A Brief History of Microprogramming

Mark Smotherman
Last updated: October 2012

Summary: Microprogramming is a technique to implement the control logic necessary to execute instructions within a processor. It relies on fetching low-level microinstructions from a control store and deriving the appropriate control signals as well as microprogram sequencing information from each microinstruction.

Definitions and Example

Although loose usage has sometimes equated the term "microprogramming" with the idea of "programming a microcomputer", this is not the standard definition. Rather, microprogramming is a systematic technique for implementing the control logic of a computer's central processing unit. It is a form of stored-program logic that substitutes for hardwired control circuitry.

The central processing unit in a computer system is composed of a *data path* and a *control unit*. The data path includes registers, function units such as shifters and ALUs (arithmetic and logic units), internal processor busses and paths, and interface units for main memory and I/O busses. The control unit governs the series of steps taken by the data path during the execution of a user-visible instruction, or *macroinstruction* (e.g., load, add, store).

Each action of the datapath is called a *register transfer* and involves the transfer of information within the data path, possibly including the transformation of data, address, or instruction bits by the function units. A register transfer is accomplished by gating out (sending) register contents onto internal processor busses, selecting the operation of ALUs, shifters, etc., through which that information might pass, and gating in (receiving) new values for one or more registers.

Register-enabling signals, which control the sending or receiving of data at the registers, and operation-selection signals, which control the actions of the functional units, are called *control signals*. These signals are supplied by the control unit. The collections of logic gates in the datapath that respond to enabling signals and allow the sending or receiving of data at the registers are called the called *control points*.

Each step in the execution of a macroinstruction thus consists of one or more register transfers, and a complete macroinstruction is executed by generating an appropriately timed sequence of groups of control signals. Individual datapath actions or sets of related actions are often called *microoperations*.

As an example, consider the simple processing unit in Figure 1. This datapath supports an accumulator-based instruction set of four macroinstructions (load, add, store, and conditional branch). The accumulator (ACC) and program counter (PC) are visible to the macroinstruction-level programmer, but the other registers are not.

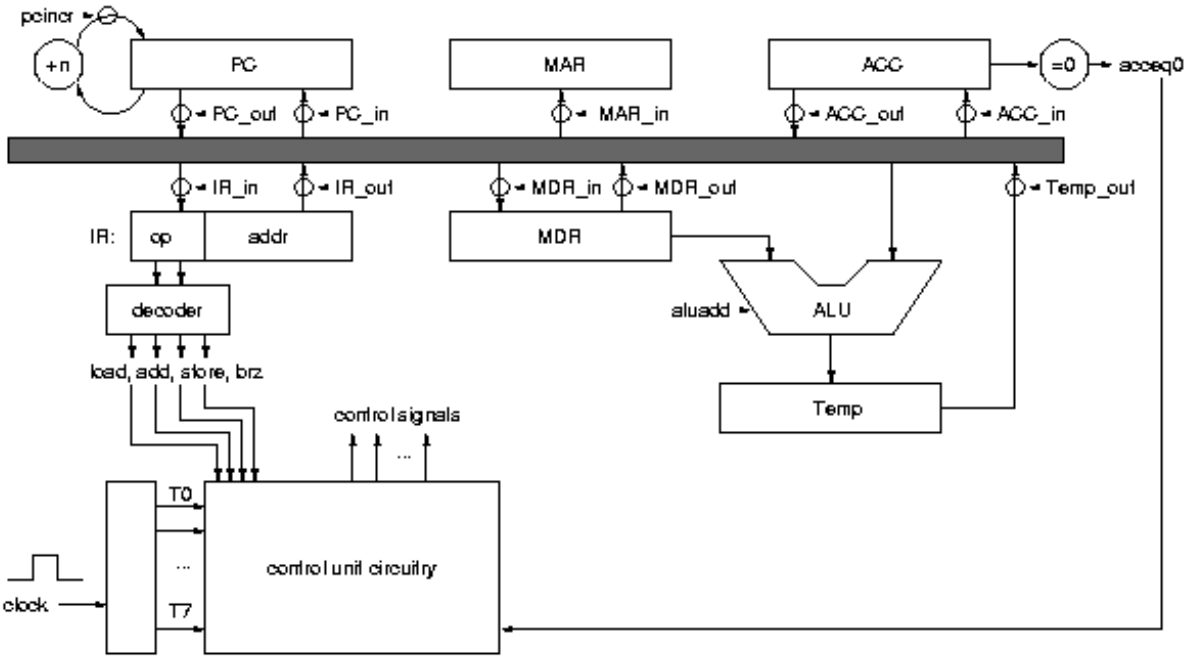


Figure 1. Simple data path for a four-instruction computer (the small circles represent control points)

The definitions of the four macroinstructions are given in Figure 2, and the definitions of the control signals used for this datapath are given in Figure 3. In this datapath, control signals such as `ACC_in` and `ACC_out` are register-enabling signals, and control signals such as `aluadd` and `pcincr` are operation-selection signals.

```
(opcode 00)  load  address  :   ACC <- memory[ address ]

(opcode 01)  add   address  :   ACC <- ACC + memory[ address ]

(opcode 10)  store address  :   memory[ address ] <- ACC

(opcode 11)  brz   address  :   if( ACC == 0 ) PC <- address
```

Figure 2. Instruction definitions for the simple computer

```
ACC_in   : ACC <- CPU internal bus
ACC_out  : CPU internal bus <- ACC
aluadd   : addition is selected as the ALU operation
IR_in    : IR <- CPU internal bus
IR_out   : CPU internal bus <- address portion of IR
MAR_in   : MAR <- CPU internal bus
MDR_in   : MDR <- CPU internal bus
MDR_out  : CPU internal bus <- MDR
PC_in    : PC <- CPU internal bus
PC_out   : CPU internal bus <- PC
pcincr   : PC <- PC + 1
read     : MDR <- memory[ MAR ]
TEMP_out : CPU internal bus <- TEMP
write    : memory[ MAR ] <- MDR
```

Figure 3. Control signal definitions for the simple datapath

To implement the processing of a macroinstruction on this datapath, a first group of control signals are needed to fetch the

macroinstruction from memory. These control signals will gate the contents of the program counter onto the internal bus and gate these bits from the internal bus into the memory address register. Either in this group of control signals (i.e., during the first time step) or the next, the program counter should be incremented, and the memory interface should be sent a signal to cause a memory read. After the read of memory is complete (which may be a synchronous or an asynchronous event), the contents of the memory data register are transferred to the instruction register.

Once the bits of macroinstruction are in the instruction register, the bits in the opcode field can be used to govern the remaining steps. These steps include fetching any operands, executing the function specified by the opcode, and storing any results. Total processing of a simple macroinstruction might require five to ten time steps on this datapath and might involve a dozen or more control signals.

Figure 4 gives control sequences for the four macroinstructions. For this simple example, it is assumed that a memory read or write completes in one time step.

```
time steps T0-T3 for each instruction fetch:

        T0:     PC_out, MAR_in
        T1:     read, pcincr
        T2:     MDR_out, IR_in
        T3:     time step (if needed) for decoding the opcode in the IR


time steps T4-T6 for the load instruction:

        T4:     IR_out(addr part), MAR_in
        T5:     read
        T6:     MDR_out, ACC_in, reset to T0


time steps T4-T7 for the add instruction:

        T4:     IR_out(addr part), MAR_in
        T5:     read
        T6:     ACC_out, aluadd
        T7:     TEMP_out, ACC_in, reset to T0


time steps T4-T6 for the store instruction:

        T4:     IR_out(addr part), MAR_in
        T5:     ACC_out, MDR_in
        T6:     write, reset to T0


time steps T4-T5 for the brz (branch on zero) instruction:

        T4:     if (acceq0) then { IR_out(addr part), PC_in }
        T5:     reset to T0
```

Figure 4. Control sequences for the four instructions

As stated above, the control unit is responsible for generating the sequences of control signals. As shown in the lower left of Figure 1, the control unit inputs consist of

1. a mutually-exclusive set of time step signals,
2. a mutually-exclusive set of decoded opcode signals, and

3. condition signals used for implementing the conditional branch instruction.

The logic expression for an individual control signal can be written in the sum-of-products form in which the terms typically consist of a given time step signal anded with an opcode signal identifying a specific instruction opcode. However, a given term may also consist merely of a time step signal, or it may have one or more of the condition signals included. The control signal will then be asserted when needed at one or more specific time steps during the instruction fetch and execution.

The logic expressions for the control signals for the simple example are given in Figure 5. For example, the logic expression for `ACC_in` has the typical form and evaluates to true for the load instruction at time step T6 and for the add instruction in time step T7. `IR_in` has a simple expression and evaluates to true for all instructions at time step T2. `PC_in` is conditionally true in time step T4 for the brz instruction since the condition signal `acceq0` is included as part of the term.

```
        ACC_in = (load & T6) + (add & T7)
        ACC_out = (store & T5) + (add & T6)
        aluadd = add & T6
        IR_in = T2
        IR_out(addr part) = (load & T4) + (add & T4) + (store & T4) + (brz & acceq0 &
T4)
        MAR_in = T0 + (load & T4) + (add & T4) + (store & T4)
        MDR_in = store & T5
        MDR_out = T2 + (load & T6)
        PC_in = brz & acceq0 & T4
        PC_out = T0
        pcincr = T1
        read = T1 + (load & T5) + (add & T5)
        TEMP_out = add & T7
        write = store & T6
```

Figure 5. Control signal definitions in sum-of-products form

When the logic expressions for the control signals are directly implemented with individual logic gates or with an incompletely-decoded gate array such as a programmed logic array (PLA), the control unit is said to be *hardwired*. A direct implementation with logic gates is depicted in Figure 6. The typically irregular placement of logic gates in direct implementations led to this approach being described as "random logic". (See also the appendix sections below on control design in the MIT Whirlwind and other early examples of hardwired control design.)

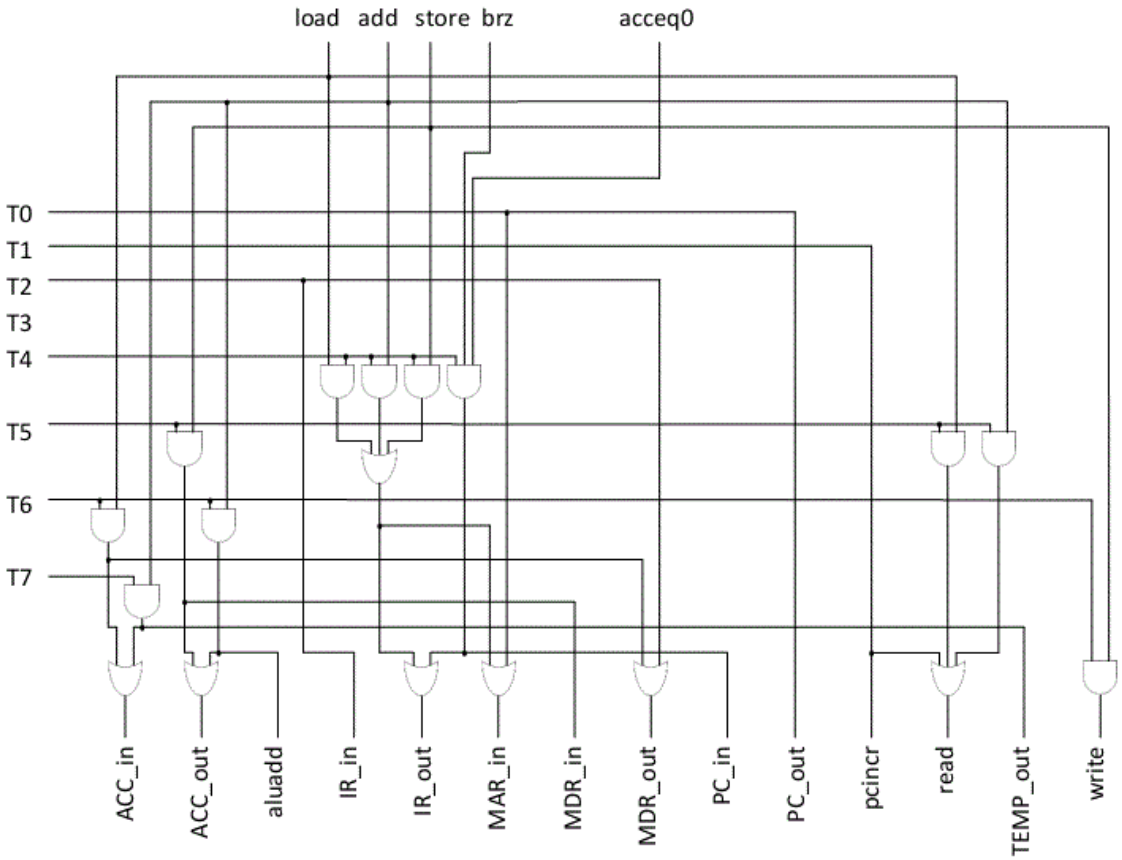


Figure 6. Hardwired control implementation for the four-instruction computer

Alternatively, in a microprogrammed control unit, the control signals that are to be generated at a given time step are stored together in a control word, which is called a *microinstruction*. The collection of control words that implement an instruction is called a *microprogram*, and the microprograms are stored in a memory element called the *control store*.

A microprogrammed implementation of control for the simple example is given in Figure 7. The control store is a 16 x 20 bit memory. It is accessed by a control store address register (CSAR), and the word that is fetched is held in a control store instruction register (CSIR). The control signals needed for a given time step are thus provided to the datapath by the CSIR.

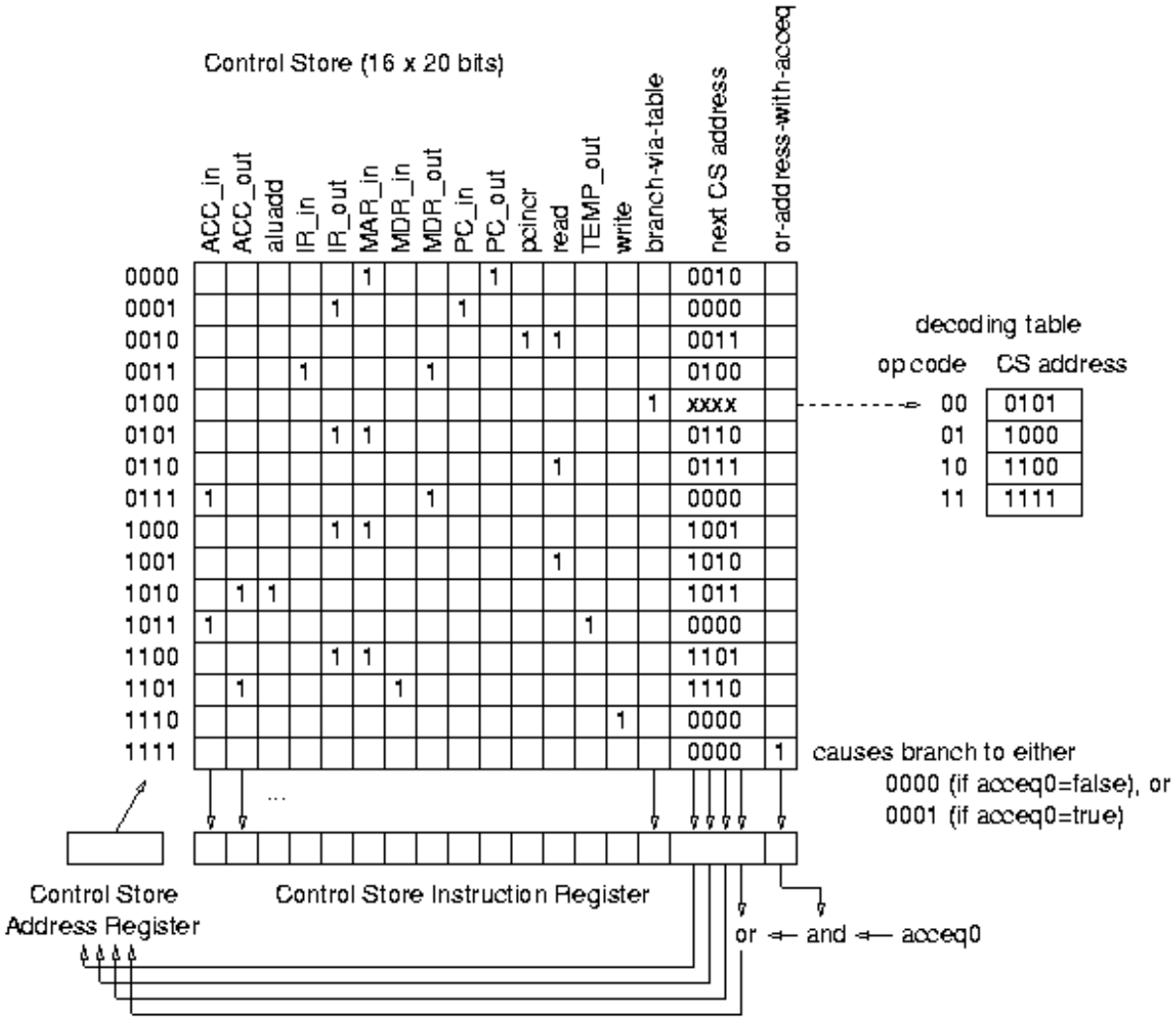


Figure 7. Control store for the four-instruction computer (control bits of zero not shown)

The first fourteen bits in each microinstruction in Figure 7 are the register-enabling and operation-selection control signals used in the Figure 1 datapath. Six additional bits are used in each microinstruction in Figure 7 to implement sequencing.

These six bits include a four-bit next-address field along with a one-bit override signal (`branch-via-table`) that causes the next address to be loaded from a separate decoding table instead of using the next-address field. In this example, the decoding table is indexed by the two-bit opcode field from the instruction register and provides the entry-point address in the control store for the microprogram associated with that particular opcode.

The sixth bit (`or-address-with-acceq`) is used to handle conditional branching by allowing the condition signal `acceq0` to conditionally alter the least significant bit in the next-address field. This occurs when the next-address field in the microinstruction ends in a 0, so that the next microinstruction will be fetched from either the address ending in 0 or the sequentially-following address ending in a 1, depending upon the condition signal being false or true, respectively.

Figure 8 illustrates the sequencing of fetches from the control store of Figure 7 for the four instructions.

```
clock    CSAR    control signals
cycle

  0      0000    PC_out, MAR_in
```

```
  1     0010    read, pcincr
  2     0011    MDR_out, IR_in
  3     0100    branch-via-table (to one of the cases below)


a load instruction causes a branch to 0101 for cycle 4:

  4     0101    IR_out(addr part), MAR_in
  5     0110    read
  6     0111    MDR_out, ACC_in (and jump to 0000)


or an add instruction causes a branch to 1000 for cycle 4:

  4     1000    IR_out(addr part), MAR_in
  5     1001    read
  6     1010    ACC_out, aluadd
  7     1011    TEMP_out, ACC_in (and jump to 0000)


or a store instruction causes a branch to 1100 for cycle 4:

  4     1100    IR_out(addr part), MAR_in
  5     1101    ACC_out, MDR_in
  6     1110    write (and jump to 0000)


or a brz instruction causes a branch to 1111 for cycle 4; the branch
  may either be untaken, in which case control returns to the start
  of the next macroinstruction fetch at address 0000:

  4     1111    or_address_with_acceq0 (yields jump to 0000)


or it may be a taken branch, in which case the true condition bit
  modifies the next-address field and directs the control store
  fetch for cycle 5 to a microinstruction that changes the PC:

  4     1111    or_address_with_acceq0 (yields jump to 0001)
  5     0001    IR_out(addr part), PC_in (and jump to 0000)
```

Figure 8. Control store fetch sequences for the four instructions

As shown for this example, in a microprogrammed control unit, the sequence of control signals needed for instruction fetch and execution derives from a series of fetches from the control store rather than the operations of hardwired circuitry. The result is a more systematic design for control that allows changes to be made by altering the contents of control store rather than rewiring or resynthesizing hardwired circuitry.

It is important to note that the control store depicted above uses completely unencoded storage of control signals. This is inefficient (note all the zero locations), and an actual implementation would encode groups of control signals. For example, all the signals used to gate the contents of registers out onto the single shared bus are mutually exclusive and can be encoded into one field. Indeed, a typical microinstruction in a commercial processor consists of a set of encoded fields of various sizes rather than containing a bit vector of individual control signals.

Since the microinstructions and microprograms stand between the logic design (hardware) and the macroinstruction program being executed (software), they are sometimes referred to as *firmware*. This term is also subject to loose usage. Ascher Opler first

defined it in a 1967 Datamation article as the contents of a *writable control store*, which could be reloaded as necessary to specialize the user interface of a computer to a particular programming language or application [Opl67]. However, in later general usage, the term came to signify any type of microcode, whether resident in read-only or writable control store. Most recently, the term has been widened to denote anything ROM-resident, including macroinstruction-level routines for BIOS, bootstrap loaders, or specialized applications.

See chapters 4 and 5 of Hamacher, Vranesic, and Zaky [Ham90] and chapter 5 of Patterson and Hennessy [Pat98] for overviews of datapath design, control signals, hardwiring, and microprogramming. Older texts devoted exclusively to microprogramming issues include Agrawala and Rauscher [Agr76], Andrews [And80], Habib [Hab88], and Husson [Hus70]. The ACM and IEEE sponsored for over twenty years an Annual Workshop on Microprogramming and published the proceedings; the workshop name was changed in 1991 to the International Symposium on Microarchitecture, to reflect the change of emphasis to the broader field of *microarchitecture*, i.e., the internal design of the processor, including such areas as branch prediction, multiple instruction execution, and low-power design.

History of Microprogramming

In the late 1940s Maurice Wilkes of Cambridge University started work on a stored-program computer called the EDSAC (Electronic Delay Storage Automatic Calculator). During this effort, Wilkes visited the MIT Whirlwind and observed the Whilrwind's diode-matrix "control store" [Wil85]. (See the Whirlwind appendix section below.) Wilkes recognized that the sequencing of control signals within the computer was similar to the sequencing actions required in a regular program and that he could use a stored program to represent the sequences of control signals [Wil85]. In 1951, he published the first paper on this technique, which he called microprogramming [Wil51].

In Wilkes' seminal paper, he described an implementation of a control store using a diode matrix. The microinstructions held in the control store had a simple format similar to the example above: the unencoded control signals were stored with a next-address field. Initial selection of the appropriate microprogram was handled by using the opcode value appended with zeros as a starting address in the control store, and normal sequencing used the contents of the next-address fields thereafter. Conditional transfers were handled by allowing conditions to select one of multiple possible next-address fields.

In an expanded paper published in 1953, Wilkes and his colleague John Stringer further described the technique [Wil53]. Figure 9 reproduces a figure from the 1953 paper. In this visionary paper, they additionally considered issues of design complexity, test and verification of the control logic, alternation of and later additions to the control logic, support of different macroinstruction sets (by using different matrices), exploiting parallelism within the data path, multiway branches, environment substitution, microsubroutines, variable-length and polyphase timing, and pipelining the access to the control store.

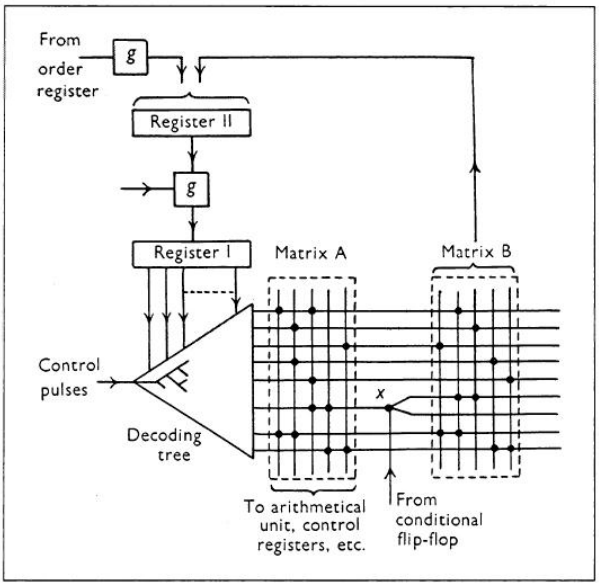


Figure 9. Microprogramming design of Wilkes [Figure 1 of [Wil53], reprinted in Chapter 28 of Gordon Bell's on-line version of Computer Structures book]

The Cambridge University group, including William Renwick and David Wheeler, went on to implement and test the first microprogrammed computer in 1957. Because diode matrices would be too large for a computer with a full instruction set, they used a magnetic core matrix instead [Whe92]. At each intersection of the matrix, a magnetic core could be switched on by coincident currents and then used to drive multiple control signals as well as sequencing signals. The matrix could also have two or four cores wired at an intersection to implement conditional control and conditional sequencing; wires from condition flip-flops would be wound around these cores in such a way that that only one of the cores could be switched on at a time [Ren56,Wil92].

For their initial tests in 1957, they used an 8-by-6 magnetic core matrix, and Wheeler describes this arrangement as having "64 microprogram steps" [Whe92]. This machine was known as the EDSAC 1½. The complete EDSAC 2 was operational in 1958 and used a 32-by-32 magnetic core matrix. Wheeler describes the EDSAC 2 as having "a microprogram of 1,024 steps including a complete order decoder of 128 steps, so that unlisted orders produced a 'report stop'" [Whe92]. EDSAC 2 also used a bit-slice design approach for the ALU and index resisters to help systematize the design of the arithmetic circuits and to make maintenance easier.

Due to the difficulty of manufacturing fast control stores in the 1950s, microprogramming did not immediately become a mainstream technology. However, several computer projects did pursue Wilkes' ideas. For example, Heinz Billing and Wilhelm Hopmann in Germany designed a microprogrammed control for the Göttingen G1a using magnetostatic delay lines [Hop00]; this work in turn influenced the control design of the Zuse Z22 and the Telefunken TR-4. In England, the EMIDEC 1100 and LEO III are examples of early transistor-based computers with control units based on microprogramming.

John Fairclough at IBM's laboratory in Hursley, England, also led a development effort in the late 1950s that explored a read-only magnetic core matrix for the control unit of a small computer. This work resulted in the SCAMP (Scientific Computer and Modulator Processor). In 1961, Fairclough's experience played a key role in IBM's decision to pursue a full range of compatible computers, which was announced in 1964 as the System/360 [Pug91]. All but two of the initial 360 models (the high-end Models 75 and 91) were microprogrammed [Tuc67].

Table 1 compares the initial System/360 models that were microprogrammed.

	M30	**M40**	**M50**	**M65**
minimum rental fee ($K/mo.)	4	7	15	35
Knight index - scientific (Kops)	7.94	33.4	187	1390
Knight index - commercial (Kops)	17.1	50.1	149	810
internal data path width (bits)	8	16	32	64
control store size (K microinsts.)	4	4	2.75	2.75
microinstruction width (bits)	50	52	85	87
control store technology	CCROS	TROS	BCROS	BCROS
control store cycle time (ns)	750	625	500	200
main memory cycle time (ns)	1500	2500	2000	750

Table 1. Comparison of IBM System/360 Models 30-65 (ca. 1965) [Phi79,Pad81,Pug91].

Because of the vital nature of microprogramming to the plan for compatibility, IBM aggressively pursued three different types of read-only control store technologies [Pug91]. The first was Balanced Capacitor Read-Only Storage (BCROS), which used two capacitors per bit position in a storage of 2816 words of 100 bits each. A second technology was Transformer Read-Only Storage (TROS), which used the magnetic core approach taken by Fairclough at Hursley in a storage of 8192 words of 54 bits each.

A third technology was developed for the Model 30, Card Capacitor Read-Only Storage (CCROS). This technology was based on mylar cards the size and shape of standard punch cards that encased copper tabs and access lines. A standard card punch could be used to remove some of the tabs from a 12 by 60 bit area in the middle of the card. Removed tabs were read as zeros by sense lines, while the remaining tabs were read as ones. An assembly of 42 boards with 8 cards per board was built into the Model 30 cabinet.

CCROS cards on the Model 30 could be replaced in the field. This made design modifications easier to accomplish, as well as providing for field diagnosis. The control store of the Model 30 was divided into a section for resident routines and a section of six CCROS cards that would hold test-specific routines [Joh71]. The reserved board could be "populated" with one of several sets of CCROS cards by an engineer running machine diagnostics.

The later System/360 Model 25 and models of the later System/370 series were configured with at least one portion of the control store being read-write for loading microcode patches and microdiagnostics. (On the smaller System/370 models, the control store was allocated in part of main memory). Indeed, IBM invented the eight-inch floppy diskette and its drive to improve the distribution of microcode patches and diagnostics; the diskette was first used in 1971 on the System/370 Model 145, which had a writable control store (WCS) of up to 16K words of 32 bits each.

During the change-over of the IBM product line to the new System/360 in the mid-sixties, the large customer investment in legacy software, especially at the assembly language level, was not fully recognized [Tuc99]. Software conversion efforts through recompilation of high-level-language programs were planned, but the pervasive use of machine-dependent codes and the unwillingness of customers (and economic infeasibility) to convert all these machine-dependent codes required efforts at providing simulators of the older computers. Initial studies of the simulators, however, indicated that, at best, performance would suffer by factors ranging from ten to two [Pug91]. Competitors were meanwhile seeking to attract IBM customers by providing less intensive conversion efforts using more compatible hardware and automatic conversion tools, like the Honeywell "Liberator" program that accepted IBM 1401 programs and converted them into programs for the 1401-like Honeywell H-200.

IBM was spared mass defection of former customers when engineers on the System/360 Model 30 suggested using an extra control store that could be selected by a manual switch and would allow the Model 30 to execute IBM 1401 instructions [McC65]. The simulator and control store ideas were then combined in a study of casting crucial parts of the simulator programs as microprograms in the control store. Stuart Tucker and Larry Moss led the effort to develop a combination of hardware, software, and microprograms to execute legacy software for not only the IBM 1401 computers but also for the IBM 7000 series [Tuc65]. Moss felt their work went beyond mere imitation and equaled or excelled the original in performance; thus, he termed their work as *emulation* [Pug91]. The emulators they designed worked well enough so that many customers never converted legacy software and instead ran it for many years on System/360 hardware using emulation.

Because of the success of the IBM System/360 product line, by the late 1960s microprogramming became the implementation technique of choice for most computers except the very fastest and the very simplest. This situation lasted for about two decades. For example, all models of the IBM System/370 aside from the Model 195 and all models of the DEC PDP-11 aside from the PDP-11/20 were microprogrammed.

At perhaps the peak of microprogramming's popularity, the DEC VAX 11/780 was delivered in 1978 with a 4K word read-only control store of 96 bits per word and an additional 1K writable region available for diagnostics and microcode patches. An extra-cost option on the 11/780 was 1K of user-writable control store.

Several early microprocessors were hardwired, but some amount of microprogramming soon became a common control unit design feature. For example, among the major eight-bit microprocessors produced in the 1974 to 1976 time frame, the MC6800 was hardwired while the Intel 8080 and Zilog Z80 were microprogrammed [Anc86]. An interesting comparison between 1978-era 16-bit microprocessors is the hardwired Z8000 [Shi79] and the microcoded Intel 8086 [McK79]. The 8086 used a control store of 504 entries, each containing a rather generic 21-bit microinstruction. Extra decoding logic was used to tailor the microinstruction to the particular byte-wide or word-wide operation. In 1978 the microprogramming of the Motorola 68000 was described [Anc86, Str78,Tre88]. This design contained a sophisticated two-level scheme. Each 17-bit microinstruction could contain either a 10-bit microinstruction jump address or a 9-bit "nanoinstruction" address. The nanoinstructions were separately stored 68-bit words, and they identified the microoperations to be performed in a given clock cycle. Additional decoding logic was used along with the nanoinstruction contents to drive the 196 control signals.

An article by Nick Tredennick in 1982 characterized the development of four major uses of microprogramming, which he termed cultures [Tre82]:

- *Commercial machine* - In this mode of usage, microprogramming is used to implement a standard computer family instruction set. There is no access to microcode by the user, but the control store is designed to be writable so that the manufacturer can make changes in the microcode. The IBM System/370 series is a major example [Pad81]. Commercial machines have also used microprogramming to implement operating system functions, varying from special-purpose assists

(e.g., IBM VM/370 assists provided 45%-65% reductions in elapsed time for OS calls [Pad81]) to major portions of an OS kernel (e.g., the French-designed Honeywell Series 60 Level 64, later called the DPS-7, had threading and semaphore operations implemented via microprogramming [Atk74]).

- *Single-chip* - For microprocessors, microprogramming is an implementation technique used to simplify the design. The control store is typically read-only and cannot be changed without a revision in the masks used in chip fabrication. The MC68000 family is an example [Tre88]. (Note, however, that more recent microprocessors contain small microcode RAMs that provide for microcode updates and patches that can override the ROM-based microcode. This is similar to, but typically much more limited than, the capability provided by commercial machines. For an example of this facility, see [Bad92], in which a 20-word RAM is used for patching a 1600-word ROM in the microsequencer for the EBOX in the single-chip DEC NVAX microprocessor.)

- *Bit-slice* - Prior to the widespread use of VLSI ASICs, designers would build application-specific machines by using commercial off-the-shelf microprogram sequencers and datapath building blocks. The datapath components were two- to four-bit wide slices of an ALU plus register file, and multiple slices could be hooked together to build a datapath of the desired width. There were several families of components and sequencers, including the Intel 3001/3002, AMD 2901/2909 and 2903/2910, TI 74S, and MC10800. See Andrews [And80] and Mick and Brick [Mic80] for more details.

- *Microprogrammable machine* - Because minicomputers like the PDP-11/60 and VAX-11/750 were becoming available and provided fairly easy user access to writable control stores, user microprogramming was gaining popularity. For example, Jim Larus published a study of pattern matching on the VAX-11/750 and reported that a microcoded implementation was 14-28 times faster in execution performance than hand-coded assembly language or a compiled C program [Lar82]. In 1978, Bell, Mudge, and McNamara estimated microprogramming productivity at 700 microinstructions per man year [Bel78]. This meant that microprogramming was five to ten times more expensive than conventional programming of same task, but it was estimated that microprogramming was usually ten times faster in performance.

  An important sub-area within the microprogrammable machine culture was the desire to produce a *universal host machine* with a general data path for efficient emulation of any macroinstruction set architecture or for various high-level-language-directed machine interfaces. Examples of this include the Nanodata QM-1 (ca. 1970) [Agr76,Nanodata,Sal76] and the Burroughs B1700 (ca. 1972) [Bar92,Burroughs,Mye78,Sal76]. ALU operations on the QM-1 were selectable as 18-bit or 16-bit, binary or decimal, and as unsigned or two's complement or one's complement. However, even with efforts to provide a generalized set of operations such as this, the unavoidable mismatches between host and target machine data types and addressing structures never allowed unqualified success for the universal host concept.

  The B1700 is probably the closest commercial machine to Opler's original vision of firmware. Multiple sets of microprograms could be held in its memory, each one presenting a different high-level macroinstruction interface. The B1700 operating system was written in a language called SDL, while application programs were typically written in Fortran, Cobol, and RPG. Thus, the control store for the B1700 could hold microprograms for an SDL-directed interface as well as microprograms for a Fortran-directed interface and a Cobol/RPG-directed interface. On an interrupt, the B1700 would use the SDL-directed microprograms for execution of operating system code, and then on process switch the operating system could switch the system to the Fortran-directed microprograms or the Cobol/RPG-directed microprograms. Baron and Higbie question the B1700's choice of 24-bit word size, lack of two's complement arithmetic support, address space limitations, and limited interrupt and I/O facilities, but they mainly attribute the limited commercial success of the B1700 to the lack of operating system documentation for developers [Bar92].

Starting in the late seventies a trend emerged in the growth of complexity in macroinstruction sets. The VAX-11/780 superminicomputer and the MC68020 microprocessor are examples of this trend, which has been labeled CISC for complex instruction set computer. The trend toward increasing complexity is generally attributed to the success of microprogramming to that date. Freed from previous constraints arising from implementation issues, the VAX designers emphasized ease of compilation [Bel78]. For example, one design criterion was the generality of operand specification. However, the result was that the VAX had complex, variable-length instructions that made high-performance implementations difficult [Bha91].

Compared to the original MC68000 (ca. 1979), the MC68020 (ca. 1984) added virtual memory support, unaligned data access, additional stack pointers and status register bits, six additional stack frame formats, approximately two dozen new instructions, two new addressing modes (for a total of 14), and 18 new formats for offsets within addressing modes (for a total of 25). To support this, microcode storage increased from 36 KB to 85 KB [Tre88]. The complex memory indirect addressing modes on the

MC68020 are representative of the design style engendered by a control store with access time only a fraction of the main memory access time. This style dictates that only a relatively few complex macroinstructions should be fetched from main memory and a multitude of register transfers can then be generated by these few macroinstructions.

Added to the trend of growing complexity within "normal" macroinstruction set architectures was an emphatic call to "bridge the semantic gap", under the assumption that higher-level machine interfaces would simplify software construction and compilation. See chapter 1 of Myers [Mye78] for an overview of this argument.

However, starting in the 1980s, there was a reaction to the trend of growing complexity. Several technological developments drove this reaction. One development was that VLSI technology was making on-chip or near-chip cache memories attractive and cost effective; indeed, dual 256-byte instruction and data caches were included on the MC68030 (ca. 1987), and dual 4K-byte instruction and data caches were part of the MC68040 (ca. 1989). Thus effective memory access time was now closer to the processor clock cycle time.

A second development was the wide use of VLSI design tools and the ability of designers to easily synthesize hardwired designs using these CAD tools. Changes to the design of a control unit became a matter of running the altered design through the simulation and synthesis toolchain with little to no need for a circuit designer to perform custom design of the control logic.

The change in the logic/memory technological ratio and the ease of implementing a hardwired control unit set the stage for a design style change to RISC, standing for reduced instruction set computer and named so as to heighten the contrast with CISC designs. The RISC design philosophy argues for simplified instruction sets for ease of hardwired, high-performance, and pipelined implementations. In fact, one could view the RISC instruction sets as quite similar to highly-encoded microinstructions and the accompanying instruction cache as a replacement for a writable control store. Thus, instead of using a fixed set of microprograms for interpretation of a higher-level instruction set, RISC could be viewed as compiling directly to microcode. A counterargument to this view, given by some RISC proponents, is that RISC has less to do with exposing the microprogram level to the compiler and more to do with (1) a rediscovery of Seymour Cray's supercomputer design style as exhibited in the CDC 6600 design of the mid-sixties, and (2) an agreement with John Cocke's set of hardware/software tradeoffs as exhibited in the IBM 801 design of the early 1980s.

Indeed, the 1980s proved to be a turning point for the use of traditional microprogramming. The RISC microprocessors introduced in the 1980s, like SPARC and MIPS, were hardwired. In the 1990s, the MC68000 macroinstruction set was downsized, so that the implementation of the remaining core macroinstruction set could be hardwired [Cir95]. Application-specific tailoring of hardware also became an ASIC or FPGA design problem rather than being seen as requiring custom microprogramming.

Microcode is still used, however, in the numerous Intel x86 compatible microprocessors like the Intel Pentium Pro and its descendants and in the AMD microprocessors. However, within these designs, simple macroinstructions have one to four microoperations (called uops or Rops, respectively) immediately generated by decoders without control store fetches. This suffices for all but the most complex macroinstructions, which still require a stream of microinstructions to be fetched from an on-chip ROM by a microcode sequencer. See Shriver and Smith [Shr98] for a detailed exposition of the internal design of an x86-compatible microprocessor, the AMD K6-2. (Note that most x86 microprocessors now contain a small amount of microcode RAM to provide for patching the microcode, called "microcode update" as well as "BIOS update" since the BIOS loads the update whenever power is turned on. Intel provides a processor update utility that can patch microcode for P6 and Pentium 4 processors. A similar approach is available for AMD Athlons and Opterons.)

IBM mainframe design was also affected by the transition back from the popularity of microprogrammed control to hardwiring. Since the mid-1990s, IBM System/390 and zSeries mainframes have had many of their macroinstructions hardwired [Web97]. The recent IBM z10 implements approximately three-fourths of its macroinstructions in hardware [Web08]. The remaining complex macroinstructions are implemented by an internal code called "millicode", which uses hardware-specific milli-instructions with instruction formats similar to those of the macroinstructions [Hel05]. (The Amdahl 580 mainframe designs of the 1980s had a similar approach but called their internal code "macrocode" [Dor88], not to be confused with the use of the term macroinstruction in this discussion. Also in the 1980s, DEC used epicode, "extended processor instruction code", in the PRISM architecture and later PALcode, "Privileged Architecture Library code", in the Alpha architecture. These approaches are reminiscent of the reserved store in the EDSAC 2 in the 1950s [Wil92] and extracodes in the Manchester Atlas in the early 1960s [Lav78].)

Current Trends Related to Microprogramming

Microprogramming or microprogramming-like approaches have appeared recently in proposals related to application-specific processors, HW/SW codesign, code compression, and low-power designs. Some examples include:

- The Tensilica Instruction Extension language allows users to define application-specific instructions and user state registers in a subset of Verilog [Gon00]. This approach can be compared to the user-microprogrammable machines of the 1970s and 1980s. The Tensilica design process then implements the extended processor in silicon.
- NISC (No Instruction Set Computer) is an approach that compiles C programs to microcode [Res05].
- PRISA (Post-manufacturing Reconfigurable ISA) provides a generic datapath along with a large control store ROM from which the user can dynamically select an application instruction set of fifteen 16-bit instructions for decoding by a programmable instruction decoder [Che08]. The programmable instruction decoder is essentially a small WCS that can contain copies of a subset of the larger control store's lines.
- The Stanford ELM project replaces a traditional instruction cache with a control and index memory (CIM) and instruction register files (IRF) [Bla08]. The CIM/IRF arrangement is comparable to a two-level control store.

Flavors of Microprogramming

The level of abstraction of a microprogram can vary according to the amount of control signal encoding and the amount of explicit parallelism found in the microinstruction format. On one end of a spectrum, a *vertical* microinstruction is highly encoded and may look like a simple macroinstruction containing a single opcode field and one or two operand specifiers. For example, see Figure 10, which shows an example of vertical microcode for a Microdata machine [Microdata].

```
; multiply two numbers

     LF  5,X'08'   ; set loop counter to 8
     MT  2         ; move X to T
     LF  4,X'00'   ; clear ZU
ADD: TZ  3,X'01'   ; if Y bit 0 set to 1
     A   4,T       ; add X to Z
     H   4,R       ; shift Z upper
     H   3,L,R     ; shift Z lower
     D   5,C       ; decrement loop counter
     TN  0,X'04'   ; if loop counter equal 0
     JP  ADD       ; jump to top of loop
```

Figure 10. Vertical microcode: Microdata example

Each vertical microinstruction specifies a single datapath operation and, when decoded, activates multiple control signals. Branches within vertical microprograms are typically handled as separate microinstructions using a "branch" or "jump" opcode. This style of microprogramming is the most natural to someone experienced in regular assembly language programming, and is similar to programming in a RISC instruction set.

At the other end of the spectrum, a *horizontal* microinstruction might be completely unencoded, with each control signal assigned to a separate bit position in the microinstruction format (as in Wilkes' original proposal and the simple example given in the definitions section above). This extreme, however, is usually impractical since groups of control signals are mutually exclusive and the resulting representation efficiency is too low. Instead, horizontal microinstructions typically use several operation fields, each encoding one of several mutually-exclusive choices. For example, the System/360 Model 50 uses 25 separate fields in each 85-bit microinstruction. The multiple operation fields allow the explicit specification of parallel activities within the datapath.

Branching in horizontal microprograms is also more complicated than in the vertical case. Each horizontal microinstruction is likely to have at least one branch condition and associated target address, and the microinstruction format may include an explicit, unconditional next-address field. The programming effect is more like developing a directed graph (with various cycles in the graph) of groups of control signals.

Horizontal microprogramming is therefore less familiar to traditional programmers and can be more error-prone. However, horizontal microprogramming typically provides better performance because of the opportunities to exploit parallelism within the datapath and to fold the microprogram branching decisions into the same clock cycle in which the datapath actions are being performed. Programming for horizontal microcode engines has been compared to the programming required for VLIW processors.

A combination of vertical and horizontal microinstructions in a two-level scheme is called *nanoprogramming* and was used in the Nanodata QM-1 and the Motorola 68000. The QM-1 used a 16K word microinstruction control store of 18 bits per word and a 1K word nanoinstruction control store of 360 bits per word [Nanodata]. The microinstructions essentially formed calls to routines at the nanoinstruction level. The horizontal-style nanoinstruction was divided into five 72-bit fields (see Figure 11).

```
SRDAI: "SHIFT RIGHT DOUBLE ARITHMETIC IMMEDIATE"

.... FETCH, KSHC=RIGHT+DOUBLE+ARITHMETIC+RIGHT CTL,  SH STATUS ENABLE
           KALC=PASS LEFT,                          ALU STATUS ENABLE
S...    A->FAIL,    A->FSID, B->KSHR, CLEAR CIN
.S..    A->FAOD, INCF->FSID, A->FSOD
..S. INCF->FSOD
...X GATE ALU, GATE SH, ALU TO COM
```

Figure 11. Horizontal microcode: Nanodata example

The first field (called K and denoted in Figure 11 by the four periods) holds a 10-bit branch address (`FETCH` in the example in Figure 11), various condition-select subfields, and some control subfields. The remaining four fields (called T1-T4 and appearing on separate lines in the figure) specify the particular microoperations to be performed. Each T field contains 41 subfields.

A nanoinstruction was executed on the QM-1 in four phases: the K field was continuously active, and the T fields were executed in sequence, one per phase (note the staggered S and X characters in Figure 11 for the T1-T4 fields). This approach allows one 360-bit nanoinstruction to specify the equivalent of four 144-bit nanoinstructions (i.e., the K field appended with each particular T field in turn). Sequencing subfields within the T fields provide for repetitive execution of the same nanoinstruction until certain conditions become true, and other subfields provide for the conditional skipping of the next T field.

Interpretation of microinstructions can become more complicated than interpretation of normal instructions due to some of the following features:

- *bit steering*, in which one field's value determines the interpretation of other field(s) in the microinstruction;
- *environment substitution*, in which fields from user-level registers are used as counts or operand specifiers for the current microinstruction;
- *residual control*, in which one microinstruction deposits control information in a special setup or configuration register that governs the actions and interpretation of subsequent microinstructions;
- *polyphase specification*, in which different fields in a microinstruction are active in different clock phases or clock cycles (e.g., Nanodata QM-1 as described above); and,
- *multiphase specification*, in which the time duration (the number or length of clock phases or cycles) for an ALU activity would be explicitly lengthened to accommodate the time required (e.g., such as lengthening the time to account for carry propagation during an addition, also a feature of the QM-1).

Sequencing of microinstructions is also complicated by a basic rule of avoiding any microinstruction address additions except as simple increments to counters. Thus, address field bits may be or'ed into the control store address register or conditionally replace all or part of the control store address register. Alternatively, a microinstruction might use multiple next address fields.

Subroutine calls may also be available at the microprogram level; however, the nesting level is typically restricted by the size of a dedicated control store return address stack.

For use in decoding macroinstructions, the initial mapping to a particular microprogram can be accomplished by one of these methods:

- appending low-order zeroes to the macroinstruction opcode, implying that the microcode routines are organized as fixed-offset segments within the control store (as in Wilkes' original scheme)
- appending high-order zeroes to the macroinstruction opcode, implying the use of a jump table in low memory
- using a table lookup via a special decoding ROM or PLA (as in the 68000)

Microprogramming Tools and Languages

Most environments for microprogramming provided standard tools, such as micro-assemblers, linkers, loaders, and machine simulators (which may provide various forms of debugging and tracing facilities). These tools were, of necessity, machine dependent. Over the years, many efforts were made toward developing high-level microprogramming languages (HLMLs), ranging from machine-dependent approaches to machine-independent approaches. The former typically could not be ported to other implementations or had large development costs required for porting; the latter typically generated rather inefficient microcode. Thus, over the years there arose no widely accepted microprogramming language.

One of the first published discussions of an attempt to provide a higher-level representation for microcode was Stuart Tucker's description of an internal IBM flowchart language for System/360 microcode [Tuc67]. Other early work included Samir Husson's 1970 text in which he described a Fortran-like HLML [Hus70], and Dick Eckhouse's 1971 Ph.D. dissertation in which he described a PL/I-like language he called MPL. Other notable efforts include STRUM in 1976, developed by David Patterson with the goal of microcode verification. STRUM was a Algol-like, block-structured language that provided loop assertions (**assert**) and pre- and post-procedure assertions (**assume** and **conclude**). The first language implemented on multiple machines (specifically the DEC VAX and the HP 300) was done by Patterson and colleagues at DEC in 1979 and was named YALL for "Yet Another Low Level Language". See Subrata Dasgupta's and Scott Davidson's survey papers for more information on these efforts [Das80, Dav86].

A major design question for any HLML is whether (and, if so, how) to express parallelism and timing constraints. On one hand, a language like Dasgupta's S* (ca. 1978) uses a Pascal-like syntax but includes two timing control structures: **cocycle** and **coend**, which identify parallel operations active in the same clock cycle; and, **stcycle** and **stend**, which identify parallel operations that start together in a new clock cycle (but do not necessarily have the same duration). On the other hand, Preston Gurd wrote a microcode compiler using a subset of the C programming language as source statements (Micro-C, ca. 1983) for his masters thesis at University of Waterloo. Gurd found that microprograms written without explicit constraints were easier to understand and debug. In fact, debugging could take place on any system having a C compiler and make use of standard debuggers. Moreover, providing a microcode compiler for an existing HLL allows preexisting codes to be converted to microcode without further programming effort.

Because the efficiency of microcode is so important to the performance of a system, a major goal of microprogrammers and consequently their tools has been optimization. This optimization is typically called *microcode compaction* and attempts to ensure both that the microprograms will fit within a given size control store and that the microprograms execute with maximum performance [And80,Hab88]. Such optimization may reorder the microoperations, while not violating data and resource dependencies. One advanced technique, called *trace scheduling* was developed by Josh Fisher for horizontal microprograms [Rau93]. Fisher's approach is to predict a likely path of execution (called a "trace") and perform major optimization on that path. Off-trace paths will likely require *compensation code* to retain correctness. Adding code to infrequently-taken paths costs relatively little in execution time, but this optimization may be constrained by control store size limits. (Trace scheduling was the foundation of Fisher's later work with VLIW.)

Summary Evaluation of Microprogramming

- Advantages
  - systematic control unit design
  - ease of implementing a code-compatible family of computers
  - ability to emulate other computer systems
  - ability to debug system using microdiagnostics

- field modification without rewiring or board replacement
- low marginal cost for additional function, given that additional microinstructions fit within the control store or do not otherwise impact the packaging constraints
- reconfigurable computing using a writable control store

- Disadvantages
  - uneconomical for small systems due to relatively high fixed cost of control store and microinstruction sequencing logic
  - another level of interpretation with associated overhead
  - instruction cycle time limited by control store access time
  - limited support tools

References

[Anc86] F. Anceau. The Architecture of Microprocessors. Workingham, England: Addison-Wesley, 1986.

[And80] M. Andrews. Principles of Firmware Engineering in Microprogram Control. Potomac, MD: Computer Science Press, 1980.

[Agr76] A.K. Agrawala and T.G. Rauscher. Foundations of Microprogramming. New York: Academic Press, 1976.

[Atk74] T. Atkinson, "Architecture of Series 60/Level 64," Honeywell Computer Journal, v. 8, n. 2, 1974, pp. 94-106.

[Bad92] R. Badeau, et al., "A 100 Mhz macropipelined CISC CMOS microprocessor," IEEE Journal of Solid-State Circuits, v. 27, n. 11, 1992, pp. 1585-1598.

See also Bob Supnik's archive of the NVAX microcode listings.

[Bar92] R.J. Baron and L. Higbie. Computer Architecture: Case Studies. Reading, MA: Addison-Wesley, 1992.

[Bel78] C.G. Bell, J.C. Mudge, and J.E. McNamara. Computer Engineering: A DEC View of Hardware Systems Design. Bedford, MA: Digital Press, 1978.

[Bha91] D. Bhandarkar and D.W. Clark, "Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization," Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems [ASPLOS], 1991, pp. 310-319.

[Bla08] D. Black-Schaffer, J. Balfour, W.J. Dally, V. Parikh,and J. Park, "Hierarchical Instruction Register Organization," Computer Architecture Letters, v. 7, n. 2, 2008. Available on-line as http://cva.stanford.edu/projects/elm/pubs/cal_Jul_2008.pdf.

[Burroughs] B1700 manuals. Available on-line at http://www.textfiles.com/bitsavers/pdf/burroughs/B1700/

[Che08] A.C. Cheng, "Amplifying Embedded System Efficiency via Automatic Instruction Fusion on a Post-Manufacturing Reconfigurable Architecture," Proceedings of the International Symposium on Quality Electronic Design [ISQED], 2008, pp. 744-749.

See also, among other papers by Cheng and colleagues on Framework-based Instruction-set Tuning Synthesis (FITS), A.C. Cheng and G.S. Tyson, "High-Quality ISA Synthesis for Low-Power Cache Designs in Embedded Microprocessors," IBM Journal of Research and Development, v. 50, n. 2/3, 2006, pp. 299-309. Available on-line as http://www.research.ibm.com/journal/rd/502/cheng.html.

[Cir95] J. Circello, "Coldfire: A Hot Processor Architecture," Byte, v. 20, no. 5, 1995, pp. 173-174.

[Das80] S. Dasgupta, "Some Aspects of High Level Microprogramming," ACM Computing Surveys, v. 12, n. 3, 1980, pp. 295-323.

[Dav86] S. Davidson, "Progress in High-Level Microprogramming," IEEE Software, v. 3, n. 4, 1986, pp. 18-26.

[Dor88] R.W. Doran, "Amdahl Multiple-Domain Architecture," IEEE Computer, v. 21, no. 10, 1988, pp. 20-28.

See also US Patent 4,967,342, "Data Processing System having Plurality of Processors and Channels Controlled by Plurality of System Control Programs through Interrupt Routing," 1990.

[Gon00] R.E. Gonzalez, "Xtensa: A Configurable and Extensible Processor," IEEE Micro, v. 20, n. 2, 2000, pp. 60-70.

[Hab88] S. Habib (ed.), Microprogramming and Firmware Engineering Methods. New York: van Nostrand, 1988.

[Ham90] V.C. Hamacher, Z.G. Vranesic, and S.G. Zaky. Computer Organization (3rd ed.). New York: McGraw-Hill, 1990.

[Hel05] L.C. Heller and M.S. Farrell, "Millicode in an IBM zSeries processor," IBM Journal of Research and Development, v. 48, n. 3/4, 2005, pp. 425-434. Available on-line as http://www.research.ibm.com/journal/rd/483/heller.html.

See also J. Maergner and H.R. Schwermer, "I370 - A New Dimension of Microprogramming," ACM SIGMICRO Newsletter, v. 19. n. 3, 1988, pp. 24-31.

[Hop00]

W. Hopmann, "The G1 and the Göttingen Family of Digital Computers," in R. Rojas and U. Hashagen, The First Computers - History and Architectures. Cambridge, MA: MIT Press, 2000, pp. 295-313.

[Hus70] S.H. Husson, Microprogramming: Principles and Practice. Englewood Cliffs, NJ: Prentice Hall, 1970.

[Joh71] A.M. Johnson, "The Microdiagnostics for the IBM System 360 Model 30," IEEE Transactions on Computers. v. C-20, n. 7, 1971, pp. 798-803.

[Lar82] J.R. Larus, "A Comparison of Microcode, Assembly Code, and High-Level Languages on the VAX-11 and RISC I," ACM Computer Architecture News, v. 10, n. 5, 1982, pp. 10-15.

[Lav78] S.H. Lavington, "The Manchester Mark I and Atlas: A Historical Perspective," Communications of the ACM, v. 21, n. 1, 1978, pp. 4-12.

See also http://www.chilton-computing.org.uk/acl/technology/atlas/p007.htm.

[McC65] M.A. McCormack, T.T. Schansman, and K.K. Womack, "1401 Compatibility Feature on the IBM System/360 Model 30," Communications of the ACM, v. 8, n. 12, 1965, pp. 773-776.

[McK79] J. McKevitt and J. Bayliss, "New Options from Big Chips," IEEE Spectrum, v. 16, n. 3, 1979, pp. 28-34.

[Mic80] J. Mick and J. Brick. Bit-Slice Microprocessor Design. New York: McGraw-Hill, 1980.

[Microdata] Microdata 1600 manuals. Available on-line at http://www.textfiles.com/bitsavers/pdf/microdata/

[Mye78] G.J. Myers. Advances in Computer Architecture. 2nd ed. New York: Wiley, 1978.

[Nanodata] Nanodata QM-1 manuals. Available on-line at http://www.textfiles.com/bitsavers/pdf/nanodata/

[Opl67] A. Opler, "Fourth-Generation Software," Datamation, v. 13, n. 1, 1967, pp. 22-24.

[Pad81] A. Padegs, "System/360 and Beyond," IBM Journal of Research and Development, v. 25, n. 5, 1981, pp. 377-390. Available on-line as http://www.research.ibm.com/journal/rd/255/ibmrd2505D.pdf.

[Pat98] D.A. Patterson and J.L. Hennessy. Computer Organization and Design: The Hardware / Software Interface (2nd ed.). San Mateo, CA: Morgan Kaufmann, 1998.

[Phi79] M. Phister, Jr. Data Processing Technology and Economics (2nd ed.). Bedford, MA: Digital Press, 1979.

[Pug91] E.W. Pugh, L.R. Johnson, and J.H. Palmer. IBM's 360 and Early 370 Systems. Cambridge, MA: MIT Press, 1991.

[Rau93] B.R. Rau and J. Fisher, "Instruction-Level Parallel Processing: History, Overview, and Perspective," Journal of Supercomputing, v. 7, 1993, pp. 9-50.

[Ren56] W. Renwick, "EDSAC II," Proceedings of the IEE, v. 103B, n. 2, April 1956, pp. 277-278.

[Res05] M. Reshadi, B. Gorjiara, and D. Gajski, "Utilizing Horizontal and Vertical Parallelism Using a No-Instruction-Set Compiler and Custom Datapaths", Proceedings of the International Conference on Computer Design [ICCD], 2005, pp. 69-76.

See also http://www.ics.uci.edu/~nisc/.

[Sal76] A.B. Salisbury. Microprogrammable Computer Architectures. New York: Elsevier, 1976.

[Shi79] M. Shima, "Demystifying Microprocessor Design," IEEE Spectrum, v. 16, n. 7, 1979, pp. 22-30.

[Shr98] B. Shriver and B. Smith. The Anatomy of a High-Performance Microprocessor: A Systems Perspective. Los Alamitos, CA: IEEE Computer Society Press, 1998.

[Str78] S. Stritter and N. Tredennick, "Microprogrammed Implementation of a Single Chip Microprocessor," Proceedings of the 11th Annual Microprogramming Workshop [MICRO-11], 1978, pp. 8-16.

See also US Patent 4,307,445, "Microprogrammed Control Apparatus having a Two-Level Control Store for Data Processor," 1981, and US Patent 4,325,121, "Two-Level Control Store for Microprogrammed Data Processor," 1982.

[Tre88] N. Tredennick, "Experiences in Commercial VLSI Microprocessor Design," Microprocessors and Microsystems, v. 12, n. 8, 1988, pp. 419-432.

[Tre82] N. Tredennick, "The Cultures of Microprogramming," Proceedings of the 15th Annual Microprogramming Workshop [MICRO-15], 1982, pp. 79-83.

[Tuc65] S.G. Tucker, "Emulation of Large Systems," Communications of the ACM, v. 8, n. 12, 1965, pp. 753-761.

[Tuc67] S.G. Tucker, "Microprogram Control for System/360," IBM Systems Journal, v. 6, n. 4, 1967, pp. 222-241. Available on-line as http://www.research.ibm.com/journal/sj/064/ibmsj0604B.pdf.

[Tuc99] S.G. Tucker, personal communication, February 1999.

[Web97] C.F. Webb and J.S. Liptay, "A High-Frequency Custom CMOS S/390 Microprocessor," IBM Journal of Research and Development, v. 41, n. 4/5, 1997, pp. 463-473. Available on-line as http://www.research.ibm.com/journal/rd/414/webb.html.

[Web08] C.F. Webb, "IBM z10: The Next-Generation Mainframe Microprocessor," IEEE Micro, v. 28, n. 3, 2008, pp. 19-29.

descriptions of subsequent IBM z196 design (July 2010):

- "The z196 offers 984 instructions, of which 762 are implemented entirely in hardware." [p. 57, IBM zEnterprise System Technical Introduction, August 2010 (draft Redbook pdf)]
- "There are 246 complex instructions executed by millicode and another 211 complex instructions cracked into multiple RISC like operations." [p. 72, IBM zEnterprise System Technical Guide, August 2010 (draft Redbook pdf)]
- *[note: 762 + 246 = 1008]*

[Whe92] D.J. Wheeler, "The EDSAC Programming Systems," IEEE Annals of the History of Computing, v. 14, n. 4, 1992, pp. 34-40.

[Wil51] M.V. Wilkes, "The Best Way to Design an Automated Calculating Machine," Manchester University Computer Inaugural Conf., 1951, pp. 16-18.
- Reprinted in: MV Wilkes, "The Genesis of Microprogramming," IEEE Annals of the History of Computing, v. 8, n. 3, 1986, pp. 116-126.

[Wil53] M.V. Wilkes and J.B. Stringer, "Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer," Proceedings of the Cambridge Philosophical Society, v. 49, 1953, pp. 230-238.
- Reprinted as chapter 11 in: D.P. Siewiorek, C.G. Bell, and A. Newell. Computer Structures: Principles and Examples. New York: McGraw-Hill, 1982.
- Also reprinted in: M.V. Wilkes, "The Genesis of Microprogramming," IEEE Annals of the History of Computing, v. 8, n. 3, 1986, pp. 116-126.

[Wil58] M.V. Wilkes, W. Renwick, and D.J. Wheeler, "The Design of the Control Unit of an Electronic Digital Computer," Proceedings of the IEE, v. 105B, n. 20, 1958, pp. 121-128.

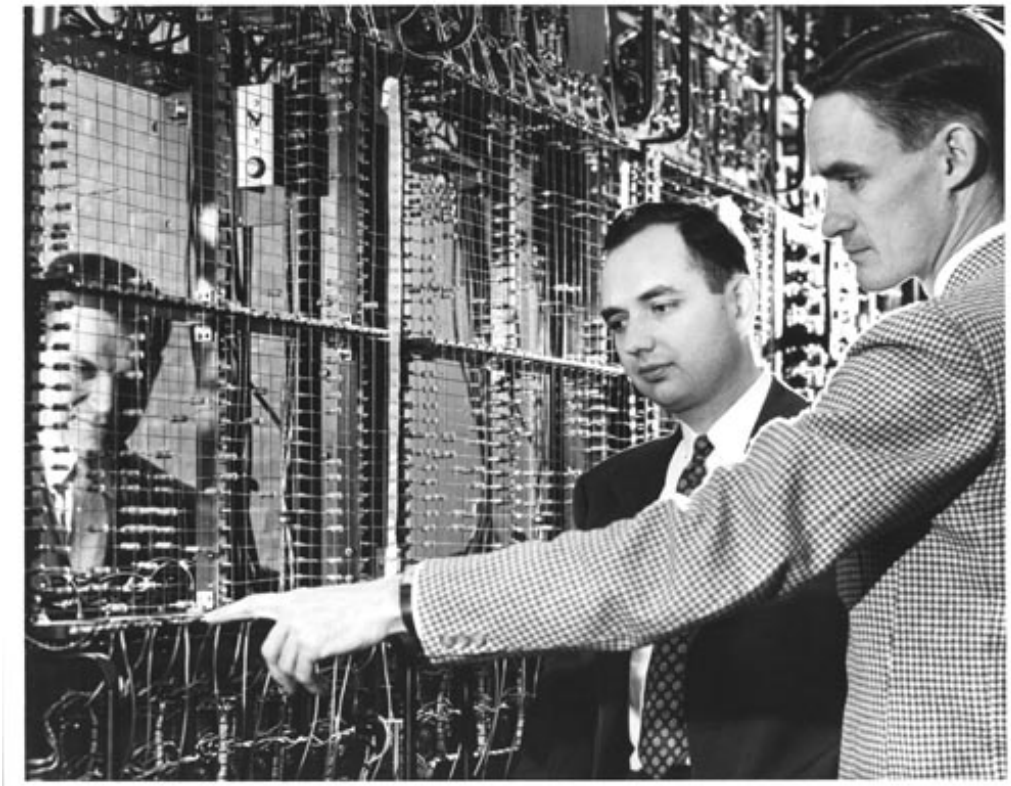
[Wil85] M.V. Wilkes, Memoirs of a Computer Pioneer. Cambridge, MA: MIT Press, 1985.

[Wil92] M.V. Wilkes, "EDSAC 2," IEEE Annals of the History of Computing, v. 14, n. 4, 1992, pp. 49-56.

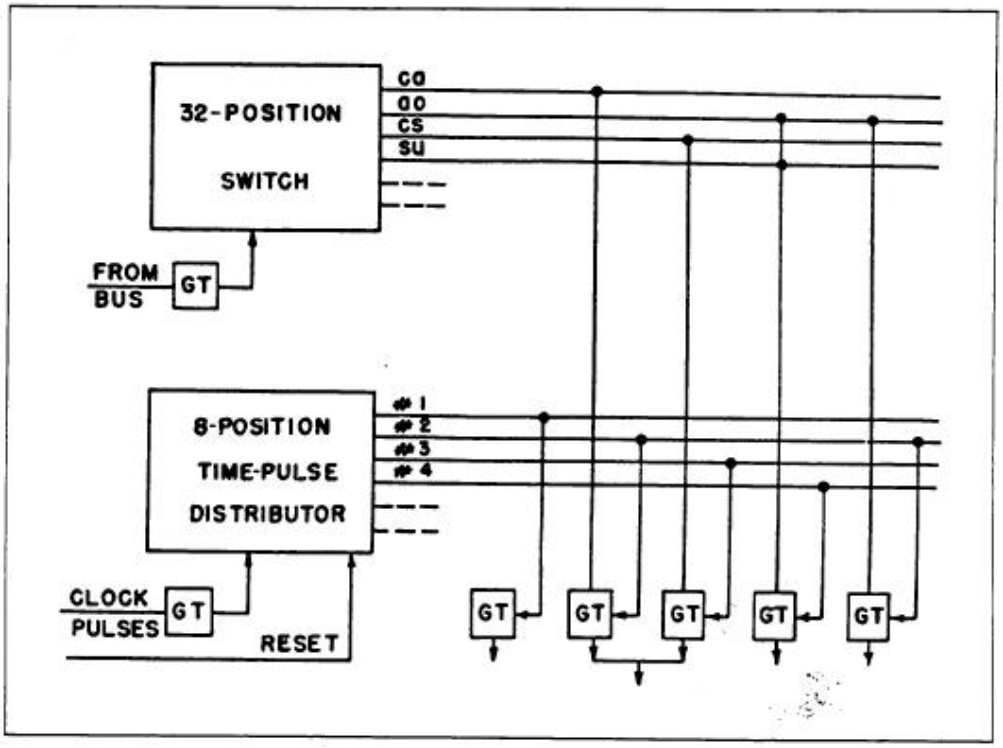
Appendix Section: Control Design of the MIT Whirlwind Computer

Whirlwind was an early real-time computer developed at MIT. Its development started in late 1945, and it was operational by 1950. The Whirlwind had a very systematic approach to control design, with control signals implemented using three diode matrices: an Operation Matrix, an Operation Timing Matrix, and a Program Timing Matrix. (See R.R. Everett and F.E. Swain, "Whirlwind I Computer Block Diagrams," Report R-127, MIT Servomechanisms Laboratory, 1947, Figures 49, 54, 55, and 58.) These diode matrices were called the "control store". One of its main designers, Robert Everett, described the Whirlwind as having "a very flexible control to permit the addition and modification of instructions. ... Soldering in diodes changed existing instructions or put in new ones." [R.R. Everett, "Whirlwind," in N. Metropolis, J. Howlett, and G.C. Rota (eds.), A History of Computing in the Twentieth Century, Academic Press, 1980, pp. 365-384.]

A photo at the Computer History Museum shows Normal Taylor (behind the panel), Robert Everett, and Gus O'Brien examining one of the control matrices.



In the 1947 report and in 1951 conference presentations (e.g., see Everett's paper reprinted in Bell and Newell), the following simplified diagram was used to convey the general approach to generating control signals in Whirlwind. The top decoder selects one of 32 instruction lines from a 5-bit opcode. An instruction line can be combined with any of eight timing pulses to generate a control pulse.



[Figure 4 of Chapter 6 from Gordon Bell's on-line version of Computer Structures book]

Saul Rosen reported in his 1969 paper "Electronic Computers: A Historical Survey" (appearing in ACM Computing Surveys):

> One of the major contributions of the Whirlwind project was a set of detailed, well annotated logical diagrams of the computer. Although not formally published, they achieved fairly wide private circulation and helped to educate many early workers in the computer field (including the author [i.e., Rosen himself]).

Wilkes visited Whirlwind in September 1950. See pp. 176ff in his book, Memoirs of a Computer Pioneer, MIT Press, 1985. An excerpt from pp. 177-178:

> It was during the trip that I have just been describing, and the few months that immediately followed it, that my ideas on the subject of microprogramming crystallized. We had tried to make the design of the control sections of the EDSAC as systematic as possible, but they contained a great deal of what is now called random logic. I felt that there must be a way of replacing this by something more systematic, perhaps along the lines of the configurations of diodes used for decoding the function digits and subsequently re-encoding them to drive various gates throughout the computer. My voyage across the Atlantic in the RMS Newfoundland gave me the opportunity to do some quiet thinking on the subject and later, when I saw the Whirlwind computer, I found that it did indeed have a centralized control based on the use of a matrix of diodes. It was, however, only capable of producing a fixed sequence of 8 pulses - a different sequence for each instruction, but nevertheless fixed as far as a particular instruction was concerned. It was not, I think, until I got back to Cambridge that I realized that the solution was to turn the control unit into a computer in miniature by adding a second matrix to determine the flow of control at the micro-level and by providing for conditional micro-instructions. Sometime during the winter the ideas fell into shape and I gave an impromptu lecture to my colleagues. I subsequently wrote them up and included them in a lecture that I gave in July 1951 to a conference held at Manchester University to mark the completion of the Ferranti Mark I computer.

See M.V. Wilkes, "The Growth of Interest in Microprogramming: A Literature Survey," ACM Computing Surveys, v.1. n. 3., 1969, pp. 139-145, in which he also describes Whirlwind's control store, as well as his comments on Charles Babbage, Whirlwind, and microprogramming in M.V. Wilkes, "The Design of a Control Unit - Reflections on Reading Babbage's Notebooks," IEEE Annals of the History of Computing, v. 3, n. 2, 1981, pp. 116-120.

Appendix Section: Other Early Examples of Hardwired Control Design

- **EDVAC**

  T. Kite Sharpless gave two lectures on the early EDVAC serial design as part of the Moore School Lectures at the Moore School of Electrical Engineering at the University of Pennsylvania in summer 1946. The diagrams accompanying the lectures depict the general idea of a 10 x 25 switch matrix used as an instruction decoder, but detailed switch connections are not given. The decoder accepts a five-bit opcode and generates "A", "S", and "M" signals for the add, subtract, and multiply instructions (i.e., "orders"), respectively. Five other signals are also depicted as being generated by the decoder.

  A separate diagram of the ALU (i.e., "high-speed binary computer") depicts the use, but not generation, of more qualified A, S, and M control signals, such as "A1P2" meaning addition instruction, minor cycle 1, pulse 2, and "M1P2-7" meaning multiplication instruction, minor cycle 1, pulses 2 through 7.

  See Lecture 47, M. Campbell-Kelly and M.R. Williams (eds.), The Moore School Lectures, MIT Press, 1985, pp. 546-560 (with two fold-out diagrams). See also this early hand-drawn EDVAC diagram.

  David Brown and Robert Everett from the MIT Whirlwind project were invited attendees for the lecture series, and Jay Forrester attended a few of the lectures also. Maurice Wilkes was another invitee but was only able to attend the final two weeks.

- **Manchester University SSEM**

  Frederic Williams, Tom Kilburn, and Geoff Tootill built a small-scale experimental computer at Manchester University (U. K.) in 1948 to test out the Williams tube memory design. A complete block diagram, including the control signals, is given for the Small Scale Experimental Machine (nicknamed "Baby") in Figure 9 of F.C. Williams, T. Kilburn, and G.C.

Toothill, "Universal High-Speed Digital Computers: A Small-Scale Experimental Machine," Proceedings of the Institution of Electrical Engineers (IEE), v. 98, part II, n. 61, February 1951, pp. 13-28. The control signals are labeled as the six signals "0/13", "1/13", "0/14", "1/14", "0/15", and "1/15", which represent whether the individual bits in the opcode (i.e., the "function number" in bit positions 13-15) are 0 or 1. There is no switch matrix. Instead, subsets of these six signals are combined together at different points in the datapath to perform control functions.

- **EDSAC**

  A reprint of Wilkes and Renwick's 1949 conference paper on EDSAC is available on-line in Peter Robinson and Karen Jones, "EDSAC 99 Commemorative Booklet," University of Cambridge Computer Laboratory, April 1999. Block diagrams from the reprint appear on pages 61 and 62 of the booklet and show the active control signals for instruction fetch and for execution of an addition operation, respectively.

  A case study of the invention of microprogramming by Wilkes is a major part of Subrata Dasgupta, Creativity in Invention and Design: Computational and Cognitive Explorations of Technological Originality, Cambridge University Press, 1994. Chapter 3 discusses both EDSAC and EDSAC 2.

- **Eckert and Mauchly BINAC**

  J. Presper Eckert and John Mauchly left the Moore School at the University of Pennsylvania in 1946 and formed the Eckert-Mauchly Computer Corporation. EMCC accepted an order for the large UNIVAC machine in 1948 and also an order for a smaller BINAC machine, which was completed in 1949. Control signals in BINAC were generated by using two diode matrices, called "function tables". These matrices are shown in detail in A.A. Auerbach, J.P. Eckert, R.F., Shaw, J.R. Weiner, and L.D. Wilson, "The BINAC," Proceedings of the IRE, v. 40, n. 1, January 1952, pp. 12-29.

- **IAS**

  The "Third Interim Progress Report on the Physical Realization of an Electronic Computing Instrument" indicates that work on the "control organ" for the IAS machine was "just starting intensively" by January 1948 (p. 111). Indeed, by the time of the sixth interim report in 1951, the control circuitry was still described as "fragmentary". Gerald Estrin discusses asynchronous timing chains and walks through the actions of an add instruction in "A Description of the Electronic Computer at the Institute for Advanced Studies," Proceedings of the 1952 ACM National Meeting, Toronto, 1952, pp. 95-109. Details of the IAS design were also presented in the 1954 "Final Progress Report on the Physical Realization of an Electronic Computing Instrument"

- **IBM 701**

  The control signal logic for the IBM 701 appears in the various drawings for US Patent 2,974,866, J.A. Haddad, R.K. Richards, N. Rochester, and H.D. Ross, Jr., "Electronic Data Processing Machine," filed March 30, 1954, and issued March 14, 1961.

  Frank Beckman, Fred Brooks, and William Lawless of IBM discussed synchronous and asynchronous control in "Developments in the Logical Organization of Computer Arithmetic and Control Units," Proceedings of the IRE, v. 94, n. 1, January 1961, pp. 53-66:

  > The classical synchronous techniques for sequencing the phases of an operation use a latch ring or counter-decoder for distributing a sequence of timed pulses from an oscillator or other clock along appropriate gating lines. These techniques are still very widely used.
  >
  > As old, and also widely used today, are the asynchronous techniques by which the completion of each operation phase causes generation of a pulse which initiates the next phase. In a widely-used variant of asynchronous technique, each operation is independently timed, and each signals the initiation of the next.

The following table compares several designs of the late 1940s and 1950s.

	EDSAC	BINAC	Whirlwind	IAS	IBM 701	EDSAC 2
year operational	1949	1949	1951	1952	1952	1958
word size	36 bits	31 bits	16 bits	40 bits	36 bits	40 bits
clocking	500 KHz	1 MHz	1 MHz	asynchronous	1 MHz	?
addition time	1.4 msec	800 •sec	49 •sec	62 •sec	60 •sec	17-42 •sec (fixed point); 100-170 •sec (floating point)
multiplication time	5.4 msec	1.2 msec	61 •sec	713 •sec	444 •sec	[add product instruction] 270-330 •sec (fixed point); 210-340 •sec (floating point)
memory access time	mercury delay lines	mercury delay lines	16 •sec (CRT); 8 •sec (1953 core memory)	25 •sec (CRT)	12 •sec (CRT)	? •sec (magnetic core)
cost			$3M - $5M	$650K	$10K+/ month lease	
average error-free running period (1955 BRL Report)	N/A	N/A	15 hours	4-8 hours	2-6 hours	N/A

Data sources for table entries:

- M.H. Weik A Survey of Domestic Electronic Digital Computing Systems," U.S. Army Ballistic Research Laboratories, Report No. 971, December 1955.
- S.H. Lavington, Early British Computers, Digital Press, 1980;
- K.C. Redmond and T.M. Smith, Project Whirlwind: The History of a Pioneer Computer, Digital Press, 1980.
- H.D. Huskey, "Hardware Components and Computer Design," in R. Rojas and U. Hashagen (eds.), The First Computers: History and Architectures, MIT Press, 2000.

Acknowledgements

My thanks to Rick Smith for pointing me to the Whirlwind diagrams, and to Stu Tucker for his help in understanding the use of microprogramming in several IBM designs.

[History page] [Mark's homepage] [CPSC homepage] [Clemson Univ. homepage]

mark@cs.clemson.edu