Compiler Design

Lecture 14: Code generation : Memory management and function call
(EaC Chapter 6&7)

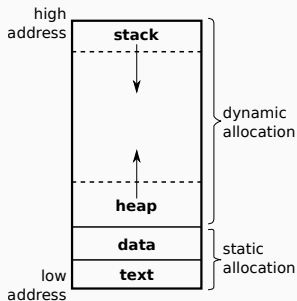Christophe Dubach

Winter 2024

Timestamp: 2024/02/26 09:21:00

Memory management

Static versus Dynamic

- **Static allocation**: storage can be allocated directly by the compiler by simply looking at the program at compile-time. This implies that the compiler can infer storage size information.
- **Dynamic allocation**: storage needs to be allocated at run-time due to unknown size or function calls.
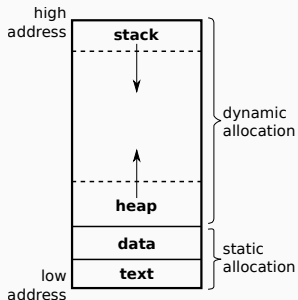
Static storage:

- Text: instructions
- Data
    - global variables
    - string literals
    - global arrays of fixed size
    - . . .

Dynamic Storage:

- Heap: `malloc`
- Stack:
    - local variables
    - function arguments/return values
    - saved registers
    - register spilling (register allocation)

Example


```
char c;                data
int arr[4];            data
void foo() {
  int arr2[3];         stack
  int* ptr =           stack
    (int*) malloc(sizeof(int)*2);   heap
  ...
  {
    int b;             stack
    ...
    bar("hello");      data
  }
  ...
}
```

Memory management

Static Allocation & Alignment

Typically in C:

- int and pointer types (*e.g.* char*, int*, void*) are 32 bits (4 byte).
- char is 1 byte

Example C code (static allocation):

```c
char c;
int i;
```

Assembly code:

```asm
.data
c: .space 1 # char
i: .space 4 # int
.text
lb $t0, c
lw $t1, i    # error!
```

Miss-aligned load, 😠 error!

```asm
.data
c:       .space 1 # char
padding: .space 3
i:       .space 4 # int
.text
lb $t0, c
lw $t1, i    # no error
```

Padding added, all good ☺

Project hint: make use of the .align directive.

In C, typically:

- all fields of a struct should be aligned to their size if smaller than the data alignment of the architecture
  (unless `packed` directive is used); and
- the size of a struct is a multiple of the data alignment of the architecture.

Example C code:

```c
struct myStruct_t {
  char c1;
  int i;
  char c2;
};
struct myStruct_t ms;
...
ms.i
...
```

Assembly code:

```
.data
ms:  .space  12

.text
  ...
  lw $t0, (4)ms
  ...
```

In contrast to structs, arrays are always compact in C.

Example C code (static allocation):

```c
char arr[7];
...
```

Corresponding assembly code:

```
.data
arr: .space 7

.text
...
```

When dealing with allocation of data:

- make sure all declarations/fields are aligned to their size

For variables: padding/alignment may be required before a global or local allocation;

For structs: the total size of the struct and the offset of each field must take into account padding/alignment.

Memory management

Stack allocation

Stack variable allocation

The compiler needs to keep track of local variables in functions.

These cannot be allocated statically (static = text section).

```
int foo(int i) {
  int a;
  a = 1;
  if (i==0)
    foo(1);
  print(a);
  a = 2;
}
void bar() {
  foo(0);
}
```

If a allocated statically, what is printed?

1

2

There must be one copy of a local variable per function call
⇒ local variables must be stored on the stack!

How to keep track of local variables on the stack?
⇒ Could use the `stack pointer`.

⚠ Problem: stack pointer can move, *e.g.*:

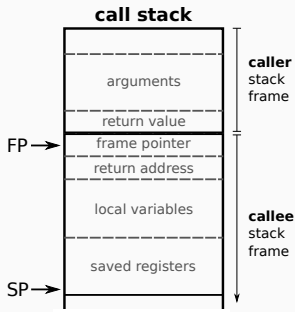- dynamic memory allocation on the stack
- push/pop instructions

Solution: use another pointer, the frame pointer

<div style="background:#e0e0e0">

Frame pointer

- The frame pointer must be initialised to the value of the stack pointer, just when entering the function.
- Access to variables allocated on the stack can then be determined as a fixed offset from the frame pointer.

</div>

The **frame pointer** (FP) always points to the start of the *callee* stack frame.

The **stack pointer** (SP) always points at the top of the stack (points to the last element pushed).

call stack
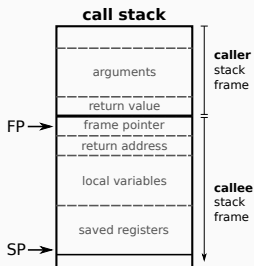
arguments	**caller** stack frame
return value	
FP → frame pointer	
return address	
local variables	**callee** stack frame
saved registers	
SP →	

Memory management

Memory allocation pass

Memory Allocator

call stack

FP →

SP →

caller stack frame	callee stack frame

arguments

return value

frame pointer

return address

local variables

saved registers

Args in reverse order from FP

```
class Allocator {
 int fpOffset;
 boolean insideFunDecl = false;

 void visit(ASTNode node) {
  switch(node) {
   case FunDecl fd -> {

    fd.retValFPOffset = +4;

    int offset = 4+fd.returnSize;
    for (param: fd.params.reverse) {
      param.fpOffset = offset;
      offset += param.size;
    }

    this.fpOffset = -4;
    this.insideFunDecl = true;
    visit(fd.body);
    this.insideFunDecl = false;
  }
  ...
```
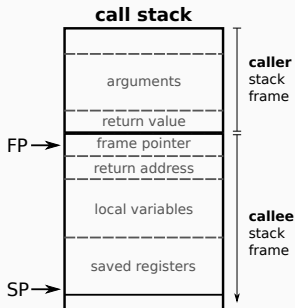
Memory Allocator

call stack

arguments
return value
frame pointer
return address
local variables
saved registers

FP → frame pointer

SP →

caller stack frame

callee stack frame

Low address is "start" of variable
(think of structs!)

```
...
case VarDecl vd -> {
  if (insideFunDecl) {// local/stack
    this.fpOffset -= vd.size;
    vd.fpOffset = this.fpOffset;
  } else {// global/static
    Label label = new Label(vd.name);
    ... // emit .space label size
    vd.label = label;
  }
}
case default -> {
  for (c: node.children)
    visit(c);
}
...
```

Memory management

Address of expressions

Generating addresses

Sometimes, the compiler needs to know the address of an expression (*e.g.* assignment, address-of operator):

```
struct vec_t {
  int x;
  int y;
};

void foo() {
  int i;
  struct vec_t v;
  int arr[10];

  i = 2;
  v.x = 3;
  arr[2] = 5;
}
```

Generating addresses

Specialized generator that produces the address of an expression:

Address Code Generator

```
class AddrCodeGen {

  Register visit(Expr e) {
    return switch(e) {
      case BinOp bo -> {
        error("No address for BinOP"); yield INVALID;}
      case IntLiteral il -> {
        error("No address for IntLiteral"); yield INVALID;}
      case VarExpr v -> {
        Register resReg = newVirtualRegister();
        if (v.vd.isStaticAllocated())
          emit("la", resReg, v.vd.label);
        else if (v.vd.isStackAllocated())
          emit("addi", resReg, Register.fp, v.vd.fpOffset);
        yield resReg;
      }
      ...
    }
  }
```

AddrCodeGen is called from the other code generator when needed:

```
ExprCodeGen
 ...
case Assign a -> {
   Register addrReg = (new AddrCodeGen()).visit(a.lhs);
   Register valReg  = visit(a.rhs);
   emit("sw", valReg, addrReg);
   return valReg;
}
...
```

Examples above have assumed variable stores an interger.

In case a variable represents an array or struct, you have to be careful:

- array are passed by reference
  (treat them exactly like pointers, no big deal)
- structs are passed by values

Assigning between two structs means copying field by field. Hence your code generator must check the type of variables when encountering an assignment and handle structures correctly.

Similar problem for struct and function call. Arguments and return values that are struct are passed by values.

Code with struct assignment:

```
struct vec_t {
  int x;   int y;
};
void foo() {
  struct vec_t v1;
  struct vec_t v2;
  v1 = v2;
}
```

Equivalent to:

```
struct vec_t {
  int x;   int y;
};
void foo() {
  struct vec_t v1;
  struct vec_t v2;
  v1.x = v2.x;
  v1.y = v2.y;
}
```

♀ Tip for coursework

The register returned by the ExprCodeGen visit method holds the expression's value. However, a structure cannot be held in a register.

- As a special case, the ExpCodeGen's visit method should return the address of a `StructType` expr;
- And the AddrCodeGen will handle the assignment (and other things).

```
class ExprCodeGen {
  Register visit(Expr e) {
    if (e.type == StructType)
      return (new AddrCodeGen()).visit(e);
    ...
} }
```

```
class AddrCodeGen {
  Register visit(Expr e) {
    return switch(e) {
      case Assign a -> {
        Register lhsAddrReg = visit(a.lhs);
        if (a.type == StructType) {
          Register rhsAddrReg = visit(a.rhs);
          // copy the struct word by word
          ...
        } else { // not a struct
          Register rhsValueReg = (new ExprCodeGen()).visit(a.rhs);
          // copy the value of the rhs into memory at address lhs
          ...
        }
        yield lhsAddrReg;
 } } }
```

Function calls

```
int bar(int a) {
    return 3+a;
}
void foo() {
    ...
    bar(4)
    ...
}
```

- foo is the caller
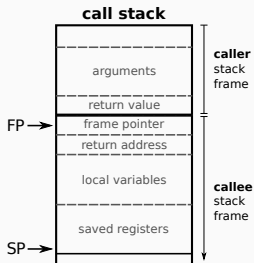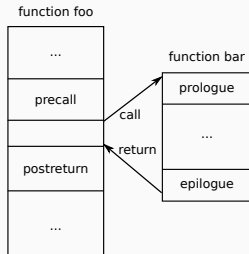- bar is the callee

What happens during a function call?

- The caller needs to pass the arguments to the callee
- The callee needs to pass the return value to the caller
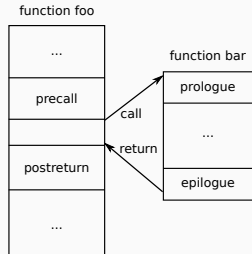
But also:

- The values stored in registers needs to be saved somehow.
- Need to remember where we came from to return to the call site.

function foo

...

precall

postreturn

...

function bar

prologue

...

epilogue

call

return

Possible convention: **Callee-save**
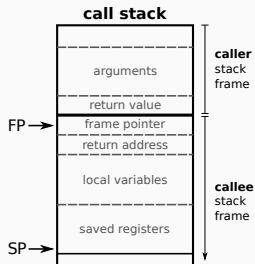
- precall:
  - pass the arguments via registers or push on the stack
  - reserve space on stack for return value (if needed)

- postreturn:
  - read the return value from dedicated register or stack
  - reset stack pointer

call stack

arguments	**caller** stack frame
return value	
FP → frame pointer	
return address	**callee** stack frame
local variables	
saved registers	
SP →	

function foo

function bar

- prologue:
    - push frame pointer onto the stack
    - initialise the frame pointer
    - push return address on the stack
      (if function makes call)
    - reserve local variables space on stack
    - push saved registers on stack

- epilogue:
    - restore saved registers from the stack
    - restore return address from the stack
      (if function makes call)
    - restore the stack pointer
    - restore the frame pointer from the stack
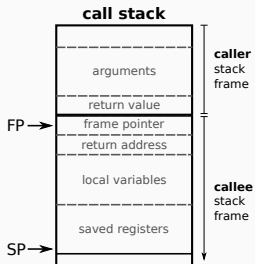
♀ Other conventions are possible.

For your project, we suggest you:

- pass all arguments and return value via the stack
  (needed anyway when more than four arguments);
- always push/restore the return address
  (needed anyway when the function calls another one).

Example (callee)

```
int bar(int a) {
    int b;
    return 3+a;
}
```

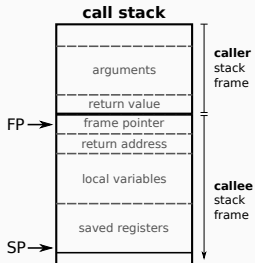**call stack**

Assembly code:

```
bar:

addi $sp, $sp, -4     #
sw   $fp, ($sp)       # push frame pointer on the stack

move $fp, $sp         # initialise the frame pointer

addi $sp, $sp, -4     #
sw   $ra, ($sp)       # push return address on stack

addi $sp, $sp, -4     # reserve space on stack for b

addi $sp, $sp, -4     #
sw   $t0, ($sp)       # push $t0 onto the stack
addi $sp, $sp, -4     #
sw   $t1, ($sp)       # push $t1 onto the stack

li   $t0, 3           # load 3 into $t0

lw   $t1, 8($fp)      # load first argument from stack
add  $t0, $t0, $t1    # add $t0 and first argument

sw   $t0, 4($fp)      # copy the return value on stack

lw   $t0, -12($fp)    # restore $t0
lw   $t1, -16($fp)    # restore $t1

addi $sp, $fp, 4      # restore stack pointer

lw   $ra, -4($fp)     # restore return address from the stack

lw   $fp, ($fp)       # restore the frame pointer

jr   $ra              # jumps to return address
```

26

Example (caller)

```
void foo() {
  ...
  bar(4)
  ...
}
```

Assembly code:

```
foo:
...

li    $t0, 4            # prepare argument

addi  $sp, $sp, -4      #
sw    $t0, ($sp)        # push argument on stack

addi  $sp, $sp, -4      # reserve space on stack for return value

jal   bar               # call function

lw    $t0, $sp          # read return value from stack

addi  $sp, $sp, 8       # reset stack pointer
...
```

call stack

arguments	**caller** stack frame
return value	
FP → frame pointer	
return address	
local variables	**callee** stack frame
SP → saved registers	

Beware of value semantic with functions!

Structs need to be passed by value in a function call.

C code example:

```c
struct vec_t {
  int x;
  int y;
}

int foo(struct vec_t v) {
  return v.x;
}

void bar() {
  struct vect_t myvec;
  int i;
  i = foo(myvec);
}
```

Beware of value semantic with functions!

Structs needs to be returned by value.

C code example:

```c
struct vec_t {
  int x;
  int y;
}

struct vec_t foo() {
  struct vec_t v;
  v.x = 0; v.y = 1;
  return v;
}

void bar() {
  struct vect_t myvec;
  myvec = foo();
}
```

💡 Coursework tip

Again, structs have to be handled differently with function calls.

- If passed as an argument, they must be copied word by word onto the stack;
- When a `return` statement is encountered, they must be copied word by word onto the stack;
- If returned from a function, they might have to be copied word by word from the stack (*e.g.* assignment).

Naive register allocator.