CSC 252: Machine Level Programming: Instruction Set Architectures

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`

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

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

Assembly Programmer’s View
- 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

CPU
- Registers
- Condition Codes
- 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
- Object Code
- Program Data
- OS Data
- Addressable array
- Code, user data, (some) OS data
- Includes stack used to support procedures

Stack

Compiling Into Assembly
- C

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

Generated 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
  - Addresses (untyped pointers)
  - Data values
  - 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

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

```
int t = x+y;
addl $(%ebp),%eax
```

Similar to expression

```
x += y
```

Object Code

```
0x401040 <sum>:
0x55 push %ebp
0x89 e5 mov %esp,%ebp
0x8b 45 0c mov 0xc(%ebp),%eax
0x03 45 08 add 0x8(%ebp),%eax
0x5d pop %ebp
0xc3 ret
```

Disassembling Object Code

```
Disassembled
0x401040 <_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,%eax
b: 5d pop %ebp
c: c3 ret
d: 8d 76 00 lea 0x0(%esi),%eax
```

- **Assembler**
  - Translates .s into .o

- **Disassembler**
  - 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
Disassembled

Object
0x401040: push %ebp
0x401041: mov %esp,%ebp
0x401043: mov 0xc(%ebp),%eax
0x401046: add 0x8(%ebp),%eax
0x401049: mov %ebp,%esp
0x40104b: pop %ebp
0x40104c: ret

Object
0x40104d: lea 0x0(%esi),%esi

Within gdb Debugger
gdb p
– Disassemble procedure
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 $0x304000c9

– 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 ’$’
    • 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

Source Destination C Analog

%eax
%edx
%ecx
%ebx
%esi
%edi
%esp
%ebp

Imm	Reg	Mem	Reg	Mem	Reg
temp = 0x4;
*p = -147;
temp2 = temp1;
temp = *p;

– Cannot do memory-memory transfers with single instruction
Simple Addressing Modes

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

- Displacement D(R) Mem[Reg[R] + D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
    \[ \text{movl} \ 8(\%\text{ebp}),\%edx \]

Using Simple Addressing Modes

```
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
```

Understanding Swap

<table>
<thead>
<tr>
<th>Register</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>yp</td>
</tr>
<tr>
<td>%edx</td>
<td>xp</td>
</tr>
<tr>
<td>%eax</td>
<td>t1</td>
</tr>
<tr>
<td>%ebx</td>
<td>t0</td>
</tr>
</tbody>
</table>

Stack:

- \%ebp
- \%esp
- \%ebp

Address:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x124</td>
</tr>
<tr>
<td>8</td>
<td>0x120</td>
</tr>
<tr>
<td>4</td>
<td>0x11c</td>
</tr>
<tr>
<td>0</td>
<td>0x118</td>
</tr>
<tr>
<td>-4</td>
<td>0x114</td>
</tr>
<tr>
<td>12</td>
<td>0x110</td>
</tr>
<tr>
<td>8</td>
<td>0x10c</td>
</tr>
<tr>
<td>4</td>
<td>0x108</td>
</tr>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

```
### Understanding Swap

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>0x120</td>
</tr>
<tr>
<td>xp</td>
<td>0x110</td>
</tr>
<tr>
<td>t0</td>
<td>0x10c</td>
</tr>
<tr>
<td>t1</td>
<td>0x108</td>
</tr>
<tr>
<td>eax</td>
<td>0x104</td>
</tr>
<tr>
<td>ebx</td>
<td>0x100</td>
</tr>
</tbody>
</table>

**Example Code:**

```assembly
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>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>0x120</td>
</tr>
<tr>
<td>xp</td>
<td>0x110</td>
</tr>
<tr>
<td>t0</td>
<td>0x10c</td>
</tr>
<tr>
<td>t1</td>
<td>0x108</td>
</tr>
<tr>
<td>eax</td>
<td>0x104</td>
</tr>
<tr>
<td>ebx</td>
<td>0x100</td>
</tr>
</tbody>
</table>

**Example Code:**

```assembly
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

```
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**
  
  \[ D(Rb,Ri,S) \text{ Mem}[\text{Reg}[Rb]+S\times\text{Reg}[Ri]]+D \]
  
  - D: Constant “displacement” 1, 2, or 4 bytes
  - Rb: Base register: Any of 8 integer registers
  - Ri: Index register: Any, except for %esp
  - S: Scale: 1, 2, 4, or 8

- **Special Cases**
  
  \[
  \begin{align*}
  \text{D}(Rb,Ri) & \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]] \\
  \text{D}(Rb,Ri) & \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]]+D \\
  \text{(Rb,Ri,S)} & \quad \text{Mem}[\text{Reg}[Rb]+S\times\text{Reg}[Ri]]
  \end{align*}
  \]

### 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 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, or 8.

Some Arithmetic Operations

Format            Computation
Two Operands Instrucntions  
addl Src, Dest      Dest = Dest + Src     
subl Src, Dest      Dest = Dest - Src     
imull Src, Dest     Dest = Dest * Src     
sall Src, Dest      Dest = Dest << Src Also called shll 
sarl Src, Dest      Dest = Dest >> Src Arithmetic 
shr1 Src, Dest      Dest = Dest >> Src Logical 
xorl Src, Dest      Dest = Dest ^ Src     
orl Src, Dest       Dest = Dest | Src

Some Arithmetic Operations

Format            Computation
One Operand Instructions  
incl Dest           Dest = Dest + 1    
decl Dest           Dest = Dest - 1    
neql Dest           Dest = ~ Dest     
notr Dest           Dest = ~ Dest     

Using leal for Arithmetic Expressions

```
int arith (int x, int y, int z)
{
    int t1 = x+y;
    int t2 = t1+1;
    int t3 = t2+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t3;
    return rval;
}
```

Using leal for Arithmetic Expressions

```
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 %eax, %ebp
popl %ebp
ret
```

`arith: `
### Understanding arith

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

#### Code Generation

```assembly
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)
```

### 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;
}
```

#### Code Generation

```assembly
movl 8(%ebp),%eax # eax = x
movl 12(%ebp),%edx # edx = y
xori 12(%ebp),%eax # eax = x^y (t1)
sarl $17,%eax # eax = t1>>17 (t2)
andl $8185,%eax # eax = t2 & mask
popl %ebp
ret
```

### CISC Properties

- Instruction can reference different operand types
  - 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></td>
<td>C</td>
<td>1) loops</td>
</tr>
<tr>
<td></td>
<td>mem</td>
<td>2) conditionals</td>
</tr>
<tr>
<td></td>
<td>proc</td>
<td>3) switch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4) Proc. call</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5) Proc. return</td>
</tr>
</tbody>
</table>

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:
1) byte
2) 2-byte word
3) branch/jump
4) call
5) address of initial byte

Whose Assembler?

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

- Intel/Microsoft Differs from GAS
  - Operands listed in opposite order
    mov Dest, Src
    movl Src, Dest
  - Constants not preceded by `$`, Denote hex 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)

Disclaimer

- Parts (most in this lecture) of the slides were developed by the course text authors: Dave O’Hallaron and Randy Bryant. The slides are intended for the sole purpose of instruction of computer organization at the University of Rochester. All copyrighted materials belong to their original owner(s).