CS1622

Lecture 18

Code Generation (2)
The Main Point

The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation record.

Thus, the AR layout and the code generator must be designed together!
Overview

- Code generation continued
- Reading Chapter 6.1
Example

The picture shows the state after the call to 2nd invocation of f returns
Discussion

- The advantage of placing the return value 1st in a frame is that the caller can find it at a fixed offset from its own frame.

- There is nothing magic about this organization.
  - Can rearrange order of frame elements.
  - Can divide caller/callee responsibilities differently.
  - An organization is better if it improves execution speed or simplifies code generation.
Discussion (Cont.)

- Real compilers hold as much of the frame as possible in registers
  - Especially the method result and arguments
Globals

- All references to a global variable point to the same object
  - Can’t store a global in an activation record

- Globals are assigned a fixed address once
  - Variables with fixed address are “statically allocated”

- Depending on the language, there may be other statically allocated values
Memory Layout with Static Data

Memory

Low Address

High Address

Code

Static Data

Stack
Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR
  
  ```
  method foo() { new Bar }
  ```
  
  The Bar value must survive deallocation of foo’s AR

- Languages with dynamically allocated data use a *heap* to store dynamic data
Notes

- The code area contains object code
  - For most languages, fixed size and read only
- The static area contains data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable
- The stack contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals
- Heap contains all other data
  - In C, heap is managed by `malloc` and `free`
Both the heap and the stack grow

Must take care that they don’t grow into each other

Solution: start heap and stack at opposite ends of memory and let the grow towards each other
Memory Layout with Heap

- **Code** (Low Address)
- **Static Data**
- **Stack**
- **Heap** (High Address)
Data Layout

- Low-level details of machine architecture are important in laying out data for correct code and maximum performance

- Chief among these concerns is *alignment*
Alignment

- Most modern machines are (still) 32 bit
  - 8 bits in a byte
  - 4 bytes in a word
  - Machines are either byte or word addressable
- Data is *word aligned* if it begins at a word boundary
- Most machines have some alignment restrictions
  - Or performance penalties for poor alignment
Alignment (Cont.)

- Example: A string
  "Hello"
  Takes 5 characters (without a terminating \0)

- To word align next datum, add 3 “padding” characters to the string

- The padding is not part of the string, it’s just unused memory
Code Generation

- Stack machines
- The MIPS assembly language
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
Example of Stack Machine Operation

- The addition operation on a stack machine
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 $\text{op}(e_1, \ldots, e_n)$ push the accumulator on the stack after computing each of $e_1, \ldots, e_{n-1}$
  - 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

```
acc

stack

acc= 7
push acc

acc= 5

acc= acc + top_of_stack
pop

12

7
...}

7
...}

5
...}

7
...}
```
### A Bigger Example: $3 + (7 + 5)$

<table>
<thead>
<tr>
<th>Code</th>
<th>Acc</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc= 3</td>
<td>3</td>
<td>&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 tutorial for more details
A Sample of MIPS Instructions

- **lw reg<sub>1</sub> offset(reg<sub>2</sub>)**
  - Load 32-bit word from address reg<sub>2</sub> + offset into reg<sub>1</sub>

- **add reg<sub>1</sub> reg<sub>2</sub> reg<sub>3</sub>**
  - reg<sub>1</sub> = reg<sub>2</sub> + reg<sub>3</sub>

- **sw reg<sub>1</sub> offset(reg<sub>2</sub>)**
  - Store 32-bit word in reg<sub>1</sub> at address reg<sub>2</sub> + offset

- **addiu reg<sub>1</sub> reg<sub>2</sub> imm**
  - reg<sub>1</sub> = reg<sub>2</sub> + imm
  - “u” means overflow is not checked

- **li reg imm**
  - reg = imm
MIPS Assembly. Example.

The stack-machine code for 7 + 5 in MIPS:

acc = 7
push acc
acc = 5
acc = acc + top_of_stack
pop

- We now generalize this to a simple language...
A Small Language

A language with integers and integer operations

\[
P \Rightarrow D; P \mid D
\]

\[
D \Rightarrow \text{def id(ARGS)} = E;
\]

\[
\text{ARGS} \Rightarrow \text{id, ARGS} \mid \text{id}
\]

\[
E \Rightarrow \text{int} \mid \text{id} \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4
\]

\[
\mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1, \ldots, E_n)
\]
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:
  
  ```python
  def fib(x) = if x = 1 then 0 else
  if x = 2 then 1 else
    fib(x - 1) + fib(x - 2)
  ```
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:

  \[ \text{cgen}(i) = \text{li} \; \$a0 \; i \]

- Note that this also preserves the stack, as required
Code Generation for Add

cgen(e_1 + e_2) =
    cgen(e_1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    cgen(e_2)
    lw $t1 4($sp)
    add $a0 $t1 $a0
    addiu $sp $sp 4

- Possible optimization: Put the result of e_1 directly in register $t1?