Modular nets (MNETS): A modular design methodology for computer timers

by R. E. Matick

This paper describes a modular, graphical, fully implemented CAD tool for building timers to model computer pipelines. The complete system is composed of three parts which can exist independently but have been fully integrated to provide a user-friendly CAD tool. These parts are, first, modular nets (MNETS), a new modeling concept for modular, graphical implementation of pipeline structures of any kind; second, the implementation of various MNETS modules and macros in a VHDL library similar to logic and circuit design libraries; and third, the integration of parts 1 and 2 into an existing graphical entry framework, the EDA Wizard graphical editor. A graphical model is constructed by interconnecting basic building blocks using the graphical tool, similarly to the way circuits and logic are designed. Selection of a menu option will produce a VHDL description of this graphical model, which can subsequently be simulated on a VHDL simulator. This paper concentrates on part 1, the features of MNETS which make it inherently modular and consequently graphical. The two crucial requirements, namely the construct for storing of state and a control mechanism for the passing of state, are unique to MNETS and are discussed in detail, with comparisons to other methodologies. A brief discussion of some features and macros available in the existing MNETS library is included, as well as one simple modeling example. This library can be accessed on the IBM Andrew file system, AFS. A detailed MNETS user/design manual is available which describes MNETS in detail, as well as the library, memory hierarchy design, and modeling.

1. Introduction

MNETS is a unique pipeline-modeling methodology with the key advantage of providing true modularity in the construction of a timer model. (A timer is a trace-driven model that counts the number of processor cycles required to pass a given instruction stream through a pipeline.)

0018-8646/98/$5.00 © 1998 IBM

This modularity is inherent in the basic structure of MNETS and provides the additional advantage of being readily adaptable to graphical-editor interfaces for constructing any model. The key features which permit this are the use of a well-defined construct for holding state, and well-defined, never-changing, *local control signals* for the passing of state. Each state of a pipeline is represented by a token in a box (latch), which is clocked like an ordinary pipeline latch; simple logic, local to each box, determines whether any given token will pass from its source box to a destination box based on the current state of the source box and simple, modular control signals received from target boxes. Every box which holds a state is a source on its output side and a target on its input side, similar to a master/slave latch. Thus, the logical structure of MNETS is very similar to that of an actual system pipeline. In essence, it is an extremely simplified version of a small portion of the logic of the actual computer pipeline being modeled. As a result, an MNETS model will look and behave much like the actual pipeline structure. To build a pipeline model, several fundamental modules are required; these are interconnected by the designer to create a specific model.

In using the full MNETS system, a user creates a model by instantiating library modules and macros on an AIX* window using a graphical editor as the user interface. Connections are made in the usual "rubber-band" fashion. Once the model is graphically constructed, the selection of a menu option produces a compilable VHDL code file. From this point on, any simulator capable of handling VHDL record data types can be used.[1] The combined MNETS, graphical editor, and VHDL system provide a widely available, versatile set of tools that can be used to enter, edit, modify, simulate, debug, check, and capture a design in a highly modular, easily understood fashion.

The MNETS concepts are independent of the input method used, and thus can be implemented in any programming language.[2] VHDL has been chosen as the underlying language primarily because graphical input methods and simulators are readily available. For the graphical user interface, the EDA Wizard graphical editor was chosen since it is also readily available, is very user-friendly, and can produce a VHDL description of a graphical model. Various graphical editors can be used, since none of them needs nor has any knowledge of MNETS; e.g., there are no constructs in Wizard itself which are specific to MNETS.[3]

Although MNETS uses the VHDL language at its basic root, the user need not know VHDL unless a custom macro is needed which is not in the design library, cannot be built from existing library functions, and cannot be obtained elsewhere.

The application of MNETS concepts is not limited to pipelines. In fact, the constructs of storing and passing of state can be used in other programming tools, as discussed in Section 11.

2. Comparison of MNETS with other timer modeling tools

Readers familiar with circuit design systems know that one can build and test software models of circuits using a completely graphical interface provided by computer-aided design (CAD) tools such as ASTAP, ASX, SPICE, and Cadence. In such systems, the user typically "instantiates" within the working window of a computer screen the various necessary transistors, capacitors, and other components available in the design library to build the model. The components are interconnected by using a mouse to click on component terminals and wire the components in a straightforward manner. The graphical editor typically provides a variety of services to permit editing and customizing the model. The model can be simulated at any stage by compiling and running it with a choice of input waveforms. Typically, the CAD system also provides various tools for analyzing and debugging the model, as well as capturing the results. In an analogous manner, a logic designer can build and test logic models using a completely graphical interface for entering, testing, and capturing logic circuits. In fact, such logic tools are typically a part of the circuit design systems mentioned above.

In order to understand the significance of MNETS, it is important to recognize that while many graphical input/editor systems exist (e.g., Summit Design, View Logic, SimScript), such systems typically require the user to define the basic objects for interconnections. The choice of objects is of fundamental importance. For circuit analysis, objects consistent with circuit design are essential: transistors, capacitors, resistors, etc. Similarly, we know the objects consistent with logic design to be AND, OR, and NOT in one logic family, and NOR gates and NAND gates in others. These building blocks inherently include a set of consistent, well-defined input and output signals. The total design system in these cases includes a library of such constructs, and usually some useful macros built entirely from the basic constructs. Thus, even before we start to construct a circuit or logic model, we have clearly in mind the basic building blocks and interconnection signals which are to be used. *For modeling computer pipelines, we need comparable basic objects which are consistent with pipeline design. Such objects are nonexistent or not obvious, and as a result, each modeler has traditionally built his/her own such objects or*

[1] A Cadence MTI (Model Technology Inc.) VHDL simulator was used in this work.
[2] Pascal was originally used. Various processor pipelines have been modeled in MNETS using Pascal programs (see [1]).
[3] Other graphical interfaces can be and have been used—Summit Design Inc. Visual VHDL, View Logic, or SimScript.

"procedures," with his/her own input, output, and control signals. MNETS provides these basic objects and signals, and has all of the desired features described above. This is the fundamental purpose of MNETS.

The lack of basic, consistent objects for pipeline design has resulted in a variety of modeling techniques. Often, the modeler builds the entire model using computer languages such as C, C++, Pascal, FORTRAN, or PL1. Invariably, such models are nonmodular, not reusable, and require the assistance of the original coder to understand, change, and debug. As a result, a second approach is to implement a pipeline design language which attempts to circumvent some of these limitations. However, these are invariably quite cryptic, and difficult to learn and to remember. The basic building blocks are higher-order logic constructs but are implemented as code words and parameters. While some systems may provide limited graphical input assistance, none have primitive graphical pipeline building blocks nor a graphical interface which even begins to resemble those available in circuit and logic design systems. As a result, such tools for building timers are not only extremely difficult to learn and use, but once the model is constructed, there is still no single, consistent, graphical view of the design, just code. It is difficult for even the designer to go back and determine just what the model is doing. There is virtually no reusability, and new models are typically constructed "from scratch."

In direct contrast to this, the inherent properties of MNETS allow the user to construct a timer model of a computer pipeline, memory hierarchy, or any "clocked" information flow process, in a manner analogous to that used by circuit and logic designers. MNETS consists of a small set of design modules (objects) from which the user can construct a timer/state flow control model of any pipeline. Just as a logic designer can build any logic function or computer from the three basic gates AND, OR, and INVERT, simple pipelining in MNETS requires three types of MNETS modules, and typically four to six fundamental types of modules can be used to build an average pipeline, including decoders and virtual-address translation modules. Such modularity greatly simplifies the overall logical complexity.

In addition to the object modules, MNETS provides a variety of macros from which a large number of models can be constructed. Some of these macros are functions, such as address-translation units, decoders, and delay macros for modeling chip-crossing delays. If the designer needs a custom macro, it can be constructed and embedded in the library. In many cases, new macros can be built from existing library modules, using the interactive graphics tool. However, if the required function cannot be built in this manner, the final logic of the macro is coded in the VHDL language.

3. Similarities and differences between MNETS and actual pipelines

The essential similarity between MNETS and an ordinary pipeline is that both use latches to hold state, the time to transfer state from one latch to the next downstream latch is the system cycle time, and the transfer/nontransfer control of state between latches is an inherent part of the pipeline structure. The difference is that an ordinary pipeline contains all of the logic, both data flow and control, to process the data. Timers assume that the data processing will be correct logically and that all logic functions will complete correctly within the allotted machine cycle time. A timer will only "time," or, literally, determine the *cycle count* to answer the question, "How many machine cycles will be required to process a given instruction trace through a given pipeline?" A trace is just a flat list (branches unrolled) of all of the actual instructions which a processor would "see" in running the program. We can think of each of the various registers (state holders) as holding a token, which is passed or not passed to the next register of the pipeline, depending on certain control signals. These control signals will stall or hold a token (token = state) in a given register whenever the downstream target register cannot accept the token (state) for whatever reason. Stalls are not predictable *a priori* and are the source of deviation of the actual processing time from any simple analytical calculation which assumes no such stalls or conflicts.

Given the idea of state as basically a token which is passed and timed through the pipeline registers, the most important difference between MNETS and usual pipeline models, as well as all other design methodologies, is the concept of *local control for the passing of state.* Other systems in the past have allowed basically an unconstrained, unstructured, global type of control logic. This invariably leads to code which can almost never be modular or reusable. In addition, such complex control structures render the model understandable only to the original coder, and changing and reuse are extremely difficult. Also, one cannot look at the model and have even the most simple idea of what is modeled, or of the general structure. MNETS simplifies all such control structures by using a standard set of four local control signals at each module of the pipeline which controls the flow of the state token. This gives MNETS the property of being modular and easily changed. For many cases, changing the pipeline consists of removing or adding a pipeline module and plugging in the same control signals again. This is illustrated later[4] in a modeling example.

[4] Users familiar with Petri nets will recognize a similarity to MNETS. However, not only do P-nets lack a well-defined structure for representing state, but there is no structured method for control of state passing. That is what "timed" P-nets attempt to do, but still in an unstructured manner and thus still unsuccessfully. See [1] for a discussion of Petri nets.

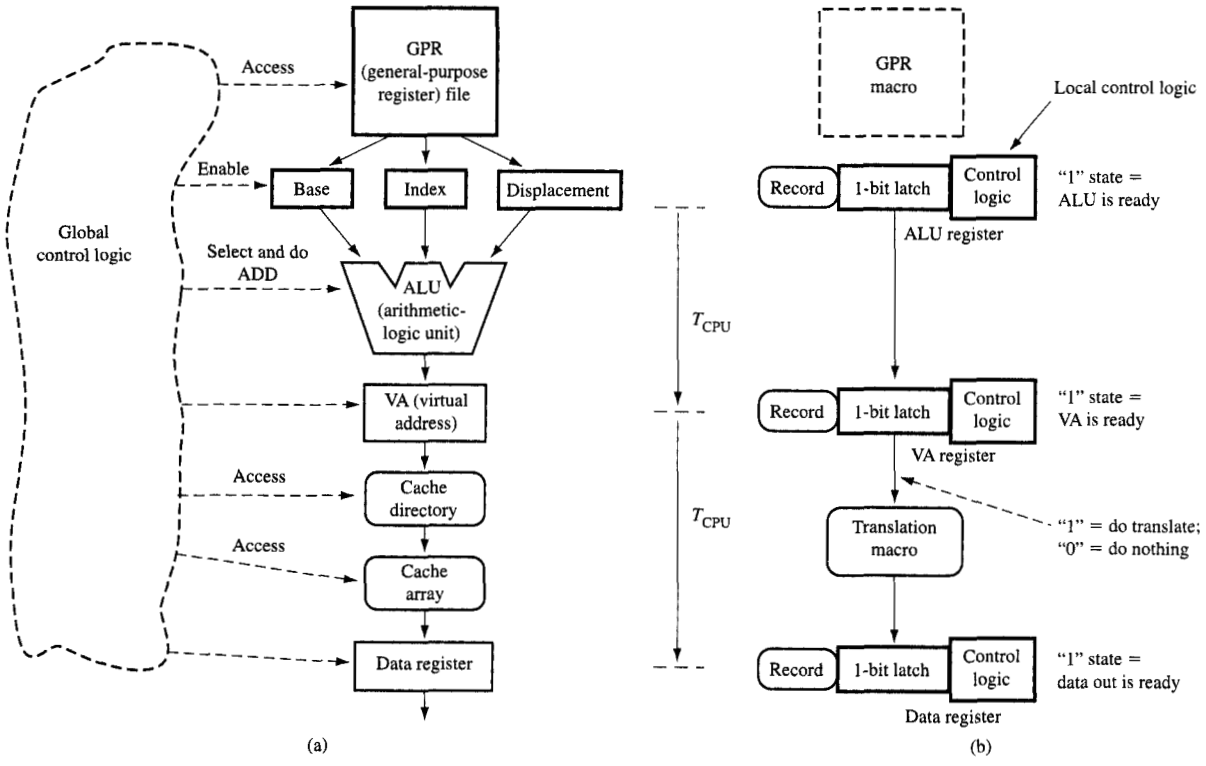


Figure 1

(a) Simple computer pipeline containing virtual address generation and cache access; (b) equivalent MNETS model.

4. Essential concepts in MNETS modeling

Let us start with a very simple portion of a computer pipeline to see how this part might be modeled in MNETS. We use a pipeline path extracted from the fixed-point unit of a generic processor, which consists of virtual address generation and cache access.[5] This is shown in **Figure 1(a)**. We start with the access of the general-purpose register (GPR) file for three operands of 32 bits each, which are latched in the base, index, and displacement registers. This represents one stage of the pipeline. At the next pipeline stage, these three operands are added together to produce a virtual address (VA) of 48 bits (size is not important) latched in the VA register. This VA is used in the next pipeline stage to access a cache directory and array. We assume a one-cycle cache, meaning that both the directory and array access are completed in one cycle, and the data will be latched in the data-out register at the end of this cycle if a cache hit results. If a cache miss results, we assume for simplicity that the entire pipeline halts (stalls) until the missed data are retrieved and the pipeline is restarted, causing the translation to be redone. In an actual pipeline, the data register would typically be written back into one of the GPRs in addition to being latched in a register, but we are neglecting this for now.

For a fast pipeline, all three operands would be accessed from the GPR file on the same cycle. These three integers would have to be added with appropriate modulo shift to give a 48-bit result. Various bit fields of this 48-bit virtual address will have to be selected appropriately in order to address the cache directory and array correctly for the particular type of organization (e.g., late-select or other, congruence class/set associativity size). The cache array is accessed, and typically 64 or 128 bits of data are latched in the data-out register.

So far, this path specifies only the flow of data. In parallel with this, and usually not specified, is the control logic. Its path is not so simple, nor clear. It must control the enabling of all of the registers, GPR file, directory, and array, and also send signals to the ALU (arithmetic/logic unit) specifying and performing the ADD, etc. so as to maintain proper flow and latching of

[5] For more details about MNETS modeling of this type of pipeline, see [1, 2]. The design manual of [3] deals mainly with memory hierarchy pipelines.

the data. This logic tends to be customized to the application and is usually much less well ordered than the data flow. In any case, a timer would model only the state flow and not the full logic of the pipeline.

To model this logic using typical state-of-the-art methodologies, no consistent method exists, nor are there any simple or graphical constructs to aid in the modeling. The programmer might decide to consider each state of the system as an entry in a two-dimensional array, or a linear list, or in any of a number of other formats. A variety of methods have been and continue to be used. There are design languages which are intended to aid the construction of pipelines, but none are graphical or have the reusability feature of MNETS. No matter what method is chosen for representing state, all lack the simple, consistent, well-defined system for storing of state and control signals for the passing of state that are needed for modularity and reusability. Control signals tend to be global, much like that of the actual pipeline (although of course simpler, since only part of the pipeline is modeled). As such, the control structure ends up being custom code, which is seldom reusable. It is not possible to give a graphical representation of the model and relate it to the corresponding parts of the pipeline, since a one-to-one correspondence does not exist.

By comparison, an MNETS model provides a simple representation of the data path, understandable by any designer, as shown in **Figure 1(b)**. For the sake of understanding, we start at the output of the GPR file. Since a timer does not typically process any data, there is no need for any of the data contained in the three operand registers. Instead, the fact that an access and a pending add are ready is indicated by a 1 in a one-bit ALU register latch, which is the state or token specifying this condition (the record part of this latch is discussed shortly). Continuing downstream along this same path, the ALU need not be modeled at all, since no data are processed. The virtual address, VA, will previously have been calculated and is contained in the input trace. This VA is passed along as an integer field of the record[6] part of the one-bit state latch. Thus, the 48-bit virtual address register in the actual pipeline is replaced by a one-bit VA register latch with an attached integer field in a record.

In most cases of interest, directory translation for hits, misses, and appropriate timing of all reloads is necessary, since these are a crucial part of the pipeline stalls. In MNETS, there are VHDL logic macros available for various directory organizations (see [3], Chapters 4 and 5) which need no special implementation. In fact, if the translation macros available in the MNETS library are suitable, the modeler has very little to do except instantiate the macro and connect the MNETS control signal, as shown later in example *FirstEx*. Thus, the cache directory hardware is replaced by an available translation macro. Since a one-cycle cache was assumed, the cache array is not needed at all, because no data are actually accessed in a timer. As detailed later, if a multicycle cache was assumed, as is necessary in other levels of cache (L2, L3), the array is still very simple, just a down counter to provide a pipeline stall for the number of cycles equal to the array access time. The data-out register is just a one-bit latch and may or may not need a record part, depending on the full model complexity. Thus we see that MNETS provides a graphical one-to-one correspondence between the actual pipeline and its model, but with an enormous simplification. This is of great value in building, testing, debugging, and reusing the model.

State passing is very complex in an actual pipeline, and still complex in typical timer models. In marked contrast, this is one of the most significant advantages of MNETS. The controls are all local in MNETS, rather than global, as in actual pipelines and other models, as indicated in Figure 1. The local control concept makes MNETS modules reusable—just plug and unplug module control signals to add, delete, or change the model. The control structure which allows this is a *set of four fundamental, unique signals common to all pipelines* that work as follows. Referring to Figure 1, we see that when the translation macro has a miss, it really is indicating that it is stalled on the current cycle, cannot complete the last request, and will not accept any new input state. A signal, called Wait Exception on Current (WEC) cycle, is used for this and is propagated back upstream to notify any module which might try to send a new state to it. This is shown in **Figure 2**. In this case, the stall/wait signal, WEC, goes back upstream to the VA register. Now, on the same cycle, the VA register's local logic interprets this input signal to mean that the target for the VA is not available, so obviously it should hold its state if it is a 1 and likewise send out a WEC = 1 signal indicating that it is stalled on this cycle. Thus, its WEC will be propagated back upstream to the ALU register. If the VA register is empty (as indicated by its state being a logical 0), it can in fact accept a new state and thus does not set its WEC to 1; i.e., it does not have to stall, since it is empty and can be filled. The same is true for all such state registers. Thus, the stall signal WEC for the current cycle is evaluated locally and propagates as a 1 (stall) only as far back upstream as an actual stall will occur. As can be seen in Figure 2, the WEC output signal from each state register is connected to a control input terminal labeled DSE, which stands for Down-Stream Exception. This input means that the target to which this state is headed is stalling on this cycle if DSE = 1, and is not stalling on this cycle if DSE = 0. The reason for different names will

[6] VHDL allows signals to be defined as records (similar to those in C or Pascal), so the modeler need not provide a hardware bit-by-bit representation of integers and other such data.

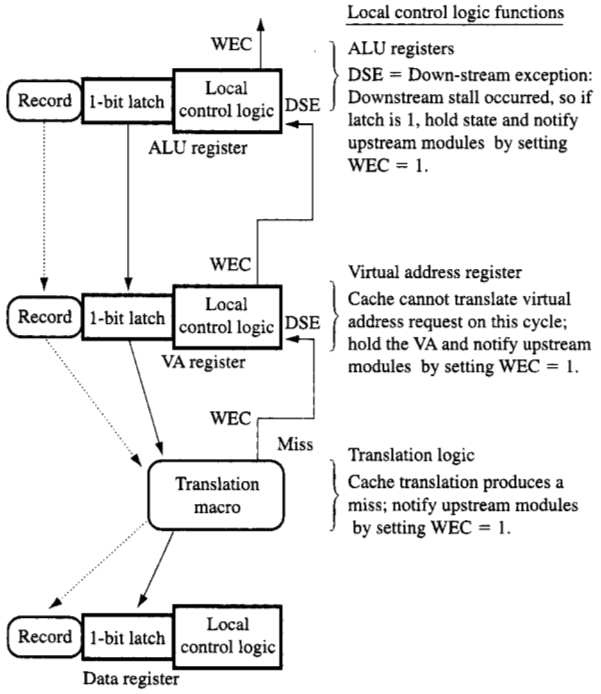


Figure 2

Local control logic functions and signals for a simple pipeline with an assumed cache miss and full stall.

become more clear below, but the fundamental reason is that for some cases, this DSE input does not always have to come from the WEC of the downstream module. It can come from other control signals, or from a combination of several other control signals, usually ANDed or ORed together.

There are two other local control signals required for general modeling of pipelines, which have the same ultimate function, namely to determine locally whether the local state should hold, reset to 0, or accept a new state. Each specifies a different logic condition, and all work together to fully model any pipeline. These two additional control signals are Dependent-Event Exception (DEE) and a Used (U) control signal, as shown in **Figure 3(a)**. The function and timing of each of these signals is detailed later, in Section 7. In essence, DEE prevents a new state from being accepted but lets the current state pass out if it can. DEE provides a means for specifying that a needed resource or condition must be met before a new input can be accepted. The Used signal, U, is needed for cases where two or more sources (states) must pass through a priority selection and only one can be chosen, the other stalling. The Used signal specifies which source is chosen so that it can pass state if other conditions permit, while the sources not Used will hold state. This is shown later in an example. The Used signal can be thought of as a type of enable signal to indicate to an upstream module that it has now, on the current cycle, been processed and can be dropped if allowed by any attached DSE signal. Thus, the fundamental unit for holding and passing state in a pipeline is the Basic Pipeline Module (BPM) shown in Figure 3(a). It consists of a one-bit latch with a bit input X, a bit output Y, and a local control logic section with bit-input signals DEE = dependent-event exception, DSE = down-stream exception, and U = Used, and a single bit-output control signal WEC = wait exception on current cycle. As was discussed above, in many paths of a pipeline it is necessary to pass along certain information (usually obtained from the input trace or translation macro) such as the virtual address, the operation to be performed (e.g., load, store, other), and sometimes additional details. Such information is carried in a separate record field attached to a one-bit latch, as indicated in **Figure 3(b)**. Its input is the record

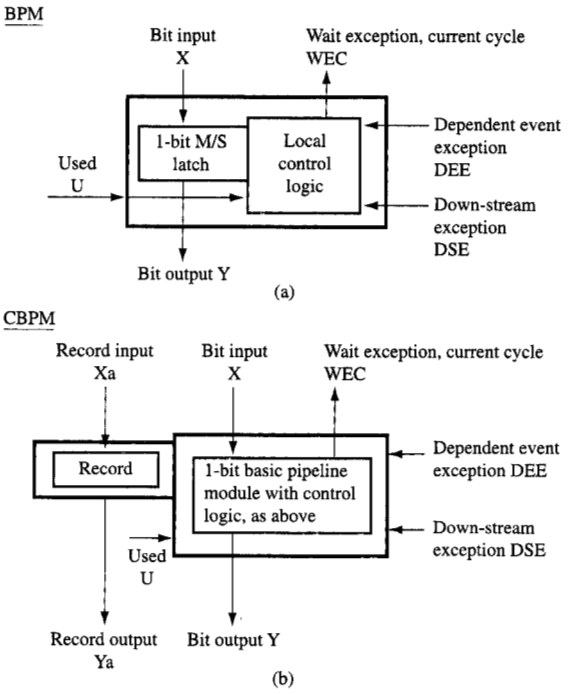


Figure 3

Schematic of the elementary structure and control signals of the most basic MNETS pipelining modules: (a) One-bit basic pipeline module (BPM) with four modular, local control signals; (b) one record plus one-bit compound basic pipeline module (CBPM) with the same four modular, local control signals.

signal Xa, and output is the same record signal, now called Ya. The record can be defined to have almost any type and number of fields (VHDL records), but certain types have been defined and are inherent in the MNETS macros in the library provided. In any case, the record passes along with the one-bit state automatically; no user intervention is needed.

There are a few (very few) other basic modules needed to model a pipeline, and there are a number of variants of these basic modules just for simplicity of modeling (e.g., not all modules need all the local control signals). But in all cases, and fundamental to this design methodology, the same local control signals are used for all modules and all macros at all levels of the design hierarchy. This is the feature which makes MNETS modular and easy to understand, defines reusable modules, and makes models simple to design and change. In addition it makes each module separately testable with a common test generator, and pluggable into the model. Although no formal derivation or confirmation is available, this set of control signals appears to be fundamental and unique. Thus far, they have been able to model all control flow structures, which is a necessity in achieving any modular, object-oriented modeling construct. An obvious result of this is that other users, if attempting to use this methodology, should *not* introduce additional control signals, since this destroys its simplicity and modularity.

5. Clocking, timing, and passing of state

The MNETS pipeline model works very much like an actual pipeline. A two-phase clock paces the pipeline state registers (an understanding of this is important as well as extremely educational in understanding pipelines in general). In an actual pipeline such as that in Figure 1, each register has a two-phase clock input. On the Begin clock TB (which clock is Begin is purely arbitrary), the clock signal rises and causes all state registers to broadcast their internal state into the combinatorial logic network. After a time equal to the maximum delay of the logic, all input signals to all state registers will have reached their correct, final value, but the input to each register (latch) is blocked during this clock phase. At this time, the Begin clock TB goes to 0 and the second clock, TE, rises. This TE clock enables the input stage of all the state registers. Whether or not a new input is accepted usually depends on a number of additional external enable signals provided by the global control logic. In MNETS a similar but simplified modular structure and timing sequence is used, as follows. All MNETS modules which latch state have implicit two-phase clock inputs. These signals do not show up as terminals on the various modules such as those in Figure 3, because they can be declared in a special way as global signals in VHDL (see [3], Section 1.11.4). As a result, any module can use these signals internally and need not declare them. This implicit use of clocks and the resulting sequence of events is illustrated in **Figure 4**.

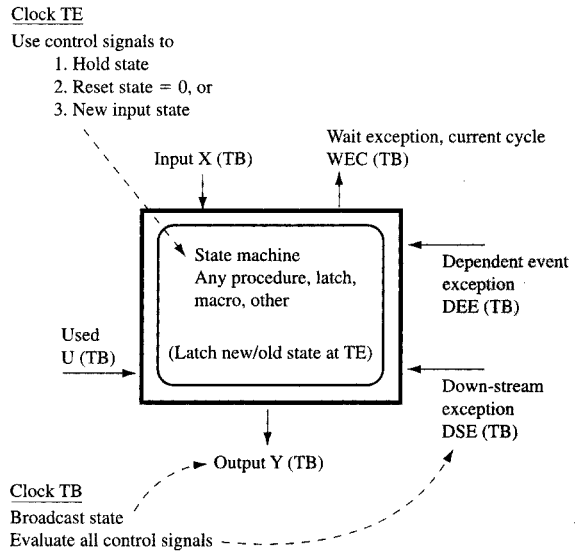
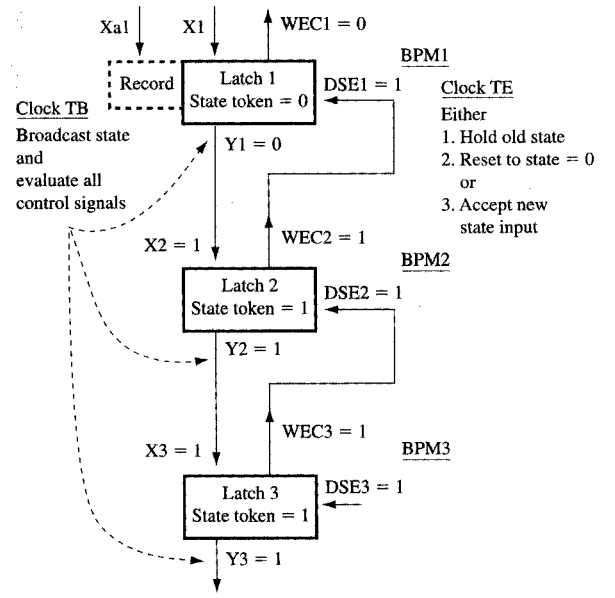


Figure 4

State machine of any kind, with four modular, local control signals for passing of state.

On the Begin clock TB goes to 1, and all modules which hold state place this state on their Y outputs, which broadcast their state into the MNETS network in a manner similar to that of an actual pipeline. These state signals flow to other modules and start generating the various control signals at time TB. Each module with control input signals DEE, DSE, and/or U looks at these inputs and its own state, and as a consequence sets its own WEC to 1 or 0 at this time (TB). *All modules complete this signal evaluation at TB.* Actually, all modules are basically "zero time" circuits and evaluate on the rise time, since no actual circuit delay is simulated. Thus, the processor clock or cycle time is totally arbitrary, picked to match the pipeline being modeled. After the first half of the cycle, the End clock time TE becomes 1, and TB goes to 0 at the same instant. The only function performed at time TE is for all modules with state to decide, on the basis of the input control signals, whether they will hold previous state, reset to 0 state, or enable a new state from the signals X/Xa. Subsequently, the clock time moves to TB again, and all modules broadcast their state on their output terminals, Y. Those modules which held state will broadcast the same state as previously. Those with new state, whether it be 0 or new X/Xa values, will likewise broadcast their states. The re-evaluation of all control signals takes place, and on the next TE, the state passing,



Pipeline section illustrating use and function of control signals DSE (down-stream exception) and WEC (wait exception, current cycle).

holding, or resetting occurs as before. Thus, the pipeline progresses in a manner completely analogous to a real pipeline. However, the control logic is all local, the modules are all modular and pluggable, and the model is graphical.

6. Control signal functions, timing, and uses

In the most general case, an MNETS module which contains a state latch can have all four control signals, namely inputs DEE, DSE, and U, and output WEC. First, an example for DSE alone is discussed (shown in **Figure 5**), then an example for DEE alone (shown in **Figure 6**) and an example for U (shown in **Figure 7**). After these, the full logic for a module with all control signals is presented. It should be kept clearly in mind that every module which holds a state is a source on its output side, and a target on its input side.

- *Control signal down-stream exception, DSE*

The primary function of this signal, as indicated previously, is to inform an upstream source module that the downstream target will not accept a new state token on this cycle. It is up to the source module to decide what to do with this information. In general, the source looks at its DSE, DEE, and U signals as well as its current state token in order to decide whether to hold, reset, or accept a new state. The simplest case occurs when the module has no DEE nor U input signal or one or both are present but do not affect the state evaluation. As an example, consider the path in which three basic pipeline modules (BPMs) are connected as shown in Figure 5, where it is assumed that initially Latch 1 = 0 while Latch 2 and Latch 3 are both 1 tokens with all U = 1 and all DEE = 0 if present (equivalent to not being present, as shown). On the first Begin clock, time TB1, all BPMs broadcast their tokens, 1 or 0, to their respective Y outputs. Assume that on this TB1 the control signal DSE3 on BPM3 is a 1, indicating that the target of BPM3 (not shown) will not accept a token on this cycle. The control logic in BPM3 looks at the DSE3 = 1, and since its state token in Latch 3 is 1, it recognizes that it will not be able to move downstream on the upcoming TE1, so it sets its WEC3 = 1. This makes DSE2 = 1. The control logic in BPM2 sees its DSE2 = 1 and its state token in Latch 2 also as 1, so it obviously will not be able to move on the upcoming TE1, and it also sets its WEC = 1. As a result, DSE1 = 1, but its control logic sees that its state token in Latch 1 = 0, meaning that it is empty, i.e., no token to move. Therefore, BPM1 can accept a new token on the upcoming TE and thus sets its WEC1 = 0. This tells any module above BPM1 that its state transfer, if any, will be successful.

Subsequently, at time TE1, Latch 3 and Latch 2 both hold the current state, while Latch 1 accepts a new input X, whatever X may be. This X could be a 1 or a 0; it does not matter, since BPM1 must accept whatever token is sent from its upstream source (not shown) and this could be 1 or 0. Note that if these BPMs were compound BPMs, having a record field input Xa, output Ya, and internal latches all having a record field, these records would follow the respective tokens. Record 2 in Latch 2 and Record 3 in Latch 3 would remain the same on TE1, while Record 1 in Latch 1, which was initially null (actually "don't care"), would accept the new Xa1 record (shown dotted) at TE1.

- *Control signal dependent-event exception, DEE*

The primary function of this control signal is to prevent any new state from entering the latch of a BPM (or any module with a state latch) for any reason. The typical reasons for a DEE input to be 1 are that some resource or a particular logical state or condition needed at this stage is not available. Thus, the entering of this state must be prevented until this needed condition is available. This logical structure can be achieved with the MNETS model of Figure 6. The dependent resource can be another MNETS module, a macro, or a separate logical construct. At clock time TB, if the dependent resource is busy, it will set its WECdr = 1. This sets DEE1 = 1, so the control

logic in BPM1 sets its WEC1 = 1. At clock time TE, this will stall any upstream source which has BPM1/Latch 1 as its target. If the Latch 1 state token was initially 1 at TB of the cycle in which DEE became 1, its state token will drop to the downstream target and Latch 1 will reset to 0 (if other control signals do not prevent this). Thus, while a new state cannot enter the module because of DEE, the current state of the latch can move out of this module independent of DEE (but dependent on DSE and U, if either is present).

- *Control signal Used, U*

An important, but not the only, function of the Used (U) control signal is in conjunction with a priority selection module, in which there are multiple sources headed for the same target and only one can/must be selected. This is illustrated in Figure 7, which shows two sources, BPM1 and BPM2, both trying to reach the same target, BPM3. The DSE inputs may or may not be needed on BPM1 and BPM2, depending on the pipeline. Regardless, the U signals play the same role. For example, suppose initially that Latch 1 and Latch 2 both have state tokens = 1. At clock time TB, both Y1 and Y2 = 1, making both XA and XB = 1. The priority logic of the module is assumed to select XA before XB, so XA is selected and transferred immediately to the Yp output (there is no internal latch, so no state is held in any priority module in the MNETS library). At the same time, at TB, U signal UA is set to 1 and UB is set to 0 by the priority module, making U1 = 1 and U2 = 0. If there are no DSE inputs on Latch 1 or Latch 2 (i.e., assume that DSE is not needed for the moment), then since U1 = 1 (it has been Used), Latch 1 logic sets its WEC1 = 0. At the same time, Latch 2 has U2 = 0 (it has not been Used), so it sets its WEC2 = 1. At clock time TE, Latch 1 drops its token to the target Latch 3 and accepts a new state input from its X1, while Latch 2 holds the current state token. If Latch 1 and Latch 2 both have DSE inputs from WEC3 of the target, the above sequence still occurs if WEC3 = 0 at clock time TB. If it should happen that WEC3 = 1 at time TB, the BPM2 behavior will not change (it holds state) but Latch 1 is different. Even though signal U1 will still be set to 1, the DSE1 = 1 input to BPM1 will cause the control logic to set WEC1 = 1 and hold the current Latch 1 state token at clock time TE. The reason, obviously, is that the target, BPM3, is not going to move at TE, since its WEC3 = 1. Thus the source, BPM1/Latch 1 for the given set of states, even though it was Used (U1 = 1), sees that its token has a DSE = 1, so it cannot and must not move.

Note that the DSE and Used signals work together as a sort of op code to *move one* or *hold both* sources. Also note that BPM2 can move only if the BPM1 state token is 0, owing to the priority logic of the priority module. While the priority modules in the MNETS library have primarily a straight ranking selection, A > B > C, etc., for input A, B, C, etc., the logic is written in extremely simple VHDL code and can easily be changed to almost any logic.

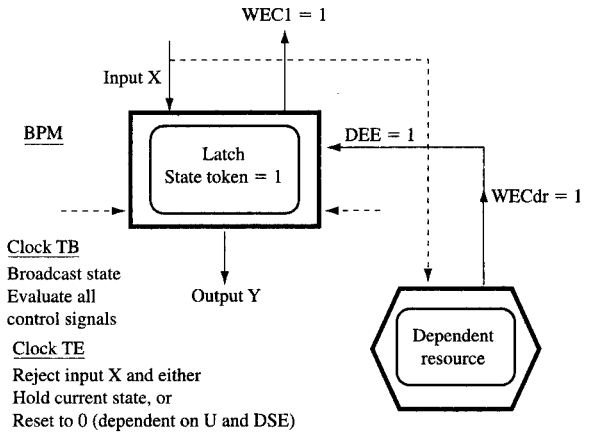


Figure 6

Control signal dependent event exception, DEE.

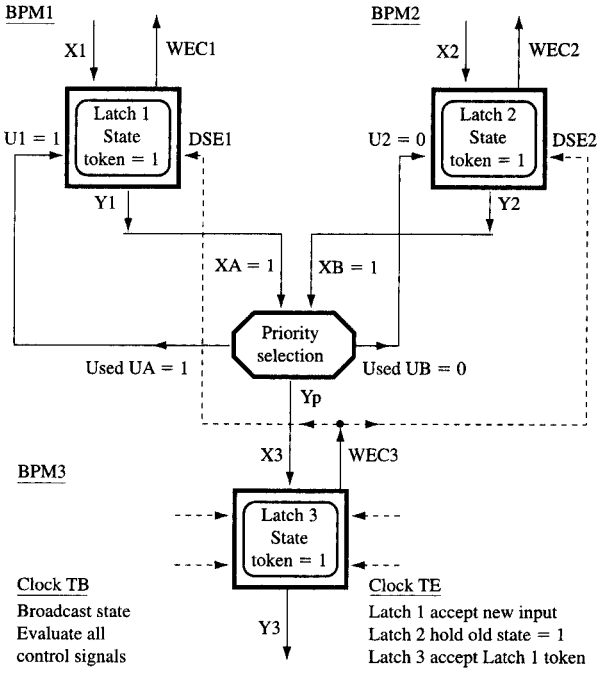


Figure 7

Control signal Used, U (and DSE) configured with priority selection module.

7. General case of all control signals present on module

In the most general case, a module can have DSE, DEE, and U inputs. The control logic in such modules looks at all of these signals at time TB in order to decide what value to assign WEC at this TB and whether the latch should hold, reset, or accept a new state at TE. The logic is as follows (see specific modules in the MNETS library for the actual VHDL code).

```
If
  state token = 1 at TB
Then
  If
    DSE = 1 or U = 0
  Then
    Set WEC = 1 at TB              (Current state cannot
    and                            move downstream,
    hold state at TE               so hold; DEE is a
                                   "don't care"
                                   in this case.)
Else
  If
    DSE = 0 and U = 1 and DEE = 0
  Then
    Set WEC = 0 at TB              (Current state can
    and                            move downstream
    accept new state input at TE   and there is no
                                   dependent event
                                   exception stall.)
Else
  If
    DSE = 0 and U = 1 and DEE = 1
  Then
    Set WEC = 1 at TB              (Current state can
    and                            move downstream
    reset state to 0 at TE         but new state cannot
                                   be accepted because
                                   of DEE stall.)
End If
If
  state token = 0 at TB
Then
  If
    DEE = 0
  Then
    Set WEC = 0 at TB              (Latch is empty and
    and                            there is no DEE stall,
    accept new state at TE         so accept new state;
                                   DSE is "don't care"
                                   in this case and
                                   U ≠ 1 if state = 0.)
Else
  If
    DEE = 1
  Then
    Set WEC = 1 at TB              (Latch is empty but
    and                            new state cannot be
    hold state at TE               accepted because of
                                   DEE stall.)
End If
```

8. Multicycle stall on a basic pipeline module using down counter

It is possible for certain stages of the pipeline to require multiple cycles before the state token can even attempt to pass from source to target. For instance, if the ALU in Figure 1 is doing a multiply or divide or certain shifts, or if a stage such as VA needs memory access to an L2 cache or main memory, or cache reloads are required, then this stage of the pipeline must stall for the appropriate number of cycles. These stalls are almost always provided by a simple, fundamental module called a down counter. The essential idea is that during the initial loading of this stage of the pipeline, an associated down counter is also loaded with an integer count value which will give the desired number of cycles of stall. On each subsequent cycle, the counter generates an output signal WEC = 1 and decrements its internal count value by 1. This down counter continues decrementing its internal count value and generating WEC = 1 on each cycle until an appropriate count is reached (0 or 1, discussed in [3]), when it will generate WEC = 0, which remains 0 until a new count value is entered. This WEC is used as DSE, DEE, or whatever inputs are necessary to stall the corresponding states of the pipeline. The down counter has an external enable signal, En, to tell it to load the integer count value, Nin. However, this enable signal is internally shut off while any previous entry is still counting down (i.e., cannot accept a new Nin value as long as it is generating a WEC = 1 output signal). Only at the end of counting down (i.e., when WEC is 0) is the enable input able to insert a new count value, Nin. The details of the down counter operation are discussed below.

- *Fixed number of cycles of stall*

An example of the typical use of a down counter is shown in **Figure 8** and works as follows. Assume that the token which enters source BPM1 always requires three cycles before dropping to target BPM2. A down counter is inserted between the two stages and provides this three-cycle fixed pipeline stall as follows. At clock time TB, if the enable input En = 1 and the current internal count value is 0 (or 1 but going to 0), the Nin input integer is accepted as the new count value at subsequent clock time TE. Also at TB, if the internal count value is greater than 1, the counter logic sets its WECdc = 1 and rejects any new input count integer Nin, regardless of the En signal value. In this particular example, this WECdc is used as a direct dependent-event-exception (DEE2) input on BPM2, and the WEC2 of BPM2 is the DSE1 of BPM1. The result of this connection is that BPM2 will reject any new input token for three cycles, and BPM1 will hold the 1 token for three cycles. At the end of the third cycle, token 1 (and any associated record, if a compound BPM is used) will

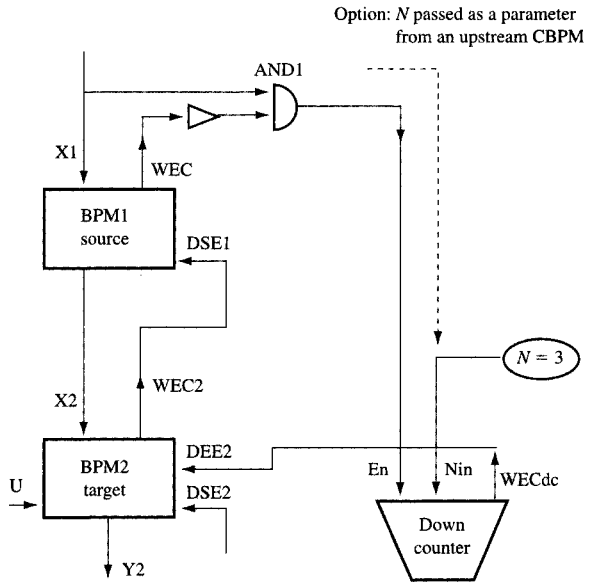


Figure 8

Multicycle pipeline stall using a down counter and target with DEE input.

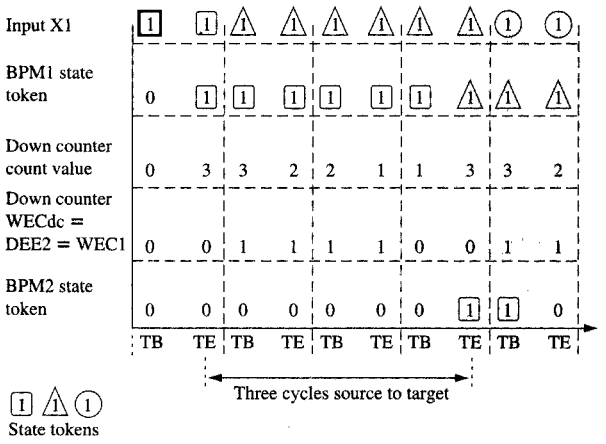


Figure 9

Token timing diagram showing a multicycle stall between two BPMs, provided by a down counter generating WEC = 1 while COUNT > 1 (at TB).

drop to BPM2, and BPM1 will accept any new input state. A state-token passing/timing diagram is shown in **Figure 9** for this case. The overall logic of the down counter is as follows:

At clock time TB If internal COUNT > 1, set WEC = 1;
Else If COUNT = 0 or 1, set WEC = 0.

At clock time TE If COUNT > 1, set COUNT = COUNT − 1;
Else If COUNT = 0 or 1,
then If En = 1, set COUNT = *N*,
Else (if En = 0) set COUNT = 0.

• *Variation of pipeline connection for multicycle stall using down counter*

In Figure 8, it was assumed that the target is another pipeline module, BPM2, which has a DEE input. This DEE control signal prevents the transfer of the 1 token from source to target until the WECdc of the down counter goes to 0. This is the simplest method of control when the target has such a DEE input. However, in some cases, the output of such a module might be the output terminal of a macro and cannot know *a priori* just what the target might be (for instance, the output terminal of the memory reload register, the fetch register of the L2Simple macro in [3], Section 5.6). The target for this

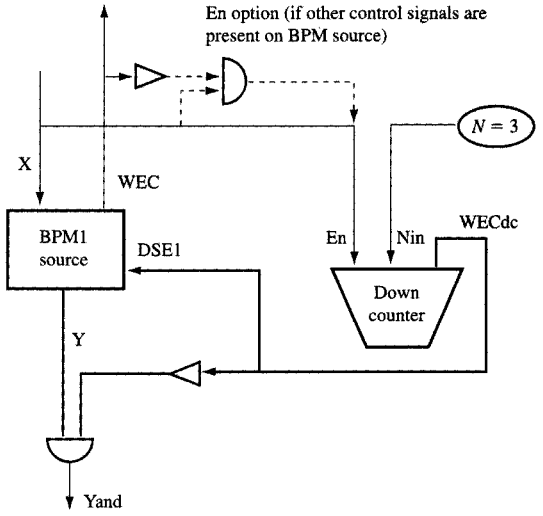


Figure 10

Alternative configuration for multicycle stall to an unknown target, or target without DEE input, using a down counter.

source, which will later be connected to this macro output, can be one of any number of destinations. To be most general, the output of the source is locally prevented from propagating any further by the use of an AND gate on its Y output, as shown in **Figure 10**. The signal WECdc is

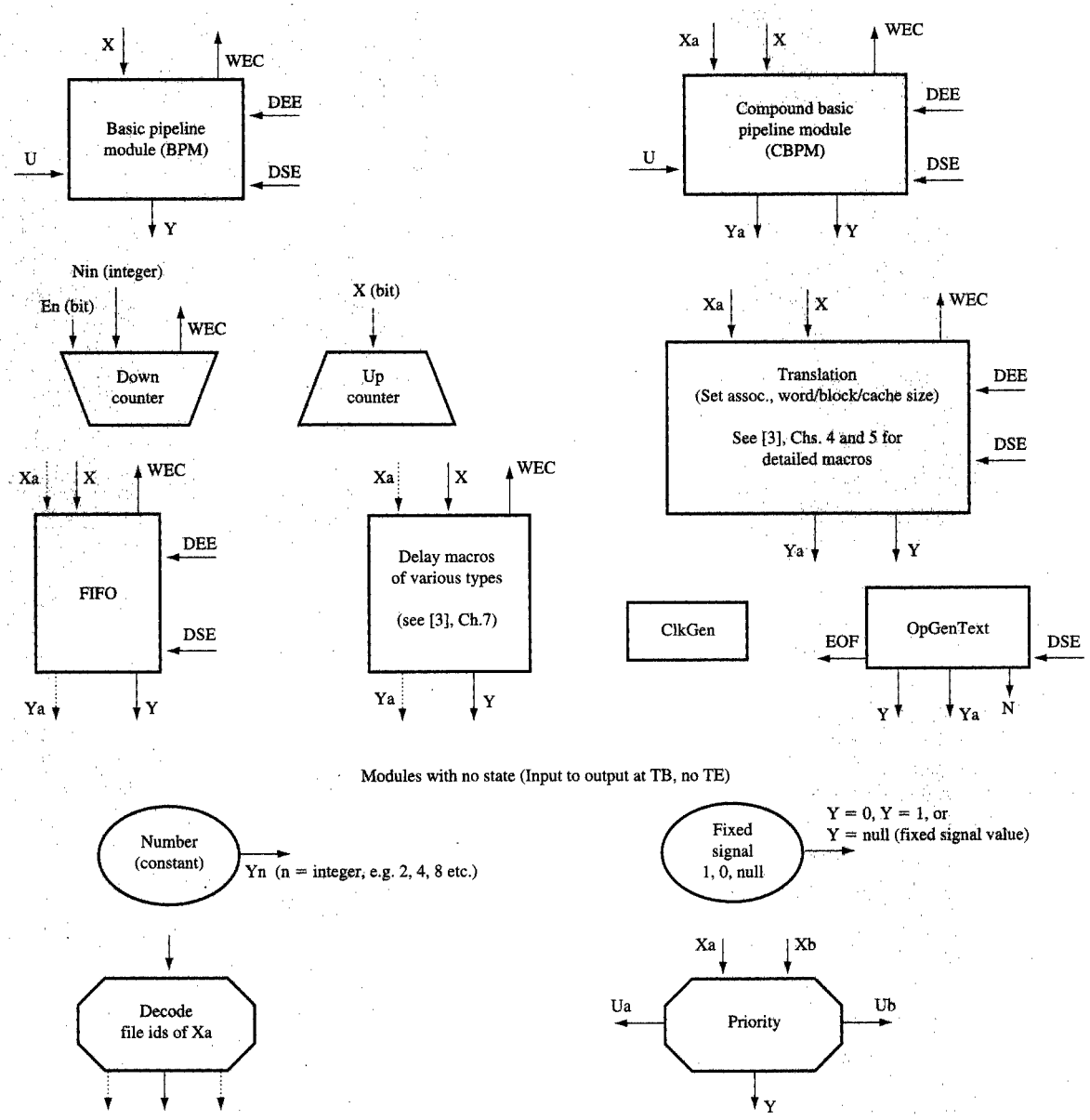


Figure 11

General classes of MNETS modules and macros.

used as a direct input to DSE1 of the source, which stalls the token in BPM1 until the down counter WECdc goes to 0. At the same time, the inverse of signal WECdc is ANDed with the output Y of BPM1, which also makes signal Yand = 0 until the down counter WECdc goes to 0. Thus, this Yand signal will go to 1 only after the counter has counted down the appropriate number of cycles. After this countdown, signal Yand will be a 1 for only one cycle unless some other signal holds the token in the source. In any case, this Yand will have the correct timing and can be used directly as an input to any module with or without any DEE input.

- *Integer numbers for down counter*

A fixed value of Nin for a down counter is typically needed and easily obtained from an MNETS library module called NUMBERn, where n is the desired integer value. For instance, modules NUMBER3 and NUMBER5

give an integer output signal of value 3 and 5, respectively. These modules can be instantiated and "wired" to the down counter Nin pin in Wizard just like any other module.

- *Variable number of cycles of stall*

While most stages of a pipeline will have a fixed value for the number of cycles of stall, there are cases for which the down counter Nin integer is not fixed and/or not known in advance. Depending on the situation, there are several simple methods for accommodating these features. If the counter delay is variable, but known or calculated when the original trace is produced, the Nin value can be passed as a field in the record part of the Xa input. When the down counter is enabled, this value is easily extracted and passed to the Nin input of the down counter, as shown by the dotted line in Figure 8.

9. Basic modules in MNETS

From the above discussion, it can already be inferred that three important modules for MNETS modeling are 1) the *basic pipeline module, BPM*, and its companion, the *compound basic pipeline module, CBPM*; 2) the *down counter module*; and 3) the *priority module*. These three, shown with the most general set of control signals in **Figure 11**, are the most useful, basic modeling components and appear repeatedly in any model. In addition to these basic modules, there are others which, while not used as often, are nevertheless required to build a pipeline model, particularly memory hierarchy models. The complete MNETS library is contained in two AFS directories, namely

- afs/watson/projects/M/MNETS/AVHDL/Admod8.
- afs/watson/projects/M/MNETS/AVHDL/Admacro8.

These fall into a few general classes, such as those illustrated in Figure 11. Some of these components are the following:

- *First-in-first-out (FIFO) stack*
  This is a typical type of FIFO register stack, but with MNETS control signals for passing or holding state and a selectable stack size.
- *Translation macros*
  Translation macros are used to translate a given virtual (linear, effective) address into the physical cache address, in a manner which mimics the actual hardware directories. The cache size, line size, set associativity, and logical word size (size of accessed data) are input parameters, and several different types are available, e.g., inclusive and noninclusive).
- *Clock generation (CLKGEN)*
  All pipeline modules require two clocks, TB and TE, as described previously. Once instantiated in a model, the CLKGEN module generates these clocks automatically. In addition, the clock signals TB and TE are global; i.e., all modules and macros in the schematic can "see" these global clocks without the user having to make any direct connections (see Figure 14, shown later).
- *Trace reader and formatter (OpGenText)*
  This module reads a text input trace from an AIX text file and puts each instruction on its Y/Ya output. This output serves as the driving input to the remainder of the model.
- *Metering counter (up counter)*
  This module is used as a type of metering device. It starts from an internal count of 0 value and adds 1 at the end (time TE) of each cycle when its input enable signal, En, is 1 at the begin time, TB, of that cycle. This is used as a means to count the number of misses, misses with castout, etc. in the translation modules and can be used as a metering device in any model, as needed.

10. Object-oriented program modeling

The most important element in MNETS modeling, the concept of local control for the passing of state, is achieved by the use of a set of four well-defined, consistent, never-changing control signals at all levels. Every module, component, or macro which contains a latch or holds and passes state in any manner via the clocks has either all four or some necessary subset of these four control signals. There are no exceptions, and any user who constructs a custom macro *must* adhere to this design rule. By so doing, one achieves a design that has a control structure at all nested levels which is identical to that at the lowest levels, as illustrated in **Figure 12**. This makes the models easy to build, edit, and reuse, and above all else, easy to understand at some time long after implementation. The use of a few well-defined control signals removes all of the complex interdependencies one normally encounters when each modeler has to choose his/her own control signal structure and definitions.

By adhering to this consistent structure, one obtains a pure, object-oriented programming methodology for pipelines. The objects all have consistent and well-defined parameters which are passed between them, namely the consistent set of control signals and input/output state signals. Modularity is just another word for objects with consistent parameters passed between *all* levels of object definition, including nested objects within objects.

11. Embedding other modeling techniques within the MNETS object-oriented framework

The essence of MNETS modularity stems from direct incorporation of the fundamental state-transition flow-control conditions required in any pipeline. The two

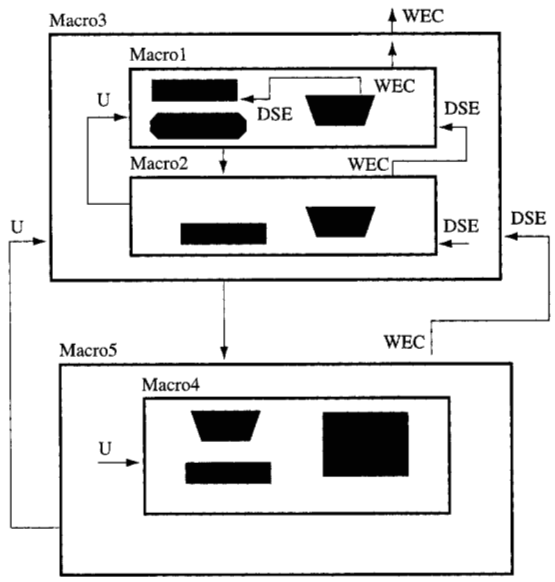


Figure 12

Design of a memory hierarchy, or any pipeline in MNETS, showing consistent, object-oriented modularity at all levels.

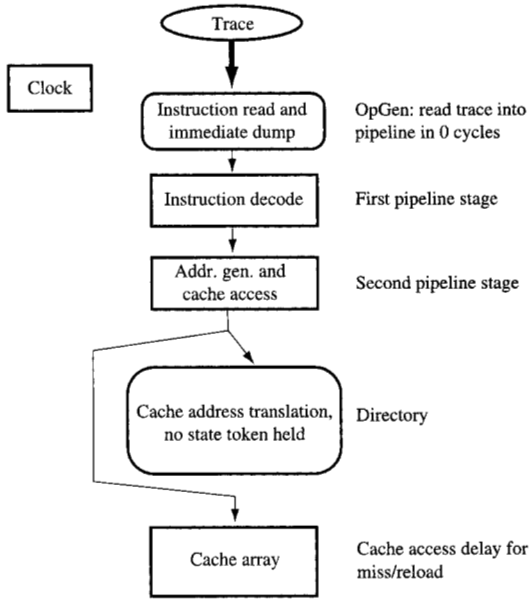


Figure 13

Schematic of pipeline stages for first example, FirstEx.

fundamental control mechanisms for state-transition determination, required in any methodology, are easily seen from the previous discussion:

1. Are all necessary resources available in order to start the state transition? (DEE = dependent-event exception)
2. Can (will) the current status allow completion of state transition?
   a. Has this state been acted upon? (U = Used)
   b. If Used, can it move to the target? (DSE = downstream exception)

These concepts and the resulting four control signals *are not, and need not be, limited* to the particular MNETS modules defined here. In fact, it is easily conceivable that other models could use these signals to make their code modular, reusable, easier to construct and use, and have all the advantages of MNETS. If properly constructed, such models could be easily and conveniently integrated into and used with the MNETS modules and macros.

12. Model example

A simple, complete modeling example presented in detail in [3], Chapter 1, is a model named *FirstEx*. It has the logical structure illustrated in **Figure 13** and the actual, final MNETS/Wizard schematic in **Figure 14**. The trace of instructions is read by the OpGenText component from a file. The instructions are placed on the output of the OpGenText component and fed into a pipeline consisting of a compound basic pipeline module CBPMDse with only a DSE control input (no U or DEE) followed by another identical CBPMDse. The first pipeline stage (first to second) can be thought of as the instruction decode stage, requiring one processor cycle; the second pipeline stage can be thought of as address generation and cache access on the same cycle. Alternatively, one can think of these two stages as address generation between first and second, with only cache access on the second stage. The actual representation would depend on the actual pipeline. In either case, the latter stage accesses a simple, one-cycle cache which is assumed to operate as follows. A full translation using a library macro *TransL1incl* (described in [3], Chapter 4) is performed. There are no castouts, no unlocks, and only simple translation hits or misses. If a hit occurs, nothing extra happens. This is equivalent to the request being found in one cycle and successfully used by the processor. If a miss occurs, the request drops through to the output YMiss, so CBPMDse2 can accept a new state, but it is subsequently stalled for NR cycles. This is achieved by loading the down counter if the translation output YMiss = 1. This simulates a fixed reload time, wherein the upstream pipeline fully stalls on a miss. The

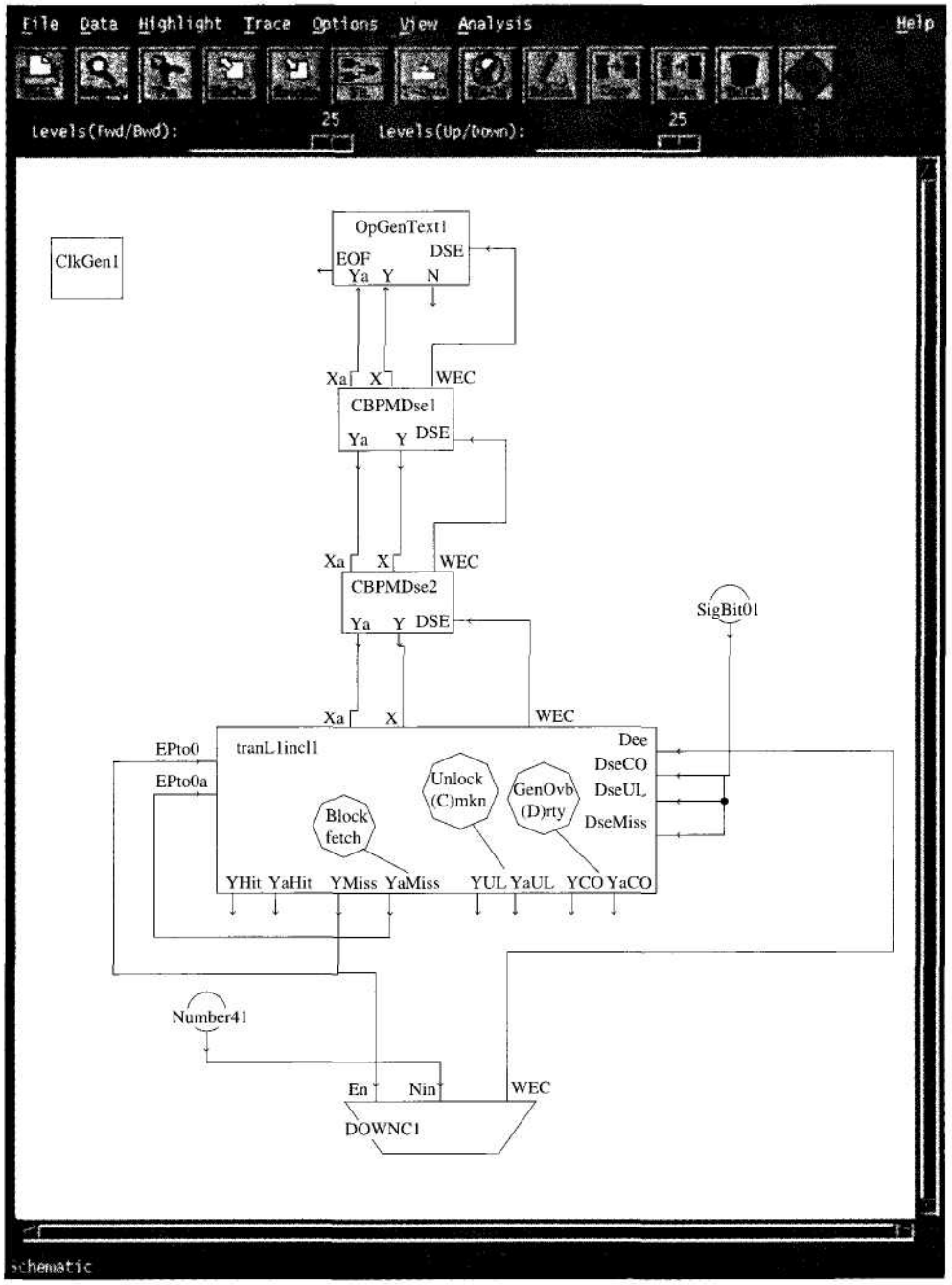


Figure 14

Wizard editor schematic of our first model, FirstEx, in MNETS, showing all component instances and wiring connections.

DEE = 1 input to translation sets its WECtr = 1 so that the WECdc stalls the translation from generating any more outputs, and stalls CBPMDse2 as well. After NR cycles, WECdc goes to 0 and the pipeline restarts.

Assuming the user has the EDA Wizard graphical editor available, and access to the Andrew file system, the construction of such a model requires only minutes. The needed components are instantiated in the usual manner

and graphically wired together. This Wizard model is nothing more than a graphical drawing consisting of a list of box (component) names, pins, signals with type definitions, interconnections, and some other needed information. Fortunately Wizard knows how to create compilable VHDL programs from such graphical drawings and requires only clicking on a VHDL button.

Thus, the model and VHDL code generation require nothing more than Wizard and the MNETS design library. To simulate the VHDL code, any simulator can be used as long as it is capable of handling record data types. The compilation, simulation, and running of this or any model are totally independent of the MNETS/Wizard design system described above. Details of running this example on the MTI system are given in [3].

It can be seen from the above that when needed components are available, it is quite simple to graphically build a pipeline model. Since much of the drudgery is automated, the user can concentrate on the important issues. Experience, both personal and general, has shown that the modeler will spend as much or more time in trying to decide or find out exactly what is to be modeled, than in actually building the model. This is because there are so many options and details in the actual pipeline that it is impossible to consider and decide them all beforehand. MNETS allows the user to start with a simpler model and add more complexity as the design becomes clearer. Also, it forces the designer to face early on those important issues which can seriously affect performance. All timing points and stalls must be included, similar to those of the actual pipeline, to obtain an accurate performance predictor.

13. Model testing

MNETS has a distinct advantage in testing over other methodologies. Each module or macro can be tested independently, similar to the way in which electronic components can be tested. We attach a signal generator, in this case an OpGen module, which supplies input signals from a special-purpose trace if necessary. Then we observe the output signals and can run various test cases. Once fully tested, the macro is entered into the library. This has been done for most modules and macros. Assuming that all underlying macros have been fully tested, a new model need only test for the new states resulting from the interconnection of these macros. If necessary, a new model can be built in steps. Even more useful, if a simulator such as the MTI VSIM is used, the module/macro signals at any point in the model can be essentially traced on a software "oscilloscope" just as for a circuit or logic design. These can be saved and studied later and manipulated in various ways, greatly facilitating the testing.

14. Interactive design tool

A major issue in all timer models is that the number of instructions which can be simulated in any reasonable time is a very small fraction of any actual running time of the system being modeled. The fundamental problem stems from the fact that any model running on a current computer, trying to simulate a future, faster computer, will be orders of magnitude slower in processing instructions. If every design change requires rerunning of very long traces, few options will be explored. The use of reduced traces to allow faster evaluation without compromising accuracy is a step in the right direction, but we still need a better system. Running long traces against some particular design merely gives a final answer of cycles per instruction for a given trace. Although some other internal statistics can be accumulated, this is a tedious design process. A better method is desired for design optimization, and a new mode of thinking is required.

The author's vision of the future design system is an interactive one in which the user looks dynamically at the MNETS graphical model on the screen, watching tokens pass from module to module. The interactive system will be capable of color-coding various tokens along paths at the choice of the user. In addition, the system will have a built-in capability to increase the color intensity of tokens as they stall at any one module, intensity increasing with each successive stall to indicate a hot spot. The user then applies certain selected and/or reduced traces and looks for hot spots in the pipeline flow. This would indicate points of significant stall and could be seen quickly for certain input conditions. The user can attempt to remove the stalls by changing the pipeline, rerunning the trace, and looking for the hot spots again. Various sets of critical trace conditions could be maintained and used for such design optimization. Once a supposedly "more optimum" pipeline is achieved, a full trace can be run for total cycles per instruction and compared with the previous designs. In this way, a more optimum design can be achieved much more rapidly, to give the designer significant insights into the pipeline operation and bottlenecks. A graphical design tool such as MNETS is readily adaptable for such an interactive, optimized design system.

Conclusions

MNETS methodology is a new approach to modeling. While the initial step is always difficult, the advantages should be clear enough to justify a new approach. Past and current techniques used to model pipelines are becoming more and more difficult without some kind of modular approach.

A fundamental point which should be kept clearly in mind is that all timer models reduce ultimately to nothing more than a method for representing state, plus a set of nested if . . . then . . . else (or essentially similar) control

statements for passing state. The different methodologies for constructing models only determine the manner in which the state variables and control statements will be selected, how they are structured, and how they are entered by the modeler. When there are no established rules or fundamental constructs, the modeler is free to pick and use state variables and control statements in any fashion; this is what makes programs nonmodular and nonreusable. In essence, MNETS is merely a way to impose a well-structured discipline on the choice of state variables as well as the structure and use of if . . . then . . . else control statements.

MNETS not only provides such a modular approach with a graphical user interface, it also contains a framework which can evolve to a dynamic, interactive design optimizer as well as incorporate other modeling tools. By incorporating the same basic clocking and local control signal structure, almost any program procedure can interface directly with MNETS models.

Appendix: Hardware implementation of MNETS

The various basic modules in MNETS look like electronic macros. In fact, modules such as the basic pipeline module, down counter, priority, as well as various decoders, FIFO, etc. could be implemented directly in electronic circuits and used to build pipelines. It has been suggested by Eric Kronstadt, Wilm Donath, and others that such modules could be mapped onto an EVE [4] or EVE-like hardware simulator to achieve orders-of-magnitude increases in processing speed. This issue has not been pursued but is an enticing thought. One problem, however, is how to "simulate" translation units. These are macros containing the MNETS control signals, which can be simulated, but state variables (directory entries) are currently implemented as arrays of records, stored in memory. It would be desirable to not have to do the full binary implementation of such macros, but the advantages and disadvantages have not been assessed. If this is not a limitation, models capable of processing very large instruction traces become possible.

Acknowledgments

Many people have been very helpful in various ways throughout this endeavor; the author is especially indebted to the following: Kenneth Sheppard and Kelvin Lewis for help with the Cadence/MTI system; Mark Williams and Arthur Weiner for extraordinary assistance and extensions to Wizard; Rene Miranda and Fateh Tipu for assistance with VHDL; Deborra Zukowski for considerable assistance in Pascal coding during some very early stages prior to the adoption of VHDL; David LaPotin for his timely suggestion to switch from Pascal to VHDL as the underlying programming language; Ravi Nair for suggesting the comparison to Petri nets; Winfried Wilcke for pointing out the similarity of MNETS to object-oriented programming, and Wilm Donath for insightful discussions about modeling and improvements in this paper.

*Trademark or registered trademark of International Business Machines Corporation.

References

1. R. E. Matick, "M-NETS: A Modular Modeling Technique for General Use," *Research Report RC-15117*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, November 7, 1989.
2. R. Matick, "Modular Technique for Constructing Control Logic of a Pipelined Processor," *IBM Tech. Disclosure Bull.* **32,** No. 7, 403–425 (1989).
3. R. Matick, "Modular NETS (MNETS) User/Design Manual; Cache/Memory Hierarchy Timer Modeling and Design Using MNETS Methodology," *Research Report RC-20288*, Ver 1.0, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, June 1997.
4. D. Beece, G. Deibert, G. Papp, and F. Villante, "The IBM Engineering and Verification Engine," *Proceedings of the 25th ACM/IEEE Design Automation Conference*, Anaheim, CA, June 12–15, 1988, IEEE Computer Society, Washington, DC, 1988.

Received June 11, 1997; accepted for publication August 17, 1998

Richard E. Matick *IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (matick@watson.ibm.com).* Dr. Matick received his B.S., M.S., and Ph.D. in electrical engineering from Carnegie Mellon University in 1955, 1956, and 1958, respectively. He joined IBM Research in 1958, and worked initially on thin magnetic films, memories, and ferroelectric devices. As Manager of the Magnetic Film Memory Group from 1962 to 1964, he received an IBM Outstanding Invention Award for the invention and development of the thick-film read-only memory. He later accepted a half-year assignment at the IBM Laboratory in Hursley, England, to develop that memory for System/360 applications. From 1965 to 1972, Dr. Matick was a member of the technical staff of the IBM Director of Research, where his assignments included responsibility for Research divisional plans and service as Technical Assistant to the Director. In 1986 Dr. Matick received an IBM Outstanding Innovation Award for his contributions as co-inventor of "video RAM," which quickly became a commodity DRAM chip used in the high-speed, high-resolution display-bit buffers required by

most PCs and many workstations. For his work on high-density CMOS cache memory design, which served as the foundation for the high-speed functional cache system of the IBM RISC System/6000 processor family, he received an IBM Outstanding Technical Achievement Award in 1990. He is currently working on VLSI memory chips, memory hierarchies, and microprocessor design. Dr. Matick is the author of *Transmission Lines for Digital Networks*, McGraw-Hill, 1969 (an IEEE Press Classic Reissue of 1995), and *Computer Storage Systems and Technology*, John Wiley, 1977. He has also written chapters on memory for *Introduction to Computer Architecture*, SRA, 1975 and 1980, and the *Electronics Engineers' Handbook*, second and third editions, McGraw-Hill, 1982 and 1989, the topical section "Cache Memory" in the Van Nostrand Reinhold *Encyclopedia of Computer Science*, third edition, 1993, and numerous papers on magnetic devices and memories, semiconductor memory and logic, and virtual memory. Dr. Matick holds numerous patents and is the author of a number of published invention disclosures; he is a member of Eta Kappa Nu and a Fellow of the IEEE.