

# Bit Shifts

## ICS312 Machine-Level and Systems Programming

Henri Casanova ([henric@hawaii.edu](mailto:henric@hawaii.edu))

# Why bitwise operations?

- Some of the coolest “tricks” in programming rely on bitwise operations, *regardless of the programming language*
  - With only a few instructions one can do a lot very quickly using judicious bit operations
  - Heavily used in fast algorithm implementations (and thus in programming competitions)
- High-level language almost all offer bitwise operators to manipulate bits
- And you can do even more in assembly
- A common bitwise operation: **shifting**

# Shift Operations

- A shift moves the bits around in a value
- A shift can be toward the left (i.e., toward the most significant bits), or toward the right (i.e., toward the least significant bits)



- There are three main kinds of shifts
  - logical shifts
  - arithmetic shifts
  - rotate shifts (typically not directly available in high-level languages)

# Logical Shifts

- Logical shifts are the simplest: bits disappear at one end and **zeros appear at the other**

original byte      

1	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---

left log. shift    

0	1	1	0	1	0	1	0
---	---	---	---	---	---	---	---

left log. shift    

1	1	0	1	0	1	0	0
---	---	---	---	---	---	---	---

left log. shift    

1	0	1	0	1	0	0	0
---	---	---	---	---	---	---	---

right log. shift   

0	1	0	1	0	1	0	0
---	---	---	---	---	---	---	---

right log. shift   

0	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

right log. shift   

0	0	0	1	0	1	0	1
---	---	---	---	---	---	---	---

# Logical Shift Instructions

- Two instructions: **shl** and **shr**
- One must specify by how many bits the data is shifted
  - Either by just passing a constant to the instruction
  - Or by using whatever is stored in the **CL register**
  - **One can shift by at most 31 bits**
- After the instruction has executed, **the carry flag (CF) contains the last bit that was shifted out**
- Example:

```
mov al, 0C6h          ; al = 1100 0110
shl al, 1             ; al = 1000 1100 (8Ch)   CF=1
shr al, 1             ; al = 0100 0110 (46h)   CF=0
shl al, 3             ; al = 0011 0000 (30h)   CF=0
mov cl, 2
shr al, cl            ; al = 0000 1100 (0Ch)   CF=0
```

# Left Shifts to Multiply

- Common use of left shifts: **quickly multiply by powers of 2**
- In decimal:
  - Multiply 13 by 10: add one 0 to the right to get 130
  - Multiply 13 by  $100=10^2$ : add two 0's to the right to get 1300
  - “adding 0's the the right” is a left shift!!
- In binary
  - Multiply 101 by 2: add one 0 to the right to get 1010
  - Multiply 101 by  $4=2^2$ : add two 0's to the right to get 10100
- This is always true mathematically, but in computer arithmetic we have a limited number of bits
  - We may not have enough bits to store the numerical result
- For instance, consider the UNSIGNED 8-bit value 10000000 (128d)
- Multiplying by 2 with a left shift gets us: 00000000, which is not right
  - We cannot encode 256d with 8 bits because  $256 > 2^8-1$
  - It's a kind of overflow

# Right Shifts to Divide

- Common use for right shifts: **quickly divide by powers of 2**
- In decimal, for instance:
  - Divide 1300 by 10 is really one right shift to obtain 0130
  - Divide 1300 by  $100=10^2$  is really two right shifts to obtain 0013
- In binary
  - Divide 1100 by 2 is really a right shift to obtain 0110
  - Divide 1100 by  $4=2^2$  is really two right shifts to obtain 0011
- When dividing odd numbers by 2 we “lose bits”, which ends up rounding to the integer division’s quotient (the remainder ends up in the carry flag!)
- For example:
  - Consider the binary number 10011 (19d)
  - One right shift: 01001 (9d: 19/2 rounded below)
  - Another right shift: 00100 (4d: 9/2 rounded below)

# What about Signed Numbers?

- In the previous slides, we have implicitly assumed unsigned numbers
- Things are not as simple for signed numbers
  - Are they ever?
- When numbers are signed, the shifts may not handle the *sign* bit correctly
  - i.e., the number may change sign due to the shift, which produces a numerically incoherent result
- Let's see this on two examples...

# What about Signed Numbers?(2)

- Left shift example: Consider the 1-byte number 63h
  - If unsigned:
    - $63h = 99d = 01100011b$
    - left shift:  $11000110b = C6h = 198d$  (which is  $99 * 2$ )
  - In signed:
    - $63h = 99d = 01100011b$
    - left shift:  $11000110b = C6h = -58d$  (which is **NOT**  $99 * 2$ )
- Right shift example: Consider the 1-byte number FE
  - If unsigned:
    - $FEh = 254d = 11111110b$
    - right shift:  $01111111b = 7Fh = 127d$  (which is  $254/2$ )
  - In signed:
    - $FEh = -2d = 11111110b$
    - right shift:  $01111111b = 7Fh = +127d$  (which is **NOT**  $-2/2$ )

# Arithmetic Shifts

- Since the logical shifts do not always work for signed numbers, we have another kind of shifts called arithmetic shifts
- Left arithmetic shift: **sal**
  - This instruction works just like shl
    - We just have another name for it so that in the program we “see/remember” that we’re dealing with signed numbers
  - **As long as the sign bit is not changed by the shift, the result will be correct** (i.e., will be multiplied by 2)
- Right arithmetic shift: **sar**
  - This instruction does NOT shift the sign bit: the new bits entering on the left are **copies of the sign bit**
- Both shifts store the last bit out in the carry flag

# Arithmetic Shift Example

- If **signed numbers**, then the operations below are correct multiplications / divisions of 1-byte quantities

```
mov    al, 0C3h      ; al = 1100 0011  (-61d)
sal    al, 1         ; al = 1000 0110  (86h = -122d)
sar    al, 3         ; al = 1111 0000  (F0h = -16d)
; (note that this is not an exact division as we
; lose bits on the right... rounding occurs)
```

- Let's say that now I want to multiply the value above by 16
- I cannot do a left shift by 4 =  $\log_2(16)$  bits: I'd get 0 because I'd lose all bits
- Instead, I should use the `imul` instruction instead (but unfortunately `imul` doesn't work on 1-byte quantities):

```
movsx  ax, al       ; sign extension!
imul   ax, 16       ; result in ax
```

- Let's see/run this example in file `ics312_arithmetic_shift.asm`

# In-Class Exercise

- Consider the following instructions

```
mov    ax, 0F471h
```

```
sar    ax, 3
```

```
shl    ax, 7
```

```
sar    ax, 10
```

- At each step, give the content of register ax (in hex and binary) and the value of CF (assuming that initially it is equal to 0)
  - Remember: CF contains the last bit that was shifted out

# In-Class Exercise

```
mov    ax, 0F471h
      ax = 1111 0100 0111 0001
      ax = F471h                                CF=?
                                              (whatever it was before)

sar    ax, 3
      ax = 1111 1110 1000 1110
      ax = FE8Eh.                                CF=0

shl    ax, 7
      ax = 0100 0111 0000 0000
      ax = 4700h                                CF=1

sar    ax, 10
      ax = 0000 0000 0001 0001
      ax = 0011h                                CF=1
```

# Rotate Shifts

- **rol** and **ror**: circular left and right shifts
  - bits shifted out on one end are shifted in the other end
  - We'll use those...
- **rcl** and **rcr**: carry flag rotates
  - the source (e.g., a 16-bit register) and the carry flag are rotated as one quantity (e.g., as a 17-bit quantity)
  - We likely won't use those
- See the book (Section 3.1.4) for more detailed descriptions and examples

# Example Using Shifts

- Let's do Example 3.1.5 in the book: say you want to count the number of bits that are equal to 1 in register EAX
  - This is useful because often we encode sets as bits (e.g., a bit set to 1 means an element is present)
  - Which saves a lot of space when compared to, say, an array of booleans
- One easy way to do this is to use shifts
  - Shift 32 times (left or right, doesn't matter)
  - Each time the carry flag contains the last shifted bit
  - If the carry flag is 1, then increment a counter, otherwise do not increment a counter
- Let's write this in x86 assembly live right now...
  - The textbook uses the loop instruction, so let's write it without it
  - On the next slide is something that should look a lot like what we're about to implement...

# Example Using Shifts

```
; Counting 1 bits in EAX
mov  bl, 0      ; bl: the number of 1 bits
mov  cl, 32     ; cl: the loop counter
loop_start:
    shl  eax, 1  ; left shift
    jnc  not_one ; if carry != 1, jump to not_one
    inc  bl      ; increment the number of 1 bits
not_one:
    dec  cl      ; decrement the loop counter
    jnz  loop_start ; if more iterations then
                    ; goto loop_start
```

# The same, with the `adc` instruction

- Convenient instruction: `adc` (add carry)
  - `adc dest, src` ; `dest += src + cf`

```
; Counting 1 bits in EAX
    mov  bl, 0    ; bl:the number of 1 bits
    mov  cl, 32  ; cl: loop counter
loop_start:
    shl  eax, 1  ; left shift
    adc  bl, 0   ; add the carry to bl
    dec  cl      ; decrement the loop counter
    jnz  loop_start ; if more iterations then
                    ; goto loop_start
```

# The same, with the loop instruction

- Remember the **loop** instruction

- `loop <label>` ; decrements loop index (in ecx)  
; and branches if ecx isn't 0

`; Counting 1 bits in EAX`

```
mov    bl, 0      ; bl: the number of 1 bits
```

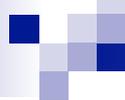
```
mov    ecx, 32   ; ecx: the loop counter
```

```
loop_start:
```

```
shl    eax, 1    ; left shift
```

```
adc    bl, 0     ; add the carry to bl
```

```
loop  loop_start ; decrement ecx and  
; then loop if needed
```



# **Making it more efficient?**

- Anybody has an idea about how to make this code more efficient?

# Making it more efficient?

- Anybody has an idea about how to make this code more efficient?
- Let's consider an example:
  - Say that EAX = 00 00 00 04 h (a single 1 bit)
  - Yet, we iterate 32 times... doesn't that seem like a waste?
  - So, anybody has any thoughts on how to avoid iterating 32 times?

# Making it more efficient?

- Anybody has an idea about how to make this code more efficient?
- Let's consider an example:
  - Say that `EAX = 40 00 00 00 h` (a single 1 bit)
  - Yet, we iterate 32 times... doesn't that seem like a waste?
  - So, anybody has any thoughts on how to avoid iterating 32 times?
- Simple idea: stop when `EAX == 0`

# Likely the best version

```
; Counting 1 bits in EAX
    mov     bl, 0      ; bl: the number of 1 bits
loop_start:
    shl    eax, 1     ; left shift
    adc    bl, 0      ; add the carry to bl
    cmp    eax, 0
    jnz    loop_start ; loop back if eax != 0
```

The code isn't shorter, but **we don't waste any register to store a loop counter, as this is now a **while** loop!**

(this idea could of course be used in high-level code as well)

Could be improved further (should do no work if EAX = 0)

# The POPCNT instruction

- It turns out that x86 has an instruction to do the bit counting
  - `popcnt eax, ebx`
    - EAX is set to the number of bits sets to one in EBX
- The x86 designers deemed this operation useful enough that they implemented it in hardware
  - Introduced circa 2008, so relatively recently
- And it can be accessed in high-level languages via compiler “intrinsics”
  - i.e., built-in functions that are not part of the language but that will call the assembly instruction for you
  - Not portable

# Important Takeaways

- Three kinds of shifts:
  - Logical shifts: always shift in zeros
  - Arithmetic shifts: when to the right, will shift in replicas of the sign bit
  - Rotate shifts: treats the bits as in a circular buffer
- Right shifts divide, left shifts multiply
  - By powers of 2 only (of course)
  - Right logical shifts for unsigned division, right arithmetic shifts for signed division

# Conclusion

- Bitwise operations are useful in general
  - Can be the bread-and-butter of the clever assembly/C/Java/Python/\* programmer
    - See the FIRST programming question in Cracking the Coding Interview!!! (we'll do it in class)
  - Is often the only way to do things efficiently for some problems (as seen in many programming competitions!)
  - And is needed when dealing with data that's encoded in binary (e.g., images, sound, video)
- A few practice problems are posted
  - Let's briefly look at them, and try to do the mystery programs...
- In the next set of lecture notes we'll talk about other bitwise operations and the use of bitmasks