Administrivia

• Semantics/type check due Thur. night
  – How’s it going?
  – Reminder: if you want to use late days (max 2 per assignment, max 4 overall), both partners need to have them available and both are charged if used

• Codegen part of the project out next week
  – High-level overview in next few lectures
  – Project-specific view in sections next week
Agenda

• Mapping source code to x86-64
  – Mapping for other common architectures is similar

• This lecture: basic statements and expressions
  – We’ll go quickly since this is review for many, fast orientation for others, and pretty straightforward

• Next: Object representation, method calls, and dynamic dispatch

Footnote: These slides include more than is specifically needed for the course project
Review: Variables

• For us, all data will be either:
  – In a stack frame (method local variables)
  – In an object (instance variables)

• Local variables accessed via %rbp
  
  movq  -16(%rbp),%rax

• Object instance variables accessed via an offset from an object address in a register
  – Details later
Conventions for Examples

• Examples show code snippets in isolation
  – Much the way we’ll generate code for different parts of the AST in a compiler visitor pass
  – A different perspective from the 351 holistic view
• Register %rax used here as a generic example
  – Rename as needed for more complex code using multiple registers
• 64-bit data used everywhere
• A few peephole optimizations shown for a flavor of what’s possible
  – Some might be easy to do in the compiler project
What we’re skipping for now

• Real code generator needs to deal with many 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
  – Dealing with different sizes of data
  – Exploiting the full instruction set
Code Generation for Constants

• Source
  17

• x86-64
  `movq $17,%rax`
  – Idea: realize constant value in a register

• Optimization: if constant is 0
  `xorq %rax,%rax`
  (but some processors do better with `movq $0,%rax` – and this has changed over time, too)
Assignment Statement

• Source
  var = exp;

• x86-64
  <code to evaluate exp into, say, %rax>
movq  %rax, offset_{var}(%rbp)
 Unary Minus

• Source
  -exp

• x86-64
  <code evaluating exp into %rax>
  negq %rax

• Optimization
  – Collapse -(-exp) to exp

• Unary plus is a no-op
Binary +

- Source
  \[ \text{exp}_1 + \text{exp}_2 \]
- x86-64
  <code evaluating \text{exp}_1 into \%rax>
  <code evaluating \text{exp}_2 into \%rdx>
  \text{addq} \ \%rdx,\%rax
Binary +

• Some optimizations
  – If $\text{exp}_2$ is a simple variable or constant, don’t need to load it into another register first. Instead:
    \[
    \text{addq exp}_2, %rax
    \]
  – Change $\text{exp}_1 + (-\text{exp}_2)$ into $\text{exp}_1 - \text{exp}_2$
  – If $\text{exp}_2$ is 1
    \[
    \text{incq } %rax
    \]
  • Somewhat surprising: whether this is better than
    \[
    \text{addq } 1, %rax
    \]
    depends on processor implementation and has changed over time
Binary -, *

• Same as +
  – Use subq for – (but not commutative!)
  – Use imulq for *

• Some optimizations
  – Use left shift to multiply by powers of 2
  – If your multiplier is slow or you’ve got free scalar units and multiplier is busy, you can do $10\times x = (8\times x) + (2\times x)$
    • But might be slower depending on microarchitecture
  – Use x+x instead of 2*x, etc. (often faster)
  – Can use leaq (%rax,%rax,4),%rax to compute $5\times x$, then addq %rax,%rax to get $10\times x$, etc. etc.
  – Use decq for x-1 (but check: subq $1$ might be faster)
Signed Integer Division

• Ghastly on x86-64
  – Only works on 128-bit int divided by 64-bit int
    • (similar instructions for 64-bit divided by 32-bit in 32-bit x86)
  – Requires use of specific registers

• Source
  \[
  \frac{\text{exp}_1}{\text{exp}_2}
  \]

• x86-64
  <code evaluating \text{exp}_1 \text{ into } \%\text{rax ONLY}> \\
  <code evaluating \text{exp}_2 \text{ into } \%\text{rbx}> \\
  \text{cqto} \quad \# \text{ extend to } \%\text{rdx:}\%\text{rax, clobbers } \%\text{rdx} \\
  \text{idivq } \%\text{rbx} \quad \# \text{ quotient in } \%\text{rax, remainder in } \%\text{rdx}
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 x86-64
  - Will have to realize with appropriate sequence of instructions to set condition codes followed by conditional jumps
  - Normally don’t need to actually generate the value “true” or “false” in a register
    - But this is a useful shortcut hack for the project
While

• Source
  while (cond) stmt

• x86-64
  test: <code evaluating cond>
    j_{false} done
    <code for stmt>
    jmp test
  done:
  – Note: In generated asm code we need to have unique labels 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 – x86-64 does this particularly well
• 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)

• x86-64
  loop: <code for stmt>
  <code evaluating cond>
  $j_{\text{true}}$ loop
If

• Source
  
  if (cond) stmt

• x86-64
  
  <code evaluating cond>
  jfalse skip
  <code for stmt>

  skip:
If-Else

• Source
  
  if (cond) stmt₁ else stmt₂

• x86-64
  
  <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 (if-else if-...-else; or nested loops or conditionals, ...)

• 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
  – Often done in peephole optimization pass after initial code generation
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
    • (Although for our project we might simplify and always produce the value)
Code for exp1 > exp2

• Basic idea: Generated code depends on context:
  – What is the jump target?
  – Jump if the condition is true or if false?

• Example: evaluate exp1 > exp2, jump on false, target if jump taken is L123
  <evaluate exp1 to %rax>
  <evaluate exp2 to %rdx>
  cmpq  %rdx,%rax     # dst-src = exp1-exp2
  jng    L123
Boolean Operators: !

- Source
  
  \(! \ exp\)

- Context: evaluate exp and jump to L123 if false (or true)

- To compile !, just reverse the sense of the test: evaluate exp and jump to L123 if true (or false)
Boolean Operators: && and ||

• In C/C++/Java/C# /many others, 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
  
  if (exp₁ && exp₂) stmt

• x86-64
  
  <code for exp₁>
  j_false skip
  <code for exp₂>
  j_false skip
  <code for stmt>

  skip:
Example: Code for ||

• Source
  
  if (exp₁ || exp₂) stmt

• x86-64

  <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 specifies 0 and 1; we’ll use that
  – Best choice can depend on machine instructions & language; normally some convention is picked during the primeval history of the architecture
Boolean Values: Example

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

• x86-64
  
  \[
  \begin{align*}
  \text{<code for bexp>} \\
  \text{j}\_\text{false} & \quad \text{genFalse} \\
  \text{movq} & \quad \text{\$1,\%rax} \\
  \text{jmp} & \quad \text{storelt} \\
  \text{genFalse:} & \\
  \text{movq} & \quad \text{\$0,\%rax} \quad \# \text{or xorq} \\
  \text{storelt:} & \\
  \text{movq} & \quad \text{\%rax,offset\_\text{var}(\%rbp)} \quad \# \text{generated by asg stmt}
  \end{align*}
  \]
Better, If Enough Registers

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

- x86-64

  ```
  xorq %rax,%rax # or movq $0,%rax
  <code for bexp>
  jfalse store
  incq %rax # or movq $1,%rax
  ```

  ```
  store:
  ```

  ```
  movq %rax,offset_var(%rbp) # generated by asg
  ```

- Better: use movecc instruction to avoid conditional jump

- Can also use conditional move instruction for sequences like
  
  ```
  x = y<z ? y : z
  ```
Better yet: setcc

• Source
  \[ \text{var} = x < y; \]

• x86-64
  \[
  \begin{align*}
  \text{movq} & \quad \text{offset}_x(\%rbp),\%rax & \# \text{load } x \\
  \text{cmpq} & \quad \text{offset}_y(\%rbp),\%rax & \# \text{compare to } y \\
  \text{setl} & \quad \%al & \# \text{set low byte } \%rax \text{ to 0/1} \\
  \text{movzbq} & \quad \%al,\%rax & \# \text{zero-extend to 64 bits} \\
  \text{movq} & \quad \%rax,\text{offset}_\text{var}(\%rbp) & \# \text{gen. by asg stmt}
  \end{align*}
  \]
Other Control Flow: switch

• Naïve: generate a chain of nested if-else if statements
• Better: switch statement is intended to allow easier generation of 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: stmts\_0;
    case 1: stmts\_1;
    case 2: stmts\_2;
}

“break” is an unconditional jump to the end of switch

• x86-64:

<put exp in %rax>

“if (%rax < 0 || %rax > 2)
    jmp defaultLabel”

movq swtab(,%rax,8),%rax
jmp *%rax

.data
swtab:
    .quad L0
    .quad L1
    .quad L2
.text
L0: <stmts\_0>
L1: <stmts\_1>
L2: <stmts\_2>
Arrays

• Several variations
• C/C++/Java
  – 0-origin: an array with n elements contains variables a[0]...a[n-1]
  – 1 dimension (Java); 1 or more dimensions using row major order (C/C++)
• Key step is evaluate subscript expression, then calculate the location of the corresponding array element
0-Origin 1-D Integer Arrays

- Source
  \[ \text{exp}_1[\text{exp}_2] \]
- x86-64
  <evaluate \text{exp}_1 (array address) in \%rax>
  <evaluate \text{exp}_2 in \%rdx>
  address is (\%rax,\%rdx,8)  # if 8 byte elements
2-D Arrays

• Subscripts start with 0 (default)
• C/C++, etc. use row-major order
  – E.g., an array with 3 rows and 2 columns is stored in sequence: a(0,0), a(0,1), a(1,0), a(1,1), a(2,0), a(2,1)
• 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/C++ code?
• Java does not have “real” 2-D arrays. A Java 2-D array is a pointer to a list of pointers to the rows
  – And rows may have different lengths (ragged arrays)
a[i][j] in C/C++/etc.

• If a is a “real” 0-origin, 2-D array, to find a[i][j], we need to know:
  – Values of i and j
  – How many columns (but not rows!) the array has
• Location of a[i][j] is:
  – Location of a + (i*(#of columns) + j) * sizeof(elt)
• Can factor to pull out allocation-time constant part and evaluate that once – no recalculating at runtime; only calculate part depending on i, j
  – Details in most compiler books
Coming Attractions

• Code Generation for Objects
  – Representation
  – Method calls
  – Inheritance and overriding

• Strategies for implementing code generators

• Code improvement – “optimization”