CSE 3430
Overview of Computer Systems For Non-Majors

(George) Michael Green
Computer Science & Engineering
Hitchcock Hall 303
E-mail: green.25@osu.edu

Important Note

- **This course is NOT intended for BS-CSE and BS-CIS majors.**
- **If you are a BS-CSE or BS-CIS major or pre-major, drop this course and take CSE 2421, followed by CSE 2431. You will NOT be able to use CSE 3430 as part of your major program.**
- **This course is meant for students pursuing a Computer Science minor, for Data Analytics majors, and possibly some others, but not majors for any degree in CSE at Ohio State.**

Course Organization

- The course consists of 3 parts:
  - Computer hardware topics: Data encoding, integer operations, Boolean logic and logic gates, assembly language, and CPU hardware.
  - C18 (Same as C11) C language programming
  - Operating systems
- Stated simply, the course is a somewhat less advanced and condensed version of the two systems courses for CSE/CIS majors at Ohio State (CSE 2421 and CSE 2431).

Some Key Points from the Syllabus

- You should read the syllabus (if you have not already) and follow the policies stated there.
- Although the optional text on C is an excellent reference, it is optional; the OS text is required (but free online!).
- You can make Piazza posts on content which will count as part of participation (but not posts on assignments, scheduling, etc.), even if you think you understand everything perfectly! "Contribution" license used for Piazza (Sorry about the requests for money!).
- Academic misconduct: See the statement on the syllabus. If you are not sure if something is allowed, ***assume it is not***, and ask before doing it, or don't do it.
- Access, disability issues, etc.: Please be sure to request assistance if you need it, and I will refer you to SLDS so they can determine what you need, and I will make sure you get it.

Other Key Syllabus Points

- You need to include the Academic Integrity Statement in homework/lab assignments you submit.
- Late assignments only accepted within 24 hours, with 25% deduction, but let me know as early as possible if you cannot submit on time, and I may be able to give you an extension if you need to submit late for some reason beyond your control.
- We will not check homework solutions before they're submitted, but let us know if you have questions about how to do problems.
- You are responsible for testing and debugging your lab code (the only way to learn debugging well is by doing it), but we can give helpful guidance.
- Comments are part of writing software; be sure to include them in lab code filles for code that you write.

2 Other Key Syllabus Points

- If you need help, please ask, and please don't wait too long! Look at assignments as soon as possible after they're made (not just a few days before they're due!), so you can ask questions early!
- Do your best to keep up with the material, and ask any questions you have, as early as possible.

Goals of the Course (and the 3 parts)

- A. Discuss key “machine-level” ideas, i.e., how computers are organized internally and how they function – this is usually described as *hardware;* also, discuss how data is *encoded* (integers, characters, and floating-point data briefly, but other types of data more generally)
- B. Develop a knowledge of “low-level” programming using C (We'll discuss why C is still a very important language, even though it's quite old)
- C. Discuss some key ideas and concepts underlying *operating systems* that serve as an interface between application programs and the computer hardware (Operating systems are implemented exclusively or mostly in C)

Background and Course Materials

- 1. This is a relatively new course (about eight years old now).
- 2. There are many books on *computer architecture* and on *operating systems* but they are *much* too detailed for the purposes of this course.
- 3. Useful references:
  - Arpaci-Dusseau, Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*
    - (Here: http://pages.cs.wisc.edu/~remzi/OSTEP/) – I will assign operating systems reading from this online text, which can be accessed using the link above.
  - *C Primer Plus, 6th Edition*, Stephen Prata, ISBN13 9780321928429
    - (This text is quite useful for coding in C, but optional)

Background and Course Materials continued

- 4. *Important:* *Don't* miss classes! The attendance and participation portion of the grade is 5%; we have a 75% ***minimum*** attendance rule for non-exam days; ***you must attend a minimum of 3/4 of the classes on non-exam days to pass***. Falling behind, and/or missing assignments and/or submitting them late and then doing badly on exams is the most common reason why students who don't pass the course end up with a non-passing grade.

Course Materials

Notes:

- ***You*** are responsible for creating your own class notes, and YOU SHOULD DO THIS!

- The slides will usually be something akin to a list of bullet points to guide class discussion. You have to create your own detailed notes based on the class/discussion sessions and discussion on Piazza.

- What we say in live class sessions overrides what you may find online or anywhere else; so when you do your homework assignments and answer questions in the exams etc., they should be based on the class coverage, not what you may have seen elsewhere.

Grading

- You can see a list of assignments on the syllabus. Here's a summary:
  - 2 midterms, 20% each (1st on Part A only, 2nd on Part B only)
  - Comprehensive final exam, 35% (covers all 3 parts, A, B, and C)
  - 3 homeworks, total 10%
  - 2 labs, total 10%
  - Attendance and participation, 5%

Course grades

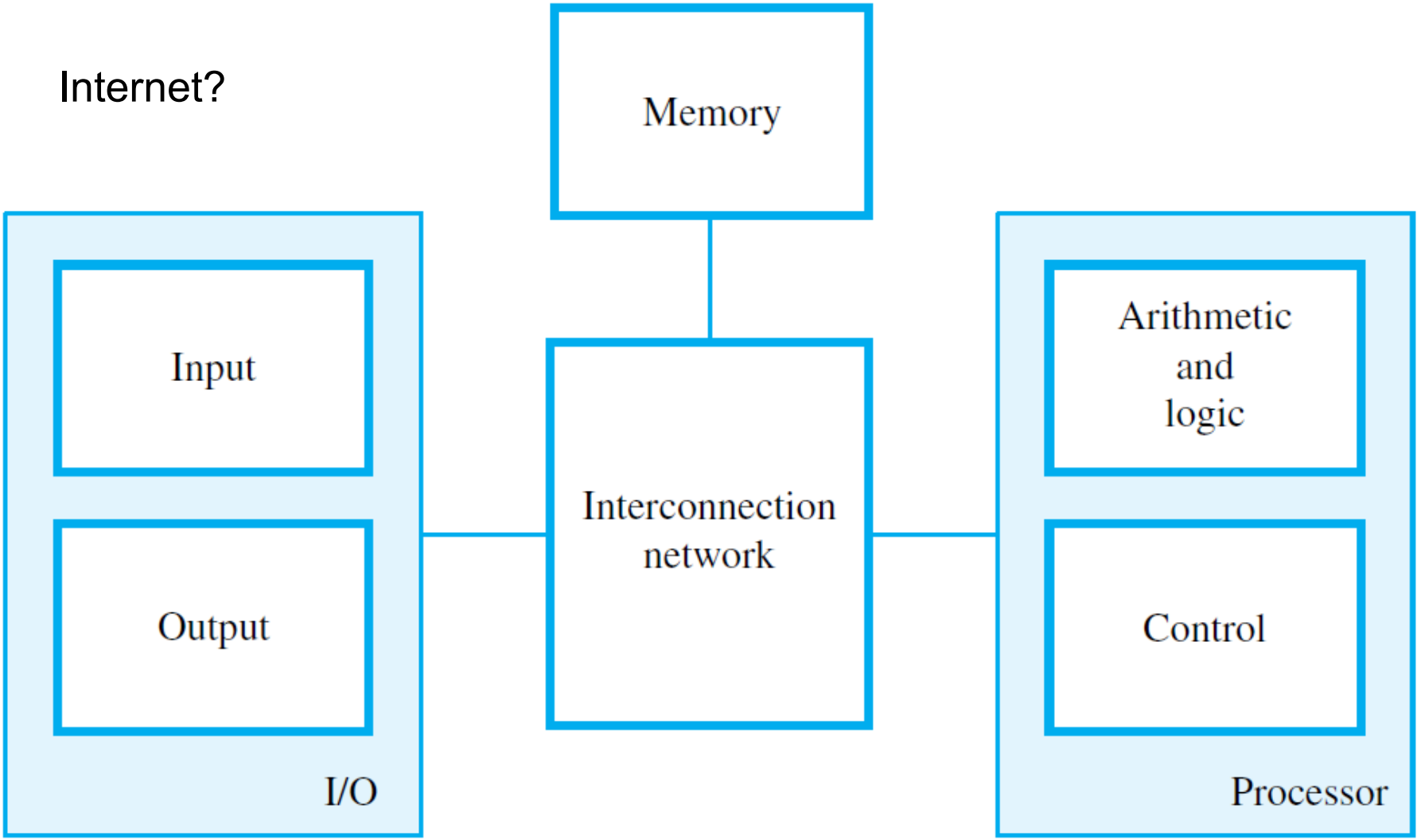
- Although I do not use "a curve," in the usual sense, if the class average is lower than expected (87%, a low B+) at the end of the semester, I will add the same number of percentage points to everyone's average on Carmen to bring the average up to approximately 87%.
- I do not make this "adjustment" till the end of the semester, but I will keep you posted on what the current "tentative adjustment" is.
- Typically, about 40% of students get an A, 40-50% a B, and 10-20% a C.
- Sometimes students may get a grade lower than a C, but this is virtually always due to missing assignments or getting very low scores on multiple exams. We can avoid this by being sure we determine why an exam score was low, and preparing better for later exams, and by making sure we don't miss assignments.

Main components of a computer

- Memory
  - Main memory (RAM, i.e., Random Access Memory) – volatile
  - Secondary memory (disks, flash memory etc.) – non-volatile
  - Cache (also volatile) memory
- Processor (CPU)
  - Arithmetic/logic circuits (ALU)
  - Timing/control circuits
  - Registers, and cache memory
- Input
  - Keyboard, mouse, touchpad, cameras, … (and many others)
- Output units
  - printers, speakers, … (and many others)

Main components of a computer

Internet?



Some initial questions

- How do the different components communicate with one another?
- Do you notice any missing component(s)/parts in the diagram?

Fundamental Concept #1

- Smallest unit of memory: *Bit* (**bi**nary digi**t**)
  - A bit may be 0 or 1 (abstraction of value)
  - There are hardware devices that can *store a bit* (these devices are also called bits)
  - You can *write* a new value (only 0 or 1!) into a bit or *read* the value currently stored in the bit; THAT"S IT!
- Computer's memory: Collection of a *huge* number of bits
  - **An ordered sequence of 1 or more bits: a *bit string*** (or string of bits)
- ***Everything*** in the computer (data of every type, instructions, etc.) is stored as a bit string (an ***ordered*** sequence of bits: 1100 and 0011 are distinct bit strings) – WHY?
- That is ***everything in the computer***; NO exceptions!
- Names, numbers, songs, pictures, emojis, *everything!* Even *programs!*
- ***Everything* in the computer is stored as a bit string!**

But … how?

- By ***encoding*** everything as a bit string …; encoding means converting the meaning/value to a bit string that can be used to represent it.
- A bit string with a single bit can encode two possible values:
  Represent the state of a light bulb *on* (1) or *off (*0); or
  whether a particular student *passed* (1) or *failed* (0) the first midterm;
  or *anything else* that has exactly two possible values.
- A bit string with two bits can encode up to four possible values (00, 01, 10, 11)
  Represent state of *two* bulbs in room (off,off / off,on/ on,off/ on,on); or
  whether a particular student failed/passed each of two midterms; or
  whether *two* students failed/passed the first midterm;
  or *anything else* that has exactly four possible values.
- *Major Problem*:
  Does "01" represent a particular student's fail/pass in two midterms? or
  fail/pass of two students in the first midterm …
  or even whether the two light bulbs are off, on respectively?
  Answer: There is *no way to tell just by looking at the bit string!* We just have to know (actually, the software that manipulates the encoded data has to know)!
  … but we will see more on this later.

Storing Information …

- Memory is organized into a sequence of *words* (an array of words)
- A word typically was a string of 32 bits in the prior generation of hardware (usually 64 today), but could be 8, 16, 32, …; there is no universal agreement about or definition of the size of a word. Unless otherwise stated, we will assume that it is 64 bits (this is the most common size in systems today, but this is an assumption we will make since it is very typical in general purpose systems today).
- Notice that what hardware manufacturers call a word may not be a word technically (we will see later that the Intel terminology is rather crazy in this way, because it's based on history)
- Often we talk of *bytes* where a "byte" is ***always*** a string of 8 bits (that is, byte has a universal definition). Thus a word (typically) consists of 8 bytes (if, as we assume, a word is 64 bits).

Storing Information (contd.)

- How do we represent a number?
  - First, why does, say, “265” denote two-hundred-and-sixty-five (and not 562, for example)?
  - Because we are using the *positional* notation; so the “5”, because it is in the “units” position, denotes 5 (times 1); the “6”, in the “tens” position, denotes *60* (6 X 10); the “2”, in the “hundreds” position, denotes 200; giving us a total of two-hundred-and-sixty-five
  - We apply the same idea to *binary numbers*; i.e., in an unsigned “binary number”, say, 11001, the rightmost digit is in the units position (2^0); the digit to its left is in the 2^1 position; the digit to its left is in the 2^2 position; the next one on the left is in the 2^3 position; etc.
  - So “11001” denotes $1*2^0 + 0*2^1 + 0*2^2 + 1*2^3 + 1*2^4$, i.e., 25
    What does “1001” denote? What does “1201” denote in binary?
  - Using the way described above, what is the *largest* number we can denote with 5 bits? Smallest?

Storing Information (contd.)

- What if we want to denote a number that is bigger than 31?

Answer: Use more bits!

- What is the largest number we can store in a *word* of memory?
  111…11 (a string of 32 bits, each being 1):

  $2^{32} - 1 = (1024 * 1024 * 1024 * 4) - 1 = 4294967295$

- What value does it denote?
- But what if we want *still larger* numbers?
- What can we do? Do you see?
- *Range* is an important concept in encoding.
- How can we describe the range of values that can be encoded using the binary number encoding scheme described above?

Formulas

- Using the way of forming (unsigned) binary numbers described above, using n bits, the largest number which can be encoded (represented) is equal to

  $2^n - 1$

- The smallest number which can be encoded is 0.
- This is referred to as the range of values which can be encoded, and the range given above always applies to unsigned binary numbers.
- The range, then, of course depends on n (how many bits we have to encode values); if n is larger, of course, we get a bigger range.

Storing Information (contd.)

- Above, we only considered non-negative numbers (The encoding scheme described is known as ***B2U***, or *binary to unsigned*). What if we want to also deal with *negative* numbers?
- Natural approach: Use the first bit as the "sign bit" and the remaining bits for the value: This kind of encoding is called ***binary to sign-and-magnitude (B2S)*. Generally, a first bit of 0 is used for non-negative, and 1 for negative,** so "0101" denotes +5 and "1101" denotes –5
- We will see later that, although B2S is intuitive, it does not work well in machines, and no system uses it for signed numbers.
- We will say more about some much better ways of encoding signed numbers below.

Arithmetic Operations

- How do we perform addition of two numbers? Subtraction?
  - The representation we choose should make it as uncomplicated as possible to do arithmetic operations on the encodings; this is one factor to be considered in choosing a good encoding scheme.
  - For the moment, let's just consider addition of two unsigned numbers, using B2U (Binary to Unsigned). Let's add two 4-bit numbers, 0101 and 0110 (First, what are these two numbers?):

    Carry: 0 1000

    1st number: 0101

    2nd number: 0110

    Sum: 1011

Notes on Addition

- Notice that we wrote carries for each pair of bits (even if 0) – we want to represent the addition operations for each pair of bits in a way that shows clearly what the hardware does.
- The carries can only be 1 or 0; it is NOT POSSIBLE to have "no carry," (which we use as a shorthand when doing decimal addition by hand) because it must be either 0 or 1 (because it is a bit value), so the carry should ***ALWAYS*** be written, whether 0 or 1 (you will lose points if you fail to write all the bit values)!
- Also notice that there is a carry used for the addition of the right-most pair of bits also (least significant bits, abbreviated ***lsbs***).
- Also, notice that a carry is generated for the addition of the last pair of bits (the most significant pair of bits, or ***msbs***); this carry is actually stored by the hardware in a one-bit flag called the carry flag, or C; we will say more about this later.

Notes on Addition (continued)

- The last carry bit is important to determine if there is OVERFLOW of some kind (we will discuss two different kinds of overflow for addition and subtraction).
- It's called overflow because we'll see later that this means there are not enough bits to store the accurate/correct result.
- OVERFLOW means that the sum generated by the addition hardware is NOT ACCURATE. Let’s see an example on the next slide.

Overflow Example

- Now, let's add the two numbers, 0110 and 1010 using B2U. with 4 bits:

  Carry: 1 1100

  1st number: 0110

  2nd number: +1010

  Sum: 0000

- Notice that the carry from the addition of the msbs is 1; this means there is overflow, and the result is incorrect. (It's obvious that the result is incorrect, right? We added 0110 (Decimal 6) and 1010 (Decimal 10), and the result is 0!).
- The problem is that the binary encoding of the correct result, which is 16, ***does not fit*** in 4 bits (that is, we need more than 4 bits to encode this value in B2U); as we saw above, the range for 4 bit B2U numbers is 0 to 15. When the result does not fit in this way, we refer to this as ***overflow***.

Why not add another bit?

- If the processor is adding two 4 bit numbers, and the sum requires 5 bits, why not just use another bit to store the result? [In other words, store a 5 bit result.]
- The answer is that this makes the hardware WAAAAAAAAY more complicated, and thus, *much more expensive* (*and slower*), but the performance is generally no better (or only marginally better)!
- Do you want to pay a lot more money for a machine that is slower and performs only a little better in terms of being able to store a slightly larger range of results? Neither does anyone else!
- This is why nobody (no manufacturer) builds a computer that uses an extra bit to store the correct result when there is overflow - it's simply not worth it!!!!

Real CPUs

- For the reasons stated above, all real CPUs generally use the same number of bits to store results that are used for the operands (the numbers being used to do the calculation).
- Also, real CPUs can generally only work on operands (and get results) of certain numbers of bits (8, 16, 32, and 64 bit operands/results are pretty much what all general-purpose CPUs have today).
- You can see the operand/result sizes are 1 byte, 2 bytes, 4 bytes and 8 bytes.
- Finally, for operations on multiple operands, all of the operands are the same size; designing a machine to work on operands of different sizes is waaaaay too complicated, and not worth it!!!

Detection of Overflow in Hardware: Flags

- All CPUs store the carry from the addition of the msbs somewhere, so that overflow can be detected.
- Processors store this last carry in a 1 bit flag called ***the carry flag***, abbreviated C.
- If C is 1 for an unsigned arithmetic operation, then the result of the last arithmetic operation is incorrect; if C is 0 for an unsigned arithmetic operation, the result of the last arithmetic operation is correct.
- Notice we referred above to ***unsigned*** arithmetic operations; processors can do both signed and unsigned operations (more later).

2's complement: A different approach

- Let's return now to signed encoding. Suppose our word size is 3 bits (for simplicity). This means there are 8 possible combinations of bits (for n bits, 2^n is the number of distinct encodings; do you see why? So, if n is 3, we can have 2 * 2 * 2 = 8 different bit strings).
- We want to have:
  - Roughly equal number of positive and negative nos.
  - As many numbers as possible (that is, every encoding should encode a unique value – we do not want 2 encodings of the same value).
  - Easy comparison for equality of numbers, and easy determination of sign of number.

2's complement (continued)

- Based on the above considerations, 000, 001, 010, 011 should be non-negative (note that 0 is neither positive nor negative); 100, 101, 110, 111 should be negative
- Evaluate this scheme using the criteria that we gave above. How does it do? [Let's look back at the prior slide.]
- This encoding scheme is called 2's complement, or ***B2T***
- Q: Which negative nos. should each of 100, …, 111 represent?
  - 100
  - 101
  - 110
  - 111

2's complement "theory"

- What happens if you subtract 1 from 000 (Notice that this is the same as adding -1 to 0)? What should the result be?
- The reason this is called 2's complement is because 000 is treated, if we are subtracting from it, as though it were (1)000, that is, with an implied 1 in the position of the ***most significant bit*** (or ***msb***).
- So, for three bit numbers, for purposes of subtraction, 0 is treated as if it were 8, 1000 in binary, but the msb is only ***implied*** (it's not really there, because we only have a three bit number!).
- So 111 should be -1 (because (1)000 – 1 = 111);
  similarly, 110 should be -2; 101 should be -3
- And 100 should be -4
- In effect, we can see what -k should be by subtracting k from 0; or, rather, from 2^3 (because we are using 3 bits here); if we were using 4 bits, for example, we would subtract from 2^4, or 16.

2's complement: negating a value

- Here is another 2-step algorithm that is much more often used to negate values (get the negative value that corresponds to some positive value *OR* get the positive value that corresponds to some negatvie value) in 2's complement:
  (1) Invert all the bits (change 1's to 0's, and 0's to 1's)
  (2) Add 1 to the result of step 1
- Suppose, for example, we want to negate 0101 (decimal 5), using a 4 bit encoding of signed numbers
  (1) Invert: 1010
  (2) Add 1: 1011
- Thus, 1011 is -5 in 4 bits in 2's complement, or B2T

Going the other way with negation

- Now, let's reverse what we did above, and see if it works.
- In other words, start with –5 in 4 bits, 1011, and negate.
- 1011 (-5)

  (1) Invert: 0100

  (2) Add 1: 0101
- It worked: We got back our original representation for 5!

1's complement

- 2's complement was developed based on 1's complement, or B2O, which is a simpler scheme for negation of integer numbers, but does not work nearly as well.
- If we have a non-negative integer in B2U, say 011 (decimal 3), we negate it using 1's complement by simply ***inverting or complementing*** all of the bits; that is, if a bit is 1, we change it to 0, and if it's 0, we change it to 1.
- This scheme looks simpler, but does not work well for arithmetic operations
  - Because we cannot use the same hardware in the ALU to do signed and unsigned operations.
  - For example, we need one adder for unsigned values, and a different adder for signed values.
  - This makes the hardware more complex, and also more expensive.
- Let's try adding two signed values using 1's complement, and see if we get the right result.

-1 in 1's complement

- First, let's get -1 in 4 bit 1's complement:
- 1 is 0001
- Invert/complement all bits to get -1: 1110
- Now let's add -1 to -1 in 1's complement (next slide).

Example in 1's complement

- Let's add -1 to -1 in 1's complement.
  - Carry: 1 1100
  - 1st number 1110 (-1)
  - 2nd number + 1110 (-1)
  - Sum 1100 (-3)

  - Wrong result!
  - Can we fix this so that the sum comes out correctly?

Mechanism to fix the problem

- For 1's complement addition, if we look at lots of examples, we will see that the result will be incorrect if both operands are negative, as in the example above, and in that case, if we add the same way we did before, the result will always be 1 less than the correct result (as in the problem above; we got -3, but the correct result is -2).
- Therefore, the simple solution is to use a carry of 1 for the 1st pair of bits when both operands are negative (have an msb of 1):
- Carry 1 110**1**
- 1st number 1110 (-1)
- 2nd number + 1110 (-1)
- Sum 1101 (-2)
- Correct!

Mechanism to fix the problem

- How can we get the hardware to use a carry of 1 to add the 1st two bits when both operands are negative? We can use an AND gate for the two msbs in the operands, and use the output of that gate as the first carry. Where both operands are negative, this works, as the example above illustrates.
- It also works if both operands are positive (0 AND 0 is 0, so 0 is used for the first carry).
- BUT what if one is positive and one is negative (0 AND 1 is 0, and 1 AND 0 is 0, so in this case, the first carry will be 0)?
- Let's try to add -3 and 2 in one's complement using the scheme described above (that is, use a first carry of 0), and see if the result is correct.

Add -3 and 2 in B2O

- Let's try adding -3 and 2 in one's complement (since 1 AND 0 is 0, the first carry is 0):
- Carry 0 0000
- 1st number 1100 (-3)
- 2nd number +0010 (2)
- Sum 1110 (-1)
- **Correct!**
- BUT does it ***always*** work for operands of different signs?

Add -3 and 4 in B2O

- Let's try adding -3 and 4 in one's complement (since 1 AND 0 is 0, the first carry is 0):
- Carry 1 1000
- 1st number 1100 (-3)
- 2nd number +0100 (4)
- Sum 0000 (0)
- Incorrect!
- Notice the result is 1 less than the correct result (which is 1).

Big Problem

- In the first case above, (-3 + 2), we used a carry of 0 for the first pair of bits, and the result was correct. In the second case, (-3 + 4), we used a carry of 0 for the first pair of bits, and the result was incorrect.
- When adding two number with different signs in B2O, getting the correct result depends on choosing the right carry for the first pair of bits, but making the right choice of the bit to use for the first carry requires more complicated hardware.
- See the next slide for a simple solution to the problem.

Adding numbers with different signs in B2O

- We can see in the two examples above (and in amny other examples), if the result is correct with a 1st carry of 0, then the LAST carry will ALSO be 0, BUT if the result is incorrect with a 1st carry of 0, then the LAST carry will be 1!
- We can use the fact above to fix the problem; ADD the last carry back to the result (do a 2nd addition) and then the result will always be correct, as long as there's no overflow (the correct result fits in the number of bits available)!
- The disadvantage of this is that it requires A SECOND ADDITION OPERATION (so 2 additions are needed) whenever we add numbers with different signs. This means addition takes twice as long in these cases, but ADDITION IS A VERY COMMON OPERATION in programs :-(

Determining if signs are different

- How can the addition unit in the processor determine if the two numbers being added have different signs (one has sign bit of 0, and the other has sign bit of 1)?
- Remember that the sign bit is the msb.
- The adder can input the 2 msbs (sign bits) to an exclusive-or (XOR) gate. XOR outputs a 1 if and only if exactly 1 of the two input bits is 1 (meaning the other must be 0).

Summary of Mechanism

- So the adder uses an XOR gate for the 2 msbs of the numbers being added to determine if it needs to do a 2nd addition. ***If the XOR gate output is 1, a second addition will be done***, using the last carry from the first addition as the first carry for the second addition.
- If the output of the XOR gate is 0 (this means the two operands have the same sign), only one addition will be done, as we described before.

Summary of B2O addition

- When adding two negative numbers in B2O, using a carry of 1 for the first pair of bits always works.
- When adding two non-negative numbers in B2O, using a carry of 0 for the first pair of bits always works also (it's just like B2U addition).
- When adding two numbers with different signs (XOR gate outputs 1), use a 1st carry of 0, but then add the last carry back to the result (do a second addition) to obtain the final correct result.

Mechanism to fix the problem

- Getting the hardware to work correctly for addition and subtraction using 1's complement is somewhat more complicated than for 2's complement, but it can be done (as we saw above).
- For multiplication and division, however, the hardware gets much more complicated, for 1's complement.
- For these reasons, among others, 1's complement is not used in modern hardware.
- Instead, 2's complement is universally used in hardware being built currently.

More on B2S

- We also looked at sign and magnitude (B2S - slide 18 above) for the encoding of signed integers.
- This scheme also looks simpler, just as 1's complement does, but it has similar disadvantages.
  - Arithmetic hardware is more complex, because different hardware is needed for signed and unsigned arithmetic.
  - As with 1's complement, since this makes the hardware much more complicated, B2S, or sign and magnitude is not used in modern hardware either.

2's complement arithmetic - more

- With 2's complement, the same hardware can be used for unsigned and signed arithmetic for ALL operations, *with only a very small wrinkle*.
- The hardware for addition, for example, can be exactly the same for unsigned and signed values
- ***Overflow***, however, must be detected differently (that's the wrinkle).
- Here's an example: 1101 (-3) + 1011 (-5):

Carry:	1	1110
1st number:		1101 (-3)
2nd number:	+	1011 (-5)
Sum:		1000

Here, the carry from the msbs is 1, so the carry flag will be 1, but the result is CORRECT, though the carry is 1!

2's complement overflow for signed addition

- As we saw above, when adding signed numbers, in some cases, if the carry from the msbs is 1 for signed addition with 2's complement, the result is still correct.
- There are also cases where the carry from the msbs is 0, and the result is incorrect! Let's add 4 (0100) and 4 (0100):

Carry:	0 1000	
1st number:	0100	(4)
2nd number:	+ 0100	(4)
Sum:	1000	(-8)

- The carry flag will be 0, but the result is -8, which is incorrect.

Overflow for signed addition

- What we actually need is to ***compare*** the carry from the bits to the right of the msbs (the second to the last carry) and the carry from the msbs (the last carry).
  - If these two carries are *the same*, there is no overflow.
  - If these two carries are *different*, there is overflow.
- Examples:
  - Look at the example above, with 0100 and 0100. You see that the carry from the bits to the right of the msbs is 1, but the carry from the msbs is 0; since the two are different, there is overflow.
  - Also look at the prior example for the addition of -3 (1101) and -5 (1011). Here, the carry from the bits to the right of the msbs is 1, but the carry from the addition of the msbs is also 1. Since the two carries are the same, there is no overflow.
- Therefore, all CPUs today have a second flag in addition to the carry flag which is used to detect overflow for signed operations.
- This flag is called the overflow flag, or O.

How to compare the last 2 carries

- To get the correct bit value for the overflow flag, or O, an exclusive-OR gate, or XOR gate is used.
- An XOR gate outputs 1 only if exactly one of the two input bits is 1. If both are 0, or both are 1, it outputs a 0.
- So, the last 2 carries are fed to an XOR gate, and the output bit is put into the O flag. This gives the correct value of the bit for the O flag.
- You can see that the scheme described above will give an O flag of 0 if the last two carries are the same (no signed overflow), or 1 if the last two carries are different (so there is signed overflow).
- This method of detecting overflow also works for 1's complement addition (see below).

The method above also works for B2O

- Comparing the last two carries using an XOR gate also works to determine overflow for B2O addition.
- Example:

Carry	1 0111
1st number	1011 (-4)
2nd number	1010 (-5)
Sum	0110 (6)

Result is incorrect (overflow), because last two carries are different, so O flag will be 1

Advantages of 2's complement

- Arithmetic is easy and uniform for positive and negative nos.
- Testing for equality or for positive/negative are easy
- Going from 3-bit to 4-bit is easy: just extend the sign bit; works for any no. of bits.
  - This is important because we often want to change the no. of bits used for storing some info
- Virtually all computers use 2's complement representation today because of these advantages.
- Range of values for n bit B2T numbers is -2^{n-1} to $2^{n-1} - 1$.
- So, for example, for 4 bit B2T numbers, the range of values is – 8 to 7.

B	Values represented		
$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	−0	−7	−8
1 0 0 1	−1	−6	−7
1 0 1 0	−2	−5	−6
1 0 1 1	−3	−4	−5
1 1 0 0	−4	−3	−4
1 1 0 1	−5	−2	−3
1 1 1 0	−6	−1	−2
1 1 1 1	−7	−0	−1

Addition and subtraction

- 2's complement has another advantage.
- Because it works for both unsigned and signed arithmetic, we can do subtraction by adding the negative of the number being subtracted.
- That is, a – b is just a + (-b).
- This means we can use an adder to do subtraction, if we can complement the second operand.
- Doing complementation with 2's complement seems a little tricky, but we can do it like this:
  - Invert the bits in the second operand (just input each one to a NOT gate, which just inverts, or "flips" the bit; if a 0 is input, 1 is output, and if a 1 is input, 0 is output);
  - Change the carry used for the first pair of bits from 0 to 1! This is equivalent to adding 1 to the complemented second operand.
- So, when the CPU does subtraction, it complements the second operand (flips the bits) and uses a carry for the first pair of bits equal to 1 (rather than 0, as usual for B2U or B2T addition).

Let's try it!

- Compute, with 4 bit operands, 4 (0100) – 3 (0011).
- Remember, we need to invert the second operand, 3, and then add using a carry for the first pair of bits of 1, not 0.
- Is the result correct?
- How do you know?

Result

Carry:	1	1001
1st number:		0100 (4)
2nd number:		1100 (-3 in B2O)
Sum:		0001

- The result is correct, because the last two carries are the same (both are 1), so the overflow flag will be 0.
- If the two bits are different, the CPU would store a 1 in the overflow flag.
- Try 4 – 5 with 4 bit operands using the method we described, and see what you get (there should not be overflow, right?).

Overflow for Subtraction

- Overflow for subtraction can be detected as discussed before, but remember that subtraction is always a signed operation (because it uses the negation of one operand).
- Therefore, overflow is detected by comparing the last two carries with an XOR gate, as previously discussed; if the last two carries are the same, there's no overflow (the O flag will be 0), but if the last two carries are different, the O flag will be 1, and the result is incorrect.
- See the example on the next slide.

Example of Overflow for Subtraction

- Let's compute, using 4 bit operands, –4 – 6 (the bold bit strings are the ones being added, and remember that the 1st carry is 1):

  Carry 1 0011

  1st number **1100** (-4)

  2nd number [0110] (6)

  2nd number

  inverted **1001**

  Difference 0110

The last 2 carries are different, so the O flag will be 1, and that means the result is incorrect, which we can see (the result obtained is 6, but the correct result is –10).