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>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>PentiumPro</td>
<td>1995</td>
<td>6.5M</td>
</tr>
</tbody>
</table>

- Added special collection of instructions for operating on 64-bit vectors of 1, 2, or 4 byte integer data
- Added conditional move instructions
- Big change in underlying microarchitecture
X86 Evolution: Programmer's 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>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2001</td>
<td>42M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></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

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

X86 Evolution: Clones

Transmeta
- Recent start-up
- Employer of Linus Torvalds
- Radically different approach to implementation
  - Translates x86 code into “Very Long Instruction Word” (VLIW) code
  - High degree of parallelism
  - Shooting for low-power market

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>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Itanium 2</td>
<td>2002</td>
<td>221M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></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
- Big performance boost
Assembly Programmer's View

Programmer-Visible State
- EIP: Program Counter
  - Address of next instruction
- Register File
  - Heavily used program data
- Condition Codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

Assembly Characteristic

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 contiguously 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

Turning C into Object Code

Code in files: p1.c, p2.c
Compile with command: gcc -O -S p1.c p2.c -o p
- Use optimizations (-O)
- Put resulting binary in file

Text
- C program (p1.c p2.c)
  - Compiler (gcc -S)
- Asm program (p1.s p2.s)
  - Assembler (gcc or as)

Binary
- Object program (p1.o p2.o)
  - Linker (gcc or ld)
- Static libraries (.a)

Compiling Into Assembly

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

Obtain with command:
gcc -O -S code.c
Produces file: code.s

Generated 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 -O -S code.c
Produces file: code.s
Object Code

Code for `sum`

```assembly
0x401040 <sum>:
0x55  push %ebp
0x89 e5  mov %esp,%ebp
0x8b 45 0c  mov 0xc(%ebp),%eax
0x03 45 08  add 0x8(%ebp),%eax
0x89 ec  mov %ebp,%esp
0x05  ret
```

Assembler
- Translates .c 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

Disassembling Object Code

Disassembled

```assembly
00401040 <_sum>:
0: 55  push %ebp
1: 89 e5  mov %esp,%ebp
3: 8b 45 0c  mov 0xc(%ebp),%eax
6: 03 45 08  add 0x8(%ebp),%eax
9: 89 ec  mov %ebp,%esp
b: 5d  pop %ebp
c: c3  ret
```

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

Machine Instruction Example

C Code
```c
int t = x+y;
```

Assembly
```assembly
addl $(%ebp),%eax
```

Similar to expression:
```c
x += y
```

Or
```c
int eax;
int *ebp;
eax += ebp[2]
```

Disassembled

```assembly
0x401046: 03 45 08
```

Alternate Disassembly

Disassembled

```assembly
0x401040 <sum>:
0x55  push %ebp
0x401041 <sum+1>:
0x8b 45 0c  mov 0xc(%ebp),%eax
0x401044 <sum+4>:
0x89 ec  mov %ebp,%esp
0x401047 <sum+7>:
0x03 45 08  add 0x8(%ebp),%eax
0x40104a <sum+10>:
0x89 ec  mov %ebp,%esp
0x40104c <sum+12>:
0x00 40 60  lea 0x0(%esi),%esi
```

Within gdb Debugger
- `gdb p`
- `disassemble sum`
- `x/13b sum`
- `Examine the 13 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

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 '0'
  - E.g., $0x400, $-533
  - 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

```
\[
\begin{array}{|c|c|c|c|}
\hline
\text{Source} & \text{Dest} & \text{Src,Dest} & \text{C Analog} \\
\hline
\text{Reg} & \text{Reg} & \text{Movl } %ax,$%ax & \text{temp } = \text{0x4}; \\
\text{Mem} & \text{Reg} & \text{Movl } %ax, %ax & \text{*p } = \text{-147}; \\
\text{Mem} & \text{Mem} & \text{Movl } %ax, %ax & \text{temp2 } = \text{temp1}; \\
\text{Mem} & \text{Reg} & \text{Movl } %ax, %ax & \text{temp } = \text{*p}; \\
\hline
\end{array}
\]
```

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 8(%ebp), %edx
Understanding Swap

```
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>%eax</th>
<th>456</th>
</tr>
</thead>
<tbody>
<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>4</td>
</tr>
<tr>
<td>%edi</td>
<td>-4</td>
</tr>
<tr>
<td>%esp</td>
<td>0x104</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x100</td>
</tr>
</tbody>
</table>

Address

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

Offset

<table>
<thead>
<tr>
<th>456</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Rtn adr

<table>
<thead>
<tr>
<th>%ebp</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>0x104</td>
</tr>
<tr>
<td>%edi</td>
<td>0x100</td>
</tr>
</tbody>
</table>

Rtn adr

<table>
<thead>
<tr>
<th>%ebp</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>0x104</td>
</tr>
<tr>
<td>%edi</td>
<td>0x100</td>
</tr>
</tbody>
</table>

%eax = yp
%edx = xp
%ecx = *yp (t1)
%ebx = *xp (t0)
%eax = *xp
*yp = ebx

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 %esp
- 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]]

Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>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 denoted by expression

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

Some Arithmetic Operations

**Format**

<table>
<thead>
<tr>
<th>Two Operand Instructions</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addl Src, Dest</code></td>
<td><code>Dest = Dest + Src</code></td>
</tr>
<tr>
<td><code>subl Src, Dest</code></td>
<td><code>Dest = Dest - Src</code></td>
</tr>
<tr>
<td><code>imull Src, Dest</code></td>
<td><code>Dest = Dest * Src</code></td>
</tr>
<tr>
<td><code>sall Src, Dest</code></td>
<td><code>Dest = Dest &lt;&lt; Src</code></td>
</tr>
<tr>
<td><code>sar1 Src, Dest</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code></td>
</tr>
<tr>
<td><code>shrl Src, Dest</code></td>
<td><code>Dest = Dest ^ Src</code></td>
</tr>
<tr>
<td><code>orl Src, Dest</code></td>
<td>`Dest = Dest</td>
</tr>
</tbody>
</table>

**One Operand Instructions**

| incl Dest | Dest = Dest + 1          |
| decl Dest | Dest = Dest - 1          |
| negl Dest | Dest = ~ Dest            |
| notl Dest | Dest = ~ Dest            |

Using `leal` for Arithmetic Expressions

```c
int arith(int x, int y, int z) {
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+y;
    int t4 = y*40;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

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

---

Page 9
int arith
(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+t4;
    int t4 = x*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 = 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)

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 = 3*y
sal $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

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;
}

movl 8(%ebp),%eax # eax = x
movl 12(%ebp),%edx # edx = y
lea (%edx,%eax),%ecx # ecx = x+y (t1)
lea (%edx,%edx,2),%edx # edx = 3*y
sal $4,%edx # edx = 48*y (t4)
addl 16(%ebp),%ecx # ecx = z+t1 (t2)
lea 4(%edx,%eax),%eax # eax = 4+t4+x (t5)
imull %ecx,%eax # eax = t5*t2 (rval)

CISC Properties

Instruction can reference different operands
- Immediate, register, memory
- Arithmetic operations can read/write memory
- Memory reference can involve complex computation
  - Rb + S* 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>1) char</td>
<td>1) loops</td>
</tr>
<tr>
<td></td>
<td>2) int, float</td>
<td>2) conditionals</td>
</tr>
<tr>
<td></td>
<td>3) double</td>
<td>3) switch</td>
</tr>
<tr>
<td></td>
<td>4) struct, array</td>
<td>4) Proc. call</td>
</tr>
<tr>
<td></td>
<td>5) pointer</td>
<td>5) Proc. return</td>
</tr>
</tbody>
</table>

Assembly

<table>
<thead>
<tr>
<th>mem</th>
<th>regs</th>
<th>alu</th>
<th>processor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>mem</td>
</tr>
<tr>
<td>Stack</td>
<td>Cond. Codes</td>
<td></td>
<td>processor</td>
</tr>
</tbody>
</table>

1) byte  3) branch/jump
2) 2-byte word 4) call
3) 4-byte long word 5) set
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
  - 12–16 cycle latency

Features (continued)

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, [ecx*ecx*2]</code></td>
<td><code>leal (%ecx, %ecx, 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 [ebp-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 Differs from GAS**

- Operands listed in opposite order
  
<table>
<thead>
<tr>
<th>Intel/Microsoft</th>
<th>GAS/Gnu</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mov Dest, Src</code></td>
<td><code>movl Src, Dest</code></td>
</tr>
</tbody>
</table>

- Constants not preceded by `$`, Denote hex with ‘h’ at end
  
<table>
<thead>
<tr>
<th>Intel/Microsoft</th>
<th>GAS/Gnu</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>100h</code></td>
<td><code>$0x100</code></td>
</tr>
</tbody>
</table>

- Operand size indicated by operands rather than operator suffix
  
<table>
<thead>
<tr>
<th>Intel/Microsoft</th>
<th>GAS/Gnu</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sub</code></td>
<td><code>subl</code></td>
</tr>
</tbody>
</table>

- Addressing format shows effective address computation
  
<table>
<thead>
<tr>
<th>Intel/Microsoft</th>
<th>GAS/Gnu</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[eax*4+100h]</code></td>
<td><code>$0x100(,%eax,4)</code></td>
</tr>
</tbody>
</table>