Agenda

- Mapping source code to x86
  - Mapping for other common architectures follows same basic pattern
- Today: basic statements and expressions
- Next time: Object representation, method calls, and dynamic dispatch

Review: Variables

- For us, all data will be in either:
  - A stack frame for method local variables
  - An object for instance variables
- Local variables accessed via ebp
  - mov eax,[ebp+12]
- Instance variables accessed via an object address in a register
  - Details later
Conventions for Examples

- Examples show code snippets in isolation
- Real code generator needs to worry about things like
  - Which registers are busy at which point in the program
  - Which registers to spill into memory when a new register is needed and no free ones are available
  - (x86: temporaries are usually pushed on the stack)
- Register eax used below as a generic example
  - Rename as needed for more complex code involving multiple registers

Peephole Optimizations

- A class of optimizations involving small numbers of instructions
- We’ll point out a few of these along the way

Assignment Statement

- Source
  - var = exp;
- x86
  - <code to evaluate exp into, say, eax>
  - mov [ebp+offsetof(var)],eax
Constants

- Source
  - 17
- x86
  - `mov eax,17`
- Optimization: if constant is 0
  - `xor eax,eax`

Unary Minus

- Source
  - `-exp`
- x86
  - `<code evaluating exp into eax>
  - `neg eax`
- Optimization
  - Collapse `-(-exp)` to `exp`
  - Unary plus is a no-op

Binary +

- Source
  - `exp1 + exp2`
- x86
  - `<code evaluating exp1 into eax>
  - `<code evaluating exp2 into edx>
  - `add eax,edx`
**Binary +**

- Optimizations
  - If exp2 is a simple variable or constant
    - add eax, exp2
  - Change exp1 + -exp2 into exp1 - exp2
  - If exp2 is 1
    - inc eax

**Binary -, * **

- Same as +
  - Use sub for –
  - Use imul for *
- Optimizations
  - Use left shift to multiply by powers of 2
  - Use x+x instead of 2*x, etc. (faster)
  - Use dec for x-1

**Integer Division**

- Ghastly on x86
  - Only works on 64-bit int divided by 32-bit int
  - Requires use of specific registers
- Source
  - x86
    - `<code>evaluating exp1 into eax ONLY</code>`
    - `<code>evaluating exp2 into ebx</code>`
    - `cdq ; extend to edcx, clobbers edx`
    - `idiv ebx ; quotient in eax; remainder in edx`
**Control Flow**

- Basic idea: decompose higher level operation into conditional and unconditional gotos
- In the following, jfalse is used to mean jump when a condition is false
- No such instruction on x86
- Will have to realize with appropriate sequence of instructions to set condition codes and perform conditional jumps
- Normally won't actually generate the value "true" or "false" in a register

**While**

- Source
  
  while (cond) stmt

- x86
  
  test: <code evaluating cond>  
jfalse done  
<code for stmt>  
jmp test  
done:

**Labels**

- In x86 assembly language we'll need to produce unique labels for each if, while, etc.
- Some assemblers allow for "local" labels that can be reused
- Ignore for now – concentrate on code shape
Optimization for While

- Put the test at the end
  jmp test
  loop: <code for stmt>
  test: <code evaluating cond>
  jtrue loop
- Why bother?
  - Pulls one instruction (jmp) out of the loop
  - Avoids pipeline stall on jmp
  - Easy to do from AST; not so easy if generating code on the fly (e.g., recursive descent 1-pass compiler)

Do-While

- Source
  do stmt while(cond);
- x86
  loop: <code for stmt>
  <code evaluating cond>
  jtrue loop

If

- Source
  if (cond) stmt
- x86
  <code evaluating cond>
  jfalse skip
  <code for stmt>
  skip:
If-Else

- **Source**
  
  `if (cond) stmt1 else stmt2`

- **x86**
  
  `<code evaluating cond>`
  
  `jfalse else`  
  `<code for stmt1>`
  
  `jmp done`

  `else: <code for stmt2>`

  `done:`

Jump Chaining

- **Observation:** naïve implementation can produce jumps to jumps
- **Optimization:** if a jump has as its target an unconditional jump, change the target of the first jump to the target of the second
- **Repeat until no further changes**

Boolean Expressions

- **What do we do with this?**
  
  `x > y`

- **It is an expression that evaluates to true or false**
  
  **Could generate the value**

  **But normally we don't want/need the value; we're only trying to decide whether to jump**
Code for exp1 > exp2

- Basic idea: designate jump target, and whether to jump if the condition is true or if it is false
- Example: exp1 > exp2, target L123, jump on false
  - <evaluate exp1 to eax>
  - <evaluate exp2 to edx>
  - cmp eax,edx
  - jng L123

Boolean Operators: !

- Source
  - ! exp
- Context: evaluate exp and jump to L123 if false (true)
- To compile !, reverse the sense of the test: evaluate exp and jump to L123 if true (false)

Boolean Operators: && and ||

- In C/C++/Java/C#, these are short-circuit operators
- Right operand is evaluated only if needed
- Basically, generate the if statements that would be needed to do this if we didn't have the short-circuit operators
Example: Code for &

Source
if (exp1 && exp2) stmt

x86
<code for exp1>
jfalse skip
<code for exp2>
jfalse skip
<code for stmt>
skip:

Example: Code for ||

Source
if (exp1 || exp2) stmt

x86
<code for exp1>
jtrue doit
<code for exp2>
jfalse skip
doit: <code for stmt>
skip:

Realizing Boolean Values
If a boolean value needs to be stored in a variable or method call parameter, generate code needed to actually produce it.

Typical representations: 0 for false, +1 or -1 for true

C uses 0 and 1; we'll use that

Best choice can depend on machine architecture, but normally some convention is established during the primeval history of the architecture.
Boolean Values: Example

- Source
  
  ```
  var = bexp;
  ```

- x86
  
  ```
  <code for bexp>
  jfalse genFalse
  mov eax,1
  jmp storeIt
  genFalse:
  mov eax,0
  storeIt: mov [ebp+offset var],eax ; generated by asg stmt
  ```

Other Control Flow: switch

- Naive: generate a chain of nested if-else if statements
- Better: switch is designed to allow an O(1) selection, provided the set of switch values is reasonably compact
- Idea: create a 1-D array of jumps or labels and use the switch expression to select the right one
  - Need to generate the equivalent of an if statement to ensure that expression value is within bounds

Switch

- Source
  
  ```
  switch (exp) {
  case 0: stmts0;
  case 1: stmts1;
  case 2: stmts2;
  }
  ```

- X86
  
  ```
  <put exp in eax>
  "if (eax < 0 || eax > 2)
  jmp defaultLabel"
  mov eax,swtab[eax*4]
  jmp eax
  .data
  swtab dd L0
  dd L1
  dd L2
  .code
  L0: <stmts0>
  L1: <stmts1>
  L2: <stmts2>
  ```
x86 Addressing Modes

- A memory address in x86 can be
  - register
    + register optionally scaled by *2, *4, or *8
    + constant offset
  - Assemblers have many syntax
    variations involving labels, register
    values in brackets, etc.

Arrays

- Several variations
- C/C++/Java
  - 0-origin; an array with n elements contains
    variables a[0]...a[n-1]
  - 1 or more dimensions; row major order
- Key step is to evaluate a subscript
  expression and calculate the location of
  the corresponding element

0-Origin 1-D Integer Arrays

- Source
  - exp1[exp2]
- x86
  - <evaluate exp1 (array address) in eax>
  - <evaluate exp2 in edx>
  - address is [eax+4*edx] ; 4 bytes per element
Fortran Arrays

- Subscripts start with 1 (default)
- Column-major order
  - E.g., an array with 3 rows and 2 columns is stored in this sequence: a(1,1), a(2,1), a(3,1), a(1,2), a(2,2), a(3,2)

a(i,j) in Fortran

- To find a(i,j), we need to know
  - Values of i and j
  - How many rows the array has
- Location of a(i,j) is
  - Location of a + (j-1)(#rows) + (i-1)
- Factor to pull out compile-time constant part and evaluate that in compiler
  - [Loc. of a – (#rows) – 1] + [j(#rows) + i]
- Compile appropriately

Coming Attractions

- Code Generation for Objects
  - Representation
  - Method calls
  - Inheritance and overriding
- Strategies for implementing code generators
- Code improvement - optimization