Machine-Level Programming I: Memory and Instructions
Lecture 4

Instructor:
Alvin R. Lebeck

Slides based on those from Randy Bryant and Dave O’Hallaron

Administrivia
- Homework #1 Due Sept 12
- Homework #2 will be up soon (C Program w/ pointers)
- We have two UTAs!
  - Michael Zhou
  - Dennis Ochei

Outline
- Arrays
- Pointers
- Linked list example
- Pointer Arithmetic
- Instruction Set Architecture

Reading
Chapter 2, start in on Chapter 3
A Program’s View of Memory

- **What is Memory?** a bunch of bits
- **Looks like** a large linear array
- **Find things by** indexing into array
  - unsigned integer
- **Most computers support byte (8-bit) addressing**
  - Each byte has a unique address (location).
  - Byte of data at address 0x100 and 0x101
- **Machine Has “Word Size”**
  - Nominal size of integer-valued data
    - Including addresses
  - 32-bit vs. 64-bit addresses
  - 32-bit word of data at address 0x100 and 0x104

Memory Partitions

- **Text for instructions**
  - add res, src1, src2
  - mem[res] = mem[src1] + mem[src2]
- **Data**
  - static (constants, globals)
  - dynamic (heap, new allocated)
  - grows up
- **Stack**
  - local variables
  - grows down
- **Variables are names for memory locations**
  - int x;
A Simple Program’s Memory Layout

...  
int result; // global var  
main()  
{  
    int x;  
    ...  
    result = x + result;  
    ...  
}  
mem[0x208] = mem[0x400] + mem[0x208]

Reference (handle) vs. Pointer

Java
- “The value of a reference type variable, in contrast to that of a primitive type, is a reference to (an address of) the value or set of values represented by the variable”  
  http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html
- Cannot manipulate the value of the reference

C
- A pointer is a memory location that contains the address of another memory location
- Can manipulate the value of pointer (double edge sword)
Pointers

- “address of” operator &
  - don’t confuse with bitwise AND operator (later today)

**Given**

```c
int x; int *p;
p = &x;
```

**Then**

```c
*p = 2;  and x = 2; produce the same result
```

- What happens for `p = 2;`?

```
|x| 0x26cf0
|p| 0x26d00
```

On 32-bit machine, `p` is 32-bits

Arrays

- In C
- Static allocation
  ```c
  int ar[100];
  ```
- Dynamic allocation
  ```c
  malloc(nbytes); /* allocates space in data segment of memory */
  free(ptr);    /* frees memory so it can be reused */
  int *ar;
ar = malloc(100*sizeof(int));
ar[27] = 1928374;
free(ar);
  ```
- No automatic growing or shrinking
- No constructor to initialize values
Address Calculation

- x is a pointer, what is x+33?
- A pointer, but where?
  - what does calculation depend on?
- Result of adding an int to a pointer depends on size of object pointed to
- Result of subtracting two pointers is an int

\[(d + 3) - d = \_\_\_\_\_\_\_\]

- Be careful, be very careful. large address values...

```
int * a = new int[100];
a = (int *) malloc (100*sizeof(int));
```

```
0 1 32 33 98 99
```

a[33] is the same as *(a+33)
if a is 0x00a0, then a+1 is 0x00a4, a+2 is 0x00a8
(decimal 160, 164, 168)

```
double * d = new double[200];
d = (double *) malloc (100*sizeof(double))
```

```
0 1 199
```

*(d+33) is the same as d[33]
if d is 0x00b0, then d+1 is 0x00b8, d+2 is 0x00c0
(decimal 176, 184, 192)

More Pointer Arithmetic

- address one past the end of an array is ok for pointer comparison only
- what’s at *(begin+44)?
- what does begin++ mean?
- how are pointers compared using < and using == ?
- what is value of end - begin?

```
char * a = new char[44];
char * begin = a;
char * end = a + 44;
```

```
0 1 15 16 42 43
```

while (begin < end)
{
    *begin = 'z';
    begin++;
}
Array Example

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int *a;
    int k;
    a = (int *) malloc(100*sizeof(int));
    int *p = a; // do this after malloc

    for (k = 0; k < 100; k++)
    {
        *p = k;
        p++;
    }

    printf("Entry 3 is %d
",a[3]);
    return(0);
}
```

C Array of Structures -> Linked List

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct node {
    int me;
    struct node *next;
} Node;

int main()
{
    Node *head, *p;
    int k;

    head = (Node *) malloc(sizeof(Node));
    p = head;
    p->me = 0;
    p->next = NULL;

    for (k = 1; k <= 9; k++)
    {
        p->next = (Node *) malloc(sizeof(Node));
        p = p->next;
        p->me = k;
        p->next = NULL;
    }

    p = head;
    while (p != NULL) {
        printf("%d 0x%lx 0x%lx
", p->me, (unsigned long) p, (unsigned long) p->next);
        p = p->next;
    }

    return(0);
}
```

- Given ar = 0x10000, what do you think memory layout looks like?
C Array of Structures -> Linked List

```c
#include <stdio.h>
#include <stdlib.h>

struct node {
    int me;
    struct node *next;
};

int main()
{
    struct node *ar;
    struct node *p;
    int k;
    ar = (struct node *) malloc(10*sizeof(struct node));
    p = ar;
    for (k = 0; k < 9; k++)
    {
        p->me = k;
        p->next = &ar[k+1];
        p++;
    }
    p->me = 9;
    p->next = NULL;
    p = ar[0];
    while (p != NULL) {
        printf("%d 0x%lx 0x%lx
", p->me, (unsigned long) p, (unsigned long) p->next);
        p = p->next;
    }
    return(0);
}
```

Given ar = 0x10000, what does memory layout look like?

<table>
<thead>
<tr>
<th>Me</th>
<th>Memory Address</th>
<th>Memory Contents</th>
<th>Source Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x26ca8</td>
<td>0</td>
<td>me</td>
</tr>
<tr>
<td>1</td>
<td>0x26cb0</td>
<td>0x26cb0</td>
<td>next</td>
</tr>
<tr>
<td>2</td>
<td>0x26cb8</td>
<td>0x26cb8</td>
<td>ar[0]</td>
</tr>
<tr>
<td>3</td>
<td>0x26cc0</td>
<td>0x26cc0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0x26cc8</td>
<td>0x26cc8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0x26cd0</td>
<td>0x26cd0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0x26cd8</td>
<td>0x26cd8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0x26ce0</td>
<td>0x26ce0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0x26ce8</td>
<td>0x26ce8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0x26cf0</td>
<td>0x0</td>
<td>ar[9]</td>
</tr>
</tbody>
</table>

- me is int (4 bytes)
- next is node* (4 bytes)

Memory Layout
Memory Manager (Heap Manager)

- `malloc / new (C/C++)`
- Library routines that handle memory management for data segment (allocation / deallocation)
- Java has garbage collection (reclaim memory of unreferenced objects)
- C/C++ must use `free/delete`, else memory leak

Strings as Arrays

- A string is an array of characters with `\0` at the end
- Each element is one byte, ASCII code
- `\0` is null (ASCII code 0)
Strlen()

- `strlen()` returns the # of characters in a string
  - same as # elements in char array?

```c
int strlen(char * s)
// pre: '\0' terminated
// post: returns # chars
{
    int count=0;
    while (*s++)
        count++;
    return count;
}
```

Memory Summary

- Computer memory is linear array of bytes
- Pointer is memory location that contains address of another memory location
- Code examples are linked to course web page
- We’ll visit these topics again throughout semester

Next

- Instruction set architecture (ISA)

Reading: Chapter 3
Outline

Machine Programming Basics
- Instruction Set Architectures
- Brief History of Intel Architecture
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64

Turning C into Object Code
- Code in files: `p1.c` `p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
  - Use basic optimizations (`-O1`)
  - Put resulting binary in file `p`

```
C program (p1.c p2.c)  
|                    |
| Compiler (gcc -S)  |
|                    |
Asm program (p1.s p2.s)  
|                    |
| Assembler (gcc or as) |
Object program (p1.o p2.o)  
|                    |
| Linker (gcc or ld) |
|                    |
Executable program (p)  
|                    |
| Static libraries (.a) |
```
Definitions

- **Architecture**: (also instruction set architecture: ISA) The parts of a processor design that one needs to understand to write assembly code.
- **Microarchitecture**: Implementation of the architecture.
- **Architecture examples**: instruction set specification, registers.
- **Microarchitecture examples**: cache sizes and core frequency.

Instruction Sets

```c
#include <stdio.h>

main()
{
    int a[100];
    int *p;
    int k;

    p = &a;
    for (k = 0; k < 100; k++)
    {
        *p = k;
        p++;
    }

    printf("entry 3 = %d\n", a[3]);
}
```

What primitive operations do we need? (i.e., What should be implemented in hardware?)
## Design Space of Instruction Sets

### Five Primary Dimensions

- **Operations**
  - add, sub, mul, ...
  - How is it specified?
- **Number of explicit operands**
  - \( 0, 1, 2, 3 \)
- **Operand Storage**
  - Where besides memory?
- **Memory Address**
  - How is memory location specified?
- **Type & Size of Operands**
  - byte, int, float, vector, ...
  - How is it specified?

### Other Aspects

- **Successor instruction**
  - How is it specified?
- **Conditions**
  - How are they determined?
- **Encodings**
  - Fixed or variable? Wide?
- **Parallelism**

---

## Basic ISA Classes

### Accumulator:

<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>add A</td>
<td>acc ← acc + mem[A]</td>
</tr>
<tr>
<td>1+x</td>
<td>addx A</td>
<td>acc ← acc + mem[A + x]</td>
</tr>
</tbody>
</table>

### Stack:

<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>add</td>
<td>tos ← tos + next (JAVA VM)</td>
</tr>
</tbody>
</table>

### General Purpose Register:

<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>add A B</td>
<td>B ← B + A (Intel x86)</td>
</tr>
<tr>
<td>3</td>
<td>add A B C</td>
<td>C ← B + A</td>
</tr>
</tbody>
</table>

### Load/Store: (MIPS, ARM)

<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>add Ra Rb Rc</td>
<td>Ra ← Rb + Rc</td>
</tr>
<tr>
<td></td>
<td>load Ra Rb</td>
<td>Ra ← mem[Rb]</td>
</tr>
<tr>
<td></td>
<td>store Ra Rb</td>
<td>mem[Rb] ← Ra</td>
</tr>
</tbody>
</table>
Adding Registers to an ISA

- A place to hold values that can be named within the instruction
- Like memory, but much smaller
  - 8-64 locations
- Intel operands in either memory or register
  - add A B

Intel x86 Processors

- Totally dominate desktop, laptop, & server computer market
  - Will it last?
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
  - Many quirks because of backward compatibility
- 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>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td>Pentium 4F</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>2667-3333</td>
</tr>
</tbody>
</table>

- First 16-bit processor. Basis for IBM PC & DOS
- 1MB address space
- First 32 bit processor, referred to as IA32
- Added “flat addressing”
- Capable of running Unix
- 32-bit Linux/gcc uses no instructions introduced in later models
- First 64-bit processor, referred to as x86-64
- Core i7

Intel’s 64-Bit

- Intel Attempted Radical Shift from IA32 to IA64
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
- AMD Stepped 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

- IA32
  - The traditional x86

- x86-64/EM64T
  - The emerging standard

Presentation

- Book presents IA32 in Sections 3.1—3.12
- Covers x86-64 in 3.13
- We will cover both simultaneously, but mostly IA32
- Most labs will be based on IA32, may on occasion be x86-64

Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
Stored Program Computer

- **Instructions**: a fixed set of built-in operations
- Instructions and data are stored in the (same) computer memory
- Fetch-Execute Cycle
  ```java
  while (!done)
      fetch instruction
      execute instruction
  ```
- This is usually done by the hardware for speed
- This is what the Java Virtual Machine does

What Must be Specified?

- Instruction Format
  - how do we tell what operation to perform?
- Location of operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Data type and Size
- Operations
  - what are supported
- Successor instruction
  - jumps, conditions, branches
- **fetch-decode-execute is implicit!**
### Turning C into Object Code

- **Code in files**: `p1.c p2.c`
- **Compile with command**: `gcc -O1 p1.c p2.c -o p`
  - Use basic optimizations (`-O1`)
  - Put resulting binary in file `p`

![Diagram of compilation process](image)

```
native text
  C program (p1.c p2.c)

  Compiler (gcc -S)

  Asm program (p1.s p2.s)

  Assembler (gcc or as)

  Object program (p1.o p2.o)

  Linker (gcc or ld)

  Executable program (p)
```

### Assembly Programmer’s View

- **Programmer-Visible State**
  - **PC**: Program counter
    - Address of next instruction
    - Called “EIP” (IA32) or “RIP” (x86-64)
  - **Register file**
    - Frequently 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
Assembly Characteristics: Data Types

- “Integer” data of 1, 2, or 4 bytes
  - Data values
  - Addresses (untyped pointers)

- Floating point data of 4, 8, or 10 bytes
  - Intel has some 80-bit floats

- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

Assembly Characteristics: Operations

- Arithmetic & Logical
  - Perform arithmetic function on register or memory data

- Data Movement: 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
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
    popl %ebp
    ret
```

Some compilers use instruction “leave”

Obtain with command

```bash
/usr/bin/gcc -O1 -S code.c
```

Produces file `code.s`

Object (Machine) Code

**Code for sum**

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040 &lt;sum&gt;:</td>
<td></td>
</tr>
<tr>
<td>0x55</td>
<td>pushl %ebp</td>
</tr>
<tr>
<td>0x89</td>
<td>movl %esp,%ebp</td>
</tr>
<tr>
<td>0xe5</td>
<td>movl 12(%ebp),%eax</td>
</tr>
<tr>
<td>0x8b</td>
<td>addl 8(%ebp),%eax</td>
</tr>
<tr>
<td>0xc3</td>
<td>popl %ebp</td>
</tr>
<tr>
<td>0x401040</td>
<td>ret</td>
</tr>
</tbody>
</table>

- Total of 11 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address 0x401040

- **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
Machine Instruction Example

- **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 \( %eax \)
  - Operands:
    - \( x \): Register \( %eax \)
    - \( y \): Memory \( M[ebp+8] \)
    - \( t \): Register \( %eax \)

- **Object Code**
  - 3-byte instruction
  - Stored at address \( 0x80483ca \)

<table>
<thead>
<tr>
<th>C Code</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{int t = x+y;}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Assembly Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{addl 8(%ebp),%eax}</td>
</tr>
</tbody>
</table>

Similar to expression:
- \( x += y \)

More precisely:
- \( \text{int eax; int *ebp; eax += ebp[2]} \)

<table>
<thead>
<tr>
<th>Object Code</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3-byte instruction</td>
<td></td>
</tr>
<tr>
<td>Stored at address ( 0x80483ca )</td>
<td></td>
</tr>
</tbody>
</table>

Disassembling Object Code

Disassembled

<table>
<thead>
<tr>
<th>Disassembled</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{080483c4 &lt;sum&gt;:}</td>
</tr>
<tr>
<td>\texttt{080483c4: 55 push %ebp}</td>
</tr>
<tr>
<td>\texttt{080483c5: 89 e5 mov %esp,%ebp}</td>
</tr>
<tr>
<td>\texttt{080483c7: 8b 45 0c mov Oxc(%ebp),%eax}</td>
</tr>
<tr>
<td>\texttt{080483ca: 03 45 08 add 0x8(%ebp),%eax}</td>
</tr>
<tr>
<td>\texttt{080483cd: 5d pop %ebp}</td>
</tr>
<tr>
<td>\texttt{080483ce: c3 ret}</td>
</tr>
</tbody>
</table>

- **Disassembler**
  - \texttt{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 \texttt{a.out} (complete executable) or \texttt{o} file
Alternate Disassembly

Object

Disassembled

Dump of assembler code for function sum:
0x080483c4 <sum+0>: push %ebp
0x080483c5 <sum+1>: mov %esp,%ebp
0x080483c7 <sum+3>: mov 0xc(%ebp),%eax
0x080483ca <sum+6>: add 0x8(%ebp),%eax
0x080483cd <sum+9>: pop %ebp
0x080483ce <sum+10>: ret

■ Within gdb Debugger

gdb p
disassemble sum
- Disassemble procedure
x/11xb sum
- Examine the 11 bytes starting at sum

What Can be Disassembled?

% 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

■ Anything that can be interpreted as executable code
■ Disassembler examines bytes and reconstructs assembly source
Machine Programming I: Basics

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

### Integer Registers (IA32)

<table>
<thead>
<tr>
<th>Register</th>
<th>Origin (mostly obsolete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>accumulate</td>
</tr>
<tr>
<td>%ecx</td>
<td>counter</td>
</tr>
<tr>
<td>%edx</td>
<td>data</td>
</tr>
<tr>
<td>%ebx</td>
<td>base</td>
</tr>
<tr>
<td>%esi</td>
<td>source</td>
</tr>
<tr>
<td>%edi</td>
<td>index</td>
</tr>
<tr>
<td>%esp</td>
<td>destination</td>
</tr>
<tr>
<td>%ebp</td>
<td>stack</td>
</tr>
<tr>
<td>%esp</td>
<td>pointer</td>
</tr>
<tr>
<td>%ebp</td>
<td>base</td>
</tr>
</tbody>
</table>

16-bit virtual registers (backwards compatibility)
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><code>Imm</code></td>
<td><code>Reg</code></td>
<td><code>movl %0x4,%eax</code></td>
<td><code>temp = 0x4;</code></td>
</tr>
<tr>
<td><code>Mem</code></td>
<td><code>Reg</code></td>
<td><code>movl $-147,(%eax)</code></td>
<td><code>*p = -147;</code></td>
</tr>
<tr>
<td><code>Reg</code></td>
<td><code>Mem</code></td>
<td><code>movl %eax,%edx</code></td>
<td><code>temp2 = temp1;</code></td>
</tr>
<tr>
<td><code>Mem</code></td>
<td><code>Reg</code></td>
<td><code>movl (%eax),%edx</code></td>
<td><code>*p = temp;</code></td>
</tr>
</tbody>
</table>

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

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address

\[
\text{movl} \ (%\text{ecx}),\%\text{eax}
\]

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

\[
\text{movl} \ 8(%\text{ebp}),\%\text{edx}
\]

Using Simple Addressing Modes

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

**swap:**

\[
\begin{align*}
\text{pushl} & \%\text{ebp} \\
\text{movl} & \%\text{esp},\%\text{ebp} \\
\text{pushl} & \%\text{ebx} \\
\text{movl} & 8(\%\text{ebp}),\%\text{edx} \\
\text{movl} & 12(\%\text{ebp}),\%\text{ecx} \\
\text{movl} & (\%\text{edx}),\%\text{ebx} \\
\text{movl} & (\%\text{ecx}),\%\text{eax} \\
\text{movl} & \%\text{eax},(\%\text{edx}) \\
\text{movl} & \%\text{ebx},(\%\text{ecx}) \\
\text{popl} & \%\text{ebx} \\
\text{popl} & \%\text{ebp} \\
\text{ret}
\end{align*}
\]

{\textbf{Set Up}}

{\textbf{Body}}

{\textbf{Finish}}
Using Simple Addressing Modes

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

Understanding Swap

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

Stack (in memory)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>yp</td>
</tr>
<tr>
<td>8</td>
<td>xp</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>-4</td>
<td>Old %ebx</td>
</tr>
</tbody>
</table>

Register Value

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>xp</td>
</tr>
<tr>
<td>%ecx</td>
<td>yp</td>
</tr>
<tr>
<td>%ebx</td>
<td>t0</td>
</tr>
<tr>
<td>%eax</td>
<td>t1</td>
</tr>
</tbody>
</table>

Assembly Code

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

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

popl %ebx
popl %ebp
ret
```
Understanding Swap

%eax
%edx 0x124
%ecx
%ebx
%esi
%edi
%esp
%ebp 0x104

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
Understanding Swap

\[
\begin{array}{c}
\%eax \\
\%edx 0x124 \\
\%ecx 0x120 \\
\%ebx 0x123 \\
\%esi \\
\%edi \\
\%esp \\
\%ebp 0x104 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{Offset} & \text{Address} & \text{Address} \\
\hline
yp & 12 & 0x120 \\
xp & 8 & 0x124 \\
\hline
\end{array}
\]

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
Understanding Swap

\[
\begin{align*}
\%eax & 456 \\
\%edx & 0x124 \\
\%ecx & 0x120 \\
\%ebx & 123 \\
\%esi & \\
\%edi & \\
\%esp & \\
\%ebp & 0x104 \\
\end{align*}
\]

\[
\begin{array}{c|c|c}
\text{Address} & 0x123 & 0x124 \\
& 0x120 & 0x11c \\
& 0x114 & 0x118 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{Offset} & \text{Rtn adr} & 0x104 \\
yp & 12 & 0x108 \\
xp & 8 & 0x10c \\
4 & & 0x100 \\
\end{array}
\]

\[
\begin{align*}
\text{movl} & \ 8(\%ebp), \ %edx \quad \# \ edx = xp \\
\text{movl} & \ 12(\%ebp), \ %ecx \quad \# \ ecx = yp \\
\text{movl} & \ (%edx), \ %ebx \quad \# \ ebx = \ast xp \ (t0) \\
\text{movl} & \ (%ecx), \ %eax \quad \# \ eax = \ast yp \ (t1) \\
\text{movl} & \ %eax, \ (%edx) \quad \# \ \ast xp = t1 \\
\text{movl} & \ %ebx, \ (%ecx) \quad \# \ \ast yp = t0
\end{align*}
\]
## Understanding Swap

### Memory Addressing Modes

### Most General Form

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

- **D**: Constant “displacement” 1, 2, or 4 bytes
- **Rb**: Base register: Any of 8 integer registers
- **Ri**: Index register: Any, except for \%esp
  - Unlikely you’d use \%ebp, either
- **S**: Scale: 1, 2, 4, or 8 (why these numbers?)

### Special Cases

- \((Rb,Ri)\quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]]\)
- \(D(Rb,Ri)\quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D]\)
- \( (Rb,Ri,S) \quad \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]] \)
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          | 16             
- char *                | 4                  | 4              | 8              
- Or any other pointer  |                    |                |                

x86-64 Integer Registers

- %rax %eax %r8 %r8d
- %rbx %ebx %r9 %r9d
- %rcx %ecx %r10 %r10d
- %rdx % edx %r11 %r11d
- %rsi %esi %r12 %r12d
- %rdi %edi %r13 %r13d
- %rsp %esp %r14 %r14d
- %rbp %ebp %r15 %r15d

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

- Long word l (4 Bytes) ↔ Quad word q (8 Bytes)

- New instructions:
  - `movl` ↔ `movq`
  - `addl` ↔ `addq`
  - `sall` ↔ `salq`
  - etc.

- 32-bit instructions that generate 32-bit results
  - Set higher order bits of destination register to 0
  - Example: `addl`

32-bit code for swap

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

```assembly
swaps:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

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

    popl %ebx
    popl %ebp
    ret
```
64-bit code for swap

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

- Operands passed in registers (why useful?)
  - First (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers
- No stack operations required
- 32-bit data
  - Data held in registers %eax and %edx
  - `movl` operation

Set Up

Body

Finish

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
    - “q” stands for quad-word

Set Up

Body

Finish
Machine Programming I: Summary

- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts
- C, assembly, machine code
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- Assembly Basics: Registers, operands, move
  - The x86 move instructions cover wide range of data movement forms
- Intro to x86-64
  - A major departure from the style of code seen in IA32