

C++ Bitwise Operators And Debugging

ENSC 254 Tutorial

Rashid Zamanshoar Heris

School of Engineering Science
Simon Fraser University
Burnaby, BC, Canada

(Slides adapted from William Xue)

May 15, 2026

A Quick Preamble

- Engineering is **hard**. Coding is **hard**.
- There is **a lot** to learn, and it takes **time and effort** to get it right.
- Here are some tips you've heard before
 - Ask for help! Use your lab times!
 - Even during code demo weeks, TAs are available for help
- Don't get discouraged, don't give up :)
- There is always more to learn :)

A note on Course Policy

- For your lab assignments, we'll have “code demos” - please take these seriously, and be able to answer questions about how your code works confidently and quickly.
 - If you forgot how you implemented it, study it before the interview.
 - This will can simulate a real interview, where you will have to confidently explain your code to an interviewer

Lab Submission Format “Demo”

- Please pay attention to your lab submission format; otherwise, you may lose easy marks.
- Assignment Submission:

Your lab assignments will be submitted electronically through Canvas. You will need to submit **a single YOUR_SFU_COMPUTING_ID.zip file**. This format makes it easy for TA to do auto-grading. Failure to comply with this format will result in a **ZERO** point for the assignment.

To zip your files in Linux:

1. Go to your lab1 folder
2. `cd part1; make clean` //make sure you clean your files in part1 sub-folder
3. `cd ../part2; make clean` //make sure you clean your files in part2 sub-folder
4. `cd ../part3; make clean` //make sure you clean your files in part3 sub-folder
5. `cd ../../` //go two levels up
6. `mv lab1 YOUR_SFU_COMPUTING_ID` //rename your lab1 folder using your SFU Computing ID
7. `zip -r YOUR_SFU_COMPUTING_ID.zip YOUR_SFU_COMPUTING_ID` //zip all your lab1 files into a single .zip

What's in this tutorial?

PART 1: How to debug programs

- Review of Basic Data types in C
- Binary
- What is a program?
- GDB (to help find logic errors)

What's in this tutorial?

PART 2:

- Structs and Unions
- Pointers
- Bitwise Operators
 - `& | << >> ^`
- Lab 1 Tutorial
 - <https://nandgame.com/>

Part 1.1 - GDB

Example program: Sum up to n

- How do we do it?
- Suppose we didn't know the mathematical formula, $n*(n+1)/2$

```
unsigned long sum_first_n(unsigned long input)
{
    unsigned long sum = 0;
    for (unsigned int i = input; i >= 0; --i)
    {
        sum += i;
    }
    return sum;
}
```

Let's try our sum function!
(demo)

- That was just one small example.
- Real-world code can be much more complicated, and much bigger.
 - Small projects contains thousands of lines
 - Big projects can be millions of lines
- Debugging with print statements works sometimes... but other times, we need more power.
- Need a way to watch the CPU go through, line by line, as it executes each instruction in sequence.
 - Debuggers!!

- Need a way to watch the CPU go through, line by line, as it executes each instruction in sequence.
 - Debuggers!!
- The debugger we're examining today is called "GDB".
 - Widely available on Linux platforms
 - Stands for the Gnu DeBugger
- Many IDEs come with built-in debuggers, as well
 - You may already be familiar with their debuggers.

- How do we install it?
 - Linux:
 - <https://installati.one/ubuntu/20.04/gdb/>
 - GDB is pre-installed on our lab computers.
 - Windows:
 - Install WSL (Windows Subsystem for Linux), which will include it
 - Install MinGW (Minimalistic GNU for Windows), which will also include it
 - Mac:
 - On MacOS need to use lldb instead of gdb
 - LLDB stands for low level debugger, it has similar functionalities to GDB

Note - if you have gcc and g++ installed, you might already have gdb.
Just try using it if you're not sure.

- How do we use it?
- Instead of running your program normally, feed your program as an input to gdb.
- In order for this to work, you need to modify your compilation slightly.
- Compile with `-g -O0` compiler flags
 - `-g`: insert extra information for a debugger to use
 - `-O0`: Optimization level 0 (i.e. do not optimize)
 - `g++ -g -O0 main.cpp -o main`
- `gdb ./main`

(Note that setting `-O0` is not necessary for GDB, can also set `-O2` for example)

Let's see GDB for ourselves!
(demo)

GDB Commands (cheatsheet)

- break ; run ; next ; step ; continue
 - To control the flow of the program
- list ; frame
 - To see where you are
- layout next ; tui disable
 - To show and disable a UI
- print ; display ; watch
 - To show values
- info breakpoints ; info watchpoints
 - To list some metadata
- backtrace ; quit
 - To show the call stack; to exit GDB

run

Starts the program.

break

break main (Pause at main)

break sum_first_n (Pause at function)

break 7 (Pause at line number)

- When the program reaches that point:
 - execution pauses
 - you can inspect variables
 - you can step through code

next

Executes the next line. (Does NOT enter functions)

step

Executes the next line. (enters functions)

continue

Continue normal execution until:

- next breakpoint
- crash
- program end

GDB Commands (cheatsheet)

- `break ; run ; next ; step ; continue`
 - To control the flow of the program
- `list ; frame`
 - To see where you are
- `layout next ; tui disable`
 - To show and disable a UI
- `print ; display ; watch`
 - To show values
- `info breakpoints ; info watchpoints`
 - To list some metadata
- `backtrace ; quit`
 - To show the call stack; to exit GDB

list

Shows source code around current location.

frame

Shows current stack frame.

Basically:

- where are we?
- which function are we inside?

layout next

Turns on text UI mode. Shows:

- source code
- Terminal
- execution position

tui disable

- Exit UI mode

print

Print variable values.

display

Automatically print a variable every step.

watch

Pause when a variable changes.

GDB Commands (cheatsheet)

- `break ; run ; next ; step ; continue`
 - To control the flow of the program
- `list ; frame`
 - To see where you are
- `layout next ; tui disable`
 - To show and disable a UI
- `print ; display ; watch`
 - To show values
- `info breakpoints ; info watchpoints`
 - To list some metadata
- `backtrace ; quit`
 - To show the call stack; to exit GDB

`info breakpoints`

Shows all breakpoints.

`info watchpoints`

Shows active watchpoints.

`backtrace`

Shows function call stack.

`quit`

Exit GDB.

Most Important Commands to Memorize

Command	Purpose
break	Set breakpoint
run	Start program
next	Next line
step	Enter function
print	Show variable
continue	Continue execution
backtrace	Show call stack
quit	Exit

Debugging Tips

- The easiest way to fix bugs is to be disciplined when you code – know what you're writing before you write it.
 - Test early, test often.
- Use the tools at your disposal.
 - Understand your tool suite (GDB, Valgrind, GTest, IDE features).
 - Please take some time to learn version control tools, such as git.
 - Ask the TAs in lab times if you want help with this!

Part 2.1 - Structs and Unions

C/C++ More Data Types

- Structs
 - In C, they let you “combine data together”, like classes but without function members
 - In C++, they’re basically the same as classes since you can add function members
- Unions
 - Let you save memory, by allowing the same section of memory be represented as different data types
 - They look a lot like structs, but they are NOT the same!

C/C++ Struct

C/C++ Struct

```
/* access rtype with: rtype.(opcode|rd|funct3|rs1|rs2|funct7) */  
struct {  
    unsigned int opcode;  
    unsigned int rd;  
    unsigned int funct3;  
    unsigned int rs1;  
    unsigned int rs2;  
    unsigned int funct7;  
} rtype;
```

- This is an r-type instruction struct found in your Lab 1 Code
- The struct contains member variables which can be accessed with the dot operator

C/C++ Struct with Bit Fields

```
/* access rtype with: rtype.(opcode|rd|funct3|rs1|rs2|funct7) */  
struct {  
    unsigned int opcode : 7;  
    unsigned int rd : 5;  
    unsigned int funct3 : 3;  
    unsigned int rs1 : 5;  
    unsigned int rs2 : 5;  
    unsigned int funct7 : 7;  
} rtype;
```

- The struct above contains a bit-field declaration
- The bit-field tells the compiler the required bit-width of the corresponding member variable

Memory Layout of Structs

```
struct Example
{
    int a;
    int b;
};
```

- Struct members are stored separately in memory.
- Memory:
 - [a][b]
- Size:
 - 8 bytes

C/C++ A Union Of Structs

```
typedef union {
    struct {
        unsigned int opcode : 7;
        unsigned int rd : 5;
        unsigned int funct3 : 3;
        unsigned int rs1 : 5;
        unsigned int rs2 : 5;
        unsigned int funct7 : 7;
    } rtype;
    struct {
        unsigned int opcode : 7;
        unsigned int rd : 5;
        unsigned int funct3 : 3;
        unsigned int rs1 : 5;
        unsigned int imm : 12;
    } itype;
} Instruction;
```

- Shown on the left is a union of two structs: a rtype struct and a itype struct
- The Instruction datatype can represent either a rtype or a itype instruction
- The instruction only requires 32 bits of memory (4 bytes)
- Example:
 - Access itype imm field:
 - instruction.itype.imm
 - Access rtype funct7:
 - instruction.rtype.funct7

Memory Layout of Union

```
union Example
{
    int a;
    int b;
};
```

- Only largest member matters.
- Memory:
 - [a / b]
- Size:
 - 4 bytes

C/C++ More Data Types

- Structs vs Unions:
- Structs keep all data elements at the same time, in contiguous memory
- Unions keeps enough memory for one data type in the union at a time
- For example, in a union of structs, accessing the data as one data type allows the programmer to access bits specific to each instruction

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type

C/C++ A Union Of Structs Question

```
typedef struct {  
    struct {  
        unsigned int opcode : 7;  
        unsigned int rd : 5;  
        unsigned int funct3 : 3;  
        unsigned int rs1 : 5;  
        unsigned int rs2 : 5;  
        unsigned int funct7 : 7;  
    } rtype;  
    struct {  
        unsigned int opcode : 7;  
        unsigned int rd : 5;  
        unsigned int funct3 : 3;  
        unsigned int rs1 : 5;  
        unsigned int imm : 12;  
    } itype;  
} Instruction;
```

- The rtype and itype both require 32 bits of memory
- How much memory does a struct of structs require (left)?
- Correct: **64 Bits!**

C/C++ A Union Of Structs Question

```
typedef union {  
    struct {  
        unsigned int opcode : 7;  
        unsigned int rd : 5;  
        unsigned int funct3 : 3;  
        unsigned int rs1 : 5;  
        unsigned int rs2 : 5;  
        unsigned int funct7 : 7;  
    } rtype;  
    struct {  
        unsigned int opcode : 7;  
        unsigned int rd : 5;  
        unsigned int funct3 : 3;  
        unsigned int rs1 : 5;  
        unsigned int imm : 12;  
    } itype;  
} Instruction;
```

- The rtype and itype both require 32 bits of memory
- How much memory does a union of structs require (left)?
- Correct: **32 Bits!**

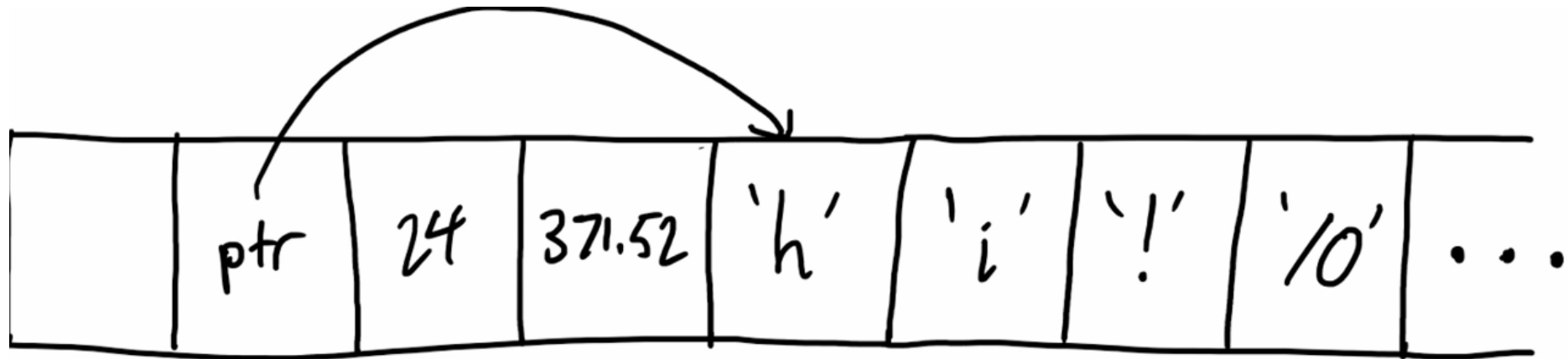
Part 2.1 - Struct Code Examples

Part 2.2 - Pointers

Pointers

```
int val;  
int *my_ptr = &val;
```

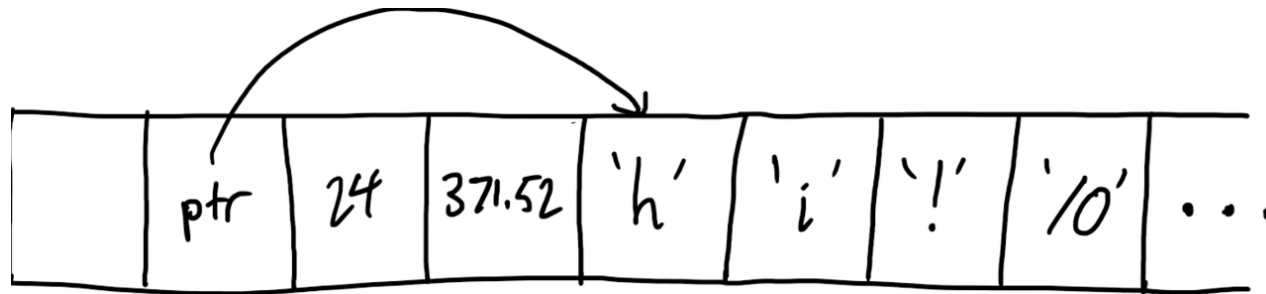
- In this simple example, my_ptr stores the memory address of val.
- “Normal” variables store data.
- Pointers store memory addresses.
- A programmer can view memory as an enormous array.
 - Pointers “point” to entries in this enormous array.



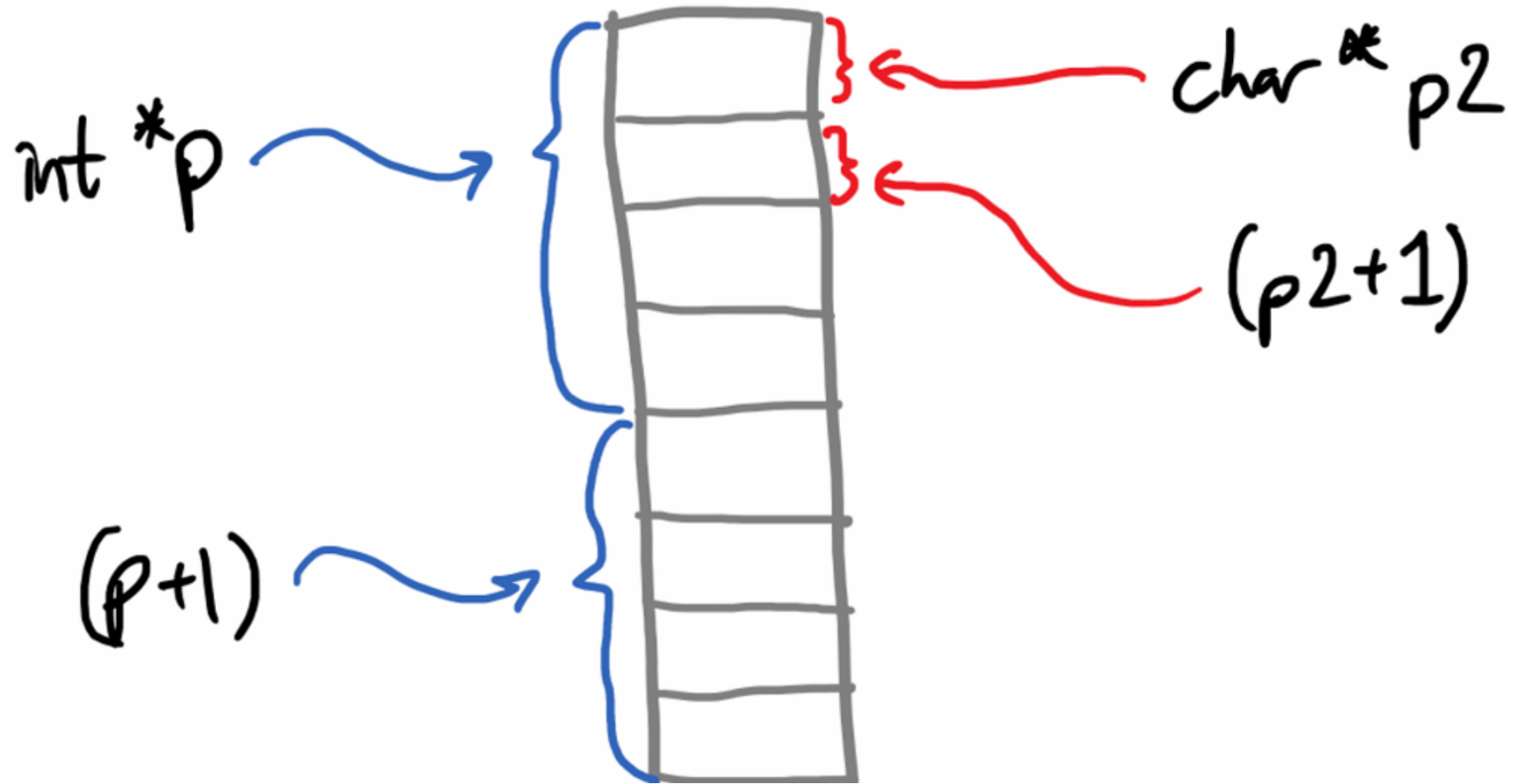
Pointer Arithmetic

```
char my_str[5] = "hi!";  
char *my_ptr = my_str;
```

- Question: what do you think (my_ptr+1) should do?
- In C, it means “give me the starting address of the next value, for whatever type my_ptr points to”



Pointer Arithmetic



Memory Visualization

```
int x = 10;
```

- Maybe memory looks like:

Address	Value
1000	10

```
int *ptr = &x;
```

- Maybe memory looks like:

Address	Value
2000	1000

- ptr stores memory address 1000
- which is where x lives

x	&x	ptr	&ptr	*ptr
10	1000	1000	2000	10

Why pointers are powerful

Pointers allow:

- direct memory access
- dynamic memory allocation
- arrays
- linked lists
- trees
- operating systems
- hardware programming

Without pointers:

- C/C++ would lose much of their power

Pointers to Structs

- You can use the arrow operator “->” to refer to members of a struct pointer.

```
struct_product_t products[5];  
struct_product_t* prod_p = products;
```

`(prod_p+i)->product_id`  `(* (prod_p+i)).product_id`

```
for (int i = 0; i < 5; ++i){  
    (prod_p+i)->product_id = i;  
    (prod_p+i)->dollars = 50*i;  
    (prod_p+i)->cents = 99;  
}
```



```
for (int i = 0; i < 5; ++i) {  
    (* (prod_p+i)).product_id = i;  
    (* (prod_p+i)).dollars = 50*i;  
    (* (prod_p+i)).cents = 99;  
}
```

Part 2.2 - Pointer Code Examples

Let's take a break?

Part 2.3 - Bitwise Operations

C/C++ Basic Data Types

- Native data types:
 - Char
 - Int
 - Float
 - Double
 - And more...
- All represented in binary
- There are also unsigned and signed versions of each of these
 - Except float

Signed and Unsigned Data

- Unsigned data types are represented in “normal” binary:
- The n'th bit represents 2^n
- 4-bit Example:
 - $1011 = 8 + 0 + 2 + 1 = 11$
 - $0101 = 0 + 4 + 0 + 1 = 5$

$$\overline{2^3} \quad \overline{2^2} \quad \overline{2^1} \quad \overline{2^0}$$

Signed and Unsigned Data

- Signed data types are represented using two's complement
 - Just like normal binary, but the highest bit represents -2^n .
 - Only works for a predetermined size.
- 4-bit example:
 - $1011 = -8 + 0 + 2 + 1 = -5$
 - $0101 = 0 + 4 + 0 + 1 = 5$

$$\overline{-2^3} \quad \overline{2^2} \quad \overline{2^1} \quad \overline{2^0}$$

Negating in two's complement

- Input: a two's complement number, n .
- Output: a two's complement number, $-n$.
- Algorithm:
 - Flip all bits (1 to 0, 0 to 1)
 - Add one

$$\overline{-2^3} \quad \overline{2^2} \quad \overline{2^1} \quad \overline{2^0}$$

Negating in two's complement

- Example: 0011 (0011=3)
 - 0011 = 3
 - Step 1: 0011 \rightarrow 1100
 - Step 2: 1100 \rightarrow 1101 = $-8 + 4 + 1 = -3$
- Example: 1101
 - 1101 = -3
 - Step 1: 1101 \rightarrow 0010
 - Step 2: 0010 \rightarrow 0011 = 3

Negating in two's complement


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

1	1	1	1	-1
1	1	1	0	-2
1	1	0	1	-3
1	1	0	0	-4
1	0	1	1	-5
1	0	1	0	-6
1	0	0	1	-7
1	0	0	0	-8

Negating in two's complement

- Can anyone find the issue?
 - There's one value that can't be negated properly.

0	0	0	0	0	1	1	1	1	-1
0	0	0	1	1	1	1	0	0	-2
0	0	1	0	1	1	0	1	0	-3
0	0	1	1	1	1	0	0	0	-4
0	1	0	0	1	0	1	1	0	-5
0	1	0	1	1	0	1	0	0	-6
0	1	1	0	1	1	0	0	1	-7
0	1	1	1	1	1	0	0	0	-8



Bitwise and Logical Operations

- Remember the basic binary operators, from ENSC 252?
 - NOT, AND, OR, XOR

A	NOT
0	1
1	0

A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1

A	B	OR
0	0	0
0	1	1
1	0	1
1	1	1

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

Bitwise and Logical Operations

- They have corresponding **logical** operators in C, except XOR.
 - `!`, `&&`, `||`
- They also have corresponding **bitwise** operators in C
 - NOT = `~`
 - AND = `&`
 - OR = `|`
 - XOR = `^`

Logical Operations

- Logical operations work by treating the arguments as a single logical value.
- **Nonzero** = logical 1
- **Zero** = logical 0
- **11111111** is treated same as
00000100 is treated the same as
10010001
- **00000000** cannot be expressed in any other ways

Bitwise Operations

- Bitwise operations work **bit-by-bit** on the arguments.

$$\begin{array}{l} \sim 1011 \\ = 0100 \\ \text{NOT} \end{array}$$

$$\begin{array}{l} 1011 \\ \& 0110 \\ = 0010 \\ \text{AND} \end{array}$$

$$\begin{array}{l} 1011 \\ | 0110 \\ = 1111 \\ \text{OR} \end{array}$$

$$\begin{array}{l} 1011 \\ \wedge 0110 \\ = 1101 \\ \text{XOR} \end{array}$$

Subtraction

- Suppose you're only allowed the following operators:
 - \sim | $\&$ | $+$
- How do we express $A - B$?
 - $-B = \sim B + 1$;
 - $A - B = A + (\sim B + 1)$;

Bit Masking

- Suppose you're looking at some number, but all you care about is a specific part of it.
 - E.g. **phone numbers: 604-123-4567**. Sometimes all you want is the area code, sometimes all you want is the “rest”.
 - E.g. **IPv4 Addresses: 192.168.0.43**. Typically, the first $\frac{3}{4}$ is the “network part”, and the last $\frac{1}{4}$ is the “host part”.
- Manipulating a **subset** of the bits is called “bit-masking”

Bit Masking

- Key idea: view the “A” argument as something we control.
- AND can be viewed as: $A \text{ ? } B : 0$
- OR can be viewed as: $A \text{ ? } 1 : B$
- XOR can be viewed as: $A \text{ ? } \sim B : B$

A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1

A	B	OR
0	0	0
0	1	1
1	0	1
1	1	1

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

Bit Masking

- Extracting only a subset of bits:
 - $0011\mathbf{10100}111 \& 0000\mathbf{1111}0000 = 0000\mathbf{1010}0000$
 - $0011\mathbf{10100}111 | 1111\mathbf{0000}1111 = 1111\mathbf{1010}1111$
- Setting only a subset of bits:
 - $0011\mathbf{10100}111 \& 1111\mathbf{0000}1111 = 0011\mathbf{0000}0111$
 - $0011\mathbf{10100}111 | 0000\mathbf{1111}0000 = 0011\mathbf{1111}0111$
- Toggling only a subset of bits:
 - $0011\mathbf{10100}111 \wedge 0000\mathbf{1111}0000 = 0011\mathbf{0101}0111$

Bitwise Operations

- There are also bit-shifting operations
 - **Arithmetic** right-shift = \gg
 - Left-shift = \ll
- $a \gg 4$
 - shift the bits in a right by 4, **with sign-extension**
- $b \ll 7$ means
 - shift the bits in b left by 7, **zero-padding the right side.**

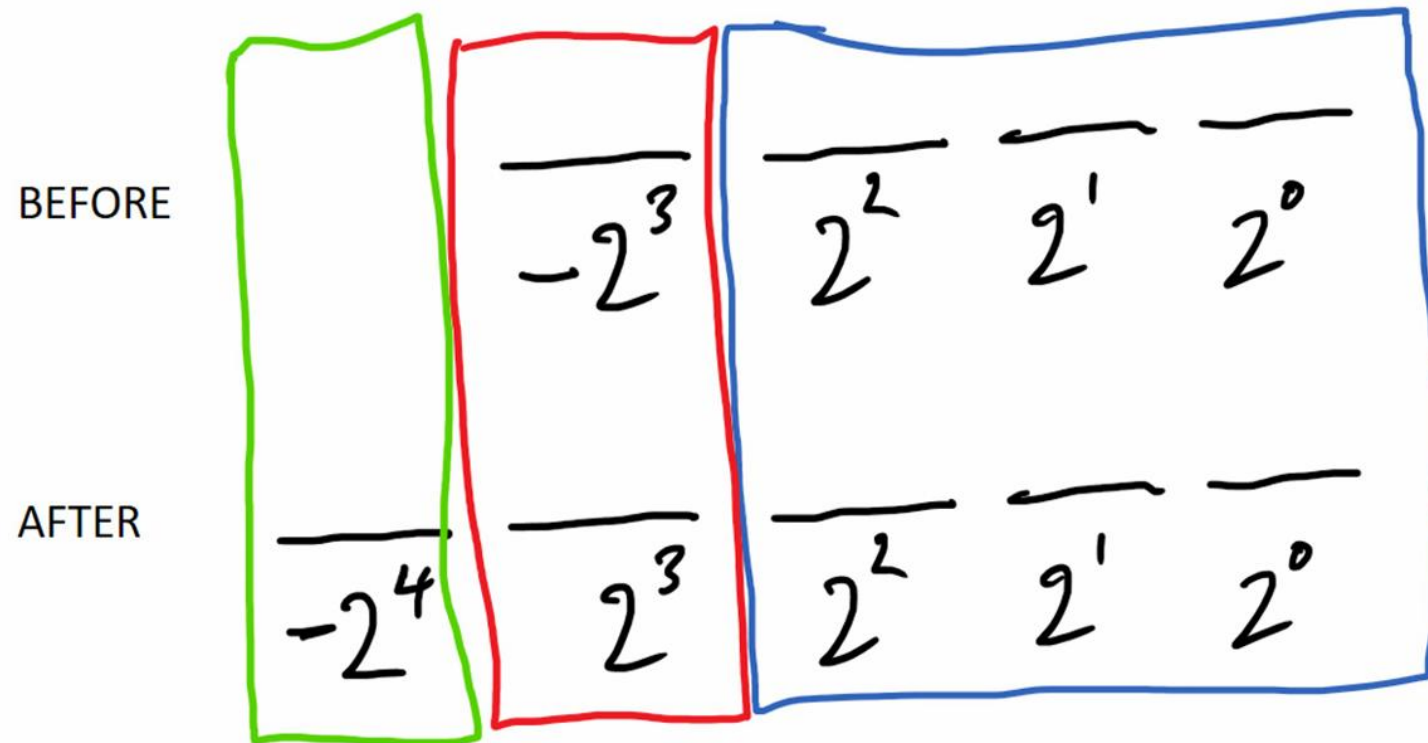
Sign Extension

- This is required when increasing bitwidth, or arithmetic-right shifting numbers in two's complement.
- Simply take the original value's most-significant bit (MSB), and copy it to the left.
- Examples:
 - $0110 = 4 + 2 = 6$
 - $00110 = 4 + 2 = 6$
 - $1001 = -8 + 1 = -7$
 - $11001 = -16 + 8 + 1 = -7$

$$\begin{array}{cccc} \overline{-2^3} & \overline{2^2} & \overline{2^1} & \overline{2^0} \\ \overline{-2^4} & \overline{2^3} & \overline{2^2} & \overline{2^1} & \overline{2^0} \end{array}$$

Sign Extension

- Proof: consider what this algorithm does to BEFORE and AFTER.
 - The blue part is unchanged.
 - Case analysis on red and green: if 1, what happens? if 0, what happens?



Part 2.3 - Code Examples And Lab 1 Tutorial