CS143: Code Generation

David L. Dill
Stanford University
Code Generation

• Semantic Analysis (wrap up)
  • MIPS Architecture
  • Stack Machine on MIPS
  • Generating Code for Expressions
  • Function Calls
  • Objects
MIPS Architecture
MIPS Architecture

One of the first "RISC" CPU chips
From Stanford research project
Led by Prof. John L. Hennessy — now Stanford President

RISC = "Reduced Instruction Set Computer"

Idea: Simple, very fast instructions
Still used in embedded processors, synthesizable cores system on a chip.
MIPS Architecture

8 bit bytes

32 general-purpose registers (32 bits each)

Arithmetic instructions

\[ \text{reg}1 \leftarrow \text{reg}2 \text{ op } \text{reg}3 \]

Load & Store instructions to move data between registers & memory

Documentation: SPIM manual
Some MIPS Instructions

```
lw reg1, offset(reg2)    "load word"
    Load 32-bit word from memory address in reg2+offset

add reg1, reg2 reg3
    reg1 ← reg2 + reg3

sw reg1, offset(reg2)    "store word"
    Store 32-bit word in reg1 to memory address in reg2+offset
```
Some MIPS Instructions

addiu reg, reg2, imm ("add immediate")
  \[ \text{reg} \leftarrow \text{reg2} + \text{imm} \in \text{constant} \]
  "u" means no check for overflow

li reg, imm ("load immediate")
  \[ \text{reg} \leftarrow \text{imm} \in \text{constant} \]

move reg1, reg2
  \[ \text{reg1} \leftarrow \text{reg2} \]
Stack Machine on MIPS
Stack Machine on MIPS

To simplify code generation, we use MIPS like a stack machine.

32-bit registers

$\text{a0} - \text{accumulator}$

$\text{sp} - \text{stack pointer}$

$\text{t1} - \text{temporary register}$

$\text{fp} - \text{frame pointer}$
Stack in MIPS

Stack grows from larger to smaller addresses, $sp$ contains address of next pushed value.

"push ace" $\Rightarrow$ sw $s0$ $0(\$sp)$
Stack in MIPS

Stack grows from larger to smaller addresses. $sp$ contains address of next pushed value.

```
"push ace"  ⇒  sw $a0 0($sp)
addiu $sp $sp -4
```

```

   Stack

   top

   $sp:

   $a0:

   ← $sp

   x
```

"push ace"  ⇒  sw $a0 0($sp)
addiu $sp $sp -4
Stack in MIPS

Stack grows from larger to smaller addresses. $sp$ contains address of next pushed value.

```
push ace
```

```
sw $a0 0($sp)
addiu $sp $sp -4
```

```
t1 ← top of stack
```

```
lw $t1 4($sp)
```
Stack in MIPS

Stack grows from larger to smaller addresses. 
$sp$ contains address of next pushed value.

```

"push acc"  =>  sw $a0 0($sp)  
                addi $sp $sp -4

$t1$ ← top_of_stack
    =>  lw $t1 4($sp)

"pop"  =>  addi $sp $sp 4
```
Generating Code for Expressions
Simple Programming Language

Program → Def+
Def → 'def' ID (ID+)
Expr → Int | ID | Expr + Expr
    | Expr - Expr
    | if Expr = Expr then Expr else Expr
    | ID (Expr+)


Example Program

def fib(x) =
    if x = 1 then Ø else
    if x = 2 then 1 else
        fib(x-1) + fib(x-2)
Generated Code for Expression

$sp$ and valid contents of stack before and after evaluation should be the same. $a0$ should contain the value of the expression.
Code for Constant Int

cgen (i):
    li $a0 i  — load i into $a0

Properties
    • Stack does not change
    • Result is in $a0
Generated Code for Expression

code for "$e_1 + e_2$":

<code for $e_1$> ← value of $e_1$ is in $a_0$

sw $a_0 0($sp) ← push acc
addiu $sp $sp -4
<code for $e_2$> ← value of $e_2$ in acc

lw $t1 4($sp) ← $t1 \leftarrow \text{top of stack}
add $a_0 $t1 $a_0 ← $a_0 ← $t1 + $a_0
addiu $sp $sp 4 ← \text{Pop}
Generating Code for $e_1 + e_2$

cgen (e_1 + e_2):

cgen (e_1)

print "sw $a0 0($sp)"
print "addiu $sp $sp -4"
cgen (e_2)

print "lw $t1 4($sp)"
print "add $a0 $t1 $a0"
print "addiu $sp $sp 4"
Code Generation for Conditional

New instruction: \texttt{beq \texttt{reg1, reg2 label}}

If \texttt{reg1} = \texttt{reg2}, jump to label

(label is a name. Assembler figures out the numerical address)

New instruction: \texttt{b label}

Jump to label
Code Generation for Conditional code for "if e₁ = e₂ then e₃ else e₄"

```assembly
<code for e₁>
sw $a0 @(sp $sp)
addiu $sp $sp -4  ] push e₁ val on stack
<code for e₂>
lw $t1 4($sp)
addiu $sp $sp 4 ] pop e₁ val into $t1
(A+ this point, we have
$a0 = e₂
$t1 = e₁ )
Code Generation for Conditional

Code for "if e₁ = e₂ then e₃ else e₄"

```assembly
<code for e₁>
sw $a0 @(sp)
addiu $sp $sp -4
<code for e₂>
 lw $t1 4($sp)
addiu $sp $sp 4
beq $a0 $t1 ltrue
```

(If $a0 = $t1, jump to label "ltrue"
else, go to next instruction)
Code Generation for Conditional code for "if e₁ = e₂ then e₃ else e₄"

<code for e₁>
sw $a₀ (@($sp))
addiu $sp $sp -4
<code for e₂>
lw $t₁ 4($sp)
addiu $sp $sp 4
beq $a₀ $t₁ ltrue
<code for e₄>
blend

]does this
if $a₀ $t₁
Code Generation for Conditional

code for "if \(e_1 = e_2\) then \(e_3\) else \(e_4\)"

\[
\begin{align*}
&\text{<code for } e_1 \text{> } \\
&\text{sw } $a0 \text{ } \& ($sp) \\
&\text{addiu } $sp \text{ } $sp \text{ } -4 \\
&\text{<code for } e_2 \text{> } \\
&\text{lw } $t1 \text{ } 4($sp) \\
&\text{addiu } $sp \text{ } $sp \text{ } 4 \\
&\text{beq } $a0 \text{ } $t1 \text{ } \text{lttrue} \\
&\text{<code for } e_4 \text{> } \\
&\text{b } \text{ldend}
\end{align*}
\]
Code Generation for Conditional

code for "if \(e_1 = e_2\) then \(e_3\) else \(e_4\)"

\[
\begin{align*}
\text{sw } &\$a0 ($sp) \\
\text{addiu } &\$sp \$sp -4 \\
\text{lw } &\$t1 4($sp) \\
\text{addiu } &\$sp \$sp 4 \\
\text{beq } &\$a0 \$t1 \text{ ltrue} \\
\text{b } &\text{lend}
\end{align*}
\]

\text{ltrue: } \langle \text{code for } e_3 \rangle

\text{lend: }

After executing "else" code \((e_4)\), it jumps here, skipping "then" code.

After executing "then" code, "lend" is next.
Function Calls
Activation Record

Result is always in 5a0
AR doesn’t need a space

Need:

Old frame pointer
Actual parameter values
Return address
New instruction: **jal label**

"jump and link" - for function calls

- save address of next instruction in $ra
- jump to label
Constructing the Activation Record

Call $f(x, y)$

Before Call

$fp \rightarrow$

$sp \rightarrow$

On entry to callee

$fp \rightarrow$

$sp \rightarrow$

Before Return

$fp \rightarrow$

$sp \rightarrow$

After Return

$fp \rightarrow$

$sp \rightarrow$
Call-side Code

code for $f(e_1, \ldots, e_n)$

sw $fp 0($sp$)$

addiu $sp $sp -4  \[ \text{save fp} \]

$fp \rightarrow$

$sp \rightarrow$
Call-side Code

code for \( f(e_1, \ldots, e_n) \)

\[
\begin{align*}
\text{sw} & \text{ } \$fp \text{ } 0(\$sp) \\
\text{addiu} & \text{ } \$sp \text{ } \$sp \text{ } -4
\end{align*}
\]

save fp

\[\begin{array}{c}
\text{old fp} \\
\$sp \rightarrow
\end{array}\]
Call-side Code

code for \( f(e_1, \ldots, e_n) \)

\[
\begin{align*}
&\text{sw } $fp 0($sp) \\
&\text{addiu } $sp $sp -4 \\
&\quad \text{[save fp]} \\
&\text{<code for } e_n \\
&\text{sw } $fp 0($sp) \\
&\text{addiu } $sp $sp -4 \\
&\quad \text{[save actuals in reverse order]} \\
&\vdots \\
&\text{<code for } e_1 \\
&\text{sw } $a0 0($sp) \\
&\text{addiu } $sp $sp -4
\end{align*}
\]
Call-side Code

code for \( f(e_1, \ldots, e_n) \)

\[ \begin{align*}
    &\text{sw } $fp 0($sp) \\
    &\text{addi } $sp $sp -4 \\
    &\langle \text{code for } e_n \rangle \\
    &\text{sw } $fp 0($sp) \\
    &\text{addi } $sp $sp -4 \\
    &\ldots \\
    &\langle \text{code for } e_1 \rangle \\
    &\text{sw } $a0 0($sp) \\
    &\text{addi } $sp $sp -4 \\
    &\text{jal } f\text{label}
\end{align*} \]

\$fp \rightarrow \\
\text{save } fp \\
\begin{array}{l}
\text{old fp} \\
\vdots \\
\vdots \\
\vdots \\
e_1 \end{array} \\
\text{save actuals} \\
\text{in reverse order} \\
\$sp \rightarrow \\
\text{jump to code for } f \\
\text{Return address will be in } \$ra
New instruction:  jr reg

Jump to address in register

Used for return from function
Function-side Code

```python
def f(x_1, \ldots, x_n) = e
move fp => sp — fp becomes current sp
```

```
old fp
  e
  ...
  e_i
$fp = $sp
```
Function-side Code

```python
def f(x_1, ..., x_n) = e
```

- `move $fp $sp` — $fp$ becomes current $sp$
- `sw $ra 0($sp)` — push return address
- `addiu $sp $sp -4` — ($fp$ points to return)
Function - side Code

def f(x_1, \ldots, x_n) = e

\begin{align*}
\text{move} \& fp & \text{ sp} & - fp \text{ becomes current sp} \\
\text{sw} \ & ra & 0(sp) & \text{push return address} \\
\text{addiu} \ & sp & sp - 4 & (fp \text{ points to return}) \\
\langle \text{code for e} \rangle & \\
\text{lw} \ & ra & 4(sp) & \text{return goes in ra}
\end{align*}
Function side Code

def f(x_1, \ldots, x_n) = e

move $fp$ $sp$  — fp becomes current sp
sw $ra$ 0($sp$)  [] push return address
addiu $sp$ $sp$ -4 (fp points to return)
< code for e >
lw $ra$ 4($sp$)  — return goes in $ra$
addiu $sp$ $sp$ e  — e = size of frame 41n + 8

Function-side Code

\[\text{def } f(x_1, \ldots, x_n) = e\]

\text{move $fp$ to $sp$} \quad \text{-- $fp$ becomes current $sp$}
\text{sw $ra$ to $0(sp)$} \quad \text{[push return address}}
\text{addiu $sp$ to $sp - 4$} \quad \text{(fp points to return)}
\langle \text{code for } e \rangle
\text{lw $ra$ to $4(sp)$} \quad \text{-- return goes in $ra$}
\text{addiu $sp$ to $sp - z$} \quad \text{-- $z$ = size of frame $4 \times n + 8$}
\text{lw $fp$ to $0(sp)$} \quad \text{-- restore old $fp$}
\text{jr $ra$} \quad \text{-- jump to return address}
Accessing Variables

```
.fp →
oldfp
y
x
return
expr
temp

← fp + 8
← fp + 4
```

Code to access value of parameter #i:

```
lw $a0 
```

Compile-time constant
Allocated Temporaries
Allocated Temporaries for Expression Evaluation

Idea:
Reserve space for temps in a block.
Assign each temp a location to use

Saves instructions because
MIPS takes 2 instructions to push.

Idea is useful for smarter register allocation
How Many Temporaries?

$NT(e) - \# \text{ of temporaries required to evaluate } e.$

$NT(e_1 + e_2) = \max (NT(e_1), 1 + NT(e_2))$

\[\text{need to preserve value of } e_1 \text{ in a temp while we use all the temps for } e_2.\]
Number of Terms

\[
NT(e_1 + e_2) = \max(NT(e_1), 1 + NT(e_2))
\]

\[
NT(e_1 - e_2) = \max(NT(e_1), 1 + NT(e_2))
\]

\[
NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) = \\
\max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4))
\]

\[
NT(f(e_1, \ldots, e_n)) = \max(NT(e_1), \ldots, NT(e_n))
\]

\[
NT(\text{Int}) = \emptyset
\]

\[
NT(\text{ID}) = \emptyset
\]
Code Gen with Temps

New parameter: index of next available temp

cgen(e₁ + e₂, nt)
cgen(e₁, nt)
sw $a₀ nt ($fp)  // save in temp
cgen(e₂, nt + 4)  // $a₀ ← e₂ value
lw $t₁ nt ($fp)  // $t₁ ← e₁ value
add $a₀ $t₁ $a₀  // $a₀ ← $t₁ + $a₀

_temps are heavily reused._
Objects
Principle

If class \( B \) is a subclass of class \( A \), then any code that operates on an object of type \( A \) must work on an object of type \( B \).

- Attributes inherited from \( A \) must be in same position
- Dispatch must find correct method (even if method is redefined in \( B \))
Object Layout

<table>
<thead>
<tr>
<th>Offset</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class tag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dispatch ptr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Offsets are known at compile time

Size & offset of each attribute are computed by analyzing class definition
Example

```java
class A {
    a: Int ← Ø;
    d: Int ← 1;
    f(): { — };
}
```

```java
class B inherits A {
    b: Int ← 2;
    f(): { — };
    g(): { — };
}
```

```java
class C inherits A {
    c: Int ← 3;
    h(): Int { — 3; }
}
```
Layout and Inheritance

Class B inherits A

Instance of A  Instance of B  Instance of C
Example

class A {  
a: Int ← ∅;  
d: Int ← 1;  
f(): { ─ };
}  

class C inherits A {  
c: Int ← 3;  
h(): Int { ─ 3};
}  

class B inherits A {  
b: Int ← 2;  
f(): { ─ };
}  
g(): { ─ 3};  
}
Dispatch

Want f at same offset in dispatch table, whether inherited or redefined.

A dispatch

B dispatch

C dispatch
Using Dispatch Tables

Dynamic dispatch e.f()

Code:

evaluate e → ptr to object
get ptr to dispatch table
get ptr to method from dispatch table
call method with self = e value