ISAs and Microarchitectures

• Instruction Set Architecture
  • The interface between hardware and software
  • “Language” + programmer visible state + I/O = ISA
  • Hardware can change underneath
  • Software can change above
  • Example: IA32, IA64, ALPHA, POWERPC

• Microarchitecture
  • An implementation of an ISA
    – Pentium Pro, 21064, G4, …
  • Can tune your code for specific microarchitectures

• Machine architecture
  • Processor, memory, buses, disks, nics, …
  • Can also tune code for this
ISAs Continued

• State
  • Memory in all its forms
    – Registers

• I/O
  • Special memory locations that are read/written by other devices
  • Processor reading/writing them causes side-effects in the devices
  • Interrupts

• Language
  • How to interpret bits as transformations from State+I/O into State+I/O
    – How to tell the “cone of logic” what to do
Different models

- **CISC = Complex Instruction Set Computer**
  - Push machine language closer to programming languages
  - Hope: more abstraction => more performance
  - IA32, VAX, IBM mainframe processors, …

- **RISC = Reduced Instruction Set Computer**
  - Push machine language closer to the hardware
  - Hope: easier for compiler to produce high performance
  - Alpha, PowerPC, …

- **Others**
  - (V)L IW = (Very) Long Instruction Word : IA64
  - Vector processors: Cray, Hitachi
  - Multithreaded: Tera
  - Reconfigurable (you write the cone of logic directly)

- **Transistors are first order effect in market**
IA32 Processors

Totally Dominate Computer Market

Evolutionary Design

- Starting in mid ’70s with 8080 (8 bit)
- 1978 – 16 bit 8086
  - 8088 version used in IBM PC – 1981
  - Growth of PC
- 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, 16 bit OS/2, and 16-bit 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, 32 bit Windows, 32 bit OS/2, ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Linux/gcc uses no instructions introduced in later models</td>
</tr>
<tr>
<td>486</td>
<td>1989</td>
<td>1.9M</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>3.1M</td>
</tr>
</tbody>
</table>
# X86 Evolution: Programmer’s View

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium/MMX</td>
<td>1997</td>
<td>4.5M</td>
<td>• Added special collection of instructions for operating on 64-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>vectors of 1, 2, or 4 byte integer data</td>
</tr>
<tr>
<td>Pentium II</td>
<td>1997</td>
<td>7M</td>
<td>• Added conditional move instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Big change in underlying microarchitecture</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>8.2M</td>
<td>• Added “streaming SIMD” instructions for operating on 128-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>vectors of 1, 2, or 4 byte integer or floating point data</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2001</td>
<td>42M</td>
<td>• Added 8-byte formats and 144 new instructions for streaming SIMD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mode</td>
</tr>
</tbody>
</table>
Why so many transistors

ISA of P4 is basically the same as 386, but it uses 150 times more transistors

Answer:

Hardware extracts parallelism out of code stream to get higher performance
  multiple issue
  pipelining
  out-of-order and speculative execution

All processors do this these days

Limits to how far this can go, hence newer ISA ideas
New Species: IA64

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Itanium</td>
<td>2000</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”

The principles of machine-level programming we will discuss will apply to current processors, CISC and RISC. Some principles will also apply to LIWs like IA64

Quantum Computers, if we can build them and if they are actually more powerful than classical computers, will be COMPLETELY DIFFERENT
ISA / Machine Model of IA32

**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

**Memory**
- Byte addressable array
- Code, user data, (some) OS data
- Includes 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) (versus -g => debugging info)
  - Put resulting binary in file `p`

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

```
Assembler (gcc or as)
```

```
Object program (p1.o p2.o)
```

```
Static libraries (.a)
```

```
Executable program (p)
```

```
Linker (gcc or ld)
```
Compiling Into Assembly

C Code

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

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`
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 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

Data Flow

Control Flow
Object Code

Code for `sum`

```plaintext
0x401040 <sum>:
  0x55
  0x89
  0xe5
  0x8b
  0x45
  0x0c
  0x03
  0x45
  0x08
  0x89
  0xec
  0x5d
  0xc3
```

- Total of 13 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

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

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

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

Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401040</td>
<td>```&lt;_sum&gt;:`</td>
<td></td>
</tr>
<tr>
<td>0:</td>
<td>55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>1:</td>
<td>89 e5</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>3:</td>
<td>8b 45 0c</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>6:</td>
<td>03 45 08</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>9:</td>
<td>89 ec</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td>b:</td>
<td>5d</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>c:</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>d:</td>
<td>8d 76 00</td>
<td>lea 0x0(%esi),%esi</td>
</tr>
</tbody>
</table>

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
Alternate Disassembly

Disassembled

Object 0x401040: 0x55 0x89 0xe5 0x8b 0x45 0x0c 0x03 0x45 0x08 0x89 0xec 0xc3

Within gdb Debugger

gdb p
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

% 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
Copying Data and Registers

Moving Data (Really Copying)

\texttt{movl \ Source, Dest}: Move 4-byte ("long") word

- Accounts for 31% of all instructions in sample (IA32 – other machines are different)

Operand Types

- **Immediate**: Constant integer data
  - Like C constant, but prefixed with ‘$’
  - 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
  - Special cases => Non-orthogonality (BAD)

- **Memory**: 4 consecutive bytes of memory
  - Various “addressing modes”
### movl Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>movl $0x4, %eax</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl $-147, (%eax)</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movl %eax, (%edx)</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax), %edx</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax), %edx</td>
</tr>
</tbody>
</table>

- Cannot do memory-memory transfers with single instruction
  - Example of NON-ORTHOGONALITY in the IA32 ISA
    » Makes it much harder to program or compile for
Simple Addressing Modes

**Normal (R) Mem[Reg[R]]**
- Register R specifies memory address
  
  ```
  movl (%ecx),%eax => int t = *p;
  ```

**Displacement D(R) Mem[Reg[R]+D]**
- Register R specifies start of memory region
- Constant displacement D specifies offset
  
  ```
  movl 8(%ecx),%edx => int t = p[2];
  ```

  ```
  movl 8(%ebp),%edx => int t = some_argument;
  ```
  - %ebp, %esp used to reference stack. Stack contains arguments to function

All instructions support addressing modes, not just moves
Using Simple Addressing Modes

```c
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
```
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
Indexed 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

All instructions support addressing modes, not just moves

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]]
Address Computation Instruction

`leal Src, Dest`

- `Src` is address mode expression
- Set `Dest` to address denoted by expression

Uses

- Computing address without doing memory reference
  - E.g., translation of \( p = \&x[i]; \)
- Computing arithmetic expressions of the form \( x + k*y \)
  - \( k = 1, 2, 4, \text{ or } 8. \)
# Some Arithmetic Operations

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Two Operand Instructions</strong></td>
<td></td>
</tr>
<tr>
<td>addl</td>
<td>Source, Destination</td>
</tr>
<tr>
<td>subl</td>
<td>Source, Destination</td>
</tr>
<tr>
<td>imull</td>
<td>Source, Destination</td>
</tr>
<tr>
<td>sall</td>
<td>Source, Destination</td>
</tr>
<tr>
<td>sarl</td>
<td>Source, Destination</td>
</tr>
<tr>
<td>shrl</td>
<td>Source, Destination</td>
</tr>
<tr>
<td>xorl</td>
<td>Source, Destination</td>
</tr>
<tr>
<td>andl</td>
<td>Source, Destination</td>
</tr>
<tr>
<td>orl</td>
<td>Source, Destination</td>
</tr>
<tr>
<td><strong>One Operand Instructions</strong></td>
<td></td>
</tr>
<tr>
<td>incl</td>
<td>Destination</td>
</tr>
<tr>
<td>decl</td>
<td>Destination</td>
</tr>
<tr>
<td>negl</td>
<td>Destination</td>
</tr>
<tr>
<td>notl</td>
<td>Destination</td>
</tr>
</tbody>
</table>
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 + 4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

**arith:**

```asm
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
```

{ } Set Up

{ } Body

{ } Finish
Understanding 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;
}
```

```
movl 8(%ebp),%eax       # eax = x
movl 12(%ebp),%edx      # edx = y
leal (%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)
leal 4(%edx,%eax),%eax  # eax = 4+t4+x (t5)
imull %ecx,%eax         # eax = t5*t2 (rval)
```
Another Example

```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
```

- Set Up
  - pushl %ebp
  - movl %esp,%ebp
  - movl 8(%ebp),%eax
  - xorl 12(%ebp),%eax
          
  Body
  - 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 (t1)
  sarl $17,%eax  
  eax = t1>>17 (t2)
  andl $8185,%eax  
  eax = t2 & 8185
CISC Properties

Instruction can reference different operand types
  • Immediate, register, memory

Arithmetic operations can read/write memory

Memory reference can involve complex computation
  • $R_b + S*R_i + D$
  • Useful for arithmetic expressions, too

Instructions can have varying lengths
  • IA32 instructions can range from 1 to 15 bytes

RISC
  • Instructions are fixed length (usually a word)
  • special load/store instructions for memory
    – Generally simpler addressing modes
  • Other operations use only registers (but there are lots of registers)
Summary: Abstract Machines

Machine Models

Data

Control

1) char
2) int, float
3) double
4) struct, array
5) pointer

1) loops
2) conditionals
3) goto
4) Proc. call
5) Proc. return

Assembly

1) byte
2) 4-byte long word
3) branch/jump
4) call
5) ret

1) byte long word
2) 8-byte quad word
3) contiguous byte allocation
4) address of initial byte
Pentium Pro (P6)

History

• Announced in Feb. ‘95
• Basis for Pentium II, Pentium III, and Celeron processors

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
PentiumPro 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
  • Difficult to predict / optimize performance at assembly level
Whose Assembler?

Intel/Microsoft Format | GAS/Gnu Format
---|---
lea eax, [ecx+ecx*2] | leal (%ecx,%ecx,2),%eax
sub esp, 8 | subl $8,%esp
cmp dword ptr [ebp-8], 0 | cmpl $0,-8(%ebp)
mov eax, dword ptr [eax*4+100h] | movl $0x100(,%eax,4),%eax

Intel/Microsoft Differs from GAS

- **Operands listed in opposite order**
  ```
  mov Dest, Src  
  movl Src, Dest
  ```

- **Constants not preceded by ‘$’, Denote hexadecimal with ‘h’ at end**
  ```
  100h  
  $0x100
  ```

- **Operand size indicated by operands rather than operator suffix**
  ```
  sub  
  subl
  ```

- **Addressing format shows effective address computation**
  ```
  [eax*4+100h]  
  $0x100(,%eax,4)
  ```