Functions

Terms

```
void foo() {
  int a, b;
  ...
  bar(a, b);
}


void bar(int x, int y) {
   ...
}
```

- foo is the *caller*
- bar is the *callee*
- a, b are the *actual parameters* to bar
- x, y are the *formal parameters* of bar
- Shorthand:
  - argument = actual parameter
  - parameter = formal parameter

Different kinds of parameters

- Value parameters
- Reference parameters

Value parameters

- “Call-by-value”
- Used in C, Java, default in C++
- Passes the value of an argument to the function
- Makes a copy of argument when function is called
- Advantages? Disadvantages?

Value parameters

```
int x = 1;
void main () {
    foo(x, x);
    print(x);
}

void foo(int y, int z) {
    y = 2;
    z = 3;
    print(x);
}
```

- What do the print statements print?
- Answer:

print(x); //prints 1

print(x); //prints 1

Reference parameters

- “Call-by-reference”
- Optional in Pascal (use “var” keyword) and C++ (use “&”)
- Pass the *address* of the argument to the function
- If an argument is an expression, evaluate it, place it in memory and then pass the address of the memory location
- Advantages? Disadvantages?

Reference parameters

```
int x = 1;
void main () {
   foo(x, x);
   print(x);
}

void foo(int &y, int &z) {
   y = 2;
   z = 3;
   print(x);
   print(y);
}
```

- What do the print statements print?
- Answer:

  print(x); //prints 3

  print(x); //prints 3

  print(y); //prints 3!

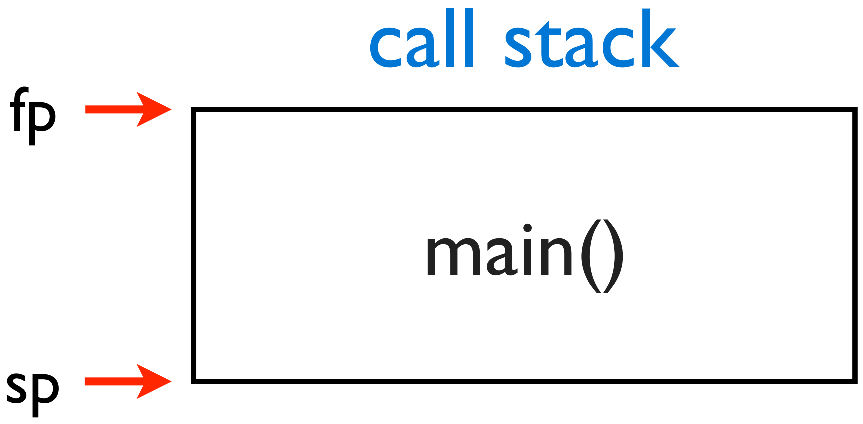
Other considerations

- Scalars
    - For call by value, can pass the address of the actual parameter and copy the value into local storage within the procedure
        - Reduces size of caller code (why is this good?)
    - For machines with a lot of registers (*e.g.*, MIPS), compilers will save a few registers for arguments and return types
        - Less need to manipulate stack

Other considerations

- Arrays
  - For efficiency reasons, arrays should be passed by reference (why?)
    - Java, C, C++ pass arrays by reference by default (technically, they pass a pointer to the array by value)
  - Callee can copy array into local storage as needed

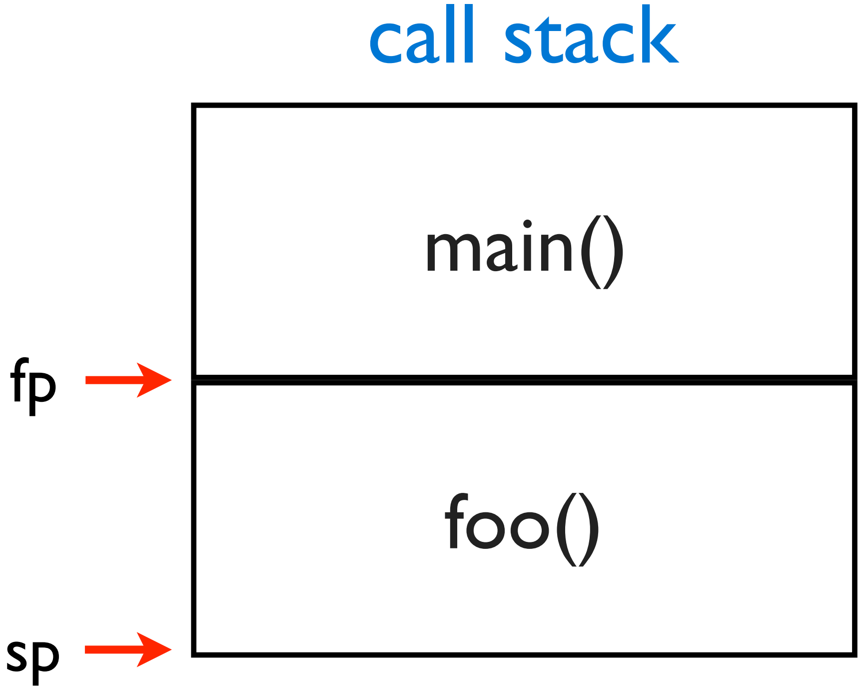
Function call behavior



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

foo() {
    bar();
    ...
    baz();
}
```

Function call behavior


call stack

main()

foo()

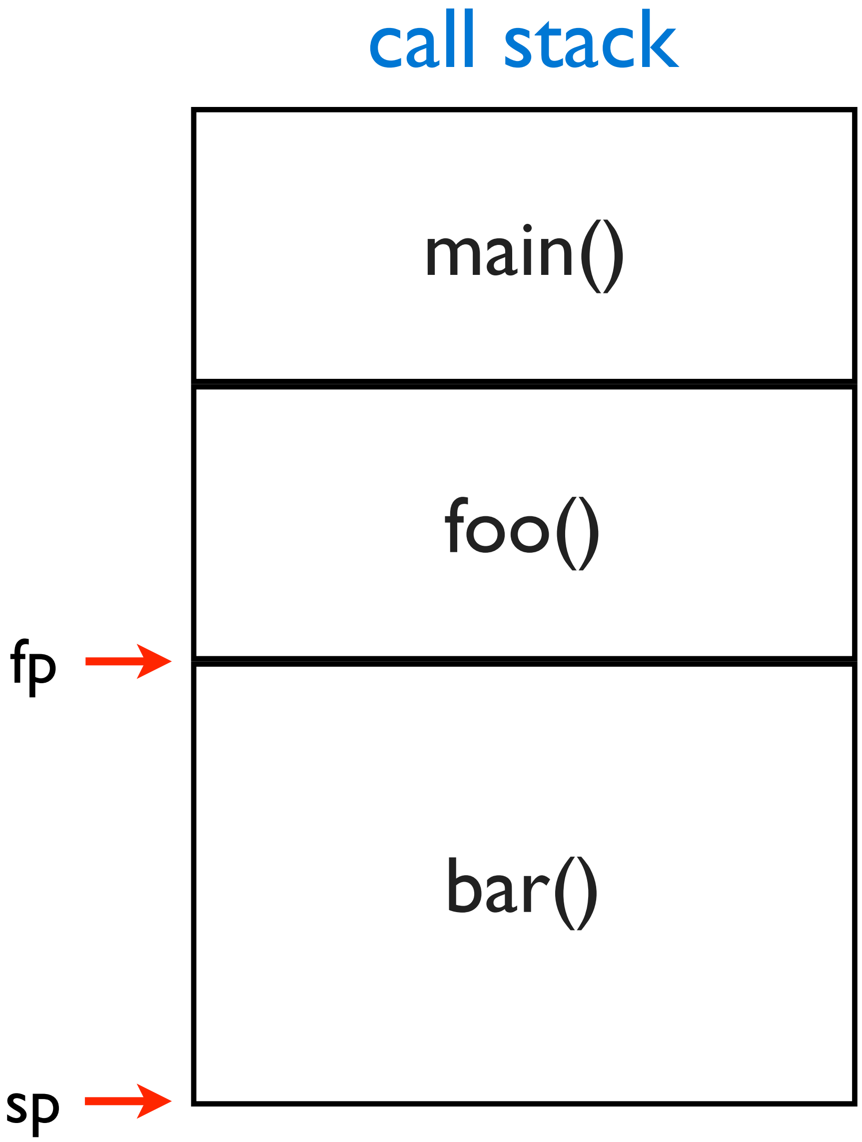
fp

sp

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

foo() {
    bar();
    ...
    baz();
}
```

Function call behavior



call stack

main()

foo()

bar()

fp

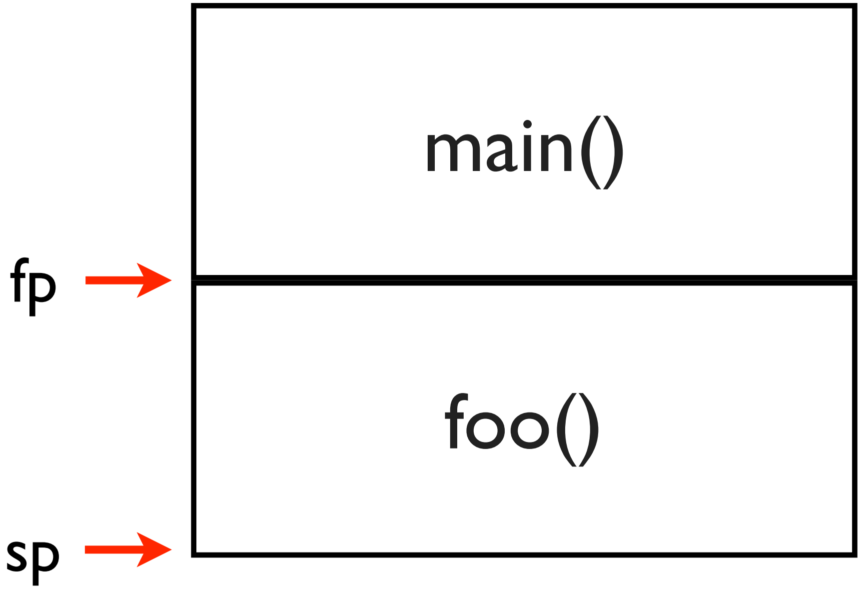
sp

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

foo() {
    bar();
    ...
    baz();
}
```

Function call behavior

call stack



main()

foo()

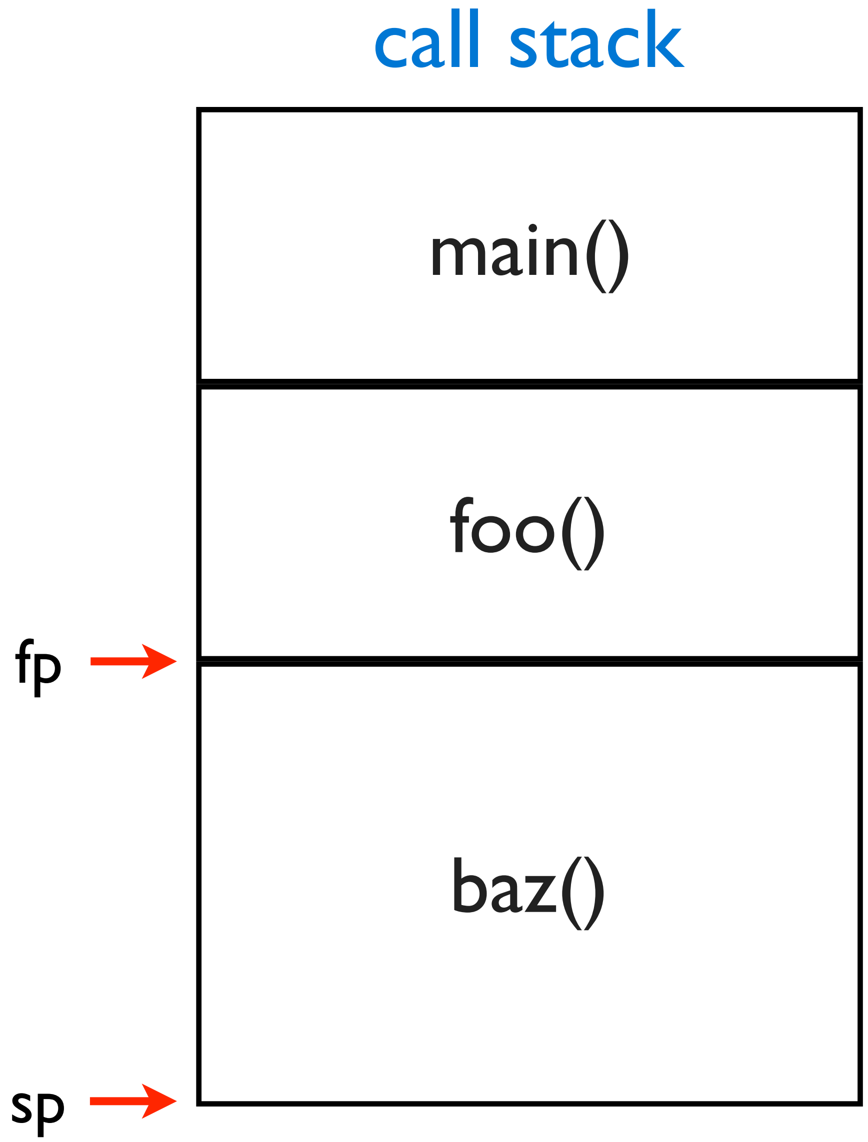
fp

sp

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

foo() {
    bar();
    ...
    baz();
}
```

Function call behavior



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

foo() {
    bar();
    ...
    baz();
}
```

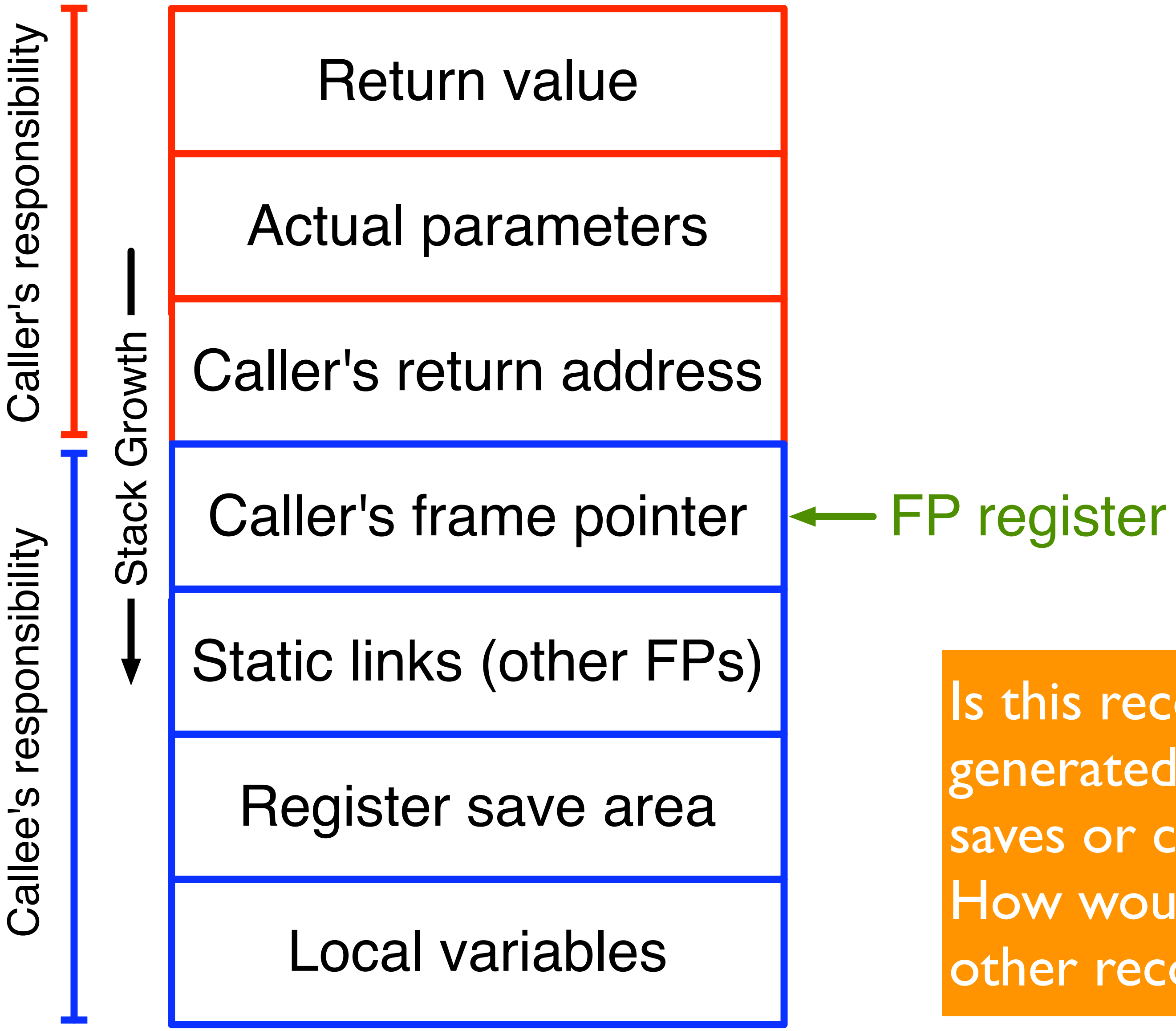
Calling a function

- What should happen when a function is called?
    - Set the *frame pointer* (sets the base of the activation record)
    - Allocate space for local variables (use the function's symbol table for this)
    - What about registers?
        - Callee might want to use registers that the caller is using

Saving registers

- Two options: *caller saves* and *callee saves*
- Caller saves
  - Caller pushes all the registers it is using on to the stack before calling function, restores the registers after the function returns
- Callee saves
  - Callee pushes all the registers it is *going to use* on the stack immediately after being called, restores the registers just before it returns
- Why use one vs. the other?
- Simple optimizations are good here: don't save registers if the caller/callee doesn't use any

Activation records



Caller's responsibility

Callee's responsibility

Stack Growth

Return value

Actual parameters

Caller's return address

Caller's frame pointer

Static links (other FPs)

Register save area

Local variables

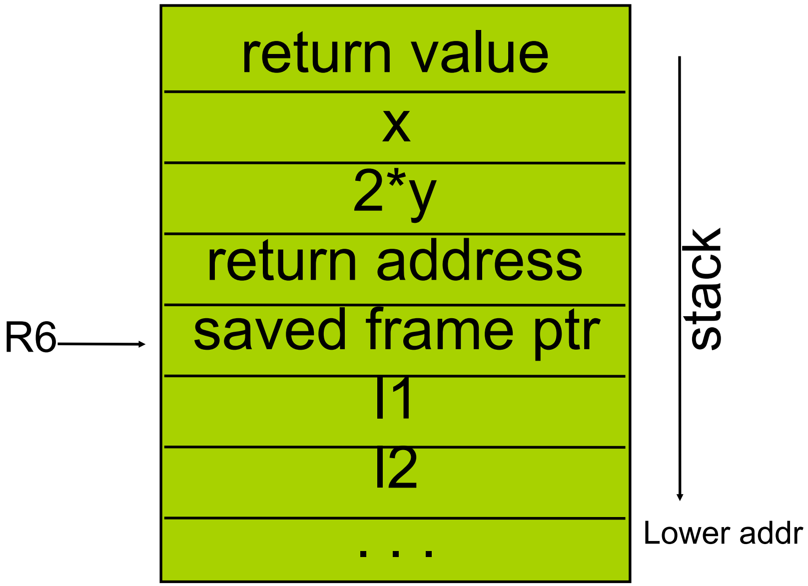
FP register

Is this record generated for callee-saves or caller-saves? How would the other record look?

The frame pointer

- Manipulate with instructions like `link` and `unlink`
  - Link: push current value of FP on to stack, set FP to top of stack
  - Unlink: read value at current address pointed to by FP, set FP to point to that value
  - In other words: link pushes a new frame onto the stack, unlink pops it off

Example Subroutine Call and Stack Frame



```
z = SubOne(x,2*y);
```

3-address code:

```
push
push x
mul  2  y  t1
push t1
jsr SubOne
pop
pop
pop  z
```

assembly code:

```
push
push x
load  y R1
muli  2 R1
push R1
jsr SubOne
pop
pop
pop  R1
store R1 z
```

```
int SubOne(int a, int b) {
      int l1, l2;
      l1 = a;
      l2 = b;
      return l1+l2;
   };
```

```
link  3
move $P1 $L1
move $P2 $L2
add $L1 $L2 t2
move t2 $R
unlink
ret
```

```
link R6 3
load  3(R6)  R1
store R1 -1(R6)
load  2(R6)  R2
store R2 -2(R6)
load  -1(R6)  R1
add   -2(R6)  R1
store R1  4(R6)
unlink
ret
```