

CS 200

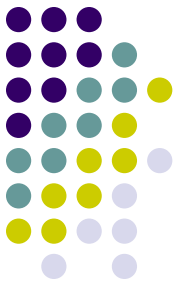
Computer Systems

Programming I

Unit 8

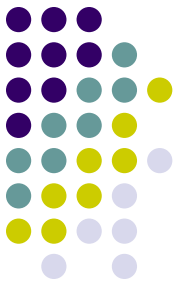
Arithmetic and Logical

Operations



IA32 leal Instruction

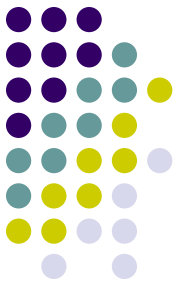
- Load Effective Address (Long)
 - A convenient variant of the movl instruction
 - $\text{leal } S, D \Rightarrow D \leftarrow \&S$
 - Loads the *address* of S in D, not the *contents*
 - Destination must be a register
 - Why would we want an address in a register?



leal Practice

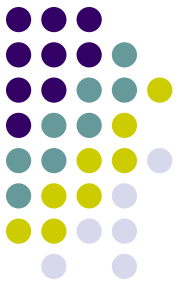
`%eax = x, %ecx = y`

Expression	Result in %edx
<code>leal 6(%eax), %edx</code>	
<code>leal (%eax, %ecx), %edx</code>	
<code>leal (%eax, %ecx, 4), %edx</code>	
<code>leal 7(%eax, %eax, 8), %edx</code>	
<code>leal 0xA(, %ecx, 4), %edx</code>	
<code>leal 9(%eax, %ecx, 2), %edx</code>	
<code>leal (%ecx, %eax, 4), %edx</code>	
<code>leal 1(, %eax, 2), %edx</code>	
<code>leal 0xFF(%ecx, %eax, 3), %edx</code>	



IA32 leal Instruction

- Compilers often use leal for simple math
 - If $\%edx = x$,
 - $\text{leal } 7(\%edx, \%edx, 4) \Rightarrow 5x + 7$
 - Multiply and add all in one instruction
 - Result ends up in edx
 - What if x is a signed integer and we get negative overflow?



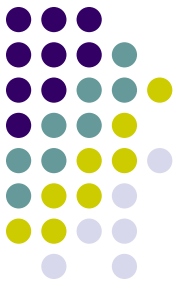
Unary Operations

- Unary $\Rightarrow$ one operand

- inc – increment $\Rightarrow D \leftarrow D + 1$
- dec – decrement $\Rightarrow D \leftarrow D - 1$
- neg – negate $\Rightarrow D \leftarrow -D$
- not – complement $\Rightarrow D \leftarrow \sim D$

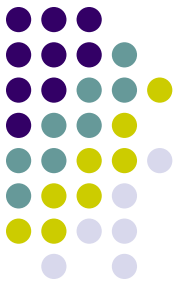
- Examples

- incl (%esp)
 - Increment 32-bit quantity at top of stack
- notl %eax
 - Complement 32-bit quantity in register %eax



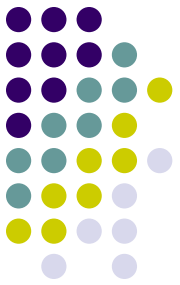
Binary Operations

- A little bit tricky
 - The second operand is used as both a source and destination
 - A bit like C operators '+=', '-=', etc.
- Format
 - $\text{<op> S, D} \Rightarrow D = D \text{ <op> S}$
- Can be confusing
 - $\text{subl S, D} \Rightarrow D = D - S$
 - Not $S - D!!$ Be careful



Binary Operations

- add S, D $\Rightarrow D = D + S$
- sub S, D $\Rightarrow D = D - S$
- imul S, D $\Rightarrow D = D * S$
- xor S, D $\Rightarrow D = D \oplus S$
- or S, D $\Rightarrow D = D | S$
- and S, D $\Rightarrow D = D \& S$

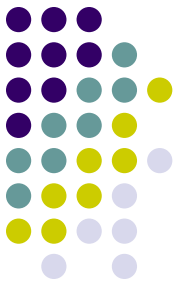


Practice with Binary Ops

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Instruction	Destination	Value
addl %ecx, (%eax)		
subl %edx, 4(%eax)		
imull \$16, (%eax, %edx, 4)		
incl 8(%eax)		
decl %ecx		
subl %edx, %eax		
xorl %eax, 4(%eax)		
addl (\$0x4, %eax), %esp		



Logical Operators in C

Bitwise operators == assembly instructions

- `&` , `|` , `^` , `~`

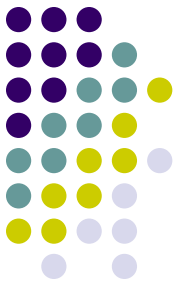
Expression logical operators give “true” or “false” result

- `&&`, `||` , `!`
- False is integer 0, true is integer non-zero

Example

```
if (x & y) {  
    ...  
}
```

The expression in parentheses is true if what?

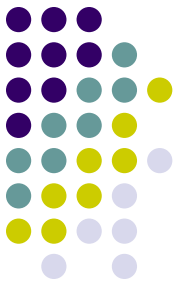


Examples

`int x, y;`

For some processor, independent of the size of an integer, write expressions without any “=” signs that are true if:

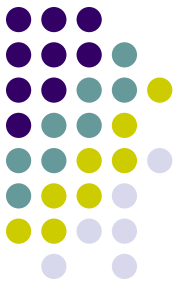
- `x` and `y` have any non-zero bits in common in their low order byte
- `x` has any 1 bits at higher positions than the low order 8 bits
- `x` has no 1 bits at higher positions than the low order 8 bits
- `x` is zero
- `x == y`



Shift Operations

- Not to be confused with their C counterparts!
- Arithmetic and logical shifts are possible
- `<op>` amount value
 - `sal k, D` $\Rightarrow D = D \ll k$
 - `shl k, D` $\Rightarrow D = D \ll k$
 - `sar k, D` $\Rightarrow D = D \gg k$, sign extend
 - `shr k, D` $\Rightarrow D = D \gg k$, zero fill
- Max shift is 32 bits, so `k` is either an immediate byte, or register `%cl`
 - `%cl` is byte 0 of register `%ecx`

Shift Example

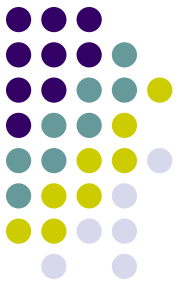


```
int shift_left_rightn(int x, int n)
{
    x <<= 2;
    x >>= n;
    return x;
}
```



```
_shift_left2_rightn:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %eax    ; get x
    movl    12(%ebp), %ecx   ; get n
    popl    %ebp

    _____ ; x <<= 2;
    _____ ; x >>= n;
    ret
```

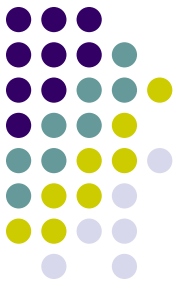


More Practice

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Instruction	Destination	Value
addw %cx, (%eax)		
subb %dl, 4(%eax)		
sarb %cl, (%eax)		
sarl \$2, (%eax)		
shrb %cl, (%eax)		
shll \$8, %edx		



Compiler “Tricks”

- The compiler will try to generate efficient code
 - Resultant assembly code may not readily map to C code, but is functionally the same

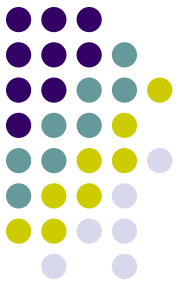
```
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z*48;
    int t3 = t1 & 0xFFFF;
    int t4 = t2 * t3;

    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
```

Things That Make You Go Hmm



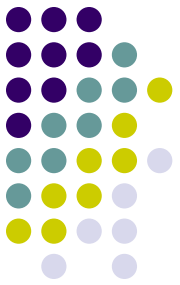
```
int junk(int n)
{
    int i, v=0;

    for (i=0; i < n; i++)
        v += i;

    return v;
}
```

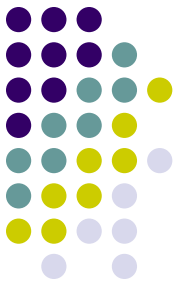


```
_junk:
    pushl   %ebp
    xorl    %eax, %eax
    movl    %esp, %ebp
    movl    8(%ebp), %ecx
    xorl    %edx, %edx
    cmpl    %ecx, %eax
    jge     L8
    .p2align 4,,15
L6:
    addl    %edx, %eax
    incl    %edx
    cmpl    %ecx, %edx
    jl      L6
L8:
    popl    %ebp
    ret
```



Integer Multiply

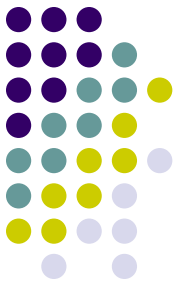
- **`imull <operand>`**
 - One operand is in `eax`
 - Or in `al` or `ax` for the shorter forms
 - Other operand in register or memory
 - 64-bit result in `eax:edx`
- **`mul` and `imul` have many forms**
 - This is truly not a RISC processor!
 - See the programmer's reference manual
 - Which still won't help you with Gnu *gas*



Integer Divide

- **`idivl <operand>`**
 - Has several forms (not as many as multiply)

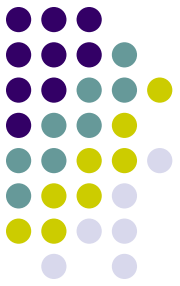
Dividend	Divisor	Quotient	Remainder
AX	rm/8	AL	AH
DX:AX	r/m16	AX	DX
EDX:EAX	r/m32	EAX	EDX



Instruction	Effect
imull S	$R[\%edx]:R[\%eax] \leftarrow S \times R[\%eax]$; signed
mull S	$R[\%edx]:R[\%eax] \leftarrow S \times R[\%eax]$; unsigned
cld	$R[\%edx]:R[\%eax] \leftarrow \text{SignExtend}(R[\%eax])$
idivl S	$R[\%edx] \leftarrow R[\%edx]:R[\%eax] \bmod S$; signed $R[\%eax] \leftarrow R[\%edx]:R[\%eax] \div S$
divl S	$R[\%edx] \leftarrow R[\%edx]:R[\%eax] \bmod S$; unsigned $R[\%eax] \leftarrow R[\%edx]:R[\%eax] \div S$

$R[\%edx]:R[\%eax]$ is viewed as a 64-bit quad word

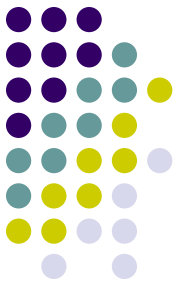
Example



Assume x is at %ebp+8, y at %ebp+12

```
movl 8(%ebp), %eax  
imull 12(%ebp)  
pushl %edx  
pushl %eax
```

```
movl 8(%ebp), %eax  
cld  
idivl 12(%ebp)  
pushl %eax  
pushl %edx
```



Example

- Write an assembly routine that multiplies two 32-bit integers and returns the 64 bit result
 - C prototype:
void product(int *a, int *b)
 - Return high 32 bits in a and low 32 bits in b
- Assume this stack setup

