CS 6410: Compilers

Fall 2023

Tamara Bonaci

t.bonaci@northeastern.edu

Thank you to UW faculty Hal Perkins. Today lecture notes are a modified version of his lecture notes.

Credits For Course Material

- **Big thank you to UW CSE faculty members, Hal Perkins and Justin Hsia**
- Some direct ancestors of this course:
  - UW CSE 401 (Chambers, Snyder, Notkin, Perkins, Ringenburg, Henry, …)
  - UW CSE PMP 582/501 (Perkins)
  - Cornell CS 412-3 (Teitelbaum, Perkins)
  - Rice CS 412 (Cooper, Kennedy, Torczon)
  - Many books (Appel; Cooper/Torczon; Aho, [[Lam,] Sethi,] Ullman [Dragon Book], Fischer, [Cytron ,] LeBlanc; Muchnick, …)

Agenda

- Overview of x86-64 architecture (core part only)
- Mapping source code to x86-64[1]
  - Basic statements and expressions
  - Object representation and layout
  - Field access
  - What is `this`?
  - Object creation – `new`
  - Method calls
    - Dynamic dispatch
    - Method tables
    - `super`
  - Runtime type information

[1] Mapping for other common architectures is similar

Reading:

Cooper and Torczon, chapters 4.1-4.4, 5.5, 6.2-6.5 and 7.1-7.4

Dragon book, chapters 6.3-6.5, 7.1-7.7, 8.1-8.4

Review: x86-64 Assembly "Data Types"

- Integral data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses
- Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
  - Different registers for those (e.g. xmm1, ymm2)
  - Come from extensions to x86 (SSE, AVX, …)
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
- Two common syntaxes
  - "AT&T": used by our course, slides, textbook, gnu tools, …
  - "Intel": used by Intel documentation, Intel tools, …
  - Must know which you're reading

Review: Intel vs. GNU Assembler

- Main differences between Intel docs and gcc assembler

	Intel/Microsoft	AT&T/GNU as
Operand order: op a,b	a = a op b (dst first)	b = b op a (dst last)
Memory address	[baseregister+offset]	offset(baseregister)
Instruction mnemonics	mov, add, push, …	movq, addq, pushq [operand size is added to end]
Register names	rax, rbx, rbp, rsp, …	%rax, %rbx, %rbp, %rsp, …
Constants	17, 42	$17, $42
Comments	; to end of line	# to end of line or /* … */

- Intel docs also include many complex, historical instructions and artifacts not commonly used by modern compilers – we won't use them either

Review: x86-64 Memory Model

- 8-bit bytes, byte addressable
- 16-, 32-, 64-bit words, double words and quad words (Intel terminology)
  - That's why the 'q' in 64-bit instructions like movq, addq, etc.
- Data should normally be aligned on "natural" boundaries for performance, although unaligned accesses are generally supported – but with a big performance penalty on many machines
- Little-endian – address of a multi-byte integer is address of low-order byte

Review: x86-64 registers

- 16 64-bit general registers
  - %rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rbp, %rsp, %r8-%r15
- Registers can be used as 64-bit integers or pointers, or 32-bit ints
  - Also possible to reference low-order 16- and 8-bit chunks – we won't for the most part
- To simplify our project we'll use only 64-bit data (ints, pointers, even booleans!)

Review: Instruction Format

- Typical data manipulation instruction

```
label:    opcode   src,dst      # comment
```

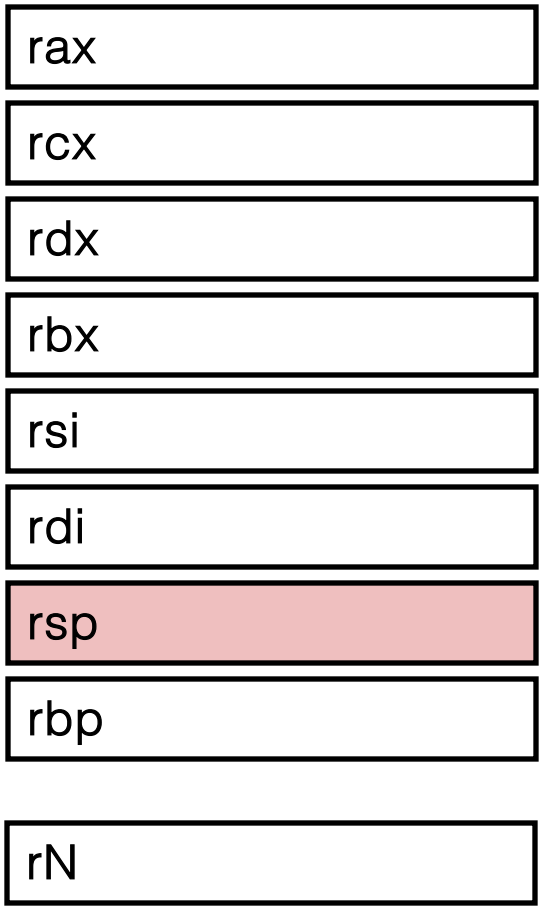
- Meaning is

  dst ⟵ dst op src

- Normally, one operand is a register, the other is a register, memory location, or integer constant
  - Can't have both operands in memory – can't encode two memory addresses in a single instruction (e.g., cmp, mov)
- Language is free-form, comments and labels may appear on lines by themselves (and can have multiple labels per line of code)

Review: Operand types

- Immediate: Constant integer data
  - Examples: `0x400, -533`
  - Encoded with 1, 2, 4, or 8 bytes depending on the instruction
- Register: 1 of 16 integer registers
  - Examples: `rax, r13`
  - But `rsp,rbp` reserved for special use
  - Others have special uses for particular instructions
- Memory: Consecutive bytes of memory at a computed address
  - Simplest example: `DWORD PTR[rax]`
  - Various other "address modes"



Review: Three Basic Kinds of Instructions

1) Transfer data between memory and register
   - Load data from memory into register
     - reg = Mem[address]
   - Store register data into memory
     - Mem[address] = reg

Remember: Memory is indexed just like an array of bytes!

2) Perform arithmetic operation on register or memory data
   - c = a + b; z = x << y; i = h & g;

3) Control flow: what instruction to execute next
   - Unconditional jumps to/from procedures
   - Conditional branches

Review: x86-64 - Instructions Overview

- Data transfer instruction (`mov`)
- Arithmetic operations
- Memory addressing modes
- Address computation instruction (`lea`)

Review: x86-64 Memory Stack

- Register %rsp points to the "top" of stack
  - Dedicated for this use; don't use otherwise
  - Points to the last 64-bit quadword pushed onto the stack (not next "free" quadword)
  - Should always be quadword (8-byte) aligned
    - It will start out this way, and will stay aligned unless your code does something bad
    - Software generally requires 16-byte alignment when function is called
  - Stack grows down

Review: Stack Instructions

pushq src

```
%rsp ← %rsp - 8;
memory[%rsp] ← src
```

(e.g., push src onto the stack)

popq dst

```
dst ← memory[%rsp];
%rsp ← %rsp + 8
```

(e.g., pop top of stack into dst and logically remove it from the stack)

Review: Stack Frames

- When a method is called, a stack frame is traditionally allocated on logical "top" of the stack to hold its local variables
- Frame is popped on method return
- By convention, %rbp (base pointer) points to a known offset into the stack frame
  - Local variables referenced relative to %rbp
  - Base pointer common in 32-bit x86 code; less so in x86-64 code where push/pop used less & stack frame has a fixed size so locals can be referenced from %rsp easily
  - We will use %rbp in our project – simplifies addressing of local variables and compiler bookkeeping

Review: Operand Address Modes (1)

- These should cover most of what we'll need

```
movq   $17,%rax          # store 17 in %rax
movq   %rcx,%rax         # copy %rcx to %rax
movq   -16(%rbp),%rax      # copy memory to %rax
movq   %rax,-24(%rbp)    # copy %rax to memory
```

- References to object fields work similarly – put the object's memory address in a register and use that address plus an offset
- Remember: can't have two memory addresses in a single instruction

Review:Operand Address Modes (2)

- A memory address can combine the contents of two registers (with one optionally multiplied by 2, 4, or 8) `plus a constant:`

```
basereg + indexreg*scale + constant
```

- Main use of general form is for array subscripting or small computations - if the compiler is clever
- Example: suppose we have an array of 8-byte ints with address of the array A in %rcx and subscript i in %rax. Code to store %rbx in A[i]

```
movq   %rbx,(%rcx,%rax,8)
```

qword ptr – Intel assembler

- Obscure, but sometimes necessary…
- If the assembler can't figure out the size of the operands to move, you can explicitly tell it to move 64bits with the qualifier "`qword ptr`"

```
mov   qword ptr  [rax+16],[rbp-8]
```

  - Similarly, for dword ptr, etc
  - Use this if the assembler complains; otherwise ignore
  - Not an issue in GNU asembler – operand sizes encoded in opcode mnemonics

Basic Data Movement and Arithmetic Instructions

`movq src,dst`

dst ← src

`addq src,dst`

dst ← dst + src

`subq src,dst`

dst ← dst – src

`incq dst`

dst ← dst + 1

`decq dst`

dst ← dst - 1

`negq dst`

dst ← - dst
(2's complement arithmetic negation)

Integer Multiply and Divide

`imulq src,dst`

dst ← dst * src

dst must be a register

`cqto`

`%rdx:%rax` ← 128-bit sign extended copy of %rax

(why??? To prep numerator for idivq!)

`idivq src`

Divide %rdx:%rax by src (%rdx:%rax holds sign-extended 128-bit value; cannot use other registers for division)

%rax ← quotient

%rdx ← remainder

(no division in MiniJava!)

Bitwise Operations

`andq src,dst`

dst ← dst & src

`orq src,dst`

dst ← dst | src

`xorq src,dst`

dst ← dst ^ src

`notq dst`

dst ← ~ dst
(logical or 1's complement)

Shifts and Rotates

`shlq  count,dst`

dst shifted left count bits

`shrq  count,dst`

dst ← dst shifted right count bits (0 fill)

`sarq  count,dst`

dst ← dst shifted right count bits (sign bit fill)

`rolq  count,dst`

dst ← dst rotated left count bits

`rorq  count,dst`

dst ← dst rotated right count bits

Uses for Shifts and Rotates

- Can often be used to optimize multiplication and division by small constants (mul/div by powers of 2)
  - If you're interested, look at "Hacker's Delight" by Henry Warren, A-W, 2nd ed, 2012
    - Lots of very cool bit fiddling and other algorithms
  - But be careful – be sure semantics are OK
    - Example: right shift is not the same as integer divide for negative numbers – shift truncates towards -∞
- There are additional instructions that shift and rotate double words, use a calculated shift amount instead of a constant, etc.

Address Computation Instruction

- `lea dst, src`
  - `lea` stands for load effective address
  - `src` is address expression (in brackets)
  - `dst` is a register
  - Sets `dst` to the address computed by the `src` expression (does not go to memory! – it just does math)
  - Example: lea rax, QWORD PTR[rdx+rcx*4+1]
- Uses:
  - Computing addresses without a memory reference
    - e.g. translation of p = &x[i];
  - Computing arithmetic expressions of the form x+k*i+d
    - Though k can only be 1, 2, 4, or 8

Load Effective Address

- The unary & operator in C/C++

```
leaq  src,dst        # dst ← address of src
```

  - `dst` must be a register
  - Address of `src` includes any address arithmetic or indexing
  - Useful to capture addresses for pointers, reference parameters, etc.
  - Also useful for computing arithmetic expressions that match r1 + scale*r2 + const

Control Flow - GOTO

- At this level, all we have is `goto` and conditional `goto`
- Loops and conditional statements are synthesized from these
- Note: random jumps play havoc with pipeline efficiency; much work is done in modern compilers and processors to minimize this impact

Unconditional Jumps

```
jmp   dst
   %rip ⟵ address of dst
```

- dst is usually a label in the code (which can be on a line by itself)
- dst address can also be indirect using the address in a register or memory location (*reg or *(reg)) – use for method calls, switch

Conditional Jumps

- Most arithmetic instructions set "condition code" bits to record information about the result (zero, non-zero, >0, etc.)
  - True of addq, subq, andq, orq; but not imulq, idivq, leaq
- Other instructions that set condition codes

```
cmpq src,dst # compare dst to src (e.g., dst-src)
testq src,dst      # calculate dst & src (logical and)
```

  - These do not alter src or dst

Conditional Jumps Following Arithmetic Operations

```
jz   label   # jump if result == 0
jnz  label   # jump if result != 0
jg   label   # jump if result > 0
jng  label   # jump if result <= 0
jge  label   # jump if result >= 0
jnge label   # jump if result < 0
jl   label   # jump if result < 0
jnl  label   # jump if result >= 0
jle  label   # jump if result <= 0
jnle label   # jump if result > 0
```

- Obviously, the assembler is providing multiple opcode mnemonics for several actual instructions

Compare and Jump Conditionally

- Want: compare two operands and jump if a relationship holds between them
- Would like to do this

  `jmp`$_{\texttt{cond}}$ `op1,op2,label`

  but can't, because 3-operand instructions can't be encoded in x86-64

  (also true of most other machines for that matter)

cmp and jcc

- Instead, we use a 2-instruction sequence

```
cmpq op1,op2
j<cc>  label
```

where `j<cc>` is a conditional jump that is taken if the result of the comparison matches the condition cc

Conditional Jumps Following Arithmetic Operations

```
je    label    # jump if op1 == op2
jne   label    # jump if op1 != op2
jg    label    # jump if op1 > op2
jng   label    # jump if op1 <= op2
jge   label    # jump if op1 >= op2
jnge  label    # jump if op1 < op2
jl    label    # jump if op1 < op2
jnl   label    # jump if op1 >= op2
jle   label    # jump if op1 <= op2
jnle  label    # jump if op1 > op2
```

- Again, the assembler is mapping more than one mnemonic to some machine instructions

Function Call and Return

- The x86-64 instruction set itself only provides for transfer of control (jump) and return
- Stack is used to capture return address and recover it
- Everything else – parameter passing, stack frame organization, register usage – is a matter of convention and not defined by the hardware

call and ret Instructions

call label

- Push address of next instruction and jump

```
%rsp ⟵ %rsp – 8;
memory[%rsp] ⟵ %rip
  %rip ⟵ address of label
```

- Call address can be in a register or memory as with jumps

ret

- Pop address from top of stack and jump

```
%rip ⟵ memory[%rsp];
%rsp ⟵ %rsp + 8
```

- WARNING! The word on the top of the stack had better be an address and not some leftover data

enter and leave

- Complex instructions for languages with nested procedures
  - `enter` can be slow on current processors – best avoided – i.e., don't use it in your project
  - `leave` is equivalent to

```
mov %rsp,%rbp
pop %rbp
```

  and is generated by many compilers. Fits in 1 byte, saves space. Not clear if it's any faster.

X86-64-Register Usage

- `%rax` – function result
- Arguments 1-6 passed in these registers in order
  - `%rdi, %rsi, %rdx, %rcx, %r8, %r9`
  - For Java/C++ "this" pointer is first argument, in %rdi
    - More about "this" later
- `%rsp` – stack pointer; value must be 8-byte aligned always and 16-byte aligned when calling a function
- `%rbp` – frame pointer (optional use)
  - We'll use it

x86-64 Register Save Conventions

- A called function must preserve these registers (or save/restore them if it wants to use them)
  - `%rbx, %rbp, %r12-%r15`
- `%rsp` isn't on the "callee save list", but needs to be properly restored for return
- All other registers can change across a function call

The Nice Thing About Standards…

- The above is the System V/AMD64 ABI convention (used by Linux, OS X)
- Microsoft's x64 calling conventions are slightly different (sigh…)
  - First four parameters in registers `%rcx, %rdx, %r8, %r9`; rest on the stack
  - Stack frame must include empty space for called function to use to store values passed in parameter registers if desired

x86-64 Function Call

- Caller places up to 6 arguments in registers, rest on stack, then executes call instruction (which pushes 8-byte return address)
- On entry, called function prologue sets up the stack frame:

```
pushq   %rbp            # save old frame ptr
movq    %rsp,%rbp       # new frame ptr is top of
                        # stack after ret addr and
                        # old rbp pushed
subq $framesize,%rsp  # allocate stack frame
```

x86-64 Function Return

- Called function:
  1. Puts result in %rax (if any)
  2. Restores any callee-saved registers (if needed)
- Called function returns with:

```
movq   %rbp,%rsp     # or use leave instead
popq   %rbp          # of movq/popq
ret
```

- If caller allocated space for arguments it deallocates as needed

Caller Example

- n = sumOf(17,42)

```
        movq    $42, %rsi    # load arguments
        movq    $17, %rdi
        call    sumOf        # jump & push ret addr
        movq    %rax, offset_n(%rbp) # store result
```

Example Function

- Source code

```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}
```

Assembly Language Version

```
int sumOf(int x, int y) {
int a,
int b;
# sumOf:
   pushq   %rbp   #prologue
   movq    %rsp,%rbp
   subq    $16,%rsp

a = x;
# a = x;
movq   %rdi,-8(%rbp)
```

```
b = a + y;
# b = a + y;
   movq    -8(%rbp),%rax
   addq    %rsi,%rax
   movq    %rax,-16(%rbp)

return b;
#  return b;
   movq    -16(%rbp),%rax
   movq    %rbp,%rsp
   popq       %rbp
   ret
#}
```

Stack for sumOf

```
int sumOf(int x, int y) {
   int a, int b;
   a = x;
   b = a + y;
   return b;
}
```

Code Shape

Review: Variables

- For us, all data is 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 today)

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
- Register %rax used here as a generic example
  - Rename as needed for more complex code
- 64-bit data used everywhere
- Examples show a few peephole optimizations
  - Some might be easy to do in the compiler project

What We're Skipping for Now

- Real code generator deals 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
  - 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

  exp_1 + exp_2

- x86-64

```
<code evaluating exp1 into %rax>
<code evaluating exp2 into %rdx>
addq %rdx,%rax
```

Binary +

- Some optimizations
  - If exp_2 is a simple variable or constant, don't need to load it into another register first. Instead:

    `addq  exp2,%rax`
  - Change exp_1 + (-exp_2) into exp_1-exp_2
  - If exp_2 is 1

    `incq  %rax`
    - Somewhat surprising: whether this is better than 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*x = (8*x)+(2*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*x, then
    ```
    addq %rax,%rax
    ```
    to get 10*x, etc. etc.
  - Use `decq` for x-1

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
  - Very slow (~50 clocks)
- Source

  exp_1 / exp_2

- x86-64

```
<code evaluating exp1 into %rax ONLY>
<code evaluating exp2 into %ebx>
cqto                # extend to %rdx:%rax, clobbers
%rdx
idivq  %ebx   # quotient in %rax, remainder in %rdx
```

Control Flow

Control Flow

- Basic idea: decompose higher level operation into conditional and unconditional gotos

- In the following, j_{false} is used to mean jump when a condition is false

  - No such instruction on x86-64
  - Will have to realize it with appropriate instruction to set condition codes followed by conditional jump
  - 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>
        jtrue  loop
```

- Why bother?
  - Pulls one `jmp` instruction out of the loop
  - May avoid 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_true   loop
```

If

- Source

  if (cond) stmt

- x86-64

```
        <code evaluating cond>
        j_false skip
        <code for stmt>
    skip:
```

If-Else

- Source

  if (cond) $stmt_1$ else $stmt_2$

- x86-64

```
        <code evaluating cond>
        j_false else
        <code for stmt_1>
        jmp  done
else:   <code for stmt_2>
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

- 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: 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
jng     L123
```

Boolean Operators

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 (exp_1 && exp_2) stmt

- x86-64

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

Example: Code for ||

- Source

  if (exp_1 || exp_2) stmt

- x86-64

```
        <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 specifies 0 and 1 if stored; we'll use that
  - Best choice can depend on machine instructions; normally some convention is established during the primeval history of the architecture

Boolean Values: Example

- Source

  var = bexp;

- x86-64

```
        <code for bexp>
        j_false    genFalse
        movq $1,%rax
        jmp  storeIt
genFalse:
        movq $0,%rax          # or xorq
storeIt:
        movq %rax,offset_var(%rbp) # generated
  by asg stmt
```

Better, If Enough Registers

- Source

  var = bexp;

- x86-64

```
        xorq    %rax,%rax
        <code for bexp>
        j_false store
        incq      %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

  var = x < y;

- x86-64

```
 movq    offset_x(%rbp),%rax      # load x
 cmpq    offset_y(%rbp),%rax   # compare to y
 setl      %al                 # set low byte %rax
to 0/1
 movzbq  %al,%rax              # zero-extend to 64
bits
 movq    %rax,offset_var(%rbp)   # gen. by asg
stmt
```

Other Control Flow: switch

- Naïve: generate a chain of nested if-else if statements
- Better: switch statement is intended to allow 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 equivalent of an if to ensure expr. value is within bounds (& avoid wild jump/segfault)

Switch

- Source

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

"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,4),%ra
    x
    jmp  *%rax
      .data
swtab:
      .quad L0
      .quad L1
      .quad L2
      .text
L0: <stmts0>
L1: <stmts1>
L2: <stmts2>
```

Arrays

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

  $exp_1[exp_2]$

- x86-64

```
<evaluate exp1 (array address) in %rax>
<evaluate exp2 in %rdx>
address is (%rax,%rdx,8)  # if 8 byte
elements
```

2-D Arrays

- Subscripts start with 0
- C/C++, etc. specify 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 specifies 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

What does this program print?

```
class One {
   int tag;
   int it;

   void setTag()  {
   tag = 1; }

   int getTag() {
   return tag; }

   void setIt(int
   it) {
   this.it = it; }

   int getIt() {
   return it; }
}
```

```
class Two extends
   One {
   int it;
   void setTag() {
    tag = 2;
    it = 3;
   }

   int getThat() {
    return it;
   }

   void resetIt() {
    super.setit(42);
   }
}
```

```
public static void main(String[] args)
   {
        Two two = new Two();
        One one = two;

        one.setTag();

System.out.println(one.getTag());

        one.setIt(17);
        two.setTag();

System.out.println(two.getIt());

System.out.println(two.getThat());
        two.resetIt();

System.out.println(two.getIt());

System.out.println(two.getThat());

 }
```

Your Answer Here

Object Representation

- The naïve explanation is that an object contains
  - Fields declared in its class and in all superclasses
    - Redeclaration of a field hides (shadows) superclass instance – but the superclass field is still there
  - Methods declared in its class and all superclasses
    - Redeclaration of a method overrides (replaces) – but overridden methods can still be accessed by super…
- When a method is called, the method "inside" that particular object is called
  - Regardless of the static (compile-time) type of the variable
    - (But we really don't want to copy all those methods, do we?)

Actual Representation

- Each object contains:
  - An entry ("slot") for each field (instance variable)
    - Including shadowed and private fields in superclasses
  - A pointer to a runtime data structure for its class
    - Key component: method dispatch table (next slide)
- Basically a C struct
- Fields hidden by declarations in subclasses are *still* allocated in the object and are accessible from superclass methods

Method Dispatch Tables

- One of these per class, not per object
- Often called “vtable”, “vtbl”, or “vtab”
  - (virtual function table – term from C++)
- One pointer per method – points to beginning of method code
- Dispatch table offsets fixed at compile time in O(1) implementations

Method Tables and Inheritance

- A really simple implementation
  - Method table for each class has pointers to all methods declared in it (a dictionary)
  - Method table also contains a pointer to parent class method table
  - Method dispatch:
    - Look in current table and use if method declared locally
    - Look in parent class table if not local
    - Repeat
    - "Message not understood" if you can't find it after search
  - Actually used/needed in typical implementations of some dynamic languages (e.g. Ruby, Smalltalk, etc.)

O(1) Method Dispatch

- Idea: first part of method table for extended class has pointers for the same methods in the same order as the parent class
  - BUT pointers actually refer to overriding methods if these exist

  ∴ Method dispatch can be done with indirect jump using fixed offsets known at compile time – O(1)
    - In C: *(object->vtbl[offset])(parameters)
- Pointers to methods added in subclass are after ptrs to inherited/overridden ones in vtable

Method Dispatch Footnotes

- Don't need vtable pointer to parent class vtable for method calls, but still useful for other purposes
  - Casts and instanceof
- Multiple inheritance requires more complex mechanisms
  - Also true for multiple interfaces

Confusing Example Revisited

```
class One {
   int tag;
   int it;
   void setTag()    { tag = 1; }
   int getTag()      { return tag; }
   void setIt(int it) {this.it = it;}
   int getIt()        { return it; }
}
class Two extends One {
   int it;
   void setTag() {
      tag = 2;  it = 3;
   }
   int getThat() { return it; }
   void resetIt() { super.setIt(42); }
}
```

```
public static void main(String[] args) {
      Two two = new Two();
      One one = two;

      one.setTag();
      System.out.println(one.getTag());

      one.setIt(17);
      two.setTag();
      System.out.println(two.getIt());
      System.out.println(two.getThat());
      two.resetIt();
      System.out.println(two.getIt());
      System.out.println(two.getThat());

   }
```

Now What?

- Need to explore
  - Object layout in memory
  - Compiling field references
    - Implicit and explicit use of “this”
  - Representation of vtables
  - Object creation – new
  - Code for dynamic dispatch
  - Runtime type information – instanceof and casts

Object Layout

- Typically, allocate fields sequentially
- Follow processor/OS alignment conventions for struct/object when appropriate/available
  - Include padding bytes for alignment as needed
- Use first word of object for pointer to method table/class information
- Objects are allocated on the heap
  - No actual bits in the generated code

Object Field Access

- Source

  int n = obj.slot;

- x86-64
  - Assuming that obj is a local variable in the current method's stack frame

```
movq    offset_obj(%rbp),%rax      # load obj ptr
movq    offset_slot(%rax),%rax     # load slot
movq    %rax,offset_n(%rbp)        # store n
```

  - Same idea used to reference fields of "this"
    - Use implicit "this" parameter passed to method instead of a local variable to get object address

Local Fields

- A method can refer to fields in the receiving object either explicitly as “this.f” or implicitly as “f”
  - Both compile to the same code – an implicit “this.” is assumed if not present explicitly

Source Level View

What you write:

```
int getIt() {
  return it;
}
void setIt(int it) {
  this.it = it;
}
…
obj.setIt(42);
k = obj.getIt();
```

What you really get:

```
int getIt(ObjType this) {
  return this.it;
}
void setIt(ObjType this, int it) {
  this.it = it;
}
…
setIt(obj, 42);
k = getIt(obj);
```

x86-64 "`this`" Convention (C++)

- "`this`" is an implicit first parameter to every non-static method
- Address of object placed in %rdi for every non-static method call
- Remaining parameters (if any) in %rsi, etc.

- We'll use this convention in our project

MiniJava Method Tables (vtbls)

- Generate these as initialized data in the assembly language source program
- Need to pick a naming convention for assembly language labels; suggest:
  - For methods, classname$methodname
    - Would need something more sophisticated for overloading
  - For the vtables themselves, classname$$
- First method table entry points to superclass table (we might not use this in our project, but is helpful if you add instanceof or type cast checks)

Method Tables For Convoluted Example (gcc/as syntax)

```
class One {
   void setTag()     { … }
   int getTag()       { … }
   void setIt(int it) {…}
   int getIt()          { … }
}

class Two extends One {
   void setTag() { … }
   int getThat() { … }
   void resetIt() { … }
}
```

```
                .data

One$$: .quad  0                #
   no superclass
       .quad  One$setTag
       .quad  One$getTag
       .quad  One$setIt
       .quad  One$getIt

Two$$: .quad  One$$       #
   superclass
       .quad  Two$setTag
       .quad  One$getTag
       .quad  One$setIt
       .quad  One$getIt
       .quad  Two$getThat
       .quad  Two$resetIt
```

Method Table Layout

Key point: first method entries in `Two's` method table are pointers to methods declared in `One` in *exactly the same order*

- Actual pointers reference code appropriate for objects of each class (inherited or overridden)

∴ Compiler knows correct offset for a particular method pointer *regardless of whether that method is overridden* and regardless of the actual (dynamic) type of the object

Object Creation – `new`

Steps needed

- Call storage manager (malloc or similar) to get the raw bits
  - (and store 0's if required by the language, e.g., Java)
- Store pointer to method table in the first 8 bytes of the object
- Call a constructor with "`this`" pointer to the `new` object in %rdi and other parameters as needed
  - (Not in MiniJava since we don't have constructors)
- Result of `new` is a pointer to the `new` object

Object Creation

- Source

```
One one = new One(…);
```

- x86-64

```
movq    $nBytesNeeded,%rdi      # obj size + 8 (include space for
vtbl ptr)
call        mallocEquiv             # addr of allocated bits
returned in %rax
leaq        One$$,%rdx              # get method table address
movq    %rdx,0(%rax)                # store vtbl ptr at beginning
of object
movq    %rax,%rdi               # set up "this" for constructor
movq    %rax,offset_temp(%rbp)  # save "this" for later
<load constructor arguments>        # arguments (if needed)
call        One$One                 # call ctr if we have one (no
vtbl lookup)
movq    offset_temp(%rbp),%rax      # recover ptr to object
movq    %rax,offset_one(%rbp)   # store object reference in
variable
```

Constructor

- Why don't we need a vtable lookup to find the right constructor to call?
- Because at compile time we know the actual class (it says so right after "new"), so we can generate a call instruction to a known label
  - Same with super.method(…) or superclass constructor calls – at compile time we know all of the superclasses (need this to compile subclass and construct method tables), so we know statically what class "super.method" belongs to

Method Calls

- Steps needed
  - Parameter passing: just like an ordinary C function, except load a pointer to the object in %rdi as the first (“this”) argument
  - Get a pointer to the object’s method table from the first 8 bytes of the object
  - Jump indirectly through the method table

Method Call

- Source

  obj.m(…);

- x86-64

```
<load arguments in registers as usual>  # as needed
movq    offset_obj(%rbp),%rdi # first argument is obj ptr ("this")
movq    0(%rdi),%rax              # load vtable address into %rax
call    *offset_m(%rax)       # call function whose address is at
                              #       known offset in the vtable *
```

- * Or can use: addq $offset$_m$,%rax
  call *(%rax)

  or : movq $offset$_m$(%rax),%rax
  call *%rax

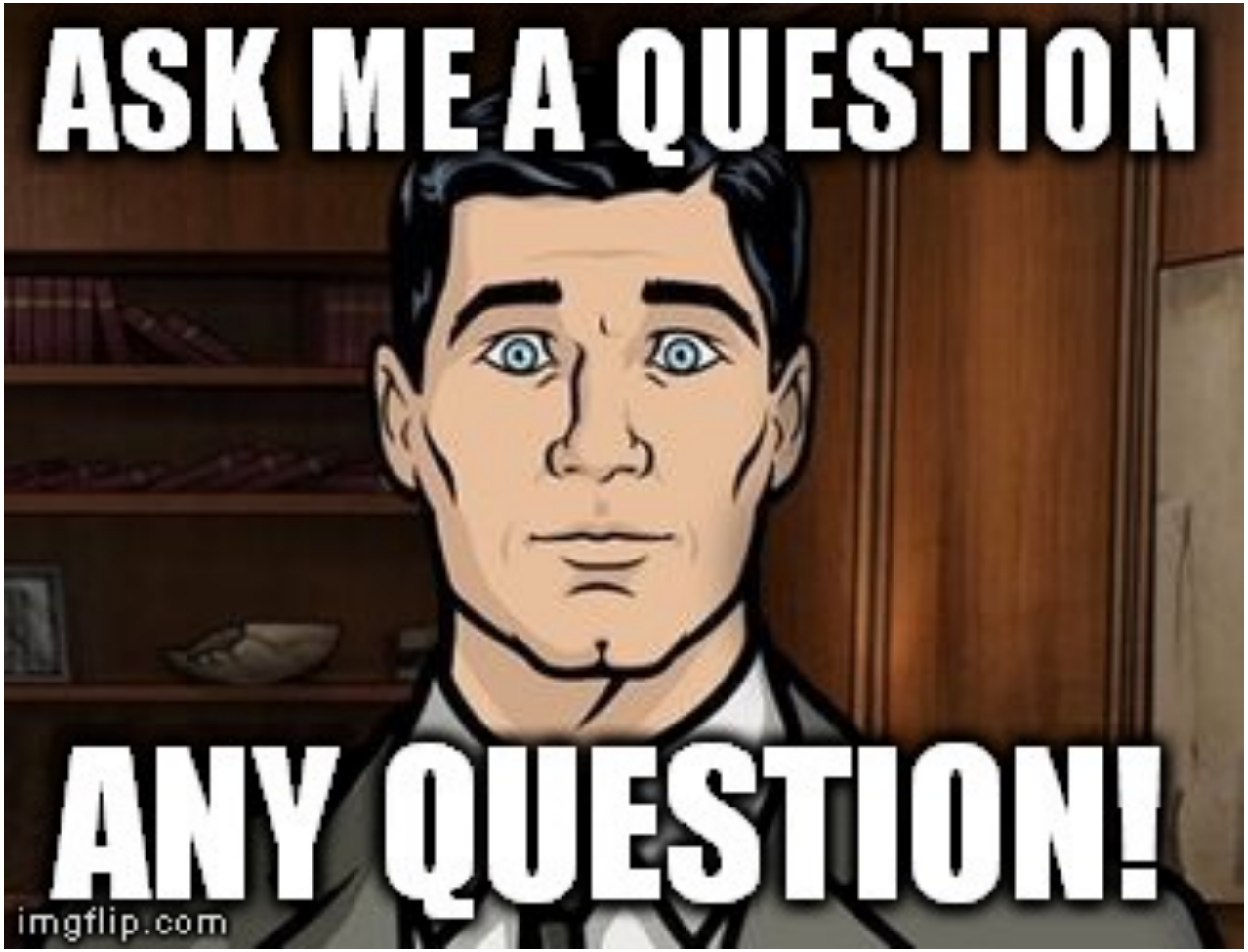
Runtime Type Checking

- We can use the method table for the class as a "runtime representation" of the class
  - Each class has one vtable at a unique address
- The test for "o instanceof C" is
  - Is o's method table pointer == &C$$?
    - If so, result is "true"
  - Recursively, get pointer to superclass method table from the method table and check that
  - Stop when you reach Object (or a null pointer, depending on whether there is a ultimate superclass of everything)
    - If no match by the top of the chain, result is "false"
- Same test as part of check for legal downcast (e.g., how to test for ClassCastException in (type)obj cast)

Coming (& Past) Attractions

- Code analysis and optimization
- Industrial-strength back end (register allocation, instruction selection & scheduling)
- Other topics as time allows
  - GC? Dynamic languages? JVM? What else?

- And simple code generation for the project



[Meme credit: imgflip.com]