Operators

Young W. Lim

2020-04-27 Mon

Outline

Based on

"Computer Architecture: A Programmer's Perspective",
Bryant & O'Hallaron

Compling 32-bit program on 64-bit gcc

- gcc -v
- gcc -m32 t.c
- sudo apt-get install gcc-multilib
- sudo apt-get install g++-multilib
- gcc-multilib
- g++-multilib
- gcc -m32
- objdump -m i386

Integer Arithmetic Operations

1. Load Effective Address
2. Inc, Dec, Neg, Complement
3. Add, Sub, Mult, Xor, Or, And
4. Left shift, Arithmetic & Logical right shift

Integer Arithmetic Operations Table (1)

incl	D	$D \leftarrow D + 1$	Increment
decl	D	$D \leftarrow D - 1$	Decrement
negl	D	$D \leftarrow -D$	Negate
not	D	$D \leftarrow \ D$	Complement
addl	S, D	$D \leftarrow D + S$	Add
suml	S, D	$D \leftarrow D - S$	Subtract
imull	S, D	$D \leftarrow D * S$	Multiply
xorl	S, D	$D \leftarrow D \oplus S$	Exclusive OR
orl	S, D	$D \leftarrow D \vert S$	OR
andl	S, D	$D \leftarrow D$	S
AND			
sall	k, D	$D \leftarrow D << k$	Left shift
shll	k, D	$D \leftarrow D << k$	Left shift
sarl	k, D	$D \leftarrow D >> k$	Arithmetic Right shift
shrl	k, D	$D \leftarrow D >> k$	Logical Left shift

Integer Arithmetic Operations Table (2)

`incl D`	D + 1 → D	Increment
`decl D`	D − 1 → D	Decrement
`negl D`	~D → D	Negate
`not D`	~D → D	Complement
`addl S, D`	S + D → D	Add
`suml S, D`	-S + D → D	Subtract
`imull S, D`	S ∗ D → D	Multiply
`xorl S, D`	S ⊕ D → D	Exclusive OR
`orl S, D`	S | D → D	OR
`andl S, D`	S & D → D	AND
`sall k, D`	D << k → D	Left shift
`shll k, D`	D >> k → D	Left shift
`sarl k, D`	D >> k → D	Arithmetic Right shift
`shrl k, D`	D >> k → D	Logical Left shift

1) Load Effective Address (1)

- `leal S, D ; &S -> D`

- lea (Load Effective Address)
- the same form as the instruction
  which reads from memory to a register
- the 1st operand looks like a memory reference
- no actual memory access
- just copy the location (effective address) to the destination
- used to generate pointers

```
leal S, D    ; &S -> D


leal 7(%edx, %edx, 4), %eax

x is the value of %edx
x + 4*x + 7 ->  %eax
```

Unary Operators

- the single operand serves both <u>source</u> and <u>destination</u>
  - a register
  - a memory location
- `incl (%esp)`
  increments the element on the top of the stack

incl	D	$D \leftarrow D + 1$	Increment
decl	D	$D \leftarrow D - 1$	Decrement
negl	D	$D \leftarrow -D$	Negate
not	D	$D \leftarrow \ D$	Complement

Binary Operators

- the second operand serves both source and destination
- the first operand (source)
  - an immediate value
  - a register
  - a memory location
- the second operand (destination)
  - a register
  - a memory location
- `movl` the two operands cannot both be memory locations

Unary & Binary Operators

```
incl (%esp)
subl %eax, %edx

addl %ecx, (%eax)
subl %edx, 4(%eax)
imull $16, (%eax, %edx, 4)
incl 8(%eax)
decl %ecx
subl %edx, %eax
```

Shift Operators (1)

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

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

Shift Operators (2)

- S.A.L (*S*hift *A*rithmetic *L*eft)
- SH.L (*Sh*ift Logical *L*eft)
- S.A.R (*S*hift *A*rithmetic *R*ight)
- SH.R (*Sh*ift Logical *R*ight)

https://www.cs.umb.edu/~bobw/CS341/i386_Assembly/Instructions.pdf

SAL (Shift Arithmetic Left)

- `salb count, dest`
- `salw count, dest`
- `sall count, dest`
- dest $\leftarrow$ dest « count

```
salw $4, %ax       ; count: idata, dest: reg
salb $4, label     ; count: idata, dest: mem
sall %cl, %eax     ; count: %cl,   dest: reg
salw %cl, label    ; count: %cl,   dest: mem
```

https://www.cs.umb.edu/~bobw/CS341/i386_Assembly/Instructions.pdf

SHL (Shift Logical Left)

- `shlb count, dest`
- `shlw count, dest`
- `shll count, dest`

- dest ← dest « count

```
shlw $4, %ax       ; count: idata, dest: reg
shlb $4, label     ; count: idata, dest: mem
shll %cl, %eax     ; count: %cl,   dest: reg
shlw %cl, label    ; count: %cl,   dest: mem
```

https://www.cs.umb.edu/~bobw/CS341/i386_Assembly/Instructions.pdf

SAR (Shift Arithmetic Right)

- `sarb count, dest`
- `sarw count, dest`
- `sarl count, dest`

- dest ← dest » count with sign extension

```
sarw $4, %ax       ; count: idata, dest: reg
sarb $4, label     ; count: idata, dest: mem
sarl %cl, %eax     ; count: %cl,   dest: reg
sarw %cl, label    ; count: %cl,   dest: mem
```

https://www.cs.umb.edu/~bobw/CS341/i386_Assembly/Instructions.pdf

SHR (Shift Logical Right)

- `shrb count, dest`
- `shrw count, dest`
- `shrl count, dest`

- dest $\leftarrow$ dest » count (without sign extension)

```
shrw $4, %ax       ; count: idata, dest: reg
shrb $4, label     ; count: idata, dest: mem
shrl %cl, %eax     ; count: %cl,   dest: reg
shrw %cl, label    ; count: %cl,   dest: mem
```

https://www.cs.umb.edu/~bobw/CS341/i386_Assembly/Instructions.pdf

Arithmetic Routine

```
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 $65536, %eax
imull %eax, %edx
movl %edx, %eax
```

Special Arithmetic Instructions (1)

- two operand `imull`
  - generates a 32-bit product from two 32-bit operands
  - when truncating the product to 32-bits
    both unsigned multiply and 2's complementary multiply
    have the same bit-level behavior
- one operand multiply instructions
  - to compute the full 64-bit product of two 32-bit values
  - one for unsigned (`mull`)
  - one for two's complement (`imull`)

Special Arithmetic Instructions

```
;; Signed Full Multiply
imull S     ;; R[%edx]:R[%eax] <- S x R[%eax]
;; Unsigned Full Multiply
mull  S     ;; R[%edx]:R[%eax] <- S x R[%eax]
;; Convert to quad word
cltd        ;; R[%edx]:R[%eax] <- SignExtend(R[%eax])
;; Signed Divide
idivl S     ;; R[%edx] <- R[%edx]:R[%eax] mod S
            ;; R[%eax] <- R[%edx]:R[%eax] / S
;; Unsigned Divide
divl S      ;; R[%edx] <- R[%edx]:R[%eax] mod S
            ;; R[%eax] <- R[%edx]:R[%eax] / S
:
```

Special Arithmetic Examples

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

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