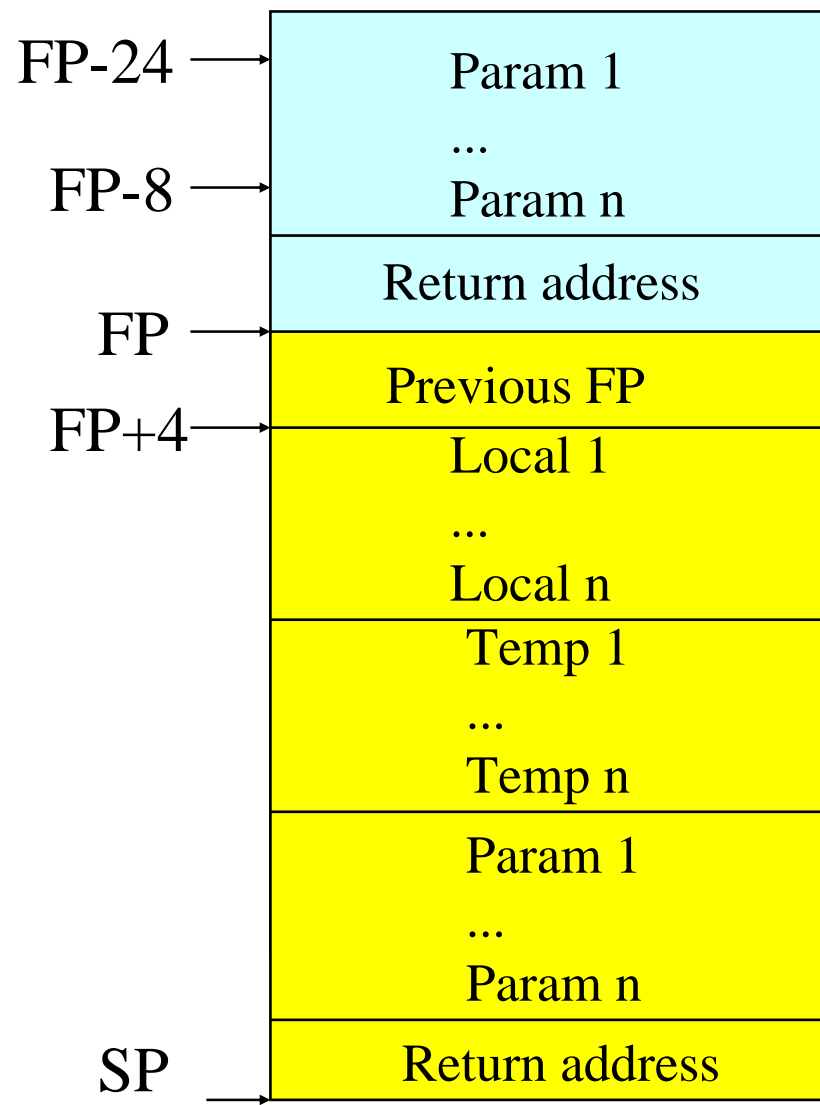
- » use return value
- » pop args: $sp = sp-12$

Accessing Stack Variables

❖ To access stack variables: use offsets from FP

❖ Example

- » $[fp-8] = \text{param } n$
- » $[fp-24] = \text{param } 1$
- » $[fp+4] = \text{local } 1$



Class Problem

Assume you are mapping this function onto a processor that has 4 general registers, r1, r2, r3, r4. r1/r2 are caller save, r3-r4 are callee save. Show how the stack frame is constructed for this recursive function. You should assume the registers r1-r4 contain useful values. What if they do not?

```
int foo(int a, int b) {  
    int x, y, z;  
    ...  
    z = foo(x,y);  
    ...  
    return z;  
}
```

Data Layout

- ❖ Naive layout strategies generally employed
 - » Place the data in the order the programmer declared it!
- ❖ 2 issues: size, alignment
- ❖ Size – How many bytes is the data item?
 - » Base types have some fixed size
 - E.g., char, int, float, double
 - » Composite types (structs, unions, arrays)
 - Overall size is sum of the components (not quite!)
 - Calculate an offset for each field

Memory Alignment

- ❖ Cannot arbitrarily pack variables into memory → Need to worry about alignment
- ❖ Golden rule – Address of a variable is aligned based on the size of the variable
 - » Char is byte aligned (any addr is fine)
 - » Short is halfword aligned (LSB of addr must be 0)
 - » Int is word aligned (2 LSBs of addr must be 0)
 - » This rule is for C/C++, other languages may have a slightly different rules

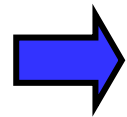
Structure Alignment (for C)

- ❖ Each field is layed out in the order it is declared using Golden Rule for aligning
- ❖ 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 can have an array of structs

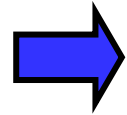
Structure Example

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

Largest field is int (4 bytes)



struct size is multiple of 4



struct starting addr is word aligned

Struct must start at word-aligned address

char w → 1 byte, start anywhere

x[3] → 12 bytes, but must start at word aligned addr,

so 3 empty bytes between w and x

char y → 1016, start anywhere

short z → 2 bytes, but must start at halfword aligned addr,

so 1 empty byte between y and z

Total size = 20 bytes!

Class Problem

How many bytes of memory does the following sequence of C declarations require (int = 4 bytes) ?

```
short a[100];  
char b;  
int c;  
double d;  
short e;  
struct {  
    char f;  
    int g[1];  
    char h[2];  
} i;
```