Agenda

- Mapping source code to x86
  - Mapping for other common architectures follows same basic pattern
- Now: basic statements and expressions
- Next: 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 often pushed on the stack, but can also be stored in a stack frame)
- Register eax used below as a generic example
  - Rename as needed for more complex code involving multiple registers
- A few peephole optimizations shown
Constants

- **Source**
  
  17

- **x86**
  
  ```
  mov   eax,17
  ```
  
  Idea: realize constant value in a register

- **Optimization**: if constant is 0
  
  ```
  xor   eax,eax
  ```
  
  Machine instructions from a compiler writer’s perspective: “I don’t care what it was designed to do, I care what it can do!”
Assignment Statement

- **Source**
  
  ```
  var = exp;
  ```

- **x86**
  
  ```
  <code to evaluate exp into, say, eax>
  mov [ebp+offset_{var}],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 – (but not commutative!)
  - Use imul for *

- Optimizations
  - Use left shift to multiply by powers of 2
    - (If your multiplier is really slow or you’ve got free scalar units and multiplier is busy, \(10\times x = (8\times x) + (2\times x)\))
  - Use \(x+x\) instead of \(2\times x\), 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
  
  \[
  \text{exp1} / \text{exp2}
  \]

- x86
  
  <code evaluating exp1 into eax ONLY>
  <code evaluating exp2 into ebx>
  
  \[
  \text{cdq} \quad ; \text{extend to edx:eax, clobbers edx} \\
  \text{idiv \ ebx} \quad ; \text{quotient in eax; remainder in edx}
  \]
Control Flow

- Basic idea: decompose higher level operation into conditional and unconditional gotos
- In the following, $j_{\text{false}}$ 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 followed by 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:

- Note: In generated asm code we’ll need to generate unique label for each loop, conditional statement, etc.
Optimization for While

- Put the test at the end
  
  ```
  jmp  test
  loop:  <code for stmt>
  test:  <code evaluating cond>
  j_{true}  loop
  ```

- Why bother?
  - Pulls one instruction (jmp) out of the loop
  - Avoids a pipeline stall on jmp on each iteration
    - Although modern processors will often predict control flow and avoid the stall
  - Easy to do from AST or other IR; 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>
  j_true loop
  ```
If

- **Source**
  
  if (cond) stmt

- **x86**
  
  <code evaluating cond>
  
  j_{false} 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 (0/1 or whatever the local convention is)
  - 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 (or true)

- To compile !, reverse the sense of the test: evaluate exp and jump to L123 if true (or 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 jump appropriately and only evaluate operands when needed
Example: Code for &&

- **Source**
  
  \[
  \text{if (exp1 && exp2) stmt}
  \]

- **x86**
  
  <\text{code for exp1}>
  
  \text{j\_false} \text{ skip}
  
  <\text{code for exp2}>
  
  \text{j\_false} \text{ skip}
  
  <\text{code for stmt}>
  
  \text{skip:}
Example: Code for ||

- **Source**
  
  ```
  if (exp1 || exp2) stmt
  ```

- **x86**
  
  ```
  <code for exp1>
  j_{true} doit
  <code for exp2>
  j_{false} 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; normally some convention is established during the primeval history of the architecture.
Boolean Values: Example

- Source
  \[ \text{var} = \text{bexp} ; \]

- x86
  
  `<code for bexp>
  \text{j}_{\text{false}} \text{ genFalse}
  \text{mov} \text{ eax},1
  \text{jmp} \text{ storeIt}

  \text{genFalse:}
  \text{mov} \text{ eax},0

  \text{storeIt: mov [ebp+offset}_{\text{var}},eax ; generated by asg stmt}`
Faster, If Enough Registers

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

- **x86**
  
  ```
  xor eax,eax
  <code for bexp>
  jfalse storeIt
  inc eax
  storeIt: mov [ebp+offset_var],eax ; generated by asg stmt
  ```

- Or use conditional move (movecc) instruction if available – avoids pipeline stalls due to conditional jumps
Other Control Flow: switch

- Naïve: 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

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

X86

```asm
<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>
```
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**
  \[\text{exp1}[\text{exp2}]\]

- **x86**
  
  <evaluate \text{exp1} (array address) in \text{eax}>
  
  <evaluate \text{exp2} in \text{edx}>
  
  address is \[\text{eax}+4*\text{edx}\] ; 4 bytes per element
2-D Arrays

- Subscripts start with 1 (default)
- C, etc. use row-major order
  - E.g., an array with 3 rows and 2 columns is stored in this sequence: a(1,1), a(1,2), a(2,1), a(2,2), a(3,1), a(3,2)
- Fortran uses column-major order
  - Exercises: What is the layout? How do you calculate location of a(i,j)? What happens when you pass array references between Fortran and C/etc. code?
- Java does not have “real” 2-D arrays. A 2-D array is a pointer to a vector of pointers to the array rows
To find $a(i,j)$, we need to know:
- Values of $i$ and $j$
- How many columns the array has

Location of $a(i,j)$ is:
$$\text{Location of } a + (i-1) \times \text{(number of columns)} + (j-1)$$

Can factor to pull out load-time constant part and evaluate that at load time – no recalculating at runtime.
Coming Attractions

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