Hello, and welcome to my paper! I'm really happy to have you here! <3

In this paper, I describe a new compiler for the C89 programming language.

For good reasons that I will explain later, this paper must be 20 pages long. Due to unreasonable SIGBOVIK deadlines, I did not produce enough technical material to fill the minimum number of pages, so I will be taking my time and I have inserted several unrelated ASCII-art drawings.

1. Typesetting note

If you receive this paper in a raw text file, it may be difficult to read because of its two-column layout. It should be typeset in a monospaced font on pages 160 characters wide and 128 characters tall (this is a typical density of a line printer from the 1980s or 1990s). Many papers, including parts of this file, include special marks outside the text body to make the correct alignment easier to verify. This file contains no carriage returns or newline characters—just contains 160 characters and is padded with spaces. If you receive this paper as a PDF, it's not typeset the right way because it's printed in a very small font to conserve paper. Quoting really hard to read tiny hard fonts is good exercise for your eyes.

Your antivirus software may detect this paper as a virus, for good reasons that I will describe later. It is not a virus. ;-) 

2. Introduction

On any normal computer, a program is just a data file. It usually contains some header information that tells the operating system about what it is (for example, to confirm that it is a program and not some other kind of file; to tell the operating system about how much memory it needs, what the language is, etc.); some optional commands for the processor to execute. I'm not talking about stuff like small scripts and system programs which are not "executables"—commands (like 10 PRINT "HELLO") interpreted by some other program. I mean real executable files. These commands are low level instructions called "opcodes," and are usually bytes long. For example, on the popular and elegant X86 architecture, the single byte 0x40 is the "HALT" instruction, which halts the computer. (Could this be why ALT-F4 is the universal key code for quitting the current program? Intriguing!) (Of course, some instructions like HALT are strictly off limits for "user space" programs. When running a program, the operating system is in a "privileged" state where it can just do anything without worrying about such rude instructions instead alerting the operating system to the program's misbehavior. We'll talk about such instructions in Section 17.) The single byte 0x04 means "INC AX"—add 1 to the "AX" register—and a multibyte instruction like 0xF4 0x40 means "PUSH 0x40." All the time, the computer is just reading the next byte out of the program (or operating system, itself a program written using these same instructions), doing what it says to do, and then going on to the next one.

I wrote the opcodes above in hexadecimal notation, but they're just stored in bytes and memory as raw bytes (like above). Byte 0xF4 is not considered "printable" because old-timey computer people couldn't agree on how it should look. In DOS, it's the top half of an integral sign, like this:

```
0xF4
```

The first half of all bytes (0x00 to 0x7F) are defined in ASCII, which is standard across almost all computers now. When you look at the picture above, they are almost certainly represented as 0xF4. If you peeked directly at the bytes in this file, you would see a lot of 0x0D and 0x0A in that region sometimes they can be the flowers of a rose, like "----". To the processor, it means INC AX, since 0x40 is that opcode.

Now, for good reasons that I will explain later, this paper must contain some bytes that are not printable. Please proceed to Page 3 to continue reading this interesting paper.
Let's look at the printable opcodes available in X86. Don't actually read this table, but I will refer to it:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>0x20</code></td>
<td>SPACE</td>
</tr>
<tr>
<td><code>0x2C</code></td>
<td>SUB AL/AX/EAX &lt;- imm</td>
</tr>
<tr>
<td><code>0x2D</code></td>
<td>SUB AL/AX/EAX &lt;- imm</td>
</tr>
<tr>
<td><code>0x42</code></td>
<td>AND AL &lt;- 0x42</td>
</tr>
<tr>
<td><code>0x68</code></td>
<td>PUSH imm</td>
</tr>
<tr>
<td><code>0x6A</code></td>
<td>POP imm</td>
</tr>
<tr>
<td><code>0x89</code></td>
<td>MOV imm, AL</td>
</tr>
<tr>
<td><code>0x99</code></td>
<td>MOV imm, AX</td>
</tr>
<tr>
<td><code>0xAE</code></td>
<td>LEA AX, [EAX+imm]</td>
</tr>
<tr>
<td><code>0xBE</code></td>
<td>LEA AX, [EBX+imm]</td>
</tr>
<tr>
<td><code>0xDD</code></td>
<td>INT 0x08</td>
</tr>
<tr>
<td><code>0xEE</code></td>
<td>CALL</td>
</tr>
</tbody>
</table>

**5. The CISC Ridiculous**

It's not clear that it will even be possible to do basic things, and it was a pretty satisfying hacking challenge to work around its limitations. If you have some experience, you might want to give a little thought to the following puzzles:

- How can we load an arbitrary number (e.g. an address constant) into a register? Note that the immediate value in something like `PUSH imm` must be printable.
- Without the MOV instruction, how do we do loads and stores?
- Without the INT instruction, how can we even exit the program?
- How do we implement bitwise OR with the given instructions?
- The 16-byte Jmp (e.g. JNZ, JAE) instructions take only an absolute displacement. How do we do function (pointed) calls and returns?
- The displacement must be printable, which means it is always a positive number. How do we even do loops?

I will explain these problems and my solutions in later sections; I think they are each interesting. If you are not going to read the whole paper, which is likely, I think "18. Loops" and "17. Exiting and returning" are the most interesting/fun parts. Various parts of the compiler's design are intertwined with the many constraints, so there is no easy path through the whole idea. For now, let's warm up with the file format.

**6. Executable file formats**

In order for the compiler's output to be executable, it needs to be in a file that the operating system recognizes as program. This means that the header of the program needs to be printable too. We can rule out several formats that cannot possibly have printable headers:

On Linux, executables are ELF files. The first byte of these files is always 0x7F "ELF", which is not printable. Several other bytes in the header have to be zero.

On MacOS, executables are Mach-O files. These files always start with 0xfeedface, an amusing example of unprintable bytes whose hexadecimal representation nonetheless spells out words. It also requires a field called Mach_EXECUTE to be 0x02, among other problems.

On Windows, most executables are EXE Files. The modern version of this format is called Portable Executable (PE) and is used for 32- and 64-bit programs. It contains a required COFF subheader which always starts with 0x04500000 (the zero bytes not printable). For backward "compatibility", PE EXE files actually start with different EXE headers, which are actually programs that print something like "This program cannot be run in DOS mode."

and then exits. Windows recognizes a secret code that tells it to ignore that part and look at the "real" program.

... this eliminates the main executable formats for the modern x86 platforms. We can see that the EICAR program is a COM file, so clearly that is a possibility.

A DOS .COM file has no header. The entire program is just inserted into memory at the address 0x0100 and starts running. This level of simplicity is a dream for a SIGBOVIK Compiler Author, but it has a fatal flaw. In order to understand, we need to take a break and talk about segmentation.

**7. Segmentation break**

DOS is a 16-bit operating system, and a 16-bit number can only denote 65,536 (64k) different values. To allow programs to address more than...
PAPER, E X E  

This results in an file size of 409,600 bytes, which I believe is the same size, b/c part of it is 64k of the program image, the program image is loaded right after it, but starts at addresses DS:0x1000 rather than DS:0x0000. In 16-bit systems when we get this for free whether we want it or not, both COM and EXE files. Since this is just part of DS, programs will be able to write the data at any value and use the command line that the program is invoked with from the DOS prompt.

It's not necessary to understand this diagram since you are looking at a 1:1 scale model right now, i.e., the program itself. I'll point these out as we encounter them.

9. The Program Segment Prefix

The Program Segment Prefix, or PSP, is 256-bytes at the beginning of the data segment. Depending on how you look at it, DOS either overwrites the first 64k bytes of our program image, or the program image is loaded right after it, but starts at addresses DS:0x1000 rather than DS:0x0000. In 16-bit systems when we get this for free whether we want it or not, both COM and EXE files. Since this is just part of DS, programs will be able to write the data at any value and use the command line that the program is invoked with from the DOS prompt.

10. Relocations

You already saw the header structure (it's the title of the paper) and the relocation table (the full page of "..."), for normal programs, the purpose of the relocation table is for DOS to patch the program so that segment values aren't located in memory. If the program is loaded it is placed in a different spot in memory. When the program is loaded, DOS goes through all of the entries in the relocation table, and modifies the appropriate entries in the program by the appropriate amount to that at the location. Usually this location is part of an instruction segment prefix. In 16-bit systems when we want to be relative to the base program segment. We can't change segment values in the instruction's encoding. The instruction always acts between a particular value of the form "AND reg|mod/rm" or "AND mod/rm <- reg". In the instruction's encoding. The instruction always acts between a register and a "mod/rm", with two adjacent opcodes determining whether this is 32-bit or 16-bit code. The relocation for the "..." file is part of the instruction's encoding. The instruction always acts between a register and a "mod/rm", with two adjacent opcodes determining whether this is 32-bit or 16-bit code. If the relocation is one of the forms "..." or "..." then gets overwritten when we get there. The locations are given as segment+offset pairs, which is nice because we have multiple ways to refer to a given location. We simply solve for some regexp such that (seg + 16 + off = addr) and both seg and off are printable.

11. Addressing modes, temporary, calling convention

In any compiler, one must decide on various conventions for how variables are loaded in memory, how registers and temporaries are used, how arguments are passed, etc. so that all of such decision in ABC; some are basically normal and some are particular to the weird problems we have to solve. Let's talk about some of the limitations of the instruction set that we have access to, because those inform the low-level design.

In Figure 1, there are several instructions that look like this:

```
AND reg|mod/rm
```

These are each a family of instructions like:

```
AND AX < BX 
AND AX < [BX+SI] 
AND EBX < [EBX+DI] 
ADD AX < BX
```

where the (source on the right) and destination are given by some bits in the instruction's encoding. The encoding defines between a register and a "mod/rm", with two adjacent opcodes determining whether this is 32-bit or 16-bit code. If the relocation is one of the forms "..." or "..." then gets overwritten when we get there. The locations are given as segment+offset pairs, which is nice because we have multiple ways to reference a given location. We simply solve for some regexp such that (seg + 16 + off = addr) and both seg and off are printable.

```
AND AX < BX
AND AX < [BX+SI] 
AND EBX < [EBX+DI] 
ADD AX < BX
```

where the (source on the right) and destination are given by some bits in the instruction's encoding. The encoding defines between a register and a "mod/rm", with two adjacent opcodes determining whether this is 32-bit or 16-bit code. If the relocation is one of the forms "..." or "..." then gets overwritten when we get there. The locations are given as segment+offset pairs, which is nice because we have multiple ways to reference a given location. We simply solve for some regexp such that (seg + 16 + off = addr) and both seg and off are printable.
The "scaled index byte" (sb) has another table with 224 entries, which we won't get into. There is also a similar, but smaller, table for 16 bit addresses and 8 bit operands. Note that only part of this table is printable (marked with !(sb)), which means we can only use a subset of addressing modes. Notably:

- We can't do any register-to-register operations, like AND AX < AX.
- Most compilers use these instructions frequently!

- As a result, exactly one of the source or destination operand is located on in memory.

- The simple addressing modes can only paired with some registers. For example, AX and DI < [ESX] is allowed, but AX and DX < [ESX] is not. ![](en):which means the memory in the location pointed to by the value in the EDI register.

This is even more annoying than sx8 usually is. That said, the fact that we don't have register-to-register operations means that register allocation is significantly easier. We'll also only be dealing with some sets of temporary locations, using the ![](en) addressing mode. ![](en) is a nice extension of the sx8 addressing modes, so it is very similar to the same segment as the stack. In fact, since we initialized the stack pointer towards the middle of sx8 (it has to be printable, the maximum value would be 0x7f7a, but we use 0x6f69 to make the title more readable), we have the entire region from that to 0xFFFF to use for temporary registers. Each function's frame is seen below (for better visibility).

To perform a basic subtraction operation, whereas a traditional compiler is likely to emit an instruction like

0x29 0x02 SUB AX < DX

where AX = DX

ABC emits a sequence like

?7 MOV AX < [EBP+0x22] ; AX = tmp2
?6 MOV AX < [EBP+0x20] ; AX = tmp0 - AX

which is not so bad. (Note that we do not have a MOV instruction; this puzzle is a spoiler below.) We can now do much more work than this to perform a basic operation and optimize it meaningfully (especially things that reduce code size).

The ![](en) addressing mode denotes the location in memory at the address location plus the 8 bit offset (above). To encode this mod/rd, we need to write the displacement byte in the opcode. Since we must be able to reach the location, it actually always point 32 bytes before the first temporary, so that temporary 0 is accessed as [EBP+0x20].

With this idea in mind, here is a summary of ABC's low-level design:

- A C pointer is represented as a 16-bit address into the data segment.
- Anything addressable therefore needs to be stored in DS. This includes global variables, local variables and function arguments.
- Global variables are just allocated at compile time to some locations near the beginning of DS.
- A traditional compiler uses the machine stack to store local variables, but since there need to be in DS, not SS, we maintain a separate lookup table for each address in DS, which, after the global variables and grows towards larger addresses. This is called the local stack. The register EBA points 32 bytes before the local stack, so that we can use (ENBA) to efficiently access values.
- EBP always points 32 bytes before the "temp stack".
- Both stacks (and the machine stack) advance when we make a function call, so that the values of locals and temporaries persist across the function call. ABC only stores the return address on the machine stack.
- Aside from EBX, EBP, and ESP (the machine stack pointer), all other registers can be used for this purpose.
- Next, we need to implement a number of low-level primitives that let our program do computation. Let's warm up with something very basic.

** 12. Putting a value in a register **

When programming SX8 like a normal person, a very common task is to put an arbitrary value into AX (the low byte of AL), AB or BC. We usually need to be able to load arbitrary values, not just printable ones (but the value is part of the instruction encoding).

We do have some ability to load values. For example, we can encode

AND AX < 0x2020

since 0x20 is printable. This clears up most of the bits in AX, and then

AND AX < 0x4000

will always clear the remainder, since (0x40 & 0x20 = 0x00). With AX containing the value 0x2000, we can use the 24 times to reach the desired value. This totally sucks, but it works.

There are often more direct routes. We can XOR and SUB and AND with immediate 16 bit quantities to add in to AX. There is probably no "closed form" solution for the quickest route to a given value (the presence of both XOR and SUB makes this rather like a cryptic function/programming problem). We do have some ability to load values. For example, we can encode

AND AX < 0x2020

since 0x20 is printable. This clears up most of the bits in AX, and then

AND AX < 0xe000

will always clear the remainder, since (0xe0 & 0x20 = 0xe0). With AX containing the value 0x2000, we can use the 24 times to reach the desired value. This totally sucks, but it works.

There are often more direct routes. We can XOR and SUB and AND with immediate 16 bit quantities to add to AX. There is probably no "closed form" solution for the quickest route to a given value (the presence of both XOR and SUB makes this rather like a cryptic function/programming problem). We do have some ability to load values. For example, we can encode

AND AX < 0xe000

will always clear the remainder, since (0xe0 & 0x20 = 0xe0). With AX containing the value 0x2000, we can use the 24 times to reach the desired value. This totally sucks, but it works.

** 13. Moving between registers and memory **

Another useful kind of instruction is MOV AX < [EBP+0x20], which moves the value in memory into any arbitrary register (offset by 0x20 into AX). This is how we read and write temporary variables; the "AX" < [EBP+0x20] part is pretty obvious. Of course, we do not have the register to register moves (coffe). Fortunately, the XOR instruction is "information-preserving," so it can be used in a similar way. Of course, we only have here in the destination, when XOR "is" a MOV. In order to load from memory we use an instruction sequence like:

- Various...
- swt ax 0xe000
- OR AX < [EBP+0x20]

To write to memory, we do:

- OR Ax < 0xe000
- PUSH AX
- save value to write
- OR Ax < 0xe000
- OR Ax < [EBP+0x20]
- save values to write
- OR Ax < 0xe000
- XOR AX < [EBP+0x20]
- zero reading
- OR AX < [EBP+0x20]
- write value

This is almost... nice! But don't worry, it gets better.

** 14. Bitwise OR **

We don't have the OR instruction, but it can be computed with this trick.

This is the table of all possible bit combinations that A and B could have; and the OR operation is of course only dependent on the pair of bits at each position. First, observe (in your mind; it's not in the table) that the XOR of two are the same as A or B (it separates A or B into the cases where both bits in the input were 1, and the case where exactly one of the two bits is 1). So the result is the same if you compute their OR with &,

```plaintext
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

implies plus is also a multi-step process, described next:
to tell it that we're done and the program can be unloaded. In DOS, you make system calls by triggering a processor interrupt with the INT instruction, which a stub will finally check the value and say 'no, go on!'. We don't have access to this instruction, whose opcode is Ox02. Also, INT Ox02 has no functionality, like printing strings and reading from the keyboard, read from and to magnetic disks, and so forth. The Ox02 instruction is sad to go without it. (The EICAB test virus uses self-modifying code to create two INT instructions; one is to print the string and the second is to exit.) In DOS, Ox21 is the most useful one; you set registers to some values to access dozens of different functions.

INT Ox21 is so common that it appears in the Program Segment Prefix that it's always loaded at the beginning of the data segment. It's just sitting there amidst some zeros:

```
  DS:0x004A 0x00 0x00  ADD [BX+SI]  AL  <AL
  DS:0x004C 0x00 0x00  ADD [BX+SI]  AL  <BL
```

It even tautologically has RETF (far return from function call) immediately after it. However, it was probably invented by some puzzlemaker of years past, exactly for this kind of situation. (I don't actually know why it's there!) RETF pops both a return address and return segment, so if we could manage to put a return address on the stack (not hard) and the code segment (we don't know it, but we could probably use the relocation table to write it somewhere) beneath it, and then somehow transfer control to DS:0x004C, we'd have a fully general INT Ox21 to use. It would even help with the loop problem (last section) since it lets us return to an arbitrary address, and could conceivably even let us escape the confines of always executing code within the initial code segment CS (because RETF modifies CS). But speaking of confines, none of this reflects how an INT Ox21 works. The interrupt instruction is like executing code out of DS. Too bad, so sad. (This idea might pan out for a future project.)

Jumping the program to a non-printable instruction is also a bit questionable, though it's not an instruction that we wrote there, so this might still work. Anyway, along the way the system waits to serve the ovo lacto vegetarian with vegetarian food that causes him to eat non-vegetarian food that the customer himself brought with him! When...

This is not hopeless. The way interrupts actually work is to stop the current execution (saving the state of the registers on the stack) and then return that value. Why do you think the table itself should be called the "interrupt vector", containing addresses of all INT Ox21 (and Ox02) code? Exactly for this kind of situation. (I don't actually know how many interrupts other systems use for example by replacing the address for INT Ox21 with the address to subscribe to and make the INT Ox21 code to do instead of the original INT Ox21 handler so that everything still works. The INT Ox21 address is not immediately useful, because we can't control it directly. The CALL DX command, which is the text that should be called the "interrupt vector", is nothing more than a table. Each interrupt has a number, and each address is a 32-bit segment:offset pair. So the address at Ox4A = 0x004A is the location of DOS's code for INT Ox4A. In 16-bit programs, there's a special register (see Figure 2) about the operating system; you can just jump directly into it if you want. The interrupt vector is an extremely brittle thing. Also, the timer interrupt handler has to perform certain functions, and in case of some interrupts, it returns the stack segment is set to Ox00 when our program starts (we can't change it), other possible uses amidst a bunch of sensible ones. This instruction is not hopeless, the way interrupts actually work is to stop the current execution (saving the state of the registers on the stack) and then return that value. Why do you think the table itself should be called the "interrupt vector", containing addresses of all INT Ox21 (and Ox02) code? Exactly for this kind of situation. (I don't actually know how many interrupts other systems use for example by replacing the address for INT Ox21 with the address to subscribe to and make the INT Ox21 code to do instead of the original INT Ox21 handler so that everything still works. The INT Ox21 address is not immediately useful, because we can't control it directly. The CALL DX command, which is the text that should be called the "interrupt vector", is nothing more than a table. Each interrupt has a number, and each address is a 32-bit segment:offset pair. So the address at Ox4A = 0x004A is the location of DOS's code for INT Ox4A. In 16-bit programs, there's a special register (see Figure 2) about the operating system; you can just jump directly into it if you want. The interrupt vector is an extremely brittle thing. Also, the timer interrupt handler has to perform certain functions, and in case of some interrupts, it returns the stack...
The ARPL instruction takes two argument bytes which just have to be printable; the instruction we actually encode is

```
0x63 0x79 0x61 ARPL [ECX+0x61] <- DI
```

The ASCII sequence is "cy", as in see ya, which we follow with an unexecuted exclamation mark for emphasis. You can find the string "cy!" in the code segment on page 16 if you're good at Where's Waldo stuff!

```
* 18. Loops *
```

The last major problem involves control flow. In printable x86 we have available a family of instructions Jcc+disp8. Jcc stands for "jump (on) condition code", and consists of 15 opcodes:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Acrobat Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>p 0x70</td>
<td>Jmp if overflow</td>
<td></td>
</tr>
<tr>
<td>r 0x72</td>
<td>Jump below</td>
<td>JNAE, JC</td>
</tr>
<tr>
<td>s 0x73</td>
<td>Jump not below</td>
<td>JNB, JNE, JNC</td>
</tr>
<tr>
<td>t 0x74</td>
<td>JZ jump zero</td>
<td>JE</td>
</tr>
<tr>
<td>u 0x75</td>
<td>Jump not zero</td>
<td>JNZ</td>
</tr>
<tr>
<td>v 0x76</td>
<td>JBE jump below or equal</td>
<td>JBA</td>
</tr>
<tr>
<td>w 0x77</td>
<td>JNB jump not below</td>
<td>JNBE</td>
</tr>
<tr>
<td>x 0x78</td>
<td>JS jump if sign</td>
<td></td>
</tr>
<tr>
<td>y 0x79</td>
<td>JNS jump not sign</td>
<td></td>
</tr>
<tr>
<td>z 0x7A</td>
<td>Jump if parity even</td>
<td>JPE</td>
</tr>
<tr>
<td></td>
<td>0x7B  JNP jump if parity odd</td>
<td>JP0</td>
</tr>
<tr>
<td></td>
<td>0x7C  JL jump less</td>
<td>JNGE</td>
</tr>
<tr>
<td></td>
<td>0x7D  JNL jump not less</td>
<td>JNL</td>
</tr>
<tr>
<td>- 0x7E  JLE jump if less or equal</td>
<td>JNG</td>
<td></td>
</tr>
</tbody>
</table>

This is a fairly full set of conditions (although we are missing the last one, JNBE/JU, with opcode 0x7F). Each of these consults the processor's FLGS register and tests for a certain condition. FLGS is updated on many operations; for example, the "zero flag" ZF is set to 1 if the result of certain operations is zero, such as if "SUB [EBP+0x24] < AK" ends up writing 0x0000 into memory, and ZF is cleared to 0 if not. The JZ instruction jumps if ZF is set, and just continues on to the next instruction or whatever JZF has an alias, J (jump equal); they are the same exact opcode because when you subtract two equal numbers, you get zero. Since it is common to want to set the appropriate FLGS without actually subtracting, the CMP (compare) instruction is like SUB but it only updates flags. We have a version of the CMP instruction in printable x86, so all is well so far.

These particular instructions are Jcc+disp8, so we provide an 8-bit displacement. The address of the current instruction is stored in the EIP ("instruction pointer") register. When EIP points at Jcc+disp8, EIP is set to the instruction immediately after it (EIP+2) and then if we jump, incremented further by disp8. The disp8 byte is treated as signed, so jumps can go upward or downward. Unfortunately, all printable displacements are positive! This allows us to conditionally skip code, but only downward, and only between 32 and 127 bytes.

This subset won't even be Turing-complete if we can't jump backwards; all programs will terminate because the instruction pointer only increases. What actually happens when we reach the end of the code segment? If EIP is 0xFFFF and we execute a single-byte instruction like INC AX, EIP just continues on to 0x00010000; the EIP register is 32-bit despite us struggling with 16-bit segments and offsets. This instruction is right after the code segment, and indeed contains whatever followed our code segment in the program image. So we could conceivably break free of the 64k code segment. Unfortunately, performing a jump when in this weird state still just jumps downward, and the situation is very brittle (see Section 31 for some ideas and problems). However, there is a special case on the processor, probably for compatibility with an earlier processor; it's right there in the pseudocode for this instruction in Intel's manual [INTC'01]:

```
IF condition
THEN
  EIP <- EIP + SignExtend(DIS8)
  IF OperandSize = 16
  THEN
    EIP <- EIP AND 0000FFFFH;
    IF;                     (sic -tom7)
    ELSE (* OperandSize = 32 *)
      IF EIP < CS.Base OR EIP > CS LIMIT
      THEN
        EIP;

Specifically, if we are right at the end of the code segment, and our jump's displacement takes us past the end, then we "wrap around" to the beginning, because EIP is bitwise-anded with 64+0xFFFF. This means that our program can do one backwards jump, from the end of the segment right back to the beginning.

We're approaching the data section now, so it's time to take another break! Here it is:

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x72</td>
<td>JB</td>
<td>Jump below or equal</td>
</tr>
<tr>
<td>0x73</td>
<td>JNP</td>
<td>Jump if parity odd</td>
</tr>
<tr>
<td>0x74</td>
<td>JZ</td>
<td>Jump zero</td>
</tr>
<tr>
<td>0x75</td>
<td>JS</td>
<td>Jump if sign</td>
</tr>
<tr>
<td>0x76</td>
<td>JBE</td>
<td>Jump below or equal</td>
</tr>
<tr>
<td>0x77</td>
<td>JNB</td>
<td>Jump not below</td>
</tr>
<tr>
<td>0x78</td>
<td>JL</td>
<td>Jump less</td>
</tr>
<tr>
<td>0x79</td>
<td>JNL</td>
<td>Jump not less</td>
</tr>
</tbody>
</table>
```

Now you're looking at the PSW.
"Putting a value in a register."
Anyway, one backwards jump is enough! We can set things up so that whenever we need to jump backwards, we instead jump forward until we’re at the end of the segment, then jump across that boundary (overflowing back to the beginning) and then keep jumping forward until we get where we need to be. This is delicate, but it works.

One other issue with jumps is that we can only jump a fixed distance; there is no equivalent to MOV EIP <- AX to jump to a computed location. We need this functionality to implement two C features:

1. Function pointers (the destination of a function call is not known at compile time) and returning from functions (the function can be called from multiple sites, so we need to know which site to return to).

2. **19. The ladder**

To solve the various problems with jumps, we build the program around what’s called a “ladder” in the code. The whole program is broken up into small blocks of code. Each one is given a sequential “number” (this has nothing to do with the memory location, just its sequence in the list of blocks). Each block starts with a “rung,” which is the following code:

```
DEC SI
JNE -disp8
```

where disp8 is a printable displacement that brings us downward to the next block. We decrement the SI register to count down to the block we want, and if it is Not Zero yet, then we jump to the next one. If zero, we execute the block. Inside a block, if we ever want to perform a jump to some arbitrary block dest_block, then we can compute:

```
offset = (dest_block - current_block) mod num_blocks
si = (if offset = 0 then num_blocks else offset)
jmp to next rung
```

Every block knows its current number, so the offset is just a constant. Note that the destination block’s number may be before the current block, which is why we need to mod by the total number of blocks (yielding a non-negative result). SI cannot be zero, because the first thing we do is DEC it, so a self-loop requires setting to num_blocks, a full cycle.

To perform a jump to a code location not known at compile time (e.g. from a return address (block number) on the stack, we can just perform the same computation as above. We do not have an efficient mod operation (implementing it seems to need loops, in fact, a circularity!), so instead we actually compute (dest_block - current_block) + num_blocks.

This is always positive as needed, but requires forward jumps to make an entire cycle around the entire ladder (“Turn the dial to the left, passing zero and the first number...”).

The blocks are laid out sequentially in the program until we get too close to the end of the segment; when we do, we make sure to perform an unconditional jump across the segment boundary, wrapping around. This jump need not DEC SI. In fact, most programs do not fill the entire code segment, so we end up padding the end and beginning of the segment with jumps to span the unused space. For these padding jumps, we definitely don’t want to DEC SI, both because that’s more instructions to execute, and because we don’t know the amount of padding ahead of time (see the section on Assembling below).

There are many annoyances! A jump cannot be too short (less than 32 bytes) or too long (127 bytes). The viable range is large enough to build nontrivial programs, but it is a significant constraint for us.

We don’t have access to a non-conditional JMP instruction. There are a few tricks for simulating it. When computing a jump to a known label, we can just know the state of flags because we’ve just performed some computation. Even when doing a jump to a computed block number, we know that the result of subtraction is not zero, so we can always use the JNZ instruction. Occasionally we need to do a jump without knowing our state at all. XOR always clears the Overflow flag, so something like

```
XOR AX <- [DI]
XOR AX <- [DI]
JNO disp
```

keeps AX unperturbed and always performs the jump. A little shorter is

```
JNO disp
JO (disp - 2)
```

which jumps to the same target whether the Overflow flag is set or not, but is more annoying because we need to keep track of two displacements.

** 20. Assembling **

Assembling the program is the process of generating actual instruction bytes (here, printable ASCII from some semi-abstract representation of instructions [in ABC, this is the LL1HOP language discussed in the next section]. Assembling has a self-dependency: In order to generate

...
A program consists of a series of labeled blocks. JumpCond pairs a condition (signed and unsigned comparisons, etc.) with a jump to a label. The possible conditions for JumpCond are: "eq", "neq", "lt", and "gt". Each condition is associated with a jump to a specific label. The jump is taken if the condition is true. For example:

```c
JumpCond cond, label
```

This language is an assembly-like language that has explicit *data* layout but does not require any explicit *data* layout. The only way to jump to a non-constant destination is with PopJumpInd, available. Since opcode 0x7F (Jump Greater) is not actually printable, condition (signed and unsigned comparisons, etc.) with a jump to a specific label. The jump is taken if the condition is true. For example:

```c
JumpCond cond, label
```

This language is an assembly-like language that has explicit *data* layout but does not require any explicit *data* layout. The only way to jump to a non-constant destination is with PopJumpInd, available. Since opcode 0x7F (Jump Greater) is not actually printable, condition (signed and unsigned comparisons, etc.) with a jump to a specific label. The jump is taken if the condition is true. For example:

```c
JumpCond cond, label
```

This language is an assembly-like language that has explicit *data* layout but does not require any explicit *data* layout. The only way to jump to a non-constant destination is with PopJumpInd, available. Since opcode 0x7F (Jump Greater) is not actually printable, condition (signed and unsigned comparisons, etc.) with a jump to a specific label. The jump is taken if the condition is true. For example:

```c
JumpCond cond, label
```

This language is an assembly-like language that has explicit *data* layout but does not require any explicit *data* layout. The only way to jump to a non-constant destination is with PopJumpInd, available. Since opcode 0x7F (Jump Greater) is not actually printable, condition (signed and unsigned comparisons, etc.) with a jump to a specific label. The jump is taken if the condition is true. For example:

```c
JumpCond cond, label
```

This language is an assembly-like language that has explicit *data* layout but does not require any explicit *data* layout. The only way to jump to a non-constant destination is with PopJumpInd, available. Since opcode 0x7F (Jump Greater) is not actually printable, condition (signed and unsigned comparisons, etc.) with a jump to a specific label. The jump is taken if the condition is true. For example:

```c
JumpCond cond, label
```

This language is an assembly-like language that has explicit *data* layout but does not require any explicit *data* layout. The only way to jump to a non-constant destination is with PopJumpInd, available. Since opcode 0x7F (Jump Greater) is not actually printable, condition (signed and unsigned comparisons, etc.) with a jump to a specific label. The jump is taken if the condition is true. For example:

```c
JumpCond cond, label
```
The CIL language is typed, with one important use of this being that we determine the calling convention for a function pointer from its type. This includes the size of the return address slot, which is set on the locals stack and shared between the caller and callee, as well as the number and sizes of the arguments, as they are on the stack. We make the representation (Word64, Word64, Word32) of integral types (which converts e.g. an 8-bit word to a 16-bit word) explicit about whether they perform sign extension. We are careful to distinguish between 8-, 16- and 32-bit quantities throughout the compiler, because printables work with all three widths, and we can produce significantly better code if we can use the correct width (e.g., a simple example, loading a 16-bit word is much cheaper than the zero-extended 32-bit version).

Some low-level ideas are threaded throughout the compiler. In the case of "out" and "exit", for example, these are available to the programmer if she simply declares them:

```c
int _out8(int int);
int _exit();
```

They can be called like _exit(1), but are translated to the Bulitin function on an exact function call. It is not permitted to take their addresses.

Unlike LLVM/NO, we have both expression forms of operators and "cond" forms. The expression forms evaluate to 1 or 0, whereas the cond forms are of use as a combination of an expression and a call.

Optimizations try to put these in the most useful form for later work.

** 24. Optimization **

CIL code is optimized via a series of conservative transformations until there is no more optimization opportunity. We do not have to worry about optimizations being dead variable removal and constant folding, which is a convention used by code generated by the compiler, but not by the one.

We are more possible here, but since these problems are not specific to printables only, we did not include the one.

The optimization phase is also responsible for eliminating some features from the language that we don't need to think about them when converting to LLVM/NO:

- Multiplication. In printable x86, we have access to the MUL instruction, but only when both operands are constant and have a direct immediate value (operands 0x6b, 0x69).

- Comparison ops. Expressions like LessEq are transformed into

GtIf(cond, ...,), since we don't have any way of comparing values without also branching.

- String literals. These are replaced with references to globally-allocated arrays.

- Global initialization. All initialization code for globals (e.g. int global = 15) is moved into a wrapper around the main function.

These tasks aren't really optimizations, but we want to perform optimizations both because it makes the translation simpler and because it needs to at least be aware of their existence so that it doesn't e.g. reinroduce string literals after they have been eliminated.

** 25. Converting to CIL **

The frontend of the compiler uses the cikl library [C177/00] to parse the input code into an ML datatype called *AST*. The details of this language are mostly uninteresting and can be built from more fundamental forms. The "syntactic" correspondence to CIL is direct, and when we convert to LLVM/NO, we remove "syntactic" (and sometimes semantic) things. For example, for a loop is broken apart into a few goto's. This is a pretty big change, and it makes the translation simpler.

The optimization phase for CIL replaces the Times expression with a call to the above function that implements multiplication by repeated addition.

** 26. Limitations **

ABC has some limitations, some of which are fundamental and some of which are simply due to the unconscionably strict SIGUVOK directives:

- Floating point is not available. We have access to none of the floating-point operations; prior to the Intel 8086, support for floating point was usually provided in software anyway, so this helps avoid anachronism.

- Standard libraries are not available. Since we can only call the DOS INT 0x21 handler one time, and we use that to exit, there is no way for us to access functions or to invoke their interfaces; we could conceivably write their own device drivers using I/O ports (see the next section), but this usually also involves using or implementing hardware interrupts, so probably wouldn't pan out.

- mallow/free. This can be supported in software, with no significant limitations other than the amount of memory available.

- Opened widths. Though ABC architected supports most operations at 8, 16, and 32 bit widths, most operations are only implemented for 16 bit operands. This is easily fixed, but should be done with some care to the correctness and performance.

- Performance. Multiplication is linear time, since we use a software routine (if we can be sure we have access to the one). It will not significantly impact the choice of which algorithm to use, but will allow us to implement certain control flow techniques; these issues can make algorithms perform asymptotically worse than they should.

- Division and modulus. These need to be done in software like multiplication, which is trickier than usual due to the lack of efficient bit shifts. Note that many computer processors don't have an integer division instruction (e.g. Alpha, 6802), so this is not even that weird.

- Struct copying. Not a huge deal, but it means emitting code that copies struct field-by-field because we don't have anything like memcpy, and around the time of a function call or return, the state of the machine is pretty delicate.

- sizeof. Actually sizeof is so easy I just went and implemented it. It is used instead of writing this sentence. I saved further time by not deleting the previous sentence.

- Bit fields. These are garbage so nobody implements them unless they have to. No fundamental limitation here, although the compiler does assume that values have an address.


** 27. Programming **

In this case, there is one nice piece of hardware that is standard on DOS-era computers, and that grabbed a standard set of port numbers which were originally intended for a /O and I/O. In fact they are little used for I/O, and in fact their names are IOP and OUPS. These are part of a family of C0 instructions that interact with peripherals on the motherboard. DOS uses these to implement some of its own system calls (e.g. to talk to the disk controller to implement the file system), but I/O ports are sometimes also used by application programs.

A natural thing to do when thinking about "printable x86" would be to have the paper print itself out, i.e., a quine. This would be quite challenging given the ratio of accessible data (64k data segment + embedded in the 64k of code) to the size of the paper itself (409k), but it might be possible. Sadly, the major obstacle is that we can't print it.

Let's imagine of kind of miracle, though, two of the opcodes available to us in printable x86 are practically made for I/O. In fact they are little used for I/O, and in fact their names are IOP and OUPS. These are part of a family of C0 instructions that interact with peripherals on the motherboard. DOS uses these to implement some of its own system calls (e.g. to talk to the disk controller to implement the file system), but I/O ports are sometimes also used by application programs.

In this case, there is one nice piece of hardware that is standard on DOS-era computers, and that grabbed a standard set of port numbers which were originally intended for a /O and I/O. In fact they are little used for I/O, and in fact their names are IOP and OUPS. These are part of a family of C0 instructions that interact with peripherals on the motherboard. DOS uses these to implement some of its own system calls (e.g. to talk to the disk controller to implement the file system), but I/O ports are sometimes also used by application programs.

A natural thing to do when thinking about "printable x86" would be to have the paper print itself out, i.e., a quine. This would be quite challenging given the ratio of accessible data (64k data segment + embedded in the 64k of code) to the size of the paper itself (409k), but it might be possible. Sadly, the major obstacle is that we can't print it.

Like some kind of miracle, though, two of the opcodes available to us in printable x86 are practically made for I/O. In fact they are little used for I/O, and in fact their names are IOP and OUPS. These are part of a family of C0 instructions that interact with peripherals on the motherboard. DOS uses these to implement some of its own system calls (e.g. to talk to the disk controller to implement the file system), but I/O ports are sometimes also used by application programs.

In this case, there is one nice piece of hardware that is standard on DOS-era computers, and that grabbed a standard set of port numbers which were originally intended for a /O and I/O. In fact they are little used for I/O, and in fact their names are IOP and OUPS. These are part of a family of C0 instructions that interact with peripherals on the motherboard. DOS uses these to implement some of its own system calls (e.g. to talk to the disk controller to implement the file system), but I/O ports are sometimes also used by application programs.

In this case, there is one nice piece of hardware that is standard on DOS-era computers, and that grabbed a standard set of port numbers which were originally intended for a /O and I/O. In fact they are little used for I/O, and in fact their names are IOP and OUPS. These are part of a family of C0 instructions that interact with peripherals on the motherboard. DOS uses these to implement some of its own system calls (e.g. to talk to the disk controller to implement the file system), but I/O ports are sometimes also used by application programs.
This section contains the C source code that was compiled into this paper. It may be interesting to see how the code (e.g., string literals) make their way into the data for the paper. You may also laugh at my many mistakes.

- I'm playing music, which has some dependency on timing, but there is no way to get access to the system clock. Instead, I use for loops with built-in constants determined empirically. At least the technique of relaying on the CPU's cycle timing for delays above strings, which is what we want to turn off the channel.

- Also, remember when we talked about the relocation table, and how it has to corrupt some pair of bytes in our file? That's right here: --> XX <--.

- I also still need to fill up 20 pages in this ridiculously small font!

- copy.Paper, Copyright (c) 2010 Tom Murphy VII P.B.D. This copyright notice must appear in the compiled version of this program. Otherwise, please distribute freely.

- Plays music in a simplified ABC notation, on the command line, or one of several built-in songs.

** 28. Running, debugging **

- Speaking of running the program, old-style EXE files no longer run on 64-bit versions of Windows. So if you do not have an old DOS computer around with a sound card, you can run DOSBox as a Windows program.

- One of your real directories as a "hard drive". To mount an emulator. DOSBox is an excellent choice. It runs on pretty much all modern CPUs, it does not run on DOS, but on GNU Linux you can just use DOS and tends to just work. You have to do something like

** 29. PAPER.C **

- Executing this program in DOS, with an AdLib-compatible sound card (such as the Sound Blaster) configured at 0x386, will play some music. The music to play is specified on the command line, using a subset of a standard text-based music format called ABC (ABC'05). For example, invoking

```
Paper.PEX C4C4G4A4A4G8C8F4E4E4A4D4A4C8
```

- Will play a segment of the "Now I know my ABC's" song and then exit.

- The language supported is as follows:

  - A-G Basic notes
  - A-3 Same, up one octave
  - Rest (Prefix) Sharp
  - Flat (Prefix) Natural - does nothing since key of C is assumed
  - Up one octave
  - Down one octave
  - 2-B (Suffix) Set dummy note to this many eighth notes

- Up to three simultaneous tracks can play, all using the same dumb-sounding organ-like instrument, by separating tracks with |. DOS tracks are also command line, so quote the argument, like PAPER.PEX "A|B|C".

- Running PAPER.PEX with arguments like "-song" will play a built-in song. Available songs include: -alphabet, -plumber, and "bluebird". There's plenty of space in the data segment for more!

- Running PAPER.PEX without any arguments will play a default song.

- to copy.Paper, con

- to copy it to your console, or COPY PAPER.PEX LPT1 to copy it to your simulated computer's printer (it's empty: it doesn't have one). But why bother? You're reading PAPER.PEX right now!

- I used DOSBox frequently during development, and modified its debugger, especially for understanding the header values are actually used. The ABC compiler outputs each of the intermediate languages for 20 or so thousand lines of code, it compiles them down into the programs installed in an emulator. DOSBox is an excellent choice. It runs on pretty much all modern CPUs, it does not run on DOS, but on GNU Linux you can just use DOS and tends to just work. You have to do something like

** Mount C:\(DOWNLOADS\ABC\)**

- in order to mount one of your real directories as a "hard drive". To verify that PAPER.PEX is printable with no blopping or funny characters, you could do

```
COPY PAPER.PEX CON
```

- Also, remember when we talked about the relocation table, and how it has to corrupt some pair of bytes in our file? That's right here: --> XX <--.

- For example, invoking

```
PAPER.PEX C4C4G4A4A4G8C8F4E4E4A4D4A4C8
```

- Will play a segment of the "Now I know my ABC's" song and then exit.

- The language supported is as follows:

  - A-G Basic notes
  - A-3 Same, up one octave
  - Rest (Prefix) Sharp
  - Flat (Prefix) Natural - does nothing since key of C is assumed
  - Up one octave
  - Down one octave
  - 2-B (Suffix) Set dummy note to this many eighth notes

- Up to three simultaneous tracks can play, all using the same dumb-sounding organ-like instrument, by separating tracks with |. DOS tracks are also command line, so quote the argument, like PAPER.PEX "A|B|C".

- Running PAPER.PEX with arguments like "-song" will play a built-in song. Available songs include: -alphabet, -plumber, and "bluebird". There's plenty of space in the data segment for more!

- Running PAPER.PEX without any arguments will play a default song.

```
** 28. PAPER.C **
```

- This section contains the C source code that was compiled into this paper. It may be interesting to see how the code (e.g., string literals) make their way into the data for the paper. You may also laugh at my many mistakes.

- I'm playing music, which has some dependency on timing, but there is no way to get access to the system clock. Instead, I use for loops with built-in constants determined empirically. At least the technique of relaying on the CPU's cycle timing for delays above strings, which is what we want to turn off the channel.

- Also, remember when we talked about the relocation table, and how it has to corrupt some pair of bytes in our file? That's right here: --> XX <--.

- If you load this program in a debugger and look at memory approximately starting at C3:FF00, you'll see the XX changed to something else (unpredictable).

- I also still need to fill up 20 pages in this ridiculously small font!

- unsigned char *meta_note = "Now this is the part of the data segment that stores the values of the parameters for this song. This is actually a string constant in " the program itself, so you'll see it again when I show you the source code later. We have almost 64kB of space to store stuff, although most of this segment is not used for the stack of local variables and arguments, and would be used for malloc as well, if it were implemented. Storing a string like this is basically free, because "everything in it is printable, aside from the terminating \0 "

- As well as lightly-commented XREFs for the command line, so quote the argument, like PAPER.PEX "A|B|C".

- Running PAPER.PEX with arguments like "-song" will play a built-in song. Available songs include: -alphabet, -plumber, and "bluebird". There's plenty of space in the data segment for more!

- Running PAPER.PEX without any arguments will play a default song.

- Also, remember when we talked about the relocation table, and how it has to corrupt some pair of bytes in our file? That's right here: --> XX <--.

- If you load this program in a debugger and look at memory approximately starting at C3:FF00, you'll see the XX changed to something else (unpredictable).

- I also still need to fill up 20 pages in this ridiculously small font!
// No sound.
return 0;
}

// We pick octave 4 as the base one; this is fairly canonical and
// benefits us since this array is all printable. Note that A4 is
// higher than C4, since octave 4 begins at the note C4. This
// array maps A...G to the corresponding MIDI note.

unsigned char "octave4 =
\"*A - 57
\"*B - 59
\"*C - 48
\"*D - 50
\"*E - 52
\"*F - 53
\"*G - 55

// Parse a character c (must be capital A,B,C,D,E,F,G) and
// interpret any suffixes as well.
int ParseNote(unsigned char *ptr, int c, int *idx) {
    int midi;
    int offset = c - (int)'A';
    int nextc;
    midi = octave4[offset];
    for (; ; ) {
        nextc = (int)ptr[(*idx)++];
        switch (nextc) {
            case 'Y':
                // Up octave.
                midi += (int)12;
                break;
            case 'V':
                // Down octave.
                midi -= (int)12;
                break;
            default:
                // Not suffix, so we're done (and don't consume
                // the character.)
                return midi;
        }
    }
    *idx = *idx + (int)1;
    return (unsigned int)200 * midi;
}

// Parse the song description (ptr) starting at *idx. Updates *idx to
// point to the next character after the parsed note. Updates *len to
// the length in some unspecified for-loop unit. Returns the MIDI note to
// play next, or 0 if the song is done.
int GetMidi(unsigned char *ptr, int *idx, unsigned int *len) {
    int c = (int)*idx;
    int midi_note = *len + ParseLength(ptr, idx);
    return midi_note;
}

// Code.

int main(int argc, unsigned char **argv) {
    Channel channels[3];
    unsigned char *song = argv[1];
    int i, n, next_channel;
    MakeArgString(argv);
    song = GetSong(argv);
    // Initialize channels. Note that this will just blow the
    // stack-allocated channels array if there are more than
    // two in the input string!
    num_channels = SplitChannels(song, (Channel *)[4]);
    Quiet();
    for (i = 0; i < num_channels; i++) InitInstrument(i);
    for (; ; ) {
        int ch, all_done = 1;
        // At each tick (whose rate is governed just by the time
        // it takes to do this loop), reduce each channel's ticksleft:
        if (ch < n_channels(ch)) {
            Channel channel = channels[ch];
            int midi_note = channel->midi_note;
            if (midi_note != (int)0) {
                int ticksleft = channel->ticksleft;
                all_done = 0;
                if (ticksleft > (int)0) {
                    channel->ticksleft = ticksleft - (int);
                } else {
                    int new_note = GetMidi(channel->song, channel->idx, channel->ticksleft);
                    channel->midi_note = new_note;
                    if (new_note == (int)0) {
                        // Quiet the channel -- forever!
                        PlayNote(ch, (int)0);
                    } else {
                        PlayNote(ch, new_note);
                    }
                }
            }
            if (all_done) break;
        }
        Quiet();
        return 0;
    }
}
** 30. Is this useful for anything? **

No. This is a SIGBOVIK paper. <3

** 31. Future work **

There are many code size optimizations possible, and while nontrivial programs can fit in 64k (such as the one in this paper!), larger ones will simply run against that boundary. Typically, a factor of about 4 can be gained through a few hard but straightforward optimizations. Can we breathe free of this 64k boundary? Earlier we noted that when our execution exceeds CS:*0FFF, it simply continues to CS:*0D010000 unless a facility is executing (a facility) that is pointing to bytes that are part of our program image (this text is there, in fact), so conceivably we could write code here. One significant issue is that interrupts, which are constantly firing, push 16-bit versions of CS and IP onto the stack, and then REET (return) to that address if an interrupt happens while we are executing in this extended address space, we will return to CS:*0000 (0x2FF0). If we had anecdotally interesting, a good way to return to the normal 16-bit code segment (i.e., to perform a backwards jump), but we do not know of any other way to be able to globally suppress interrupts, like by using our single illegal instruction interrupt during initialization, with the interrupt handler pointing "just to code" that we control (and not from from it). This leaves the interrupt flag cleared, as discussed. The compiler will be non-functional may very, because the operating system will no longer run, but we might still be able to do very remarkable 8-bit/16-bit code.

With interrupts suppressed, we can't use the interrupt trick to return to CS:*0000. However, my reading of the Intel manual [INTC] seems to imply that our interrupter's code can be forced into 16-bit mode (thus being subject to the 0x2FF0 overflow) with an address size prefix. However, we thus not seen as possible in DOSBox.

Given how unusual this situation is, it may even be a bug in DOSBox's C/S/86 jump table. Having access to a full megabyte of code (it still is a fit to fit in the EXE container) would be exciting, since it would allow us to build much more significant systems (e.g., standard library and a floating point emulator); more investigation is warranted here.

I initially designed CIL with the thought that it could be used for multiple small "compile C to X" projects. This has been true, but can occasionally be of legitimate use for low-level tasks or tasks where the reasonable and familiar high-level syntax pays for the effort of writing a simple backend. (When making such a decision I like to also weigh the effort by the enjoyment of each task: i.e., the cost is like (1 - fun of writing backend) * time writing backend vs (pain of writing low-level code by hand) * time writing low-level code by hand... but I have been informed that not all computer work is done purely for fun.) This *portable assembler* application of C remains relevant today, and CIL or LLVM/MIP is a much simpler GCC or LLVM.

Anyway, I discovered that the design of such a thing is not so easy. While it is possible to "compile away" certain features by turning them into something "simpler," it's not straightforward what feature set you wish to compile away. For example, associating |, &&, and r~ in one setting, |, ~, and r~ in another setting, may very well be present instead of in. We normally think of the | and << shift operators as being fundamental, but in ABC they are inaccessible. I find the expression forms like "a < b" much easier to think about than the combined term-and-branch version, but the latter is much better when targeting x86, important for producing readable code in ABC. I do that it could be possible to write code for this niche where certain constructs could be compiled away in favor of others, at the direction of the compiler author, but such a thing is firmly future work.

On the topic of taking away, one might ask: What is the minimal subset of C by which we could imagine using ABC?

There are some trivial subtractions: We omit the BEGIN and END (CGX,62, lowerc b and it does not see useful; a few of these comments present construct names are also removed. The "ASCII Adjust After Addition" are currently unused, but they are on AA in a predictable way, they could provide ways to improve the readability of load immediate values. But we're also removing the surface, not increasing it. And speaking ofload immediate values, we certainly make more use from a known starting point by INC and DEC, taking at most 0x7FFF instructions (half the size of the MOV instruction). Others are also removed. For example, our lowerc or register to a known value, which today requires two or more printable values whose bitwise AND is 0. Sadly, though we could go through some pains to remove bytes from the front of the code, it's unfortunate is we make a label that looks like "lowercase letters" or "alphanumeric" jumps out; we rely on the contrast in the late lowercase letters (LOC) and the basic ops in the early punctuation (ABD/XOR), not to mention that the EXE header bar is written within the existing constraints with access to both "email" (0x20A0) and "large" (0x267e) constants.

Others have produced compilers for high-level languages with very reduced instruction sets. In an extreme case, Dolan shows [MOV13] that his machine language instructions on its own are complete (note however that this requires a "single absolute jump" to the top of the program, an implementation to what we encounter in printable X86, only we do not cheat by inserting any out-of-gamut instructions). Another extreme machine language example is MIP, implemented a C compiler that produces only MOV instructions (MVP16). I didn't look at it while writing ABC (sillier!) but he avoids using any JMP instruction the same way that I avoid the "program generation" instructions but rewriting the interrupt handler). While awesome, the problem is somewhat different from what we solve; here we fundamentally concerned with how bytes appear in the executable, which influences what operands are accessible and their arguments and addressing modes, but is not the only constraint created. For example, in MOV-only compilation, the program's header does not need to consist only of MOV instructions, and the compiler's output does not suffer the same severe code and data limitations that DOS EXEs do. (The executables it produces are extremely large and slow; they also seem to have non-MOV initialization code.) The MOV instruction is also very rich, and no versions of them are printable!

Of course, everyone knows that even unary numbers (just like one symbol a repeat given a number of times) is Turing complete, via Godel encoding. So what's the big deal?

** 32. Acknowledgements **

The author would like to thank the fastidious SIGBOVIK "Program" Committee for "Evaluating" my paper.

** 33. Bibliography **


Please see http://tom7.org/abc for supplemental material.
** Appendix **

Here is a histogram of every character that appears in this file. There are no non-printable bytes.

<table>
<thead>
<tr>
<th>char</th>
<th>byte</th>
<th>number of occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>!</td>
<td>175076</td>
</tr>
<tr>
<td>0x21</td>
<td>@</td>
<td>1242</td>
</tr>
<tr>
<td>0x22</td>
<td>#</td>
<td>1771</td>
</tr>
<tr>
<td>0x23</td>
<td>$</td>
<td>3216</td>
</tr>
<tr>
<td>0x24</td>
<td>%</td>
<td>557</td>
</tr>
<tr>
<td>0x25</td>
<td>&amp;</td>
<td>3922</td>
</tr>
<tr>
<td>0x26</td>
<td>*</td>
<td>113</td>
</tr>
<tr>
<td>0x27</td>
<td>(</td>
<td>467</td>
</tr>
<tr>
<td>0x28</td>
<td>)</td>
<td>8770</td>
</tr>
<tr>
<td>0x29</td>
<td>*</td>
<td>955</td>
</tr>
<tr>
<td>0x2A</td>
<td>+</td>
<td>526</td>
</tr>
<tr>
<td>0x2B</td>
<td>,</td>
<td>233</td>
</tr>
<tr>
<td>0x2C</td>
<td>.</td>
<td>2088</td>
</tr>
<tr>
<td>0x2D</td>
<td>/</td>
<td>27690</td>
</tr>
<tr>
<td>0x2E</td>
<td>0x7E</td>
<td>7339</td>
</tr>
<tr>
<td>0x2F</td>
<td>252</td>
<td></td>
</tr>
<tr>
<td>0x30</td>
<td>0</td>
<td>1045</td>
</tr>
<tr>
<td>0x31</td>
<td>1</td>
<td>1618</td>
</tr>
<tr>
<td>0x32</td>
<td>2</td>
<td>697</td>
</tr>
<tr>
<td>0x33</td>
<td>3</td>
<td>1346</td>
</tr>
<tr>
<td>0x34</td>
<td>4</td>
<td>2594</td>
</tr>
<tr>
<td>0x35</td>
<td>5</td>
<td>373</td>
</tr>
<tr>
<td>0x36</td>
<td>6</td>
<td>321</td>
</tr>
<tr>
<td>0x37</td>
<td>7</td>
<td>185</td>
</tr>
<tr>
<td>0x38</td>
<td>8</td>
<td>296</td>
</tr>
<tr>
<td>0x39</td>
<td>9</td>
<td>102</td>
</tr>
<tr>
<td>0x3A</td>
<td>;</td>
<td>1423</td>
</tr>
<tr>
<td>0x3B</td>
<td>&lt;</td>
<td>396</td>
</tr>
<tr>
<td>0x3C</td>
<td>&gt;</td>
<td>482</td>
</tr>
<tr>
<td>0x3D</td>
<td>?</td>
<td>1048</td>
</tr>
<tr>
<td>0x3E</td>
<td>@</td>
<td>49</td>
</tr>
<tr>
<td>0x3F</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>0x40</td>
<td>0x7F</td>
<td>11867</td>
</tr>
<tr>
<td>0x41</td>
<td>@</td>
<td>559</td>
</tr>
<tr>
<td>0x42</td>
<td>B</td>
<td>366</td>
</tr>
<tr>
<td>0x43</td>
<td>C</td>
<td>664</td>
</tr>
<tr>
<td>0x44</td>
<td>D</td>
<td>1516</td>
</tr>
<tr>
<td>0x45</td>
<td>E</td>
<td>2762</td>
</tr>
<tr>
<td>0x46</td>
<td>F</td>
<td>860</td>
</tr>
<tr>
<td>0x47</td>
<td>G</td>
<td>200</td>
</tr>
<tr>
<td>0x48</td>
<td>H</td>
<td>291</td>
</tr>
<tr>
<td>0x49</td>
<td>I</td>
<td>422</td>
</tr>
<tr>
<td>0x4A</td>
<td>J</td>
<td>118</td>
</tr>
<tr>
<td>0x4B</td>
<td>K</td>
<td>192</td>
</tr>
<tr>
<td>0x4C</td>
<td>L</td>
<td>261</td>
</tr>
<tr>
<td>0x4D</td>
<td>M</td>
<td>490</td>
</tr>
<tr>
<td>0x4E</td>
<td>N</td>
<td>773</td>
</tr>
<tr>
<td>0x4F</td>
<td>O</td>
<td>248</td>
</tr>
<tr>
<td>0x50</td>
<td>0x50</td>
<td>2784</td>
</tr>
<tr>
<td>0x51</td>
<td>P</td>
<td>8234</td>
</tr>
<tr>
<td>0x52</td>
<td>Q</td>
<td>147</td>
</tr>
<tr>
<td>0x53</td>
<td>R</td>
<td>375</td>
</tr>
<tr>
<td>0x54</td>
<td>S</td>
<td>372</td>
</tr>
<tr>
<td>0x55</td>
<td>T</td>
<td>154</td>
</tr>
<tr>
<td>0x56</td>
<td>U</td>
<td>81</td>
</tr>
<tr>
<td>0x57</td>
<td>V</td>
<td>127</td>
</tr>
<tr>
<td>0x58</td>
<td>W</td>
<td>2821</td>
</tr>
<tr>
<td>0x59</td>
<td>X</td>
<td>62</td>
</tr>
<tr>
<td>0x5A</td>
<td>Y</td>
<td>143</td>
</tr>
<tr>
<td>0x5B</td>
<td>Z</td>
<td>105</td>
</tr>
<tr>
<td>0x5C</td>
<td>\</td>
<td>132</td>
</tr>
<tr>
<td>0x5D</td>
<td>]</td>
<td>459</td>
</tr>
<tr>
<td>0x5E</td>
<td>^</td>
<td>946</td>
</tr>
<tr>
<td>0x5F</td>
<td>_</td>
<td>2311</td>
</tr>
<tr>
<td>0x60</td>
<td>`</td>
<td>38</td>
</tr>
<tr>
<td>0x61</td>
<td>a</td>
<td>5342</td>
</tr>
<tr>
<td>0x62</td>
<td>b</td>
<td>1268</td>
</tr>
<tr>
<td>0x63</td>
<td>c</td>
<td>2527</td>
</tr>
<tr>
<td>0x64</td>
<td>d</td>
<td>2370</td>
</tr>
<tr>
<td>0x65</td>
<td>e</td>
<td>8806</td>
</tr>
<tr>
<td>0x66</td>
<td>f</td>
<td>1381</td>
</tr>
<tr>
<td>0x67</td>
<td>g</td>
<td>4943</td>
</tr>
<tr>
<td>0x68</td>
<td>h</td>
<td>3456</td>
</tr>
<tr>
<td>0x69</td>
<td>i</td>
<td>5406</td>
</tr>
<tr>
<td>0x6A</td>
<td>j</td>
<td>1638</td>
</tr>
<tr>
<td>0x6B</td>
<td>k</td>
<td>513</td>
</tr>
<tr>
<td>0x6C</td>
<td>l</td>
<td>3072</td>
</tr>
<tr>
<td>0x6D</td>
<td>m</td>
<td>1988</td>
</tr>
<tr>
<td>0x6E</td>
<td>n</td>
<td>5008</td>
</tr>
<tr>
<td>0x6F</td>
<td>o</td>
<td>5159</td>
</tr>
<tr>
<td>0x70</td>
<td>p</td>
<td>1763</td>
</tr>
<tr>
<td>0x71</td>
<td>q</td>
<td>209</td>
</tr>
<tr>
<td>0x72</td>
<td>r</td>
<td>4376</td>
</tr>
<tr>
<td>0x73</td>
<td>s</td>
<td>4917</td>
</tr>
<tr>
<td>0x74</td>
<td>t</td>
<td>7297</td>
</tr>
<tr>
<td>0x75</td>
<td>u</td>
<td>3625</td>
</tr>
<tr>
<td>0x76</td>
<td>v</td>
<td>677</td>
</tr>
<tr>
<td>0x77</td>
<td>w</td>
<td>1219</td>
</tr>
<tr>
<td>0x78</td>
<td>x</td>
<td>837</td>
</tr>
<tr>
<td>0x79</td>
<td>y</td>
<td>1035</td>
</tr>
<tr>
<td>0x7A</td>
<td>z</td>
<td>361</td>
</tr>
<tr>
<td>0x7B</td>
<td>{</td>
<td>73</td>
</tr>
<tr>
<td>0x7C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7D</td>
<td>}</td>
<td>1188</td>
</tr>
<tr>
<td>0x7E</td>
<td>~</td>
<td>17126</td>
</tr>
<tr>
<td>total</td>
<td></td>
<td>409600</td>
</tr>
</tbody>
</table>

The following characters were inserted to make the above converge: 000004569