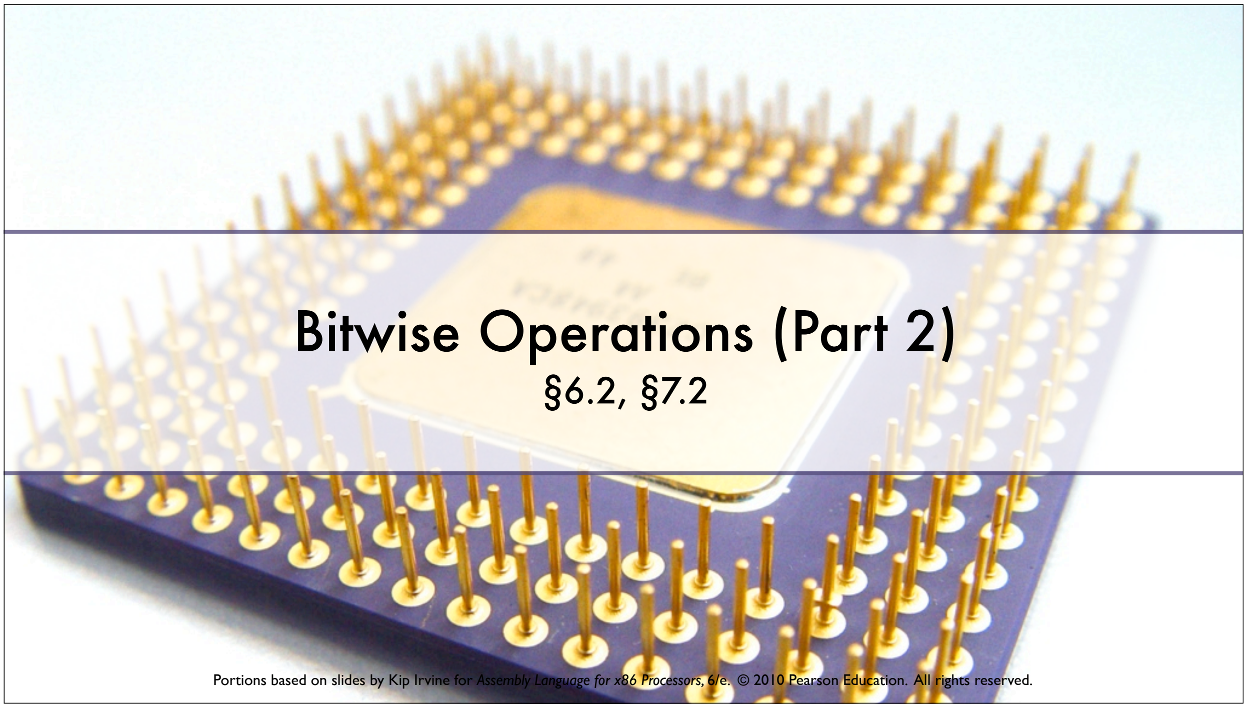


Bitwise Operations (Part 2)

§6.2, §7.2

Administrivia

- **Exam 2** Wednesday, November 5
  - Make-up exams must be scheduled **before** the exam is given in class; no make-ups afterward
- **Homework 5** out today
- Reading on procedures (no reading questions):

6/e	7/e	Title
§5.4	§5.1	Stack Operations
§5.5	§5.2	Defining and Using Procedures
§8.2	§8.2	Stack Frames
§8.3	§8.3	Recursion – not covered in lecture

(Review) TEST: Is a bit set?

- Recall: a bit mask can be used with a bitwise AND to determine whether bits are set
  - Example: Is bit 0 or 1 set (or both)?
  - ```
      1010   < Original number
    & 0011   < Bit mask
      0010
    ```
  - Result is nonzero ⇒ at least one of those bits was set
- The TEST instruction sets flags the same as a bitwise AND
  - TEST against a mask; then, zero flag will be clear if the bit(s) were set
  - Typically followed by JZ/JNZ

Topics Covered in Notes:

- TEST instruction
    ```

AND Masks – Clearing Bits

- Notice what happens to each bit when you apply an AND mask...
  - ```
      1010   < Original number
    & 0011   < Bit mask
      0010
    ```
  - If there is a 0 bit in the mask, the corresponding bit is cleared
  - If there is a 1 bit in the mask, the corresponding bit is retained
- Bitwise **AND** can be used to **clear** particular bits
  - ```
      101101011110011001110110   < Original number
    & 111111110000000011111111   < Bit mask
      101101010000000001110110
    ```

OR Masks – Setting Bits

- A bit mask can be used with **OR** to **set** particular bits
  - ```
      1010   < Original number
    | 0011   < Bit mask
      1011
    ```
  - If there is a 0 bit in the mask, the corresponding bit is retained
  - If there is a 1 bit in the mask, the corresponding bit is set
  - ```
      101101011110011001110110   < Original number
    | 000000001111111100000000   < Bit mask
      101101011111111101110110
    ```

XOR Masks – Flipping Bits

- A bit mask can be used with **XOR** to **flip** particular bits
  - `1010` < *Original number*
    ⊕ `0011` < *Bit mask*
    `1001`
  - If there is a 0 bit in the mask, the corresponding bit is retained
  - If there is a 1 bit in the mask, the corresponding bit is flipped
  - `101101011110011001110110` < *Original number*
    ⊕ `0000000011111111000000000` < *Bit mask*
    `1011010100011001011110110`

Expressions with AND, OR, XOR

- Sometimes we will find it helpful to write mathematical expressions using bitwise operators
  - Example: Given a 4-bit integer *n,* give an expression that is equal to *n* with bit 0 cleared and bit 3 set
    - *n* with bit 0 cleared is: *n* & 1110
    - *n'* with bit 3 set is: *n'* | 1000
    - So, *n* with bit 0 cleared and bit 3 set is given by both of these:

      (*n* & 1110) | 1000 (n | 1000) & 1110
    - Implementation in assembly language is straightforward:

```
; Suppose n is in AL
and al, 1110b
or al, 1000b
; Result is in AL
```

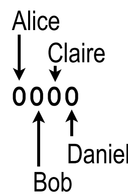
Expressions with AND, OR, XOR

- Sometimes we will find it helpful to write mathematical expressions using bitwise operators
  - Example: Given an 8-bit unsigned integer, give an expression that evaluates to the largest even number not greater than *n*
    - I.e., 0 ↦ 0, 1 ↦ 0, 2 ↦ 2, 3 ↦ 2, 4 ↦ 4, 5 ↦ 4, 6 ↦ 6, 7 ↦ 6, 8 ↦ 8, ...
    - $0 = 00000000_2$ $\quad$ $1 = 00000001_2$
$2 = 00000010_2$ $\quad$ $3 = 00000011_2$
$4 = 00000100_2$ $\quad$ $5 = 00000101_2$
$6 = 00000110_2$ $\quad$ $7 = 00000111_2$
$8 = 00001000_2$ $\quad$ $9 = 00001001_2$
    - Even numbers: bit 0 clear $\quad$ Odd numbers: bit 0 set
      - Every odd number is equal to the previous even number + 1
    - So, the solution is to clear bit 0: the expression is *n* & 11111110

Activity 13 #1

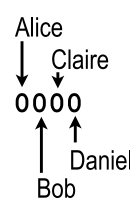
Application: Bit Sets

- Bit strings can represent sets
- Each bit represents one element
- The bit is 1 if that element is present
- The bit is 0 if that element is absent



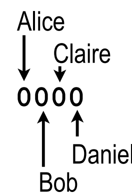
Application: Bit Sets

- **Combine the elements in two sets using | (*set union*)**
  - 1010 denotes { Alice, Claire }; 0110 denotes { Bob, Claire }
  - 1010 | 0110 = 1110, which denotes { Alice, Bob, Claire }
  - In set theory notation:
    { Alice, Claire } ∪ { Bob, Claire } = { Alice, Bob, Claire }



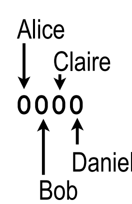
Application: Bit Sets

- **Find common elements using & (*intersection*)**
  - 1010 denotes { Alice, Claire }; 0110 denotes { Bob, Claire }
  - 1010 & 0110 = 0010, which denotes { Claire }
  - In set theory notation,
    { Alice, Claire } ∩ { Bob, Claire } = { Claire }



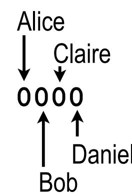
Application: Bit Sets

- **A set is empty iff the bits are all zeroes**
  - 0000 denotes { }
- **Do two sets have any elements in common?**
  - Test whether their intersection is nonempty
  - In set theory notation, determine whether $S \cap T \neq \varnothing$
  - Use a bitwise AND, then determine if the result is nonzero
    - 1010 denotes { Alice, Claire }; 0110 denotes { Bob, Claire }
    - 1010 & 0110 = 0010, which denotes { Claire }
    - Since 0010 is nonzero, the two sets have elements in common



Application: Bit Sets

- **Test for membership using &**
  - Bob is represented by bit 2
  - So, test if bit 2 is set
    - 1110 denotes { Alice, Bob, Claire }
    - 1110 & 0100 = 0100, which is nonzero
    - So, Bob *is* in { Alice, Bob, Claire }
    - `test al, 0100b` ; Bit set in AL – is Bob in it?
      `jnz some_label` ; Jump if Bob was in the set



- Note that the expression (1110 & 0100) above is also used to compute the intersection of { Alice, Bob, Claire } and { Bob }
  - So, testing whether Bob is a member of { Alice, Bob, Claire } is the same as testing if { Alice, Bob, Claire } $\cap$ { Bob } $\neq \varnothing$

Application: Bit Sets

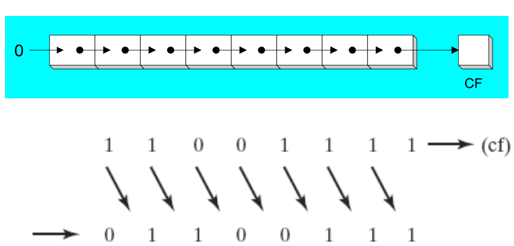
- Bit sets are good for representing sets where
  - there is a **finite** universe of elements
    - i.e., every element that could be in the set is known ahead of time
  - and the sets are **small** (very few possible elements) or **dense** (many of the possible elements will be in the set)
  - E.g., there are 24 students in this class; can represent any set of these students with 24 bits!
- To represent sets from an infinite universe, or sets that are large and sparse, do not use bit sets: use, e.g., hash tables or binary trees
  - E.g., there are 7.076 billion people in the world – more are being added all the time (so the universe of elements is not fixed), and anyway a bit set that large would occupy 844 MB of memory
- In the Java library: BitSet, HashSet, TreeSet

Bit Shifting

- The bits in a number can be *shifted* left or right
- When writing formulas, we will denote this by
  - $a \ll n$ Left shift by n bits
  - $a \gg^{u} n$ Right shift by n bits with 0-fill (logical shift)
  - $a \gg^{s} n$ Right shift by n bits with sign-fill (arithmetic shift)
- Examples (8-bit numbers):
  - $10111001 \ll 3 = 11001000$
  - $10111001 \gg^{u} 2 = 00101110$
  - $10111001 \gg^{s} 2 = 11101110$
  - $00111001 \gg^{s} 2 = 00001110$

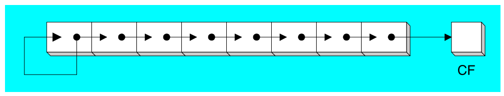
Logical Shift

- A *logical* shift fills the newly created bit position with zero:



Arithmetic Shift

- An *arithmetic* right shift fills the newly created bit position with a copy of the number's sign bit:



- An arithmetic left shift is the same as a logical left shift!
  (nothing special about the sign bit when shifting left)

SHL Instruction

- The SHL (shift left) instruction performs a logical left shift on the destination operand, filling the lowest bit with 0.

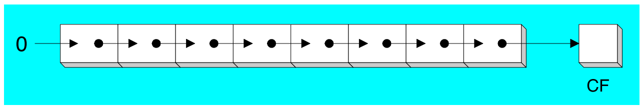
Operand types for SHL:

```
SHL reg,imm8
SHL mem,imm8
SHL reg,CL
SHL mem,CL
```

(Same for all shift and rotate instructions)

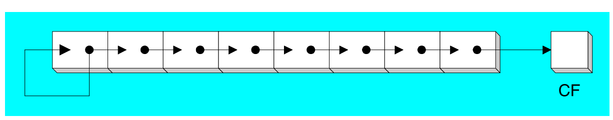
SHR Instruction

- The SHR (shift right) instruction performs a logical right shift on the destination operand. The highest bit position is filled with a zero.



SAL and SAR Instructions

- SAL (shift arithmetic left) is identical to SHL.
- SAR (shift arithmetic right) performs a right arithmetic shift on the destination operand.



An arithmetic right shift preserves the number's sign.

```
mov dl,-80
sar dl,1                        ; DL = -40
sar dl,2                        ; DL = -10
```

Application: Powers of 2

- What happens when you left shift the number 1?
- $1 \ll n$ has a 1 in bit position n and zeros elsewhere
  - $1 \ll 0 \quad = \quad 00000001_2 \quad = \quad 1$
  - $1 \ll 1 \quad = \quad 00000010_2 \quad = \quad 2$
  - $1 \ll 2 \quad = \quad 00000100_2 \quad = \quad 4$
  - $1 \ll 3 \quad = \quad 00001000_2 \quad = \quad 8$
  - $1 \ll 4 \quad = \quad 00010000_2 \quad = \quad 16$
  - $1 \ll 5 \quad = \quad 00100000_2 \quad = \quad 32$
  - $1 \ll 6 \quad = \quad 01000000_2 \quad = \quad 64$
  - $1 \ll 7 \quad = \quad 10000000_2 \quad = \quad 128$
- So, to compute powers of 2: $\mathbf{1 \ll \boldsymbol{n} = 2^{\boldsymbol{n}}}$

Activity 13 #2

Application: Shifting the Sign Bit

- Example: Given an SDWORD *n,* give an expression that is equal to 1 if *n* is negative and 0 otherwise
  - *n* is negative iff the sign bit is set
  - So, shift the sign bit into the ones' position
  - The expression is: $\boldsymbol{n \gg^{u} 31}$
- Example: Given an SDWORD *n,* give an expression that is equal to –1 if *n* is negative and 0 otherwise
  - Recall that –1 is represented by all one bits
  - The expression is: $\boldsymbol{n \gg^{s} 31}$

Activity 13 #3