# CS3005D
# Compiler Design
# Lecture #35
# Run time Environments

Winter 2024

Saleena N

CSED NITC

# Code generation: mapping names to addresses

`x=y+z` translated to (hypothetical machine architecture)

$\texttt{LD R1, } \texttt{y}_{\texttt{addr}}$

$\texttt{LD R2, } \texttt{z}_{\texttt{addr}}$

……….

Actual addresses not known during compilation.

What is known?

The size(number of bytes required) of each variable.

Information is in the Symbol Table.

Map each name to a relative address.

# Run-time memory

Run-time memory divided into code area and data area

# Code generation: mapping names to addresses

```
LD R1, y_addr
LD R2, z_addr
……….
```

Map each name to a relative address - relative to the *base* of the data area in memory - offset with respect to the base of the data area.

`x` at *base+0*

`y` at *base+4*…

Is base address known during compilation?

# Code generation: mapping names to addresses

source code:

```
int b=3,c=5;
```

compiler generated code:

```
movl  $3, -12(%rbp)
movl  $5, -8(%rbp)
```

# Mapping names to addresses

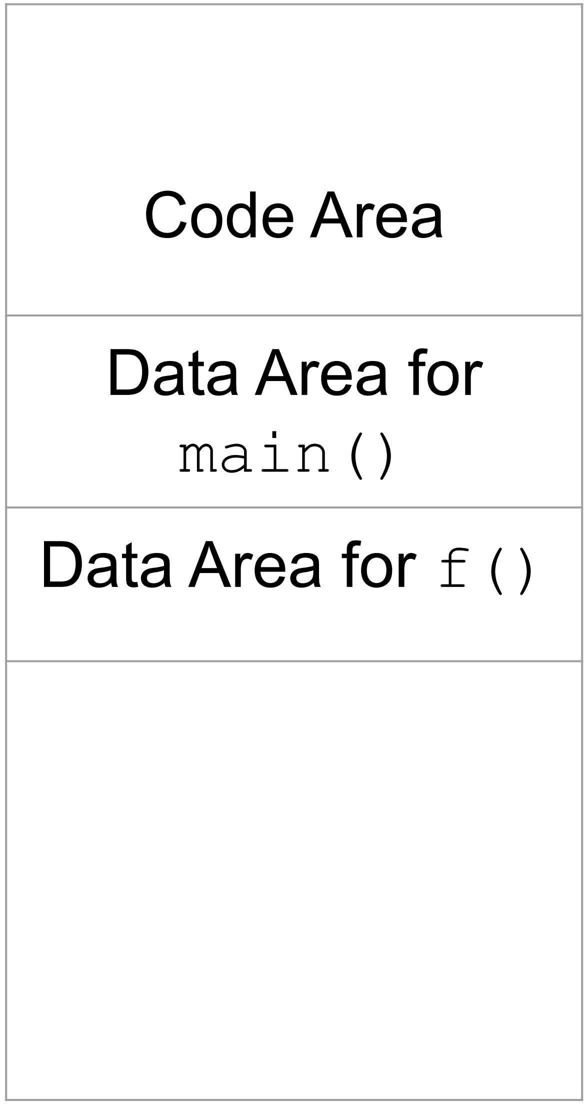
```
int main(){int a; ...   f(...)  ...};
...
int f(int y) { int z; ... x=y+z ...}
```

`x` is global
`y` is a parameter to function
`z` is a local variable to a function

relative addresses?

Separate data area for each function?

Address computation?



# `f()` is recursive

```
int main(){int a; ...   f(...)   ...};
...
int f(int y) { int z; ... x=y+z … f(...)...}
```

Multiple activations of `f()`

# `f()` is recursive

```
int main(){int a; ...   f(...)   ...};
...
int f(int y) { int z; ... x=y+z … f(...)...}
```

Multiple activations of `f()`.

Each activation requires separate space for storing its local variables, parameters etc.

# `f()` is recursive

```
int main(){int a; ...   f(...)   ...};
...
int f(int y) { int z; ... x=y+z … f(...)...}
```

A separate data area for each activation.

How many such data areas? Number of activations not known.

Base address of each area?

# Programming Language - Abstractions

Compiler must accurately implement the *abstractions* in the source language.

Abstractions:

Name, scope, binding, data type, operators, procedures ….

Name is an abstraction that gets mapped to a concrete store location

# Run-Time Environment

Compiler creates and manages a *run-time environment* in which it assumes its target programs are being executed. This environment deals with issues like

- allocation of storage locations for the objects named in the program
- mechanisms used by the target program to access variables
- mechanisms for parameter passing
- linkages between procedures
- interfaces to the operating system

# Run-Time Environment

Specifically look at:

- allocation of storage locations
- access to variables
- implementing procedure calls

# Storage Organization

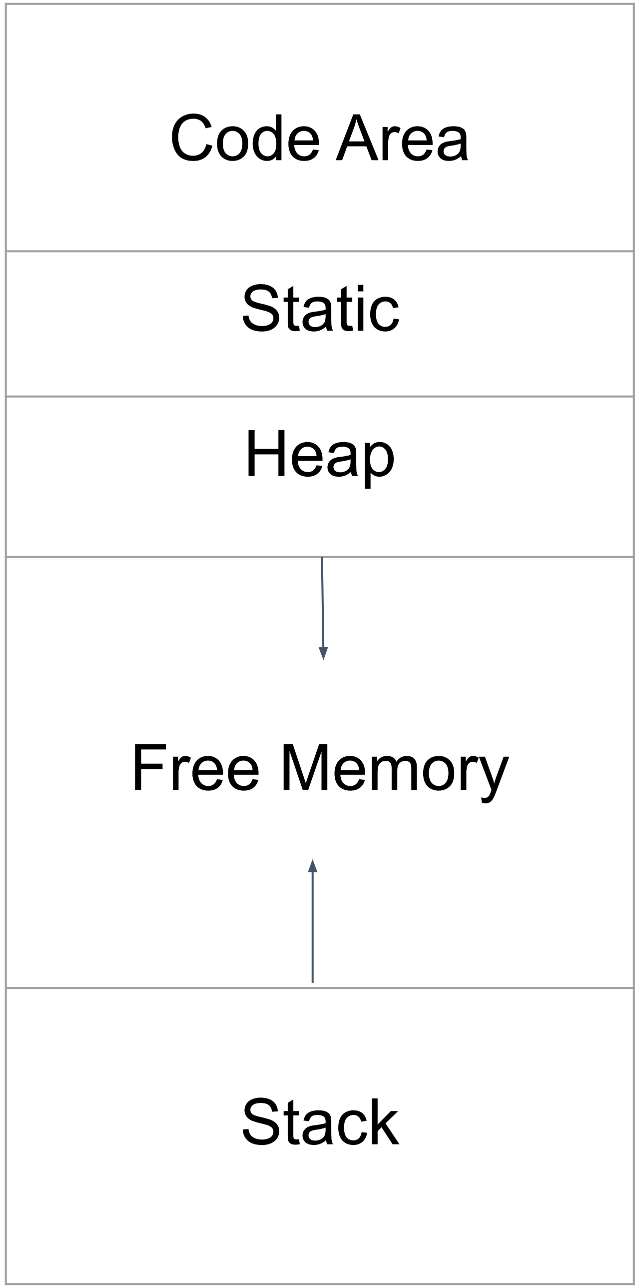
Compiler Writer's Perspective:
executing target program runs in its own logical address space

Logical Address Space

*Code Area* and *Data Area*

Logical Address Space mapped by OS into physical address space

# Run-time memory: Typical subdivision



# Data Area

Code Area - executable target program

Static Area - global constants, static variables etc.

Heap Area - for dynamically allocated data

Stack Area - storage for local variables, parameters and other data items for active functions

# Life time

Life time - time during which memory is allocated for the variable

A data item can be local/global/static/dynamically allocated - decides its lifetime

Global - lifetime is entire execution time of the program

Local to a function - lifetime of an activation of the function

Dynamic - from allocation to freeing of memory

# Stack Allocation

To support procedures/functions

Each live activation of a procedure has an *activation record* in the stack

# Activation Record (AR)

**Activation Record** (also known as **Frame**) - an area in memory allocated for storing data pertaining to an activation of a function.

- Storage in Stack Area (Control Stack).
- One Activation record for each active function
- Multiple activations of the same function - multiple ARs
- The AR of the current active function is on top of the stack
- AR memory allocated upon call and releases after returning

# A general activation record

| |
|---|
| Actual Parameters |
| Return value |
| Control link |
| Access link |
| Saved machine status |
| Local data |
| Temporaries |

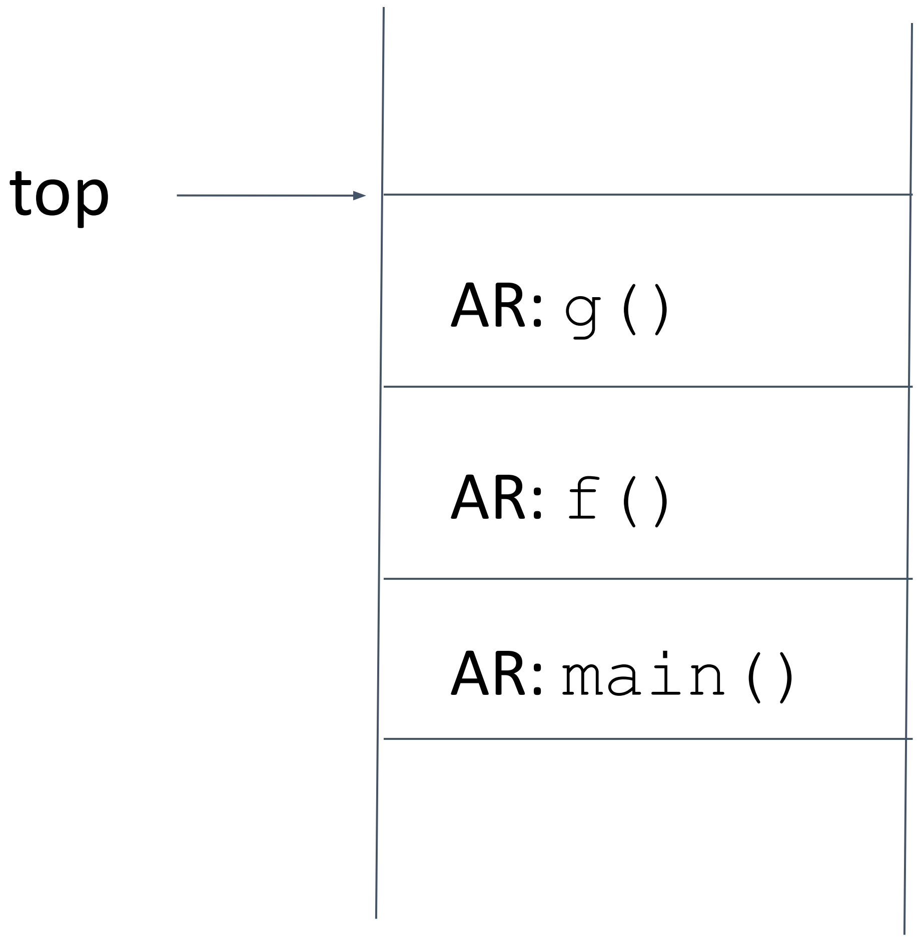
# Activation Record (AR)

- Temporaries which can not be held in registers
- Local Data of the procedure that is active
- Saved machine status - information about the state of the machine just before the call - return address and contents of registers
- Space for return value
- Actual Parameters
- Control link pointing to the AR of the caller
- Access Link - to access non local data - nested procedures - points to the AR an enclosing procedure

# Activation Record

Suppose `main()` calls `g()` and `g()` calls `f()`

How many Activation records in the stack? Topmost AR?


top
AR: g()
AR: f()
AR: main()

# Function Calls

Declaration/ Prototype: `int g(int i, int j);`

Calls / invocations:

```
g(a, b)
g(s[i], b+c)
g(g(a), h(b))
```

# Semantics of Function Calls

```
int g(int i, int j);
```

calls: `g(a, b)....g(s[i], b+c)... g(g(a), h(b))`

Evaluate each argument, copy the value of the argument to the formal parameters `i` and `j`.

Space allocated for `i` and `j` in the AR of `g()`.

# Semantics of Function Calls

Function Definition:

```
int g(int i, int j){
   int x, y; ....};
```

Local variables `x, y` in AR of `g()`.

# Semantics of Function Calls

```
int g(int i, int j);
```

Suppose `f()` calls `g()`.

`f()` is the *caller* and `g()` is the *callee*.

Who evaluates the arguments of `g()`? `f()` or `g()`?

# Parameter Passing

`f()` calling `g()`:

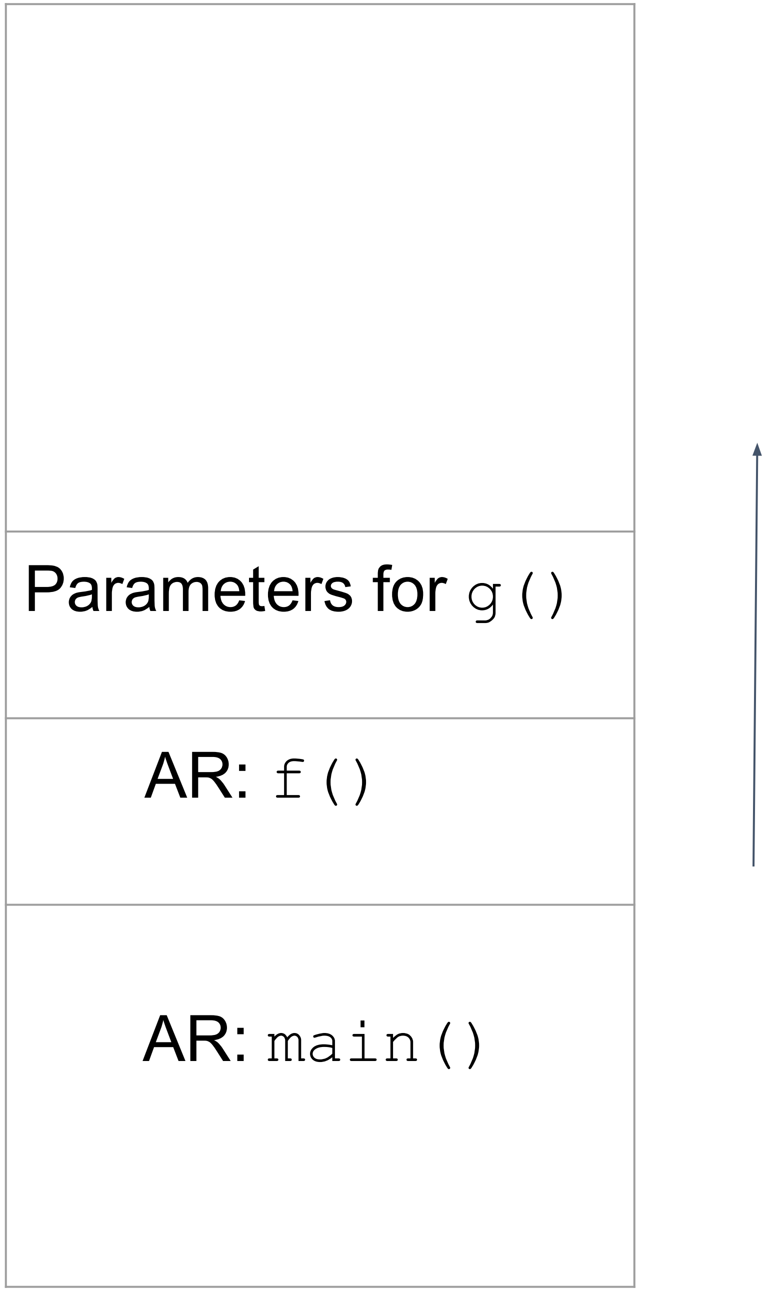
`f()` is the current active function. Its AR is on top of the stack.

`f()` evaluates arguments

`f()` passes arguments to `g()` - how?

`f()` starts setting up AR for `g()` on top of its own AR.

`f()` writes the values of parameters to the space just above its AR



# Semantics of Function Calls

While `g()` is active, it stores its local variables and temporaries in its AR.

When `g()` finishes it should return value to `f()`, and control should return back to `f()`.

Return address ?

# Calling Sequence

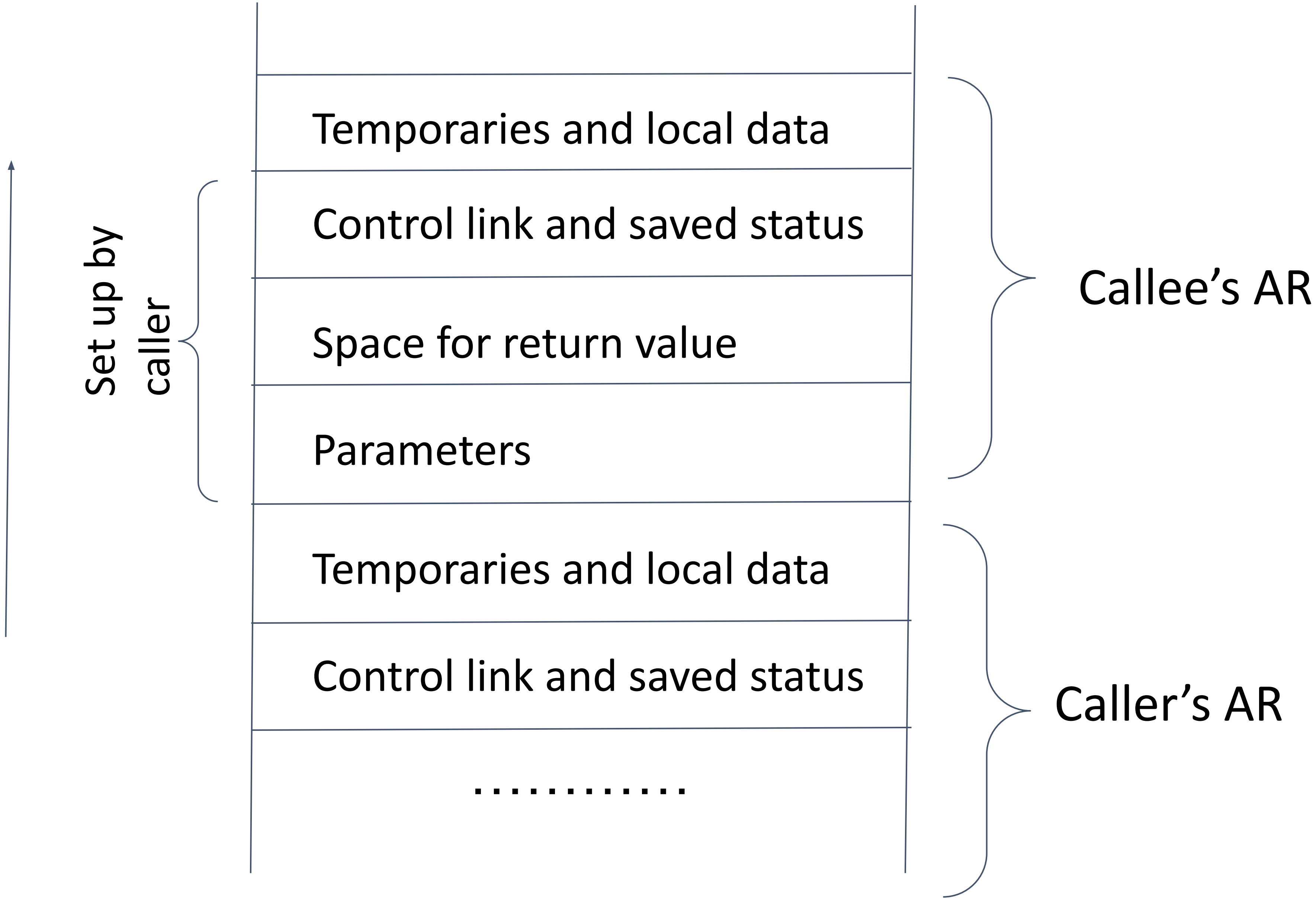
Calling Sequence (also known as call sequence)

- Code that allocates AR and enters information into it.
- Divided between the caller and the callee
- Desirable to put as much of the calling sequence into the callee as possible - why?

# Calling Sequence

Calling  Sequence

- Caller evaluates actual parameters
- Caller stores return address, current value of SP into callee's AR
- Caller increments SP to point to the top of caller's AR
- Callee saves machine status
- Callee initializes local data and begins execution



Temporaries and local data
Control link and saved status
Space for return value
Parameters
Temporaries and local data
Control link and saved status
............
Set up by caller
Callee’s AR
Caller’s AR

# Layout of AR

- Values communicated between the caller and the callee  - at the beginning of the callee's AR
  - parameters, return value

- Fixed-length items in the middle
  - control Link, access link, machine status

- Local variables and temporaries at the end of the AR

# Return Sequence

Callee places return value in the AR

Using the saved machine status, callee restores SP and other register values

Even though control returned to the caller, caller knows where the return value is, with respect to its own AR and can use it.

# Frame Pointer (FP)

**Frame Pointer**(also known as **Base Pointer**) - a register that points to a fixed position in the current AR. (in addition to SP)

Caller  is responsible for setting up value of FP before transferring control to the callee

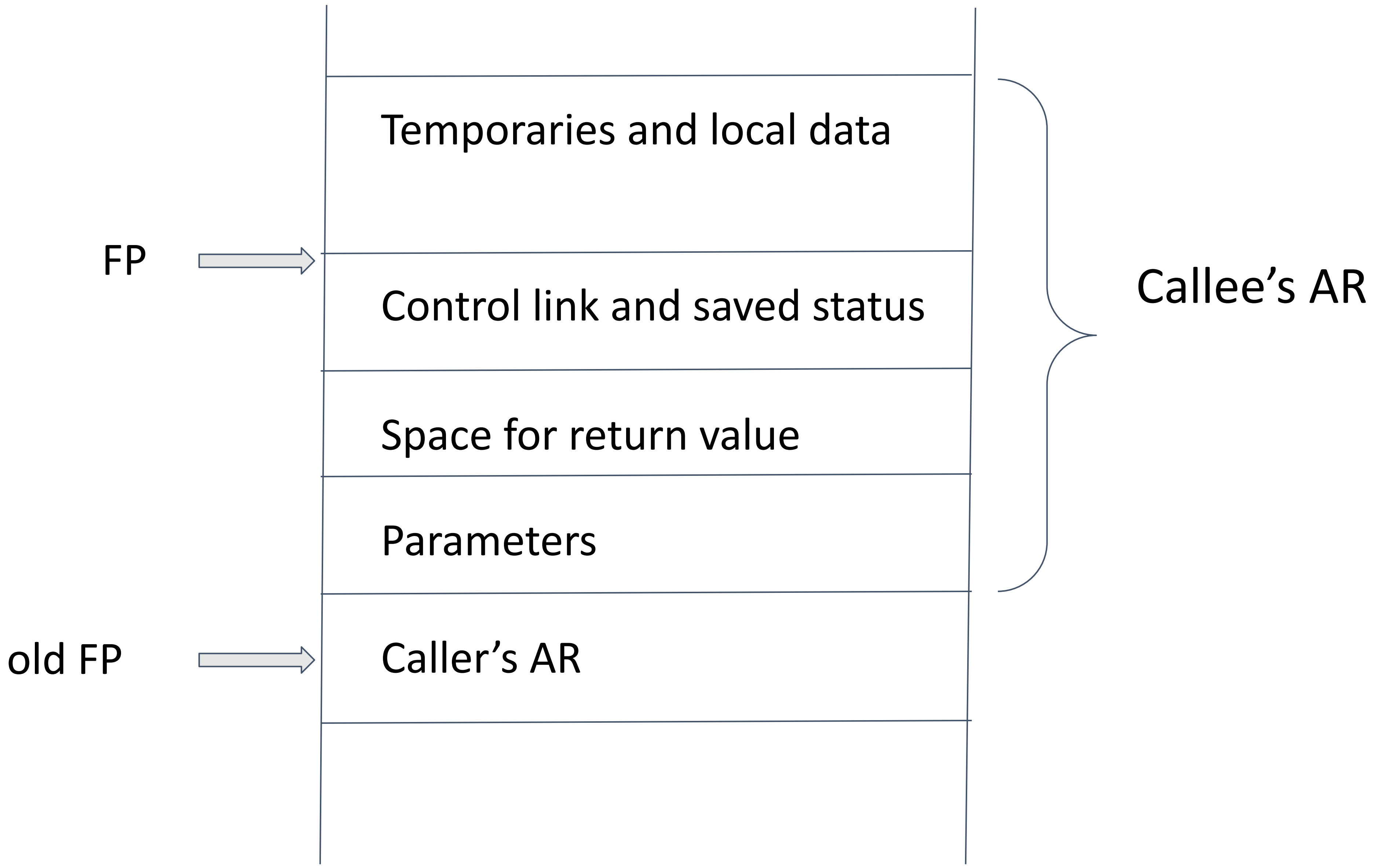
Along with machine status, current value of FP (known as *Control link*) should be stored by the caller in callee's AR.

# Calling Sequence

1. The caller evaluates the actual parameters and writes into the callee's AR
2. The caller stores a return address and the old value of FP into the callee's AR
3. Caller increments value of FP to point to the position above the status field in callee's AR
4. Callee saves register values and other status information
5. Callee initializes its local data and begins execution

# Return Sequence

1. Callee places the return value next to the parameters (in callee’s AR)
2. Using the saved machine status information, callee restores FP and other registers, and then branches to the return address that the caller placed in in the status field
3. The caller knows where the return value is, relative to the current value of FP, and can use it

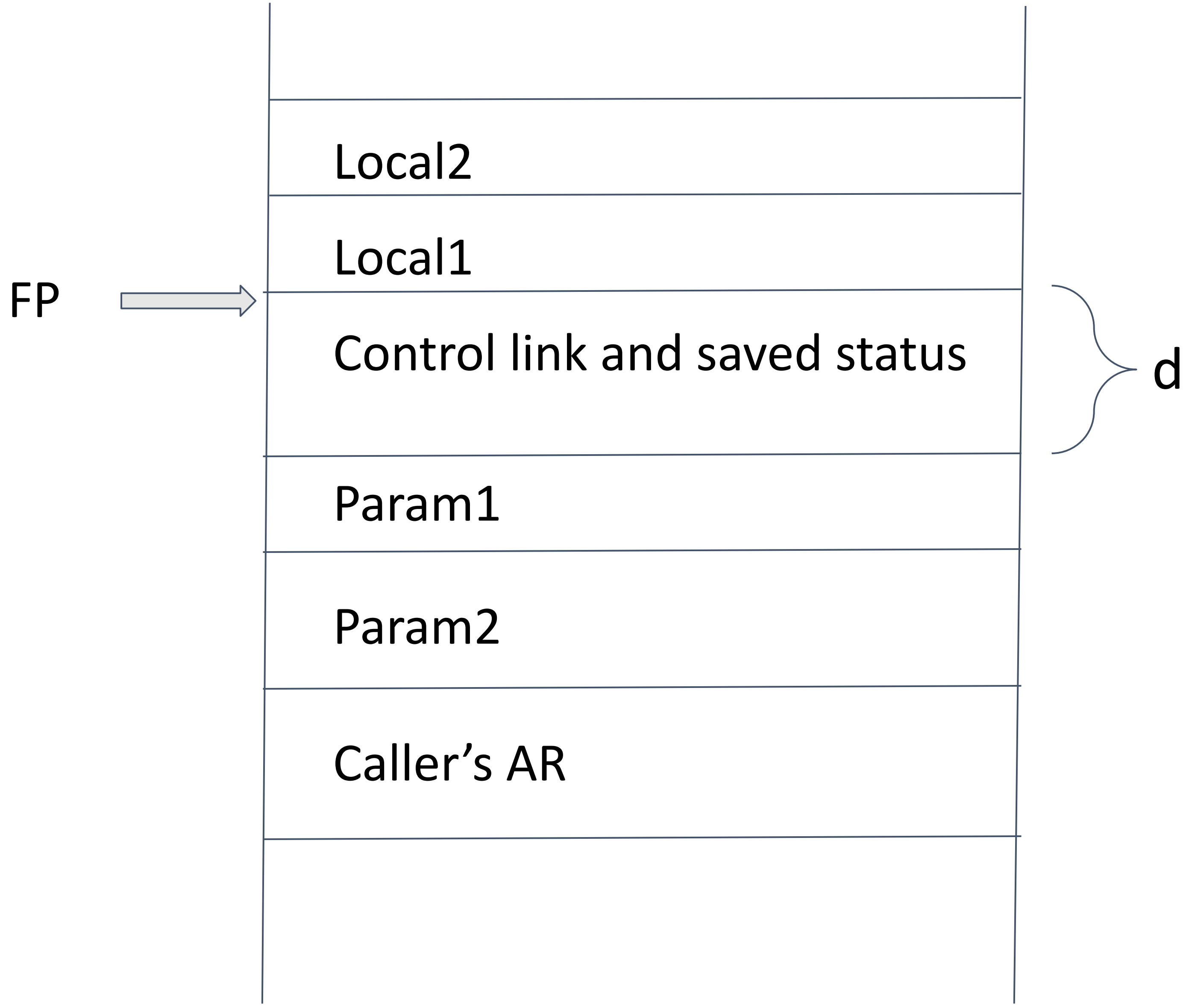

Temporaries and local data
FP
Control link and saved status
Callee’s AR
Space for return value
Parameters
old FP
Caller’s AR

# Addresses of local data and parameters

The addresses of local data and parameters can be specified with respect to offsets from FP.

Parameters at negative offsets and local data at positive offsets from FP.

Code generator computes addresses as fixed offsets from FP.




Local1 at FP
Local2 at FP+4 ….

Param1 at FP-d
Param2 at FP-d-4

# Code generation: mapping names to addresses

source code:

```
int b=3,c=5;
```

compiler generated code:

```
movl  $3, -12(%rbp)
movl  $5, -8(%rbp)
```

# Architecture Dependent

Calling Conventions

Caller save / Callee save registers

Using registers to pass parameters / return values

# Reference

ALSU Chapter 7