Class Announcements

1. Urmish's office hours from 9-10 AM was cancelled. Alternated office hours details soon.
2. If you need alternate Midterm 1 email me with your name and the reason.
3. Questions about Prog. Assign. 1?
4. Details about hands-on intro to C Program. relevant to Prog. Assign. 1 soon.

Lecture Overview

- IEEE Floating Point
- ISA history and intro
- Assembly Intro, Disassembly
- IA32 Registers
- IA32 Operand forms
- IA32 Data Movement Instructions

> Assume a set of 4 chars. are in an integer-sized variable (X).
> Assume an instruction exists to print out the character all the way to the right...

\[
\begin{array}{cccc}
X & A & B & C & D \\
\end{array}
\]

\[
\text{putc} \ X \quad \text{(prints D)}
\]

> Invent instructions, and write code to print ABDC, without changing X.
Floating Point (Won’t be on exam)

1. Fractional Binary Notation

\[ \frac{1}{2} \frac{1}{4} \frac{1}{8} \ldots \]

IEEE Floating Point (Won’t be on exam)

- Limitations with binary Notation:
  - Can only exactly represent numbers of the form \( x/2^k \)
  - Just one setting of binary point within the \( w \) bits
- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point arithmetic
  - Nice standards for rounding, overflow, underflow

IEEE Floating Point (Won’t be on exam)

- Numerical Form:
  \[ (-1)^s \times M \times 2^E \]
  - \( S \) is sign bit: negative or positive
  - Significant \( M \) normally a fractional value in range \([0, 2)\).
  - Exponent \( E \) weights value by power of two
- Encoding
  - MSB \( s \) is sign bit \( s \)
  - exp field encodes \( E \) (but is not equal to \( E \))
  - frac field encodes \( M \) (but is not equal to \( M \))
IEEE Floating Point (Won’t be on exam)

1. Normalized values: Exp neither all zeroes nor all ones
   \[ E = \text{Exp} - \text{Bias} \text{ and } M = 1 + f \]
2. Denormalized values: Exp is all zeroes
   \[ E = 1 - \text{Bias} \text{ and } M = f \]
3. Special values: Exp is all ones: Inf, NaN
   (Bias is \(2^{k-1} - 1\), E.g. for \(k=4\) bit exp field, Bias = 7)

---

IEEE Floating Point (Won’t be on exam)

- Single precision: 32 bits
  
<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8-bits</td>
<td>23-bits</td>
</tr>
</tbody>
</table>

- Double precision: 64 bits
  
<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11-bits</td>
<td>52-bits</td>
</tr>
</tbody>
</table>

- Extended precision: 80 bits (Intel only)
  
<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15-bits</td>
<td>63 or 64-bits</td>
</tr>
</tbody>
</table>

---

Focus: \textbf{x86 architecture}

- 1960s: CISC
  System/360(IBM), B5000(Burroughs), Motorola 68000

- 1970s: Large Scale Integration
  8008, 8080, 8086 (Intel), PDP-11, VAX(DEC)

- 1980s: RISC, Instruction Level Parallelism, Pipelining
  80286, 80386, 80486(Intel), Motorola 68020

- 1990s - today: Multi-threading , Multi-Core, Open source processors
  Pentium, Pentium Pro, Intel Core(Intel), Athlon series(AMD)
Assembly Programmer’s view

- **Programmer-Visible State**
  - 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, heap, (some) OS data
  - Includes stack used to support procedures

Assembly Language

- Why learn assembly?
  - Preferred for low level tasks: boot loaders, system calls
  - Less overhead than with HLL code
  - Helpful while debugging
  - Can access some new features of processor only through assembly until compilers add support.
  - One of the oldest tools in Programmers toolbox
- We will use the ATT syntax and not the Intel syntax. Consider:
  - Intel: `mov eax,1; mov ebx,0fh; int 80h`
  - AT&T: `movl $1,%eax; movl $0xff,%ebx; int $0x80`

32-bit architecture
- 2-address instruction set
- CISC (not RISC, load/store)
- 8 registers (depending on how we count)
- Uses condition codes for control instructions

Moore’s Law

- Entire architecture on 1 slide:
Characteristics of Assembly Programs: Data Types

1. Integer data of 1, 2, or 4 bytes (data values, addresses)
2. Floating point data of 4, 8, or 10 bytes
3. No aggregate types such as arrays or structures (Just contiguously allocated bytes in memory)

Characteristics of Assembly Programs: Operations

1. Arithmetic operations on memory or registers
2. Transfer data between memory and registers: Load and Store
3. Transfer control: Unconditional jumps, Conditional branches

Example Assembly Program

```
.include "defines.h"
data
hw: .string "hello world\n"
text .globl main
main:
    movl $SYS_write,%eax
    movl $1,%ebx
    movl $hw,%ecx
    movl $12,%edx
    int $0x80
    movl $SYS_exit,%eax
    xorl %ebx,%ebx
    int $0x80
    ret
```

Generating Assembly Code from C

```
#include <stdio.h>

int a = 10, b = 20;
int main(){
    int t = a;
    a = b;
    b = t;
    printf("%d %d\n",a,b);
    return 0;
}
```
Disassembly of Executables

1. objdump –S
2. gdb and then disassembly command
3. Compile with –g for source code info:
   man gcc says:
   
   `-g` Produce debugging information in the operating system's native format
   (stabs, COFF, XCOFF, or DWARF 2).
   GDB can work with this debugging information.

Registers

<table>
<thead>
<tr>
<th>31</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%eax</td>
<td>%ah</td>
<td>%al</td>
<td></td>
</tr>
<tr>
<td>%ecx</td>
<td>%ecx</td>
<td>%ch</td>
<td>%cl</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td>%edx</td>
<td>%dh</td>
<td>%dl</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td>%ebx</td>
<td>%bh</td>
<td>%bl</td>
<td></td>
</tr>
</tbody>
</table>

4 More Registers

<table>
<thead>
<tr>
<th>31</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esi</td>
<td>%si</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td>%dl</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td></td>
</tr>
</tbody>
</table>

Registers

1. %esp, %ebp : stack pointer, base pointer
2. %eip : instruction pointer
3. x86-64 : %rax, %rbx etc. (64 bits)

What to do when there are not enough registers?
Answer: Store temporarily in memory.
On to the instruction set. Our coverage will be of a small subset.

Classify instructions:
- data movement
- arithmetic
- logical (and shift)
- control

### Operands

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Addressing mode name</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>$imm$</td>
<td>immediate</td>
<td>value in machine code</td>
</tr>
<tr>
<td>$rR$</td>
<td>register</td>
<td>value in register $r$</td>
</tr>
<tr>
<td>$imm$</td>
<td>absolute</td>
<td>address given by $imm$</td>
</tr>
<tr>
<td>($rR$)</td>
<td>register direct (incased in textbook)</td>
<td>address in $rR$</td>
</tr>
<tr>
<td>$imm$, $rR$</td>
<td>base displacement</td>
<td>address is $imm + rR$</td>
</tr>
</tbody>
</table>

### Some more operand formats in IA32

<table>
<thead>
<tr>
<th>Operand</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>
<tr>
<td>$0x11$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$%eax$</td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>$0x108$</td>
<td></td>
</tr>
<tr>
<td>(@%eax)</td>
<td></td>
</tr>
<tr>
<td>4(%%eax)</td>
<td></td>
</tr>
<tr>
<td>9(%%eax,%edx)</td>
<td></td>
</tr>
<tr>
<td>260(%%ecx,%edx)</td>
<td></td>
</tr>
<tr>
<td>0xPC(%%eax,4)</td>
<td></td>
</tr>
<tr>
<td>(%%eax,%edx,4)</td>
<td></td>
</tr>
</tbody>
</table>
### Five possible combination of Source and Destination Types

<table>
<thead>
<tr>
<th>Operand</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x100</td>
<td>Register</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
<td>Absolute address</td>
</tr>
<tr>
<td>$0x108</td>
<td>0x108</td>
<td>Immediate</td>
</tr>
<tr>
<td>(%eax)</td>
<td>0xFF</td>
<td>Address 0x100</td>
</tr>
<tr>
<td>4(%eax)</td>
<td>0xAB</td>
<td>Address 0x104</td>
</tr>
<tr>
<td>9(%eax,%edx)</td>
<td>0x11</td>
<td>Address 0x10C</td>
</tr>
<tr>
<td>260(%ecx,%edx)</td>
<td>0x13</td>
<td>Address 0x108</td>
</tr>
<tr>
<td>0xFC(%edx,4)</td>
<td>0xFF</td>
<td>Address 0x100</td>
</tr>
<tr>
<td>(%eax,%edx,4)</td>
<td>0x11</td>
<td>Address 0x10C</td>
</tr>
</tbody>
</table>

### Data Movement Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source</th>
<th>Destination</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb</td>
<td>S</td>
<td>D</td>
<td>Nondestructive copy of S to D</td>
</tr>
<tr>
<td>movw</td>
<td>S</td>
<td>D</td>
<td>Sign-extended, nondestructive copy of S to D</td>
</tr>
<tr>
<td>movl</td>
<td>S</td>
<td>D</td>
<td>Non-destructive copy of S to D</td>
</tr>
<tr>
<td>movsb</td>
<td>S</td>
<td>D</td>
<td>Byte to word</td>
</tr>
<tr>
<td>movsh</td>
<td>S</td>
<td>D</td>
<td>Byte to double word</td>
</tr>
<tr>
<td>movsl</td>
<td>S</td>
<td>D</td>
<td>Word to double word</td>
</tr>
<tr>
<td>pushl</td>
<td>S</td>
<td></td>
<td>Push double word S onto the stack</td>
</tr>
<tr>
<td>popl</td>
<td>D</td>
<td></td>
<td>Pop double word off the stack into D</td>
</tr>
</tbody>
</table>