Bit Shifts

ICS312
Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Why bit operations

- Assembly languages all provide ways to manipulate individual bits in multi-byte values
- Some of the coolest “tricks” in assembly rely on bit operations
  - With only a few instructions one can do a lot very quickly using judicious bit operations
  - And you can do them in almost all high-level programming languages!
- Let’s look at some of the common operations, starting with shifts
  - logical shifts
  - arithmetic shifts
  - rotate shifts

Shift Operations

- A shift moves the bits around in some data
- 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 two kinds of shifts:
  - Logical Shifts
  - Arithmetic Shifts

Logical Shifts

- The simplest shifts: 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 specifies 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
- After the instruction executes, 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
```

Shifts and Numbers

- The common use for shifts: quickly multiply and divide by powers of 2
- In decimal, for instance:
  - multiplying 0013 by 10 amounts to doing one left shift to obtain 0130
  - multiplying by $100=10^2$ amounts to doing two left shifts to obtain 1300
- In binary
  - multiplying by 00101 by 2 amounts to doing a left shift to obtain 01010
  - multiplying by $4=2^2$ amounts to doing two left shifts to obtain 10100
- If numbers are too large, then we'd need more bits and multiplication doesn't produce valid results
  - e.g., 10000000 (128d) cannot be left-shifted to obtain 256 using 8-bit values
- Similarly, dividing by powers of two amounts to doing right shifts:
  - right shifting 10010 (18d) leads to 01001 (9d)
- Note that when dividing odd numbers by two we "lose bits", which amounts to rounding to the lower integer quotient
  - Consider number 10011 (19d)
  - Right shift: 01001 (9d: 19/2 rounded below)
  - Right shift: 00100 (4d: 9/2 rounded below)

Shifts and Unsigned Numbers

- Using shifts works only for unsigned numbers
- When numbers are signed, the shifts do not handle the sign bits correctly and cannot be interpreted as multiplying/dividing by powers of 2 anymore
- Example: Consider the 1-byte number FE
  - If Unsigned:
    - FE = 254d = 11111110b
    - right shift: 01111111b = 7Fh = 127d (which is 254/2)
  - In Signed:
    - FE = - 2d = 11111110b
    - right shift: 0111111b = 7Fh = +127d (which is NOT -2/2)

Arithmetic Shifts

- Since the logical shifts do not 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 code we "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!)
```

- The following is not a correct multiplication by 16!

```
sal     al, 4           ; al = 0000 0000  (0d, which can't be right)
```

- One 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

Rotate Shifts

- There are more esoteric shift instructions
- rol and ror: circular left and right shifts
  - bits shifted out on one end are shifted in the other end
- 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)
- See the book (Section 3.1.4) for more detailed descriptions and examples

Conclusion

- In the next set of lecture notes we'll talk about bit-wise operations and the use of bitmasks
- This is useful in general, and not only in assembly
  - Can be the bread-and-butter of the clever assembly/C/Java/Python/* programmer