Lecture 11

Representation of Programs

CPSC 275
Introduction to Computer Systems

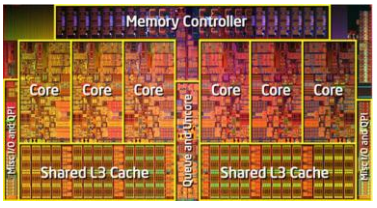
Intel x86 Processors

- Totally dominate today's 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
  - Hard to match performance of Reduced Instruction Set Computers (RISC). Read CSaPP pp. 342-344 for more coverage on this.

Intel x86 Evolution: Milestones

Name	*Date*	*Transistors*	*MHz*
8086	1978	29K	5-10
– First 16-bit processor. Basis for IBM PC & DOS			
– 1MB address space			
386	1985	275K	16-33
– First 32 bit processor, referred to as IA32			
– Added *flat addressing*			
– Capable of running Unix			
– 32-bit Linux/gcc generates instructions for this model by default			
Pentium 4F	2004	125M	2800-3800
– First 64-bit processor, referred to as x86-64			

Intel Core i7-970



- Year: 2010
- # Transistors: > 700M
- CPU Clock: 3.2-3.4 GHz

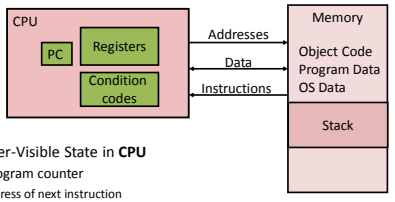
x86 Clones: Advanced Micro Devices (AMD)

- 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

Some Definitions

- Architecture: (also instruction set architecture: ISA) The parts of a processor design that one needs to understand to write assembly code.
  - Examples: instruction set specification, registers.
- Microarchitecture: Implementation of the architecture.
  - Examples: cache sizes and core frequency.
- Example ISAs (Intel): x86, IA-32
- We will focus only on IA-32.

Assembly Programmer's View



Programmer-Visible State in **CPU**

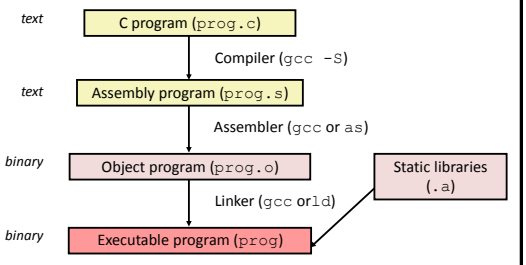
- PC: Program counter
  - Address of next instruction
  - Called "EIP" (IA32) or "RIP" (x86-64)
- Register file
  - Heavily used program data
- Condition codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

Memory

- Byte addressable array
- Code, user data, (some) OS data
- Includes stack used to support procedures

Turning C into Object Code

- Source **`prog.c`**
- Compile with command: **`gcc -o prog prog.c`**



Compiling Into Assembly

C Code

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

Generated IA-32 Assembly

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

Obtain with command

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

Produces file `code.s` where `-O1` indicates level-one optimization.

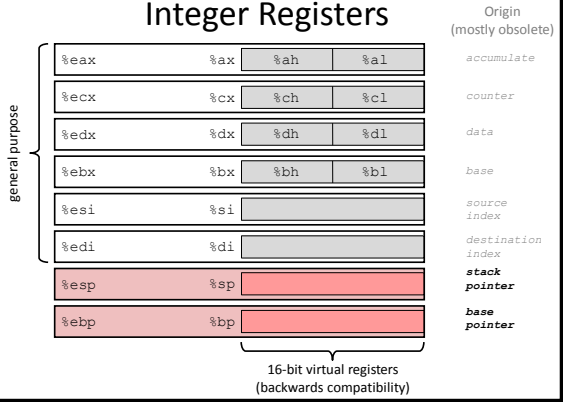
Assembly Characteristics: Data Types

- Integral data of 1, 2, or 4 bytes
  - Data values
  - Addresses
- Floating point data of 4, 8, or 10 bytes (more on this later)
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

Assembly Characteristics: Operations

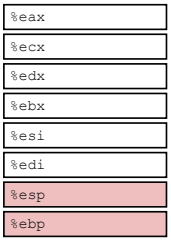
- Perform arithmetic function on register or memory data
- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

Integer Registers



Moving Data

- 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:*** consecutive memory at address given by register
    - Simplest example: **`(%eax)`**
    - Various other "address modes"



`movl` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movl`	*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) Mem[Reg[R]]
  - Register R specifies memory address

```
movl (%ecx),%eax
```

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

```
movl 8(%ebp),%edx
```

Using Simple Addressing Modes

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

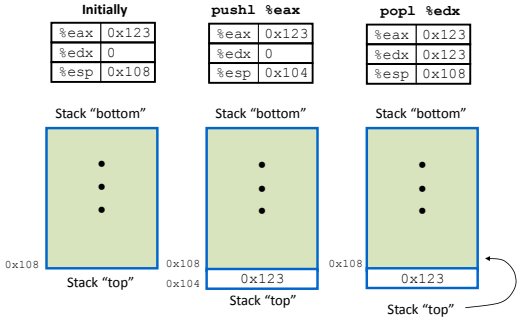
```
swap:
  pushl %ebp
  movl  %esp,%ebp
  pushl %ebx

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

  popl  %ebx
  popl  %ebp
  ret
```

Set Up

Body

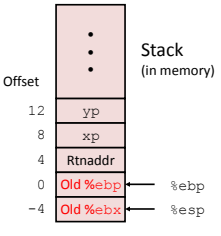
Finish

Moving Data To/From Stack



Understanding Swap

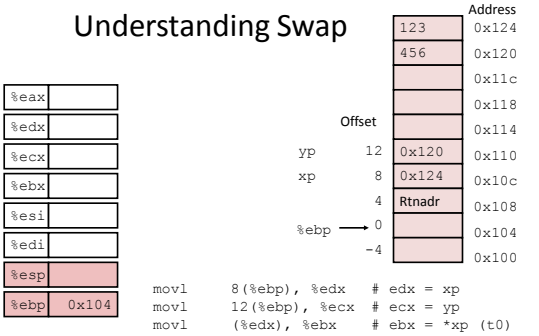
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%edx	xp
%ecx	yp
%ebx	t0
%eax	t1

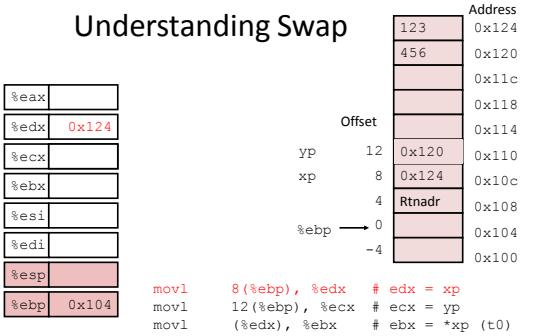
```
movl    8(%ebp), %edx   # edx = xp
movl    12(%ebp), %ecx  # ecx = yp
movl    (%edx), %ebx    # ebx = *xp (t0)
movl    (%ecx), %eax    # eax = *yp (t1)
movl    %eax, (%edx)    # *xp = t1
movl    %ebx, (%ecx)    # *yp = t0
```

Understanding Swap



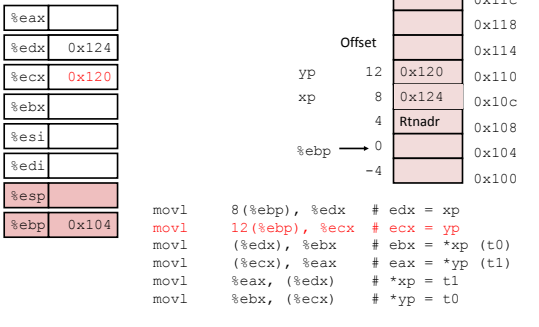

```
movl    8(%ebp), %edx   # edx = xp
movl    12(%ebp), %ecx  # ecx = yp
movl    (%edx), %ebx    # ebx = *xp (t0)
movl    (%ecx), %eax    # eax = *yp (t1)
movl    %eax, (%edx)    # *xp = t1
movl    %ebx, (%ecx)    # *yp = t0
```

Understanding Swap



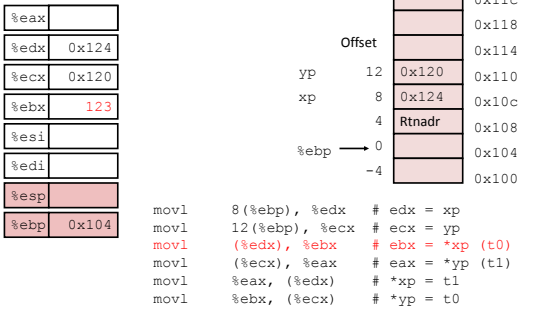

```
movl    8(%ebp), %edx   # edx = xp
movl    12(%ebp), %ecx  # ecx = yp
movl    (%edx), %ebx    # ebx = *xp (t0)
movl    (%ecx), %eax    # eax = *yp (t1)
movl    %eax, (%edx)    # *xp = t1
movl    %ebx, (%ecx)    # *yp = t0
```

Understanding Swap



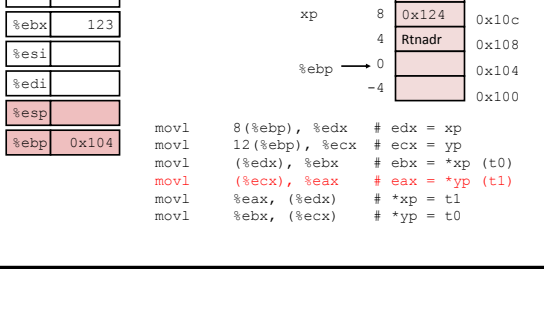

```
movl    8(%ebp), %edx   # edx = xp
movl    12(%ebp), %ecx  # ecx = yp
movl    (%edx), %ebx    # ebx = *xp (t0)
movl    (%ecx), %eax    # eax = *yp (t1)
movl    %eax, (%edx)    # *xp = t1
movl    %ebx, (%ecx)    # *yp = t0
```

Understanding Swap



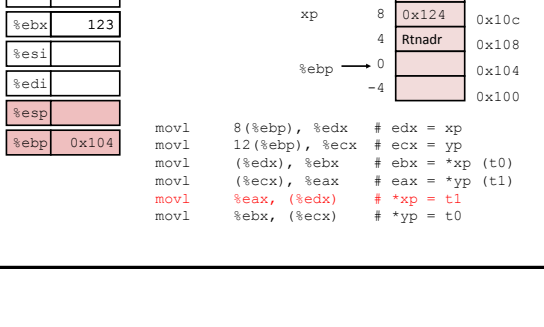

```
movl    8(%ebp), %edx   # edx = xp
movl    12(%ebp), %ecx  # ecx = yp
movl    (%edx), %ebx    # ebx = *xp (t0)
movl    (%ecx), %eax    # eax = *yp (t1)
movl    %eax, (%edx)    # *xp = t1
movl    %ebx, (%ecx)    # *yp = t0
```

Understanding Swap



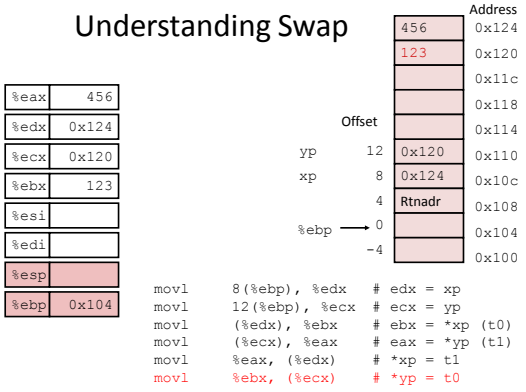

```
movl    8(%ebp), %edx   # edx = xp
movl    12(%ebp), %ecx  # ecx = yp
movl    (%edx), %ebx    # ebx = *xp (t0)
movl    (%ecx), %eax    # eax = *yp (t1)
movl    %eax, (%edx)    # *xp = t1
movl    %ebx, (%ecx)    # *yp = t0
```

Understanding Swap




```
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




```
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
```

Practice Problems

- Read CSaPP Sec. 3.0 (Intro to the chapter), 3.1-3.4 and try the following problems:

  3.1, 3.2, 3.3, 3.4, 3.5