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

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

School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213, U. S. A.

Abstract

SML is a language for describing complex finite-state hardware controllers. It provides many of the standard control structures found in modern programming languages. The state tables produced by the SML compiler can be used as input to a temporal logic model checker that can automatically determine whether a specification in the logic CTL is satisfied. We describe extensions to SML for the design of modular controllers. These extensions allow a compositional approach to model checking which can substantially reduce its complexity. To demonstrate our methods, we discuss the specification and verification of a simple CPU controller.

[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.

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 that we call *compositional* SML (or CSML), which is intended to allow the modular design of finite-state controllers. 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 that 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 important aspect of modularity which relates specifically to finite-state controllers and their verification. Because of the state explosion phenomenon, a controller which coordinates a number of concurrent activities may have an extremely large number of states. In fact, this number may grow exponentially with the number of parallel threads of control. It is therefore important to be able to design such controllers as systems of communicating finite-state machines instead of as single, purely sequential, state machines.

The state explosion problem also has an important bearing on automatic verification. A *model checking algorithm* ([2, 8]) can determine the truth of CTL formulas with respect to a given finite state model, however the complexity of verification is related to the number of states in the model. If a system is composed of communicating modules, we can use a theorem called the *interface rule* to reduce the state space of the model without affecting the truth value of CTL formulas. The interface rule states a set of conditions under which a module may be replaced by a reduced version

called an *interface process*.

To illustrate our new approach, we describe the design and verification of the controller for a simple CPU with decoupled access and execution units. We describe (in part) the implementation of the controller in CSML and give 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 [7]. 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 examples, we consider two 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. $\forall G(send \rightarrow \forall(send \mathcal{U} rcvd))$ 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. A counterexample is a sequence of states in the model which shows the formula is false. This can be of tremendous help in locating 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 paths along which certain conditions hold infinitely often. This can be necessary when checking properties of systems which have external inputs; for example, in verifying our example CPU, we used fairness constraints to represent the assumption that when the CPU executes a memory read, the external memory system eventually returns the desired data.

We deal with the state explosion in concurrent systems using 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 is equivalent to P_1 when observed via the wires S (and similarly for A_2 and P_2). The key point here is that A_1 must be equivalent ($\equiv$) to P_1 on S in an appropriate sense, in order to preserve the truth value of logical formulas in the composition. 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),

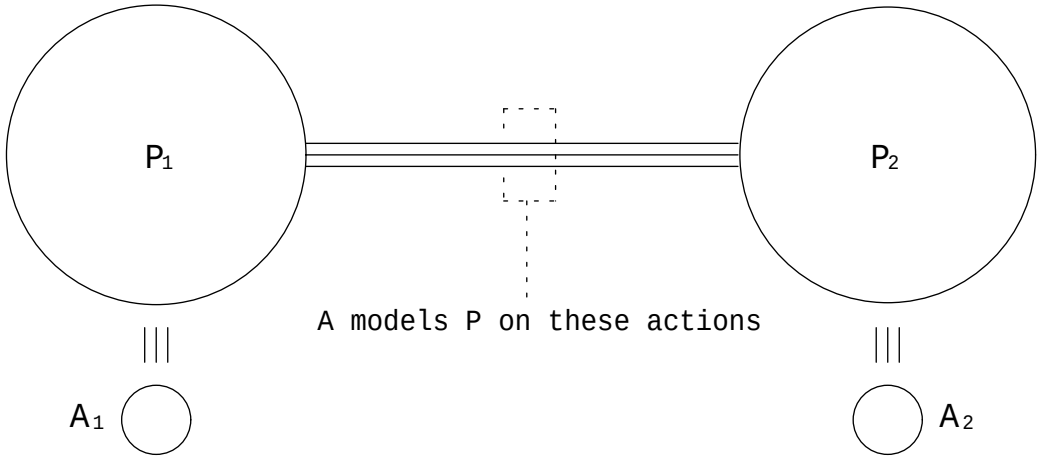

Figure 1: The interface rule

then φ is true in $P_1 \| P_2$. In a loosely coupled system, A_1 will almost always have far fewer states than P_1, and thus $A_1 \| P_2$ will be much smaller than $P_1 \| P_2$. The interface rule can be extended to apply 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]. Other state machine languages are described in [13, 12, 10, 1]. Although SML was developed for specifying complicated finite state machines, it has many of the standard control structures found in modern imperative programming languages, including a while statement, a conditional, a case 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 finite state transition table.

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, the basic data type is boolean. Each boolean variable my be declared to be either

(1) an input changed only by the external world but visible to the program, (2) an output changed only by the program but visible to the external world, or (3) an internal variable changed and seen only by the program. 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.

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. According to the operational sematics of SML, most statements execute in zero time. Changing a variable value, however, takes one unit of time (one clock cycle).

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*. We will not discuss the conditional statement or the *switch* statement, as they are similar to constructs in common imperative programming languages.

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

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

The statements in the *parallel* construct execute concurrently in lockstep. The *parallel* terminates when all of the statements in the *parallel* have finished executing or a *break* is executed. 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 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

In this section we describe two extensions we have made to SML which allow the hierarchical definition and interconnection of modules. A *process* in CSML has the following syntax:

process ⟨identifier⟩;
  ⟨declaration list⟩
  ⟨statement list⟩
endproc

A process compiles into a separate Moore machine which communicates with other processes running concurrently (in lockstep). The ⟨statement list⟩ in a program or process may be replaced by a list of processes. The scoping of variables in CSML is similar to that of *PASCAL*. A process may access the variables of any process which lexically contains it (All processes may access the variables of the main program). In order to alter a variable of another process, however, a process must declare that variable as an output in its declaration list. By requiring that no variable may be declared as an output by more than one process, we insure that CSML processes can be implemented as communicating Moore machines. Variables which are not declared as outputs by any process are considered inputs to the program.

The other way in which CSML extends SML is the *processtype* statement, which defines a reusable process type. The *processtype* statement may appear in the declaration list of a program or process, and has the following form:

processtype ⟨identifier⟩ (⟨formal parameter list⟩);
  ⟨declaration list⟩
  ⟨statement list⟩
endtype

A process type is instantiated as a process by a statement of the following form:

process ⟨process identifier⟩ : ⟨processtype identifier⟩ (⟨actual parameter list⟩);

The scoping rules for process type identifiers and variables referenced in process types are the same as for variables referenced in processes. In particular, since lexical scoping is used, an instantiated process operates in the context in which it was defined, not in the context in which it is instantiated.

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. The formal model of communicating Moore machines which underlies the semantics of CSML is described in [9].

5 Application: a simple CPU

To illustrate to the use of CSML to describe modular controllers, we present the 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 endeavored to include enough detail so that the reader can judge how close our example CPU is to the complexity to a real design, and yet not enough to be overwhelming.

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 execution unit (EU), in order to increase its performance by carrying out memory accesses and instruction executions in parallel. The AU'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 EU's function is to interpret instructions of the CPU's machine code (which is stack based).

The instruction set has two addressing modes: stack, and immediate, and 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 one of eight conditions codes and select either direct or program counter relative addressing.

```
program prodcom;
  output produce,consume;
  internal req,ack;

  processtype Producer(request,acknowledge,produce);
    input request;
    output acknowledge=false,produce=false;
    loop
      while(!request) do loop skip endloop;
      raise(produce); lower(produce);
      raise(acknowledge);
      while(request) do loop skip endloop;
      lower(acknowledge)
    endloop
  endtype

  processtype Consumer(acknowledge,request,consume);
    input acknowledge;
    output request=false,consume=false;
    loop
      raise(request);
      while(!acknowledge) do loop skip endloop;
      raise(consume); lower(consume);
      lower(request);
      while(acknowledge) do loop skip endloop
    endloop
  endtype

  process producer1: Producer(req,ack,produce);
  process consumer1: Consumer(ack,req,consume)
endprog
```

Figure 2: Producer-consumer program

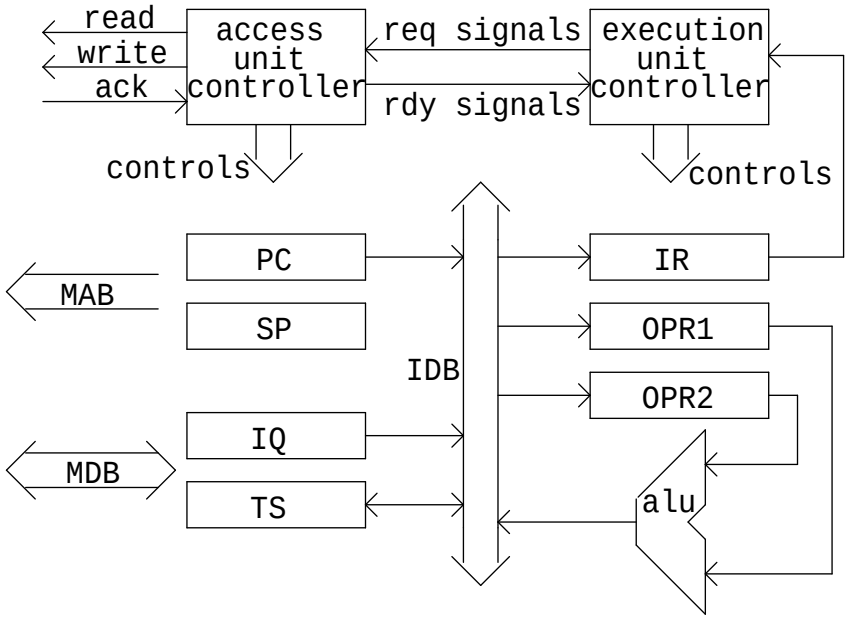

Figure 3: CPU block diagram

The access unit has four registers: the program counter (PC), the stack pointer (SP), the instruction queue (IQ), which can hold two instruction words, 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 RTL functions are summarized in table 1. These definitions will be of interest later when we discuss the formal specification of the controller.

The execution unit has two operand registers (OPR1 and OPR2), an instruction register (IR), a condition code register (CCR), and an ALU. There is a 16-bit internal data bus (IDB) by which data are communicated between the EU and AU. The function of the ALU and the signals which control the execution unit data path will not be described here for the sake of brevity.

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 execution unit signals its intention to perform a push, pop or (instruction) fetch operation

Signal	Function
fetch	$PC \leftarrow PC + 1 \qquad (fetch = fetch\text{-}req \wedge fetch\text{-}rdy)$
PC-MAB	*MAB* (memory address bus) $\leftarrow PC$
PC-IDB	*IDB* (internal data bus) $\leftarrow PC$
branch	$PC \leftarrow IDB$
push	$SP \leftarrow SP - 1 \qquad (push = push\text{-}req \wedge push\text{-}rdy)$
pop	$SP \leftarrow SP + 1 \qquad (pop = pop\text{-}req \wedge pop\text{-}rdy)$
SP-MAB	$MAB \leftarrow SP$
MDB-IQ	$IQ \leftarrow MDB$
IQ-IDB	$IDB \leftarrow IQ$
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

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. The AU first asserts one of the memory control signals (*mem-rd* for a read, and *mem-wr* for a write), and 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 (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 Implementing the controllers

In this section, we give an informal specification of the access unit controller and describe some of the CSML code. The AU controller has two

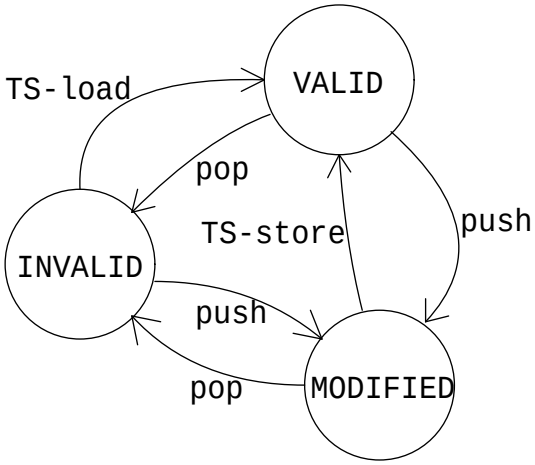

Figure 4: TS manager state diagram

```
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 5: Code for TS manager

functions, which it performs conceptually in parallel: the management of the instruction queue and the management of the top-of-stack cache. We will examine the latter function in some detail. We distinguish 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, the AU is not ready for a push operation when the TS is MODIFIED, because previously pushed data would be lost, and it is not ready for a pop operation when the TS is INVALID, because incorrect data would be read. Figure 4 gives an abstract state diagram which defines the effects of the AU controller operations on the TS register state. This serves as our model of the data path when designing the controller. The CSML code in figure 5 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.

In its other capacity as instruction queue manager, the AU 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. We will not discuss this function in detail.

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. 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 6. Note that when the TS register is in the INVALID state, we allow a push request to take priority over a TS-load operation (line `*`), but once the TS-load operation is started, we lower `push-rdy` to prevent push operations from interfering with the memory cycle (line `**`). 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 7. Since the calls to `read` appear inside compress statements, only the while loop actually takes time.

The overall structure of the AU controller code is a three-way *parallel* statement as shown in figure 8. The job of the execution unit is more straight-

```
    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 6: 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 7: Routine `read`.

forward. 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. When compiled as separate modules, the AU and EU controllers have 13 and 98 states respectively.

```
process AU;
  ... declarations ...
  ... procedures ...
  parallel
  ... memory access monitor ...
  ||
  ... TS manager ...
  ||
  ... IQ manager ...
 endparallel
end proc
```

Figure 8: Overall structure of AU controller code.

5.3 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. Before proceeding we define a few predicates which will simplify the specifications and the following discussion:

$$\begin{aligned} \textit{push} &\equiv \textit{push-req} \wedge \textit{push-rdy} \\ \textit{pop} &\equiv \textit{pop-req} \wedge \textit{pop-rdy} \end{aligned}$$

$$\begin{aligned} \textit{TS-load} &\equiv \textit{mem-rd} \wedge \textit{SP-MAB} \wedge \textit{MDB-TS} \\ \textit{TS-load-done} &\equiv \textit{TS-load} \wedge \textit{mem-ack} \\ \textit{TS-store} &\equiv \textit{mem-wr} \wedge \textit{SP-MAB} \wedge \textit{TS-MDB} \\ \textit{TS-store-done} &\equiv \textit{TS-store} \wedge \textit{mem-ack} \end{aligned}$$

The predicate *push* indicates that a data word is being pushed onto the stack from the internal data bus. Likewise, *pop* indicates that a data word is being popped off the stack. *TS-load* is true when a memory cycle is in progress which is loading the TS. It indicates that the stack pointer contents are being driven onto the memory address bus (*SP-MAB*), and that the data on the memory data bus are being gated into the TS register (*MDB-TS*).

TS-load-done is true on the last clock cycle of such a memory cycle (when *mem-ack* is asserted). In a similar fashion, *TS-store* is true when a memory cycle is in progress which is storing the TS value into memory, and *TS-store-done* indicates the last clock cycle of the TS store operation.

The conditions for correct management of the TS manager are derived from the state transition diagram of figure 4. If the TS is 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). This state thus gives rise to no correctness conditions. In the MODIFIED state, however, we cannot allow another *push* operation, or a *TS-load* operation to occur before either a *pop* or *TS-store* is completed. This condition is expressed by the following formula:

$$MODIFIED \equiv \forall[\neg(push \vee TS\text{-}load)\mathcal{W}(pop \vee TS\text{-}store\text{-}done)]$$

where $\mathcal{W}$ denotes the *weak until* operator, which does not require eventual occurence of its second argument. Since the MODIFIED state is entered if and only if a *push* operation occurs, we specify the following formula:

$$\forall G(push \rightarrow \forall X(MODIFIED))$$

In the INVALID state, *pop* or *TS-store* must not occur before either a *push* or *TS-load* is completed. We express this condition in CTL as

$$INVALID \equiv \forall[\neg(pop \vee TS\text{-}store)\mathcal{W}(TS\text{-}load\text{-}done \vee push)]$$

Since the INVALID state is entered if and only if a *pop* operation occurs, we specify the following:

$$\forall G(pop \rightarrow \forall X(INVALID))$$

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 stack 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 a *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 *TS-load* and *TS-store* cycles, the stack pointer not change. These requirements are expressed in the following formulas:

$$\forall G(\mathit{TS\text{-}load} \rightarrow \forall(\mathit{TS\text{-}load} \wedge \neg(\mathit{push} \vee \mathit{pop})\,\mathcal{U}\,\mathit{TS\text{-}load\text{-}done})),$$
$$\forall G(\mathit{TS\text{-}store} \rightarrow \forall((\mathit{TS\text{-}store} \wedge \neg(\mathit{push} \vee \mathit{pop}))\,\mathcal{U}\,\mathit{TS\text{-}store\text{-}done})).$$

Six more formulas, which we omit here, define correct management of the instruction queue. The following two formulas state that no spurious memory accesses occur.

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

All of 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:

$$\forall G \forall F \mathit{fetch}.$$

5.4 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. After compiling the CSML code, we use a separate utility to compute the parallel composition of the EU and AU modules (along with a small module which models the behavior of the memory system). Since this parallel composition 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 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 original 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 substantial reduction is that, while the EU interprets a large number of instructions, the memory accesses for these instructions fall into a few basic patterns. For this reason, little of the complexity of the EU is observable via the signals connecting it to the AU. We view this as a principle of good interface design: that interfaces should reveal as little of the complexity of the underlying modules as possible. Design according to this principle will reduce the global effects of local changes in the design, and simplify the verification process. The interface rule provides a way of quantifying this effect in terms of the number of states in the interface processes. 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, illustrating 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, during a branch, the EU did not check to make sure that a *IQ-load* operation was not in progress before modifying the PC. This caused the address on the MAB to change during a memory cycle. The second bug 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 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 [11]. More research is needed to handle this problem within our current framework.

Clearly, 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. As an example, 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 critical path timing of modules can be verified independently. Nonetheless, a language with Mealy machine semantics might be more useful for practical designs.

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. We believe, however, that it will be possible to use the model checker to verify temporal formulas over complex models, which can then be used as lemmas in simple hand-constructed proofs.

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 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.

[8] 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.

[9] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of Fourth Symposium on Logic in Computer Science*, 1989. To appear.

[10] D. Harel. Statecharts: A visual approach to complex systems. Technical Report CS84-05, The Weizmann Institute of Science, February 1984.

[11] R. P. Kurshan. Reducibility in analysis of coordination. In *LNCIS*, volume 103, pages 19–39. Springer-Verlag, 1987.

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

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