K. A. Duke
H. D. Schnurmann
T. I. Wilson

System Validation by Three-level Modeling Synthesis

Abstract: The experimental three-level system modeling technique discussed in this paper can be used during the design stage of a system for identifying mismatches among the architectural, microprogramming, and hardware logic levels. Compatible switching between modeling levels is emphasized. Execution of an application program by the architectural and microprogramming level models with switching between levels is illustrated.

Introduction

To cope with the increased complexity of computers during their design and development stages, the responsibility for the system elements is usually delegated to separate groups. Examples of such elements include microprogramming, hardware logic design, and system programming. System designers achieve compatibility among these elements by defining interfaces from one to the other. Thus the designer is not only interested in the individual functioning of the parts, but also in their functioning in concert. This requires system validation, which is usually accomplished by modeling the computer either in hardware or software [1–5].

In hardware models, it is fairly simple to demonstrate the functioning of a number of interconnected system elements. However, these models tend to lack flexibility. Programming models, although more flexible, are often incompatible between system elements. For example, during the development of System/360 Model 40, the data flow was agreed upon between the microprogrammers and the logic designers, and the System/360 instruction set was defined by the system architects. The logic designers embodied the data flow together with its controls into hardware, and the microprogramming group developed the required microprogram [6, 7]. Each group used its own simulator to debug its design, but the correlations between designs were performed manually. As a result of the manual correlation mismatches occurred, for example, while developing the IBM 1401 emulator on the Model 40; the microprogram simulator gave incorrect results due to the incomplete specification of decimal operations in the microprogramming and hardware logic designs. Similarly, the correlation between the instruction set usage and microprogram was done manually. The validation of those designs and the correction of the mismatch was not effectively completed until the test model of the computer was built and running.

This paper discusses an approach to preliminary compatibility validations of separately designed system elements. We implemented an experimental method for correlating system elements while a system is still in the design and specification stage. Our purpose was to explore the idea that mismatches can be identified before being cast in hardware. The three-level technique described in this paper integrates program models of a system design and the switching between models. To test this concept, we programmed the action of a simplified version of System/360 Model 40 at two design levels using the APL language [8, 9]. The detailed action of the system is discussed first from the architectural point of view (level I) and then from the viewpoint of microprogramming design (level II.)

Also discussed is the hardware logic design (level III), which was planned but not fully implemented. We ran application programs at both the architectural and microprogramming levels, and performed switching between

levels at arbitrary points in a program. An annotated example of such a two-level run, together with switching between levels, is given in the Appendix.

System synthesis

The system chosen by us as a test vehicle can be viewed in three distinct ways, which correspond to recognized computer design disciplines. These viewpoints are those of the system architect, the microprogrammer, and the logic designer. Other aspects are possible, but these three were chosen as being particularly well-defined and having considerable practical significance. The technique of system synthesis consists mainly of designing the three modeling programs (referred to as levels I, II, and III) and providing for switching between levels. Each level has its own initialization and data entry procedures. Input data at any level may be set up directly by the user, or may be an output from either the same level or a different level.

- *Level I: Architecture*

The system architect views the CPU as a device capable of executing System/360 instructions. He is concerned primarily with functional capabilities, but not with timing or the order in which operations are performed during the execution of a single instruction. The architect's view of the CPU, shown in Fig. 1, includes the following:

- byte-oriented main storage with up to 2^{24} bytes,
- sixteen 32-bit, fixed-point general registers,
- four 64-bit, floating-point registers,
- a 64-bit program status word (PSW) and
- a processor that recognizes and executes instructions.

System/360 allows four distinct types of processing:

1) logical operations on bits, character strings, and words,
2) decimal arithmetic on digit strings,
3) floating-point arithmetic and
4) fixed-point, binary arithmetic.

Instructions, such as tests and branches, are also provided for directing the flow of a program.

The system architecture model is initialized by setting zeros into all registers, the entire main storage, and the PSW. Also set up is a *navigation vector* [8], which is a table used to direct level I, wherein the instruction operation code is the table-lookup argument. The program and data are loaded into simulated main storage, and the user enters his application program in symbolic form. An assembly routine recodes the application program into 8-bit bytes, and determines where they are stored. The instruction address in the PSW is set to indicate the location of the first program instruction.

Instruction execution consists of two distinct parts—fetch-decode and execution. A fetch-decode program locates the first byte of the instruction (the operation code) by reference to the PSW. By inspection of that byte, the instruction format and the special data formats are determined. The operation code is then used as the table-lookup argument for the navigation vector. The modeling process then branches to the appropriate part of the execution program.

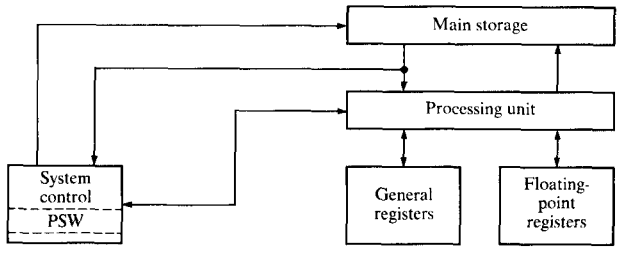


Figure 1 Architectural model of the CPU.

The level I execution program first calculates the length in bytes of each instruction, and then determines the value by which the instruction address in the PSW must be incremented. However, the PSW is not updated at this time because the instruction may later be aborted. The effective addresses of operands in main storage are determined, and the appropriate action is taken if they are not valid, e.g., an address value exceeds the highest address in main storage or an address does not lie on an appropriate word boundary. Operands are then located, the instruction function is executed, and the result is returned to the appropriate location. This result is tested to determine the setting of the condition code bits in the PSW. The next instruction address is updated, and the execution program branches back to the fetch-decode program, which fetches the next instruction. The loop is repeated until the instruction address points to an empty storage location.

The outputs of level I consist of the following system contents: 1) main storage (expecially the calculated results), 2) the PSW and 3) the floating-point and general registers. These outputs may be arranged in any desired format, and displayed or recorded at the end of a system modeling exercise or after the execution of any or all instructions. Alternatively, only those items that have changed may be displayed. Such displays constitute a useful trace facility.

Level I may be used by itself to validate a system architecture and to determine the functional accuracy of programs written to run on a system having that architecture. Alternatively, level I may be used to set up the input data for level II (microprogramming).

- *Level II: Microprogramming*

The microprogrammer deals with the first level of the physical implementation of the system and the flow of data

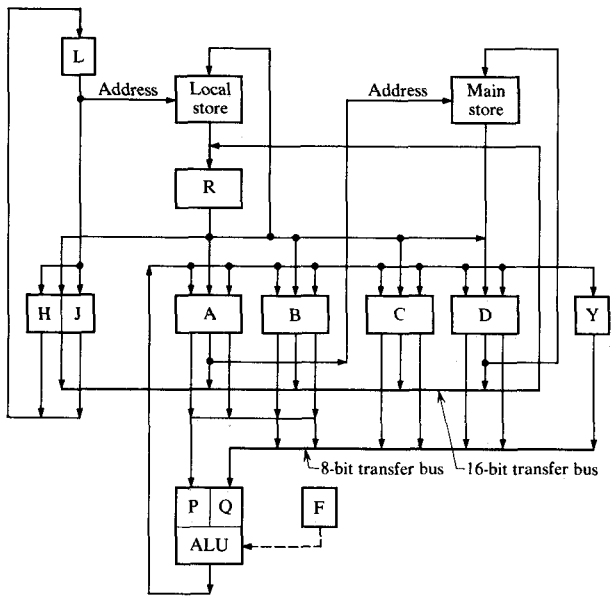


Figure 2 Microprogramming model of the CPU.

among the physical storage devices. A microprogramming model of the CPU data flow is illustrated in Fig. 2. Main storage contains 2^{15} two-byte words. The fixed-point and floating-point registers are contained in the local store, which consists of 128 words of two bytes each. The PSW is also held in the local store. Hardware registers A, B, C, D, H, J and Y are used as temporary storage for data during the execution of an instruction. In addition, register A functions as the address register for main storage and register D is its data register. Registers H and J act as address sources for the local store.

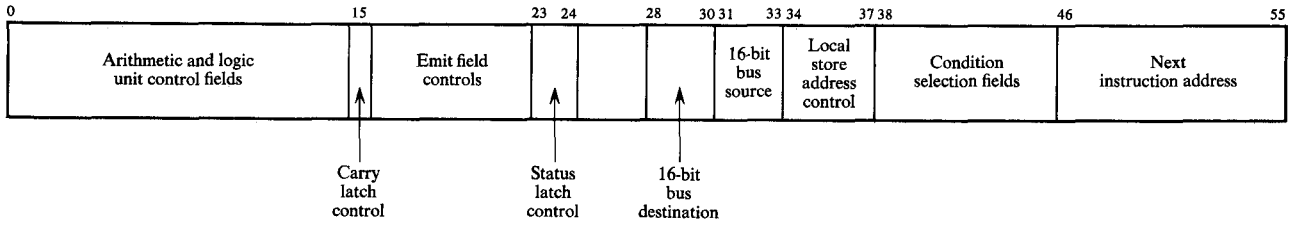
Arithmetic and logical operations take place in the arithmetic and logical unit (ALU), and the hardware registers are its input sources and output destinations. The function executed by the ALU is determined by the content of the 4-bit register F. The set of ALU functions includes binary and decimal addition over 8-bit fields, shifts, and various logical operations. Although the overall Model 40 was not implemented in level III, the ALU was modeled in a manner close to that of hardware logic. The reason is that the ALU is so complex that the only way to represent its function unambiguously is at (or close to) the hardware logic rather than at a higher level.

Data are moved around the CPU under the control of a microprogram stored in the read-only store (ROS). The microprogram is written in a symbolic notation that is translated into binary values for each microinstruction field when the microprogram is loaded into the ROS. Formatting of a microinstruction is indicated in Fig. 3. Each word of the ROS contains one microinstruction, which determines the function to be performed during one control cycle, that is, the manner in which data are to be moved among the hardware elements of the system. Each microinstruction also determines the ROS location of the next microinstruction to be executed. Some bits of the ROS address may be modified by selected items of data in the hardware registers, thus achieving conditional branching in the microprogram.

A microinstruction is divided into fields, each of which controls a particular segment of the data flow or a particular set of functions. The number of different combinations of functions that a specified field can control depends on the number of bits in that field. The combinations are chosen to be as simple as possible, with the stipulation that no two combinations are required at the same time. For example, a single field of the microinstruction controls the gating of data onto the 16-bit transfer bus, allowing only one hardware register to be gated at a time.

Initialization of the microprogramming model begins by setting the dimensions of the arrays representing main storage, the local store, hardware registers and ROS, and by loading all but ROS with zeros. The control program is then entered in the appropriate ROS locations, and the ROS address register is loaded with the address of the first microinstruction of the instruction fetch routine. (ROS controls the data flow shown in Fig. 2; therefore, the ROS itself is not shown.) Input data to the microprogramming model are a representation of the state of the CPU at the beginning of the instruction to be executed. The System/360 program containing that instruction is loaded into main storage. Data being operated on by the level II model are transferred into main storage in the state at which it would exist at that point in the program, as

Figure 3 Microinstruction format.



shown, for example, by data t_1 in Fig. 4. Partial results are also loaded at this time. Those locations in the local store that represent the PSW, the floating-point, and general registers are set with the appropriate values. Finally, the two latches of register Y that represent the condition code are set to the correct values. Execution of the microprogramming model begins by locating the first microinstruction defined by the contents of the ROS address register shown in Fig. 5.

Each field of the microinstruction is decoded and used to control the appropriate section of the data flow. Since the control cycle consists of two phases, care must be taken that all operations performed in phase I of the control cycle are executed before any of the phase II operations. The modeled functions include the 8-bit and 16-bit data transfers, local and main storage addressing and accessing, and the calculation of the next microinstruction address. Because this calculation may depend on a number of machine conditions, which may be set at different times in the control cycle, it is necessary that the appropriate values of those conditions be sensed. Conflicts can occur when two data transfers take place simultaneously into the same hardware register. These conflicts must be resolved by performing dominant transfers after those that they override. Main storage read or write operations require more than one control cycle. Therefore, data transfers initiated in previous control cycles must be represented in addition to those transfers initiated by the current microinstricution. Moreover, checks are made to determine that conflicts in the operation of main storage do not occur. Certain operations, such as the arithmetic and logical functions, are fairly complex and difficult to express precisely in any concise way. It was therefore decided to implement these functions in terms of their hardware logical implementation. That is, part of the level II microprogramming is expressed in level III hardware logic terms.

At the end of the execution of each microinstruction, level II is ready to begin executing the next microinstruction, as in going from m to $m + 1$ in level II of Fig. 4. The output data from one cycle are exactly the input data to the next. After the last microinstruction of a machine instruction ($m + q$ in Fig. 4), a subset of the output data represents the machine instruction output data. This subset is the contents of the local store that represent the general and floating-point registers, the PSW and the data in the main storage.

Level II may be used by itself to validate the design of a microprogram. However, it is easier to derive level II input data from level I, and then to switch to the microprogramming level for that instruction whose microprogram is to be tested. This is illustrated between t_1 and t_2 in Fig. 4. In order to determine the accuracy of a given system implementation, the results of executing an entire program sequence at the architectural level may be compared

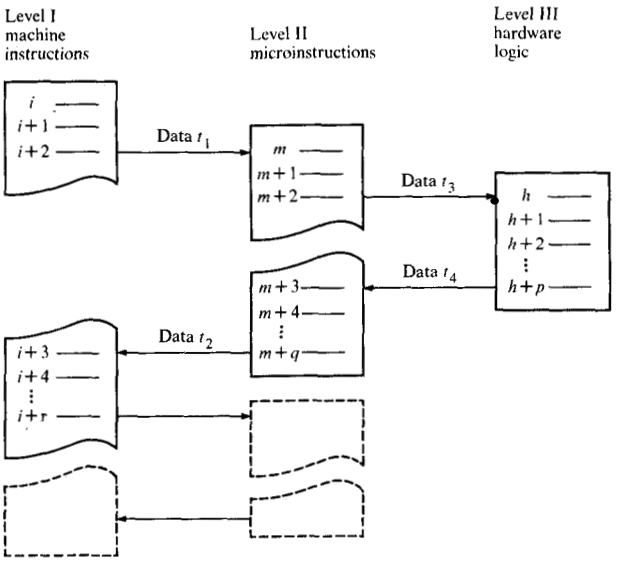
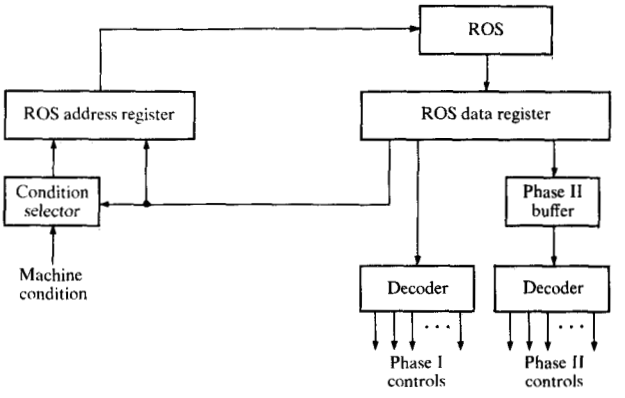


Figure 4 Switching between levels.

Figure 5 Machine cycle control system.



bit-for-bit with the results of executing the same sequence with one instruction modeled at level II. (Such a comparison is made between level-I and level-II results in the Appendix.) The microprogramming model may also be used to provide input data to the level III hardware logic model, as illustrated for data t_3 in Fig. 4.

• *Level III: Hardware logic*

The hardware logic designer's view of the system differs fundamentally in many respects from both of the previously described views. The hardware designer deals with continuously operating physical devices. Thus, he is concerned not only with the elements of the data flow as seen by the microprogrammer, but also with the control

of those elements. He must implement the set of functions required by the microprogrammer, and he must ensure that no functions occur other than those that are called for in any given machine cycle.

Each machine cycle is divided into two phases, and data are latched at the end of each phase. During phase I, data may be transferred from the data registers (A, B, C, D, H and J) via the appropriate buses to the buffer registers P, Q, R and L. During phase II, the results of logical operations may be returned to the data registers. Data registers and buffer registers are shown in Fig. 2.

The machine cycle control system is described with reference to Fig. 5. The output from the ROS, which is used to control the CPU during a particular cycle, is available in the ROS data register at the beginning of phase I of that cycle. Those fields that are used to control data transfers taking place in phase I are directly decoded. However, since the contents of the ROS data register are changed during phase II of each cycle, those fields used to control data transfers during phase II must be buffered. Transfer to the buffer takes place during phase I, and the fields are decoded for use in phase II.

Also during phase I, the ROS address register is changed by transferring the next instruction address field (shown in Fig. 3) from the ROS data register to the address register. The least significant two bits of the ROS address are used to effect microprogram branches. This is accomplished by using each condition selector to select one of 16 one-bit data sources, shown in Fig. 5 as machine conditions.

Hardware logic modeling differs in detail from architectural and microprogramming modeling in that hardware logic consists of a set of interconnected functional units, each of which responds to its input signals at all times. Therefore, during hardware modeling, it is necessary to ensure that each unit performs its particular function only when required, and to constantly check the control signals to all units. The function of each unit is described in terms of its inputs and appropriate internal storage. It is particularly difficult to model the inputs to a unit that can be modified by its own output, however indirectly, in a time interval undivided by clock pulses. (In the System/360 Model 40, this situation occurs only within a latch circuit.) The hardware logic model, therefore, treats latches as single storage cells.

In addition to the units described in the microprogramming model (level II), level III contains control signal generators, decoders, timing generators and buffers. Each of these is assumed to be continously operative. Provided no malfunctions occur, the examination of clock pulse inputs to the latches is frequently sufficient to enable large parts of the logic to be ignored at any one time. To permit data checking, every register and buffer has one or more parity bits associated with it. Also the arithmetic and logical unit is described in two-rail logic, wherein every function is defined by its true and its complement values. Checking logic is introduced at various points in the data flow, and must be sampled at appropriate times.

The hardware logic model is initialized by specifying the dimensions of the arrays that represent main storage, the local store, and all registers and buffers. The timing generator is set to phase I of the machine cycle. The contents of the ROS, local store, and main storage are defined, including parity bits. The contents of the registers are set appropriately including the ROS address register. Flags are set to indicate any incomplete storage operations to ensure their correct modeling. The appropriate word of the ROS is then loaded into the ROS output register, and the model of phase I may proceed.

The output of each logic unit between the registers and the buffers must be determined, and it is necessary to order those units to ensure that all the inputs to any unit have been defined for a given cycle. This is done by ordering the sequence of each unit in such a way that its level is at least one greater than the unit feeding it. A unit may not be assigned a level until those of all its inputs have been assigned. The model simulates units in order of their increasing level. At the end of phase I, the buffers are set to whatever values their inputs indicate. Similarly, at the end of phase II, the registers are set according to their inputs. Simulation terminates only at the end of phase II. At that time, the output of the model consists of the contents of the registers recognized by the microprogram, and the contents of both the local store and main storage.

The hardware logic model would be useful for exercising the machine logic under various conditions by using level III alone. Similarly, the effect of component malfunctions may be observed by suitably modifying the hardware logic. However, as previously suggested, a single machine cycle that uses the hardware logic model may be embedded into a microprogram model. This technique (illustrated in Fig. 4) may be used to validate the logical operation of the CPU, or to examine the effects of an intermittent malfunction on the microprogram, and subsequently on the system operation.

We have mentioned that one of the objectives of the three-level system model is to validate a computer design, i.e., to determine whether a machine has been designed so as to perform its intended function. This is accomplished by comparing the performance of a machine design at three levels under the same conditions. The resultant states of the models at each level should be identical. The simplest way to obtain the same initial conditions for the different levels of modeling is to generate them from the same source. This is done by substituting a microprogram model for one or more instructions in an architectural model, or by substituting a hardware logic model for one or more microinstructions. Such substitutions involve switching among the three levels. An example of levels I and II and

the switching between levels is presented in the Appendix. When it is required to switch modeling levels, the input data to the next level are completely and unambiguously specified by the output data from the current level. It is necessary to change only the formats and to set up certain predefined initial conditions. For example, when switching from level I to level II, the contents of the machine registers, PSW, and main storage are transformed to their appropriate formats, and the ROS address register is set to the beginning of the instruction fetch routine. The program execution may then continue at level II. At the end of the more detailed program, the reverse process takes place.

System implementation

The three-level model was implemented in APL\360 because of the power of the language [9] and because it allows interactive debugging of the modeling program. (Since APL programs are executed in an interpretive mode, it is also possible to alter a program and continue execution without recompiling.) The elements of a machine being modeled are described as arrays, which are convenient to reformat when switching levels. For example, the main storage—described at the architectural level as an 8192 × 8-bit matrix—is transformed to a 4096 × 16-bit matrix at the microprogram level using one APL statement.

The modeling programs are carefully segmented so that a change in any part can be made without affecting other parts by spurious interactions. For example, the execution of an instruction or microinstruction is separated from the selection of the next instruction. For simplicity, application programs and microprograms are entered in symbolic form. An elementary assembler converts these programs into appropriate bit patterns, and determines the format of each instruction and the value to be inserted in each field. The assembler also determines the length of each instruction, which it uses to compute the location of the next instruction. Similarly, the microprogram assembler analyzes the symbolic statement of each microinstruction and computes the value to insert into each field of the appropriate ROS word.

Trace routines enable one to follow the course of each exercise in as much detail as he requires. Printouts of the data stored in the system elements modeled may be obtained at any prescribed point, and may include data from all system elements or only values that have changed since the last printout.

Originally, certain restrictions were imposed by workspace limitations on APL\360. This implied that only one level of modeling could be contained in each workspace. As a result, the switching of levels required manual intervention and the use of APL system commands. Such interventions, which are illustrated in the Appendix, are limitations of the specific APL model and not of the modeling system or the APL language itself.

Concluding remarks

We developed the experimental three-level system validation model to ensure that the machine design will perform its intended function as defined by its architecture. The architectural model (level I) has been used by itself to evaluate system implementation, and in combination with the microprogram model (level II). Congruence of the model outputs for the same function at both levels I and II has validated the design of the microprogram data flow for such functions on System/360 Model 40. A similar comparison between levels II and III may be used to verify the logic design.

The multilevel model has also been used to generate input data for a detailed model to test a section of microprogram under a variety of application programming conditions. Although setting up such tests can be very tedious, the tedium may be conveniently avoided by using the architecture model and by switching to the microprogramming level at appropriate points.

Another motivation for the three-level modeling system was to study the effects of machine failures (especially intermittent failures) on the execution of a program. This can be done by simulating the program at the architectural level, then by switching to the microprogram and logic levels at points where failures occur. Both the timing and location of failures may be selected at random or in a predefined way, particularly if an item is to be tested for vulnerability.

Our system makes it possible to design computers in sequence, with well defined interfaces, from the architecture to the hardware logic levels. This and similar techniques are being increasingly used in computer design, and can possibly be extended to higher levels such as operating systems where the user has no access to the machine language in which the operating system is implemented.

Appendix: An example

We now describe System/360 Model 40 CPU modeling in terms of an illustrative example called DEMO. Written in the APL language, DEMO resides in level I, as shown in Fig. 6. It includes a number of statements that load a few program steps written in System/360 instructions (indicated by the shaded area of Fig. 6) into main storage. In this example, the contents of storage location 52 are doubled. If the result is negative, it is stored in location 60; if positive, it is stored in the location determined by 60, plus twice the contents of storage location 56. Model 40 validation at the architectural level is described first.

Two integers are inputs to the variable IN shown in Fig. 6. In statement [1], VALUES calls an APL function to initialize the contents of the general registers, main storage, and the PSW to all zeros, and to set values appropriate to level I. Statement [2] of DEMO causes the input data to be loaded into main storage, and statement [3] sets the binary

```
          ∇DEMO[□]∇
        ∇ DEMO IN
[1]     VALUES
[2]     52 4 MLOAD IN
[3]     PSW[40+⍳24]←(24⍴2)⊤20
[4]     L PR 20 8 0 0 52
[5]     AR PR 8 8
[6]     BC PR 4 0 0 36
[7]     SR PR 10 10
[8]     A PR 10 0 0 56
[9]     ST PR 8 10 10 60
[10]    SUP
[11]    0 80 4 XDISPLAYM XM[⍳96;]
        ∇
```

Figure 6 Illustrative example.

Figure 7 Execution at the architectural level.

```
      )LOAD LEVEL1
SAVED  13.40.14 06/05/70

      SUPERTRACE←1
      STOPΔPR←⍳0
      DEMO 357 4
VALUES...SYSTEM INITIALIZATION.
MLOAD..DATA BEING LOADED.
L          8      0      0     52......STORED...LOC: 20 - 23
AR         8      8.....................STORED...LOC: 24 - 25
BC         4      0      0     36......STORED...LOC: 26 - 29
SR        10     10.....................STORED...LOC: 30 - 31
A         10      0      0     56......STORED...LOC: 32 - 35
ST         8     10     10     60......STORED...LOC: 36 - 39
**************************
        20...L          8      0      0     52
357(R8) ← MEM.LOC.(52)
        24...AR         8      8
714 = 357 + 357
CC←2
        26...BC         4      0      0     36
BRANCH NOT TAKEN.
        30...SR        10     10
0 = 0 - 0
CC←0
        32...A         10      0      0     56
4 = 0 + 4
CC←2
        36...ST         8     10     10     60
714(R8) ← MEM.LOC.(68)
PROGRAM HALT: 30.07 SECONDS CONNECT...11.8 SECONDS CPU.

HEXADECIMAL DISPLAY OF CONTENTS OF MAIN STORAGE

     0 = 00000000              40 = 00000000
     4 = 00000000              44 = 00000000
     8 = 00000000              48 = 00000000
    12 = 00000000              52 = 00000165
    16 = 00000000              56 = 00000004
    20 = 58800034              60 = 00000000
    24 = 1A884740              64 = 00000000
    28 = 00241BAA              68 = 000002CA
    32 = 5AA00038              72 = 00000000
    36 = 508AA03C              76 = 00000000
```

Figure 8 Start execution at the architectural level.

```
      )LOAD LEVEL1
SAVED   8.45.13 06/08/70

      STOPΔPR←24
      DEMO 357 4
        20...L         8      0      0     52
357(R8) ← MEM.LOC.(52)
STOP: IA=24

SUP[7]
      )SAVE
   8.48.30 06/08/70 LEVEL1

      )LOAD LEVEL2
SAVED   8.46.27 06/08/70

      DATATRANSFERΔLEV1ΔTOΔLEV2
INITVAL...NOW BEING EXECUTED.
TYPE: )COPY LEVEL1 LEVEL1ΔXMΔXRΔPSW
□:    )COPY LEVEL1 LEVEL1ΔXMΔXRΔPSW
DATA TRANSFER NOW COMPLETE.
TYPE:  SUPEXC ...TO CONTINUE SIMULATION.
```

representation of the decimal value 20 into the last 24 bits of the PSW. Since this is the instruction address field, the model begins executing the instruction in storage location 20.

Execution of the example begins by a call to the SUP function in statement [10] of DEMO. Figure 7 shows the execution of DEMO at level I. The level I workspace is brought into the active workspace, and the APL system responds with the time and date this workspace was last saved. SUPERTRACE is a switch that is set to 1 to permit tracing the loading and execution of the example. The STOPΔPR switch, which can stop execution at any desired instruction, is turned off. Input data to DEMO are 357 (number to be doubled) and 4 (store location displacement).

The execution phase is signaled by the line of asterisks. During execution, the trace facility prints each instruction executed and the result. Execution in Fig. 7 shows 357 loaded into register 8; doubling to yield 714; the branch not taken; register 10 set to zero (to which the input 4 is added); and the result (714 in register 8) stored in location 68. At this point, the program halts. The terminal was connected for 30.07 seconds, of which the CPU usage was 11.8 seconds. (CPU utilization varies with system configuration; for a given configuration, CPU time is essentially constant.) After the execution stops, the contents of storage are printed, the input data are in locations 52–55 and 56–59, and the result is in locations 68–71. (Note that $357_{10} = 165_{16}$, and $714_{10} = 2CA_{16}$.)

In Figs. 8 through 11, DEMO illustrates the switching from modeling level I to II and back to level I. In Fig. 8, STOPΔPR is set to 24. This halts the simulation prior to execution of the instruction in location 24. The same input data are used here as in the previously described level I example.

The load instruction L is executed, then STOP IA = 24 signals the simulation to halt. At this point, the command is given to SAVE the active workspace. This freezes the state of the machine, and stores the state data for later use. To see the execution of L (load) in greater detail, we switch to level II in Fig. 9, and execute this instruction at the microprogramming level.

The switching between levels I and II is illustrated at the bottom of Fig. 8, and proceeds as follows. Level II is loaded, and the DATATRANSFER steps are executed by first initializing level II and printing the request that the contents of storage, registers, and the PSW (all of the level I) be copied into the active workspace. When the experimenter enters the)COPY command, the contents of storage XM are transferred to the level II workspace using the level I format. That is, the storage contents are one byte wide, and are expressed as a matrix of eight columns and a number of rows (N) equal to the number of bytes in storage. Moreover, XM in the level II workspace must exist in the same

form as it does in the Model 40; that is, storage must be two bytes wide. Thus the XM matrix is transformed by the DATATRANSFER program from an $N \times 8$ structure to an $N/2 \times 16$ structure. Since the 16 general registers XR and the PSW do not exist as separate entities in the Model 40 (but appear as particular words in the local storage), the same procedure must be followed in level II. Registers XR are thus placed into the first 32 rows of the LOST (local storage) matrix, and the PSW into rows 32–35 by the commands:

```
XM←((((⍴XM)[0])×2),8)⍴,XM
XR← 16 32 ⍴,LOST[⍳32;]
PSW←,LOST[32 33 35 34 ;]
```

Data transfer is now complete, and level II (Fig. 9) is ready to continue executing DEMO. The command is given to stop the program execution at storage location 26. Instruction L in storage locations 20–23 has already been executed, and level II is now ready to execute the AR instruction in locations 24–25. The STOPΔPR command permits the execution of just one instruction.

The level II model emulates the execution of each System/360 instruction by executing many microinstructions. Hence, the execution of the AR instruction may be thought of as a test to determine the validity of the microprogram that is contained in the read-only or control storage (ROS). The microprogram for the AR instruction consists of several hundred microinstructions. For purposes of this demonstration, a small segment of the microprogram in mnemonic form necessary to execute the AR instruction is shown in Fig. 10. To illustrate the execution of the AR instruction in locations 44 to 49, the trace is turned on, and each microinstruction shown being executed in Fig. 9 is identified by its address in ROS in Fig. 10. Each microinstruction in Figure 9 is preceeded by a printout of the contents—in hexadecimal—of each of the Model 40 hardware registers (shown in Fig. 2) prior to its execution.

The first column of Fig. 10 gives the ROS location of the microinstruction. The column headed X ← P/Q* shows controls for the ALU and the 8-bit data path. In the fourth column are the controls for the LOST and the 16-bit data paths. Field E contains immediate data for use in the microinstruction. One of the MISC functions is to initiate main-storage read or write cycles. Branching and next-instruction information is contained in the last three columns.

Each of the major registers (A, B, C, D and H) consists of two bytes. The 16-bit control field refers to each register as a whole. Thus the microinstruction in location 44 results in a transfer of the contents of the entire LOST word into register B. The 8-bit control field, however, differentiates between most-significant and least-significant bytes of the registers. In location 47, for example, the 8-bit destination register is B1 (least-significant byte of register B), and the two source registers are A1 and B1.

```
      STOPΔPR←26
      STOPΔMU←⍳0
      TRACEΔMU←(⍳16),44+⍳7
      MICROTRACE←NANOTRACE←0
      SUPEXC
**************************
EXECUTING: INSTRUCTION IN XM[24]
        24...AR        8     8
 E=3 HJ=3342 A=0000 B=2208 C=1A10 D=0165 Y=1C F=F
44....
 E=0 HJ=3343 A=0000 B=0165 C=1A10 D=0165 Y=1C F=F
 LOST[67]=0000
45....
 E=0 HJ=3341 A=0000 B=0165 C=1A10 D=0165 Y=1C F=F
 LOST[67]=0000
46....
 E=0 HJ=3340 A=0165 B=0165 C=1A10 D=0165 Y=1C F=F
 LOST[65]=0000
47....
 E=0 HJ=3340 A=0165 B=01CA C=1A10 D=0165 Y=1C F=F
48....
 E=0 HJ=3341 A=0165 B=02CA C=1A10 D=0165 Y=1C F=F
 LOST[64]=0000
49....
 E=0 HJ=3340 A=0165 B=02CA C=1A10 D=0165 Y=1C F=F
 LOST[65]=02CA
50....
```

```
EXECUTING: INSTRUCTION IN XM[26]
        26...BC        4     0     0     36
STOP: IA=26

SUPEXC[11]

      )SAVE
   8.55.29 06/08/70 LEVEL2

      DISPLAYREG
GENERAL FIXED-POINT REGISTERS:
    8  714
```

Figure 9 Continue execution at the microprogramming level.

Figure 10 Sample of mnemonic representation of the system microprogram in the ROS.

LOC	X←P/Q *	C←D(X)	YX*	E	MISC	RC	BCN	CCN	NIA
44		B←(JA)		3					45
45		(J2)							46
46		A←(J-)							47
47	B1←A1⍳B1								48
48	B0←A0⍳B0	(J+)							49
49		(J-)←B							50

Upon completion of the execution of the AR instruction, the program STOP facility halts execution prior to beginning the instruction in storage 26 as shown in Figure 9. The level II workspace is saved via the)SAVE APL system command. A decimal display of the contents of the nonzero registers reveals that register 8 contains the proper result of 714 at this time.

We now switch back to level I to complete the DEMO program. The process of switching and transferring the data from level II to level I is just the reverse of the previous

```
        26...BC         4      0      0     36
BRANCH NOT TAKEN.
        30...SR        10     10
0 = 0 - 0
CC←0
        32...A         10      0      0     56
4 = 0 + 4
CC←2
        36...ST         8     10     10     60
714(R8) → MEM.LOC.(68)
PROGRAM HALT: 526.47 SECONDS CONNECT...190.98 SECONDS CPU.

HEXADECIMAL DISPLAY OF CONTENTS OF MAIN STORAGE

     0 = 00000000          40 = 00000000
     4 = 00000000          44 = 00000000
     8 = 00000000          48 = 00000000
    12 = 00000000          52 = 00000165
    16 = 00000000          56 = 00000004
    20 = 58800034          60 = 00000000
    24 = 1A884740          64 = 00000000
    28 = 00241BAA          68 = 000002CA
    32 = 5AA00038          72 = 00000000
    36 = 508AA03C          76 = 00000000
```

Figure 11 Completion of execution at the architectural level.

switching operation. The data transformation commands are:

```
XM←((((ρXM)[0])÷2),16)ρ,XM
LOST[ι32;]← 32 16 ρ,XR
LOST[32 33 35 34 ;]← 4 16 ρ,PSW
```

We now resume executing level I shown in Fig. 11, the last four instructions of which are executed and printed out. The result 714 computed by level II is stored at location 68.

At this point, the program halts. There were 190.98 seconds of CPU time used, as compared with 11.8 seconds when the entire simulation took place at level I with no switching between modeling levels. The resulting printout of the contents of main storage is identical to that of Fig. 7. This demonstrates the validity of the microprogram as well as the complete compatibility of the two simulation levels.

References

1. E. W. Dijkstra, "Structure of the THE—multiprogramming system," *Proc. ACM Symp. Operating System Principles,* Gatlinburg, Tennessee (October 1–4, 1967).
2. D. Fox and J. L. Kessler, "Experiments in software modeling," *AFIPS Conf. Proc., Fall Joint Computer Conference* **31,** 429–436 (November 1967).
3. N. R. Nielsen, "Computer simulation of computer system performance," *Proc. 22nd ACM Nat. Conf.* 581–590 (1967).
4. D. L. Parnas and J. A. Darringer, "SODAS and methodology for computer system design," *AFIPS Conf. Proc., Fall Joint Computer Conference* **31,** 449–474 (November 1967).
5. F. W. Zurcher and B. Randell, "Iterative multi-level modeling—a methodology for computer system design," *IFIP Congress 1968,* Edinburgh, Scotland, D138-D142 (August 5–10, 1968).
6. S. G. Tucker, "Microprogram control for System/360," *IBM Sys. J.* **6,** No. 4, 222–241 (1967).
7. M. A. McCormack, T. T. Schansman, and K. K. Womack, "1401 compatibility feature on the IBM System/360 Model 30," *Comm. ACM* **8,** No. 12, 773–776 (December 1965).
8. A. D. Falkoff, K. E. Iverson, and E. H. Sussenguth, "A formal description of System/360," *IBM Sys. J.* **3,** Nos. 2 and 3, 198–261 (1964).
9. A. D. Falkoff and K. E. Iverson, *APL 360 User's Manual* (1968) may be obtained from the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

Received March 25, 1970

K. A. Duke is with the Systems Development Division at the laboratory in Poughkeepsie, New York; H. D. Schnurmann is at the Components Division laboratory in East Fishkill, New York; and T. I. Wilson is with the Data Processing Division in White Plains, New York.