COOL Project
Code Generation

CS2210

Stack Machines
- A simple evaluation model
- No variables or registers
- A stack of values for intermediate results
- Each instruction:
  - Takes its operands from the top of the stack
  - Removes those operands from the stack
  - Computes the required operation on them
  - Pushes the result on the stack

Code Generation Models
- Evaluate all expression on stack
  - Stack machine
  - Conceptually very simple
    - Very slow
    - COOL provides support routines for this (ok for a toy compiler w/o optimization)
- Use processor registers to compute expressions
  - Used in practice
  - Much faster
  - Easier to optimize
  - Have to to (simple) register allocation
Example of Stack Machine Operation

- The addition operation on a stack machine

```
5 7 9 ...
---
5 7 9 ...
```

pop add push

Example of a Stack Machine Program

- Consider two instructions
  - push i: place the integer i on top of the stack
  - add: pop two elements, add them and put the result back on the stack

- A program to compute 7 + 5:
  push 7
  push 5
  add

Why Use a Stack Machine?

- Each operation takes operands from the same place and puts results in the same place

- This means a uniform compilation scheme

- And therefore a simpler compiler
Why Use a Stack Machine?

- Location of the operands is implicit
  - Always on the top of the stack
  - No need to specify operands explicitly
- No need to specify the location of the result
- Instruction "add" as opposed to "add r_1, r_2"
  - Smaller encoding of instructions
  - More compact programs
- This is one reason why Java Bytecodes use a stack evaluation model

Optimizing the Stack Machine

- The add instruction does 3 memory operations
  - Two reads and one write to the stack
  - The top of the stack is frequently accessed
- Idea: keep the top of the stack in a register (called accumulator)
  - Register accesses are faster
- The "add" instruction is now
  - acc ← acc + top_of_stack
    - Only one memory operation!

Stack Machine with Accumulator

Invariants

- The result of computing an expression is always in the accumulator
- For an operation op(e_1, ..., e_n) push the accumulator on the stack after computing each of e_1, ..., e_{n-1}
  - After the operation pop n-1 values
- After computing an expression the stack is as before
Stack Machine with Accumulator. Example

- Compute $7 + 5$ using an accumulator

<table>
<thead>
<tr>
<th>acc</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>+</td>
<td>12</td>
</tr>
</tbody>
</table>

acc ← 7
push acc
acc ← 5
acc ← acc + top_of_stack
pop

A Bigger Example: $3 + (7 + 5)$

<table>
<thead>
<tr>
<th>acc</th>
<th>Code</th>
<th>Acc, init, Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>3, &lt;init&gt;</td>
</tr>
<tr>
<td>push acc</td>
<td>3</td>
<td>3, &lt;init&gt;</td>
</tr>
<tr>
<td>acc ← 7</td>
<td>7</td>
<td>3, &lt;init&gt;</td>
</tr>
<tr>
<td>push acc</td>
<td>7</td>
<td>7, 3, &lt;init&gt;</td>
</tr>
<tr>
<td>acc ← 5</td>
<td>5</td>
<td>7, 3, &lt;init&gt;</td>
</tr>
<tr>
<td>acc ← acc + top_of_stack</td>
<td>12</td>
<td>7, 3, &lt;init&gt;</td>
</tr>
<tr>
<td>pop</td>
<td>12</td>
<td>3, &lt;init&gt;</td>
</tr>
<tr>
<td>acc ← acc + top_of_stack</td>
<td>15</td>
<td>3, &lt;init&gt;</td>
</tr>
<tr>
<td>pop</td>
<td>15</td>
<td>&lt;init&gt;</td>
</tr>
</tbody>
</table>

Notes

- It is very important that the stack is preserved across the evaluation of a subexpression
  - Stack before the evaluation of $7 + 5$ is 3, <init>
  - Stack after the evaluation of $7 + 5$ is 3, <init>
  - The first operand is on top of the stack
From Stack Machines to MIPS

- The compiler generates code for a stack machine with accumulator
- We want to run the resulting code on the MIPS processor (or simulator)
- We simulate stack machine instructions using MIPS instructions and registers

Simulating a Stack Machine...

- The accumulator is kept in MIPS register $a0
- The stack is kept in memory
- The stack grows towards lower addresses
  - Standard convention on the MIPS architecture
- The address of the next location on the stack is kept in MIPS register $sp
  - The top of the stack is at address $sp + 4

MIPS Assembly

MIPS architecture
- Prototypical Reduced Instruction Set Computer (RISC) architecture
- Arithmetic operations use registers for operands and results
- Must use load and store instructions to use operands and results in memory
- 32 general purpose registers (32 bits each)
  - We will use $sp, $a0 and $t1 (a temporary register)

Read the SPIM handout for more details
A Sample of MIPS Instructions

- \text{lw \ reg, offset(reg_2)}
  - Load 32-bit word from address \text{reg_2} + \text{offset} into \text{reg_1}
- \text{add \ reg_1, reg_2, reg_3}
- \text{reg_1 ← reg_2 + reg_3}
- \text{sw \ reg_1, offset(reg_2)}
  - Store 32-bit word in \text{reg_1} at address \text{reg_2} + \text{offset}
- \text{addiu \ reg_3, reg_2, imm}
  - \text{reg_3 ← reg_2 + imm}
  - "u" means overflow is not checked
- \text{li \ reg, imm}
- \text{reg ← imm}

MIPS Assembly. Example.

The stack-machine code for \texttt{7 + 5} in MIPS:

\begin{equation*}
\begin{aligned}
acc & ← 7 \\
push & acc \\
acc & ← 5 \\
acc & ← acc + top\_of\_stack
\end{aligned}
\end{equation*}

\begin{equation*}
\begin{aligned}
\text{li} & \$a0 7 \\
\text{sw} & \$a0 0(\$sp) \\
\text{addiu} & \$sp \$sp -4 \text{ li } \$a0 5 \\
\text{lw} & \$t1 4(\$sp) \\
\text{add} & \$a0 \$a0 \$t1 \\
\text{addiu} & \$sp \$sp 4
\end{aligned}
\end{equation*}

We now generalize this to a simple language...

A Small Language

- A language with integers and integer operations

\begin{equation*}
P \rightarrow D; \ P \mid D
\end{equation*}

\begin{equation*}
D \rightarrow \text{def id(ARGS) = E;}
\end{equation*}

\begin{equation*}
\text{ARGS} \rightarrow \text{id, ARG} \mid \text{id}
\end{equation*}

\begin{equation*}
E \rightarrow \text{int l id l if } E_0 \text{ then } E_1 \text{ else } E_4
\end{equation*}
A Small Language (Cont.)

- The first function definition \( f \) is the "main" routine
- Running the program on input \( i \) means computing \( f(i) \)
- Program for computing the Fibonacci numbers:
  - \( \text{def } \text{fib}(x) = \begin{cases} 
  0 & \text{if } x = 1 \\
  1 & \text{if } x = 2 \\
  \text{fib}(x - 1) + \text{fib}(x - 2) & \text{else}
\end{cases} \)

Code Generation Strategy

- For each expression \( e \) we generate MIPS code that:
  - Computes the value of \( e \) in \( $a0 \)
  - Preserves \( $sp \) and the contents of the stack
- We define a code generation function \( \text{cgen}(e) \) whose result is the code generated for \( e \)

Code Generation for Constants

- The code to evaluate a constant simply copies it into the accumulator:
  - \( \text{cgen}(i) = li \ $a0\ i \)
- Note that this also preserves the stack, as required
Code Generation for Add

cgen(e₁ + e₂) =
cgen(e₁)
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e₁)
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp -4

- Possible optimization: Put the result of e₁ directly in register $t1 ?

Code Generation for Add.
Wrong!

- Optimization: Put the result of e₁ directly in $t1?

cgen(e₁ + e₂) =
cgen(e₁)
move $t1 $a0
cgen(e₂)
add $a0 $t1 $a0

- Try to generate code for: 3 + (7 + 5)

Code Generation Notes

- The code for + is a template with "holes" for code for evaluating e₁ and e₂.
- Stack machine code generation is recursive.
- Code for e₁ + e₂ consists of code for e₁ and e₂ glued together.
- Code generation can be written as a recursive-descent of the AST.
  - At least for expressions.
Code Generation for Sub and Constants

New instruction: \texttt{sub reg1, reg2, reg3}

- Implements \( reg_1 \leftarrow reg_2 - reg_3 \)
- \( \text{cgen}(e_1 - e_2) = \text{cgen}(e_3) \)
  - \( \text{sw} \ $a0 0($sp) \)
  - \( \text{addiu} \ $sp \ $sp -4 \)
  - \( \text{cgen}(e_4) \)
  - \( \text{lw} \ $t1 4($sp) \)
  - \( \text{sub} \ $a0 \ $t1 \ $a0 \)
  - \( \text{addiu} \ $sp \ $sp 4 \)


Code Generation for Conditional

- We need flow control instructions

- New instruction: \texttt{beq reg1, reg2, label}
  - Branch to label if \( reg_1 = reg_2 \)

- New instruction: \texttt{b label}
  - Unconditional jump to label


Code Generation for If (Cont.)

\[ \text{cgen}(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) = \]
\[ \begin{align*}
\text{false_branch:} & \\
\text{cgen}(e_1) & \\
\text{sw} \ $a0 0($sp) & \\
\text{addiu} \ $sp \ $sp -4 & \\
\text{cgen}(e_4) & \\
\text{lw} \ $t1 4($sp) & \\
\text{addiu} \ $sp \ $sp 4 & \\
\text{beq} \ $a0 \ $t1 \text{ true_branch} & \\
\end{align*} \]
The Activation Record

- Code for function calls and function definitions depends on the layout of the activation record

- A very simple AR suffices for this language:
  - The result is always in the accumulator
  - No need to store the result in the AR
  - The activation record holds actual parameters
    - For \( f(x_1, \ldots, x_n) \) push \( x_n, \ldots, x_1 \) on the stack
    - These are the only variables in this language

The Activation Record (Cont.)

- The stack discipline guarantees that on function exit \( sp \) is the same as it was on function entry

- We need the return address

- It’s handy to have a pointer to the current activation
  - This pointer lives in register \( sp \) (frame pointer)
  - Reason for frame pointer will be clear shortly

The Activation Record

- Summary: For this language, an AR with the caller’s frame pointer, the actual parameters, and the return address suffices

- Picture: Consider a call to \( f(x, y) \). The AR will be:

```
    SP         FP
    |          |
    | old fp   |
    | y        |
    | x        |

AR of f
```
Code Generation for Function Call

- The calling sequence is the instructions (of both caller and callee) to set up a function invocation

- New instruction: jal label
  - Jump to label, save address of next instruction in $ra
  - On other architectures the return address is stored on the stack by the "call" instruction

Code Generation for Function Call (Cont.)

cgen(f(e1, ..., en)) =
    sw $fp 0($sp)
    addiu $sp $sp -4
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    ...
    cgen(en)
    sw $a0 0($sp)
    addiu $sp $sp -4
    jal f_entry

- The caller saves its value of the frame pointer
- Then it saves the actual parameters in reverse order
- The caller saves the return address in register $ra
- The AR so far is 4*n+4 bytes long

Code Generation for Function Definition

- New instruction: jr reg
  - Jump to address in register reg

cgen(def(x1, ..., xn) = e) =
    move $fp $sp
    sw $ra 0($sp)
    addiu $sp $sp -4
    cgen(e)
    lw $ra 4($sp)
    addiu $sp $sp z
    lw $fp 0($sp)
    jr $ra

- Note: The frame pointer points to the top, not bottom of the frame
- The callee pops the return address, the actual arguments and the saved value of the frame pointer
- z = 4*n + 8
Calling Sequence. Example for \( f(x,y) \).

```
Before call | On entry | Before exit | After call
------|----------|-------------|----------
FP       | FP       | FP          | FP       
SP       | SP       | SP          | SP       
old fp   | y        | old fp      | y        
x        | x        | return      | return   
```

Code Generation for Variables

- Variable references are the last construct
- The "variables" of a function are just its parameters
  - They are all in the AR
  - Pushed by the caller

- Problem: Because the stack grows when intermediate results are saved, the variables are not at a fixed offset from \$sp

Solution: use a frame pointer
- Always points to the return address on the stack
- Since it does not move it can be used to find the variables
- Let \( x_i \) be the \( i^{th} \) (\( i = 1, \ldots, n \)) formal parameter of the function for which code is being generated

\[
cgen(x_i) = lw \ $a0 \ z($fp) \quad (z = 4^{i+1})
\]
Example: For a function $\text{def } f(x,y) = e$ the activation and frame pointer are set up as follows:

- Old fp
- Y
- X
- FP
- Return
- SP

Summary

- The activation record must be designed together with the code generator
- Code generation can be done by recursive traversal of the AST
- Recommend to not use a stack machine
  - You learn more
  - Alternative not much more complicated

A Better Way

- Idea: Keep temporaries in the AR
- The code generator must assign a location in the AR for each temporary
Example

def fib(x) = if x = 1 then 0 else if x = 2 then 1 else fib(x - 1) + fib(x - 2)

- What intermediate values are placed on the stack?
- How many slots are needed in the AR to hold these values?

How Many Temporaries?

let NT(e) = # of temps needed to evaluate e

- NT(e_1 + e_2)
  - Needs at least as many temporaries as NT(e_1)
  - Needs at least as many temporaries as NT(e_2) + 1
- Space used for temporaries in e_1 can be reused for temporaries in e_2

The Equations

$$
\begin{align*}
  NT(e_1 + e_2) &= \max(NT(e_1), NT(e_2) + 1) \\
  NT(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) &= \max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4)) \\
  NT(id(e_1, \ldots, e_n)) &= \max(NT(e_1), \ldots, NT(e_n)) \\
  NT(id) &= 0 \\
  NT(int) &= 0
\end{align*}
$$

Is this bottom-up or top-down?
What is NT(...code for fib...)?
The Revised AR

- For a function definition \( f(x_1, \ldots, x_n) = e \) the AR has \( 2 + n + NT(e) \) elements
  - Return address
  - Frame pointer
  - \( n \) arguments
  - \( NT(e) \) locations for intermediate results

---

Revised Code Generation

- Code generation must know how many temporaries are in use at each point
  - Add a new argument to code generation: the position of the next available temporary

---

Picture

<table>
<thead>
<tr>
<th>Old FP</th>
<th>( x_2 )</th>
<th>( x_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Addr.</td>
<td>Temp NT(e)</td>
<td>Temp 1</td>
</tr>
</tbody>
</table>

---
Code Generation for +
(original)

cgen(e₁ + e₂) =
cgen(e₁)
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e₂)
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4

Code Generation for +
(revised)

cgen(e₁ + e₂, nt) =
cgen(e₂, nt)
sw $a0 nt($fp)
cgen(e₂, nt + 4)
lw $t1 nt($fp)
add $a0 $t1 $a0

Notes

■ The temporary area is used like a small, fixed-size stack
■ Can construct cgen for other constructs
Implementation Alternative

- Do expression evaluation in registers
  - much faster
  - easier to optimize
  - have to write your own code generation support routines :-(
    - But not too difficult and you may find it easier in some respects

Suggested Code Generation Algorithm (cf. Aho ch. 9)

- Walk tree and generate CFG with basic blocks of 3-address code
- Use new temporary name for every sub-expression t1, t2, ... don’t worry about actual registers and reusing temporary locations
- Extension: transform this to SSA form
  - Have to compute: dominators and iterated DF
  - Do this only once you have the first part done
- Generate code for each basic block
  - 3-address code statement by statement

One optimization you can do here:
- Do not use getreg/putreg allocation but perform register allocation on this CFG and generate code from it
  - Good speedups possible

Register Allocation

- Write a support routine that gives you (virtual) registers to do computation
  - Getreg / putreg (can be found in Aho ch. 9.6)
  - May return a real register or a stack memory location
    - Since on RISC all operations have to be performed in registers:
      - Reserve 2 registers for evaluation
      - Bring in-memory (stack) operands first into this expression register(s)
    - Evaluate and store back
- Alternative: perform register allocation on the CFG
  - This counts as an optimization
Expression Evaluation

- Use an address descriptor
  - a map of names of variables & temporaries to register locations
- \( x := y \text{ op } z \)
  - \( L = \text{getreg()} \) for the result of the computation
  - \( y' = \text{location of } y \text{ if not in a register call getreg()} \)
    - same for \( z \)
  - generate \( \text{ld-instruction} \)
  - generate \( L := y' \text{ op } z' \)
  - update \( x \)'s address map to indicate that \( x \) is in \( L \) now

Object Layout

- OO implementation = Stuff from last lecture + More stuff
- OO Slogan: If \( B \) is a subclass of \( A \), than an object of class \( B \) can be used wherever an object of class \( A \) is expected
- This means that code in class \( A \) works unmodified for an object of class \( B \)

Two Issues

- How are objects represented in memory?
- How is dynamic dispatch implemented?
Object Layout Example

Class A {
  a: Int <- 0;
  d: Int <- 1;
  f(): Int ( a <- a + d );
};

Class B inherits A {
  b: Int <- 2;
  f(): Int ( a );
  g(): Int ( a <- a - b );
};

Class C inherits A {
  c: Int <- 3;
  h(): Int ( a <- a * c );
};

Object Layout (Cont.)

- Attributes a and d are inherited by classes B and C
- All methods in all classes refer to a
- For A methods to work correctly in A, B, and C objects, attribute a must be in the same "place" in each object

Object Layout (Cont.)

An object is like a struct in C. The reference foo.field
is an index into a foo struct at an offset corresponding to field

Objects in Cool are implemented similarly
- Objects are laid out in contiguous memory
- Each attribute stored at a fixed offset in object
- When a method is invoked, the object is self and the fields are the object's attributes
Cool Object Layout

- The first 3 words of Cool objects contain header information:
  - Class Tag 0
  - Object Size 4
  - Dispatch Ptr 8
  - Attribute 1 12
  - Attribute 2 16
  - ...

Cool Object Layout (Cont.)

- Class tag is an integer
  - Identifies class of the object
- Object size is an integer
  - Size of the object in words
- Dispatch ptr is a pointer to a table of methods
  - More later
- Attributes in subsequent slots
- Lay out in contiguous memory

Subclasses

Observation: Given a layout for class A, a layout for subclass B can be defined by extending the layout of A with additional slots for the additional attributes of B

Leaves the layout of A unchanged (B is an extension)
Subclasses (Cont.)

- The offset for an attribute is the same in a class and all of its subclasses.
- Any method for an \( A_i \) can be used on a subclass \( A_j \).
- Consider layout for \( A_n < ... < A_3 < A_2 < A_1 \).

Dynamic Dispatch

- Consider the following dispatches (using the same example):
Object Layout Example
(Repeat)

Class A {
    a: Int <- 0;
    d: Int <- 1;
    f(): Int (a <- a + d);
};

Class B inherits A {
    b: Int <- 2;
    f(): Int (a);
    g(): Int (a <- a - b);
};

Class C inherits A {
    c: Int <- 3;
    h(): Int (a <- a * c);
};

Dynamic Dispatch Example

- e.g()
  - g refers to method in B if a is a B
- e.f()
  - f refers to method in A if f is an A or C
    (inherited in the case of C)
  - f refers to method in B for a B object

The implementation of methods and dynamic dispatch strongly resembles the implementation of attributes

Dispatch Tables

- Every class has a fixed set of methods (including inherited methods)
- A dispatch table indexes these methods
  - An array of method entry points
  - A method f lives at a fixed offset in the dispatch table for a class and all of its subclasses
Dispatch Table Example

<table>
<thead>
<tr>
<th>Offset</th>
<th>Class</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>TA</td>
</tr>
<tr>
<td>B</td>
<td>FB</td>
<td>g</td>
</tr>
<tr>
<td>C</td>
<td>TA</td>
<td>h</td>
</tr>
</tbody>
</table>

- The dispatch table for class A has only 1 method
- The tables for B and C extend the table for A to the right
- Because methods can be overridden, the method for f is not the same in every class, but is always at the same offset

Using Dispatch Tables

- The dispatch pointer in an object of class X points to the dispatch table for class X
- Every method f of class X is assigned an offset O_f in the dispatch table at compile time

Using Dispatch Tables (Cont.)

- To implement a dynamic dispatch e.f() we
  - Evaluate e, giving an object x
  - Call D[O_f]
    - D is the dispatch table for x
    - In the call, self is bound to x