CS 33 – Computer Organization

Week 0 Discussion

Friday, 29 September 2017

Slides adapted from Uen-Tao Wang

Contact Info

- TA: Brian Hill
- Email: blhill@ucla.edu
- Office Hours: Tues/Wed 10:30-11:30am (tentative)
- TA Office Hours Location: BH 2432

Schedule

- Administrative information
- Linux overview and accessing the SEASnet Linux servers
- C (aka unlearning C++)
- Binary representation
- Binary operators

Course Administration

- Course Website: http://web.cs.ucla.edu/classes/fall17/cs33/
- Syllabus link available on course website
- Professor Eggert's Office Hours: Mon 2-3pm, Thur 10-11am
- Textbook: Computer Systems: A Programmer's Perspective (3rd edit), Randal Bryant & David O'Hallaron
  - Note: 2^{nd} edition homework problems are different numbers than 3^{rd}
- Grading:

5 Homeworks	5%	(1% each)
4 Labs	40%	
2 Midterms	25%	(12.5% each)
1 Final Exam	30%	

Getting Started

- This class is based around C, not C++.
- As a result, you are highly recommended to ditch Visual Studio and work in a Linux environment, specifically the SEASnet Linux servers.
- Your assignments will be tested on the SEASnet Linux servers
  - `lnxsrv06, lnxsrv07, lnxsrv09` have newer version of gcc you'll want to use
- The class lectures are likely to be Linux heavy.
- Linux is love. Linux is life.

Source: http://apprize.info/linux/pic.jpg

Getting Started: Accessing the SEASnet

- To login to SEASnet you need to be connected to wireless network on campus

OR

- Login with VPN Software

https://www.it.ucla.edu/bol/services/virtual-private-network-vpn-clients

Getting Started: Accessing the SEASnet

- SSH stands for Secure Shell and is a protocol that is used to initiate text based access to a remote server.
- For Windows users:
  - PuTTY (http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html): An SSH client
- For Mac/Linux:
  - SSH is a command that can be issued directly. Open a terminal (for Mac: Applications -> Utilities -> Terminal)

Getting Started: Accessing the SEASnet

- For Windows users:
  - In “host name”, enter [username]@lnxsrv.seas.ucla.edu
- For Mac and Linux:
  - ssh [username]@lnxsrv.seas.ucla.edu
- If you are in the Henry Samueli School of Engineering, you should already have a SEASnet account. Otherwise go to the SEASnet office at 2684 Boelter Hall to get one.

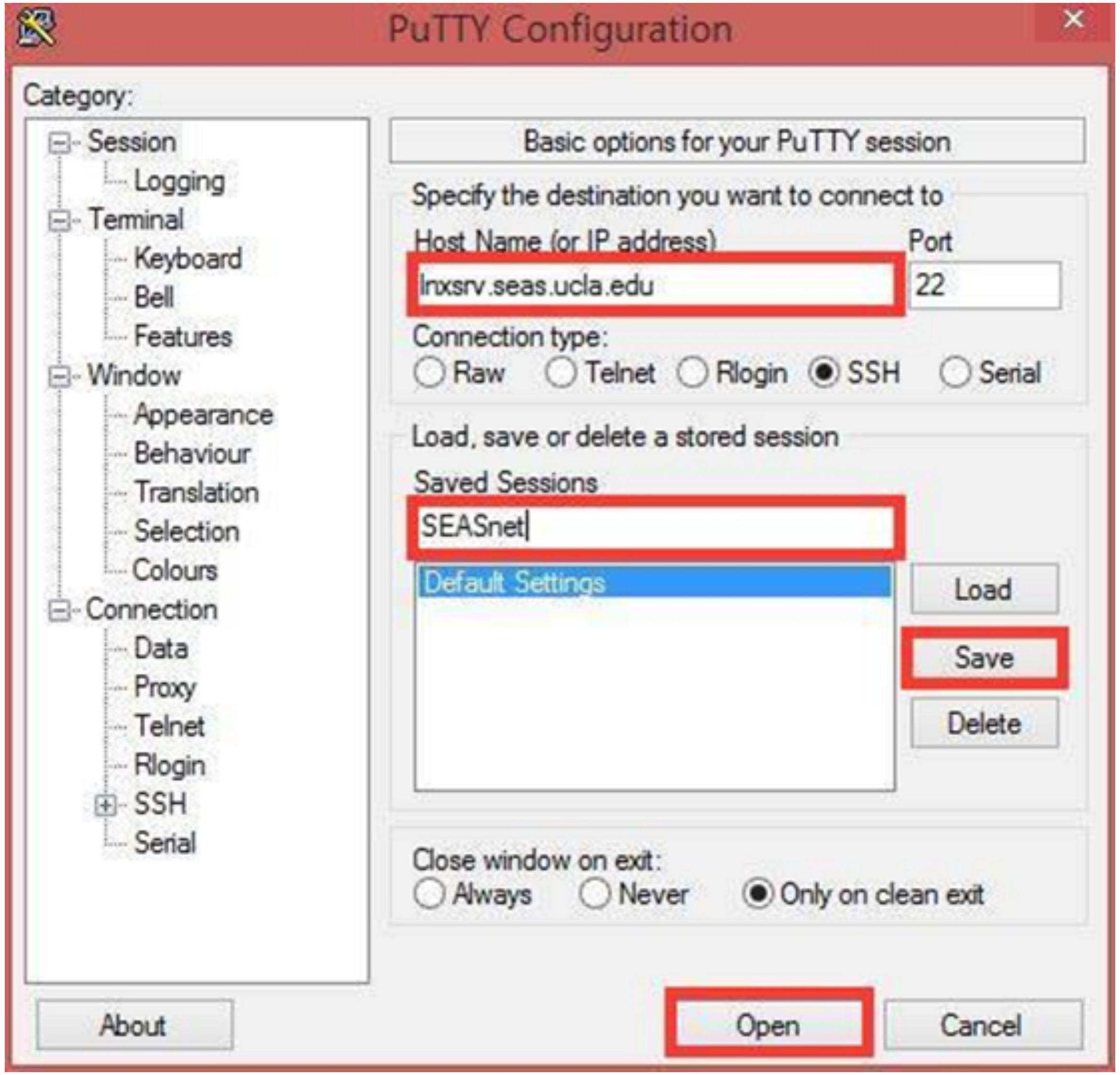
PuTTY

First Run

- Type *lnxsrv@seas.ucla.edu* for Host Name
- Type SEASnet for Saved Sessions
- Click Save
- Click Open
- Type your SEASnet username and password

Double-click SEASnet under Saved Sessions

in the future



Getting Started: Useful Linux Commands

- Typical Linux command format: [command name] -X -Y -Z [argument]
  - (X, Y, and Z are optional flags)
- Flags modify/specify the behavior of the command.
- Ex:
  - `ls` Lists files in current directory
  - `ls -l` Lists files in current directory, in “long” format
- **VERY** useful command: `man <command name>`
  - Opens manual page for the command
  - Ex: `man ls`

Getting Started: Useful Linux Commands

Command	Example	Explanation
`pwd`	`pwd`	Print the working directory (current directory)
`ls [flags] [path to directory]`	`ls -l /tmp`	List contents of directory
`cd <path to directory>`	`cd /tmp`	Change working directory to argument
`mkdir [new directory name]`	`mkdir /tmp/test_folder`	Create new directory
`rm [flags] [file/directory]`	`rm -rf /tmp/test_folder`	Remove file/directory (be careful!!)
`exit`	`exit`	Exit terminal

Getting Started: Useful Linux Commands

- Editing files. If you're interested familiarizing yourselves with Linux (which will have to happen eventually), it is recommended that you use “vim” or “emacs”.
  - vim text.txt
  - emacs text.txt
- Useful vim commands:

`i`	Insert mode (for editing text)
`esc (Escape)`	Get out of “insert mode”
`:q`	Exit vim
`:wq`	Write changes and exit vim
`:q!`	Do NOT save changes and exit vim

Getting Started: Useful Linux Commands

- The standard Linux C compiler is gcc
  - `gcc main.c` (compile the file main.c into an executable file with default name “a.out”)
  - `gcc main.c -o main` (compile the file main.c into an executable file called “main”)
  - `gcc main.c -O2` (compile the file with optimizations, level 2)
  - `gcc -S main.c` (dump assembly code)
  - `gcc -E main.c` (show code after pre-processing)
- Executing executables
  - ./main (executes the executable file called “main”)

C (as opposed to C++)

- In a (very simplified) nutshell, C++ is an extension to C.

- The syntax of the language is nearly identical, but you will find that C lacks certain features, namely the “Object Oriented” paradigm.

- Some features are analogous, but have different names.

C (as opposed to C++)

- In C++:

– for(int i = 0; i < size; i++)

- By default, gcc uses a 1990's C standard which prohibits declarations in “for” loops. As a result, you will have to do either

– int i;

– for(i = 0; I < size; i++)

- Or explicitly use gcc to compile with a different C standard

– gcc -std=c99 temp.c

C (as opposed to C++)

Dynamic memory allocation

In C++:

– char * c_arr = new char[10];

– delete c_arr;

– “new” allows you to specify repetitions of a specific data type.

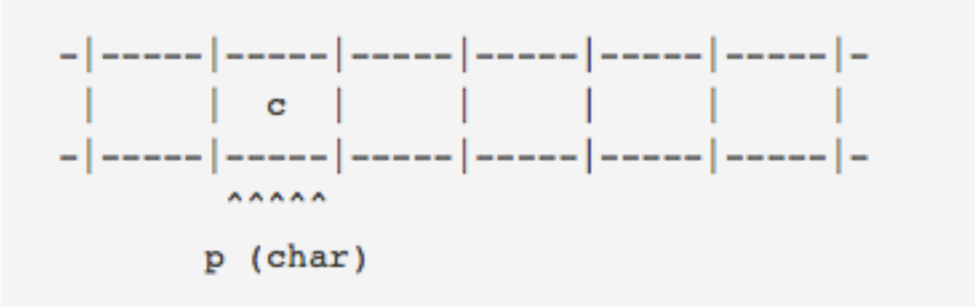
C (as opposed to C++)

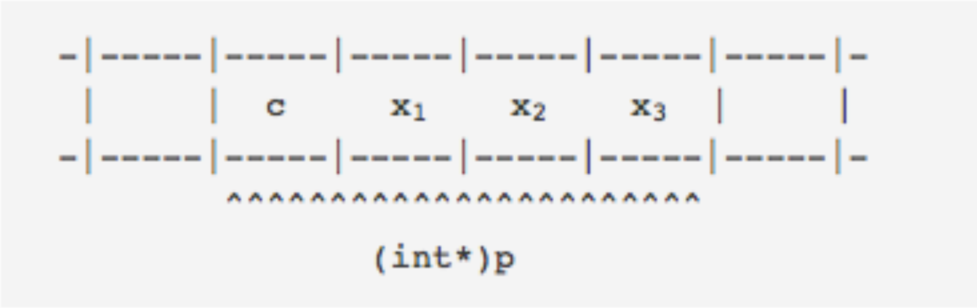
- In C, these declarations force you to be more specific. Instead of “new”, use “malloc” and instead of “delete”, use “free”.
  - char * c_arr = (char *) malloc(sizeof(char) * 10);
  - free(c_arr);
- Note: These are analogous but not the same.
- “malloc” and other “_alloc” variations operate on the principle that you're specifying a specific amount of memory to allocate rather than a specific data type.

C (as opposed to C++)

- Pointer casting: doesn’t change address pointed to, but how we interpret data at that address
- Ex:
  - char * c_arr = (char *) malloc(sizeof(char) * 10);
- Casts void pointer (void *) to (char *)

- Example on right shows difference between casting pointer as (char *) vs (int *)





Source: https://prateekvjoshi.com/2014/08/02/what-is-pointer-casting/

C (as opposed to C++)

- Instead of:
  - int x = 10;
  - cout << x;
- You'll use “printf”
  - printf(“hello”);
  - printf(“%d”, x);
- printf takes in as the first parameter a string to print out that is populated with format codes that correspond to the remaining arguments.

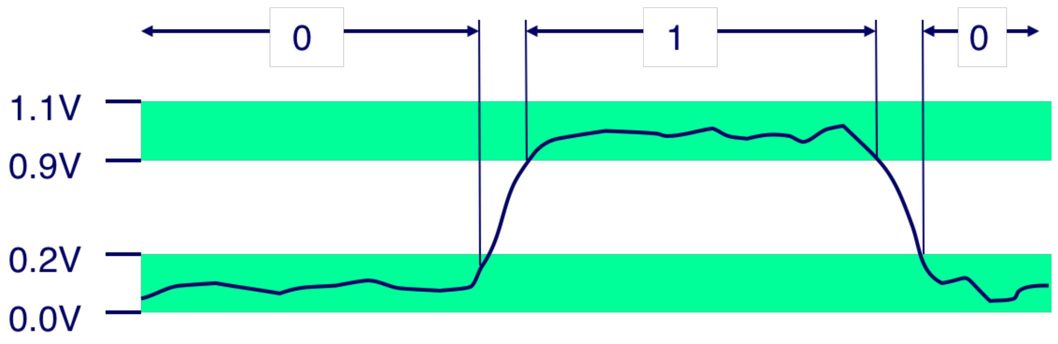
Documentation: http://www.cplusplus.com/reference/cstdio/printf/

-fwrapv and -ftrapv Flags

- fwrapv - instructs the compiler to assume that signed arithmetic overflow of addition, subtraction and multiplication wraps around using twos-complement representation. Enabled by default for Java.
- ftrapv - generates traps for signed overflow on addition, subtraction, multiplication operations

Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers, sets, strings, etc…
- Why bits? Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



For example, can count in binary

- Base 2 Number Representation
  - Represent 15213_{10} as 11101101101101_2
  - Represent 1.20_{10} as $1.0011001100110011[0011]\ldots\ldots\ldots_2$
  - Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$

Encoding Integers

Unsigned

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

Two's Complement

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

Sign Bit

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

- C short 2 bytes long

	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

Two's 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**

Numeric Ranges

Unsigned Values

- *UMin* = 0
  - 000…0
- *UMax* = $2^w - 1$
  - 111…1

- **Two's Complement Values**
  - *TMin* = -2^{w-1}
    - 100…0
  - *TMax* = $2^{w-1} - 1$
    - 011…1
- **Other Values**
  - Minus 1
    - 111…1

Values for *W* = 16

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Values for Different Word Sizes

	W			
	8	**16**	**32**	**64**
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

Observations

- $|TMin| = TMax + 1$
- $UMax = 2 * TMax + 1$

C Programming

- #include <limits.h>
- Declares constants, e.g.,
  - ULONG_MAX
  - LONG_MAX
  - LONG_MIN
- Values platform specific

Unsigned & Signed Numeric Values

X	**B2U(*X*)**	**B2T(*X*)**
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

Equivalence

- Same encodings for nonnegative values

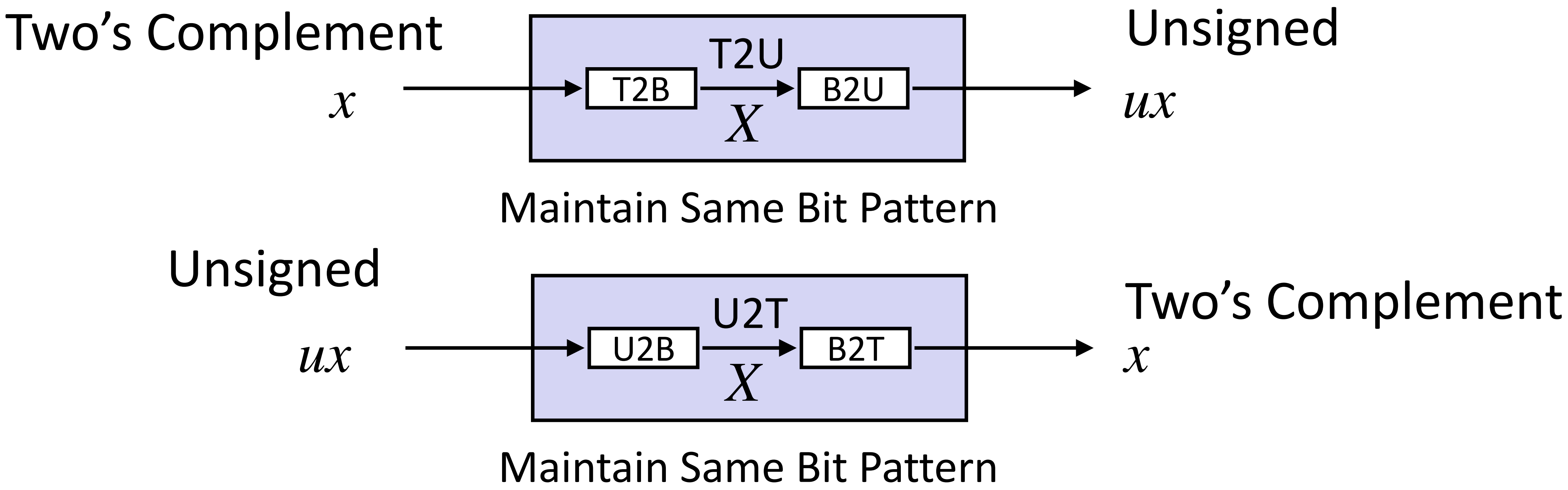
Uniqueness

- Every bit pattern represents unique integer value
- Each representable integer has unique bit encoding

⇒ **Can Invert Mappings**

- U2B(x) = B2U^{-1}(x)
  - Bit pattern for unsigned integer
- T2B(x) = B2T^{-1}(x)
  - Bit pattern for two's comp integer

Mapping Between Signed & Unsigned



Mappings between unsigned and two's complement numbers:
Keep bit representations and reinterpret

Mapping Signed ↔ Unsigned

Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

→ T2U →

← U2T ←

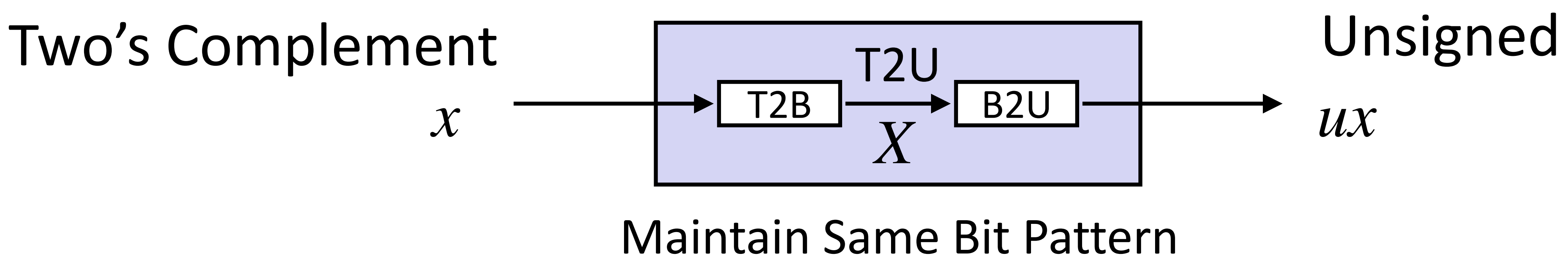
Mapping Signed ↔ Unsigned

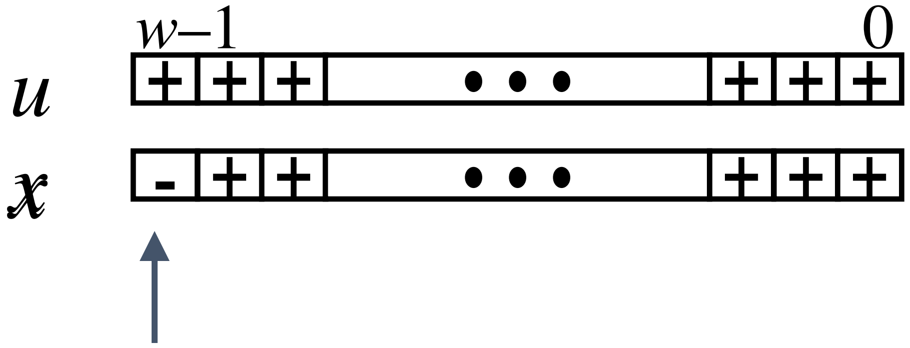
Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

=

+/-
16

Relation between Signed & Unsigned



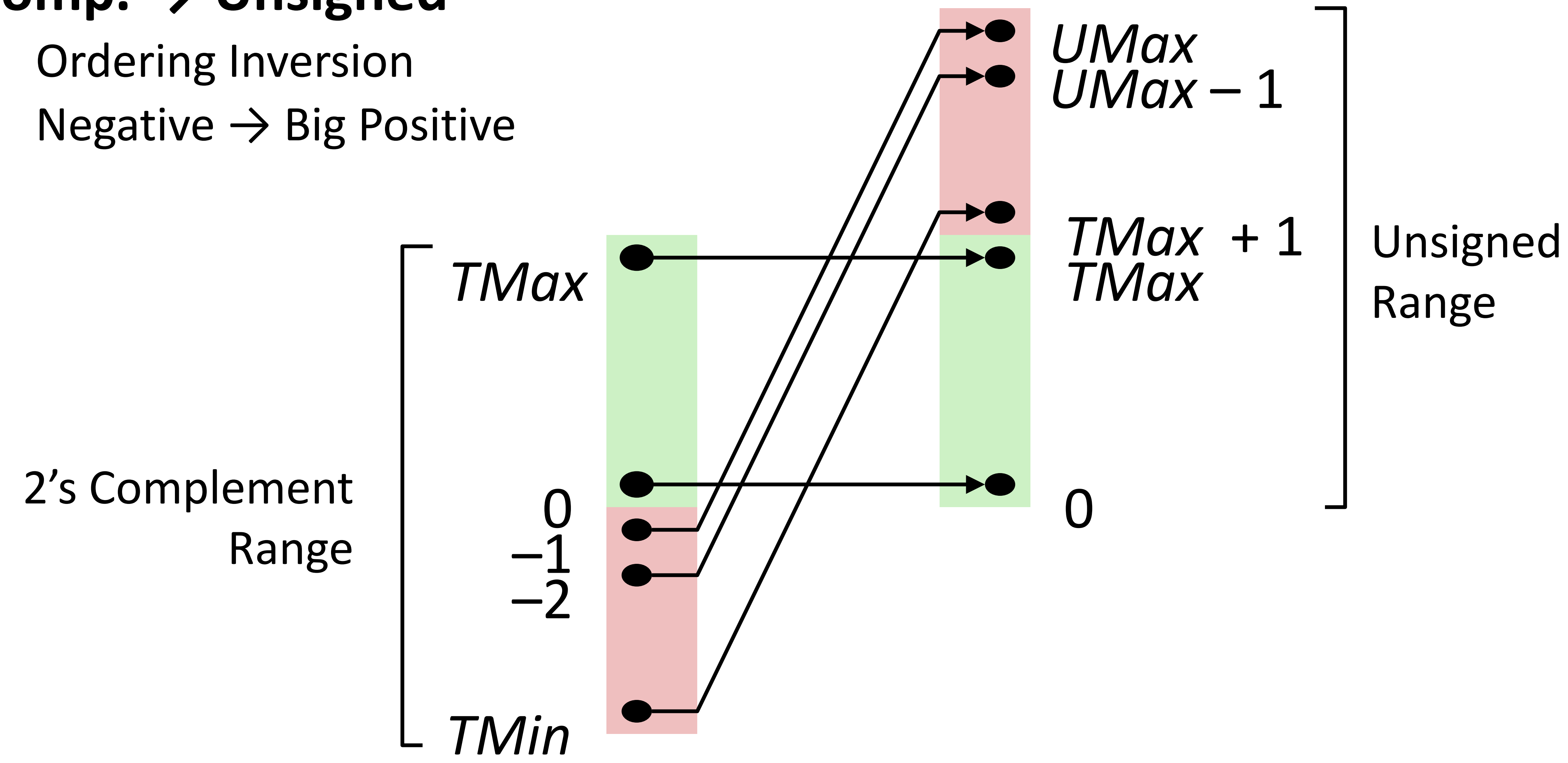


Large negative weight
becomes
Large positive weight

Conversion Visualized

2's Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive



Signed vs. Unsigned in C

Constants

- By default are considered to be signed integers
- Unsigned if have “U” as suffix

```
0U, 4294967259U
```

Casting

- Explicit casting between signed & unsigned same as U2T and T2U

```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

- Implicit casting also occurs via assignments and procedure calls

```
tx = ux;
uy = ty;
```

Casting Surprises

Expression Evaluation

- If there is a mix of unsigned and signed in single expression, ***signed values implicitly cast to unsigned***
- Including comparison operations **<, >, ==, <=, >=**
- Examples for W = 32: **TMIN = -2,147,483,648 ,** **TMAX = 2,147,483,647**

Casting Surprises

Constant$_1$	Constant$_2$	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	signed
2147483647	-2147483647-1	>	signed
2147483647U	2147483647U	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

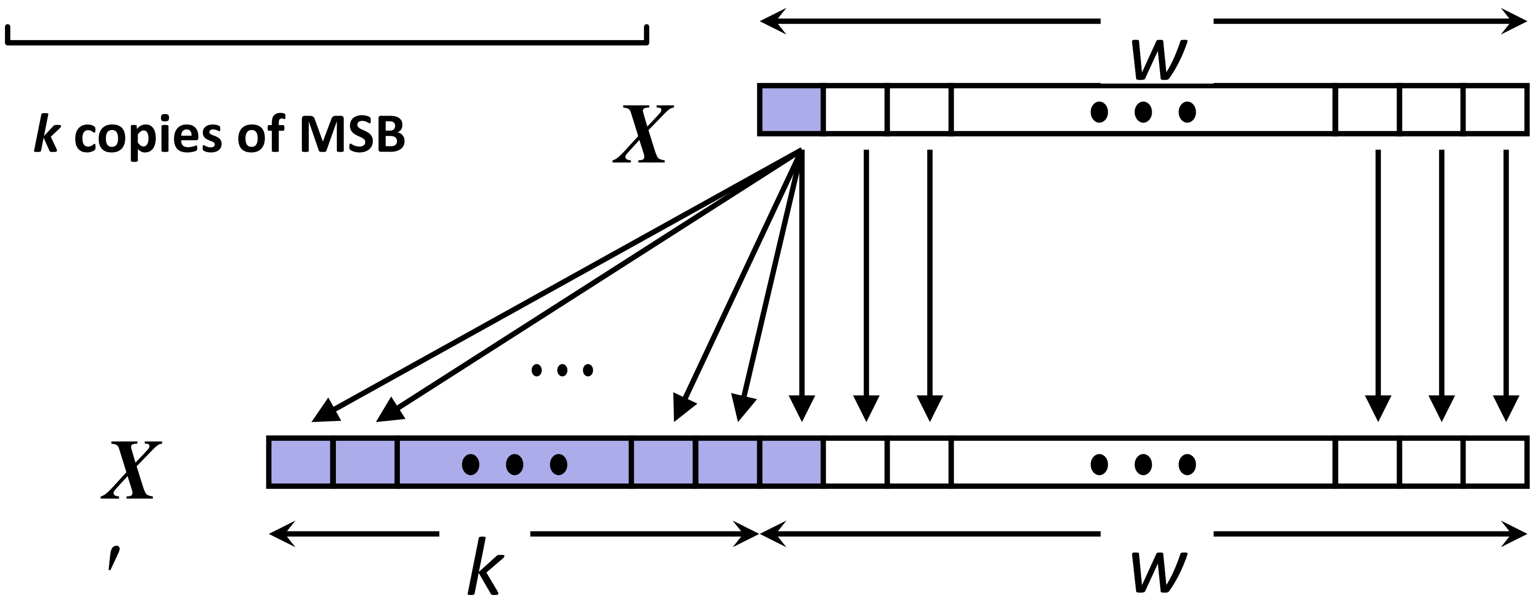
Sign Extension

Task:

- Given w-bit signed integer x
- Convert it to w+k-bit integer with same value

Rule:

- Make k copies of sign bit:
- $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$



Sign Extension Example

```
short int x =  15213;
int       ix = (int) x;
short int y = -15213;
int       iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

Converting from smaller to larger integer data type

C automatically performs sign extension

Summary: Expanding, Truncating: Basic Rules

Expanding (e.g., short int to int)

- Unsigned: zeros added
- Signed: sign extension
- Both yield expected result

Truncating (e.g., unsigned to unsigned short)

- Unsigned/signed: bits are truncated
- Result reinterpreted
- Unsigned: mod operation
- Signed: similar to mod
- For small numbers yields expected behavior

Signed Binary - Two's Complement

How do we represent negative numbers?

The two's complement of a number is technically it's value subtracted from 2^N.

In two's complement, most bits have the same contribution as in unsigned. The value of the i-th bit is 2^i (assuming i starts from 0).

However, the most significant bit of an N bit number has a value of -2^(N-1) instead of 2^(N-1)

Signed Binary: Two's Complement

- Assume we're dealing with four bit numbers.
- Consider the unsigned binary number 1010:
  - $1010 = 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 10$
- Now consider the signed binary number 1010:
  - $1010 = 1*(-(2^3)) + 0*2^2 + 1*2^1 + 0*2^0 = -6$

- Same sequence of bits, but depends on how we interpret!

How to convert between negative and positive

- The method: take the bitwise inverse and add one. Consider 0101 (5).
- 1. 0101
- 2. Bitwise inverse of 0101 = 1010.
- 3. 1010 + 0001 = 1011.
- 4. Confirm: 1011 = $(-(2^3)) + 2^1 + 2^0 = -5$

Signed Binary: Notes

- The value of a signed binary number depends on the number of bits there are.
  – Four bit signed: 1111 = -1
  – Five bit signed: 01111 = 15
- An N-bit signed binary number has 2^N possible values with a range of $[-2^{(N-1)}, 2^{(N-1)}-1]$
- REMEMBER THIS: The range of a two’s complement signed binary number is **not** symmetrical around 0.
- Henceforth all signed binary is two's complement unless otherwise specified.

Binary arithmetic

- What does it mean to add bits?
- The idea is the same as in decimal. Let's try with unsigned.

```
      0001            0001          0001          0011
+     0010          + 0001        + 0111        + 0111
      ----------      --------      ----------    --------
      0011            0010          1000          1010
```

Note that adding two or three 1 bits will produce a carry bit that must be added to the next bit over.

Binary arithmetic

● How about subtraction? You can generalize the decimal method for binary, but...
● The simplest way to do X – Y is to do X + (-Y).
● Take 0110 (6) – 0010 (2)
● This becomes 0110 (6) + 1110 (-2)

```
        0110
+       1110
        ----------
        0100
```

Unsigned overflow in C

● With signed arithmetic, we saw that a carry out bit was completely valid, but what about unsigned?
● Say we have 4-bit numbers and we add 6 + 12.
● 6 = 0110, 12 = 1100

```
     0110
+    1100
     --------
     10010 = 18
```

● ...but this requires 5 bits to represent. We only have 4.
● How does the wise and venerable C respond?

Unsigned overflow in C

- Just drop bits.
- If an unsigned operation of an n-bit number requires more than n bits, the resulting number will consist only of the n least significant bits.

– Ex. In the previous example:

– 0110 + 1100 = (1) 0010, the leading one is dropped and instead of the right answer of 18, you get the incredibly wrong answer of 2

- More formally, if you have n-bits, the computation of x + y is (x + y) % 2^n.

– Ex. (6 + 12) % 2^4 = 18 % 16 = 2

Unsigned overflow in C

- This is also true of unsigned multiplication overflow:
- If we have 4 bits, 6 * 12 = 72
- In binary, 72 is 1001000.
- Truncate bits beyond 4 and the result is 1000 = 8.
- 72 % 2^4 = 8
- Keep in mind, this is for unsigned numbers only.
- Let's not think about signed numbers for now.

Datatypes in C

- Each native datatype in C is expressed by a sequence of bits.
- Simple data types such as ints and shorts come in unsigned and signed variants where signed is the default (ie. int is actually a signed int)
- However, the number of bits used to express these numbers differs depending on whether the processor is 32 or 64-bit.
- ...but more on the processor definitions later.

Datatypes in C

- char/unsigned char : 8-bits
- short/unsigned short : 16-bits
- int/unsigned (int) : 32-bits

Here's where it gets weird.

- In 32-bit machines:

– long/unsigned long : 32-bits

– long long/unsigned long long (proof that a five year old named these) : 64-bits

- In 64-bit machines:

– long/unsigned long : 64-bits

– long long/unsigned long long (ugh) : 64-bits

Boolean Operators

- Boolean operators operate on a single bit.
- AND : &

– Result is 1 if both inputs are 1.

- OR : |

– Result is 1 if either of the inputs are 1.

- XOR : ^

– Result is 1 if one input is 1 and the other is 0

- NOT : ~

– Result is 1 if the input is 0.

Bitwise Operators

- Bitwise operators perform repeated boolean operations on each bit of a number or pair of numbers.
- Bitwise invert (not the same as logical invert or '!')

– ~(1011) = 0100

- Bitwise AND/OR (not the same as logical AND/OR or &&/||)

– 1010 & 1100 = 1000

– 1010 | 1100 = 1110

- Bitwise XOR

– 1010 ^ 1100 = 0110

Bitwise Operators

● Left shift/right shift (arithmetic vs logical)
● Left shift
– 0111 << 1 = 1110
● Right shift
– 1011 >> 1 = 0101 (logical)
– 1011 >> 1 = 1101 (arithmetic)
● Why two different right shifts?

Logical Operators

- Where bitwise operators operate on each individual bit of a number, logical operators operate on the number as a whole
  - II, &&, !
- To invert a bit sequence x, you would use ~x.

What happens if you use the logical invert '!'?

- !(1010) = 0
- !(0111) = 0
- !(0) = 1

Logical Operators

- What happens when you use logical operators on numbers?

  1011 && 1100?
- Non-zero numbers are interpreted as 1 and 0 is interpreted as 0.

  1011 && 1100 <=> 1

  1011 && 0 <=> 0

  1011 || 0 <=> 1

Useful Tips

x is a bit vector

```
if(x == 0)
	return 0;
else
	return 1;
```

OR

```
return !!x;
```

a, b, and c are bits

```
if(a)
	return b;
else
	return c;
```

OR

```
return (a & b) | (~a & c)
```

De-Morgan’s Law

```
a & b = ~(~a | ~b)
a | b = ~(~a & ~b)
```

Multiplication by Shifting

- Consider the 4-bit unsigned number 0110.

– 0110 = 2^3 * 0 + 2^2 * 1 + 2^1 * 1 + 2^0 * 0 = 2^2 + 2^1 = 6

- 0110 << 1 = 1100

– 1100 = 2^3 * 1 + 2^2 * 1 + 2^1 * 0 + 2^0 * 0 = 2^3 + 2^2

– 2^3 + 2^2 = 2*(2^2 + 2^1) = 12

- x << n = x * 2^n

Multiplication by Shifting

How can we think of multiplying two arbitrary (ie non-powers of two) numbers in binary?

0110 * 1011 (= 6 * 11 = 66)

= 0110 * (1000 + 0010 + 0001)

= 0110 * 1000 + 0110 * 0010 + 0110 * 0001

= 0110 << 3 + 0110 << 1 + 0110

= 0110000 + 01100 + 0110

= 1000010

Division by shifting

By the same logic, this ought to work for division right?

Consider 4-bit unsigned:

– 1100 = 12

– 1100 >> 1 = 0110 = 6

Consider 4-bit signed:

– 1100 = -4

– 1100 >> 1 = 0110 = 6 (??)

Division by shifting

- Previously, we tried logical right shifting (shift in zeros, but that didn't seem to pan out). This is where arithmetic right shifting steps in.
- Consider 4-bit signed:
  – 1100 = -4
  – 1100 >> 1 = 1110 = -2
- Logical shifting maintains correct values for unsigned operations while arithmetic shifting maintains correct values for signed operations.

Division by shifting

- Consider the 4-bit signed number 1101.
  – $1101 = -2^3 * 1 + 2^2 * 1 + 2^1 * 0 + 2^0 * 1 = -(2^3) + 2^2 + 2^0 = -3$
- 1101 >> 1 = 1110
  – $1110 = -2^3 * 1 + 2^2 * 1 + 2^1 * 1 + 2^0 * 0 = -2$
- -3 /(integer) 2 = -1, not -2
- How do you resolve this?

Access specific bits

Say you have the binary value 1010 and you only want to consider bits 1 and 2, that is, you want to transform 1010 into 0010.

```
      1010
&     0110
   ------------
      0010
```

References

- Slides modified from DJ Kim, UT Wang and Shikhar Malhotra
- http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/schedule.html

Thank You