SIMULATION TECHNIQUES FOR MICROPROCESSORS

James R. Armstrong and Garry Woodruff
Virginia Polytechnic Institute and State University
Electrical Engineering Department
Blacksburg, VA 24061

Abstract

In recent years the popularity of microprocessors has increased to a point where there exists the need to effectively simulate a microprocessor in conjunction with other logic elements. The two major problems of simulating the microprocessor are the level of detail in the simulation and the method of implementation. A simulation model is developed that allows effective simulation at the chip level with the other logic elements.

Introduction

The advent of the micro-processor is having a major impact on digital system design. With the cost of these devices in the rapid decline, more and more designers are using microprocessors to perform functions previously performed by hard wired logic. With this greater usage of microprocessors, there exists the need for methods and tools to aid in the design, implementation, and maintenance of systems employing these devices.

Simulation has played a large role in the development and maintenance of digital systems. However most logic simulators do not possess the capability of effectively simulating a microprocessor, i.e., simulating the microprocessor in terms of its relationship to other logic devices. There are presently simulators available to perform simulation of a user's program, but these simulators do not attempt to realistically reproduce the actual timing of the microprocessor or its signal interface.

Because of the limitations of present simulation, the authors, during the past year have been involved in a research program to develop a "chip" level microprocessor simulator which effectively simulates the operation of a microprocessor in terms of how it reacts with other devices in the system.

The research has been supported by the Naval Surface Weapons Center at Dahlgren, Virginia. The Microprocessor Simulation Routine (MSR) that has been developed is intended to be a basic component routine in a Logic Simulation Program (LSP)[1] which the Navy uses for design verification and fault diagnosis of the digital systems in ship board equipment. As such, MSR has been designed to interface with the Logic Simulator Program. However, the program that was developed can also be used as a stand-alone routine.

The microprocessor simulator development has been carried out in two phases. The first phase consisted of the design of the necessary program structures and an assessment of their complexity. The second phase consisted of actually coding and checking out a routine for a specific microprocessor (the Intel 8080).

It is the purpose of this paper to describe the results of the program design efforts in the hope that the techniques and data structures developed there will be of use to others in designing their own microprocessor simulators.

General Model Characteristics

An essential characteristic of a microprocessor (in fact, any processor, is the fact that it is recursively dynamic, i.e., when given conditions at earlier times, the present state of the processor can be determined.[2] This means that a state table can be derived for the processor and used in implementation of the simulation. The use of a state table requires two basic procedures, one procedure that determines whether a variable or variables should change and another procedure that updates the variables according to these changes at the proper time. A terminology frequently used in simulation work is to say that any state change is an event. Implementation of the state table can then be said to involve scheduling of events and event occurrence. The scheduling of events is effected by placing an event in a queue to cause the occurrence of the event at some later time. (Generally, this queue is a time ordered structure with a lowest time in/first out (LTIFO) arrangement.) Event occurrence consists of updating the appropriate system variables at the proper time as specified by the queue entry. As will be shown, some events result in signal changes at the processor interface while others reflect purely internal state changes.

Another consideration in the design of any simulator is the manner of time advancement. The advancement of simulated time can be performed by two different procedures: The "time slice" approach and the "next event" approach.[3] Advancement of time in the time slice method is effected by using a fixed time increment to update the simulator clock. This method is advantageous for short simulation times and in simulations where there are many events happening at each time interval. The next event method advances time by using a LTIFO (lowest time in - first out) structure. At each step, the simulator clock is advanced to the first time in the structure. By advancing the clock to the next event occurrence time, the timing accuracy is limited by the computational accuracy of the host machine. This is in contrast to the time slice approach where the time increment and thus the accuracy is fixed. The next event approach is advantageous for longer simulation times wherein the events are distributed randomly in time.

As was indicated above, the microprocessor simulation routine (MSR) was designed as a callable subroutine within a logic simulation program (LSP). LSP uses a "hybrid" method of time advancement in that the time slice approach is used for near events and the next event approach is used for distant events.[4] As shall be shown, MSR schedules its events by placing them in the LSP queue.

Perhaps the most important consideration in choosing the simulation model for a microprocessor is the level of detail in the simulation. While many levels of simulation are possible, the three most basic are: the system level, the chip level, and the register level.

The system level model has two essential characteristics. The first is that one is simulating the functions of the processor in terms of the final results, but no attempt is made to simulate the actual sequence of events. Because of this, the system level model is the simplest of the three to implement. The second is that simulation at the system level implies that one is simulating a microcomputer.[5] Figure 1 shows the system

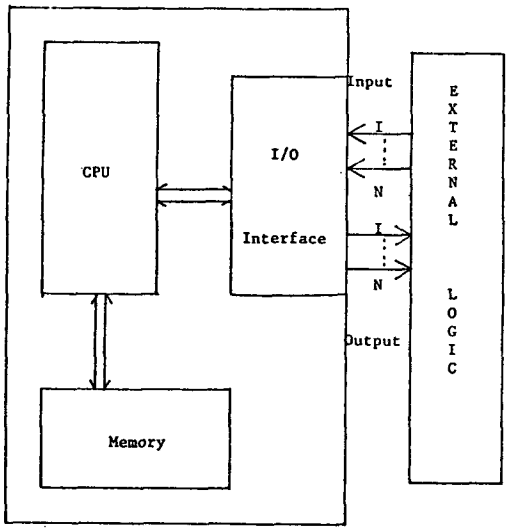

Figure 1
System Level Model

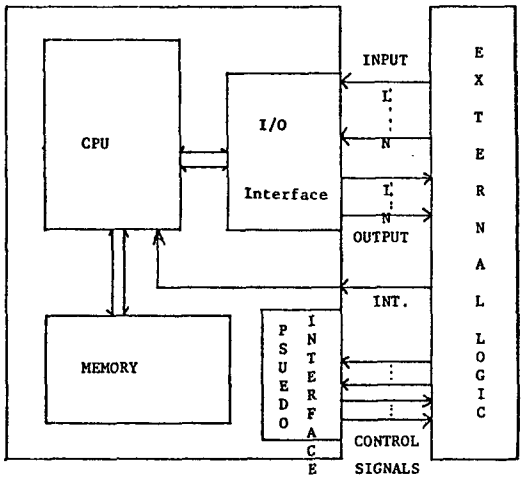

Figure 2
Modified System Level Model

level model. Note that memory and I/O logic are considered to be within the model boundaries, and that the interaction between the microcomputer simulator and any external logic or devices occurs at the I/O interface. This is essentially the model used in the simulators available from microprocessor manufacturers. One widely known example is INTERP/80, which performs a simulation of an INTEL 8080 system. INTERP/80 provides means to perform interactive program debugging. However, this simulator makes no attempt to simulate actual instruction timing or a realistic I/O interface, and thus is not useful as a logic level simulator.

The model shown in Figure 1 has some obvious deficiencies. It has no provisions for handling interrupts or the control signal interface necessary to adequately simulate the interaction between the microprocessor and external logic and devices. If one makes provision for this, the system model shown in Figure 2 results. Note that this model is the original system model with the addition of interrupts and a pseudo interface to simulate control signals. In adding interrupts and control signals to the model, one enhances its capability, but much of the simplicity of the original model is lost because program code must now be included to handle the logic and timing of these signals.

There is another major deficiency in the system level approach and that is: How do you define a generalized model of a microcomputer system? This problem is not evident in Figures 1 and 2, but there are a myriad of ways to set up a microcomputer system depending on the type of support chips used and the manner in which they are connected. More importantly, the resultant interface is peculiar to the configuration chosen.

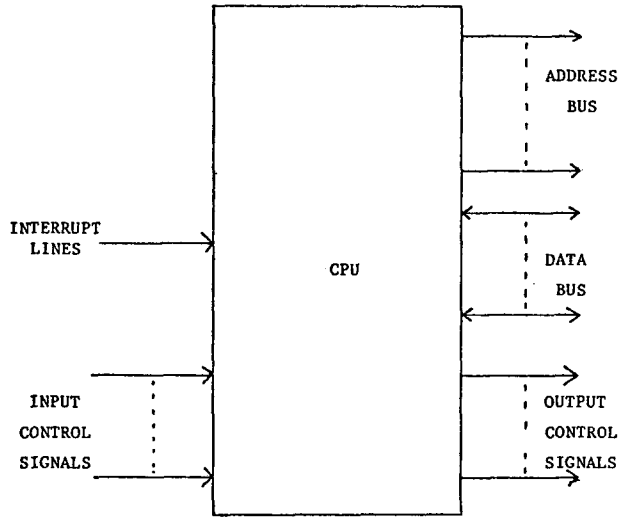

Figure 3
Chip Level Model

This deficiency is not shared by the chip level model shown in Figure 3. The chip interface consists, first of all, of address and data buses, which are signal interfaces common to all microprocessors. The

same can be said for interrupt lines. The control signal interface at the chip level also contains signals which are common to almost all microprocessors, e.g., system clock, processor synchronization pulses, status lines, and slow device ready lines. Thus, it is possible to develop a chip level model that is applicable to a wide range of microprocessors.

Another motivation for using the chip level model is that at this level, the interaction between the microprocessor and other devices takes place primarily via the data and address buses. This fundamental mode of interaction is shared by almost all interfaces with a microprocessor and thus simulation at this level should be applicable to a wide range of system configurations.

It should be noted that chip level simulation requires that there be other routines that simulate the other elements in the microcomputer system, i.e., RAM, ROM and I/O circuitry. However, most logic simulators have subroutines for simulating these devices, or they can easily be added.

One could simulate at the register level also, but this level of simulation is of little interest to the microprocessor user. He is concerned with chip input/output characteristics and does not care a great deal as to the manner in which a particular operation is carried out internal to the chip. (If one were a designer of microprocessor chips a register level simulator would be very useful, however.) Also, this level of simulation also adds unnecessarily to the complexity of the simulation, for as will be shown, when simulating at the chip level, one does not attempt to emulate completely the processor behavior but only what is necessary to simulate the signal interface.

In summary, the chip level model was chosen because of its ease of generalization, wide applicability to many system configurations, and reasonable level of complexity.

Program and Data Structures

Using the chip level model of the microprocessor, appropriate program and data structures need to be developed to implement the model. The microprocessor simulator routine (MSR) is composed of three routines and five data structures. The three routines in MSR are:

1) ACTGEN - the action generator. It's functions are analogous to functions performed by the control unit in a processor. It maintains the state of the processor, decodes instructions and calls an arithmetic logic simulator routine (ALUSIM) to perform arithmetic and data operations. In addition, ACTGEN serves as the main event scheduler for MSR. It schedules events which either cause signal line changes at the simulated chip interface or events which cause a return to ACTGEN for future activity.

2) ALUSIM - This routine is the arithmetic logic unit simulator. It is called by the action generator to perform arithmetic and data operations. The call statement specifies: 1) the type of operation to be performed, 2) the source of operand and destination of the results and 3) which processor flags are affected by the instruction.

3) SIGCHAN - This routine causes lines at the simulated chip interface to assume their appropriate values. As will be shown, its main function is to transfer designated portions of an internal buffer to an external line buffer.

The data structures can be grouped into three classes:

1) Translation Tables - Two translation tables are used to look up signal changes and actions for each instruction. They are static in operation, i.e., constant throughout the simulation. The specific functions of each of the tables will be discussed in a later example.

2) Internal Buffers - There are two internal buffers, one to hold the present value of the processor's registers and its present state (INTBUF) and another to temporarily hold future line changes (TLBUF).

3) External Buffer - The external line buffer (ELBUF) stores the simulated chip interface signals and as such is the interface for signal values between the microprocessor simulator routine (MSR) and logic simulator program (LSP).

System Operations

Figure 4 illustrates the system operation of the microprocessor simulation routine (MSR) and its interface with the logic simulator program (LSP). Note that there are two levels of interaction between MSR and LSP. The first level of interaction is the communication of signal values via the external line buffer (ELBUF). The event scheduling for the processor events comprises the second level of interaction, i.e., the processor events are placed in the LSP time queue. A minimal time queue entry for MSR has three elements:

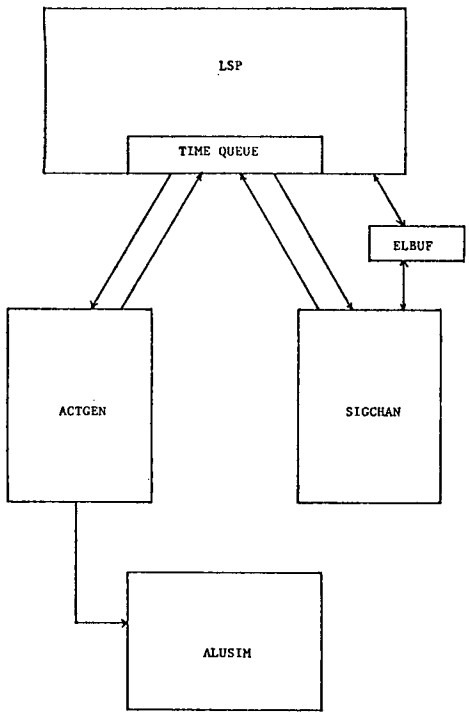

Figure 4
MSR System Operation

event time, subroutine code, and signal number. The event time is needed to denote when the event is to occur. The subroutine code is used to send control to the correct MSR routine. The signal number is a parameter passed to the routine to be used to identify the signal to be changed.

As designed, only ACTGEN and SIGCHAN interact directly with LSP. Each routine, ACTGEN and SIGCHAN, has a code associated with it. No signal value is needed for the ACTGEN routine since all of its information is self contained in INTBUF. However, a signal number is passed to SIGCHAN denoting which signal or signal group is to be changed.

The ACTGEN routine schedules events for SIGCHAN and for itself, and interacts directly with ALUSIM. SIGCHAN interacts only with the time queue mechanism of LSP and schedules only two events; one is the system clock signal and the other is for a system reset. ALUSIM is called by ACTGEN to perform the data operations of the processor and does not schedule events.

Translation Tables

A method to determine "what to do" and "when to do it" during the processing of an instruction is needed. The translation tables perform this task. TRANALL contains the information needed to generate the fetch cycle and all later cycles of an instruction. It contains information about signal changes and operations by instructions. TRAN1 determines the data operations for the particular instruction during the later part of the simulated fetch cycle. Entry into TRAN1 is by creating an integer index from the instruction code. Each entry in TRAN1 also contains a pointer back into TRANALL for signal generation and operations to be carried out during subsequent machine cycles (if any).

TBLPTR, the pointer into TRANALL, is used to chain through the subsequent machine cycles of a particular instruction, if any. The value of TBLPTR is determined by the value of the pointer field in TRAN1 and TRANALL. Whenever TBLPTR assumes the value of one, simulation of the instruction is complete and another fetch cycle begins.

Figure 5 illustrates the decoding of a multiple cycle instruction. With TBLPTR set to one, TRANALL is accessed for information necessary to simulate the fetch cycle. A result of the fetch cycle is that an instruction code is placed in the instruction register. The code for instruction i is used to compute a pointer to the ith table entry of TRAN1. This entry specifies the actions that arithmetic logic unit simulator (ALUSIM) must perform. The pointer entry in the ith word contains the value of TBLPTR required to access the information for the next machine cycle from TRANALL. Subsequent chaining through TRANALL is evident from the diagram. As each table access is made, the entry is used to schedule signal changes and actions during the simulated machine cycle.

Implementation Techniques

A number of techniques were employed in order to improve the efficiency of the microprocessor simulation. As was stated above the values of internal and external signals are stored in three buffers, INTBUF, TLBUF, and ELBUF. These buffers store register contents as well as individual signals. In our mechanization, register contents are encoded with one register bit stored in one element of the buffer, e.g., an 8-bit register requires 8 buffer elements. This was done to facilitate logical and arithmetic

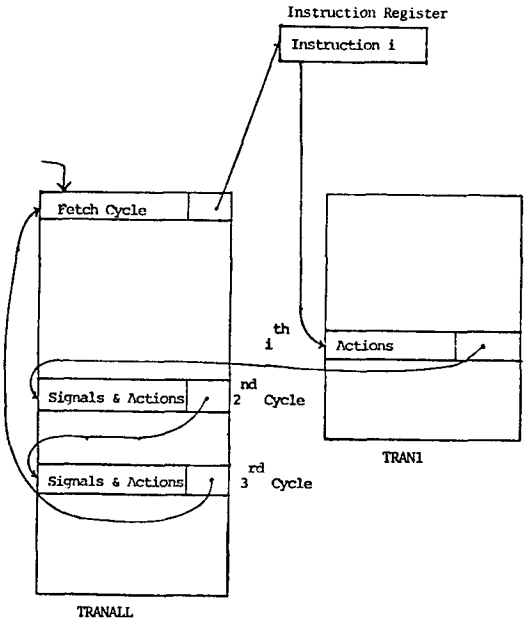

Figure 5
Multiple Cycle Instruction Decoding

operations on the registers in the ALUSIM routine. The effect of this "linear encoding" is to make these operations bit serial at the program level. This approach was found to be more efficient than storing a whole register contents in a single word because it avoids "packing" and "unpacking" operations. Also, the increase in buffer size necessitated by linear encoding is not significant.

As was indicated in the discussion of model characteristics, the simulation does not attempt to completely emulate the internal processor behavior. One example of this is the manner in which the simulation "compresses" processor cycles. If the effect of a particular processor cycle is purely internal, then the activities of that cycle are scheduled to occur during the preceding cycle, thus eliminating time queue entries and the resulting calls for these "internal" cycles. The simulator also compresses the instructions into eighteen (18) different types. When ALUSIM is called, the "type" parameter in the CALL directs control to a routine for each type. In addition, however, a routine for one type of an instruction can call another if the second type is a "subfunction" of the first, e.g., an increment instruction can utilize the add routine.

The use of the chip level model implies that memory operation is simulated elsewhere. As indicated above most logic simulators have subroutines that simulate RAMs and ROMs. However, in the case of the ROM, an initialization step is necessary. The ROM, of course, contains the microprocessor program and

fixed data. All manufacturers provide cross-assemblers for their processors. The output of the cross-assembler can be used to create a file of initialization data for the ROM.

Some other considerations in the design of MSR are the incorporating of tri-state capability, and a method for fault analysis.

Since the microprocessor has tri-state capability for most of its external lines, a method was devised to simulate this property. When the processor enters a tri-state situation, MSR will "float" the values in ELBUF, the external line buffer. The floating of the lines is accomplished by setting the appropriate lines in ELBUF to a logic "1" and then by making no further changes to those lines until the tri-state mode is excited. (Since ELBUF is the signal interface with other elements in the simulated logic net, the "floating" lines can be forced to 0 by other logic elements tied to these lines.)

A method for the handling of fault analysis is to associate a fault buffer with ELBUF. The buffer is used for stuck faults only. This fault buffer is set during each initialization phase of fault analysis in LSP. Thus when updating of ELBUF occurs, the fault buffer is used to mask any stuck faults into ELBUF causing a simulated stuck fault on the line(s).

Complexity

It would seem that the simulation of a logic net containing a microprocessor could result in simulation running times that are prohibitive from a cost point of view. However, if one is already committed to analyzing a logic net at, let us say, a 10 nsec time scale, then the addition of a microprocessor simulator does not significantly effect the running time. This is because the microprocessor clock has a period of microseconds, while the basic simulator clock has a period in the nanosecond range. Thus the ratio of microprocessor events to other logic events in the time queue at any given time is relatively small.

As far as memory usage is concerned, for a host machine with a word size of 32 bits or greater, the translation tables and buffers require less than 4k of memory, while the code for the three routines ACTGEN, ALUSIM and SIGCHAN requires less than 3k locations. This assumes that the coding is done with a higher level language such as FORTRAN. Programming at the assembly language level could reduce these size requirements further.

Summary

Techniques for the design of a microprocessor simulation routine have been presented. A routine for the 8080 microprocessor, employing these techniques, has been written and is currently in operation. Bench mark tests have shown it to be 2 times faster than Interp 80, as well as providing a signal level simulation (which Interp 80 does not).

The present version of MSR is limited to a two-level simulation. Extension to multi-level simulation (to allow for signals with an unknown state and timing analysis) would require changes to ALUSIM to implement the multi-valued data operations as well as change in element width in the dynamic buffers (INTBUF, ELBUF, and TLBUF). Neither of these problems appear to be significant.

The techniques presented here can quite easily be applied to other microprocessors. Writing of a routine to simulate the Motorola 6800 microprocessor will be carried out this summer as a follow-up task under our present research contract.

References

1. Keiner, W. L., The NWL Logic Simulation Program: A Tool for Maintainable Digital Design, NWL Technical Report TR-3024, April 1974.

2. Emshoff, J. R., and R. L. Sisson, Design and Use of Computer Simulation Models, New York: MacMillan Publishing Co., 1970.

3. Gordon, Geoffery, System Simulation, Englewood Cliffs, N.J.: Prentice Hall, 1969.

4. Szygena, S. A. and E. W. Thompson, "Digital Logic Simulation in a Time-Based Table-Driven Environment, Part 1. Design Verification," Computer, March 1975.

5. Soucek, Branko, Microprocessors and Microcomputers, New York: John Wiley and Sons, 1976.