Lecture 3

CIS 341: COMPILERS
Announcements

- HW1: Hellocaml!
  - is due tonight at 11:59:59pm.

- HW2: X86lite
  - Will be available soon… look for an announcement on Piazza
  - Pair-programming project
The target architecture for CIS341

X86LITE
Intel’s X86 Architecture

• 1978: Intel introduces 8086
• 1982: 80186, 80286
• 1985: 80386
• 1989: 80486
• 1993: Pentium
• 1995: Pentium Pro
• 1997: Pentium II/III
• 2000: Pentium 4
• 2003: Pentium M, Intel Core
• 2006: Intel Core 2
• 2008: Intel Core i3/i5/i7
• 2011: SandyBridge / IvyBridge
• 2013: Haswell

• AMD has a parallel line of processors
X86 vs. X86lite

• X86 assembly is very complicated:
  – 8-, 16-, 32-, 64-bit values + floating points, etc.
  – Intel 64 and IA 32 architectures have a huge number of functions
  – “CISC” complex instructions
  – Machine code: instructions range in size from 1 byte to 17 bytes
  – Lots of hold-over design decisions for backwards compatibility
  – Hard to understand, there is a large book about optimizations at just the instruction-selection level

• X86lite is a very simple subset of X86:
  – Only 64 bit signed integers (no floating point, no 16bit, no …)
  – Only about 20 instructions
  – Sufficient as a target language for general-purpose computing
X86 Schematic

Processor

Instruction Decoder

Instruction

Decoder

Control

ALU

Flags

OF

SF

ZF

OF

SF

ZF

RIP

Memory

Code & Data

Heap

Stack

0x00000000

0xffffffff

Larger Addresses

Registers

rax rbx rcx rdx
rsi rdi Rpb Rsb
r08 r09 r10 r11
r12 r13 r14 r15

0xffffffff

Zdancewic  CIS 341: Compilers
X86lite Machine State: Registers

• Register File: 16 64-bit registers
  - rax  general purpose accumulator
  - rbx  base register, pointer to data
  - rcx  counter register for strings & loops
  - rdx  data register for I/O
  - rsi  pointer register, string source register
  - rdi  pointer register, string destination register
  - rbp  base pointer, points to the stack frame
  - rsp  stack pointer, points to the top of the stack
  - R08–r15 general purpose registers

• rip    a “virtual” register, points to the current instruction
  - rip is manipulated only indirectly via jumps and return.
Simplest instruction: \texttt{mov}

- \texttt{movq} SRC, DEST \hspace{1cm} \textit{copy} SRC into DEST

- Here, DEST and SRC are \textit{operands}
- DEST is treated as a \textit{location}
  - A location can be a register or a memory address
- SRC is treated as a \textit{value}
  - A value is the \textit{contents} of a register or memory address
  - A value can also be an \textit{immediate} (constant) or a label

- \texttt{movq} \$4, \%rax \hspace{1.2cm} // move the 64-bit immediate value 4 into \texttt{rax}
- \texttt{movq} \%rbx, \%rax \hspace{1.2cm} // move the contents of \texttt{rbx} into \texttt{rax}
A Note About Instruction Syntax

• X86 presented in *two* common syntax formats

• AT&T notation: source *before* destination
  – Prevalent in the Unix/Mac ecosystems
  – Immediate values prefixed with ‘$’
  – Registers prefixed with ‘%’
  – Mnemonic suffixes: `movq` vs. `mov`  
    • `q` = quadword (4 words)
    • `l` = long (2 words)
    • `w` = word
    • `b` = byte

  ```
  movq $5, %rax
  movl $5, %eax
  ```

  **Note:** X86lite uses the AT&T notation and the 64-bit only version of the instructions and registers.

• Intel notation: destination *before* source
  – Used in the Intel specification / manuals
  – Prevalent in the Windows ecosystem
  – Instruction variant determined by register name

  ```
  mov rax, 5  
  mov eax, 5  
  ```

  dest src
X86lite Arithmetic instructions

- `negq DEST` two’s complement negation
- `addq SRC, DEST` \( \text{DEST} \leftarrow \text{DEST} + \text{SRC} \)
- `subq SRC, DEST` \( \text{DEST} \leftarrow \text{DEST} - \text{SRC} \)
- `imulq SRC, Reg` \( \text{Reg} \leftarrow \text{Reg} \times \text{SRC} \) (truncated 128-bit mult.)

Examples as written in:

\begin{verbatim}
addq %rbx, %rax // rax ← rax + rbx
subq $4, rsp // rsp ← rsp - 4
\end{verbatim}

- Note: Reg (in `imulq`) must be a register, not a memory address
X86lite Logic/Bit manipulation Operations

• notq DEST logical negation
• andq SRC, DEST DEST ← DEST && SRC
• orq SRC, DEST DEST ← DEST || SRC
• xorq SRC, DEST DEST ← DEST xor SRC

• sarq Amt, DEST DEST ← DEST >> amt (arithmetic shift right)
• shlq Amt, DEST DEST ← DEST << amt (arithmetic shift left)
• shrq Amt, DEST DEST ← DEST >>> amt (bitwise shift right)
X86 Operands

- Operands are the values operated on by the assembly instructions

- **Imm** 64-bit literal signed integer “immediate”

- **Lbl** a “label” representing a machine address the assembler/linker/loader resolve labels

- **Reg** One of the 16 registers, the value of a register is its contents

- **Ind** [base:Reg][index:Reg,scale:int32][disp] machine address (see next slide)
In general, there are three components of an indirect address:

- **Base**: a machine address stored in a register
- **Index * scale**: a variable offset from the base
- **Disp**: a constant offset (displacement) from the base

\[
\text{addr}(\text{ind}) = \text{Base} + [\text{Index} \times \text{scale}] + \text{Disp}
\]

- When used as a *location*, \( \text{ind} \) denotes the address \( \text{addr}(\text{ind}) \)
- When used as a *value*, \( \text{ind} \) denotes \( \text{Mem}[\text{addr}(\text{ind})] \), the contents of the memory address

Example:

- \(-4(\%\text{rsp})\) denotes address: \( \text{rsp} - 4 \)
- \((\%\text{rax}, \%\text{rcx}, 4)\) denotes address: \( \text{rax} + 4 \times \text{rcx} \)
- \(12(\%\text{rax}, \%\text{rcx}, 4)\) denotes address: \( \text{rax} + 4 \times \text{rcx} + 12 \)

Note: **Index cannot be \( \text{rsp} \)**

*Note*: X86lite does not need this full generality. It does not use index * scale.
X86lite Memory Model

• The X86lite memory consists of \(2^{64}\) bytes numbered \(0x00000000\) through \(0xffffffff\).
• X86lite treats the memory as consisting of 64-bit (8-byte) quadwords.
• Therefore: legal X86lite memory addresses consist of 64-bit, quadword-aligned pointers.
  – All memory addresses are evenly divisible by 8

• `leaq Ind, DEST` `DEST ← addr(Ind)` loads a pointer into DEST

• By convention, there is a stack that grows from high addresses to low addresses
• The register `rsp` points to the top of the stack
  – `pushq SRC` `rsp ← rps - 8`; `Mem[rsp] ← SRC`
  – `popq DEST` `DEST ← Mem[rsp]; rsp ← rsp + 8`
X86lite State: Condition Flags & Codes

- X86 instructions set flags as a side effect
- X86lite has only 3 flags:
  - **OF**: “overflow” set when the result is too big/small to fit in 64-bit reg.
  - **SF**: “sign” set to the sign or the result (0=positive, 1 = negative)
  - **ZF**: “zero” set when the result is 0

- From these flags, we can define *Condition Codes*
  - To compare SRC1 and SRC2, compute SRC1 – SRC2 to set the flags
  - **e** equality holds when **ZF** is set
  - **ne** inequality holds when (not **ZF**)
  - **g** greater than holds when (not **ZF**) and (not **SF**)  
    • Equivalently: ((**SF** && not **OF**) || (not **SF** && **OF**))
  - **ge** greater or equal holds when (not **SF**)
  - **le** than or equal holds when **SF** <> **OF** or **ZF**
• X86 assembly code is organized into *labeled blocks*:

```plaintext
label1:
   <instruction>
   <instruction>
   ...
   <instruction>

label2:
   <instruction>
   <instruction>
   ...
   <instruction>
```

• Labels indicate code locations that can be jump targets (either through conditional branch instructions or function calls).
• Labels are translated away by the linker and loader – instructions live in the heap in the “code segment”
• An X86 program begins executing at a designated code label (usually “main”).
Conditional Instructions

• `cmpq SRC1, SRC2`  Compute SRC2 – SRC1, set condition flags

• `setbCC DEST`  DEST’s lower byte ← if CC then 1 else 0

• `jCC SRC`  rip ← if CC then SRC else fallthrough

• Example:

  ```
  cmpq %rcx, %rax  Compare rax to ecx
  je __truelbl  If rax = rcx then jump to __truelbl
  ```
Jumps, Call and Return

- **jmp SRC** \( \text{rip} \leftarrow \text{SRC} \)** Jump to location in SRC

- **callq SRC** Push \( \text{rip} \); \( \text{rip} \leftarrow \text{SRC} \)
  - Call a procedure: Push the program counter to the stack (decrementing \( \text{rsp} \)) and then jump to the machine instruction at the address given by SRC.

- **retq** Pop into \( \text{rip} \)
  - Return from a procedure: Pop the current top of the stack into \( \text{rip} \) (incrementing \( \text{rsp} \)).
  - This instruction effectively jumps to the address at the top of the stack
Notation

• Different assemblers use different notation

• This lecture has used one variant: `Mov DEST SRC`
  – It is close to the X86lite implementation in the course projects

• Another common variant: `mov SRC, DEST`
  – Pronounce the comma as “into”
  – Write `EAX` as `%eax`
  – Write immediate 4 as `$4`

• Note: “true x86” uses different instruction suffixes for different word sizes:
  `mov` vs. `movl`
  – All of our assembly uses the “l” suffix
  – `Mov EAX -4(ECX)` is displayed as `movl -4(%ecx), %eax`

• Similarly, “true x86” adds the condition codes as suffixes for the instructions: `Jeq` for “Jump if Eq”
IMPLEMENTING X86LITE

See file: x86.ml
See: runtime.c

**DEMO: HANDCODYING X86LITE**
Compiling, Linking, Running

• To use hand-coded X86:
  1. Compile main.ml (or something like it) to either native or bytecode
  2. Run it, redirecting the output to some .s file, e.g.:
     ./handcoded.native >> test.s
  3. Use gcc to compile & link with runtime.c:
     gcc -o test runtime.c test.s
  4. You should be able to run the resulting executable:
     ./test

• If you want to debug in gdb:
  – Call gcc with the –g flag too