Compiler Construction: X86lite
Course structure

Borrowed liberally from UPenn CIS 341
THE X86 ARCHITECTURE
History

1978: Intel 8086
   – Introduced the x86 architecture

1985: Intel 80386
   – First 32-bit x86 processor

1995: Intel Pentium Pro
   – μ-op translation, speculative execution, &c.

2003: AMD Athlon 64
   – First 64-bit x86 processor
History

1978: Intel 8086
1985: Intel 80386
1995: Intel Pentium Pro
2003: AMD Athlon 64
2006: Intel Core 2
  – Intel accepts the demise of single core processors
2010: AMD FX
  – First consumer 8-core processor
Moore’s law
Features of X86 assembly

- 8-, 16-, 32-, 64-bit values, varying-precision floating point, vectors
- CISC: Intel 64 and IA 32 architectures have a large number of functions
- Binary encoding: instructions range in size from 1 byte to 17 bytes
- Design constrained by backwards compatibility
- Complexity makes simple decisions hard: whole books just about optimizations in instruction selection
Features of X86lite assembly

X86:
• 8-, 16-, 32-, 64-bit values
• Large number of functions
• Instructions range in size from 1 byte to 17 bytes
• Design constrained by backwards compatibility
• Complexity makes simple decisions hard

X86lite:
• 64-bit signed integers
• 20 operations
• Uniform instruction encoding
• No concerns about compatibility
• Complexity (mostly) removed

But still sufficient for general purpose computing
### "General purpose" registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose (maybe)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>Accumulator</td>
</tr>
<tr>
<td>rbx</td>
<td>Base (pointer)</td>
</tr>
<tr>
<td>rcx</td>
<td>Counter (for strings and loops)</td>
</tr>
<tr>
<td>rdx</td>
<td>Data (for I/O)</td>
</tr>
<tr>
<td>rsi</td>
<td>String source (pointer)</td>
</tr>
<tr>
<td>rdi</td>
<td>String destination (pointer)</td>
</tr>
<tr>
<td>rbp</td>
<td>Base pointer (bottom of the stack)</td>
</tr>
<tr>
<td>rsp</td>
<td>Stack pointer (top of the stack)</td>
</tr>
<tr>
<td>r08-r15</td>
<td>General purpose</td>
</tr>
</tbody>
</table>

### Special registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>rip</td>
<td>Instruction pointer</td>
</tr>
<tr>
<td>rflags</td>
<td>Conditions after last op</td>
</tr>
</tbody>
</table>
Our first instruction

\texttt{movq \textit{src} \textit{dst}}

- Copy from \textit{src} into \textit{dst}

- \textit{dst} is treated as a location
  - Either a register or a memory address

- \textit{src} is treated as a value
  - \textit{Contents} of a register or memory address
  - Or a constant or label (called an \textit{immediate})
Our first instruction

\texttt{movq src dst}

– Copy from \texttt{src} into \texttt{dst}

• \texttt{movq $4, \%rax}
  – Moves the 64 bit value 0...0100 into register \texttt{rax}

• \texttt{movq \%rbx, \%rax}
  – Moves the \textit{contents} of register \texttt{rbx} into register \texttt{rax}

• \texttt{movq (\%rbx), \%rax}
  – Moves the \textit{contents} of the memory location \textit{pointed to} by register \texttt{rbx} into register \texttt{rax}
It’s already gone complicated

AT&T syntax

- Source before destination
- Prefixes for immediates ($), registers (%)
- Mnemonic suffixes for sizes

\[
\begin{align*}
\text{movq $5, %rax} \\
\text{movl $5, %eax} \\
\text{movq $5, (%rax)}
\end{align*}
\]

- Prevalent in Unix (derived) ecosystems

We’ll stick to AT&T syntax in EECS665

Intel syntax

- Destination before source
- Instruction variant determined by register name
- Size directives, sometimes

\[
\begin{align*}
\text{mov rax, 5} \\
\text{mov eax 5} \\
\text{mov dword ptr [rax], 5}
\end{align*}
\]

- Used in Intel specifications/manuals
- Prevalent in Windows ecosystem
X86lite arithmetic

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Schematic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>negq dst</code></td>
<td><code>dst ← ~dst</code></td>
<td>Two’s complement negation</td>
</tr>
<tr>
<td><code>addq src dst</code></td>
<td><code>dst ← dst + src</code></td>
<td>Addition</td>
</tr>
<tr>
<td><code>subq src dst</code></td>
<td><code>dst ← dst - src</code></td>
<td>Subtraction</td>
</tr>
<tr>
<td><code>imulq src dst</code></td>
<td><code>dst ← dst * src</code></td>
<td>Truncated 128-bit multiplication</td>
</tr>
</tbody>
</table>

- The destination in `imulq` must be a register!

- `addq %rax, (%rbx)`
  - Memory at rbx gets rax + (memory at rbx)

- `imulq $4, %rax`
  - rax gets 4 * rax
## X86lite logical operators (the easy ones)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Schematic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>notq dst</td>
<td>dst ← ¬dst</td>
<td>Bitwise negation</td>
</tr>
<tr>
<td>andq src dst</td>
<td>dst ← src &amp; dst</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>orq src dst</td>
<td>dst ← src</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>xorq src dst</td>
<td>dst ← src ⊕ dst</td>
<td>Bitwise XOR</td>
</tr>
</tbody>
</table>
### X86lite logical operators (the hard ones)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Schematic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shlq imm dst</td>
<td>dst ← dst &lt;&lt; amt</td>
<td>Logical (or arithmetic) shift left</td>
</tr>
<tr>
<td>shrq imm dst</td>
<td>dst ← dst &gt;&gt; amt</td>
<td>Logical shift right</td>
</tr>
<tr>
<td>sarq imm dst</td>
<td>dst ← dst &gt;&gt;&gt; amt</td>
<td>Arithmetic shift right</td>
</tr>
</tbody>
</table>

- movb $-8, %al
  - %al = 1111 1000
- sarb $1, %al
  - %al = 1111 1100
- shrb $1, %al
  - %al = 0111 1110
X86(lite) operands

So what are src and dst really?

• *Immediate* values: 64-bit literal signed integers
• *Labels*: names for addresses (resolved before execution by assembler/linker/loader)
• *Registers*: of the general-purpose variety
• *Indirect* references: memory
What does “memory” mean?

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

<table>
<thead>
<tr>
<th>AT&amp;T syntax</th>
<th>Intel syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>(%rax)</td>
<td>[rax]</td>
</tr>
<tr>
<td>-4(%rax)</td>
<td>[rax-4]</td>
</tr>
<tr>
<td>(%rax, %rcx, 4)</td>
<td>[rax+rcx*4]</td>
</tr>
<tr>
<td>12(%rax, %rcx, 4)</td>
<td>[rax+rcx*4+12]</td>
</tr>
</tbody>
</table>
X86lite operands: indirect references

What does “memory” mean?

– *Base*: a machine address, stored in a register

– *Index*scale*: a variable offset from the base register

– *Displacement*: a constant offset from the (indexed) base register

<table>
<thead>
<tr>
<th>AT&amp;T syntax</th>
<th>Intel syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>(%rax)</td>
<td>[rax]</td>
</tr>
<tr>
<td>-4(%rax)</td>
<td>[rax-4]</td>
</tr>
<tr>
<td>(%rax, %rcx, 4)</td>
<td>[%rax+rcx*4]</td>
</tr>
<tr>
<td>12(%rax, %rcx, 4)</td>
<td>[%rax+rcx*4+12]</td>
</tr>
</tbody>
</table>

X86lite doesn’t have index*scale addressing
**X86(lite): indirect references**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Schematic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>leaq src dst</code></td>
<td><code>dst ← addr(src)</code></td>
<td>Load effective address</td>
</tr>
</tbody>
</table>

- Gives access to computation of indirect addresses
  - `src` must be an indirect reference

- `leaq -4(%ebx), %eax`  
  \[ \text{eax} \leftarrow \text{ebx}-4 \]  

- `leaq 4(%ebx, %ecx, 12), %eax`  
  \[ \text{eax} \leftarrow \text{ebx}+\text{ecx} \times 12 + 4 \]
X86lite condition flags

X86(lite) instructions set flags as a side effect

<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Meaning (set if...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OF</td>
<td>Overflow</td>
<td>Result doesn’t fit in 64 bits</td>
</tr>
<tr>
<td>SF</td>
<td>Sign</td>
<td>Result was negative</td>
</tr>
<tr>
<td>ZF</td>
<td>Zero</td>
<td>Result was zero</td>
</tr>
</tbody>
</table>
## X86 condition flags: comparisons

Flags can be used to define comparison.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
<th>Flags after src1-src2</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Equality</td>
<td>ZF</td>
</tr>
<tr>
<td>NE</td>
<td>Inequality</td>
<td>¬ZF</td>
</tr>
<tr>
<td>G</td>
<td>Greater than</td>
<td>(¬ZF&amp;¬SF)⊕OF</td>
</tr>
<tr>
<td>L</td>
<td>Less than</td>
<td>SF⊕OF</td>
</tr>
<tr>
<td>GE</td>
<td>Greater than or equal</td>
<td>¬SF⊕OF</td>
</tr>
<tr>
<td>LE</td>
<td>Less than or equal</td>
<td>(SF⊕OF)</td>
</tr>
</tbody>
</table>

Note: SF, ZF, OF are condition flags with meanings:

- SF (Sign Flag): Indicates the sign of the result. 0 indicates positive, 1 indicates negative.
- ZF (Zero Flag): Indicates whether the result is zero. 0 indicates non-zero, 1 indicates zero.
- OF (Overflow Flag): Indicates whether there was an overflow during the operation. 0 indicates no overflow, 1 indicates overflow.

These flags can be used to control the flow of a program based on the outcome of the comparison.
## Conditional instructions (part 1)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Schematic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpq src1 src2</td>
<td>cmpq src1 src2</td>
<td>Set flags based on src2-src1</td>
</tr>
<tr>
<td>setCC dst</td>
<td>dst ← 1 if CC, 0 otherwise</td>
<td>dst set based on given condition code</td>
</tr>
<tr>
<td>movq $4, %rbx</td>
<td></td>
<td>%rbx = ... 0100</td>
</tr>
<tr>
<td>movq $5, %rcx</td>
<td></td>
<td>%rcx = ... 0101</td>
</tr>
<tr>
<td>cmpq %rbx, %rcx</td>
<td></td>
<td>of = 0, zf = 0, sf = 0</td>
</tr>
<tr>
<td>setg %rax</td>
<td></td>
<td>%rax = ... 0001</td>
</tr>
</tbody>
</table>
Code blocks and labels

- X86 assembly is organized into labeled blocks
- Labels indicate jump targets (either through conditionals or function calls)
- Labels are translated away by the linker and loader
- Designated label to start execution

```assembly
_factorial:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl 8(%ebp), %eax
movl %eax, -4(%ebp)
movl $1, -8(%ebp)
LBB0_1:
cmpl $0, -4(%ebp)
jle LBB0_3
movl -8(%ebp), %eax
imull -4(%ebp), %eax
movl %eax, -8(%ebp)
movl -4(%ebp), %eax
subl $1, %eax
movl %eax, -4(%ebp)
jmp LBB0_1
LBB0_3:
movl -8(%ebp), %eax
addl $8, %esp
popl %ebp
retl
```
### Conditional instructions (part 2)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Schematic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cmpq src1 src2</code></td>
<td><code>dst ← 1 if CC, 0 otherwise</code></td>
<td>Compare <code>src1</code> and <code>src2</code></td>
</tr>
<tr>
<td><code>setCC dst</code></td>
<td><code>dst ← 1 if CC, 0 otherwise</code></td>
<td><code>dst</code> set based on given condition code</td>
</tr>
<tr>
<td><code>jmp src</code></td>
<td><code>rip ← src</code></td>
<td>Jumps to <code>src</code></td>
</tr>
<tr>
<td><code>jCC dst</code></td>
<td><code>rip ← dst if CC</code></td>
<td>Jump if condition</td>
</tr>
<tr>
<td>...</td>
<td><code>%ebx = 4, %ecx = 5</code></td>
<td></td>
</tr>
<tr>
<td><code>cmpq %rbx, %rcx</code></td>
<td><code>of = 0, zf = 0, sf = 0</code></td>
<td></td>
</tr>
<tr>
<td><code>jg _dest</code></td>
<td><code>rip = _dest</code></td>
<td></td>
</tr>
</tbody>
</table>
The X86lite/C memory model

X86lite assumes $2^{64}$ bytes of memory.

Conventionally divided into three parts:

- The code & data (or "text") segment stores compiled code, constant data, &c.
The X86lite/C memory model

The heap:
• Starts low in memory and grows upwards.
• Contains dynamically allocated objects

Heap management in C:
• Objects allocated by “malloc”
• Deallocated via “free”

Heap management in Haskell:
• “Bump” allocation
• Deallocation via garbage collection
The X86lite/C memory model

The stack:
- Starts high, grows downwards
  - Register `rsp` points to the “top” of the stack, `rbp` points to the bottom of the current stack frame.
- Stores function arguments, return addresses, and local variables

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Schematic</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pushq src</code></td>
<td><code>rsp ← rsp – 8; Mem[rsp] ← src</code></td>
</tr>
<tr>
<td><code>popq dst</code></td>
<td><code>dst ← Mem[rsp]; rsp ← rsp + 8</code></td>
</tr>
</tbody>
</table>
Call, and return

Procedure calls are implemented using the stack:
• To call a procedure: push the current rip to the stack, then jump
• To return: jump to the address on top of the stack

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Schematic</th>
<th>Description</th>
</tr>
</thead>
</table>
| callq src   | push rip
rip ← src | Procedure call          |
| retq        | pop rip            | Return from procedure               |
Calling conventions

Implement function calls in terms of callq/ret: need to specify

• Locations for function arguments

• Treatment of registers:
  – “Caller-save”: freely usable by called code; caller responsible for saving values
  – “Callee-save”: called code responsible for restoring values at call

• Protocol for stack-allocated arguments
  – Caller cleans
  – Callee cleans: variable argument functions harder
32-bit calling conventions

- EAX, ECX, EDX are caller-save. All others are callee-save
- Return value in EAX, or in EAX and EDX

**cdecl:**
- Arguments passed right-to-left
- Caller cleans parameters after return
- Allows variable-length argument lists
- Standard in stand-alone C programs and Unix-y operating systems

**pascal:**
- Arguments passed right-to-left
- Callee cleans parameters before return
- “Fractionally faster” (in 1985)
- Used in Win32 API calls

We’re only going to use cdecl-like conventions
32-bit call stacks

Scenario: $f(x_1, x_2)$ with local variable $v$ calls $g(y)$ with local variables $w_1, w_2$

Stack frame (at EBP) contains:
- Local variables \((\text{above EBP})\)
- Callee-save registers \((\text{above EBP})\)
- Return address \((\text{below EBP})\)
- Parameters \((\text{below EBP})\)
To call function $f(e_1, e_2, ..., e_n)$

1. Save caller-save registers
2. Push values of $e_n$...$e_1$ onto the stack
3. callq $f$

After $f$ returns:

1. Clean values of $e_n$...$e_1$ from the stack
2. Restore caller-saved registers

Note: return value in eax, edx.
32-bit function calls: caller protocol

To call function $f(e_1, e_2, e_3)$:

- push %edx
- push %ecx
- push -$4(%ebp)
- push $42
- push %ebx
- call _f
- addl $12, %esp
- pop %ecx
- pop %edx

```
push %edx
push %ecx
push -4(%ebp)
push $42
push %ebx
call _f
addl $12, %esp
pop %ecx
pop %edx
```

Save registers
Arguments
Function call
Clean arguments
Restore registers

ESP

Return addr.

$\mathbf{v}_1$
$\mathbf{v}_2$
$\mathbf{v}_3$
eax
ecx
edx
...local variables...

Larger addresses
32-bit function calls: callee protocol

To implement function f(e1,...,en):
1. Save old frame pointer
2. Set up new frame pointer
3. Allocate space for local variables
4. Save callee-save registers

To return from f:
1. Restore callee-save registers
2. Deallocate local variables
3. Restore frame pointer
4. Return
32-bit function calls: callee protocol

To implement function \( f(x_1, \ldots, x_n) \):

\[
\_f:
\]

- push `%ebp`
- mov `%esp`, `%ebp`
- sub $12, `%esp`
- push `%esi`
- push `%edi`
- ...
- pop `%edi`
- pop `%esi`
- mov `%ebp`, `%esp`
- pop `%ebp`
- ret

**Frame setup**
- **Local variables**
- **Save registers**
- **Function body**
- **Restore registers**
- **Local variables**
- **Old frame pointer**
- **Return**
64-bit function calls

• Callee-save: rbp, rbx, r12-r15
• `Return value in rax

• Parameters
  – 1-6: rdi, rsi, rdx, rcx, r8, r9
  – 7+: on the stack

• 128 byte “red zone”