Bit Operations

Ray Seyfarth

June 29, 2012

64 Bit Intel Assembly Language

©2011 Ray Seyfarth

Outline



1 Introduction to bits









- 6 Shift operations
- Rotate instructions (7)
- 8 Bit testing and setting

- A bit can mean one of a pair of characteristics
- True or false
- Male or female
- Bit fields can represent larger classes
- There are 64 squares on a chess board, 6 bits could specify a position
- The exponent field of a float is bits 30-24 of a double word
- We could use a 4 bit field to store a color from black, red, green, blue, yellow, cyan, purple and white
- Should you store numbers from 0-15 in 4 bits or in a byte?

Bit operations

- Individual bits have values 0 and 1
- There are instructions to perform bit operations
- Using 1 as true and 0 as false
 - ▶ 1 and 1 = 1, or in C, 1 && 1 = 1
 - 1 and 0 = 0, or in C, 1 && 0 = 0
 - ▶ 1 or 0 = 1, or in C, 1 || 0 = 1
- We are interested in operations on more bits
 - ▶ 10101000b & 11110000b = 1010000b
 - ▶ 10101000b | 00001010b = 10101010b
- These are called "bit-wise" operations
- We will not use bit operations on single bits, though we will test individual bits

Not operation

- C uses ! for a logical not
- C uses ~ for a bit-wise not

```
!0 == 1
!1 == 0
~0 == 1
~1 == 0
~10101010b == 01010101b
~0xff00 == 0x00ff
!1000000 == 0
```

- The not instruction flips all the bits of a number one's complement
- not leaves the flags alone
- There is only a single operand which is source and destination
- For memory operands you must include a size prefix
- The sizes are byte, word, dword and qword
- The C operator is

not	rax	;	invert	all bit	s of	rax
not	dword [x]	;	invert	double	word	at x
not	byte [x]	;	invert	a byte	at x	

- C uses & for a logical and
- C uses && for a bit-wise and

```
11001100b & 00001111b == 00001100b
11001100b & 11110000b == 11000000b
0xabcdefab & 0xff == 0xab
0x0123456789abcdef & 0xff00ff00ff00ff00 == 0x010045008900cd00
```

Bit-wise and is a bit selector

And instruction

- The and instruction performs a bit-wise and
- It has 2 operands, a destination and a source
- The source can be an immediate value, a memory location or a register
- The destination can be a register or memory
- Not both destination and source can be memory
- The sign flag and zero flag are set (or cleared)

mov	rax,	0x12345678		
mov	rbx,	rax		
and	rbx,	Oxf	;	rbx has the low nibble 0x8
mov	rdx,	0	;	prepare to divide
mov	rcx,	16	;	by 16
idiv	rcx		;	rax has 0x1234567
and	rax,	Oxf	;	rax has the nibble 0x7

Or operation

- C uses | for a logical and
- C uses || for a bit-wise and

```
11001100b | 00001111b == 11001111b
11001100b | 11110000b == 11111100b
0xabcdefab | 0xff == 0xabcdefff
0x0123456789abcdef | 0xff00ff00ff00 == 0xff23ff67ffabffef
```

• Or is a bit setter

Or instruction

- The or instruction performs a bit-wise or
- It has 2 operands, a destination and a source
- The source can be an immediate value, a memory location or a register
- The destination can be a register or memory
- Not both destination and source can be memory
- The sign flag and zero flag are set (or cleared)

mov	rax,	0x1000					
or	rax,	1	;	make	the	number	odd
or	rax,	0xff00	;	set b	oits	15-8	

Exclusive or operation

C uses ^ for exclusive or

```
00010001b ^ 0000001b == 00010000b
01010101b ^ 11111111b == 10101010b
01110111b ^ 00001111b == 01111000b
0xaaaaaaaa ^ 0xffffffff == 0x55555555
0x12345678 ^ 0x12345678 == 0x00000000
```

Exclusive or is a bit flipper

Exclusive or instruction

- The xor instruction performs a bit-wise exclusive or
- It has 2 operands, a destination and a source
- The source can be an immediate value, a memory location or a register
- The destination can be a register or memory
- Not both destination and source can be memory
- The sign flag and zero flag are set (or cleared)
- mov rax, 0 uses 7 bytes
- xor rax, rax uses 3 bytes
- xor eax, eax uses 2 bytes

mov	rax,	0x1234567812345678		
xor	eax,	eax	;	set rax to O
mov	rax,	0x1234		
xor	rax.	Oxf	:	change to 0x123b

Shift operations

- C uses << for shift left and >> for shift right
- Shifting left introduces low order 0 bits
- Shifting right propagates the sign bit in C for signed integers
- Shifting right introduces 0 bits in C for unsigned integers
- Shifting left is like multiplying by a power of 2
- Shifting right is like dividing by a power of 2

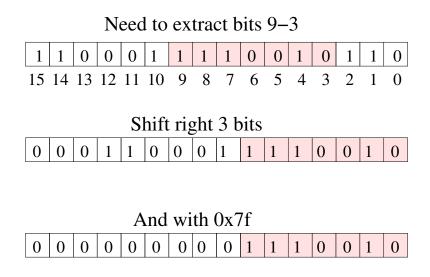
101010b >> 3 == 10b 111111b << 2 == 11111100b 125 << 2 == 500 0xabcd >> 4 == 0xabc

Shift instructions

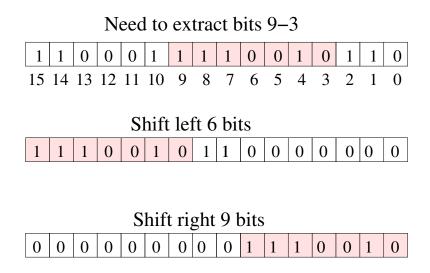
- Shift left: shl
- Shift right: shr
- Shift arithmetic left: sal
- Shift arithmetic right: sar
- shl and sal are the same
- shr introduces 0 bits on the top end
- sar propagates the sign bit
- There are 2 operands
 - A destination register or memory
 - In immediate number of bits to shift or cl
- The sign and zero flags are set (or cleared)
- The carry flag is set to the last bit shifted out

- There are at least 2 ways to extract a bit field
- Shift right followed by an and
 - To extract bits m k with $m \ge k$, shift right k bits
 - And this value with a mask of m k + 1 bits all set to 1
- Shift left and then right
 - Shift left until bit m is the highest bit
 - With 64 bit registers, shift left 63 m bits
 - Shift right to get original bit k in position 0
 - With 64 bit registers, shift right 63 (m k) bits

Extracting a bit field with shift/and



Extracting a bit field with shift/shift



Rotate instructions

- The ror instruction rotates the bits of a register or memory location to the right
- Values from the top end of the value start filling in the low order bits
- The rol instruction rotates left
- Values from the low end start filling in the top bits
- These are 2 operand instructions like the shift instructions
- The first operand is the value to rotate
- The second operand is the number of bits to rotate
- The second operand is either an immediate value or cl
- Assuming 16 bit rotates

1 ror 2 = 0100000000000000 0xabcd rol 4 = 0xbcda 0x4321 ror 4 = 0x1342

Filling a field

- There are at least 2 ways of filling in a field
- You can shift the field and a mask and then use them
 - Working with a 64 bit register, filling bits m k
 - Prepare a mask of m k + 1 bits all 1
 - Shift the new value and the mask left k bits
 - Negate the mask
 - And the old value and the mask
 - On in the new value for the field
- Use rotate and shift instructions and or in new value
 - Rotate the register right k bits
 - Shift the register right m k + 1 bits
 - Rotate the register left m k + 1 bits
 - Or in the new value
 - Rotate the register left k bits

- It takes a few instructions to extract or set bit fields
- The same technique could be used to test or set single bits
- It can be more efficient to use special instructions operating on a single bit
- The bt instruction tests a bit
- bts tests a bit and sets it
- btr tests a bit and resets it (sets to 0)
- These are all 2 operand instructions
- The first operand is a register or memory location
- The second is the bit to work on, either an immediate value or a register

Set operations example code

- rax contains the bit number to work on
- This bit number could exceed 64
- We compute the quad-word of data which holds the bit
- We also compute the bit number within the quad-word

mov	rbx, rax	;	copy bit number to rbx
shr	rbx, 6	;	qword index of data to test
mov	rcx, rax	;	copy bit number to rcx
and	rcx, 0x3f	;	extract rightmost 6 bits
xor	edx, edx	;	set rdx to O
bt	[data+8*rbx],rcx	;	test bit
setc	dl	;	edx equals the tested bit
bts	[data+8*rbx],rcx	;	set the bit, insert into set
btr	[data+8*rbx],rcx	;	clear the bit, remove