Machine Level Programming: x86-64 History
Intel x86 Processors

- Dominate laptop/desktop/server market
- Evolutionary design
  - Backwards compatible up until 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 typical programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
    - But, Intel has done just that!
      - In terms of speed. Less so for low power.
# Intel x86 Evolution: Milestones

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
<td>5-10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>First 16-bit Intel processor. Basis for IBM PC &amp; DOS</td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>First 32 bit Intel processor, referred to as IA32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added “flat addressing”, capable of running Unix</td>
<td></td>
</tr>
<tr>
<td>Pentium 4E</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>First 64-bit Intel x86 processor, referred to as x86-64</td>
<td></td>
</tr>
<tr>
<td>Core 2</td>
<td>2006</td>
<td>291M</td>
<td>1060-3500</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>First multi-core Intel processor</td>
<td></td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>1700-3900</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Four cores</td>
<td></td>
</tr>
</tbody>
</table>

---
x86 Clones: Advanced Micro Devices (AMD)

- **Historically**
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper

- **Then**
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension to 64 bits

- **Recent Years**
  - Intel got its act together
    - Leads the world in semiconductor technology
  - AMD has fallen behind
    - Relies on external semiconductor manufacturer
Intel’s 64-Bit History

- **2001: Intel Attempts Radical Shift from IA32 to IA64**
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing

- **2003: AMD Steps in with Evolutionary Solution**
  - x86-64 (now called “AMD64”)

- **Intel Felt Obligated to Focus on IA64**
  - Hard to admit mistake or that AMD is better

- **2004: Intel Announces EM64T extension to IA32**
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!

- **All but low-end x86 processors support x86-64**
  - But, lots of code still runs in 32-bit mode
Our Coverage

- x86-64
  - The standard
  - Emitted by commands like…
    - $ gcc hello.c

- Book
  - Book covers x86-64
  - This is why the latest edition is critical.
  - Prior to this it was 32-bit
  - Rare case where new edition of textbook is actually worth it!
C, Assembly & Machine code
Definitions

- **Architecture:** (also ISA: instruction set architecture): The parts of a processor design that one needs to understand to write assembly/machine code.
  - Ex. instruction set specification, registers…..
  - *Example ISAs*
    - Intel: x86, IA32, x86-64
    - ARM: Used in almost all mobile phones

- **Code Forms:**
  - *Machine Code*: The byte-level programs that a processor executes.
    (target of compiler)
  - *Assembly Code*: A text representation of machine code
Machine-level programmer-visible state

- Program counter
  - Address of next instruction
  - Called “RIP” (x86-64)
- Register file
  - Heavily used program data
- Condition codes
  - Status info on most recent operation
  - Used for conditional branching

- Memory
  - Byte addressable array
  - Code and user data
  - Stack to support procedures
Turning C into Machine/Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc –Og p1.c p2.c -o p`
  - Use basic optimizations (`-Og`)
  - Put resulting binary in file `p`
Compiling Into Assembly

- Generated using command: `gcc -Og -S sum.c`
  - `-Og` tells gcc “do very little optimization”.
- Produces file `sum.s`
- Note: Will get very different results different machines due to different versions of gcc and different compiler settings.
- Note: For now we ignore all instructions that begin with a dot (.)
Assembly Characteristics: Data Types

- **Integers**
  - 1, 2, 4, or 8 bytes
  - Bit values (unsigned or not, doesn’t matter!)
  - Addresses (void pointers)

- **Floating point numbers**
  - Floating point data of 4 or 8 bytes.
  - We will skip this in our treatment of MLP

- **Code**
  - Byte sequences encoding series of instructions.

- **Data structures**
  - No aggregate types such as arrays or structures, *just contiguously allocated bytes in memory.*
  - Constructions of the compiler
Assembly Characteristics: Operations

- **Perform arithmetic functions** on register or memory data

- **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 and loops
  - All built from combinations of simple instructions!

- **Note**: Very limited in what can be done in one instruction - does only one thing: move data, single simple arithmetic operation, memory dereference.
Object Code (Machine Code)

- **Assembler**
  - 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

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0400595:</td>
<td></td>
</tr>
<tr>
<td>0x53</td>
<td></td>
</tr>
<tr>
<td>0x48</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0xd3</td>
<td></td>
</tr>
<tr>
<td>0xe8</td>
<td></td>
</tr>
<tr>
<td>0xf2</td>
<td></td>
</tr>
<tr>
<td>0xff</td>
<td></td>
</tr>
<tr>
<td>0xff</td>
<td></td>
</tr>
<tr>
<td>0xff</td>
<td></td>
</tr>
<tr>
<td>0x48</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0x03</td>
<td></td>
</tr>
<tr>
<td>0x5b</td>
<td></td>
</tr>
<tr>
<td>0xc3</td>
<td></td>
</tr>
</tbody>
</table>
```

Total of 14 bytes. Each instruction 1, 3, or 5 bytes. Starts at address 0x0400595
Object Code Example

**C Code**
- Store value \( t \) where designated by \( \text{dest} \)

**Assembly**
- Move 8-byte value to memory
- Operands:
  - \( t: \) Register \( %rax \)
  - \( \text{dest}: \) Register \( %rbx \)
  - \( \*\text{dest}: \) Memory \( M[\%rbx] \)

**Object Code**
- 3-byte instruction
- Stored at address \( 0x40059e \)
Disassembling Object Code

- **Disassembler**: `objdump --d sum`
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file

```
0000000000400595 <sumstore>:
    400595:  53                   push   %rbx
    400596:  48 89 d3             mov    %rdx,%rbx
    400599:  e8 f2 ff ff ff       callq  400590 <plus>
    40059e:  48 89 03             mov    %rax,(%rbx)
    4005a1:  5b                   pop    %rbx
    4005a2:  c3                   retq
```
Alternate Disassembler

- Within **gdb** debugger **disassemble** **sumstore**
- Disassemble procedure **x/14xb sumstore**
  - Examine the 14 bytes starting at **sumstore**

Dump of assembler code for function sumstore:

```
0x0000000000400595 <+0>:   push   %rbx
0x0000000000400596 <+1>:   mov    %rdx,%rbx
0x0000000000400599 <+4>:   callq  0x400590 <plus>
0x000000000040059e <+9>:   mov    %rax,(%rbx)
0x00000000004005a1 <+12>:  pop    %rbx
0x00000000004005a2 <+13>:  retq
```
What Can be Disassembled?

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source
- Might be illegal

% objdump -d WINWORD.EXE

WINWORD.EXE:  file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55             push   %ebp
30001001:  8b ec          mov    %esp,%ebp
30001003:  6a ff          push   $0xffffffff
30001005:  68 90 10 00 30 push   $0x30001090
3000100a:  68 91 dc 4c 30 push   $0x304cdc91
Assembly Basics: Registers, Operands, Move
## x86-64 Integer Registers

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

Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
Register Operators: Moving Data

- **Moving data:** `movq src, 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`
    - But `%rsp` reserved for special use
    - Others have special uses for particular instructions
  - **Memory:** 8 bytes at address in register
    - Used parens like a dereference (%rax)
### movq Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Imm</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reg</td>
<td></td>
<td>movq $0x4,%rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td></td>
<td>movq $-147,(%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reg</td>
<td></td>
<td>movq %rax,%rdx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Mem</td>
<td></td>
<td>movq %rax,(%rdx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq (%rax),%rdx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>

Cannot do memory-memory transfer with a single instruction
Simple Memory Addressing

- **Normal**  \((R)\)  \(\text{Mem}[\text{Reg}[R]]\)
  - Contents of register R specifies memory *address*
  - Aha! Pointer dereferencing in C
    
    \[
    \text{movq} \ (%\text{rcx}),\%\text{rax}
    \]

- **Displacement**  \(D(R)\)  \(\text{Mem}[\text{Reg}[R]+D]\)
  - Contents of register R specifies *start of memory region*
  - Constant displacement D specifies offset
    
    \[
    \text{movq} \ 8(%\text{rbp}),\%\text{rdx}
    \]

**Note**: the normal mode is a special case of displacement mode in which \(D = 0\)
Example of Simple Addressing

```c
void swap (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```assembly
swap:
    movq (%rdi), %rax
    movq (%rsi), %rdx
    movq %rdx, (%rdi)
    movq %rax, (%rsi)
    ret
```

```
gcc -S -Og swap.c
```
Understanding swap()

void swap (long *xp, long *yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>xp</td>
</tr>
<tr>
<td>%rsi</td>
<td>yp</td>
</tr>
<tr>
<td>%rax</td>
<td>t0</td>
</tr>
<tr>
<td>%rdx</td>
<td>t1</td>
</tr>
</tbody>
</table>

Memory

Registers

swap:
    movq (%rdi), %rax       # t0 = *xp
    movq (%rsi), %rdx       # t1 = *yp
    movq %rdx, (%rdi)       # *xp = t1
    movq %rax, (%rsi)       # *yp = t0
    ret
Understanding swap() con’t

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
</tbody>
</table>

swap:

```
movq   (%rdi), %rax     # t0 = *xp
movq   (%rsi), %rdx     # t1 = *yp
movq   %rdx, (%rdi)     # *xp = t1
movq   %rax, (%rsi)     # *yp = t0
ret
```
Understanding swap() con't

Registers

| %rdi | 0x120 |
| %rsi | 0x100 |
| %rax | 123  |
| %rdx |

Memory

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>

```
swap:
    movq (%rdi), %rax       # t0 = *xp
    movq (%rsi), %rdx       # t1 = *yp
    movq %rdx, (%rdi)       # *xp = t1
    movq %rax, (%rsi)       # *yp = t0
    ret
```
Understanding swap() con’t

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
</tbody>
</table>

Assembly code:

```
swap:
  movq   (%rdi), %rax  # t0 = *xp
  movq   (%rsi), %rdx  # t1 = *yp
  movq   %rdx, (%rdi)  # *xp = t1
  movq   %rax, (%rsi)  # *yp = t0
  ret
```
Understanding swap() con’t

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>0x120</th>
<th>0x118</th>
<th>0x110</th>
<th>0x108</th>
<th>0x100</th>
</tr>
</thead>
<tbody>
<tr>
<td>456</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
swap:
movq (%rdi), %rax  # t0 = *xp
movq (%rsi), %rdx  # t1 = *yp
movq %rdx, (%rdi)  # *xp = t1
movq %rax, (%rsi)  # *yp = t0
ret
```
Understanding swap() con’t

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>456</td>
</tr>
<tr>
<td>0x118</td>
<td>0x110</td>
</tr>
<tr>
<td>0x108</td>
<td>0x100</td>
</tr>
</tbody>
</table>

```
swap:
    movq   (%rdi), %rax        # t0 = *xp
    movq   (%rsi), %rdx        # t1 = *yp
    movq   %rdx, (%rdi)        # *xp = t1
    movq   %rax, (%rsi)        # *yp = t0
    ret
```
Complete Memory Addressing

- General form

\[ \text{D(Rb, Ri, S)} \quad \text{Mem}[ \ D + \text{Reg[Rb]} + \text{Reg[Ri]} \ast S \ ] \]

- D: Constant “displacement”
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for \%rsp
- S: Scale: 1, 2, 4, or 8

- Scale becomes useful when dealing with arrays and structs, as we will see later.
### 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>
</tbody>
</table>

“Base” register

“Index” register
Address Computation Instruction

- leaq *src*, *dest*
  - *src* is an address computation expression
  - set *dest* to address denoted by expression

- use case 1
  - Computing addresses without a memory reference
    - E.g., translation of `p = &x[i];`

- Example

```c
char* a2(char* x){
    return &x[2];
}
```

```assembly
leaq 2(%rdi), %rax  # return &x[2]
ret
```
Address Computation Instruction con’t

- **leaq** *src, dest*
  - *src* is an address computation expression
  - set *dest* to address denoted by expression

- (ab)use case 2
  - Computing arithmetic expressions of the form \( x + k \times y \)
    - \( k = 1, 2, 4, \) or 8

- Example
  ```c
  long m12(long x){
    return x * 12;
  }
  leaq (%rdi,%rdi, 2), %rax # t = x + x * 2 (3x)
  salq $2, %rax             # return t << 2 (4x)
  ret
  ```
Some Arithmetic Operations - Binary

- **Two Operand Instructions:**

  - **Format** | **Computation**
    - `addq src, dest` | `dest = dest + src`
    - `subq src, dest` | `dest = dest − src`
    - `imulq src, dest` | `dest = dest * src`
    - `salq src, dest` | `dest = dest << src` (also called `shlq`)
    - `sarq src, dest` | `dest = dest >> src` (arithmetic)
    - `shrq src, dest` | `dest = dest >> src` (logical)
    - `xorg src, dest` | `dest = dest ^ src`
    - `andq src, dest` | `dest = dest & src`
    - `orq src, dest` | `dest = dest | src`

- See book for explanations

- No distinction between signed and unsigned int (except right shift)
Some Arithmetic Operations - Unary

- One Operand Instructions:
  - Format          Computation
    - incq dest     dest = dest + 1
    - decq dest     dest = dest − 1
    - negq dest     dest = −dest
    - notq dest     dest = ~dest

- See book for explanations