CS 33:
Introduction to Computer Organization

Week 1

Agenda

- Class/Homework Questions
- Lab 1: Datalab
- Boolean operations
- Boolean/Bitwise techniques
- Type casting

Lab 1: datalab

- Setting up the environment:
  - Again, highly recommended (required) that you use the SEASnet server: lnxsrv09.seas.ucla.edu
  - The version of gcc you need to use is 5.2.0
  - Check by using “gcc --version”
- This version of gcc is located in: /usr/local/cs/bin of lnxsrv09.seas.ucla.edu.
- If 5.2.0 is not the version that you see (and it probably won't be), you will have to add /usr/local/cs/bin to the path environment variable.

Lab 1: datalab

- Environment variables are variables that are known by the operating system or by an instance of a terminal.
- The path environment variable defines the directories in which to look for programs.
  - If you type “gcc”, it finds the installed version of gcc by looking through the directories specified by the path.

Lab 1: datalab

- First, you will have to determine what shell (type of command line interface) you're using.
- Try the commands:
  - "echo $0”
  - "ps -p $$"
- If echo $0 prints out “-bash”, you're using bash. If it prints out “-tcsh” and “-csh”, you're using tcsh/csh.
- “ps -p $$” should print out a table with a single row whose “CMD” field should match the output of echo $0.

Lab 1: datalab

- Side note:
- “echo $0” - Will tell you your current shell.
  - ex. -tcsh
- “echo $SHELL” - Will tell you your default shell.
  - ex. /usr/local/bin/tcsh
- If the results of these commands differ, you may have some script that is executing another shell upon logging in.
- If so, use the shell version that you see from the command “echo $SHELL”.

Lab 1: datalab

- If you're using csh or tcsh, in your home directory, do the following:
  - Open the file “.login” from a text editor.
  - Add the line: “set path=(/usr/local/cs/bin $path)”
  - Restart
- If you're using bash, in your home directory, do the following:
  - Open the file “.bash_profile” from a text editor.
  - Add the line: “export PATH=/usr/local/cs/bin:$PATH”
  - Restart

Lab 1: datalab

- Setting up the datalab:
  - Download datalab.tgz from CCLE.
  - Copy it to a directory of your choosing.
  - .tgz is a compressed file type.
  - Uncompress it with the following:
  - “tar -zxvf datalab.tgz”

Lab 1: datalab

- Running the datalab:
  - Compiling: “make”
  - Testing your code (correctness): “./btest”
  - Testing your code (follows rules): “./dlc -e bits.c”
- See INSTRUCTIONS.txt and README for more.

Lab 1 Errata

- “rotateRight(int x, int n) - Rotate x to the right by n”
- “Examples: rotateRight(0x87654321,4) = 0x76543218”
- The example demonstrates left rotation.
- Implement right rotation and ignore the example.

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

- Consider 4-bit binary numbers
- 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

- Whereas bitwise operators operate on each individual bit of a number, logical operators operate on the number as a whole
  - ||, &&, !
- To invert a bit sequence x, you would use ~x. What happens if you use the logical invert '!'?
- !1010 = 0
- !0111 = 0
- !0000 = 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 Tricks: Conditioning without "if"

- a, b, and c are bits
- if(a)

  return b;
- else

  return c;

Useful Tricks: Conditioning without "if"

- We know that we want to return b only if a == 1.
  - This can be represented as a & b.
- We want to return c only if a == 0.
  - This can be represented as ~a & c
- return (a & b) | (~a & c)

Useful Tricks: Representing AND expressions with OR and vice versa

- How can we represent c = a & b without using AND?
- DeMorgan's Law:
  - a & b = ~(~a | ~b)
  - a | b = ~(~a & ~b)

Useful Tricks: Multiplication via 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
  - $1100 = 2^3 * 1 + 2^2 * 1 + 2^1 * 0 + 2^0 * 0 = 2^3 + 2^2$
  - $2^3 + 2^2 = 2*(2^2 + 2^1) = 12$
- $x << n = x * 2^n$

Useful Tricks: Multiplication via 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 (correct!)

Useful Tricks: Division via shifting

- By the same logic, this ought to work for division right?
- Consider 4-bit unsigned:
  - 1100 = 12
  - 1100 >> 1 = 0110 = 6 (looks good...)
- Consider 4-bit signed:
  - 1100 = -4
  - 1100 >> 1 = 0110 = 6 (hrm?)

Useful Tricks: Division via 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.

Useful Tricks: Division via 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?

Useful Tricks: Extracting 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.

Useful Tricks: Extracting specific bits

- 1010
- & 0110
- ----------
- 0010
- Can be extended to hexadecimal. Recall that 0xF is 1111. If we have 0x33221100 and we want to extract bits 8-15:
- 0x33221100
- & 0x0000FF00
- ----------
- 0x00001100

Useful Tricks: Checking for Signed Overflow

- A signed operation between a and b has overflowed if a and b are the same sign and the result is of a different sign.
- Side note: Is this sufficient to account for all operation types?
- Consider 4-bit signed:
- 0100 (4) * 0100 (4)
- = 0100 << 2 = 0000.

Useful Tricks: Checking for Signed Overflow

- Consider T = A + B;
- Overflow if:
  - (A < 0) == (B < 0) && (T < 0) != (A < 0)
- Informally, this is two conditions:
  - If sign of A is the same as sign of B.
  - If sign of T is different from sign of A.
- Note: In order to check the sign of a number, we only need to consider the most significant bit.

Useful Tricks: Checking for Signed Overflow

- As a result, we only need an operation that does the following:
- “if the sign bit of A and B are the same, return 1 (could overflow). If they are different, return 0 (can't overflow)”

```
A B | ?
--------
0 0 | 1
0 1 | 0
1 0 | 0
1 1 | 1
```

Useful Tricks: Checking for Signed Overflow

- Recall that:
  - 0 XOR 0 = 0
  - 1 XOR 1 = 0
- When we xor the most significant bits of A and B together, if they are the same (both positive or both negative), we get 0.
- We can invert the result to get '1' if A and B were the same sign.
- Do we need to then extract the sign bits and do the comparison?

Useful Tricks: Checking for Signed Overflow

- We can simply do (~A^B). All of the other bits will be XOR'd, but it will be okay since we're only really looking at the sign bit.
- The same method works for checking if the sign of A is different from the sign of T.
- = (~A ^ B) & (A ^ T) < 0
- Note that the numerical value won't mean much, the but if the conditions are met, the sign bit will be 1 (which means the value of the entire expression will be “less than 0).
- How can we do “less than 0” without the '<' (a la the datalab)?

Useful Tricks: Checking for Signed Overflow

- = ((~A^B) & (A ^ T) >> 31) & 1;

Useful Tricks: Creating bit mask

- Generate given int lowbit and int highbit, generate a mask of 1's ranging from the bit range specified by lowbit to highbit.
- int bitMask(int highbit, int lowbit)
- ex. int bitMask(5, 3) = ..00111000
- Assume lowbit and highbit range from 0 to 31. If highbit < lowbit, return 0.
- Legal ops: ! ~ & ^ | + << >> (no subtraction)
- No integer constants greater than 0xFF
- Max ops: 16

Useful Tricks: Creating bit mask

- No problem right?
- We need: a bunch of zeros, (highbit – lowbit + 1) 1s, and then (lowbit) zeros.
- Using the mask creation method described in class, we can do:
- 1. (1 << (highbit – lowbit + 1)) – 1
  - ex. highbit = 5, lowbit = 3
  - (1 << (5 – 3 + 1)) – 1 = (1000) – 1 = 111
- 2. Then, shift the result by lowbit.
  - ex. 111 << 3 = 000...000111000

Useful Tricks: Creating bit mask

```
int bitMask(int highbit, int lowbit)
{
  int diff = highbit – lowbit;
  return ((1 << (diff + 1)) - 1) << lowbit;
}
```

- Hmm, not allowed to use '-' (also can't make an integer constant like 0xFFFFFFFF).

Useful Tricks: Creating bit mask

```
int bitMask(int highbit, int lowbit)
{
  int diff = highbit + ~lowbit + 1;
  return ((1 << (diff + 1)) + (~0)) << lowbit;
}
```

- 8 ops. Nice.
- Does this work in all cases?

Useful Tricks: Creating bit mask

- What if highbit < lowbit?
- Ex. highbit = 3, lowbit = 5.
- int diff = 3 – 5;
- (1 << -1) is undefined!
- We need to respond differently if highbit – lowbit is negative.
- How can we identify if highbit – lowbit is negative?

Useful Tricks: Creating bit mask

- int diff = highbit + ~lowbit + 1;
- int is_neg = diff >> 31;
- Now, is_neg is 111...111 if highbit – lowbit is negative and 000...000 otherwise.
- We need something like:

```
if(is_neg)
{
  //shift by 0
}
else
{
  //shift by diff + 1
}
```

Useful Tricks: Creating bit mask

- int diff = highbit + ~lowbit + 1;
- int is_neg = diff >> 31;
- int shamt = ((~is_neg) & (diff + 1)) | (is_neg & 0);
- return ((1 << shamt) + ~0) << lowbit;

Useful Tricks: Creating bit mask

- int diff = highbit + ~lowbit + 1;
- int is_neg = diff >> 31;
- int shamt = (~is_neg) & (diff + 1);
- return ((1 << shamt) + ~0) << lowbit;
- 11 ops... starting to sweat now...
- Does this work in all cases?

Useful Tricks: Creating bit mask

- What if highbit = 31 and lowbit = 0?
- (1 << (31 – 0 + 1)) = (1 << 32)
- Attempting to shift by a number that is greater than or equal to the data type size is undefined (likely a no-op).
- The result should be a mask of all 1s, but it's not. It's wrong.
- Just kill me now.

Useful Tricks: Creating bit mask

- The annoying thing is, this line:
  - return ((1 << shamt) + ~0) >> lowbit;
- ...would work C allowed for shifting amounts that are >= the data type size.
- Ex. if highbit = 31 and lowbit = 0, if shamt is 32.
  - ((1 << 32) + ~0) >> lowbit
  - (0 + ~0) >> lowbit
  - 0xFFFFFFFF >> lowbit = 0xFFFFFFFF
  - (lowbit = 0 in this case)
- Basically, the behavior is correct if we could shift by 32. How can we it to shift more than 31?

Useful Tricks: Creating bit mask

- int diff = highbit + ~lowbit + 1;
- int is_neg = diff >> 31;
- int shamt = (~is_neg) & (diff + 1);
- return ((1 << shamt) + ~0) << lowbit;
- We always want to shift it diff + 1 and diff + 1 is at most 32...

Useful Tricks: Creating bit mask

- int diff = highbit + ~lowbit + 1;
- int is_neg = diff >> 31;
- int shamt = (~is_neg) & (diff + 1);
- return ((1 << shamt) + ~0) << lowbit;
- We always want to shift it diff + 1 and diff + 1 is at most 32...
- Shift twice!

Useful Tricks: Creating bit mask

- int diff = highbit + ~lowbit + 1;
- int is_neg = diff >> 31;
- int shamt1 = (~is_neg) & diff;
- int shamt2 = (~is_neg) & 1;
- return (((1 << shamt1) << shamt2) + ~0) << lowbit;
- 13 ops.
- Can we make this better?

Useful Tricks: Creating bit mask

- int diff = highbit + ~lowbit + 1;
- int is_pos = ~(diff >> 31);
- int shamt1 = is_pos & diff;
- int shamt2 = is_pos & 1;
- return (((1 << shamt1) << shamt2) + ~0) << lowbit;
- 12 ops. Not bad.

Useful Tricks: Creating bit mask

- Could we do even better?
- Sometimes you may reach the limit of what one approach can accomplish.
- We had a pretty good solution up until we had to check for when highbit < lowbit.

Useful Tricks: Creating bit mask

- Another approach:
- If lowbit = 3 and highbit = 5, create two masks: a lower_mask and an upper_mask.
- lower_mask = 0...00111111 (based on highbit)
- upper_mask = 1...11111000 (based on lowbit)
- return lower_mask & upper_mask
- This resolves issue of the lowbit being greater than highbit.
  - Ex if lowbit is 5 and highbit is 3
  - lower_mask = 0...00000111
  - upper_mask = 1...11000000

Useful Tricks: Creating bit mask

- upper_mask:
- 1. 00...0000
- 2. ~(00...0000) = 11...1111
- 3. ~(00...0000) << lowbit = 11...1100..0
  - If lowbit = 3, then upper_mask = 11...111000
- int upper_mask = ~0 << lowbit;

Useful Tricks: Creating bit mask

- lower_mask:
- 1. 00...0000
- 2. ~(00...0000) = 11...1111
- 3. ~(00...0000) << (highbit + 1) = 11...1100..0
- 4. ~(~(00...0000) << (highbit + 1)) = 00..0011..1
  - If highbit = 5, then upper_mask = 111000000
- int lower_mask = ~(~0 << (highbit + 1));

Useful Tricks: Creating bit mask

- int upper_mask = ~0 << lowbit;
- int lower_mask = ~(~0 << (highbit + 1));
- return upper_mask & lower_mask;
- Does this work in all cases?

Useful Tricks: Creating bit mask

- What happens if highbit is 31?
- int lower_mask = ~(~0 << (highbit + 1));
- int lower_mask = ~(~0 << 32);
- Same issue as before, but we know how to deal with it now.

Useful Tricks: Creating bit mask

- int upper_mask = ~0 << lowbit;
- int lower_mask = ~((~0 << highbit) << 1);
- return upper_mask & lower_mask;
- 7 ops. Not bad at all.
- Wait...

Useful Tricks: Creating bit mask

- int neg_one = ~0;
- int upper_mask = neg_one << lowbit;
- int lower_mask = ~((neg_one << highbit) << 1);
- return upper_mask & lower_mask;
- 6 ops! That's more like it.

“Useful” Optimizations

- Say 'a' is a bit and we're not allowed to use '-'
- int count = ...;
- if(a)
- 	count = count + x;
- else
- 	count = count - x;

“Useful” Optimizations

- int a_mask = (a << 31) >> 31;
- int diff = (a_mask & x) | (~a_mask & (~x + 1))
- count = count + diff;
- 9 ops. Can we do better?

“Useful” Optimizations

- We either want to add x or we want to add ~x + 1. In other words, we want to add by x + 0 or ~x + 1.
- It might be better if we could do something like:
- count = count + x(OP1) + (OP2)

“Useful” Optimizations

- count = count + x(OP1) + (OP2)
- If a_mask is 111..111, we want x. If a_mask is 000...000, we want ~x.
- OP1?

“Useful” Optimizations

- count = count + x(OP1) + (OP2)
- If a_mask is 111..111, we want x. If a_mask is 000...000, we want ~x.
- count = count + x^(~a_mask) + …
- P xor 1 = ~P
- P xor 0 = P

“Useful” Optimizations

- count = count + x^(~a_mask) + (OP2)
- If a_mask is 111...111, we want to add by 0. If a_mask is 000...000, we want to add by 1.
- OP2?

“Useful” Optimizations

- count = count + x^(~a_mask) + (OP2)
- If a_mask is 111...111, we want to add by 0. If a_mask is 000...000, we want to add by 1.
- count = count + x^(~a_mask) + !a_mask.

“Useful” Optimizations

- int a_mask = (a << 31) >> 31;
- count = count + x^(~a_mask) + !(a_mask)
- We're down to 7 ops. Not bad.

Useful Tricks: Whatever this is

- From the desk of Bryant and O'Hallaron, CS:APP 3rd Edition, question 2.65:
- /*Return 1 when x contains an odd number of 1s; 0 otherwise. Assume w=32*/
- int odd_ones(unsigned x)
- You are prohibited from using:
  - if, while, division, multiplication, modulo, comparison operators, sorcery.
- You can use fewer than 12 arithmetic/bitwise/logical operators.

Useful Tricks: Whatever this is

- Where to even begin?
- The CS 31 way would be to go bit by bit.
- You could do this without a loop since you know the data type is 32-bits.
- However, you're definitely going to need more than 12 operations. In fact you're going to need 32+.
- The time complexity of this is O(n) where n is the number of bits.

Useful Tricks: Whatever this is

- We need an approach that is faster than order n. In order to do this, we will need an aggregating approach that will allow us to do at least some work in parallel.
- But first, we need to figure out how to keep track of if there are even or odd bits.
- We can do it by counting, sure, but is there an operator that will tell us if the inputs contain an odd number of bits?

Useful Tricks: Whatever this is

- Consider the truth table of xor:

```
 x y | x^y
-----------
 0 0 |  0
 0 1 |  1
 1 0 |  1
 1 1 |  0
```

- Another way of describing x^y is that the result is 1 if there is an odd number of 1's in the inputs.
- How suspiciously convenient.

Useful Tricks: Whatever this is

- Is this true for multiple xors?
  - ex. (w^x)^y, (w^x)^(y^z)?

“Useful” Tricks: Whatever this is

- Is this true for multiple xors?
  - ex. (w^x)^y, (w^x)^(y^z)?

```
 w  x  y  |  (w^x)^y
----------------------
 0  0  0  |    0
 0  0  1  |    1
 0  1  0  |    1
 0  1  1  |    0
 1  0  0  |    1
 1  0  1  |    0
 1  1  0  |    0
 1  1  1  |    1          Looks good!
```

“Useful” Tricks: Whatever this is

- It seems like we may be able to use this, but we'll need to xor every bit together.
- Not to mention, we need to do so non-linearly.
- Let's say we're trying this out on 8-bit unsigned:
  - 10110101
- Can we somehow xor multiple bits in one operation?

“Useful” Tricks: Whatever this is

- The ^ (xor) operation itself operates on an entire bit vector. We can simply use the xor itself to operate on the entire bits of the vector as follows:
- 10110101 → split into upper (1011) and lower(0101)
- upper ^ lower = (1011)^(0101) = 1110
- Now, we simply need to repeat the process with the remaining bits.

“Useful” Tricks: Whatever this is

- 1110 → upper (11) and lower (10)
- 11^10 = 01
- 01 → upper (0) and lower (1)
- 0 ^ 1 = 1 (original value 10110101 has an odd number of 1s)
- How would this look in C?

“Useful” Tricks: Whatever this is

```
int odd_ones(unsigned x)
{
  unsigned temp = x ^ (x >> 16);
  temp = temp ^ (temp >> 8);
  temp = temp ^ (temp >> 4);
  temp = temp ^ (temp >> 2);
  temp = temp ^ (temp >> 1);
  return temp&0x1;
}
```

- 11 ops!

“Useful” Tricks: Whatever this is

- But wait, when we do x ^ (x >> k), isn't this going to preserve too much number?
- 8-bit example:
  - 10110101 >> 4 = 00001101.
  - 10110101 ^ 00001011 = 1101 1110, not 0000 1110.

“Useful” Tricks: Whatever this is

- But wait, when we do x ^ (x >> k), isn't this going to preserve too much number?
- 8-bit example:
  - 10110101 >> 4 = 00001101.
  - 10110101 ^ 00001011 = 1101 1110, not 0000 1110.
- We only care about the lower bits. As the operations continue, eventually the bit that we care about is only the least significant bit.
- ...after which we AND the result with 1, ignoring the remaining bits.

Variable Placement in Memory

- Variables have memory addresses.
- Hence, pointers.
- char c = 0xBA;
- &c would be the address that c is stored in.
- Typically, memory is byte-addressable, which means that for each memory address, a single byte is stored there.
- Say &c = 0x100. If you looked into memory address 0x100, you'd find 0xBA.

Variable Placement in Memory

- int i = 0xFEEDBACC;
- Say &i is at address 0x100.
- Address 0x100 can only fit one byte. How can int i be stored at that address?
- In order to contain the four bytes of the int, we require four different addresses.
- Since &i begins at address 0x100, we will also need addresses 0x101, 0x102, and 0x103.
- But which byte belongs where?

Endianness

- Big Endian

Addr	0x100	0x101	0x102	0x103
Value	FE	ED	BA	CC

- Little Endian

Addr	0x100	0x101	0x102	0x103
Value	CC	BA	ED	FE

Variable Placement in Memory

- Say you have the following:
- int i = 0x11223344;
- int j = i & 0xFF;
- What is j if we're on a Little Endian machine?

Variable Placement in Memory

- Say you have the following:
- int i = 0x11223344;
- int j = i & 0xFF;
- What is j if we're on a Little Endian machine?
  - 0x44
- What is j if we're on a Big Endian machine?

Variable Placement in Memory

- Say you have the following:
- int i = 0x11223344;
- int j = i & 0xFF;
- What is j if we're on a Little Endian machine?
  - 0x44
- What is j if we're on a Big Endian machine?
  - Yup, this was a trick question. It's 0x44.

Variable Placement in Memory

- When you use operations at the C level, you're asking the compiler to do a set of bitwise operations on numerical values.
- The endianness does NOT influence the operations. Endianness only affects how the values are stored in memory.
- As a result, when you do a non-memory operation such as the one in the previous slide, the endianness is already sorted out.

Variable Placement in Memory

- However, what about this:
- int i = 0x11223344;
- char * c_point = (char *) &i;
- char c = *c_point;
- int j = (int) c;
- What's going on here?

Variable Placement in Memory

- Say int i (0x11223344) is at address 0x100. First, we get the pointer of i (0x100) and cast it to a char pointer.
- Then, we dereference the new char pointer to get the byte at that address 0x100.
- Now, j is equal to the byte at address 0x100.
- In a little endian machine, what is j?
- How about a big endian machine?

Variable Placement in Memory

- Say int i is at address 0x100. First, we get the pointer of i (0x100) and cast it to a char pointer.
- Then, we dereference the new char pointer to get the byte at that address 0x100.
- Now, j is equal to the byte at address 0x100.
- In a little endian machine, what is j?
  - 0x44
- How about a big endian machine?
  - 0x11

C: Type Casting

- Say we have the following declaration:
  - int i = 0xFFFFFFFF
- What's the bit configuration of i?
- What's the mathematical value?

C: Type Casting

- Say we have the following declaration:
  - int i = 0xFFFFFFFF
- What's the bit configuration of i?
  - All 1's of course
- What's the mathematical value?
  - -1

C: Type Casting

- Now say we have the following:
  - int i = 0xFFFFFFFF;
  - unsigned int ui = (unsigned int) i;
- What's the bit configuration of ui?

C: Type Casting

- Now say we have the following:
  - int i = 0xFFFFFFFF;
  - unsigned int ui = (unsigned int) i;
- What's the bit configuration of ui?
  - Still all 1's
- So... did that type casting actually do anything?

C: Type Casting

- Type casting between signed and unsigned is NOT negation (ie. type casting signed -1 to unsigned is not intended to produce 1).
- Type casting between the integer data types of the same size is merely telling the system and compiler how a bit configuration should be interpreted.
- The bitwise configuration does not change when converting between a signed and unsigned integer of the same length.

C: Type Casting

- A useful property of two's complement arithmetic is that the common operations behave the same between signed and unsigned values (assuming the same width).
- Consider 4-bit signed value of -4 (1100) and the 4-bit unsigned value of 11 (1011).

  1100 (-4)

  \+ 1011 (11)

  0111 (7)

- Despite the fact that we're adding a signed and unsigned number, the correct answer is produced using the normal bitwise addition.

C: Type Casting

- As a result (again, assuming the data types are of the same width), arithmetic operations will produce the same bitwise configuration regardless of whether the inputs are signed or unsigned.

- What are z1, z2, and z3?

```
int x = 0xFFFFFFF8; // -8
int y = 0x4;
int z1 = x + y;
int z2 = x + (unsigned int) y;
unsigned int z3 = (unsigned int) x + (unsigned int) y;
```

C: Type Casting

- All operations will produce the same bitwise result (0xFFFFFFFC).
- The mathematical “value” of the result, however, will depend on how you use the bitwise result afterwards.
- If you use it as a signed value, you get -4.
- If you use it as an unsigned value, you'll get some big number that I'm too lazy to precisely write out.

C: Type Casting

- As a result of this, type casting only matters in the case where the correctness of the operation depends explicitly on the mathematical value rather than just the bitwise configurations.
- What is such an operation?

C: Type Casting

- As a result of this, type casting only matters in the case where the correctness of the operation depends explicitly on the mathematical value rather than just the bitwise configurations.
- What is such an operation?
  - <, >, <=, >=
- Because the correctness of these operations depends on whether the inputs bit patterns are to be considered signed or unsigned, type casting matters.

C: Type Casting

- The point of no return:
  - The behavior I've described above is true if you subscribe to nice wrap semantics.
  - In general, C is not as nice as it could be.
- The truth is, in the following:

```
int x = 0xFFFFFFF8; // -8
int y = 0x4;
int z1 = x + y;
int z2 = x + (unsigned int) y;
unsigned int z3 = (unsigned int) x + (unsigned int) y;
```

- ...one of these zX technically commits the worst sin of all: undefined behavior... ooooOOOOooOOOoohh...

C: Type Casting

- The ugly truth: each data type has a “casting” rank.
- When an arithmetic operation is performed on two variables of differing data types, the variable of a “lower” rank type is implicitly cast to the “higher” rank type.
- ...plus a bunch of other rules.

C: Type Casting

- The rankings:
- long long = unsigned long long > long = unsigned long > int = unsigned int > short = unsigned short > char = unsigned char
- However, what if you have <signed int> + <unsigned int> (the signed and unsigned of the same data type are the same rank).

C: Type Casting

- From the C standard:
  - “Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.”
- In other words:
  - “signed data types are implicitly cast to unsigned if both are the same length”

C: Type Casting

- Thus:

```
int z2 = x + (unsigned int) y;
unsigned int z3 = (unsigned int) x + (unsigned int) y;
```

- In both cases, the addition will be tantamount to (unsigned int) x + (unsigned int) y.
- So what's the problem?
- (unsigned) x (0xFFFFFFF8) + (unsigned) y (0x4) is 0xFFFFFFFC
- z3 is going to be unsigned 0xFFFFFFFC, but z2 is going to be implicitly cast back to a signed int.

C: Type Casting

- But 0xFFFFFFFC is a very positive number as unsigned.
- When converted to signed, the value is going to be different, right? It's going to be a negative number.
- This is something of an overflow. What does the C standard say about type casting-based overflow?

C: Type Casting

- What the C standard says about casting:
  - “1. [...] if the value can be represented by the new type, it is unchanged”
  - “2. Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.” (what the...)
  - “3. Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.”
- Let's translate this:

C: Type Casting

- What the C standard says about casting:
  - “1. [...] if the value can be represented by the new type, it is unchanged”
  - 2. Otherwise, when casting from signed to unsigned, the resulting value is the unsigned value that has the same bit pattern (truncating if necessary).
  - 3. Otherwise when converting TO a signed and the value cannot be represented in it, the result is “implementation-defined” or an “implementation-defined signal” is raised. IE undefined.

C: Type Casting

- int z2 = x + (unsigned int) y.
- (unsigned) x (0xFFFFFFF8) + (unsigned) y (0x4) is unsigned 0xFFFFFFFC.
- Because we're saving to z2, we're implicitly type casting from unsigned to signed int.
- As an unsigned int, 0xFFFFFFFC is too large to fit in an signed int, hence, undefined.

C: Type Casting

- The reality is that most compilers will follow the nice bitwise rules and simply allow for the two's complement conversion.
- However, technically:
  - int i = -1; // OK...
  - unsigned int ui = (unsigned int) i; //OK...
  - i = (int) ui; // HOW DARE YOU?!
- Again, why does it matter?

C: Type Casting

- Code portability:
  - When behavior is undefined, that means a compiler writer or implementation of a language is free to do whatever it wants if that behavior occurs.
  - If you write code that expects an undefined behavior to act in some way (for example, if you assume that signed overflow wraps), then if you take this code and compile it with a compiler that implements different behavior, the program is wrong.

C: Type Casting

- Final notes:
  - Signed overflow is considered “undefined”. This includes overflow caused by shifts.
    - Consider 4-bit signed:
    - 1010 (-6) << 1 ?= 0100 (4)
    - This may be the result, but technically it's undefined.
  - Presumably as an act of mercy, the book treats signed overflow as if it always follows “wrap” behavior.
  - Converting between data types of different lengths has its own set of rules.

End of

The First Week

-Nine Weeks Remain-