

CS620T: Compilers - Principles and Implementation

Autumn 2026

Week 6: Semantic processing (contd.) :
Intermediate Representation (for Functions),
Local Optimizations

Functions

- **Typical Syntax and Usage**

```
FUNCTION VOID bar(INT x, FLOAT y) BEGIN
```

```
END
```

Keywords

Return type

comma separated parameter declarations.

Declarations (string or variable decl) followed by statement declarations

```
FUNCTION void foo() BEGIN
```

```
INT a;
```

```
FLOAT b;
```

```
...
```

```
bar(a, b);
```

← Calls function bar

```
END
```

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
- Reference
- Result
- Value-Reference
- Read-only
- Call-by-Name

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?

Advantage: ‘side-effect’ free – caller can be sure that the argument is not modified by the callee

Disadvantage: Not efficient for larger sized arguments.

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?

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?

Advantage: Efficiency – for larger sized arguments

Disadvantage: results in clumsy code at times (e.g. check for null pointers)

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?

Result Parameters

- To capture the return value of a function
- Copied at the end of function into arguments of the caller
- E.g. output ports in Verilog module definitions

Result Parameters

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

- What do the print statements print?

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

Value-Result Parameters

- “Copy-in copy-out”
- Evaluate argument expression, copy to parameters
- After subroutine is done, copy values of parameters back into arguments
- Results are often similar to pass-by-reference, but there are some subtle situations where they are different

Value-Result Parameters

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

- What do the print statements print?

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

Read-only Parameters

- Used when callee will not change value of parameters
- Read-only restriction must be enforced by compiler
- E.g. `const` parameter in C/C++
- Enforcing becomes tricky when in the presence of aliasing and control flow. E.g.

```
void foo(readonly int x, int y) {  
    int * p;  
    if (...) p = &x else p = &y  
    *p = 4  
}
```

Call-by-name Parameters

- The arguments are passed to the function before evaluation
 - Usually, we evaluate the arguments before passing them
- Not used in many languages, but Haskell uses a variant

```
int x = 1
void main () {
    foo(x+2);
    print(x);
}
```

```
void foo(int y) {
    z = y + 3; //expands to z = x + 2 + 3
    print(z);
}
```

Call-by-name Parameters

- Why is this useful?
 - E.g. to analyze certain properties of a program/function – termination

```
void main () {  
    foo(bar());  
}
```

```
void foo(int y) {  
    z = 3;  
    if(z > 3)  
        z = y + z;  
}
```

- Even if bar has an infinite loop, the program terminates.

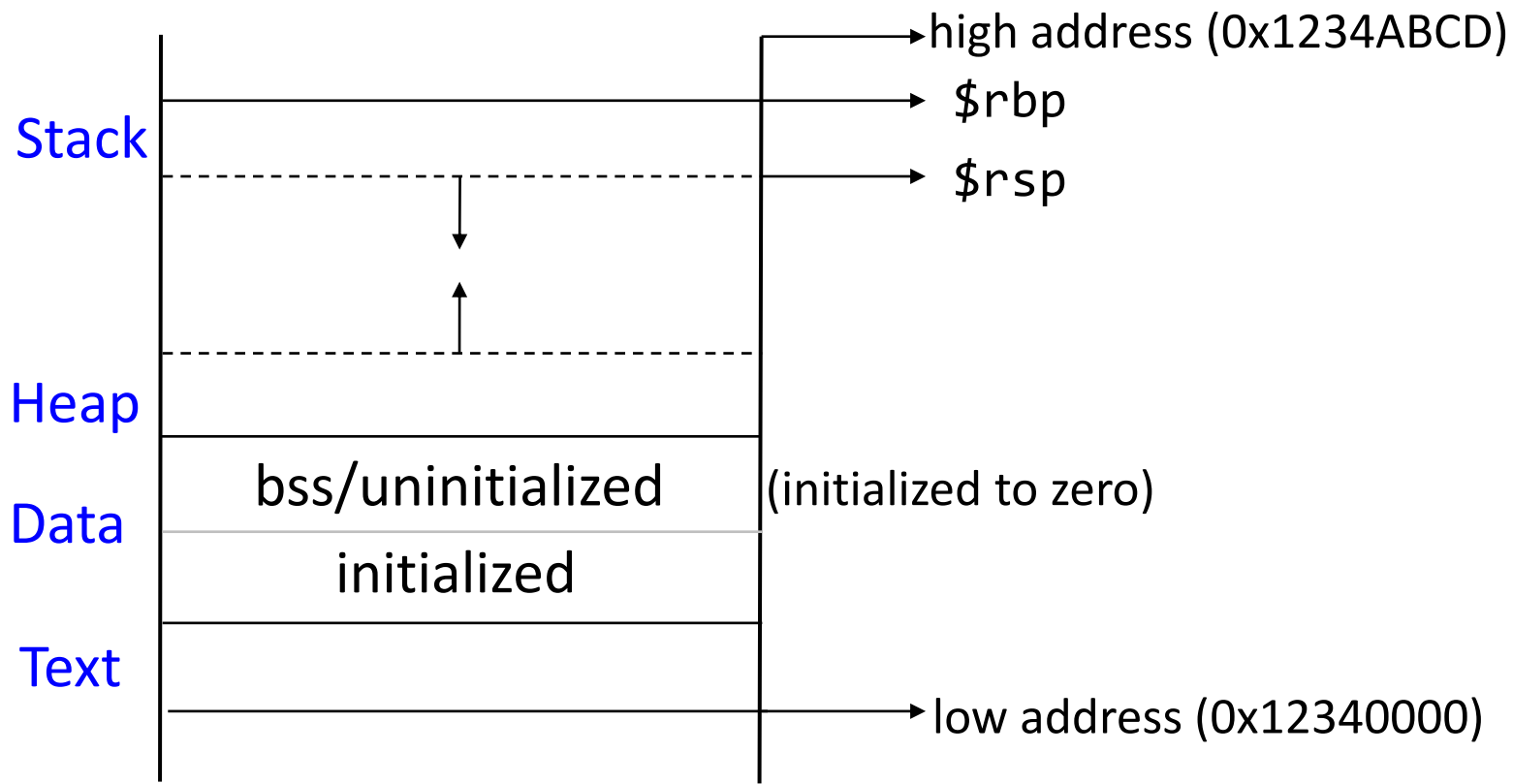
Program Layout in Memory

- Compiler assumes a *runtime environment* for execution of the program.
- A C/C++ program in Linux OS has 4 segments of memory
 - Every memory location is a *box* holding *data/instruction*

Program Layout in Memory

- A program's memory space is divided into four segments:
 1. Text
 - source code of the program
 2. Data
 - Broken into *uninitialized* and *initialized* segments; contains space for global and static variables. E.g. `int x = 7; int y;`
 3. Heap
 - Memory allocated using `malloc/calloc/realloc/new`
 4. Stack
 - Function arguments, return values, local variables, [special registers](#).

Program Layout in Memory



Activation

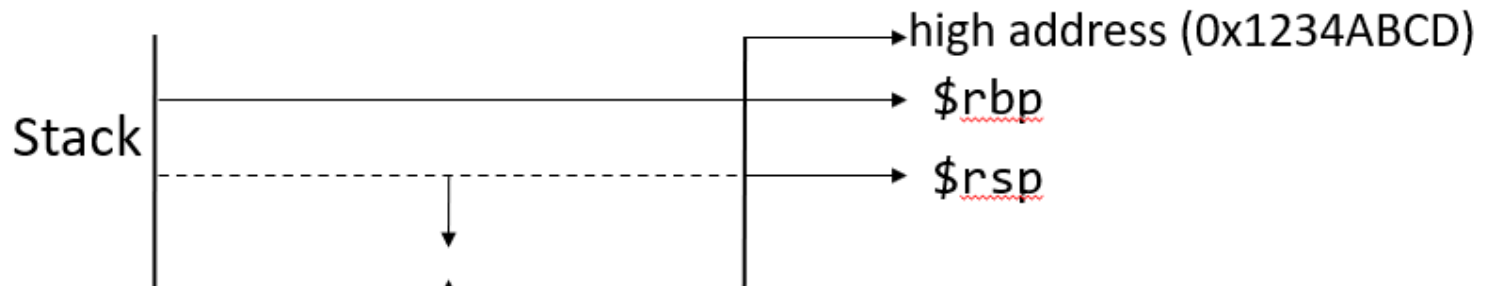
- A function call or invocation is termed an *activation*
- Calls to functions in a program form *activation tree*
 - Postorder traversal of the tree shows return sequence i.e. the order in which control returns from functions
 - Preorder traversal of the tree shows calling sequence
- In a sequential 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*)

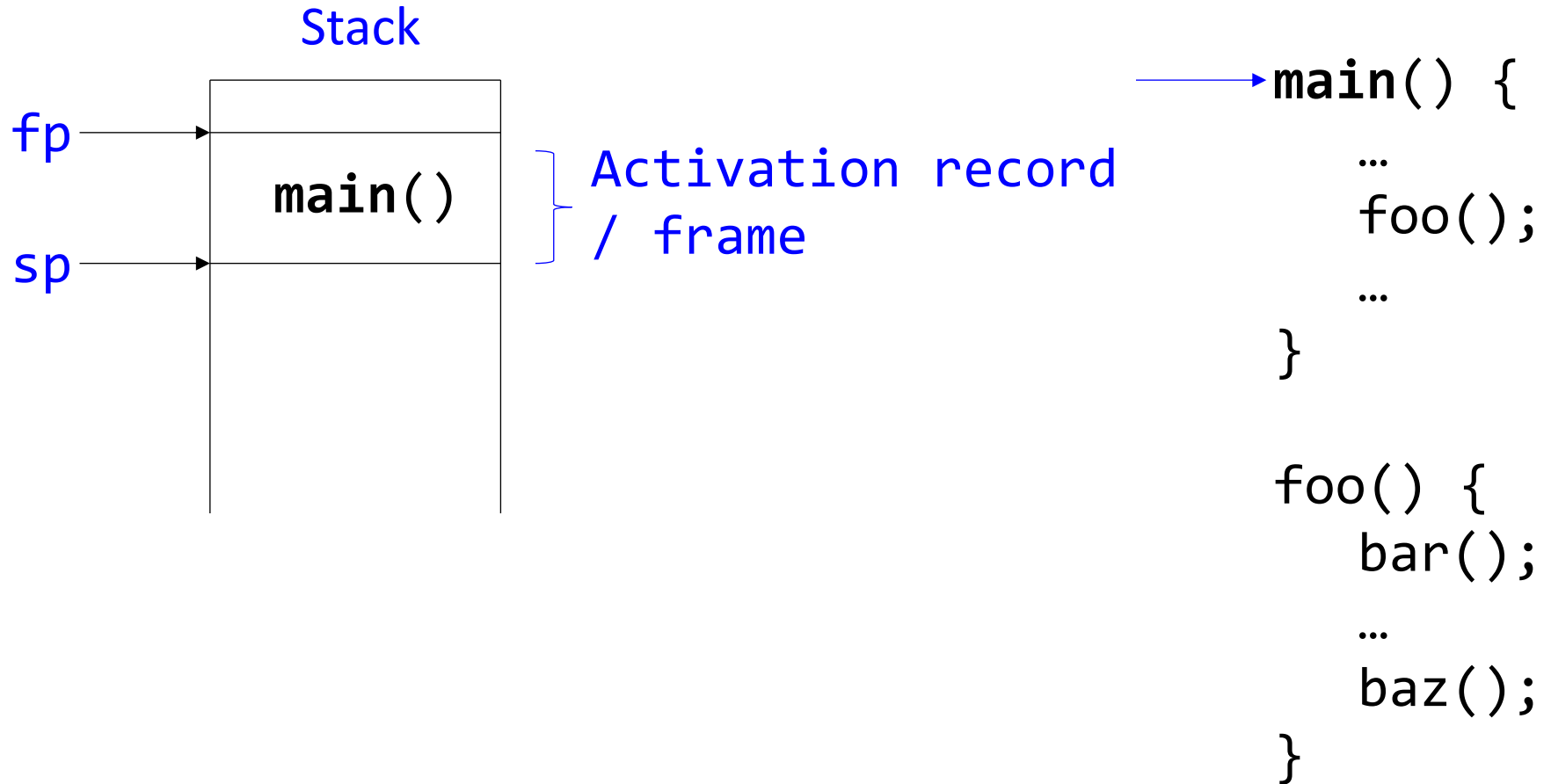
Activation Record

- 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.

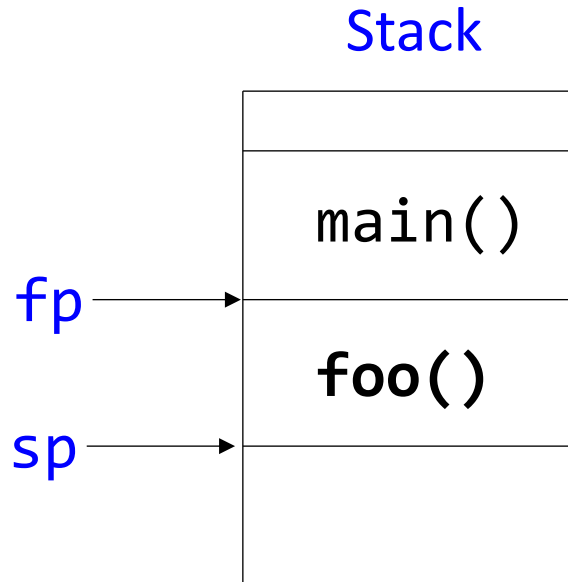


- `$rbp` – base pointer or frame pointer (**fp**)
- `$rsp` – stack pointer (**sp**)

Activation Record - Example



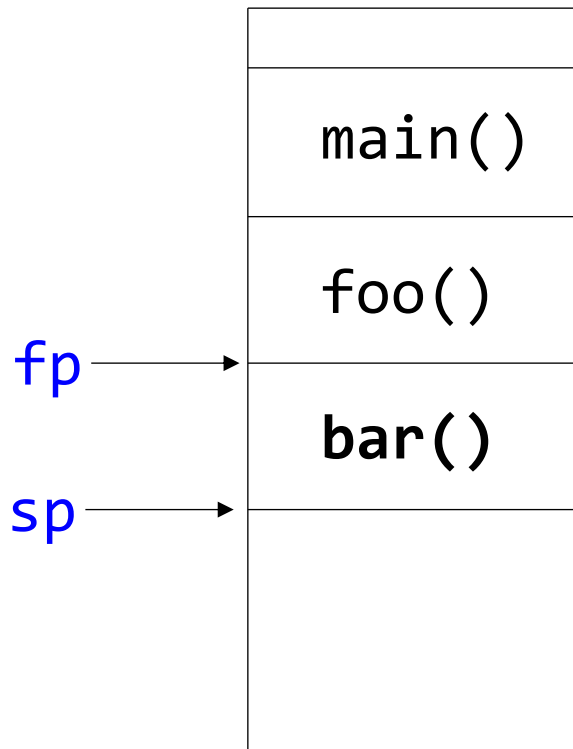
Activation Record - Example



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

Activation Record - Example

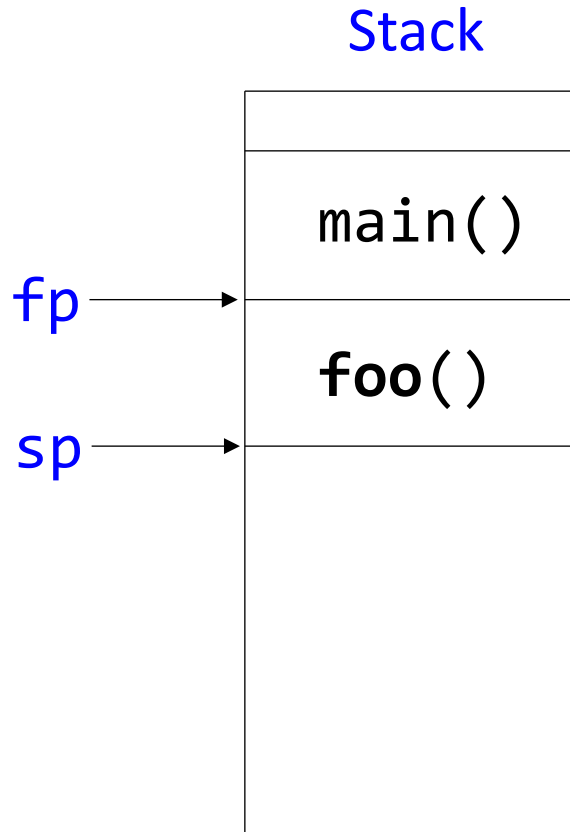
Stack



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

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

Activation Record - Example



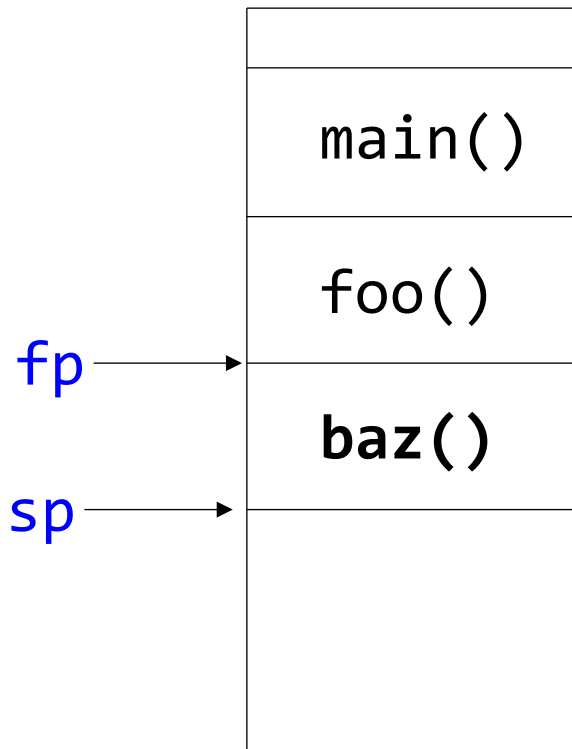
```
main() {  
    ...  
    foo();  
    ...  
}
```

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



Activation Record - Example

Stack



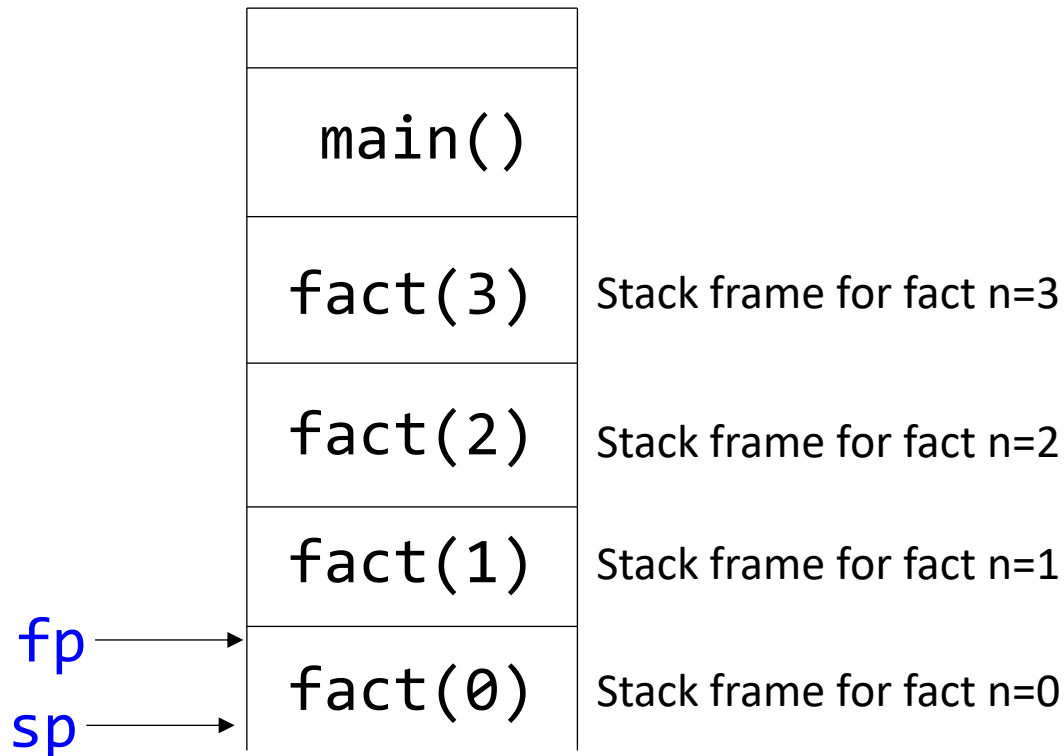
```
main() {  
    ...  
    foo();  
    ...  
}
```

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

```
    → baz();  
}
```

Activation Record – Example (Recursive Functions)

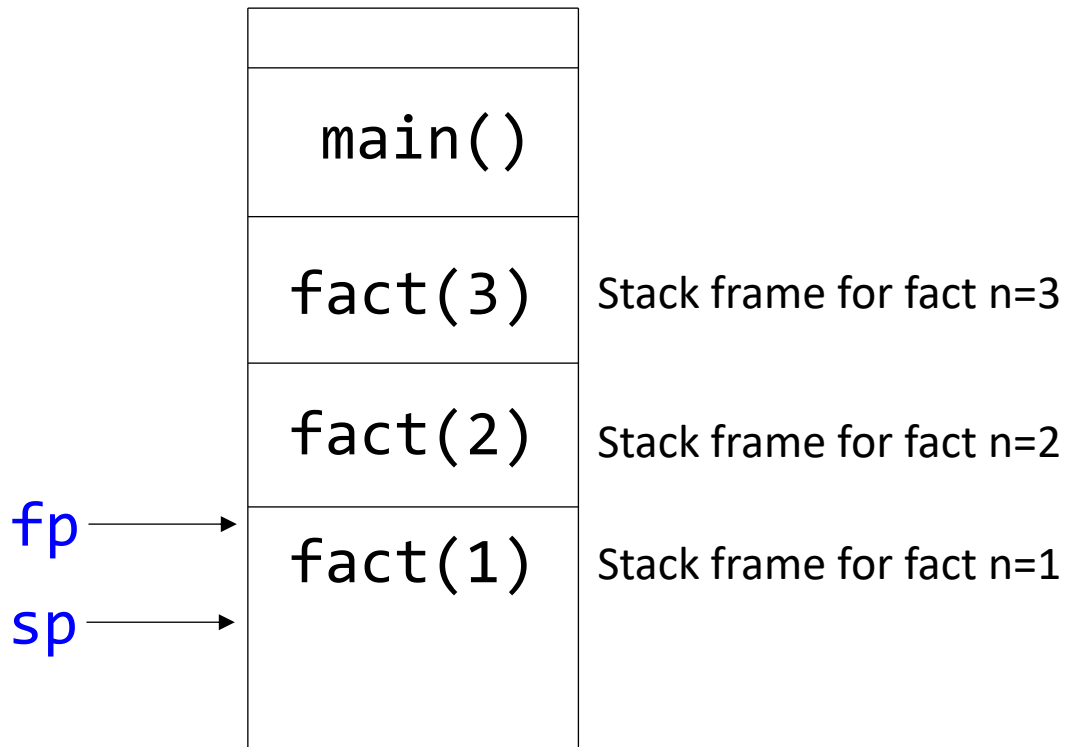
Stack



```
main() {  
    ...  
    fact(3);  
    ...  
}  
  
fact(int n) {  
    if (n=0) return 1  
    return n*fact(n-1)  
}
```

Activation Record – Example (Recursive Functions)

Stack



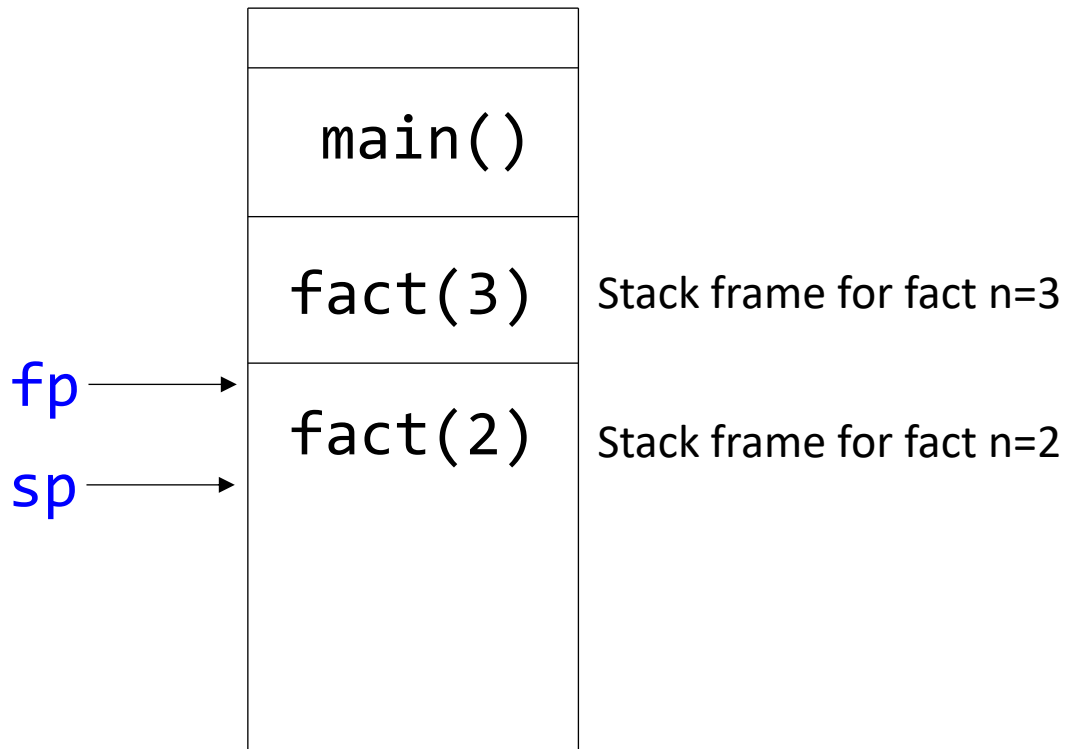
```
main() {  
    ...  
    fact(3);  
    ...  
}
```

```
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)

Stack



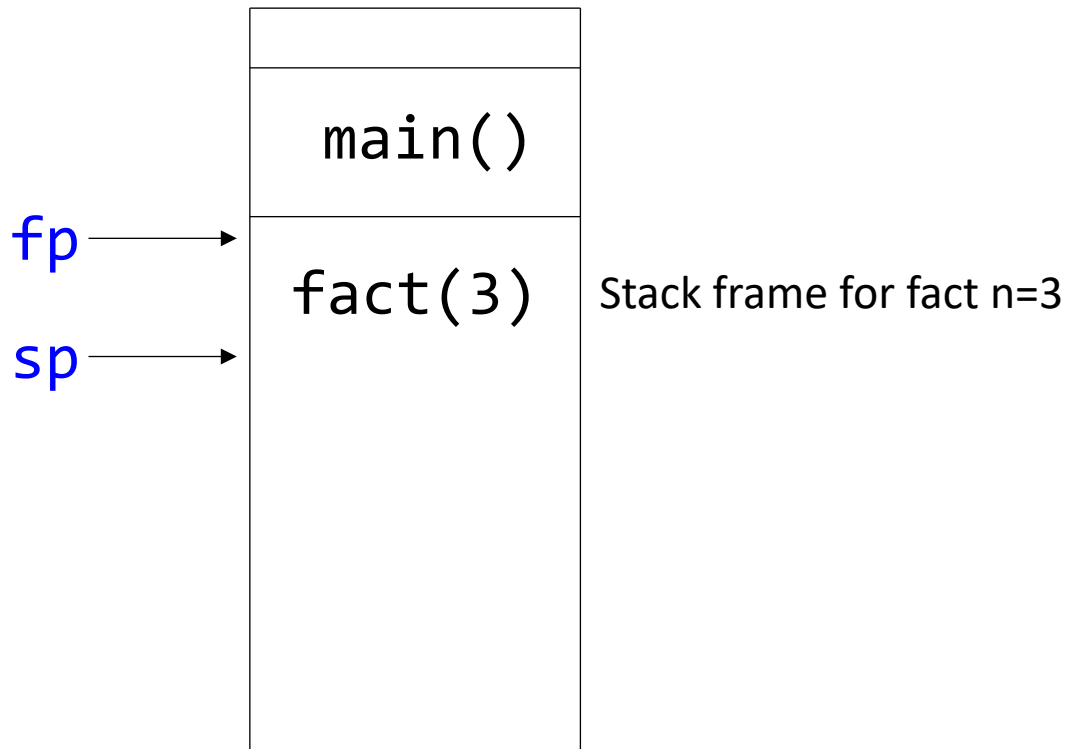
```
main() {  
    ...  
    fact(3);  
    ...  
}
```

```
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)

Stack

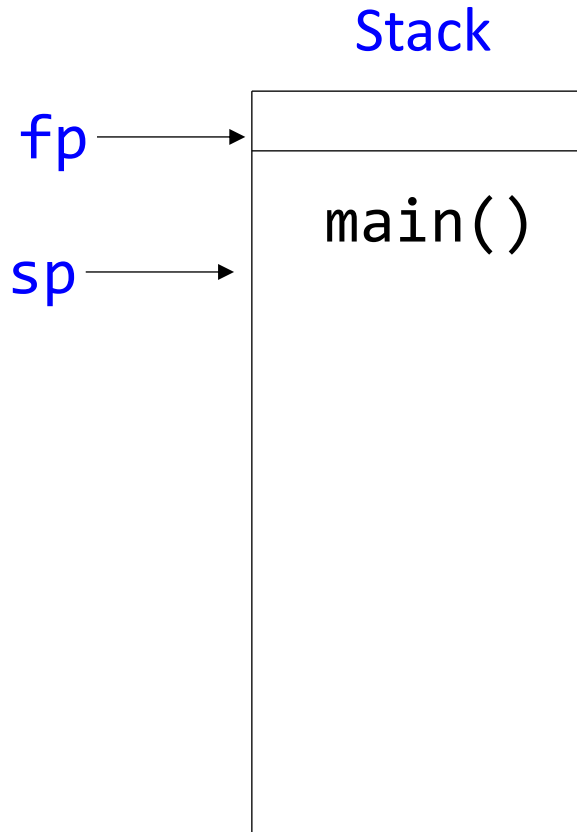


```
main() {  
    ...  
    fact(3);  
    ...  
}
```

```
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:

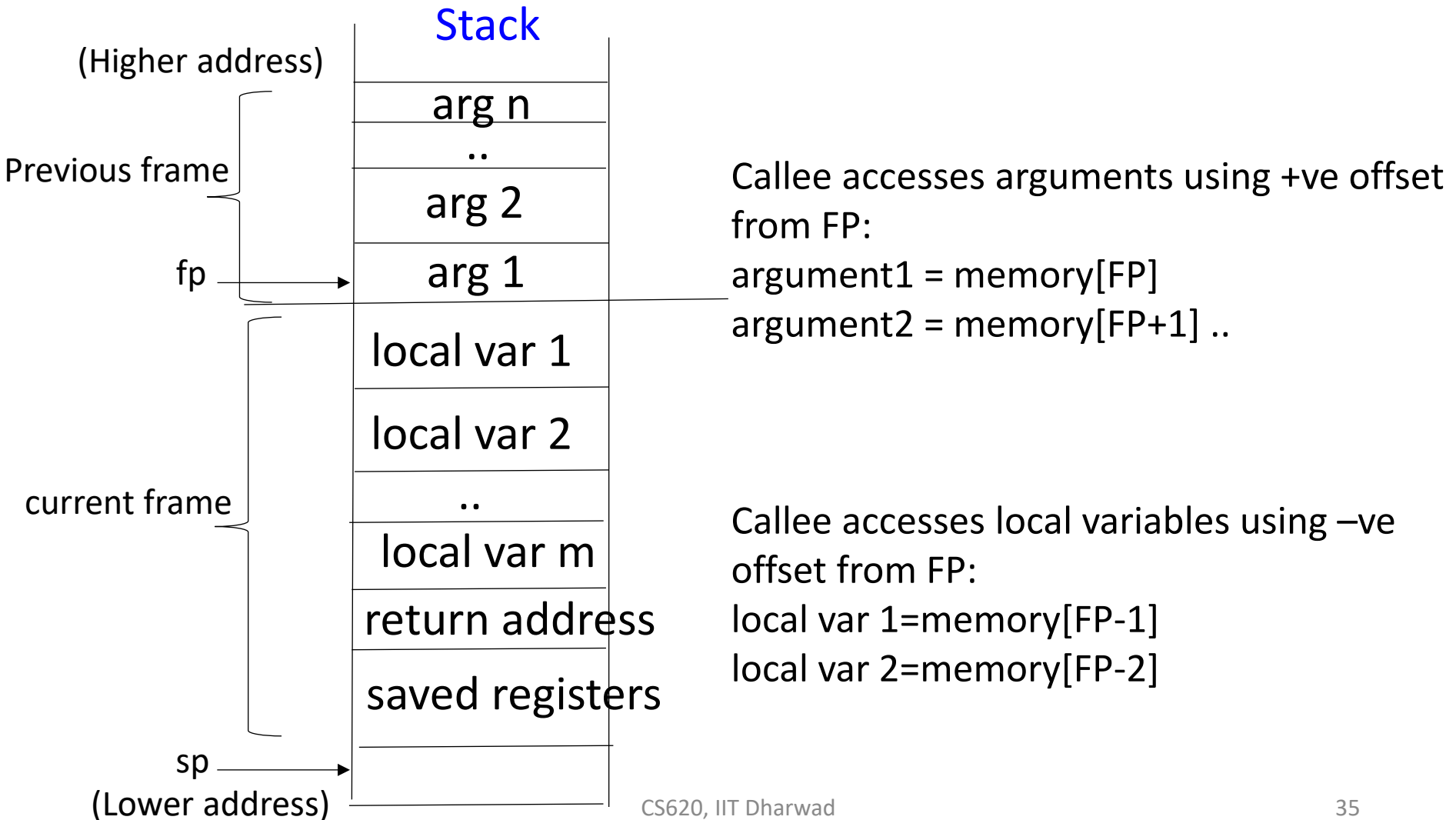
- Temporaries
 - Local vars
 - Saved registers
 - Return address, previous fp
 - Return value
 - Actual Params
- Who stores this information?
 - Caller } together execute *calling sequence* and *return*
 - Callee } *sequence*

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



Function call: Peeking at Activation Record

- When `main` calls function `foo`

1. The following are pushed on to the stack:

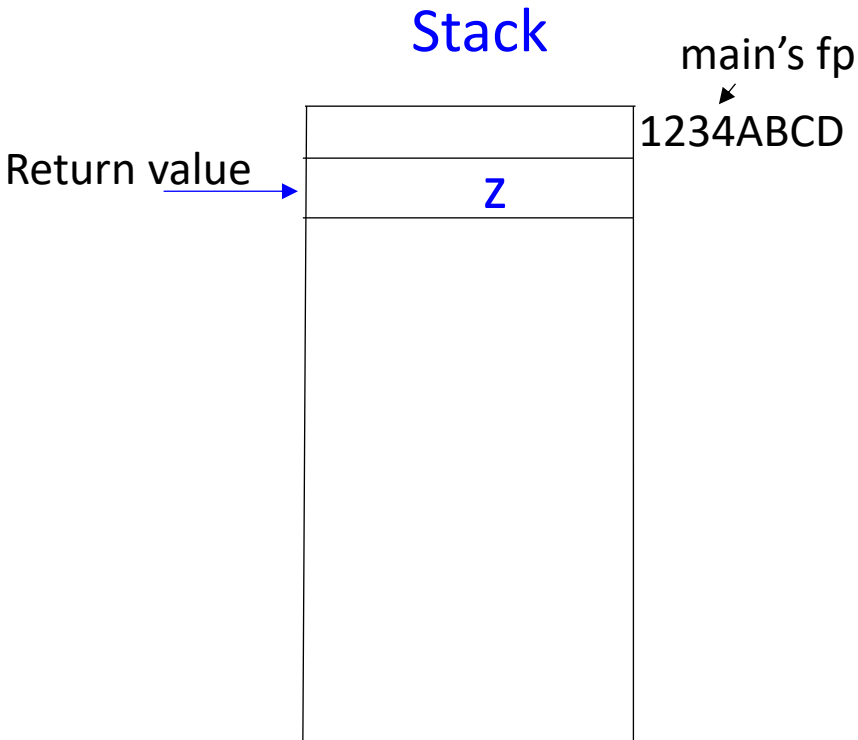
1. `foo`'s **arguments**
2. Space to hold `foo`'s **return value**
3. Address of the next instruction executed (in `main`) when `foo` returns (**return address**)
4. Current value of `$rbp` (**frame pointer**)

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

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

2. `$rbp` is assigned the value of `$rsp`

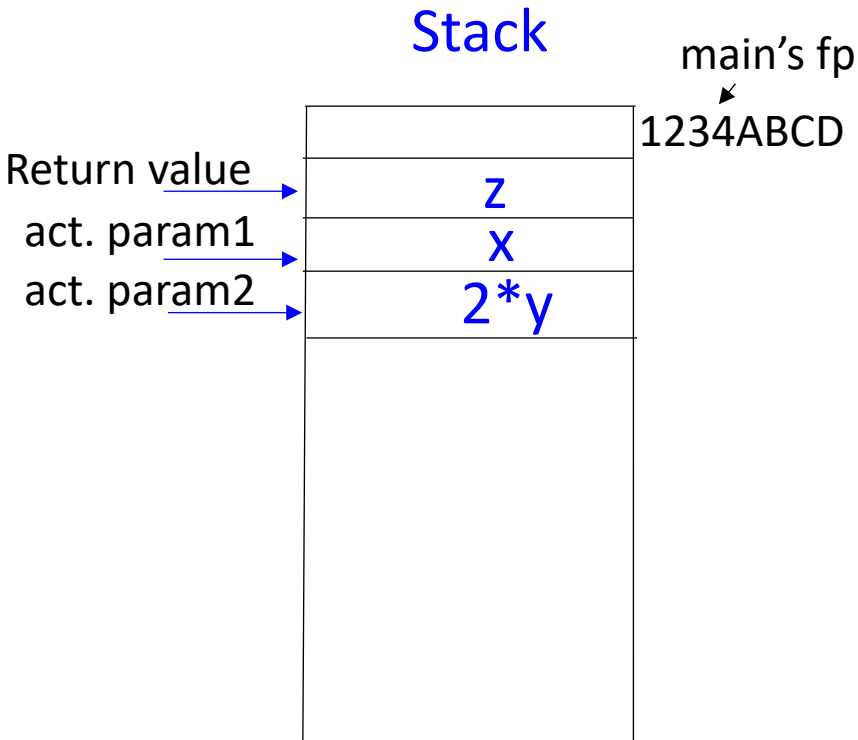
Function call: Peeking at Activation Record



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

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

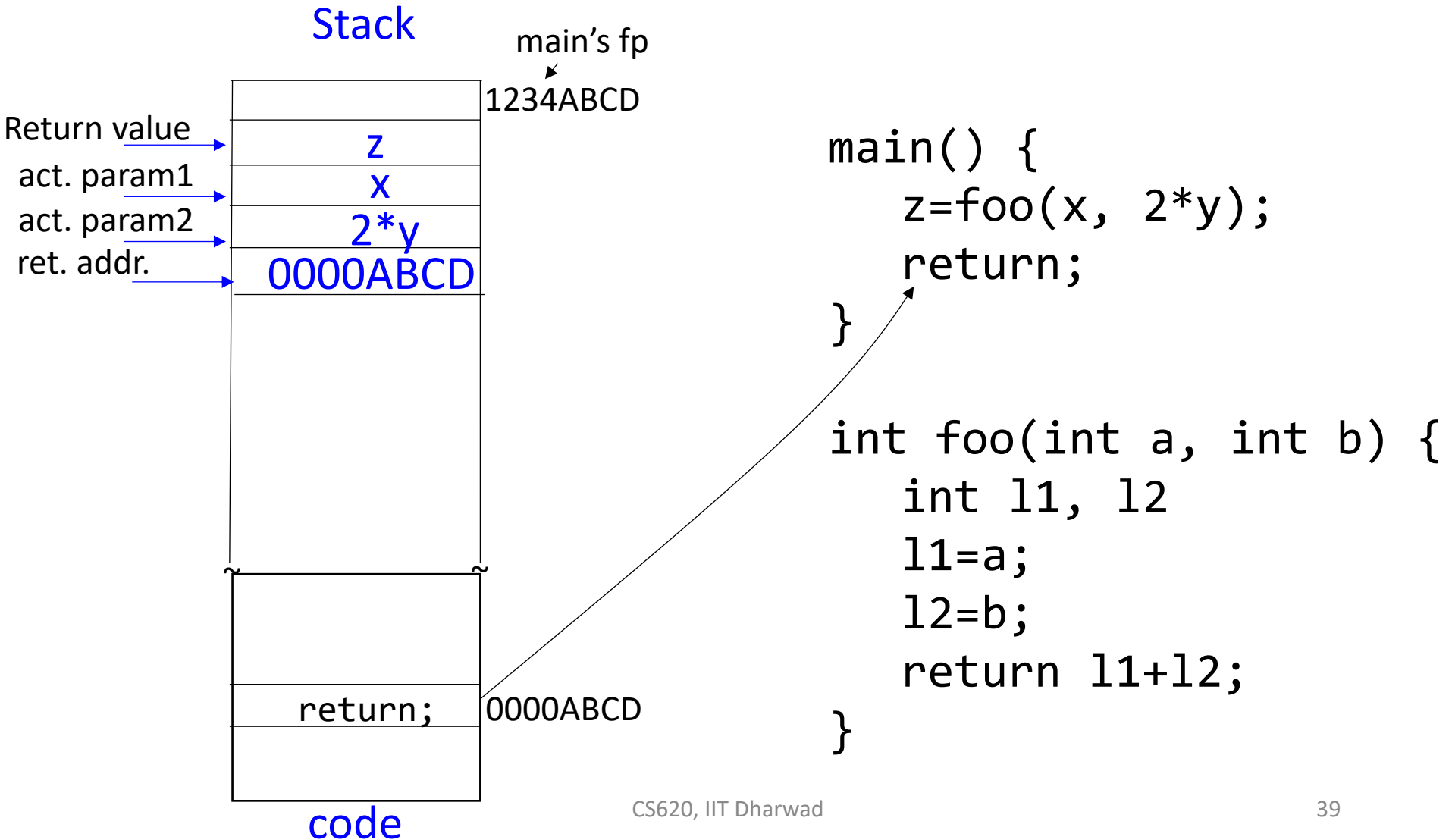
Function call: Peeking at Activation Record



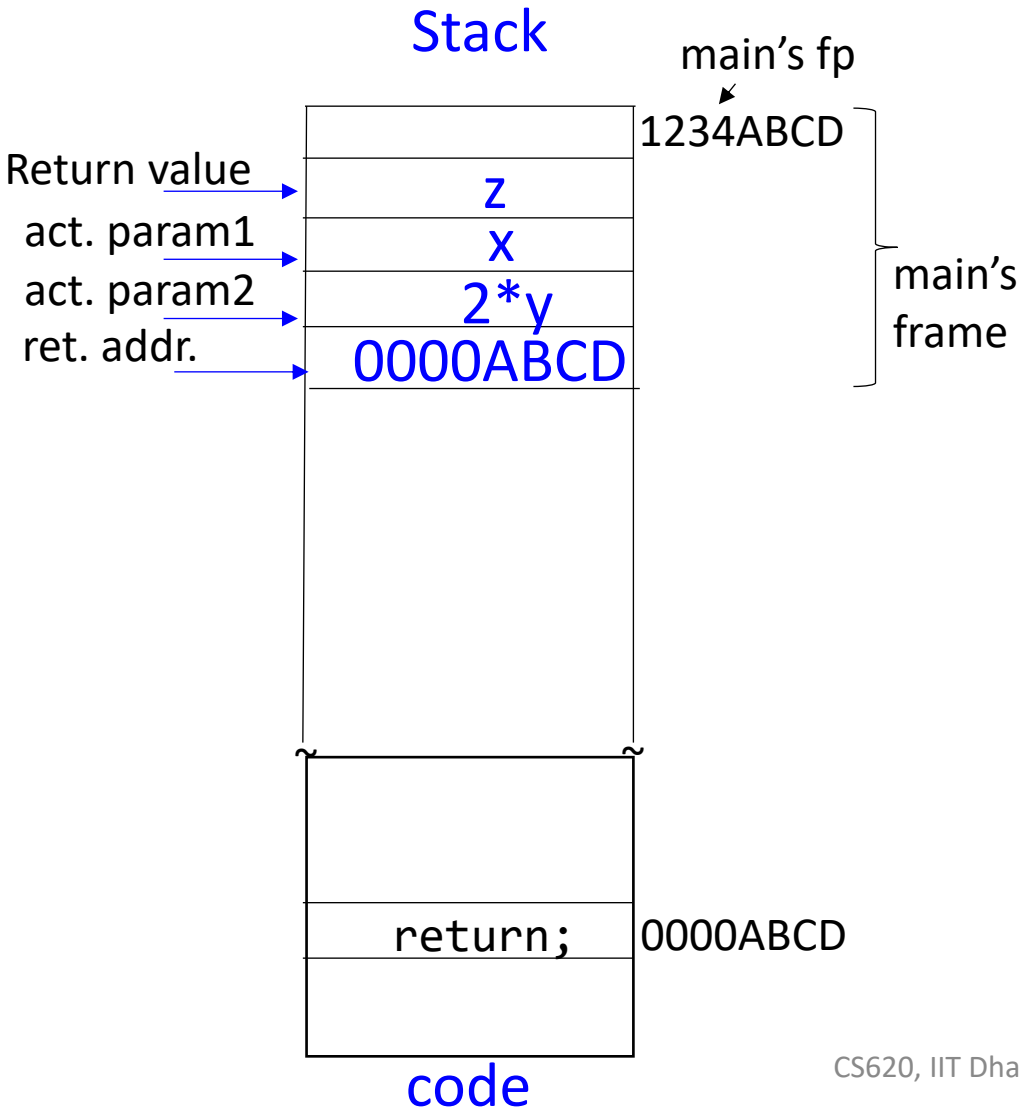
```
main() {  
    z=foo(x, 2*y);  
    return;  
}
```

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

Function call: Peeking at Activation Record



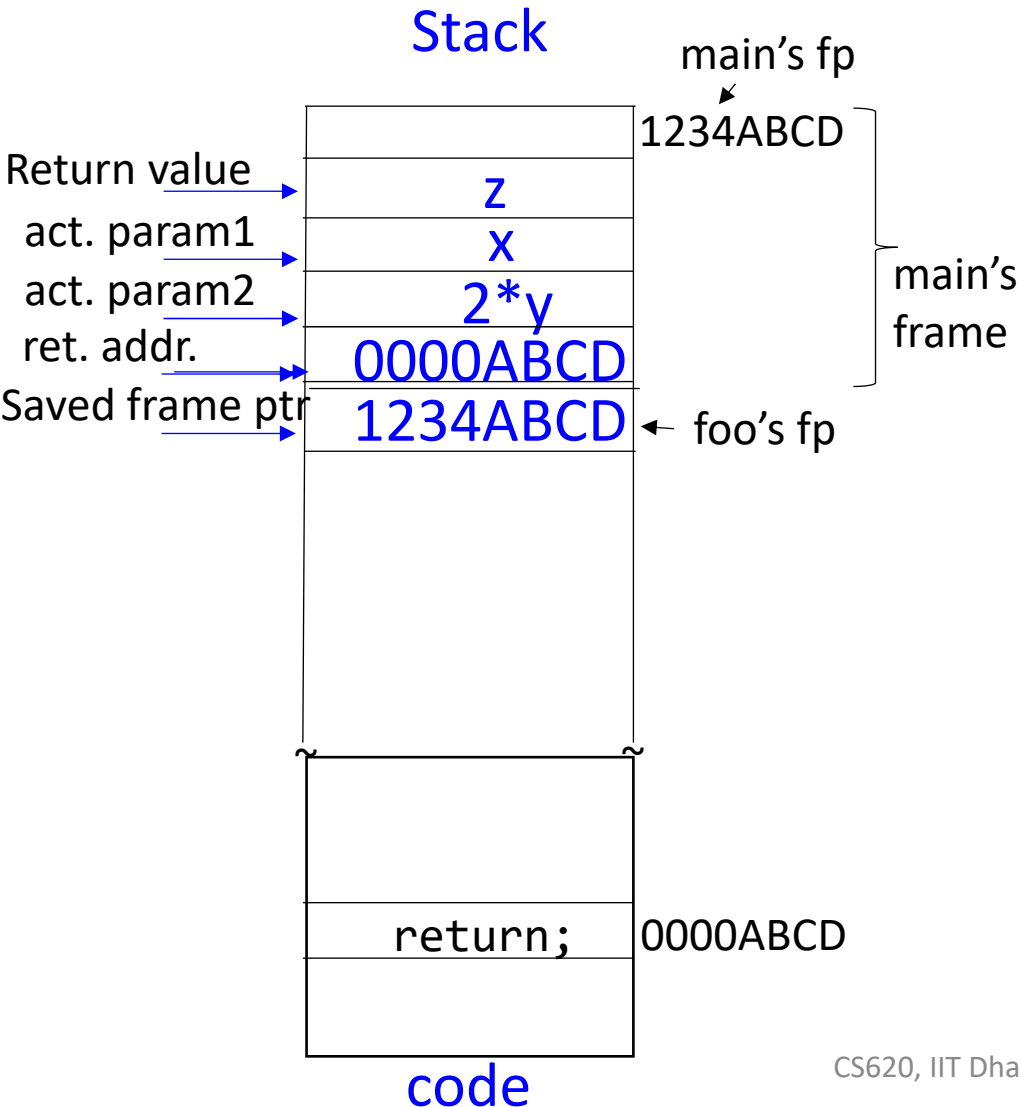
Function call: Peeking at Activation Record



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

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

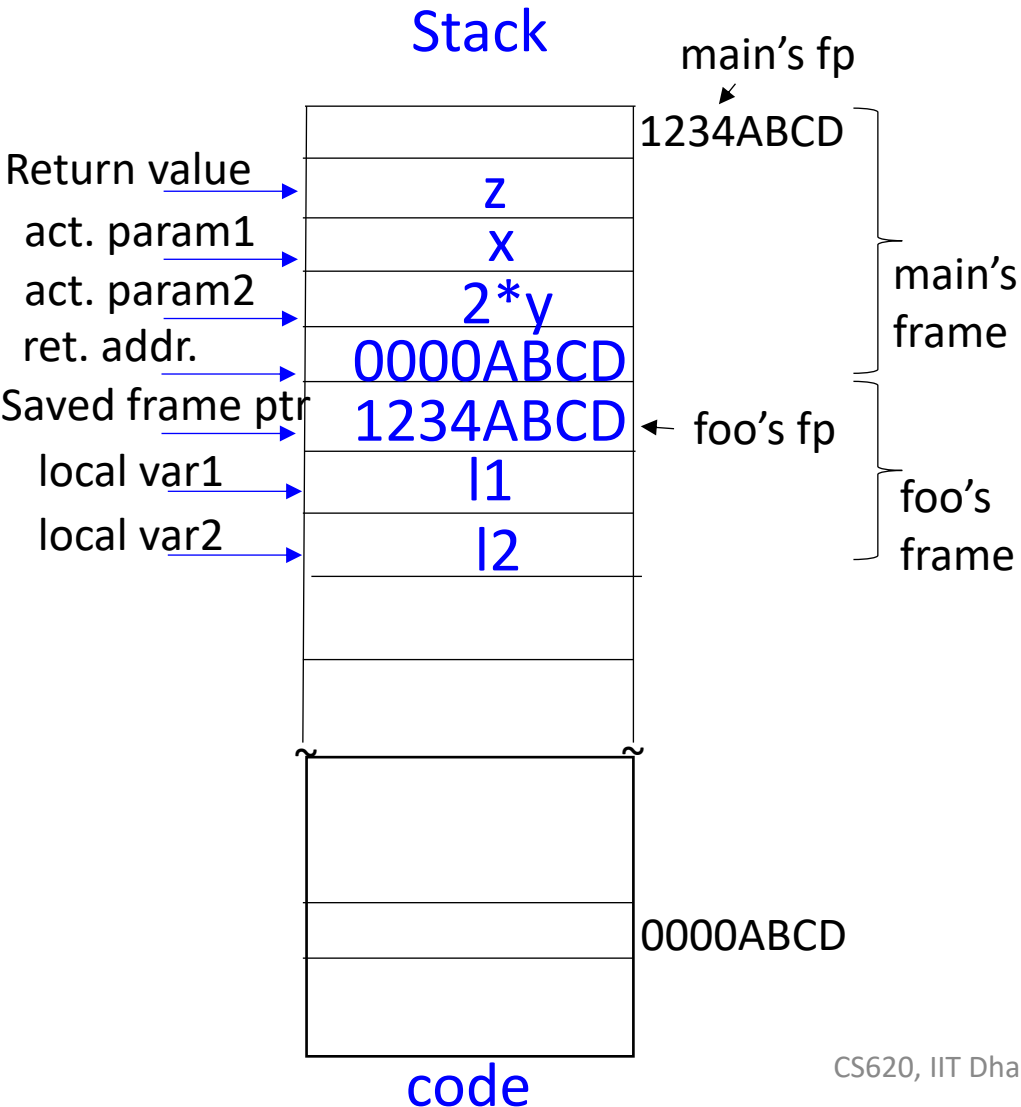
Function call: Peeking at Activation Record



```
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;
}
```

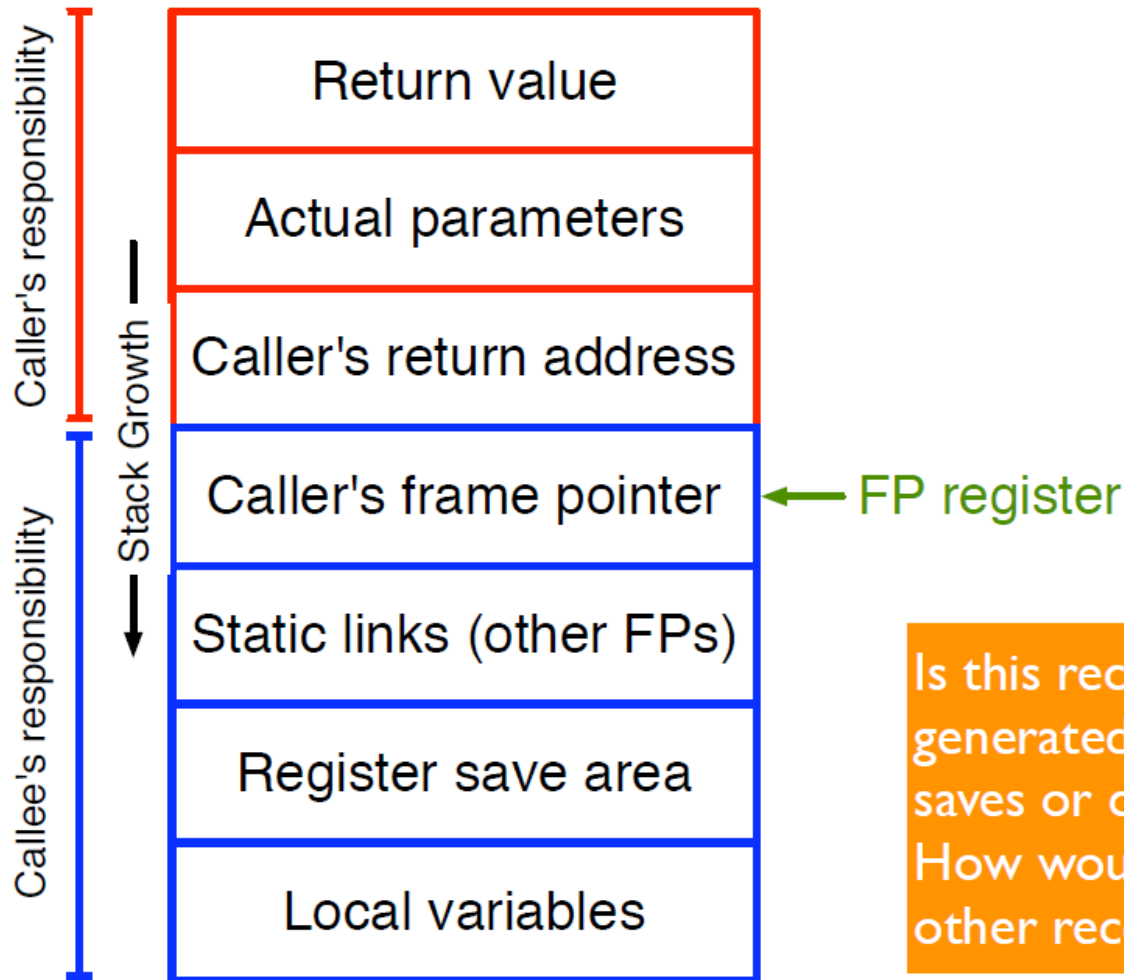
Function calls – Register Handling

- **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

- 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

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

3-address code:

push

stack

Lower addr

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

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

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)

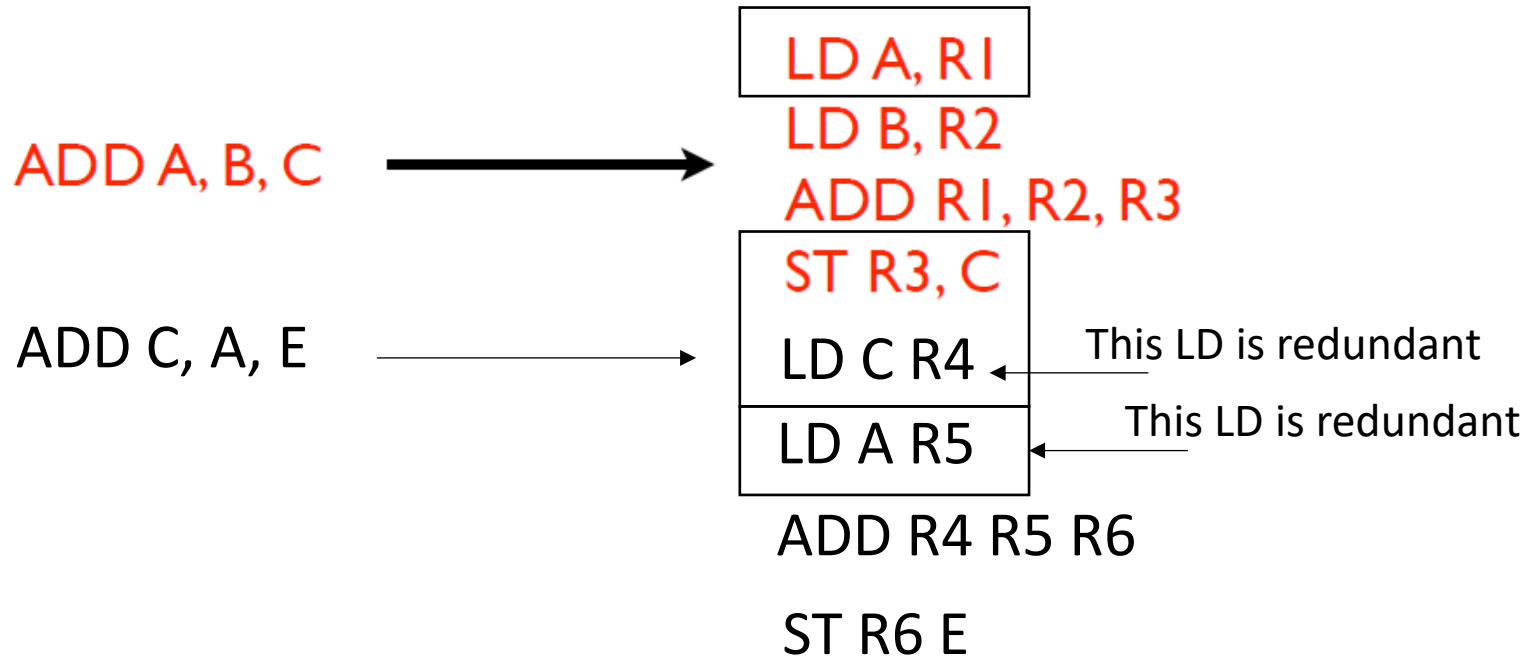
MUL A, 4, B →
LDA A, R1
MOV 4, R2
MUL R1, R2, R3
ST R3, B

MUL A, 4, B →
LDA 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)



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 $A + B$

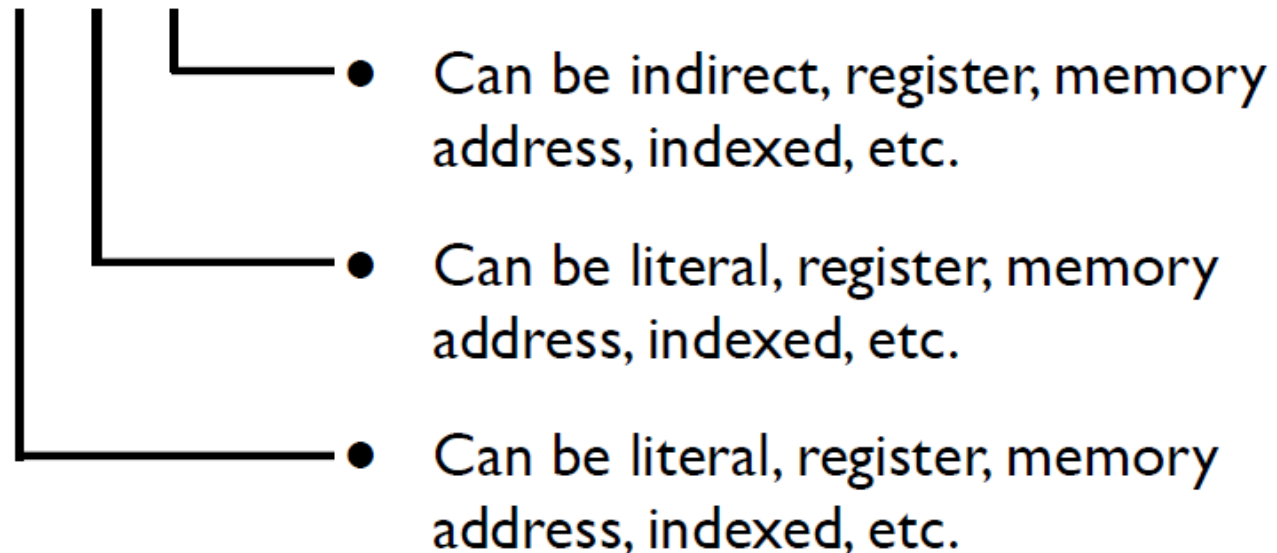
How do we address this?

- 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



- 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 $\longrightarrow$ MOV lit1 + lit2, Rx
MOV lit1, Rx $\longrightarrow$ MOV lit1 + lit2, Ry
ADD li2, Rx, Ry

- Strength reduction

MUL operand, 2, Rx $\longrightarrow$ SHIFTL operand, 1, Rx
DIV operand, 4, Rx $\longrightarrow$ SHIFTR operand, 2, Rx

- Null sequences

MUL operand, 1, Rx $\longrightarrow$ MOV operand, Rx
ADD operand, 0, Rx $\longrightarrow$ MOV operand, Rx

Peephole optimizations

- Combine operations

JEQ L1
JMP L2
L1: ...

- Simplifying

SUB operand, 0, Rx $\longrightarrow$ NEG Rx

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

ADD 1, Rx, Rx $\longrightarrow$ INC Rx
SUB Rx, 1, Rx $\longrightarrow$ DEC Rx

- Address mode operations

MOV A R1
ADD 0(R1) R2 R3 $\longrightarrow$ ADD @A R2 R3

Superoptimization

- Peephole optimization/instruction selection write 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: $B = \text{<new value>}$

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. 65

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)
ADD A B T1	{}		ld a r1; ld b r2; add r1 r2 r1
ADD T1 C T2	{"A + B"}	ld c r5;	add r1 r5 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"}	ld c r5;	add r1 r5 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"}		

Suggested Reading

- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman: Compilers: Principles, Techniques, and Tools, 2/E, AddisonWesley 2007
 - Chapter 2 (2.3, 2.5, 2.7, 2.8), Chapter 4 (4.6), Chapter 5 (5.1, 5.2.3, 5.2.4, 5.4), Chapter 6 (6.2-6.4)
- Fisher and LeBlanc: Crafting a Compiler with C
 - Chapter 6 (6.2-6.4), Chapter 7 (7.1, 7.3), Chapter 8 (8.2, 8.3), Chapter 11 (11.2)