Bitwise !

Operators

Bitwise Operators

- A set of **operators** for working with a value's **bit**-level representation
  - Many other languages as well, including Java, Python, JavaScript, etc

- Three classes of bitwise operators:

  - Unary - used as a prefix to a bit vector (like negation on a number)

    ```
    ~ one's complement / invert
    ```

  - Binary operators operating on two vectors of bits:

    ```
    & and
    | or
    ^ exclusive or
    ```

  - Binary shift operators whose LHS is a bit vector and RHS is an integer

    ```
    << shift left
    >> shift right
    ```

~ Bitwise Complement Operator

- This unary complement operator flips each bit in its operand
  - 0s become 1s, 1s become 0s
  - Example: $\sim[0,1,0,1] = [1,0,1,0]$

bit a	~a
`0`	`1`
`1`	`0`

& Bitwise *And* Operator

- The bitwise **and** operator takes two bit vectors, $\vec{a}\&\vec{b}$, and produces $\vec{c}$
  - Each of $\vec{a}$ and $\vec{b}$'s place value bits i are compared
  - When *both* $\overrightarrow{a_i}$ and $\overrightarrow{b_i}$ are **1**, then $\overrightarrow{c_i}$ is **1**
  - Otherwise, $\overrightarrow{c_i}$ is **0**

  - Example:

$$\vec{a} = [1,1,0,0]$$
$$\vec{b} = [1,0,1,0]$$
$$\vec{a}\&\vec{b} = [1,0,0,0]$$

a	b	a & b
1	1	1
1	0	0
0	1	0
0	0	0

& Bitwise *Or* Operator

- The bitwise **or** operator takes two bit vectors, $\vec{a} \mid \vec{b}$, and produces $\vec{c}$
  - Each of $\vec{a}$ and $\vec{b}$'s place value bits i are compared
  - When *either* $\overrightarrow{a_i}$, or $\overrightarrow{b_i}$, or *both*, are **1**, then $\overrightarrow{c_i}$ is **1**
  - Otherwise, $\overrightarrow{c_i}$ is **0**

  - Example:

$$\vec{a} = [1,1,0,0]$$
$$\vec{b} = [1,0,1,0]$$
$$\vec{a}|\vec{b} = [1,1,1,0]$$

a	b	a | b
1	1	1
1	0	1
0	1	1
0	0	0

^ Bitwise *Exclusive Or* Operator

- The bitwise **xor** operator takes two bit vectors, $\vec{a}$ ^ $\vec{b}$, and produces $\vec{c}$
  - Each of $\vec{a}$ and $\vec{b}$'s place value bits i are compared
  - When $\overrightarrow{a_i}$ is not equal to $\overrightarrow{b_i}$ then $\overrightarrow{c_i}$ is **1**
  - Otherwise, $\overrightarrow{c_i}$ is **0**

  - Example:

$$\vec{a} = [1,1,0,0]$$
$$\vec{b} = [1,0,1,0]$$
$$\vec{a}\wedge\vec{b} = [0,1,1,0]$$

a	b	a ^ b
1	1	0
1	0	1
0	1	1
0	0	0

<< Bitwise *Left Shift* Operator

- The bitwise **left shift** operator <<
  - Takes a bit vector $\vec{a}$ on the left-hand side
  - A magnitude m integer on the right-hand side
  - Produces a bit vector $\vec{b}$

- Its effect is: $\overrightarrow{a_i} = \vec{b}_{i+m}$
  - $\vec{a}_{w-m}$ through $\vec{a}_{w-1}$ are truncated
  - $\vec{b}_0$ through $\vec{b}_{m-1}$ are zeroed

`a`	`m`	`a<<m`
`0101`	`1`	`1010`
`0101`	`2`	`0100`
`0101`	`3`	`1000`
`0101`	`4`	`0000`

>> Bitwise *Right Shift* Operator

- The bitwise **right shift** operator >>
  - Takes a bit vector $\vec{a}$ on the left-hand side
  - A magnitude m integer on the right-hand side
  - Produces a bit vector $\vec{b}$
- Its effect is: $\overrightarrow{a_i} = \vec{b}_{i-m}$
  - $\vec{a}_{m-1}$ through $\vec{a}_0$ are truncated
  - $\vec{b}_{w-1}$ through $\vec{b}_{w-m}$ are **sign-extended**
    - If $\vec{a}_{w-1}$ is 0, then 0s will fill
    - If $\vec{a}_{w-1}$ is 1, then 1s will fill
    - Why? To make sure negative numbers in two's complement retain their sign bit

Sign extension with a 0 high-order bit.

a	m	a>>m
0101	1	0010
0101	2	0001
0101	3	0000

Sign extension with a 1 high-order bit.

a	m	a>>m
1010	1	1101
1010	2	1110
1010	3	1111

Bitwise Assignment Operators

- Bitwise assign operators are for when need to perform a bitwise operation on a variable and assign result back to the variable itself
  - Just as with in *arithmetic assignment operators*!
    `i = i + 1;` same as `i += 1;`

- Works with all the binary bitwise operators:

```
&= bitwise AND assignment
|= bitwise OR assignment
^= bitwise XOR assignment
<<= left shift assignment
>>= right shift assignment
```

- Example: `a >>= 2;` is the same as `a = a >> 2;`

~ Complement

bit a	~a
0	1
1	0

& AND

a	b	a & b
1	1	1
1	0	0
0	1	0
0	0	0

\| OR

a	b	a | b
1	1	1
1	0	1
0	1	1
0	0	0

^ XOR
(eXclusive OR)

a	b	a^b
1	1	0
1	0	1
0	1	1
0	0	0

Hex_{16}	$Binary_2$	Dec_{10}
0	0000	00
1	0001	01
2	0010	02
3	0011	03
4	0100	04
5	0101	05
6	0110	06
7	0111	07
8	1000	08
9	1001	09
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

USASCII code chart

b_7 b_6 b_5 →					000	001	010	011	100	101	110	111
b_4	b_3	b_2	b_1	Column → / Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Left Shift <<

a	m	a<<m
0101	1	1010
0101	2	0100
0101	3	1000
0101	4	0000

Right Shift >>
0 sign extended

a	m	a>>m
0101	1	0010
0101	2	0001
0101	3	0000

Right Shift >>
1 sign extended

a	m	a>>m
1010	1	1101
1010	2	1110
1010	3	1111