CS323: Compilers

Spring 2023

Week 9: Functions, Local Optimizations

Recap

Activation

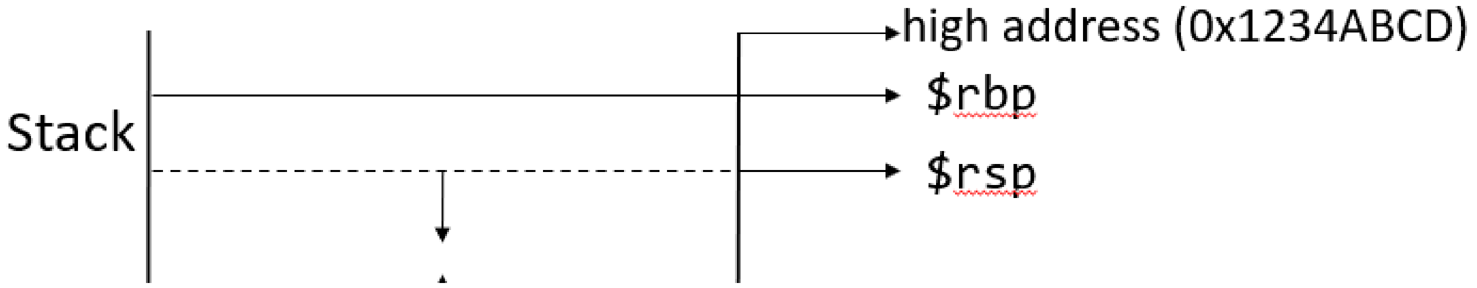
- A function call or invocation is termed an *activation*
- Calls to functions in a program form *activation tree*
  - Postorder traversal of the tree shows return sequence i.e. the order in which control returns from functions
  - Preorder traversal of the tree shows calling sequence
- In a sequential program, at any point in time, *control of execution is in any one activation*
  - All the ancestors of that activation are active i.e. have not returned

Activation

- Activations are managed through the help of *control stack*
- A function call (activation) results in allocating a chunk of memory called *activation record* or *frame* on the stack (also called *stack frame*)

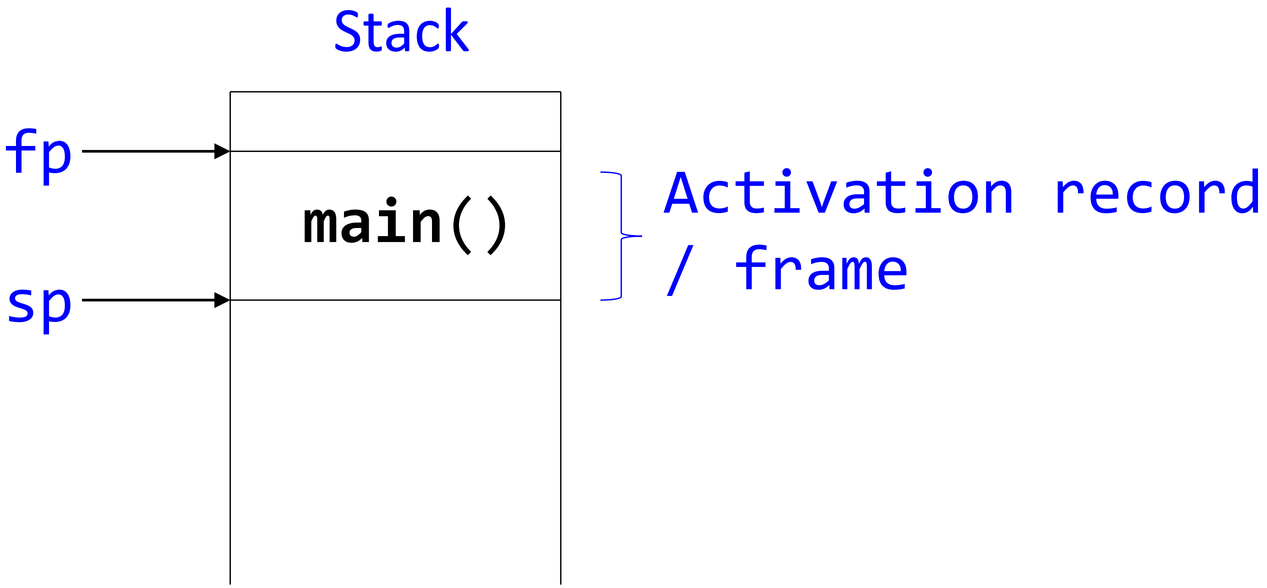
Activation Record

- A *sub-segment* of memory on the stack
  - Special registers `$rbp` and `$rsp` track the bottom and top of the stack frame. These are the names in x86 architecture.



- `$rbp` – base pointer or frame pointer (`fp`)
- `$rsp` – stack pointer (`sp`)

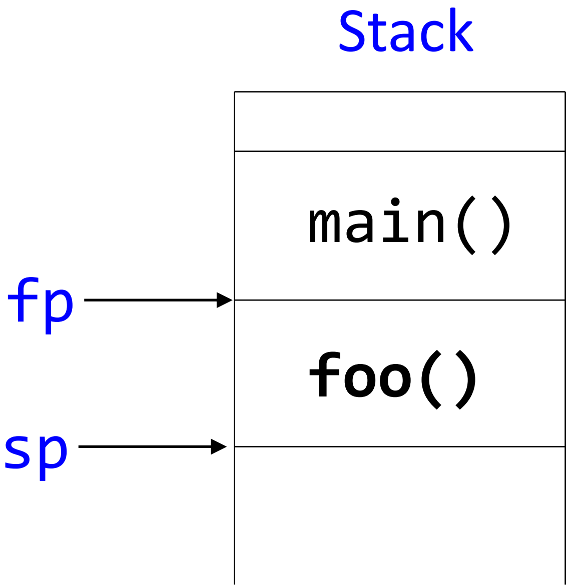
Activation Record - Example




```
main() {
    …
    foo();
    …
}

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

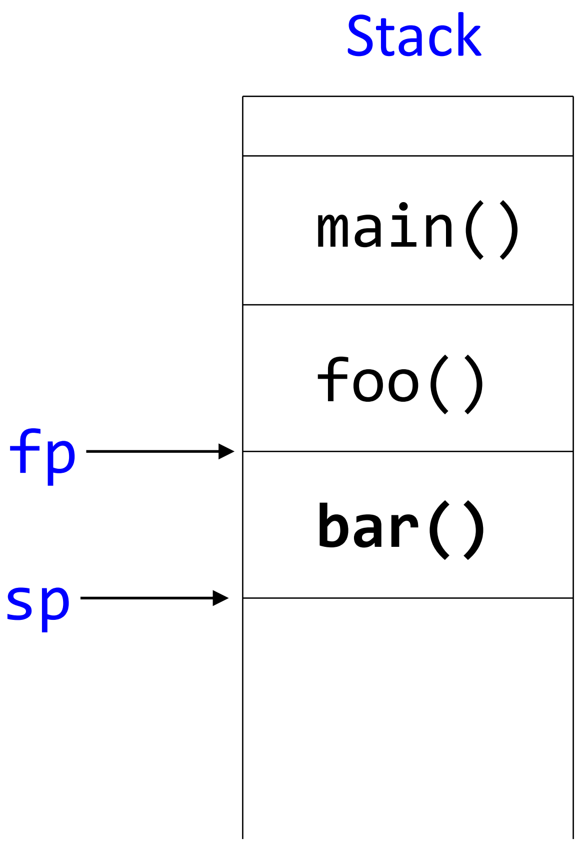
Activation Record - Example




```
main() {
    …
    foo();
    …
}

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

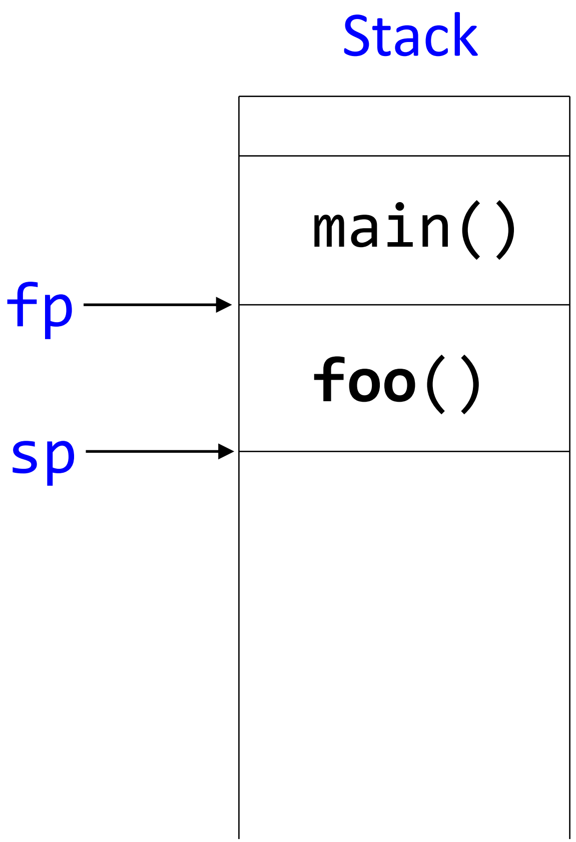
Activation Record - Example




```
main() {
   …
   foo();
   …
}

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

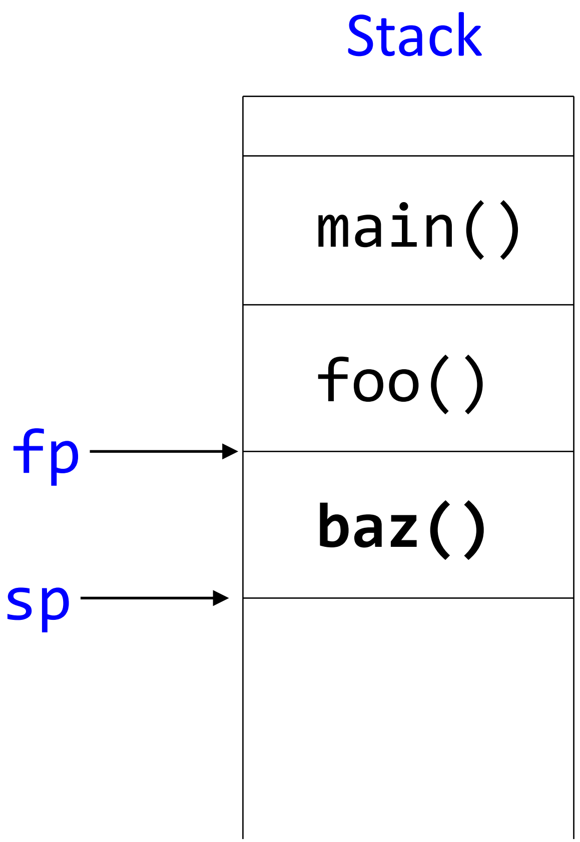
Activation Record - Example




```
main() {
   …
   foo();
   …
}

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

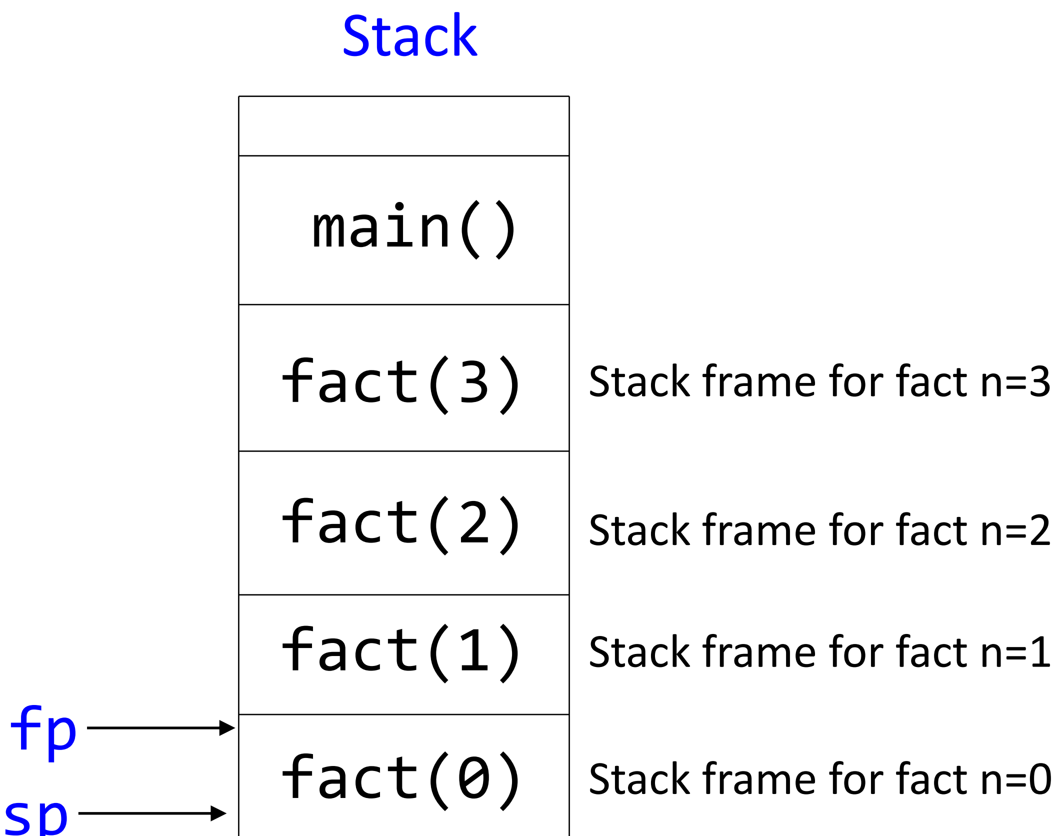
Activation Record - Example




```
main() {
    …
    foo();
    …
}

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

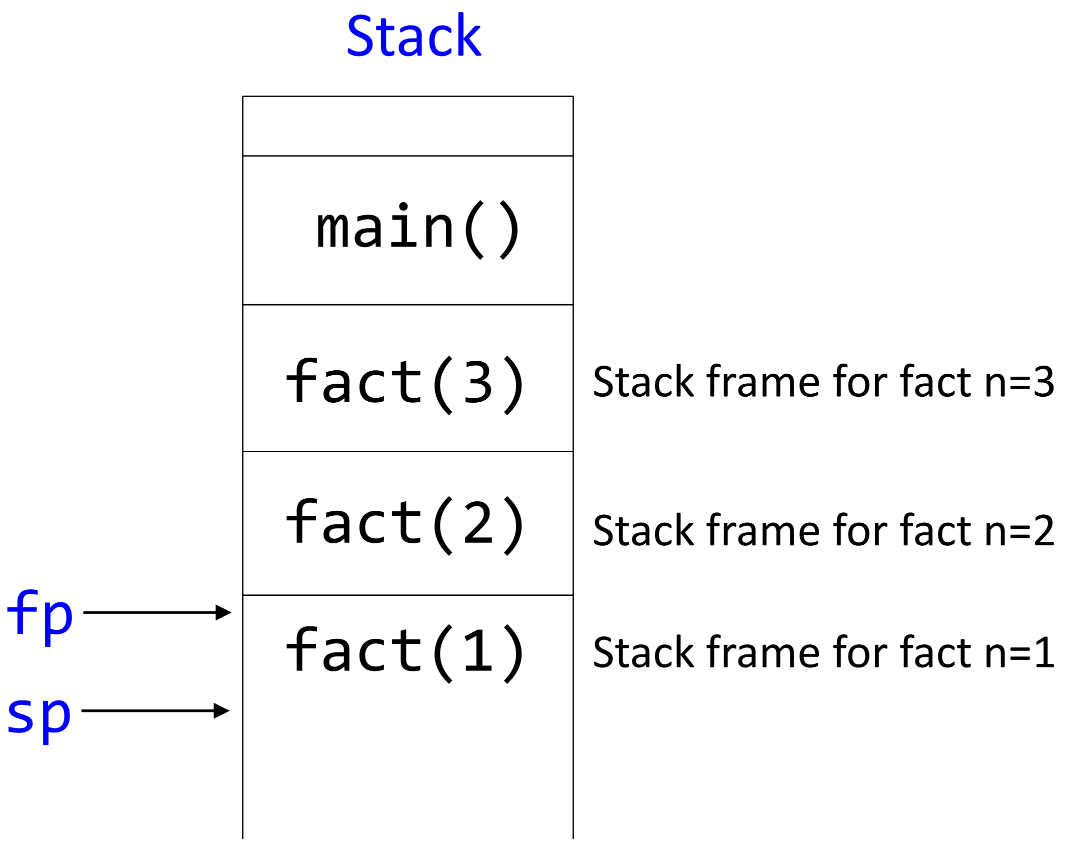
Activation Record – Example (Recursive Functions)




```
main() {
    …
    fact(3);
    …
}

fact(int n) {
if (n=0) return 1
return n*fact(n-1)
}
```

Activation Record – Example (Recursive Functions)

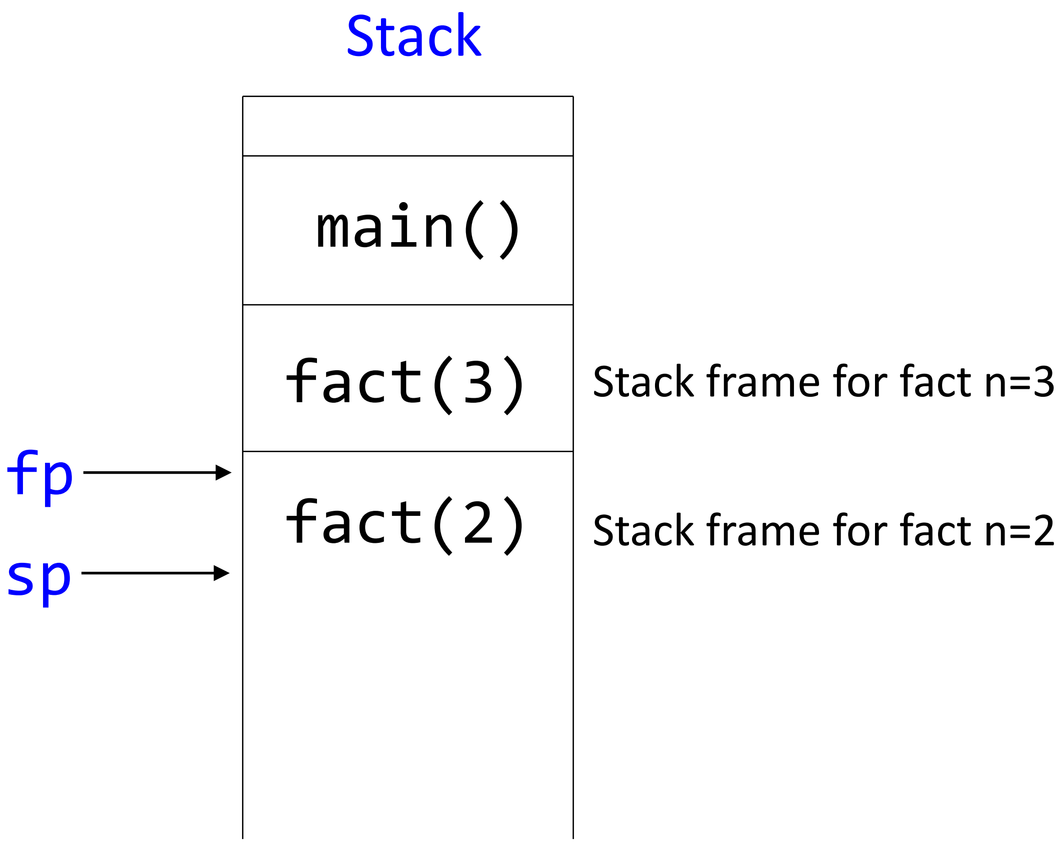



```
main() {
    …
    fact(3);
    …
}

fact(int n) {
if (n=0) return 1
return n*fact(n-1)
}
```

Stack frame for n=0 popped off. 1 Returned.

Activation Record – Example (Recursive Functions)

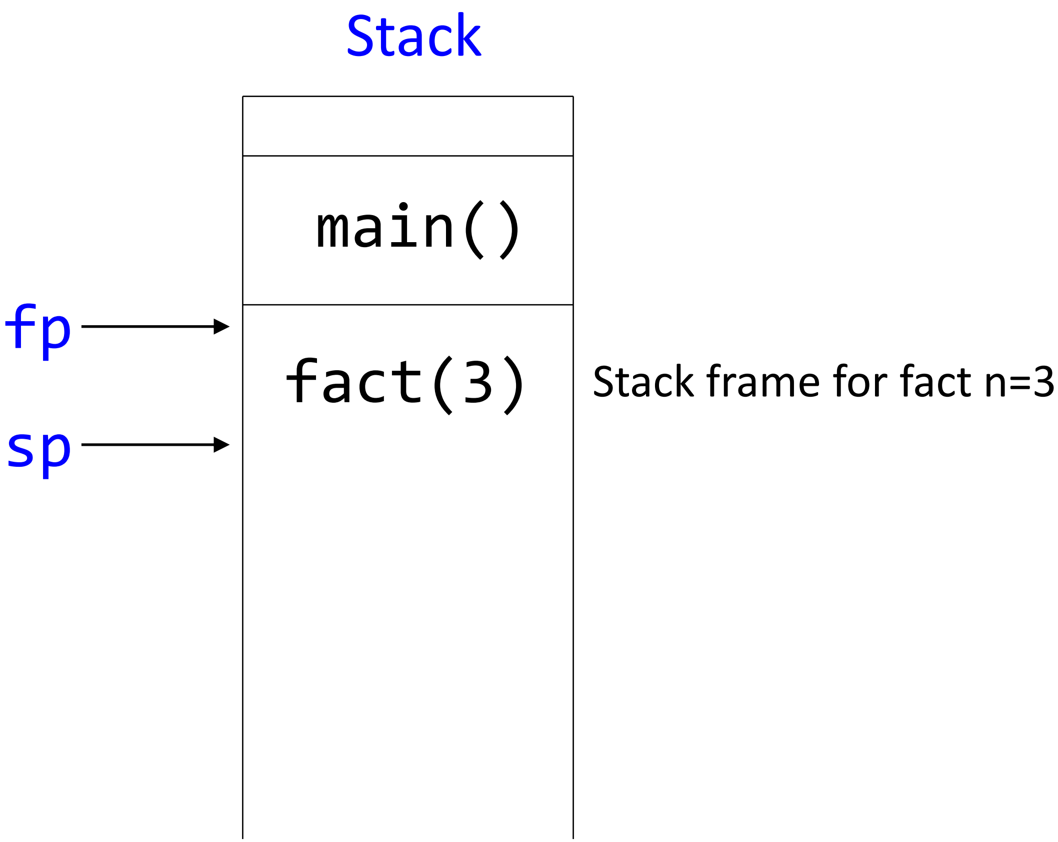



```
main() {
   …
   fact(3);
   …
}

fact(int n) {
if (n=0) return 1
return n*fact(n-1)
}
```

Stack frame for n=1 popped off. 1 Returned.

Activation Record – Example (Recursive Functions)

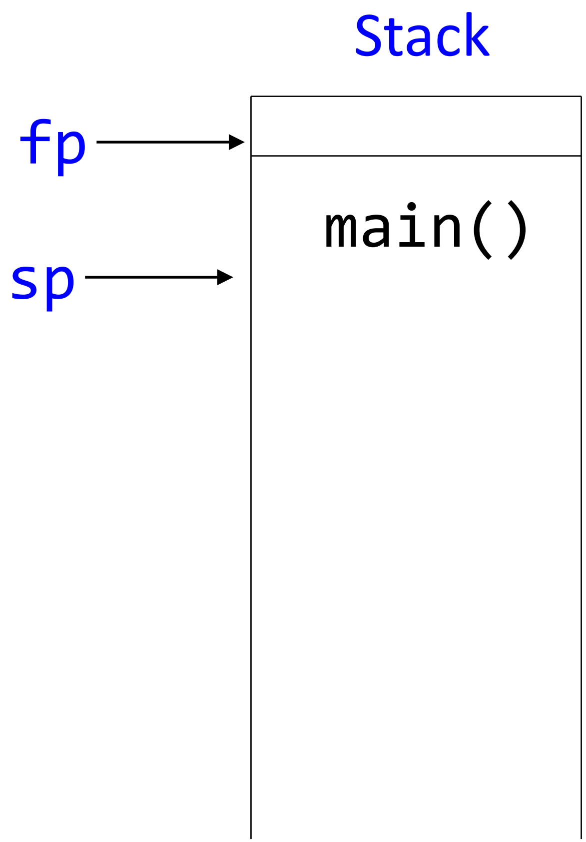



```
main() {
   …
   fact(3);
   …
}

fact(int n) {
if (n=0) return 1
return n*fact(n-1)
}
```

Stack frame for n=2 popped off. 2 Returned.

Activation Record – Example (Recursive Functions)




```
main() {
   …
   fact(3);
   …
}

fact(int n) {
if (n=0) return 1
return n*fact(n-1)
}
```

Stack frame for n=3 popped off. 6 Returned.

Activation Record

- What happens when a function is called?
  1. `fp` and `sp` get adjusted
  2. Memory for the activation record is allocated on stack
     - The size of the memory allocated depends on local variables used by the called function (consult function's symbol table for this)
  3. Each invocation of a function has its own instantiation of local variables
- When the function call returns:
  - Memory for the activation record is destroyed

Activation Record

- What is stored in the activation record?

  Depends on the language being implemented:
  - Temporaries
  - Local vars
  - Saved registers
  - Return address, previous `fp`
  - Return value
  - Actual Params
- Who stores this information?
  - Caller
  - Callee

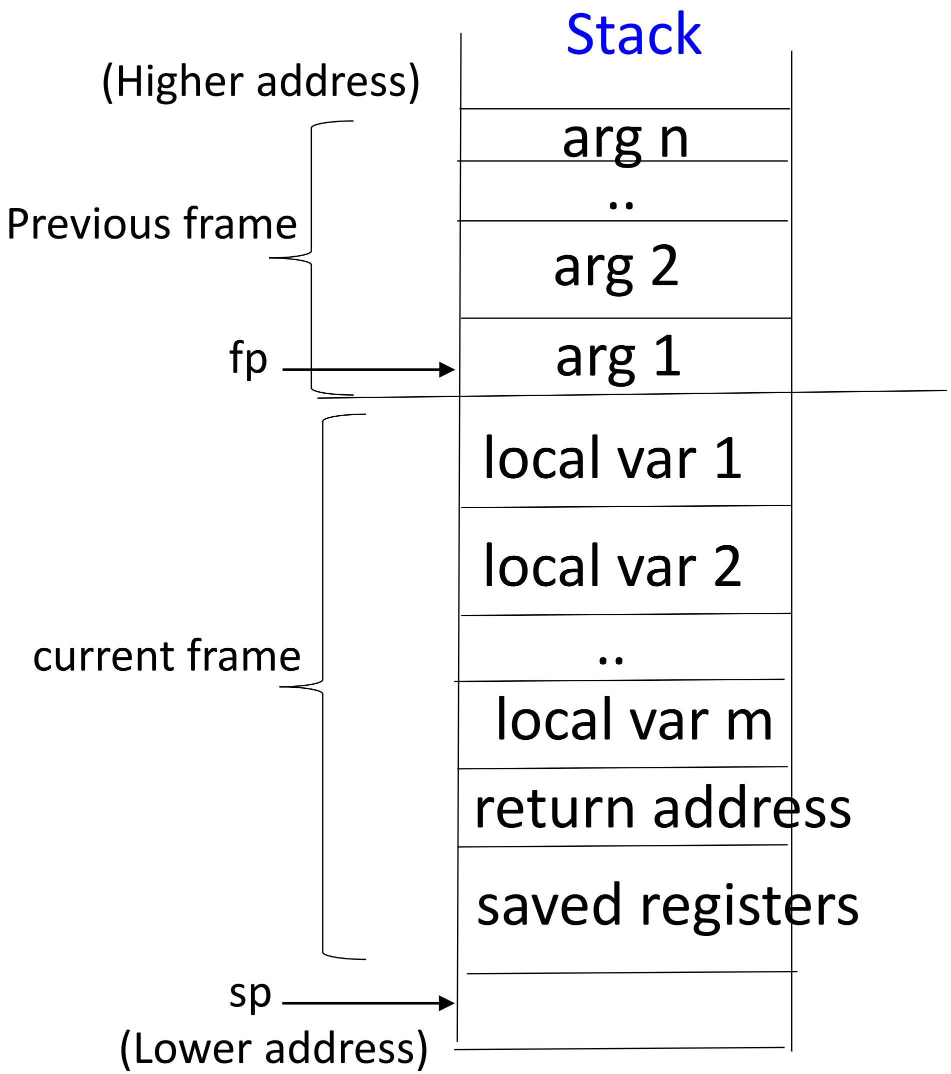
  together execute *calling sequence* and *return sequence*

Application Binary Interface (ABI)

- How is data organized on the activation record?
  - ABI is the specification on how data is provided to functions
    - Caller saves or callee saves
  - ABI is meant to deliver interoperability between different compilers
    - Compile the function using one compiler to create an object code, Link object code with other code compiled using a different compiler

forms the *calling convention*

Typical Activation Record



Callee accesses arguments using +ve offset from FP:
argument1 = memory[FP]
argument2 = memory[FP+1] ..

Callee accesses local variables using –ve offset from FP:
local var 1=memory[FP-1]
local var 2=memory[FP-2]

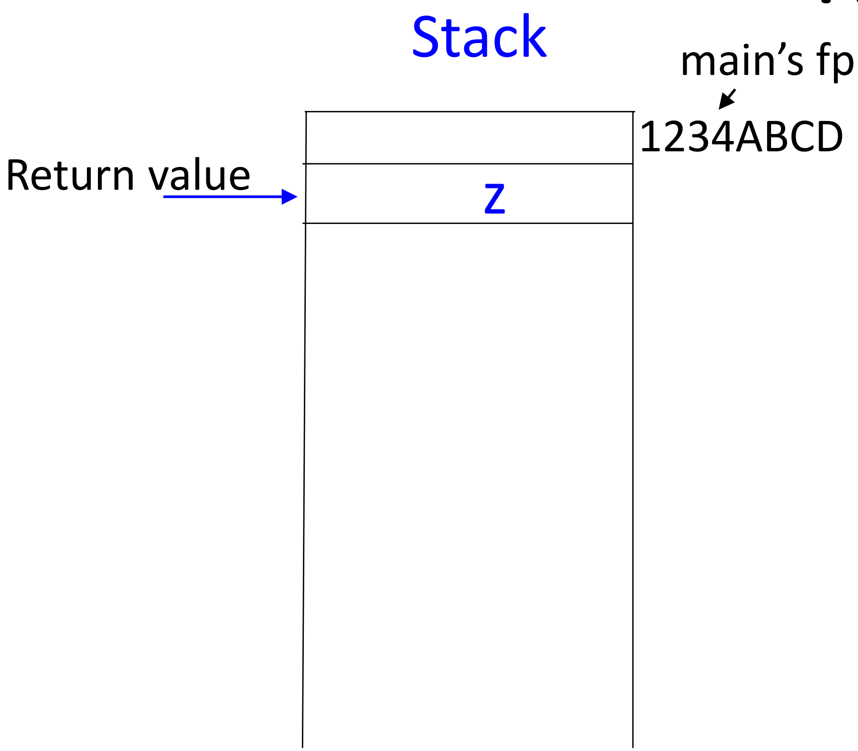
Function call: Peeking at Activation Record

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

- When `main` calls function `foo`
  1. The following are pushed on to the stack:
     1. `foo`'s arguments
     2. Space to hold `foo`'s return value
     3. Address of the next instruction executed (in `main`) when `foo` returns (return address)
     4. Current value of `$rbp` (frame pointer)

     `$rsp` is automatically updated (decremented) to point to current top of the stack.
  2. `$rbp` is assigned the value of `$rsp`

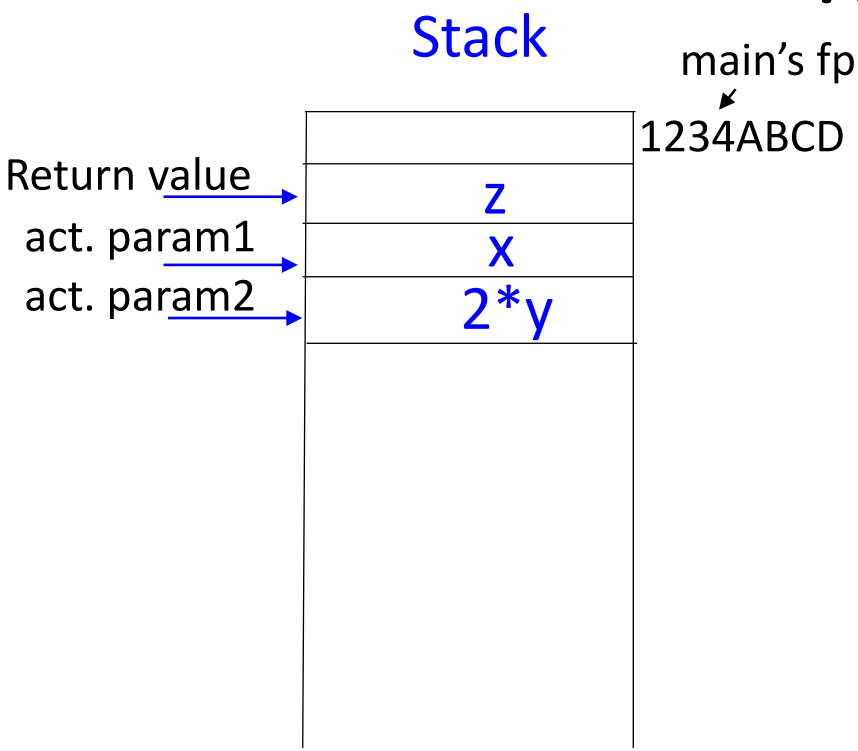
Function call: Peeking at Activation Record




```
main() {
   z=foo(x, 2*y);
   return;
}

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

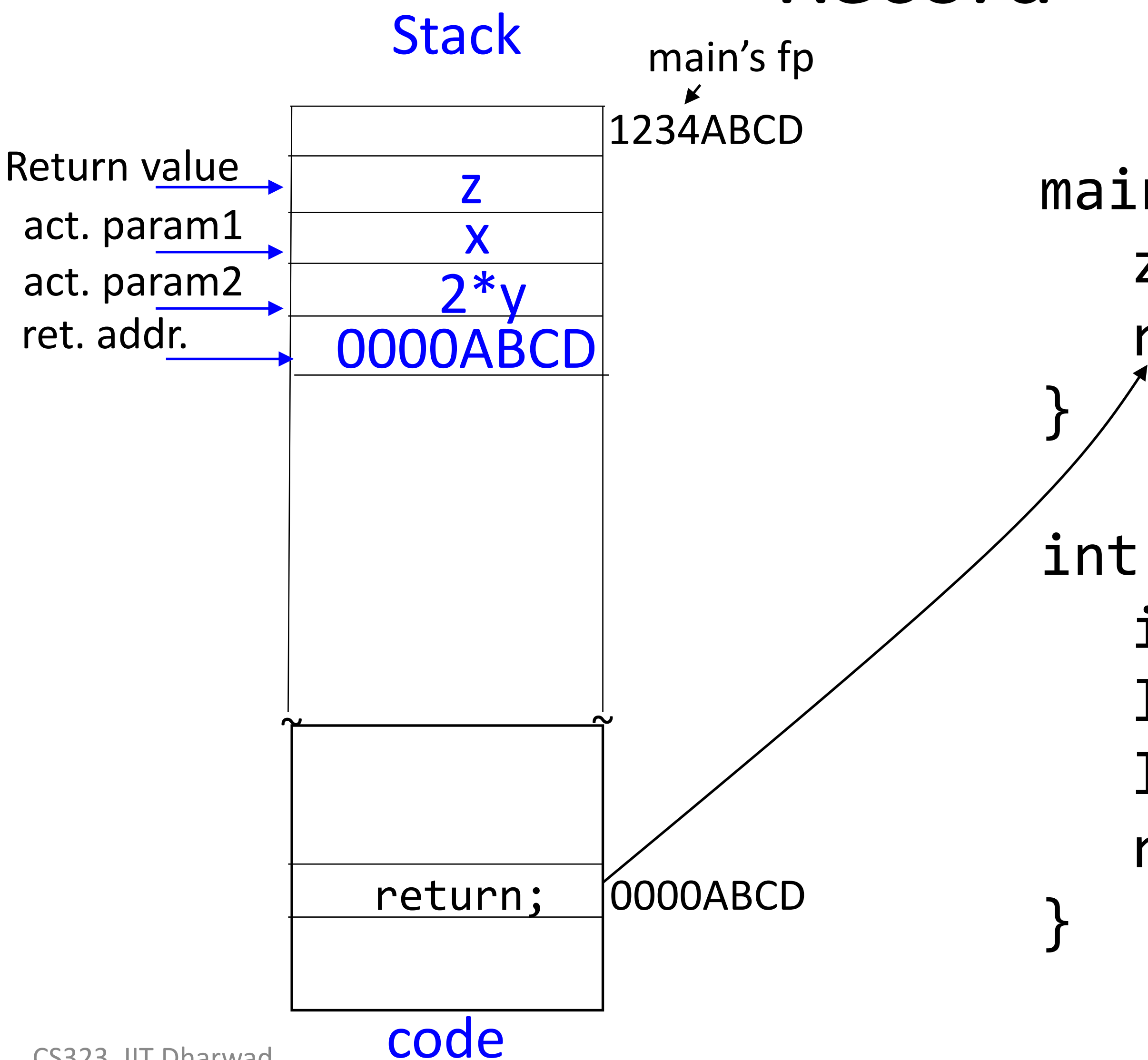
Function call: Peeking at Activation Record




```
main() {
   z=foo(x, 2*y);
   return;
}

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

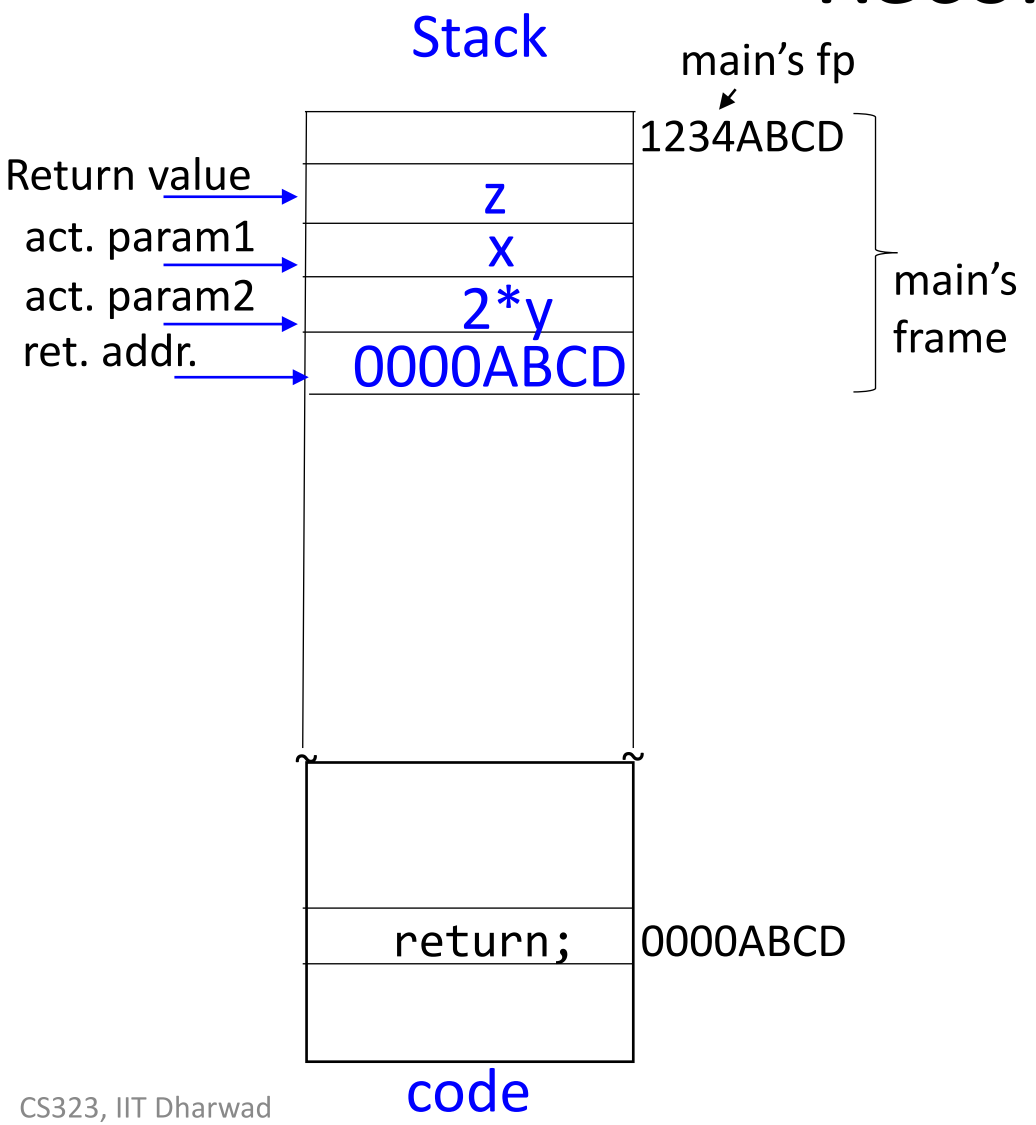
Function call: Peeking at Activation Record




```
main() {
   z=foo(x, 2*y);
   return;
}

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

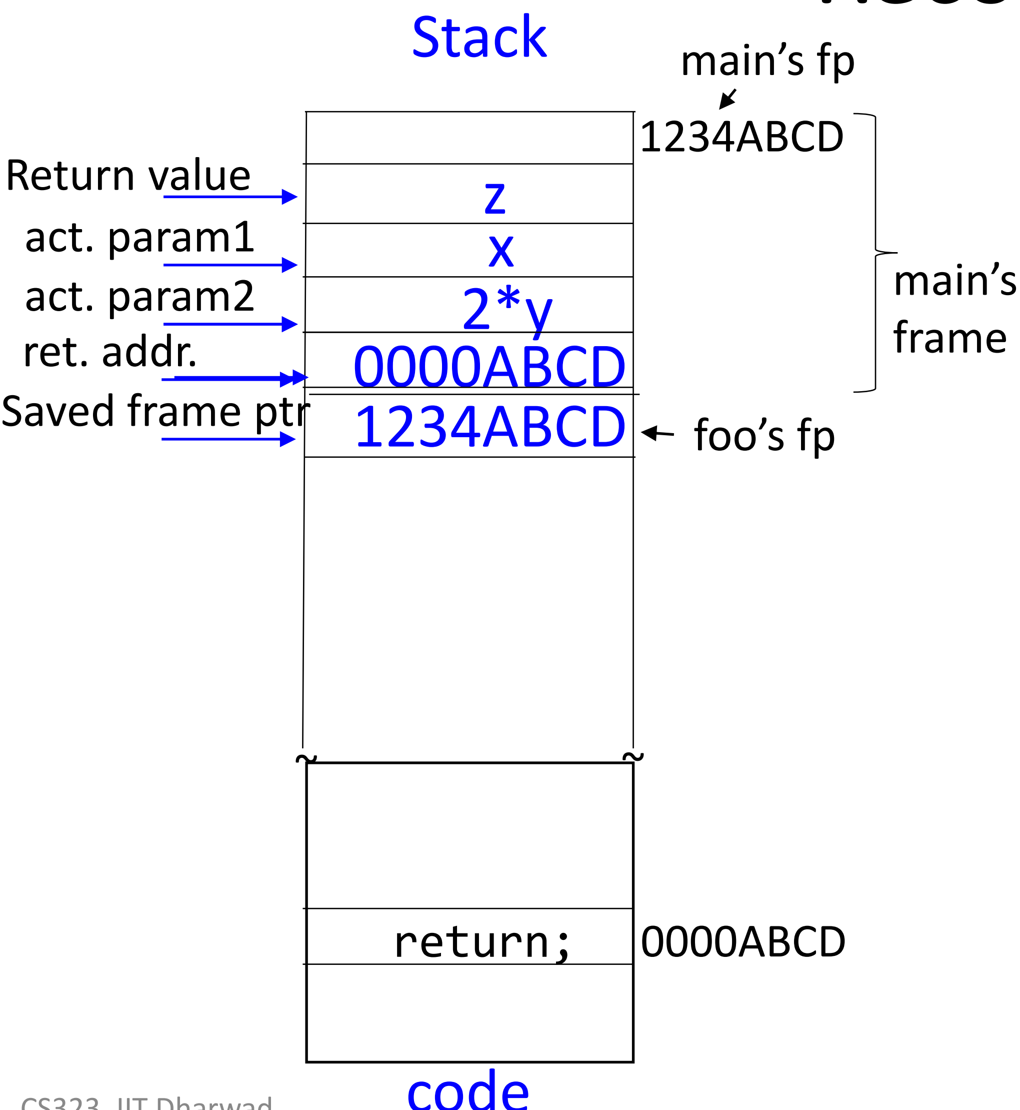
Function call: Peeking at Activation Record




```
main() {
    z=foo(x, 2*y);
    return;
}

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

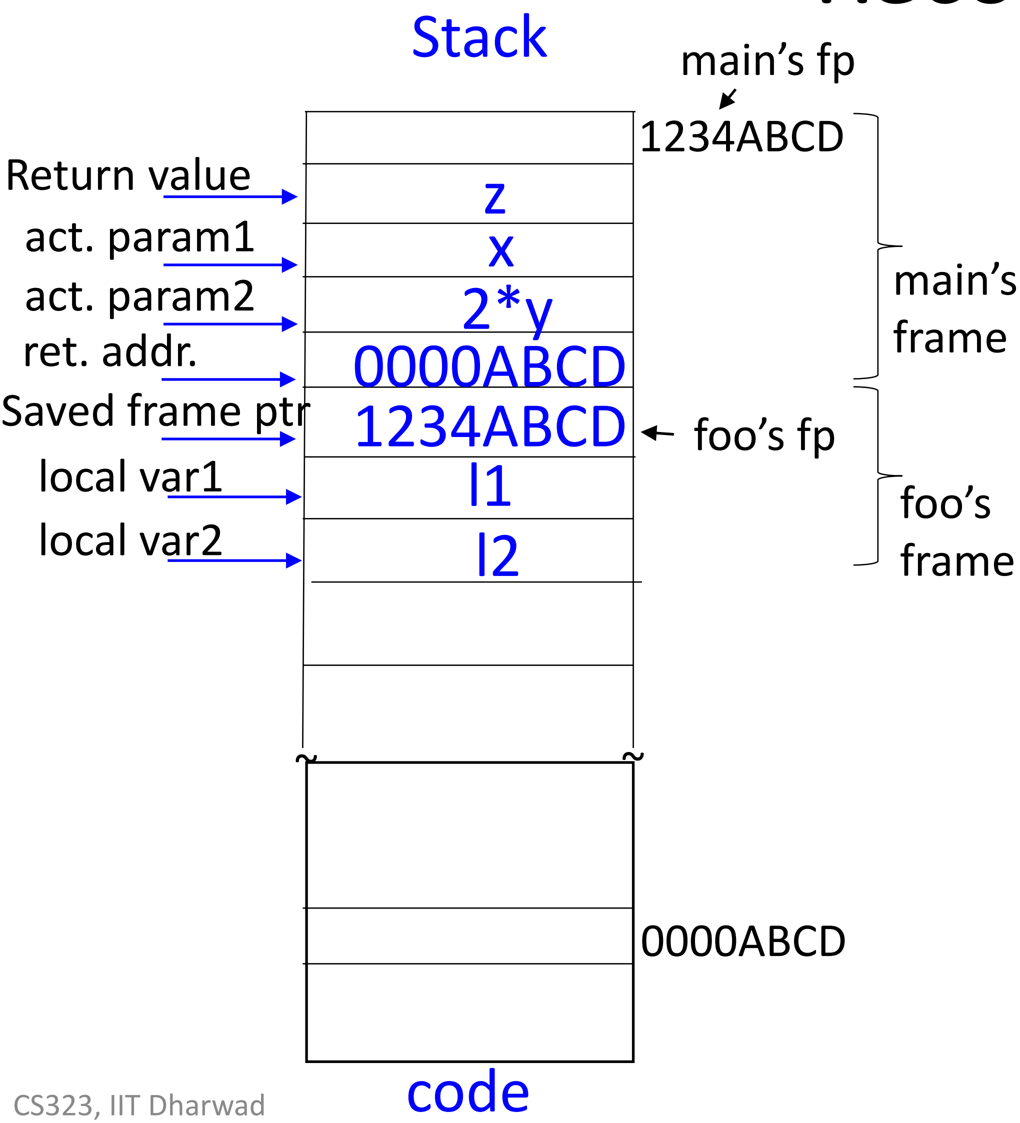
Function call: Peeking at Activation Record




```
main() {
    z=foo(x, 2*y);
    return;
}

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

Function call: Peeking at Activation Record




```
main() {
    z=foo(x, 2*y);
    return;
}

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

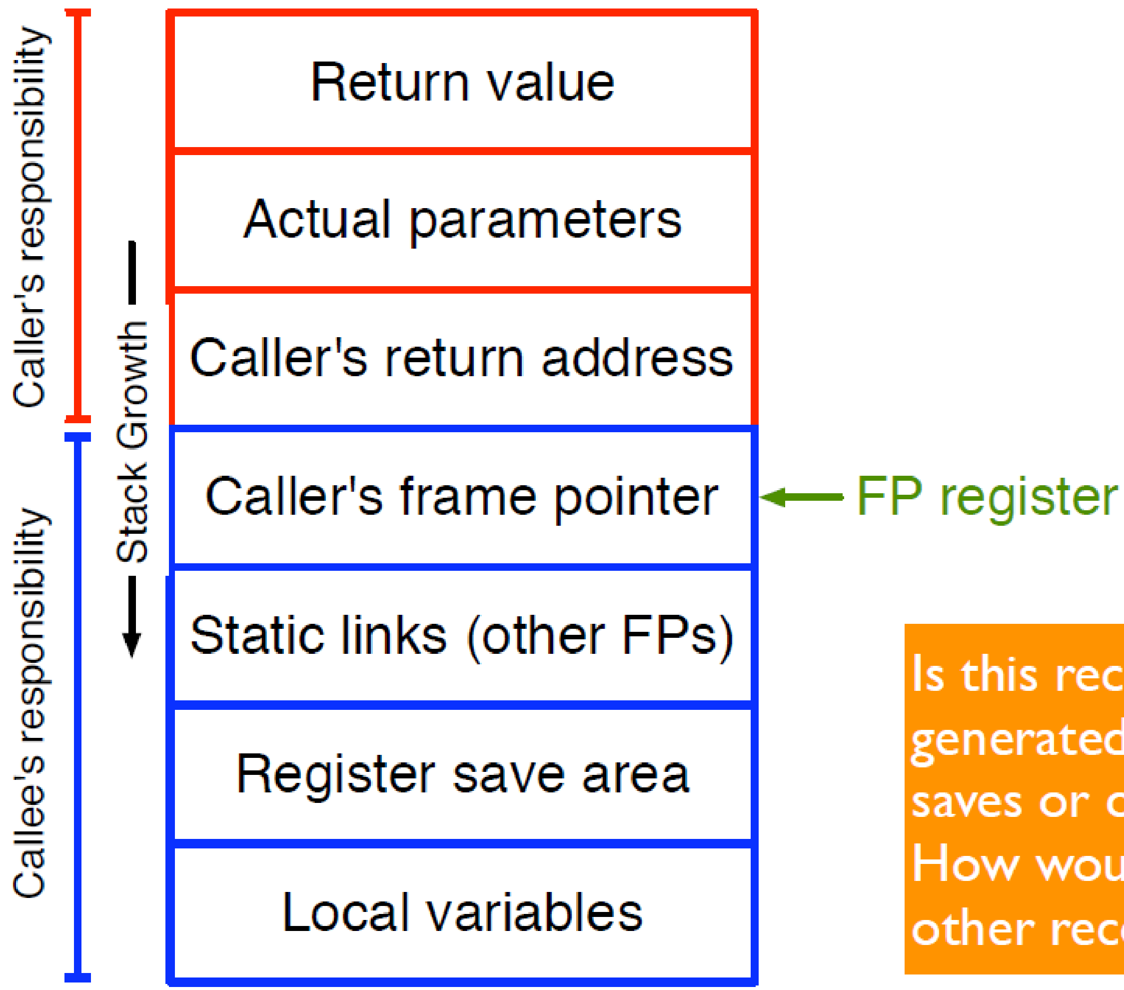
Function calls – Register Handling

- **Did not use registers** in the previous example (for parameter passing)
- Registers are faster than memory. So, compiler should keep parameters in registers whenever possible
- Modern calling convention places first few arguments in registers (arg1 in r1, arg2 in r2, arg3 in r3…) and the remaining in memory.
    - In x86 C-ABI, first 6 arguments are passed in registers
- What if callee wants to use registers r1, r2, r3 etc. for local computation? Callee must save the registers in its stack frame.

Function calls – Register Handling

- Two options: caller saves or callee saves
- Caller Saves
  - Caller pushes all the registers it is using on to the stack before calling the 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

Activation records



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

Activation Record – Return Address and Return Value

- Callee must be able to return to the caller when done
- Return address is the address of the instruction following the function call
- Return address can be placed on the stack or on register
- The `call` instruction on modern machines places the return address in a specific register
- Return value is placed in a specific register by the callee function

This week

- Functions continued
- Local Optimizations

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

Stack Pointer

- SP is manipulated through push and pop instructions

```
Push x:
stack_pointer--
Memory[stack_pointer] = x


Pop x:
x = Memory[stack_pointer]
stack_pointer--
```

Example Subroutine Call and Stack Frame

return value

stack

Lower addr

3-address code:

push

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

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

Question ?

Where are the command-line arguments stored?

How about environment variables such as LD_LIBRARY_PATH and PATH?

Local Optimizations

Naïve approach

- "Macro-expansion"
  - Treat each 3AC instruction separately, generate code in isolation

```
ADD A, B, C   ──────►   LD A, R1
                        LD B, R2
                        ADD R1, R2, R3
                        ST R3, C

MUL A, 4, B   ──────►   LD A, R1
                        MOV 4, R2
                        MUL R1, R2, R3
                        ST R3, B
```

Why is this bad? (I)

MUL A, 4, B → LD A, R1
MOV 4, R2
MUL R1, R2, R3
ST R3, B

MUL A, 4, B → LD A, R1
MULI R1, 4, R3
ST R3, B

There is a better instruction available!

Too many instructions
Should use a different instruction type

Why is this bad? (II)

ADD A, B, C → LD A, R1
LD B, R2
ADD R1, R2, R3
ST R3, C

ADD C, A, E → LD C R4
LD A R5
ADD R4 R5 R6
ST R6 E

This LD is redundant

This LD is redundant

Why is this bad? (III)

ADD A, B, C → LD A, R1
LD B, R2
ADD R1, R2, R3
ST R3, C

ADD A, B, C
ADD A, B, D
→
LD A, R1
LD B, R2
ADD R1, R2, R3
ST R3, C
LD A, R4
LD B, R5
ADD R4, R5, R6
ST R6, D

Wasting instructions recomputing A + B

How do we address this?

- Several techniques to improve performance of generated code
  - *Instruction selection* to choose better instructions
  - *Peephole optimizations* to remove redundant instructions
  - *Common subexpression elimination* to remove redundant computation
  - *Register allocation* to reduce number of registers used

Instruction selection

- Even a simple instruction may have a large set of possible address modes and combinations

+ A B C

- Can be indirect, register, memory address, indexed, etc.
- Can be literal, register, memory address, indexed, etc.
- Can be literal, register, memory address, indexed, etc.

- Dozens of potential combinations!

More choices for instructions

- Auto increment/decrement (especially common in embedded processors as in DSPs)
  - e.g., load from this address and increment it
  - Why is this useful?
- Three-address instructions
- Specialized registers (condition registers, floating point registers, etc.)
- "Free" addition in indexed mode

  MOV (R1)offset R2

  - Why is this useful?

Peephole optimizations

- Simple optimizations that can be performed by pattern matching
  - Intuitively, look through a "peephole" at a small segment of code and replace it with something better
  - Example: if code generator sees `ST R X; LD X R`, eliminate load
- Can recognize sequences of instructions that can be performed by single instructions

  `LDI R1 R2; ADD R1 4 R1` replaced by

  `LDINC R1 R2 4` //load from address in R1 then inc by 4

Peephole optimizations

- Simple optimizations that can be performed by pattern matching
  - Intuitively, look through a "peephole" at a small segment of code and replace it with something better
  - Example: if code generator sees `ST R X; LD X R`, eliminate load

Get the data present at address in R2 and put it in R1

be

`LDI R1 R2; ADD R1 4 R1` replaced by

`LDINC R1 R2 4` //load from address in R1 then inc by 4

Peephole optimizations

- Constant folding

```
ADD lit1, lit2, Rx ⟶ MOV lit1 + lit2, Rx
```

```
MOV lit1, Rx
ADD li2, Rx, Ry      ⟶ MOV lit1 + lit2, Ry
```

- Strength reduction

```
MUL operand, 2, Rx ⟶ SHIFTL operand, 1, Rx
DIV operand, 4, Rx ⟶ SHIFTR operand, 2, Rx
```

- Null sequences

```
MUL operand, 1, Rx ⟶ MOV operand, Rx
ADD operand, 0, Rx ⟶ MOV operand, Rx
```

Peephole optimizations

- Combine operations

```
JEQ L1
JMP L2
L1: ...
```

- Simplifying

```
SUB operand, 0, Rx  ⟶  NEG Rx
```

- Special cases (taking advantage of ++/--)

```
ADD 1, Rx, Rx       ⟶  INC Rx
SUB Rx, 1, Rx       ⟶  DEC Rx
```

- Address mode operations

```
MOV A R1
ADD 0(R1) R2 R3     ⟶  ADD @A R2 R3
```

Superoptimization

- Peephole optimization/instruction selection writ large
- Given a sequence of instructions, find a different sequence of instructions that performs the same computation in less time
- Huge body of research, pulling in ideas from all across computer science
  - Theorem proving
  - Machine learning

Common subexpression elimination

- Goal: remove redundant computation, don't calculate the same expression multiple times

1: A = B * C
2: E = B * C

Keep the result of statement 1 in a temporary and reuse for statement 2

- Difficulty: how do we know when the same expression will produce the same result?

1: A = B * C
2: B = <new value>
3: E = B * C

B is "killed." Any expression using B is no longer "available," so we cannot reuse the result of statement 1 for statement 3

- This becomes harder with pointers (how do we know when B is killed?)

Common subexpression elimination

- Two varieties of common subexpression elimination (CSE)
- Local: within a single basic block ➡ Maximal sequence of instructions that are executed one after another (i.e. there are no jump instructions OR no instruction is the target of a jump)
  - Easier problem to solve (why?)
- Global: within a single procedure or across the whole program
  - Intra- vs. inter-procedural
  - More powerful, but harder (why?)
  - Will come back to these sorts of "global" optimizations later

Local optimizations are done on basic blocks. Global optimizations on control flow graphs (CFGs), where the basic blocks are the nodes of the graph. Then, there are inter-procedural optimizations, which span function calls. Later on CFGs and other kinds of optimizations.

CSE in practice

- Idea: keep track of which expressions are "available" during the execution of a basic block
  - Which expressions have we already computed?
  - Issue: determining when an expression is no longer available
    - This happens when one of its components is assigned to, or "killed."
- Idea: when we see an expression that is already available, rather than generating code, copy the temporary
  - Issue: determining when two expressions are the same

Maintaining available expressions

- For each 3AC operation in a basic block
  - Create name for expression (based on lexical representation)
  - If name not in available expression set, generate code, add it to set
    - Track register that holds result of and any variables used to compute expression
  - If name in available expression set, generate move instruction
  - If operation assigns to a variable, kill all dependent expressions

Example

3 Address Code	Available expression(s)	Killed expression(s)	Generated Code (assembly)
			ld a r1; ld b r2;
ADD A B T1	{}		add r1 r2 r1
ADD T1 C T2	{“A + B”}		add r1 c r2
ADD A B T3	{“A + B”, “T1 + C”}		mov r1 r3
ADD T1 T2 C	{“A + B”, ~~“T1 + C”}~~	{“T1+C”}	add r1 r2 r5 st r5 c
ADD T1 C T4	{“A + B”, “T1 + T2”}		add r1 c r4
ADD T3 T2 D	{“A + B”, “T1 + T2”, “T1 + C”}		add r3 r2 r6 st r6 d
	{“A + B”, “T1 + T2”, “T1 + C”, “T3 + T2”}		

Downsides (CSE)

- What are some downsides to this approach? Consider the two highlighted operations

Three address code

```
+ A B T1
+ T1 C T2
+ A B T3
+ T1 T2 C
+ T1 C T4
+ T3 T2 D
```

Generated code

```
ADD A B R1
ADD R1 C R2
MOV R1 R3
ADD R1 R2 R5; ST R5 C
ADD R1 C R4
ST R5 D
```

T1 and T3 compute the same expression. This can be handled by an optimization called *value numbering.*

Aliasing

- One of the biggest problems in compiler analysis is to recognize aliases – different names for the same location in memory

exercise: are T1 and T3 aliased in previous example?

- Why do aliases occur?
    - Pointers referring to the same location
    - Function calls passing the same reference in two arguments
    - Arrays referencing the same element
    - Unions
- What problems does aliasing pose for CSE?
    - when talking about "live" and "killed" values in optimizations like CSE, we're talking about particular variable names
    - In the presence of aliasing, we may not know which variables get killed when a location is written to

Memory disambiguation

- Most compiler analyses rely on *memory disambiguation*
  - Otherwise, they need to be too conservative and are not useful
- Memory disambiguation is the problem of determining whether two references point to the same memory location
  - *Points-to* and *alias* analyses try to solve this
  - Will cover basic pointer analyses in a later lecture

Single assignment form and its use in local optimizations

x=z+y
a=x
x=2*x

replace x with b

b=z+y
a=b
x=2*b

Single assignment form: a variable is assigned only once i.e. appears only once in LHS.

Aids CSE:

x=z+y
. . .
x=z+y

Neither z nor y can appear on the LHS here in single assignment form.

So, can be sure that this z+y is the same expression as earlier. In the original code, if z or y were assigned to in between the two expressions, then we would have used different names, say, z1=..; y1=; then the last expression would have to be rewritten as x=z1+y1.

Aids copy propagation: can replace all the uses of a variable downstream

Aids dead code elimination: if the variable is never used later, can safely remove the statement where the variable is defined/assigned to.

Example – Local Optimizations

```
a=x**2
b=3
c=x
d=c*c
e=b*2
f=a+d
g=e*f
```

Example – Local Optimizations

- Algebraic simplification – exploiting mathematical properties of operators involved

```
a=x*x
b=3
c=x
d=c*c
e=b<<1
f=a+d
g=e*f
```

Example – Local Optimizations

- Copy and constant propagation

```
a=x*x
b=3
c=x
d=c*c
e=b<<1
f=a+d
g=e*f
```

```
a=x*x
b=3
c=x
d=x*x
e=3<<1
f=a+d
g=e*f
```

Example – Local Optimizations

- Constant folding

```
a=x*x
b=3
c=x
d=c*c
e=b<<1
f=a+d
g=e*f
```

⇨

```
a=x*x
b=3
c=x
d=x*x
e=3<<1
f=a+d
g=e*f
```

⇨

```
a=x*x
b=3
c=x
d=x*x
e=6
f=a+d
g=e*f
```

Example – Local Optimizations

- CSE

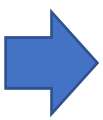
```
a=x*x
b=3
c=x
d=x*x
e=6
f=a+d
g=e*f
```

```
a=x*x
b=3
c=x
d=a
e=6
f=a+d
g=e*f
```

Example – Local Optimizations

- Copy and Constant Propagation

```
a=x*x
b=3
c=x
d=a
e=6
f=a+d
g=e*f
```



```
a=x*x
b=3
c=x
d=a
e=6
f=a+a
g=6*f
```

Example – Local Optimizations

- Dead code elimination

```
a=x*x
b=3
c=x
d=a
e=6
f=a+a
g=6*f
```

⇒

```
a=x*x
f=a+a
g=6*f
```

Anything else?

```
a=x*x
f=2*a
g=6*f
```

⇒

a=x*x
~~f=2*a~~
g=12*a