A Question about Compression (0:00-4:00)

- What if I take my essay and I compress it using WinZip again and again and again? Can I whittle it down to a single bit?
- No, of course this makes no sense, because we wouldn't be able to recover all that data from a single bit.
- But why do we hit the point at which we can compress no further?
- Huffman and other algorithms take advantage of redundancies, or patterns. If you've already represented your data with as few bits as possible, Huffman cannot find additional redundancies in the file to exploit.
- For instance, a file might have a lot of e's, so the uncompressed file has the bit pattern representing "e" over and over again. Huffman exploits this by representing e with a shorter encoding, say, a single bit.
- But once we have eliminated this pattern, there will no longer be anything to take advantage of. If our file has, say, 1000 e's, it will still consist of at least 1000 bits in no particular pattern.

Bitwise Operators (4:00-14:00)

- How do we usually change an uppercase letter to lowercase? Add 32, so A (65) becomes a (97)
- We can do this another way using bitwise operators.
- What do you notice about the difference between A and a, Z and z?

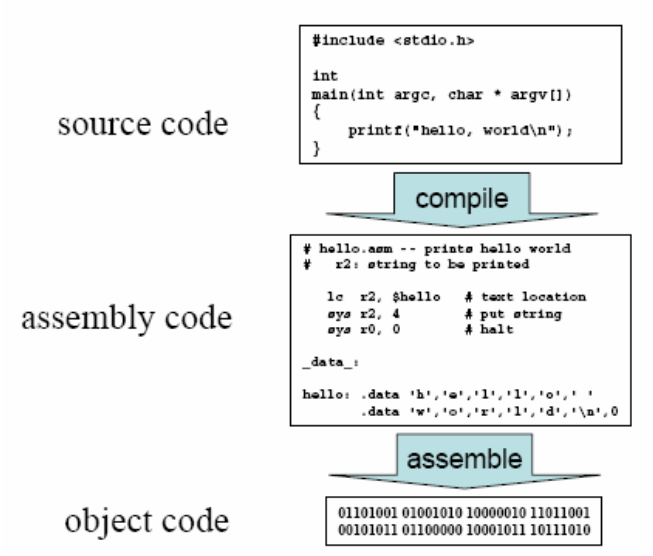
  A = 01000001 (65)
  a = 01100001 (97)
  Z = 01011010 (90)
  z = 01111010 (122)

- A and a only differ in one bit, the third from the left. Same with Z and z.
- So if we could find a way to "turn on" that bit, we could change uppercase letters to lowercase.
- We can do this by bitwise OR-ing a capital letter with 00100000. This is because 00100000 has a 1 in the bit that we want to "turn on."
- We call 00100000 a mask.
- How do we represent this mask when we are coding in C? Recall that 00100000 is 32 in decimal.
- So if we know that char c is in the range of capital letters, we can simply say c = c | 32 to apply the mask and turn on the third bit from the left.
- 01000001 | 00100000, or A |32, yields 0110001, or a.
- What if we want to make a lowercase letter uppercase? Now we need to turn off the third bit from the left.
- This we can do by bitwise AND-ing the lowercase letter with the mask 11011111. This will have the effect of "turning off" any bit that lines up with the zero.

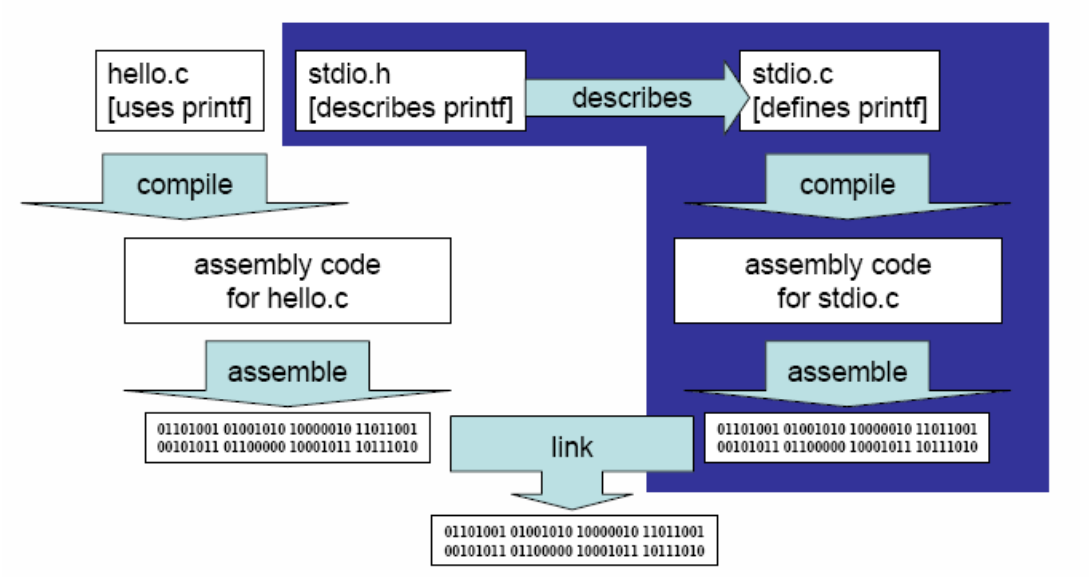
- How do we make this mask? We know that it is the 1's complement of our old mask. So instead of figuring out exactly what it is in decimal, we can represent it as ~32.
- Now we would say c = c & ~32.
- 01111010 & 11011111, or z & ~32, yields 01011010, or Z.
- You can view implementations of these algorithms in toupper.c and tolower.c
- Home exercise: use bitwise operators to swap two variables without using any temporary variables. (Hint: use the XOR operator.) See solution in swap2.c.

Underneath It All (14:00-25:00)

- What's going on under the hood?
- There are five processes that go into making code into an executable file:
  - Pre-processing
  - Compiling
  - Assembly
  - Linking
  - Executing
- When you #define a constant or #include a library, you are writing a pre-processor directive. This means that GCC first processes all the things that start with #.
- For instance, if you say #define FOO 3, GCC goes through all your code and copies and pastes 3 into every place that FOO appears in your code. Only then does compilation begins.
- So what's compilation all about? As we know, compilation at a high level takes your source code in C and spits out object code—zeros and ones that are understandable by your CPU and operating system.
- Actually, there is an intermediate step in which your compiler first transforms your source code into assembly code.



- Assembly code is much closer to the hardware than source code, but is still understandable to humans. In fact, we can write (something like) assembly code using a language called Ant.
- When you're writing assembly code, you have to explicitly work with memory registers in the CPU. So if you want to add two numbers, you have to tell the CPU from which memory registers to fetch the numbers and in which register to put the result.
- So when you write assembly code, you're working at a much lower level than when you write C. To go any lower would essentially be to hardcode in zeros and ones.
- Finally, GCC takes that assembly code, and writes it in zeros and ones that the computer can understand.
- But there's a catch. As we know, hello.c uses printf. Where is printf defined? In the library stdio.c. We indicate this to GCC by #include-ing stdio.h at the top of hello.c.
- What's really happening is this: GCC compiles hello.c into assembly code, which gets assembled into object code. *Meanwhile*, in parallel, GCC sees stdio.h in hello.c, finds stdio.c, compiles stdio.c into assembly code, and assembles that into object code. At the end of this, we have two object files hello.o and stdio.o which get combined into the executable a.out.

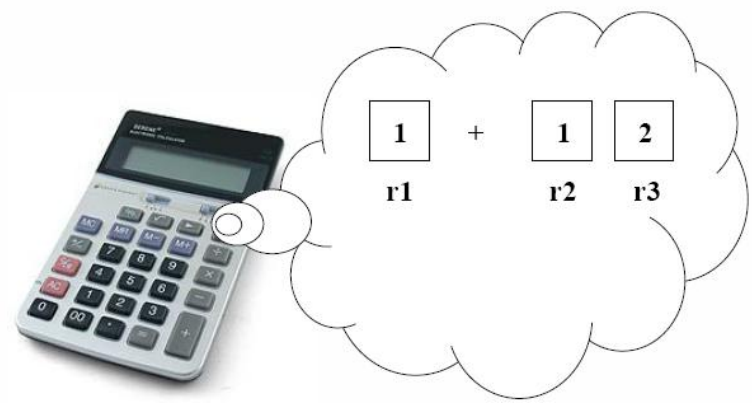


- The same thing happens whenever you include cs50.h or any other library.
- OK, we lied a little. GCC doesn't actually recompile things like stdio.c every single time someone calls printf. That would be silly!
- In reality, there are some things that are already compiled ahead of time and can just be linked to as necessary.

Underneath the Hood: Hardware (25:00-39:00)

- Let's think about a calculator as a piece of hardware with three memory registers: r1, r2, r3.
- If you press 1, 1 gets put in r1.

- Then you press + and 1 again. 1 gets put into r2.
- Finally, you press =. The calculator figures out what instruction what was called.
- It happens to be add. The calculator then knows to take the contents of r1 and r2, add them, and put the result in r3.



- The add function is hardcoded into the calculator.
- So when we say a CPU has an instruction set, we mean that it has some hardwired connections that tell the computer what certain sequences of bits should induce it to do.
- Well what else is in the box?
  - Registers: usually not that many, only 16 or 32, just for temporary memory
  - Program Counter: variable that keeps track of where in the program the computer is currently executing
  - Memory: what we know as RAM, what you control with malloc()
  - Control Unit and Data Path: implement the actual logic when you say + or &
  - I/O Devices: deals with keyboard, mouse, etc., that interact with user
- What can a CPU do? Here is the basic instruction set for Ant, which is a type of assembly code:

:: Arithmetic
- `add dst,src1,src2`
- `sub dst,src1,src2`
- `mul dst,src1,src2`
- `inc dst,const8`

:: Load constant
- `lc dst,const8`

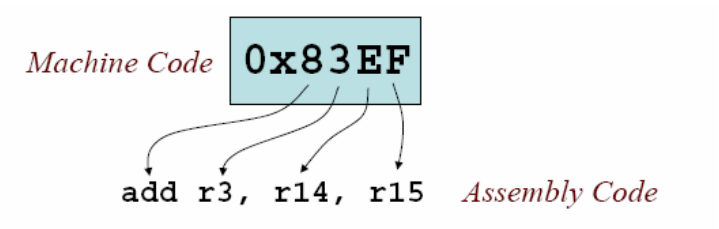
:: Bitwise/Logical
- `and dst,src1,src2`
- `nor dst,src1,src2`
- `shf dst,src1,src2`

:: Load and store
- `ld1 dst,base,uconst4`
- `st1 src,base,uconst4`

:: Branch
- `beq tgt,src1,src2`
- `bgt tgt,src1,src2`
- `jmp uconst8`

:: System
- `hlt`
- `in dst,channel`
- `out src,channel`

- Notice that basic instructions in a CPU are largely mathematical.
- Any time you write an instruction in Ant, you write it as one of the commands above followed by 3 arguments.
- Say you wanted to add the contents of r14 and r15 and put the result in r3. You would write this as "add r3 r14 r15".
- But since an instruction is just four "words," it is also just a 16-bit sequence that can be translated into a hexadecimal number:



- Let's see how to translate a C program into Ant. This is similar to what GCC does when it translates your C source code into assembly code. (Actually, the assembly instructions are a bit different because they are in x86 code rather than Ant.)

```
/* C code */
int test1 = 77;
int test2 = 96;
int totalPoints = test1 + test2;
```

```
# ANT code
    lc   r5, $test1     # address of test1 in mem
    ld1  r14, r5, 0     # value of test1
    ld1  r15, r5, 1     # value of test2
    add  r3, r14, r15   # sum of test scores
    st1  r3, r5, 2      # store in mem
    hlt                 # end of program code
_data_:
test1:          .byte 77
test2:          .byte 96
totalPoints:    .byte 0
# end of assembly file
```

- The first instruction means, "load the address of test1, which contains a constant, into the register r5."
- At the bottom of the program, we have placed 77 into test1 as a constant.
- Putting the address of test1 into the memory register is a lot like making a pointer. So C is kind of like assembly code with some nicer syntax. Using a low level language like C gives you more control, but also makes it easier to break your program.
- The next instructions mean "load the value at the address in r5 into r14" and "load the value at the location after the address in r5 into r15"
- At the bottom of the program, we have placed 96 right after 77 in memory

- In the third instruction, 1 is an offset indicating to go the address in r5 (test1), and then go 1 address later (test2)
- Finally, we add the values located in r14 and r15 and put the result in r3.
- Then we transfer the contents of r3 to the address two addresses after the address in test1 (totalPoints)

Viewing Assembly Code (39:00-41:00)

- We can see what your code looks like when it is translated into assembly language by executing the following command at the prompt:

```
gcc -S -o hello.c
```

- This outputs a file called hello.s which contains the contents of hello.c translated into assembly language
- It will be in the assembly language understood by your processor (probably x86), which you'll notice is not just like Ant
- We can also execute the following command:

```
gcc -o -g -Wa, -a, -ad hello.o > hello.lst
```

- Then open up hello.lst to see C interweaved with assembly language.
- Home exercise: execute these commands on your favorite programs to see how C translates into the assembly language of your processor