The Stored Program Computer and Some Early Computational Abstractions

Andru Luvisi

Sonoma State University

ABSTRACT

The stored program computer held instructions and data in the same memory. This enabled programs to manipulate other programs (or even themselves!) and made it possible to decouple the external representation of a program from the actual instructions executed by the machine. This in turn allowed new forms of abstraction to be created and used by implementing them in software. This talk will discuss the creation of the stored program computer and some early software and hardware ideas related to computational abstraction.

The latest version of this handout can be found at http://www.sonoma.edu/users/l/luvisi/storedprogram/

1. What I Mean by Abstraction

In this context, by "abstraction" I refer to tools and concepts that allow a programmer to disregard details that are unrelated to the essential nature of the problem being solved.

As Steele and Sussman[17] argue, a good abstraction should be more than an abbreviation for typing. It should be an idea that has meaning independent of its implementation:

> One decomposition strategy is the packaging of common patterns of the use of a language. For example, in Algol a *for* loop captures a common pattern of *if* and *goto* statements. Packages of common patterns are not necessarily merely abbreviations to save typing. While a simple abbreviation has little abstraction power because a user must know what the abbreviation expands into, a good package encapsulates a higher level concept which has meaning independent of its implementation. Once a package is constructed the programmer can use it directly, without regard for the details it contains, precisely because it corresponds to a single notion he uses in dealing with the programming problem.

2. On The Difficulty of Examining Familiar Concepts

Marshal McLuhan is said to have once observed that "Whoever discovered water, it wasn't a fish."[1] In this talk, I will be discussing some ideas that are so common in the world of computing as to be part of the water we swim in and the air we breath. This very familiarity creates multiple challenges because it can be difficult to appreciate the nature of a familiar idea if we have never been aware of any other ways that things have been or could be.

In an effort to make the significance of these ideas more visible, I will begin today's story just a little bit before the beginning to set some context.

3. Background and Immediate Predecessors

The Harvard Mark I[15] was a relay computer that interpreted instructions fed to it on paper tape. It was relatively easy to change the program being run by feeding a different tape into the machine, but it performed less than 10 additions per second.

The ENIAC[8] was capable of performing 5,000 additions per second, but it was programmed by manually connecting wires between arithmetic elements and data busses and between control units and control busses. Changing programs could take hours.

4. The Big Idea

In late 1943 and early 1944, the design of the ENIAC had been frozen so that construction could proceed and the machine could be finished as soon as possible to support the war effort. The team was, however, aware of the ENIAC's limitations, and was already considering ways to address them in their next design for a computer.

The two main issues were the time required to set the machine up to work on a new problem and varying requirements among the programs that might run on the machine. For example, some programs might require a large amount of intermediate storage (supplied by accumulators in the ENIAC) but little or no storage for lookup tables (supplied by banks of manual switches in the ENIAC) while other programs might require very little intermediate storage but large lookup tables. Similarly, some programs might have simple algorithms while others might have complex algorithms (set up on the ENIAC by manually connecting control units together with cables hooked up to common control busses).

How could they decide how much of the new machine should be dedicated to storage, how much to lookup tables, and how much to the program being run, when different programs could vary so wildly in their requirements?

[1] I have been unable to identify a primary reference for this quotation.

On January 29, 1944 J. Presper Eckert wrote a memo[7] explaining how information could be stored on spinning disks, using one large memory to hold intermediate results, lookup tables, and program information.

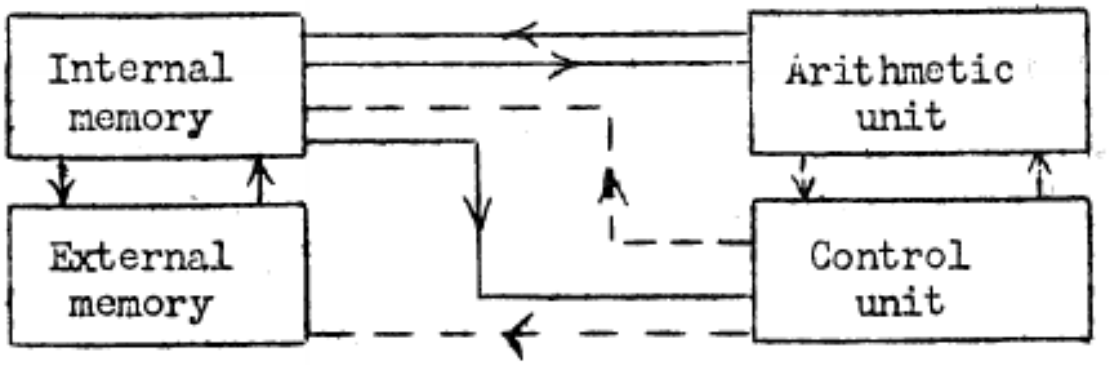
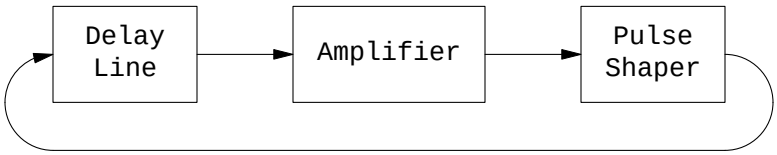


Diagram of The Stored Program Design, From a 1946 Lecture[9]

Later proposals switched to using ultrasonic delay lines as a storage device. Timed ultrasonic pulses, injected at one end of a tube of mercury, could be read out of the other end of the tube after a delay. Multiple pulses could be passing through one tube at the same time, with electronic circuits detecting the pulses coming out at one end and reinjecting them back into the other end. The presence or absence of a pulse at a specific moment would correspond to a "1" or a "0" being stored in a specific place in memory.



Delay Line Memory

A given word of memory would be stored in a series of pulses in the same tank, one after the other, with the least significant bit coming first. When a word was being transfered between parts of the machine, or added to another word, one bit was processed at a time in a serial fashion.

Using data as instructions to indicate the actions to perform would make it easier to change programs. It would also allow a programmer to think in terms of the actions that would be needed to solve a problem instead of thinking about wire connections.

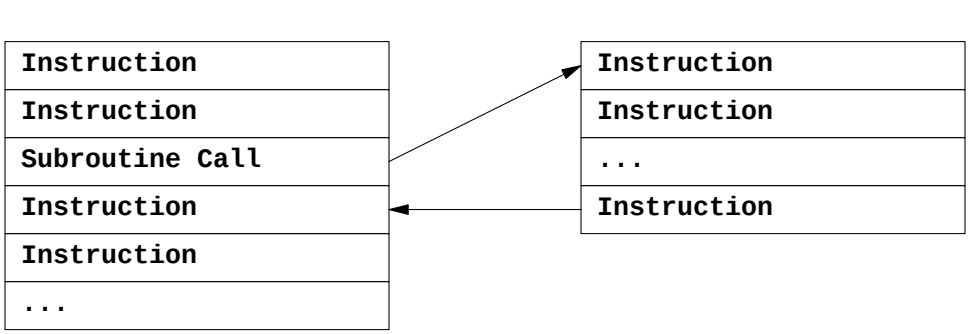
5. The von Neumann Report

John von Neumann joined the project in September of 1944 and wrote a first draft of a report on the proposed new machine[16]. Called the EDVAC (Electronic Discrete Variable Automatic Computer) the new machine

was to use mercury delay lines for memory and to hold both instructions and data in the same memory.

The report was never formally published, but it was widely shared on an informal basis, giving many people in the automatic computer community their first introduction to the stored program computer.

6. Subroutines, Flowcharts, and Early algorithms

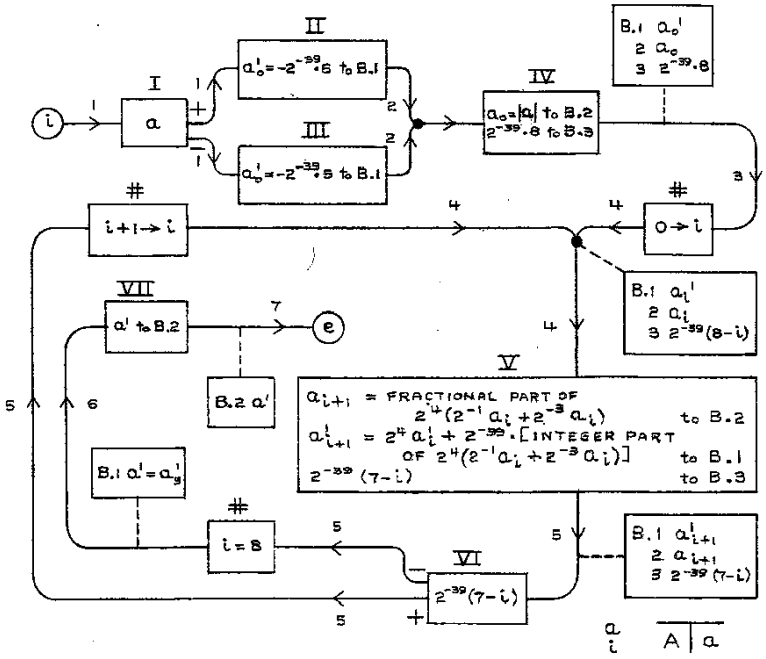


Subroutine Call

Even before the first stored program computers were operational, the need for some abstractions became apparent. John Mauchly mentions subroutines in one of his 1946 lectures[9].

Subroutines would allow a programmer to think about the overall action performed by a subroutine instead of the detailed instructions that would execute at run time.

While documenting proposed types of programs that could run on stored program computer, von Neumann and Goldstine created flowcharts as an abstraction to allow humans to visualize a program at a higher level than just a linear list of instructions[6, 10].



Flowchart of a Binary to Decimal Conversion Algorithm

Some of the first problems they tackled are still familiar today, including programs for sorting, arbitrary precision arithmetic, and binary/decimal conversion, to prove that it would be practical to create a binary computer and have humans interact with it using decimal notation.

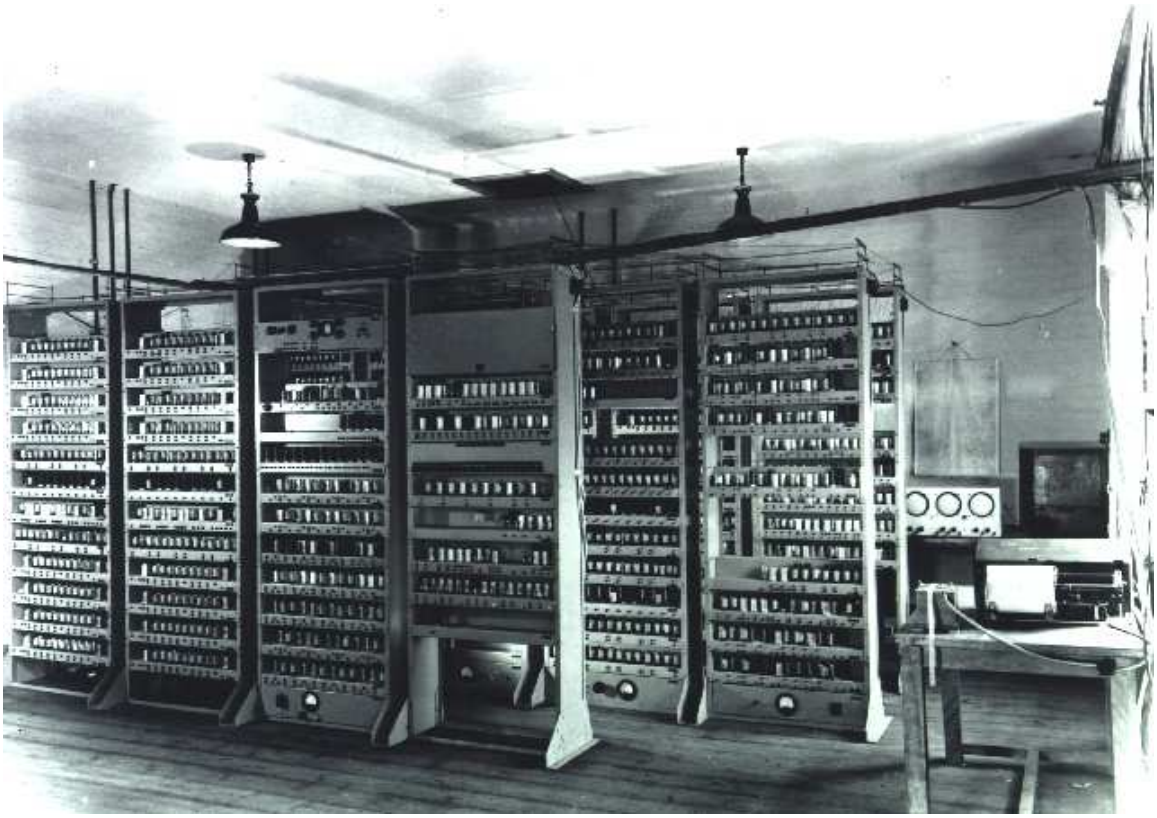
7. Work towards a Stored Program Computer

The ENIAC had proven the feasability of a large scale electronic computer, and the von Neumann report had revealed the team’s plans for the EDVAC to the world. Multiple institutions began working towards implementing a stored program computer and the literature of the time[11, 12, 1, 13, 4] took on a distinctly optimistic tone. There was no question of whether this new kind of device could be built, only when they would start working, and whose would be first.

8. The Manchester Baby

The Small Scale Experimental Machine, also known as The Manchester Baby, made its first successful run of a program on June 21, 1948. This was the first working demonstration of the stored program concept[2]. It was a prototype that demonstrated the viability of the stored program concept.

9. The EDSAC



The EDSAC

The EDSAC[20, 21, 25, 26, 27], (Electronic Delay Storage Automatic Computer) built in Cambridge, England by a team under the leadership of Maurice Wilkes, came online on May 6, 1949. It was the first practical stored program computer to become operational, and also the proving ground for many early advances in programming.

The EDSAC used many key ideas from the design of the EDVAC. It used ultrasonic delay memory made out of mercury tubes and it operated in a serial fashion.

The EDSAC used single address instructions, where five bits of each word identified the operation to be performed. The mapping between codes and instructions provided a simple single letter mnemonic device, "A" meaning "add," "S" meaning "subtract," and so on.

9.1. EDSAC Initial Orders

When first turned on, the EDSAC would load a statically defined set of "Initial Orders" into memory that were responsible for loading a program from paper tape and running it[18].

Because the Initial Orders were themselves a program capable of performing calculations and making decisions, they were able to perform arbitrary conversions between the format of programs punched on tape and the format of programs running in memory.

The first set of Initial Orders could read a letter indicating the instruction portion of a word and a decimal number indicating the address portion of the same word. The decimal number was converted into binary and the instruction described was created and entered into memory. They also provided for running the program once all of the instructions had been loaded in.

This implemented two very important kinds of abstraction. It allowed programmers to think of instructions in terms of their single letter names rather than their five bit binary values, and it allowed programmers to think of addresses and numbers in terms of their decimal representations rather than their binary representations.

```
 0      T   S        11     I  2 S        22     A  1 S
                                           23     L    L
 1      H  2 S       12     A  2 S        24     A    S
                     13     S  5 S
 2      T   S        14     E 21 S        25     T 31 S
 3      E  6 S
                     15     T  3 S

 4      P  1 S       16     V  1 S
 5      P  5 S                            26     A 25 S
 6      T   S                             27     A  4 S
 7      I   S        17     L  8 S        28     U 25 S
 8      A   S
 9      R 16 S                            29     S 31 S
                     18     A  2 S        30     G  6 S
                     19     T  1 S
                     20     E 11 S
10      T   L
                     21     R  4 S
```

First EDSAC Initial Orders

A second set of Initial Orders was created in September 1949 that added support for several "control combinations" that affected the loading process rather than entering instructions into the memory. One key feature was the ability to add an offset to an address while an instruction was being loaded into memory. This made it possible to write subroutines that could be loaded into anywhere in memory.

```
0     T   F        13    L   D        22    T 43 F       32    A 22 F
                                                           33    T 42 F
1     E 20 F       14    S 39 F
                                       T n F
2     P  1 F       15    E 17 F   or    E m F
3     U  2 F
                   16    S  7 F   23    A 22 F       34    I 40 D
4     A 39 F       17    A 35 F   24    A  2 F
5     R  4 F                      25    T 22 F
                                  26    E 34 F       35    A 40 D
6     V   F       18    T 20 F                      36    R 16 F
                   19    A   F   27    A 43 F       37    T 40 D
7     L  8 F                     28    E  8 F
                   20    H  8 F                      38    E  8 F
8     T   F
9     I  1 F                     29    A 42 F       39    P  5 D
                                                     40    P   D
                   A (24+b) F
10    A  1 F   or   E(16+b) F    30    A 40 F
11    S 39 F
12    G  4 F       21    A 40 F
                                  31    E 25 F
```

Second EDSAC Initial Orders

10. Subroutines

The EDSAC team defined two different kinds of subroutines, Open Subroutines, and Closed Subroutines. Open Subroutines were useful pieces of code in the subroutine library that a programmer could copy into new code, kind of like copy and paste programming. Closed subroutines were self contained pieces of code that could be called from, and could return to, code that was using them.

While Mauchly, von Neumann, and Goldstine had discussed the concept of subroutines as a design tool, they had not published any methods for implementing them as a technical reality. The EDSAC team, and David Wheeler in particular, solved two main problems, namely how to write a subroutine in such a way that it could be loaded into different places in memory, and how to manage the transfers of control into and out of the subroutine[18, 19, 26, 27].

10.1. Relocation

Relocation was implemented using a feature of the Initial Orders. The control combination "G K" would save the address of the next instruction into memory location 42, and any instruction that ended with the character θ would have the value from location 42 added to its address field.

For example, if a subroutine began with the control combination "G K," then within the subroutine the conditional jump instruction "E 0 θ" would become a jump to the beginning of the subroutine, wherever it was

loaded into memory, and the instruction "E 7 θ" would become a jump to the location 7 addresses after the beginning.

entry on the input tape	action during input
$G\quad K$	this control combination puts the number $n \times 2^{-15}$ in storage location 42
$A\ 1\ F$	this is placed in location n
$E\ \ 4\ \theta$	$n \times 2^{-15}$ is added and the result $E(n+4)\ F$ is placed in location $n+1$
$T\ 10\ F$ $S\ 10\ F$ $T\ 10\ F$	these are not modified but are placed in locations $n+2$, $n+3$ and $n+4$

Example of relocatable code

This allowed a programmer to think about the address of instructions relative to the beginning of the subroutine, instead of the absolute addresses into which the subroutine would be loaded.

10.2. Control Transfer

When using a subroutine, the calling code would need to load the address of the call into the accumulator and then jump to the beginning of the subroutine. The first instructions of the subroutine would then store the return address into a jump instruction at the end of the subroutine. The θ feature also made it easy for an instruction to contain its own address regardless of where in memory it had been loaded.

location	order	notes
		orders calling in the sub-routine
		accumulator assumed empty
n	$A\quad n\quad F$	this order adds itself into the accumulator
$n+1$	$G\quad m\quad F$	control to m since $C\ (Acc)$ is negative, as the numerical equivalent of the character A is negative
$n+2$		programme after sub-routine
		sub-routine
		accumulator contains $A\ n\ F$
m	$A\quad 3\quad F$	adds $U\ 2\ F$ to the accumulator forming $(U+A)(2+n)\ F$, numerically equal to the order $E\ n+2\ F$
$m+1$	$T\quad m+r\quad F$	transfers $E\ (n+2)\ F$ to the operational end of the sub-routine where it is used to return control to the main programme
...		accumulator empty
$m+r$	$(E\qquad F)$	becomes $E\ n+2\ F$ as a result of the operation of the order in $(m+1)$. When $(m+r)$ is reached in the sequence of operations of the sub-routine, control is switched to $(n+2)$

Example of calling a subroutine

11. Interpretive Routines

One special kind of subroutine was dubbed the Interpretive Routine[3, 5], and would interpret instructions for a virtual machine different than the physical machine actually being used. For example, there were floating point interpretive routines that would allow the programmer to write code for a computer that was very similar to the EDSAC, except the accumulator and the memory addresses each held a floating point number, and the arithmetical operations acted upon floating point numbers.

This allowed a programmer to solve a problem while thinking about operations in a machine specifically suited to the problem, instead of the actual instructions that would be executed to solve the problem.

The arithmetical 'orders'

$A\ m\ D$	Add $F(mD)$ to $F(A)$
$B\ m\ D$	Subtract $F(mD)$ from $F(A)$
$H\ m\ D$	Replace $F(R)$ by $F(mD)$
$V\ m\ D$	Add the product $F(R) \cdot F(mD)$ to $F(A)$
$N\ m\ D$	Subtract the product $F(R) \cdot F(mD)$ from $F(A)$
$D\ m\ D$	Replace $F(A)$ by $F(A)/F(mD)$

Sample instructions from a virtual machine implemented by an interpretive subroutine

12. B-Lines/Index Registers

A B-Line or B-Register, is what we now call an Index Register. It stores an address that can be added to the address in an instruction at execution time[14, 27].

This removed much of the need for self modifying code, and allowed a programmer to think of the program text as written, instead of considering the entire execution history and how it had impacted the current instructions in memory.

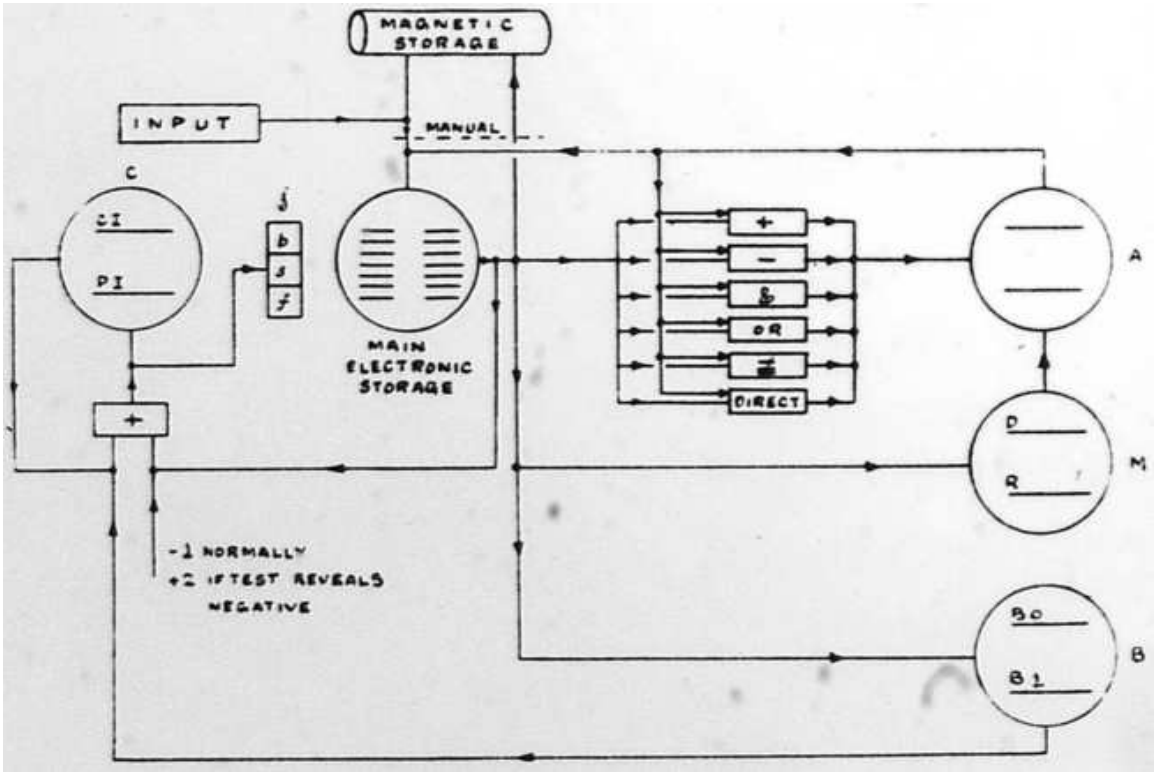


Diagram of the Manchester Mark I.
The B lines are in the B tube in the lower right corner.

13. "Assemblers"

The term "assembler" was used in a different way than it is generally used today. It referred to a subroutine that read in multiple other subroutines and assigned each one a number as it was read in[26, 27]. These subroutines could then call each other by number, rather than by address.

This allowed a programmer to think about what routine was being called, rather than the address into which the routine had actually been loaded.

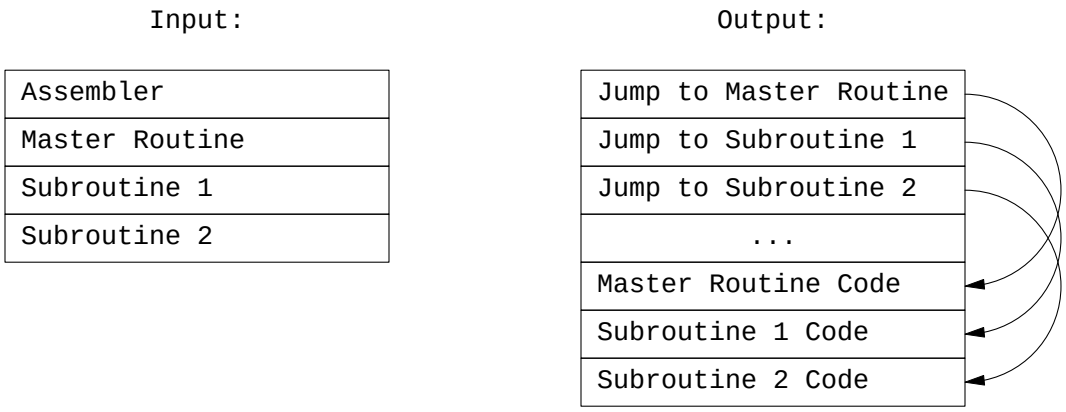


Illustration of assembler input and output
Subroutines were called by number using the jump table

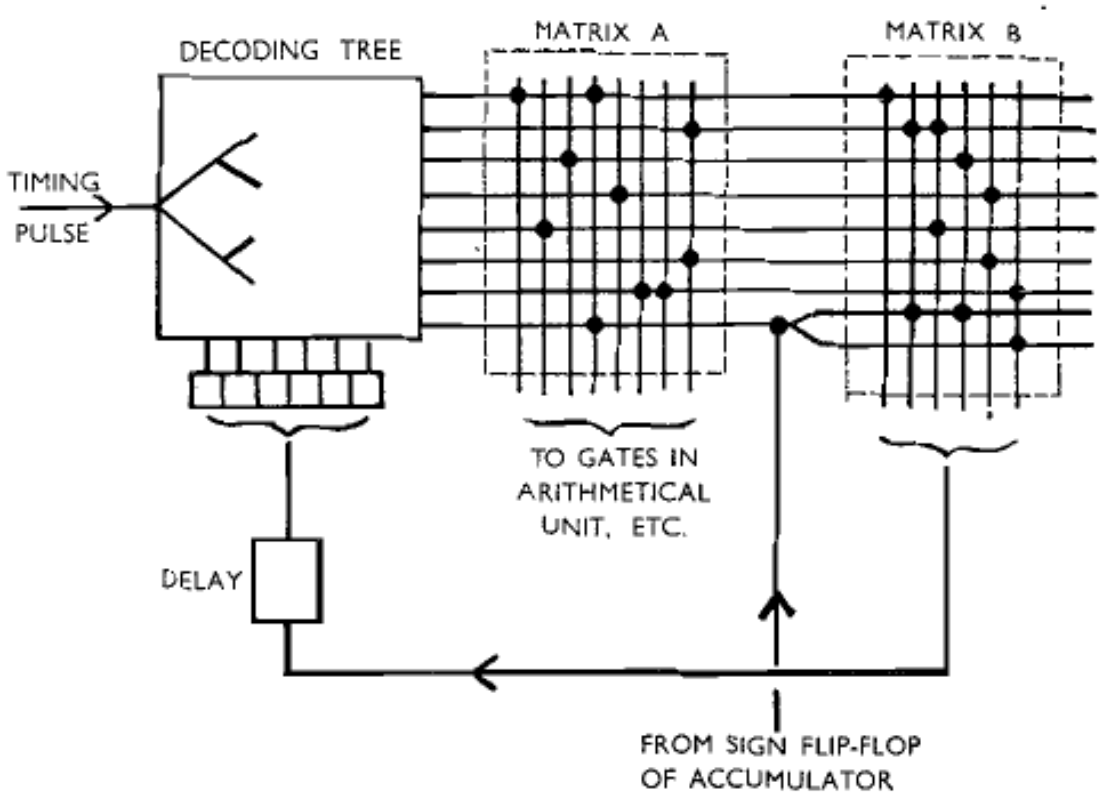
14. Microcode

In a microcode architecture, the design of the read/execute loop is separated into two portions: The hardware design and the microcode[22].

The hardware design implements various microinstructions, each of which can perform an extremely simple operation, like moving a value from memory to the accumulator or testing the sign bit of the accumulator.

The microcode is a program written in microinstructions that implements the computer's read/execute loop and interprets the machine code of programs run on the computer.

This allowed the computer designer to create a simpler control unit and to think about microinstructions instead of wiring diagrams when designing the fetch/execute cycle.

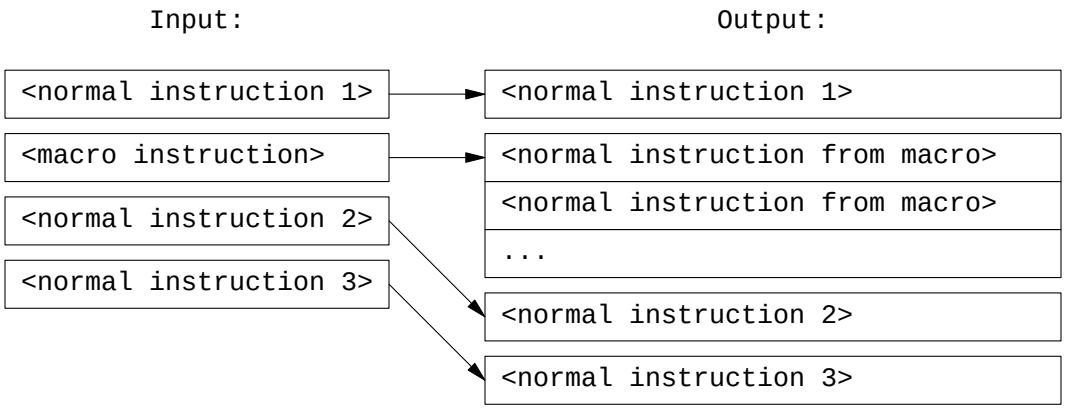


Microcode Architecture

15. Synthetic Orders/Macros

The phrase "Synthetic Orders" referred to what we would now call macros[23]. A loading routine would expand a synthetic order into multiple actual orders while the program was being loaded into the computer.

Like subroutines, this allowed a programmer to think about the overall action performed by a synthetic order instead of the detailed instructions that would execute at run time.



Macro expansion

16. Floating Addresses

The "subroutine relative" addressing implemented by the Initial Orders allowed a subroutine to be loaded into different places in memory, but the programmer still needed to code internal references as an offset from the beginning of the subroutine. If an instruction was added or removed at the beginning of the subroutine, the offset of all later instructions would change.

A "Floating Address" system was developed to enable programmers to label instructions in subroutines and to use the labels rather than offsets[23, 24, 27].

This allowed a programmer to think about the instruction being targeted by a jump rather than the address or offset of that instruction.

```
1*  A   2* S   ]
    P   1  F   ]  modify orders
    A   3* S   ]
    P   1  F   ]

    A 200  F   ]
    H  99  F   ]
2*  H  99  F   ]
3*  V  99  F   ]  form  a_n^2
    T 200  F   ]

    C   1* S   ]
    P 100  F   ]  count

    Z      F      stop
```

Code using floating addresses.
1*, 2*, and 3* are labels for locations in the code.

17. Summary of Abstraction Hierarchy Circa 1952 (three years after the EDSAC came on line)

What I find most amazing about all of the abstractions described in this talk is that they were almost all invented within the first three years of programming stored program computers, between 1949 and 1952.

<table>
<tr><td colspan="3">Floating Labels (1952)</td></tr>
<tr><td colspan="3">Program Assembler (1951)</td></tr>
<tr><td>Subroutines (1946)</td><td>Macros (1952)</td><td>Interpretive Subroutines/
Virtual Machines (1949)</td></tr>
<tr><td>Mnemonics (1949)</td><td>Relocation (1949)</td><td>Decimal
Addresses (1949)</td></tr>
<tr><td colspan="3">Microcode (1951)</td></tr>
<tr><td colspan="3">B-Lines/Index Registers (1949)</td></tr>
<tr><td colspan="3">Hardware Interpreting Data (1944)</td></tr>
</table>

Summary of Abstractions Developed by 1952

References

1. *Proceedings of a Symposium on Large-Scale Digital Calculating Machinery,* Harvard University Press, Cambridge, Massachusetts (1948).
2. *The Manchester Small Scale Experimental Machine.* http://www.computer50.org/mark1/new.baby.html.
3. Bennett, J. M., Prinz, D. G., and Woods, M. L., “Interpretative Sub-routines,” *Proceedings of the 1952 ACM national meeting (Toronto),* pp. 81-87, ACM (1952).
4. Berkeley, Edmund Callis, *Giant Brains, or Machines That Think,* John Wiley & Sons, Inc., New York (1949). http://www.archive.org/details/GiantBrains.
5. Brooker, R. A. and Wheeler, D. J., “Floating Operations on the EDSAC,” *Mathematical Tables and Other Aids to Computation* **7**(41), pp. 37-47 (Jan., 1953).
6. Burks, W., Goldstine, Herman H., and von Neumann, John, *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument,* The Institute for Advanced Study, Princeton, NJ (1946). http://library.ias.edu/ecp.
7. Eckert, J. Presper, “The ENIAC” in *A History of Computing in the Twentieth Century,* ed. Metropolis, M., Howlett, J., and Rota, Gian-Carlo, Academic Press, New York (1980).
8. Eckert, Jr., John Presper and Mauchly, John W., *Electronic Numerical Integrator and Computer (US Patent No. 3120606)* (Jun 26, 1947).

http://www.freepatentsonline.com/3120606.html.

9. Martin Campbell-Kelly (ed) and Michael R. Williams (ed), *The Moore School Lectures,* MIT Press, Cambridge, Massachusetts (1985).
10. Goldstine, Herman H. and von Neumann, John, *Planning and Coding of Problems for an Electronic Computing Instrument,* The Institute for Advanced Study, Princeton, NJ (1947). http://library.ias.edu/ecp.
11. Hartree, Douglas R., *Calculating Machines: Recent and Prospective Developments and their impact on Mathematical Physics,* Cambridge University Press, London (1947).
12. Hartree, Douglas R., *Calculating Instruments and Machines,* University of Illinois Press, Urbana (1949).
13. Engineering Research Associates Inc., *High-Speed Computing Devices,* McGraw-Hill Book Company Inc., New York (1950). http://www.archive.org/details/HighSpeedComputingDevices.
14. Kilburn, T., "The University of Manchester Unviersal High-Speed Digital Computing Machine," *Nature* **164**, pp. 684-687 (October 22, 1949).
15. The Staff of the Harvard Computation Laboratory, *A Manual of Operation for the Automatic Sequence Controlled Calculator,* Harvard University Press, Cambridge, Massachusetts (1946). http://www.bitsavers.org/pdf/harvard/MarkI_operMan_1946.pdf.
16. von Neumann, J., "First Draft of a Report on the EDVAC" in *The Origins of Digital Computers, Third Edition,* ed. Randell, Brian, Springer-Verlag, New York (1982).
17. Steele, Guy and Sussman, Gerald, *The Art Of The Interpreter or, The Modularity Complex,* Massachusetts Instute of Technology Artificial Intelligence Laboratory (May 1978). ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-453.pdf.
18. Wheeler, D. J., "Organization and Initial Orders for the EDSAC," *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences* **202**(1071), pp. 573-589 (Aug. 22, 1950).
19. Wheeler, D. J., "The use of sub-routines in programmes" in *ACM '52 Proceedings of the 1952 ACM national meeting (Pittsburgh),* pp. 235-236, Association for Computing Machinery (1952).
20. Wilkes, M. V. and Renwick, W., "The EDSAC (Electronic Delay Storage Automatic Calculator)," *Mathematical Tables and Other Aids to Computation* **4**(30), pp. 61-65 (Apr., 1950).
21. Wilkes, M. V., *Automatic Digital Computers,* John Wiley & Sons, Inc., New York (1957).
22. Wilkes, M. V., "The Best Way to Design an Automatic Calculating Machine" in *Computer Design Development: principal papers,* ed. Swartzlander, Jr., Earl E., Hayden Book Company, Inc., Rochelle Park, New Jersey (1976 (originally 1951)).
23. Wilkes, M. V., "Pure and applied programming" in *ACM '52 Proceedings of the 1952 ACM national meeting (Toronto),* pp. 121-123, Association for Computing Machinery (1952).
24. Wilkes, M. V., "The Use of a 'Floating Address' System for Orders in an Automatic Digital Computer," *Mathematical Proceedings of the Cambridge Philosophical Society* **49**(1), pp. 84-89 (January 1953).

25. Wilkes, Maurice V., *Memoirs of a Computer Pioneer,* MIT Press, Cambridge, Massachusetts (1985).

26. Wilkes, Maurice V., Wheeler, David J., and Gill, Stanley, *The Preparation of Programs for an Electronic Digital Computer,* Addison-Wesley, Cambridge, Mass. (1951).

27. Wilkes, Maurice V., Wheeler, David J., and Gill, Stanley, *The Preparation of Programs for an Electronic Digital Computer (2nd ed.),* Addison-Wesley, Reading, Mass. (1957).