Bitstring Operations

CS 350: Computer Organization & Assembler Language Programming

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

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.

C. Sections of Bitstrings:

- Bits are numbered 0, 1, … starting from the right.
- We can specify individual bits using an index in square brackets; we can select a subsequence of bits using index : index in square brackets.
- **Example 1**: If $X = 011$, then $X[0] = X[1] = 1$, $X[2] = 0$, and $X[1{:}0] = 11$, $X[2{:}1] = 01$, $X[2{:}0] = X = 011$.
- Why do this? Actual bitstring data is often broken up into individual **fields**.
  - For example, if B is a 32-bit string, then to read it as an IEEE floating-point number, we break it up into three fields: The (overall) sign of the number in $B[31]$, the exponent in $B[30{:}23]$, and the fraction in $B[22{:}0]$.
  - Later, when we study the LC-3, we can read a 16-bit string S as an instruction with an identifying 4-bit op code in $S[31{:}28]$; depending on the op code, we'll need to break up $S[27{:}0]$ in one of a number of different ways.
- There are a couple of basic kinds of operations we'll do on bitstrings.
  - **Shifting** a string of bits left / right: We'll drop some bits on the left / right and fill in the holes that open up on the right / left.

- **Bitwise** operations, where we perform the same operation on each bit of a string (as in flipping all the bits of a string), or we take two strings and perform the same operation using bits from corresponding positions in the strings (we'll see this in detail below).
- We can use these operations to do multiplication and division by powers of 2 and to isolate/select/change a field of a bitstring. For example, if we want to select bits 3 to 1 of a string $b_5\, b_4\, b_3\, b_2\, b_1\, b_0$, we can shift to make them the rightmost bits of the string (leaving, say, $0\, b_5\, b_4\, b_3\, b_2\, b_1$) and then zero out the bits we don't want (to get $0\, 0\, 0\, b_3\, b_2\, b_1$). Note how we treat those three bits will depend on the context: They might represent an unsigned integer or maybe a signed integer or maybe three individual true/false flags.

D. Left Shifting With Zero Fill

- Let X be an n-bit string. We can write $X[n-1:0]$ if we want to be explicit about X being a bitstring.
- To **left shift *X* one bit position with zero fill**, set $X[n-1:1] \leftarrow X[n-2:0]$ and $X[0] \leftarrow 0$.
- To left shift k bit positions with zero fill, we shift left one bit k times.
- In C and Java, $X << k$ is the result of shifting X left k bits with zero fill.
  - **Example 2**: if $k = 0, 1, 2, \ldots$, then `1` << k = `1`, `2`, `4`, `8`, `16`, … ($= 2^k$). You can't write binary constants in C or Java, but if you could, you'd see `00…0001` << `1` = `00…0010`, then `00…0001` << `2` = `00…0100`, etc.
- In general, left shifting k bits with 0 fill corresponds to multiplying by 2^k for unsigned and 2's complement binary integers.
- For unsigned integers, if a 1 is "shifted out" of the number, then overflow has occurred.
  - **Example 3**: Treating 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 $\neq$ the old sign bit, then overflow has occurred.

  - **Example 4** : Treating 00011 = 3 as a 2's complement integer, repeated left shifting yields 00110 = 6, 01100 = 12, 11000 = −8 (with overflow), 10000 = −16, and 00000 = 0 (with overflow)
- Note the multiplying by 2 works for negative 2's complement integers too.
  - **Example 5**: With 5 bits, 11111 = −1 in 2's complement. Repeated left shifting yields 11110 = -2, 11100 = -4, 11000 = -8, 10000 = -16, 00000 = 0 with overflow.

E. Circular Shifting

- In circular shifting (left or right), the bit that falls off the end of the string is swung around to fill the hole that opens up.
  - In left circular shift, the old sign bit[*] is used to fill in position 0.
    - $X[n-1:1] \leftarrow X[n-2:0]$ and (simultaneously) $X[0] \leftarrow X[n-1]$.
  - In right circular shift, the old rightmost bit becomes the new leftmost bit.
    - $X[n-2:0] \leftarrow X[n-1:1]$ and (simultaneously) $X[n-1] \leftarrow X[0]$.
  - **Example 6**: Repeatedly left circular-shifting 00011 one bit yields 00110, 01100, 11000, 10001, 00011, 00110, ….
  - Note the difference w.r.t. shifting with zero fill: (00011), 00110, 01100, 11000, 10000, 00000, ….
- Circular shifting an *n*-bit string *n* positions yields the original value.
- Circular shift is nice for repeatedly cycling through the bits of a bitstring.

F. Right (Non-Circular) Bit Shifting

- To right shift an n bit string, the leftmost $n-1$ bits become the rightmost $n-1$ positions: $X[n-2:0] \leftarrow X[n-1:1]$. The sign bit can be filled in various ways:
- **Right shift with zero fill** (a.k.a. **logical right shift**): $X[n-1] \leftarrow 0$
- **Right circular shift**: $X[n-1] \leftarrow X[0]$ (we saw this above).
- **Right shift with sign fill** (a.k.a. **arithmetic right shift**): Keep $X[n-1]$ unchanged.

[*]If the string represents an unsigned integer, then the leftmost bit is just the leftmost bit.

- If the leftmost bit of X is 0, then then logical and arithmetic shift of k bits are the same and correspond to division by 2^k, whether X is signed or unsigned.
    - **Example 7**: Logical and arithmetic right-shifting 01111_2 = 15 both yield 00111 = 7, 00011 = 3, 00001 = 1, 00000 = 0, 00000 = 0, ….
- If X represents a negative integer, then only arithmetic right shift corresponds to division because it keeps the sign bit of 1.
    - **Example 8**: 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 division and remainder operators (`/` and `%`) work in C and Java.)
- A logical right shift of a negative int X treats X as the bitstring for an unsigned int, which we then divide by 2.
    - **Example 9**: Using 16 bit integers, −13 = $-000D_{16}$ equals FFF3. One-bit logical right shift yields 7FF9, which is a large positive number.

Right Bit Shifting in C and Java

- In Java, logical right shift is the >>> operator and arithmetic right shift is the >> operator.
    - **Example 10**: In Java, `-13` >> `1` is `-7`; `-13` >> `2` is `-3`; `-13` >> `3` is `-1`.
- For right shift, C is weird: The C standard says that whether >> stands for arithmetic or logical right shift is up to the implementation.

G. 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.
- **Bitwise NOT**: To get the Bitwise NOT of a bitstring X, you take the *NOT* of each bit of X. So Y = Bitwise NOT X means each $Y[i] = NOT\ X[i]$. (Remember, $Y[i]$ means the i'th bit of Y, counting from the right end.)
    - Bitwise NOT is the same as taking the 1's complement of an integer.

  - **Example 11**: Bitwise NOT of 101101 = 010010.
- **Bitwise AND** , etc: To get the Bitwise AND of two bitstrings *X* and *Y*, you take the one-bit *AND* on each corresponding pair of bits from *X* and *Y*. So $Z = X$ Bitwise AND Y means each $Z[i] = X[i]$ *AND* $Y[i]$.
- The bitwise versions of OR, XOR, etc. are similar: For bitwise OR, you take the OR of each pair of corresponding bits; for bitwise XOR, it's the XOR of each pair of bits, and so on.
  - **Example 11**:
    - Bitwise AND of 101100 and 100001 is 100000.
    - Bitwise OR of 101100 and 100001 is 101101.
    - Bitwise XOR of 101100 and 100001 is 001101.
- **Bitwise operations in C and Java**: In these languages, unary ~ (tilde) and binary `&`, `|`, and ^ (circumflex) are the operators for bitwise NOT, AND, OR, and XOR respectively.
  - Contrast these with `!`, `&&`, and `||`, and `!=` (logical *NOT*, *AND*, *OR*, and *XOR*).
  - In C, integer `0` represents false and any integer ≠ `0` represents true. (The built-in logical operators always yield 0 or 1.) So `x && y` is true iff both `x` and `y` are nonzero. On the other hand, `x & y` behaves like true iff any pair of corresponding bits in `x` and `y` are both `1`.
  - **Example 12**: In C, we have `1 && 2 == 1`, but `1 & 2 == 0`.

H. Testing and Manipulating One Bit Of A String

- A **bit mask** is a bitstring of some specific pattern used to help carry out an operation, such as selecting or modifying parts of a bitstring.
- We rely on the following properties of *AND*, *OR*, and *XOR*: If *b* is a bit, then

Y	$b \wedge Y$	$b \vee Y$	$b \oplus Y$
0	0	b	b
1	b	1	$\neg b$

Inspecting a bit

- Given X and k, to inspect (view) $X[k]$, there are a couple of techniques.
- One is to shift X rightward k bits and view bit 0 of the result using a bitwise AND with 1: $(X >> k)$ `& 1` equals $X[k]$.
- The other is to build a mask M that is all 0's except $M[k] = 1$. (As an integer, $M = 2^k$.) Then X `&` M is integer 0 if $X[k] = 0$; if X `&` $M \neq$ integer 0 then it equals M, and this indicates that $X[k] = 1$.
- **Example 13**: (In 16 bits), let $X = \text{FF34}_{16} = 1111\ 1111\ 0011\ 0100_2$ and let $k = 2$.
  - Using the right-shift technique, $(X >> k)$ `& 1` = ??11 1111 1100 1101 `&` 0001_{16} = 0001_{16}, so $X[2] = 1$. (The two question mark bits are 0 or 1 depending on whether >> is an arithmetic or logical right shift.)
  - Using the left shift technique, we want our mask M to be all 0's except $M[k] = M[2] = 1$, so we want $M = 0004_{16} = 2^2 = 2^k = (1 << k)$. Then X `&` M = FF34_{16} & $0004_{16} = 0004_{16}$, so $X[k] = 1$. (Note the test is X `&` M `!= 0`, not X `&` M `= 1`.)

Setting a bit (to 1)

- To set $X[k] \leftarrow 1$ and not change any other bits of X, we use the same mask $M = 2^k$ (0's everywhere except that position k is 1) and use bitwise OR: $X \leftarrow X \mid M$. Then for bit position k, $X[k] \leftarrow X[k] \vee 1 = 1$, and for the bits at positions $i \neq k$, $X[i] \leftarrow X[i] \vee 0 = X[i]$.
- **Example 14**: If again $X = \text{FF34}_{16}$ and $k = 2$, then assigning $X = X \mid M = \text{X} \mid (1 << k)$ makes $X = \text{FF34} \mid 0004 = \text{FF34}$. (Bit 2 of FF34 is already 1, so setting it to 1 doesn't change it.) If instead $X = \text{FF30}$, then $X = \text{FF30} \mid 0004 = \text{FF34}$, which is FF30 but with bit 2 set to 1.

Clearing a bit (making it 0)

- To set $X[k] \leftarrow 0$ and not change any other bits of X, we flip the mask from all 0's except at position k to be all 1's except at position k, and we use bitwise AND, so $X \leftarrow X$ `&` $\sim(2^k) = X$ `&` $\sim(1 << k)$. (Recall that ~ is bitwise NOT.)

- **Example 15**: Again using $X = \mathrm{FF34}_{16}$ and $k = 2$, then setting bit k to 0 gives FF30: $X = X$ `&` `~(1` $<<$ k`)` equals FF34 `&` ~0004 equals FF34 `&` FFFB equals FF30. (If we look at just the rightmost hex digit, 4 `&` B_{16} = 0100 `&` 1011 = 0000.)

Flipping a bit

- To set $X[k] \leftarrow \neg X[k]$ and not change any other bits of X, we use the bitwise *XOR* with our usual mask; we set $X \leftarrow X \mathbin{\char`^} 2^k = X \mathbin{\char`^} (1 << k)$.
- **Example 16**: Again using $X = \mathrm{FF34}_{16}$ and $k = 2$, flipping bit 2 (which is 1) should give us FF30: FF34 ^ 0004 = FF30. (In the last digit, 4 ^ 4 = 0100 ^ 0100 = 0000.)

I. 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 $X[2:0] \leftarrow 111$, we can use $X \leftarrow X \mid 00\ldots0111$. If we want to inspect the three bits of $X[7:5]$, we can shift X rightwards 5 bits and capture the 3 bits we want: (X >> 5) `&` 0…0111.
- To build a mask M that ends with a run of m 1's, we can use $2^m - 1 = (1 << m) - 1$. E.g., if $m = 3$, we get $2^3 - 1 = 8 - 1 = 7 = 0\ldots0111_2$. If $M = (1 << m) - 1$, then
  - X `&` M yields the last m bits of X.
  - $X \leftarrow X \mid M$ sets the last m bits of X to 1's.
  - $X \leftarrow X$ `&` ~M sets the last m bits of X to 0's.
  - $X \leftarrow X$ ^ M flips the last m bits of X.
- More generally, if we want a mask with 0's, then m 1's, then p 0's, we can just build the mask that ends with m 1's and left shift it p positions: we need $(2^m-1) << p = ((1 << m) - 1) << p$.
  - **Example 17**: If we want the mask 0078_{16} = 0000 0000 0111 1000 (so $m = 4$ and $p = 3$), then $((1 << m) - 1) << p = ((1 << 4) - 1) << 3 = (16 - 1) << 3 = \mathrm{000F}_{16} << 3 = 0078_{16}$.

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. Problems

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 $X = 0111\ 1001$ and $Y = 1100\ 0101$. Give the values of (a) X `&` Y (b) $X \mid Y$ and (c) X ^ Y. (I.e., the bitwise *AND*, *OR*, and *XOR* of *X* and *Y*, respectively.)

5. Let $X = 1011\ 0110$ and $Y = 31_{10} = 0001\ 1111$. What are the results of
(a) $Z \leftarrow X$ `&` Y (b) $Z \leftarrow X \mid Y$ (c) $Z \leftarrow X$ `&` ~Y (d) $Z \leftarrow X$ ^ Y

6. Let $X = 1EB5_{16}$ (a) Rewrite X in binary and break up the 16 bits into 4-bit, 3-bit, 3-bit, 1-bit, and 5-bit sections (by hand) (b) Calculate (X >> 12) `&` 15 and verify that it yields the initial 4-bit section of *X*. (c) Write similar expressions to yield the 3-bit, 3-bit, 1-bit, and 5-bit sections of *X*.