USING A COMPUTER TO SIMULATE A COMPUTER

By GEORGE R. TRIMBLE

ABOUT THIS ARTICLE

Innovations in computer design, which motivate us to replace existing computer systems with new and better systems, also compel us to develop techniques for simulating the new computer before it arrives. Called simulation and emulation, these techniques enables us to imitate all the functions of the new computer using the old computer. Not only does this facilitate conversion from one system to another, but it also enables us to test programs for a new computer even before the machine is built. Simulation mainly concerns the software which enables the use of existing machine applications on the new system prior to reprogramming. Emulation involves hardware features specifically designed within the new computer to promote simulation compatibility. Many problems arise because of basic differences in machine operation and simulation objectives.

One of the more exotic applications of digital computers is to simulate a digital computer on another entirely different type of computer. Using a simulation program, application programs developed for the first computer, the source computer, may be executed on a second computer, the object computer.

Simulation obviously provides many advantages in situations where a computer is replaced by a different computer, for which the applications have not yet been programmed. Simulation techniques enable an installation to continue solving problems using existing programs after the new computer has been installed and the old one removed.

Simulation of the replaced computer is obviously a much less efficient means of solving the problem than reprogramming the applications for the new computer. When such a switchover is made, however, it is not practical to reprogram all the applications for the new machine immediately. Use of a simulator permits the installation to continue running its programs as reprogramming proceeds on a reasonable schedule. In fact, for some of the very infrequently used programs, reprogramming may not be worthwhile.

Another situation in which simulation is advantageous is during the development of a new computer. Once specifications for the new computer have been established, programming of applications for the computer can proceed in parallel with hardware development. The use of a simulator in this situation enables the users to debug their applications before the hardware is actually available.

Simulation of an unbuilt computer is advantageous for other reasons as well. Sometimes new equipment has bugs in it which may not be discovered during assembly and final test procedures. Debugging a computer program on a new piece of equipment frequently results in errors which can not be firmly associated with either the equipment or the program. If, however, the program has been debugged on a simulator, greater assurance can be placed in the program and it is more likely that problems encountered are equipment problems.

Another advantage of simulation is that sophisticated debugging aids can be built into the simulator which would be very difficult, if not impossible, to build into a debugging system for the computer being simulated. Selective dumping and tracing are examples of the type of debugging facilities which can be incorporated into a simulator.

Simulation Objectives

A simulator has two principal objectives. First and

foremost, the simulator must faithfully duplicate the functions of the computer being simulated. The results of arithmetical and logical operations must be precisely the same as those which would be obtained if the program were run on the source computer itself. Simulation of the timing of asychronous functions may not be as critical, but in some situations even the timing must be simulated as precisely as possible.

By faithful simulation it is not to be implied that functions on the circuit level must be duplicated, but rather functions which are pertinent to the programmer must be simulated. For example, the contents of the arithmetic registers and the status of various indicators must be duplicated. The functions of an adder or a shift register do not have to be duplicated precisely, but the effects of these devices on the contents of the registers which are accessible to the programmer must be duplicated.

The second principal objective of a simulator is speed of simulation. Since the process of simulation is inherently inefficient, it may occur that the program may operate slower on the object computer than it does on the source computer even though the object computer is considerably faster than the source computer. Advantage must be taken of every feature of the object computer to increase speed of simulation. This implies that great care must be taken in the coding of critical portions of the simulator, such as the instruction fetch and basic instruction execution routines. It frequently occurs that a function which is common to many subroutines should not be made a subroutine itself but instead should be programmed in its entirety each time that function is required. For example, a subroutine to simulate a subtract instruction differs from the subroutine to simulate an add instruction only in that the sign of the operand is reversed. It would be much more efficient from a storage view point to reverse the sign of the operand and then transfer control to some point within the subroutine which simulates an add instruction. This approach, however, would require at least one additional branch instruction to be executed when a subtract instruction is simulated. It is, therefore, faster to write out the subtract subroutine in its entirety, separate and distinct from the add subroutine, even though 90 per cent of the instructions may be identical.

This philosophy could be carried out to an extreme, however, and judgment of when to duplicate coding must be based on the frequency with which specific functions are performed. If the object computer is limited in its storage available, it may be necessary to sacrifice speed in order to make the program even fit within the memory space available.

Approach to Simulation

The characteristics of simulators described above point out the necessity for having two types of simulators. Changing an installation from one computer to another means that the programs to be executed have already been debugged and are in a production state.

ABOUT THE AUTHOR

George R. Trimble, Jr., was graduated from St. John's College in 1948 and obtained a B.A. and an M.A. in mathematics from the University of Delaware in 1951. Formerly with the Computing Laboratory of the Ballistics Research Laboratories and a senior staff member in the Applied Science Div. of IBM, Mr. Trimble joined Computer Usage Co., Inc., in 1955. Utilizing experience on a multitude of computer systems, he now performs analyses and supervises major programs for many applications using a wide variety of machines, including IBM 360, 650, 702, 704, 1401, 7030, 7070, 7090, 7740, Honeywell 800, Bendix G-15, ISI-609, Univac 1107 and ASI-420.

Speed is the most important factor in this situation and debugging facilities are minimal or non-existent. This type of simulator is called a production simulator.

The other type of simulator is used to debug programs for an unbuilt computer and is called a debugging simulator. Speed is also important here but many additional features are built into the simulator to facilitate debugging of programs. In fact, the debugging facilities made available in the simulator can be so significantly superior to those available in the object computer that it is preferable to debug programs using the simulator even if the object computer has been built and is available.

The classical organization of digital computers is into control, arithmetic, memory, and I/O sections. The development of a simulator very much parallels this organization

The memory of a computer may consist of words or characters. A portion of the object computers memory is set aside so that the words or characters of the source computers memory can be simulated. If both computers have word memories, for example, one word of the object computer may be used to represent one word of the source computer.

Obviously this depends upon the word length of the two computers. In some cases it may be possible to represent two source computer words in one object computer word, while in other cases it may be necessary to use two object computer words to represent one source computer word. An analogous comparison can be made for simulating a character memory in a word memory in which several characters of the source computer may be packed in one word of the object computer's memory for efficient utilization of memory.

The control section of the computer is simulated by having a register which performs the same functions as the program location counter in the source computer. An instruction fetch routine is used to refer to the contents of the simulated program location counter in order to determine the location of the next instruction to be simulated. The instruction fetch routine then refers to simulated memory to obtain the instruction and the contents of the simulated memory location are analyzed to determine what must be done. The instruc-

tion code is examined and control is transferred to a subroutine whose function is to simulate the specific instruction being simulated. Other functions performed in this process may include, for example, determination of which index register is to be used as a modifier, computation of the effective address based on the index register modification, and obtaining the data from the simulated location or storing the results of the operation in the simulated location determined as a result of this address computation.

The arithmetic section is simulated by the many subroutines which are required to simulate the individual instructions. The registers which must be simulated in the arithmetic unit are those which are significant from the programmers viewpoint. For example, a machine which has an accumulator register, and an auxiliary arithmetic register would require that locations in the object computer be used to simulate the contents of these registers. All operations which perform some modification of these registers in the source computer would, therefore, require that the subroutines which simulate those operations perform the same modifications upon the locations used to simulate these registers.

Auxiliary registers such as index registers, additional arithmetic registers, or temporary registers for buffering must also be simulated. Locations in the object computer are set aside to simulate the corresponding registers in the source computer. In addition to the various registers and address counters which are required, it is also necessary to simulate many of the indicators of the source computer by having a bit in the object computer simulate the function of the corresponding indicator in the source computer. For example, overflow in arithmetic operations may cause an overflow indicator to come on. Therefore, it is necessary that the object computer use a switch of some sort to simulate the overflow indicator. Similarly, such indicators as high, low and equal indicators for comparison purposes may have to be simulated.

Simulation of the input/output facilities of the source computer frequently present the most difficult problem. To simulate a card reader, for example, the object computer should have a card reader. It is not always necessary that this exact simulation take place, however, since it is possible to simulate a card reader by having card images on magnetic tape which are then read by the simulator program in the object computer. The simulator program simply reads in the next card image from this simulated card input data tape when the source computer program calls for a "read card" instruction. Through use of magnetic tapes on the object computer, it is possible to simulate a wide variety of input/output devices. It is necessary only to read from or write on the specific tape being used to simulate the device when the corresponding instruction is simulated in the source computers program.

One of the most difficult functions to simulate is the manual operation of the source computer console. In general, console operations are not duplicated, because the consoles of the two computers are usually quite different. It is possible to simulate most console operations by using control cards to set program switches, initialize registers, or perform similar functions. This implies that console operations which would normally be performed during a run would not be performed during a simulation run unless they had been preprogrammed in the input data in such a way that the simulator could recognize this control data and take the required action. If, however, the console facilities of the object computer are such as to permit easy duplication of the console functions of the source computer, it certainly is possible to permit on-line simulation of manual console operations during execution of the program.

The approach taken in the development of production simulators is somewhat different from that taken in the development of debugging simulators. A production simulator requires simply that the programs for the source machine be loaded and execution of these programs through simulation be initiated. The results of the computations may be recorded on magnetic tape or given directly as output via a printer.

A debugging simulator requires that considerable additional information be provided to the simulator in the form of control information specifying where dumps and traces are to occur, specifying the initial conditions of switches and registers, and providing for the changing of switches and registers during a particular simulation run. In order to accommodate these control cards, a program is required to read in the control cards, analyze the data on the control cards, and set up tables and indicators required for performance of the functions indicated on the control cards. This program is a preprocessor which is executed prior to the particular simulation run. Upon completion of a simulation run, control is returned to the pre-processor in order to read in the control cards for the next simulation run to be made. It is the pre-processor which enables the addition of debugging facilities, providing flexible control over the entire simulation run.

In some situations it is necessary to perform post-processing on the output generated by the simulation run. The results obtained during a simulation run may be recorded in a highly compact form for efficient simulation. However, such compact information is not amenable to easy interpretation by the user so that a post-processing run is required to convert the output to a form which the user can understand.

Another important characteristic of a debugging simulator is that it can be intimately connected with the assembly system which produces the object code which is the input to the simulator run. Although ultimately the output of the assembly program must be used as input to the source computer itself for execution, the output of the assembly program may be modified to be more convenient for use in the object computer which is being used in the simulation. For example, instead of paper tape for input to the source computer, a binary magnetic tape can be generated for direct input

to the simulator.

The relationship between the assembly program and the simulator can be extended even further. The symbol tables generated by the assembly program can be used by the pre-processor so that debugging specifications defining dumps and traces need not be in absolute machine language form. They can be specified using symbols which were used in writing the program originally. The pre-processor uses the symbol tables produced by the assembly program to convert symbolic debugging specifications to the absolute specifications which are used in the simulation run.

Simulator Generators

A logical consequence of the common characteristics of simulators is that one can abstract those features which are common to many different computers. A prime example is the instruction fetch routine which uses the program location counter to fetch the next instruction and then updates the instruction location counter. In the same way that emulators capitalize on this function and perform it by hardware, it is possible to write a simulator generator in which the same functions are performed by a general purpose routine. The routine to performing instruction fetch would be common to many simulators.

There are many common characteristics of computers which could be built into generalized subroutines. For example, binary computers using two's complement arithmetic, having a fixed word length and two arithmetic registers, as is commonly the practice, would all perform the basic operations of addition, subtraction, multiplication and division in essentially the same manner. Thus, the subroutines for performing these operations would vary depending upon only the word length of the computer. Carries from the low order arithmetic register to the high order arithmetic register, handling of overflows, underflows, operands with different signs, and storage of results, would all have common characteristics.

A simulator generator is a program in which the user can specify the characteristics of the source machine that he wishes to simulate, and the generator will produce specific subroutines which will perform the required functions in a fairly efficient manner.

There are many other functions which are common to simulators to which the same type of reasoning applies. Included are indexing addresses, indirect addressing, and conversion from decimal to binary or binary to decimal.

Obviously, any general system will suffer in that generality will reduce the efficiency of the specific routines written for performing that function. By restricting the routines to a class of machines, this loss of efficiency can be minimized. If the source machines are sufficiently similar, there can be great savings in the construction of simulators. Thus, it would not be economical to construct a simulator generator which would handle fixed-word-length binary machines as well as being able to handle variable-word-length decimal machines. However, if the source machines are all binary but of different word lengths, even if the number representation may be different in the sense of one's complement vs. two's complement vs. magnitude and sign; the routines to perform these functions can still be written with very little loss of efficiency.

A simulator generator would be useful for an installation which has one machine which is to be used as the object machine for simulation and several source machines which are of different characteristics. Rather than write a specific simulator for each of the source machines on the object machine, a simulator generator enables one to easily build simulators for each of the source machines to run on the one object machine.

Emulation

A recent development in the computer industry has been the use of emulation to facilitate the conversion from one computer system to another computer system. Emulation is a means of facilitating simulation through the use of hardware features in the object computer. Emulation is obviously economical only when a large family of machines is replaced by another family of machines. An example of this situation is the IBM System/360 in which emulators are available for various models of the 360 to assist simulation by many of the 7000 series IBM equipment.

In terms of the simulators discussed previously, emulation is obviously of value only as an aid in performing the functions of a production simulator. Use of hardware features to aid simulation is obviously much more efficient and results in faster execution of the source computers program than does pure simulation. As was indicated, speed is the most important factor in a production simulator so that emulators are of great value here.

On the other hand, emulation is not an efficient approach to building a debugging simulator. One characteristic of unbuilt computers is that design changes frequently occur. These changes necessitate only program modifications in a program simulator, but may require major hardware changes in an emulator.

The description of simulation and the functions performed in simulation provide the basis for determining those functions which an emulator can be the greatest value in performing. The heart of any simulator is the loop in which the simulated, program location counter determines the location of the next instruction. Since this loop must be performed for every instruction executed, the use of hardware facilities to perform these functions would vastly improve the efficiency of the simulator. This is one of the functions of a simulator which is performed by hardware in an emulator.

Having fetched the instruction, the operation code for the source machine is examined and control branches to one of a series of subroutines to perform the instruction. Depending upon the complexity of the source machine and the similarity of the object machine's hardware to the source machine's hardware, some of the operations can be performed by the addition of other

Continued on next page

hardware. In some cases however, the source machine instruction is so different, that the structure of the object machine does not enable economical implementation of the instruction in the object machine. In this case, a subroutine is executed which performs the functions required to simulate the source machines instruction. This subroutine is identical to what would be required if a full simulation of the source machine were done.

Simulators and emulators are not two mutually exclusive ways of performing the same function, but are instead two techniques which implement each other. Emulation is able to perform functions much more efficiently and quickly than a simulation program can do. This speed is of great significance in a production simulator, but, it is not economical to emulate all of the source machines instructions. Simulation should be used to perform the functions which cannot be economically performed by hardware emulation.

Simulation of the Control Data Digital Control Computer on the IBM 7030 Stretch

The Control Data DCC is a binary machine having a 24 bit word length. This computer is one of the fire control computers used in the Polaris system and has input/output instructions which are specifically related to weapon control functions. Since it is a real time computer, the timing of the program is very significant so that the simulator must take these timing considerations into account.

The heart of the simulator is a table called the object-time table. Events which are time dependent cause an entry to be placed in this table so that the event can be activated by the simulator when the proper time occurs. The simulator keeps track of time in a simulated DCC clock by accumulating the time required to execute each DCC instruction.

Entries in the object-time table are kept sorted in increasing time sequence. The value of the programmed DCC clock is compared with the time associated with the entry at the top of the object-time table at completion of simulation of each DCC instruction. When the comparison indicates that the event is to be activated, control is transferred to a subroutine to perform the event. The events which can be specified by the object-time table are primarily associated with input/output operations, such as reading a character into memory, writing a character to the printer, transmitting data to the missile, or an interrupt caused by the return of data from the missile.

The approach used in the DCC simulator permits a detailed timing simulation not only of the internal program functions but also of the asychronous input/output operations. Users of the DCC were able to debug their programs before the machine was actually available. Included in the debugging of DCC programs was the checking of intricate timing relationships required to assure successful operation of the real-time program.

Simulation of the IBM 650 on the Honeywell 800

The IBM 650 is a decimal-digit drum-memory computer whereas the Honeywell 800 is a 48-bit core-memory computer which can perform both decimal and binary arithmetic. The approach to interpretation of 650 instructions on the H-800 was determined by the fact that the H-800 could perform both decimal and binary arithmetic.

Each 650 word is simulated in one H-800 word. The format of the word simulated, however, depends upon whether the word is an instruction or data. By storing 650 data in decimal form, the H-800 decimal arithmetic features were utilized. This speeded up the simulation of arithmetic operations.

On the other hand, the H-800 requires binary addresses to access the words of memory which are used to simulate the 650 words. Addresses in the 650 program are in decimal so that they must be converted to binary for the H-800. This problem is solved by representing 650 instructions in a blocked binary format. This means that the two-digit operation code of the 650 is represented by seven binary bits in the H-800, and each of the four-digit addresses of the 650 instruction is represented by fourteen binary bits of the H-800. Three additional bits are used in the H-800 representation to tag the addresses.

Using this technique, instructions for the 650 are stored in a convenient form so that they can be picked up and interpreted without further conversion. Data which these instructions operate upon is in decimal format so that the decimal operation capabilities of the H-800 can be employed. Problems arise when the 650 instructions are modified arithmetically. Since they are not in correct format, it is necessary to go through a conversion from the blocked binary format to the decimal format, to perform the modification of the 650 instruction. Similarly, if data is ever interpreted as an instruction, it is necessary to convert from decimal format to the blocked binary format. The additional bits in H-800 words are used to indicate the format so that the simulator performs the necessary conversion when required.

Simulation of the IBM 650 on the IBM 704

The IBM 704 is a 36-bit binary machine whereas the 650 is a 10-digit decimal machine. The approach taken for representation of a 650 word on the 704 is as follows: 1 bit represents the sign of the 650 word, 7 bits represent the two-digit 650 operation code, and 14 bits represent each of the four decimal-digit addresses. Thus, the entire 36 bits of a 704 word are required to represent a single 650 word. This format is called a blocked binary format and is similar to the format in the H-800 simulation of the 650 except that no additional bits are available for control purposes.

One characteristic of the 650 programs being simulated is that the input programs required for simulation utilize a floating-point interpretative system for the 650.

Continued on next page

This system is capable of including pseudo-instructions in addition to 650 machine-language instructions. In these programs the data which the floating-point interpretative system operates upon is always segregated from the instructions which operate upon that data. Thus, no arithmetic operations are ever performed on the instructions using the floating-point interpretative system. Nor does the reverse take place, that is, in no case are 650 machine-language instructions used to operate upon the decimal data words which are the data that the floating-point interpretative system operates upon.

Segregation of 650 instructions and floating-point data makes it possible to provide a much more efficient simulation system. First, the 650 operation codes are simulated to perform exactly the functions required by the 650. Only 33 of the basic operation codes need to be simulated since the operations of multiplication and division and their variations were always performed in the floating-point interpretative system.

A second part of the simulator is a 704 interpretative system which interprets the pseudo-operations as the 650 does with one exception. The interpretation is implemented directly by 704 subroutines, rather than through simulation of the 650 routines which interpret the pseudo-instructions.

The approach of simulating only the basic 650 instructions and providing a 704 floating-point interpretive system results in much greater efficiency in executing 650 programs on the 704. As a result 650 programs operate approximately 40 times faster on the 704 than on the 650. If a straightforward interpretation of the 650 without a second floating-point interpretative system is used, the simulation would run at a ration of 3 or 4 to 1 rather than 40 to 1. □