CSC 425 - Principles of Compiler Design I

Code Generation

Outline

- Stack machines
- Abstract assembly code
- A stack machine implementation example

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 to the stack

Example of a Stack Machine Program

- Consider two instructions:
  - `push i` – place the integer `i` on top of the stack
  - `add` – pop the topmost two elements, add them and put the result back on the stack
- Example 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 top of the stack
- No need to specify operands explicitly
- No need to specify the location of the result
- Instruction encoding is more compact than instructions with registers
- Many bytecode interpreters use a stack machine model, for example, Java and Python

Optimizing the Stack Machine

- The `add` instruction does three memory operations:
  - Two read operations and one write operation
  - The top of the stack is frequently accessed
- Idea: keep the top of the stack in a dedicated register (called the "accumulator")
- The `add` instruction is now

  ```
  acc := acc + top
  ```

  which is only one memory operation

Stack Machine with Accumulator: Invariants

- The result of computing an expression is always placed in the accumulator
- For an operation $op(e_1, \ldots, e_n)$, compute each e_i and then push the accumulator (the result of evaluating e_i) on the stack
- After the operation, pop $n - 1$ values
- After computing an expression, the stack is as before

Stack Machine with Accumulator: Example

- Compute $3 + (7 + 5)$ using an accumulator:

Code	Accumulator	Stack
`acc := 3`	3	$\langle init \rangle$
`push acc`	3	3, $\langle init \rangle$
`acc := 7`	7	3, $\langle init \rangle$
`push acc`	7	7, 3, $\langle init \rangle$
`acc := 5`	5	7, 3, $\langle init \rangle$
`acc := acc + top`	12	7, 3, $\langle init \rangle$
`pop`	12	3, $\langle init \rangle$
`acc := acc + top`	15	3, $\langle init \rangle$
`pop`	15	$\langle init \rangle$

From Stack Machines to Three-address Code

- The compiler generates code for a stack machine with an accumulator
- Here we use an abstract RISC assembly language for simplicity
- The generated assembly code simulates the stack machine instructions with instructions and registers

Simulating a Stack Machine with Assembly

- The accumulator is kept in a register, we will call it `acc`
- The stack is kept in memory
- The stack grows towards lower addresses
- The address of the next location on the stack is kept in a register, we will call it `sp` for stack pointer
- Memory is accessed with `load` and `store` instructions
- Assume a machine word is 32-bits
- Assume an arbitrary number of registers named `t1`, ..., `tn`

Sample Instructions

- Load word: load a 32-bit word from address $\textit{register}_1 + \textit{offset}$ into $\textit{register}_2$

  ```
  lw r1 offset(r2)
  ```

- Store word: store a 32-bit word in $\textit{register}_1$ at address $\textit{register}_2 + \textit{offset}$

  ```
  sw r1 offset(r2)
  ```

- Load immediate value

  ```
  li reg imm
  ```

- Add $\textit{register}_2$ and $\textit{register}_3$ and store the result in $\textit{register}_1$

  ```
  add r1 r2 r3
  ```

Example

- The stack machine code for $7 + 5$:

```
acc := 7              li acc 7
push acc              sw acc 0(sp)
                      li t1 -4
                      add sp sp t1
acc := 5              li acc 5
acc := acc + top      lw t1 4(sp)
                      add acc acc t1
pop                   li t1 4
                      add sp sp t1
```

A Small Language

- We will generalize the previous example to a simple language; a language with only integers and integer operations
- Grammar

$$
\begin{aligned}
Program &\rightarrow Function Program \\
&\mid Function \\
Function &\rightarrow id(Args)\ begin\ E\ end \\
Args &\rightarrow id, Args \\
&\mid id \\
E &\rightarrow int \\
&\mid id \\
&\mid if\ \ E_1 = E_2\ then\ E_3\ else\ E_4 \\
&\mid if\ \ E_1 + E_2 \\
&\mid if\ \ E_1 - E_2 \\
&\mid id(E_1, \ldots, E_n)
\end{aligned}
$$

A Small Language

- The first function definition f is the "main" function
- Running the program on input i means computing $f(i)$
- Example program: Fibonacci numbers:

```
fib(x)
begin
  if x = 1 then 0 else
  if x = 2 then 1 else fib(x-1) + fib(x-2)
end
```

Code Generation Strategy

- For each expression *e* we generate assembly code that:
  - Computes the value of *e* in `acc`
  - Preserves `sp` and the contents of the stack
- We define a recursive code generation function *cgen*(*e*) whose result is the code generated for *e*

Code Generation for Constants

- The code to evaluate an integer constant simply copies it into the accumulator:

  $cgen(int) = \texttt{li acc } int$

- Note that this also preserves the stack, as required

Code Generation for Addition

$cgen(e_1 + e_2) =$

```
    cgen(e1)             ; acc := the value e1
    sw acc 0(sp)         ; push that value on the stack
    li t1 -4
    add sp sp t1
    cgen(e2)             ; acc := the value of e2
    lw t1 4(sp)          ; retreive the value of e1
    add acc t1 acc       ; perform the addition
    li t1 4              ; pop the stack
    add sp sp t1
```

Code Generation Notes

- The code for $e_1 + e_2$ is a template with "holes" for the code that evaluates e_1 and e_2
- Stack machine code generation is recursive
- The code for $e_1 + e_2$ consists of code for e_1 and e_2 glued together
- Code generation can be written as a recursive descent of the AST (at least for arithmetic expressions)

Code Generation for Subtraction

- New instruction: subtract *register*$_2$ and *register*$_3$ and store the result in *register*$_1$

```
sub r1 r2 r3
```

$cgen(e_1 - e_2) =$

```
    cgen(e1)            ; acc := the value e1
    sw acc 0(sp)        ; push that value on the stack
    li t1 -4
    add sp sp t1
    cgen(e2)            ; acc := the value of e2
    lw t1 4(sp)         ; retreive the value of e1
    sub acc t1 acc      ; perform the subtraction
    li t1 4             ; pop the stack
    add sp sp t1
```

Code Generation for Conditionals

- We need flow control instructions and labels
- A label is a symbolic name that indicates a point in the code that can be jumped to
- The code for $e_1 + e_2$ consists of code for e_1 and e_2 glued together
- New instructions:
  - Branch to label if $register_1 = register_2$

    ```
    beq r1 r2 label
    ```

  - Unconditional jump to label

    ```
    jump label
    ```

Code Generation for If Then Else

$cgen(if\ e_1 = e_2\ then\ e_3\ else\ e_4) =$

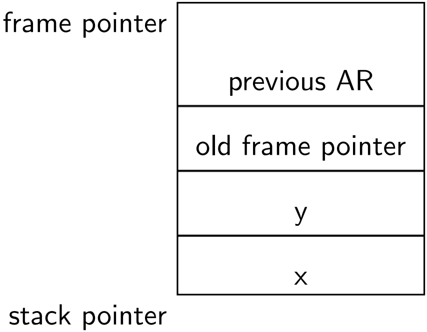
```
    cgen(e1)
    sw acc 0(sp)
    li t1 -4
    add sp sp t1
    cgen(e2)
    lw t2 4(sp)
    li t1 4
    add sp sp t1
    beq acc t2
false_branch:
    cgen(e4)
    jump end_if
true_branch:
    cgen(e3)
end_if:
```

Code Generation for Functions

- Code for function calls and function definitions depends on the layout of the activation record
- A simple activation record is sufficient for the example language
  - The result is always in the accumulator; there is no need to store the result in the activation record
  - The activation record holds the actual parameters; for $f(x_1, \ldots, x_n)$ push the arguments $x_1, \ldots, x_n$ onto the stack
- The stack machine invariants guarantee that on function exit the stack is the same as it was before the arguments got pushed
- We need the return address
- It is also convenient to have a pointer to the currect activation; this pointer will be stored in the register `fp` (frame pointer)

Layout of the Activation Record

- For the example language, an activation record with the caller's frame pointer, the actual parameters, and the return address is sufficient
- Consider a call to $f(x, y)$, the activation record would be:



Code Generation for a Function Call

- The calling sequence is the instructions (of both caller and callee) to set up a function invocation
- New instructions:
  - Jump to label and save the address of the next instruction in a special register `ra` (return address)

    ```
    jumpal label
    ```

  - Jump to address in $register_1$

    ```
    jumpr r1
    ```

  - Copy the value of $register_2$ to $register_1$

    ```
    move r1 r2
    ```

Code Generation for a Function Call

$cgen(f(e_1, \ldots, e_n)) =$

```
sw fp 0(sp)        ; the caller saves the value of the
li t1 -4           ; frame pointer
add sp sp t1
cgen(e_n)          ; push the actual parameters in
sw acc 0(sp)       ; reverse order
li t1 -4
add sp sp t1
...
cgen(e_1)
sw acc 0(sp)
li t1 -4
add sp sp t1
jumpal f_entry     ; jump and save return address in ra
```

Code Generation for a Function Definition

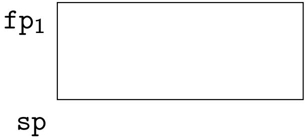
$cgen(f(x_1, \ldots, x_n) \; begin \; e \; end) =$

```
f_entry
    move fp sp
    sw acc 0(sp)
    li t1 -4
    add sp sp t1
    cgen(e)
    lw ra 4(sp)
    li t1 frame_size        ; frame size is 4n + 8
    add sp sp t1
    lw fp 0(sp)
    jumpr ra
```

- The callee saves the old return address, evaluates its body, pops the return address, pops the args, and then restores the fram pointer

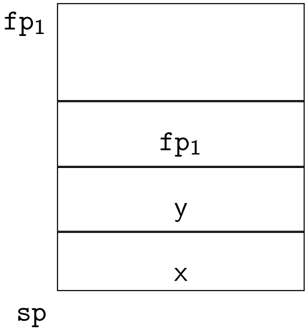
Calling Sequence: Example for $f(x, y)$

Before call



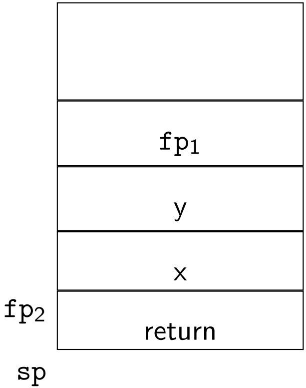
Calling Sequence: Example for $f(x, y)$

On entry

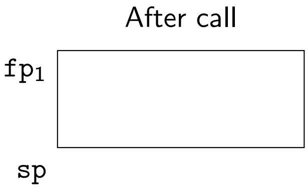


Calling Sequence: Example for $f(x, y)$

After body



Calling Sequence: Example for $f(x, y)$



Code Generation for Variables/Parameters

- Variable references are the last construct
- The "variables" of a function are its parameters:
  - They are in the activation record
  - Pushed by the caller
- Problem: because the stack grows when intermediate results are saved, the variables are not at a fixed offset from `sp`

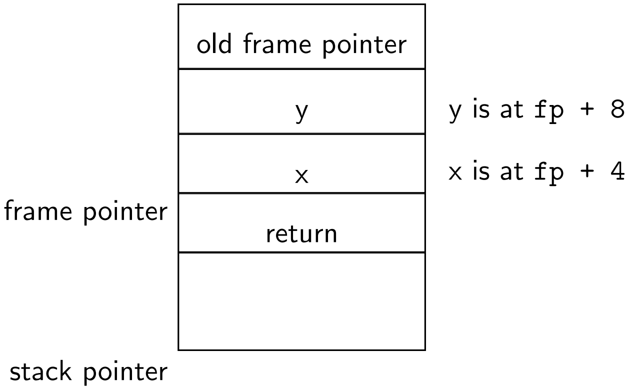
Code Generation for Variables/Parameters

- Solution: use the 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} formal parameter of the function for which code is being generated

  $cgen(x_i) =$ `lw acc offset(fp)`   ; offset = 4 * i

Code Generation for Variables/Parameters

- Example: for a function $f(x, y)$ *begin e end*, the activation and frame pointer are set up as follows (when evaluating e)



Activation Record and Code Generation Summary

- The activation record must be designed together with the code generator
- Code generation can be done by recursive traversal of the AST
- Note: production compilers do different things:
  - emphasis is on keeping values in registers
  - intermediate results are laid out in the activation record, not pushed and popped from the stack
  - as a result, code generation is often performed in synergy with register allocation