Why do I care for machine code?

- Chances are, you’ll never write programs in machine code
  - Compilers are much better & more patient than you are
- But: understanding machine code is useful and important
  - Implementing system software
    - Compiler has machine code as target
    - Operating systems do weird things (save/restore process state)
  - Access special hardware features
    - Processor model-specific registers
  - Tuning program performance
    - Understand optimizations done / not done by the compiler
    - Understanding sources of program inefficiency
  - Creating / fighting malware
    - Machine/assembly code is the language of choice!

Not A Computer Architecture Course

- This course only teaches the interaction with machine hardware, not designing/implementing the hardware
- If you are really into the processor architecture
  - Read chapter 4
  - A guest lecture by Prof. Dwarkadas on Feb 19
  - Take the Computer Architecture course (ECE 401)

Intel/AMD x86 Processors

- Totally dominate laptop/desktop/server market
  - Does do as well in smartphones!
- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
  - Compared to Reduced Instruction Set Computers (RISC), an instruction can do more but require more complex hardware implementation (potentially slow, more chip area and power consumption)
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>First 16-bit 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>First 32 bit processor, referred to as IA32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Capable of running Unix</td>
<td></td>
</tr>
<tr>
<td>Pentium 4F</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td></td>
<td></td>
<td>First 64-bit processor, referred to as x86-64</td>
<td></td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>2667-3333</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Multicore</td>
<td></td>
</tr>
<tr>
<td>Haswell</td>
<td>today</td>
<td>1.4B</td>
<td>Not faster</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Improve on core count, power reduction, not frequency increase</td>
<td></td>
</tr>
</tbody>
</table>

Evolution into 64-Bit

- 64-bit: support larger programs, more register space, ...
- Intel attempted radical shift from IA32 to IA64
  - Itanium: executes IA32 code only as legacy
  - Performance disappointing
- AMD stepped in with evolutionary solution
  - x86-64
- 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

Machine Programming I: Basics

- Overview of x86 processors and architectures
- Relation of C, assembly, machine codes
- Assembly Basics: Registers, operands, move
- x86-64

Definitions

- **Architecture**: (also instruction set architecture: ISA) The parts of a processor design that one needs to understand to write assembly code.
  - Including instruction set specification, registers.
  - Example ISA: IA32, x86-64
- **Microarchitecture**: Implementation of the architecture.
  - Examples: cache sizes and core frequency.
Machine Programmer’s View

- **CPU**
  - **Registers**
  - **Condition Codes**
- **Memory**
  - **Object Code**
  - **Program Data**
  - **O’S Data**
  - **Stack**

**Programmer-visible state**
- **PC**: Program counter
  - Address of next instruction
  - Called “EIP” (IA32) or “RIP” (x86-64)
- **Register file**
  - Heavily used program data
  - **Condition codes**
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

Turning C into Machine Code

- Code in files **p1.c p2.c**
- Compile with command: **gcc -O1 p1.c p2.c -o p**
- Put resulting binary in file **p**

---

Compiling Into Assembly

**C Code**
```c
int sum(int x, int y)
{
    int t = x*y;
    return t;
}
```

**Generated IA32 Assembly**
```
sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    popl %ebp
    ret
```

---

Assembly Characteristics: Data Types

- “Integer” data of 1, 2, or 4 bytes
  - Data values
  - Addresses (pointers)
- Floating point data of 4, 8, or 10 bytes
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
Assembly Characteristics: Operations

- Perform arithmetic function 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

Code for `sum`

```
0x401040 <sum>:
0x550x890xe50x8b0x45
```

Object/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

Disassembling Object Code

```
Disassembled:
080483c4 <sum>:
0483c4:  55  push  %ebp
0483c5:  89 e5  mov  %esp, %ebp
0483c7:  8b 45 0c  mov  0xc(%ebp), %eax
0483ca:  03 45 08  add    0x8(%ebp), %eax
0483ce:  c3  ret
```

Machine Instruction Example

```c
int t = x+y;
addl 8(%ebp),%eax
```

C code
- Add two signed integers

Assembly
- Add 2 4-byte integers
  - "Long" words in GCC parlance
  - Same instruction whether signed or unsigned
- Operands:
  - x: Register %eax
  - y: Memory M[ebp+8]
  - t: Register %eax
- Return function value in %eax

Object code
- 3-byte instruction
- Stored at address 0x80483ca

Disassembler
```
objdump -d p
```
- 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
Machine Programming I: Basics

- Overview of x86 processors and architectures
- Relation of C, assembly, machine codes
- Assembly Basics: Registers, operands, move
- x86-64

Integer Registers (IA32)

- Origin (mostly obsolete)
  - Accumulate
  - Counter
  - Data
  - Base
  - Source
  - Index
  - Destination
  - Index
  - Stack
  - Pointer
  - Base
  - Pointer

Moving Data: IA32

- Moving Data
  - `movl 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 8 integer registers
  - Example: %eax, %edx
  - But %esp and %ebp reserved for special use
  - Others may have special uses for particular instructions

- Memory: 4 consecutive bytes of memory at address given by register
  - Simplest example: (%eax)
  - Various other “address modes”

movl Operand Combinations

- Source Dest Src, Dest C Analog
  - Imm
    - Reg movl $0x4,%eax temp = 0x4;
    - Mem movl $-147,(%eax) *p = -147;
  - Reg
    - Reg movl %eax,%edx temp2 = temp1;
    - Mem movl %eax,(%edx) *p = temp;
  - Mem
    - Reg movl (%eax),%edx temp = *p;

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

- **Normal (R) Memory [Reg[R]]**
  - Register R specifies memory address
  - \texttt{movl (%ecx), %eax}

- **Displacement D(R) Memory [Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
  - \texttt{movl 8(%ebp), %edx}

Memory Addressing Example

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

Full Memory Addressing Modes

- **Most general form**
  - \texttt{D(Rb,Ri,S) Memory [Reg[Rb]+S*Reg[Ri]+D]}  
  - D: Constant “displacement”
  - Rb: Base register: Any of 8 integer registers
  - Ri: Index register: Any, except %esp (unlikely you’d use %ebp either)
  - S: Scale: 1, 2, 4, or 8 (why these numbers?)

Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%edx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(%edx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1d080</td>
</tr>
</tbody>
</table>
Address Computation Instruction

- **leal Src, Dest**
  - Src is address mode expression
  - Set Dest to address denoted by expression

- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of \( p = \&x[i] \)
  - Computing arithmetic expressions of the form \( x + k \cdot y \)
    - \( k = 1, 2, 4, \text{ or } 8 \)

- **Example**

```c
int mul2(int x) {
    return x*12;
}
```

Converted to ASM by compiler:
```
leal (%eax,%eax,2), %eax  ;t <- x+x*2
sall $2, %eax             ;return t<<2
```

Understanding Swap

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

Example:
```
int key = 0x12345678;
```

Converted to ASM by compiler:
```
movl 8(%ebp), %edx # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx # ebx = \&xp (t0)
movl (%ecx), %eax # eax = \&yp (t1)
movl %eax, (%edx) # *xp = t1
movl %ebx, (%ecx) # *yp = t0
```

Machine Programming I: Basics

- Overview of x86 processors and architectures
- Relation of C, assembly, machine codes
- Assembly Basics: Registers, operands, move
- x86-64

Data Representations: IA32 + x86-64

- **Sizes of C Objects (in Bytes)**
  - **C Data Type** | **Generic 32-bit** | **Intel IA32** | **x86-64**
  - unsigned        | 4               | 4              | 4
  - int             | 4               | 4              | 4
  - long int        | 4               | 4              | 8
  - char            | 1               | 1              | 1
  - short           | 2               | 2              | 2
  - float           | 4               | 4              | 4
  - double          | 8               | 8              | 8
  - long double     | 8               | 10/12          | 10/16
  - char*           | 4               | 4              | 8
  - Or any other pointer
### x86-64 Integer Registers

- `%rax` ← `eax`
- `%rbx` ← `ebx`
- `%rcx` ← `ecx`
- `%rdx` ← `edx`
- `%r8` ← `r8d`
- `%r9` ← `r9d`
- `%r10` ← `r10d`
- `%r11` ← `r11d`
- `%r12` ← `r12d`
- `%r13` ← `r13d`
- `%r14` ← `r14d`
- `%r15` ← `r15d`

- `%rsp` ← `esp`
- `%rbp` ← `ebp`

- **Extend existing registers. Add 8 new ones.**
- **Make `%ebp`/`%rbp` general purpose**

### New 64-bit Instructions

- Long word 1 (4 Bytes) ↔ Quad word q (8 Bytes)
- New instructions:
  - `movl` ↔ `movq`
  - `addl` ↔ `addq`
  - `sall` ↔ `salq`
  - etc.

### 32-bit code for swap

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

#### Body

- `movl %esp, %ebp`
- `movl %esp, %ebx`
- `movl %ebp, %edx`
- `movl %edx, %ebx`
- `movl %ecx, %eax`
- `movl %eax, (%edx)`
- `movl %ebx, (%ecx)`
- `popl %ebx`
- `popl %ebp`
- `ret`

#### Finish

### 64-bit code for swap

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

#### Body

- `movl (%rdi), %edx`
- `movl (%rsi), %eax`
- `movl %eax, (%rdi)`
- `movl %edx, (%rsi)`
- `ret`

#### Finish

- Operands passed in registers
  - First (`xp`) in `%rdi`, second (`yp`) in `%rsi`
- No stack operations required
- 64-bit pointers, 32-bit data
64-bit code for long int swap

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

- 64-bit data
- Data held in registers `%rax` and `%rdx`
- `movq` operation

Machine Programming I: Summary

- Overview of x86 processors and architectures
  - Evolutionary design, backward compatibility
- Relation of C, assembly, machine codes
  - High-level language, machine-level code, human friendliness
- Assembly basics: registers, operands, move
  - Key is data location and semantics
- x86-64
  - More than 64-bits, more register space allows additional optimization

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of “Computer Systems: A programmer’s Perspective” by Bryant and O’Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.