PROGRAMMING THE IBM 701 EMULATOR

Marc Rochkind
15-July-2019; minor updates 24-Feb-2024
Second Edition

Introduction

This document explains how to program the IBM 701 using the emulator that you can download from mrochkind.com/mrochkind/a-701.html.

It's essential to also study the *Principles of Operation Type 701 and Associated Equipment*, referred to here as the PrincOp. References of the form PrincOp-NN are to a specific page in the PrincOp. The PrincOp was probably the first of the excellent IBM manuals to be distributed widely, and is worth every minute you spend with it. However, it's not necessary to read the whole thing before you read this document, as I'm going to present some basic 701 programming step-by-step. A PDF of the PrincOp is on my site as well.

This document, *Programming the IBM 701*, starts at the very beginning, showing how to input some instructions using the front panel alone. Then it goes onto a simple machine-language program stored on punched cards. That's enough to motivate the use of an assembler, also included in the emulator. All further examples are in assembly language; that novelty of machine language very quickly wears off. There's also a PL/0 compiler available (https://github.com/MarcRochkind/pl0compiler), which is built into the Windows version of the emulator.

About the 701

The 701, IBM's first commercial computer, was first shipped to a customer in April, 1953. An average of one machine a month was shipped; a total of 19 were installed. It had 2048 36-bit words, addressable by the half-word, for a total of 4096 addresses. Instructions were half-words: 12 bits for the address, 5 bits for the instruction code, and a sign bit. Logic was implemented with vacuum tubes; electrostatic memory was implemented with Williams tubes. Input-output components included 4 tape drives, 4 drums (each the same size as electrostatic memory), a card reader, a card punch, and a line printer. All interaction was via a control panel. Monthly rental was about $15,000 (about $130,000 in 2013 dollars).

Although the 701 was marketed as IBM's scientific computer, which it certainly was, because of its speed, it had no floating-point instructions. Programming was awkward because there were no index registers and because the sign bit didn't participate in shifts, greatly complicating the processing of non-numeric data, such as characters packed 6 to a 36-bit word, as was required for printing. These limitations were alleviated by the 704, which came out just a couple of years later. There is a clear progression from the 704 to the 709 to the 7090/94, but the 701 and the 704 are very different. That makes the 701 at once one of the most important and most obscure of IBM's early computers. I started on a 7090/94, on which I was an experienced assembly-language programmer. When I first looked into the 701, I expected it to be somewhat familiar, but I was surprised at how primitive it was. I have new appreciation for the 7090/94. (If you're

new to all this, the IBM 360 came out in the mid-1960s, about six years after the 7090. When I first used a 7090, it was already obsolete.)

IBM introduced the 650 about a year-and-a-half after the 701, for an entirely different market: Those who wanted a programmable computer that could replace unit record (card) equipment. The 650 was about 10 to 20 times slower than the 701, but it rented for about 1/5 of what the 701 cost, making it much more affordable. About 2000 650s were shipped, so it was a much better-known machine. In contrast to the 701, the 650 was a decimal machine and used a drum for its main memory. Donald Knuth dedicated his *The Art of Computer Programming* to the 650; nobody ever paid similar attention to the 701. The 650 was the first computer to sell over a thousand units. (The 1401, which came much later, was the first to sell over 10,000.)

Probably fewer than a thousand people ever programmed an actual 701, and probably fewer than a hundred of those are still alive. (These are just guesses.) Unlike the 704 and later machines in the 700/7000 series, the 701 wasn't emulated by newer models. There doesn't seem to be any mention of a 701 emulator on the web, unlike emulators for nearly every one of IBM's other computers, for which emulators are easily found. So, it's possible that my 701 emulator is one of the first ever created, and very likely that it's the only one anyone can actually use today.

Because it has only 33 instructions and a simple programming model, the 701 makes an ideal computer on which to learn programming at the machine and assembly language level. Few young programmers have ever had the experience of operating a computer from the front panel, but now, with my 701 emulator, everyone can. It’s one of the few emulators that includes a completely functional front panel. It also operates at speeds comparable to the 701 (I had to slow it down to achieve that), and has the same (emulated) I/O components. It's exactly as difficult to print on the 701 emulator as it was on an real 701, although I've written a printing subroutine that does the hard part (for 36 columns, anyway).

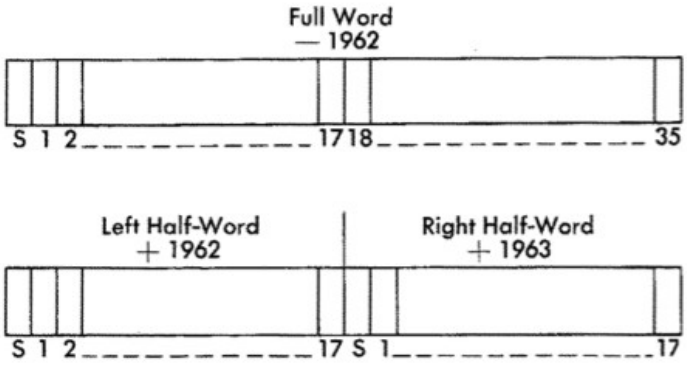
If you'd like a comparison of the 701 with today's computers, here are some fun facts:

- All of the 19 701s together couldn't store even one digital photo in main memory. You'd need about 200 of them.
- A fast desktop today is at least one million times faster than the 701, and sells for about 1/100 of the 701's monthly rental.
- The 701 would fill a large-sized living room. When I want to use my laptop I often have to look for it, as I'm not sure where I last left it. At least my iPhone, itself tens-of-thousands of times faster than a 701, is usually in my pocket.

Memory and Register Layout

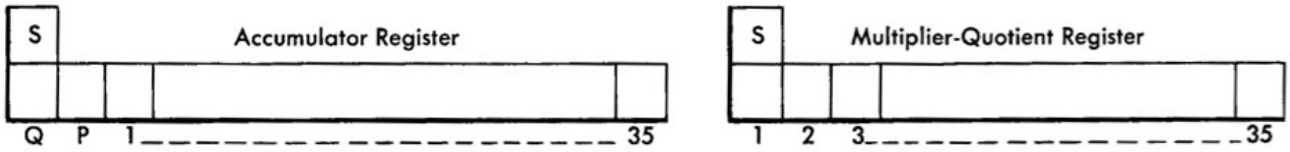
You really have to read the PrincOp, but I’ll say just a few words here to get you started.

The 701 has 2048 36-bit words of electrostatic (Williams tube) memory, addressed by 18-bit half-words, so there are 4096 addressable locations, which is all the 12-bit address part of an instruction can handle. Full (36-bit) words have a sign bit and 35 data bits; half-words have a sign bit and 17 data bits, as shown by this diagram taken from the PrincOp:

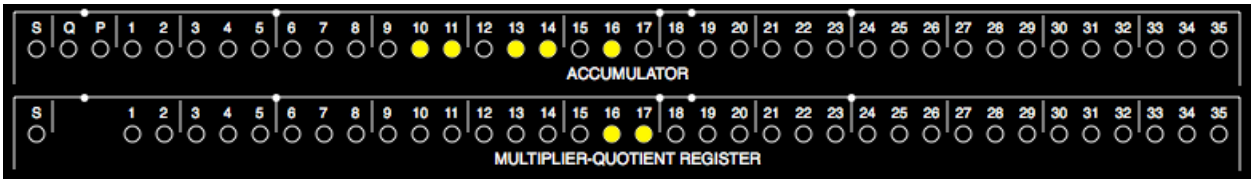


Most instructions use the sign bit of the instruction to indicate whether the address refers to a half-word, with a plus sign (sign bit of 0) or a full-word, with a minus sign (sign bit of 1).

There are two registers you can manipulate with instructions: an accumulator (AC) with a sign bit and 37 data bits, and a multiplier-quotient (MQ) with a sign bit and 35 data bits. The two high-order bits of the AC are known as Q and P, and get set when an overflow or shift occurs, but otherwise don't participate in arithmetic or in other operations. Again, from the PrincOp comes this diagram:



Long right and left shifts treat the AC and MQ as one continuous 70-bit register. That's also how they behave when they hold the product of a multiplication or the numerator of a division. At all times, the contents of the AC and MQ are shown on the front panel, as in this example:

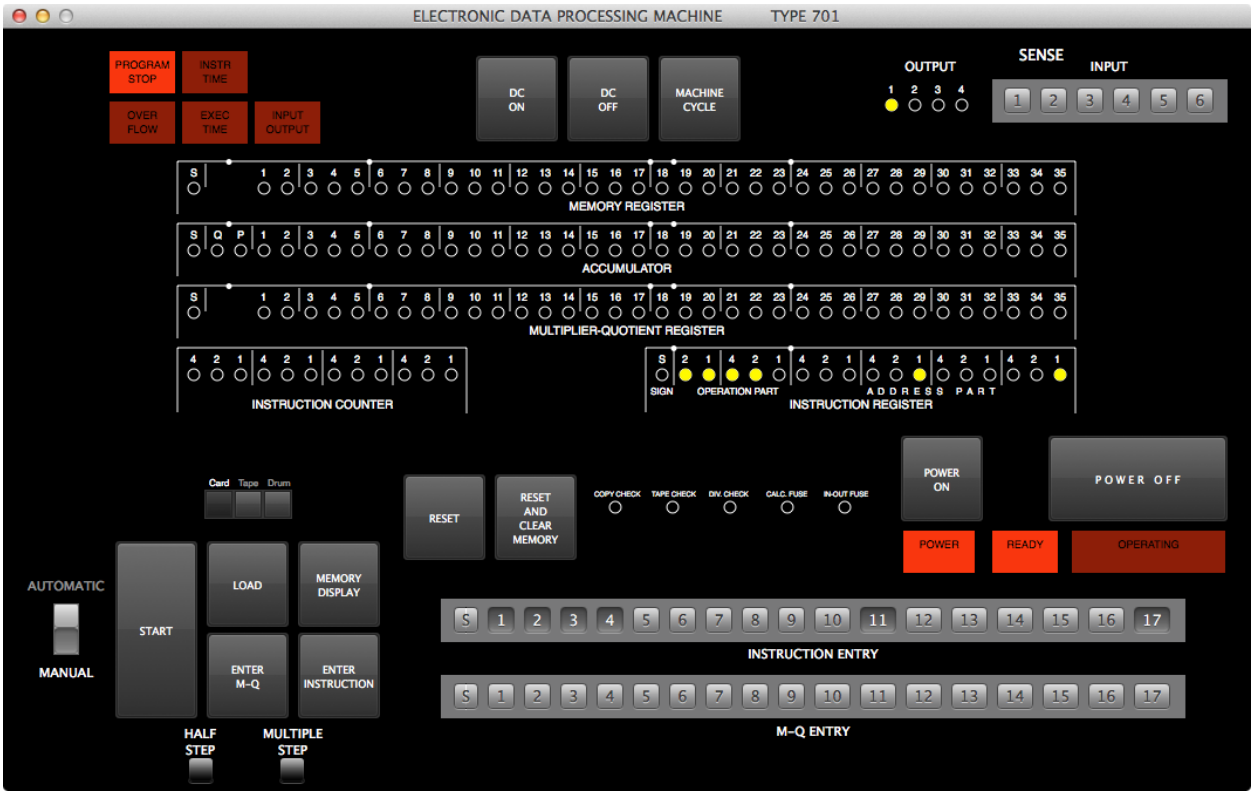


Entering and Running a Machine-Language Program

First, hit the POWER ON button to turn on the 701. You enter an instruction by setting it with the INSTRUCTION ENTRY (IE) keys and pressing ENTER INSTRUCTION (EI). For example, hit RESET AND CLEAR MEMORY make sure the AUTOMATIC/MANUAL switch is set to MANUAL. The instruction SENSE 65 lights the SENSE OUTPUT 1 light. PrincOP-69 says that the code for SENSE is 30, which is 036 in octal, and 11110 in binary. The instruction code is bits 1 – 5. The address is 65, 0101 octal, or 1000001 binary, which goes into bits 6 – 17, the rightmost 12 bits.

You set it on the IE keys like this:

Then you hit the EI key to enter and execute that instruction, after which the front panel looks like this:



The INSTRUCTION REGISTER lights show the instruction (SENSE 65), and SENSE OUTPUT 1 is on. The 701 has just executed one instruction.

For a more advanced example, you can enter the following program into the first 8 half-words of memory (addresses 0 through 7) and execute it. Memory locations are shown at the start of each line. (Remember that each location is a half-word, or 18 bits.) The binary codes for the instructions and addresses are shown in brackets, with a bullet between the instruction and the address, to make reading easier. The sign bit is represented in the brackets with an S.

```
0: -RADD          4                [S01010•000000000100]
1: -ADD           6                [S01001•000000000110]
2: LRIGHT         35               [ 10101•000000100011]
3: STOP                            [ 00000•000000000000]
4: 0                               [ 00000•000000000000]
5: 2                               [ 00000•000000000010]
6: 0                               [ 00000•000000000000]
7: 3                               [ 00000•000000000011]
```

The first instruction, –RADD, resets (clears) the accumulator (AC) and adds the word (because of the minus sign) at location 4, which is a 2. Since a word is two half-words, the high-order half-word at location 4 contains a 0, and the low-order half-word, at location 5, contains a 2.

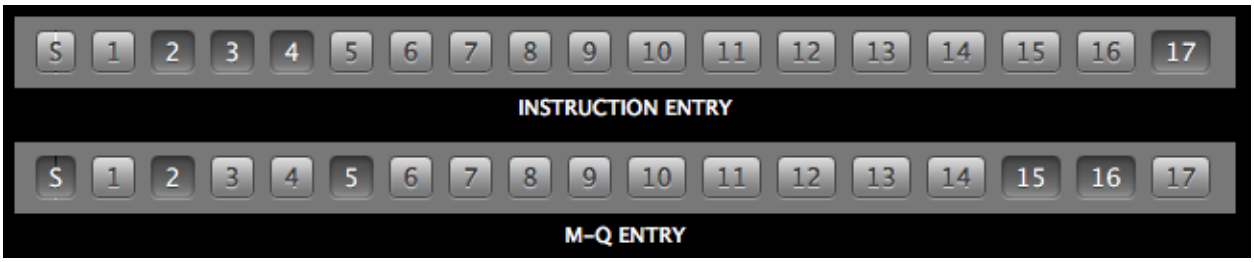
(The 701 had what would decades later be referred to as big-endian addressing.) The next instruction, –ADD, adds to the AC the word at location 6, which contains a 3. The result is in the AC. The third instruction, LRIGHT, shifts the combined 70-bit AC and multiplier-quotient register (MQ) to the right 35 bits, putting the result in the MQ. The fourth instruction stops the computer.

To enter these 8 half-words into memory, each half-word is set on the M-Q ENTRY keys and the ENTER M-Q button is hit to enter that data into the MQ, and you'll then see it the MULTIPLIER-QUOTIENT REGISTER lights. To enter it into memory, a STOREMQ instruction has to be executed, which is code 14 decimal, 016 octal, and 01110 binary. To load the left-most 18 bits of the MQ into location 0, a STOREMQ instruction is executed with an address of 0. Then the next instruction is set on the M-Q ENTRY keys, entered into the MQ, and the same STOREMQ instruction is executed with an address of 1, and so on until all 8 half-words are entered into memory. Here's the procedure step-by-step:

1. Set –RADD 4 on the M-Q ENTRY keys, like this:

2. Hit ENTER M-Q to enter the instruction into the MQ
3. Set STOREMQ 0 on the IE keys, like this:

4. Hit the EI key to enter and execute the instruction, thus storing –RADD 4 into location 0.
5. Set the next instruction, –ADD 6, on the M-Q ENTRY keys.
6. Hit ENTER M-Q.
7. Leave the STOREMQ instruction on the IE keys, but change the address to 1. The IE and M-Q ENTRY keys now should look like this

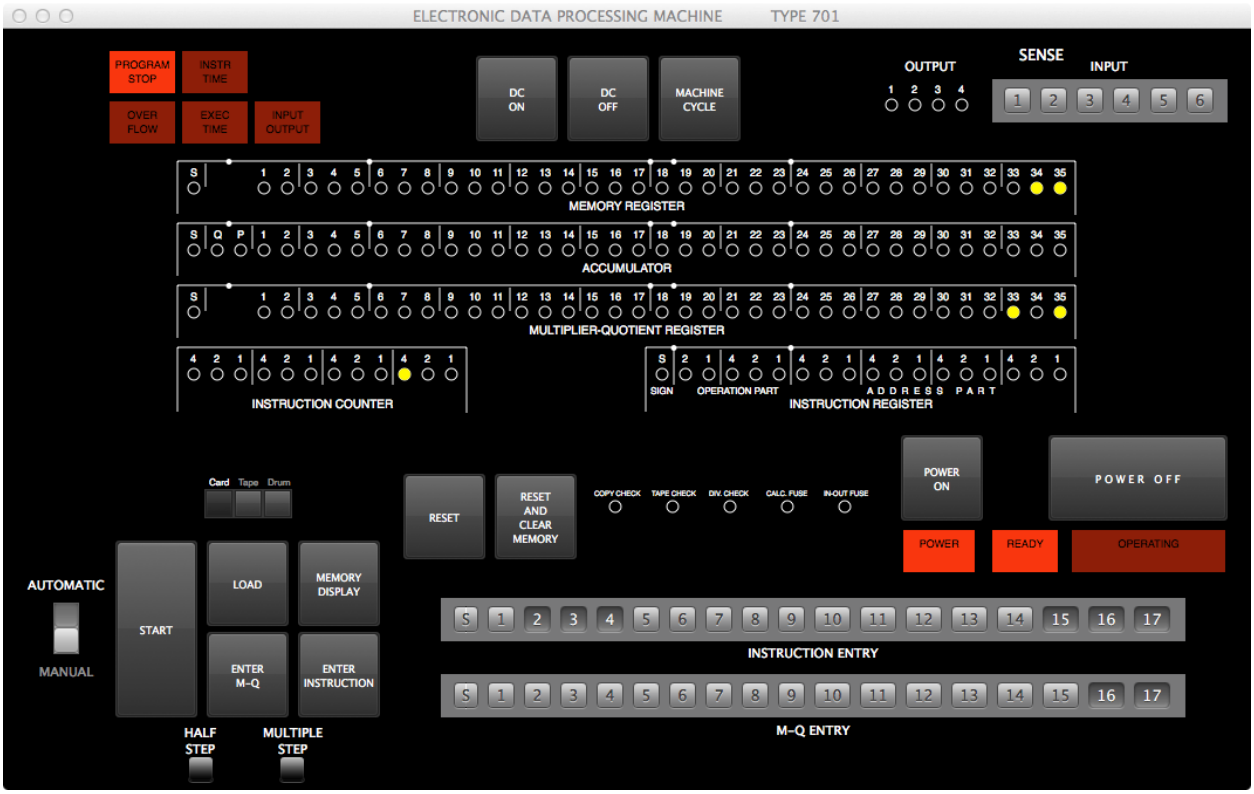


8. Hit the EI key to execute the STOREMQ, thus entering the instruction into location 1.
9. Continue in this way until all 8 half-words are entered (4 instructions and 2 words of data).
10. You can check that you've entered everything correctly with the Dump Electrostatic Memory command on the emulator's Debug menu (not part of the 701 itself), which shows the following in the Log window:

```
0: 172036 0520004     0: - RADD     4
1: 167942 0510006     1: - ADD      6
2:  86051 0250043     2: + LRIGHT   35
3:      0 0000000     3: + STOP     0
... same as above ...
5:      2 0000002     5: + STOP     2
6:      0 0000000     6: + STOP     0
7:      3 0000003     7: + STOP     3
8: 262143 0777777     8: - COPY     4095
... rest same as above ...
```

(If you don't see the Log window, choose Show Log on the Window menu or Show All Windows on the View menu.)

11. If there are any errors, correct them by entering something different at that memory location.
12. Check that the INSTRUCTION COUNTER is at 0 (all lights off), switch the computer to AUTOMATIC mode, and hit START. The program will execute starting at location 0, and you should see 5 (2 + 3) in the MQ, like this:



Loading a Program from Binary Cards

If you've actually entered and executed the program in the previous section, you've already received one benefit of having an accurate emulator: You've realized that entering instructions via the front panel is extremely tedious, something a programmer is willing to do at most once.

Fortunately, the 701 can also load a program from cards, tape, or even a drum. Cards were the most common way, but first you had to get the program onto the cards, which means punching binary cards.

The Model 029 keypunch that I used at the University of Maryland in 1967 punched what were sometimes referred to as Hollerith cards, which means that an 80-column card could hold 80 characters, one per column. There were 12 rows on the card, so each character was represented by a pattern of up to 12 punches. (You can try out an 029 at www.masswerk.at/keypunch).

However, those weren't binary cards. A 701 binary card considers each of the 12 rows as two pairs of 36-bit words, using only the first 72 columns. A card holds 24 words or 48 half-word-sized instructions (18 bits each). On a card, the order is from bottom-to-top (9 row to 12 row), and left-to-right within each row. Ignoring the card layout, and just considering the 24 words, a card is a sequence of 24 36-bit words.

The 701 never does anything in hardware that it can do in software, as you saw when you had to enter a STOREMQ instruction just to get the contents of the MQ stored into memory. The same idea applies to loading a program from cards: Hitting the LOAD button doesn't read a whole card, but only executes three instructions:

```
READ            2048
-COPY           0
TR              0
```

READ starts reading I/O component 2048, which is the card reader, which must already be loaded with cards. The 24 words on the card are made available to the program in sequence: 9-left, 9-right, 8-left, 8-right, and so on, up to 12-right. One a READ has been executed, the program must execute a COPY instruction for each word; each transfers one word from the card to the memory address specified on the COPY. In the sequence above, exactly one word, 9-left, is transferred to addresses 0 and 1. Recall that the minus sign means that the COPY transfers a whole word. (The three instructions executed by LOAD are not themselves in memory.) The third instruction then transfers to address 0, executing whatever instruction was in the half-word at 9-left on the card. As a full word was loaded, a second instruction (the other half-word at 9-left) was also loaded.

So, hitting LOAD causes two instructions to be read from the card into memory addresses 0 and 1, and the first is then executed. As an example, suppose the two instructions are:

```
0: SENSE          66                [ 11110•000001000010]
1: STOP           255               [ 00000•000011111111]
```

You punch a binary card (I'll explain how shortly) that looks like this:

You can see that 9-left is punched with the 18 binary digits for the SENSE instruction (11110000001000010) followed by the 18 binary digits for the STOP instruction (00000000011111111). The rest of the card is empty, but nothing else on it would have been read anyway, because only a single COPY instruction was executed.

If you place this card into the card reader and hit LOAD, the SENSE will cause SENSE OUTPUT 2 to be lit (its number is 66), and then the computer will STOP. The address 255 is the next instruction to be executed if you hit START after the computer stops; I have it there just so I can see it in the Log window and verify that this is the instruction that caused the stop.

Given that only two instructions are loaded by the LOAD button, and programs are much longer than that, clearly those two instructions must include at least another COPY, to read in a second word (9-right). In fact, if you punch the following six instructions on the card (from 9-left through 8-left), a LOAD will cause the whole card to be read:

```
0: -COPY          2
1: RADD           3
2: ADD            0
3: -COPY          4
4: STOREA         3
5: TR             2
```

Here's what happens:

1. Hitting LOAD causes the first two instructions (–COPY and RADD) to be loaded into addresses 0 and 1.
2. The –COPY instruction at 0 is executed, which causes two more instructions (ADD and another –COPY) to be loaded into addresses 2 and 3.
3. The RADD at 1 is executed, causing the half-word at 3, the instruction –COPY 4, to be loaded into the leftmost 18 bits of the AC.
4. The ADD at 2 (just copied there by the –COPY at 0) causes the half-word at 0, which is a –COPY 2 instruction, to be added to the AC. All we really care about is the address part, which now equals 6 (4 + 2).

5. The –COPY at 3 is executed, which causes another word (the instructions STOREA and TR) to be loaded into 4 and 5. At this point all 6 of the instruction shown have been loaded.
6. The STOREA at 4 stores the address in the AC, 6, into the address part of the –COPY at 3, which changes it from –COPY 4 to –COPY 6.
7. The TR causes a jump to address 2, which again adds the half-word at 0, the –COPY 2, to the AC, changing the address part in the AC from 6 to 8.
8. The –COPY at 3, whose address has been changed to 6, copies another word to addresses 6 and 7. These are the first two instructions of the program that follows the instructions shown above, whatever they may be.
9. The address in the AC, now 8, is stored into the –COPY at 3.
10. Looping continues with the TR 2. Each time the –COPY at 3 is executed, another word is copied into memory, as its address keep increasing (10, 12, 14, ...).
11. When the –COPY at 3 attempts to copy the word at address 24, instead of copying, it senses an end-of-record condition, as there are no more words on the card. That causes execution to skip to the *second* instruction after the –COPY, which is at address 6. That is in fact the first instruction of the program that follows the six-instruction self loader, and the program then executes.

A few items to note about the execution of the self-loader, things very common to 701 programming:

- Lots of address arithmetic, with instructions added to other instructions, not because adding op codes gets you anywhere, but because it's the address arithmetic that's of interest.
- Lots of modification of instructions, as the 701 has no index registers.
- The self-loader just barely works. Initially, instructions are copied in just before they're executed. Just barely working is entirely satisfactory.

As this six-instruction self-loader can load a whole card, all you have to do is put it at the front of any program you write, punch the card, and you can load and execute it. That gives you 42 instructions to do anything you want. Clearly, some programs are longer, and I'll show a self-loader that can load a whole deck of cards later.

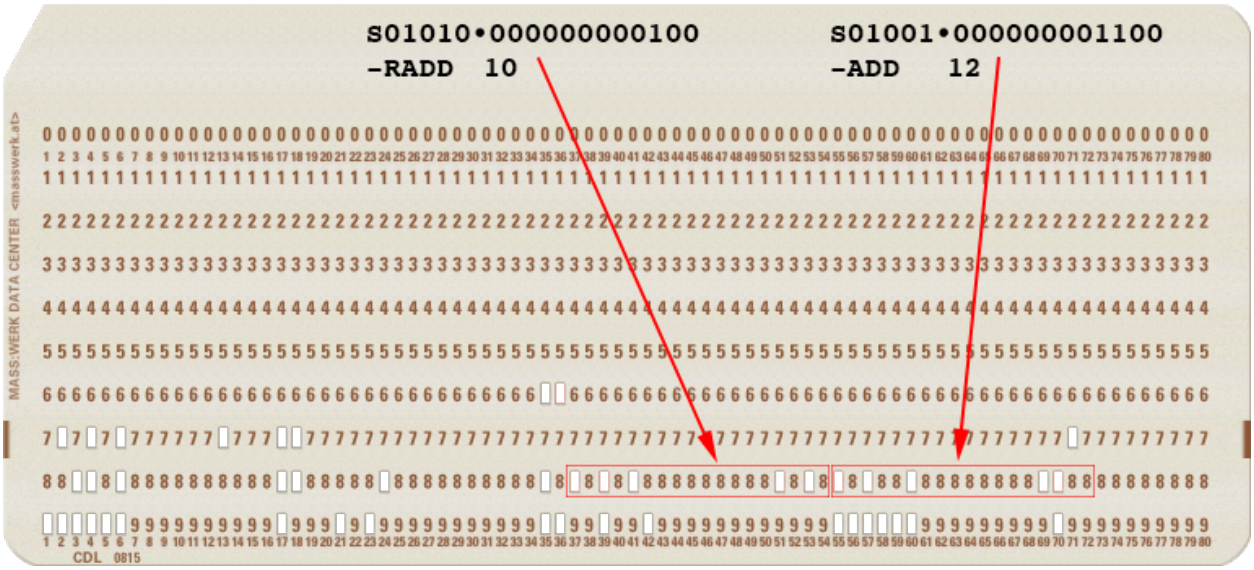
(By the way, I didn't come up with any of the self-loaders. You can see them in PrincOp-78-81.)

Now we can precede the program I showed before that added 2 and 3 with the six-instruction self loader:

```
 0: -COPY          2                [S11111•000000000010]
 1: RADD           3                [ 01010•000000000011]
 2: ADD            0                [ 01001•000000000000]
 3: -COPY          4                [S11111•000000000100]
 4: STOREA  3                       [ 01101•000000000011]
 5: TR             2                [ 00001•000000000010]
 6: -RADD          10               [S01010•000000001010]
 7: -ADD           12               [S01001•000000001100]
 8: LRIGHT         35               [ 10101•000000100011]
 9: STOP                            [ 00000•000000000000]
10: 0                               [ 00000•000000000000]
11: 2                               [ 00000•000000000010]
12: 0                               [ 00000•000000000000]
```

You can see that programming in machine language is a real pain: Because the addition program had to be moved to start at address 6, to make room for the self-loader, I had to change the address on the –RADD from 4 to 10, and the address on the –ADD from 6 to 12.

Now, if we could only get these 14 half-words punched onto a binary card, we'd be set. The card would look like the following one. To help you find the instructions at 6 and 7, I've annotated the card; the printing at the top certainly is not on the card.



The emulator doesn't have a reader that can read real cards, so, instead, binary cards are in a file with a ".deck" suffix that contains a sequence of binary words, each corresponding to a word on the card. One card follows another in the file, so the first 24 words are card 1, the next 24 are card 2, and so on. To make it easier to read and write these files, each word is 64 bits, not 36, which is awkward to read and write. The leftmost 28 bits of each word are wasted. The emulator handles the translation between 64-bits and 36-bits automatically.

Still, it's not easy to prepare such a file, even with a hex editor. So, the emulator has a Convert to Binary command on the File menu. It takes a text file of half-words in octal notation, one per line, and punches a binary card deck as though it were punched by the emulated Type 721 Punched Card Recorder that you can see in the INPUT-OUTPUT COMPONENTS window. Then you can click the Save Deck button to save the card deck for later loading by the Load Binary Deck button on the emulated Type 711 Punched Card Reader.

Let's go through it step-by-step:

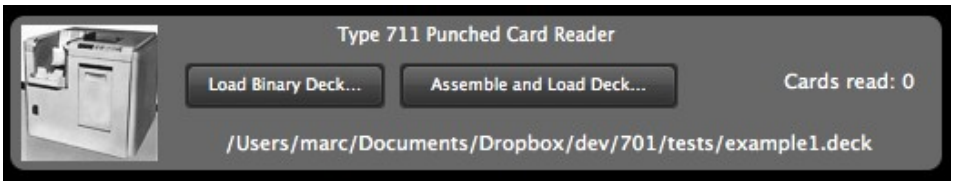
First, you prepare the text file of octal half words. Looking at the 14 half-words shown in the program above, that file looks like this:

```
0770002
0120003
0110000
0770004
0150003
0010002
0520012
```

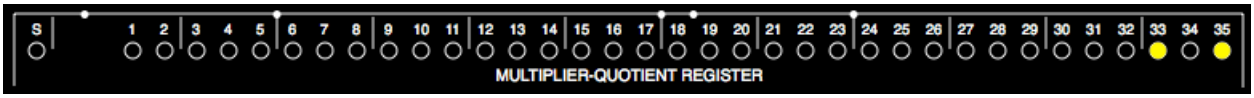
```
0510014
0250043
0000000
0000000
0000002
0000000
0000003
```

You can use any text editor, such as TextEdit or NotePad, to prepare this file, which you save anywhere you like, with a ".txt" extension. Next, you choose Convert to Binary on the File menu, select the text file, and you'll see that one or more cards appear on the card punch:

You click Save Deck and save the deck anywhere you like, with a ".deck" extension. Then, you click Load Binary Deck on the card reader, select that deck file, and it will show as loaded into the card reader. I called the deck example1.deck, so it looked like this:



With the deck loaded into the card reader, you can hit the LOAD button on the front panel (make sure the 701 is on), and the self-loader will execute, load itself and the adding example, and the program will run, displaying a 5 in the MQ:



If you want a picture of a card, similar the ones I showed, on the Mac you can hold down the Option key while clicking on the Load Binary Deck button and the deck shown in the card reader will be punched by the web site www.masswerk.at/keypunch. (This feature isn't available on the Windows version.) It's really fun to see this in action, so much fun that I've posted a movie that you can see at mrochkind.com/701/cardpunch.m4v.

The Assembler

You can't go very far coding instructions in binary, or even octal, and referring to addresses by absolute numbers, whatever the base. So, while I wasn't there, my guess is that only two groups of people ever coded a 701 that way, and only briefly: (1) the engineers debugging the hardware, and (2) some early programmers who wrote a primitive assembler, to be replaced very shortly by those same programmers using that assembler to program a better one.

Such a primitive assembler has two main jobs: To allow you to use mnemonics for instructions, and to allow you to refer to memory addresses symbolically.

Obviously, the original 701 assembler ran on the 701, but the emulator's 701 assembler doesn't; it runs on the same Mac that's running the emulator. You run it by clicking the Assemble and Load Deck button on the INPUT-OUTPUT COMPONENTS window, and it magically produces a card deck loaded in the card reader, reader for you to hit the LOAD button on the front panel. (Make sure the computer is on.) The assembler source must be in a text file with a "txt" or "a" extension, somewhere accessible on your computer.

The mnemonics that the assembler accepts are the same ones I've been showing, as in this earlier example:

```
0: -COPY          2
1: RADD           3
2: ADD            0
3: -COPY          4
4: STOREA         3
5: TR             2
```

They can be in upper or lower case. Symbolic address allow you to replace the absolute references with symbols, so you could write the program like this:

```
a         -COPY       b
          RADD        c
b         ADD         a
c         -COPY       d
d         STOREA      c
          TR          b
```

The complete set of instruction mnemonics are the same as the abbreviations listed on PrincOp-69, but without the spaces. For example, R ADD becomes RADD or radd. There are two exceptions: the instruction TR + is written TRPLUS, and TR 0 is written TRZERO.

Since the assembler isn't part of the 701 emulation, and therefore doesn't have to be historically accurate, it has an include facility, which would be impossible with punched cards. The pseudo-op INCL causes another file to be read in that must be in the same folder as the main source file. This is really handy for including the self loader at the start of each file. You can put the six-instruction self loader in a file called *self-load-card.a* and then just include it in any program that needs it, such as this assembly version of the 2 + 3 example that I showed earlier:

```
          INCL        self-load-card.a
          -RADD       a
          -ADD        b
          LRIGHT      35
          STOP
a         WORD        2
b         WORD        3
```

Note that both the loader in *self-load-card.a* and the program just above both use the labels *a* and *b*, but the assembler treats labels as local to their file, so there's no conflict. If you want a label to be global, you have to start it with a period, as in *.print*.

Also, note the psudo-op WORD which simply assembles its address in a word. The similar HWORD would assemble its address in a half-word. If you just want some empty memory, you

can use the psudo-op RES which reserved a number of half-words equal to its address. For example:

```
x         res          10              # reserve 10 half-words
```

reserves 10 half-words of memory, the first of which is at address *x*. This example also shows a comment: Anything on a line starting with "#" is a comment. Blank lines are ignored and treated as comments as well.

The parts of a line—label, op code, and address—can be separated by spaces or tabs.

In in the 2 + 3 example I explicitly coded words to hold the constants 2 and 3. The assembler can do that automatically if you use a special address that starts with a "="; this is called an immediate constant. The example could then be rewritten like this:

```
          INCL        self-load-card.a
          -RADD       =2
          -ADD        =3
          LRIGHT      35
          STOP
```

Keyboard Shortcuts (Mac only)

When you're debugging a 701 program, you have to keep editing the source and rerunning it, and it's tiresome to keep going through the assemble and load sequence. You can do it in one step with Assemble and Load on the Debug menu. The first time, it prompts you for the file, just as the Assemble and Load Deck button does, but subsequently it just keeps re-assembling and loading the same file. So the steps are: save in your editor, activate the 701 app, and type ⌘-G.

If you've programmed a STOP, so you can check the results before continuing, you can type ⌘-R instead of hitting START.

With these two shortcuts, on a small screen you can expand the Log window and do all your work while you watch it, ignoring the front panel. You're debugging your 701 program using facilities not part of the 701 (assembler, keyboard, and log window), but in the end you get a legitimate 701 program, so what the heck.

A More Complete Self Loader

The self loader I showed can load only a card, but on PrincOp-80 there's a loader that can load the whole deck, whatever is loaded in the card reader. It's in the file *self-load.a* and looks like this, in assembler (I've kept the numeric addresses from the original):

```
# Self loader to read until end of input (read will fail)
# See PrincOp-80
         -COPY       2
         -COPY       4
         RADD        5
         ADD         0
         STOREA      5
         -COPY       4
         TR          2
         TR          10
         READ        2048
         TROV        5
# Copy skips to here on end of input
```

From here on out, I'll include this file in every assembly instead of *self-load-card.a*. You can read a very complete explanation of how it works starting on PrincOp-80. There's no problem with its absolute addresses, as the self loader always goes first and starts at address 0.

Subroutine Calling Sequence

The recommended basic subroutine calling sequence is described on PrincOp-76 and goes like this: The calling subprogram executes these two instructions, assuming *.sub* is the label of the first instruction of the subroutine:

```
a        RADD        a
         TR          .sub
```

The RADD puts the address just preceding the TR into the AC. Then the TR transfers to the subroutine, which has this form:

```
.sub     ADD         =2              # address 2 past caller's RADD
         STOREA      return          # setup to return there
         ...
return   TR                          # return to the caller
```

The subroutine calculates the address 2 past the RADD, which is just after the TR that transferred to the subroutine. Then it stores this address in its own TR that is the last instruction it will execute, causing control to pick up just after the caller's TR.

The caller's RADD, referring to its own address, is essentially "put me in the AC." All that's needed is "put my address in the AC," but there's no instruction for that, and having the op code there does no harm. To avoid a proliferation of addresses like the *a* here, the assembler allows you to refer to the address of an instruction with an asterisk, making the calling sequence:

```
         RADD        *
         TR          .sub
```

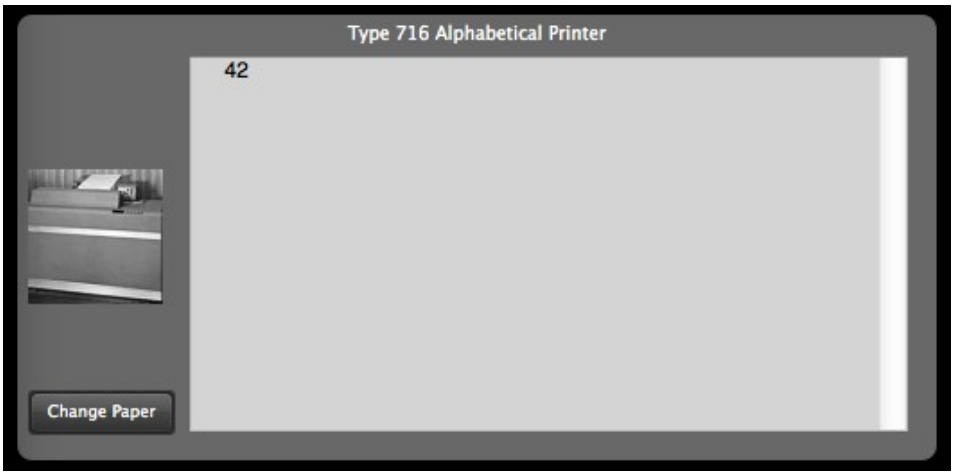
There are other places where an increment off of the current address is useful, so you can also code **+n* or **–n*, such in this instruction that stops the computer and then goes to the next instruction when START is hit:

```
         STOP        *+1
```

If there's only one argument, it can usually go into the MQ. (The calling sequence itself claims the AC.) For example, here's a program that prints the number 42, using two subroutines that are supplied as built-in examples that you copy with Copy Examples on the File menu:

```
INCL        self-load.a
-LOADMQ     =42
RADD        *
TR          .print-number
STOP
INCL        print-number.a
INCL        print.a
```

This was the output:



Note that *print-number.a* and *print.a* are not part of the 701, but were subroutines written by me.

An alternative calling sequence, useful if there's more than one argument, is also described in the PrincOp. Here, the arguments are in order after the TR that transfers to the subroutine. For example, if there are three arguments, 22, 33, and 44, the calling sequence would be:

```
RADD        *
TR          .sub
HWORD       22
HWORD       33
HWORD       44
```

The subroutine can access the arguments by adding 2, 3, and 4 to the address in the AC when it's called, but then must transfer to the address in the AC plus 5 to return, instead of that address plus 2, which worked when the only argument was in the MQ. Naturally, the subroutine has to be coded for the exact number of arguments. You can't just call a subroutine expecting its argument in the MQ with the argument someplace else.

Additional Assembler Features and Limitations

If you want an address as a constant, you can use an @ in an immediate constant to refer to an address, like this:

```
loadmq      =@hdr1
```

This creates a half-word somewhere in memory loaded with the address *hdr1* and assembles a reference to that half-word. The result of the instruction is the address *hdr1* (some number) loaded into the MQ. Note that the above line is completely different from:

```
        loadmq      hdr1
```

which loads the contents of address *hdr1* into the MQ. And it's also different from:

```
        loadmq      =hdr1
```

which is illegal.

Constants used with HWORD or WORD are taken as octal if they start with a 0. This doesn't apply to immediate constants (e.g., =012 is 12 decimal, not 12 octal).

Immediate constants and constants used with HWORD and WORD are limited to 18 bits; that is, they must be less than 262,144 in absolute value. If you need a larger constant, assemble two HWORDs, and use the address of the first of them (remember, the 701 is big-endian), like this, from the *sqrt.a* example:

```
y0      hword       0377777         # 1 - 2 ** -35
        hword       0777777         # (e.g., bits 1-35 all 1)
```

This assembles at y0 a 36-bit constant of all ones, since this would be disallowed:

```
        word         0377777777777 # not allowed
```

The CHAR pseudo-op assembles characters into memory as constants:

```
str     char        "PRIMES LESS THAN 1000"
```

The address *str* refers to a word that contains the first 6 characters, followed by as many additional words as it takes to contain the whole string. The quotes aren't included.