Lecture 3

COMPILER DESIGN
Announcements

• HW1: Hellocaml
  – Due Thursday, 1 Oct. at 23:59

• HW2: X86lite
  – Will be available next week on our course Moodle
  – Simulator / Loader for x86 assembly subset
(Simplified) Compiler Structure

Source Code (Character stream)
if (b == 0) a = 0;

Lexical Analysis

Token Stream

Parsing

Abstract Syntax Tree

Intermediate Code Generation

Intermediate Code

Code Generation

Assembly Code
CMP ECX, 0
SETBZ EAX

Front End
(machine independent)

Middle End
(compiler dependent)

Back End
(machine dependent)
Back End

Compiler Intermediate Representation

Assembly Code
Multiple Back Ends

Code in Intermediate Representation

ISAs

X86 (IA-32)

X86_64

ARMv7

ARMv8

...
Instruction Set Architecture

ISA

Instruction Semantics?
Datatypes?
Addressing Modes?
...

Programmer

Compiler

Hardware
The target architecture

**X86LITE**
• Binary Compatibility (old code runs on new hardware)

• Complex ISA (1500+ instructions)

• Multiple extensions

• Non-Intel implementations (e.g., AMD, Via)
X86 assembly is very complicated!

- 8-, 16-, 32-, 64-bit values + floating point, 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 backward 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 16-bit, no …)
- Only about 20 instructions
- Sufficient as a target language for general-purpose computing
X86 Schematic

Processor

Memory

Instruction Decoder

ALU

Control

RIP

Code & Data

Heap

Stack

Registers

Flags

rax,
rbx,
rcx,
rax

Flags:
OF,
SF,
ZF

Table:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>rbx</td>
<td>rcx</td>
<td>rdx</td>
</tr>
<tr>
<td>rsi</td>
<td>rdi</td>
<td>Rpb</td>
<td>Rsb</td>
</tr>
<tr>
<td>r08</td>
<td>r09</td>
<td>r10</td>
<td>r11</td>
</tr>
<tr>
<td>r12</td>
<td>r13</td>
<td>r14</td>
<td>r15</td>
</tr>
</tbody>
</table>

Larger Addresses

0x00000000

0xffffffff
Simplest instruction: \textit{mov}

- \texttt{movq} \texttt{SRC}, \texttt{DEST}

- Here, \texttt{DEST} and \texttt{SRC} are \textit{operands}

- \texttt{DEST} is treated as a \textit{location}
  - A location can be a register or a memory address

- \texttt{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
A Note About Instruction Syntax

AT&T notation: source before destination
- Prevalent in the Unix ecosystems
- Immediate values prefixed with ‘$’
- Registers prefixed with ‘%’
- Mnemonic suffixes: movq vs. mov
  - q = quadword (4 words)
  - l = long (2 words)
  - w = word (16-bit)
  - b = byte (8-bit)

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

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

\[
\begin{align*}
&\text{movq }$5, \%rax \quad \text{movq }$5, \%rax \\
&\text{movl }$5, \%eax \quad \text{movl }$5, \%eax \\
&\text{mov rax, 5} \quad \text{mov rax, 5} \\
&\text{mov eax, 5} \quad \text{mov eax, 5}
\end{align*}
\]
## X86 Operands

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>64-bit literal signed integer “immediate”</td>
<td>movq $4, %rax</td>
</tr>
<tr>
<td>Lbl</td>
<td>a “label” representing a machine address, the assembler/linker/loader resolves labels</td>
<td>callq FOO</td>
</tr>
<tr>
<td>Reg</td>
<td>One of the 16 registers, the value of a register is its contents</td>
<td>movq %rbx, %rax</td>
</tr>
<tr>
<td>Ind</td>
<td>machine address: [base:Reg][index:Reg][disp:int32] (base + index*8 + disp)</td>
<td>movq 12(%rax, %rcx), %rbx</td>
</tr>
</tbody>
</table>
### Arithmetic instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Example</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>negq DEST</td>
<td>2’s complement negation</td>
<td>negq %rax</td>
<td></td>
</tr>
<tr>
<td>addq SRC, DEST</td>
<td>DEST ← DEST + SRC</td>
<td>addq %rbx, %rax</td>
<td></td>
</tr>
<tr>
<td>subq SRC, DEST</td>
<td>DEST ← DEST – SRC</td>
<td>subq $4, %rsp</td>
<td></td>
</tr>
<tr>
<td>imulq SRC, Reg</td>
<td>Reg ← Reg * SRC (truncated 128-bit mult.)</td>
<td>imulq $2, %rax</td>
<td>Reg must be a register, not a memory address</td>
</tr>
</tbody>
</table>

```
movq $2, %rax
movq $3, %rbx
imulq %rbx, %rax
// rax = 6, rbx = 3
```
### Logic/Bit Manipulation instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Explanation</th>
<th>Example</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>notq DEST</td>
<td>logical negation</td>
<td>notq %rax</td>
<td>bitwise not</td>
</tr>
<tr>
<td>andq SRC, DEST</td>
<td>DEST ← DEST &amp; SRC</td>
<td>andq %rbx, %rax</td>
<td>bitwise and</td>
</tr>
<tr>
<td>orq SRC, DEST</td>
<td>DEST ← DEST</td>
<td>orq $4, %rsp</td>
<td>bitwise or</td>
</tr>
<tr>
<td>xorq SRC, DEST</td>
<td>DEST ← DEST xor SRC</td>
<td>xorq $2, %rax</td>
<td>bitwise xor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Explanation</th>
<th>Example</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>sarq Amt, DEST</td>
<td>DEST ← DEST &gt;&gt; Amt</td>
<td>sarq $4, %rax</td>
<td>arithmetic shift right</td>
</tr>
<tr>
<td>sh1q Amt, DEST</td>
<td>DEST ← DEST &lt;&lt;&lt; Amt</td>
<td>sh1q %rbx, %rax</td>
<td>logical shift left</td>
</tr>
<tr>
<td>shrq Amt, DEST</td>
<td>DEST ← DEST &gt;&gt;&gt; Amt</td>
<td>shrq $1, %rsp</td>
<td>logical shift right</td>
</tr>
</tbody>
</table>
Some X86 instructions set flags as side effects

- **OF**: “overflow” set when the result is too big/small to fit in 64-bit reg.
- **SF**: “sign” set to the sign of 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

<table>
<thead>
<tr>
<th>Code</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>e (equality)</td>
<td>ZF is set</td>
</tr>
<tr>
<td>ne (inequality)</td>
<td>(not ZF)</td>
</tr>
<tr>
<td>g (greater than)</td>
<td>(not ZF) and (SF = OF)</td>
</tr>
<tr>
<td>l (less than)</td>
<td>SF &lt;&gt; OF</td>
</tr>
<tr>
<td>ge (greater or equal)</td>
<td>(SF = OF)</td>
</tr>
<tr>
<td>le (less than or equal)</td>
<td>SF &lt;&gt; OF or ZF</td>
</tr>
</tbody>
</table>

Examples

```
movq $INTMAX, %rax
subq $1, %rax
//rax=INTMAX-1
//OF=0, SF=0, ZF=0
```

```
movq $INTMIN, %rax
subq $1, %rax
//rax=INTMAX
//OF=0, SF=0, ZF=0
```

```
movq $INTMAX, %rax
subq $-1, %rax
//rax=INTMIN (neg)
//OF=1, SF=1, ZF=0
```

```
movq $INTMIN, %rax
subq $1, %rax
//rax=INTMAX
//OF=1, SF=0, ZF=0
```
Conditional Instructions

### Instruction | Description
--- | ---
`cmpq SRC2, SRC1` | Compute SRC1 – SRC2, set condition flags
`setbCC DEST` | DEST’s lower byte ← if CC then 1 else 0
`jCC SRC` | rip ← if CC then SRC else fallthrough

```c
if (a == b){
    //something
} else {
    //somethingElse
}
//commonCode
```

```c
cmpq %rax, %rbx
je something

somethingElse:
    <instruction>
    ...
    jmp commonCode

something:
    <instruction>
    ...

commonCode:
    <instruction>
    ...
```
• X86 assembly code is organized into labeled blocks.

• 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”).

cmpq %rax, %rbx
je something

somethingElse:
  <instruction>
  ...
  jmp commonCode

something:
  <instruction>
  ...

commonCode:
  <instruction>
  ...


### Jumps, Call and Return

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp SRC</td>
<td>rip ← SRC</td>
<td>Jump to location in SRC</td>
</tr>
<tr>
<td>call SRC</td>
<td>Push rip; rip ← SRC (Call a procedure)</td>
<td>Push the program counter to the stack (decrementing rsp), and then jump to the machine instruction at the address given by SRC</td>
</tr>
<tr>
<td>ret</td>
<td>Pop into rip (Return from a procedure)</td>
<td>Pop the current top of the stack into rip (incrementing rsp). This instruction effectively jumps to the address at the top of the stack</td>
</tr>
</tbody>
</table>

```
bar:
  <instruction>
  ...
  <instruction>
  ret

foo:
  <instruction>
  ...
  <instruction>
  call bar
  ...
```
X86Lite Addressing

- In general, there are three components of an indirect address
  - **Base**: a machine address stored in a register
  - **Index**: a variable offset from the base
  - **Disp**: a constant offset (displacement) from the base

- `addr(ind) = Base + [Index * 8] + Disp`
  - When used as a *location*, `ind` denotes the address `addr(ind)`
  - When used as a *value*, `ind` denotes `Mem[addr(ind)]`, the contentsof the memory address

- **Examples:**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8(%rsp)</td>
<td>rsp - 8</td>
</tr>
<tr>
<td>(%rax, %rcx)</td>
<td>rax + 8*rcx</td>
</tr>
<tr>
<td>8(%rax, %rcx)</td>
<td>rax + 8*rcx + 8</td>
</tr>
</tbody>
</table>

- **Note**: Index cannot be `rsp`

---

// Array [0,42,2020]
// Array address 0xBEEF

```asm
movq %0xBEEF, %rax
movq %rax, %rbx  // rbx=0xBEEF
movq (%rax), %rbx  // rbx=0
movq 16(%rax), %rbx  // rbx=2020

movq $1, %rcx
movq (%rax, %rcx), %rbx  // rbx=42
movq 8(%rax, %rcx), %rbx  // rbx=2020
```
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` \( \text{DEST} \leftarrow \text{addr(Ind)} \) loads a pointer into DEST

By convention, the stack grows from high addresses to low addresses

The register `rsp` points to the top of the stack

- `pushq SRC` \( \text{rsp} \leftarrow \text{rsp} - 8; \text{Mem[rsp]} \leftarrow \text{SRC} \)
- `popq DEST` \( \text{DEST} \leftarrow \text{Mem[rsp]}; \text{rsp} \leftarrow \text{rsp} + 8 \)
DEMO: HANDCODING X86LITE
IMPLEMENTING X86LITE

See file: x86.ml