Machine-Level Representation: Basic

CSCI 2021: Machine Architecture and Organization
Pen-Chung Yew
Department of Computer Science and Engineering
University of Minnesota
http://www.cs.umn.edu/~yew

With Slides from Randy Bryant, David O'Hallaron, and Antonia Zhai

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 Linux 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>
</tbody>
</table>
| First 16-bit Intel processor, Basis for IBM PC & DOS
  1MB address space
| 386          | 1985 | 275K        | 16-33|
| First 32 bit Intel processor, referred to as IA32
  Added “flat addressing”, capable of running Unix
| Pentium 4E   | 2004 | 125M        | 2800-3800|
| First 64-bit Intel x86 processor, referred to as x86-64
| Core 2       | 2006 | 291M        | 1060-3500|
| First multi-core Intel processor
| Core i7       | 2008 | 731M        | 1700-3900|
| Four cores   |      |             |      |

Our Coverage

- IA32
  - The traditional 32-bit x86
- x86-64
  - The standard
  - gcc hello.c
  - gcc -m64 hello.c
- Presentation
  - Book covers x86-64
  - Web aside on IA32
  - We will only cover x86-64

Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
Definitions

- **Architecture**: (Instruction set architecture - ISA)
  - 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.
  - Examples: cache sizes and core frequency.

- **Code Forms**:
  - **Machine Code**: The byte-level programs (binaries, or object code) that a processor executes.
  - **Assembly Code**: A text representation of machine code.

- **Example ISAs**:
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Used in almost all mobile phones
  - ARM: Used in almost all mobile phones
  - Intel: x86, IA32, Itanium, x86-64

Assembly/Machine Code View

- **Programmer-Visible State**
  - PC: Program counter
  - Register file
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic or logical operation
    - Used for conditional branching

- **CPU (Central Processing Unit)**
  - Addresses
  - Data
  - Code
  - Memory
    - Byte addressable array
    - Code and user data
    - Stack to support procedures

- **Registers**
  - Data	
    - Code Data Stack
  - Instruction Codes
  - PC

- **Condition Codes**
  - Used for conditional branching

Turning C into Object Code

- **Code in files**: `p1.c p2.c`
- **Compile with command**: `gcc -Og p1.c p2.c -o p` [New to recent versions of GCC]
- **Put resulting binary in file p**

<table>
<thead>
<tr>
<th>Text</th>
<th>Compiler (gcc -Og -S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C program <code>p1.c p2.c</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text</th>
<th>Assembler (gcc or aas)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asm program <code>p1.s p2.s</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary</th>
<th>Static libraries <code>.a</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Object program <code>p1.o p2.o</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary</th>
<th>Linker (gcc or 1d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executable program <code>p</code></td>
<td></td>
</tr>
</tbody>
</table>

Compiling Into Assembly

- **C Code** (`sum.c`)
- **Generated x86-64 Assembly**

  ```c
  long plus(long x, long y);
  void sumstore(long x, long y, long *dest);
  {
    long t = plus(x, y);
    *dest = t;
  }
  ```

  ```assembly
  sumstore:
  pushq %rbx
  movq %rdx, %rbx
  call plus
  movq %rax, (%rbx)
  popq %rbx
  ret
  ```

<table>
<thead>
<tr>
<th>Text</th>
<th>Compiler (gcc -Og -S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc -Og -S <code>sum.c</code></td>
<td></td>
</tr>
</tbody>
</table>

- **Produces file**: `sum.s`
- **Warning**: Will get very different results on different machines (Linux, Mac OS-X, ...) due to different versions of gcc and different compiler settings.

Assembly Characteristics: Data Types

- “Integer” data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- 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 register or memory data
- Transfer data between memory and register (mov instructions)
- Load data from memory into register
- Store register data into memory
- Transfer control
  - Unconditional jumps to/from procedures (call instruction)
  - Conditional branches (jmp instructions)
Review: Compiling 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` `-o p`.

<table>
<thead>
<tr>
<th>Text</th>
<th>Compiler (gcc -Og -S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asm program</td>
<td>Assembler (gcc or as)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Object Code

```c
Code for sumstore
0x0400595:
0x0a33: push rbx
0x0a48: movq %rbx, %rbx
0x0a89: callq long printf
0x0a03: retq
0x0a5b: .isum: long plus(long x, long y)
```

- Assembler
  - Translates .s into .o
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files

- Linker (to be covered in Chpt 7)
  - 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

Machine Instruction Example

```c
#dest = t;

movq %rax, (%rbx)
```

- C Code
  - Store value `t` to the memory location designated by `dest`
- Assembly
  - Move 8-byte value to memory
  - Quad words in x86-64 parlance
- Operands:
  - `t`: Register %rax
  - `dest`: Register %rbx
  - `%dest`: Memory [%rbx]

```
0x040059a: 48 89 03
```

- Object Code
  - 3-byte instruction
  - Stored at address `0x040059a`

Disassembling Object Code

```
Disassembled (from Object Code to Assembly)
00000000000400595 <sumstore>:
400595: 53
400596: 48 89 d3
400597: 48 89 f2
400598: 48 89 ff
400599: 48 89 00
40059a: 48 89 03
40059b: 48 89 0f
40059c: 48 89 89
40059d: 48 89 7f
40059e: 48 89 7f
4005f0: 48 89 00
4005f1: 48 89 03
4005f2: 48 89 0f
4005f3: 48 89 89
4005f4: 48 89 7f
4005f5: 48 89 7f
4005f6: 48 89 00
4005f7: 48 89 03
4005f8: 48 89 0f
4005f9: 48 89 89
4005fa: 48 89 7f
4005fb: 48 89 7f
4005fc: 48 89 00
4005fd: 48 89 03
4005fe: 48 89 0f
4005ff: 48 89 89
```

- 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 `.out` (complete executable) or `.o` file

Review: Assembly/Machine Code View

Programmer-Visible (Accessible) Machine State

- PC: Program counter
  - Address of next instruction
  - Called `next` in x86-64
- Register file
  - Heavy used program data
- Condition codes
  - Store status information after execution of current arithmetic or logical instr.
  - Used for conditional branching

Memory

- Code
  - Byte addressable array
- Data
  - Code and user data
- Stack
  - To support procedures

ASM/Compiler View

- Text
  - C program `p1.c` `p2.c`
- Binary
  - Object program `p1.o` `p2.o`
- Executable program `p`

System View

- CPU (Central Processing Unit)
  - Registers
  - Data
  - Instruction
  - Condition Codes

- Memory
  - Addresses
  - Code
  - Data
  - Stack

Machine Instruction Example

```
*dest = t;

movq %rax, (%rbx)
```

- C Code
  - Store value `t` to the memory location designated by `dest`
- Assembly
  - Move 8-byte value to memory
  - Quad words in x86-64 parlance
- Operands:
  - `t`: Register %rax
  - `dest`: Register %rbx
  - `%dest`: Memory [%rbx]

```
0x040059a: 48 89 03
```

- Object Code
  - 3-byte instruction
  - Stored at address `0x040059a`

- 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 `.out` (complete executable) or `.o` file
Disassembled 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
```

• In gdb Debugger
gdb sum
  disassemble sumstore
• Disassemble procedure
  x/14xb sumstore
• Examine the 14 bytes starting at sumstore

Object 0x0400595:
```
0x53 0x48 0x89 0xd3 0xe8 0xf2 0xff 0xff 0xff 0x48 0x89 0x03 0x5b 0xc3
```

**Alternate Disassembly**

**Object**
- 0x0400595:
  - 0x53
  - 0x48
  - 0x89
  - 0xd3
  - 0xe8
  - 0xf2
  - 0xff
  - 0xff
  - 0xff
  - 0x48
  - 0x89
  - 0x03
  - 0x5b
  - 0xc3

**Disassembled**
- 0x0000000000400595 <+0>: push %rbx
- 0x0000000000400596 <+1>: mov %rdx,%rbx
- 0x0000000000400599 <+4>: callq 0x400590 <plus>
- 0x000000000040059e <+9>: mov %rax,(%rbx)
- 0x00000000004005a1 <+12>: pop %rbx

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

**Machine Programming I: Basics**

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

**CPU (Central Processing Unit)**
```
<table>
<thead>
<tr>
<th>PC</th>
<th>Registers</th>
<th>Data</th>
<th>Code</th>
<th>Data</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Addresses**
```
|    |    |      |      |      |       |
|    |    |      |      |      |       |
```

**Memory**
```
|    |    |      |      |      |       |
|    |    |      |      |      |       |
```

**x86-64 Integer Registers**
```
%rax %rbx %rcx %rdx %rsi %rdi %rsp %rbp
%rbx %rcx %rdx %rsi %rdi %r8 %r9 %r10
%rdi %rsi %r8 %r9 %r10 %r11 %r12
%r13 %r14 %r15 %esp %ebp %eip
```

• Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

**Some History: IA32 Registers**
```
%eax %ecx %edx %ebx %esi %edi %esp %ebp
%eax %ecx %edx %ebx %esi %edi %esp %ebp
%eax %ecx %edx %ebx %esi %edi %esp %ebp
%eax %ecx %edx %ebx %esi %edi %esp %ebp
%eax %ecx %edx %ebx %esi %edi %esp %ebp
%eax %ecx %edx %ebx %esi %edi %esp %ebp
```

**Operand Types**
- **Immediate**: Constant integer data
  - Example: $0x400, $0x533
  - Like C constant, but prefixed with '$'
  - Encoded with 1, 2, or 4 bytes
- **Register**: One of 16 integer registers
  - Example: %rax, %r13
  - But %esp reserved for special use
  - Others have special uses for particular instructions
- **Memory**: 8 consecutive bytes of memory at address given by register
  - Simplest example: (%rax)
  - Various other “address modes”

**Moving Data**
```
movq  %rax, %rdx
movq  %rdx, %r8
movq  %r8, %rdx
movq  %rdx, %rsp
movq  %rsp, %rdx
```

**movq Operand Combinations**
```
Source | Dest  | Src,Dest  | C Analog
---|---|---|---
Reg | Mem | movq $0x4,%rax | temp = 0x4;
Reg | Mem | movq $-147,(%rax) | *p = -147;
Mem | Reg | movq %rax,%rdx | temp2 = temp1;
Mem | Reg | movq %rax,%rdx | *p = temp;
```

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

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

Understanding Swap()

```assembly
swap:
    movq (%rdi), %rax  # t0 = *xp
    movq (%rsi), %rdx  # t1 = *yp
    movq %rdx, (%rdi)  # *xp = t1
    movq %rax, (%rsi)  # *yp = t0
    ret
```
Understanding Swap()

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi: 0x120</td>
<td>456</td>
</tr>
<tr>
<td>%rsi: 0x100</td>
<td>123</td>
</tr>
<tr>
<td>%rax: 123</td>
<td>0x120</td>
</tr>
<tr>
<td>%rdx: 456</td>
<td>0x118</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

Announcement 2/10/2016

- Homework Assignment #1 due before lecture today
- Bomb Lab issued today, due Friday 2/26/2016
  - Recitation session Thursday will cover Bomb Lab
  - Will cover GDB – debugger needed in Bomb Lab

Review: 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>Reg</td>
<td>Mem</td>
<td>Mem</td>
<td>D(R)</td>
</tr>
<tr>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
<td>D(R)</td>
</tr>
<tr>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
<td>D(R)</td>
</tr>
<tr>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
<td>D(R)</td>
</tr>
</tbody>
</table>

Cannot do Memory-Memory transfer with a single instruction

Simple Memory Addressing Modes

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

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

Complete Memory Addressing Modes

- Most general form for an operand
  - \( D(Rb,Ri,S) \) Mem[Reg[Rb]+S*Reg[Ri]+D]
  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for %rsp
  - S: Scale: 1, 2, 4, or 8 (for different data types)
- Special cases
  - \( (Rb,Ri) \) Mem[Reg[Rb]+Reg[Ri]]
  - \( D(Rb,Ri) \) Mem[Reg[Rb]+Reg[Ri]+D]
  - \( (Rb,Ri,S) \) Mem[Reg[Rb]+S*Reg[Ri]]

Review: Word-Oriented Memory Organization

<table>
<thead>
<tr>
<th>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0001</td>
<td>0002</td>
<td>0003</td>
</tr>
<tr>
<td>0004</td>
<td>0005</td>
<td>0006</td>
<td>0007</td>
</tr>
<tr>
<td>0008</td>
<td>0009</td>
<td>0010</td>
<td>0011</td>
</tr>
<tr>
<td>0012</td>
<td>0013</td>
<td>0014</td>
<td>0015</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>0x0000 + 0x8</td>
<td>0x0008</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0x0000 + 0x100</td>
<td>0x1000</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>0x0000 + 4*0x100</td>
<td>0x4000</td>
</tr>
<tr>
<td>0x80(%rdx,2)</td>
<td>2*0x0000 + 0x80</td>
<td>0x1080</td>
</tr>
</tbody>
</table>

Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

Some Arithmetic Operations

- Two Operand Instructions:
  - `leaq Src,Dst` # load effective address (lea)
  - `Src` is address mode expression
  - `Set Dst to address denoted by expression`
- Uses
  - Computing addresses without a memory reference
    - E.g., translation of \( p = a[1] \)
  - Computing arithmetic expressions of the form \( x + k\cdot y \)
    - \( k = 1, 2, 4, \text{ or } 8 \)
- Example
  - Converted to ASM by compiler:
    ```
    long m12(long x)
    {
      return x*12;
    }
    leaq (%rdi,%rdi,2), %rax
    # t <- x + x*2
    salq $2, %rax
    # return t<<2
    ```

Review: Two’s Complement Addition

- `TAdd(u,v)`
- `UAdd(u,v)`

Some Arithmetic Operations

- One Operand Instructions:
  - `incq Dest` Dest = Dest + 1
  - `decq Dest` Dest = Dest - 1
  - `negq Dest` Dest = -Dest
  - `notq Dest` Dest = ~Dest

- Watch out for argument order!
- No distinction between signed and unsigned `int` (why?)
**Arithmetic Expression Example**

```
long arith
(long x, long y, long z) {
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

**Instructions Used**
- `lea`: address computation
- `add`: addition
- `sal`: shift
- `imul`: multiplication
  - But, only used once

**Understanding Arith Expression Example**

```
long arith
(long x, long y, long z) {
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

**Register Use(s)**
- `%rdi`: Argument x
- `%rsi`: Argument y
- `%rdx`: Argument z
- `%rax`: `t1`, `t2`, `rval`
- `%rax`: t4
- `%rcx`: t5

**Where is t3 calculated?**

**Machine Programming I: 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
- Arithmetic
  - C compiler will figure out different instruction combinations to carry out computation

**Move on to Program Control**