Definitions

- **Architecture**: (also instruction set architecture or ISA)
  The parts of a processor design that one needs to understand to write assembly code
  - “What is directly visible to software”
- **Microarchitecture**: Implementation of the architecture
  - CSE 352

- Is cache size “architecture”?
- How about CPU frequency?
- And number of registers?
Assembly Programmer’s View

- **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

- **Memory**
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures (we’ll come back to that)
Turning C into Object Code

- Code in files \( p1.c \) \( p2.c \)
- Compile with command: \( \text{gcc} \ -O1 \ p1.c \ p2.c \ -o \ p \)
  - Use basic optimizations (\(-O1\))
  - Put resulting machine code in file \( p \)

```
<table>
<thead>
<tr>
<th>text</th>
<th>C program (p1.c p2.c)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compiler (gcc -S)</td>
</tr>
<tr>
<td>text</td>
<td>Asm program (p1.s p2.s)</td>
</tr>
<tr>
<td></td>
<td>Assembler (gcc or as)</td>
</tr>
<tr>
<td>binary</td>
<td>Object program (p1.o p2.o)</td>
</tr>
<tr>
<td></td>
<td>Linker (gcc or ld)</td>
</tr>
<tr>
<td>binary</td>
<td>Executable program (p)</td>
</tr>
</tbody>
</table>
```

Static libraries (.a)
Compiling Into Assembly

C Code

```c
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

Generated IA32 Assembly

```assembly
sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Obtain with command

```
gcc -O1 -S code.c
```

Produces file `code.s`
Machine Instruction Example

C Code: add two signed integers

Add two 4-byte integers
- “Long” words in GCC speak
- 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 0x401046
## Object Code

### Code for `sum`

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td><code>&lt;sum&gt;</code>:</td>
</tr>
<tr>
<td>0x55</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0xe5</td>
<td></td>
</tr>
<tr>
<td>0x8b</td>
<td></td>
</tr>
<tr>
<td>0x45</td>
<td></td>
</tr>
<tr>
<td>0xc</td>
<td></td>
</tr>
<tr>
<td>0x0c</td>
<td></td>
</tr>
<tr>
<td>0x03</td>
<td></td>
</tr>
<tr>
<td>0x45</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0xec</td>
<td></td>
</tr>
<tr>
<td>0x5d</td>
<td></td>
</tr>
<tr>
<td>0xc3</td>
<td></td>
</tr>
</tbody>
</table>

- Total of 13 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address 0x401040
- Not at all obvious where each instruction starts and ends

### Assembler

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

### Linker

- Resolves references between object files and (re)locates their data
- 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

<table>
<thead>
<tr>
<th>Address</th>
<th>OPCODE</th>
<th>Mnemonic</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0041040</td>
<td>55</td>
<td>push</td>
<td>%ebp</td>
</tr>
<tr>
<td>1:</td>
<td>e5</td>
<td>mov</td>
<td>%esp,%ebp</td>
</tr>
<tr>
<td>3:</td>
<td>45 0c</td>
<td>mov</td>
<td>0xc(ebp),%eax</td>
</tr>
<tr>
<td>6:</td>
<td>45 08</td>
<td>add</td>
<td>0x8(ebp),%eax</td>
</tr>
<tr>
<td>9:</td>
<td>ec</td>
<td>mov</td>
<td>%ebp,%esp</td>
</tr>
<tr>
<td>b:</td>
<td>5d</td>
<td>pop</td>
<td>%ebp</td>
</tr>
<tr>
<td>c:</td>
<td>c3</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

Disassembler

`objdump -d p`

- Useful tool for examining object code (`man 1 objdump`)
- Analyzes bit pattern of series of instructions (delineates instructions)
- Produces near-exact rendition of assembly code
- Can be run on either p (complete executable) or p1.o / p2.o file
## Alternate Disassembly

### Object

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040:</td>
<td>0x55</td>
</tr>
<tr>
<td></td>
<td>0x89</td>
</tr>
<tr>
<td></td>
<td>0xe5</td>
</tr>
<tr>
<td></td>
<td>0x8b</td>
</tr>
<tr>
<td></td>
<td>0x45</td>
</tr>
<tr>
<td></td>
<td>0x0c</td>
</tr>
<tr>
<td></td>
<td>0x03</td>
</tr>
<tr>
<td></td>
<td>0x45</td>
</tr>
<tr>
<td></td>
<td>0x08</td>
</tr>
<tr>
<td></td>
<td>0x89</td>
</tr>
<tr>
<td></td>
<td>0xec</td>
</tr>
<tr>
<td></td>
<td>0x5d</td>
</tr>
<tr>
<td></td>
<td>0xc3</td>
</tr>
</tbody>
</table>

### Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td>push %ebp</td>
<td></td>
</tr>
<tr>
<td>0x401041</td>
<td>mov %esp,%ebp</td>
<td>%esp,%ebp</td>
</tr>
<tr>
<td>0x401043</td>
<td>mov 0xc(%ebp),%eax</td>
<td>%eax</td>
</tr>
<tr>
<td>0x401046</td>
<td>add 0x8(%ebp),%eax</td>
<td>%eax, %eax</td>
</tr>
<tr>
<td>0x401049</td>
<td>mov %ebp,%esp</td>
<td></td>
</tr>
<tr>
<td>0x40104b</td>
<td>pop %ebp</td>
<td></td>
</tr>
<tr>
<td>0x40104c</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

### Within gdb debugger

```bash
    gdb p
disassemble sum
    (disassemble function)
x/13b sum
    (examine the 13 bytes starting at sum)
```
What Can be Disassembled?

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

```
% 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
```
Roadmap

C:
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();

Assembly language:
get_mpg:
pushq %rbp
movq %rsp, %rbp
...
popq %rbp
ret

Machine code:
0111010000011000
100011010000010000000010
1000100111000010
110000011111110100001111

Computer system:

Data & addressing
Integers & floats
Machine code & C
x86 assembly
programming
Procedures &
stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Next x86 topics

- Move instructions, registers, and operands
- Memory addressing modes
- swap example: 32-bit vs. 64-bit
- Arithmetic operations
What Is A Register (again)?

- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)

- Registers have names, not addresses.

- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but especially x86
Integer Registers (IA32)

%eax
%ecx
%edx
%ebx
%esi
%edi
%esp
%ebp

32-bits wide

general purpose

Origin
(mostly obsolete)

accumulate
counter
data;base
source
index
destination
index
stack
pointer
base
pointer
## Integer Registers (IA32)

<table>
<thead>
<tr>
<th>General Purpose</th>
<th>Origin (mostly obsolete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax</td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx</td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
</tr>
</tbody>
</table>

16-bit virtual registers (backwards compatibility)
## 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>

- Extend existing registers, and add 8 new ones; all accessible as 8, 16, 32, 64 bits.
Assembly Data Types

- “Integer” data of 1, 2, 4 (IA32), or 8 (just in x86-64) bytes
  - Data values
  - Addresses (untyped pointers)

- Floating point data of 4, 8, or 10 bytes

- What about “aggregate” types such as arrays?
  - Just contiguous memory locations
Three Basic Kinds of Instructions

- **Transfer data between memory and register**
  - *Load* data from memory into register
    - %reg = Mem[address]
  - *Store* register data into memory
    - Mem[address] = %reg

- **Perform arithmetic function on register or memory data**
  - c = a + b;  
  - z = x << y;  
  - i = h & g;

- **Transfer control: what instruction to execute next**
  - Unconditional jumps to/from procedures
  - Conditional branches

Remember: memory is indexed just like an array[] of bytes!
Moving Data: IA32

- **Moving Data**
  - `movx Source, Dest`
  - `x` is one of `{b, w, l}`
  - `movl Source, Dest:`
    - Move 4-byte “long word”
  - `movw Source, Dest:`
    - Move 2-byte “word”
  - `movb Source, Dest:`
    - Move 1-byte “byte”

- Lots of these in typical code

Confusing historical terms... not the current machine word size
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 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

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>movl</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Imm</strong></td>
<td>Reg</td>
<td>movl $0x4,%eax</td>
<td>var_a = 0x4;</td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td>Reg</td>
<td>movl $-147,(%eax)</td>
<td>*p_a = -147;</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td>Mem</td>
<td>movl %eax,(%edx)</td>
<td>var_d = var_a;</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td>Mem</td>
<td>movl %eax,(%edx)</td>
<td>*p_d = var_a;</td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td>Reg</td>
<td>movl (%eax),%edx</td>
<td>var_d = *p_a;</td>
</tr>
</tbody>
</table>

*Cannot do memory-memory transfer with a single instruction.*

*How would you do it?*
Memory vs. registers

- What is the main difference?
- Addresses vs. Names
- Big vs. Small
Memory Addressing Modes: Basic

- **Indirect** (R) \(\text{Mem}[\text{Reg}[R]]\)
  - Register R specifies the memory address

  \texttt{movl \%ecx,\%eax}

- **Displacement** \(D(R)\) \(\text{Mem}[\text{Reg}[R]+D]\)
  - Register R specifies a memory address
    - (e.g. the start of some memory region)
  - Constant displacement D specifies the offset from that address

  \texttt{movl 8(\%ebp),\%edx}
Using Basic Addressing Modes

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

swap:

pushl %ebp
movl %esp,%ebp
pushl %ebx

movl 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax,(%edx)
movl %ebx,(%ecx)

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret

Set Up

Body

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

movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax  # eax = *yp (t1)
movl (%edx),%ebx  # ebx = *xp (t0)
movl %eax,(%edx)  # *xp = eax
movl %ebx,(%ecx)  # *yp = ebx
Understanding Swap

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td></td>
</tr>
<tr>
<td>%ecx</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>0x104</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

```
movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax  # eax = *yp (t1)
movl (%edx),%ebx  # ebx = *xp (t0)
movl %eax,(%edx)  # *xp = eax
movl %ebx,(%ecx)  # *yp = ebx
```
Understanding Swap

%eax
%edx
%ecx 0x120
%ebx
%esi
%edi
%esp
%ebp 0x104

movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax  # eax = *yp (t1)
movl (%edx),%ebx  # ebx = *xp (t0)
movl %eax,(%edx)  # *xp = eax
movl %ebx,(%ecx)  # *yp = ebx
Understanding Swap

<table>
<thead>
<tr>
<th></th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>0x124</td>
</tr>
<tr>
<td>456</td>
<td>0x120</td>
</tr>
<tr>
<td>0x11c</td>
<td>0x118</td>
</tr>
<tr>
<td>0x114</td>
<td>0x110</td>
</tr>
<tr>
<td>0x124</td>
<td>0x120</td>
</tr>
<tr>
<td>0x10c</td>
<td>0x108</td>
</tr>
<tr>
<td>0x104</td>
<td>0x104</td>
</tr>
<tr>
<td>0x100</td>
<td>0x100</td>
</tr>
</tbody>
</table>

Variables:
- `%eax`
- `%edx` 0x124
- `%ecx` 0x120
- `%ebx`
- `%esi`
- `%edi`
- `%esp` 0x104
- `%ebp` 0x104

### Offset

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>-4</td>
<td></td>
</tr>
</tbody>
</table>

### Assembly Code

1. `movl 12(%ebp),%ecx`  
   # ecx = yp
2. `movl 8(%ebp),%edx`  
   # edx = xp
3. `movl (%ecx),%eax`  
   # eax = *yp (t1)
4. `movl (%edx),%ebx`  
   # ebx = *xp (t0)
5. `movl %eax,(%edx)`  
   # *xp = eax
6. `movl %ebx,(%ecx)`  
   # *yp = ebx
### Understanding Swap

#### Variables

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

#### Offset Table

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x110</td>
</tr>
<tr>
<td>8</td>
<td>0x10c</td>
</tr>
<tr>
<td>4</td>
<td>0x108</td>
</tr>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

#### Instructions

1. `movl 12(%%ebp),%ecx`  
   
   # ecx = yp

2. `movl 8(%%ebp),%edx`  
   
   # edx = xp

3. `movl (%ecx),%eax`  
   
   # eax = *yp (t1)

4. `movl (%edx),%ebx`  
   
   # ebx = *xp (t0)

5. `movl %eax,(%edx)`  
   
   # *xp = eax

6. `movl %ebx,(%ecx)`  
   
   # *yp = ebx
Understanding Swap

| `%eax` | 456 |
| `%edx` | 0x124 |
| `%ecx` | 0x120 |
| `%ebx` | 123 |
| `%esi` | |
| `%edi` | |
| `%esp` | |
| `%ebp` | 0x104 |

```
movl 12(%ebp),%ecx       # ecx = yp
movl 8(%ebp),%edx        # edx = xp
movl (%ecx),%eax         # eax = *yp (t1)
movl (%edx),%ebx         # ebx = *xp (t0)
movl %eax,(%edx)         # *xp = eax
movl %ebx,(%ecx)         # *yp = ebx
```
Understanding Swap

| %eax  | 456 |
| %edx  | 0x124 |
| %ecx  | 0x120 |
| %ebx  | 123 |
| %esi  | |
| %edi  | |
| %esp  | 0x104 |
| %ebp  | 0x104 |

```
movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax   # eax = *yp (t1)
movl (%edx),%ebx   # ebx = *xp (t0)
movl %eax,(%edx)   # *xp = eax
movl %ebx,(%ecx)   # *yp = ebx
```
## Understanding Swap

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>YP</th>
<th>XP</th>
<th>Rtn adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
<td>0x120</td>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
<td>0x124</td>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x11c</td>
<td>4</td>
<td></td>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x118</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- `movl 12(%ebp),%ecx` # ecx = yp
- `movl 8(%ebp),%edx` # edx = xp
- `movl (%ecx),%eax` # eax = *yp (t1)
- `movl (%edx),%ebx` # ebx = *xp (t0)
- `movl %eax,(%edx)` # *xp = eax
- `movl %ebx,(%ecx)` # *yp = ebx
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>64-bits wide</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<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>

- Extend existing registers, and add 8 new ones; all accessible as 8, 16, 32, 64 bits.
32-bit vs. 64-bit operands

- **Long word** $l$ (4 Bytes) $\leftrightarrow$ **Quad word** $q$ (8 Bytes)

- **New instruction forms:**
  - `movl` $\rightarrow$ `movq`
  - `addl` $\rightarrow$ `addq`
  - `sall` $\rightarrow$ `salq`
  - etc.

- x86-64 can still use 32-bit instructions that generate 32-bit results
  - Higher-order bits of destination register are just set to 0
  - Example: `addl`
Swap Ints in 32-bit Mode

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

### Swap

```
swap:
pushl %ebp
movl %esp,%ebp
pushl %ebx
movl 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax,(%edx)
movl %ebx,(%ecx)
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
Swap Ints in 64-bit Mode

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

- **Arguments passed in registers (why useful?)**
  - First (*xp*) in `%rdi`, second (*yp*) in `%rsi`
  - 64-bit pointers
- **No stack operations required: faster**
- **32-bit data**
  - Data held in registers `%eax` and `%edx`
  - `movl` operation (the `l` refers to data width, not address width)
Swap Long Ints in 64-bit Mode

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

text

64-bit data

- Data held in registers %rax and %rdx
- movq operation
- “q” stands for quad-word
Complete Memory Addressing Modes

- Remember, the addresses used for accessing memory in `mov` (and other) instructions can be computed in several different ways.

- Most General Form:
  \[
  D(Rb,Ri,S) \rightarrow \text{Mem}[\text{Reg}[Rb] + S*\text{Reg}[Ri] + D]
  \]
  - **D:** Constant “displacement” value represented in 1, 2, or 4 bytes
  - **Rb:** Base register: Any of the 8/16 integer registers
  - **Ri:** Index register: Any, except for `%esp` or `%rsp`; `%ebp` unlikely
  - **S:** Scale: 1, 2, 4, or 8 (*why these numbers?*)

- Special Cases: can use any combination of D, Rb, Ri and S
  - `(Rb,Ri) \rightarrow \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]]` (S=1, D=0)
  - `D(Rb,Ri) \rightarrow \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D]` (S=1)
  - `(Rb,Ri,S) \rightarrow \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]]` (D=0)
Address Computation Examples

<table>
<thead>
<tr>
<th>Example</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>0x1e080</td>
</tr>
</tbody>
</table>
Address Computation Instruction

- `leal Src, Dest`
  - `Src` is address mode expression
  - Set `Dest` to address computed by expression
    - (lea stands for *load effective address*)
  - Example: `leal (%edx, %ecx, 4), %eax`

**Uses**

- Computing addresses without a memory reference
  - E.g., translation of `p = &x[i];`
- Computing arithmetic expressions of the form `x + k*i`
  - `k = 1, 2, 4, or 8`
Some Arithmetic Operations

- **Two Operand (Binary) Instructions:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subl</td>
<td>Dest = Dest - Src</td>
</tr>
<tr>
<td>imull</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>shll</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarl</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shrl</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorl</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andl</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orl</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

- **Watch out for argument order! (especially subl)**
- **No distinction between signed and unsigned int (why?)**
  - except arithmetic vs. logical shift right
Some Arithmetic Operations

- One Operand (Unary) Instructions

  - incl `Dest`\n    \[ Dest = Dest + 1 \]\n    increment
  
  - decl `Dest`\n    \[ Dest = Dest - 1 \]\n    decrement
  
  - negl `Dest`\n    \[ Dest = -Dest \]\n    negate
  
  - notl `Dest`\n    \[ Dest = \sim Dest \]\n    bitwise complement

- See textbook section 3.5.5 for more instructions: `mul`, `cltd`, `idivl`, `divl`
Using `leal` for Arithmetic Expressions (IA32)

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

```assembly
arith:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sall $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax
    movl %ebp,%esp
    popl %ebp
    ret
```
Understanding arith (IA32)

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

```
movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y  (t1)
leal (%edx,%edx,2),%edx  # edx = y + 2*y = 3*y
sall $4,%edx  # edx = 48*y  (t4)
addl 16(%ebp),%ecx  # ecx = z+t1  (t2)
leal 4(%edx,%eax),%eax  # eax = 4+t4+x  (t5)
imull %ecx,%eax  # eax = t5*t2  (rval)
```
Understanding arith (IA32)

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

```
movl 8(%ebp),%eax       # eax = x
movl 12(%ebp),%edx      # edx = y
leal (%edx,%eax),%ecx   # ecx = x+y  (t1)
leal (%edx,%edx,2),%edx  # edx = y + 2*y = 3*y
sall $4,%edx            # edx = 48*y  (t4)
addl 16(%ebp),%ecx      # ecx = z+t1  (t2)
leal 4(%edx,%eax),%eax  # eax = 4+t4+x  (t5)
imull %ecx,%eax         # eax = t5*t2  (rval)
```
Understanding arith (IA32)

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

- movl 8(%ebp),%eax  # eax = x
- movl 12(%ebp),%edx  # edx = y
- leal (%edx,%eax),%ecx  # ecx = x+y (t1)
- leal (%edx,%edx,2),%edx  # edx = y + 2*y = 3*y
- sall $4,%edx  # edx = 48*y (t4)
- addl 16(%ebp),%ecx  # ecx = z+t1 (t2)
- leal 4(%edx,%eax),%eax  # eax = 4+t4+x (t5)
- imull %ecx,%eax  # eax = t5*t2 (rval)
Understanding arith (IA32)

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

```
movl 8(%ebp),%eax        # eax = x
movl 12(%ebp),%edx      # edx = y
leal (%edx,%eax),%ecx   # ecx = x+y (t1)
leal (%edx,%edx,2),%edx  # edx = y + 2*y = 3*y
sall $4,%edx            # edx = 48*y (t4)
addl 16(%ebp),%ecx      # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax  # eax = 4+t4+x (t5)
imull %ecx,%eax         # eax = t5*t2 (rval)
```
Observations about arith

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

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Get exact same code when compile:
- \((x+y+z) \times (x+4+48*y)\)

```assembly
movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx # edx = y
leal (%edx,%eax),%ecx # ecx = x+y (t1)
leal (%edx,%edx,2),%edx  # edx = y + 2*y = 3*y
sall $4,%edx  # edx = 48*y (t4)
addl 16(%ebp),%ecx # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax # eax = 4+t4+x (t5)
imull %ecx,%eax  # eax = t5*t2 (rval)
```
Another Example (IA32)

```c
int logical(int x, int y) {
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

### Logical Function

#### Set Up
- `pushl %ebp`
- `movl %esp,%ebp`

#### Body
- `movl 8(%ebp),%eax`
- `xorl 12(%ebp),%eax`
- `sarl $17,%eax`
- `andl $8185,%eax`

#### Finish
- `movl %ebp,%esp`
- `popl %ebp`
- `ret`

---

```
movl 8(%ebp),%eax  # eax = x
xorl 12(%ebp),%eax # eax = x^y
sarl $17,%eax     # eax = t1>>17
andl $8185,%eax   # eax = t2 & 8185
```

---

The stack layout shows:
- `ebp` at offset 0
- `x` at offset 4
- `y` at offset 8
- `rtn adr` at offset 12
- `old %ebp` at offset 16

---

48
Another Example (IA32)

```c
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

```assembly
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret

movl 8(%ebp),%eax          eax = x
xorl 12(%ebp),%eax         eax = x^y (t1)
sarl $17,%eax              eax = t1>>17 (t2)
andl $8185,%eax            eax = t2 & 8185
```
Another Example (IA32)

```c
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

```assembly
logical:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

```assembly
movl 8(%ebp),%eax   ; eax = x
xorl 12(%ebp),%eax  ; eax = x^y (t1)
sarl $17,%eax       ; eax = t1>>17 (t2)
andl $8185,%eax     ; eax = t2 & 8185
```
Another Example (IA32)

```c
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

```
2^{13} = 8192,
2^{13} - 7 = 8185
...0010000000000000, ...0001111111111001
```

```
2^{13} = 8192,
2^{13} - 7 = 8185
...0010000000000000, ...0001111111111001
```

Logical:
```
pushl %ebp
movl %esp,%ebp
```

Set Up
```
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8158,%eax
```

Body
```
movl %ebp,%esp
popl %ebp
ret
```

Finish

```
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8158,%eax
```

Compiler optimization

```
eax = x
```

```
eax = x^y (t1)
```

```
eax = t1>>17 (t2)
```

```
eax = t2 & 8158
```