Runtime Environments

Roadmap

Type checking

– Went through a couple of type system design points
– Inferred the types of expressions in our language
– Showed how to propagate type errors

Today

– Begin looking at how to lower code down to assembly

Outline

Talk about what a runtime environment is

Discuss the “semantic gap”

– The difference between level of abstraction in source code and executables

How memory is laid out in an abstract machine

WYSINWYX

What You See (in source code) Is Not What You eXecute

- We think in terms of high-level abstractions
- Many of these abstractions have no explicit representation in machine code

What Abstractions are we missing?

Loops

Variables

Scope

Functions

- Flat list of opcodes
- Byte-addressable memory

Runtime Environment

Underlying software and hardware configuration assumed by the program

- May include an OS (may not!)
- May include a virtual machine

The Role of the Operating System

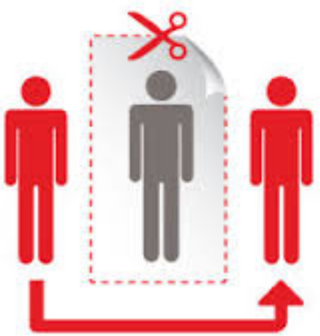
Program piggybacks on the OS

- Provides functions to access hardware
- Provides illusion of uniqueness
- Enforces some boundaries on what is allowed

Mediation is Slow

It's up to the compiler to use the runtime environment as best it can

- Limited number of very fast registers with which to do computation
- Comparatively large region of memory to hold data
- Some basic instructions from which to build more complex behaviors



Conventions

Assembly code enforces very few rules

- We'll have to structure the way we access memory ourselves

These conventions help to guarantee that isolated code can work together

- Allows modularity
- Increase efficiency

Issues to consider

Variables
- How do we store them?
- How do we access them?

Functions as straight-line code
- How do we simulate function calls?
- How do we simulate function entry?
- How do we simulate function return?

General Memory Layout

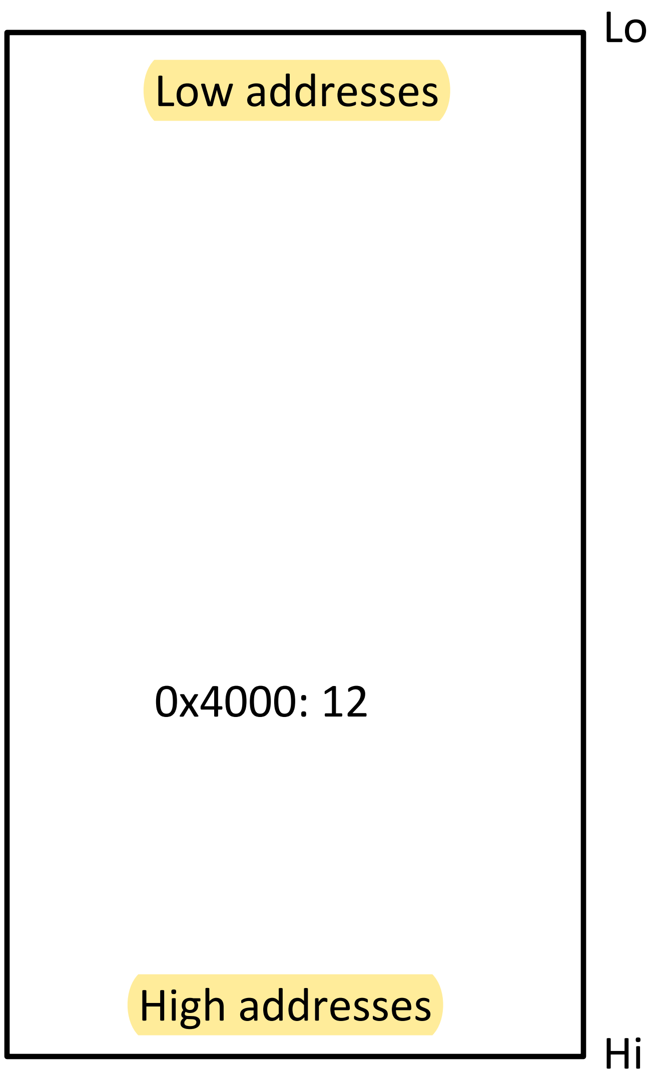
We can think of program memory as a single array

Addressable via memory cell

- Represent using a hex value

Very common to represent program memory as a "tower"

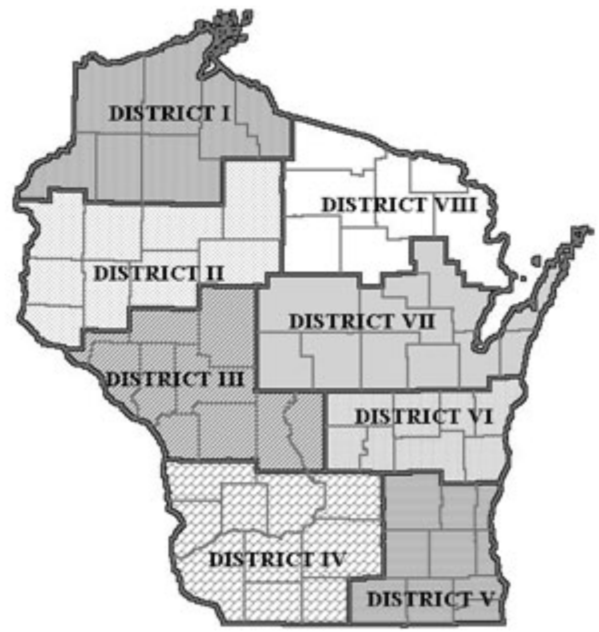
- Low addresses at the "top"
- High addresses at the "bottom"



How do we divide up memory?

Goals

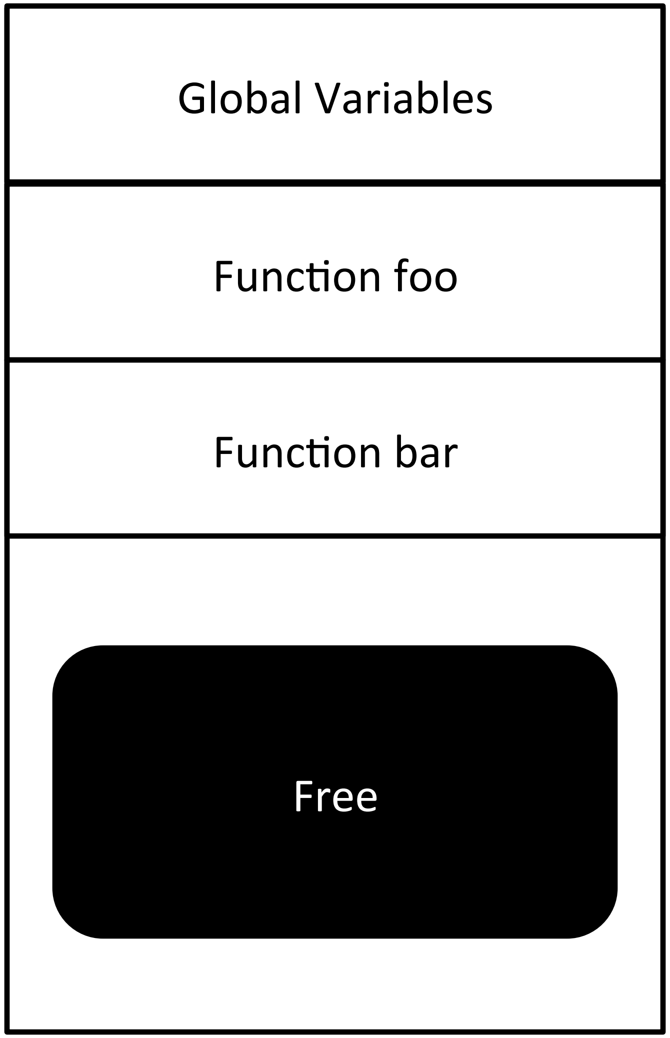
- Flexibility
- Efficiency
- Speed



Memory Layout : Static Allocation

- Region for global memory
- 1 “frame” for each subroutine of the program
  - Memory “slot” for each local, param
  - “slot” for caller
- Fast but
  - Any drawbacks?

Recursion is impossible.
Suppose bar calls itself. Then the local parameters would be over-written.



Memory Layout: The Stack

Keep the function frame idea, but allocate per invocation

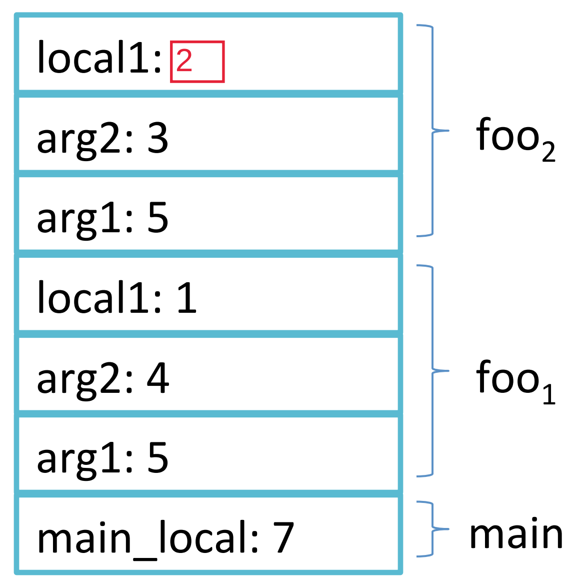
- AKA activation records
- We don't statically know how many frames we might have
- Fix a point in memory grow from there

A Closer look at Activation Records (ARs)

- Push a new frame on function entry
- Pop the frame on function exit
- To keep size down, we can put static data in the global area
  - In particular, strings
- Allows conceptually infinite recursion depth
  - In practice, we'll eventually hit the global data

```
foo(int arg1, int arg2){
    int local1 = arg1 - arg2;
    if (local1 > 0) { foo( arg1, 3); }
}
main(){
    int main_local = 7;
    foo(5, 4);
}
```

Disclaimer:
High-level idea only



In reality, only the values are stored on the stack. You have to keep track of which variables refer to which values.

Activation Records: Dynamic Locals

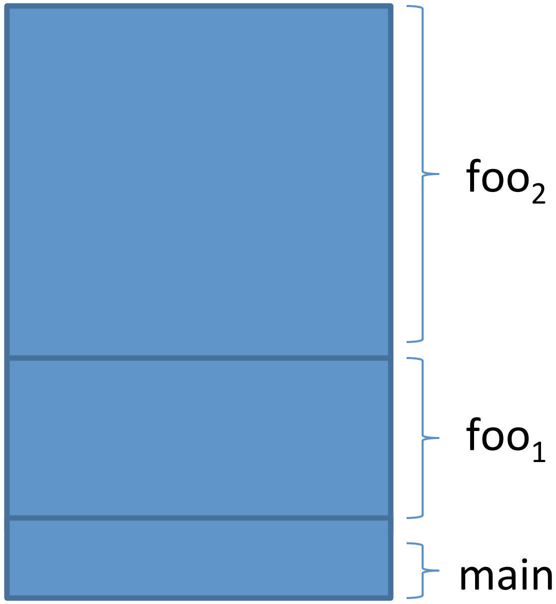
The stack can handle local variables whose size is unknown

– Grow the frame as needed during its execution

This means stack size is unknown at compile time!

– Store the previous frame's boundaries in the current frame

```
foo(int arg){
    int locArr[arg];
    …
    foo(arg * 2);
}
main(int argc, char * argv[]){
    int main_local = 7;
    foo(argc);
}
```

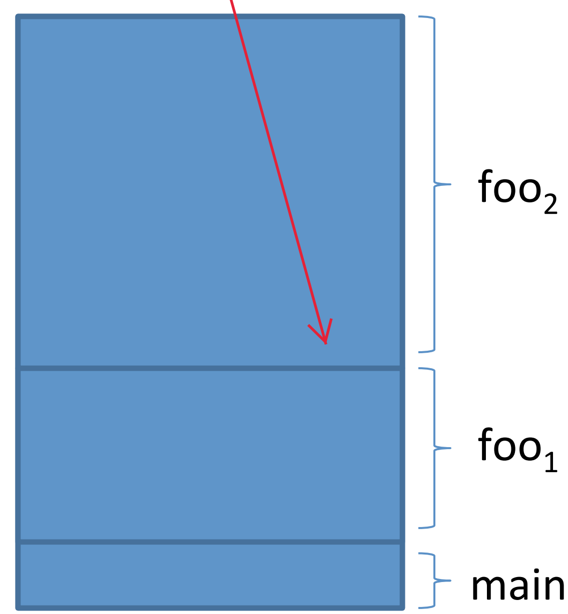


The size of each stack frame is unknown at compile time. The stack frame is constructed and destructed at runtime!

Activation Record: Summary

- Things in the frame
  - Local variable values
  - Space for the caller's frame
    - Data context
      - Enough info to remember the boundaries of the frame we called from (the caller)
    - Control context
      - Enough info to know what line of code we were at when we made the call

The line doesn't exist in reality. We need to remember ourself.



Non-Local Dynamic Memory

Surely we don't want *all* data allocated in a function call to disappear on return

Don't know how much space we'll need

- Can allocate many such objects
- Can be sized dynamically

```
public makeList(){
    Node n = new Node();
    Node t = new Node();
    n.next = t;
    return n;
}
```

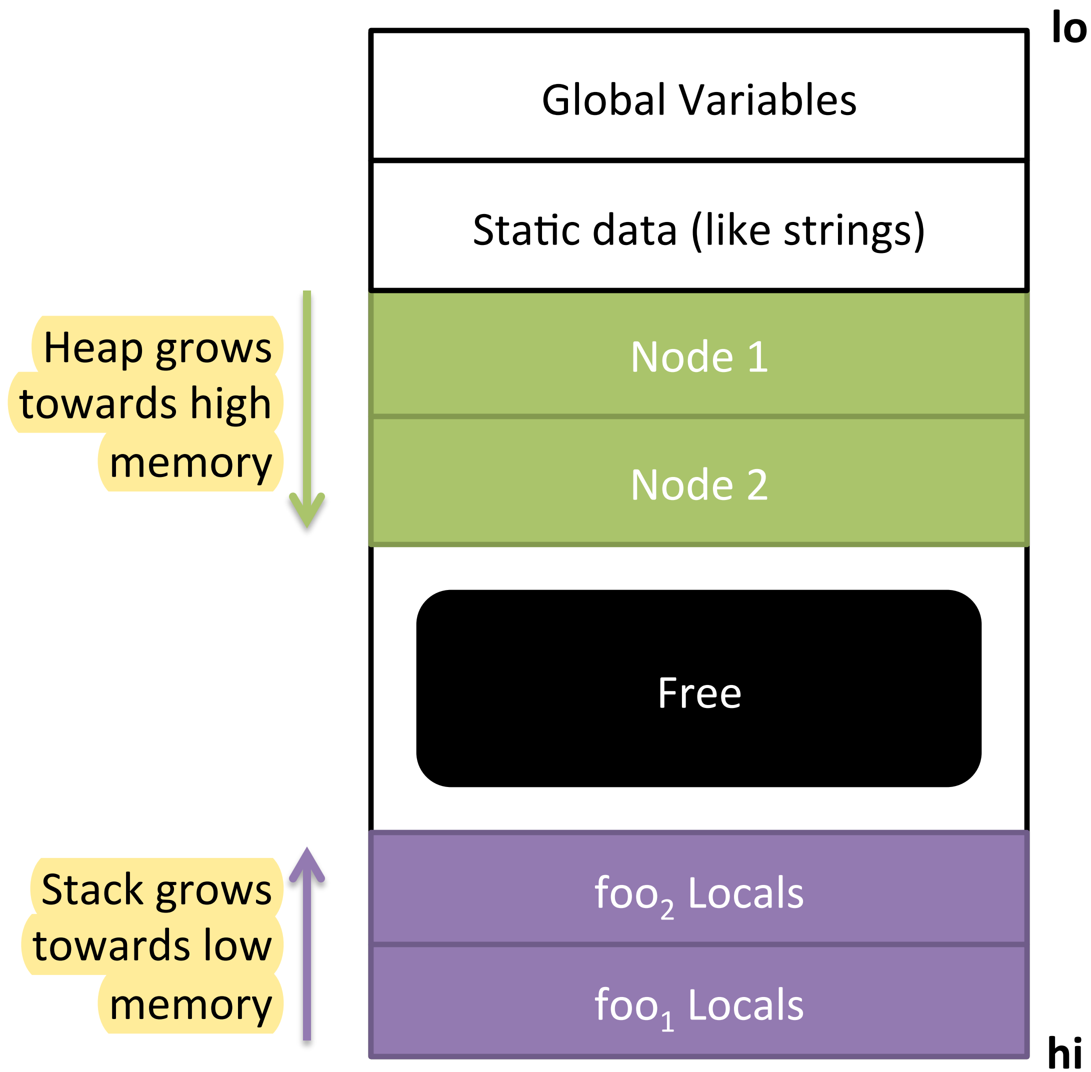
The Heap

Region of memory independent of the stack

Allocate at program’s command

How do we get rid of it?

- Ask programmer to specify when it’s unused
- Can track automatically when it’s unused GC



Function Calls

Where convention meets implementation

- Function calls are so common that their semantics are partially encoded into architecture
- Registers often have "nicknames" that hint at their purpose in representing ARs
- Some instructions implement "shortcuts" for building up and breaking down ARs

When are we “in” a function?

$ip the *instruction pointer* tracks the line of code we are executing. It tracks “where we are at” in the program

If the instruction pointer points to code that was generated for some function, we’ll say we’re in that function

```
#1  int summation(int max){
#2    int sum = 1;
#3    for (int k = 1 ; k <=
max ; k++){
#4      sum += k;
#5    }
#6    return sum;
#7  }
#8  void main(){
#9    int x = summation(4);
#10   cout << x;
#11 }
```

$ip: *#2*

Caller / Callee relationship

- Caller
  - The function doing the invocation
- Callee
  - The function being invoked
- Note that this is a per-call relationship
  - main is the caller at line 5
  - v is the callee at line 5

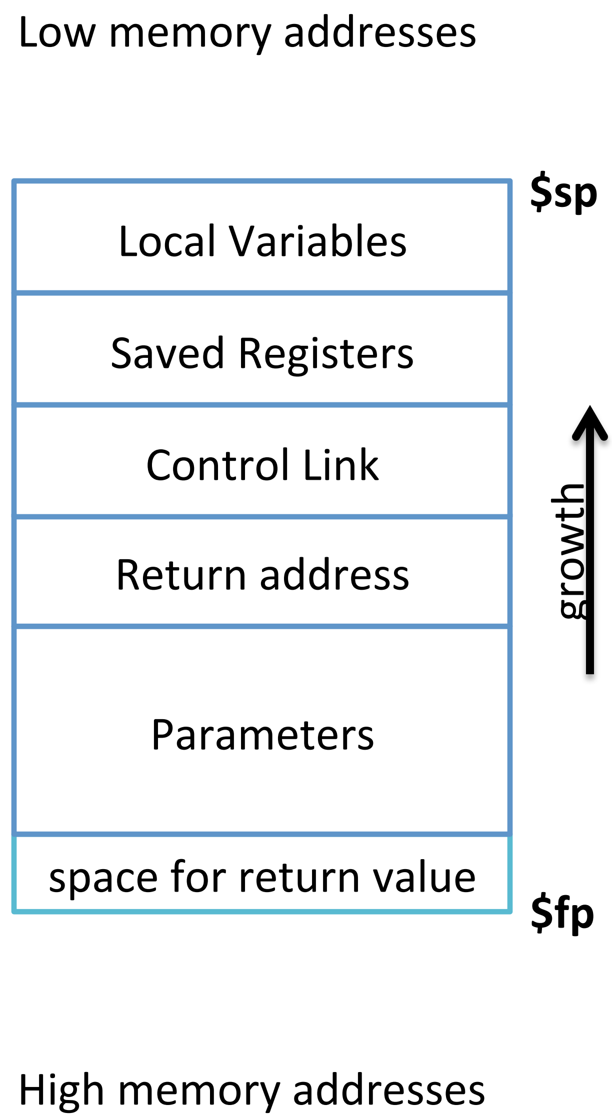
$ip

```
1. void v(){
2. }
3.
4. int main(){
5.     v();
6. }
```

How ARs are *Actually* Implemented

Two registers track the stack

- Frame pointer ($fp) tracks the base of the frame
- Stack pointer ($sp) tracks the top of the stack



Function Entry: Caller Responsibilities

Store the *caller-saved* registers in it's own AR

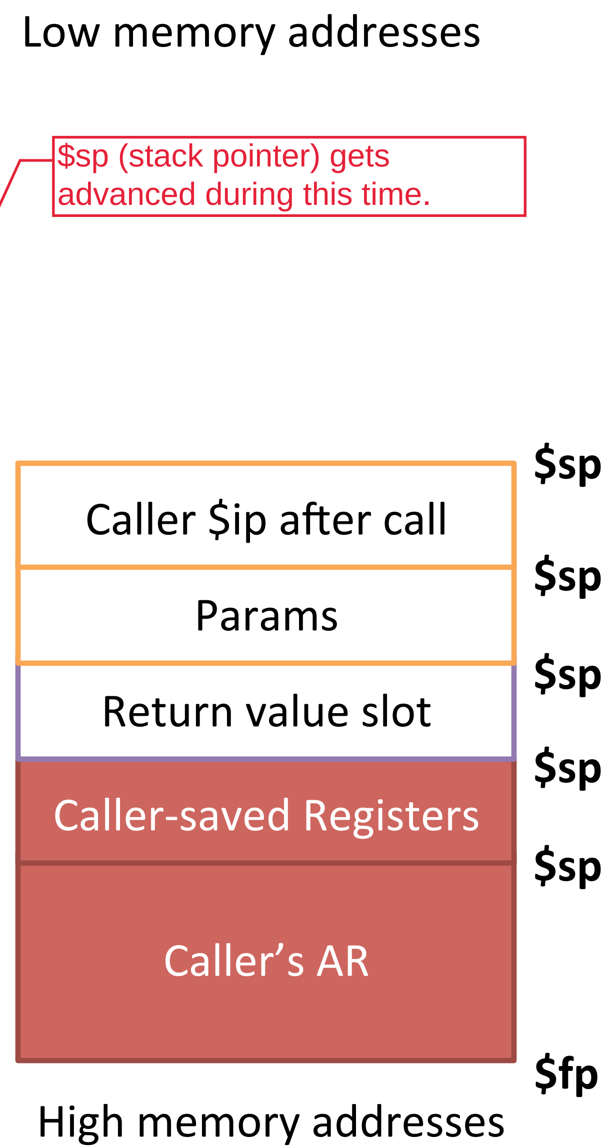
AR: activation record.

$sp (stack pointer) gets advanced during this time.

Set up actual params
- Set aside a slot for the return value
- Push parameters onto the stack

Copy return address out of **$ip**
- It's about to get obliterated

Jump to the Callee's first instruction



Function Entry: Callee Responsibilities

Save **$fp** since we need to restore it later

Update the base of the new AR to be to end of the old AR

Save *callee-saved* registers if necessary

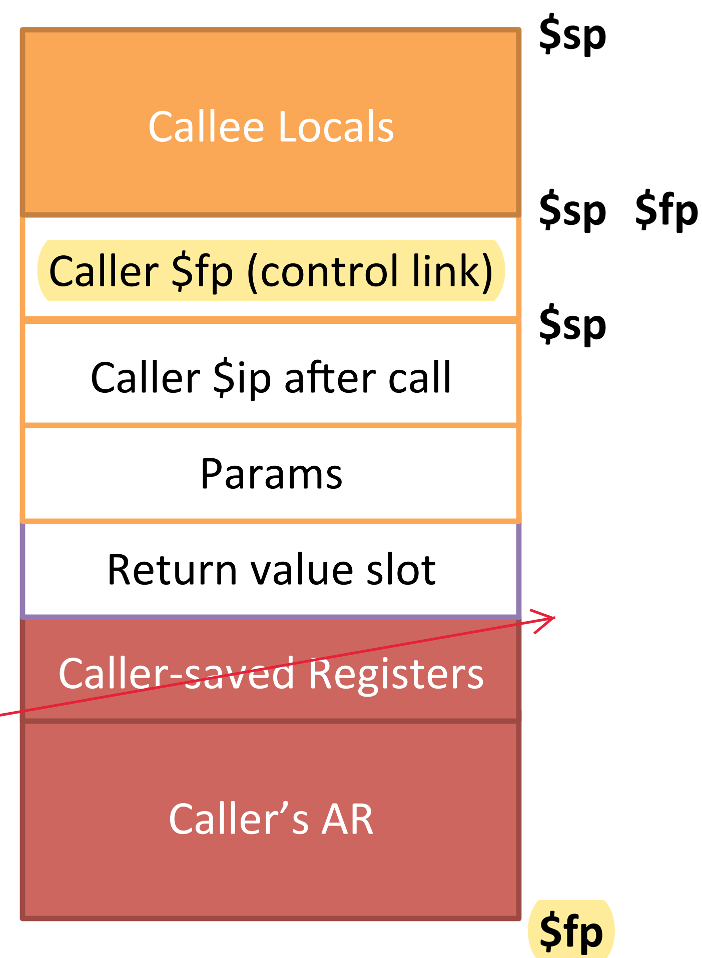
Make space for locals

Sometimes you might see $fp being marked here. This is valid as the white region is of static size (known at compile time). So whether marking the white region as part of the caller's frame or part of the callee's frame is a design choice.

The white slots are somewhat shared between the caller and the callee.

$ip Callee entry

Low memory addresses



High memory addresses

Function Exit: Callee Responsibilities

Set the return value

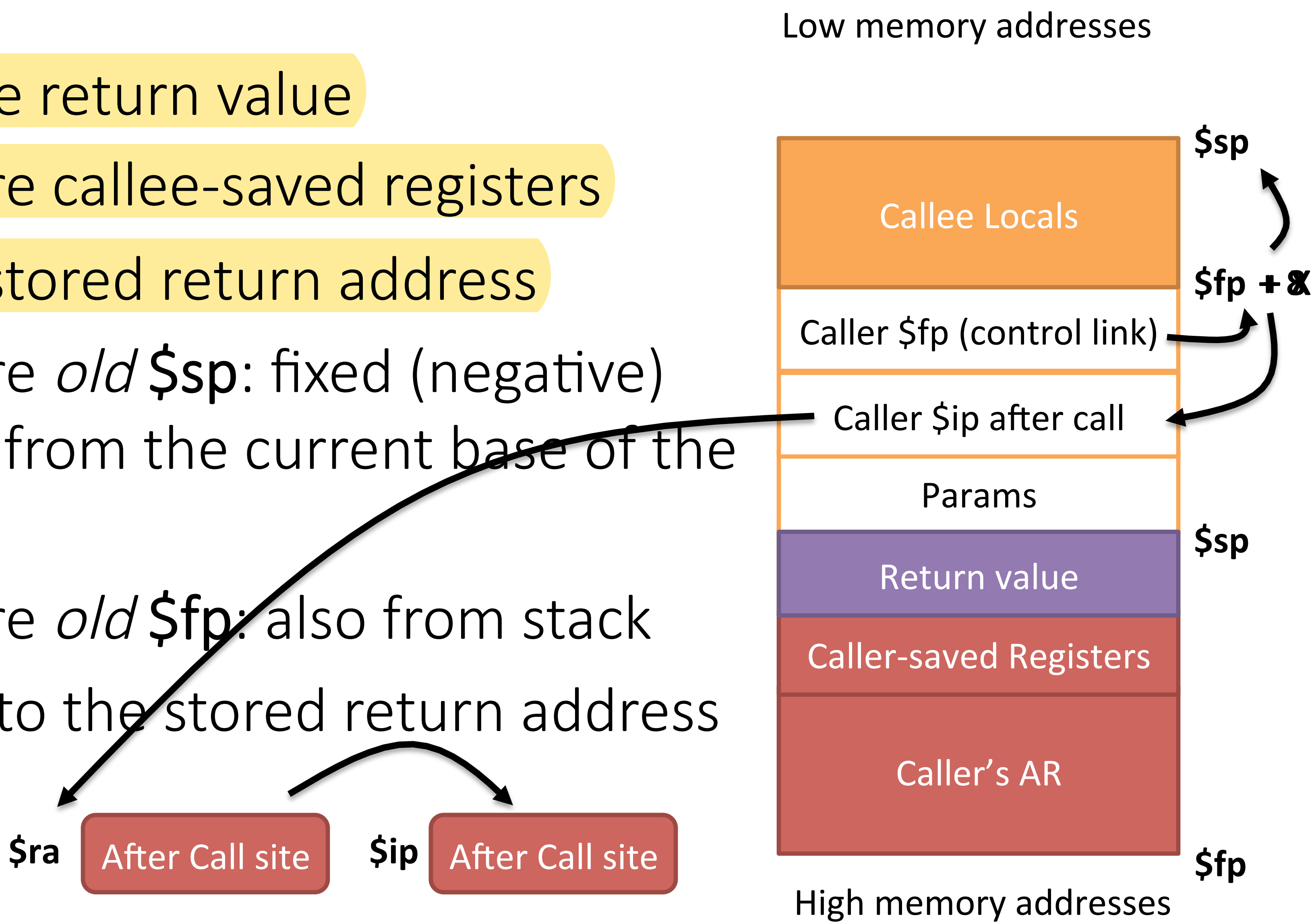
Restore callee-saved registers

Grab stored return address

Restore *old* **$sp**: fixed (negative) offset from the current base of the stack

Restore *old* **$fp**: also from stack

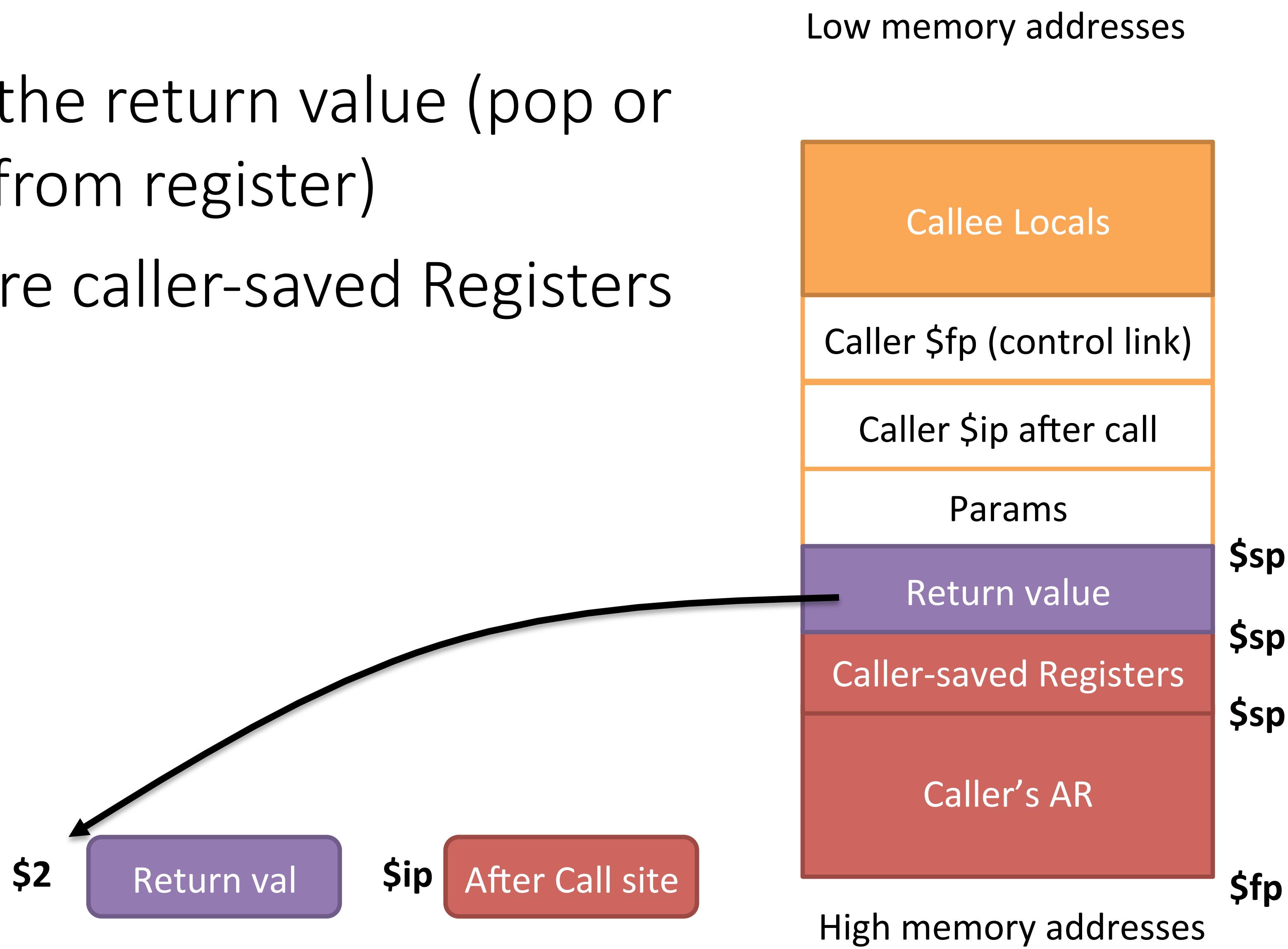
Jump to the stored return address



Function Exit: Caller Responsibilities

Grab the return value (pop or copy from register)

Restore caller-saved Registers



Example

```
int summation(int max){
    int sum = 1;
    for (int k = 1 ; k <= max ; k++){
        sum += k;
    }
    return sum;
}
void main(){
    int x = summation(4);
    cout << x;
}
```

Hardware Support for Functions

Calls

- JAL (Jump and Link): MIPS instruction that puts **$ip** in **$ra** then, sets **$ip** to a given address
- Call: x86 instruction that pushes **$ip** directly onto the stack, then sets **$ip** to given address

Return

- JR (Jump Return): MIPS instruction that sets **$ip** to **$ra**
- ret: x86 instruction that pops directly off the stack into **$ip** SPARC “Sliding Windows”
- Crazy system where caller registers are automatically saved, new set of callee saved registers automatically exposed

Next Time

- MIPS
  - We will fix a concrete runtime environment, not just a pseudo-code machine
- Variable access
  - We've shown how to store variables
  - How do we actually access them?
    - What about scope?