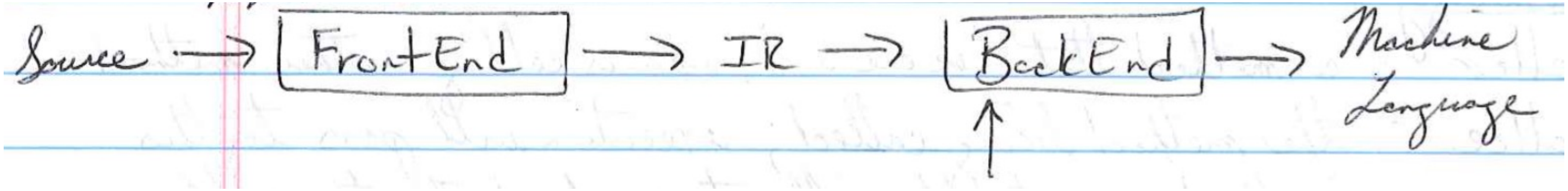
CSC 416/565: DESIGN AND CONSTRUCTION OF COMPILERS

West Chester University
Dr. Richard Burns
Fall 2023

Today

Where are we?



Towards the translation of Intermediate Code to Assembly Language

- **Activation Records** and **Memory Organization**

Runtime Organization / Environment

- Created and managed by compiler.
- Target programs will be executed in this runtime environment.
- Placing *local variables*, *function parameters (formals)*, etc. into <u>activation records</u> <u>(stack frames)</u> in a *machine-dependent* way.
- Let's consider AST as our IR:
  - IR should be *independent* of back-end

Examples of back-ends:
- SPARC
- Intel x86 (CISC)
- MIPS (RISC)
- JVM

Issues

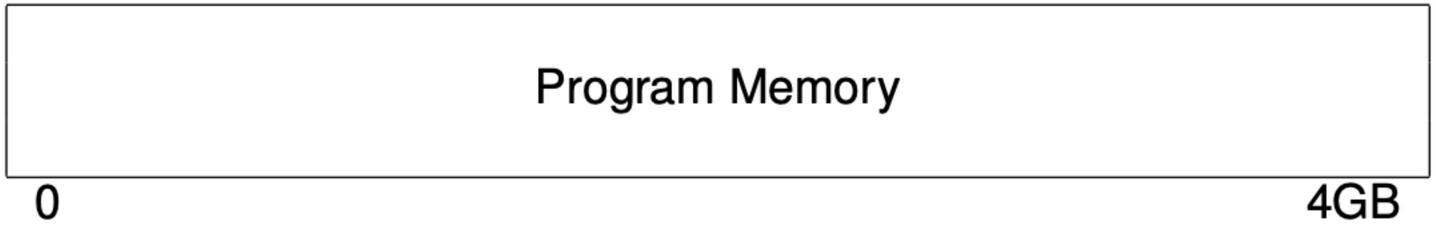
1. Allocation of storage locations
2. Access to variables and data

Ultimately need to be able to *emit* machine instructions such as:

```
LD R1, a(R0)
MUL R1, R1, 8
```

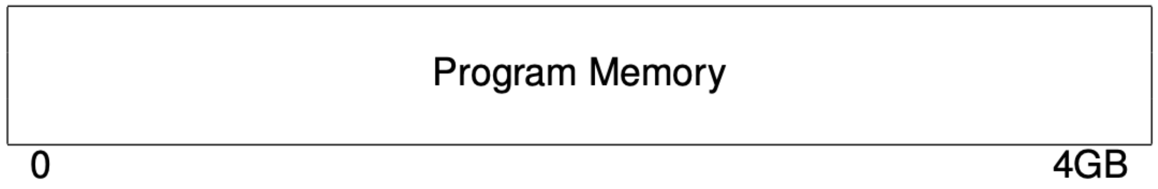
Introduction

- *In Theory:* a program/process has freedom to use memory in any way that it wants to



- *In Practice:* a convention has developed that divides the area of a program into logical segments
- We'll be talking about memory addresses from low to high

What goes in Program Memory?

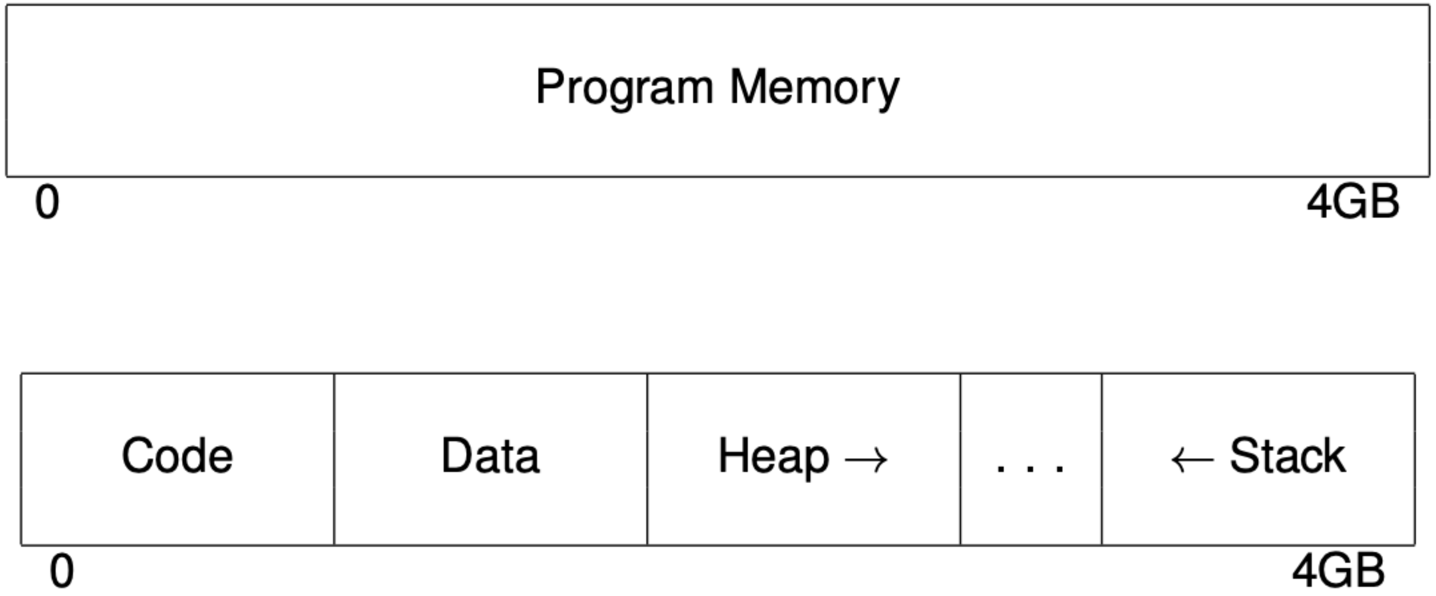


<u>Program memory is allocated by the OS at runtime</u>

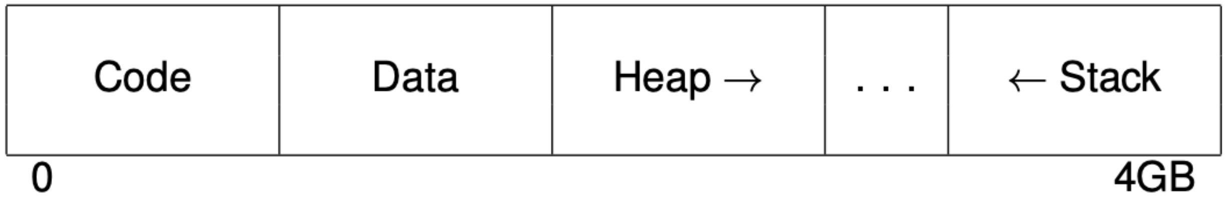
1. Machine Code: assembly instructions to execute
2. Global data and variables
3. Space to dynamically allocate memory at runtime
   - (`new` in Java for object instances, `malloc` in C, …)
4. Local variables currently use, function calls, current execution state of the program

Logical Segmentation

- *In Practice:* a convention has developed that divides the area of a program into logical segments



Logical Segmentation



<u>Program memory is allocated by the OS at runtime</u>

1. [Code] Machine Code: assembly instructions to execute
   - Sometimes called **code segment** or **text segment**
2. [Data] Global data and variables
3. [Heap] Space to dynamically allocate memory at runtime
   - (`new` in Java for object instances, `malloc` in C, …)
   - Grows "up" from <u>low</u> memory addresses to <u>high</u> addresses
4. [Stack] Local variables currently use, function calls, current execution state of the program
   - Grows "down" from <u>high</u> addresses to <u>low</u> addresses

Memory in-between heap and stack is currently unused memory.

Storage Organization

Logical Space:

Code	Data	Heap →	. . .	← Stack
0				4GB

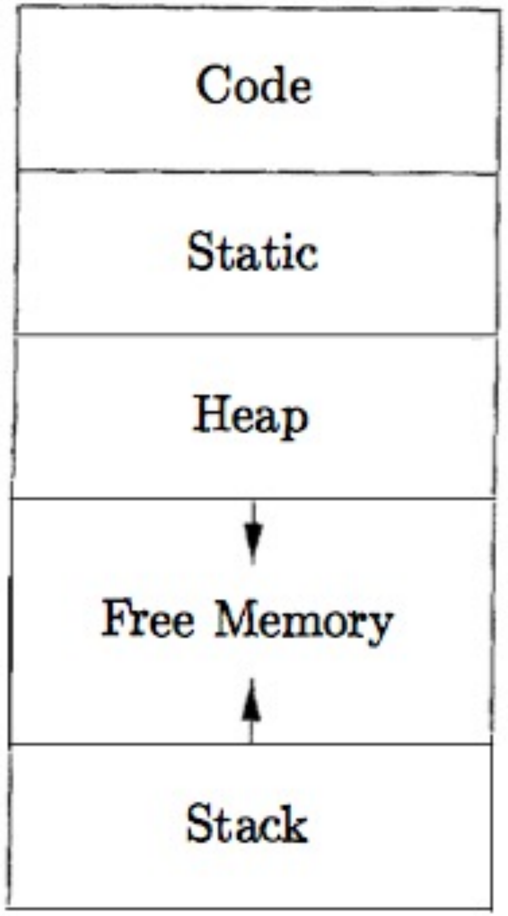
- *Compiler perspective:* executing target program runs in its own logical address space in which each program value has a location.
- *Operating system:* responsible for executing multiple target programs simultaneously and mapping logical into physical address space.
    - physical address space for a program is usually spread across memory (non contiguous)

Multiple Simultaneous Programs:

OS Kernel		Process 1				Process 2			
Code	Data	Code	Data	Heap	Stk	Code	Data	Heap	Stk
Virtual Addrs:		0	$\longrightarrow$		4GB	0	$\longrightarrow$		4GB

Runtime Logical Memory Organization of Code and Data Areas

Low address



High address

- 1 byte is the smallest unit of *addressable* memory

  1 byte = 8 bits

  4 bytes = 1 word

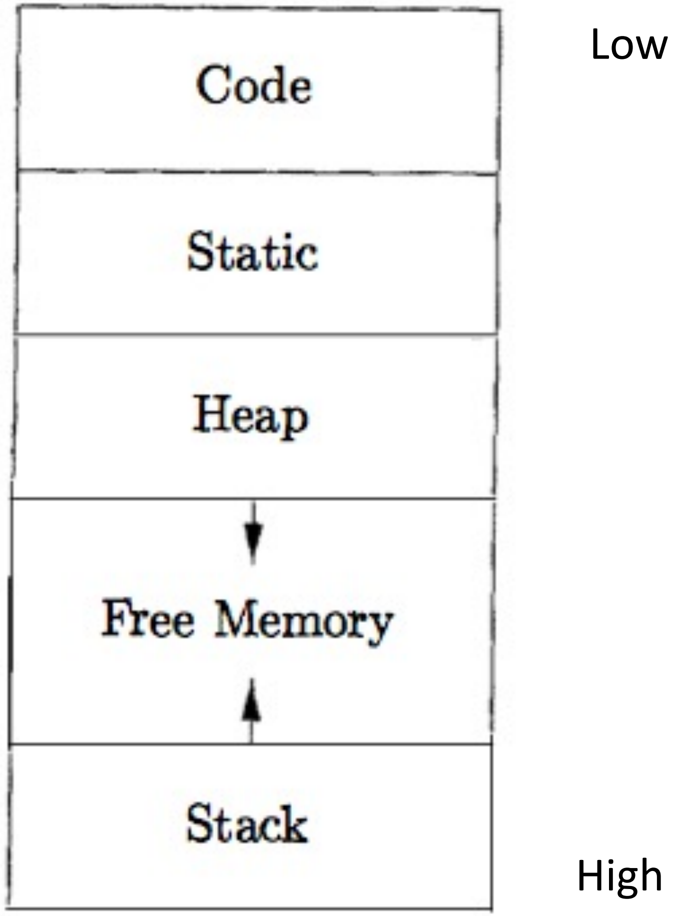
- Multibyte objects are stored in consecutive bytes and given the address of the *first byte*.
- Amount of storage needed for a *named variable* is determined from its type.

What’s In Each Part?

Today: will primarily discuss **stack management**

- will talk about **heap management** later

- Generated target code.
- Static: global constants, information used for garbage collection, etc.
- *Dynamic Portions:* Stack and Heap begin at opposite ends of the address space and grow towards each other
  - Stack: holds activation record data structures and data local to a procedure
  - Heap: holds data that may outlive the call to a procedure. Garbage collection assists with heap management by detecting useless data elements.



Big idea: stack grows as local variables enter scope and functional calls are performed.

Stack Management

Properties:

- Space for local variables is pushed onto stack when a procedure is called.
- Local variables are destroyed when a function returns.
- *The Benefit:* **Allows space to be shared by procedure calls whose durations do not overlap in time.**

(Other high-level languages (ML, Scheme, …) are more complicated to represent.)

Activation Tree

Represents the activations of procedures during the running of a program.

- Each node is one activation.
- Children of a node correspond to activations of procedures called by that node.

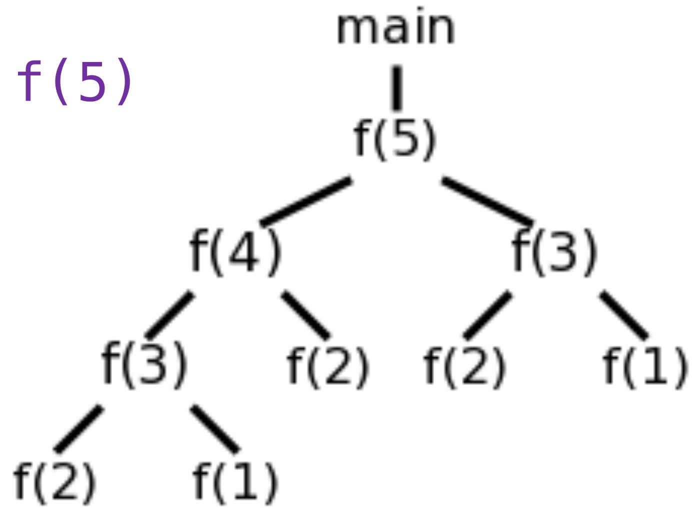
Big Idea: space to be shared by procedure calls whose durations do not overlap in time

Example: Fibonacci Sequences

```
System starts main
    enter f(5)
        enter f(4)
            enter f(3)
                enter f(2)
                exit f(2)
                enter f(1)
                exit f(1)
            exit f(3)
            enter f(2)
            exit f(2)
        exit f(4)
        enter f(3)
            enter f(2)
            exit f(2)
            enter f(1)
            exit f(1)
        exit f(3)
    exit f(5)
main ends
```

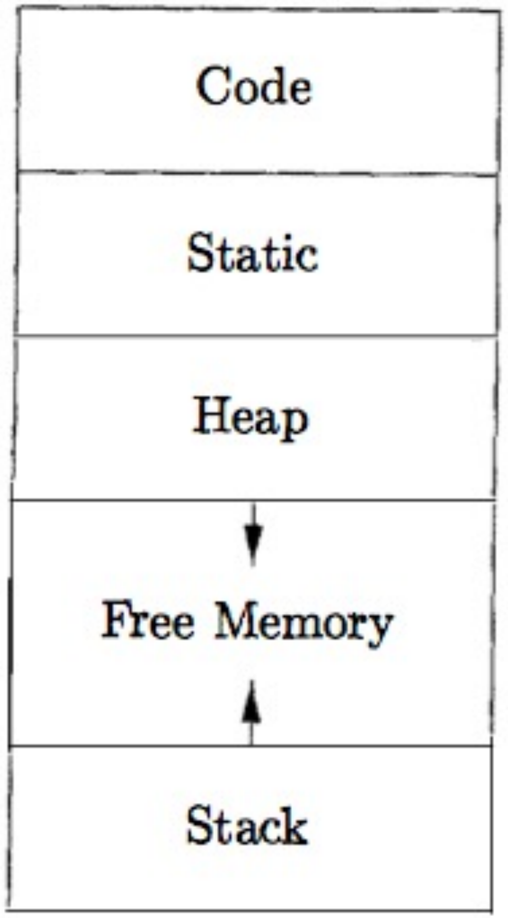
```
public int fibonacci(int n)  {
    if(n == 0)
        return 0;
    else if(n == 1)
      return 1;
   else
      return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Initial Call: `f(5)`



Terminology

- Caller: a method that "we are in", who is calling another method (the callee)
- Callee: the method being called; execution will pass to this method; eventually will return back to the caller
- Stack Push: on entry to functions, parameters are pushed in large batches
- Stack Pop: when method returns, variables popped in large batches



Low

High

Stack Management

- Most CPUs have a specialized register that stores the address where the next item will be pushed/popped
  - **$SP Stack pointer**

- *Entry to a function:* stack grows and $SP decreases in value
- *Exit from a function:* stack shrinks and $SP increases in value

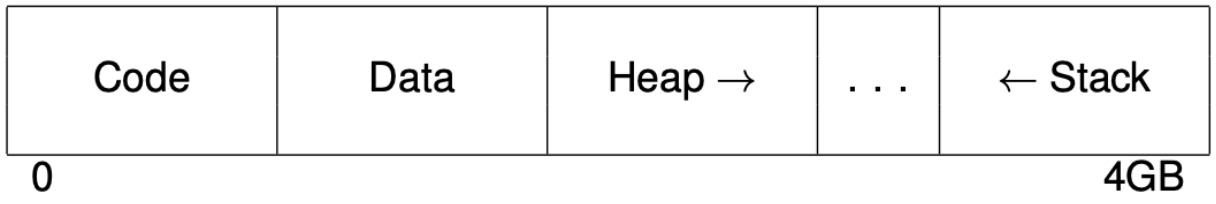
The value (memory location) of $SP changes during execution of a program. *(e.g. the top of the stack grows and shrinks)*

- *Entry to a function:* stack grows and $SP decreases in value
- *Exit from a function:* stack shrinks and $SP increases in value

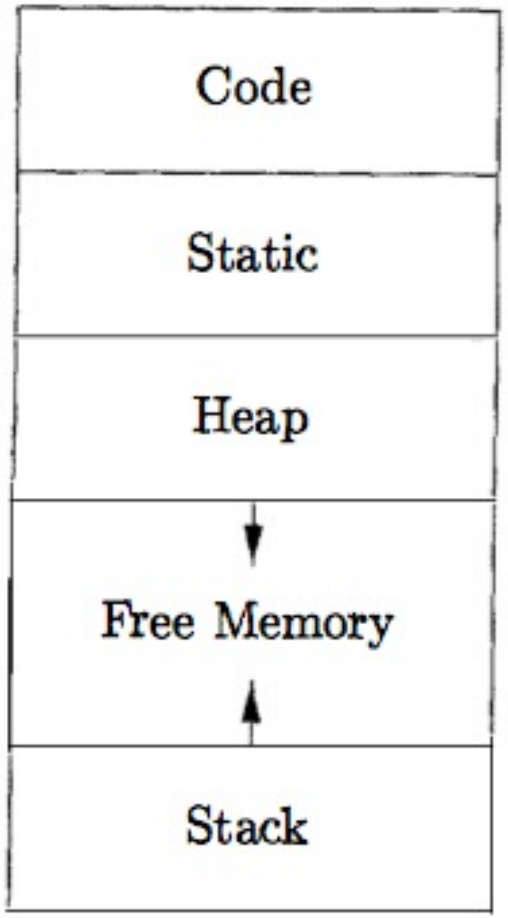
The stack is usually designed to start at a high memory address and "grow" towards smaller addresses.

Convention is that stack always “grows” towards lower addresses

Thain



Low

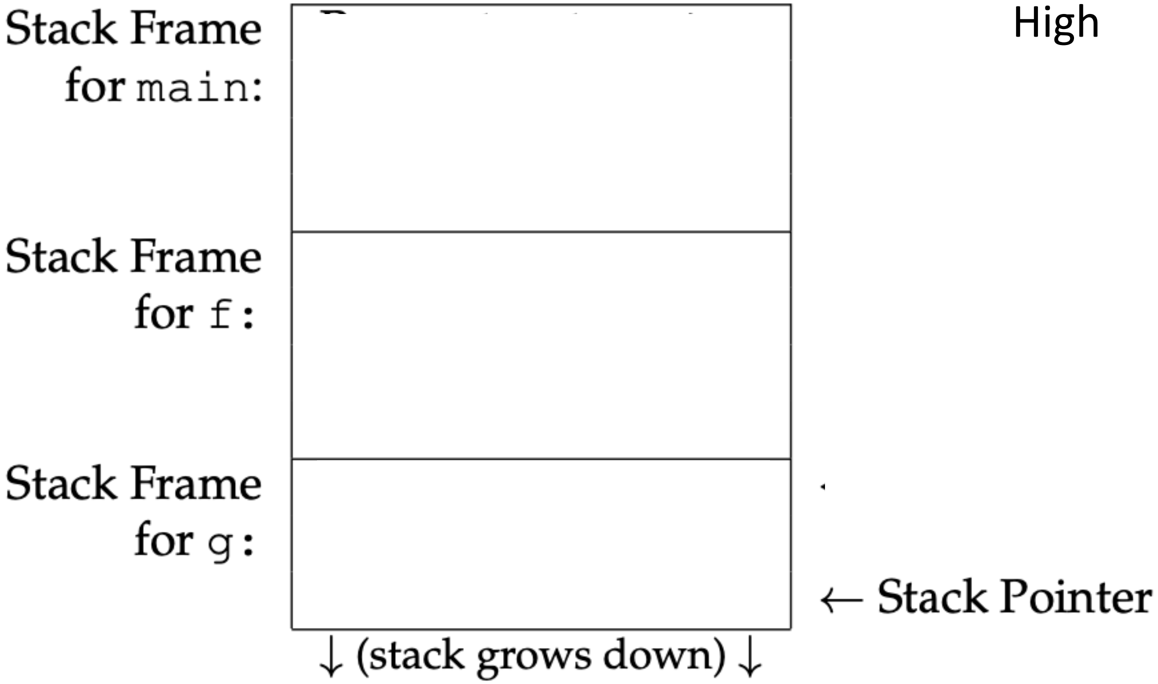


High

Dragon Book

Thain

High



Low

Stack Frame / Activation Record

Stack frame: memory on the stack used for an invocation of a function

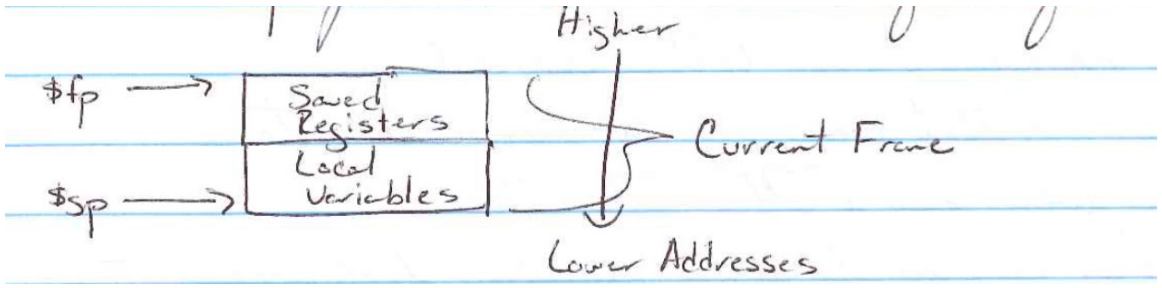
- Contains parameters and local variabes used by that function
- When a function is called, a new stack frame is pushed
- When a function returns, the stack frame is popped

Typical Layout of a Stack Frame

- $sp stack pointer points to top of stack
  - Program also has to know where local variables exist in memory/on the stack. Tricky because $sp can move.
- $fp frame pointer register stores memory address of beginning current frame
  - Does not move for the duration of the subroutine call.
  - *Big Idea:* Parameters that are passed into the subroutine can remain at a constant spot relative to the frame pointer.
  - *(Terminology: the frame pointer is sometimes called the base pointer)*

Common Activation Frame Layout

- $fp points to *first* word (highest memory location) in the currently executing procedure's stack frame
- $sp points to the *last* word of the frame ("top" of the stack)



Stack Frame for `main`:	Parameters to `main`	
	Old Frame Pointer	
	Local Variables	
	Return Address	
Stack Frame for `f`:	Parameters to `f`	
	Old Frame Pointer	
	Local Variables	
	Return Address	
Stack Frame for `g`:	Parameters to `g`	← Frame Pointer
	Old Frame Pointer	
	Local Variables	← Stack Pointer

↓ (stack grows down) ↓

Final Thoughts

- Nothing magic about this organization.
  - Could rearrange order of frame elements.
  - Details of the elements in a stack frame differ somewhat between CPU architectures and OS’s
  - As long as caller/callee agree on what goes in the stack frame, then any function may call another
  - BUT, one frame layout is better than another if it improves execution speed or simplifies code generation.
- Real compilers hold as much of the frame as possible in registers, not the stack (memory).

After the Break

Stack Calling Convention

A Deeper Look at Stack Frames and Calling Conventions

When one function calls another:

1. Place function arguments on the stack
2. Place function arguments into machine registers
3. Variation of strategies (1) and (2)

- Will focus mainly on the **stack calling convention**

Stack Calling Convention

Conventional approach:

1. Push arguments on stack in reverse order
2. Jump to address of function
3. Also set the return address on stack
4. Saves old frame pointer
5. Create space for any needed local variables

Function Call:

```
f(10,20);
```

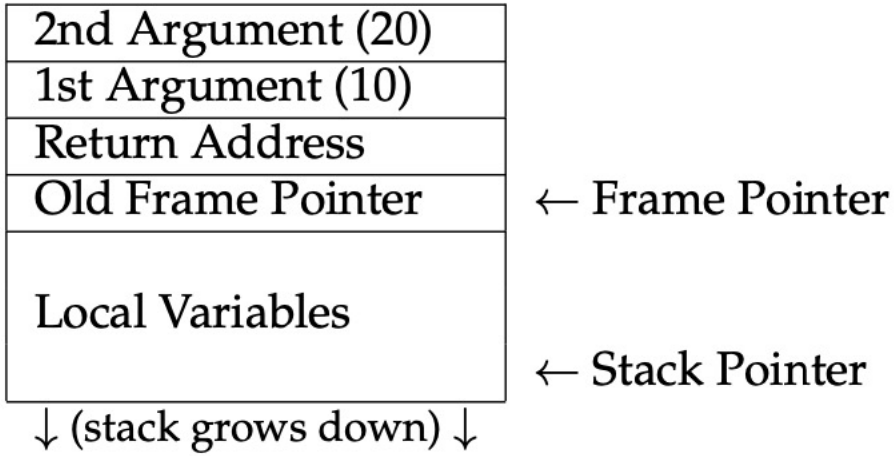
Potential Assembly Code:

```
PUSH $20
PUSH $10
CALL f
```

Function Call:
f(10,20);

Potential Assembly Code:

```
PUSH $20
PUSH $10
CALL f
```



- Arguments and local variables are loaded by `f` from memory/stack relative to the frame pointer
  - Arg1 is always two words above $fp
  - Arg2 is always three words above $fp
- This convention allows for a *variable* number of arguments

It’s Arbitrary!

- Runtime organization design is arbitrary.
- Frame layout design is arbitrary.
- *(google “frame design MIPS”)*

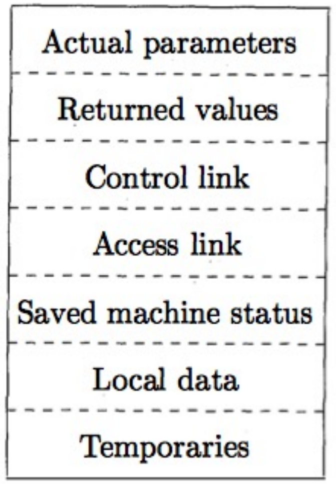
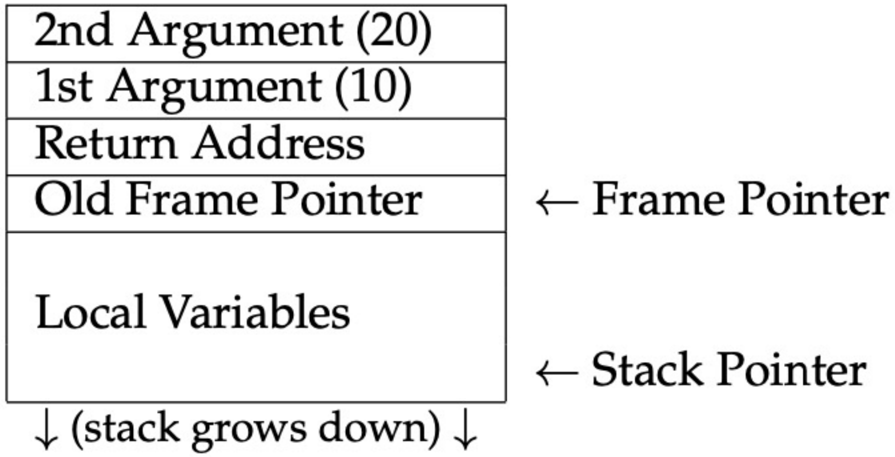


Figure 7.5: A general activation record

Dragon



Thain

Appel

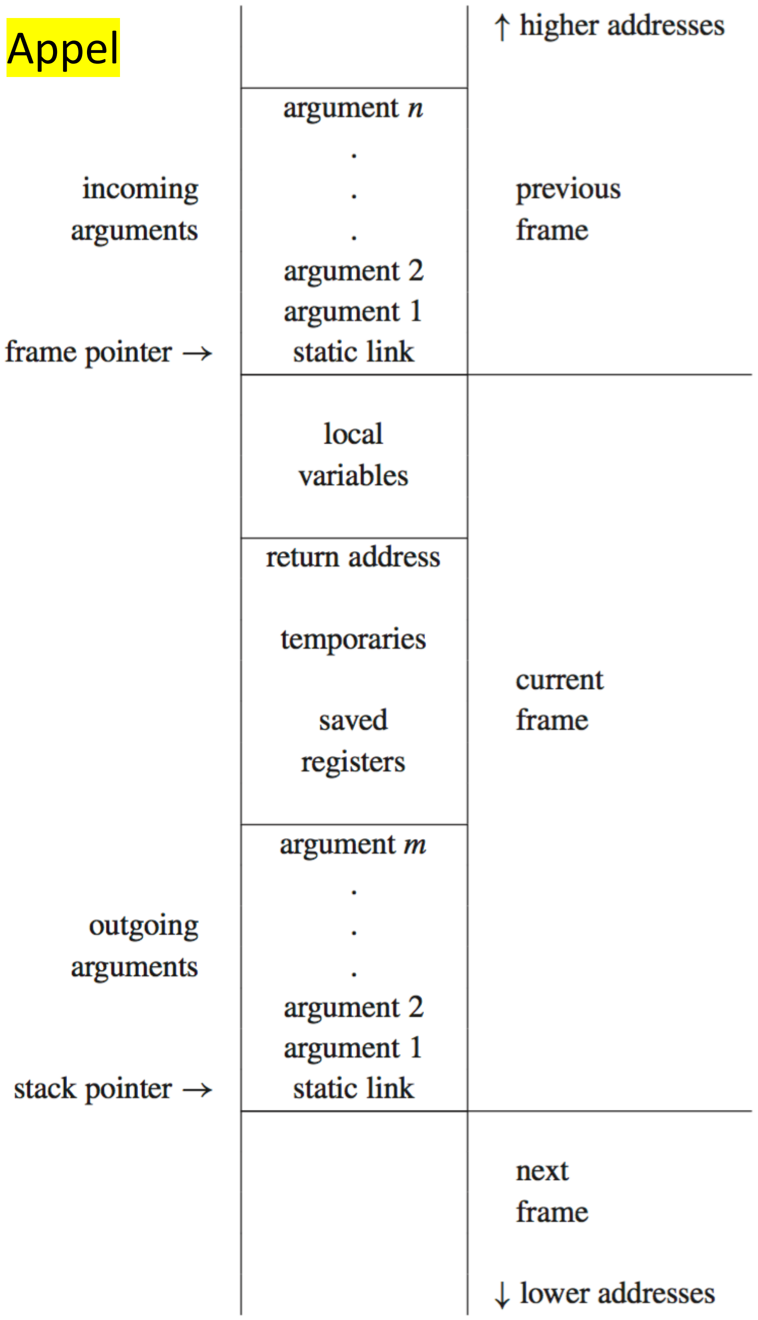


FIGURE 6.2. A stack frame.

Contents of Current Frame

- **Local Variables:** with respect to the current frame
  - Some in frame; others possibly in registers
- **Return Address:** memory location holding where execution should go to in calling function when callee returns
- **Saved Registers:** if a register needs to be saved into the frame to make room for other registers
- **Temporaries:** if we need to store additional intermediary data than what we can fit in registers

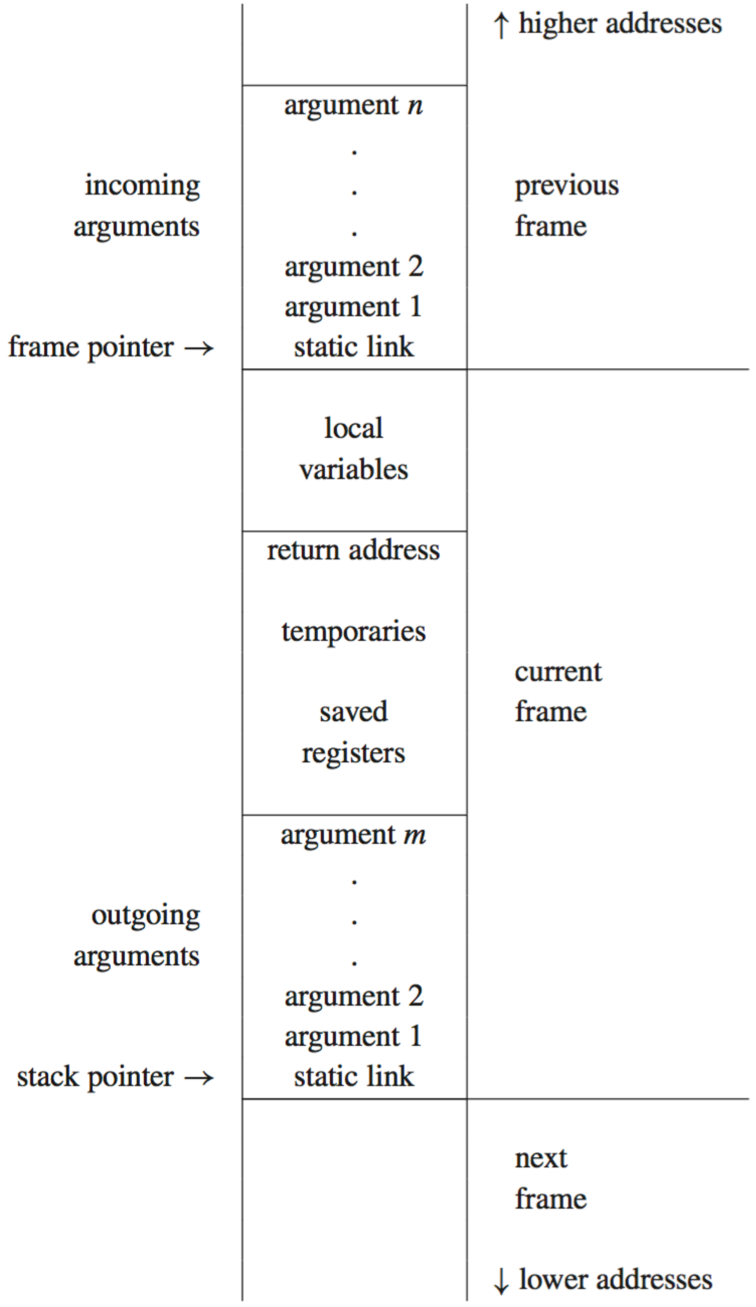


FIGURE 6.2. A stack frame.

Registers

- Most modern machines have a large set of registers.
- Typically around 32 or so.
- <u>MIPS target architecture</u> (our compiler’s back-end) has 32 integer registers.

Why MIPS?

- MIPS is a RISC (**R**educed **I**nstruction **S**et **C**omputer) architecture
  - As opposed to CISC (Complex Instruction Set Computer), e.g. Intel
- RISC is simplified – and also has some advantages – over CISC
  - Only *simple* instructions (code sizes then to be larger…)
  - *All* instructions can be executed in one clock cycle
  - *Fixed* 32 bit instructions (first 6 bits are opcode)
  - Must use LOAD/STORE instructions to copy data to/from registers

```
MUL $r1, $r1, 8
010101 00001 00001 0000000000001000
```

- Intel x86 still uses CISC
- MIPS used in lots of embedded systems (Sony PlayStation 2, Nintendo 64)

When a Function Call Happens:
How to Pass Arguments from Caller to Callee?

Can't we use registers?

- It gets complicated.
- Potential problem: most machine only have *one* set of registers.
- So, different procedures and functions need to use the *same* set of registers.

Example Scenario:

- Method x is:
  1. Storing a local variable in register $r1
  2. Needs to call method y
- Method y also needs to use register $r1

Solution:

- Write x's local variable to memory so that the register value is not overwritten and lost
- Later, restore the value back into the register from memory

Who should save and restore $r1? Method x or method y?

MIPS Calling Convention

Q: Who should save and restore $r1? Method x or method y?

A: It depends.

1. Registers 16-23 (with symbolic names $s0 ... $s7), by convention, should be <u>preserved</u> across function calls:
   - referred to as: <u>callee-save registers</u>
   - It is the callee's responsibility to save/restore registers if it wishes to use $s0 ... $s7.
2. All other registers are not preserved.
   - referred to as: <u>caller-save registers</u>

- Wisely selecting to use callee-save or caller-save registers for local variables and temporaries can reduce number of needed loads/stores (writing variables to memory).
- *Example:* if method x knows that a local variable will not be needed after some method call, it can be placed in a caller-save register -- and purposely not save it to memory before the method call.

Parameter Passing Using Registers

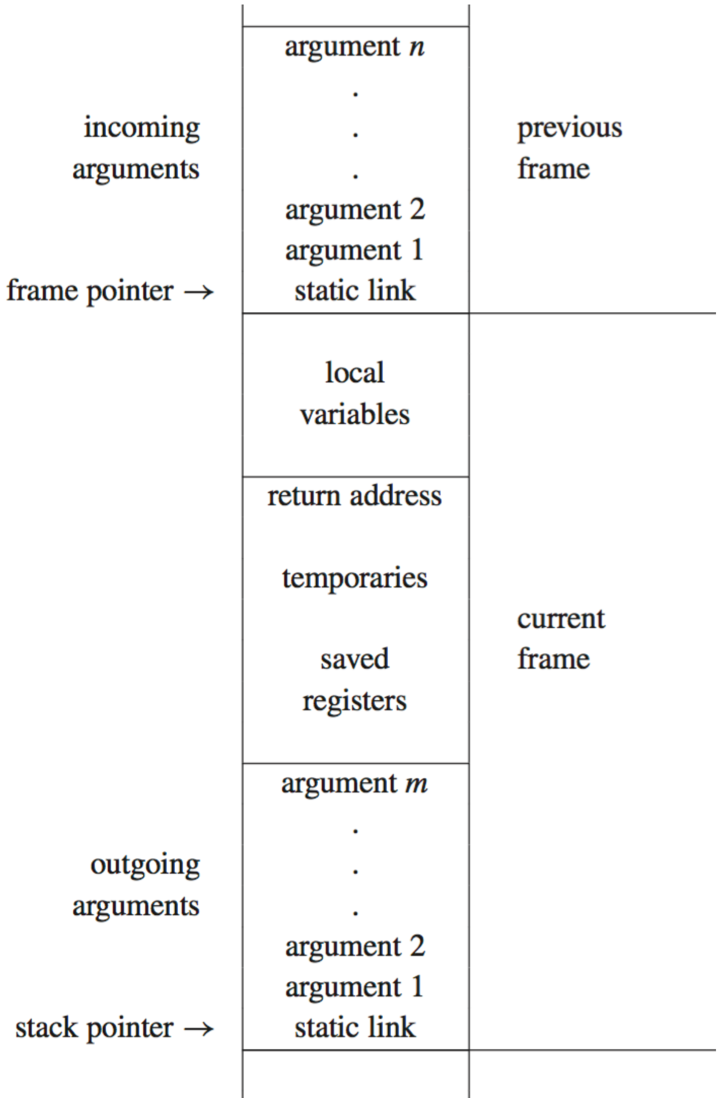
- Would like to pass arguments to subroutines using registers (rather than the stack) to save time.
- On MIPS, 4 registers, by convention, are used for passing arguments:

`$a0`	`arg1`
`$a1`	`arg2`
`$a2`	`arg3`
`$a3`	`arg4`

An Argument Passing Convention Using <u>Both</u> Registers and Memory

<u>*One approach:*</u>

- caller passes arguments $a_1 ... a_4$ in registers
- caller passes arguments $a_5 ... a_n$ at end of its *own* frame (in section marked Outgoing Arguments)
  - becomes Incoming Arguments of callee
- What if callee needs to write incoming register args to memory?
  - When would this happen? If callee needs to call another method.



So, writing to memory cannot be avoided?

But the solution is not terrible.

1. leaf procedure: a procedure that doesn't call another procedure. What proportion of procedures are leaves? It depends.
2. the callee may be finished with its use of an argument by the time it needs to perform a function call

What if callee needs to write incoming register args to memory?

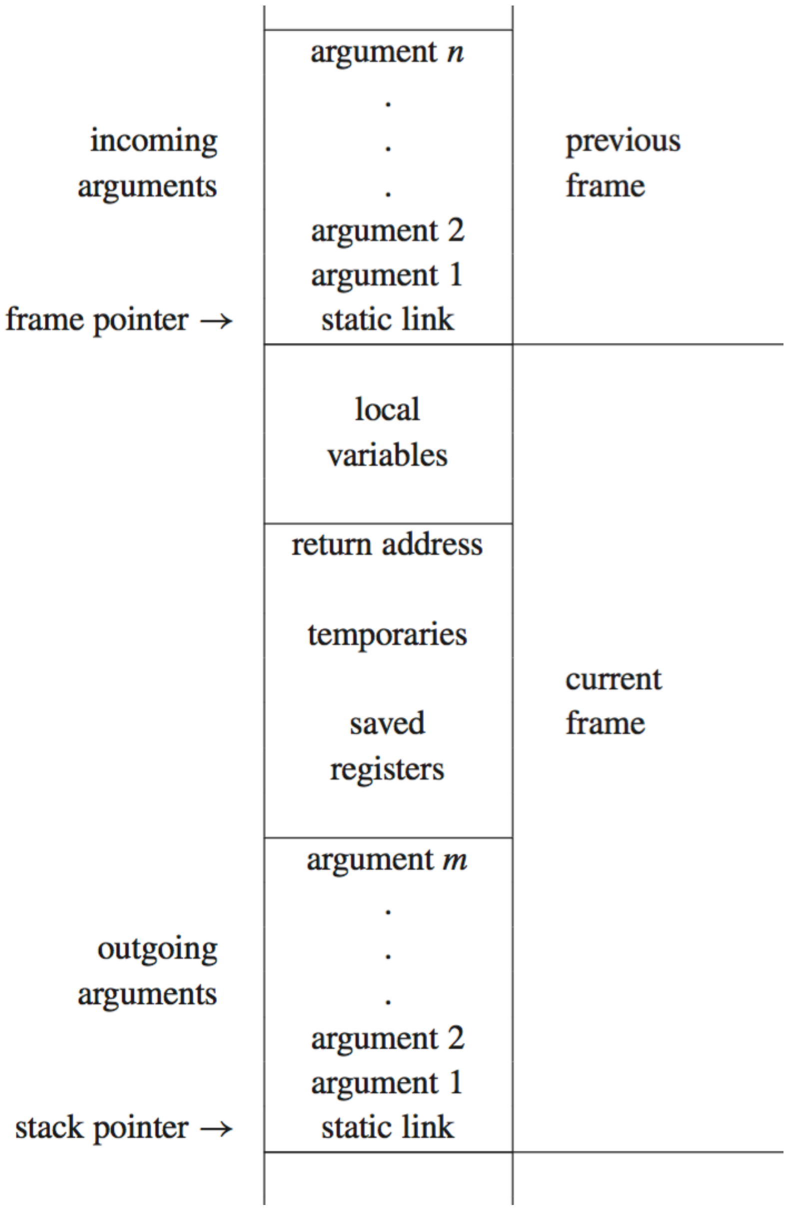
- caller also reserves space in its own frame for arg_1 .. arg_4 (before arg_5)
- doesn't write anything there
- callee can use the designated space to write to memory

Return Address

- When a function returns, where in memory is the next instruction to execute?
- Store the *address* where the next instruction exists.
    1. Either in the activation record
    2. Or, in a register for faster access

```
$ra        # register 31
```

- Non-leaf procedures will *have* to write the return address to the stack frame.



Conclusion

Ideally, most values are maintained in registers, but:

1. Pass By Reference requires storing values in memory – variable "escapes"
2. Spilled Variables: not enough registers exist – value written temporarily to the stack frame