Computer Systems Principles

Data Representation in C

UMASSCS

SCHOOL OF COMPUTER SCIENCE

Today

- Overflow and underflow
- Data representation in C
  - basic data types
  - cast
- Bit Manipulation
  - bitwise *and*, *or*, *exclusive-or*, and *not, shift*

OVERFLOW AND UNDERFLOW

Ariane 5

Exploded 37 seconds after lift-off with cargo worth 500 million

Why..

- Computed horizontal velocity as 64-bit floating-point number
- Converted to 16-bit integer
- Worked for Ariane 4
- Overflowed for Ariane 5

Two’s Complement Overflow & Underflow

- Overflow is caused by a value near the upper limit of the range, while an underflow is caused by values near the lower limit of the range.

Overflow: Example

Consider the 8-bit two's complement addition:

				1	1	1	1	1	1	
1	2	7	0	1	1	1	1	1	1	1
	+	1							+	1
1	2	8	1	0	0	0	0	0	0	0

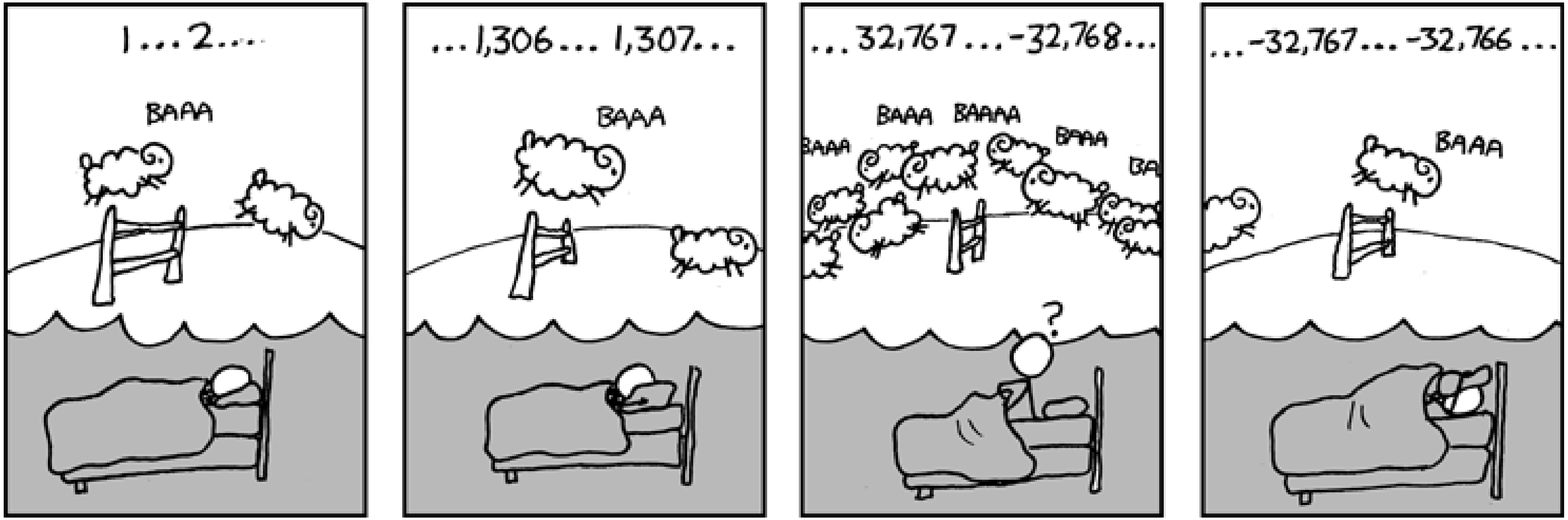
- The result should be +128, but the leftmost bit is 1, therefore the result is -128!
- This is an overflow: an arithmetic operation that should be positive gives a negative result.

Underflow: Example

Consider the 8-bit two’s complement addition:

-	1	2	8		1	0	0	0	0	0	0	0
		-	1	-								1
-	1	2	9		0	1	1	1	1	1	1	1

- The result should be -129, but the leftmost bit is 0, therefore the result is +127!
- This is an underflow: as an arithmetic operation that should be negative gives a positive result.



Is this dynamic ram??

Source: http://xkcd.com/571/

DATA TYPES IN C

Data types in C

```c
int x;
```

- first IBM PC: `int` [16 bits]
- today's PC: `int` [32 bits]
  (even on 64-bit PCs – but be careful!)

Data types in C (for gcc)

C Data Type	Typical 32-bit	Intel IA 32	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	4	8
`long long`	8	8	8
`float`	4	4	4
`double`	8	8	8
`long double`	8	10/12	10/16
`pointer`	4	4	8

Code Portability?

Notice that `long` and `pointer` data types are different on different processors (and maybe compilers).

A simple to print data type size

```
# include <stdio.h> // This is needed to run printf()
int main()
{
    int a;
    short int b;
    unsigned int c;
    char d;
    // size-of displays the size of the data type
    printf(“Size of int=%d bytes\n”,sizeof(a));
    printf(“Size of short int=%d bytes\n”,sizeof(b));
    printf(“Size of unsigned int=%d bytes\n”,sizeof(c));
    printf(“Size of char=%d bytes\n”,sizeof(d));
    return 0;
}
```

Casting between Signed and Unsigned

C allows conversions between signed (two's complement) and unsigned.

```
unsigned short int ux =  15213;
short int x           = (short int) ux;
short int y           = -15213;
unsigned short int uy = (unsigned short) y;
```

Resulting Value

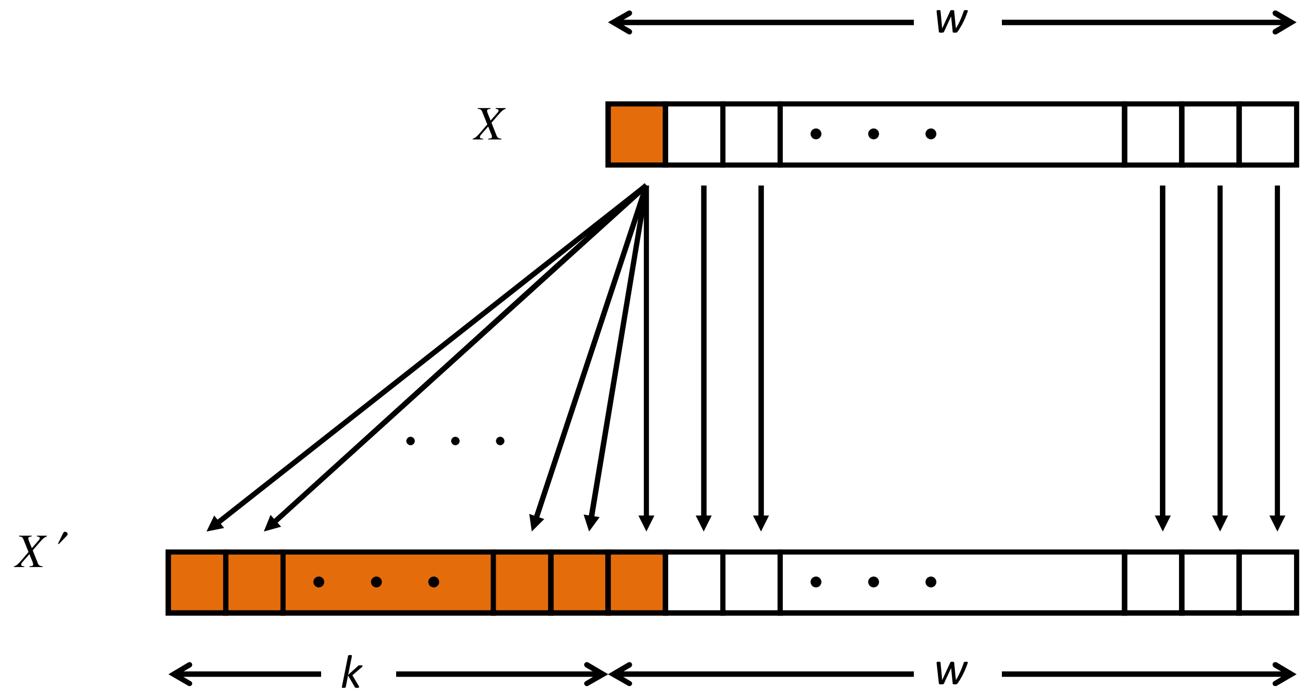
- No change in bit representation!
- Results reinterpreted

Signed vs. Unsigned in C

- Declaration for two signed and unsigned integers
  - ```
    int tx, ty;        // signed
    unsigned ux, uy; // unsigned
    ```
- Explicit casting between signed & unsigned
  - ```
    tx  = (int) ux;
    uy  = (unsigned) ty;
    ```
- Implicit casting also occurs via assignments and procedure calls
  - ```
    tx  =  ux;
    uy  =  ty;
    ```
    ```

Expanding Bit representation : Sign Extension

- Given w-bit signed integer x:
  - Convert it to $w+k$-bit integer with same value
  - Make k copies of sign bit: $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



Expanding Bit representation : Sign Extension

- Converting from smaller to larger integer data type
- C automatically performs sign- or zero- extension

```
short int sx =  -12345;
int    x   = sx;
unsigned short int usx = sx;
unsigned int ux = usx;
```

Variables	Value	Hexadecimal representation	Binary representation
sx	-12345	cf c7	11001111 11000111
usx	53191	cf c7	11001111 11000111
x	-12345	ff ff cf c7	11111111 11111111 11001111 11000111
ux	53191	00 00 cf c7	00000000 00000000 11001111 11000111

Expanding Bit representation : Sign Extension

- Converting from smaller to larger integer data type
- C automatically performs sign- or zero- extension

```
short int sx =  -12345;
int    x   = sx;
unsigned short int usx = sx
unsigned int ux = usx;
```

Expanding:
Unsigned: zero added
Signed: sign extension

Variables	Value	Hexadecimal representation	resentation
sx	-12345	cf c7	11001111 11000111
usx	53191	cf c7	11001111 11000111
x	-12345	ff ff cf c7	11111111 11111111 11001111 11000111
ux	53191	00 00 cf c7	00000000 00000000 11001111 11000111

Signed and unsigned integer constants in C

- By default an integer is assumed to be signed integer
  - Example: 3
- An integer constant may be suffixed by the letter u or U, to specify that it is unsigned.
  - Example: 3u

Casting Surprises: Expression evaluation

- If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!

Casting Surprises: Expression evaluation

- If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!
- Including comparison operations <,>,==,<=,>=
- E.g.: *W* = 32 TMIN = -2,147,483,648 (2^31) TMAX = 2,147,483,647 (2^31-1)

Constant-1	Relation	Constant-2	Evaluation
0		0u	
-1		0	
-1		0u	
2147483647		-2147483648	
2147483647u		-2147483648	
-1		-2	
(unsigned) -1		-2	
2147483647		2147483648u	
2147483647		(int) 2147483648u	

Casting Surprises: Expression evaluation

- If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!
- Including comparison operations <,>,==,<=,>=
- E.g.: *W* = 32 TMIN = -2,147,483,648 (2^31) TMAX = 2,147,483,647 (2^31-1)

Constant-1	Relation	Constant-2	Evaluation
0	==	0u	unsigned
-1		0	
-1		0u	
2147483647		-2147483648	
2147483647u		-2147483648	
-1		-2	
(unsigned) -1		-2	
2147483647		2147483648u	
2147483647		(int) 2147483648u	

Casting Surprises: Expression evaluation

- If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!
- Including comparison operations <,>,==,<=,>=
- E.g.: *W* = 32 TMIN = -2,147,483,648 (2^31) TMAX = 2,147,483,647 (2^31-1)

Constant-1	Relation	Constant-2	Evaluation
0	==	0u	unsigned
-1	<	0	signed
-1		0u	
2147483647		-2147483648	
2147483647u		-2147483648	
-1		-2	
(unsigned) -1		-2	
2147483647		2147483648u	
2147483647		(int) 2147483648u	

Casting Surprises: Expression evaluation

- If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!
- Including comparison operations <,>,==,<=,>=
- E.g.: *W* = 32 TMIN = -2,147,483,648 (2^31) TMAX = 2,147,483,647 (2^31-1)

Constant-1	Relation	Constant-2	Evaluation
0	==	0u	unsigned
-1	<	0	signed
-1	>	0u	unsigned
2147483647		-2147483648	
2147483647u		-2147483648	
-1		-2	
(unsigned) -1		-2	
2147483647		2147483648u	
2147483647		(int) 2147483648u	

Casting Surprises: Expression evaluation

- If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!
- Including comparison operations <,>,==,<=,>=
- E.g.: *W* = 32 TMIN = -2,147,483,648 (2^31) TMAX = 2,147,483,647 (2^31-1)

Constant-1	Relation	Constant-2	Evaluation
0	==	0u	unsigned
-1	<	0	signed
-1	>	0u	unsigned
2147483647	>	-2147483648	signed
2147483647u		-2147483648	
-1		-2	
(unsigned) -1		-2	
2147483647		2147483648u	
2147483647		(int) 2147483648u	

Casting Surprises: Expression evaluation

- If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!
- Including comparison operations <,>,==,<=,>=
- E.g.: *W* = 32 TMIN = -2,147,483,648 (2^31) TMAX = 2,147,483,647 (2^31-1)

Constant-1	Relation	Constant-2	Evaluation
0	==	0u	unsigned
-1	<	0	signed
-1	>	0u	unsigned
2147483647	>	-2147483648	signed
2147483647u	<	-2147483648	unsigned
-1		-2	
(unsigned) -1		-2	
2147483647		2147483648u	
2147483647		(int) 2147483648u	

Casting Surprises: Expression evaluation

- If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!
- Including comparison operations <,>,==,<=,>=
- E.g.: *W* = 32 TMIN = -2,147,483,648 (2^31) TMAX = 2,147,483,647 (2^31-1)

Constant-1	Relation	Constant-2	Evaluation
0	==	0u	unsigned
-1	<	0	signed
-1	>	0u	unsigned
2147483647	>	-2147483648	signed
2147483647u	<	-2147483648	unsigned
-1	>	-2	signed
(unsigned) -1		-2	
2147483647		2147483648u	
2147483647		(int) 2147483648u	

Casting Surprises: Expression evaluation

- If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!
- Including comparison operations <,>,==,<=,>=
- E.g.: *W* = 32 TMIN = -2,147,483,648 (2^31) TMAX = 2,147,483,647 (2^31-1)

Constant-1	Relation	Constant-2	Evaluation
0	==	0u	unsigned
-1	<	0	signed
-1	>	0u	unsigned
2147483647	>	-2147483648	signed
2147483647u	<	-2147483648	unsigned
-1	>	-2	signed
(unsigned) -1	>	-2	unsigned
2147483647		2147483648u	
2147483647		(int) 2147483648u	

i-clicker question

What is the relationship between two integer 2147483647 and 2147483648u?

Assume each integer has 32 bit. MINIMUM = -2,147,483,648 (2^31) MAXIMUM = 2,147,483,647 (2^31-1)

A. ==
B. <
C. >

i-clicker question

What is the relationship between two integer 2147483647 and (int) 2147483648u?

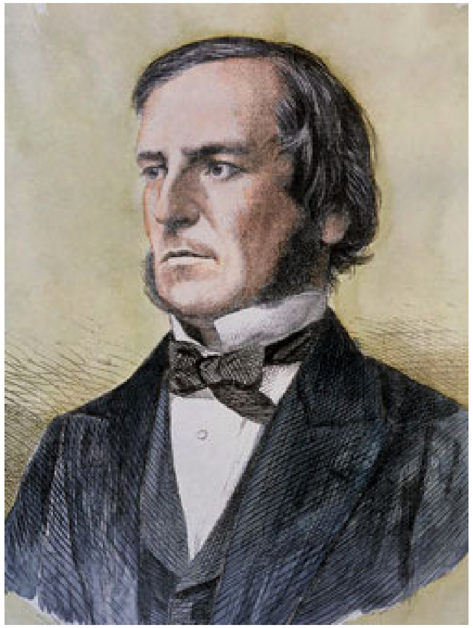
Assume each integer has 32 bit. MINIMUM = -2,147,483,648 (2^31) MAXIMUM = 2,147,483,647 (2^31-1)

A. ==
B. <
C. >

BOOLEAN ALGEBRA

Boolean Algebra

- Developed by George Boole in the 19th Century and applied to Digital Systems by Claude Shannon

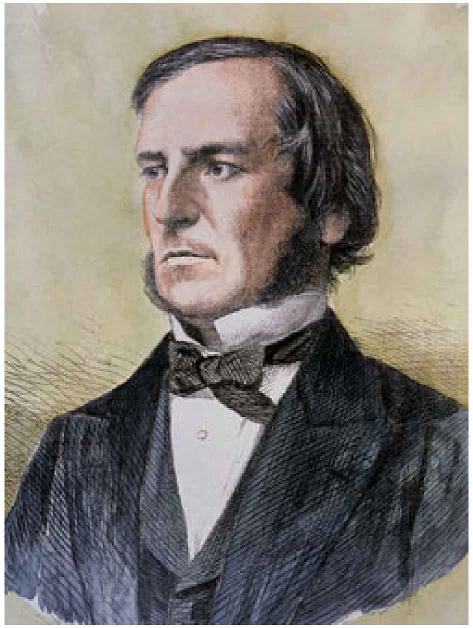


"Laws of Thought"
image source: Wikipedia

Bit-Manipulations

Boolean Algebra:

- Developed by George Boole in the 19th Century and applied to Digital Systems by Claude Shannon
- Encode “True”/”On”/”Yes” as 1 and “False”/”Off”/”No” as 0



“Laws of Thought”
image source: Wikipedia

Bit-Manipulations

Not (~A)

~	
0	1
1	0

Bit-Manipulations

Not (~A)

~	
0	1
1	0

And (A & B)

&	0	1
0	0	0
1	0	1

Bit-Manipulations

Not (~A)

~	
0	1
1	0

And (A & B)

&	0	1
0	0	0
1	0	1

Or (A|B)

<table>
<tr><th>|</th><th>0</th><th>1</th></tr>
<tr><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>

Bit-Manipulations

Not (~A)

~	
0	1
1	0

And (A & B)

&	0	1
0	0	0
1	0	1

Or (A|B)

|	0	1
0	0	1
1	1	1

Xor A^ B

^	0	1
0	0	1
1	1	0

Bit-Manipulations

Boolean operations are applied *bitwise* on the bit sequences (i.e., by columns)

Not (~A)	And (A & B)	Or (A|B)	Xor A^ B
	0 1 1 0	0 1 1 0	0 1 1 0
~ 1 0 1 0	& 1 0 1 0	I 1 0 1 0	^ 1 0 1 0
0 1 0 1	0 0 1 0	1 1 1 0	1 1 0 0

Bit Manipulations

Boolean algebra obeys some of the properties of integer algebra.. but not all!

Boolean	Boolean	Integer
Sum and product identities	$A \mid 0 = A$ $A \& 1 = A$	$A+0 = A$ $A*1 = A$
Zero is product annihilator	$A \& 0 = 0$	$A *0 = 0$
Cancellation of negation	$\sim(\sim A) = A$	$-(-A) = A$
Laws of Complements	$A \mid \sim A = 1$	$A + -A \neq 1$
Every element has an additive inverse	$A \mid \sim A \neq 0$	$A + -A = 0$

SHIFT OPERATION

Shift unsigned integers

- Left shift x<<y
  - Discard bits on the left
  - Fill with 0s on the right
  - 00000010<<2=00001000
- Right shift x>>y
  - Discard bits on the right
  - Fill with 0s on the left
  - 10000010>>3=00010000

Shift unsigned integers

- Left shift x<<y
  - Discard bits on the left
  - Fill with 0s on the right
  - 00000010<<2=00001000
- Right shift x>>y
  - Discard bits on the right
  - Fill with 0s on the left
  - 10000010>>3=00010000

Shift unsigned integers

- Left shift x<<y
  - Discard bits on the left
  - Fill with 0s on the right
  - 00000010<<2=00001000
- Right shift x>>y
  - Discard bits on the right
  - Fill with 0s on the left
  - 10000010>>3=00010000

Shift unsigned integers

- Left shift x<<y
  - Discard bits on the left
  - Fill with 0s on the right
  - 00000010<<2=00001000
- Right shift x>>y
  - Discard bits on the right
  - Fill with 0s on the left
  - 10000010>>3=00010000

Shift unsigned integers

- Left shift x<<y
  - Discard bits on the left
  - Fill with 0s on the right
  - 2 00000010<<2=00001000 8
- Right shift x>>y
  - Discard bits on the right
  - Fill with 0s on the left
  - 10000010>>3=00010000
- Left shift y equivalent to multiplying by 2^y

Shift unsigned integers

- Left shift x<<y
  - Discard bits on the left
  - Fill with 0s on the right
  - 00000010<<2=00001000
- Right shift x>>y
  - Discard bits on the right
  - Fill with 0s on the left
  - 130 10000010>>3=00010000 16
- Left shift y equivalent to multiplying by 2^y
- Right shift y equivalent to dividing by 2^y

Shift signed integers

- Left shift: x << k : Shift bit-vector x left by k positions
  - Throw away extra bits on the left
  - Fill with 0's on the right.
  - x << k is equivalent to x * 2^k

Bit Manipulations: shift operators

- Right shift: x >> k : Shift bit-vector x right k positions.
  - Throw away extra bits on the right

TWO KINDS:

- Logical Shift: Fill with 0’s on the left.
- Arithmetic Shift : Replicate with most significant bit on the left.
  - Copies the sign bit
  - Arithmetic shift is equivalent to logical shift for positive numbers

Bit Manipulations: shift operators

- Right shift: x >> k : Shift bit-vector x right k positions.
  - Throw away extra bits on the right

TWO KINDS:

- Logical Shift: Fill with 0’s on the left.
- Arithmetic Shift : Replicate with most significant bit on the left.
  - Copies the sign bit
  - Arithmetic shift is equivalent to logical shift for positive numbers
  - 0100 1000 >> 2 = 0001 0010

Bit Manipulations: shift operators

- Right shift: x >> k : Shift bit-vector x right k positions.
  - Throw away extra bits on the right

TWO KINDS:

- Logical Shift: Fill with 0's on the left.
- Arithmetic Shift : Replicate with most significant bit on the left.
  - Copies the sign bit
  - Arithmetic shift is equivalent to logical shift for positive numbers
  - 0100 1000 >> 2 = 0001 0010
  - 1001 0001 >> 3 = 1111 0010
  - x >> k corresponds to $x/2^k$ for rounding.
    - 1001 0001 >> 3 in decimal: (-111) / 2^3 = -13.875
    - 1111 0010 in decimal: -14

Bit Manipulations: shift operators

- Right shift: x >> k : Shift bit-vector x right k positions.
  - Throw away extra bits on the right

TWO KINDS:

- Logical Shift: Fill with 0's on th
- Arithmetic Shift : Replicate ... eft.
  - Copies the sign bit
  - Arithmetic shift is equivalent to
  - 0100 1000 >> 2 = 0001 0010
  - 1001 0001 >> 3 = 1111 0010
  - x >> k corresponds to $x/2^k$ for rounding.
    - 1001 0001 >> 3 in decimal: (-111) / 2^3 = -13.875
    - 1111 0010 in decimal: -14

$x<<y$ or $x>>y$
undefined when $y<0$
or $y>=$# bits of x

Comparison with shifting in Java

Both use << to shift left

Comparison with shifting in Java

Both use << to shift left

C:

- Has signed and unsigned integer

Java:

- Has only *signed* integer

Comparison with shifting in Java

Both use << to shift left

C:

- Has *signed* and *unsigned* integer
- Shift operator (>>) is implementation defined
- In our Virtual Machine, >> operates according to the *type* of the operand
  - When shifting an unsigned value, >> is a logical shift.
  - When shifting a signed value, >> is an arithmetic shift.

Java:

- Has only *signed* integer
- >> is arithmetic shift, >>> is logical shift

iClicker Question

Compute this *arithmetic* right shift:

1001 0001 >> 2

a) 1111 0010
b) 1110 0100
c) 1110 0101
d) 0010 0100

APPLICATION OF SHIFT OPERATION

Ranges of bits

Sometimes you will encounter situations where multiple smaller numbers are *packed* into a single larger one. Some reasons for this are:

- To save space in memory
- To fit a quantity into a single register
- Because some hardware was designed that way and you have to talk to it
- To save sending bits over a network or to/from a device (such as a disk)

This leads to requirements to *extract* ranges of bits in a number, and to *update* ranges of bits.

Extracting a range of bits: Method 1

- Number bits starting from 0 on the right:
  bit7 bit6 bit5 bit4 bit3 bit2 bit1 bit0
- Suppose you want bits 2 through 4
- <u>Step 1</u>: Isolate the bits using a *mask*:
  Bitwise And (&): b7 b6 b5 b4 b3 b2 b1 b0
  With 0 0 0 1 1 1 0 0
- The mask has 1 bits for the positions you want
  - And 0 elsewhere
- <u>Step 2</u>: Shift masked value right to get rid of unwanted 0 bits on the right.
  - In this case, >> 2

Extracting bits L through R

- Mask has L-R+1 bits that are 1
  - Can form by: (1 << (L-R+1)) – 1
    - … or by writing it out
  - For L == 4 and R == 2, we have (1 << 3) – 1
    - In binary: 1 << 3 is 00001000
    - Subtract 1 and you get: 00000111
- It is shifted left by R bits
  - In this case ((1 << 3) – 1) << 2
  - In binary, shift 00000111 left by 2: 00011100

Whole sequence in C

```
int mask = ((1 << 3) - 1) << 2;
// or: int mask = 0b00011100;
// (…b is a gcc extension to C)
int range =
  (unsigned)((full & mask)) >> 2
```

Whole sequence in C

```
int mask = ((1 << 3) - 1) << 2;
// or: int mask = 0b00011100;
// (…b is a gcc extension to C)
int range =
   (unsigned)((full & mask)) >> 2
```

Why do we cast signed int to unsigned int?

Extracting a range of bits: Method 2

- Number bits starting from 0 on the right:
  bit7 bit6 bit5 bit4 bit3 bit2 bit1 bit0
- Suppose you want bits 2 through 4
- Step 1: Shift left, eliminating unwanted high bits:
  b7 b6 b5 b4 b3 b2 b1 b0 << 3
  b4 b3 b2 b1 b0 0 0 0
- Step 2: Shift right logical to get desired bits.
  b4 b3 b2 b1 b0 0 0 0 >> 5
  0 0 0 0 0 b4 b3 b2

  Desired field may be signed, or unsigned; cast to the right type *before* shifting right.

Extracting bits L through R

- Shift left by n-(L+1) bits
- Shift right by n-(L-R+1) bits
- n is the number of bits in the data type (8, 16, 32, 64)
- For L == 4 and R == 2, with n == 8, we have

  (x << 3) >> 5

Extracting bits L through R

- Shift left by n-(L+1) bits
- Shift right by n-(L-R+1) bits
- n is the number of bits in the data type (8, 16, 32, 64)
- For L == 4 and R == 2, with n == 8, we have

  (x << 3) >> 5

Both methods are equally good

Updating bits L through R of N with M

You can assume that the bits L through R of N have enough space to fit all of M.

For example, you are given a 11-bit number N=10000000000 and a 5-bit number M=10011, update N such that M starts at bit L=2 and ends at bit R=6.
Output:
N = 10001001100

Updating bits L through R of N with M

Steps:

- Clear the bits L through R in N
- Shift M so that it lines up with bits L through R
- merge M and N

Updating bits L through R of N with M

Steps:

- Clear the bits L through R in N
- Shift M so that it lines up with bits L through R
- merge M and N

Updating bits L through R of N with M

Steps:

- Clear the bits L through R in N
- Shift M so that it lines up with bits L through R
- merge M and N
- In C:

```
int mask = ((1 << (L-R+1) - 1) << R);
int newN = N & (~mask);
int newM = M << R;
int result = newN | newM;
```

Updating bits L through R of N with M

Steps:

- Clear the bits L through R in N
- Shift M so that it lines up with bits L through R
- merge M and N
- In C:

```
int mask = ((1 << (L-R+1) - 1) << R);
int newN = N & (~mask);
int newM = M << R;
int result = newN | newM;
```

Updating bits L through R of N with M

Steps:

- Clear the bits L through R in N
- Shift M so that it lines up with bits L through R
- merge M and N
- In C:

```
int mask = ((1 << (L-R+1) - 1) << R);
int newN = N & (~mask);
int newM = M << R;
int result = newN | newM;
```

Updating bits L through R of N with M

- Let L == 4, R == 2, M = 0b<u>101</u>
- Let N = 0b 011 <u>011</u> 01

```
int mask = ((1 << (L-R+1)) - 1) << R);
// mask == 0b00011100
int newN = N & (~mask);
// newN == 0b01100001
int newM = M << R;
// M << R == 0b10100
int result = newM | newN;
// result == 0b01110101
```

Summary

- Bit representation and manipulation is extremely useful in a wide variety of applications like compiler analyses, network programming, cryptography and many more!
- The same binary sequence can be used to represent ASCII characters, unsigned binary, and two's complement integers. Their interpretation is based on the context in which they are defined!
- C has different data types to store integers and floating point numbers that have different memory sizes on different operating systems.
- Typecasting operations between two different data types can be explicit or implicit.
  - Casting surprises when changing between data types can change the numeric value.
  - Casting surprises also occur if we use arithmetic and relational operators on two different data types.
- Boolean algebra includes {not, and, or and x-or} operations and left and right shifts.
  - Not to be confused with conditional operators !
- Using &, |, <<, and >> you can extract and replace ranges of bits using masks
- Next class we will cover more programming in C!