Machine-Level Programming: Introduction

CMPU 224 – Computer Organization
Jason Waterman
Intel x86 Processors

• Dominate laptop/desktop/server market

• Evolutionary design
  • Backwards compatible with the 8086, introduced in 1978
  • Added more features as time goes on

• Complex instruction set computer (CISC)
  • Many different instructions with many different formats
    • But only small subset encountered with Linux programs
  • Hard to match performance of Reduced Instruction Set Computers (RISC)
  • But Intel has done just that!
    • In terms of speed at least, less so for low power
## Intel x86 Processors

### Machine Evolution

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1979</td>
<td>29k</td>
<td>5-10</td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>0.3M</td>
<td>16-33</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>3.1M</td>
<td>60-300</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2000</td>
<td>45M</td>
<td>1400-1500</td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>2006</td>
<td>291M</td>
<td>1860-2670</td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>1700-3900</td>
</tr>
<tr>
<td>Core i7 Skylake</td>
<td>2015</td>
<td>1.75B</td>
<td>2800-4000</td>
</tr>
</tbody>
</table>

### Added Features

- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores
- Built-in Graphics Processor
Definitions

- **Architecture:** (also ISA: instruction set architecture) The parts of a processor design that one needs to understand or write assembly/machine code
  - Examples: instruction set specification, registers
- **Microarchitecture:** Implementation of the architecture
  - Can have many microarchitectures implement the same ISA e.g., different cache sizes and core frequencies
- **Code Forms:**
  - **Machine Code:** The byte-level programs that a processor executes
  - **Assembly Code:** A text representation of machine code
- **Example ISAs:**
  - Intel/AMD: IA32, x86-64
  - ARM: ARMv6, ARMv7E, ARMv8
  - RISC-V: RV32I, RV64I, RV64G
Assembly/Machine Code View

**Programmer-Visible State**

- **Register file**
  - Heavily used program data

- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

- **PC: Program counter**
  - Address of next instruction
  - Called “RIP” (Instruction Pointer Register) in X86-64

- **Memory**
  - Byte addressable array
  - Code and user data
Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Use basic optimizations (`-Og`) [New to recent versions of GCC]
  - Put resulting binary in file `p`
Compiling Into Assembly

C Code (mult_and_add.c)

```c
long mult_and_add(long x, long y, long z) {
    long product = x * y;
    return z + product
}
```

Generated x86-64 Assembly

```assembly
mult_and_add:
    imulq %rsi, %rdi
    leaq (%rdi, %rdx), %rax
    retq
```

Obtain (on a lab machine) with command

```
gcc -Og -S mult_and_add.c
```

Produces the file `mult_and_add.s`

**Warning**: You will get very different results on other machines (e.g., MacOS) due to different versions of gcc and different compiler settings
Assembly Characteristics: Data Types

• Integer data of 1 (char), 2 (short), 4 (int), or 8 (long) bytes
  • Data values (signed and unsigned)
  • Addresses (pointers)

• Floating point data of 4 (float) or 8 (double) bytes
  • Stored in a different set of registers

• Code: Byte sequences encoding series of instructions

• No aggregate types such as arrays or structures
  • Just contiguously allocated bytes in memory
Assembly Characteristics: Operations

- Perform arithmetic function on registers or memory data
  - Math and logic operations

- Transfer data between memory and register
  - Load data from memory into register **READ**
  - Store register data into memory **WRITE**

- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches
## Object Code

### Code for `mult_and_add`

<table>
<thead>
<tr>
<th>Hex Value</th>
<th>Binary Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x48</td>
<td>01001000</td>
</tr>
<tr>
<td>0x0f</td>
<td>00000111</td>
</tr>
<tr>
<td>0xaf</td>
<td>01011011</td>
</tr>
<tr>
<td>0xfe</td>
<td>01111110</td>
</tr>
<tr>
<td>0x48</td>
<td>01001000</td>
</tr>
<tr>
<td>0x8d</td>
<td>01000001</td>
</tr>
<tr>
<td>0x04</td>
<td>00000100</td>
</tr>
<tr>
<td>0x17</td>
<td>00010111</td>
</tr>
<tr>
<td>0xc3</td>
<td>11000011</td>
</tr>
</tbody>
</table>

- **Assembler:** `gcc -c mult_and_add.s`
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
    - Missing linkages between code in different files
    - The number of bytes per instruction varies

- **Linker**
  - Resolves references between files
  - Combines with code from run-time libraries
    - E.g., code for `malloc()`, `printf()`
  - Some libraries are dynamically linked
    - Linking occurs when program begins execution
Disassembling Object Code

• Disassembler
  
  `objdump -d mult_and_add`

  • Useful tool for examining object code
  • Analyzes bit pattern of series of instructions
  • Produces a rendition of the assembly code
  • Can be run on either executable binary program or .o file

• Disassembled

```
00000000004004e7 <mult_and_add>:
 4004e7: 48 0f af fe       imul   %rsi,%rdi
 4004eb: 48 8d 04 17       lea    (%rdi,%rdx),%rax
 4004ef: c3                retq
```
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%r8</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%r9</td>
</tr>
<tr>
<td>%rcx</td>
<td>%r10</td>
</tr>
<tr>
<td>%rdx</td>
<td>%r11</td>
</tr>
<tr>
<td>%rsi</td>
<td>%r12</td>
</tr>
<tr>
<td>%rdi</td>
<td>%r13</td>
</tr>
<tr>
<td>%rsp</td>
<td>%r14</td>
</tr>
<tr>
<td>%rbp</td>
<td>%r15</td>
</tr>
</tbody>
</table>
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Can reference low-order 4 bytes
Integer Registers

• The lower portion of each 64-bit register can be referred to by alternate register names.

• All 64-bit registers start with r.

• All named 32-bit registers start with e.
Assembly instructions

• Instruction Format: **ins Source, Dest**
  • **ins**: opcode (instruction)
  • **source, dest**: operands → VALUE
    • Most opcodes have two operands, but some only have one

• Operand Types
  • **Immediate**: Constant integer data
    • Example: $0x400, $−533
    • Like C constants, but prefixed with $
    • Encoded with either 1, 2, or 4 bytes
  • **Register**: One of the integer register names prefixed with a %
    • Example: %rax, %r13
    • Some registers have special uses for particular instructions
  • **Memory**: Consecutive bytes of memory at a given address
    • Simplest example: (%rax)
    • Various other “addressing modes”
    • **Note**: an address can also be specified with as a constant without the $ prefix
Operand Combinations

Cannot do memory-memory operations with a single instruction

Specific instructions may have other operand restrictions
Our first instruction: load effective address (lea)

• `leaq mem, reg`
  • Computes the address of the source operand and saves it in the destination register
• Computes the memory address for array and structure access
• Compiler will also use it to perform simple arithmetic
Normal and Simple Memory Addressing Modes

• Normal (R) \( \text{Mem}[\text{Reg}[R]] \)
  • Register R specifies memory address

  \texttt{leaq (\%rcx),\%rax}

• Displacement D(R) \( \text{Mem}[\text{Reg}[R]+D] \)
  • Register R specifies start of memory region
  • Constant displacement D specifies offset

  \texttt{leaq 8(\%rbp),\%rdx}
Indexed Memory Addressing Modes

• Indexed \((R_b, R_i)\) \(\text{Mem}[\text{Reg}[R_b] + \text{Reg}[R_i]]\)
  • Register \(R_b\) often specifies base memory address
  • Register \(R_i\) often acts as an index
  • Often used in accessing arrays
    \texttt{leaq (\%rcx, \%rdx),\%rax}

• Scaled Indexed \((R_b, R_i, s)\) \(\text{Mem}[\text{Reg}[R_b] + \text{Reg}[R_i]*s]\)
  • \(s\) is called the scaling factor
  • Must be 1, 2, 4, 8 (\textit{why these numbers?})
  • \texttt{leaq (\%rcx, \%rdx, 8),\%rax}
  • \(R_b\) is optional \((, R_i, s)\) is a valid operand
Complete Memory Addressing Modes

• Most General Form

\[ D(R_b, R_i, S) \quad \text{Mem}[\text{Reg}[R_b] + S*\text{Reg}[R_i] + D] \]

• D: Constant “displacement” 1, 2, or 4 bytes
• \( R_b \): Base register: Any of 16 integer registers
• \( R_i \): Index register: Any, except for \%rsp
• S: Scale: 1, 2, 4, or 8
### leaq arithmetic

- Compilers often use the `leaq` instruction for performing arithmetic instead of computing addresses.
- `leaq` can perform arithmetic in the form of $x + y \times s$.
  - Where $s$ is 1, 2, 4, or 8.

```plaintext
mult_and_add:
    imulq %rsi, %rdi
    leaq (%rdi, %rdx), %rax
    retq
```
### Address Computation Examples

<table>
<thead>
<tr>
<th>Address Computation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%%rdx)</td>
<td>0xf000</td>
</tr>
<tr>
<td>(%rdx, %rcx)</td>
<td>0x0100</td>
</tr>
<tr>
<td>(%rdx, %rcx,4)</td>
<td></td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td></td>
</tr>
</tbody>
</table>
Address Computation Examples

<table>
<thead>
<tr>
<th>%rdx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rcx</td>
<td>0x0100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Address Computation Examples

<table>
<thead>
<tr>
<th>%rdx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rcx</td>
<td>0x0100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80(%rdx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(%rdx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Address Computation Examples

### Variables

<table>
<thead>
<tr>
<th>%rdx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rcx</td>
<td>0x0100</td>
</tr>
</tbody>
</table>

### Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>
Instruction suffixes

• Most assembly instructions take a suffix:
  • b (byte: 1 byte)
  • w (word: 2 bytes)
  • l (long word: 4 bytes)
  • q (quad word: 8 bytes)

• Often used with the low-order registers (e.g., %eax, %ax, %al)
  • leab (%rax), %al
  • leaw (%rax), %sp
  • leal (%rax), %eax

• In general, only the specific register bytes or memory locations are modified
  • Exception: “l” instructions that have a register as a destination will set the upper order bits to 0
Machine Programming Basics: Summary

• History of Intel processors and architectures
  • Evolutionary design leads to many quirks and artifacts

• C, assembly, machine code
  • New forms of visible state: program counter, registers, ...
  • Compilers transform statements, expressions, procedures into low-level instruction sequences

• Assembly Basics: Registers, operands, address computation

• Next time: more instructions