CS 4400 Computer Systems

LECTURE 5

Machine-level code

Accessing information

Arithmetic and logical operations

Machine-Level Rep of Programs

- High-level programming languages (PL) shield us from machine-level details (expressed in the assembly-code program).
  - exactly how memory is managed
  - functions are called
  - the low-level instructions used to carry out computation

Machine-Level Rep of Programs

- It is the job of a compiler to translate high-level programs to assembly code, so why must we understand it?
  - to analyze underlying inefficiencies in the code
  - to learn about the (hidden) run-time behavior of a program
- We will focus on how C programs are translated to assembly code using the IA32 (aka x86) instruction set.

Machine-Level Code

- Assembly code is very close to machine code.
  - but in a (more readable) text format, instead of binary
- Parts of processor state (hidden from high-level PL):
  - program counter (`%eip`), gives address of next instruction
  - integer registers, 8 named locations for 32-bit values
  - floating-point registers, 8 locations for floating-point data

Machine-Level Code

- A single machine instruction is very simple.
  - such as adding two numbers stored in registers, changing the program counter, etc.

- Compiler first job is to generate sequences of such instructions to implement high-level constructs (e.g., loops).

Machine-Level View of Memory

- Assembly code views memory simply as a large, byte-addressable array.
  - C arrays and structures are contiguous collections of bytes
  - no distinction between signed and unsigned data, pointers and integers, different types of pointers, …

Machine-Level View of Memory

- Program memory contains:
  - object code for the program
  - (some info required by the OS)
  - a run-time stack for managing procedure calls and returns
  - blocks of memory allocated by the user (via `malloc`)

Example: Machine-Level Code

code.c

```
int accum = 0;

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

code.s

```
...
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  addl %eax, accum
  movl %ebp, %esp
  popl %ebp
  ret
...
```

code.o

```
...55 89 e5 8b 45 0c 03
45 08 01 05 00 00 00 00
89 ec 5d c3...
```

- Assembly and object code from the text.
- gcc generates assembly code in "AT&T" format.

```
unix> gcc -S -O2 code.c
unix> ls
      code.c
      code.s
```

```
unix> gcc -c -O2 code.c
unix> ls
      code.c
      code.s
      code.o
```

Data Formats

- An instruction has a suffix denoting the size of the operand. (`movl`: move double word)

- Because Intel started as a 16-bit architecture:
  - "byte" 8-bit data type, instruction suffix is `b` (e.g., `char`)
  - "word" 16-bit data type, instruction suffix is `w` (e.g., `short`)
  - "double word" 32-bit data type, instruction suffix is `l` (e.g., `int`)
  - "quad word" 64-bit data type, instruction suffix is `q` (e.g., `long`)

- Suffix for single-precision floating point is `s`, for double precision suffix is `l`. Confusion with 4-byte integer?

32-Bit Registers

- 8 registers
- store integer data and pointers
- low-order bits may be independently read/written
- conventions for using “general purpose” registers to be covered

to access	double word	word	byte	byte
“general purpose”	`%eax`	`%ax`	`%ah`	`%al`
	`%ecx`	`%cx`	`%ch`	`%cl`
	`%edx`	`%dx`	`%dh`	`%dl`
	`%ebx`	`%bx`	`%bh`	`%bl`
	`%esi`	`%si`		
	`%edi`	`%di`		
stack pointer	`%esp`	`%sp`		
frame pointer	`%ebp`	`%bp`		

Operand Specifiers

- Most instructions have one or more operands.
  - specify source values to reference in performing operation
  - specify destination location into which to place the result
- Source values
  - immediate (constant), e.g., `$-577` or `$0x1F`
  - register, e.g., `%eax` (double-word op) or `%al` (byte op)
  - memory reference, e.g., `7(%eax)` (addr in `%eax+7`)
- Destination locations
  - register
  - memory reference

Addressing Modes

- M[*addr*] denotes the value stored at byte address *addr*.
- R[*reg-id*] denotes the value of the contents of register *reg-id*.

  - `0x2a3`, absolute, M[0x2a3]
  - `(%eax)`, indirect, M[R[`%eax`]]
  - `7(%edx)`, base + displacement, M[7 + R[`%edx`]]
  - `(%eax,%ecx)`, indexed, M[R[`%eax`] + R[`%ecx`]]
  - `7(%eax,%ecx)`, indexed, M[7 + R`[%eax`] + R[`%ecx`]]
  - `(,%eax,4)`, scaled indexed, M[R[`%eax`] * 4]
  - `7(,%eax,4)`, scaled indexed, M[7 + R[`%eax`] * 4]
  - `(%eax,%ecx,4)`, scaled indexed, M[R[`%eax`] + R[`%ecx`] * 4]
  - `7(%eax,%ecx,4)`, scaled indexed, M[7 + R[`%eax`] + R[`%ecx`] * 4]

Exercise: Addressing Modes

address	value
`0x200`	`0x12`
`0x204`	`0x2a`
`0x208`	`0xd4`
`0x20c`	`0xfd`

register	value
`%eax`	`0x200`
`%ecx`	`0x41`
`%edx`	`0x4`

operand	location	value
`(%edx,%ecx,8)`		
`$0x204`		
`%eax`		
`0x1f8(,%edx,4)`		
`(%eax)`		
`0x208`		

Exercise: Addressing Modes

address	value
0x200	0x12
0x204	0x2a
0x208	0xd4
0x20c	0xfd

register	value
%eax	0x200
%ecx	0x41
%edx	0x4

operand	location	value
(%edx,%ecx,8)		
$0x204		
%eax		
0x1f8(,%edx,4)		
(%eax)		
0x208		

0x2a3, absolute, M[0x2a3]
(%eax), indirect, M[R[%eax]]
7(%edx), base + displacement, M[7 + R[%edx]]
(%eax,%ecx), indexed, M[R[%eax] + R[%ecx]]
7(%eax,%ecx), indexed, M[7 + R [%eax] + R[%ecx]]
(,%eax,4), scaled indexed, M[R[%eax] * 4]
7(,%eax,4), scaled indexed, M[7 + R[%eax] * 4]
(%eax,%ecx,4), scaled indexed, M[R[%eax] + R[%ecx] * 4
7(%eax,%ecx,4), scaled indexed, M[7 + R[%eax] + R[%ecx] * 4]

Exercise: Addressing Modes

address	value
0x200	0x12
0x204	0x2a
0x208	0xd4
0x20c	0xfd

register	value
%eax	0x200
%ecx	0x41
%edx	0x4

operand	location	value
(%edx,%ecx,8)	M[0x4+0x41*8]=M[0x20c]	0xfd (253)
$0x204		
%eax		
0x1f8(,%edx,4)		
(%eax)		
0x208		

Exercise: Addressing Modes

address	value
0x200	0x12
0x204	0x2a
0x208	0xd4
0x20c	0xfd

register	value
%eax	0x200
%ecx	0x41
%edx	0x4

operand	location	value
(%edx,%ecx,8)	M[0x4+0x41*8]=M[0x20c]	0xfd (253)
$0x204	—	0x204 (516)
%eax		
0x1f8(,%edx,4)		
(%eax)		
0x208		

Exercise: Addressing Modes

address	value
`0x200`	`0x12`
`0x204`	`0x2a`
`0x208`	`0xd4`
`0x20c`	`0xfd`

register	value
`%eax`	`0x200`
`%ecx`	`0x41`
`%edx`	`0x4`

operand	location	value
`(%edx,%ecx,8)`	`M[0x4+0x41*8]=M[0x20c]`	`0xfd (253)`
`$0x204`	—	`0x204 (516)`
`%eax`	—	`0x200 (512)`
`0x1f8(,%edx,4)`		
`(%eax)`		
`0x208`		

Exercise: Addressing Modes

address	value
`0x200`	`0x12`
`0x204`	`0x2a`
`0x208`	`0xd4`
`0x20c`	`0xfd`

register	value
`%eax`	`0x200`
`%ecx`	`0x41`
`%edx`	`0x4`

operand	location	value
`(%edx,%ecx,8)`	`M[0x4+0x41*8]=M[0x20c]`	`0xfd   (253)`
`$0x204`	—	`0x204  (516)`
`%eax`	—	`0x200  (512)`
`0x1f8(,%edx,4)`	`M[0x1f8+0+0x4*4]=M[0x208]`	`0xd4   (212)`
`(%eax)`		
`0x208`		

Exercise: Addressing Modes

address	value
0x200	0x12
0x204	0x2a
0x208	0xd4
0x20c	0xfd

register	value
%eax	0x200
%ecx	0x41
%edx	0x4

operand	location	value
(%edx,%ecx,8)	M[0x4+0x41*8]=M[0x20c]	0xfd (253)
$0x204	—	0x204 (516)
%eax	—	0x200 (512)
0x1f8(,%edx,4)	M[0x1f8+0+0x4*4]=M[0x208]	0xd4 (212)
(%eax)	M[0x200]	0x12 (18)
0x208		

Exercise: Addressing Modes

address	value
`0x200`	`0x12`
`0x204`	`0x2a`
`0x208`	`0xd4`
`0x20c`	`0xfd`

register	value
`%eax`	`0x200`
`%ecx`	`0x41`
`%edx`	`0x4`

operand	location	value
`(%edx,%ecx,8)`	`M[0x4+0x41*8]=M[0x20c]`	`0xfd (253)`
`$0x204`	—	`0x204 (516)`
`%eax`	—	`0x200 (512)`
`0x1f8(,%edx,4)`	`M[0x1f8+0+0x4*4]=M[0x208]`	`0xd4 (212)`
`(%eax)`	`M[0x200]`	`0x12 (18)`
`0x208`	`M[0x208]`	`0xd4 (212)`

Data Movement Instructions

- Generality of the operand notation allows a simple move instruction to perform many different kinds of moves.

- `movl` src, dest (`movw` for 16-bit, `movb` for 8-bit)

```
– movl $0x4050,%eax ;immediate to register
– movl %ebp,%esp    ;register to register
– movl $-17,(%esp)    ;immediate to memory
– movl %eax,-12(%ebp) ;register to memory
```

- IA32 does not allow both operands to be memory locations.
  - How can we move the contents at address `0xa3` to address `0x7b`?

Question

Is this a valid IA32 instruction?

```
movb $0xF, (%bl)
```

Question

Is this a valid IA32 instruction?

```
movb $0xF, (%bl)
```

- Move Byte
  - From: Immediate value 0xf
  - To: M[%bl]

Question

Is this a valid IA32 instruction?

```
movw %ax, (%esp)
```

Question

Is this a valid IA32 instruction?

```
movw %ax, (%esp)
```

- Move Word (16-bit)
  - From: Word Register %ax
  - To: M[%esp]

Data Movement to/from Stack

- `pushl` and `popl` push data onto and pop data from the run-time stack. (more on the stack next week)
- `%esp` (the stack pointer) holds the address of the top stack element.
- `pushl %ebp` is equivalent to (why?):

```
subl $4,%esp
movl %ebp,(%esp)
```

- `popl %eax` is equivalent to (why?):

```
movl (%esp),%eax
addl $4,%esp
```

Exercise: Data Movement

```
int exchange(int *xp, int y)
{
  int x = *xp;
  *xp = y;
  return x;
}
```

```
movl 8(%ebp),%eax
movl 12(%ebp),%edx
movl (%eax),%ecx
movl %edx,(%eax)
movl %ecx,%eax
```

- What does the exchange function do?
- (Instructions for allocating / deallocating the stack frame are omitted.)
- When procedure begins, xp and y are stored at offsets 8 and 12 from the address in %ebp (the frame pointer).
- What do each of the assembly code instructions do?

Load Effective Address

- `leal` copies the address of the first operand to the destination (the second operand).

- Can be used to generate pointers.

- Can be used to compactly describe arithmetic operations.

Load Effective Address

- `leal 7(%edx,%edx,4),%eax`
  - `%eax=7+%edx+%edx*4`
  - if register `%edx` contains value `x`, then `%eax` is set to `5x+7`

Load Effective Address

- Let `%eax` hold value `x`
- Let `%ecx` hold value `y`

- What value is stored in `%edx` after instruction `leal 0xa(%eax,%ecx,4),%edx` ?

Load Effective Address

- Let `%eax` hold value `x`
- Let `%ecx` hold value `y`

- What value is stored in `%edx` after instruction `leal 0xa(%eax,%ecx,4),%edx` ?

```
0xa + x + y*4
```

Unary Operations

- One operand (register or memory location) serves as the source and destination.
  - `incl %eax`, increment value in register `%eax` by one
  - `decl (%eax)`, decrement value at M[R[`%eax`]] by one
  - `negl 6(%eax)`, negate value at M[6 + R[`%eax`]]
  - `notl (,%eax,4)`, complement value at M[R[`%eax`] * 4]

Binary Operations

- Right operand (register or memory location) serves both as the first source and the destination.
  - Left operand (immed, reg, memory) is the second source.
  - Cannot have both operands as memory locations.

- `subl %eax,(%ecx)`
  - `M[R[%ecx]] = M[R[%ecx]] - R[%eax]`

Binary Operations

- Other examples:
  - `addl` add
  - `imull` multiply
  - `xorl` exclusive or
  - `orl` or
  - `andl` and

Shift Operations

- First operand indicates by how much to shift.
  - immediate between 0 and 31 or in single-byte register `%cl`
- Second operand is value to shift.
- Arithmetic shifts
  - `sall $4,%ecx`, left shift equivalent to `R[%ecx]=R[%ecx]<<4`
  - `sarl`, right shift (fill with copies of MSB)
- Logical Shifts
  - `shll`, left shift, same as `sall`
  - `shrl`, right shift (fill with zeros)

Question

Which of the following is not equivalent to the C function?

```
int subtract(int x, int y) {
    return y - x;
}
```

```
A.  movl 8(%ebp), %ecx
    movl 12(%ebp), %eax
    subl %ecx, %eax

B.  movl 8(%ebp), %eax
    subl 12(%ebp), %eax

C.  movl 12(%ebp), %eax

D.  subl 8(%ebp), %eax
```

E. All are equivalent.

F. More than one are not equivalent.

Exercise: Arithmetic

```
int arith(int x, int y, int z) {
  int t1 = x + y;
  int t2 = z * 48;
  int t3 = t1 & 0xffff;
  int t4 = t2 * t3;
  return t4;
}
```

```
movl 12(%ebp),%eax
movl 16(%ebp),%edx
addl 8(%ebp),%eax
leal (%edx,%edx,2),%edx
sall $4,%edx
andl $65535,%eax
imull %eax,%edx
movl %edx,%eax
```

- The values of `x`, `y` and `z` are stored at offsets 8, 12 and 16 from the address in `%ebp`.
- Which assembly instruction(s) correspond to each C statement?

More Arithmetic Operations

- The two-operand `imull` generates a 32-bit product.
  - `imull src`, gives full 64-bit product of `%eax` and `src`, 2's comp
  - `mull src`, does the same for unsigned
  - both store the result in `%edx` (high order) and `%eax` (low order)

More Arithmetic Operations

- `cltd`, sign extends the value in `%eax` to 64-bit
  - stores the result in `%edx` (high order) and `%eax` (low order)

More Arithmetic Operations

- `idivl src`, takes as a dividend the 64-bit value in `%edx` (high order) and `%eax` (low order), `src` is divisor
  - quotient stored in `%eax` and remainder in `%edx`
  - `divl` is the same except unsigned

Exercise: Data Movement

```
void decode1(int *xp, int *yp, int *zp) {
  // FILL IN


}
```

```
movl 8(%ebp),%edi
movl 12(%ebp),%ebx
movl 16(%ebp),%esi
movl (%edi),%eax
movl (%ebx),%edx
movl (%esi),%ecx
movl %eax,(%ebx)
movl %edx,(%esi)
movl %ecx,(%edi)
```

- The values of `xp`, `yp` and `zp` are stored at offsets 8, 12 and 16 from the address in `%ebp`.
- Fill in `decode1` such that it has an equivalent effect.

Exercise: Shift Operations

```
int shift_left2_rightn(int x, int n) {
  x <<= 2;
  x >>= n;
  return x;
}
```

```
movl 12(%ebp),%ecx ;move n to %ecx
movl 8(%ebp),%eax  ;move x to %eax

??                 ;x<<=2

??                 ;x>>=n
```

- Fill in the above assembly code such that it is generated from the above C function.