CS412/CS413

Introduction to Compilers

Tim Teitelbaum

Lecture 21: Generating Pentium Code

12 March 07

Simple Code Generation

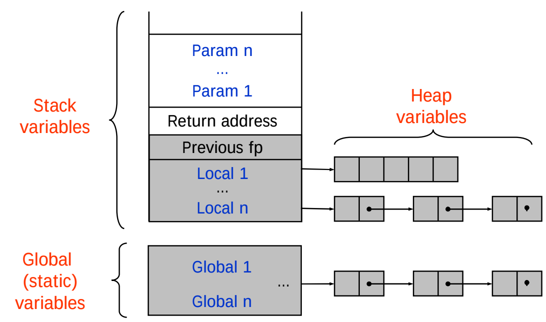
- Three-address code makes it easy to generate assembly
  - Complex expressions in the input program already lowered to sequences of simple IR instructions
  - Just need to translate each low IR instruction into a sequence of assembly instructions

    e.g. a = p+q ➡ mov 16(%ebp), %ecx
    add 8(%ebp), %ecx
    mov %ecx, -8(%ebp)
- Need to consider many language constructs:
  - Operations: arithmetic, logic, comparisons
  - Accesses to local variables, global variables
  - Array accesses, field accesses
  - Control flow: conditional and unconditional jumps
  - Method calls, dynamic dispatch
  - Dynamic allocation (new)
  - Run-time checks

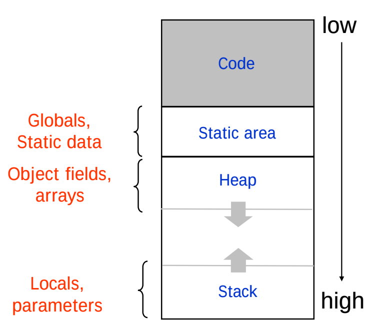
x86 Quick Overview

- Registers:
  - General purpose 32bit: eax, ebx, ecx, edx, esi, edi
    - Also 16-bit: ax, bx, etc., and 8-bit: al, ah, bl, bh, etc.
  - Stack registers: esp, ebp
- Instructions:
  - Arithmetic: add, sub, inc, mod, idiv, imul, etc.
  - Logic: and, or, not, xor
  - Comparison: cmp, test
  - Control flow: jmp, jcc, jecz
  - Function calls: call, ret
  - Data movement: mov (many variants)
  - Stack manipulations: push, pop
  - Other: lea

Big Picture of Program Memory



Memory Layout



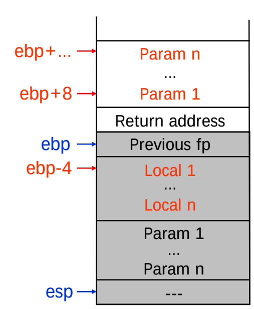
Accessing Stack Variables

- To access stack variables: use offsets from ebp
- Example:

  8(%ebp) = parameter 1

  12(%ebp) = parameter 2

  -4(%ebp) =local 1



Accessing Stack Variables

- Translate accesses to variables:
  - For parameters, compute offset from %ebp using:
    - Parameter number
    - Sizes of other parameters
  - For local variables, decide upon data layout and assign offsets from frame pointer to each local
  - Store offsets in the symbol table
- Example:
  - a: local, offset-4
  - p: parameter, offset+16, q: parameter, offset+8
  - Assignment a = p + q becomes equivalent to:
    -4(%ebp) = 16(%ebp) + 8(%ebp)
  - How to write this in assembly?

Arithmetic

- How to translate: p+q ?
  - Assume p and q are locals or parameters
  - Determine offsets for p and q
  - Perform the arithmetic operation
- Problem: the ADD instruction in x86 cannot take both operands from memory; notation for possible operands:
  - mem32: register or memory 32 bit (similar for r/m8, r/m16)
  - reg32: register 32 bit (similar for reg8, reg16)
  - imm32: immediate 32 bit (similar for imm8, imm16)
  - At most one operand can be mem !
- Translation requires using an extra register
  - Place p into a register (e.g. %ecx): mov 16(%ebp), %ecx
  - Perform addition of q and %ecx: add 8(%ebp), %ecx

Data Movement

- Translate a = p+q:
  - Load memory location (p) into register (%ecx) using a move instr.
  - Perform the addition
  - Store result from register into memory location (a):

```
mov 16(%ebp), %ecx     (load)
add 8(%ebp), %ecx      (arithmetic)
mov %ecx, -8(%ebp)     (store)
```

- Move instructions cannot take both operands from memory

  Therefore, copy instructions must be translated using a an extra register:

  a = p ⇒ mov 16(%ebp), %ecx
  mov %ecx, -8(%ebp)

- However, loading constants doesn't require extra registers:

  a = 12 ⇒ mov $12, -8(%ebp)

Accessing Global Variables

- Global (static) variables are not allocated on the run-time stack
- Have fixed addresses throughout the execution of the program
  - Compile-time known addresses (relative to the base address where program is loaded)
  - Hence, can directly refer to these addresses using symbolic names in the generated assembly code
- Example: string constants

  str: .string "Hello world!"

  - The string will be allocated in the static area of the program
  - Here, "str" is a label representing the address of the string
  - Can use $str as a constant in other instructions:

  push $str

Accessing Heap Data

- Heap data allocated with new (Java) or malloc (C/C++)
  - Such allocation routines return address of allocated data
  - References to data stored into local variables
  - Access heap data through these references
- Array accesses in Java
  - access a[i] requires:
    - To compute address of element: a + i * size
    - And access memory at that address
  - Can use indexed memory accesses to compute addresses
  - Example: assume size of array elements is 4 bytes, and local variables a, i (offsets –4, -8)

```
a[i] = 1  ⇒  mov –4(%ebp), %ebx      (load a)
             mov –8(%ebp), %ecx      (load i)
             mov $1, (%ebx,%ecx,4)   (store into the heap)
```


Control-Flow

- Label instructions
  - Simply translated as labels in the assembly code
  - E.g., label2: mov $2, %ebx
- Unconditional jumps:
  - Use jump instruction, with a label argument
  - E.g., jmp label2
- Conditional jumps:
  - Translate conditional jumps using test/cmp instructions:
  - E.g., tjump b L   cmp %ecx, $0
    jnz L

    where %ecx hold the value of b, and we assume booleans are represented as 0=false, 1=true

Run-time Checks

- Run-time checks:
  - Check if array/object references are non-null
  - Check if array index is within bounds
- Example: array bounds checks:
  - if v holds the address of an array, insert array bounds checking code for v before each load (…=v[i]) or store (v[i] = …)
  - Assume array length is stored just before array elements:

```
cmp $0, -12(%ebp)          (compare i to 0)
jl ArrayBoundsError        (test lower bound)
mov –8(%ebp), %ecx         (load v into %ecx)
mov –4(%ecx), %ecx         (load array length into %ecx)
cmp –12(%ebp), %ecx        (compare i to array length)
jle ArrayBoundsError       (test upper bound)
. . .
```


X86 Assembly Syntax

- Two different notations for assembly syntax:
  - AT&T syntax and Intel syntax
  - In the examples: AT&T syntax
- Summary of differences:

Order of operands	op a, b : b is destination	op a, b : a is destination
Memory addressing	disp(base,offset,scale)	[base + offset*scale + disp]
Size of memory operands	instruction suffixes (b,w,l) (e.g., movb, movw, movl)	operand prefixes (byte ptr, word ptr, dword ptr)
Registers	%eax, %ebx, etc.	eax, ebx, etc.
Constants	$4, $foo, etc	4, foo, etc