Proceedings of the 1985 Winter Simulation Conference
D. Gantz, G. Blais, S. Solomon (eds.)

An Experiment in Multi-level Modelling

Michael Whelan
Siemens Corporate Research & Support Inc
Princeton, New Jersey

Abstract
The primary goal of this experiment was to determine the advantages which multi-level modelling would provide to the designer of a digital system. It is, however, readily apparent that providing multi-level modelling facilities will complicate the tool builders task, since tools may be required to cooperate with other tools in order to 'gain' information concerning the behavior of the system which is modelled. Hence, a secondary goal was to examine the problems presented to the design system implementor, by multi-level modelling.

1. Introduction

Multi-level modelling is the process of constructing, and gaining information from, a model of a system, wherein subparts of that system are modelled at different levels of abstraction. Multi-level modelling has proven to be useful at the logic/circuit abstraction interface[5], and it has been hypothesized that multi-level modelling offers advantages to the designers of complex systems. This paper concerns itself only with the design of digital hardware, and in this context, 'system' will be used to describe digital hardware systems only.

More specifically, the goals of this experiment were to construct a multi-level model (mlm for short) which used two levels of abstraction, and to provide an environment wherein tools appropriate to each level could cooperate in such a way that the behavior of the composite system could be observed. The levels of abstraction to be supported were the logic and register transfer (RT) levels. The task of building the aforementioned environment would yield a measure of the difficulties which providing multi-level modelling facilities imposes on the tool builder. Using the multi-level model to study the behavior of the composite system would provide an indication of the advantages to the designer of such a capability.

This report presents details of the experiment and a discussion of what was learned. Section 2 details the system which was modelled, and how the submodels at the different levels of abstraction were intended to interact. Section 3 discusses the manner in which the multi-level modelling environment was provided. Section 4 discusses the user's interactions with the multi-level model. Section 5 discusses the results of the experiment in the context of the goals, and Section 6 concludes with a discussion of what was learned, and directions of future work.

2. Models

For the purposes of the experiment, we chose to model a Fibonacci number generator. This was constructed from a 16-bit microprocessor, a memory subsystem, and an assembly language program which computed Fibonacci numbers. The microprocessor used was the Texas Instruments TI9900. This was chosen because we already had a register transfer level model for this machine, and we also had an assembler. This microprocessor was to be broken into two sections: an ALU for the implementation of arithmetic and logical operations (which would be the subject of a detailed logic level implementation), and a control portion which modelled instruction fetch and decode, and the execution of program flow (e.g. jumps/branches) instructions. This is shown in Figure 2.1.

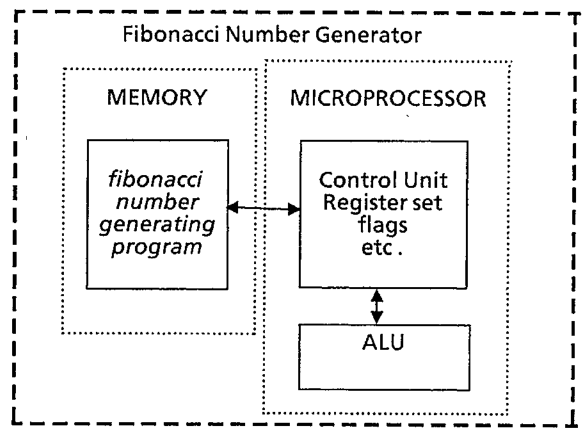

Fibonacci Number Generator
MEMORY
MICROPROCESSOR
fibonacci number generating program
Control Unit Register set flags etc.
ALU

Fig 2.1 Fibonacci number generating system.

The ALU was modelled at a boolean expression level of abstraction. The two models were to interact via a 24-bit bidirectional bus. In operation, the multi-level model would work as follows. When the register transfer level model for the control portion of the microprocessor encountered an instruction which required ALU action (e.g. an ADD instruction), it would pass the appropriate operands and control signals over the 24-bit bus. The boolean level model would get this information and would then proceed to do whatever it was instructed, returning a result to the control section over the bus. The conversion of data between the two models and the synchronization of the operations in the models was to be handled by two interface processes, one for mapping from the lower to the higher level of abstraction, and the other for mapping from the higher level to the lower level of abstraction. The operation of all these tools was to be orchestrated by a kernel communications processor, with the user interface being provided by a multi-window interface system. This is shown in Figure 2.2.

The models for the control and ALU portions of the microprocessor were respectively in the **N.mPc** register transfer level simulation system [3,4], and the **STUSIM** boolean level simulation system[2]. The mapping from the lower to the higher level process (identified as '*upmap*' in Figure 2.2) was provided by the TT theorem testing system [2]. The mapping from the higher to lower level model (identified as '*downmap*' in Figure 2.2) was a specially written piece of software. The communications kernel ('**cproc**') and the user interface processes were provided by the **vti** system [1].

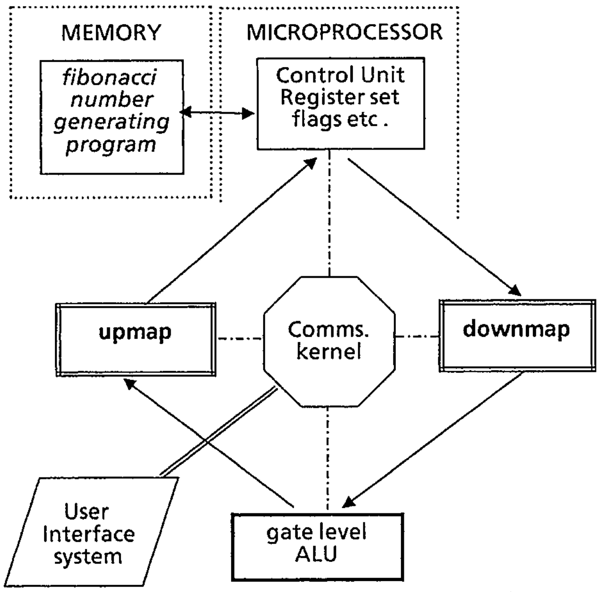


Fig. 2.2 Organization of the multi level model.

Let us first examine the single level model for the TI9900 at the Register Transfer (RT) level of abstraction. The following sections of ISP' (the register transfer level language used by the N.mPc system) illustrate the manner in which instructions are implemented in the single level register transfer model. The text in bold face is drawn from the ISP' model, while the italics are added here for the purpose of explanation.

The code excerpts shown in Figure 2.3 illustrate the algorithmic nature of the instruction decoding phase. The functionality of the ALU for this microprocessor is implemented by a combination of algorithmic control flow, and predefined (by the N.mPc system) arithmetic and logical operators which are applicable to arbitrarily sized operands. Such issues as variable data size arithmetic (e.g. byte versus word operations), and flags (e.g. overflow and carry) are handled by appropriate algorithmic control flow. For this reason, the code for **AOP** extends 16 bit fields to 17 bits, performs 17 bit arithmetic, explicitly tests the 17 bit result for overflow, and truncates the result to a 16 bit entity.

The intent of this experiment was to extract the functionality of the ALU from the ISP' model for the TI9900, to model this at the boolean level, and to have the two models communicate with each other in a simulation. The code for **AOP** which was discussed above illustrates several differences between the ISP' model for the TI9900 and the manner in which such a microprocessor would actually be built. First, 8 and 16 bit addition / subtraction are handled by different branches in the **AOP** code. In a hardware implementation, these different branches would share some hardware, i.e. there would only be one 'adder.' Another difference arises in the manner in which the potential for overflow is handled. The philosophy used in the **AOP** code is essentially that all fields are expanded so that no overflow can occur, then the result is checked to see if it can fit into the result field. This differs from a real implementation wherein one would not have a 17 bit adder and after the fact flag setting, rather, the logic which detected overflow would be part of a 16 bit adder.

```
main : =                    the main fetch - execute loop
  (
  fetch-1 . execute .  delay(1)
  )
depending on op-code branch to appropriate code
---
---
The AOP code handles addition and subtraction
of word and byte objects
AOP(op1<word>,op2<word>,op<bit>,size<bit>)
<word> : =
  (     code to handle 8 vs 16 bit data
  First sign extend operands
  OP1<0:16> = op1<0:15> ext 17 .
  ....
  Then perform addition on larger entities
  RESULT<0:16> = OP1<0:16> + OP2<0:16>
  AOP = RESULT<1:16> .            the result
  C = RESULT<0>                   a carry bit
  )
```

Fig 2.3 Excerpts from the ISP' code for the ALU.

In extracting the functionality of the ALU from the single level register transfer model for the TI9900, it was first necessary to define a logical interface between the ISP' model and the proposed logic level model. It was decided that the ALU would be connected to the remainder of the microprocessor by a 16 bit bidirectional data bus, an 8 bit unidirectional control bus, and ancilliary control signals as shown in Figure 2.4. Furthermore, it was necessary to decide on the internal storage capacity of the ALU (i.e. temporary registers). Once this had been completed, the connection of the two models could begin.

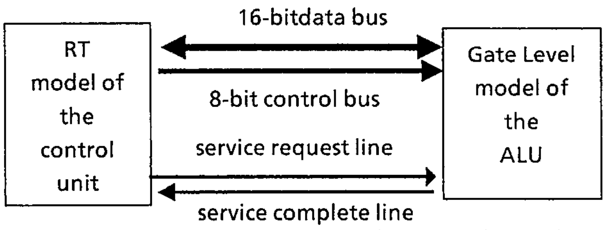


Fig. 2.4 Logical communication between the models.

The ISP' description of the TI9900 was altered so that whenever an arithmetic operation was required, the operands and control for this operation were placed onto a bus, and the result read from the bus. This was effected by replacing the original **AOP** code with the code as shown in Figure 2.5.

The 'xtoalu' routine wrote one 16 bit operand, and one 8 bit control byte to the 'external ALU' . This routine returns a 16 bit argument which is the data returned from the ALU over the bidirectional data bus. The **AOP** code works by transferring the operands of the operation to the external ALU, with instructions to load them into temporary registers (i.e. LRA means load register A with the contents of the data bus). Once both operands have been stored in temporary registers in the external ALU, an instruction to add (or subtract) the contents of the registers is issued and the result is passed back over the data bus. The xtoalu routine will handle the handshaking with the external simulator, and ISP' semantics are such that the **AOP** routine will be forced to wait until the xtoalu routine returns.

```
AOP(op1<word>,op2<word>,op<bit>,sz<bit>) <word> : =
  (
   .......
   RES<1:16> = xtoalu(OP1<1:16>,LRA); put op1 in temp reg A
   RES<1:16> = xtoalu(OP2<1:16>,LRB); put op2 in temp reg B
next ;
case op
   sub :(
    RES<1:16> = xtoalu(OP1<1:16>,SUB); issue subtract instruction
   )
   add :
   (
    RES<1:16> = xtoalu(OP1<1:16>,ADD); issue add instruction
   )
esac ;
next ;
AOP = RES<1:16>          return the result of the operation
)
```

Fig. 2.5 Modified ISP' code for the ALU.

The 'xtoalu' code actually converts the output data into a sequence of bytes which it writes to a memory. In the topology file for the N.mPc model, this memory is declared to be a raw memory, connected to a file name, and so becomes a serial link. The semantics of the 'raw memory' components in N.mPc are such that they must be 8 bits wide, and hence the data must be explicitly converted into byte streams for reading and writing.

The STUSIM model for the ALU has a 16 bit input bus, an 8 bit control bus, a 16 bit output bus and various status and handshaking signals. The model has three temporary registers, namely the A, B, and O registers. The A and B registers are used to contain operands, while the O register is used to latch the results of instructions. A state machine is used to latch data into the various registers and to interact with the outside world via a 'request' and 'ready' pair of handshaking lines. A state transition diagram, showing the interpretation of the state machine in the ALU is shown in Figure 2.6. For the purposes of this paper internal details of the ALU model are unimportant.

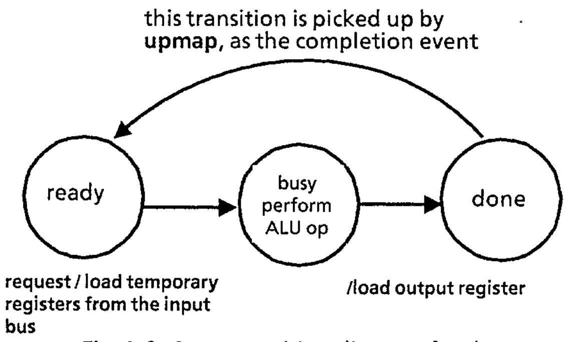

this transition is picked up by **upmap**, as the completion event
ready
busy perform ALU op
done
request / load temporary registers from the input bus
/load output register

Fig. 2.6 State transition diagram for the ALU.

2.1 Tieing the models together

Given that we now have a boolean level model for the ALU, and an ISP' model for everything else in the TI9900 microprocessor, we must consider the problem of running simulations of these models in concert in such a way that the boolean level ALU model actually performs the arithmetic operations which are required by the ISP' model. The components of the multi-level model are :

downmap
obtains the data which is to be passed over the data and control bus from the RT model to the logic model. The format of this data must be converted into commands to the STUSIM simulation of the ALU which achieve the desired setting of logic values in the boolean level model.

upmap
observes the boolean level simulation and at the appropriate time, must gather together the logic signals which define the entities of interest to the ISP' model, convert them back into a form which is meaningful to the ISP' model and pass them to it. In addition, this process collects data on the internal state of the boolean level model, and presents it to the user as indicated in Figure 4.1. The **upmap** process is implemented as a TT [2] specification.

vti
provides a multi window interface to the different processes with which a user may interact during the course of a simulation[1].

cproc
Each of the tools involved in the multi-level model is to be as ignorant as possible of the fact that it is not acting in isolation. This meant that an environment had to be created which allowed the individual tools to operate as if they were working independently, but which coordinated the tools so that the multi-level model could be simulated. The creation and management of this environment was the task of cproc [1].

3. System Implementation

In a single level simulation, each simulator will usually have as input a source of stimuli and, as output, will produce the response of the system being simulated to that stimuli. These input and output data streams are normally files. In a multi-level simulation however the input and output data streams will come from and go to other tools. Thus, part of the response of the boolean level simulator will be fed into the register transfer level simulator as stimulus, and visa versa. Each of the tools used in the multi-level simulation is set up so that its interactions to other tools is via single input and output data streams.

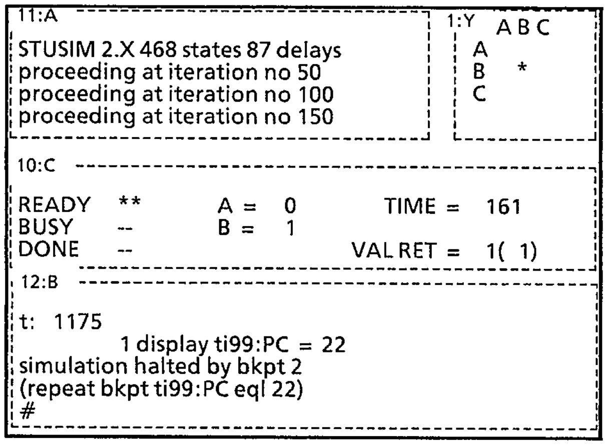
The inter-model timing is handled in the following way. When **downmap** sends a set of signal line stimuli to the **stusim** model, it adds commands to that model so that it will simulate for 6 time units (nominal gate delays). The **upmap** process observes the **stusim** simulation; and at the end of every simulation cycle, it sends one of two messages to the **cproc** process. If **upmap** has observed the completion of the requested operation (by observing the state of the state machine in the **stusim** model), then it sends a message to that effect, along with the value of the output bus, back to **cproc**. **Cproc** will then send that information on to the **ISP'** model. Otherwise, if at least 6 time cycles have been simulated, **upmap** will send a message to **cproc** which will cause a command to simulate for an additional time cycle to be sent to the **stusim** model. In addition a count of these signals is kept, and every tenth one causes an *incomplete* signal to be sent to the **ISP'** model. The effect of the *incomplete* signal at the **ISP'** model is to cause a 'delay(1)' to be executed. The combined effect is to enforce a 10 - 1 time ratio between the two models, when the ALU is running.

On the other hand, when the **ISP'** model is running, time stands still for the ALU. The latter situation is not very satisfactory but was forced on us due to practical considerations of run time and the fact that STUSIM is not an event driven simulator. Provided one can guarantee that there is no possibility of 'clock skew' being introduced into the ALU, (i.e. that the clock for the ALU is in the same phase every time a request is received from the **ISP'** model) then the multi-level model will operate correctly. This latter restriction seems reasonable for synchronous systems.

4. Results

The display of a users terminal during the course of a simulation run is shown in Figure 4.1. The top left window, which has the label '11:A' is associated with the boolean level model. The top right window, which has the label '1:Y' is used to indicate the communications which are taking place between the portions of the model. The '10:C' window is associated with the **upmap** process, and the '12:B' window with the RT level simulation. The number portion of the window labels are used in giving **vti** commands to interact with various windows. The letter parts are used by the '1:Y', window to indicate the interprocess communication traffic.

When the register transfer model decodes either an ADD or a SUB instruction, data is passed to the boolean level model where the operation is performed. Figure 4.1 illustrates the window contents just after the boolean level model had finished processing an 'INC x' instruction when x was zero. This indicates that the value returned was 1. The N.mPc model has been halted by a breakpoint, and is again awaiting user instructions. This breakpoint is at the head of the loop which computes the Fibonacci numbers.



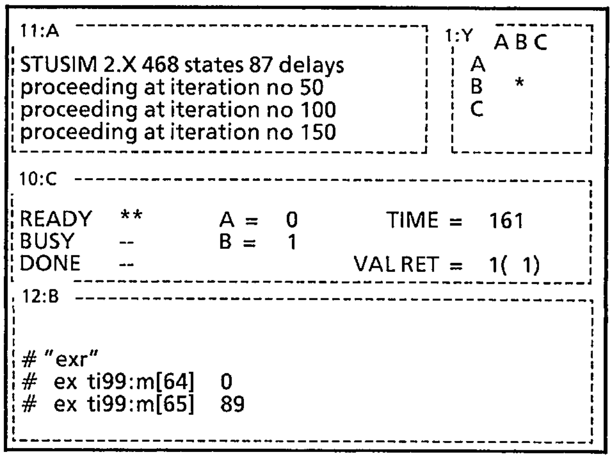
```
11:A                                        1:Y  A B C
STUSIM 2.X 468 states 87 delays             A
proceeding at iteration no 50               B    *
proceeding at iteration no 100              C
proceeding at iteration no 150

10:C
READY   **     A =  0        TIME =   161
BUSY    --     B =  1
DONE    --                VAL RET =   1( 1)

12:B
t:  1175
          1 display ti99:PC = 22
simulation halted by bkpt 2
(repeat bkpt ti99:PC eql 22)
#
```

Fig. 4.1 User terminal during the course of simulation.

Figure 4.2 shows the result of instructing the N.mPc system to execute the 'exr' command file, when the multi-level model is in the state illustrated in Figure 4.1. This command file causes the contents of the in memory registers for the TI9900 to be displayed. The memory is byte organized, and the workspace pointer was set to 0100_8. This means that memory bytes 64 and 65 are respectively the high and low order bytes of register 0. Figure 4.2 indicates that the contents of this register are 89, which can be verified as correct by examining the code for the Fibonacci generator.



```
11:A                                        1:Y  A B C
STUSIM 2.X 468 states 87 delays             A
proceeding at iteration no 50               B    *
proceeding at iteration no 100              C
proceeding at iteration no 150

10:C
READY   **     A =  0        TIME =   161
BUSY    --     B =  1
DONE    --                VAL RET =   1( 1)

12:B
# "exr"
#  ex ti99:m[64]  0
#  ex ti99:m[65]  89
```

Fig 4.2 After giving the 'exr' command to N.mPc. simulator.

5. Discussion

The intent of this experiment was to gain information on the utility of multi-level modelling to the designer and on the difficulty of providing the requisite features to the CAD system implementer. Let us first consider the difficulties which providing a multi-level modelling system poses to the CAD system implementers.

5.1 Tool Modifications

The multi-level modelling system which was constructed during the course of the work reported here attempted to change the tools which were used (namely N.mPc, STUSIM, TT) as little as possible from their normal stand alone versions. An additional complication arose in terms of the N.mPc system: since we did not have the source code available, we were not in a position to change this system even if we wanted to. The alterations which were made to the tools for the purposes of multi-level modelling were :

TOOL	CHANGES for multi-level modelling
N.mPc	none
STUSIM	a) provision of an auto initialization facility c) provision for creating auto initialization specifications d) flags processing for invocation of the above features
TT	none

In retrospect, one might argue that the changes made to the STUSIM system were in providing facilities which were lacking in the initial version. While this may to a certain extent be true, it had managed to serve a useful purpose when used in stand alone fashion without these features; whereas, these features were prerequisites for the multi-level modelling system.

Thus changes to the individual tools did not pose a serious barrier to the provision of the multi-level modelling facility. It is worth pointing out, however, that the tools were used in such a way that each had essentially one data input/output to the system which was used for interacting with the models at other levels. While one could imagine scenarios where this might be unappropriate, we do not feel that it is a real restriction under some assumptions about how such a multi-level modelling system would be used.

5.2 The utility of multi-level modelling to a designer

In order to discuss the advantages to the designer of a multi-level modelling system, it is first necessary to describe a design methodology wherein the use of the modelling facilities can be discussed. It should be borne in mind, however, that the tools discussed in this paper can be used in other design methodologies, and the potential advantages to the designer would have to be gauged in that light.

The design methodology which we shall assume is set forth in Figure 5.1 . As indicated, it is assumed that the design is being performed in a top down fashion starting with the design of a register transfer level model for the system. This model of the system is verified by running test programs on a simulation model and examining the results. This set of tests should be such that the functionality as described in the register transfer model is thoroughly tested.

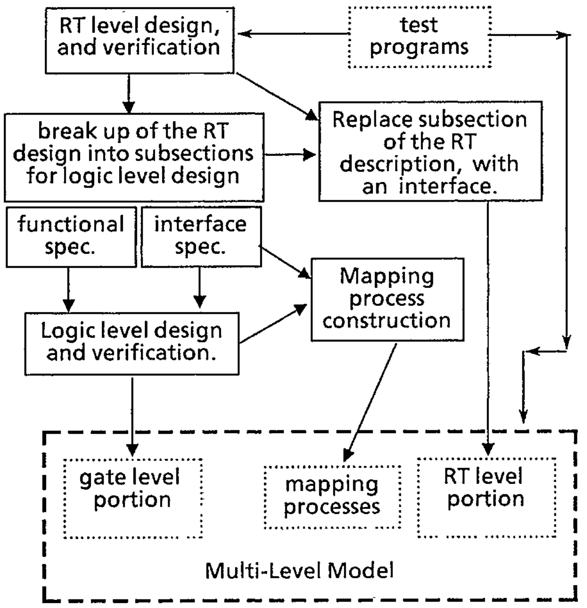

RT level design, and verification
test programs
break up of the RT design into subsections for logic level design
Replace subsection of the RT description, with an interface.
functional spec.
interface spec.
Mapping process construction
Logic level design and verification.
gate level portion
mapping processes
RT level portion
Multi-Level Model

Fig 5.1 Design methodology spanning the register transfer to gate levels.

When the design at the register transfer level has been completed, and confidence gained that it meets its requirements, the next step is to partition the functionality of the register transfer model into pieces which will then be designed at the logic level. This partitioning results in a set of submodule specifications, where the specifications can be broken into two parts, the functional specification and the interface specification. Figure 5.1 illustrates the continued design of one of these submodules, however, in practice, several would be designed simultaneously. The procedure as shown in Figure 5.1 would then apply to each submodule design independently.

The functional specification is used by the logic designer to design the submodule and to verify that it meets its functional specification. Parenthetically, the designer verifies that an 'adder adds' but does not yet consider the manner in which the adder must communicate with the rest of the system. Once confidence has been gained in the functional correctness of the design, the interface specifications are used to create mapping processes; and the submodule is verified 'in context' by constructing a multi-level model with the remainder of the system being modelled using the already verified register transfer level design and running the register transfer test cases.

An interesting point that comes up here is the 'start up' phase of simulation for the logic level system in the multi-level model. In the register transfer level of abstraction, the state of the model is automatically initialized as specified in the model (or by default). This means that there is no reason for an initialization process to get the model into a known state. Quite the opposite is true of a logic level design. In this case the state of the system is initially unknown, and it must be somehow 'set' into a known state. This is normally accomplished by a combination of control signal inputs which are applied at power on to get the system into a known state.

In the multi-level model then, the logic portion is initially in an unknown state and needs some control signal sequence to get it into a known state; whereas, the register transfer model is devoid of any notion of initialization sequences. How then is the logic level subsystem to be initialized properly so that it can take part in the multi-level model simulation. The initialization procedure is the logic level designer's responsibility (it is part of the design); whereas, the initial state of the logic system should be part of the submodule specification.

The way this problem was tackled in this experiment was to allow the logic level model to be 'auto initialized' to a known state upon invocation. Thus, when invoked in a multi-level modelling environment, the logic model would be initialized to a known state. This initialization state, however, had to specify the state of every node in the logic network, i.e. it was not simply a specification of something like 'areg = 10'. In order to construct this state description, the STUSIM simulator was provided with the capability to write the final state of a simulation to a file in the appropriate format. The logic level designer then simulated the logic level design with the initialization sequence to construct the auto initialization file which would later be used in multi-level modelling.

The latter issue illustrates that the design methology outlined in Figure 5.1, and supported by the multi-level modelling facilities as outlined here allocates responsibility for meeting specifications to the designers who are responsible, and ensures that a design (including initialization sequences) meets all of its specifications (including initialization, function , and interface) before the design can pass the multi-level modelling tests.

There are 'fall out' advantages to using the design methodology illustrated in Figure 5.1. As pointed out in [2], using formal, function specifications for design validation has the desirable side effect that the submodules being specified tend to be such that they have a functionality which can be neatly described at a level of abstraction above the level in which they are being designed, i.e. the design tends to be highly modular. The same effect is apparent in the case of interface specifications. Thus, alternative designs for a submodule may be examined simply by plugging them into the multi-level module. This ability is particularly desirable when examining the cost/performance effect of a submodule on a complex system.

As a final verification step, after each submodule, at the logic level, has passed its multi-level modelling tests, the entire system may then be simulated at the logic level as a single level model. The hope would be that by this stage almost all errors would have been detected, and so the simulation task would not have to be re-run frequently as errors were detected and corrected. Finding errors at this stage would also indicate the quality of the functional and interface specifications. This information would be valuable since interface specifications (and possibly functional specifications) will tend to be reused across many designs (e.g. an S-100 bus interface) and so improving them would yield bonuses in future designs.

6. Conclusion

The results of this experiment indicate that providing multi-level modelling facilities does not pose insurmountable problems to the CAD system designer, while the capability made available to the designer is very valuable. There are, however, several caveats. The points made in the conclusion of [2] in connection with drawing assumptions from a 'dummy' design example also apply here. Namely, the experiment which was performed did differ from a real design in several important respects.

1. The TI9900 model was not designed by us, we made use of one that was available.
2. The logic level model was not completed, i.e. it did not implement all the instructions which would be required in the real microprocessor.

These points are not independent. Indeed, it was our intent on starting this experiment to implement all of the functionality of the ALU in the boolean level model. We initially implemented the ADD and SUB instructions (and indirectly then the INC and DEC instructions). This did not cause any difficulty, since by coincidence, these instructions all used the **AOP** code in the ISP' description for their implementation. This gave us a 'handle' to extract their functionality neatly. The same, however, could not be said of other classes of instruction such as the logical instructions (AND, OR, etc.), which were implemented directly by ISP' operators as they were decoded.

When we attempted to implement the flags in the ALU we ran up against a similar problem. The setting of flags was also spread out throughout the ISP' code, and to make matters even worse, it was context dependent. For example, in determining whether to set the overflow flag after an INC instruction, advantage would be taken of the knowledge that one of the operands was '1', whereas after an ADD, a more powerful test had to be carried out. This caused great difficulty in trying to move the flag functionality to the ALU.

In retrospect many of the problems recounted above arose because of point 1. That is, we used a register transfer model which was constructed solely for the purposes of 'emulating' the behavior of an existing microprocessor. Consequently, the description in no way attempted to impose a design structure of the model's components. Thus, flag setting was done wherever convenient. Had the TI9900 model been restructured to impose a modular partitioning of functionality (or had it been written that way to begin with), the design methodology would have fared much better. One would assume that in an actual design (as opposed to a model written to emulate an existing component), the register transfer description would reflect the modularization of functionality.

We view the results of this experiment as encouraging both in terms of the utility of multi-level modelling in the design process, and in terms of the ease with which tools can be modified so that they can provide multi-level modelling facilities. Our future work will be moving the multi-level modelling facilities to higher levels of abstraction.

References

[1] 'A Design Environment that Integrates Tools, Database, and User Interface',
A Hsu, L-H. Hsu , P Ulrich, ICCD Port Chester, NY 1984

[2] 'Theorem Specification &Testing as an aid to Design Verification' ,
Michael Whelan, ICCD, Port Chester, NY 1984

[3] 'An Introduction to the N.mPc Design Environment',
Frederic I. Parke, 16th Design Automation Conference, San Diego , June 1979

[4] 'ISP' User's Manual' ,
Ralph Straubs, Computer Engineering and Science, Case Western Reserve University, 1978

[5] 'The INMOS Hardware Description Language and Interactive Simulator',
B Collins & A Gray, VLSI 81, Academic Press, London 1981, Editor John P. Gray

MICHAEL WHELAN
Michael Whelan received his Ph.D. in Electrical Engineering from the State University of New York at Stony Brook in 1982. Since then he has been employed by Siemens Corp. Research and Support in their Princeton New Jersey research laboratory. He is an adjunct faculty member in the computer science department at Stevens Instutite of Technology. His professional affiliations include the IEEE, the ACM and the New York Academy of Sciences.

Siemens Corp.
105 College Road East,
Princeton, NJ 08540

(609) 734 6561