CSE P 501 – Compilers

x86-64 Lite for Compiler Writers
A quick (a) introduction (b) review
[pick one]
Hal Perkins
Winter 2016
Agenda

• Overview of x86-64 architecture
  – Core part only, a little beyond what we need for the project, but not much

• Upcoming lectures...
  – Mapping source language constructs to x86
  – Code generation for MiniJava project (later)

• Rest of the quarter...
  – More sophisticated back-end algorithms
  – Compiler optimizations, analysis, and more
Some x86-64 References
(Links on course web - * = most useful)

• **x86-64 Instructions and ABI
  – Handout for University of Chicago CMSC 22620, Spring 2009, by John Reppy
• *x86-64 Machine-Level Programming
  – Earlier version of sec. 3.13 of Computer Systems: A Programmer’s Perspective, 2nd ed. by Bryant & O’Hallaron (CSE 351 textbook)
• Intel architecture processor manuals
• www.x86-64.org:
  – System V Application Binary Interface
  – Gentle Introduction to x86-64 Assembly
x86 Selected History

• Almost 40 Years of x86
  – 1978: 8086 16-bit, 5 MHz, 3µ, segmented
  – 1982: 80286 protected mode, floating point
  – 1985: 80386 32-bit, VM, 8 “general” registers
  – 1993: Pentium MMX
  – 1999: Pentium III SSE
  – 2000: Pentium IV SSE2, SSE3, HyperThreading
  – 2006: Core Duo, Core2 Multicore, SSE4, x86-64
  – 2013: Haswell 64-bit, 4-8 core, ~3 GHz, 22 nm, AVX2

• Many micro-architecture changes over the years:
  – pipelining, super-scalar, out-of-order, caching, multicore
And It’s Backward-Compatible!!

• Current processors can run 8086 code
  – You can get VisiCalc 1.0 on the web & run it!!!
• Intel descriptions of the architecture are engulfed with modes and flags; the modern processor is fairly straightforward
• Modern processors have a RISC-like core
  – Load/Store from memory
  – Register-register operations
• We will focus on basic 64-bit instructions
  – Simple instructions preferred; complex ones exist for backward-compatibility but can be slow
x86-64 Main features

• 16 64-bit general registers; 64-bit integers (but int is 32 bits usually; long is 64 bits)
• 64-bit address space; pointers are 8 bytes
• 16 SSE registers for floating point, SIMD
• Register-based function call conventions
• Additional addressing modes (pc relative)
• 32-bit legacy mode
• Some pruning of old features
x86-64 Assembler Language

• Target for our compiler project
But, the nice thing about standards…

• Two main assembler languages for x86-64
  – Intel/Microsoft version – what’s in the Intel docs
  – AT&T/GNU assembler – what we’re generating and what’s in the linked handouts
    • Use gcc –S to generate asm code from C/C++ code

• Slides use gcc/AT&T/GNU syntax
# Intel vs. GNU Assembler

- **Main differences between Intel docs and gcc assembler**

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

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

• To simplify our project we’ll use only 64-bit data (ints, pointers, even booleans!)
Processor Fetch-Execute Cycle

• Basic cycle (same as every processor you’ve ever seen)

  while (running) {
    fetch instruction beginning at rip address
    rip <- rip + instruction length
    execute instruction
  }

• Sequential execution unless a jump stores a new “next instruction” address in rip
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)
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
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)
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
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
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:
  
  \[ \text{basereg} + \text{indexreg} \times \text{scale} + \text{constant} \]

• 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]
  
  \[ \text{movq} \ %\text{rbx},(\%\text{rcx},\%\text{rax},8) \]
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”
  
  \[
  \text{mov qword ptr [rax+16],[rbp-8]}
  \]
  – Similarly for dword ptr, etc.
  – Use this if the assembler complains; otherwise ignore
  – Not an issue in GNU assembler – operand sizes encoded in opcode mnemonics
Basic Data Movement and Arithmetic Instructions

\[
\begin{align*}
\text{movq src, dst} & \quad \text{incq dst} \\
& \quad \text{dst} \leftarrow \text{src} \quad \text{dst} \leftarrow \text{dst} + 1 \\
\text{addq src, dst} & \quad \text{decq dst} \\
& \quad \text{dst} \leftarrow \text{dst} + \text{src} \quad \text{dst} \leftarrow \text{dst} - 1 \\
\text{subq src, dst} & \quad \text{negq dst} \\
& \quad \text{dst} \leftarrow \text{dst} - \text{src} \quad \text{dst} \leftarrow - \text{dst} \\
& \quad \text{(2’s complement arithmetic negation)}
\end{align*}
\]
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

\[
\begin{align*}
\text{andq src, dst} & \quad \text{dst} \leftarrow \text{dst} \& \text{src} \\
\text{orq src, dst} & \quad \text{dst} \leftarrow \text{dst} \mid \text{src} \\
\text{xorq src, dst} & \quad \text{dst} \leftarrow \text{dst} \wedge \text{src} \\
\text{notq dst} & \quad \text{dst} \leftarrow \sim \text{dst} \\
& \quad \text{(logical or 1’s complement)}
\end{align*}
\]
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)
    • 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.
Load Effective Address

• The unary & operator in C/C++
  
  \[
  \text{leaq src,dst} \quad \# \quad \text{dst} \leftarrow \text{address of src}
  \]
  
  – 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 \( r1 + \text{scale*r2} + \text{const} \)
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
jnll  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
  \[ \text{jmp}_{\text{cond}} \text{ op1,op2,label} \]
  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
  
jcc 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
– \( %\text{rsp} \leftarrow %\text{rsp} - 8 \); \( \text{memory[%rsp]} \leftarrow \%\text{rip} \)
  \( \%\text{rip} \leftarrow \text{address of label} \)
– Call address can be in a register or memory as with jumps

ret

– Pop address from top of stack and jump
– \( \%\text{rip} \leftarrow \text{memory[%rsp]} \); \( \%\text{rsp} \leftarrow %\text{rsp} + 8 \)
– **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
    \begin{verbatim}
    mov %rsp,%rbp
    pop %rbp
    \end{verbatim}
  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 puts result in %rax (if any) and restores any callee-save registers if needed
• Called function returns with:
  
  \[
  \text{movq} \, %rbp, %rsp \quad \# \text{ or use leave instead} \\
  \text{popq} \, %rbp \quad \# \text{ of movq/poplq} \\
  \text{ret}
  \]
• 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

```c
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}
```
# int sumOf(int x, int y) {
#  int a, int b;
sumOf:
  pushq  %rbp  # prologue
  movq   %rsp,%rbp
  subq   $16,%rsp
  
#  a = x;
  movq   %rdi,-8(%rbp)
  
#  b = a + y;
  movq   %rax,-16(%rbp)
  addq   %rsi,%rax
  movq   %rax,-16(%rbp)
  movq   -16(%rbp),%rax
  movq   %rbp,%rsp
  popq   %rbp
  ret
  
#  }

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

• Now that we’ve got a basic idea of the x86-64 instruction set, we need to map language constructs to x86-64
  — Code Shape

• Then need to figure out how to get compiler to generate this and how to bootstrap things to run the compiled programs (later)