Datatypes II

Outline

- Logical Operations on Bits
  - AND, OR, NOT, XOR, (NAND, NOR)
  - Shift
- Other Representations
  - Bit vectors
  - Hexadecimal
  - Octal
  - ASCII
  - Floating Point

AND

AND	0	1
0		
1		

AND

AND	0	1
0	**0**	**0**
1	**0**	**1**

Truth Table

A	B	A AND B A & B AB
0	0	
0	1	
1	0	
1	1	

Truth Table

A	B	A AND B A & B AB
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

What is "Bitwise"?

- Traditional Boolean functions are defined on Boolean values (i.e. True and False)
- When we have two strings of bits, we often apply a Boolean function to pairs of respective bits in the two strings
- We refer to this operation on two arrays of bits as a "bitwise" operation
- So we might write

  0110_2 AND $0011_2 = 0010_2$

  meaning that we should apply the AND function to each pair of bits

Bitwise AND

`0101 AND 0110`　　　　`(5 & 6)`

```
0101
0110
----
0100
```

OR

A	B	A OR B A | B A + B
0	0	
0	1	
1	0	
1	1	

OR

A	B	A OR B A | B A + B
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise OR

`0101 OR 0110` `(5 | 6)`

```
0101
0110
----
0111
```

NOT

A	NOT A ~A A’
0	
1	

NOT

A	NOT A ~A A’
0	**1**
1	**0**

Bitwise NOT (Complement)

```
NOT 0101                    ~5
```

```
0101

1010
```

XOR

A	B	A XOR B A ^ B
0	0	
0	1	
1	0	
1	1	

XOR

A	B	A XOR B A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Bitwise XOR

`0101 XOR 0110`

`5^6`

```
0101
0110
----
0011
```

NAND

A	B	A NAND B ~(A & B)
0	0	
0	1	
1	0	
1	1	

NAND

A	B	A NAND B ~(A & B)
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**

Bitwise NAND

`0101 NAND 0110` **`No C/Java Operator`**

`~(5 & 6)`

```
 0101
 0110
 ----
 1011
```

NOR

A	B	A NOR B ~(A | B)
0	0	
0	1	
1	0	
1	1	

NOR

A	B	A NOR B ~(A | B)
0	0	1
0	1	0
1	0	0
1	1	0

Bitwise NOR

`0101 NOR 0110` **No C Operator**

`~(5 | 6)`

```
0101
0110
----
1000
```

How Many?

- How many two-argument boolean functions do you think there are?
- How can you prove it?
- Enumerate them!

All the Boolean Functions?

P	Q	FALSE	P AND Q	~(P -> Q) P AND ~Q	P	~(Q -> P) ~P AND Q	Q	P != Q P XOR Q	P OR Q
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

P	Q	P NOR Q	P == Q ~(P XOR Q)	~Q	Q -> P P OR ~Q	~P	P -> Q ~P OR Q	P NAND Q	TRUE
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

Left Shift

0111 Leftshift 10   7 << 2

Before   0111

After   1100

Right Shift

0111 Rightshift 10 **7 >> 2**

Before 0111

After 0001

Note Well

`1101 Rightshift 10        13 >> 2`

`Before   1101`

`After    0011    or`

`         1111`

How would you negate in 2's-complement using bitwise operators and the plus operator?

How would you negate in 2’s complement using bitwise operators and the plus operator?

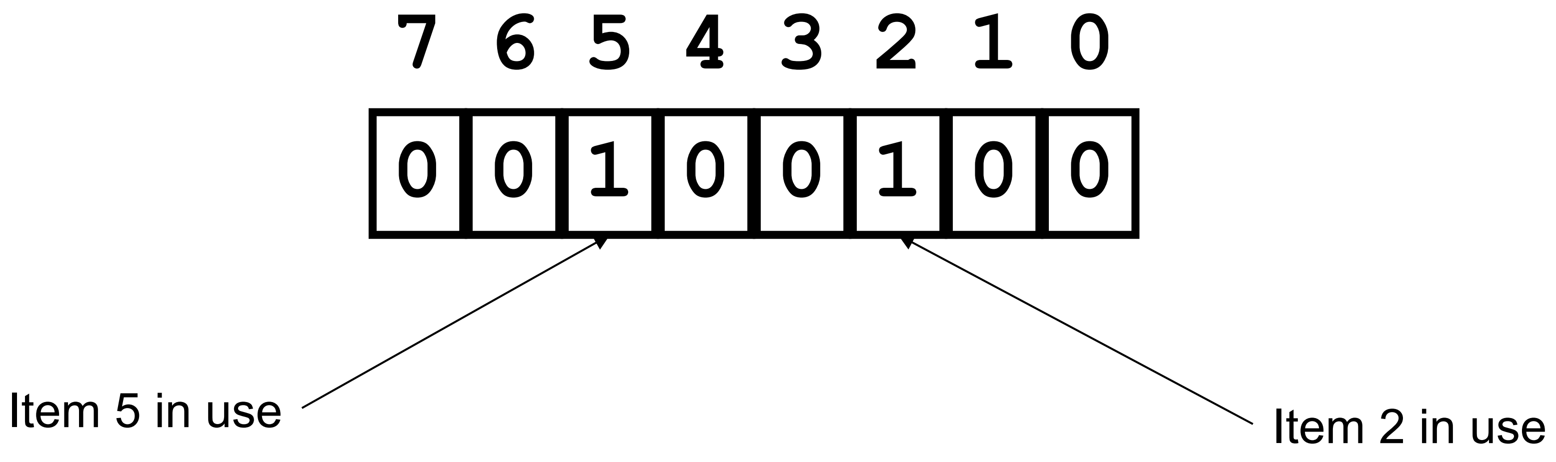
$$\sim x+1$$

Other Representations

Bit Vectors

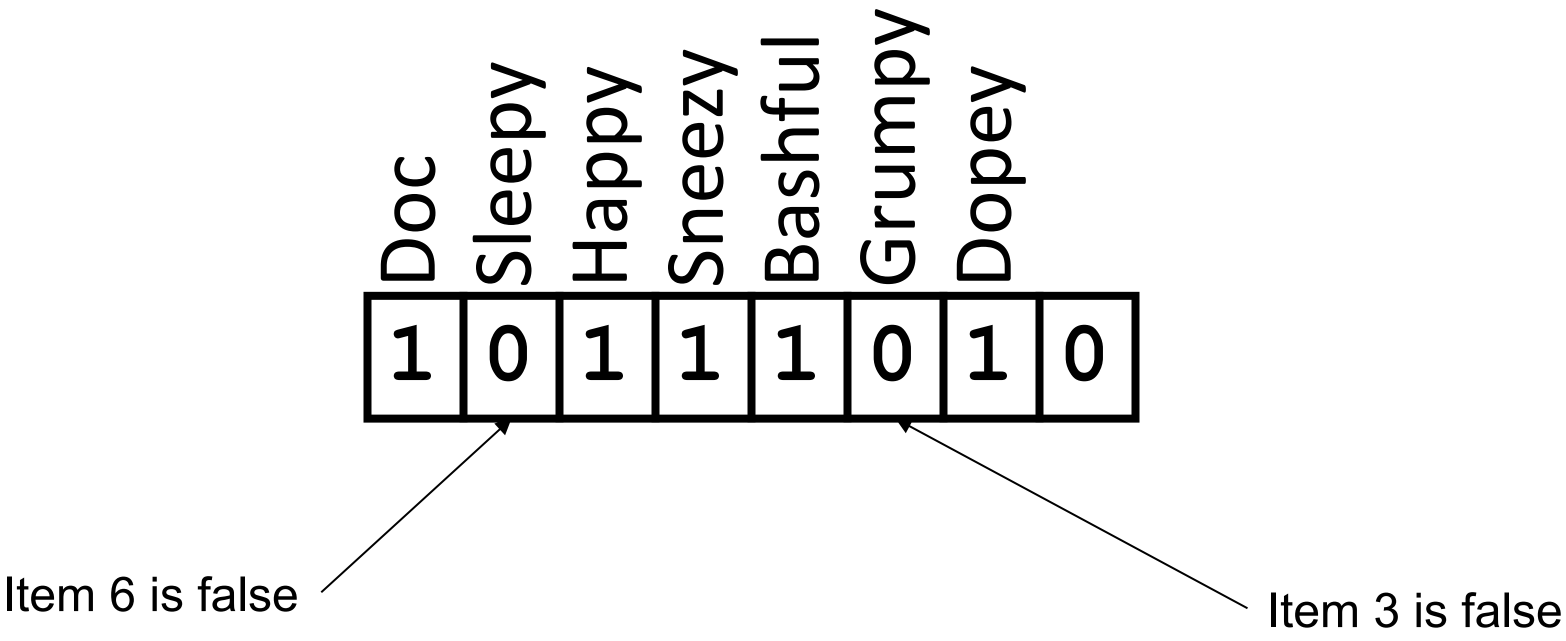
Bit Vectors

- Sometimes for reasons of space efficiency we can effectively store a group of booleans packed together in a single byte/word/etc.



Bit Vectors

- So who's washed their hands!



Bit Vectors

- How do we manipulate the individual bits in the bit vector?
- Examples
  - How do we set bit 6?
  - How do we clear bit 2?
  - How do we toggle a bit?
  - How do we test a bit?

7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0

Tallying the Dwarfs

```
w = 0                      // all bits cleared to 0
w = w | 0b10000000         // Doc
w = w | 0b00100000         // Happy
w = w | 0b00010000         // Sneezy
w = w | 0b00001000         // Bashful
w = w | 0b00000010         // Dopey
```

Alternately,

```
w = ~0                     // all bits set to 1
w = w & 0b10111111         // Sleepy
w = w & 0b11111011         // Grumpy
```

7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0

0b means base 2

Manipulating Bits

- Often we use a constant (a.k.a. **mask**) with a boolean function (four bit examples)
- CLEAR :: Identity: $wxyz_2$ & 1111_2 == $wxyz_2$
  - So put a zero in any bit you want to clear
    - $wxyz_2$ & 1101_2 == $wx0z_2$
- SET :: Identity: $wxyz_2$ | 0000_2 == $wxyz_2$
  - So put a one in any bit you want to set
    - $wxyz_2$ | 0100_2 == $w1yz_2$
- TOGGLE: $wxyz_2$ ^ 1111_2 = $w'x'y'z'_2$
  - So put a one in any bit you want to toggle
    - $wxyz_2$ ^ 1000_2 == $w'xyz_2$

More Manipulating Bits

- To test a bit, clear all the rest
  - $wxyz_2$ & $0010_2 == 00y0_2$
  - Now you can test $00y0_2 == 0000_2$
- To put a *1* in any bit position *n* in a mask, shift left by n
  - $1 << 2 == 0100_2$
- To put a zero in position in a mask, put a one in that position and complement
  - $\sim(1 << 2) == 1011_2$
  - (creates as many leading ones as you need)

Question

We're using 32-bit unsigned binary representation for integers; what value would result from

$$(12 \mid 7) \wedge 63$$

$12 = 1100_2$
$7 = 111_2$
$63 = 111111_2$

$1100_2 \mid 111_2$
$= 1111_2$

$1111_2 \wedge 111111_2$
$= 110000_2$

$110000_2 = 48$

A. 15

B. 48

C. 31

D. 0

Today's Number: 16,384

You are given a 16-bit unsigned binary number, *x*. To test if bit 8 (numbered from right to left starting at 0) is 1, you could use

x & 0000000100000000$_2$!= 0

A. ~x + 256 == ~1

B. x | 256 != 0

C. x & (1<<8) != 0

D. x & (1>>8) != 0

Hexadecimal & Octal

Difficult

- Using binary numbers is both a blessing a curse!
  - One can examine directly any particular bit
  - Reading, writing, etc. prone to error

Solution

- Turns out that mathematics has an answer for us
- Group bits and assign a single digit to represent each group
- How big should each group be?

Hint: Use a Base That's a Power of 2!

- Base 2^3, a.k.a. Base 8, a.k.a. Octal!
  - Use the digits 0 – 7 and group bits in 3s

Base 2	000	111	010	100	101
Base 8	0	7	2	4	5

  - 15 bits in 5 octal digits!
  - And it works backwards, too!

Base 8	6	4	3	0	2
Base 2	110	100	011	000	010

Use a Base That's a Power of 2!

- Base 2^4, aka Base 16, aka Hexadecimal!
  - Use the digits 0 – 15 and group bits in 4s
  - Oops! Digits 10-15! We'll just use A-F.

Base 2	1000	1111	0011	1100	0001
Base 16	8	F	3	C	1

  - $10001111001111000001_2 = 8F3C1_{16}$
  - And it too works backwards:

Base 16	F	0	0	D	5
Base 2	1111	0000	0000	1101	0101

What is 111010001010100_2 in octal?

A. 75624_8

B. 07212_8

C. 74540_8

D. 72124_8

$111\ 010\ 001\ 010\ 100_2$

$7\quad 2\quad 1\quad 2\quad 4_8$

What is 72124_8 in hexadecimal?

A. 7454_{16}

B. $E8A8_{16}$

C. 6343_{16}

D. $3A2A_{16}$

$7 \quad 2 \quad 1 \quad 2 \quad 4_8$

$111\ 010\ 001\ 010\ 100_2$

$0111\ 0100\ 0101\ 0100_2$

$7 \quad 4 \quad 5 \quad 4_{16}$

In Java (and C)

- Constant integers
  - 456 is decimal
  - **0**456 is octal
  - **0x**456 is hexadecimal

  - **0b**010101110 is sometimes used for binary,
    but is <u>not</u> standard in Java and C
- This notation often shows up instead of typographical subscripts!

ASCII

Why two codes?

Dec	Hx	Oct	Char		Dec	Hx	Oct	Char	Dec	Hx	Oct	Char	Dec	Hx	Oct	Char
0	0	000	NUL	(null)	32	20	040	SPACE	64	40	100	@	96	60	140	`
1	1	001	SOH	(start of heading)	33	21	041	!	65	41	101	A	97	61	141	a
2	2	002	STX	(start of text)	34	22	042	"	66	42	102	B	98	62	142	b
3	3	003	ETX	(end of text)	35	23	043	#	67	43	103	C	99	63	143	c
4	4	004	EOT	(end of transmission)	36	24	044	$	68	44	104	D	100	64	144	d
5	5	005	ENQ	(enquiry)	37	25	045	%	69	45	105	E	101	65	145	e
6	6	006	ACK	(acknowledge)	38	26	046	&	70	46	106	F	102	66	146	f
7	7	007	BEL	(bell)	39	27	047	'	71	47	107	G	103	67	147	g
8	8	010	BS	(backspace)	40	28	050	(	72	48	110	H	104	68	150	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	73	49	111	I	105	69	151	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	74	4A	112	J	106	6A	152	j
11	B	013	VT	(vertical tab)	43	2B	053	+	75	4B	113	K	107	6B	153	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	76	4C	114	L	108	6C	154	l
13	D	015	CR	(carriage return)	45	2D	055	-	77	4D	115	M	109	6D	155	m
14	E	016	SO	(shift out)	46	2E	056	.	78	4E	116	N	110	6E	156	n
15	F	017	SI	(shift in)	47	2F	057	/	79	4F	117	O	111	6F	157	o
16	10	020	DLE	(data link escape)	48	30	060	0	80	50	120	P	112	70	160	p
17	11	021	DC1	(device control 1)	49	31	061	1	81	51	121	Q	113	71	161	q
18	12	022	DC2	(device control 2)	50	32	062	2	82	52	122	R	114	72	162	r
19	13	023	DC3	(device control 3)	51	33	063	3	83	53	123	S	115	73	163	s
20	14	024	DC4	(device control 4)	52	34	064	4	84	54	124	T	116	74	164	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	85	55	125	U	117	75	165	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	CAN	(cancel)	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM	(end of medium)	57	39	071	9	89	59	131	Y	121	79	171	y
26	1A	032	SUB	(substitute)	58	3A	072	:	90	5A	132	Z	122	7A	172	z
27	1B	033	ESC	(escape)	59	3B	073	;	91	5B	133	[	123	7B	173	{
28	1C	034	FS	(file separator)	60	3C	074	<	92	5C	134	\	124	7C	174	|
29	1D	035	GS	(group separator)	61	3D	075	=	93	5D	135	]	125	7D	175	}
30	1E	036	RS	(record separator)	62	3E	076	>	94	5E	136	^	126	7E	176	~
31	1F	037	US	(unit separator)	63	3F	077	?	95	5F	137	_	127	7F	177	DEL

Why is this at the end?

What about these?

Fun ASCII Facts

'A' = 65 = 41_{16} = 0100 0001

'a' = 97 = 61_{16} = 0110 0001

'0' = 48 = 30_{16} = 0011 0000

'1' = 49 = 31_{16} = 0011 0001

'2' = 50 = 32_{16} = 0011 0010

…

'9' = 57 = 39_{16} = 0011 1001

Fun ASCII Facts

'A' = 65 = 41_{16} =	0100 0001	
'a' = 97 = 61_{16} =	0110 0001	
'A' + 32 = 61_{16} =	0110 0001	= 'a'
'B' + 32 = 62_{16} =	0110 0010	= 'b'
'z' – 32 = $5A_{16}$ =	0101 1010	= 'Z'
'5' – '0' = 5 =	0000 0101	= 5
5 + '0' = 35_{16} =	0011 0101	= '5'

Fun ASCII Facts

‘A’ = 65 = 41_{16} = 0100 0001

Ctrl-A = SOH = 1 = 1_{16} = 0000 0001

…

‘J’ = 74 = $4A_{16}$ = 0100 1010

Ctrl-J = LF = 10 = A_{16} = 0000 1010

…

‘M’ = 77 = $4D_{16}$ = 0100 1101

Ctrl-M = CR = 13 = D_{16} = 0000 1101

Question

717742_8
$= 111\ 001$
$111\ 111$
$100\ 010$

$= 111\ 001$
$111\ 111$
$100\ 010$

$= 0011$
1001
1111
1110
0010

$= 39FE2_{16}$

Which of these also represents the value 717742_8?

A. $39FE2_{16}$

B. $E7F88_{16}$

C. 717742_{16}

D. $8D5_{16}$

Question

'P' = 0x50

0x50 + 32
= 0x70

0x50 + 0x20
= 0x70

0x50 | 0x20
= 0x70

0x70 = 'p'

If you are given an 8-bit number containing an ASCII representation of the letter 'P', which expression below will compute the ASCII representation of 'p'.

A. 'P' + 32
B. 'P' + 0x20
C. 'P' | 0b00100000
D. All of the above

Floating Point

Transcendental Numbers

- In 1897, a bill was proposed to the Indiana Legislature that set the value of π to exactly 3.2
- (To their credit they didn't consider it for long; it never came to a vote)
- Can computers represent π correctly?
- Can we at least do better than 3.2?

http://www.agecon.purdue.edu/crd/Localgov/Second%20Level%20pages/indiana_pi_bill.htm

Historically

- Initially hardware manufacturers used whatever they thought was appropriate given their market and/or technology required.
- IBM, DEC, CDC, Burroughs, Univac, NCR, Honeywell, GE, RCA, etc. each had their own formats and in fact multiple formats
- Typical implementations might range from 32 up to 128 bits. Common to find multiple formats available (i.e. float and double)
- 1985 IEEE published Floating Point Standard
  - *ANSI/IEEE Standard 754-1985,*
    Standard for Binary Floating Point Arithmetic

Scientific Notation

$$\underbrace{-}_{\text{Sign}}\underbrace{6.023}_{\text{Normalized Mantissa}} \times \underbrace{10}_{\text{Base}}{}^{\overset{\text{Sign of Exponent}}{-}\underbrace{23}_{\text{Exponent}}}$$

?

How would you do it?

Binary Floating Point Representation

- Same basic idea as scientific notation
- Modifications and improvements based on a long history: IEEE-754 standard
  - Precise representation at the bit level
  - Precise behavior of arithmetic operations
  - Efficiency (Space & Time)
  - Additional requirements
    - Special values: not-a-number, +/-infinity, etc.
    - Correct rounding
    - Sortable without FP hardware

IEEE-754

31	30 … 23	22 … 0
0	00000000	00000000000000000000000
sign	exponent	mantissa (significand)

$$\underbrace{(-1)^{S}}_{\text{Sign}} * \underbrace{1}_{\text{1 is understood}}.\underbrace{M}_{\text{Mantissa (w/o leading 1)}} * \underbrace{2}_{\text{Base}}{}^{\overbrace{E-127}^{\text{Exponent}}}$$

Sign

1 is understood

Mantissa (w/o leading 1)

Base

Exponent

IEEE-754

sign	exponent	mantissa (significand)
0	00000000	00000000000000000000000

$$\underbrace{(-1)^{S}} * \underbrace{1.M} * \underbrace{2^{E-127}}$$

Can any of these equal 0?

So how can we represent 0?

0	00000000	00000000000000000000000
sign	exponent	mantissa (significand)

$$(-1)^{S} * 1.M * 2^{E-127}$$

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

Can be written...

0	00000000	00000000000000000000000
sign	exponent	mantissa (significand)

$$(-1)^{S} * 2^{E-127} * 1.M$$

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

0 00000000 00000000000000000000000
= 0
1 00000000 00000000000000000000000
= -0

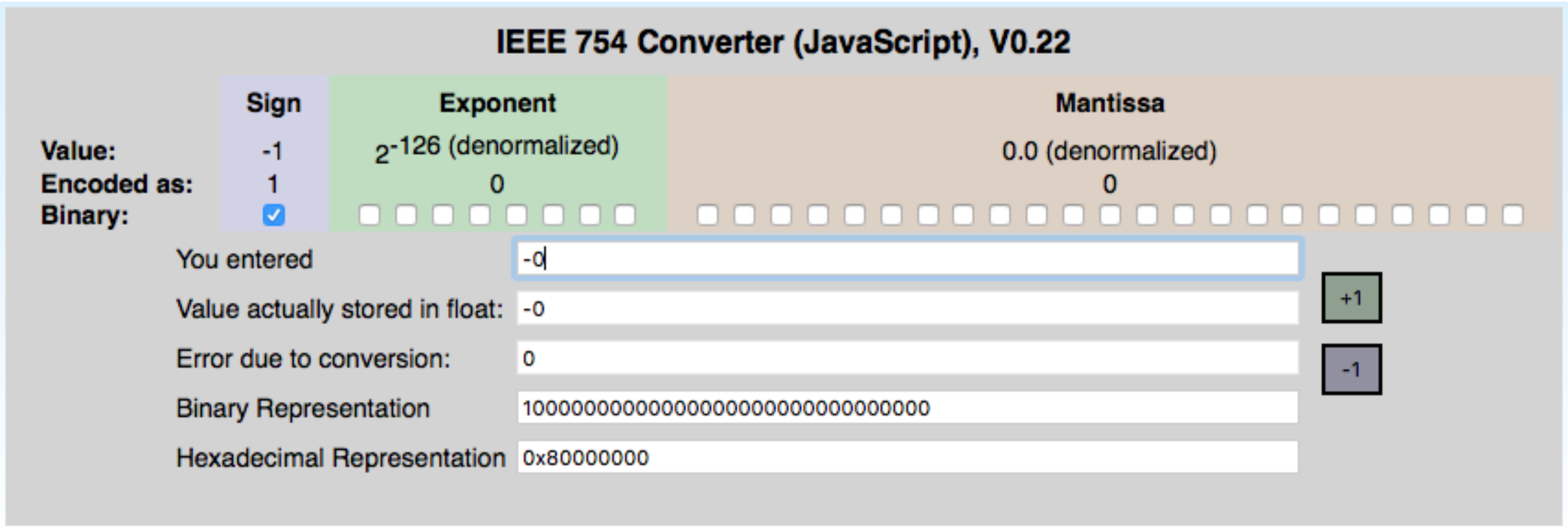
0	00000000	00000000000000000000000
sign	exponent	mantissa (significand)

$$(-1)^{S} * 2^{E-127} * 1.M$$

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

Perhaps Some Automation?

https://www.h-schmidt.net/FloatConverter/IEEE754.html



```
0 11111111 00000000000000000000000  = Infinity
1 11111111 00000000000000000000000  = -Infinity
```

0	00000000	00000000000000000000000
sign	exponent	mantissa (significand)

$$(-1)^{S} * 2^{E-127} * 1.M$$

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

`0 11111111 00000100000000000000000  = NaN`

`1 11111111 00100010001001010101010  = NaN`

0	00000000	00000000000000000000000
sign	exponent	mantissa (significand)

$$(-1)^{S} * 2^{E-127} * 1.M$$

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

Not a Number (NaN)

- Suppose A is a floating point number set to NaN
- A != B is *true*, when B is another floating point number, NaN, infinity, or anything
- Interestingly A != A is *true* as well
  - Not equal to itself
- Also, if A or B is NaN, the following are always *false:*

  $A < B$, $A > B$, $A == B$

`0 10000000 00000000000000000000000 = +1 * 2^(128-127) * 1.0 = 2`

0	00000000	00000000000000000000000
sign	exponent	mantissa (significand)

$$(-1)^{S} * 2^{E-127} * 1.M$$

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

```
0 10000001 10100000000000000000000 = +1 * 2^(129-127) * 1.101 = 6.5

1 10000001 10100000000000000000000 = -1 * 2^(129-127) * 1.101 = -6.5
```

0	00000000	00000000000000000000000
sign	exponent	mantissa (significand)

$$(-1)^{S} * 2^{E-127} * 1.M$$

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

Edge case for tiny numbers: E==0 is special, so 2^{-126} is smallest exponent; we use underflow case for anything smaller, like 2^{-127}; note when E==0 it's computed as if E==1

```
0 00000001 00000000000000000000000 = +1 * 2^(1-127) * 1.0 = 2^(-126)
0 00000000 10000000000000000000000 = +1 * 2^(-126)  * 0.1 = 2^(-127)
```

0	00000000	00000000000000000000000
sign	exponent	mantissa (significand)

$$(-1)^{S} * 2^{E-127} * 1.M$$

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

0 00000000 00000000000000000000001

= +1 * $2^{(-126)}$ * 0.00000000000000000000001

= $2^{(-149)}$ (Smallest positive value)

0	00000000	00000000000000000000000
sign	exponent	mantissa (significand)

$$(-1)^{S} * 2^{E-127} * 1.M$$

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

E = 10001110_2 = 128+8+4+2 = 142

142 - 127 = 15

M = 0

1_2 followed by 23 zeros = 1

$1 * 2^{15}$ = 32768

What number is represented by this 32-bit IEEE-754 encoding?

0 10001110 00000000000000000000000

A. 0

B. 32768

C. +Infinity

D. 2^{142}

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

0 11111111 00000100000000000000000 = NaN

1 11111111 00100010001001010101010 = NaN

0 11111111 00000000000000000000000 = Infinity

0 10000001 10100000000000000000000 = +1 * $2^{(129-127)}$ * 1.101 = 6.5

0 10000000 00000000000000000000000 = +1 * $2^{(128-127)}$ * 1.0 = 2

0 00000001 00000000000000000000000 = +1 * $2^{(1-127)}$ * 1.0 = $2^{(-126)}$

0 00000000 10000000000000000000000 = +1 * $2^{(-126)}$ * 0.1 = $2^{(-127)}$

0 00000000 00000000000000000000001

= +1 * $2^{(-126)}$ * 0.00000000000000000000001

= $2^{(-149)}$ (Smallest positive value)

0 00000000 00000000000000000000000 = 0

1 00000000 00000000000000000000000 = -0

1 10000001 10100000000000000000000 = -1 * $2^{(129-127)}$ * 1.101 = -6.5

1 11111111 00000000000000000000000 = -Infinity

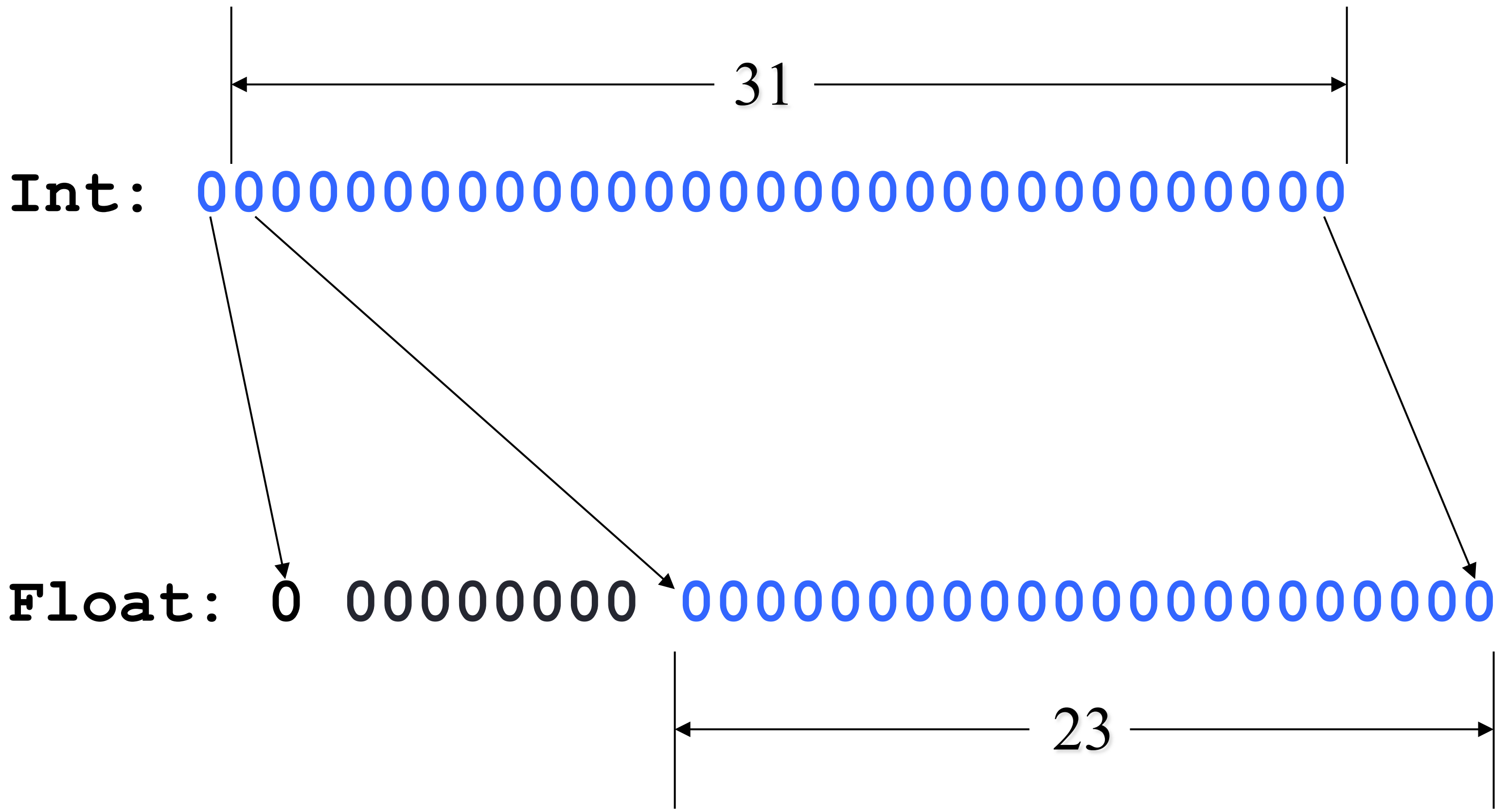
What's Up Here!

```
#include <stdio.h>

int main()
{
        float f;
        int i = 1234567897;
        int j;
        f = i;
        j = (int)f;
        printf("i = %d  f = %f  j = %d\n", i, f, j);
        return 0;
}

$ ./demo
i = 1234567897  f = 1234567936.000000  j = 1234567936
```

Reality



31
Int: 0000000000000000000000000000000
Float: 0 00000000 0000000000000000000000
23

IEEE-754 Double

63	62 52	51 0
0	000...00	000000000...00000000000
sign	exponent	mantissa (significand)

$$\underbrace{(-1)^{S}}_{\text{Sign}} * \underbrace{1}_{\text{1 is understood}}.\underbrace{M}_{\text{Mantissa (w/o leading 1)}} * \underbrace{2}_{\text{Base}}{}^{\underbrace{E-1023}_{\text{Exponent}}}$$

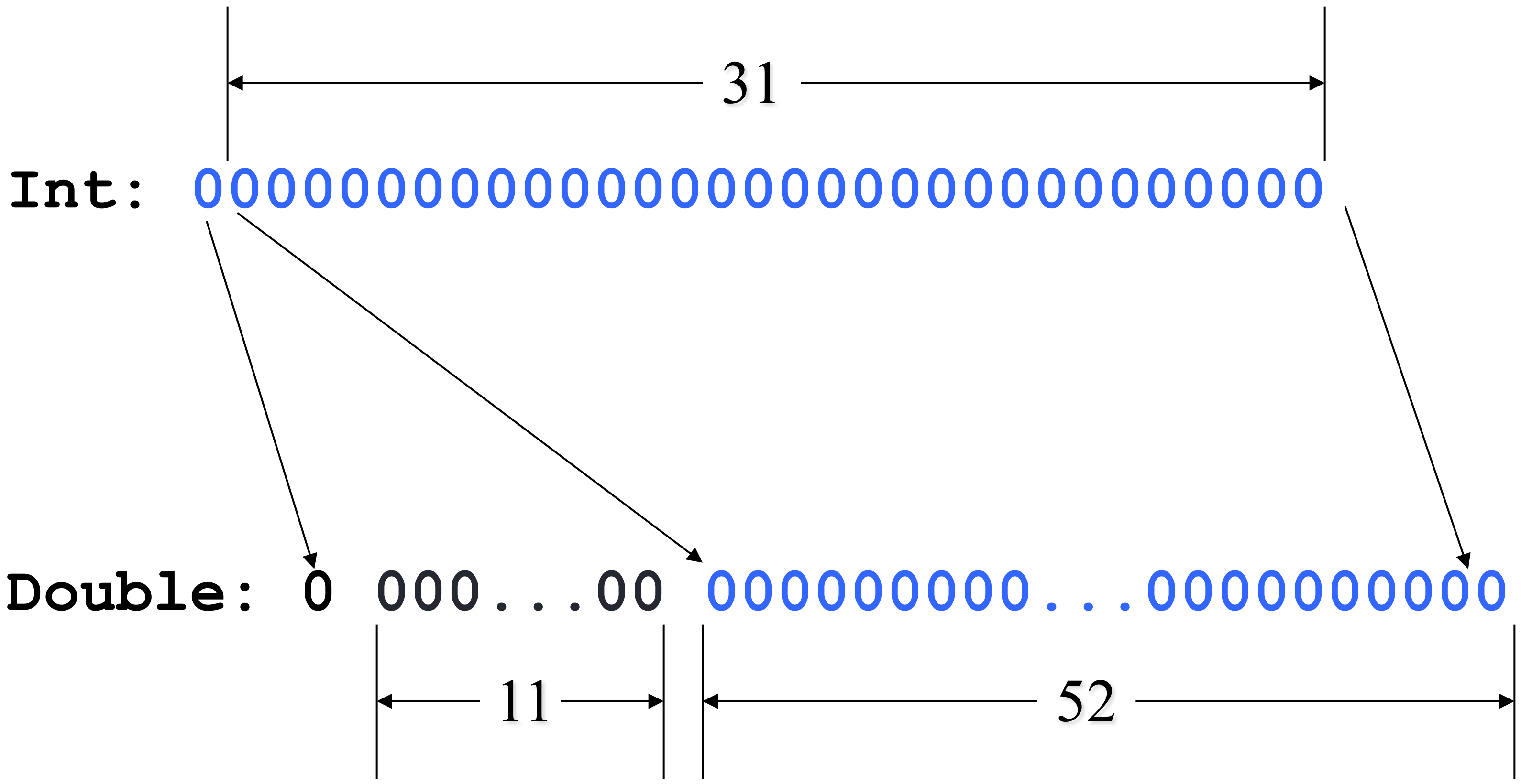
Special Cases

63	62 … 52	51 … 0
0	00000000	000000000000000000000000
sign	exponent	mantissa (significand)

$$(-1)^{S} * 1.M * 2^{E-1023}$$

	E == 0	0 <E< 2047	E == 2047
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

Double (64 bits)



Better?

```
#include <stdio.h>

int main()
{
        double d;
        int i = 1234567897;
        int j;
        d = i;
        j = (int)d;
        printf("i = %d  d = %f  j = %d\n", i, d, j);
        return 0;
}

$ ./demo
i = 1234567897  d = 1234567897.000000  j = 1234567897
```

Comparing FP Numbers

- The layout of the representation allows certain operations (like >) to be performed with no conversion
- Compare:

  3.67×10^{14} = 0x57a841ab

  2.89×10^{16} = 0x5acd58cb
- Notice the exponent bits come first in the representation, so an integer comparison can work if the numbers are the same sign

Comparing Two FP Numbers

- If either is NaN, the comparison is defined as “unordered” *(all comparisons to it except != are false)*
- If either is -0.0, replace with +0.0
- If the signs (high bit) are different, the positive number is bigger
- Compare the remaining bits as integers to get <, ==, or >
- If the signs are both negative reverse the comparison result

FP Comparison

```
A 0 10111000 11001010 11001010 1100101
B 0 10111000 11101010 11001010 1100101
```

= 2.58277016531e+17

= 2.76291415041e+17

```
|B| > |A|
```

- Treat as 31-bit unsigned whole numbers
  - Without the sign bits (bit 31)
- And compare the two magnitudes bit by bit, left to right – until you find different values
- Both are positive, so B>A

What do you need to know?

- Given the 6-case chart of FP numbers, be able to
  - Know what each case means and recognize encoded FP numbers that fit each case
  - Know that in the 32-bit form the exponent is biased by 127
  - From an encoded FP number, be able to show its value in the form of $M * 2^E$
  - Be able to compare encoded FP numbers for <,>,==,!=
- Understand
  - Converting decimal FP to IEEE-754 and vice versa
  - Precision issues in converting between integer and FP representations

Given these four IEEE-754 representations, which are the largest and smallest?

NaN #1: 1 11111111 00100010001001010101010
6.5 #2: 0 10000001 10100000000000000000000
2 #3: 0 10000000 00000000000000000000000
-Infinity #4: 1 11111111 00000000000000000000000

A. #1 is largest, #2 is smallest

B. #2 is largest, #4 is smallest

C. #3 is largest, #1 is smallest

D. #4 is largest, #2 is smallest

Bitwise Review

- Can only be applied to integral operands
- *that is,* **`char`**, **`short`**, **`int`** *and* **`long`**
- **`(signed`** *or* **`unsigned)`**

`&`	`Bitwise AND`
`|`	`Bitwise OR`
`^`	`Bitwise XOR`
`<<`	`Shift Left`
`>>`	`Shift Right`
`~`	`1's Complement (Inversion)`

Bitwise Questions

```
1 & 3
    1
1 & 2
    0
x << -2  Legal?
    No!
x << 2  Write it another way?
    x * 4
What does right shifting do to signed vars?
    depends
x = x & ~077
 Clears last six bits of x to zero!
```

Shifting Integers Right

```
int main() {
        printf("%10s %10s %10s %10s %10s %10s\n", "Dec", "Hex",
                "arith", "arith", "logical", "logical");
        printf("%10s %10s %10s %10s %10s %10s\n", "", "",
                ">> 1 dec", ">> 1 hex", ">> 1 dec", ">> 1 hex");

        for (int i = 10; i > -10; i--) {
                printf("%10d %10x %10d %10x %10d %10x\n",
                        i, i, i >> 1, i >> 1,
                        (unsigned) i >> 1, (unsigned) i >> 1);
        }
}
```

Shifting Two's-Complement Integers Right

This fact is shown in the output of the code, for non-negative numbers, we get the same result when we use either >> or >>> as shown in the first 11 rows in the output. However, when we divide negative numbers, we need to be sure that we are using the arithmetic right shift >> so that the compiler knows to interpret the dividend as a signed number and yield the correct results. If we use the logical right shift >>> with the intention to divide negative numbers, the compiler will assume those numbers to be unsigned and will just treat the sign bit as a normal bit and thus will not give you the result that you would expect. This is shown in rows 12 - 20 in the output of the code.

Dec	Hex	arith >> 1 dec	arith >> 1 hex	logical >> 1 dec	logical >> 1 hex
10	a	5	5	5	5
9	9	4	4	4	4
8	8	4	4	4	4
7	7	3	3	3	3
6	6	3	3	3	3
5	5	2	2	2	2
4	4	2	2	2	2
3	3	1	1	1	1
2	2	1	1	1	1
1	1	0	0	0	0
0	0	0	0	0	0
-1	ffffffff	-1	ffffffff	2147483647	7fffffff
-2	fffffffe	-1	ffffffff	2147483647	7fffffff
-3	fffffffd	-2	fffffffe	2147483646	7ffffffe
-4	fffffffc	-2	fffffffe	2147483646	7ffffffe
-5	fffffffb	-3	fffffffd	2147483645	7ffffffd
-6	fffffffa	-3	fffffffd	2147483645	7ffffffd
-7	fffffff9	-4	fffffffc	2147483644	7ffffffc
-8	fffffff8	-4	fffffffc	2147483644	7ffffffc
-9	fffffff7	-5	fffffffb	2147483643	7ffffffb

Bitwise Questions

```
Why is x = x & ~077 better than
      x = x & 0177700
```

Why is x = x & ~077 better than x = x & 0177700

```
x    =  1010101010101010

077  =  0000000000111111
~077 =  1111111111000000

x    =  1010101010101010
~077 =  1111111111000000   &
        ----------------
        1010101010000000
```

Why is x = x & ~077 better than x = x & 0177700

```
x        =   1010101010101010

077      =   0000000000111111


x        =   1010101010101010  ⎫
                               ⎬ &
0177700  =   1111111111000000  ⎭
            ___________________

             1010101010000000
```

But what if the word size is bigger than 16 bits?

Why is x = x & ~077 better than x = x & 0177700

```
x    =  11110000111100001010101010101010

077  =  00000000000000000000000000111111
~077 =  11111111111111111111111111000000

x    =  11110000111100001010101010101010
~077 =  11111111111111111111111111000000   &
        --------------------------------
        11110000111100001010101010000000
```

Why is x = x & ~077 better than x = x & 0177700

```
x       =  11110000111100001010101010101010

0177700 =  00000000000000001111111111000000

x       =  11110000111100001010101010101010
0177700 =  00000000000000001111111111000000    &
           --------------------------------
           00000000000000001010101010000000
```

Bitwise Questions

What does this do?

```
(x >> (p+1-n)) & ~(~0 << n);
```

What does this do? (x >> (p+1-n)) & ~(~0 << n);

```
P=15, N=3

  3322222222221111111111
  10987654321098765432109876543210
x=00101010100100101010101101010101
(x >> (p+1-n)) ==> (x >> (15+1-3))
                   (x >> 13)
  00000000000000010101010010010101
------------------------------------
~0
  11111111111111111111111111111111
~0 << n   ==>  ~0 << 3
  11111111111111111111111111111000
               ~(~0 << 3)
  00000000000000000000000000000111
------------------------------------
  00000000000000000000000000000101
```

&

Bitwise Questions

```
Why is x = x & ~077 better than
       x = x & 0177700
First one is independent of word length
(no extra cost...evaluated at compile time)

What does this do?
     (x >> (p+1-n)) & ~(~0 << n);
/* getbits: get n bits from position p */
unsigned getbits(unsigned x, int p, int n)
{
  return (x >> (p+1-n)) & ~(~0 << n);
}
```

Questions?

- Logical Operations on Bits
  - AND, OR, NOT, XOR, (NAND, NOR)
  - Shift
- Other Representations
  - Bit vectors
  - Hexadecimal
  - Octal
  - ASCII
  - Floating Point
- Bitwise review