Compiler Design

Lecture 15: Naive register allocator

Christophe Dubach

Winter 2023

Timestamp: 2023/02/19 17:03:00

The need for register allocation

So far, we have assumed that we have access to an unlimited set of registers: virtual registers.

- This simplifies greatly the design of the code generator.

Problem: real machines have a finite set of architectural registers

Proper register allocation

Map all virtual registers onto the architectural registers (if possible).

Naive "register allocation"

Just make it work.

Naive register allocator

Let's not try to be smart $\Rightarrow$ naive approach.

Main idea:

- map each virtual register to a static memory location using a label;
- use `load`/`store` instructions to read/write the value of the virtual register.

Code generated with virtual registers

C expression:

```
int a; // static allocation
int c; // static allocation
...
2+a-c
```

⇒

Generated code

```
.data
a: .space 4
c: .space 4

.text
li  v0, 2
lw  v1, a
add v2, v0, v1
lw  v3, c
sub v4, v2, v3
```

Def & Use set

Assembly instructions can:

- define registers: write values into them
- use registers: read their values

Example:

```
add v2, v0, v1
```

- `Def(insn)` = {`v2`}
- `Use(insn)` = {`v0, v1`}

Naive register allocator

Our naive register allocator will emit:

- a `load` instruction for each register $\in$ `Use(insn)`
- a `store` instruction for each register $\in$ `Def(insn)`

Generated code
(with virtual registers)

```
.data
a: .space 4
c: .space 4

.text
li  v0, 2
lw  v1, a
add v2, v0, v1
lw  v3, c
sub v4, v2, v3
```

⇒

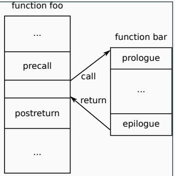
After naive register allocation

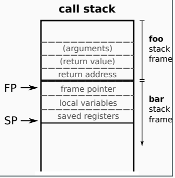
```
.data
a:  .space 4
c:  .space 4
v0: .space 4
v1: .space 4
v2: .space 4
v3: .space 4
v4: .space 4

.text
# li  v0, 2
li  $t0, 2
sw  $t0, v0

# lw  v1, a
lw  $t0, a
sw  $t0, v1

# add v2, v0, v1
lw  $t0, v0
lw  $t1, v1
add $t2, $t0, $t1
sw  $t2, v2

# lw  v3, c
lw  $t0, c
sw  $t0, v3

#sub v4, v2, v3
lw  $t0, v2
lw  $t1, v3
sub $t2, $t0, $t1
sw  $t2, v4
```

Dealing with function calls

In our previous lecture, we have seen that all the registers used by a function should be saved on the stack.

- prologue:
  - push frame pointer onto the stack
  - initialise the frame pointer
  - reserve space on the stack for local variables
  - save all the saved registers onto the stack
- epilogue:
  - restore saved registers from the stack
  - restore the stack pointer
  - restore the frame pointer from the stack





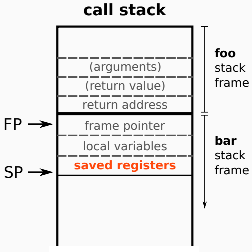
Problem: during code generation, we do not yet know how many registers will be used by the function.

⇒ This depends on the register allocator.

Solution:

- Save the used registers on the stack after the local variables;
- Use two “pseudo-instructions” that pushes and retrieve all the used registers on the stack:
  - `pushRegisters`
  - `popRegisters`

  Then, let the register allocate replace these pseudo-instructions with actual instructions once we know which registers are used.



Example : callee revisited

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

```
bar:

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

move $fp, $sp       # initialise frame pointer

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

pushRegisters

li   v0, 3

lw   v1, 12($fp)   # load first argument
add  v2, v0, v1

sw   v2, 8($fp)    # copy return value on stack

popRegisters

addi $sp, $sp, 16   # restore stack pointer

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

jr   $ra            # jumps to return address
```

After naive register allocation:

```
.data
v0: .space 4
v1: .space 4
v2: .space 4

.text
bar:
addi $sp, $sp, -4   #
sw   $fp, ($sp)     # push frame pointer
move $fp, $sp       # initialise frame pointer
addi $sp, $sp, -4   # reserve stack space for b
pushRegisters

li   $t0, 3         # li   v0, 3
sw   $t0, v0

lw   $t0, 12($fp)   # lw   v1, 12($fp)
sw   $t0, v1

lw   $t0, v0        # add  v2, v0, v1
lw   $t1, v1
add  $t2, $t0, $t1
sw   $t2, v2

lw   $t0, v2        # sw   v2, 8($fp)
sw   $t0, 8($fp)

popRegisters
addi $sp, $sp, 16   # restore stack pointer
lw   $fp, ($fp)     # restore frame pointer
jr   $ra            # jumps to return address
```

Example : callee revisited

After naive register allocation:

```
.data
v0: .space 4
v1: .space 4
v2: .space 4

.text
bar:
addi $sp, $sp, -4   #
sw   $fp, ($sp)     # push frame pointer
move $fp, $sp       # initialise frame pointer
addi $sp, $sp, -4   # reserve stack space for b
pushRegisters

li   $t0, 3         # li   v0, 3
sw   $t0, v0

lw   $t0, 12($fp)   # lw   v1, 12($fp)
sw   $t0, v1

lw   $t0, v0        # add  v2, v0, v1
lw   $t1, v1
add  $t2, $t0, $t1
sw   $t2, v2

lw   $t0, v2        # sw   v2, 8($fp)
sw   $t0, 8($fp)

popRegisters
addi $sp, $sp, 16   # restore stack pointer
lw   $fp, ($fp)     # restore frame pointer
jr   $ra            # jumps to return address
```

After expansion:

```
.data
v0: .space 4
v1: .space 4
v2: .space 4

.text
bar:
...

# pushRegisters
lw   $t0, v0        # load content of v0
addi $sp, $sp, -4   #
sw   $t0, ($sp)     # push v0 onto the stack
lw   $t0, v1        # load content of v1
addi $sp, $sp, -4   #
sw   $t0, ($sp)     # push v1 onto the stack
lw   $t0, v2        # load content of v2
addi $sp, $sp, -4   #
sw   $t0, ($sp)     # push v2 onto the stack

...

# popRegisters
lw   $t0, 0($sp)   # restore v2
sw   $t0, v2
lw   $t0, 4($sp)   # restore v1
sw   $t0, v1
lw   $t0, 8($sp)   # restore v0
sw   $t0, v0

...
```

All this looks horribly inefficient… but it works!

We will see later how “proper” register allocation actually works (and you will implement it in part 4).

Next lecture

- Control-Flow Graph / Basic blocks
- Liveness Analysis
- Proper register allocation