CS330 - Computer Organization and Assembly Language Programming

Lecture 5

Professor : Mahmut Unan – UAB CS

Agenda

- Logical Operations in C
- Shift Operations in C
- Integer Representations
  - Integral Data Types
  - Unsigned Encodings
  - Two’s Compliment Encodings
  - Conversions between Signed and Unsigned

Bit-Level Operations in C

- Operations &, |, ~, ^ Available in C
  - Apply to any "integral" data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
  - Arguments applied bit-wise
- Examples (Char data type)
  - ~0x41 → 0xBE
    - $\sim 01000001_2 \rightarrow 10111110_2$
  - ~0x00 → 0xFF
    - $\sim 00000000_2 \rightarrow 11111111_2$
  - 0x69 & 0x55 → 0x41
    - $01101001_2 \;\&\; 01010101_2 \rightarrow 01000001_2$
  - 0x69 | 0x55 → 0x7D
    - $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

Exercise 5

```
void inplace_swap (int *x, int *y){
*y =*x ^ *y;  /*Step 1*/
*x = *x ^ *y;    /*Step 2*/
*y = *x ^ *y;    /*Step 3*/
}
```

Step	*x	*y
Initially	*a*	*b*
Step 1		
Step 2		
Step 3		

Exercise 5

```
void inplace_swap (int *x, int *y){
*y =*x ^ *y;  /*Step 1*/
*x = *x ^ *y;    /*Step 2*/
*y = *x ^ *y;    /*Step 3*/
}
```

Step

Initially
Step 1
Step 2
Step 3

Rules to remember

a^a = 0

a^0 = a

Exercise 5

```
void inplace_swap (int *x, int *y){
*y =*x ^ *y;  /*Step 1*/
*x = *x ^ *y;    /*Step 2*/
*y = *x ^ *y;    /*Step 3*/
}
```

Step	*x	*y
Initially	*a*	*b*
Step 1	*a*	*a^b*
Step 2		
Step 3		

Exercise 5

```
void inplace_swap (int *x, int *y){
*y =*x ^ *y;  /*Step 1*/
*x = *x ^ *y;   /*Step 2*/
*y = *x ^ *y;   /*Step 3*/
}
```

Step	*x	*y
Initially	*a*	*b*
Step 1	*a*	*a^b*
Step 2	*a^(a^b)*	*a^b*
Step 3		

Exercise 5

```
void inplace_swap (int *x, int *y){
*y =*x ^ *y;  /*Step 1*/
*x = *x ^ *y;   /*Step 2*/
*y = *
}
```

$a \wedge (a \wedge b) = (a \wedge a) \wedge b = b$

Step		
Initially		
Step 1	a	$a \wedge b$
Step 2	$a \wedge (a \wedge b)$	$a \wedge b$
Step 3		

Exercise 5

```
void inplace_swap (int *x, int *y){
*y =*x ^ *y;  /*Step 1*/
*x = *x ^ *y;    /*Step 2*/
*y = *x ^ *y;    /*Step 3*/
}
```

Step	*x	*y
Initially	*a*	*b*
Step 1	*a*	*a^b*
Step 2	*a^(a^b)*	*a^b*
Step 3	*b*	*b^(a^b)*

Exercise 5

```
void inplace_swap (int *x, int *y){
*y =*x ^ *y;  /*Step 1*/
*x = *x ^ *y;    /*Step 2*/
*y = *x ^ *y;    /*Step 3*/
}
```

Step	*x	*y
Initially	*a*	
Step 1	*a*	
Step 2	*a^(a^b)*	
Step 3	*b*	*b^(a^b)*

b^(a^b) = (b^b)^a = a

Exercise 5

```
void inplace_swap (int *x, int *y){
*y =*x ^ *y;  /*Step 1*/
*x = *x ^ *y;    /*Step 2*/
*y = *x ^ *y;    /*Step 3*/
}
```

Step	*x	*y
Initially	*a*	*b*
Step 1	*a*	*a^b*
Step 2	*a^(a^b)*	*a^b*
Step 3	***b***	***a***

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - **&&, ||, !**
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1
    - Early termination
- Examples (char data type)
  - `!0x41 → 0x00`
  - `!0x00 → 0x01`
  - `!!0x41 → 0x01`
  - `0x69 && 0x55 → 0x01`
  - `0x69 || 0x55 → 0x01`
  - `p && *p` (avoids null pointer access)

Watch out for && vs. & (and || vs. |)…
one of the more common oopsies in C programming

Exercise 6

Suppose `x = 0x66` and `y=0x39`

Expression	Value	Expression	Value
`x & y`		`x &&y`	
`x|y`		`x||y`	
`~x | ~y`		`!x || !y`	
`x & !y`		`x && ~y`	

Exercise 6

Suppose `x = 0x66` and `y=0x39`

0110 0110 0011 1001

Expression	Value	Expression	Value
`x & y`		`x &&y`	
`x|y`		`x||y`	
`~x | ~y`		`!x || !y`	
`x & !y`		`x && ~y`	

0110 0110
0011 1001
+ _____
0010 0000
=0x20

Sup 0x39

11 1001

Expressi	Value	Expression	Value
x & y		x &&y	
x|y		x||y	
~x | ~y		!x || !y	
x & !y		x && ~y	

Exercise 6

Suppose `x = 0x66` and `y=0x39`

0110 0110 0011 1001

Expression	Value	Expression	Value
`x & y`	`0x20`	`x &&y`	
`x|y`		`x||y`	
`~x | ~y`		`!x || !y`	
`x & !y`		`x && ~y`	

Exercise 6

Suppose

.001

Expressi **Value**

`x & y`	`x &&y`
`x|y`	`x||y`
`~x | ~y`	`!x || !y`
`x & !y`	`x && ~y`

0110 0110
0011 1001
0111 1111
=0x7F

Exercise 6

Suppose `x = 0x66` and `y=0x39`

0110 0110 0011 1001

Expression	Value	Expression	Value
`x & y`	`0x20`	`x &&y`	
`x|y`	`0x7F`	`x||y`	
`~x | ~y`		`!x || !y`	
`x & !y`		`x && ~y`	

Exercise 6

Suppose

~x = 1001 1001
~y = 1100 0110
|
1101 1111
=0xDF

001

Expressio	Value
x & y	
x|y	
~x | ~y	
x & !y	
	x||y
	!x || !y
	x && ~y

Exercise 6

Suppose `x = 0x66` and `y=0x39`

0110 0110 0011 1001

Expression	Value	Expression	Value
`x & y`	`0x20`	`x &&y`	
`x|y`	`0x7F`	`x||y`	
`~x | ~y`	`0xDF`	`!x || !y`	
`x & !y`		`x && ~y`	

Exercise 6

Suppose x = 0x66 and y=0x39

.001

Expressi	Value
x & y	
x|y	
~x | ~y	
x & !y	x && ~y

!y = 0X00
we don't need to check the others

Exercise 6

Suppose `x = 0x66` and `y=0x39`

0110 0110 0011 1001

Expression	Value	Expression	Value
`x & y`	`0x20`	`x &&y`	
`x|y`	`0x7F`	`x||y`	
`~x | ~y`	`0xDF`	`!x || !y`	
`x & !y`	`0x00`	`x && ~y`	

Suppose x = 0x6

0110

x and y are nonzero
so the result is 0x01

Expression	Value	Expression	Value
`x & y`	`0x20`	`x &&y`	
`x|y`	`0x7F`	`x||y`	
`~x | ~y`	`0xDF`	`!x || !y`	
`x & !y`	`0x00`	`x && ~y`	

Exercise 6

Suppose `x = 0x66` and `y=0x39`

0110 0110 0011 1001

Expression	Value	Expression	Value
`x & y`	`0x20`	`x &&y`	`0x01`
`x|y`	`0x7F`	`x||y`	
`~x | ~y`	`0xDF`	`!x || !y`	
`x & !y`	`0x00`	`x && ~y`	

Exercise 6

Suppose x = 0x6

0110

x and y are nonzero
so the result is 0x01

Expression	Valu		
x & y	0x20		0x01
x|y	0x7F	x||y	
~x | ~y	0xDF	!x || !y	
x & !y	0x00	x && ~y	

Exercise 6

Suppose `x = 0x66` and `y=0x39`

0110 0110 0011 1001

Expression	Value	Expression	Value
`x & y`	`0x20`	`x &&y`	`0x01`
`x|y`	`0x7F`	`x||y`	`0x01`
`~x | ~y`	`0xDF`	`!x || !y`	
`x & !y`	`0x00`	`x && ~y`	

Exercise 6

Suppose `x = 0x66`

`0110 0`

Expression	Value	
`x & y`	`0x20`	
`x|y`	`0x7F`	`x|` `0x01`
`~x | ~y`	`0xDF`	`!x || !y`
`x & !y`	`0x00`	`x && ~y`

x and y are nonzero
!x and !y will be both zeros
0x00 || 0x00 = 0x00

Exercise 6

Suppose `x = 0x66` and `y=0x39`

0110 0110 0011 1001

Expression	Value	Expression	Value
`x & y`	`0x20`	`x &&y`	`0x01`
`x|y`	`0x7F`	`x||y`	`0x01`
`~x | ~y`	`0xDF`	`!x || !y`	`0x00`
`x & !y`	`0x00`	`x && ~y`	

Exercise 6

Suppose x = 0x66 and y=0x39

0110 0

Expression	Value	Expression	Value
x & y	0x20		
x|y	0x7F		
~x | ~y	0xDF	!x | !y	0x00
x & !y	0x00	x && ~y	

x is nonzero
~x = 1001 1001 (non zero)
result is 0x01

Exercise 6

Suppose `x = 0x66` and `y=0x39`

0110 0110 0011 1001

Expression	Value	Expression	Value
`x & y`	`0x20`	`x &&y`	`0x01`
`x|y`	`0x7F`	`x||y`	`0x01`
`~x | ~y`	`0xDF`	`!x || !y`	`0x00`
`x & !y`	`0x00`	`x && ~y`	`0x01`

`y = 0011 1001`

`~y = 1100 0110 =non zero`

Summary

- It's all about bits & bytes
  - Numbers
  - Programs
  - Text
- Different machines follow different conventions
  - Word size
  - Byte ordering
  - Representations
- Boolean algebra is mathematical basis
  - Basic form encodes "false" as 0, "true" as 1
  - General form like bit-level operations in C
    - Good for representing & manipulating sets

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
    - Useful in two's compliment
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument **x**	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument **x**	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Integer & Boolean algebra

- Integer Arithmetic
  - 〈Z, +, *, –, 0, 1〉 forms a mathematical structure called “ring”
  - Addition is “sum” operation
  - Multiplication is “product” operation
  - – is additive inverse
  - 0 is identity for sum
  - 1 is identity for product
- Boolean Algebra
  - 〈{0,1}, |, &, ~, 0, 1〉 forms a mathematical structure called “Boolean algebra”
  - Or is “sum” operation
  - And is “product” operation
  - ~ is “complement” operation (not additive inverse)
  - 0 is identity for sum
  - 1 is identity for product

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Boolean Algebra ≈ Integer Ring

Commutative	A | B = B | A A & B = B & A	A + B = B + A A * B = B * A
Associativity	(A | B) | C = A | (B | C) (A & B) & C = A & (B & C)	(A + B) + C = A + (B + C) (A * B) * C = A * (B * C)
Product distributes over sum	A & (B | C) = (A & B) | (A & C)	A * (B + C) = A * B + B * C
Sum and product identities	A | 0 = A A & 1 = A	A + 0 = A A * 1 = A
Zero is product annihilator	A & 0 = 0	A * 0 = 0
Cancellation of negation	~ (~ A) = A	– (– A) = A

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Boolean Algebra ≠ Integer Ring

Boolean: Sum distributes over product	A | (B & C) = (A | B) & (A | C)	A + (B * C) ≠ (A + B) * (B + C)
Boolean: Idempotency	A | A = A A & A = A	A + A ≠ A A * A ≠ A
Boolean: Absorption	A | (A & B) = A A & (A | B) = A	A + (A * B) ≠ A A * (A + B) ≠ A
Boolean: Laws of Complements	A | ~A = 1	A + –A ≠ 1
Ring: Every element has additive inverse	A | ~A ≠ 0	A + –A = 0

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Properties of & and ^

- Boolean ring
  - ⟨{0,1}, ^, &, I, 0, 1⟩
  - Identical to integers mod 2
  - I is identity operation: I (A) = A
    - A ^ A = 0
- Property: Boolean ring
  - Commutative sum — A ^ B = B ^ A
  - Commutative product — A & B = B & A
  - Associative sum — (A ^ B) ^ C = A ^ (B ^ C)
  - Associative product — (A & B) & C = A & (B & C)
  - Prod. over sum — A & (B ^ C) = (A & B) ^ (B & C)
  - 0 is sum identity — A ^ 0 = A
  - 1 is prod. identity — A & 1 = A
  - 0 is product annihilator — A & 0 = 0
  - Additive inverse — A ^ A = 0

Integer Representations

- Two different way bits can be used to encode integers;
  - Only represent nonnegative numbers
  - Represent negative, zero, and positive numbers

Integral Data Types

- C supports several integral data types (ones that represent finite ranges of integers)
- Based on the byte allocations, the different sizes allow different ranges of values to be presented (32-bit vs 64-bit)
  - Note the unsigned modifier
  - Also note the asymmetric ranges

Typical Ranges for C Integral Data Types for 32-bit Programs

C data type (32b)	Size	Minimum	Maximum
char	1	-128	127
unsigned char	1	0	255
short int	2	-32,768	32,767
unsigned short int	2	0	65,535
int	4	-2,147,438,648	2,147,438,647
unsigned int	4	0	4,294,967,295
long int	4	-2,147,438,648	2,147,438,647
unsigned long int	4	0	4,294,967,295
long long int	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long int	8	0	18,446,744,073,709,551,615

Typical Ranges for C Integral Data Types for 64-bit Programs

C data type	Minimum	Maximum
[signed] char	−128	127
unsigned char	0	255
short	−32,768	32,767
unsigned short	0	65,535
int	−2,147,483,648	2,147,483,647
unsigned	0	4,294,967,295
long	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long	0	18,446,744,073,709,551,615
int32_t	−2,147,483,648	2,147,483,647
uint32_t	0	4,294,967,295
int64_t	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint64_t	0	18,446,744,073,709,551,615

Unsigned Encodings

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

(Binary To Unsigned)

e.g. B2U ([1011]) = $1 * 2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 11$

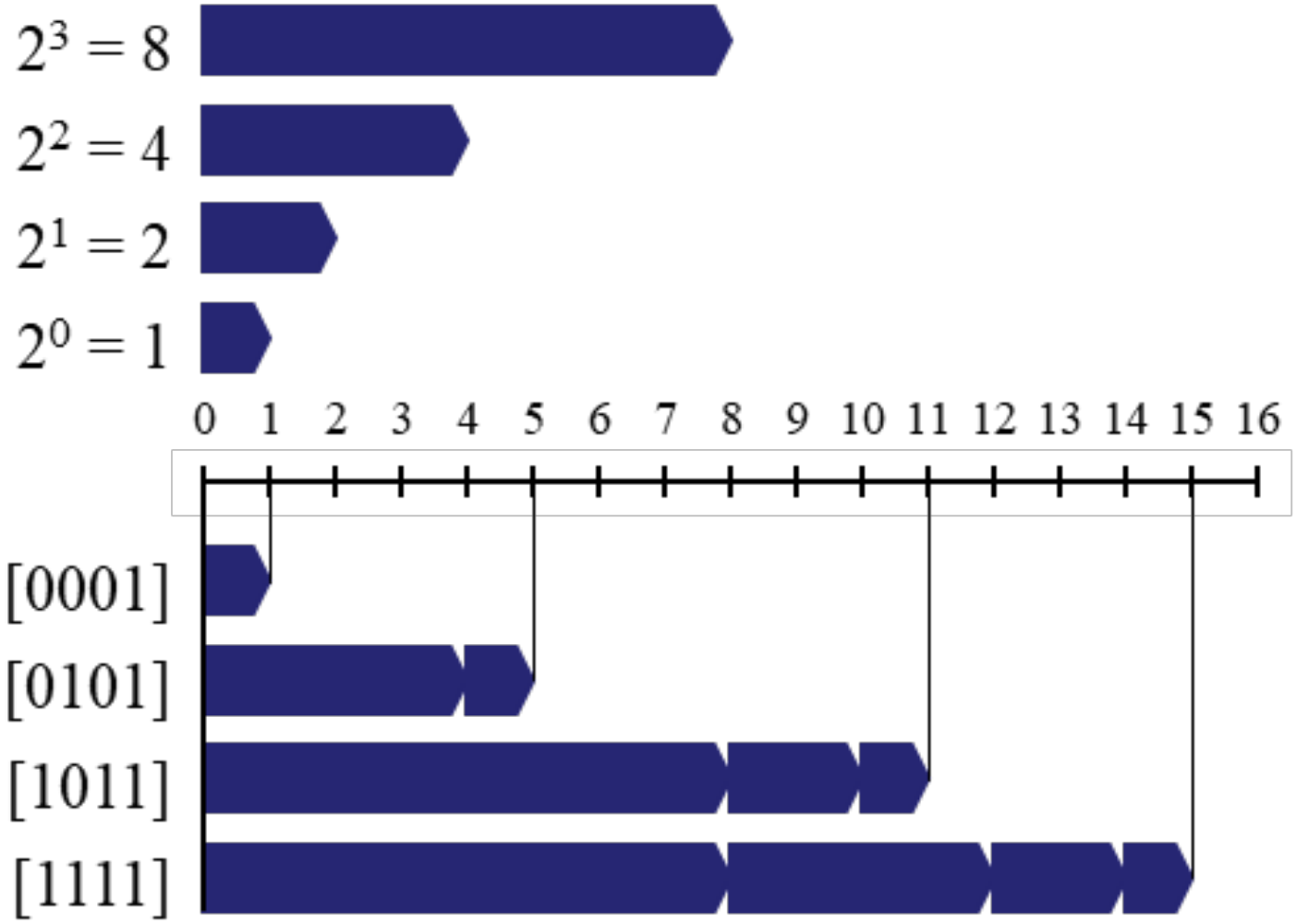
– C `short` 2 bytes long

```
short int x = 15213;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101

http://www.aqualab.cs.northwestern.edu/component/attachments/download/168

Examples


$2^3 = 8$
$2^2 = 4$
$2^1 = 2$
$2^0 = 1$
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
[0001]
[0101]
[1011]
[1111]

Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign Bit

- e.g. B2T ([1011]) = $-1 * 2^3 + 0*2^2 + 1*2^1 + 1*2^0 = -5$
- C `short` 2 bytes long

```
short int y = -15213;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

- Sign bit
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative; 1 for negative

http://www.aqualab.cs.northwestern.edu/component/attachments/download/168

Two-complement Encoding Example (Cont.)

```
x =      15213: 00111011 01101101
y =     -15213: 11000100 10010011
```

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum		**15213**		**-15213**

Two's Complement

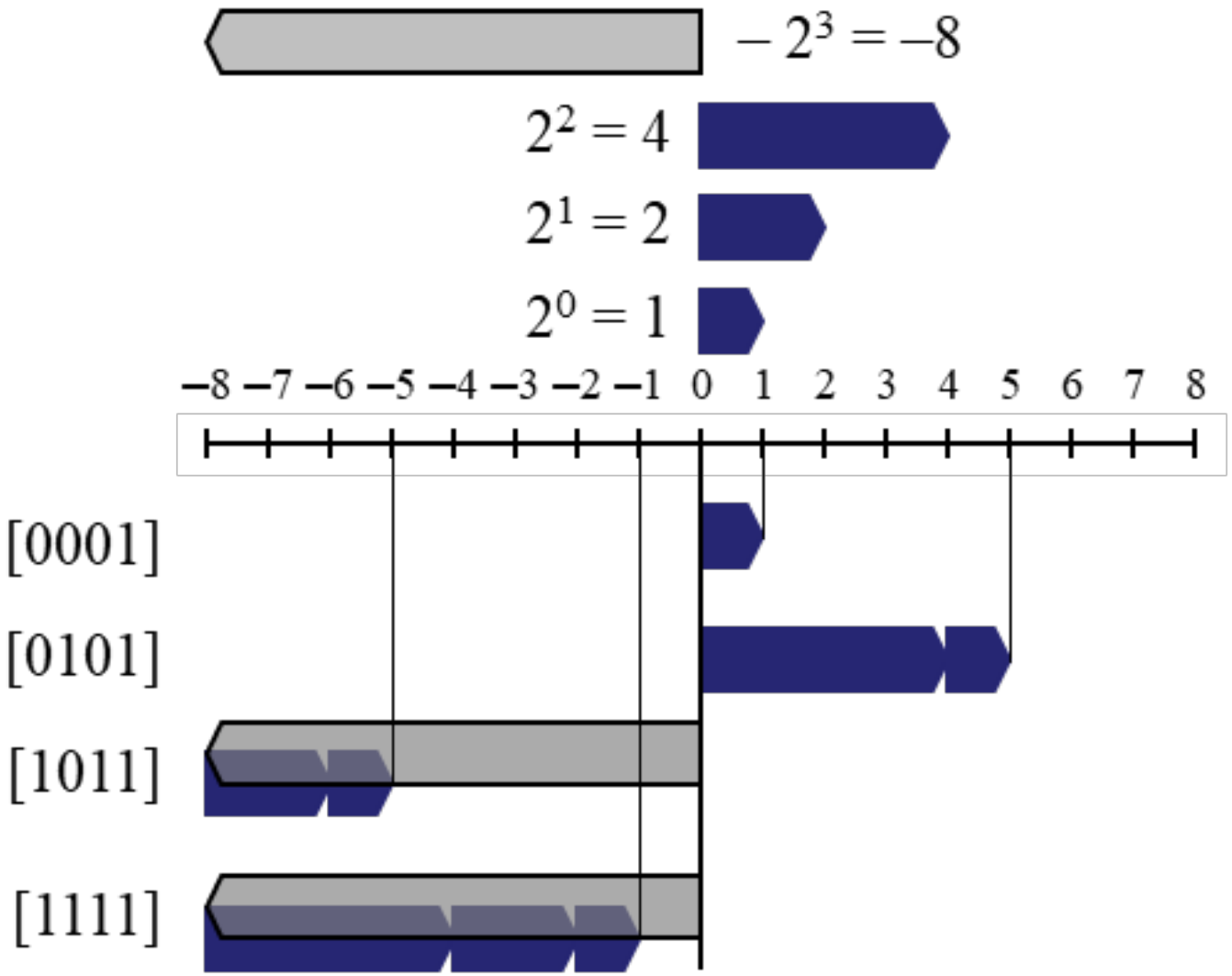
- Invert and add **one**

Suppose we're working with 8 bit quantities and suppose we want to find how **-28** would be expressed in two's complement notation.

- First we write out 28 in **binary form.**
  00011100
- Then we **invert the digits**. 0 becomes 1, 1 becomes 0.
  11100011
- Then we **add 1.**
  11100100

That is how one would write -28 in 8 bit binary.

https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html


$-2^3 = -8$
$2^2 = 4$
$2^1 = 2$
$2^0 = 1$
−8 −7 −6 −5 −4 −3 −2 −1 0 1 2 3 4 5 6 7 8
[0001]
[0101]
[1011]
[1111]

Characteristics of Twos Complement Representation and Arithmetic

Range	-2_{n-1} through $2_{n-1} - 1$
Number of Representations of Zero	One
Negation	Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
Expansion of Bit Length	Add additional bit positions to the left and fill in with the value of the original sign bit.
Overflow Rule	If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.
Subtraction Rule	To subtract *B* from *A*, take the twos complement of *B* and add it to *A*.

–128	64	32	16	8	4	2	1

(a) An eight-position two's complement value box

–128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

–128 +2 +1 = –125

(b) Convert binary 10000011 to decimal

–128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	0

–120 = –128 +8

(c) Convert decimal –120 to binary