Machine-Level Programming: Basics

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

• 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: IA32, Itanium, 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`

![Diagram showing the process from C program to executable program](image)
Compiling Into Assembly

C Code (mult_and_add.c)

```c
long add(long x, long y);

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

Generated x86-64 Assembly

```
mult_and_add:
    imulq %rdi, %rsi
    movq %rdx, %rdi
    call add
    retq
```
Assembly Characteristics: Data Types

• “Integer” data of 1 (char), 2 (short), 4 (int), or 8 (long) bytes
  - Data values
  - 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
  • Store register data into memory

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

Code for `mult_and_add`

```
0x040049c:
0x48
0x0f
0xaf
0xf7
0x48
0x89
0xd7
0xe8
0xef
0xff
0xff
0xff
0xc3
```

- Total of 13 bytes
- Each instruction 1, 3, 4, or 5 bytes
- Starts at address 0x0400595

```
mult_and_add:
imulq %rdi, %rsi
movq %rdx, %rdi
call add
retq
```

- **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

- **Linker**
  - Resolves references between files
  - Combines with static 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 approximate rendition of assembly code
  • Can be run on either executable binary program or `.o` file

• Disassembled

```
000000000040049c <mult_and_add>:
40049c:  48 0f af f7     imul  %rdi,%rsi
4004a0:  48 89 d7     mov  %rdx,%rdi
4004a3:  e8 ef ff ff     callq 400497 <add>
4004a8:  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

- Can reference low-order 4 bytes
## Integer Registers

<table>
<thead>
<tr>
<th>64-bit register</th>
<th>Lower 32 bits</th>
<th>Lower 16 bits</th>
<th>Lower 8 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>eax</td>
<td>ax</td>
<td>al</td>
</tr>
<tr>
<td>rbx</td>
<td>ebx</td>
<td>bx</td>
<td>bl</td>
</tr>
<tr>
<td>rcx</td>
<td>ecx</td>
<td>cx</td>
<td>cl</td>
</tr>
<tr>
<td>rdx</td>
<td>edx</td>
<td>dx</td>
<td>dl</td>
</tr>
<tr>
<td>rsi</td>
<td>esi</td>
<td>si</td>
<td>sil</td>
</tr>
<tr>
<td>rdi</td>
<td>edi</td>
<td>di</td>
<td>dil</td>
</tr>
<tr>
<td>rbp</td>
<td>ebp</td>
<td>bp</td>
<td>bpl</td>
</tr>
<tr>
<td>rsp</td>
<td>esp</td>
<td>sp</td>
<td>spl</td>
</tr>
<tr>
<td>r8</td>
<td>r8d</td>
<td>r8w</td>
<td>r8b</td>
</tr>
<tr>
<td>r9</td>
<td>r9d</td>
<td>r9w</td>
<td>r9b</td>
</tr>
<tr>
<td>r10</td>
<td>r10d</td>
<td>r10w</td>
<td>r10b</td>
</tr>
<tr>
<td>r11</td>
<td>r11d</td>
<td>r11w</td>
<td>r11b</td>
</tr>
<tr>
<td>r12</td>
<td>r12d</td>
<td>r12w</td>
<td>r12b</td>
</tr>
<tr>
<td>r13</td>
<td>r13d</td>
<td>r13w</td>
<td>r13b</td>
</tr>
<tr>
<td>r14</td>
<td>r14d</td>
<td>r14w</td>
<td>r14b</td>
</tr>
<tr>
<td>r15</td>
<td>r15d</td>
<td>r15w</td>
<td>r15b</td>
</tr>
</tbody>
</table>
Assembly instructions

• Instruction Format:

\[ \text{ins } \text{Source, Dest} \]

• Operand Types
  • **Immediate:** Constant integer data
    • Example: \(0x400, -533\)
    • Like C constant, but prefixed with ‘\$’
    • Encoded with 1, 2, or 4 bytes
  • **Register:** One of 16 integer registers
    • 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 “address modes”
    • Note: It can also be a constant without dollar sign ($)
Our first instruction: Move (mov)

- `movq Source, Dest`
  - Moves (copies) the source operand to the destination operand
  - Has many purposes
    - Load an immediate value (number) into a register
    - Copy a value from one register into another register
    - Read a value from a memory address
    - Write a value from a memory address

- In other hardware architectures, these operations are done with several different instructions
**movq Operand Combinations**

```
Source          Dest          Src,Dest

movq

\{ 

\text{Imm} 
\quad \begin{cases} 
\text{Reg} & \text{movq } 0x4, %rax \\
\text{Mem} & \text{movq } -147, (\%rax) 
\end{cases}

\text{Reg} 
\quad \begin{cases} 
\text{Reg} & \text{movq } \%rax, \%rdx \\
\text{Mem} & \text{movq } \%rax, (\%rdx) 
\end{cases}

\text{Mem} 
\quad \text{Reg} & \text{movq } (\%rax), \%rdx 
\}\n```

**Cannot do memory-memory transfer with a single instruction**
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)
  • movb $-17, %al
  • movw %bp, %sp
  • movl $0x4050, %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
Normal Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
  - Pointer dereferencing in C

\[
\text{movq} \ (\%rcx),\ %rax
\]
Simple Memory Addressing Modes

• Normal (R) Mem[Reg[R]]
  • Register R specifies memory address
  • Pointer dereferencing in C

  movq (%rcx),%rax

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

  movq 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
    \(\text{movq (}\%\text{rcx, }\%\text{rdx}),\%\text{rax}\)

- 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 (why these numbers?)
  - \(\text{movq (}\%\text{rcx, }\%\text{rdx, 8}),\%\text{rax}\)
  - \(R_b\) is optional \((, R_i, s)\) is a valid operand
Complete Memory Addressing Modes

• Most General Form

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

• **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
## 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></td>
<td></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>

<table>
<thead>
<tr>
<th>%rdx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rcx</td>
<td>0x0100</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><code>0x8(%rdx)</code></td>
<td><code>0xf000 + 0x8</code></td>
<td><code>0xf008</code></td>
</tr>
<tr>
<td><code>(rdx,rcx)</code></td>
<td><code>0xf000 + 0x100</code></td>
<td><code>0xf100</code></td>
</tr>
<tr>
<td><code>(rdx,rcx,4)</code></td>
<td><code>0xf000 + 4*0x100</code></td>
<td><code>0xf400</code></td>
</tr>
<tr>
<td><code>0x80(,%rdx,2)</code></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>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>
## Address Computation Examples

<table>
<thead>
<tr>
<th>Operand</th>
<th>Address</th>
<th>Value at Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%rax)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4(%rax)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9(%rax, %rdx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>260(%rcx, %rdx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFC(, %rcx, 4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%rax, %rdx, 4)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x100</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x1</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
</tr>
</tbody>
</table>
### Address Computation Examples

<table>
<thead>
<tr>
<th>Operand</th>
<th>Address Calculation</th>
<th>Value at Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x104</td>
<td>0x104</td>
<td>0xAB</td>
</tr>
<tr>
<td>(%rax)</td>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>4(%rax)</td>
<td>0x100 + 4 = 0x104</td>
<td>0xAB</td>
</tr>
<tr>
<td>9(%rax, %rdx)</td>
<td>0x100 + 3 + 9 = 0x10C</td>
<td>0x11</td>
</tr>
<tr>
<td>260(%rcx, %rdx)</td>
<td>1 + 3 + 260 * 264/8 = 0x108</td>
<td>0x13</td>
</tr>
<tr>
<td>0xFC(, %rcx, 4)</td>
<td>1*4 + 0xFC = 0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>(%rax, %rdx, 4)</td>
<td>3*4 + 0x100 = 0x10C</td>
<td>0x11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x100</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x1</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
</tr>
</tbody>
</table>
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, ...
  • Compiler must transform statements, expressions, procedures into low-level instruction sequences

• Assembly Basics: Registers, operands, move
  • The x86-64 move instructions cover wide range of data movement forms