LINUX VOICE TUTORIAL

GRACE HOPPER AND UNIVAC: BEFORE THERE WAS COBOL

JULIET KEMP

In the days before cheap silicon chips, valves ruled the roost – and it took a special kind of brain to handle these magnificent beasts.

After Babbage and the (never actually built) Analytical Engine in the 19th century, computer development languished for a while. During the first half of the 20th century, various analog computers were developed, but these solved specific problems rather than being programmable. In 1936, Alan Turing developed the idea of the 'Universal Machine', and the outbreak of World War II shortly afterwards was a driver for work on developing these machines, including UNIVAC, famously worked on by Grace Hopper.

Grace Hopper, born in New York in 1906, was an associate professor of mathematics at Vassar when WWII broke out. Volunteering for the US Navy Reserve, she was assigned to the Bureau of Ships Computation Project, where she worked on the Harvard Mark I project (a calculating machine used in the war effort), from 1944–9, co-authoring several papers.

In 1949, she moved to the Eckert-Mauchly Computer Corporation (later acquired by Remington Rand, and later still by Unisys), and joined the UNIVAC team. UNIVAC, which first ran in 1951, was the second commercially available computer in the US, and the first designed for business and admin rather than for scientific use. That meant that it was intended to execute many simple calculations rapidly, rather than performing fewer complex calculations. Punch-card calculating machines already existed, but crucially, UNIVAC was programmable. The first customers included the US Census Bureau and the US Air Force (who had the first on-site installation, in 1952). In 1952, as a promotional stunt, they worked with CBS to have UNIVAC predict the result of the 1952 US presidential election. It correctly (and quickly!) predicted an Eisenhower win, beating out the pollsters who had gone for Stevenson. So let's take a look at what it was and what it was doing.

Grace Hopper, who studied mathematics and physics at Harvard and Vasser universities, at a later UNIVAC in 1960. By Unknown (Smithsonian Institution) (Flickr: Grace Hopper and UNIVAC)

UNIVAC: mercury and diodes

UNIVAC weighed about 13 tons, and needed a whole garage-sized room to itself, with a complicated water cooling system and fans. It had 10 UNISERVO tape drives for input and output, 5,200 electron (vacuum) tubes, 18,000 diodes, and a 1,000 word memory (more on that in a moment); it required about 125kW of power to run. (A modern laptop uses around 0.03kW. It also required a lot of maintenance; replacing diodes, contacts and tubes, not to mention keeping the cooling systems running.

UNIVAC's memory and operational registers were both based on mercury delay lines. The main memory consisted of seven 'long tanks', each containing eighteen ten-word channels. Each channel was a column of mercury with quartz crystals at each end, and held 910 bits (840 bits for the words and 70 for the spaces between each word). The main clock (at 2.25MHz) was in sync with the carrier wave of the column (11.25MHz) and acted as the timer for all UNIVAC operations.

To store data in a channel, the sending crystal (at one end of the channel) was vibrated with the data bits (ones and zeros) of the word. The rate was controlled by the main clock, then the signal was mixed with the carrier wave. The whole signal would move through the column to the receiving crystal, where a bunch of circuitry picked it up, amplified it, analysed it, and sent it back to the sending crystal for another trip through the mercury. So the data was constantly rotating through the mercury, which meant that you could only access a word when it popped out at the receiving crystal end. The average access time for a word was 222 microseconds, so a fair amount of UNIVAC's time involved waiting for word access, with obvious practical programming implications.

You may have noticed that seven lots of 18 channels gives 126 channels of 10 words each; so why only 1,000 words of memory? The remaining 26

channels were used for input and output buffering, for the register, and for the vitally important mercury temperature control. The mercury had to be at an exact operating temperature for the correct transit time and to avoid bit creep; from a cold start, it could be up to half an hour before the tanks were able to hold memory.

Control and computation operations were also run via mercury delay lines, each tank working with a single 12-character word. This made access much quicker, and they also had distribution delay lines to allow multi-bit access to characters. There were four types of register:

- Four Input/Output Synchronizers, used for the 60 word read/write buffers.
- Three Control registers, used for controlling program instruction flow.
- One two-word register (rV) used as a holding area during a two-word move.
- Several of these registers were duplicated, then compared bit-by-bit, to increase accuracy.

Finally, it had an operator's console and an oscilloscope connected. The console had switches that allowed any of the memory locations to be displayed and monitored on the oscilloscope. A typewriter and printer were also connected for output.

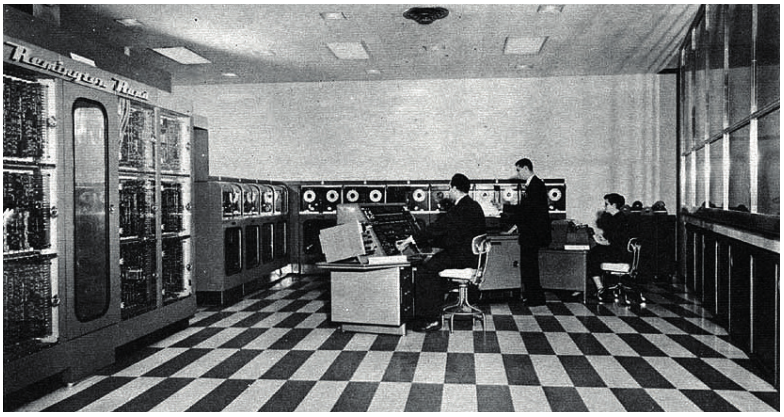
Programming UNIVAC

UNIVAC had quite a big instruction set, which covered transferring the contents of memory into registers, moving the contents of registers around, performing operations, jumping to specific memory addresses, shifting contents of registers a given number of digits, and controlling input/output. The full list (with explanations) is available at **https://wiki.cc.gatech.edu/folklore/index.php/UNIVAC_I_Instruction_Set**. Unfortunately there's no Linux-compatible emulator (see boxout), but here is a small example, with comments:

```
L00 101
   loads contents of memory register 101 into register A.
A00 102
   loads contents of register 102 into register X, adds X to A.
C00 103
   stores contents of register A into register 103, clears A.
P00 103
   print contents of register 103 on the console printer.
```

If you read last month's article on Ada Lovelace and the Analytical Engine, this may look familiar. The instructions (L, A, P) are made up to 3 digits with zero padding. So if register 101 contains the value 2, and register 102 contains the value 3, this will add 2 to 3, store 5 in register 103, and output 5 to the console.

To run this program, it would have been typed onto a program tape as a series of numeric words ('translated' from the programmer's mnemonics given here). The tape (and any needed data tapes) would be latched onto the UNISERVOs, and the operator would manually set various options and begin the booting process from the console. The first 60 instructions would be read into the input buffer, then transferred into memory. The operator would then set the machine back into 'normal' mode, hit the Start Bar, and UNIVAC would begin executing the instructions from memory, beginning with the first block. So the programmer would have to make sure that from then on in, everything that the program needed to do (including reading in more instructions or data from tape) was referenced in the program itself. The operator's role would be limited (at least in theory!) to replacing tape reels as indicated by console messages, and rescuing any minor problems such as a dropped tape loop. Breakpoints could be set in the code (instruction **,**) to aid recovery from problems.



UNIVAC I at the Franklin Imstitute, Philadelphia.

Here's a longer example from the 1954 UNIVAC operating manual. The far-left number is the memory register that contains the instruction. Instructions were saved in memory in pairs, as shown, with the left-hand six-digit instruction run first, then the right-hand six-digit instruction. In this example, registers 100–999 contain a set of numbers, and the code adds them all together.

```
000  C00 099  C00 099
001  B00 099  A00 100
002  C00 099  B00 001
003  L00 007  Q00 006
004  A00 008  C00 001
005  000 000  U00 001
006  900 000  U00 001
007  B00 099  A00 999
008  000 000  000 001
```

Line by line, here's how that code works:

000 C 099 stores register A into memory and zeroes it; so repeating this twice zeros register 099.

001 B 099 loads register 099 into register A; A 100 loads register 100 into register Z, then adds it to register A.

002 C 099 stores register A (now containing A+Z) into register 099, and zeroes register A. B 001 loads the contents of register 001 into register A. The contents of register 001 are the program instructions in step 001; so we are preparing to alter the instructions themselves.

003 L 007 loads the contents of register 007 (see step 007 below) into both register L and register X. Q 006

Grace Hopper remains a source of quotable quotes, our favourite being: "It's easier to ask forgiveness than it it to get permission."

checks whether register L is equivalent to register A; if so, it jumps to register 006 (that is, step 006).
004 A 008 loads the contents of register 008 into register X, and adds it to register A. As register A currently holds the instruction from register 001, and register 008 holds (effectively) a single 1, this alters the instruction from register 001 to read B00 099 A00 101 instead of B00 099 A00 101. In other words, next time we run step 001 we'll add the contents of the next register in the list to our running total. C 001 dumps the contents of register A back into register 001, so we've edited the program on the fly.
005 The LHI is blank; the RHI (U 001) is an unconditional jump back to register 001, ready to add the next number in the list.
006: This simply stops the computer. (Remember from 003: we jump here if the program is finished.)
007 B 099 A 999. This instruction is never actually run. It is used in 003 to check against register A. If register A looks like this at step 003, then we have added the final number (in register 999) and our program is done. 003 will then jump to 006 and the program ends.
008 End of program.

Fundamentally, this is a for loop that sums each element in an array. But UNIVAC programmers had to physically rewrite the instruction inside the loop each time.

One apparently excellent emulator for UNIVAC does exist. It's by Peter Zilahy Ingerman and is described at **www.ingerman.org/niche.htm#UNIVAC**. Unfortunately it's written in Visual Basic 6 and only runs on Windows. The download link on that page doesn't work, but it can be obtained by contacting the author on the given email address. The code above should run on it, but as it's Windows we haven't been able to test it.

Grace Hopper created the first operational compiler, in 1952, while working on the UNIVAC project. Initially, no one believed her. "I had a running compiler and nobody would touch it," she said later. "They told me computers could only do arithmetic." In fact, the A-0 system was more like what we would today call a loader or a linker than a modern 'compiler'. For A-0, Hopper transferred all her subroutines to tape, each identified with a call number, so that UNIVAC could find it. She then wrote down the call numbers, and any arguments, and this was converted into machine code to be run directly. Effectively, A-0 allowed the programmer to reuse code and to write in a more human-readable way, and get the machine to do more of the work.

Programming with A-0

The next versions were A-1 and A-2, with A-2 the first compiler to be in more general use. I found a short paper from a 1954 MIT course, which Hopper also tutored. In it she describes A-2 as handling two types of subroutine: static ones (stored in memory or on tape, either from a general library or specific to the problem) and dynamic ones. Dynamic subroutines could be generated from a 'skeleton' stored subroutine and some specific parameters, or could be generated to handle data. A-2 had four phases of operation:

1. Expands/translates the provided code, adding in data such as call-numbers and operation numbers. (In later compilers this translated from 'code' into 'machine code'.)
2. Divides the result from phase 1 into segments which can be processed in a single storage load, and creates references to each subroutine.
3. Creates the jump instructions needed to complete the necessary jumps (for example between storage loads and subroutines) required by the result of phase. After this phase, you have a complete description of the program, but not a complete program that can be run sequentially.
4. Main compilation. All subroutines are read in and transformed as necessary from 'general' to 'specific' (so any parameters are included), all jumps, reads, and writes are included, and a complete program is generated which can now be run as-is.

(If you check out the PDF course notes in the resources, there are some very cute line drawings illustrating the four phases.)

FLOW-MATIC & English-language programming

A couple of iterations later, Hopper and her team produced FLOW-MATIC, which was the first English-language-like data processing language. (Meanwhile, FORTRAN was completed at IBM in 1957, and is generally agreed to be the first complete compiler.)

Here's a quick sample of FLOW-MATIC code, taken from the FLOW-MATIC product brochure).

```
(0) INPUT INVENTORY FILE-A PRICE FILE-B ; OUTPUT
PRICED-INV FILE-C UNPRICED-INV
    FILE-D ; HSP D .
```

```
(1)  COMPARE PRODUCT-NO (A) WITH PRODUCT-NO (B) ; IF
GREATER GO TO OPERATION 10 ;
   IF EQUAL GO TO OPERATION 5 ; OTHERWISE GO TO
OPERATION 2 .
(2)  TRANSFER A TO D .
(3)  WRITE-ITEM D .
(4)  JUMP TO OPERATION 8 .
(5)  TRANSFER A TO C .
(6)  MOVE UNIT-PRICE (B) TO UNIT-PRICE (C) .
(7)  WRITE-ITEM C .
(8)  READ-ITEM A ; IF END OF DATA GO TO OPERATION 14 .
(9)  JUMP TO OPERATION 1 .
(10)  READ-ITEM B ; IF END OF DATA GO TO OPERATION 12 .
(11)  JUMP TO OPERATION 1 .
(12)  SET OPERATION 9 TO GO TO OPERATION 2 .
(13)  JUMP TO OPERATION 2 .
(14)  TEST PRODUCT-NO (B) AGAINST ZZZZZZZZZZZZ ; IF EQUAL
GO TO OPERATION 16 ;
   OTHERWISE GO TO OPERATION 15 .
(15)  REWIND B .
(16)  CLOSE-OUT FILES C ; D .
(17)  STOP . (END)
```

The **PRODUCT-NO** and **UNIT-PRICE** fields would have been defined separately, in the **DIRECTORY** section of the program. This is just the executable part. Let's step through it:

(0) Load in two input files (A is inventory, B is price), and set two output files (C is priced inventory, D is unpriced).

(1) The key part: this compares the current product number from file A with that from file B:

If they match, then product 1 has a matching price, and we go to section (5)–(9).

If A is greater, we go to section (10)–(13).

If B is greater, we go to section (2)–(4).

(2)–(4) this implies that we have an unpriced product (product 1, for example, exists on list A but on list B the lowest number is product 2). We write it out on the unpriced file D. We then jump to (8), read in the next item A and return to (1).

(5)–(9) Items A and B match; the product has a price. We write it, together with its price, on file C. Then we read in the next item A and go back to (1).

(10)–(13) Item A is greater than item B. Read in the next item B, if there is an item B, and go back to (1). Note that the result of (5)-(9) (a matching pair) will be to read in the next A product but not the next B product, so this balances that out. If we have run out of B data, we rewrite (9) so that all the rest of the products go directly to the unpriced output file.

(14)-(16): close the output files and/or rewind input file B; stop the program.

So this would generate a list of priced items with their prices, and a list of unpriced items. As you can see, FLOW-MATIC was squarely aimed at the business market.

When is a bug not a bug?

Famously, Grace Hopper popularised the term "debugging" about computer programs, after an error while working on the Mark II in 1947 was tracked down to an actual bug (a moth) stuck in a relay. The term "bug" had been used before in engineering, but Hopper brought it into popularity.

A UNIVAC at US Steel in Indiana, on the other hand, had a bug that was in fact a fish; its cooling system, which used water from Lake Michigan, got its intake blocked by a fish and thereby overheated.

COBOL and later

After FLOW-MATIC came COBOL, which Hopper and her team designed from 1959 onwards. COBOL is still in use today, with the 2002 update including OO features, and the compiler GNU Cobol (formerly OpenCOBOL) is available for Linux, with plenty of online resources available. COBOL was intended to be comprehensible by non-programmers, hence its use of English-like syntax and structure. Modern COBOL is still recognisably the same language, and indeed recognisably inherits from FLOW-MATIC. (The first COBOL compiler was itself written in FLOW-MATIC, and was the first compiler to be written in a high-level language.)

"Grace Hopper created the first operational compiler while working on the UNIVAC project."

Grace Hopper moved back into the Navy in the late 1960s. She was on active duty for several years beyond mandatory retirement with special approval of Congress, eventually retiring in 1986, at the age of 79, as a Rear Admiral. She continued to lecture widely on early computing and other aspects of user-friendly computing until she died in 1992.

More resources

My great thanks to Allan Reiter, whose page at **http://univac1.0catch.com** is invaluable for technical details of UNIVAC operation. Check it out for much more detailed info and plenty of photos and diagrams.

There are some other wonderful UNIVAC resources available online:

- The 1951 'Introduction to UNIVAC' leaflet.
- Remington Rand UNIVAC advertising film from 1950-2.
- Notes from the 1954 MIT special program on Digital Computers (see A-0 section above).
- Bitsavers have a whole bunch of documents from the early days of UNIVAC. These include operating manuals, programming references, and the course materials for an Advanced Programming Course.
- The FLOW-MATIC brochure from 1957 (includes the FLOW-MATIC sample code above.

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