Gate-level emulation of an Intel 4004 in 4004 bytes of C

by **Nicholas Carlini** 2025-08-04

The International Obfuscated C Code Contest results were announced this last weekend, and I was thrilled to see that my entry was among the winners. In this article I want to walk through what it is: a feature-complete gate-level emulator for the Intel 4004 that's capable of emulating the original Busicom 141-pf calculator ROM---the application for which the 4004 was originally designed.

To make things exciting, this program doesn't emulate the 4004 the "normal" way. Instead, this program is actually just a logic gate simulator, but where I've designed an entire 4004 as circuit (microcoded for space!) that's embedded in the program and then emulated. And because we're 50 years in the future, this gate-level emulated calculator runs at a few hundred instructions per second which is actually fast enough to be usable, just as long as you're not in a hurry.

This is part two in a two (or maybe three?) part series of posts. Last time, I explained the design of miniHDL, a small python DSL that allows you to build and design circuits. This time, I'm going to use miniHDL to design my 4004 (along with an intel 4003, 4002, and 4001), and then write some scaffolding so that it has the correct "hardware" to run the original program ROM, and then somehow crush a 15,000 byte program and 200,000 byte circuit into the below <4004 bytes. Source code, as always, is on GitHub.

```
#include<stdlib.h>
#include<stdio.h>
#include<fcntl.h>

typedef int _;
#define P 1<<24
#define Z while(
#define W for(i=0;i<K;i++)
#define A *Q++

char*Q = "7JD^A^[!^X^.{Z^\25^VZA^XT$C^W==^EJjUX^EW<|}/*^B7;(bz^Dz).%;y=F4,^ZW}^K^Se%^^^KpV1JU   e^Y/^[Q^Hy=^
C^[hSf^Z^?n}jW^R.^?P!E^R<SO^TNBKuhWy<e:db^YW9'0=/aB(K^^nZ!^Ynh,Ye=+^Cv`^Ko^N!3^O}%^Q ,L.5[M#e^S-^R={^_o^Uz2G
(a!jz?^NQY+QA$4@S6_^RAV^F^QlVO^^yv^V>^UD^_^Vz^_8N^S^[KM[A?i~<,oHaa^W9^DV3^H8c^B^[   zEK[^\^P-^AZ5*K^H> u{^~X
N8}QsGf$nd^U^TN<],5d}hWs;5^TTa^\^N[^_5-VHje6^SDjM^O^X&',K^Gd$w^LtV-^CZ^NJ^Snj^VE={^P?=c*^VT^B^Cb(2I$^KPq9^R&
^^RW^Lu:E11qqH5^SuPx^D^^D^L9Xk>V<^H^X~k^EAe^]@Su}D&F6^H;^G^WX,^D^]^TV]#K^Y,9XS>Y-Dt@-3T^ZSJ,^Q(+7^W$-X^L]]]q
^Nb}^?g^[^Ruzz^Ov@^T8'R[NIEZB66w~^G{A1CC@^W^XM]^W.^Sxq|`'^F/oFlk^_W^G^QDVg{`wx[uEO,^CYQ^H5^N%^RW3,^?_Q^ON/*^
Ks`8 ?]J/wfT_/^Pj]EdLMfgdeJ4xL5^]x^G!5$k)iB.b`*^F@P6B5bj^[^x'+^H~=9p^V`OL1Fu[Su^FuV.^[^E%C7*oIO^Wb^PV^DgIA^
G'!v^P:R)_^D(:qd}B@^BK`$1.^H^H#6PD;cTvjR^_Ov^_'>C^Fu &^L^Y^^^B^B9ck8^]3Y^P4]FJ*Jz,# ^W^D1}Y$ZK^UU~L?y^Q`N^]z
^R%^Gb/^^rD^Gg^L^W~z^H_K^Ri^BAO  ]89@jD$YLfG&8^QzDNy?p1L2vp^?^HZkLO$;^FnxG2r<tf@J[W.MCn$h!B^T1^Y^K^DH6G(Sv^K
4)nB7^X_^?x^TsQ^C({Duth]kt>=k^C%Duq^W[-nVqG.&NEXWs^U:^?Xj#^Y.t^AJ2g^[!Q^O>X(jK^B{cTC(Vk^X~^PYS?U^Y;5^Y^L^TO&
^VN*e^CU^NaY^UvDYdJtV3q^V&NGn^Z^P^Q^Bt>G6^N{BR%F^Q^UBYLeP^Zx^_q^AJ(&L%`T^K,f^^KQ^O^NV^Z^?6.T;})1S8QCg}g4k]/^
Wc:bb/>|x^YO^E^^G$^]Pn=^EK^D4joC^A^'N1HH($,eow^[1-^AINyY^ZYthdT=T$o+o)n^[cyCO^F}:^F^LvdZ^H   7s7^Q^Bd`5^Ch.t
LtvIow|VL)^?~k# OrdL^N7 ^[}z|w^HZ@   Q3^D^[pcQ^K)^BU^G`Gw{^A^RX^_;^D-82pL^Z^4L^]^USNBW^O^B^V^]Ua^A(X^S2^F&^C
^Z^\e?Q^L@6c/KzQ^Q9x^NS|k@gP^T^H$=Bgn]^R,^\^TglLOW&^ETa{Ck^ZI,!&tP!Tz^KMzqM^C^O^Po^_TGHe^Hs_ v^T^O^HUGWt^Ef^
T&V|o`^H[GR^Y^\Hm$^?^_X^Q5GI^[gE^Pme^A5:^H1!^Q^QB&?5e@^A&WB^D-J^[r^S%[b:iv=^Fyr  E;^O`^WsVol`P^^Y)^\g^^~^?)l
=*$  e8^D^?l6^K>j1]8%GQl^A^K+MU<M b,qb^Y#:^D=^_^G&fC5nCj^^tpk/8b^D&n!83CvF`(<uK^^^BH=kaf)^\(c*[^D-;zvb_`R^\^
ZT^DT^ZXVF}^B`id~^E.b,93n^Sd_^HSTrl^^.eO^T*^\;^R^Hcx $qu^OZq^_88sP^\ky.*@^?q.^Pf9`W^K^OK6^NA^N\?#Yi^Hj^O#^D`
^Kyp(L^Q(jz^Du^U]kVCWIOM~%}&QhCK^Q^K^[^WS/uY5B^E9[s*cptY?-]9U^Cy^?O^T}}j/-3~oMn^CA{(p^Bu^?=^W`'%w1Jw?o^Udzb^
_:Ok?@7>nN^HO9s=A,^R^Y$^R^[C^L$-'9(2^Wb'^F^B^SST^Y^AEIx(yq&Iyf^G D^Cg^O9@:a^^Tn/+(pe~^G?^O^P^_3Ahh<*jX^^HB7^
K^B4Wt^K7R>*^U^K^XzAl^B'eV^FOO6X=PgrDIE!),0^_^Av%y^U^C7^^9/8Vx)NAC^N^S(5O^]^X7^]^TW^Zk$i}^G?p?C^l-^G^CUQ@C6^
O3/^TT!^W^P4zJ^PX?H^?^Y^L^Uf1n^FoL:^A^A^A";

_ i,q[P],B[P],R[P],l[9],*z,K,I,m[P],p,D=123,S,n,g,*t[P],k[P],c,e,E,*u,y,F; char T[P],*v; long C=1; _*w(_*Q){
_ N,*f=Q,*o,p; Z(N=A)<9){ _ a=(z-B)/4,b=0; o=Q-1; if(N<6){ *z=N; K=1+(N>1)+(N>4); W{ I=A; N=z[i+1]=I<2e5?a-
I:m[I=I-2e5]; if(N)t[N][k[N]++]=a; } z+=4; } else if(N^7){ I=A; K=A; W m[i]=a+=A; K=P; W R[i]=q[i]; w(R+l
[I]); } else{ a=0; I=--A; p=K=A; if(I){ W{ b+=Q[p]; f[a+=A]+=b; } Q=f; } else Q+=p*2; } f=o; } return Q; } _
s(_ O){ if(C<D){ C*=D; p*=D; K=A; p+=(K-1-(K>10)-(K>13)-(K>34)-(K>92)); } _*x=B+O*2; K=C*-~*x/(*x+x[1]+2); I
=p>=K; p-=K*I; C=I?C-K:K; x[I]++; return I; } _ h(_ K,_ O){ _ g=1<<K,f=1; O*=99; Z!s(++K+O)); K--; W f=f*2|s
(O); return f-g; } void d(_ H,_ O){ for(p=0; p<H; p++){ K=1<<O; W B[g++*4+1]=R[i]>>p&1; } } _ main(_ x,char*
*O){ if(x<2)exit(42); fcntl(0,4,O_NONBLOCK); _*Q=q,K,i,j=h(9,0); Z j--){ if(s(1)){ z=Q-h(4,2)-1; K=h(1,3)+1;
W A=*z++; } else A=(1-2*s(8))*h(3,9); } Q=q; K=S=A; v=T+A; n=A; W t[i]=malloc(7e4); K=A; W l[i]=A; I=A; z=B;
Q=w(q+I); g=A; FILE*J=fopen(O[1],"r"); K=4096; W R[i]=fgetc(J); d(8,12); g=A; K=16; W{ j=256; Z j--)R[j]=Q[j
*16+i]; d(6,8); } u=Q+4096; unsigned long*f=calloc(S,16),G; Z 1){ _ r,M=0,g,*Q; Z M<S){ K=3; Z K--){ I=1<<6*
K; g=M/I; G=f[y+S/64*K+g/64]>>g%64; Z G&1){ M=M-M%I+I; G/=2; K=0; } } Q=B+M*4; r=A; i=T[A]; K=T[A]; I=T[A];
p=T[M]; T[M]=r<2?i^r:r<4?r^3?i|K:i&K:r^5?i^K:i?K:I; if(p^T[M]){ I=k[M]; Z I--){ g=t[M][I]; K=3; W f[(y^S*(g<
=M))+S/64*i+(g>>6*~-i)]&=~(1L<<(g>>6*i)%64); } } M++; } K=S/16; W f[y+i]=-1L; y^=S; { char*Q=v; if(A){ F|=A;
K=A; I=!A; E+=I&!K; if(K&I&A){ K=18; putchar(32+F*13); W putchar(u[i*18+*(_*)(T+n+((i+3)%18-(i>15))*21)%1
8]); F=0; puts(""); } } if(E-c>128){ c=E; e=!e; I=e?0:u[325+getchar()]; K=8; W A=I>>i&1; } } } }
```

A very brief primer on the Busicom 141-pf and Intel 4004

The Intel 4004 is an exceptionally simple CPU by modern standards. It is a 4-bit processor, with 16 general purpose registers. There are a total of just 45 instructions, and many of them are nearly identical to others, so in practice there are more like only 20 unique instructions. It's accumulator-based, which means that almost all instructions either operate on a special 4-bit accumulator, or load data into the accumulator or transfer it out. And the program counter can address up to 12 bits of ROM. All in, this makes it one of the simpler instruction sets out in the world, which makes sense given its age.

You can attach three other chips to the 4004 CPU that provide:

- ROM, via an Intel 4001: This stores the program code to be executed, or any other read-only data.
- RAM, via an Intel 4002: This gives additional storage of more than just the 4*16 bits from the registers.
- A shift register, via an Intel 4003, to temporarily hold a queue of data which can be attached to physical hardware to help manage I/O.

The 4004 was originally built for use in the Busicom 141-pf calculator. While initially the 4004 was going to be application-specific just to make the calculator possible, some forward looking engineers at Intel decided to build a general-purpose chip that could perform arbitrary calculations, and then just program the chip to be a calculator. And thus was born the general-purpose microprocessor.

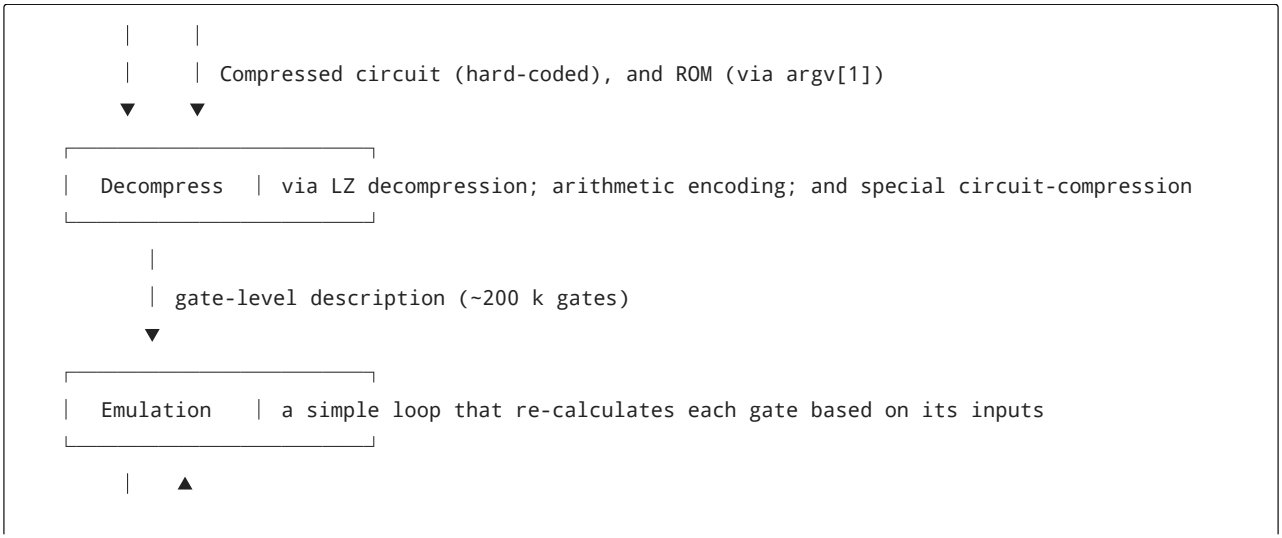
The Busicom calculator itself is a fairly simple device. It has an 8x4 keypad the operator can use to enter numbers or perform numerical calculations, and an output tape on which the result of the calculation is printed. The keypad is simple and looks familiar to any calculator today. But the output works through an esoteric (by today's standards) mechanism. A spinning print-drum rotates continuously above a paper tape; when the calculator wishes to print a number, it strikes hammers from one of the 18 columns at the exact moment the print drum has the desired number above the paper tape, and then advances the tape to be visible to the operator.

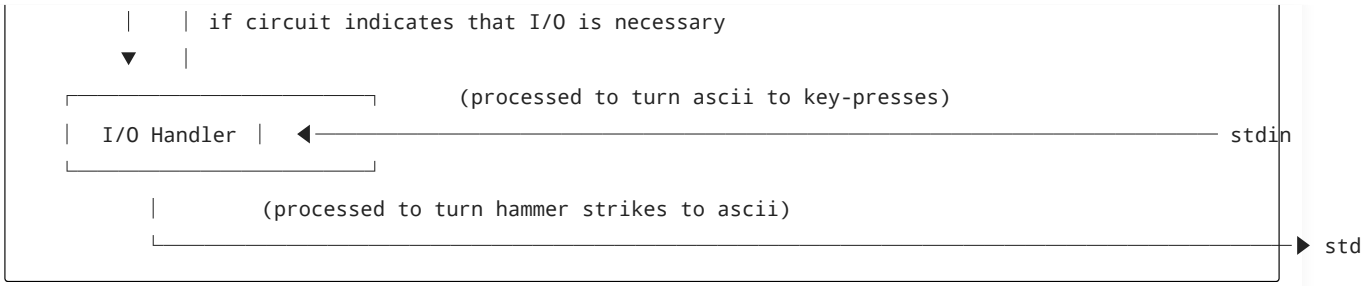
Overall structure of the program

The program is split, roughly, into three distinct components:

1. First, it takes a compressed string and decompresses it to recover a circuit that, when executed, emulates the Intel 4004 (and a few Intel 4001s, 4002s, and 4003s). As part of this process, my code inserts the provided program ROM from argv[1] into the circuit.
2. Then, the program actually emulates the decompressed circuit by computing the output of each gate in a loop, effectively running the CPU and associated RAM/ROM chips. (With a few associated tricks to make things run fast.)
3. Finally, the program simulates the original calculator hardware and handles converting the input text from ascii text passed via stdin to the original hardware keyboard presses, and also converting the shift register causing hammers to strike paper tape back to ascii outputs.

The remainder of this section documents the implementation of each of these three components. Here's a small visual to show what's going on.





Decompressing the compressed circuit representation (part 1)

The circuit is compressed twice. At the outer level is an LZ-compressor with arithmetic encoding, and then on the inner level is a special-purpose logic-gate compressor that allows creating small sub-circuits that can be called.

The first part of the decompression is the more straightforward of the two. The data is compressed with a special-purpose LZ-style compressor that I wrote which, instead of being bit- or byte-level, operates on unbounded integers as the data stream. The decompressor itself fits in just 107 bytes (plus some utilities I'll define in a bit). Let's walk through how it works.

Defining a new LZ77 format

Okay so let's get started with the basic setup of my compreressor. Instead of being a bit-level compress (like LZMA) or byte-level compressor (like GZip), this is a variably-sized int level compressor. That is, it directly processes a sequence of arbitrary-sized integers.

My compression format starts with an integer representing the number of total "operations" in the compressed file. The format itself is just a sequnce of operations. Each operation is one of the following:

1. LITERAL: the next integer being stored is directly the integer that should be written to the output. A single bit indicates if the value is positive or negative.
2. MATCH: the next two integers represent how many bytes back to go, and then how many bytes of prior output to next copy to the output.

Let's walk through an example. If compressed intput stream was the following `[0, 0, 8, 0, 1, 12, 1, 2, 2]` Then this would represent the following:

- 0: a boolean that indicates the first operation is a literal
  - 0: the literal is a positive number
  - 8: we're storing the number 8
- 0: The second operation is also a literal
  - 1: the literal is a negative number
  - 12: we're storing the number -12
- 1: Now we're going to copy some bytes
  - 2: Start by backing up two bytes (to the initial 8)
  - 4: And then copy two bytes. So we repeat 8 and -12

This finally decodes to the sequence `[8, -12, 8, -12]`

Now you may say "But Nicholas that's longer than just storing the original sequence". And yes, for this toy case it is, but for longer inputs that have a lot of repeats it makes a big difference.

Building an LZ77 decompressor

The entire reason I set up this file format this way is because of how easy it is to decompress. The decompressor is, almost literally, just the following:

```
int output_circuit[big];
void lz() {
```

```
  int* ptr = output_circuit;
  int num_bytes = get_integer();

  while (num_bytes--) {
    int is_match_op = get_bit();
    if (is_match_op) {
      int* ref = ptr - get_integer();

      int copy_len = get_integer() + 1;
      for (int i = 0; i < copy_len; i++) {
        *ptr++ = *ref++;
      }
    } else {
      *ptr++ = (1-2*get_bit()) * get_integer();
    }
  }
}
```

where the command get_integer turns a sequence of bits into an integer with a Golomb encoding:

```
int get_integer(int length, int ctx) {
  int subtract_it = 1&lt;&lt;length;
  int result_ans = 1;
  ctx*=99;
  while (!get_bit(++length+ctx));
  for (int i = 0; i < length-1; i++) {
    result_ans = result_ans*2 | get_bit(ctx);
  }
  return result_ans - subtract_it;
}
```

And that's literally all there is to it. (We'll return, in just a moment, to how we actually implement get_bit.)

Building an LZ77 compressor

Okay well now we have our own LZ77 *de*compressor, but that isn't much good without a compressor to go along with it. How are we going to do that?

Turns out LZ77 compression is actually a rather simple algorithm to implement through dynamic programming. We'll process the data one token at a time, at every moment in time remembering the best way to compress the data up to and including this token. And then, we'll try to extend this match by one more token either by just adding a literal byte, or continuing a match object one further.

So let's suppose you've compressed the string S[:K] and then want to add one more token S[K]. There are a just two operations you could take:

1. You could just use the compressed output up to this point and tack this token on the end, i.e., compress(S[:K]) + [('Literal', S[K])]
2. Alternatively, if it turns out the last K tokens (denoted S[K:]) are already contained somewhere else in the string S[:K], we might want to output compress(S[:K]) + [('Match', where_it_is, K)].

Now if you know anything about algorihtmic complexity you might say: But Nicholas, isn't this an O(N^3) compression algorithm? Why yes. Yes it is. But do you know the thing about asymptotic analysis? As long as it runs fast enough for your purpose it really doesn't matter. And given that I'm only ever compressing a sequence that's roughly 10000 ints long, a cubic algorithm is actually entirely reasonable.

Arithmetic Coding

Now let's turn to the arithmetic encoding algorithm. Actually, this algorithm just lifts directly from Fabrice Bellard's wonderful submission to IOCCC 2018, as un-obfuscated by Andrew Kensler, with a few tweaks of my own to reduce its length, and to cut out some features I don't need. Briefly, arithmetic encoding keeps a history of how likely each bit was to be zero or one, and then uses this prior to inform the decoding of the output stream. This version of arithmetic encoding uses a base-1968 encoding abusing the fact that IOCCC counts whitespace special, and IOCCC doesn't allow use of high-bit codepoints.

```
int get_bit(int ctx) {
  if (range < radix) {
    range *= radix;
    fraction *= radix;

    tmp = *bitstream++;
```

```
        fraction += (tmp - 1 - ( tmp > 10 ) - ( tmp > 13 ) - ( tmp > 34 ) - ( tmp > 92 )) << 4;
        tmp = *bitstream++;
        fraction += tmp < 33 ? ( tmp ^ 8 ) * 2 % 5 : ( tmp ^ 6 ) % 3 * 4 + ( *bitstream++ ^ 8 ) * 2 % 5 + 4;
      }
      int *counts = history_buffer + ctx * 2;
      int split = range * ~-*counts / (*counts + counts[ 1 ] + 2);
      int the_bit = fraction >= split;
      fraction -= split*the_bit;
      range = the_bit ? range-split : split;

      counts[the_bit]++;
      return the_bit;
    }
```

Decompressing the compressed circuit representation (part 2)

Once the first level of decompression is complete, the circuit now appears like a sequence of integers that represent the logic gates. It's now time to actually expand this out to the logic gates themselves. Specifically, each logic gate is given a unique integer {DIRECT_WIRE, NOT, OR, AND, XOR, MUX} and then this operation is followed by the input gates. To improve compressibility, the input gates are expressed in a relative indexing.

So, as an example of this, consider the following circuit:

```
v0: NOT v4
v1: OR v0, v0
v2: OR v1, v1
v3: OR v2, v2
v4: OR v3, v3
```

This circuit acts like a clock of sorts, with the four entries v0..v3 cycling through between 0s to 1s. This circuit would be represented with the following sequence:

Instead of each gate referencing other gates by their absolute position, the program has each gate reference gates based on how many gates backwards (or forwards) we should go. So the above circuit becomes the following:

```
0: NOT +4
1: OR -1, -1
2: OR -1, -1
3: OR -1, -1
4: OR -1, -1
```

By storing delta-encoded, because most gates reference other gates that are at least fairly nearby, we usually only have to store a somewhat smaller constant, reducing the size of the circuit by roughly a factor of two. And so instead of requiring 18 bits for each reference, it only takes 7.7 bits. But the much more important win is that now this allows for more advanced forms of compression.

The major gain is that LZ-compression now can repeat the `OR -1, -1` many times for free. A more minor gain is a special-purpose repeat handler built in to the gate-uncompression. Specifically, the program implements a special instruction called REPEAT that means "go back one entry and repeat that one again, possibly making a constant change to its value". This second part is what makes repeat valuable even though we already have LZ compression.

As an example, suppose we had a circuit that looked like this

```
v0: v0 & v10
v1: v1 & v10
v2: v2 & v10
...
v9: v9 & v10
```

Then we could express this as

```
v0: v0 & v10
REPEAT the above instruction 9 times, incrementing argument 2 by 1
```

Note we're incrementing argument 2 because after delta-encoding the access to the value of gate 10 is what's changing.

The implementation of this is relatively straightforward, and relies on modifying the instruction stream as its being processed. Specifically, the decode function is defined as below. Initially we read off the number of times to run the REPEAT loop, then we have a somewhat nasty compressed loop that

```
int* decode(int* data_stream) {
  int* prev_start;
  // ...
  if (*data_stream++ == 6) // repeat
    // A repeat is a loop that runs the prior instruction a variable number of times,
    // and also allows for updating the arguments to the prior instruction by a
    // constant scalar on each step.

    // To implement this, we use some self-modifying code:
    // 1. Decrement the argument that says how many repeats to do
    // 2. Adjust the other arguments from the prior instruction as necessary
    // 3. Go back 2 instructions if the counter is greater than zero.
    //    (on the next iteration of the loop we'll modify the arguments again).
    int number_of_times_to_duplicate = *data_stream++;
    num_change = tmp = *data_stream++;
    if (number_of_times_to_duplicate) {
      for (int i = 0; i < tmp; i++) {
        int increment_constant = data_stream[*data_stream++];
        prev_start[*data_stream++] += increment_constant;
      }
      data_stream = prev_start;
    } else {
      data_stream += num_change * 2;
    }
  }
}
```

Finally, there's one more special code that acts like a function call, and says "go copy that small sub-circuit defined somewhere else to here. This behaves similarly to repeat. The program defines five of these sub-circuits that are used throughout the program; the most important of these being a full adder, but also a single register (so it can be duplicated 256 times) and various helper circuits.

The implementation of the function-call decoding is again fairly simple: the program pulls off the function ID to be called, the number of arguments it takes, and then sets a global `arguments` array with these arguments. Then, we just jump to the relevant function and start decoding from there. (With a tiny twist: because functions can have REPEAT ops, and REPEAT uses in-place self-modifying code to work, we have to actually make a complete copy of the function and then execute it so that we can call it multiple times without breaking REPEAT.)

```
int* decode(int* data_stream) {
  ...
  if (*data_stream++ == 7) { // function call
    // A function call goes and runs a small number of instructions and returns back.
    // Nested functions aren't possible.
    // So to call a function, we just set the global arguments buffer with the correct arguments,
    // write out the data of the function, and then call cluster() on that location.

    int which_function = *data_stream++;
    int num_args = *data_stream++;
    for (int i = 0; i < num_args; i++) {
      arguments[i] = *data_stream++;
    }

    // Copy to buf the current compressed ops
    // we'd like to just do a reference,
    // but if the called function uses loops then they modify the code in-place
    // so we have to copy before the function call
    for (int i = 0; i < BIG; i++) {
      buf[i] = prog_src[i];
    }
    decode(buf+functions[which_function]);
  }
}
```

After compression, the entire circuit of 200,000 gates is crammed down to just over 1000 bytes. How's that for an efficient compression algorithm?

Decompressing the compressed circuit representation (part 3)

The final part of decompressing the compressed circuit is to load the program ROM from argv[1] and insert it into the circuit. This is relatively simple; the circuit contains a big MUX tree that encodes placeholders for the program ROM, and so all the code does is directly edit the placeholder variables that will be used to load the ROM with the following simple function.

```
// buf contains the program text that is to be inserted into the circuit
// gates is the array that has the actual logic gates
void fix_const_help(int depth, int writesize) {
  for (i = 0; i < depth; i++) {
    for (int j = 0; j < 1&lt;&lt;writesize; j++) {
      // byte offset*4 is necessary because gates can have up to 3 arguments
      // and so are represented as [operation], [arg1], [arg2], [arg3]
      // this ensures we are writing into the proper location each time
      gates[byte_offset++*4+1] = buf[j]>>i&1;
    }
  }
}
```

(In an identical manner, this is also how the microcode is written in to the circuit so that each instruction can be implemented with the extra simple circuit.)

Emulating the instructions

The second component of the program is an emulator to actually run each of the (now decompressed) AND/OR/NOT/XOR gates that are used as primitive building blocks to implement the circuit. This is by far the simplest portion of the circuit, and is just a compressed switch statement that selects the command we're running and does the correct corresponding action. Again, the weird compression is necessary to save space.

```
command = insns[ctr*4];

// Load the potential arguments for this instruction.
arg1 = gate_value[insns[ctr*4+1]];
arg2 = gate_value[insns[ctr*4+2]];
arg3 = gate_value[insns[ctr*4+3]];

// We have six possible commands REF, NOT, OR, AND, XOR, MUX
// Here we're going to evaluate with a series of ternary ops.
gate_value[ctr] = command<2 ?
  arg1^command // either 0 (DIRECT) or 1 (NOT)
  :
  // either 2 (OR), 3 (AND), 4 (XOR), or 5 (MUX)
  command<4 ?
    command^3 ?
      arg1|arg2 :
      arg1&arg2
    :
    command^5 ?
      arg1^arg2 :
      arg1?arg2:arg3;
```

This loops basically forever until the user gets bored and kills the process.

Handling input and output

Finally, we need to be able to actually hook this circuit to the outside world. Inside the circuit I set aside one "wire" whose only purpose is to signal to the outside world that it wants to take some action. Think of this like a trap in an operating system, but at the hardware level: whenever the circuit wants to take some action in the outside world, it pulls one of the "wires" high for a single clock cycle and the C program watches for this to handle the response. The entirety of this trap handler is implemented as below, explained in more detail as follows.

```
if (gate_value[trap_id]) {

  int is_write = gate_value[trap_id+1];
  int rom_or_ram = !gate_value[trap_id+2];

  jcn_counter += rom_or_ram & !is_write;
  if (is_write & rom_or_ram & gate_value[trap_id+3]) {
    for (int i = 0; i < 18; i++) {
      int idx = ((i+3)%18-(i>15));
```

```
        putchar(charset[i*18+*(int*)(gate_value+print_uid+idx*21)%18]);
      }
      puts("");
    }

  // Every 128 JCN instructions we should send the next keypress.
  // 128 is hard-coded in the circuit. Don't change.
  if (jcn_counter - reset_key_matrix > 128) {
    reset_key_matrix = jcn_counter;
    read_or_reset = !read_or_reset;
    int read_char = read_or_reset ? 0 : charset[325 + getchar()];
    for (int i = 0; i < 8; i++) {
      // this is where the circuit expects the input data to be written
      gate_value[trap_id+8+i] = read_char>>i&1;
    }
  }
```

A trap can be for one of two reasons:

- First, it be because the program wants to read some data from the outside world. When connected to the Busicom calculator, the 4004 will either want to read any key that might be pressed by the user on the input, or will want to read a single bit to detect when the spinning printer drum has advanced to the next location.
- And second, it can be because the program wants to write some data to make a change to the outside world. The 4004 uses writes to strike the hammers against the print drum, advance the paper tape (a "new line"), and to energize one of the 10 possible lines of the keyboard to enable that corresponding line to be read.

The program contains two compressed lookup tables to enable this functionality. First, it remaps the ASCII characters that the C program reads with getchar() to (row, column) locations of the corresponding key on the physical keyboard. For example, the number 9, 6, and 3 are at location (4,0), (4,1) and (4,2). This lookup table encodes each ASCII character into a six bit number, which itself is then compressed along with the circuit using the same LZ compression.

The program also contains a lookup table to map each of the hammer-strike outputs to ASCII characters that can be putchar()'d. This table, if implemented in the naive manner, would be fairly straightforward, because each print drum just repeats the numbers 0 to 9 in order (two columns of the drum contain special characters). To enable a much more efficient lookup, the data is stored permuted according to the following permutation {0, 1, 6, 7, 2, 3, 0, 0, 12, 13, 8, 9, 14, 15, 10, 11, 4, 5}.

The purpose of permuting the data in this way (and storing some data duplicated! notice that 0 is stored several times) is that now we can read it out of memory much more concisely, with the following line:

```
  for (int idx = 0; idx < 18; idx++) {
    putchar(charset[i*18+*(int*)(circuit_memory+offset+idx*21)%18]);
  }
```

Specifically, for each of the 18 drum locations, we print the character at that drum. `charset[i*18]` offsets us to the correct drum. The remaining constant is where the magic happens: circuit_memory is normally an array of uint8; by casting to (int*) we can load four values at a time, which as it happens is exactly the number of index bits we need. What we get is then an integer like 0x01000101. And it turns out that if you multiply this number by 21 and then take the result mod 18, you end up exactly being able to index into that permuted ASCII table.

(This is just one of the many pieces of the program that has to be implemented in convoluted ways to cut down on program length. I'll try to omit most of the rest for brevity.)

Now there's one final question: when to make input available. On the original calculator, the print drum was actually spinning and so that set the rate of inputs that were pushed to the circuit. But we have no actual spinning drum and so we need to decide the rate that input will be passed. Instead of making running based on wall-clock time, the C program does something somewhat confusing, and tracks the number of conditional jump operations that have been issued. Every 128 conditional jumps (JCN's) that branch on the 4004's TEST pin, it makes one more character of STDIN available to the program. This is the fastest rate possible at which the ROM can handle input without dropping characters.

Making it go fast(er)

The CPU itself has over 200,000 gates (I'll describe why in detail later---but it's to save space when compressed). Now given that we have to loop over every gate 16 times for one tick of the clock, and each instruction takes 16 inner clocks of the circuit, we're going to run into a problem. If we would like it to run at anything faster than one instruction per second, we'll have to make some changes while still keeping things in our space constraint.

As has been said many times before, you can't make computers perform the same amount of work faster. You can only make them do less.

So the program does just that. Instead of re-calculating every gate on every iteration of the loop, it keeps track of which inputs have changed last time, and then only re-calculates the gates whose input changed during the prior round. This requires that when we first load the circuit off of disk that we build a table in memory for which gates depend on which others. In order to minimize space here, this is a dense table with shape 200,000 x 70,000 (it's 70,000 wide because there's exactly one gate---the clock---that 64k other gates depend on).

Once we've built the table the logic gets somewhat complicated. The program maintains a 3-deep hierarchical bitmask to indicate how far can be skipped ahead. At each level is an array of 64-bit words where each bit corresponds to whether or not there is any gate that needs to be updated at the next level "down" in the tree. (So, for example, at the lowest level if a bit is 0 anywhere then that gate doesn't need to be updated; at the second level if a bit is 0 then all of the 64 bits below it don't need to be updated, and at the highest level the 64*64 bits below don't need to be updated.)

The logic to actually set the bits that need to be updated is as follows. Initially needs_update is set to 0xFFFFFFFFFFFFFFFF and then bits are cleared when gates they depend on are changed

```
int num_depends = depends_ctr[ctr];
while (num_depends--) {
  int gate = depends[ctr][num_depends];
```

This compressed code isn't that hard to reason about. First we select the right address by taking the gate index and dividing by 2^(6*(depth-1)). Then, we select the bit address within the word by looking at the remaining bits in the gate. We do this for all three depths, and make sure to only ever clear bits, so the needs_update gets sparser and sparser as the circuit goes forward.

And the corresponding logic to check if we need to process the current data is implemented here. Again this works by just masking out to check if all of the bits in needs_update are zero, and skips if any are set to 1. If a bit at the lowest level of the tree is 1 then it skips one gate forward; if a bit at the medium level is 1 then it skips 64 gates forward, and at the highest level 64*64 gates forward. This function looks simple, but there are several cases that are all magically handled here in very subtle ways. Try to figure out why it's exactly right!

```
    needs_update[total_gate_count/64*depth + (gate>>6*--depth)] &= ~(1L<<(gate>>6*depth)%64);
  }
}

for (int depth = 3; depth-->0; ) {
  int divide = 1&lt;&lt;6*depth;
  int gate = ctr/divide;

  unsigned long tmpll = needs_update[total_gate_count/64*depth + gate/64] >> gate%64;
  while (tmpll&1) {
    ctr = ctr - ctr%divide + divide;
    tmpll/=2;
    gate = 0; // break out of the above loop
  }
}
```

This ends up giving about a 100x-500x performance boost on average, and allows the emulated 4004 to run at a few hundred instructions per second, which isn't all that bad! (But even at 100x speed it still takes roughly a minute for the calculator to perform the addition of two numbers. Slow, but actually only 60x slower than the original calculator that took one second to do the same addition.)

And thus concludes how the actual C program itself works. There are a lot more tricks going on, but this is hopefully enough detail to understand most of the important pieces.

How the circuit works

The primary CPU construction is exceptionally simple, in order to keep it as small as possible.

The CPU is a load-store stack(like) RISC machine that encodes each of the Intel 4004 instructions as a sequence of at most 16 micro-ops. Therefore, each clock cycle of the circuit performs 1/16th of a 4004 instruction. The program maintains a 4-bit counter to keep track the current micro-op that's executing, and a 96-bit shift register that holds each of the 6-bit micro-ops that correspond to the current instruction.

The program maintains 65536 (2^16) bits of ROM split equally into two 32768-bit banks. The first of these stores the program code: up to 4096 (2^12) 8-bit Intel 4004 instructions. The second bank stores a lookup table mapping each of the 256 possible 8-bit instructions to a 96-bit micro-op vector encoding the 16 6-bit micro-ops.

The ALU itself is implemented as a 3-deep queue. On every clock cycle, each micro-op pushes one value to the top of the queue, and the last value falls off the back. (Think of this like a stack machine, but a queue is simpler to wire than a stack.) Each micro-op can index into exactly one register, and either load its current value to the top of the queue or replace the current register with the value with whatever is on the top of the queue. Alternatively, each micro-op can perform some computation using the values in the queue.

The register file for the CPU holds 256 twelve-bit ints. This might seem somewhat counterintuitive because the 4004 is a 4-bit machine, but consistent patterns are more easily compressed with LZ compression, and 12-bit registers are required for the program counter already, so it's cheaper to make everything a 12-bit register. The first 16 of these registers hold the 16 four-bit general purpose registers for the Intel 4004 (the upper 8 bits are ignored). The next 16 registers hold a few special-purpose registers:

- The 4-bit accumulator
- The 1-bit carry flag (when an ADD overflows)
- The 12-bit program program counter
- Three 12-bit program counter stack registers for call/return support
- A 8-bit RAM pointer indicating which RAM address we want to update
- A 12-bit temporary register for holding some intermediate state

The remaining 224 registers hold RAM values of the Intel 4002, discussed later.

To keep the micro-ops space-efficient, they are 6-bits each. Micro-ops that begin with a 1 index into 32 different possible queue-operation. The most commonly used micro-ops include operations to

- ADD the top two values on the queue (with a carry of 0, 1, or from the carry flag)
- INCREMENT the top value of the queue
- DOUBLE the top value of the queue
- NEGATE the top value of the queue
- MUX to conditionally keep either the top or second-to-top value.
- BIT_SHIFT_RIGHT_4 the top value on the queue
- BIT_SHIFT_LEFT_4 the top value on the queue

Micro-ops that begin with a 01 read from a register and load it into the register file, and micro-ops that start with 00 take the top value of the queue and write it to a register. Because this leaves just 4-bits to index into a register file with 256 entries, these 4-bits actually index into a lookup table that decide how the register should be located. Some of these ops, for example, allow for

- Direct indexing to 8 special purpose registers ACC/CARRY/TMP/RAM_ADDR/PC/PC1/PC2/PC3
- Direct indexing into the general-purpose register specified in the low-4 bits of the 4004 instruction
- Direct into the next register after the one specified in the low-4 bits of the 4004 instruction
- Indirect indexing that loads whatever is pointed to by the special RAM_ADDR instruction

The rest of the ALU then just consists of a few multiplexers to decide how to route the data around either loading from the RAM to storing the queue, loading from the queue and storing to the queue, or loading from the queue and storing to RAM.

Defining the circuit

The main circuit is built using miniHDL, the Python DSL I built in the last post. For the remainder of this post, you don't need to have read the last post completely, but some familiarity with circuit design would be valuable.

As I mentioned last time, here's the design of a full adder:

```
@make_circuit("full adder")
def full_add(in1, in2, in3):
    low1 = in1^in2
    low2 = low1^in3
    high = (in1 & in2) | (low1 & in3)
    return low2, high

@make_circuit("add_16")
def add(in1, in2, carry):
    out = []
    for a,b in zip(in1.bits, in2.bits):
        c, carry = full_add(a, b, carry)
        out.append(c)
    return Bits(out)
```

When compressed this circuit actually is compressed to just the following few bytes:

```
function_1:
4, 200000, 200001  // First XOR (idxs >200000 are argument indexs)
4, 1, 200002       // Second XOR with the prior result and arg2
3, 200000, 200001  // Now compute the AND of the first two args
3, 3, 200002       // And also AND with the third argument
2, 2, 1            // Finally take the OR and return this.

main:

// Initialize variables here

6, 0, 3, -10, 4, 5    // FUNCTION CALL to the full adder
7, 4, 2, 3, 2, -4, 8  // REPEAT the call to the add routine
```

Notice the value of the function call and REPEAT together allow for much more efficient compression than would otherwise be possible.

Register File

The register file is probably the most complicated component of the CPU. Unlike last time where I designed a register file just to be simple, here I'm going to redesign it to be small when compressed. To begin, the circuit implement a function `to_onehot` that takes an 8-bit integer and outputs a 256-bit output that's zero everywhere except the index of the 8-bit integer. (So, for example `to_onehot(00000010)` will return `[0 0 1 0 0 ... 0 0 0]`.

```
def to_onehot(reg):
    out = []
    counter = Bits([const(0) for _ in range(len(reg.value))])

    # 16 NOPs here for ... reasons.
    [const(0) for _ in range(16)]

    def eq_by_inc(counter):
        return inc(counter), ~any_set(counter^reg)

    for i in range(256):
        counter, to_return = eq_by_inc(counter)
        out.append(to_return)
```

```
    return Bits(out)
```

Notice again that this program is aiming for the minimum description length not the minimum number of gates. Putting 256 full adders here, each of which could be an incrementer, which aren't even necessary because you could just do this with a few MUXes, may seem wasteful. But the reason I'm doing it this way is that I already need an adder circuit, so re-using it is basically free, and so why build an incrementer when you can just re-use the adder for free. And why build any new multiplexing stuff when you can just repeat 256 adders mostly for free. LZ compression really does give us so many opportunities for optimization.

The next step is to build a single register

```
def one_reg(clock, enable_write, do_write, data, output1):
    out = dff(data, clock & enable_write & do_write, 12)
    return out
```

And finally, we're done! We can write our register file:

```
def regfile(clock, which_reg, data, enable_reg_write):
    mem = []
    mem_set = []
    reg_onehot = to_onehot(which_reg).value

    for i in range(256):
        out = one_reg(clock, enable_reg_write, reg_onehot[i], data)
        mem.append(out)

    return mem
```

The ALU

The final component of the circuit is the ALU. For this we need the actual queue where the operations will take place, defined like this:

```
queue_in = Bits([Signal() for _ in range(12)])

queue = [queue_in]

spacer = Bits([Signal() for _ in range(12)])

for _ in range(3):
    clock_inner = clock_inner.clone()
    queue.append(dff(queue[-1], clock_inner, 12))
```

The spacer here is again an example of how wasting space actually saves space. Because each iteration of the loop, when unrolled, has a d-flip-flop that refers to gates that were defined exactly 24 gates above, by placing a 12-gate "nop" we can make the circuit compress better.

Finally, we have to define the actual lookup table that the ALU uses. This is the least-clean part of the circuit, and is truly awful to look at. It just takes each of the different possible results we might want, and extracts out the correct one. See, I'll show you:

```
output_table = [
    zero,
    Bits([alu_op.value[0]^x for x in queue_a]),

    add_result, add_result,

    Bits(queue_b.value[4:8] + [const(0)]*8).clone(),
    mux(u_op.value[0], daa, Bits(kbp(*queue_a.value[:4])+[const(0)]*8).clone()),

    mux_result, mux_result,

    Bits(queue_a.value[:8] + addr.value[8:]).clone(),
    from_rom,

    full_jcn, full_jcn,

    queue_a_plus,
    queue_a_plus,

    add_two, add_two
]
```

```
    alu_output = Bits([do_mux16(Bits(u_op.value[1:5]),
                                *[x.value[i] for x in output_table]) for i in range(12)])
```

The Input/Output circuitry

The last piece of the circuit is the part that handles input and output. This is relatively simple, all things considered. It mainly consists of a two 4003 shift registers to handle the printing, and a 4003 shifter to handle the keyboard reading.

But first, it's necessary to talk about the part of the circuitry that enables I/O passing back and forth data between the C program and the circuit. Here's the implementation of the special operations as defined in the circuit:

```
    # First, we need to figure out if we want to TRAP.
    # TRAP is a special u-op that can be identified with this bitmask.
    trap_possible = (u_op.value[5] & u_op.value[4] & ~u_op.value[3] & u_op.value[1])
    # Next, we need to figure out the reason. Is it trap because of RAM/ROM read/write?
    trap_ramrom = u_op.value[2] | u_op.value[0]
    # Alternatively, maybe it's a trap because we're reading the TEST pin via JCN.
    trap_jcn_flag_set = insn_stream.value[0]

    # The C program will now directly inspect these values.
    # A trap acutally happens if we have the right u-op, and one of these conditions
    trap = trap_possible & (trap_ramrom | trap_jcn_flag_set)
    # Is it a write? or a read?
    trap_is_write = u_op.value[2].clone()
    # Is it the ram? or the rom?
    trap_rom_or_ram = u_op.value[0].clone()
    # which register are we referencing in the trap?
    which_reg = Bits(regs[RAM_REG].value[4:8])
    # And what's the data value?
    data = Bits(acc.value[3:4]).clone()

    # The C program will finally write here, with the value of any button.
    pressed_button = Bits([Signal() for _ in range(12)])
    pressed_button.connect(pressed_button)
```

The way this works is that the circuit will pull the `trap` variable high to indicate that it wants to do something. Then, as shown earlier, the C program will actually handle whatever needs to be done. Finally, after this returns back, it's the circuit's turn to handle the input from the C program. This happens as follows:

The printer shifter is just a completely naive implementation connected directly to the ROM output port:

```
    printer_clock.connect(rom_output.value[2])
    keyboard_clock.connect(rom_output.value[0])

    for _ in range(21):
        printer_clock = printer_clock.clone()
        printer_shift.append(dff(printer_shift[-1], printer_clock))
    printer_shift = Bits(printer_shift[1:]).clone()
```

This holds the hammers that are going to be struck when the corresponding command is given, and when that happens, they update en-mass a bunch of flip-flops that hold the current value that's going to be printed to stdout. This is the actual logic that computes the gates that was already shown above.

```
    which_print = []
    for i in range(20):
        out = dff_circ(store_print_row, printer_shift.value[i])
        which_print.append(out)

    which_print = Bits(sum([x.value for x in which_print], []))
```

The keyboard handling actually cheats, and doesn't use a 4003 at all. Instead, because the Busicom calculator guarantees that only one bit is active at a time, it just stores the location of the active bit, and then whenever the first bit of rom_output is cleared it reset the row back to zero. This saves a bunch of space in the circuit without breaking any functionality (for the Busicom).

```
    keyboard_counter = Bits([Signal() for _ in range(12)])
    keyboard_counter.connect(dff(mux(rom_output.value[1],
                                     inc(keyboard_counter),
                                     Bits([zero]*12)),
                                keyboard_clock,
```

```
12))
```

The Micro-Op Table

The final component of the circuit is the micro-op lookup table that defines how each of the instructions should be implemented. Most instructions are relatively straightforward to implement. The simplest instructions are trivial. For example, to implement the LDM instruction that takes the low-4 bits of the instruction and sets the accumulator to that value it decodes in this way.

```
LOAD IMMEDIATE_4 // load the contents of the low-4 bits of the 4004 instruction
STORE ACC        // and directly store this into the accumulator register
LOAD PC          // load the 12-bit program counter to the queue
INC              // increment this value on the top of the stack
STORE PC         // store the updated program counter
```

Notice that the last three micro-ops actually tell the CPU how to advance to the next instruction. Most of the time this is obvious, but for jump and call instructions, this allows the program to handle them specially without any extra circuitry. I will show an example of that briefly, but first let me show another example of a slightly more complex op. Here, for example, is the encoding of the instruction IAC that increments the contents of the accumulator (and then writes an overflow to the carry flag).

```
LOAD ACC       // push the accumulator onto the front of the queue
INC            // increment the top value on front of the queue
STORE ACC      // store the top value on the queue to the accumulator
SHIFT_RIGHT_4  // shift the top of the stack right by 4 bits (to get the high bit)
STORE CARRY    // store that shifted value into the carry flag
LOAD PC        // load the 12-bit program counter to the queue
INC            // increment this value on the top of the stack
STORE PC       // store the updated program counter
```

Now let's show some complicated instructions. Here's the definition of the instruction JUN (a direct jump), for example. First its important to understand that this function is encoded as `0100AAAA BBBBBBBB`, where `0100` is the op-code to jump, and then the value AAAABBBBBBBB is the 12-bit jump target. Because instructions are 8 bits, this is actually represented with AAAA as the low four bits of the original instruction, and BBBBBBBB as the entire next instruction.

```
LOAD PC           // load the current program counter address
INC               // increment it so we can look at the next byte
STORE PC          //   and store it so that loads now refer to the next instruction
LOAD IMMEDIATE_4  // load the four lower bits of the *original* instruction
SHIFTLEFT4        // shift left 4 times
SHIFTLEFT4        // shift left 4 times again, for a total of 8
LOAD IMMEDIATE_8  // load the low 8 bits of the jump target
ADD               // compute (low-4 of instruction) << 8 + (low-8 of next byte)
STORE PC          // this is the jump target, go there
```

Other instructions are even more complicated than this. For example, the JMS instruction is what we'd think of now as a CALL, but instead of pushing the return address to a stack in memory (what memory??) the 4004 has a built-in 4-deep call stack in registers. So the JMS instruction has to handle the stack operations directly

```
LOAD PC2     // Start the stack shift,
STORE PC3    //   moving PC2 to PC3.
LOAD PC1     // Continue the stack shift,
STORE PC2    //   moving PC1 to PC2.
LOAD PC      // Now load the current PC
INC          //   increment to get the next instruction
STORE PC     //   and store it into the PC
STORE PC1    //   and also store it onto the PC stack
LOAD IMM     // First load the low nibble from the instruction (i.e., *PC)
SHIFTLEFT4   // Shift this value left 4 bits
SHIFTLEFT4   // And then shift another 4 bits so it's now *PC << 8
LOAD IMMBYTE // Now load the following byte *(PC+1)
ADD          // Add (*PC << 8) + *(PC+1)
STORE PC     // And finally store this value into the PC
```

But perhaps the most complicated instruction is the FIN instruction, encoded as 0011RRR0, which fetches the values in ROM pointed to by the the 0th general purpose register (for the low-4 bits) and the first (for

the high-4 bits), and stores that into the register pair RRR0 (for the low-4 bits of the word) and RRR1 (for the high-4 bits of the word). Here's the encoding of this instruction:

```
LOAD PC       // Load the PC to the queue
STORE TMP     // Save it aside so that we can come back later
LOAD R1       // Load contents of general purpose register 0 to the queue
LOAD R0       // And now load the contents of register 1 to the queue
MAKEBYTE      // A special instruction that computes queue[0]<<4 + queue[1]
PCHIGH        // Another special instruction that computes (PC&F00) | queue[0]
              // At this point, the top of the queue has the new address to load
STORE PC      // Actually write this value to the top of the queue
LOAD IMMBYTE  // Load whatever's pointed to by the PC to the queue
STORE REG1    // Store the low 4 bits to the register RRR1
SHIFTRIGHT4   // Shift the value on the queue to the right 4 bits
STORE REG     // And finally store this to the value to the register RRR0
LOAD PC       // load the 12-bit program counter to the queue
INC           // increment this value on the top of the stack
STORE PC      // store the updated program counter
```

And with that, we've completed our tour of the circuitry too. There are other instructions, but they are fairly obvious. And there are other tricks in the circuitry, but we're getting quite long already. So let's just call that a wrap.

Limitations

To the best of my knowledge, this is a nearly feature-complete implementation of the Intel 4004 and the original Busicom calculator should run more or less exactly as it did fifty years ago.

There are a few limitations with the CPU that should never matter in practice:

- I didn't implement the DCL instruction. This instruction allows you to switch I/O when you have more than one different devices attached to the ROM gates. But the Busicom ROM does not call this instruction and so it is omitted for space.
- The 4004 only can listen to ROM port 0; again not a problem for the original ROM, but other ROM files may not run correctly if they use more than this.
- If you type numbers in STDIN too fast (faster than 10 numbers per minute or so) the processor won't be able to keep up with you. So please be patient.

Keybindings

The calculator has mostly typical keybindings for entering digits 0..9 and for computing addition `+`, multiplication `+`, subtraction `-`, and division `/`. Then there are special keys that perform the other calculator functions:

- M acts like M+ and adds to the memory
- m acts like m- and subtracts from the memory
- r acts like RM and is the recall memory function
- o is a special sub-total function that has various uses
- S computes the square root function

Example calculations

You might want to try a few of these example calculations. On my computer these take somewhere between 3 and 10 minutes to complete.

- Calculate 2+3: `2+3+=` (should return 5)
- Calculate 85*72: `85*72=` (should return 6120)
- Calculate 85*72*4: `85*72*o4=" (should return 24480)
- Calculate 85/72: `85/72=` (should return 1.1805555555555)
- Use the memory functions `5M4Mr` (should return 9)
- Memory with multiply `12*5=M3Mr` (should return 63)
- Calculate square roots: `2S` (should return 1.4142135623730)

- Use the sub-totals: `1+2+o5+o=` (should return 8)

Longer calculations can be performed but should be done without the use of echo, because the Busicom ROM only can hold only a limited number of buffered keystrokes at a time. Care must be taken to not get unlucky when entering keystrokes manually or else the wrong calculation will be performed. (This is not a bug in the above C program, but a bug in the original ROM, which had subtle timing race conditions that in practice were not real problems because the calculator ran several thousand times faster than this emulated version does.)

Conclusion

If one were to rank programs by human-hours-per-byte, this program would be certainly top the ranking for the all the code I've ever written. I didn't keep good track of this, but I'd estimate I spent probably a few hundred hours over the last four years working on it. (Honestly I felt kind of empty after submitting the code back in June. I've been working on this since almost the start of the pandemic, and whenever I felt like I had to take a break from my large (mostly Python) actual work this was the perfect place to come and fiddle with the bytes of a program that fits comfortably on a single piece of paper.)

I don't think I'll have anything to submit to the IOCCC next year, or at least I don't have a plan to submit anything right now, but I had a great time with it again this year. If you liked this, I'd encourage you to go take a look at the other winners, which are similarly clever and deraged C programs.

If you want to be notified the next time I write something (maybe like this, maybe not) enter your email address here.
you@example.com Senden
There's also an RSS Feed if that's more of your thing.