

An EDSAC 2 emulator

EDSAC 2 was an early computer, the successor to EDSAC 1, in the University Mathematical Laboratory, Cambridge. There are descriptions and emulators of EDSAC 1 on the web and there is a current project to build a replica, but as far as I know no emulator for EDSAC 2 exists. These notes describe an emulator for EDSAC 2 which recreates some of the atmosphere of computing at that time.

Overview

EDSAC 2 was a valve computer originally with a magnetic core memory of 2×1024 words¹ of length 40 bits. Later further memory, the **Main Store**, was added. It had an order code of approximately 100 instructions including extensive floating point operations and allowing some calculations to be performed to 80 bit precision. It was not until the 1990s that single chips included comparable hardware. One of the memories was hard-wired and contained an assembler and other utilities together with mathematical functions and some scientific software.

The machine orders were implemented by a matrix of 32×32 fast magnetic cores. When a core was switched pulses were generated on wires passing through it; two wires determined the next core to be switched and another operated a gate in the hardware as part of the sequence necessary to perform the required arithmetical and logical operations. There was even some overlapping of instructions with a new order being started while the previous one was finishing. The actual hardware gates were contained in two sets of 41 and 11 identical chassis each handling one bit of the 40 bit arithmetic or the 11 bit memory address. This meant that in the event of a hardware failure a single chassis could be quickly changed. More technical details are given in the accompanying documents.

The machine was operated by a human from a console with switches, buttons and lights. Input was from one of two 5-track paper tape readers and output again to various 5-track paper tape punches. There was also a line printer (including an interrupt flag which allowed programs to detect when its buffer was empty) and a digitally addressed cathode ray tube. An X-Y plotter also existed but I have no recollection of how it was connected or the exact orders used to control it.

¹In fact there was a gap in the addresses in the reserved store

Emulation Strategy

The most perfect emulation would follow the above description of the control matrix determining the sequence of operations on simulated electronic circuits. This would ensure that every bit in the emulation was exact. Unfortunately I do not have details of the control matrix wiring. The alternative which has been adopted is to emulate the 100 or so machine instructions as described in ‘Programming for EDSAC 2’. This does not contain all the necessary details, since some were irrelevant to the ordinary user, but these can be deduced from existing programs or guessed. The limitation of this approach is that issues like rounding in floating point operations or the state of the hardware after arithmetic overflow could be wrong.

The emulator is written in Python. The store is a list of Python integers and the fixed point arithmetic done with Python long integers. Floating point operations are done using Python floats; these use 64 bits and so should reproduce exactly the 40 bits used in EDSAC. Some of the switches, buttons and lights on the operators’ control panel are emulated using Tkinter so one uses the emulator in a way that resembles the real machine. It is possible to take advantage of the fact that one is using an emulator to get debugging information that was not available on the real machine. The speed of the emulator is approximately that of the original machine and a few demonstration programs are included.

Difficulties with the Main Store

The **Main Store**, which was added later was a commercial core store of 16384 40 bit words. The address part of the EDSAC 2 instruction does not contain enough bits to address this and a hardware modification was made to get round this. At the same time a new assembler was written to translate written instructions using main store addresses into a form appropriate for the modified hardware. This resided on magnetic tape and I do not have a copy of it and therefore writing an emulator that will handle programs written to use the **Main Store** poses a problem. It is possible, although very tedious, to write orders with main store addresses and use the built-in assembler and a very simple translator could be written to simplify this. Unfortunately some other new features were added in the new assembler that became possible with increased memory during assembly. Since any existing main store programs will probably use these features it would not be possible to run such programs.

The solution adopted is a compromise. First the main store hardware can be turned on or off; the reason for this is that interpreting the extra address bits slows the emulator. Secondly an small assembler has been written in Python which implements some of the features of the original main store

assembler. Being realistic, no one will want to run a large program that uses the main store and so all the assembler permits is demonstrating the use of the store. The assembler makes extensive use of regular expression matching as a simple way of producing its output. The disadvantage is that it does not provide rigorous syntax checking and while it should assemble correct programs it may also accept incorrect ones without detecting errors and illegal input may cause the assembler to crash rather than producing proper error messages. No attempt has been made to implement some features of the original assembler, for example the facility for breaking a large program into sections and placing these in the main store in a form ready to be copied to the free store for execution.

The original main store assembler was read from magnetic tape by the directive `25/s2` and this is no longer appropriate. When the emulator is switched to main store mode the external assembler is automatically selected but it is possible to override this and use the built-in assembler in the reserved store.

Other limitations in the emulations

Magnetic Tape

EDSAC 2 had four magnetic tape decks. Orders 114-119 controlled these but no details are given in the Programming Guide. (Order 98 is also described as “used in magnetic tape control routines”; this is correct but it is also used elsewhere and therefore its guessed function has been implemented.) I have been unable to find any detailed specifications for the orders controlling the magnetic tape and so the only way to discover their actions is from the software subroutines in the reserved store. The detailed arguments by which the operation of the orders has been deduced are given in the appendix. The built-in subroutines appear to work as described in the “Users Guide to the EDSAC Magnetic Tape System”² and I am reasonably confident that the specification of the underlying hardware instructions is correct.

The four tapes are implemented as a python object. They are stored in the file `magtapes.txt` in python ‘pickle’ format. There is the option for the emulator to save them overwriting the file `magtape.txt` and it would be possible to replace the default input by one containing additional data. A back-up copy of the original tape is provided. On the original machine tape 1 contained library subroutines that could be incorporated into users programs and the default tape contains an example (see below).

²I am grateful to the librarian of the Computer Laboratory, Cambridge for permission to include this document

Knobs, switches etc. on the control panel

The important ones are `set start`, `set start and clear` and `run`. Either of the first two initialises everything and the second clears the store to all ones in addition. The real run key could be lowered to start the machine and raised to temporarily halt it. This is emulated by a radio button marked up and down.

To run a program copy the code to `input_file.txt`, or browse to one of the sample programs, press the `set start and clear` button and the `run down` button.

There is little point in emulating some of the features of the control panel. Those omitted from the panel are:

- The main on-off switches
- Switches connecting different punches and the line printer to output channels
- Switches to run blank tape on different punches
- Paper feed switch, on the line printer?
- Punch Error light
- Switch to exchange tape readers
- Various unimportant neons
- The `not-writing`, `control`, `margins` and `eng.key` lights which are irrelevant

Interrupt controls In practice this was relevant only to the line printer or paper tape punches although it was possible to manually interrupt a program, store the state of the machine and later resume.

SET BINARY key Software to read input tapes in base 32 directly into the store was built in. This could also be accessed by an appropriate software jump, so an original tape could still be read.

SET REPORT key This is of little use.

TRACE key This was a hardware debugging facility. If the TRACE key was down details of jump instructions were recorded in the reserved store by a jump to location 1025. The details could be printed using service routines which I do not have, so although the built-in software is there and this facility could in principle be implemented I have not done so.

Loudspeaker volume control EDSAC had a loudspeaker whose main purpose was to make an encouraging noises about what was happening. In particular a continuous loop in a program could be detected. With a bit of ingenuity it was possible to play music and R A Jennings wrote a ‘compiler’ which enables data to be entered in a form closely following the music score. Sadly it will be difficult to emulate this facility but experiments are in progress.

Input/Output

The input to EDSAC 2 was 5 hole paper tape. The optical tape readers made in the Mathematical Laboratory read 1000 characters per second and there were also some commercial slower readers. The paper tape code was unique to EDSAC and the assembler included appropriate look-up tables. Since the assembler requires input in the original code but no current software supports this input ‘tapes’ are actually files written in ascii and translated at the start of a program into lists of integers representing EDSAC code. The emulation of the 69 order which reads the character currently under the tape reader gets the next entry in these lists. A few of the characters have no ascii equivalent and alternatives have been used, these are:

blank tape	b
carriage return	\r
line feed	\n
letter shift	"
figure shift	\$
subscript 10	e
subscript 2	%

Note that there were separate carriage return and line feed symbols so the any \r characters are removed and newline (\n) is then replaced by (\r\n) when the input files are processed.

The names of the ascii files containing the input for the two tape readers can be set on the emulator control panel; the default filenames are `input_tape1.txt` and `input_tape2.txt`.

The output is stored internally as the 5 bit characters that would have been sent to a paper tape punch but the reverse translation of this to ascii is done after the program has stopped and the result is sent to the terminal or to a file called `output_tape.txt`.

There were in fact two **output channels** and output could be sent to either or both as set by the program (106 order). Any of three paper tape punches

or the line printer could be connected to either channel by switches on the control panel. There seems little point in emulating this flexibility and output is sent to internal buffers connected to the **channels** and the output of both channels concatenated before being sent to the terminal or the output file.

Because of the use of a separate internal code for characters it is difficult for the external assembler (which by-passes the internal code) to implement the 'title' facility. Also data tapes read in by a program need to be translated into internal code; this is done automatically if the data is appended to the program.

Debugging

Finding errors in programs caused significant headaches for users of the original machine but additional aids have been incorporated in the emulator. The current machine order, the state of the modifier registers, the accumulator and the overflow flip-flop can be printed on the terminal. This output could easily be piped to a file if it was too much to view.

By default this facility is turned on if the machine is obeying orders in the free store but not for the reserved store so the operation of the assembler, print routines and built-in subroutines is normally suppressed.

It is also possible to step through orders one at a time. To do this one should insert a **wait** (102) order in a program and run to this point then press the single step on button before lowering the **run** switch to continue. The **step** button then continues one order at a time.

The only signal of an error state was the **report stop** which printed very limited information. Where possible, errors leading to a report are trapped by the emulator and the cause is displayed in the emulator control panel. Errors during assembly triggered a report by obeying an illegal order in the reserved store and it is not possible to add any more information in these cases. The value of modifier **t** contains some information about the type of error but this is not given in the later editions of the programming manual which assume the main store assembler is being used.

Testing

It is inconceivable that a project of this size contains no bugs. A good test is that the emulator runs the built-in assembler code and the assembled programs appear to work correctly. The magnetic tape subroutines are also fairly complex and these also appear to work. The main store assembler is

not written in a way which permits proper syntax checking and has not been exhaustively tested. It almost certainly contains significant bugs.

Some sample programs

In the sample programs the file extension `.edsc` has been used to distinguish EDSAC 2 programs that can be run. In some cases the same root with extension `.txt` is an annotated version.

Floating Arithmetic

The program `series.edsc` evaluates the four series

$$\begin{aligned}\sum_{n=1}^{\infty} \frac{1}{n^2} &= \frac{\pi^2}{6} \\ \sum_{n=0}^{\infty} \frac{1}{(2n+1)^2} &= \frac{\pi^2}{8} \\ \sum_{n=1}^{\infty} \frac{(-1)^{(n+1)}}{n^2} &= \frac{\pi^2}{12} \\ \sum_{n=1}^{\infty} \frac{1}{n^4} &= \frac{\pi^4}{90}\end{aligned}$$

Note the geeky technique for obtaining the alternating signs by constructing add and subtract orders and then obeying these! The value of $\pi/2$ is obtained from the reserved store.

The program `linear_eqns.edsc` solves a set of 3 linear equations using the built-in subroutine 59f9 and checks the result by substituting the answer. This provides a more rigorous check on the floating point operations than the previous example.

Square Roots

The program `sqrt1000.edsc` computes $\sqrt{2}$, $\sqrt{3}$ and $\sqrt{5}$ to 1000 figures. This is a program that has survived from the time the machine existed and is a nice demonstration which anyone can appreciate of a calculation that looks very impressive. It is probably best to turn off the debug output while running this program as it slows the emulator. The algorithm, which was suggested by R.L.Parker, relies on the following identity

$$\sqrt{n} = x[1 - (n - x^2)/n]^{-1/2}$$

which is evaluated as a binomial series

$$x[1 - \epsilon]^{-1/2} = x + x\frac{1}{2}\frac{\epsilon}{1!} + x\frac{1.3}{4}\frac{\epsilon^2}{2!} + x\frac{1.3.5}{8}\frac{\epsilon^3}{3!} + \dots$$

The trick is to construct a number x such that $\epsilon = (n - x^2)/n$ is a binary fraction whose bits can be fitted into the word length of the arithmetic unit. We start with an approximate square root $s \simeq \sqrt{n}$, compute s/n , truncate the last few bits and multiply by n . If this is used as x then $(n - x^2)$ will be a small binary fraction, and it is clearly exactly divisible by n . It can be scaled by a power of 2 to fit into a single word.

The point of this is that the computation of the next term in the binomial expansion from the current one involves multiplications and divisions by single length integers and by ϵ which is constructed to be single length. The terms and sum of the series are stored as digits $d_0, d_1, d_2 \dots$ in base $b = 2^{39}$; a number f can be written

$$f = d_0 + d_1b^{-1} + d_2b^{-2} + d_3b^{-3} + \dots$$

and multiplication or division of a number in this format by a single length number are straightforward. The only multiple length operation required is the addition of the term to the sum and the implementation of the carry.

Two python programs doing the same calculation in ways that closely resemble the EDSAC code are included. `sqrt1.py` illustrates the method and is the easier to understand; it represents a fraction as a list of strings of 10 characters (the ‘digits’ in base 10^{10}). `sqrt2.py` is closer to the EDSAC code and uses names resembling those in the machine code program.

The number e to 1000 figures

The program `e1000.edsc` computes the mathematical constant e to 1000 figures using the usual series. The multi-length arithmetic involved is similar to the square root example but simpler. A python program doing a similar calculation is included.

Hello World

The program `hello_world.edsc` displays the text “HELLO WORLD” on the (simulated) cathode ray tube. The purpose of the program is to draw attention to this hardware feature.

Test Orders

The program `test_orders.edsc` does no useful calculation but checks the consistency of the emulation of the order code. Each of the 100 orders is

obeyed at least once and the result checked. In a few cases, input and output for example, it is meaningless check the consistency of the result but such orders are implicitly tested in the code in the reserved store. The main use of this test program is to ensure that any changes made in the emulator do not have unanticipated side effects. If no errors are found the machine stop on a 101f0 instruction so that no address neons are illuminated. Inconsistencies are flagged by stop instructions with other addresses. The parameter `p10` set at the start may be changed to any integer value between 1 and 99 and the emulator should still pass the test. The annotated version of the program explains what is happening.

Bessel functions and library subroutines

A number of useful subroutines were kept on one of the magnetic tapes and could be inserted into a user's program. By chance one has survived: for the calculation of Bessel functions. This is believed to have been written by H.P.F. Swinnerton-Dyer and uses a method based on recurrence. There are two entry points computing $J_n(x)$ and $e^{-x}I_n(x)$; entry at the first line of the code produces the I function and at the second the J function. On entry x is in the accumulator and t is set to n . The result is placed in the accumulator.

The program `bessel_fn.edsc` prints a small table and includes the library code explicitly. Note that relative addresses are used so that the code can be placed anywhere in the user's program. The default magnetic tape provided includes this code in block 10 so that it is possible to use it as originally intended. This is done in the program `test_library.edsc` which does the same calculation as the previous program but loads the library from magnetic tape using the directive `10/` . Note that the external assembler does not support loading libraries from tape and so the one in the reserved store must be used.

Another small library program for producing pseudo-random numbers can be tried in the program `random.edsc`. It uses a multiplicative congruence method and the quality of the numbers is suspect since the Marsaglia effect had not been discovered at the time.