CS 536 Announcements for Monday, April 15, 2024

Last Time

- compiler backend design issues
  - we're going directly from AST to machine code
- start looking at code generation
  - global variables
  - function preamble
- start looking at details of MIPS

Today

- continue code generation
  - function declaration
  - function call and return
  - expressions
  - literals
  - assignment
  - I/O

Next Time

- wrap up code generation
  - tuple access
  - control-flow constructs

Recall

Global variables

- one way

```
        .data
        .align 2
_name:  .space 4
```

← # bytes

- simpler form for primitives ← fine for P6 (base)

```
        .data
_name:  .word value
```

↳ initial value

Function Declarations

Need to generate

- preamble
- prologue
- body
- epilogue

See last lecture for special handling of main's preamble

Preamble

```
integer f{integer a, integer b}[
   integer c.
   c = a + b - 7.
   return c.
]
```

```
.text
_f:
# ... function body ...
```

This label (_f) gives us a place to jump to: jal _f

Prologue

Need to

1. save the return address

```
sw $ra, 0($sp)
subu $sp, $sp, 4
```

push $ra onto stack

2. save the frame pointer

```
sw $fp, 0($sp)
subu $sp, $sp, 4
```

push $fp onto stack

3. update the frame pointer

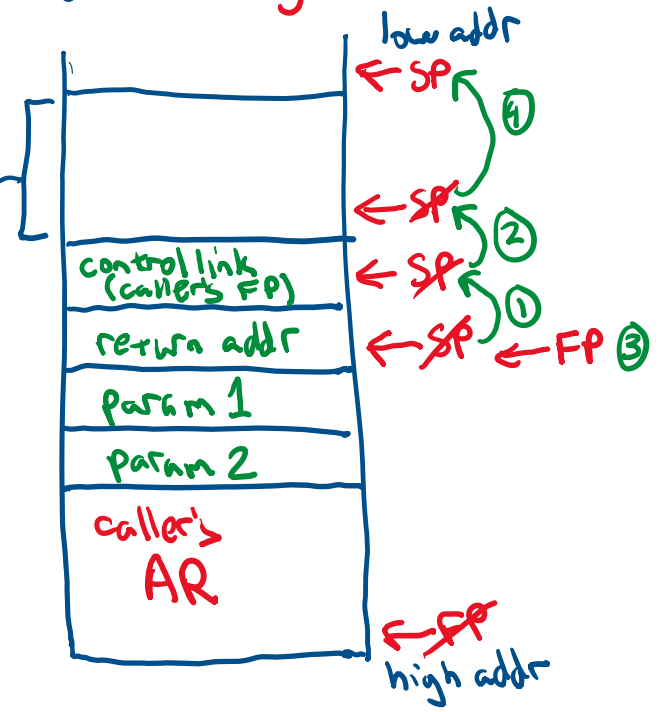
```
addu $fp, $sp, 8
```

$fp = $sp + 8

4. make space for locals

```
subu $sp, $sp, size
```

not initialized



During name analysis
- compute offsets of params & locals
- extend to compute size of all locals

Function Declarations (cont.)

Epilogue

Need to

1. restore return address

```
lw $ra, 0($fp)
```

2. restore the frame pointer

```
move $t0, $fp
lw $fp, -4($fp)
```

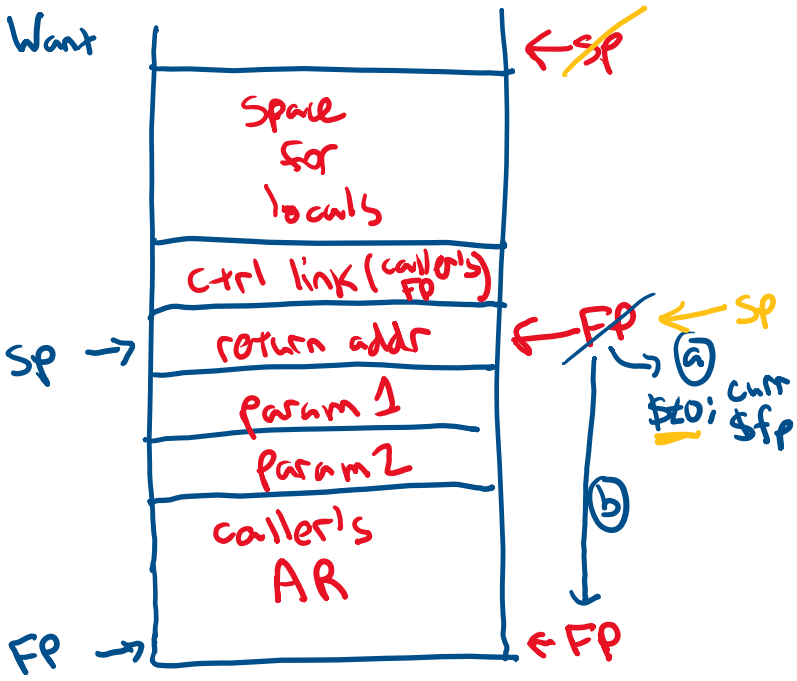
3. restore the stack pointer

```
move $sp, $t0
```

4. return control

```
jr $ra
```

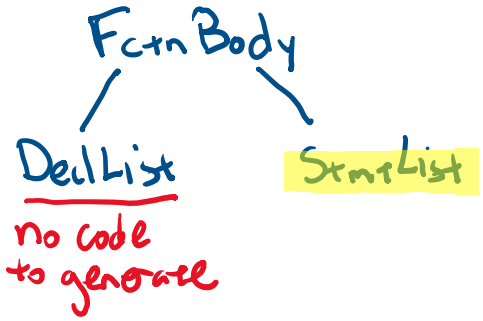
① $ra: addr to return to



Body of function

Generate code for each statement in StmtListNode

- higher-level data constructs
  - loading parameters, setting return
  - evaluating expressions
- higher-level control constructs
  - performing a function call
  - while loops
  - if-then and if-then-else statements



Accessing local variables and parameters

```
lw $t0, offset($fp)
```

sw $t0, offset($fp)

→ positive for params
negative for locals

Function Returns

Function returns when

- hit a return statement
- "fall off" end of function body

Approach

- label epilogue

```
_fctnName_exit:
# ... epilogue ... #
```

- have each return jump to label

```
# ... prologue ... #
    ...
# ... function body ... #
    ...
# code for evaluating return expression
    ...
lw $v0, 4($sp)
addu $sp, $sp, 4

j _fctnName_exit
```

if have a return value

& leaving result on stack

pop stack into $v0

jump to epilogue

About functions that return a value…

```
void main{} [
    integer x.
    x = f().
]
```

Consider 3 possibilities for function `f`

1
```
integer f{} [

]
```
code flow analysis required to verify that every path through fcn w/ non-void return type actually returns a value

2
```
integer f{} [
    return.
]
```
non-void fcn & no return value in return statement

3
```
integer f{} [
    return True.
]
```
wrong type of return value

type checker catches these

Code Generation for Expressions

Categories of expression nodes

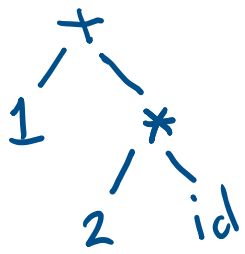
- literals — integer literals, string literals, True, False
- IDs
- tuple-access — not for P6
- call
- assignment
- non-short-circuited operators
- short-circuited operators (&, |)

Goal: evaluate expression leaving result on the stack

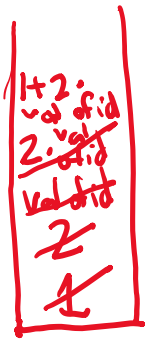
To do this, linearize ("flatten" expression tree)

- use a work stack and post-order traversal — like SDT during parsing
- at operand: push value onto stack
- at operator: pop source values from stack, push result

Example: `1 + 2 * id`



eval + node
  eval L child
    push 1 (on to stack)
  eval R child (* node)
    eval L child
      push 2
    eval R child
      push id — i.e., value of id
    pop into $t1
    pop into $t0
    multiply: $t0 * $t1 into $t1
    push $t1
  pop into $t1
  pop into $t0
  add: $t0 + $t1 into $t0
  push $t0



Code Generation for Literals

Integer (and logical) literals

```
li $t0, value
# code to push $t0 on stack
```

String literals

- stored in static data area

```
    .data
label: .asciiz string_value
```

must include quotes

↑ unique label

- to access, push *address* on to stack

load addr into $t0 (la $t0, label), push $t0 onto stack

- two strings with same sequence of characters are considered equal (ie, using ==)

base code: if "abc" == "abc" [...

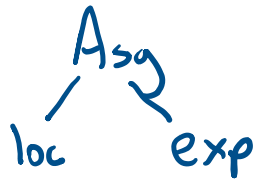
generate only one copy of .data

_L35: .asciiz "abc"

Code Generation for Assignments

Code generation for AssignExpNode

- compute address of LHS location; leave result on stack
- compute value of RHS expr; leave result on stack
- pop RHS into $t1
- pop LHS into $t0
- store value in $t1 at address held in $t0
- push $t1 on stack



Need to be able to do: a = b = c = 7.

Code generation for AssignStmtNode

1) call codeGen on AsgExpNode
2) pop stack

Code Generation for Function Calls

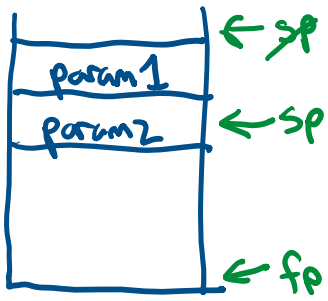
Precall

- put argument *values* on the stack
  pass-by-value semantics
- save *live* registers
  (note: we don't have any in a stack machine)
- jump to callee preamble label
  jal _fctnName (jump & link)

CallExp
IdNode ExpList

Postcall

- tear down the actual parameters → move sp
- retrieve and push result value → onto stack
  ↳ from $v0



CallStmtNode
|
CallExpNode

Handle like AsgStmtNode
- pop stack after calling codeGen on CallExpNode

x = f(y) + 4 * z. CallExp Node

g(a,b,c). CallStmt Node

Code Generation for I/O

Example (in base)

```
write << a + b.
read >> c.
```

MIPS I/O is done using `syscall`

Algorithm

- load system call code into $v0
  - 1 to print integer
  - 4 to print string
  - 5 to read integer
- put argument into $a0

eg if printing, codeGen on exp
pop into $a0

- do syscall ie, generate MIPS instruction: syscall