A Language for Compositional Specification and Verification of Finite State Hardware Controllers

E.M. Clarke, D.E. Long, K.L. McMillan

January, 1989

CMU-CS-89-110 3

Dept. of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

A Language for Compositional Specification and Verification of Finite State Hardware Controllers

E. M. Clarke, D. E. Long, K. L. McMillan

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

1 Introduction

In several previous papers ([3,4,5]), we have described a programming language called SML (State Machine Language) that provides a concise notation for specifying complicated finite state machines. Our language has many of the standard control structures found in modern high-level imperative programming languages. Programs in SML may be compiled into state transition tables that can be implemented in hardware as PALs, PLAs, or ROMs. The state transition tables can also be used as input to a *temporal logic verifier* that allows various safety and liveness properties of the program, expressed in the temporal logic CTL, to be verified automatically.

In this paper, we present an evolution of the SML language which we call *compositional* SML (or CSML), which attempts to address the issue of modularity in finite-state controller design. Although SML has a procedure mechanism, which allows a program segment to be inserted into the flow of control using call-by-name semantics, there is no construct which corresponds to the hardware designer's notion of a module—an encapsulated sub-system which runs concurrently with other subsystems, communicating with them over a well-defined interface.

The importance of modularity from a design perspective is evident—it allows a large system to be broken down into a hierarchy of smaller modules which can be designed separately. However, there is another aspect of modularity which relates specifically to finite-state controllers. Any designer of finite-state controllers recognizes that finding the right decomposition of the state space into components will greatly reduce the complexity of the design (this is an aspect of the state

[0] This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under Contract Number F33615-87-C-1499, monitored by the:

Avionics Laboratory
Air Force Wright Aeronautical Laboratories
Aeronoautical Systems Division (AFSC)
United States Air Force
Wright-Patterson AFB, Ohio 45433-6543

The National Science Foundation also sponsored this research effort under Contract Number CCR-8722633. The second author is supported by an NSF graduate fellowship.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

assignment problem). The reason is a phenomenon known as the state explosion problem—in a finite state system composed of a number of concurrently executing processes, the number of states can grow geometrically with the number of processes in the system. Since finite-state controllers frequently coordinate a number of concurrent activities, it is important to be able to design them as systems of coordinating finite-state machines instead of as single, monolithic state machines.

The state explosion problem has an important bearing on automatic verification of finite-state systems, since the complexity of verification is related to the number of states in a system. We use a *model checker* ([2,9]) to determine automatically the truth of CTL formulas with respect to a given finite state system. In practice the model checker can verify properties of state machines at a rate of approximately 100 states/second. It has been successfully used to find subtle errors in a fairly large number of examples—including a DMA controller with more than 500 states and 1300 transitions ([3,5,7]). In order to verify much larger systems, however, it is necessary to try to reduce the state space. If a system is composed of coordinating modules, techniques exist to accomplish this. One such technique makes use of a theorem called the *interface rule*. The interface rule states a set of conditions under which a module in a system of coordinating processes may be replaced by a reduced version called an *interface process* while still preserving the truth value of the formulas in the logic.

To illustrate our new approach, we describe the design and verification of the controller for a simple CPU with decoupled access and execute units. We give the implementation of the controller in CSML and a formal specification (in CTL) of one module. We then describe the application of the interface rule to reduce the complexity of automatically verifying the design. In our example, we use the interface rule to reduce the number of states by approximately a factor of 6.

Our paper is organized as follows. Section 2 describes the logic CTL and the interface rule. The SML language is described briefly in section 3, and the new language CSML is covered in section 4. Finally, section 5 describes the CPU example and the results of the model checking procedure.

2 The logic

The logic we use for formal specification is a branching-time temporal logic called CTL [8]. Formulas in CTL are built from atomic propositions (the signals of the system), boolean connectives ($\wedge$, $\vee$, $\rightarrow$ and $\neg$), and temporal operators which are used to specify timing relationships. Each temporal operator consists of a quantifier ($\forall$ or $\exists$) and a modality (F, G, X, or $\mathcal{U}$). The quantifier indicates whether the operator applies to all computation paths, or whether it specifies the existence of a single path. The modalities denote the desired timing relationship along the paths and have the following meanings:

i. $F\varphi$ means that φ is true at some point in the future.

ii. $G\varphi$ means that φ holds in the present and at all points in the future.

iii. $X\varphi$ means that φ is true at the next state.

iv. $\varphi \mathcal{U} \psi$ means that ψ holds at some point in the future, and that until that point, φ is true.

As an example, we consider several CTL formulas and the relationships they express.

i. $\forall G(req \rightarrow \forall F ack)$ specifies that along every path, if the signal *req* occurs, then eventually *ack* occurs also.

ii. $\neg\exists F(\neg go \wedge \exists X(go \wedge \forall X done))$ says that there is no point where *go* occurs and where *go* must immediately be followed by *done*.

iii. $\forall G(send \rightarrow \forall(send \mathcal{U} rcvd))$ states that along every path, if *send* occurs, then *rcvd* must eventually occur and *send* must remain asserted until *rcvd* occurs.

Given a finite state machine, the model checking program [2] can quickly determine whether a CTL formula is true or false. When a desired property does not hold, the model checker will also provide a counterexample if there is one. This counterexample can be of tremendous help in pinpointing the source of the problem. Additionally, the model checker allows the specification of fairness constraints. Fairness constraints are used to restrict the quantifiers in temporal operators to certain paths. This is often necessary when checking properties of systems which have external inputs; for example, in verifying our example CPU, one fairness constraint we specify is that when the CPU executes a memory read, the external memory system must eventually present the desired data.

One problem with the using CTL and the model checker to verify a system with many components is that the state explosion problem may give rise to finite state machines which exceed the capacity of the model checker. In order to deal with this problem, we can often use the *interface rule* to reduce the number of states. The idea is to form simple abstractions of the modules in the system and to use these abstractions when building a state graph for the model checker. Figure 1 illustrates the principle. In this figure, P_1 and P_2 represent the components of the system we wish to reason about. The components are connected by a set of wires S. A_1 and A_2 are interface processes (abstractions) of P_1 and P_2. Intuitively, A_1 represents everything P_2 can observe about P_1 via the wires S (and similarly for A_2). The key point here is that A_1 must be chosen so that it is equivalent ($\equiv$) to P_1 on S in an appropriate sense. For the logic we are using, the ordinary notion of Moore machine equivalence is sufficient; therefore we can apply the standard algorithms for Moore machine minimization to obtain A_1 and A_2. The interface rule states that if:

i. $P_1 \equiv A_1$ on the set S,

ii. φ is a CTL formula whose atomic propositions denote signals of P_2, and

iii. φ is true in $A_1 \parallel P_2$ (the composition of A_1 and P_2),

then φ is true in $P_1 \parallel P_2$. In a loosely coupled system, A_1 will almost always have far fewer states than P_1, and thus $A_1 \parallel P_2$ will be much smaller than $P_1 \parallel P_2$. Note that the interface rule can be extended to handle boolean combinations of CTL formulas in a straightforward manner.

3 The SML programming language

Since the SML language forms the basis of our new language, we give a brief and informal description of it here. A full description is contained in [3]. Although SML was developed for specifying complicated finite state machines, it has many of the standard control structures found in modern high-level imperative programming languages, including a while statement, a conditional, a case

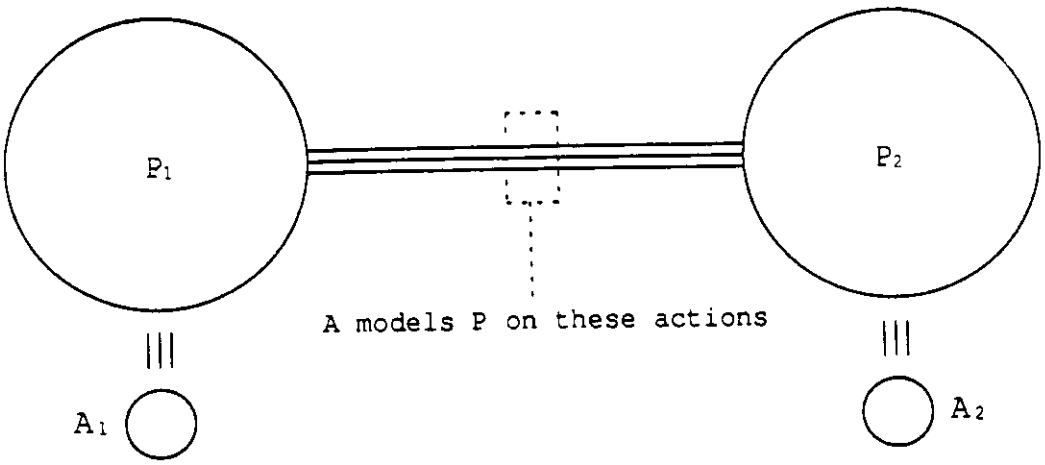

Figure 1: The interface rule

statment, and a parallel execution statement. There is even a simple mechanism for declaring non-recursive procedures. However, the only data types allowed are booleans and fixed width integers. Thus, any program written in SML has only a finite number of states and can be compiled into a state transition table.

A number of other languages have been proposed for specifying complicated finite state machines. Most of these including AMAZE, CUPL, and SLIM [12] represent state machines at a very low level, and some even require an explicit description of the state-transition behavior of the machine being specified. Clearly, if the number of states is large, this can be a tedious and error-prone process. There has also been some work on high level languages for specifying state machines. Apparently, the first such language was developed by Parnas [11]. Two recent languages of this sort are Harel's Statecharts [10] and Berry's Esterel [1]. Esterel, in particular, has had a significant influence on the design of our language. Of all these languages, however, only SML allows various properties of the state machine to be verified automatically.

All SML programs represent synchronous circuits. At a clock transition, the program examines its input signals and changes its internal state and output signals accordingly. Since we are dealing with digital circuits where wires are either high or low, the basic data type is boolean. Each boolean variable my be declared to be either an input changed only by the external world but visible to the program, an output changed only by the program but visible to the external world, or an internal changed and seen only by the program. The hardware implementation of boolean variables may also be declared to be either active high or active low. A ".H" or ".L" suffix on the variable name in its declaration determines which case applies. Internal non-negative integer variables are also provided but are not discussed in this brief survey of the language.

An SML program has the following form:

program ⟨identifier⟩;
⟨declaration list⟩
⟨statement list⟩
endprog

where ⟨identifier⟩ is the name of the program, ⟨declaration list⟩ is a sequence of variable and

procedure declarations separated by semicolons, and ⟨statement list⟩ is a sequence of statements separated by semicolons. We refer the reader to [3] for the exact syntax of variable and procedure declarations.

The semantics of SML programs are different from most programming languages, since we are not only interested in what a statement does, but also how much time the statement takes to execute. The basic idea in SML is that computation is instantaneous, but changing a variable takes one clock cycle. When we refer to the "time" that a statement takes, we are referring to the execution time of the finite state machine. Hence, it is possible for computation to take no execution time since the computation is actually done at compile time.

Boolean input variables cannot be assigned new values, since inputs are changed by the environment only. Boolean output and boolean internal variables may be changed by:

raise (⟨variable⟩)
lower (⟨variable⟩)
invert (⟨variable⟩)

Each of these statements delays until the next clock transition, at which time the value of ⟨variable⟩ will be changed. The *raise* statement will assert ⟨variable⟩ (make it active), *lower* will negate it, and *invert* will force a change of value.

There are two types of looping statements in SML: the *while* statement and the *loop* statement. The *while* statement has the following syntax:

while ⟨boolean expression⟩ *do loop*
    ⟨statement⟩
endloop

At the beginning of the *while*, the ⟨boolean expression⟩ is evaluated, and nothing is done (in zero time) if the expression is false. If it is true, ⟨statement⟩ is executed. If ⟨statement⟩ completes execution in no time, the *while* statement delays until the next clock transition and then restarts the loop. If ⟨statement⟩ completes execution after some delay, the *while* statement is immediately restarted. The *exit* statement is used to jump out of the smallest enclosing *while* or *loop* statement. We will not discuss the syntax and semantics of the *loop* statement, since its behavior is similar to the *while*. For essentially the same reason we will not discuss the conditional statement or the *switch* statement.

The *parallel* statement provides a form of synchronous parallelism. This statement has the form:

parallel
 ⟨statement1⟩ ||
 ⟨statement2⟩ ||
 ...
endparallel

Each statement in the *parallel* examines the inputs and the current state and determines what changes it should be make to the output state at the next clock transition. The semantics of the *parallel* statement determine which of these changes are actually made. The rules are as follows.

i. If one or more of the statements executes a *break*, the *parallel* does nothing and the *break* causes a jump to the statement following the *parallel*.

ii. If none of the statements tries to change a variable, the variable remains unchanged.

iii. If exactly one statement tries to change a variable, this change is made at the next clock transition.

iv. If two or more statements try to change a boolean variable and they all agree on the new value, this change is made at the next clock transition.

v. If two or more statements try to change the same boolean variable and they do not agree on the new value, the result is undefined.

The *parallel* terminates when all of the statements in the *parallel* have finished executing or a *break* or *exit* is executed. The *exit* statement was discussed previously. The effect of the *break* statement is to immediately jump out of the smallest enclosing *switch* or *parallel* statement. One of the major uses of the *break* statement is to stop normal processing when an "interrupt" occurs.

In some cases, the timing rules of SML prevent complicated relationships from being simply described without delaying for more than one clock cycle. To alleviate this problem, SML has a compress statement of the form:

compress ⟨statement⟩ *endcompress*

The effect of the *compress* statement is calculated as if variable assignment takes no time in ⟨statement⟩. Then, after delaying one clock cycle, the changes made by the *compress* statement actually take effect.

Although our description of the language has been quite brief, it should be sufficient to understand the example in the next section. The compilation of SML programs in to Moore Machines is described in more detail in [3]. Considerable effort has spent in making the compiler as fast and efficient as possible. The state transition tables produced by the compiler may be implemented in hardware as PALs, PLAs, or ROMs. Various programs have been developed to make this last phase largely automatic. For example, a post-processor is available that produces output which is compatible with the Berkeley VLSI design tools.

4 Compositional SML

4.1 Language issues for synchronous finite-state machines

In this section we describe the extensions we have made to SML in order to allow modular design. In defining the new language, we have attempted to address a number of issues relating to the design of modular digital systems, the most important being concurrency and communication. In SML, the parallel statement creates threads of control which run conceptually in parallel, even though they are compiled into a single sequential machine. This technique is quite useful for describing systems which control concurrent activities, in much the same way that coroutines are useful in describing the function of an operating system. In order to create modular designs, however, we will

need to describe the interaction of subsystems which truly run in parallel. This has an important bearing on our model of communication between processes.

The means of communication between threads of control in the SML language is shared variables. Although this is often a straightforward way of conceptualizing systems, it has a number of drawbacks from the point of view of implementing modular systems. An obvious approach to the hardware implementation of shared variables between modules would be to assign a register for each shared variable, and a finite-state machine for each process. The difficulty with this approach is that, in any given clock cycle, any or all of the registers may be written or read. Thus, the complexity of the network interconnecting controllers and registers could quickly get out of hand. The problem with shared variables in a modular system is that, although they are conceptually simple, they do not correspond well to the the way communication occurs in digital systems.

For this reason, modules in CSML communicate via signals. Each signal is in the output set of exactly one module, and in the input set of zero or more modules. A signal takes on a simple boolean value of 0 or 1 for each clock cycle. Although using signals instead of shared variables for coordination of processes can be conceptually more complex, it allows for a straightforward implementation of modular designs, and yields a mathematically simple composition rule which lends itself well to formal verification (as will be seen below). Shared variables are preserved in CSML as the means of communication between parallel threads of control within a module.

4.2 The CSML process model

In order to define the semantics of CSML, we present a formal model of coordinating processes. A process in this model is a Moore machine, which we define as a 6-tuple $(I, O, Q, q_0, \phi, \theta)$ where I is the input set, O is the output set, Q is the set of states, $q_0 \in Q$ is the initial state, $\phi : Q \times 2^I \rightarrow Q$ is the transition function, and $\theta : Q \rightarrow 2^O$ is the output function. The interpretation of this structure is as follows: if $q \in Q$ is the current state and $i \subset I$ is the set input signals which are currently asserted, then the subset of outputs asserted by the machine is $\theta(q)$ and the next state is $\phi(q, i)$.

We define $P_1 \parallel P_2$, the parallel composition of processes P_1 and P_2, as follows. If $O_{P_1} \cap O_{P_2} = \emptyset$, then

$$
\begin{aligned}
I_{P_1 \parallel P_2} &\stackrel{\text{def}}{=} (I_{P_1} - O_{P_2}) \cup (I_{P_2} - O_{P_1}) \\
O_{P_1 \parallel P_2} &\stackrel{\text{def}}{=} O_{P_1} \cup O_{P_2} \\
Q_{P_1 \parallel P_2} &\stackrel{\text{def}}{=} Q_{P_1} \times Q_{P_2} \\
q_{0P_1 \parallel P_2} &\stackrel{\text{def}}{=} (q_{0P_1}, q_{0P_2}) \\
\phi_{P_1 \parallel P_2}((q_1, q_2), i) &\stackrel{\text{def}}{=} (\phi_{P_1}(q_1, (i \cup \theta_{P_2}(q_2)) \cap I_{P_1}), \phi_{P_2}(q_2, (i \cup \theta_{P_1}(q_1)) \cap I_{P_2})) \\
\theta_{P_1 \parallel P_2}(q_1, q_2) &\stackrel{\text{def}}{=} \theta_{P_1}(q_1) \cup \theta_{P_2}(q_2).
\end{aligned}
$$

The intuitive interpretation of this definition is as follows. The input set of $P_1 \parallel P_2$ is the set of signals which are inputs of P_1 or P_2, but not outputs of the other. The output set is the set of signals which are outputs of P_1 or P_2. For the obvious reason, we require that the output sets of P_1 and P_2 are disjoint (i.e., we don't allow connecting outputs to outputs). A state of $P_1 \parallel P_2$ is a state of P_1 combined with a state of P_2. The set of outputs of $P_1 \parallel P_2$ which are asserted is

simply the union of those asserted by P_1 and P_2. To determine the next state, each component examines its inputs, both those which are inputs of the composition, and those which are outputs of the other component, and independently decides on its next state. The next state of $P_1 \| P_2$ is then the combination of the next states of P_1 and P_2.

We also define a restriction operator $\downarrow_{out}$, which is used to hide outputs of a process. We define $P \downarrow_{out} S$ (read "P with outputs restricted to S") to be identical to P except that

$$O_{P \downarrow_{out} S} \stackrel{\text{def}}{=} O_P \cap S$$
$$\theta_{P \downarrow_{out} S}(q) \stackrel{\text{def}}{=} \theta_P(q) \cap S.$$

4.3 CSML syntax and semantics

We now describe the syntax and semantics of CSML. In addition to the statements of the SML language, CSML has three statements which are used for defining and interconnecting modules: *MODULE*, *DEFINE*, and *SYSTEM*. The *MODULE* statement is used to create a process. It takes the place of the *program* statement in SML. Its syntax is

MODULE (⟨input list⟩ ; ⟨output list⟩)
⟨body⟩
ENDMODULE.

The input and output sets for the process are given by ⟨input list⟩ and ⟨output list⟩. The ⟨body⟩ of the MODULE statement has the same syntax and semantics as a program in the SML language, as described in section 3. The *DEFINE* statement creates an abstract process from a concrete process and gives it a name. A separate name space is created for the process, and its input and output lists become formal parameter lists. The syntax of the *DEFINE* statement is:

DEFINE ⟨name⟩ ⟨module⟩ *or*
DEFINE ⟨name⟩ ⟨system⟩

The *SYSTEM* statement creates a process which is the parallel composition of two or more processes. Its syntax is:

SYSTEM (⟨input list⟩ ; ⟨output list⟩)
⟨component⟩$_1$
⟨component⟩$_2$
⋮
⟨component⟩$_n$
ENDSYSTEM.

where each ⟨component⟩ is either a *MODULE* statement, another *SYSTEM* statement, or an instantiation of a definition, which has the form

⟨name⟩ (⟨actual input list⟩ ; ⟨actual output list⟩)

```
DEFINE Producer MODULE(request; acknowledge,produce)
  input request;
  output acknowledge=false;

  loop
    while(!request)do loop skip endloop;
    raise(produce); lower(produce);
    raise(acknowledge);
    while(request)do loop skip endloop;
    lower(acknowledge)
  endloop
ENDMODULE

DEFINE Consumer MODULE(acknowledge; request, consume)
  input acknowledge;
  output request=false;

  loop
    raise(request);
    while(!acknowledge)do loop skip endloop;
    raise(consume); lower(consume);
    lower(request);
    while(acknowledge)do loop skip endloop
  endloop
ENDMODULE

SYSTEM(;produce,consume)
  Producer(req; ack,produce)
  Consumer(ack; req,consume)
ENDSYSTEM
```

Figure 2: Producer-consumer program

If the latter form is used, a concrete process is created from the named abstract process by substitution of actual parameters for formal parameters. Semantically, if the components of the *SYSTEM* statement represent processes $P_1, P_2, ..., P_n$, then the meaning of the statement is:

$$(P_1 \,||\, P_2 \,||\, \ldots \,||\, P_n)\downarrow_{out}\langle\text{output list}\rangle.$$

Since inputs cannot be hidden, we require that the ⟨input list⟩ contains precisely the input set of $P_1 \,||\, P_1 \,||\, \ldots \,||\, P_n$. Figure 2 gives a simple example of a CSML program—a system composed of a producer module and a consumer module which synchronize using a four-phase handshake.

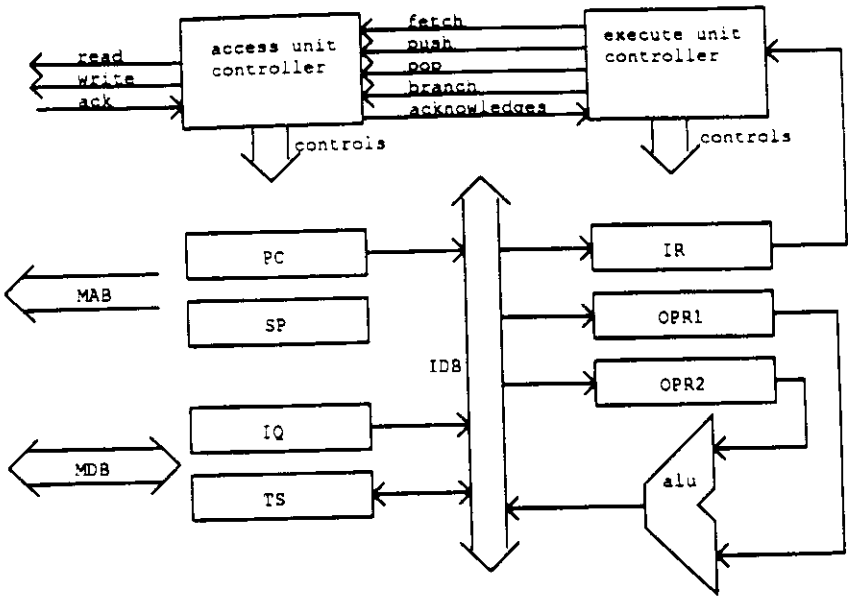

Figure 3: CPU block diagram

5 Application: a simple CPU

To illustrate to the use of CSML to describe modular controllers, we present the detailed design and verification of the controller for a simple CPU. Part of the motivation for this exercise is to guage how effective our tools and methodologies would be in doing real digital designs. We have attempted to include enough detail so that the reader can judge how close our example CPU is in complexity to a real CPU, and yet not enough to be overwhelming. On a first reading, the reader might wish to skim subsections 5.1–5.4 to arrive at the conclusions in subsection 5.5.

5.1 Architectural description

A block diagram of the CPU is given in figure 3. The CPU is divided into two modules, the access unit (AU) and execute unit (EU), in order to increase its performance by carrying out memory accesses and instruction executions in parallel. The access unit's function is to fetch instructions and store them in the instruction queue (IQ), and to maintain a cache of the top location of the stack in a special top-of-stack register (TS). The execution unit's function is to interpret instructions of the CPU's machine code (which it stack based).

The machine instructions are 8 bits, and are packed two per 16 bit machine word. There are two addressing modes: stack, and immediate. In order to implement a useful CPU, we would also require, for example, direct and stack indirect modes. However, in order to simplify the design and exposition, and to avoid such complications as fetching words on non-word boundaries, we have not included them. There are three basic classes of instructions: control, one-operand, and two-operand. Instructions that take one operand specify an addressing mode for both source and destination. Instructions that take two operands specify both source addressing modes, and use stack mode implicitly for the destination. The control instructions (branch, call, and return) specify

Signal	Function
fetch	$PC \leftarrow PC + 1$ $\quad (fetch = fetch\text{-}req \wedge fetch\text{-}rdy)$
PC-MAB	*MAB* (memory address bus) $\leftarrow PC$
PC-IDB	*IDB* (internal data bus) $\leftarrow PC$
branch	$PC \leftarrow IDB$
push	$SP \leftarrow SP - 1$ $\quad (push = push\text{-}req \wedge push\text{-}rdy)$
pop	$SP \leftarrow SP + 1$ $\quad (pop = pop\text{-}req \wedge pop\text{-}rdy)$
SP-MAB	$MAB \leftarrow SP$
MDB-IQ	$IQ \leftarrow MDB$
IQ-IDB	if *PC* is even, $IDB \leftarrow IQ_{8-15}$ if *PC* is odd, $IDB \leftarrow IQ_{0-7}$
TS-MDB	$MDB \leftarrow TS$
TS-IDB	$IDB \leftarrow TS$
MDB-TS	$TS \leftarrow MDB$
IDB-TS	$TS \leftarrow IDB$

Table 1: Access unit control signals

Signal	Function
IDB-IR	$IR \leftarrow IDB_{0-7}$
IDB-OPR1	$OPR1 \leftarrow IDB$
IDB-OPR2	$OPR2 \leftarrow IDB$
RES-IDB	$IDB \leftarrow RESULT$ (of ALU)

Table 2: Execute unit control signals

one of eight conditions codes and select either direct or program counter relative addressing.

The access unit has four 16-bit registers: the program counter (PC), stack pointer (SP), instruction queue (IQ) and top-of-stack register (TS) (see figure 3). The PC is equipped with an incrementer, and the SP with an incrementer/decrementer. The control signals for these registers and their functions are summarized in table 1.

The execute unit has two 16-bit operand registers (OPR1 and OPR2), an 8-bit instruction register (IR), a 3-bit condition code register (CCR), and a 16-bit ALU. There is a 16-bit internal data bus (IDB) by which data are communicated between the EU and AU. A block diagram of the ALU and the definition of its control signals are given in figure 4. The remaining control signals of the execution unit are summarized in table 2.

The access and execute unit controllers communicate via three request signals, *push-req*, *pop-req* and *fetch-req*, three corresponding ready signals, *push-rdy*, *pop-rdy* and *fetch-rdy*, as well as the signal *branch*, which causes the PC to be loaded and the instruction queue to be flushed. The

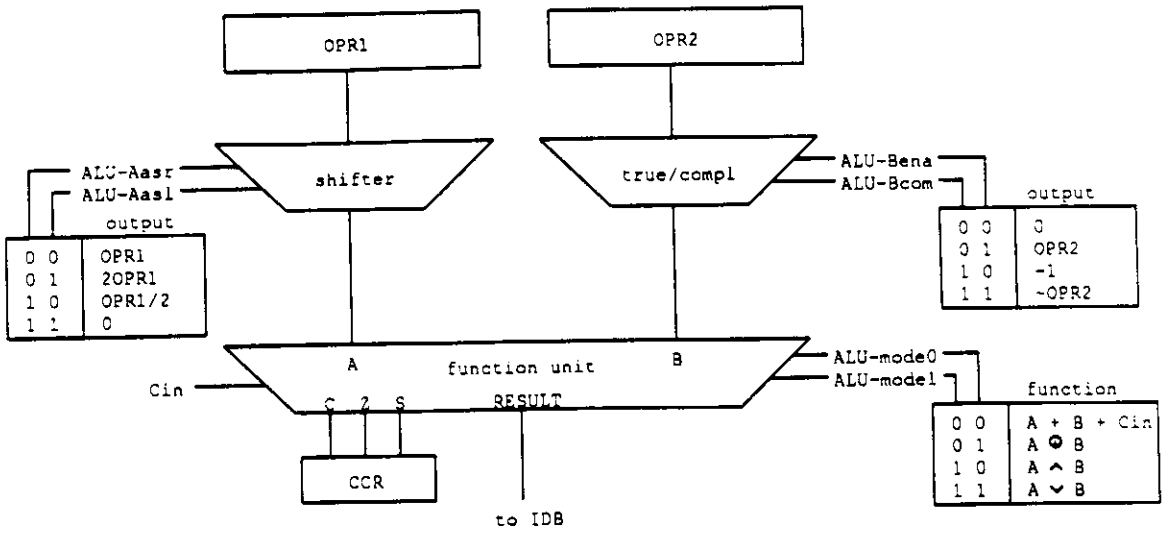

Figure 4: ALU block diagram

execution unit signals its intention to perform a push, pop or (instruction) fetch operation by asserting the appropriate request signal. If the ready signal is already asserted it proceeds, otherwise it waits for the ready signal to be asserted.

The AU communicates with memory via two buses, the memory data bus (MDB) and the memory address bus (MAB), and via three control signals: *mem-rd*, *mem-wr* and *mem-ack*. The protocol for a memory access is as follows. First, the AU asserts one of the memory control signals (*mem-rd* for a read, and *mem-wr* for a write), and simultaneously causes the appropriate address to be driven onto the MAB (using signals *PC-MAB* or *SP-MAB*). On a write, the AU drives the MDB with the contents of the top-of-stack register (using the signal *TS-MDB*). On a read, it loads the MDB data into one of its registers (using signals *MDB-IQ* or *MDB-TS*). It then waits for *mem-ack* to be asserted by the memory system, at which time it completes the access by lowering its control signals.

5.2 Informal specification of controllers

In this section, we give an informal specification for the controllers, from which we derive a design expressed in CSML. We will not give the entire CSML code (which is about five pages long), but will illustrate the following discussion with fragments from the code. Our informal specification will be used in subsection 5.4 to derive a formal specification in CTL for the access unit.

5.21. The access unit controller

We begin with the access unit controller. The AU controller has two functions, which it performs conceptually in parallel. The first is managing the instruction queue (IQ). The controller must keep track of the status of the IQ register, fetching a new instruction word when the IQ becomes empty, and flushing the queue when a branch occurs. Figure 5 gives an abstract state diagram that describes how the various operations coordinated by the AU controller affect the status of the

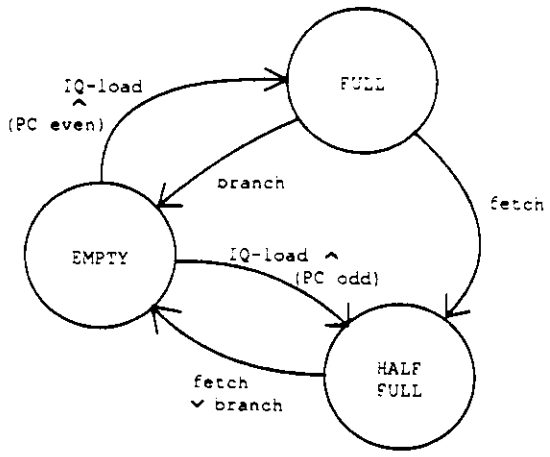

Figure 5: IQ manager state diagram

IQ register. An *IQ-load* operation (loading the IQ register from the value in memory pointed to by the PC) results in the queue being full if the PC is even, or half full if the PC is odd, since memory is only read on word boundaries. A fetch of an instruction byte by the EU results in a full queue becoming half full, or a half full queue becoming empty. A branch always results in an empty queue. Note that the AU controller can determine when the IQ is half full by examining bit 0 of the PC, to see if it is even or odd. The CSML code in figure 6 computes the status of the IQ and stores it in a variable called `IQ-st`. It also manages an output called `fetch-rdy` which signals to the EU that an instruction byte is ready in the queue. The code that actually carries out the *IQ-load* operation will be discussed later.

The other function of the AU controller is to manage the top-of-stack cache. To do this, it distinguishes three states of the TS register: INVALID, VALID, and MODIFIED. The TS is in the VALID state when its contents match the value in memory pointed to by the SP, it is MODIFIED when the TS has been written, but the contents have not yet been copied back to memory, and it is INVALID otherwise. In particular, we cannot perform a push operation when the TS is MODIFIED, because previously pushed data would be lost, and we cannot perform a pop operation when the TS is INVALID, because incorrect data would be read. Figure 7 gives an abstract state diagram that describes the effects of AU controller operations. The CSML code in figure 8 computes the status of the TS and stores it in a variable called `TS-st`. It also manages the outputs `push-rdy` and `pop-rdy` which signal to the EU that the TS register is ready for a push or pop operation respectively.

Finally, we define a third parallel thread of control, which acts like a monitor, insuring the the TS manager and IQ manager do not attempt to access memory at the same time. Although monitors *per se* are not a feature CSML, it is easy to effect this simple monitor using shared variables within a module. It would also have been possible to make the TS and IQ managers separate modules and have them communicate with a separate arbiter module using signals, but the resulting program would have been more difficult to understand. The monitor thread waits in a loop for either the IQ to become EMPTY, or the TS to become MODIFIED or INVALID. It then performs the appropriate memory access: *IQ-load*, *TS-load*, or *TS-store*, respectively. The CSML code appears in figure 9. Note that when the TS register is in the INVALID state, we allow a push request to take priority over a TS-load operation, but once the TS-load operation is

```
loop
  compress
    switch
      case branch:
        lower(fetch-rdy); IQ-st := EMPTY; break;
      case fetch:
        if (IQ-st == FULL-OR-HALF-FULL) & PCO then
         IQ-st := EMPTY; lower(fetch-rdy);
         endif; break;
       case IQ-load-done:
         IQ-st := FULL-OR-HALF-FULL; raise(fetch-rdy);
       default: skip;
    endswitch
  endcompress
endloop
```

Figure 6: CSML code for the IQ manager. Note that the literals **fetch** and **IQ-load-done** are actually macro names which expand to conditional expressions. Their definitions can be found in section 5.4. Also, note that the **compress** statement causes each excecution of the loop to occur in one clock cycle.

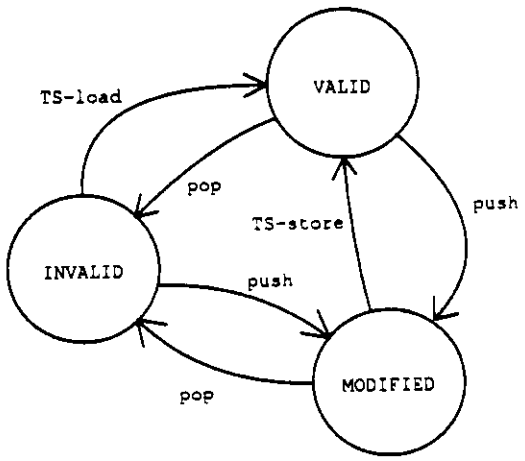

Figure 7: TS manager state diagram

```
loop
  compress
    switch
      case push:
        lower(push-rdy); raise(pop-rdy);
        TS-st := MODIFIED; break;
      case pop:
        lower(pop-rdy); raise(push-rdy);
        TS-st := INVALID; break;
      case TS-load-done:
      case TS-store-done:
        raise(push-rdy); raise(pop-rdy);
        TS-st := VALID; break;
      default: skip;
    endswitch
  endcompress
endloop
```

Figure 8: Code for TS manager

started, we lower push-rdy to prevent push operations from interfering with the memory cycle. A corresponding relationship exists between TS-store and pop.

The routine read takes as its arguments a control signal to raise to drive the MAB bus, and a control signal to raise to load the IQ or TS registers. It is defined in figure 10. follows: Since the calls to read appear inside compress statements, this routine only takes time if the wait loop has to be executed.

The overall structure of the AU controller code is a three-way *parallel* statement as shown in figure 11.

5.22. The execution unit controller

The job of the execution unit is more straightforward. It has only one thread of control, and proceeds as follows. It first loads an instruction from the IQ into the IR (i.e., performs a fetch operation). It then decodes the instruction, and jumps to an appropriate routine to interpret that instruction. When the instruction is completed, it starts again. The `dofetch` routine (figure 12) is used to load the instruction. The procedure takes a parameter `ctl`, which is a control signal that loads a register from the IDB. This allows `dofetch` to be used to load the IR or an operand register from the instruction queue.

The main loop of the EU controller has the structure of figure 13. Literals such as BR and CALL in figure 13 are macros, defined as boolean expressions on bits 4-7 of the IR, which hold the operation code field of the instruction. The actual instruction routines break down into three categories: control instructions, one operand instructions, and two operand instructions. The instructions and their functions are summarized in table 3. We take as an example the CALL instruction, which is

```
loop
  switch
    case IQ_st == EMPTY:
      compress read(PC-MAB,MDB-IQ) endcompress;
      break;
    case TS_st == INVALID & !push-req:
      compress lower(push-rdy); read(SP-MAB,MDB-TS) endcompress;
      break;
    case TS_st == MODIFIED & !pop-req:
      compress lower(pop-rdy); write(SP-MAB,TS-MDB) endcompress
      break;
    default: skip;
  endswitch
endloop
```

Figure 9: CSML code for memory access "monitor"

```
procedure read(addrctl,datactl)
        raise(mem-rd); raise(addrctl); raise(datactl);
        while !mem-ack do loop skip endloop;
        lower(mem-rd); lower(addrctl); lower(datactl);
endproc;
```

Figure 10: Routine read.

```
DEFINE AU MODULE(...;...)
  ... declarations ...
  ... procedures ...
  parallel
  ... memory access monitor ...
  ||
  ... TS manager ...
  ||
  ... IQ manager ...
 endparallel
ENDMODULE
```

Figure 11: Overall structure of AU controller code.

```
procedure do-fetch(ctl)
  parallel raise(fetch-req) || raise(ctl) endparallel;
  while !fetch-rdy do loop skip endloop;
  parallel lower(fetch-req) || lower(ctl) endparallel
endproc;
```

Figure 12: The procedure dofetch.

```
loop
  do-fetch(IDB-IR);
  switch
    case BR:
      ... execute branch instruction ...
    case CALL:
      ... execute call instruction ...
    ... other instructions ...
    case XOR
      ... execute exclusive-or instruction ...
  endswitch
endloop
```

Figure 13: The structure of the EU controller code.

interpreted by the code of figure 14.

First, do-fetch is called to load the branch target into the operand register OPR1. The condition is then decoded (the literals C, NC, Z, NZ, N, NN, and T are defined as predicates on the IR bits 0-2 which hold the condition field of the instruction). If the condition evaluates to true, the routine do-push is called to push the current PC onto the stack (do-push is similar to do-fetch), and do-branch is called to load the PC (note that the branch target flows through the ALU unchanged, since all of the ALU controls are set to zero at the conclusion of each instruction–see figure 4).

5.3 Some observations on implementing hardware in CSML

When compiled as separate modules, the AU and EU controllers have 13 and 98 states respectively. It is clear that there is no need to decompose the AU into modules, since the number of states is quite small (although early, incorrect versions had as many as 60 states). On the other hand, merging the AU and EU into one module would result in a very complex machine with 1274 states. Although, obviously, the state-space of this large machine would have a simple decomposition, the information about how to decompose it would be lost. As we will see in the next section, this modular decomposition will be an advantage in automatic verification.

This example also illustrates the difference between true parallelism (between modules) and

```
case CALL:
  do-fetch(IDB-OPR1);
  if (C&carry | NC&!carry | Z&zero | NZ&!zero | N&sign | NN&!sign | T) then
    do-push(PC-IDB);
    do-branch(RES-IDB)
  endif;
  break;
```

Figure 14: Code for the CALL instruction.

Mnemonic	Function	Mnemonic	Function
Control instructions			
BR	Branch on condition	CALL	Call on condition
RET	Return on condition		
One-operand instructions			
MOVE	source $\rightarrow$ dest	INC	source + 1 $\rightarrow$ dest
DEC	source $-$ 1 $\rightarrow$ dest	COM	$\sim$ source $\rightarrow$ dest
NEG	$-$source $\rightarrow$ dest	ASL	2source $\rightarrow$ dest
ASR	source/2 $\rightarrow$ dest		
Two-operand instructions			
ADD	$src_1 + src_2 \rightarrow$ stack	SUB	$src_1 - src_2 \rightarrow$ stack
AND	$src_1 \wedge src_2 \rightarrow$ stack	OR	$src_1 \vee src_2 \rightarrow$ stack
XOR	$src_1 \oplus src_2 \rightarrow$ stack		

Table 3: Instruction set summary

conceptual parallelism (between threads of control on the same module), and provides some motivation for the existence of two constructs for parallel execution in CSML: the *MODULE* and *parallel* statements. Although we feel that it is important to break up large controllers into modules, this does not mean that every thread of control should be broken out into a separate module. In the case of the AU, which has three tightly coupled threads of control, shared variables provide a conceptually simple mode of interaction, and the resulting number of states is not large. Thus, we feel justified, at the risk of seeming unparsimonious, in providing both mechanisms in the language.

Obviously, the CPU design presented here was not intended to be a practical one. From a practical point of view, however, at least one criticism of CSML should be made. The Moore-machine semantics of CSML (and its predecessor SML) require that raising or lowering a signal always involves one clock cycle of delay. The astute reader may have noticed that in the instruction fetch routine of the EU, one clock cycle is simply wasted in order to raise the signal *fetch-req*. This same consideration also made it necessary to use "ready" signals (essentially a pre-acknowledge), since it is not possible to respond to a request with an acknowledge in the same clock cycle. One advantage of the Moore-machine semantics is that all signals between modules effectively pass through a pipeline register. This means that timing of modules can be verified independently. Nonetheless, it seems to be a serious limitation for practical design.

5.4 Formal specification for the access unit

In this section we present a formal specification of the AU in CTL. The formal specification for the EU is not presented. It is extremely detailed (i.e., there are a large number of cases), but not very interesting. Before proceeding we would like to define a few predicates which will simplify the specifications and the following discussion. They are defined as follows:

$$
\begin{aligned}
\mathit{fetch} &\equiv \mathit{fetch\text{-}req} \wedge \mathit{fetch\text{-}rdy} \\
\mathit{push} &\equiv \mathit{push\text{-}req} \wedge \mathit{push\text{-}rdy} \\
\mathit{pop} &\equiv \mathit{pop\text{-}req} \wedge \mathit{pop\text{-}rdy}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{IQ\text{-}load} &\equiv \mathit{mem\text{-}rd} \wedge \mathit{PC\text{-}MAB} \wedge \mathit{MDB\text{-}IQ} \\
\mathit{IQ\text{-}load\text{-}done} &\equiv \mathit{IQ\text{-}load} \wedge \mathit{mem\text{-}ack} \\
\mathit{TS\text{-}load} &\equiv \mathit{mem\text{-}rd} \wedge \mathit{SP\text{-}MAB} \wedge \mathit{MDB\text{-}TS} \\
\mathit{TS\text{-}load\text{-}done} &\equiv \mathit{TS\text{-}load} \wedge \mathit{mem\text{-}ack} \\
\mathit{TS\text{-}store} &\equiv \mathit{mem\text{-}wr} \wedge \mathit{SP\text{-}MAB} \wedge \mathit{TS\text{-}MDB} \\
\mathit{TS\text{-}store\text{-}done} &\equiv \mathit{TS\text{-}store} \wedge \mathit{mem\text{-}ack} \\
\mathit{PC\text{-}roll\text{-}over} &\equiv \mathit{fetch} \wedge PC_0.
\end{aligned}
$$

The predicate *fetch* indicates that an instruction is being loaded into the IR and the PC is being incremented. Similar meanings are associated with *push* and *pop*. *IQ-load* is true when a memory cycle is in progress which is loading the IQ, and *IQ-load-done* is true on the last clock cycle of a memory cycle. Similar meanings are associated with *TS-load*, *TS-load-done*, *TS-store* and *TS-store-done*. *PC-roll-over* is true when the PC is crossing a word boundary.

The basic safety requirement for the IQ manager is that, whenever a fetch occurs, the output of the IQ matches the memory location pointed to by the PC. We assume that the stack does not

overwrite program memory. Thus, the only conditions which can violate the above requirement are:

i. The PC rolls over to the next word boundary.

ii. A branch occurs.

iii. The system is reset to its initial state.

We would like to assert that, between the time one of these three things happens and IQ load is completed, that a fetch never occurs. In order to express this in CTL, it is convenient to define a macro-operator

$$(p \ BEFORE \ q) \equiv (\neg q \mathcal{U} p)$$

that is, p occurs *BEFORE* q if and only if $\neg q$ is true *until* p is true. Note that we intend the *strong* until in this case – if p never occurs, it cannot be said to have occurred before q. Our requirement for case (i) above can now be expressed in *CTL* as follows

$$\forall G(\textit{PC-roll-over} \rightarrow \neg \exists X \exists (\textit{fetch BEFORE IQ-load-done}))$$

that is, globally, if the PC rolls over to the next word boundary, then (beginning in the next state) there exists no path along which an instruction fetch occurs before the instruction queue has been loaded. We can state the requirement for case (ii) as follows

$$\forall G(\textit{branch} \rightarrow \neg \exists X \exists (\textit{fetch BEFORE IQ-load-done}))$$

and for case (iii) simply as

$$\neg \exists (\textit{fetch BEFORE IQ-load-done})$$

This specifies that, from the initial state, there is no path along which a *fetch* operation occurs before the first *IQ-load* operation completes (the truth value of all formulas is implicitly referenced to the initial state). Of course, we also require that the IQ manager never drives the MAB or overwrites the IQ spuriously. These conditions are expressed by the following CTL formulas:

$$\forall G(\textit{PC-MAB} \rightarrow \textit{IQ-load})$$
$$\forall G(\textit{MDB-IQ} \rightarrow \textit{IQ-load}).$$

The first formula states, for example, that the PC is only driven onto the memory address bus when an *IQ-load* operation is in progress.

The correctness conditions for the TS manager may at first seem more complex, however it is relatively straightforward to derive them from the state transition diagram of figure 7. In the VALID state, any of the operations *push*, *pop*, *TS-load* and *TS-store* are allowable (the latter two are not present in the diagram, but executing them in this state will cause no harm, since the memory contents match the TS register). From the MODIFIED state, which is entered only by a *push* operation, another *push* or a *TS-load* may not occur before either a *pop* or *TS-store* is completed. We can express this condition with the following formula

$$\forall G(\textit{push} \rightarrow \neg \exists X \exists (\textit{push} \vee \textit{TS-load BEFORE TS-store-done} \vee \textit{pop}))$$

From the INVALID state, which is entered only by a *pop* or a reset to the initial state, *pop* and *TS-store* may not occur before either a *push* or *TS-load* is completed. Thus we have the following formulas:

$$\forall G(pop \rightarrow \neg\exists X\exists(pop \vee TS\text{-}store\ BEFORE\ TS\text{-}load\text{-}done \vee push))$$
$$\neg\exists(pop \vee TS\text{-}store\ BEFORE\ TS\text{-}load\text{-}done \vee push)$$

Of course, we also require that the TS manager not spuriously drive the MAB or MDB buses or overwrite the TS register:

$$\forall G(MDB\text{-}TS \rightarrow TS\text{-}load)$$
$$\forall G(TS\text{-}MDB \rightarrow TS\text{-}store)$$
$$\forall G(SP\text{-}MAB \rightarrow (TS\text{-}load \vee TS\text{-}store))$$

The first of these, for example, states that the top-of-stack register is loaded from the memory data bus only during a *TS-load* operation.

In order for memory cycles to operate correctly, we have the following requirements. First, the address, data and control signals must remain stable during an entire memory cycle. This means that, if an *IQ-load*, *TS-load* or *TS-store* condition occurs, that condition must persist up to and including the clock cycle when *mem-ack* is asserted by the memory system. Further, as the address must not change during a memory cycle, we require that during an *IQ-load* cycle, the program counter not change words, and that during *TS-load* and *TS-store* cycles, the stack pointer not change. These requirements are expressed in the following formulas:

$$\forall G(IQ\text{-}load \rightarrow \forall((IQ\text{-}load \wedge \neg(PC\text{-}roll\text{-}over \vee branch))\mathcal{U}(IQ\text{-}load \wedge mem\text{-}ack)))$$
$$\forall G(TS\text{-}load \rightarrow \forall(TS\text{-}load \wedge \neg(push \vee pop)\mathcal{U}(TS\text{-}load \wedge mem\text{-}ack)))$$
$$\forall G(TS\text{-}store \rightarrow \forall((TS\text{-}store \wedge \neg(push \vee pop))\mathcal{U}(TS\text{-}store \wedge mem\text{-}ack)))$$

Second, we allow no spurious memory accesses.

$$\forall G(mem\text{-}wr \rightarrow TS\text{-}store)$$
$$\forall G(mem\text{-}rd \rightarrow (TS\text{-}load \vee IQ\text{-}load))$$

The above formulas represent safety properties, i.e. they are characterized by the statement "nothing bad ever happens." Unfortunately, they cannot form a complete specification, since a controller which did nothing at all would satisfy all of the above assertions. Thus, we include the following liveness requirement, which states, in effect, that the CPU always eventually executes another instruction:

$$\forall G\forall F fetch$$

5.5 Summary of model checking results

Finally, we describe the application of the CTL model checker to automatically verify that our controller meets the above specification. Since the composition of the AU and EU has 1274 states, while the AU by itself has only 13 states, one might be tempted to check the formulas on the AU in isolation, and then infer that their correctness holds in the composition. Unfortunately,

this inference would not necessarily be a valid one. The CTL properties of a module are not always preserved when the module is composed with other modules. In any case, the correctness of a module very probably depends in some part on properties of the modules with which it is composed. It is possible, however, to apply the *interface rule* to replace the EU module with a reduced module EU', which has only 17 states (as opposed to the 98 states of the EU). We derive EU' by first hiding the outputs of the EU controller which control the data path of the EU, then applying a Moore machine minimization algorithm. The reason for the large reduction is that, while the EU interprets a large number of instructions, the memory access for these instructions fall into a few basic patterns. The interface rule guarantees that our specification will hold in the composition of AU and EU if and only if it holds in the composition of AU and EU'. This latter composition has only 221 states. This illustrates the point made earlier that modular design can increase the efficiency of automatic verification.

In the process of verification, the model checker pointed out two bugs in the original design. The first was that the do-branch routine did not check to make sure that a *IQ-load* operation was not in progress before modifying the PC. The second was that the *TS-store* code in the memory access monitor incorrectly asserted *MDB-TS* instead of *TS-MDB*. Counterexamples produced by the model checker made it a straightforward task to find and correct the errors. The total time to verify the 16 formulas of the AU specification on our (corrected) 221 state machine was 36.2 seconds, running on a Sun-3 workstation.

6 Directions for Future Research

We should point out that the task of verifying the CPU does not end with the verification of the controllers. It is necessary, of course, to provide a formal specification of the CPU as a whole, and to prove on the basis of the controller specification and a formal model of the data path circuitry that the CPU specification is valid. Unfortunately, we do not have the machinery to do this in an automated way. The state space of the data path section is far too large to apply model checking techniques, and in any case, CTL is most likely not expressive enough to specify the CPU as a whole. One approach to this problem might be to integrate the CTL model checker with an automatic theorem prover (or proof checker), which could perform the final step. Bryant's method of symbolic simulation [6] would probably be of considerable use in this endeavor. We leave the problem of integrating control and data as an open one here, and an area for future research.

Even with the module feature CSML has some limitations. Perhaps the most difficult issue is how to deal with nondeterminism. Currently, SML processes are completely synchronous and deterministic. In practice, however, it is important to be able to reason about processes that run on different clocks or execute asynchronously. Another important use of nondeterministic processes is to form an abstract representation of a class of deterministic machines. Such a process can be used to prove properties of the entire class, often with greatly reduced complexity. More research is needed to handle this problem within our current framework.

Finally, additional research is needed on techniques for compositional reasoning about SML processes. The interface rule handles formulas that are boolean combinations of temporal properties of the individual processes. We are currently unable to handle more general properties involving temporal assertions about several processes. Furthermore, in some verification problems it may be necessary to combine the use of the interface rule with proofs of validity for certain CTL formulas. In general, such proofs require a complicated decision procedure. Fortunately, we believe that it

will be possible to use the model checker together with informal manual reasoning to handle most of these cases.

References

[1] G. Berry and L. Cosserat. *The ESTEREL Synchronous Programming Language and its Mathematical Semantics.* Technical Report, Ecole Nationale Superieune des Mines de Paris, 1984.

[2] M. C. Browne. An improved algorithm for automatic verification of finite state machines using temporal logic. In *Proceedings of the Conference on Logic in Computer Science,* June 1986.

[3] M. C. Browne and E. M. Clarke. SML: a high level language for the design and verification of finite state machines. In *IFIP WG 10.2 International Working Conference from HDL Descriptions to Guaranteed Correct Circuit Designs, Grenoble, France,* IFIP, September 1986.

[4] M. C. Browne, E. M. Clarke, and D. L. Dill. Automatic circuit verification using temporal logic: two new examples. In *Formal Aspects of VLSI Design,* Elsevier Science Publishers, 1986.

[5] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers,* C-35(12), December 1986.

[6] R. E. Bryant. *Two Papers on a Symbolic Analyzer for MOS Circuits.* Technical Report 87-106, Carnegie Mellon University, 1987.

[7] E. M. Clarke, S. Bose, M. C. Browne, and O. Grumberg. *The Design and Verification of Finite State Hardware Controllers.* Technical Report 87-145, Carnegie Mellon University, 1987.

[8] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *ACM Transactions on Programming Languages and Systems,* 8(2):244-263, 1986.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems,* 8(2):244-263, 1986.

[10] D. Harel. *Statecharts: A Visual Approach to Complex Systems.* Technical Report CS84-05, The Weizmann Institute of Science, February 1984.

[11] D. L. Parnas. A language for describing the functions of synchronous systems. *Communications of the Association of Computing Machinery,* 9:72-75, February 1966.

[12] J. D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1984.