This paper describes the kind of microprogram control that has been used in several models of SYSTEM/360. *A microprogramming language, as well as some of the main techniques used in "assembling" and testing microprograms, are discussed. Applications of microprogram control to the design of emulators, to compatibility features, and to special modifications are summarized.*

Microprogram control for System/360

by S. G. Tucker

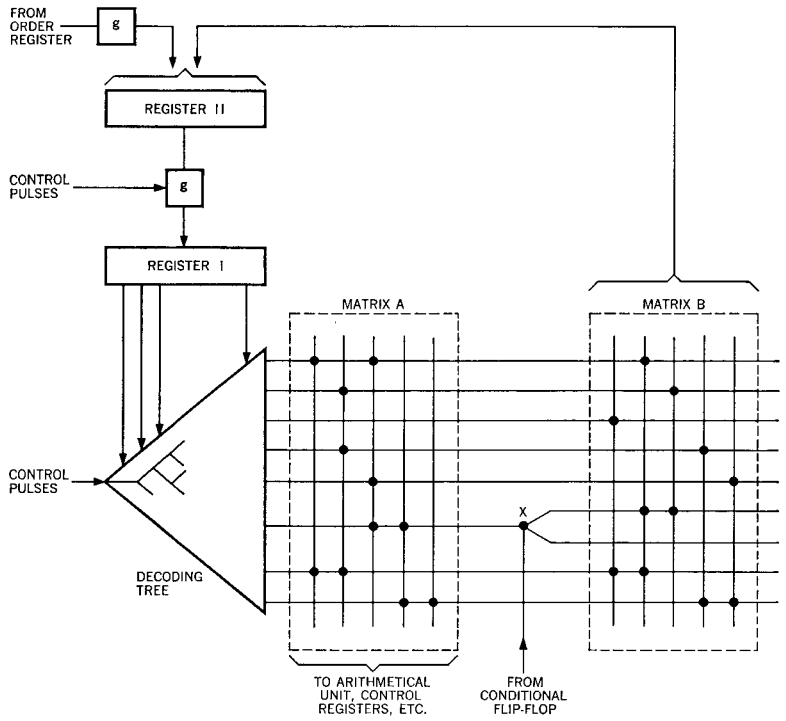
Microprogram control has received a heavy design emphasis in several of the SYSTEM/360 models because it makes an extensive instruction set economically feasible in relatively small computers. Microprogram control offers additional advantages for the designers of emulators and compatibility modifications. This paper reports on the main features of the microprogram design approach employed in SYSTEM/360. To help place the subject in perspective, some of the literature on microprogramming is briefly reviewed at the outset. The outlines of a microprogram language are also discussed.

Background

The idea of microprogramming is normally attributed to Wilkes.[1–3] He was clearly concerned with the rather ad hoc manner in which computer controls had previously been designed. Especially for computers with complex instruction sets, he foresaw both design and maintenance benefits in a more orderly approach to the design of the controls.

He noted that all of the instructions of the typical digital computer were constructed from a number of more elementary operations. These elementary operations, he noted, consisted essentially of transfers of numbers from one register to another, either direct or via an adder, and with or without shifting. The mechanism he suggested for controlling a sequence of these elementary orders to form a machine instruction is shown in Figure 1. A micro-order is placed in Register I, the output of which feeds a decoder. One and

Figure 1 Wilkes' microprogram control*



*USED WITH THE PERMISSION OF PROFESSOR WILKES AND THE UNIVERSITY OF MANCHESTER

only one output of the decoder is raised, in accordance with the micro-order in Register I when a clock pulse is applied. The output of the decoder drives two matrices, Matrix A and Matrix B. (These were originally considered diode matrices, but Wilkes suggested that other implementations might be reasonable.) When a decoder line to Matrix A is driven, output pulses governed by the diode pattern cause the appropriate elementary operations to take place in the arithmetic unit. Similar pulses from Matrix B set Register II. At the start of the next cycle, Register II is gated into Register I, which provides the address of the next micro-order. Thus Matrix A provides control lines to the arithmetic unit, whereas Matrix B provides the sequence control. The microprogram for controlling a machine instruction is then the sequence of micro-orders which performs the instruction.

A few additional observations by Wilkes and Stringer[2] are well worth noting. One concerned the utility of a conditional branch in the microprogram. A technique suggested for accomplishing this is shown in Figure 1. A line from the decoder is shown splitting before entering Matrix B. The leg which is active is considered to be controlled by flip-flop X, a storage element in the arithmetic unit. Thus, X determines which "next-address" is read out of Matrix B. Branching, it was noted, would be useful for things like testing

multiplier bits to assist in microprogramming a multiply. The use of branching to break out of a microprogram loop was mentioned. It was further noted that a similar branch before Matrix A would permit data-dependent micro-orders to be performed.

Another significant suggestion was that the sequencing of a microprogram for controlling a machine instruction could be started by inserting the operation code of the instruction in Register II (and placing zeros in the low order bits). In this way, Register II could serve directly as the address of the first micro-order of the microprogram for that machine instruction. This would eliminate the need for any form of operation decoder. Wilkes and Stringer also noted that the arithmetic unit could be designed to allow many or few elementary operations. For a comprehensive instruction set, a more flexible switching system (an arithmetic unit allowing more elementary operations) was thought to offer potential savings in the number of micro-orders required to microprogram the instruction set. Other avenues toward efficiency were also suggested,[3] one being to use the same micro-operations for both fixed- and floating-point division reduction, say, switching to and from them by branching. In effect, a microsubroutine is thus accomplished.

Since 1951, the literature has made many references to microprogramming. Various forms of control are considered, most of which stay close to Wilkes' idea except in terms of what the elementary operations should be. Here, there seems to be a wide variation; frequently, it is hard to determine just what elementary operations are assumed. Blankenbaker[4] seems to be considering a system in which the elementary operations are much simpler than those of Wilkes. In fact, the elementary operations seem closer to basic Boolean connectives. In 1958, Dinneen et al.[5] described the logical design of the CG24 computer at MIT Lincoln Labs. The CG24 is controlled by a diode-matrix read-only store. All the elementary operations are described as register-to-register transfers, but in this context the output of an adder with fixed inputs is considered a source register. The diode matrix is read out every four machine cycles and contains enough information to control four cycles.

Kampe (1960) mentioned two techniques of which we shall see more.[6] First, as part of the arithmetic and logical unit being controlled, there is a unit that can form all sixteen Boolean connectives between bit pairs. It is controlled by a four-bit field in the microinstruction (output of Matrix A in Wilkes' model). Here we have, explicitly, the grouping of bits in the output of a read-only store to form a field which controls a particular function. Second, he mentioned groups of bits which have no predetermined use but serve as emit fields (i.e. fields that can be gated into a data path). The microprogrammer is free to use these fields as a source of constants. Graselli (1962) was even more explicit about grouping fields in the microinstruction and then decoding the fields to control particular portions of the data path.[7]

The main motivation in the microprogramming reviewed thus far seems to have been in an organized approach to the design of

controls. An offshoot of this line of thinking seems to have stemmed from the desire to exploit the changeability of the microprogram storage media and economically provide a selection of instruction sets for a given machine. Glantz[8] (1956) suggested a machine that has, in addition to its fixed instruction set, a section of control store that can be written under program control. Thus for various applications, additional performance can be gained by tailoring specialized machine instructions (microprogrammed via the writable control store) to the application.

In at least four machines (References 9–12), the entire control store is described as writable. The term "stored logic" is frequently applied to these machines. The elementary operations resemble simple machine instructions with an operation code and an address field. The PB-440, for example, has 64 micro-orders stored in the low locations of main core storage, which are designed with extra speed to aid performance. The TRW133 uses the term "logand" to describe the steps of a microprogram. Logands become as complicated as a 19-cycle divide.

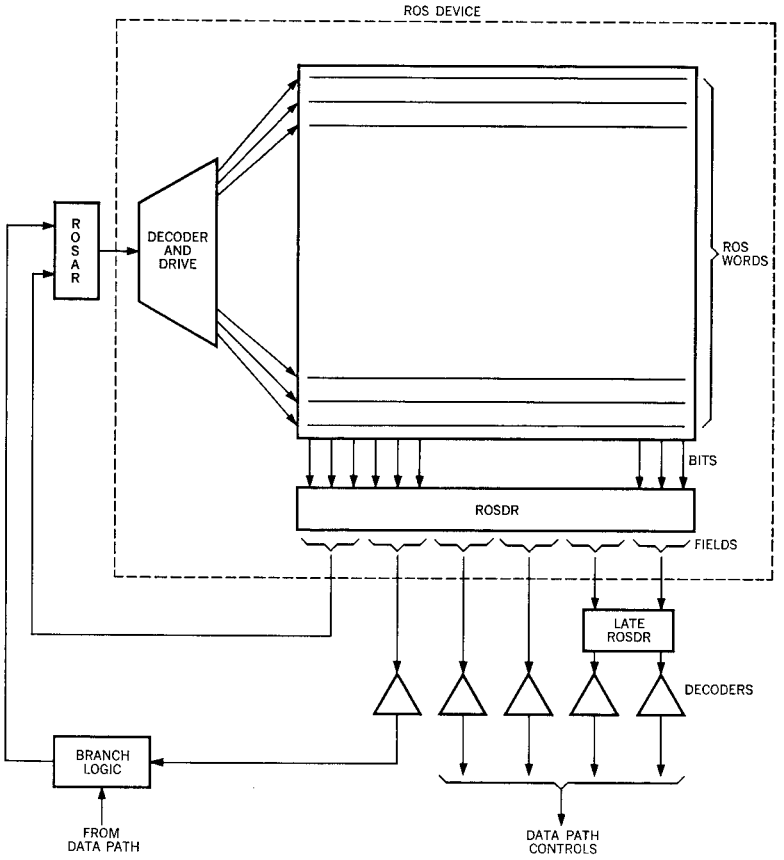
These microprogrammed control systems were designed to meet various objectives (e.g., custom-tailored instruction sets, cost reduction, control system simplification, etc.). This diversity of objectives resulted in such a great variation of "elementary operations" that the control systems, although all microprogrammed, were indeed different.

SYSTEM/360 microprogramming

Microprogramming in the SYSTEM/360 line is not meant to provide the problem programmer with an instruction set that he can custom-tailor. Quite the contrary, it has been used to help design a fixed instruction set capable of reaching across a compatible line of machines in a wide range of performances. The programmer has no way of telling from the external specifications of a SYSTEM/360 processing unit whether or not it is microprogram controlled. The use of microprogramming has, however, made it feasible for the smaller models of SYSTEM/360 to provide the same comprehensive instruction set as the large models.

This is due to the following. As the instruction set of a conventionally controlled processor is made more comprehensive, the cost of the controls goes up in a roughly linear manner. In a Read-Only Store (ROS) microprogram-controlled processing unit, a base cost for the ROS device and supporting hardware must be borne, after which the marginal cost of adding the words needed to microprogram more machine instructions is relatively small. Thus there is a cross-over point. As an instruction set is made more comprehensive, microprogram control becomes more attractive. Thus, in SYSTEM/360, microprogramming stays largely in the province of the engineers designing the processors; added flexibility is passed on to the programmer in the form of a more comprehensive instruction set and special features that become economically feasible with ROS control.

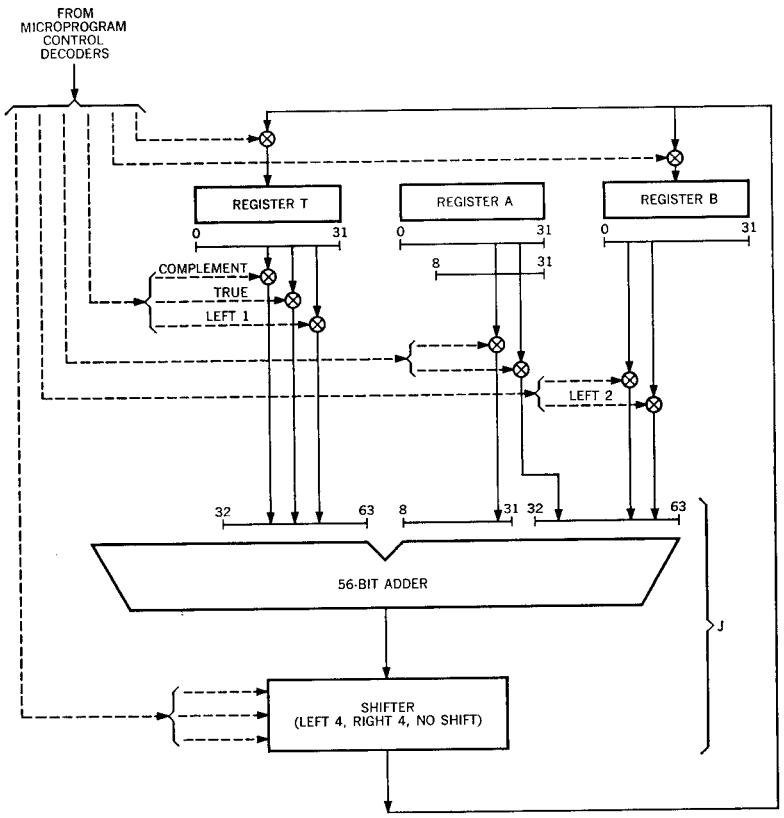
Figure 2 Part of a SYSTEM/360 ROS microprogram control



The microprogram controls used in SYSTEM/360 are very similar in technique to those described by Wilkes. A word from ROS not only controls the arithmetic unit for a single cycle, but also contains information for accessing an ROS word to control the following cycle.

Figure 2 reflects the manner in which SYSTEM/360 ROS control systems are usually shown. The unit within the dotted lines is an ROS device. (Various physical devices are used on the different SYSTEM/360 models.) Typically, they contain a few thousand words from 56 to 100 bits each. The longer words are used on the larger models to control their more complex arithmetic units. In all cases, the ROS cycle time is the same as the basic machine cycle time. (ROS access times are somewhat less than machine cycle times.) On each cycle, a word is read out of the ROS array into the Read-Only Store Data Register (ROSDR), where it is latched up and held for the duration of the cycle. It might be noted at this point that a significant portion of the control section of the machine can be checked by including in each ROS word one or more parity bits. Thus, the contents of the ROSDR can be checked for correct parity on each cycle. This much checking is difficult to accomplish on a conventionally controlled machine.

Figure 3 Part of SYSTEM/360 data path



Thus far, the terms microinstruction, micro-order, and elementary operation have been used very loosely. We can now define a *microinstruction* as a single word, or if you prefer, the information which is contained in a single word of the ROS. A microinstruction controls a single basic processor cycle.

microinstruction

In order to see how the information in a microinstruction is used to control the action of the data path (arithmetic unit), it is instructive to look at a simplified portion of a particular SYSTEM/360 data path. Figure 3 shows three 32-bit registers: T, A, and B. Provision is made to gate the contents of these registers into a 56-bit parallel adder in various ways. The output of the parallel adder is fed into a shifter, which may shift it left four bits, right four bits, or pass it straight through. The shifter output can be returned to either Register T or Register B. On a single machine cycle, we can gate various fields from the registers into the adder, shift the resulting adder output, and return the result to a register.

Table 1 is a list of functions the data path can perform. Each of these is given a symbol. The symbols are based on a few conventions and are not too obscure after one gets a bit used to them. The adder and shifter are generally indicated by the letter J. Entire registers

Table 1 Functions of a SYSTEM/360 data path

Symbol	*Function*
AJ	Gate Register A bit positions 0–31 to adder right side positions 32–63
BJ	Gate Register B bit positions 0–31 to adder right side positions 32–63
BJL2	Gate Register B bit positions 0–31 (shifted left two bit positions) to adder right side positions 30–61
AJ13	Gate Register A bit positions 8–31 to adder right side positions 8–31
TJ	Gate Register T bit position 0–31 in true form to adder left side bit positions 32–63
TJL1	Gate Register T bit positions 0–31 (shifted left one bit position) to adder left side positions 31–62
TJC	Gate Register T bit positions 0–31 in complement form to adder left side positions 32–63
(blank)	Pass adder output straight through shifter
JSR4	Shift adder output right four bit positions
JSL4	Shift adder output left four bit positions
JB	Set shifter output into Register B
JT	Set shifter output into Register T

are referred to by their assigned letters (T, A, and B). A field within a register is referred to by the numbers of the bytes in the field. (A 32-bit register contains four bytes 0, 1, 2, 3.) Thus, the low-order three bytes of Register A are designated A13. The numerals represent the first and last bytes and are understood to be inclusive.

Having gone this far, let us write a microinstruction to add the complement of Register T to Register B and put the result into Register B. It would look like this:

TJC, BJ, JB

Roughly speaking, TJC, BJ, JB are micro-orders. A microinstruction is the collection of micro-orders necessary to control a single machine cycle.

Now let us write a microprogram to add Register A to Register B and put the sum in Register T. This is a two-cycle microprogram, and thus requires two microinstructions.

BJ, JT — Move the contents of Register B to Register T. (The contents of Registers A and B cannot be added directly, because they both gate to the right side of the adder.)

TJ, AJ, JT — Gate the contents of Registers A and T to the Adder, and place the sum in Register T.

grouping

One way we could control our data path is for one bit of the word read out of ROS to control each of the gates in our data path. If the bit were a "1," the gate would be opened, and if it were a "0," it would be left closed. For our data path, that would require twelve bits of the ROS word. For an arithmetic unit as complex as the SYSTEM/360 Model 65, it would require about two-hundred-fifty bits per ROS word. In order to reduce the number of bits required, the bits of

Table 2 Code points for the left side of the adder

Code point	*Symbol*	*Function*
00	(blank)	All zeros to left side of adder
01	TJ	Gate Register T in true form to adder
10	TJL1	Gate Register T shifted left one position to adder
11	TJC	Gate Register T in complement form to adder

Table 3 Code points for the right side of the adder

Code point	*Symbol*	*Function*
000	(blank)	All zeros to right side of adder
001	AJ	Register A to adder right side
010	BJ	Register B to adder right side
011	BJL2	Register B shifted left two bit positions to adder right side
100	AJ13	Register A bit positions 8–31 to adder right side bit position 8–31
101	ABJ	Register A bit positions 8–31 to adder right side bit positions 8–31; Register B bit positions 0–32 to adder right side bit positions 32–63

the ROS words are grouped into fields, and the fields are then decoded into the original control lines. The grouping process consists largely of searching for control lines that are never used at the same time. For example, the three gates into the left side of the adder can never be used at the same time; therefore, they can be controlled by a two-bit field in the ROS, as shown in Table 2. Here, we have a savings of one ROS bit since we use only two bits to control three gates. Nothing is lost since the combinations of three control lines which open more than one gate into the adder are not allowed anyway.

Since this is not always the case, consider the four gates into the right side of the adder. We have to use a three-bit field to allow enough code points for a null state (nothing gated to the right side of the adder) and the four gates individually. Thus, we have used up five of the eight code points available in a three-bit field. However, there is a case where we wish to open two of the gates during the same cycle: namely, Register A bit positions 8 to 31 to adder bit positions 8 to 31 and Register B bit positions 0 to 31 to adder bit positions 32 to 63. This is the manner in which a 56-bit floating-point fraction held in Registers A and B is sent to the adder. Therefore, we pick one of the unused code points and let it cause both gates to open. We can assign it a symbol, say, ABJ. The field which controls inputs into the right side of the adder is shown in Table 3. Note that the other two code points remain undefined.

micro-order

Now, we are able to define what we mean by a *micro-order*. A micro-order is a control function for which a code point in an ROS word field is defined. A microinstruction is a collection of micro-orders encoded in a single ROS word and controlling a single machine cycle.

next address

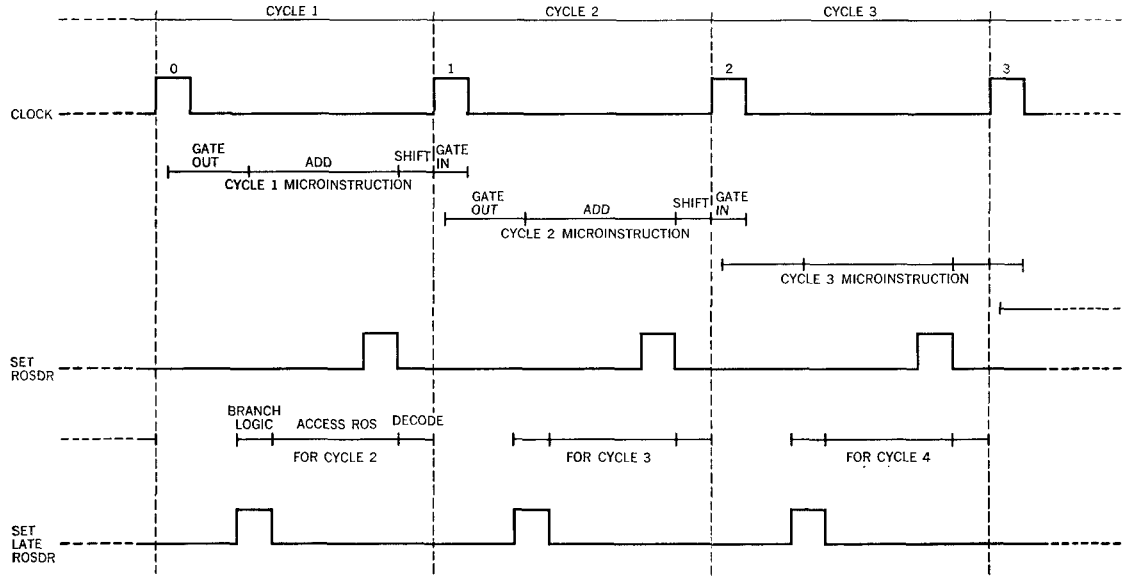
Sequencing ROS control is handled in a manner similar to Wilkes' model. Referring to Figure 2 again, notice that one of the fields of the ROS word is returned to the Read-Only Store Address Register (ROSAR). Basically, the ROSAR is used to address the next ROS word read. In this respect, the "next-address field" corresponds to Matrix B in Wilkes' model. There are, however, a few differences.

The number of bits in the next-address field is less than the number of bits required to address the full ROS. The next-address field is used as the high-order part of an address in ROSAR. The low-order bit or bits must be supplied by other means. Call the low-order bit in the ROSAR the "Y bit." We can then have a field in the ROS word which specifies how the Y bit of the address is to be determined. If we were to choose a three-bit field, there would be eight code points available in the field. Two of the code points, 000 and 001, could be used to force the Y bit directly to 0 and 1 respectively. Thus, by using the next-address field and two code points of the Y bit control field, any address in the ROS could be specified for the next cycle. The other six code points may now be used to specify other means of determining the Y bit. For example, one code point might be used to "make the Y bit 1 if there were a carry out of the adder." In effect, the carry out of the adder is used as the low-order bit of the address of the next ROS word. This provides a two-way branch in the microprogram depending on a bit in the arithmetic unit. The branch condition does not, of course, have to be restricted to a single bit in the arithmetic unit. For example, another code point in the "Y-branch field" might "make the Y bit 1 if bits 0–3 of Register T are all 0's." This would be useful for normalizing a hexadecimal number in Register T. Another type of condition that is frequently tested might "make the Y bit 1 if Register T bit position 0 is the same as Register A bit position 0." Since the leftmost bit in a SYSTEM/360 fixed-point word is a sign bit, this amounts to a "branch-on-signs-alike." The three-bit Y-branch field now looks similar to that shown in Table 4. What we have just done is to specify

Table 4 Code points for the Y-branch field

Code point	*Symbol*	*Function*
000	Y0	Set Y bit to 0 unconditionally
001	Y1	Set Y bit to 1 unconditionally
010	YCJ	Set Y bit to 1 if there is a carry out of the adder
011	YT03Z	Set Y bit to 1 if Register T bit positions 0–3 are all 0
100	YLS	Set Y bit to 1 if Registers T and A have like signs i.e., bit positions 0 are the same for both registers

Figure 4 Timing of microinstruction flow



some "Y-branch" micro-orders. By allowing one bit of the ROSAR to be determined by the Y-branch field, we allow a two-way branch in the microprogram for any data path condition for which we provide a micro-order. This technique is somewhat restrictive compared to the branch Wilkes described in that only a single bit of the next address is altered by the branch, whereas Wilkes' model showed a whole alternate address selected.

Since two-way branching turns out to be insufficient, the control systems used in SYSTEM/360 generally allow two low-order bits to be specified independently by two branch fields. This allows a four-way branch based on two independent conditions.

Normal branching allows up to a four-way branch. Higher-order branches are also used. They are referred to as "function branches." A micro-order that specifies a function branch gates several data conditions to the ROSAR. Function branching, for example, is used during instruction fetch (I fetch) to effect a multiway branch on some of the operation code bits.

timing

Thus far, the ROS control system has been discussed as though, for any given machine cycle, an ROS word were read out at the start of the cycle and held in the ROSDR until the cycle was completed. When designing a machine for the minimum cycle, a particular ROS device, and a particular circuit family in the arithmetic unit, it soon becomes apparent that this is not the best design approach.

The second line in Figure 4 shows the basic machine clock pulse. The data path in Figure 3 relates to the clock pulse as follows. Registers T, A, and B are set by the clock pulse. During the portion of the cycle immediately after the rise of the clock pulse, information flows through dc logic to the adder. During the center portion

of the cycle, information propagates through the adder; at the end of the cycle, it goes through the shifter and back to a register. In general, information must be latched so that it is held at the register input through the duration of the following clock pulse.

Now consider the ROS relative to Cycle 2. If the gating-out of the registers for Cycle 2 is to start during the clock pulse numbered "1," the microinstruction for Cycle 2 must be in the ROSDR some time before clock pulse 1 in order to allow time to decode the micro-orders that control the outgating. Hence, the pulse that sets the ROSDR is shown somewhat before the main clock pulse. The ROS address must be available in the ROSAR long enough before the clock pulse to allow for the access time of the ROS. Furthermore, a data condition that controls a branch must be available before this to allow time for the logic which controls the branch. It is now obvious that if a data condition is to be branched on, it must be available very early in the cycle. Normally, a carry which occurs late in Cycle 1 is saved in a trigger. The Cycle 2 microinstruction may then specify a branch on the carry trigger which determines what microinstruction is read out to control Cycle 3. Needless to say, this can be a frustration to the microprogrammer who normally likes to branch on the result of the current cycle. However, the alternative is a large increase in the basic cycle time.

Where performance is degraded too much by the time taken to branch, the situation can be helped with added hardware. In the example, Cycle 2 could contain a micro-order which is conditional on the state of the carry trigger. Thus, Cycle 2 can perform the recomplement function by means of the following micro-order: "Gate out in true form if the carry trigger is on; gate out in complement form if the carry trigger is off."

Note that, since the ROSDR is set some time before the clock pulse, provision must be made for saving the micro-orders which control the gate-in during the clock pulse. This is done by transferring them into the LATE ROSDR before the ROSDR is set to the next microinstruction (see Figure 2).

Other control techniques

bit saving

Several ROS control techniques are used regularly and seem worthy of brief mention. Considerable effort is normally devoted to trying to reduce the number of bits in the ROS word to reduce the ROS cost. Most of this effort goes into an attempt to find the best possible grouping of micro-orders into fields. Care must be taken not to overly restrict the micro-orders which can be used together or make decoding them too complex. Another bit-saving trick is called "dual usage." The SYSTEM/360 Model 50, for example, has an I/O mode which is entered when the CPU data path is used to handle multiplex channel functions. When in I/O mode, some of the code points take on a different meaning. Here some added complexity in the decoding of micro-orders is traded for a savings in ROS word length.

word saving

Much effort is also devoted to reducing the number of words of ROS required, that is to say, using as few microinstructions as possible. Many times, this not only results in using fewer microinstructions, but also saves cycles. It is the practice to use the same series of microinstructions for things like normalization that are common to more than one instruction. Allowing microprogrammed branches on some operation code bits allows breaking out of the common sequence.

Status triggers (called STATS) are also very useful. Rather than branching on a data condition when it is first available, the condition can be set into a STAT. This frequently allows a branch to be taken on the STAT several cycles later, thus avoiding unnecessary duplication of microinstructions before the branch is actually necessary. Frequently, the microprogrammer may explicitly set a STAT he may want to branch on later. In essence, the STAT allows some of the sequence information to be held external to the ROS control in order to save ROS words.

Another word-saving technique involves branching on a counter's going to zero to break out of a microprogrammed loop. Thus, when a cycle is to be repeated several times, a loop can be used rather than a straight-line coding of the microinstructions. Here again, sequencing information is held external to the ROS control in order to save ROS words. The use of a counter in the microprogram is very similar to the use of an index register in a regular program.

Another handy technique involves the "emit field," i.e., a field in ROS that can be gated into a data path and has no one assigned function. Some provision is made for gating the emit field into the data path. The microprogrammer is then free to put whatever constants or bit patterns he desires in the emit field.

A final-word saving device is called a "function register." The SYSTEM/360 Model 50 has an eight-bit data path which can AND, or OR, or Exclusive-OR. Its function is controlled by a "function register." All the storage-to-storage (SS) logical instructions on the Model 50 are controlled by the same microprogram except for the first microinstruction, which uses the emit field to set the function register appropriately.

In summary, the microinstruction used in SYSTEM/360 controls a single cycle. However, the structure of the data path being controlled tends to be fairly complex, and this is reflected in the structure of the microinstruction. The microprogrammer tailors each microinstruction by choosing the micro-orders he desires. Whereas Wilkes implies that there are fewer microinstructions than machine instructions, this is not the case in SYSTEM/360. For example, the Model 65 has 378 micro-orders, which can be used to form about 5×10^{21} different microinstructions. The Model 30 microprogrammer can construct about 10^{10} microinstructions. It is a matter of conjecture how many of these are reasonable.

Also, the approach was not to design a universal data path and then microprogram the SYSTEM/360 instruction set on it. Rather, the data path was specifically designed for the efficient execution of

SYSTEM/360 instructions. Fields were grouped and micro-orders assigned to permit efficient microprogramming of the SYSTEM/360 instruction set. As design proceeded, the data path and micro-orders were changed when microprogramming showed that poor choices had been made.

Microprogram language

Microprograms for SYSTEM/360 are written in a flowchart language. A box is drawn for each microinstruction and contains the symbols for all the micro-orders that make up the microinstruction.

Using the data path of Figure 3 and the micro-orders developed for it, a three-cycle, three-microinstruction microprogram (to add the contents of Register A to the contents of Register T and place the result shifted right one place into Register T) might appear as shown in Figure 5. Reading the microprogram from left to right, the first microinstruction adds the contents of Registers A and T and puts the result in Register T. The second microinstruction puts the contents of Register T, shifted left one bit position, into Register B. In the simplified data path of Figure 3, the leftmost bit is shifted off the end and lost. The third microinstruction shifts the contents of Register B right two bit positions (i.e. left two bit positions into the adder and right four bit positions in the shifter) and puts the result into Register T.

Notice that where no micro-orders are explicitly stated for one of the fields, the null-state micro-order is implied. For example, in the first two microinstructions, no micro-order is given for the shifter field; this implies a "00" bit coding or no shift. Similarly, the second microinstruction has no micro-order for the right-side adder input field; this implies a "000" coding or zeros into the right side of the adder.

Table 5 A higher-level microprogram language

Symbols	*Higher-level symbols and print positions*			
	1	*2*	*3*	*4*
AJ	A			
TJ		+T		
JT				T
JB				B
TJL1		+TL1		
BJL2	BL2			
JSR4			,R4 →	
Blank in shifter field			→	
Blank in left adder input field		+0		
Blank in right adder input field	0			
TJC		−T		

Figure 5 Simple microprogram

Figure 6 Higher-level microprogram in printer format

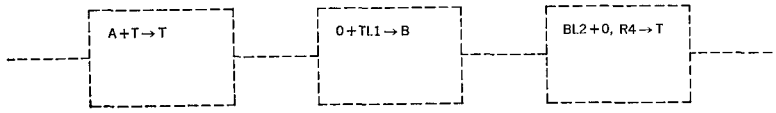
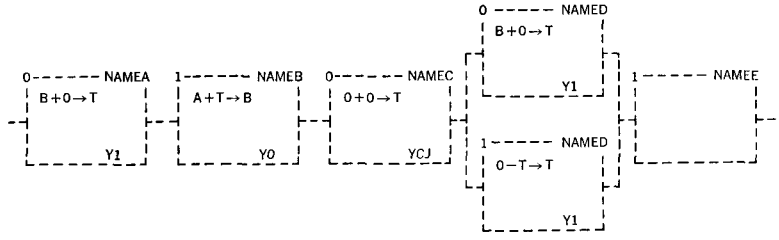


Figure 7 Microprogram branching



The microprogram can be given the appearance of a somewhat higher-level language by a mere change in the symbols chosen for some of the micro-orders. Taking the three-microinstruction example, we can substitute symbols as shown in Table 5.

Each of the micro-orders is part of an expression in a kind of algorithmic language, and our three-microinstruction microprogram now looks as shown in Figure 6. Notice that T is used to indicate both "Register T to adder gate" and the "shifter to Register T gate." With a language of this form, meaning is derived not only from the symbols, but also from their positions within an expression. This becomes a consideration when a program is written to "compile" the microprogram. In practice, microprogram languages for the various SYSTEM/360 models include a number of variations.

Microprogram branching is shown by a split in the flow lines connecting the microinstruction boxes, and branch micro-orders indicate which path is to be taken. Again a simple example is in order, based upon the data path shown in Figure 3 and branch micro-orders shown in Table 4.

Consider the microprogram in Figure 7 to add the contents of Registers A and B and put the result in Register T if there is no overflow, or set Register T to all 1's (maximum value) if there is an overflow. The address of each microinstruction is shown in the top line of the box in two parts. The 0 or 1 on the left indicates, in absolute form, the low-order bit of the address. On the right side of the

box, the high-order bits of the address are represented symbolically. Actual bit patterns are assigned when the microprogram is "compiled." The branch micro-orders (Y0, Y1, YCJ) are shown at the bottom of the blocks.

The microprogram involves the following operations. Movement of the contents of Register B to Register T is indicated by the first block, and the second block shows the addition. In the third block, 0's are entered into Register T in case there is a carry. If so, the 0's are complemented to give the required 1's. Block three also specifies a branch (YCJ) on the carry out of the add, which took place during the preceding cycle. Thus, which of the next two microinstructions is used depends on the carry. If there is no carry, the upper block is used, and the sum, which is held in Register B, is moved to Register T. If there is a carry, the lower block is used, and the 0's in Register T are gated out in complement form to provide the required 1's, which are returned to Register T.

Since only the low-order bit is determined by the branch, the high-order portion of the addresses of the two blocks branched to are the same. The two blocks (NAMED0 and NAMED1) are referred to collectively as a "branch set" (NAMED–). The next-address field of the third microinstruction is NAMED, and the low-order bit is supplied by the branch condition.

A similar specification is made in the non-branching microinstructions. For example, the second microinstruction specifies Y0 (set low-order address bit to 0), and goes to the microinstruction at NAMEC0. Both of the branch-set microinstructions specify Y1 and go to NAMEE1.

Thus, the "compiler" assigns unique bit-values to the high-order portion of the address, and the microprogrammer handles the low-order portion when required for branching.

Although the SYSTEM/360 models that use a microprogram language have a structure similar to the one illustrated in this paper, the various models have different data paths and different sets of micro-orders and associated symbols for them. The differences between microprograms written for the various SYSTEM/360 models are somewhat analogous to the differences in assembly-language programs written for computers with different instruction sets.

higher-level languages

There has been much talk, but little success, in providing higher-level languages for microprograms. There seem to be a number of factors which contribute to this. Primarily, almost no inefficiency is tolerated in microprograms. The mere fact that something is worth microprogramming is an indication that high usage is expected. Furthermore, there is a problem of compiling efficiently into a language which has the flexibility inherent in the structure of SYSTEM/360 microinstructions. Basically, a compiler would generally be forced to compete with a microprogrammer who can justifiably spend hours trying to squeeze a cycle out of his code and who may make changes in the data path to do so. As long as microprograms continue to be written in this environment, a successful higher-level microprogram language seems unlikely.

Design automation

Although the design automation supporting a microprogram system is often overlooked, its importance to the success of the design of an ROS control system is great. Basically, microprogram design automation consists of three parts: a file-update system for maintaining microprograms, a simulation system for assuring their accuracy, and a program which generates the bit pattern used to manufacture the ROS device.

file update

Microprograms for a computing system are written in flowchart form similar to Figure 7, and each flowchart page is given a number. As a microprogram exceeds the size of a page, flow lines connecting the blocks are brought to the edge of the page, and their page-block destination is indicated on the edge of the page. This information is key-punched, and a master file of all pages is formed. The microprogram for each machine instruction has its last microinstruction go to the instruction-fetch microprogram, which branches out to the individual microprogram for each instruction. In this manner, all the individual microprograms are built up into a single large microprogram which controls all machine functions. The file-update system is able to provide for:

- Automated printing of all pages
- Changing any page
- Checking for reasonableness of information on a single page and connections between pages
- Retaining a current master file from which information can be extracted by other programs

simulation

A rather general simulation program is provided. In order for a particular microprogram designer to make use of the simulator he must have a "machine-description" of the particular data path and micro-orders to be simulated. The machine description names each facility and specifies its length in bits. (For the data path of Figure 3, the facilities are Registers A, B, and T, the left input to the adder, the right input to the adder, the adder output, and the shifter output.) The machine description also states, in terms of the facilities named, the action of each micro-order to be simulated and how it moves data between these facilities. Generally, the facilities also include several words of main storage.

Once a machine description is available, the simulator can be driven by a set of initial conditions (starting contents of some of the facilities) and microprograms that are extracted from the master file.

Thus, cycle-by-cycle simulation of microprograms becomes possible. To test his microprogram, a microprogrammer sets up initial data conditions in main storage and/or in the registers. The result is a cycle-by-cycle trace. The microprogrammer can also specify facility contents he wants printed with the trace. Any facility can be requested either for selected cycles or for all cycles, and the microprogrammer need not be inundated with useless output. Generally speaking, the microprogrammer looks at the last line to see if

he got the expected results. If he did not, he goes back through the cycle-by-cycle printed output to see where his microprogram went astray.

This form of simulation is at a high enough level to be convenient because whole machine instructions, and sometimes even small groups of instructions, are simulated. The technique is successful in debugging microprograms before hardware is built.

bit-pattern generation

Once the simulation is complete, the master file contains reasonably debugged microprograms. Another program translates them into the bit pattern used in the ROS device. This process is essentially a table lookup on all the micro-orders to determine their bit codings and fields. Also, in a manner analogous to an assembly program, actual addresses can be assigned to the symbolic addresses (e.g., NAMEB) and the next-address fields can be filled in using this information. Finally, this bit-pattern information can be further translated into instructions to control a piece of automated equipment which produces the physical ROS device.

The automation system, thus, provides means for maintaining a master file, checking the validity of the master file, and building physical devices which accurately represent the microprograms on the master file. A more detailed description of the automation system is available.[13]

After-the-fact microprograms

Thus far the discussion of microprograms has centered around the task of simultaneously developing a data path and an ROS control system to accomplish the predetermined task of implementing the SYSTEM/360 instruction set. The problems of writing microprograms for some other function, after-the-fact, on a system which has already been designed presents some significantly different problems.

Generally speaking, the smaller SYSTEM/360 models have more general data paths. The larger models have data paths which are more closely tailored to efficient execution of the SYSTEM/360 instruction set. Thus, adding new functions on the smaller machines tends to be easier. In either case, a new function that stays close to the SYSTEM/360 data formats is easier to implement than one which differs radically. A function using eight-bit bytes is simpler than one which uses six-bit bytes, and a ten-bit byte structure is much more difficult to handle. The SYSTEM/360 data paths were not built for ten-bit bytes. Similarly, the handling of a 36-bit data format presents significant problems within the 8–32–64-bit structure of SYSTEM/360. Nevertheless, the compatibility features and emulators on SYSTEM/360 attest to the flexibility provided by an ROS system.

compatibility

The SYSTEM/360 Model 30 is microprogrammed to run IBM 1401 programs.[14] To do this, each six-bit 1401 character is represented by its equivalent eight-bit EBCDIC (Extended Binary-Coded-Decimal Interchange Code) equivalent. Thus, the basic Model 30 eight-bit data path can move 1401 characters. The first bit position of the EBCDIC code (which is 1 for all 1401 characters) is used to hold the

1401 word marks. Although almost no change was made in the Model 30 data path, a branch micro-order was added to assist in recognizing word marks. The decimal 1401 addresses are converted to binary by a microprogram that uses table lookup of the binary equivalents of the decimal address digits. In order to provide space for the additional microinstructions, an extra module of ROS was provided. The success of the 1401 compatibility feature indicates both the flexibility of the ROS-controlled Model 30 and the ingenuity of the designers of the compatibility feature.

emulation

A similar approach to running IBM 7000 series programs on the SYSTEM/360 Model 65 ran into three areas of difficulty. Since the data paths of the 7000 series machines and the Model 65 are more tailored to high speed in executing their own instruction sets, gross inefficiencies were encountered in the microprograms required to overcome the differences. Also, it was physically impractical to add another module of ROS to the Model 65. Only ROS words that were not used by the base machine could be used. Since the Model 65 did not have microprogrammed channels, other means had to be found to handle I/O. In the emulation of large systems, these difficulties were overcome with a combination of techniques.[15]

Some hardware was added to the data path. For example, in the case of the IBM 7090 emulator, triggers were added to hold the sign of the 7090 accumulator and MQ register (S_{AC}, S_{MQ}). A special gate was added to shift a 7090 address field into the SYSTEM/360 address field. A special decoder was added to convert 7090 operation codes to either an ROS address or a main memory address, which held routines to simulate them with either a microprogram or a SYSTEM/360 program. In all, thirty-six new micro-orders were added and fourteen were modified. In addition, the local store, which holds the general-purpose registers and floating-point registers in the SYSTEM/360 mode, was made explicitly addressable with an emit field. Thus, inasmuch as the data path was modified, the 7090 emulator does not constitute an after-the-fact microprogram.

Functions that can not be done fast enough by program are done by microprograms using both existing micro-orders and those added specifically for the 7090 emulator. Many functions are accomplished by program. These included most of the I/O (except character translation and packing into 36-bit words), the interpretive console routine, and many of the easy-to-simulate or low-usage 7090 instructions.

Although the ROS controls made the 7090 emulator economically reasonable, the emulator was by no means an all-microprogram function. Hardware, microprogramming, and software are all used where they work well.

Concluding remarks

There is no doubt that a microprogram control system makes it easier to add special functions to a machine to tailor it to a particular application. There are already many cases where features have

been added which would have been impractical on a conventionally controlled machine. However, the feeling that anything can be microprogrammed at greatly improved performance is overly optimistic. This is particularly the case for large machines.

The concept of microprogram and ROS control discussed here is in no way dependent upon the control store's being an ROS. Although there are economic and practical considerations which may dictate that the control store be made read only, this is by no means logically required. It is quite feasible to make a control store writable under program control. The potential benefits and dangers of this are being debated and will probably continue to be for some time. Although it might be reasonable if kept under tight systems program control, the remarks of Wilkes at the 1958 Eastern Joint Computer Conference still seem to warrant attention. In reference to changeable control stores he said, ". . . the many problems involved in running a computing laboratory are bad enough as it is without the additional license which would be created by a system of private order codes."

Probably the primary reason for the use of ROS control in SYSTEM/360 is one of economics. A microprogram-controlled system has a base cost for the ROS and support hardware; after that, the marginal cost of an additional ROS word is relatively small. Thus, ROS becomes attractive for controlling a comprehensive instruction set, particularly on a small machine. Furthermore, the low marginal cost of additional function in an ROS-controlled system makes compatibility features and emulators feasible where they might not have been in a conventionally controlled system.

Additional benefits accrue from the more orderly approach to control design. Checking a large part of the control system becomes feasible, and printed microprogram pages are excellent control documentation and worthwhile service documents. Also, simulation has proved to be a significant design aid, and debugging time on engineering models is thereby reduced.

Microprogram control in SYSTEM/360 is not used to provide problem programmers with tailored instruction sets. It is, rather, hidden from direct view of the programmer. It remains in the province of the design engineer and is used to economically provide the programmer with a comprehensive instruction set for the series of compatible computers having a wide range of performance.

ACKNOWLEDGMENT

The author wishes to note that the work of many people, too numerous to mention here, has been reported.

CITED REFERENCES

1. M. V. Wilkes, "The best way to design an automatic calculating machine," *Manchester University Computer Inaugural Conference*, 16–18 (July 1951).
2. 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* **49**, Part 2, 230–238 (1953).

3. M. V. Wilkes, "Microprogramming," *Proceedings of the Eastern Joint Computer Conference*, 18–20 (December 1958).
4. J. V. Blankenbaker, "Logically microprogrammed computers." *IRE Transactions on Electronic Computers* **EC-7,** 103–109 (June 1958).
5. G. P. Dineen, I. L. Lebow, I. S. Reed, "The logical design of CG24," *Proceedings of the Eastern Joint Computer Conference*, 91–94 (December 1958).
6. T. W. Kampe, "The design of a general-purpose microprogram-controlled computer with elementary structure," *IRE Transactions on Electronic Computers* **EC-9,** 208–213 (June 1960).
7. A. Graselli, "The design of program-modifiable microprogrammed control units," *IEEE Transactions on Electronic Computers* **EC-11,** 336–339, (June 1962).
8. H. T. Glantz, "A note on microprogramming," *Journal of the Association for Computing Machinery* **3,** No. 2, 77–84 (April 1956).
9. H. M. Semarne and R. E. Porter, "A stored logic computer," *Datamation* **2,** No. 5, 33–36 (May 1961).
10. W. C. McGee, "The TRW-133 computer" *Datamation* **5,** No. 2, 27–29 (February 1964).
11. E. O. Boutwell Jr., "The PB 440" *Datamation* **5,** No. 2, 30–32 (February 1964).
12. L. Beck and F. Keeler, "The C-8401 data processor," *Datamation* **5,** No. 2, 33–35 (February 1964).
13. B. R. S. Buckingham, W. C. Carter, W. R. Crawford, G. A. Nowell, "The controls automation system," *Sixth Annual Symposium on Switching Circuit Theory and Logical Design*, 279 (October 1965).
14. M. A. McCormack, T. T. Schansman, and K. K. Womack," "1401 Compatibility Feature on the IBM SYSTEM/360 Model 30" *Communications of the Association for Computing Machinery* **8,** No. 12, 773–776 (December 1965).
15. S. G. Tucker, "Emulation of large systems" *Communications of the Association for Computing Machinery* **8,** No. 12, 753–761 (December 1965).