Giving credit where credit is due

- Most of slides for this lecture are based on slides created by Drs. Bryant and O’Hallaron, Carnegie Mellon University.
- I have modified them and added new slides.

IA32 Processors

Totally Dominate Computer Market

Evolutionary Design
- Starting in 1978 with 8086
- Added more features as time goes on
- Still support old features, although obsolete

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!

X86 Evolution: Programmer’s View

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16-bit processor. Basis for IBM PC &amp; DOS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Limited to 1MB address space. DOS only gives you 640K</td>
</tr>
<tr>
<td>80286</td>
<td>1982</td>
<td>134K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added elaborate, but not very useful, addressing scheme</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Basis for IBM PC-AT and Windows</td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Extended to 32 bits. Added “flat addressing”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Capable of running Unix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Linux/gcc uses no instructions introduced in later models</td>
</tr>
</tbody>
</table>

X86 Evolution: Programmer’s View

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>486</td>
<td>1989</td>
<td>1.9M</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>3.1M</td>
</tr>
<tr>
<td>Pentium/MMX</td>
<td>1997</td>
<td>4.5M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added special collection of instructions for operating on 64-bit vectors of 1, 2, or 4 byte integer data</td>
</tr>
<tr>
<td>PentiumPro</td>
<td>1995</td>
<td>6.5M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added conditional move instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Big change in underlying microarchitecture</td>
</tr>
</tbody>
</table>
X86 Evolution: Programmers View

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>8.2M</td>
</tr>
</tbody>
</table>
|          |      | - Added “streaming SIMD” instructions for operating on 128-bit vectors of 1, 2, or 4 byte integer or floating point data  
|          |      | - Our fish machines  
| Pentium 4 | 2001 | 42M         |
|          |      | - Added 8-byte formats and 144 new instructions for streaming SIMD mode  

X86 Evolution: Clones

Advanced Micro Devices (AMD)
- Historically
  - AMD has followed just behind Intel  
  - A little bit slower, a lot cheaper  
- Recently
  - Recruited top circuit designers from Digital Equipment Corp.  
  - Exploited fact that Intel distracted by IA64  
  - Now are close competitors to Intel  
  - Developing own extension to 64 bits  

New Species: IA64

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Itanium</td>
<td>2001</td>
<td>10M</td>
</tr>
</tbody>
</table>
|          |      | - Extends to IA64, a 64-bit architecture  
|          |      | - Radically new instruction set designed for high performance  
|          |      | - Will be able to run existing IA32 programs  
|          |      | - On-board “x86 engine”  
|          |      | - Joint project with Hewlett-Packard  
| Itanium 2 | 2002 | 221M        |
|          |      | - Big performance boost  

Assembly Programmers View

- CPU  
  - EIP: Program Counter  
  - Register File  
  - Condition Codes  
  - Most recent arithmetic operation  
- Memory  
  - Object Code  
  - Program Data  
  - OS Data  
  - Stack  
- Addresses  
  - Data  
  - Instructions  
- Programmer-Visible State  
  - EIP  
  - Program Counter  
  - Address of next instruction  
  - Register File  
  - Heavy used program data  
  - Condition Codes  
  - Store status information about most recent arithmetic operations  
  - Used for conditional branching  
- Memory  
  - Byte addressable array  
  - Code, user data, (some) OS data  
  - Stack used to support procedures  

Turning C into Object Code

- Code in files p1.c, p2.c  
- Compile with command: gcc -O p1.c p2.c -o p  
- Use optimizations (-O)  
- Put resulting binary in file p  
- C program (p1.c p2.c)  
  - Compiler (gcc -S)  
  - Assembler (gcc or as)  
- Static libraries (.a)  
- Object program (p1.o p2.o)  
  - Linker (gcc or ld)  
  - Executable program (p)
In integer data of 1, 2, or 4 bytes, useful tool for examining object code. Some libraries are translated into .o

Useful 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.

Compiling Into Assembly

C Code

Generated Assembly

Assembly Characteristics

Minimal Data Types
- "Integer" data of 1, 2, or 4 bytes
- Data values
- Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- No aggregate types such as arrays or structures
- Just contiguous allocated bytes in memory

Primitive 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

Object Code

Code for sum

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 %ebp
  - y: Memory %esp
- Return function value in %eax

Object Code
- 3-byte instruction
- Stored at address 0x401046

Disassembling Object Code

Disassembled

Alternate Disassembly

Object

Disassembled

Within gdb Debugger
- disassemble sum
- Disassemble procedure
- 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

Moving Data

Moving Data
movl Source,Dest:
- Move 4-byte ("long") word
- Lots of these in typical code

Operand Types
- Immediate: Constant integer data
  - Like C constant, but prefixed with 't'
  - E.g., $0x400, $0x533
  - Encoded with 1, 2, or 4 bytes
- Register: One of 8 integer registers
  - But %esp and %ebp reserved for special use
  - Others have special uses for particular instructions
- Memory: 4 consecutive bytes of memory
  - Various "address modes"

movl Operand Combinations

movl Source Destination C Analog

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg</td>
<td>Mem[Reg][R]</td>
<td>movl $-147, (%eax) *p = -147;</td>
</tr>
<tr>
<td>Mem</td>
<td>Mem[Reg][R]</td>
<td>movl $0x4, %eax temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl %eax, %edx temp1 = %eax;</td>
</tr>
<tr>
<td>Mem</td>
<td>Mem</td>
<td>movl %eax, (%edx) *p = temp1;</td>
</tr>
</tbody>
</table>

Simple Addressing Modes

Normal (R) Mem[Reg][R]
- Register R specifies memory address
  movl (%ecx), %eax

Displacement D(R) Mem[Reg][R]+D
- Register R specifies start of memory region
  - Constant displacement D specifies offset
  movl B(%ebp), %edx

Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10C</td>
<td>0x11</td>
<td>%eax</td>
<td>0x100</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
<td>%ecx</td>
<td>0x1</td>
</tr>
<tr>
<td>0x104</td>
<td>0x00</td>
<td>%eax</td>
<td>0x0</td>
</tr>
<tr>
<td>0x100</td>
<td>0xFF</td>
<td>%ecx</td>
<td>0x104</td>
</tr>
</tbody>
</table>

Exercise

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10C</td>
<td>0x3</td>
<td>%eax</td>
<td>0x104</td>
</tr>
<tr>
<td>0x108</td>
<td>0x2</td>
<td>%ecx</td>
<td>0x0</td>
</tr>
<tr>
<td>0x104</td>
<td>0x1</td>
<td>%eax</td>
<td>0x104</td>
</tr>
<tr>
<td>0x100</td>
<td>0x0</td>
<td>%ecx</td>
<td>0x100</td>
</tr>
</tbody>
</table>
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;
}
```

Register Variable

- `tbea`: `yp`
- `tbed`: `xp`
- `tbec`: `t1`
- `tbsd`: `t0`
Understanding Swap

Address

<table>
<thead>
<tr>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>0x120</td>
<td>0x11c</td>
</tr>
<tr>
<td>0x118</td>
<td>0x114</td>
<td>0x110</td>
</tr>
<tr>
<td>0x10c</td>
<td>0x100</td>
<td>0x010</td>
</tr>
<tr>
<td>0x014</td>
<td>0x010</td>
<td>0x00c</td>
</tr>
<tr>
<td>0x008</td>
<td>0x004</td>
<td>0x000</td>
</tr>
</tbody>
</table>

Offset

<table>
<thead>
<tr>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-4</td>
</tr>
</tbody>
</table>

Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%edx)</td>
<td>0x0000 + 0x8</td>
<td>0x008</td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>0x0000 + 0x100</td>
<td>0x100</td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td>0x0000 + 4*0x100</td>
<td>0x400</td>
</tr>
<tr>
<td>%eax + (%edx,2)</td>
<td>2*0x0000 + 0x80</td>
<td>0x8000</td>
</tr>
</tbody>
</table>

Indexed Addressing Modes

Most General Form

D(Rb,Ri,S) & Mem[Reg[Rb]+S*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 %ebp
- S: Scale: 1, 2, 4, or 8

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]]

Another Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x00</td>
</tr>
<tr>
<td>0x104</td>
<td>0x00</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x100</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x1</td>
</tr>
<tr>
<td>%edx</td>
<td>0x3</td>
</tr>
</tbody>
</table>

Operand | Value
---|------
9(%eax,%edx) | 0x11
0x100(%ecx,4) | 0x00
(%eax,%edx,4) | 0x11
252(%ecx,%edx) | 0xFF
Exercise

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
<td>%eax</td>
<td>0x100</td>
</tr>
<tr>
<td>0x104</td>
<td>0x00</td>
<td>%ecx</td>
<td>0x104</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
<td>%edx</td>
<td>0x1</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
<td>%ebx</td>
<td>0x8</td>
</tr>
</tbody>
</table>

Operand | Value
---|---
3(%eax,%edx) | 0x00
254(%edx,2) | 0xFF
(%eax,%edx, 4) | 0x00
(%ecx,%ebx) | 0x11

More on Data Movement

MOVSB and MOVZBL
- MOVSB sign-extends a single byte, and copies it into a double-word destination
- MOVZBL expands a single byte to 32 bits with 24 leading zeros, and copies it into a double-word destination

Example:
%eax = 0x12345678
%edx = 0xAAAA BBB

MOV %dh, %al
MOVSB %dh, %eax
MOVZBL %dh, %eax

More on Data Movement

MOVBL moves two bytes, when one of its operands is a register, it must be one of the 8 two-byte registers

MOVZBL moves a single byte, when one of its operands is a register, it must be one of the 8 single-byte registers

Exercise

%eax = 0x12345678
%edx = 0xAAAA22CC

MOV %dh, %al %eax = #2
MOV %dh, %eax %eax = #3
MOV %dh, %eax %eax = #3
MOVBL %dl, %eax %eax = #5

Example

Assume register %eax holds value X
%ecx holds value Y

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result in %edx</th>
</tr>
</thead>
<tbody>
<tr>
<td>lea 8(%eax), %edx</td>
<td>X+8</td>
</tr>
<tr>
<td>lea 8(%eax,%ecx), %edx</td>
<td>X+Y</td>
</tr>
<tr>
<td>lea 8(%eax,%ecx), %edx</td>
<td>X+Y+8</td>
</tr>
<tr>
<td>lea 8(%eax,%ecx, 4), %edx</td>
<td>5X+8</td>
</tr>
<tr>
<td>lea 8(%eax,%ecx, 2), %edx</td>
<td>X+2Y+8</td>
</tr>
</tbody>
</table>

Address Computation Instruction

lea Src, Dest
- Src is an address mode expression
- Set Dest to address denoted by expression

Uses
- Computing address without doing memory reference
- E.g., translation of p = arr[i];
- Computing arithmetic expressions of the form x + k*y
  - k = 1, 2, 4, or 8.
Some Arithmetic Operations

Format | Computation
--- | ---
Two Operand Instructions | 
addl Src,Dest | Dest = Dest + Src
subl Src,Dest | Dest = Dest - Src
imull Src,Dest | Dest = Dest * Src
sarl Src,Dest | Dest = Dest >> Src
also called shrll
shrl Src,Dest | Dest = Dest >> Src Logical
xorl Src,Dest | Dest = Dest ^ Src
orl Src,Dest | Dest = Dest | Src

Using `leal` for Arithmetic Expressions

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

Understanding `arith`

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

Another Example

```
int logical(int x, int y)
  {
    int t1 = x>y;
    int t2 = t1 >= 17;
    int mask = (~0<<13) - 7;
    int rval = t2 & mask;
    return rval;
  }
```

Logical:

```
int logical(int x, int y)
  {
    int t1 = x>y;
    int t2 = t1 >= 17;
    int mask = (~0<<13) - 7;
    int rval = t2 & mask;
    return rval;
  }
```
**Push and Pop**

PUSHL takes a single operand: the data source, and store it to the top of stack.

For example,
- `PUSHL %eax` has equivalent behavior as
  - `subl $4, %esp` : stack grows downward
  - `movl %eax, (%esp)`
- POPL takes the data destination, and pop the top element of stack onto the destination.
- `POPL %eax` has equivalent behavior as
  - `movl (%esp), %eax`
  - `addl $4, %esp`

**CISC Properties**

Instruction can reference different operand types
- Immediate, register, memory
- Arithmetic operations can read/write memory
- Memory reference can involve complex computation
  - `Rb = 5:Ri + D`
  - Useful for arithmetic expressions, too
- Instructions can have varying lengths
  - IA32 instructions can range from 1 to 15 bytes

---

**Summary: Abstract Machines**

<table>
<thead>
<tr>
<th>Machine Models</th>
<th>Data</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mem</td>
<td>proc</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Assembly</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mem</td>
<td>proc</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 1) byte
- 2) 2-byte word
- 3) 4-byte long word
- 4) contiguous byte allocation
- 5) address of initial byte

---

**Pentium Pro (P6)**

**History**
- Announced in Feb. '95
- Basis for Pentium II, Pentium III, and Celeron processors
- Pentium 4 similar idea, but different details

**Features**
- Dynamically translates instructions to more regular format
  - Very wide, but simple instructions
  - Executes operations in parallel
    - Up to 5 at once
    - Very deep pipeline
    - 12-18 cycle latency

**Pentium Pro Operation**

Translates instructions dynamically into “Uops”
- 118 bits wide
- Holds operation, two sources, and destination

Executes Uops with “Out of Order” engine
- Uop executed when
  - Operands available
  - Functional unit available
  - Execution controlled by “Reservation Stations”
  - Keeps track of data dependencies between uops
  - Allocates resources

**Consequences**
- Indirect relationship between IA32 code & what actually gets executed
- Tricky to predict / optimize performance at assembly level
Whose Assembler?

<table>
<thead>
<tr>
<th>Intel/Microsoft Format</th>
<th>GAS/Gnu Format</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lea eax, [eax*4]</code></td>
<td><code>lea (eax, eax, 2), eax</code></td>
</tr>
<tr>
<td><code>sub esp, 8</code></td>
<td><code>subl $8, esp</code></td>
</tr>
<tr>
<td><code>cmp dword ptr [esp-8], 0</code></td>
<td><code>cmp $0, -8(%ebp)</code></td>
</tr>
<tr>
<td><code>mov eax, dword ptr [eax*4+100h]</code></td>
<td><code>movl $0x100, (eax, 4), eax</code></td>
</tr>
</tbody>
</table>

Intel/Microsoft Diffs from GAS

- Operands listed in opposite order
  - `mov Dest, Src` vs `movl Src, Dest`
- Constants not preceded by `$`, Denote hex with `h` at end
  - `100h` vs `$0x100`
- Operand size indicated by operands rather than operator suffix
  - `sub` vs `subl`
- Addressing format shows effective address computation
  - `[eax*4+100h]` vs `$0x100, (eax, 4)`