LINUX VOICE TUTORIAL

ALAN TURING: AND THE MANCHESTER MARK I

JULIET KEMP

Program the post-war machine used for early computer music, chess and proto-artificial intelligence.

WHY DO THIS?
- Take a trip back to the 1940s
- Search for large prime numbers (very slowly)
- Use the logic that first inspired the Turing Test

Alan Turing's work at GCHQ with Colossus during WWII is well-known, as of course is the Turing Test, but those were far from his only involvements with early computing developments. In the late 1940s he was working on designing a stored-program computer (Colossus couldn't store programs, and in any case was still very secret), and in 1948 moved to Manchester where the Manchester Baby and Manchester Mark I were being developed.

The Manchester Baby (aka the Manchester Small Scale Experimental Machine) wasn't a full general-purpose computer, but a small-scale test of Williams tube memory (cathode ray memory, using the charge well created by drawing a dot or dash on the tube). However, it is considered to be the world's first stored-program computer, running its first program on 21 June 1948, when it found the highest proper divisor of 2^18 (262,144), and took 52 minutes to run. Only two more programs were written for it: an amended version of this, and a program written by Turing to carry out long division.

The Baby was condensed, even primitive: only 32 words of memory and 8 hardware instructions, covering only subtraction and negation (using these, addition can be implemented in software, as -(-x-y) = x+y). It had no paper-tape reader, so programs had to be entered painstakingly, a bit at a time, with a 32-switch input device.

The Manchester Mark I

Once the Baby was successful, the team (Frederic C Williams, Tom Kilburn and Geoff Tootill, who had designed the Baby, together with research students DBG Edwards and GE Thomas) started work on the Manchester Mark I (also known as the Manchester Automatic Digital Machine, or MADM). The Mark I first ran in April 1949, with a program written to look for Mersenne primes. Max Newman wrote this version, but Turing later wrote an improved version known as the Mersenne Express.

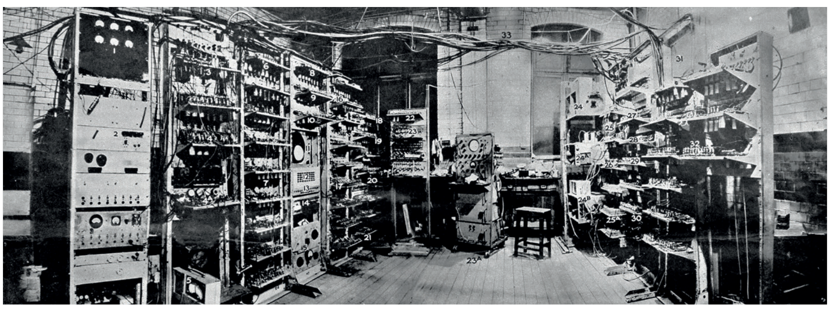
One of Turing's projects while working on the Mark I was to write code to investigate the Riemann hypothesis, which has to do with the distribution of prime numbers.

The Mark I had 40-bit words (longer than the Baby's 32 bits), and inspired by the success of the Baby, its main storage was Williams tubes. Initially, it had two Williams tubes each capable of holding two 'pages' of 32 words (so four 'pages', or 128 words, in total), which was later increased to eight pages. It also had a magnetic drum as backup storage, which contained an extra 32 pages (later, 128 pages). Initially, to transfer data from the drum to the Williams tubes, the machine had to be stopped and the transfer initiated manually, but in the final version, this could be done as part of a program.

The drum itself consisted of a series of parallel magnetic tracks, which each held two pages and had its own read/write head, which read and wrote as the drum revolved. (A little reminiscent of a modern magnetic hard disk.) Latency depended on the drum speed, which was synchronised with the main processor clock.

The most significant aspect of the Mark I, though, was that it introduced index registers. An index register holds a memory offset, which is added to an instruction to create a full memory address. Effectively, it alters the instruction as the program goes along. It is useful for very rapidly stepping

Panoramic shot of the Original Baby (copyright University of Manchester).



through memory addresses, such as to access an array sequentially, or to handle looping – the index register allows you to add one (or two or…) to the memory location to access a new location each time. Index registers are very commonly used on modern computers.

Turing and code

The Mark I had no assembly language; programmers had to write their programs in binary form, encoded as a series of five-bit characters. The programmers' handbook for the Mark II is available online at the Turing Archive, which illustrates the instruction conventions.

Each program instruction had 20 bits, of which 10 bits held the instruction code and 10 bits the address. The initial instruction set had 26 instructions (later 30, when the magnetic drum transfer instructions were added). Initially, just as with the Baby, instructions had to be keyed in, but the Final version had a teleprinter with five-hole paper tape reader and punch. Turing created a base-32 encoding scheme, which meant that programs and data could be written to and read from the paper tape. This was largely based on the existing ITA2 five-bit teleprinter code, which maps each of the 32 binary values in a five-bit system to a character. One of the characters Turing changed was binary zero (00000), which he wrote as **/** and which was very common in programs, and 01000, which he wrote as **@**. Each 40-bit word was therefore represented as eight five-bit characters, so could be written (eg) ABC//F@G. Without an assembly language, programmers had to produce their programs in this format, translating binary instructions to ITA2, and were encouraged to memorise the ITA2 table. The Mark I, like the Baby, also wrote its storage right-to-left, rather than left-to-right, so decimal one was 10000 rather than 00001 as would be expected today. Negative numbers were represented with two's complement (so the value of the most significant bit indicates sign: 0 for positive and 1 for negative).

An early user suggested that the frequently seen **//////** in early programs was an unconscious reflection of Manchester's wet weather – reminiscent of rain seen through a dirty window.

Simulating the Manchester Baby

There's a great Java simulator of the Manchester Baby available online at **www.davidsharp.com/baby**. It has a photo-realistic GUI so you can press the typewriter keys and flick various switches to set Williams tube bits just like the original team.

The red round typewriter keys each set a single bit of a particular line. They run top–bottom and left–right; so the top-left key sets bit 0, and the bottom-right would adjust bit 39, but only 0–31 are connected.

The write/erase switch at bottom-right sets whether the typewriter sets the bit as 1 or 0. (The KSC switch at the bottom clears the store.)

Williams and Kilburn with the original machine. (Copyright University of Manchester)

The switches (labelled 1, 2, 4…) underneath the typewriter choose the line to be edited, via binary coding. Line 0 is at the top of the screen; set all the switches up to access it, then flick switch 1 down for the next line down, switch 1 up and switch 2 down for the next after that, and so on. This is how you enter a program, a line at a time, into the store: pick the line and set each bit with the typewriter keys.

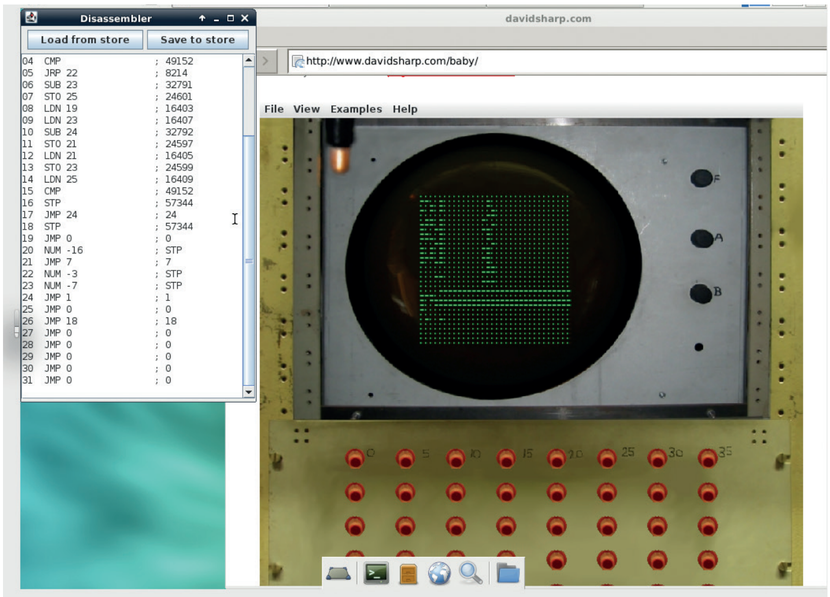
The KLC / KSC / KAC buttons at the bottom clear the current action line, the store, and the accumulator, respectively. The C, A, Sc red buttons at the bottom let allow you to look at the control, the accumulator, or the store, respectively. Flicking the CS switch to Run will run through all the stored program lines, until a **Stop** instruction is reached. To clear the **Stop** light, hit the KC button. This will also execute a single line at a time.

"The most significant aspect of the Mark I was that it introduced index registers."

To get started, try out one of the sample programs from the menu. **primegen.asc** generates primes, and stores them in lines 21 and 22. Check out the View > Disassembler menu to see a translation of the dots on the screen. Note that line 0 is at the top of the screen, and that numbers are stored least-significant-digit-to-right, so **-.-...** on the screen is 5 in decimal. To run the program, clear everything first, load it into the store from the menu, then flick the CS switch to run. Once the red stop light goes on, you have a prime stored in lines 21 and 22. Hit KC to clear the stop light, then flick CS up and down again to run again until the next stop and the next prime.

Next, we're going to transcribe the first ever program into the simulator, and run that. There's a useful online guide to the Baby from the Computer Conservation Society. This includes the structure of an instruction:

Line #		Function	
0 1 2 3 4	5 .. 12	13 14 15	16 .. 31



Here's the Baby simulator with first (buggy!) version of our program loaded up.

Only the first five bits and bits 13–15 are used at all. The line number is the line of the store to which the function is to be applied. The instruction set looks like this (CI is the Control Instruction, and A is the accumulator):

Function	Binary (LSD right)	1948 mnemonic	Modern mnemonic	Description
0	000	s,C	JMP	Copy content of store line s to CI
1	100	c+s,C	JRP	Add content of store line s to CI
2	010	-s,A	LDN	Copy content of store line s, negated, to A
3	110	a,s	STO	Copy content of A to store line s
4	001	a-s,A	SUB	Subtract content of store line s from A
5	101			As 4.
6	011	Test	CMP	Skip next instruction if content of A is negative.
7	111	Stop	STOP	Stop machine and light Stop bulb.

So, here's that first program, based on the reconstruction by Geoff Toothill and Tom Kilburn from Toothill's lab book notes (**www.cs.man.ac.uk/CCS/res/res20.htm#e**). It finds the highest factor of a given number (**a**), by starting with **b = a - 1**, and testing that and every smaller number until it succeeds. Early test cases included a = 19, a = 31 (both primes), a = 3141 (final b = 1047), a = 4537 (final b = 349), and the full trial case of 2^{18}. The final b value for that was 2^{17}, and finding it took 52 minutes.

In the Instruction column, I've shown the notebook transcription with the 1948 handbook mnemonic in brackets. The Assembly encoding is a 'fake', in the sense that it will work with the simulator but no such thing existed for the original programmers. They had to work with binary, and I've given that in the binary column (least significant digit to right, as already mentioned, and **0..0** or **1..1** means fill in the rest of the line with 0s or 1s). Finally there's a description of what each instruction does.

An overview of the program stages is below. The lines in italics are lines from the lab book notes, which I altered to get the program to run, so leave them out when typing it in.

Line	Instruction	Assembly encoding	Binary encoding	Description
0	-	JMP 0	0..0	Empty line apparently needed by simulator, not present in notebook
1	-18, C (-18, A)	LDN 18	01001 0000 0000 010 0..0	Copy empty line to accumulator to clear it
2	-19, C (-19, A)	LDN 19	11001 0000 0000 010 0..0	Load +a into accumulator
3	Sub 20 (a-20, A)	SUB 20	00101 0000 0000 001 0..0	Trial subtraction: a - current b
4	Test	CMP	00000 0000 0000 011 0..0	is difference negative?
5	Add 21, Cl (c+21, Cl)	JRP 21	10101 0000 0000 100 0..0	Still positive. Jump back two lines (by adding -3 to the current instruction line)
6	Sub 22 (a-22, A)	SUB 22	01101 0000 0000 001 0..0	Difference is negative, so we subtracted too many. Add back current b.
7	c, 24 (a,24)	STO 24	00011 0000 0000 110 0..0	Store remainder
8	-	LDN 18	01001 0000 0000 010 0..0	Clear accumulator again using empty line. This wasn't in the original notes but seems to be needed. (Note: line numbers from here do not match notebook.)
9	-22, C (-22, A)	LDN 22	01101 0000 0000 010 0..0	Load current b.
10	Sub 23 (a-23, A)	SUB 23	11101 0000 0000 001 0..0	Create next b = current b - 1.
11	c, 20 (a, 20)	STO 20	00101 0000 0000 110 0..0	Store next b.
12	-20, C (-20, A)	LDN 20	00101 0000 0000 010 0..0	Load negative next b.
13	c, 22 (a-22, A)	STO 22	01101 0000 0000 110 0..0	Store negative next b.
14	-24, C (-24, A)	LDN 24	00011 0000 0000 010 0..0	Load negative of remainder (see line 7).
15	Test	CMP	00000 0000 0000 011 0..0	Is remainder negative? If not, it must be zero.
16	-	STP	00000 0000 0000 111 0..0	Remainder is zero, so stop. I found this line 16 worked where the one below didn't.
16.5	25, CI (25, C)	JMP 25	10011 0000 0000 000 0..0	Original line in notebook which I couldn't make work; remainder is zero, so jump to line [16]
17	23, CI	JMP 23	11101 0000 0000 000 0..0	Remainder is negative, so jump back to line 1 (number in location 23). In notebook this was line 2, but that left the accumulator un-zeroed and caused errors.
17.5	Stop	STP	00000 0000 0000 111 0..0	Original line 17. This would be line 18 given the line number errors, but with the replacement line 16, it's not needed.
18	init	leave blank	0..0	0 (blank)
19	init	NUM -3141	11011 1011 1001 1..1	-a (value to be tested); here -3141

```
20          init        NUM 3140  00100 0100 0110 0..0  initial
b (a-1); here 3140
21          init        NUM -3      10111 1..1 -3
22          init        NUM -314000111 1011 1001 1..1
-(initial b); here -3140
23          init        NUM 1       10000 0..0 1
24          init        leave blank0..0          0 (blank)
25          init        NUM 16      00001 0..0 16 -- in notebook
for use in line 16.5
```

Negative numbers are stored as two's complement (to produce this yourself, write the number out in binary, swap all the 1s for 0s and 0s for 1s, and add 1). I've added lines 0 and 8, affecting line numbering for lines 8–17, and have altered lines 16–17.

The program works like so:

Lines 0–2: Clear the accumulator, and load **a** (the value whose highest divisor we are trying to find).

Lines 3–5: Subtract **b** repeatedly until you reach a negative number.

Lines 6–7: The negative number means we've gone too far, so add **b** back again once. This means that the accumulator now contains whatever is left over (the remainder) when you divide **a** by **b**. Store that remainder.

Lines 8–13: Clear the accumulator again and get the next **b** value, by subtracting one from the current **b**. Store that as both positive and negative values for use in the next loop.

Lines 14–17: Load the remainder back again, and test it. If it's zero, then we've found a divisor, and the program stops. If not, we loop back to the beginning.

Lines 18–25: Data values, both fixed and altered as the program runs.

To input this, you can input the binary directly with the switches, to really get a feel for how it was in 1948! Alternatively, you can use the Assembly encoding, by saving it in a file called **babyfactor.asm** with line numbers, as shown:

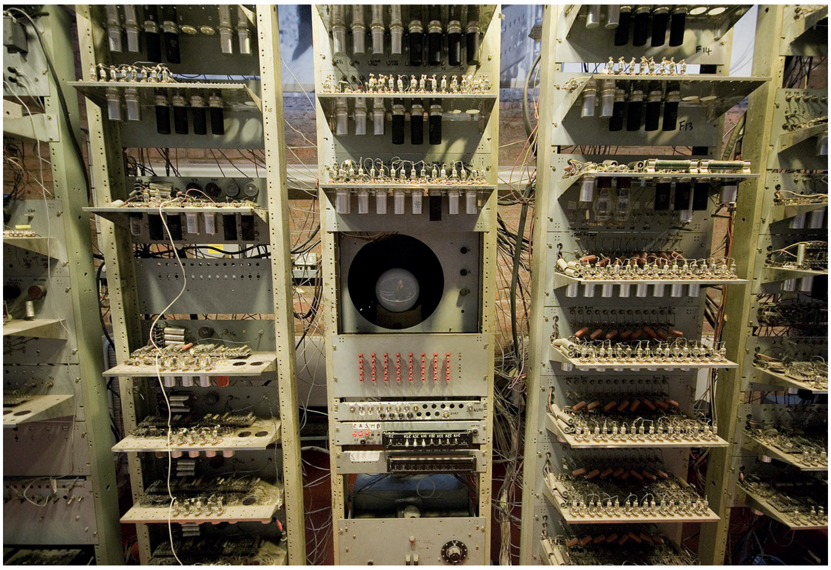
```
25
0  JMP 0
1  LDN 18
...
```

Load this from the file menu. The number at the top is the number of lines of code in the file and is necessary. Arguably it's cheating a bit, but bugfixing is much easier using assembly!

To run it, flick the 'Run' switch at the bottom to run the whole thing until the lightbulb lights (this may take a while for the given value of **a**). Once the lightbulb lights, read the **b** value from line 21 (use the Disassembler for ease!). The highest divisor is **b+1** (because **b** is decremented ready for the next time before the **STOP** line is reached). For the value of **a** given here (3141) this should be 1047.

Sometimes I found I had to hit the KC switch once before the Run switch. The simulator is a little temperamental; if you have problems, reload the simulator and start over.

For debugging or to watch the program working, you can step through it one line at a time with the KC button. (In which case I recommend using smaller **a** and **b** values.) You could also check out Toothill's article to see how they improved the program over time and make your own edits.



Close-up of the working replica at the Manchester Museum of Science and Industry. Image CC-BY-SA ,Parrot of Doom.

Further developments

The Manchester Mark I was the basis for the Ferranti Mark I, the first commercially available general-purpose electric computer. It just beat out UNIVAC, which was handed to the US Census Bureau on 31 March 1951; the first Ferranti Mark I was delivered to the University of Manchester in February 1951. The oldest recorded computer music was played on a Ferranti, using its **hoot** command, and is available from **www.digital60.org/media/mark_one_digital_music**.

One of the oldest computer games, a chess-playing program, was also written for the Ferranti by Dr Prinz in 1951, although it could handle only mate-in-two chess problems, not whole games. Turing had also been experimenting with chess computer programs, and wrote a program for a non-existent computer between 1948 and 1950. In 1952 he tried to implement it on the Ferranti, but the computer wasn't sufficiently powerful. Instead, Turing simulated it by hand, taking around 30 minutes per move. The game was recorded, but the computer lost.

After the Manchester Mark I first ran, neurosurgeon Sir Geoffrey Jefferson argued in 1949 that no machine could ever feel emotion or truly 'think'. This undoubtedly had an effect on Turing's thinking about machine intelligence. Turing explicitly disagreed with Jefferson, arguing that while this might be true now, it was not necessarily true for ever. The debate is, of course, still live today.

Juliet Kemp is a programming polyglot, and the author of O'Reilly's *Linux System Administration Recipes*.