ICS08

Instructor: Aleksandar Kuzmanovic

TA: Ionut Trestian

Recitation 3
Machine-Level Programming I: Introduction

Topics

- Assembly Programmer’s Execution Model
- Accessing Information
  - Registers
  - Memory
- Arithmetic operations
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>- Added special collection of instructions for operating on 64-bit vectors of 1, 2, or 4 byte integer data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PentiumPro</td>
<td>1995</td>
<td>6.5M</td>
</tr>
<tr>
<td>- Added conditional move instructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Big change in underlying microarchitecture</td>
<td></td>
<td></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>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>
<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

- 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
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>Extends to IA64, a 64-bit architecture</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radically new instruction set designed for high performance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Will be able to run existing IA32 programs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>On-board “x86 engine”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joint project with Hewlett-Packard</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>
<tr>
<td>Big performance boost</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

**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)
  - 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)  Static libraries (.a)

Linker (gcc or ld)          Executable program (p)
```

Binary

Text
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

```bash
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
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`

Similar to expression:

```c
x += y
```

```assembly
addl 8(%ebp),%eax
```

```assembly
0x401046: 03 45 08
```
Disassembling Object Code

Disassembled

```
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
 d: 8d 76 00 lea    0x0(%esi),%esi
```

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
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
CISC Properties

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

Arithmetic operations can read/write memory

Memory reference can involve complex computation
- $R_b + S \times R_i + D$
- Useful for arithmetic expressions, too

Instructions can have varying lengths
- IA32 instructions can range from 1 to 15 bytes
Summary: Abstract Machines

Machine Models

C

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

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

Assembly

mem
Stack

reg
Cond. Codes
alu
processor

mem
proc

1) byte
2) 2-byte word
3) 4-byte long word
4) contiguous byte allocation
5) address of initial byte
PentiumPro Block Diagram
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
Machine-Level Programming II: Control Flow

Topics

- Condition Codes
  - Setting
  - Testing

- Control Flow
  - If-then-else
  - Varieties of Loops
  - Switch Statements
Condition Codes

Single Bit Registers

- **CF** Carry Flag
- **ZF** Zero Flag
- **SF** Sign Flag
- **OF** Overflow Flag

Implicitly Set By Arithmetic Operations

- `addl Src, Dest`
  
  C analog: \( t = a + b \)
  
  - **CF** set if carry out from most significant bit
    - Used to detect unsigned overflow
  
  - **ZF** set if \( t == 0 \)
  
  - **SF** set if \( t < 0 \)
  
  - **OF** set if two’s complement overflow
    - \((a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)\)

*Not Set by `leal` instruction*
Explicit Setting by Compare Instruction

\texttt{cmp}l \ Src2,Src1

- \texttt{cmp}l b,a like computing a-b without setting destination
- CF set if carry out from most significant bit
  - Used for unsigned comparisons
- ZF set if a == b
- SF set if (a-b) < 0
- OF set if two’s complement overflow
  \[(a>0 \ \&\& \ b<0 \ \&\& \ (a-b)<0) \ \| \ \| \ (a<0 \ \&\& \ b>0 \ \&\& \ (a-b)>0)\]
Explicit Setting by Test instruction

```c
testl Src2,Src1
```
- Sets condition codes based on value of `Src1 & Src2`
  - Useful to have one of the operands be a mask
- `testl b,a` like computing `a&b` without setting destination
- ZF set when `a&b == 0`
- SF set when `a&b < 0`
## Reading Condition Codes

### SetX Instructions

- Set single byte based on combinations of condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF^OF) &amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Jumping

**jX Instructions**
- Jump to different part of code depending on condition codes

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF^OF) &amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}

_max:
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%edx
movl 12(%ebp),%eax
cmpl %eax,%edx
jle L9
movl %edx,%eax
L9:
movl %ebp,%esp
popl %ebp
ret
“Do-While” Loop Example

C Code

```c
int fact_do(int x)
{
    int result = 1;
    do {
        result *= x;
        x = x-1;
    } while (x > 1);
    return result;
}
```

Goto Version

```c
int fact_goto(int x)
{
    int result = 1;
    loop:
    result *= x;
    x = x-1;
    if (x > 1)
        goto loop;
    return result;
}
```

- Use backward branch to continue looping
- Only take branch when “while” condition holds
“Do-While” Loop Compilation

Goto Version

```c
int fact_goto(int x)
{
    int result = 1;
    loop:
        result *= x;
        x = x-1;
        if (x > 1) goto loop;
    return result;
}
```

Registers

- `%edx`  `x`
- `%eax`  `result`

Assembly

```
_fact_goto:
    pushl %ebp                     # Setup
    movl %esp,%ebp                 # Setup
    movl $1,%eax                   # eax = 1
    movl 8(%ebp),%edx             # edx = x

L11:
    imull %edx,%eax                # result *= x
    decl %edx                     # x--
    cmpl $1,%edx                  # Compare x : 1
    jg L11                        # if > goto loop

movl %ebp,%esp                  # Finish
popl %ebp                       # Finish
ret                             # Finish
```
General “Do-While” Translation

**C Code**

```c
do
    Body
while (Test);
```

**Goto Version**

```c
loop:
    Body
    if (Test)
        goto loop
```

- **Body** can be any C statement
  - Typically compound statement:
    ```c
    {
        Statement_1;
        Statement_2;
        ...
        Statement_n;
    }
    ```

- **Test** is expression returning integer
  - \( = 0 \) interpreted as false
  - \( \neq 0 \) interpreted as true
General “While” Translation

C Code

while (Test)
    Body

do
    Body
while (Test);
done:

Do-While Version

Goto Version

if (!Test)
goto done;
do
    Body
while (Test);
done:

if (!Test)
goto done;
loop:
    Body
if (Test)
goto loop;
done:
“For” Loop Example

```c
int result;
for (result = 1;
p != 0;
p = p>>1) {
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```

General Form

```
for (Init; Test; Update)
    Body
```

- **Init**
  - `result = 1`

- **Test**
  - `p != 0`

- **Update**
  - `p = p >> 1`

**Body**

```c
{
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```
"For" → "While"

**For Version**

```c
for (Init; Test; Update)
    Body
```

**While Version**

```c
Init;
while (Test) {
    Body
    Update;
}
```

**Do-While Version**

```c
Init;
if (!Test)
goto done;
do {
    Body
    Update;
} while (Test)
done:
```

**Goto Version**

```c
Init;
if (!Test)
goto done;
loop:
    Body
    Update;
    if (Test)
goto loop;
done:
```
Switch Statements

Implementation Options

- Series of conditionals
  - Good if few cases
  - Slow if many

- Jump Table
  - Lookup branch target
  - Avoids conditionals
  - Possible when cases are small integer constants

- GCC
  - Picks one based on case structure

- Bug in example code
  - No default given

```c
typedef enum
{ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
    switch (op) {
    case ADD :
        return '+';
    case MULT:
        return '*';
    case MINUS:
        return '-';
    case DIV:
        return '/';
    case MOD:
        return '%';
    case BAD:
        return '?';
    }
}
```
Jump Table Structure

Switch Form

```c
switch(op) {
    case val_0:
        Block 0
    case val_1:
        Block 1
    ...
    case val_n-1:
        Block n-1
}
```

Jump Table

- **jtab:**
  - Targ0
  - Targ1
  - Targ2
  - ...
  - Targn-1

Jump Targets

- **Targ0:** Code Block 0
- **Targ1:** Code Block 1
- **Targ2:** Code Block 2
- ...
- **Targn-1:** Code Block n-1

Approx. Translation

```c
target = JTab[op];
goto *target;
```
Jump Table

Table Contents

```
.section .rodata
.align 4
.L57:
    .long .L51 #Op = 0
    .long .L52 #Op = 1
    .long .L53 #Op = 2
    .long .L54 #Op = 3
    .long .L55 #Op = 4
    .long .L56 #Op = 5
```

Enumerated Values

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>0</td>
</tr>
<tr>
<td>MULT</td>
<td>1</td>
</tr>
<tr>
<td>MINUS</td>
<td>2</td>
</tr>
<tr>
<td>DIV</td>
<td>3</td>
</tr>
<tr>
<td>MOD</td>
<td>4</td>
</tr>
<tr>
<td>BAD</td>
<td>5</td>
</tr>
</tbody>
</table>

Targets & Completion

```
.L51:
    movl $43,%eax # '+'
    jmp .L49
.L52:
    movl $42,%eax # '*'
    jmp .L49
.L53:
    movl $45,%eax # '-'
    jmp .L49
.L54:
    movl $47,%eax # '/'
    jmp .L49
.L55:
    movl $37,%eax # '%'
    jmp .L49
.L56:
    movl $63,%eax # '?'
    # Fall Through to .L49
```
Jump Table

- Doesn’t show up in disassembled code
- Can inspect using GDB

```
gdb code-examples
(gdb) x/6xw 0x8048bc0
  • Examine 6 hexadecimal format “words” (4-bytes each)
  • Use command “help x” to get format documentation
```

```
0x8048bc0 <_fini+32>:
  0x08048730
  0x08048737
  0x08048740
  0x08048747
  0x08048750
  0x08048757
```
Extracting Jump Table from Binary

Jump Table Stored in Read Only Data Segment (.rodata)

- Various fixed values needed by your code

Can examine with objdump

```
objdump code-examples -s --section=.rodata
```

- Show everything in indicated segment.

Hard to read

- Jump table entries shown with reversed byte ordering

```
Contents of section .rodata:
8048bc0 30870408 37870408 40870408 47870408 0...7...@...G...
8048bd0 50870408 57870408 46616374 28256429 P...W...Fact(%d)
8048be0 203d2025 6c640a00 43686172 203d2025 = %ld..Char = %
...
```

- E.g., 30870408 really means 0x08048730
Disassembled Targets

- `movl %esi,%esi` does nothing
- Inserted to align instructions for better cache performance
## Matching Disassembled Targets

<table>
<thead>
<tr>
<th>Entry Address</th>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08048730</td>
<td>b8 2b 00 00 00</td>
<td>movl</td>
</tr>
<tr>
<td>0x08048735</td>
<td>eb 25</td>
<td>jmp</td>
</tr>
<tr>
<td>0x08048737</td>
<td>b8 2a 00 00 00</td>
<td>movl</td>
</tr>
<tr>
<td>0x0804873c</td>
<td>eb 1e</td>
<td>jmp</td>
</tr>
<tr>
<td>0x0804873e</td>
<td>89 f6</td>
<td>movl</td>
</tr>
<tr>
<td>0x08048740</td>
<td>b8 2d 00 00 00</td>
<td>movl</td>
</tr>
<tr>
<td>0x08048745</td>
<td>eb 15</td>
<td>jmp</td>
</tr>
<tr>
<td>0x08048747</td>
<td>b8 2f 00 00 00</td>
<td>movl</td>
</tr>
<tr>
<td>0x0804874c</td>
<td>eb 0e</td>
<td>jmp</td>
</tr>
<tr>
<td>0x0804874e</td>
<td>89 f6</td>
<td>movl</td>
</tr>
<tr>
<td>0x08048750</td>
<td>b8 25 00 00 00</td>
<td>movl</td>
</tr>
<tr>
<td>0x08048755</td>
<td>eb 05</td>
<td>jmp</td>
</tr>
<tr>
<td>0x08048757</td>
<td>b8 3f 00 00 00</td>
<td>movl</td>
</tr>
</tbody>
</table>
Summarizing

C Control
- if-then-else
- do-while
- while
- switch

Assembler Control
- jump
- Conditional jump

Compiler
- Must generate assembly code to implement more complex control

Standard Techniques
- All loops converted to do-while form
- Large switch statements use jump tables

Conditions in CISC
- CISC machines generally have condition code registers

Conditions in RISC
- Use general registers to store condition information
- Special comparison instructions
- E.g., on Alpha:
  \[ \text{cmple } 16,1,1 \]
  - Sets register $1$ to 1 when Register $16 \leq 1$