6. Instruction Set Architecture – -from C to Assembly – function calls -translation software

EECS 370 – Introduction to Computer Organization - Winter 2016

Profs. Valeria Bertacco & Reetu Das

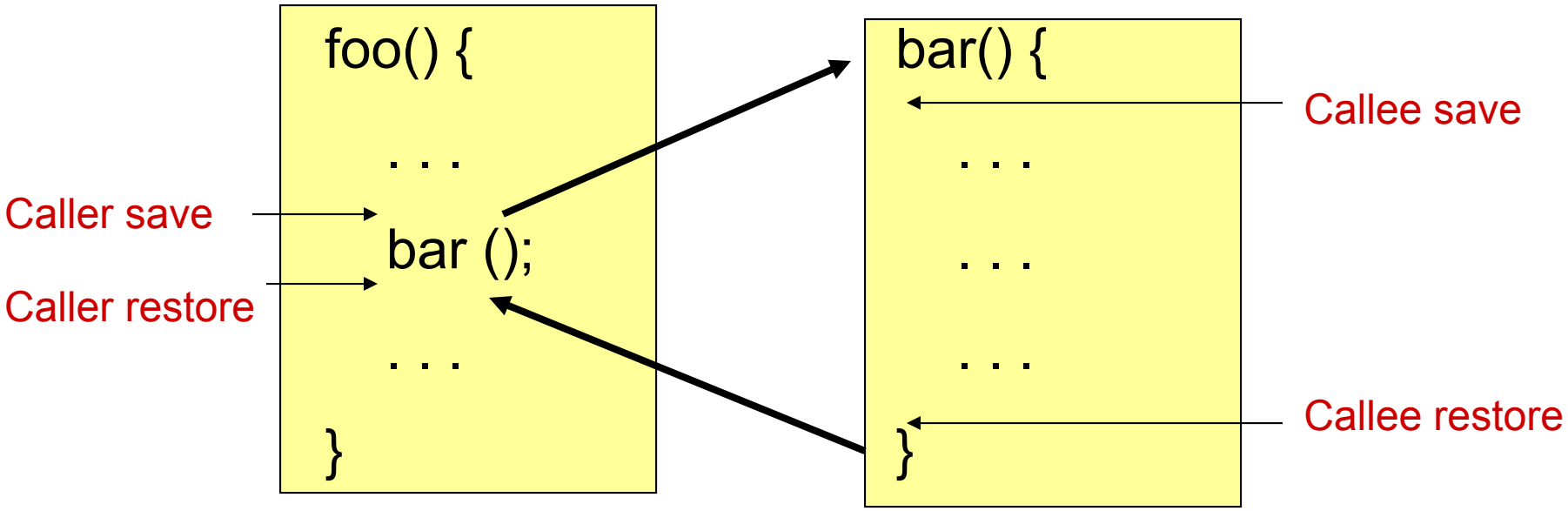
EECS Department
University of Michigan in Ann Arbor, USA

Recap

- C –to assembly: if-then-else, loops
- C-to assembly: function calls:
  - Where do variables go in memory: stack, heap, static or text segment?
  - The inner structure and operation of the mysterious stack frame
  - Caller/callee – which registers will you use for your local variables?

Caller-Callee save/restore

CALLER-CALLEE

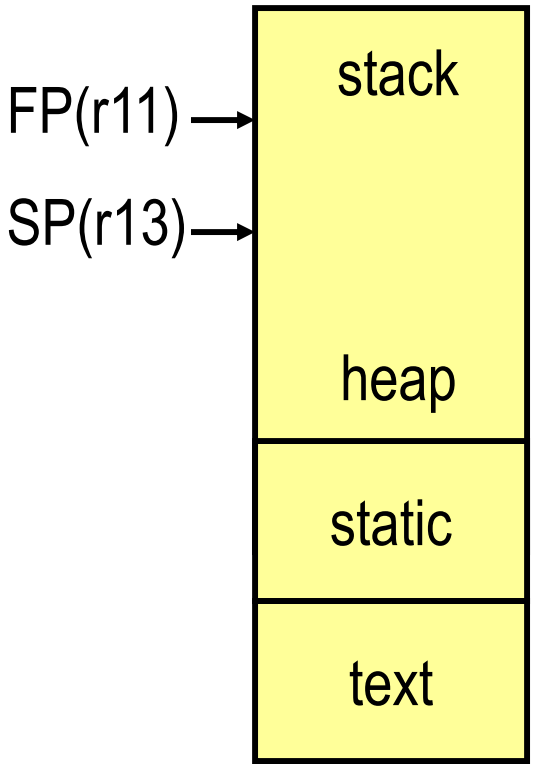


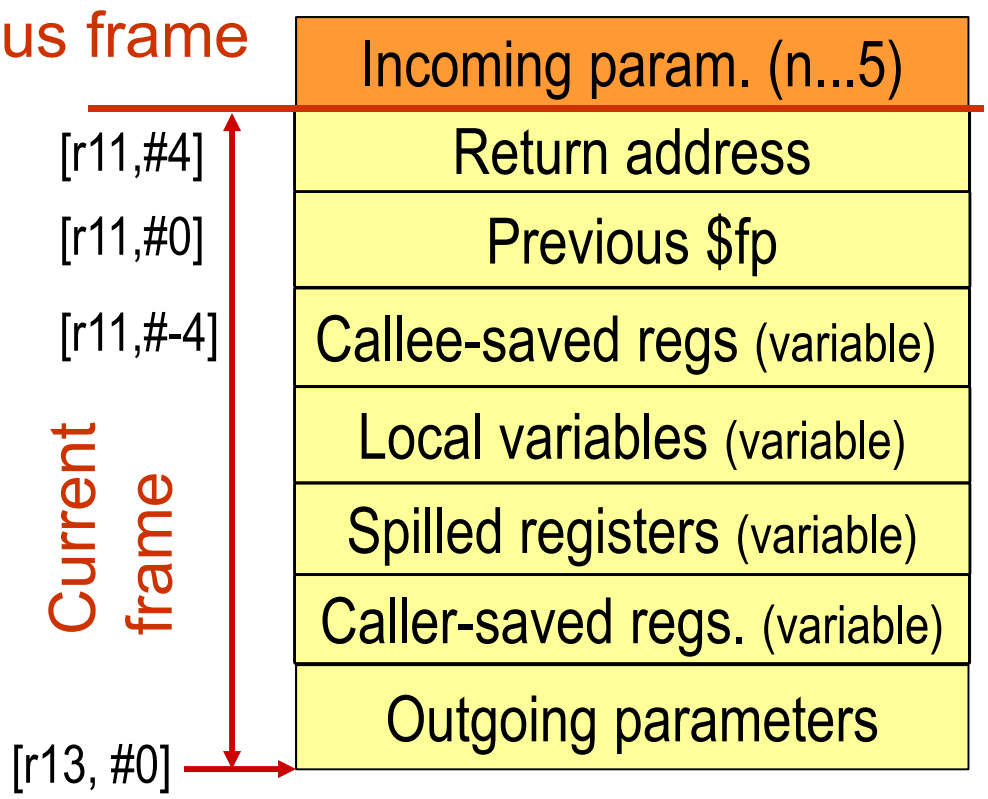
Caller save: Callee may change, so caller responsible for saving immediately before call and restoring immediately after call

Callee save: Callee may not change, so callee (called function) must leave these unchanged. Can be ensured by inserting saves at the start of the function and restores at the end

ARM Stack Frame (typical organization)

CALLER-CALLEE





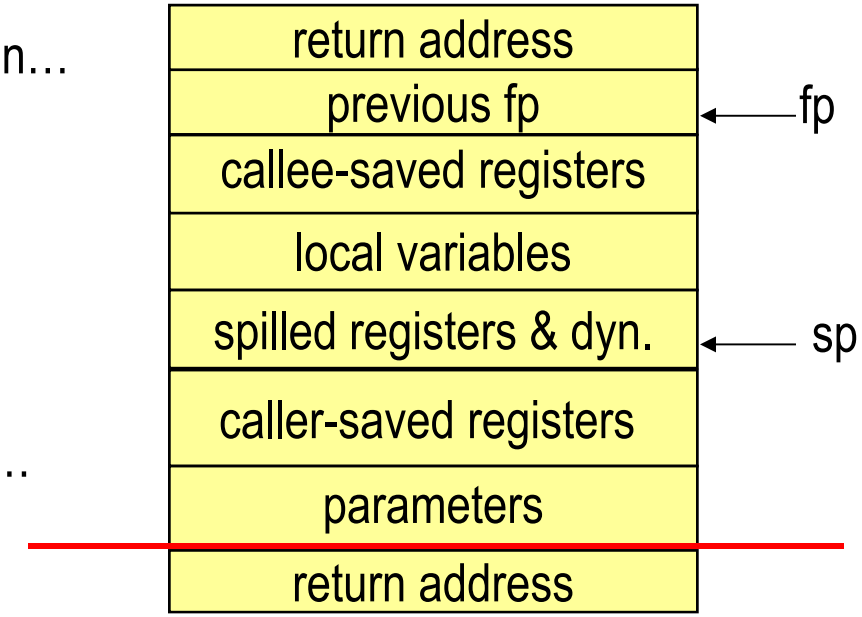
Note 1: in ARM, the first 4 parameters are passed via registers. Other ISAs have ≠ conventions

Note 2: why is the last parameter first on the stack ?

The stack during program execution…

CALLER-CALLEE

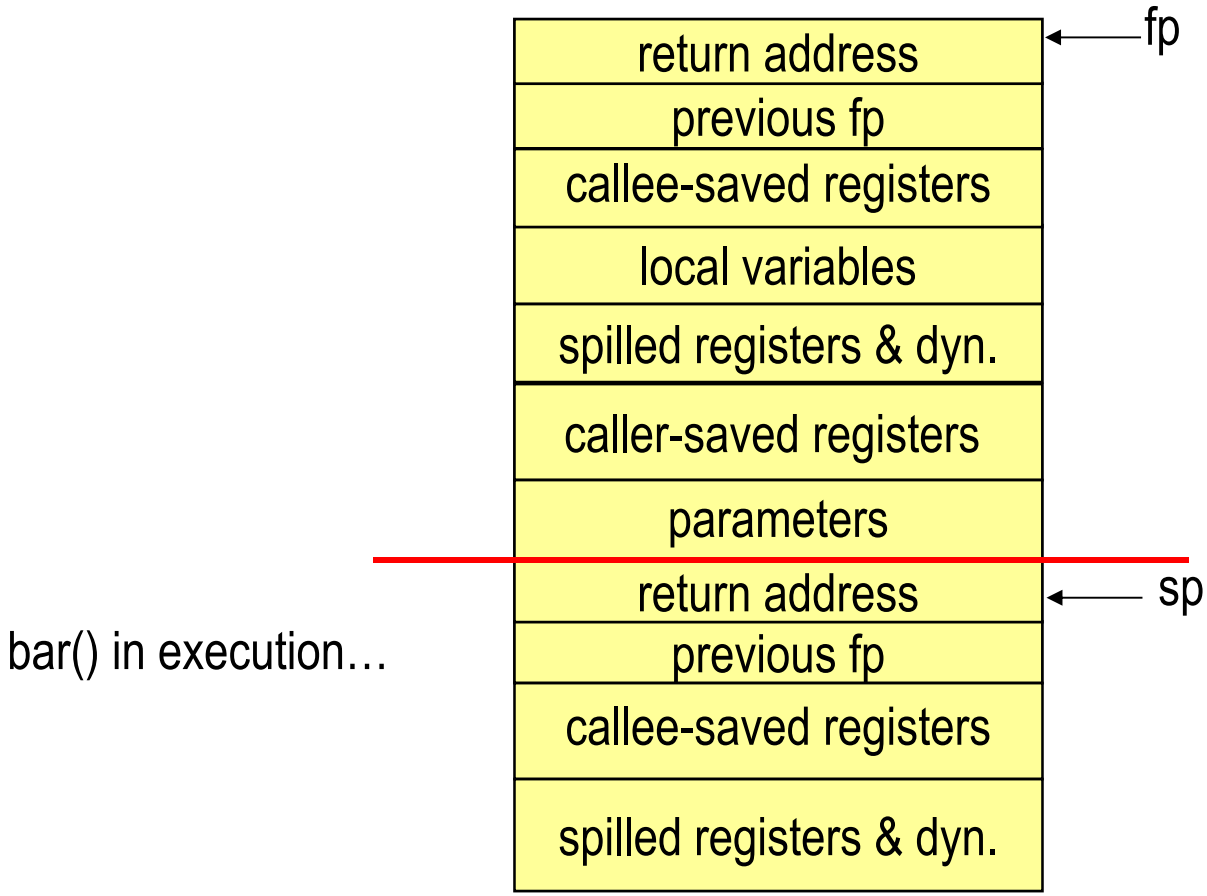
foo() is in execution…



…foo() calls bar…

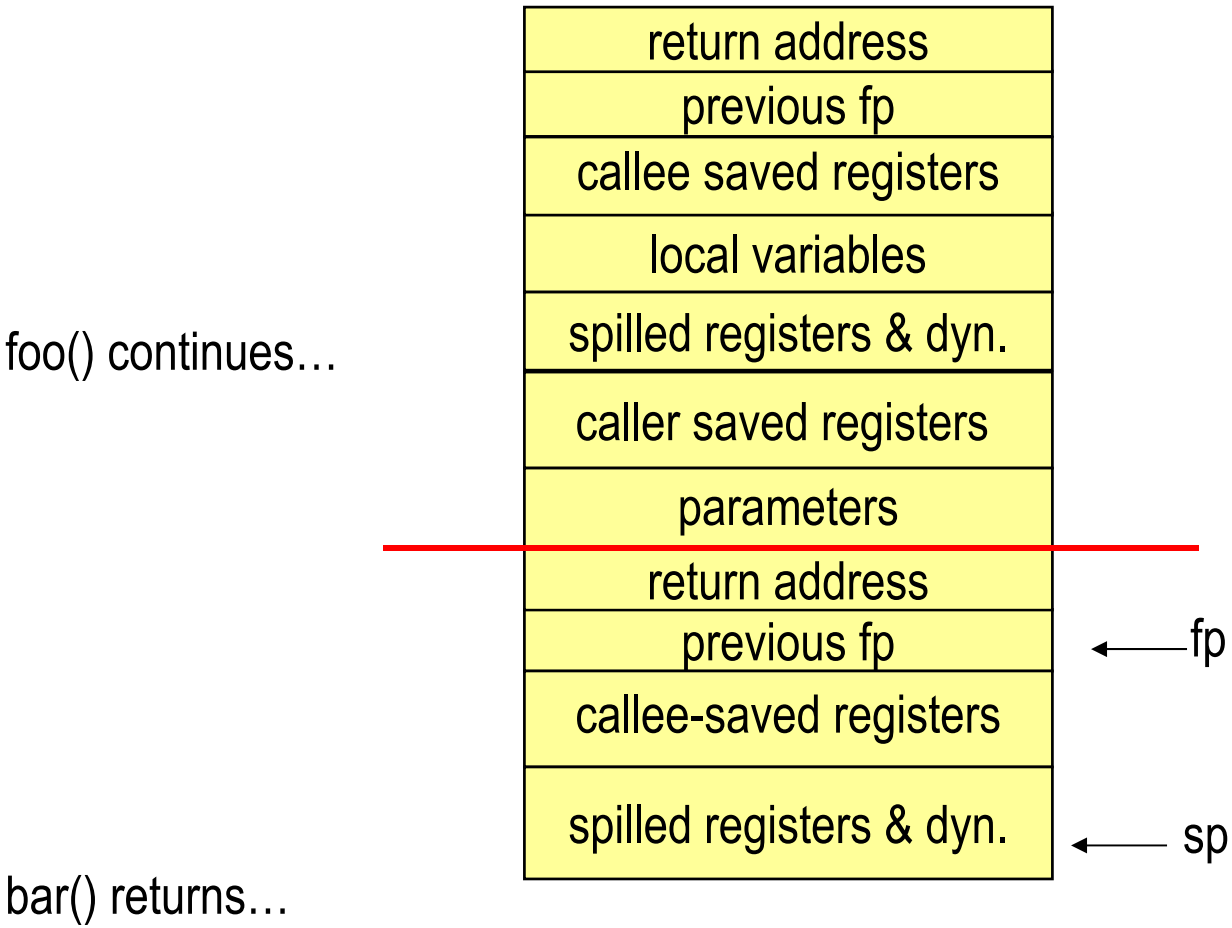
The stack

CALLER-CALLEE



The stack

CALLER-CALLEE



Putting it all together (using activation records)

```
mov   r0, #1000          //  param “hello world\n”
str   r12, [r13, #24]    //  caller save r12
bl    _printf            //  call printf()
                         // r0 holds return value (ignored)
ldr   r12, [r13, #24]    // restore caller-saved r12 value
```

```
sub   r13, r13, #16      // allocate space for 2 locals +
str   r4, [r13, #8]      // callee save r4
str   r14, [r13, #12]    // save return address
…                        // function body
mov   r0, #0             // return value 0
ldr   r4, [r13,#8]       // restore callee-saved r4
ldr   r14, [r13, #12]    // restore return address
add   r13, r13, #16      // deallocate call frame
mov   r15, r14           // return to calling function
```

Calculating Caller/Callee Costs

Consider the cost of placing each variable **v** from function **f** in a callee register and a caller register:

Cost = number of store/load instructions to be executed to accomplish the required saving/restoring

Callee_cost → save at the start of the function, restore at end
= 2 * number of invocations of f

Caller_cost → potentially save/restore across each funct. call in f
Caller cost = 0
For each function call in f, $call_i$
if (v is live) caller_cost += 2 * number of times $call_i$ is executed

Caller/Callee Selection

- Select assignment of variables to registers such that the sum of caller/callee costs is minimized
  - Execute fewest save/restores
- Each function greedily picks its own assignment ignoring the assignments in other functions
  - Calling convention assures all necessary registers will be saved
- 2 types of exam/class problems
  1. Given a single function → Assume it is called 1 time
  2. Set of functions or program → Compute number of times each function is called if it is obvious (i.e., loops with known trip counts or you are told)

Assumptions

CALLER-CALLEE

- A function can be invoked by many different call sites in different functions.

- Assume no inter-procedural analysis (hard problem)
  - A function has no knowledge about which registers are used in either its caller or callee
  - Assume main() is not invoked by another function

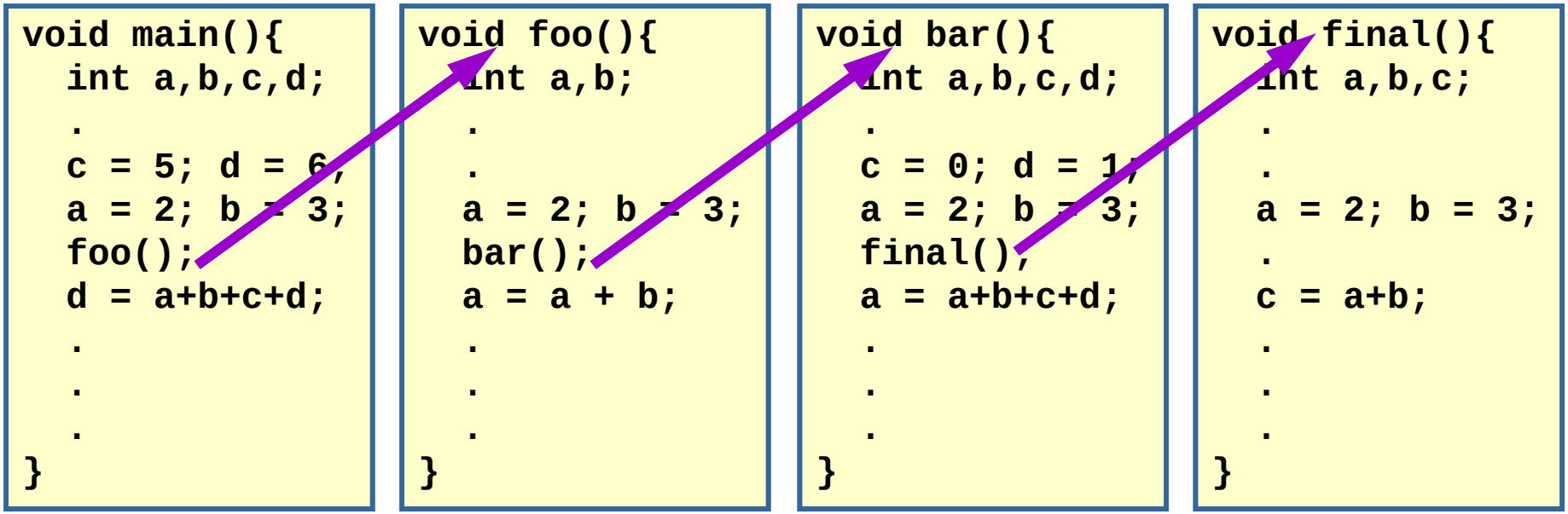
- Implication
  - Any register allocation optimization is done using function local information

Class Problem 3

```
foo() {
    a = …
    b = …
    bar();
    … = a;
    … = b;
    for (1 to 15) {
      c =  …
      d =  …
                  … = c;
      printf();
                  … = d;
    }
}
```

Assume that you have 2 caller and 2 callee save registers. Pick the best assignment for a, b, c, d. Assume each requires its own register.

Caller-saved vs. callee saved – Multiple function case



Note: assume main does not have to save any callee reg. (that is really the case for _start)

Caller-saved vs. callee saved – Multiple function case

Questions:

1. How many store/load instructions are executed in total for registers if we use a caller-save convention ?
2. How many store/load instructions are executed in total for registers if we use a callee-save convention ?
3. How many store/load instructions are executed in total for registers if we use a mixed caller/callee-save convention with 3 callee-s. and 3 caller-s. registers ?

4. Assume bar() is in a loop inside foo() and the loop is iterated 10 times ? When the program is executed, how many regs. need to be stored/loaded in total for each of the above three scenarios?

Question 1: Caller-save

```
void main(){
   .
   .
   .
   [4 str]
   foo();
   [4 ldr]
   .
   .
   .
}
```

```
void foo(){
   .
   .
   .
   [2 str]
   bar();
   [2 ldr]
   .
   .
   .
}
```

```
void bar(){
   .
   .
   .
   [4 str]
   final();
   [4 ldr]
   .
   .
   .
}
```

```
void final(){
   .
   .
   .
   .
   .
   .
   .
   .
}
```

Total: 10 str / 10 ldr

Question 2: Callee-save

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

```
void foo(){
  [2 str]
  .
  .
  .
  bar();
  .
  .
  .
  [2 ldr]
}
```

```
void bar(){
  [4 str]
  .
  .
  .
  final();
  .
  .
  .
  [4 ldr]
}
```

```
void final(){
  [3 str]
  .
  .
  .
  .
  .
  .
  [3 ldr]
}
```

Total: 9 str / 9 ldr

Question 3: Mixed 3 caller / 3 callee

```
void main(){
  .
  .
  .
  [1 str]
  foo();
  [1 ldr]
  .
  .
  .
}
```

1 caller r.
3 callee r.

```
void foo(){
  [2 str]
  .
  .
  .
  bar();
  .
  .
  .
  [2 ldr]
}
```

```
void bar(){
  [3 str]
  .
  .
  [1 str]
  final();
  [1 ldr]
  .
  .
  [3 ldr]
}
```

```
void final(){
  .
  .
  .
  .
  .
  .
  .
  .
}
```

3 caller r.

Total: 7 str / 7 ldr

Caller-saved vs. callee saved – Question 4

- Mixed 3 caller / 3 callee

```
void main(){
  .
  .
  .
  [1 str]
  foo();
  [1 ldr]
  .
  .
  .
}
```

1 caller r.
3 callee r.

```
void foo(){
  [2 str]
  .
  .
  .
  bar();
  .
  .
  [2 ldr]
}
```

loop x10

2 callee r.

```
void bar(){
  [4 str]
  .
  .
  .
  final();
  .
  .
  .
  [4 ldr]
}
```

x10

```
void final(){
  .
  .
  .
  .
  .
  .
  .
  .
}
```

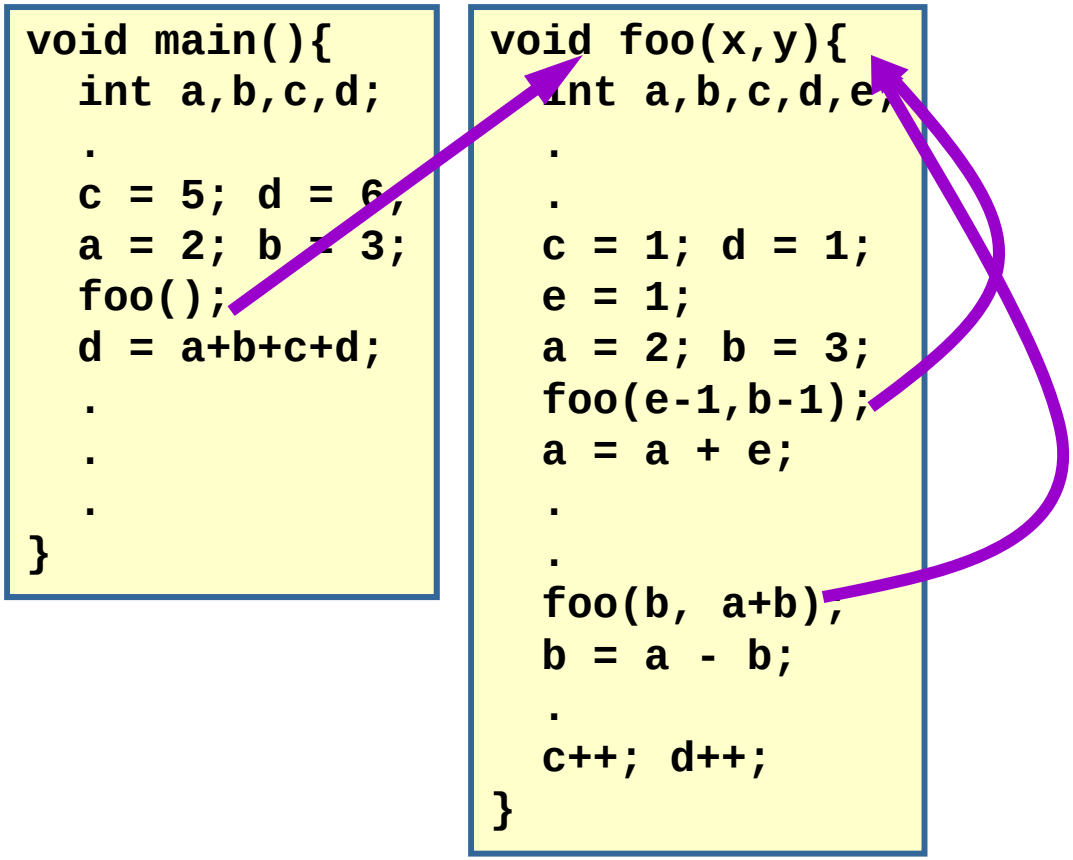
3 caller r.

x10

Total: 43 str / 43 ldr

Pure caller: (4+20+40+0) str / ldr - Pure callee (0+2+40+30) str / ldr

Class problem – Caller-saved vs. callee saved



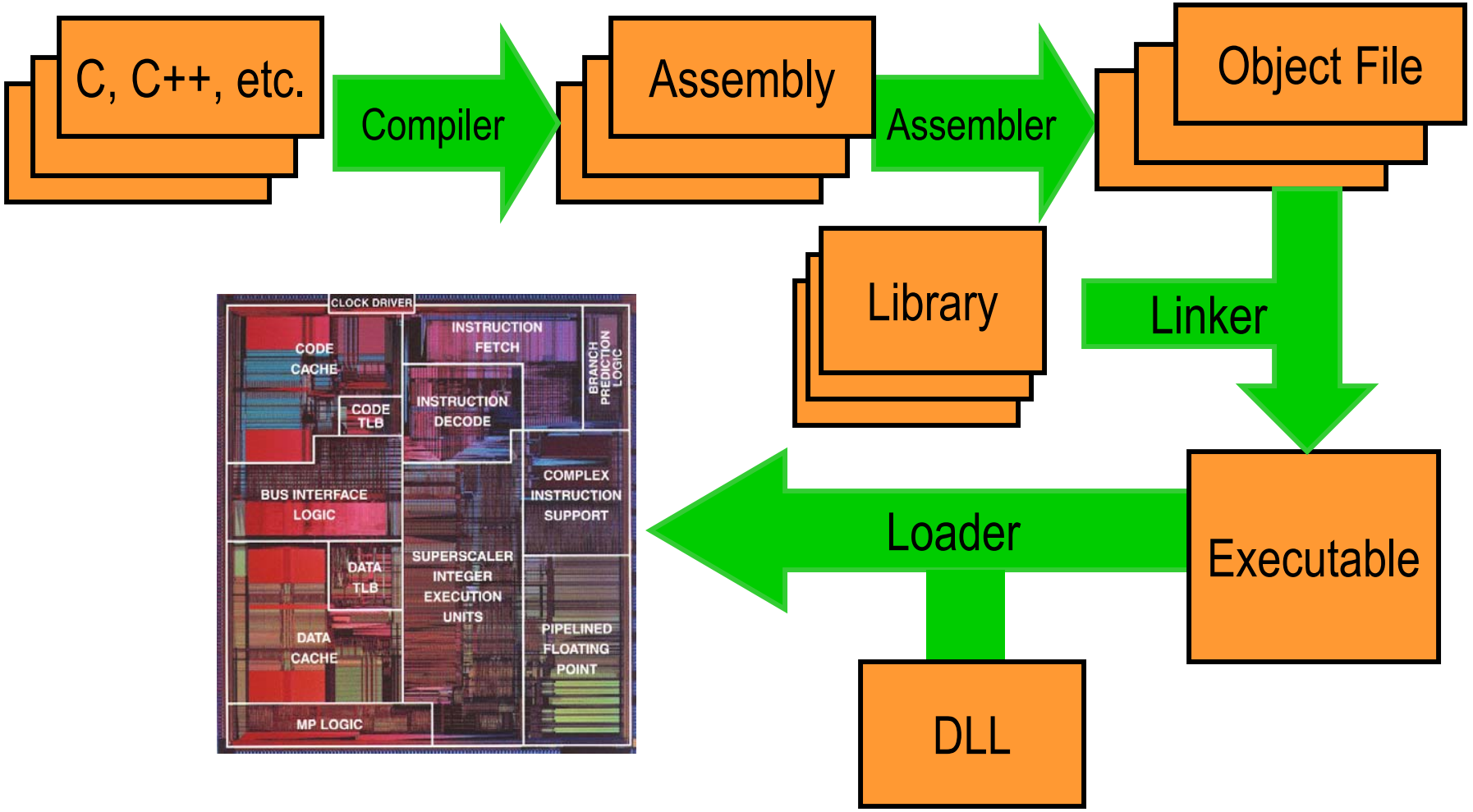
Caller-saved vs. callee saved – the interesting case

- Assume the function foo() is called recursively 14 times (so it is called once from main, and then 7 times from the first call point, 7 times from the second call point)
- When the program is executed, how many regs need to be stored/loaded in total for the following scenarios:
  - Use a **caller-save** convention ?
  - Use a **callee-save** convention ?
  - Use a mixed **caller/callee-save** convention with 3 callee-s. and 3 caller-s. registers ?

Instruction Set Architecture (ISA) Design Lectures

- Lecture 2: Storage types and addressing modes
- Lecture 3 : LC-2K and ARM architecture
- Lecture 4 : Converting C to assembly – basic blocks
- Lecture 5 : Converting C to assembly – functions
- Lecture 6 : Translation software; libraries, memory layout

Source Code to Execution



Compiler

- C, C++, etc… → assembly code
- 2 major parts
  - Frontend
    - Parsing C syntax
    - Source-level analysis
  - Backend
    - Machine independent assembly → optimizations
    - Binding variables/instructions to processor resources
- Relevant courses
  - EECS 483 (Compiler Construction)
  - EECS 583 (Advanced Compilers)

Assembler

- Converts assembly (.s file) to machine code or object file (.o file)
- Generally a simple translation
  - Most assembly instructions map to 1-1 to machine code instructions
  - Pseudo-instructions map to 1 or more machine code instructions
    - Easy to write assembly program with them
    - e.g. ARM pseudo-instr.: “nop” --- assembler translates it to “mov r0, r0”
  - Assembler directives tell the assembler to do something
    - Allocate space for a variable/array/constants
    - Associate a symbol with a value in the symbol table
    - #define variables/labels/registers to convenient names

Assembly Issues

- Reference to a label in another file
  - Data or jump address
  - Only global labels can be referenced from another file
- Assume start at fixed address (say 0) for each procedure
  - But what about multiple procedures?
- Machine code produced by assembler is NOT executable!

Linux (ELF) object file format

Object files contain more than just machine code instructions!

Header: (this is an object file) contains sizes of other parts

Text: machine code

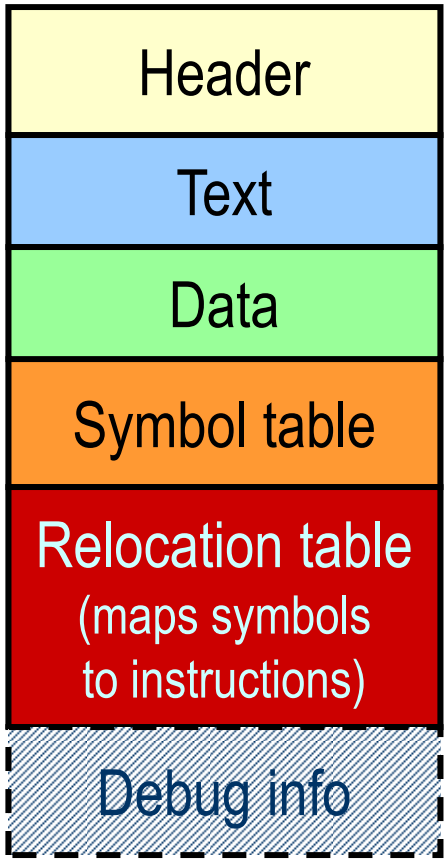
Data: global and static data

Symbol table: symbols and values

Relocation table: references to addresses that may change

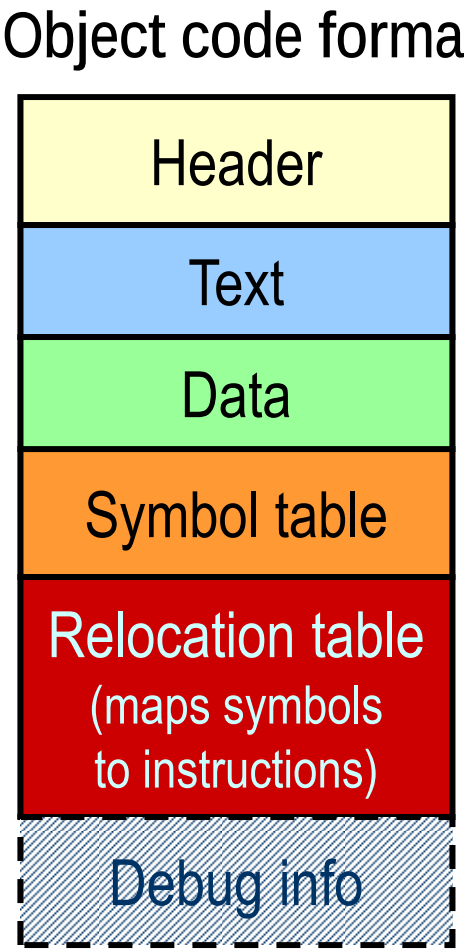
Debug info: mapping of object back to source (only exists when debugging options are turned on)

Object code format



Linux (ELF) object file format (2)

Header
- size of other pieces in file
- size of text segment
- size of static data segment
- size of uninitialized data segment
- size of symbol table
- size of relocation table

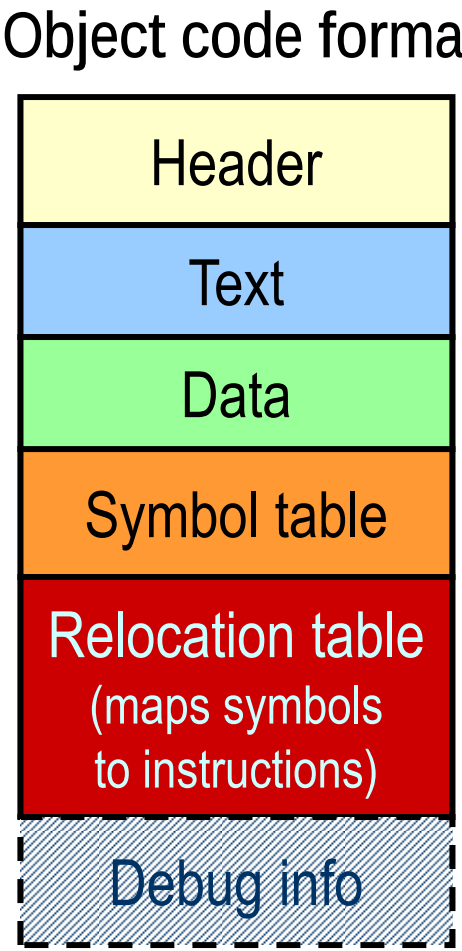


Linux (ELF) object file format (3)

Text segment

- machine code

By default this segment is assumed to be read-only and that is enforced by the OS

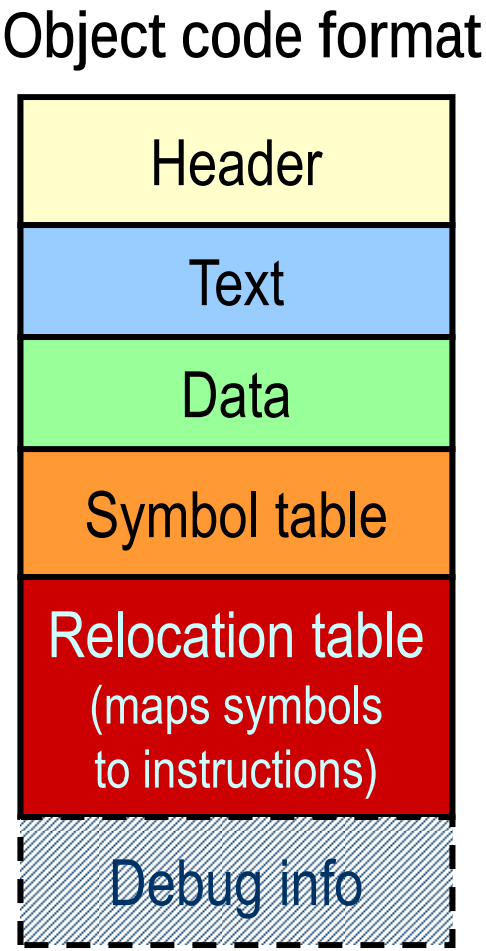


Linux (ELF) object file format (4)

Data segment (Initialized static segment)

- values of initialized globals
- values of initialized static locals

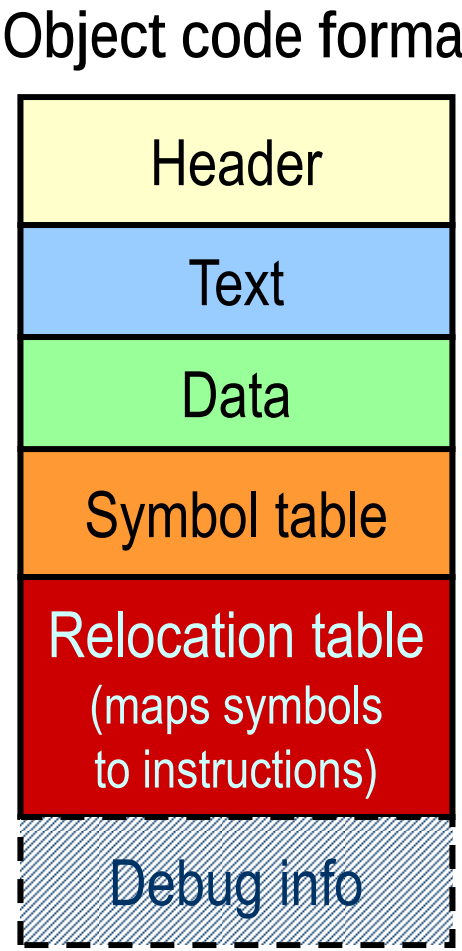
Doesn't contain uninitialized data.
Just keep track of how much memory is needed for uninitialized data



Linux (ELF) object file format (5)

Symbol table:

- It is used by the linker to bind public entities within this object file (function calls and globals)
- Maps string symbol names to values (addresses or constants)
- Associates addresses with global labels. Also lists unresolved labels

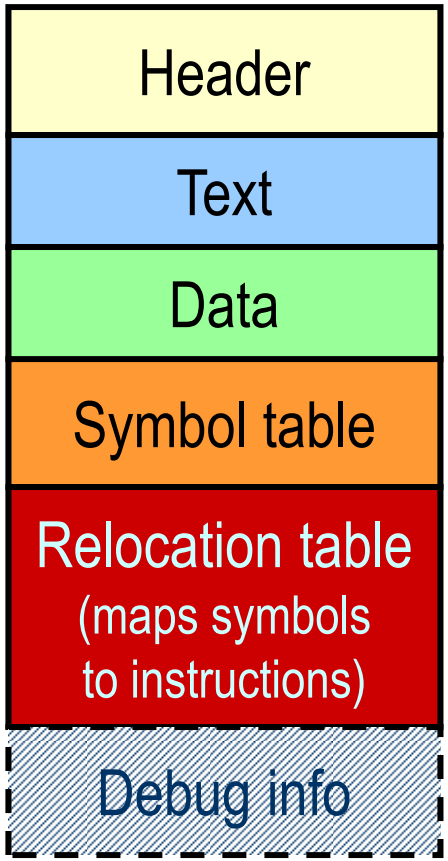


Linux (ELF) object file format (6)

Relocation table :
identifies instructions and data words that rely on absolute addresses. These references must change if portions of program are moved in memory

Used by linker to update symbol uses (e.g., branch target addresses)

Object code format



Linux (ELF) object file format (7)

Debug info (optional) :
Contains info on where variables are in stack frames and in the global space, types of those variables, source code line numbers, etc.
Debuggers use this information to access debugging info at runtime