Bitstring Operations

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Bitstring shifts can correspond to multiplication and division by two.
- Bitwise operations with masks let us manipulate particular bits of a bitstring.

B. Outcomes

After this lecture, you should

- Know the common bitstring operations and be able to perform them.

Note: From now on, we're using 2's complement unless said otherwise.

C. Sections of Bitstrings:

- Text numbers the bits 0, 1, … starting from the right.
- Can specify individual bits using index in square brackets; can select ranges using index : index in square brackets
- **Example**: If $X = 011$, then $X[0] = X[1] = 1$, $X[2] = 0$, and $X[0{:}1] = 11$, $X[1{:}2] = 01$, $X[0{:}2] = X = 011$.

D. Left Bit Shifting

- If X is a bitstring of length n, then **left-shifting** X one bit to get a result Y involves
  - $Y[n\text{-}1{:}1] \leftarrow X[n\text{-}2{:}0]$ and $Y[0] \leftarrow$ something
    - For **left shift with zero fill**, $Y[0] \leftarrow 0$
    - For **left circular shift**, $Y[0] \leftarrow X[n-1]$. (The leftmost bit of the operand becomes the rightmost bit of the result.)
- C and Java support left shift with zero fill via the **<<** operator: X **<<** i shifts X left i bits with zero fill.

- **Examples**:
  - If $i = 1, 2, 3, \ldots$, then *00011* **<<** i yields *00110* then *01100*, then *11000*, then *10000*, then *00000*, then *00000*, ….[1]
  - Repeatedly left circular-shifting *00011* one bit yields *00110*, *01100*, *11000*, *10001* (note the difference w.r.t. shifting with zero fill), *00011*, ….
- **Shift left** k bits with zero fill corresponds to multiplying by 2^k.
- For unsigned integers, if a 1 is "shifted out" of the number, then overflow has occurred.
  - **Example**: If we treat *00011* (= *3*) as an unsigned 5-bit integer, then repeated left shifting yields *00110* (= *6*), *01100* (= *12*), *11000* (= *24*), *10000* (= *16*, with overflow), and *00000* (= *0*, with overflow).
- For signed integers, if the new sign bit ≠ the old sign bit, then overflow has occurred.
  - **Example**: If we treat *00011* (= *3*) as a signed 5-bit number, then repeated left shifting yields *00110* (= *6*), then *01100* (= *12*), then *11000* (= *–8*, with overflow), then *10000* (= *–1 6*). Note that left-shifting *-8* produced *-16*, which is indeed -8×2.
  - Note that the sequence of bitstrings is the same in the two examples; it's our interpretation of the bitstrings that's different.

E. Right Bit Shifting

- **Right-shifting** *X[0:n]* one bit to yield *Y* involves
  - *Y[n-2:0]* ← *Y[n-1:1]* and *Y[n–1]* ← something
  - **Right shift with zero fill** (a.k.a. **logical right shift**), *Y[n–1]* ← 0.
  - **Right circular shift**, *Y[n–1]* ← *X[0]*
  - **Right shift with sign fill** (a.k.a. **arithmetic right shift**), *Y[n–1]* ← *X[n–1]*.

[1] I'm mixing notations here — you can't write `00011 << i` in C or Java; the closest you can get is `3 << i` (and get 32 bits) or `(char) (3 << i)` (and get 8 bits)

- If X represents an unsigned integer or a nonnegative signed integer, then logical and arithmetic shift of k bits are the same and correspond to division by 2^k.
  - **Example**: Logical and arithmetic right-shifting *01111* (= *15*) both yield *00111* (= *7*), *00011* (= *3*), *00001* (= *1*), *00000*, *00000*, ….
- If X represents a negative integer, then only arithmetic right shift corresponds to division because it fills in the sign bit of the result with a 1 (a copy of the sign bit of the value being shifted).
  - **Example**: Arithmetic right-shifting *10011* (= *–13*) yields *11001* (= *–7*), *11100* (= *–4*), *11110* (= *–2*), *11111* (= *–1*), *11111*, ….
  - **Note**: For this to work as division, you have to consider the remainder to be *0* or *1* (not *0* or *–1*). E.g., *–13* = *–14* + *1*, so *–13/2* = *–14/2* = *–7*. (This isn't how the / and % operators work in C and Java.)

- **Bit Shifting in C and Java**
  - Both C and Java support left shift using the `<<` operator; the bitstring goes on the left, and the number of places to shift goes on the right.
    - **Example**: Let $i = 1, 2, 3, \ldots$, then *15* `<<` *i* is *30*, *60*, *90*, *180*, ….
- For right shift, C is weird: The C standard says that whether `>>` stands for arithmetic or logical right shift is up to the implementation.
- In Java, logical right shift is the `>>>` operator and arithmetic right shift is the `>>` operator.
  - **Example**: In Java, `-13 >> 1` is `-7` (i.e., `0xfffffff3 >> 1` is `0xfffffff9`), and repeating the right shift yields `-3` (= `0xfffffffc`), `-1` (= `0xffffffff`), `-1` (= `0xffffffff`), ….
  - **Example**: In Java, `-13 >>> 1` is the same as `0xfffffff3 >>> 1` (substituting the bitstring for `-13`), and `0xfffffff3 >>> 1` is `0x7ffffff9`. Repeating the shift gives us `0x3ffffffc`, `0x1ffffffe`, `0x0fffffff`, `0x07ffffff`, ….
  - (To get the signed equivalents of these bitstrings, take `0x7ffffff9` = $(80000000_{16} - 7) = 8 \times 16^7 - 7 = 2^{31} - 7 = 2147483648 - 7 = 2147483641$.

Repeated right shifting does division: We get 1073741820, 536870910, 268435455, 134217727, ….)

F. Bitwise operations

- With bitwise operations, we extend operations on a single bit to each bit of a string or to corresponding bits of two strings.
- C and Java support bitwise *NOT*, *AND*, *OR*, and *XOR* as `~` (tilde), `&`, `|`, and `^` (circumflex). (Contrast `-` (arithmetic negative), `&&`, and `||` (logical *AND* and *OR*).)
- **Bitwise** *NOT*: Perform *NOT* on each bit of a string: $Y = \sim X$ means each *Y[i]* = ¬ *X[i]*. (Here, *Y[i]* means the i'th bit of *Z*, not an array indexing expression.)
  - **Example**: bitwise *NOT* of *101101 = 010010*.
  - Bitwise *NOT* is the same as taking the 1's complement of an integer.
- Given two bitstrings, we can perform a binary operation on each pair of corresponding bits: *Z[0:n−1] = X[0:n−1]* bitwise *AND Y[0:n−1]* means each *Z[i] = X[i] AND Y[i]*. Bitwise *OR* and *XOR* are similar.
  - **Bitwise** *AND* of *101100* and *100001* is *100000*.
  - **Bitwise** *OR* of *101100* and *100001* is *101101*.
  - **Bitwise** *XOR* of *101100* and *100001* is *001101*.
- **Using Bitwise Operations to Test and Manipulate Bitstrings**
  - Bitwise *AND* and *OR* are used in conjunction with "bit masks" to select and modify parts of a bitstring. A **bit mask** is a bitstring of some specific pattern used to help carry out an operation.
  - The bitwise operations rely on the following properties of *AND*, *OR*, and *XOR*: If *b* is a bit, then
    - $(b\ AND\ 1) = (b\ OR\ 0) = (b\ XOR\ 0) = b$
    - $(b\ AND\ 0) = 0$
    - $(b\ OR\ 1) = 1$
    - $(b\ XOR\ 1) = \neg b$.

- **Inspecting a bit**: Given X, to see if $X[k]$ is 0 or 1, we use the mask $M = (1$ `<<` $k)$ (the string that is all 0's except that bit $k = 1$): If X `&` M `==` (the integer) 0 then $X[k] = 0$; if X `&` $M \neq$ (the integer) 0, then $X[k] = 1$. **Note**: Don't test the result for = the integer 0 or 1; that only works for testing $X[0]$.
  - **Example**: To get bit 1 (the next-to-rightmost bit) of *110100*, we calculate *110100* `&` *000010* = *000000*, so we know bit 1 was 0. To get bit 1 of *110011*, we calculate *110011* `&` *000010* = *000010* ≠ *000000*, so we know bit 1 was 1.
- Another way to inspect bit k is to right-shift it to position zero and AND it with integer 1: $X[k] = (X$ `>>` $k)$ `&` 1.
  - **Example**: To get bit 1 of *110100*, we calculate *(110100* `>>` *1)* `&` *1* = *?11010* `&` *0…01* = *0*.[2] To get bit 1 of *110011*, we calculate *(110011* `>>` *1)* `&` *1* = *?111001* `&` *0…01* = *1*.)
- **Setting a bit**: To set $X[k] \leftarrow 1$ and not change any other bits of X, we use the same mask $M = (1$ `<<` $k)$ and set $X \leftarrow X \mid M$. Then $X[k] \leftarrow X[k]$ *OR* $1 = 1$, and for $i \neq k$, $X[i] \leftarrow X[i]$ *OR* $0 = X[i]$.
  - **Example**: If X = *110100* and $k = 1$, then $X \leftarrow X \mid$ *000010* = *110110*.
- **Clearing a bit**: To set $X[k] \leftarrow 0$ and not change any other bits of X, we use the bitwise *NOT* of the mask $M = (1$ `<<` $k)$; we set $X \leftarrow X$ `&` `~`$(1$ `<<` $k)$.
  - **Example**: If X = *110110*, then $X \leftarrow X$ `&` *111101* = *110100*.
- **Flipping a bit**: To set $X[k] \leftarrow \neg X[k]$ and not change any other bits of X, we use the bitwise *XOR* of the mask $M = (1$ `<<` $k)$; we set $X \leftarrow X$ `^` `~`$(1$ `<<` $k)$.
  - **Example**: If X = *110100* and $k = 1$ then $X \leftarrow X$ `^` *000010* = *110110*. Doing another $X \leftarrow X$ `^` *000010* changes X back to *110100*.

G. Manipulating More Than One Bit

- We can extend the bitwise operations and masks to work on larger parts of a bitstring. E.g., if we want to set $X[0{:}2] \leftarrow 111$, we can use $X \leftarrow X \mid$ *00…*

[2] The sign bit is marked ? because it depends on whether logical or arithmetic shift is being done.

0111. If we want to inspect *X[7:5]*, we can shift *X* rightwards and capture the 3 bits we want by calculating *(X* `>>` *5)* `&` *0…0111*.

- In C and Java, we can build a mask of zeros with one 1 bit at position *k* using `(1 <<` *k*`)`, but it's trickier to build a mask of zeros with a sequence of 1 bits.
  - Say we want a bitstring of length n that begins with all zeros and ends in m 1 bits (where $m < n$). One way to get this is subtracting 1 from 2^m to get $2^{m-1} + 2^{m-2} + \ldots + 2^1 + 2^0$. The C/Java syntax is `(1 << m) - 1`.
  - If for our bitstring, we want some zeros, then m 1's, and finally p 0's, we can take the mask we just built (all 0's except for the last m bits) and left-shift it p positions. The C/Java syntax is `((1 << m) - 1) << p`.
- To build a mask of all 1's except for a sequence of 0 bits, it's probably easiest to calculate it using its bitwise *NOT*. E.g., For the mask 1…100011, we take the bitwise NOT of 0…011100: can be built via

```
~(((1 << 4) - 1) << 2)
```

We get `~(((1 << 4) - 1) << 2)` = `~((0x00000010 - 1) << 2)` = `~(0x0000000f << 2)` = `~(0x0000003c)` = `0xffffffc3`.

Bitstring Operations

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Bitstring shifts can correspond to multiplication and division by two.
- Bitwise operations with masks enable us to manipulate particular bits of a bitstring.

B. Outcomes

After this activity, you should be able to:

- Perform bit shifting and bitwise operations.

C. Questions

1. Fill in the table below, to show the result of repeatedly shifting 10011011 left with zero fill. Treat the bits as unsigned when translating to decimal. Indicate any overflows that occur.

k	*1001 1011 << k*	*In Hex*	*In Decimal*
0	1001 1011	9B	155
1	0011 0110	36	54 (overflow)
2			
3			
4			
5			
6			
7			
8			

2. Fill in the table below, to show the result of repeatedly shifting 1001 1011 right arithmetically (with sign fill). Use 2's complement.

k	***1001 1011 >>> k***	***In Hex***	***In Decimal***
0	1001 1011	9B = -65_{16}	–101
1	1100 1101	CD = -33_{16}	-51
2			
3			
4			
5			
6			
7			
8			

3. Fill in the table below, to show the result of repeatedly shifting 1001 1011 circularly left. (Note you don't need to translate the result into decimal.)

k	***After k left circular shifts***	***In Hex***
0	1001 1011	9B
1	0011 0111	37
2	0110 1110	
3		
4		
5		
6		
7		
8	1001 1011	9B

4. Let $X = 1011\ 0110$, $Z = 0000\ 0000$, and $N = 1111\ 1111$. Give the values of (a) ~ X (b) X & N (c) X & Z (d) $X | Z$ (e) $X \mid N$ (f) X ^ Z (g) X ^ N (where ~, &, |, and ^ stand for bitwise *NOT*, *AND*, *OR*, and *XOR* respectively).

5. Let $X = 0111\ 1001$ and $Y = 1100\ 0101$. Give the values of (a) X & Y (b) $X | Y$ (c) X ^ Y.

Solution

1. (Left shift)

k	*1001 1011 << k*	*In Hex*	*Unsigned Decimal*
0	1001 1011	9B	155
1	0011 0110	36	54 (overflow)
2	0110 1100	6C	108
3	1101 1000	D8	216
4	1011 0000	B0	176 (overflow)
5	0110 0000	60	96 (overflow)
6	1100 0000	C0	192
7	1000 0000	80	128 (overflow)
8	0000 0000	00	0 (overflow)

2. (Arithmetic right shift)

k	*1001 1011 >>> k*	*In Hex*	*Signed Decimal*
0	1001 1011	9B = -65_{16}	–101
1	1100 1101	CD = -33_{16}	-51
2	1110 0110	E6	–26
3	1111 0011	F3	-13
4	1111 1001	F9	-7
5	1111 1100	FC	-4
6	1111 1110	FE	-2
7	1111 1111	FF	-1
8	1111 1111	FF	-1

3. (Circular left shift)

k	***After k left circular shifts***	***In Hex***
0	1001 1011	9B
1	0011 0111	CD
2	0110 1110	6E
3	1101 1100	DC
4	1011 1001	B9
5	0111 0011	73
6	1110 0110	E6
7	1100 1101	CD
8	1001 1011	9B

4. If $X = 1011\ 0110$, $Z = 0000\ 0000$, and $N = 1111\ 1111$, then (a) ~ $X = 0100\ 1001$ (b) X & $N = X$ (c) X & $Z = Z$ (d) $X \,|\, Z = X$ (e) $X \,|\, N = N$ (f) X ^ $Z = X$ (g) X ^ $N =$ ~$X = 0100\ 1001$

5. If $X = 0111\ 1001$ and $Y = 1100\ 0101$, then (a) X & $Y = 0100\ 0001$ (b) $X \,|\, Y = 1111\ 1101$ (c) X ^ $Y = 1011\ 1100$.

