IN-60
61479
P-58

NASA Contractor Report 187504

# Towards Composition of Verified Hardware Devices

*E. Thomas Schubert*
*K. Levitt*
*University of California*
*Davis, California*

*G. C. Cohen*
*Boeing Military Airplanes*
*Seattle, Washington*

*NASA Contract NAS1-18586*
*November 1991*

National Aeronautics and
Space Administration
**Langley Research Center**
Hampton, Virginia 23665

# PREFACE

This document was generated in support of NASA Contract NAS1-18586, Design and Verification of Digital Flight Control Systems Suitable for Fly-By-Wire Application, Task Assignment 3. Task 3 is associated with formal verification of embedded systems.

Computers are being used in areas where no affordable level of testing is adequate. Safety and life-critical systems must find a replacement for exhaustive testing to guarantee their correctness. Through a mathematical proof, hardware verification can formally demonstrate that a design satisfies its specification. However, hardware verification research has focused on device verification and has largely ignored system composition verification. To address these deficiencies, we examine how the current hardware verification methodology can be extended to verify complete systems.

The NASA technical monitor for this work is Sally Johnson of the NASA Langley Research Center, Hampton, Virginia.

The work was done at Boeing Military Airplanes, Seattle, Washington and the University of California, Davis, California. Personnel responsible for this work include:

Boeing Military Airplanes:

D. Gangsaas, responsible manager

T. M. Richardson, program manager

G. C. Cohen, principal investigator

University of California:

Dr. K. N. Levitt, chief researcher

E. Thomas Schubert, PhD candidate

## TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

# 1.0 INTRODUCTION

This research is directed towards developing a methodology to verify a hardware base for a safety critical system. Such systems are composed of many interacting devices. The top level hardware specification is apt to suggest a unitary implementation. While this abstraction is convenient for verifying the correctness of software executing on the hardware base, the implementation consists of many different interacting components (central processing unit, memory, coprocessors, input/output devices, bus controllers, interrupt controllers, etc.)

Previous approaches to system verification have formed *vertically verified systems*. These efforts have aimed at illustrating how hardware verification can be used to close the semantic gap between high level languages and the computer's instruction set. Systems are divided into layers that can be verified independently. Each layer consists of an implementation and a more abstract specification. Implementation layers serve as a specification for the next lower level (ref. 1). In this way, higher levels of the stack define new functionality by composing the next lower level's functionality. The base for these systems (a microprocessor-memory pair) has been an unrealistic hardware platform.

This research focuses on the development of verification techniques which support *horizontal* integration of devices. In addition to demonstrating horizontal integration verification methods, the research will contribute new abstraction techniques, integrate differing proof techniques, and extend the interpreter model to describe communicating devices.[1] The interpreter theory (refs. 2 and 3) is well suited to the problem of verifying the correctness of a single device, but does not provide an adequate model to reason about composed systems. A complementary technique will be developed so that device intercommunication properties can be verified.

We consider three device composition problems which current hardware verification methodologies do not adequately address: shared state, time granularity (temporal abstraction) and synchronization/communication specification problems. To investigate verified device integration we have examined three examples of composition: A CPU with a

[1]To describe the extensions, the term *communicating interpreters* has been coined.

memory management unit; a CPU with a floating point coprocessor; and a CPU with a "sampling" peripheral device. These examples will be described in subsequent sections.

Support chips we plan to incorporate include an interrupt controller and a direct memory access (DMA) device. These devices are being verified as independent processors by other members of the verification group. We would also like to verify and integrate other coprocessors (UART, network interface device). In general, communication between these devices is through a bus controller which must also be verified. The potential interference problems, mentioned above, must be addressed.

These devices can modify shared state in ways which make the verification of the integrated system difficult. For example, the meaning of the load instruction is that at abstract time *t+1* the value in some register is the same as the value in the memory location at time *t*. Consider interactions at a finer grain of time. Suppose during a load instruction of the interrupt controller's status register the interrupt controller changes the status register value (which is forwarded to the CPU). The interrupt controller providing the latest update (post *t*) would violate the semantics of the load instruction.

As another example, consider a DMA specification. Abstractly we would like to say that the effect of a DMA read operation is to change a region of memory. This specification states that the effect of its operation is to transfer some amount of data from a peripheral device (e.g., secondary storage) to some location in memory and generate an interrupt when the transfer is complete. This operation will often require sufficient time so that some other device could modify the specified memory region before the operation is complete—this would violate the abstract DMA read specification.

These examples suggest that composing a system with high-level abstract specifications of device operations is not always possible. The verification of the system inherits from the separate proofs a set of assumptions which must be enforced at a higher level.

# 2.0 BACKGROUND

## 2.1 RELIABLE SYSTEMS

Life-critical systems are becoming increasingly dependent on computer systems. The need for verified systems is quite real. To cite a few recently published accounts:

a. AT&T suffered significant losses in January, 1990 when a software problem caused large parts of their long-distance telephone switching network to fail (ref. 4).

b. A "fly-by-wire" Airbus A320 aircraft in Bangalore, India crashed in February, 1990. A definitive cause for the accident has not yet been established. Preliminary investigations suggest that the control computers might well have performed to specifications. However, the fact that A320 employs a fully electronic cockpit with no mechanical backup for control has raised much speculation about the computers' role in the crash (ref. 5).

c. During congressional hearings last year, it was established that incorrect software caused an x-ray machine to emit lethal doses of x-rays.

d. The 80386 has had several releases to correct for errors which were undetected before shipment. Thirty one errors were found in the first revision (A1 stepping) ranging from alignment problems to instructions performing incorrectly. Many of these errors permitted protection violations. The second revision (B0 stepping) had twelve additional errors. Fourteen errors were found in the third revision (B1 stepping) including a well publicized multiplication error. This failure is extremely pattern sensitive and not exhibited in all B1 CPUs (ref. 6).

Faults resulting from design errors are especially difficult to protect against and can compromise critical functionality (ref. 7). There are two main techniques to ensure reliability: fault tolerance and fault exclusion (ref. 2). Fault tolerance is concerned with designing systems to recognize and handle faults when they occur. Through redundant components, fault-tolerant systems present an illusion of ultrareliability. These systems do not exclude errors due to specification or implementation flaws. Building reliable systems out of unreliable components does not guarantee a safe and secure system. Fault exclusion however, attempts to exclude design faults. Simulation is the most frequently used method to exclude faults. While simulation may discover the presence of errors, it cannot guarantee the absence of errors. Hardware verification can be used to uncover all design errors.

## 2.2 SECURE SYSTEMS

A significant body of work exists that covers the software necessary for secure systems (refs. 8 and 9). Operating systems must provide efficient and reliable protection features. Not only must an operating system protect against malicious software, but also against unintentionally errant programs. Using security kernels (reference monitors) is the most frequently used technique to build a highly secure operating system (see section 2.2.1).

Secure operating systems are designed based upon an abstract model defining security requirements. To prove a system secure, it is necessary to show that there is no information leak and that the operating system implements the model. The lower level hardware base has not received as much attention. Yet protection flaws in the hardware can easily be exploited by programs which circumvent the operating system.

Most hardware mechanisms necessary to implement security are also required by conventional operating systems. User programs must be inhibited from accessing system resources without prior authorization. This requirement can be satisfied by systems with memory protection, protection rings (or modes), and restricted I/O instructions.

The need for hardware design verification is recognized by the DoD Trusted Computer System Evaluation Criteria, however, hardware design verification has been informal and incomplete to date. The Honeywell SCOMP (Secure Communications Processor) operating system is the first system to have been classified A1 (highest assurance) by the National Computer Security Center. The verification effort concentrated on showing that the operating system software enforced a Bell-LaPadula security model. The informal hardware verification approach has two problems: 1) an incomplete coverage of design (and implementation) analysis and testing; and 2) an incomplete formal top-level specification of the hardware functions visible at the trusted computing base (TCB) interface (ref. 10). Guttman proposes extending the SCOMP verification to lower levels (abstract hardware specification) where several components (process table, page table, CPU, memory management unit) all interact. This work would discharge the assumptions made by the software, complementing the SCOMP verification (ref. 11).

### 2.2.1 SECURITY KERNEL

A security kernel isolates all security mechanisms in a small distinct software layer between the hardware and the conventional operating system. A security kernel must rely on the hardware for several functions.

As efficiency is critical, hardware protection support is essential. Certainly the operating system software cannot check each instruction before allowing it to execute. The security kernel requires both process support and memory protection support from the hardware.

A security kernel must satisfy three fundamental requirements:

a. completeness.

b. isolation.

c. verifiability.

For the security kernel to satisfy the completeness requirement, user processes must be unable to bypass the reference monitor. The reference monitor must be invoked before any subject references an object. Objects include files, memory and I/O buffers which operating systems typically manage. Other types of information also exist (file names, directory structures, status registers, system use data). These data must also be described in the reference monitor's security policy specification.

The completeness requirement imposes the following demands on the underlying hardware.

a. The kernel must trust the hardware to make checks during the execution of untrusted programs. All references to memory, registers and I/O subsystems must be checked.

b. Processes must be isolated from one another.

c. All communication between processes must be mediated by the kernel.

The isolation property requires that the security monitor must be tamperproof. The operating system must be able to protect itself from accidental (or otherwise) attempts to accesses. Hardware memory management must prevent user accesses to kernel code or data. Processor instructions that the kernel uses to provide memory management must be privileged—executable only by the kernel.

The hardware design must also be verified. The kernel and trusted software rely on a system's hardware and some of the kernel mechanisms are directly provided by the hardware itself. Design flaws in the hardware protection mechanism may become visible to processes running on top of the kernel—making the implementation of a reference monitor impossible.

## 2.3 HARDWARE VERIFICATION

Hardware verification requires that the design of a system is formally shown to satisfy its specification through a mathematical proof. Using theorem proving techniques, an expression describing the behavior of a device is proven to be equivalent in some sense to an expression describing the implementation structure of the device. These expressions concisely describe the behavior of devices in an unambiguous way. An additional benefit of hardware verification is that the behavioral semantics of the hardware are clearly defined. This provides an accurate basis for building correct software systems (ref. 1).

Verification is expensive and requires a substantial amount of time. While all development efforts would benefit from the use of formal methods, presently only the verification of life-critical and security properties merits the expense. Tools such as HOL are still under development. The theory libraries are not yet sufficient for general use. Further, techniques and methodologies to verify large systems are not available.

The HOL system is described in appendix A. Circuits and devices are described in HOL using a mixture of functions and predicates. Universally quantified variables are used to specify input and output device lines while internal device lines are existentially quantified. The specifications are generally defined to model a state transition system. A specification defines the state and environment at time *t+1*, as a function of the state and environment at time *t*.

## 2.4 RELATED WORK

Newman proposes a unified hierarchy that accommodates all critical requirements (ref. 12). Responsibility to satisfy each requirement can then be delegated to an appropriate layer of the design. The layers remain interdependent; the more abstract layers relying on the correctness of the lower levels. Formal proofs about the hardware level discharge some of the assumptions made by higher, software levels. Similarly, hardware level proofs often make assumptions about the behavior of the software which are discharged when the level is composed (ref. 11).

Hardware verification efforts thus far have focused primarily on a microprocessor as the base for computer systems (refs. 2, 13, 14 and 15). Perhaps the best known verification effort is that of the VIPER microprocessor (refs. 13, 16 and 17). VIPER is the first microprocessor intended for commercial distribution where a formal verification has been attempted. However, these processors are quite limited. The processors verified have modeled small instruction sets and, generally, have not included modern CPU features such as pipelines, multipled functional units, and hardware interrupt support. Tamarack-3 (ref. 2) and AVM-1 (ref. 3) do provide sufficient interrupt support to connect with an interrupt controller. However, no system currently verified provides the memory management functions necessary to support a secure operating system.

Previous efforts to verify systems have attempted to construct vertically verified systems with a microprocessor/memory as the system base. Joyce has verified a compiler level whose target machine description is the specification of the verified Tamarack-3 microprocessor (ref. 18). Computational Logic Inc. has attempted to verify a "stack" of interpreters where the implementation of a level is the specification of the next lower level (ref. 1). The "stack" consists of a compiler (Micro-Gypsy) an assembler and linking loader, an operating system and a microprocessor. The operating system KIT, is not actually a true part of the stack. The KIT project designed and verified an operating system which supports multiple processes and asynchronous I/O. User processes are able to communicate only through message passing which is implemented by the kernel. This ensures that tasks are isolated from one another. However, the hardware base has not been designed nor verified. Bevier assumes extensions to the FM8502 microprocessor (ref. 15) to provide interrupts, asynchronous I/O, memory management and supervisor-mode instructions.

## 2.5 MEMORY MANAGEMENT UNITS

The memory management unit (MMU) is a critical component within a computer system necessary to ensure security. The MMU may be a subcomponent of the CPU (e.g., Intel 80386), but it may also be a separate processor (e.g., Motorola 68851). The memory management function can prevent an executing process from accessing or modifying main memory locations which were not allocated to the executing process. An extended memory management scheme such as LOCK, also protects secondary storage from undesired manipulation.

The MMU enforces security constraints by validating each CPU request to memory. When a request is outside of the bounds allocated to an executing process, the MMU alerts the CPU of the memory violation. The operating system kernel software can then terminate the process.

The MMU also supports dynamic code relocation and virtual memory. From the perspective of each process, memory is a contiguous sequence of addresses. However, for efficient memory management, a multiple process operating system will divide each process memory space into pieces (segments, pages). Each piece can be placed in noncontiguous locations within the real memory. During execution, the MMU translates a process requested virtual address into a real address.

The area accessible to a process is defined by a table present in memory which is cached by the MMU. The operating system kernel (executing on a verified CPU) is responsible for configuring these tables in an appropriate manner. For example, the kernel should not permit a user process to access this table.

We have verified two simple memory management units (ref. 19). The verification demonstrates the use of hierarchical decomposition and abstract theories. Both devices authorize memory requests and translate virtual addresses to real addresses. The first unit is designed and verified to the gate level. The second memory management unit is implemented with an abstract representation and provides greater operating system support. Memory requests are validated based on a memory resident segment table.

The design permits operation in two different modes: user (process) mode and supervisor mode. In supervisor mode, the MMU passes each request through to memory without validation. In user mode, the MMU performs a table lookup to determine if the request is legitimate. Each address consists of a segment identifier and a segment offset component; the segment identifier serves as the index into the memory resident (segment) table. Each segment table entry will define the availability, access rights (read-only, read-write, executable), length, and the physical address in memory for the segment. Assuming the request satisfies the segment table constraints, the MMU validates and constructs a physical address for the request.

The operating system (executing in supervisor mode) can construct a unique table in memory for each process. Before a process executes, the MMU must be informed of where in physical memory the table is located (segment table pointer register).

The addition of an MMU to a memory system modifies the behavior of the memory extensively. In user mode, the MMU presents an abstract view of memory to a process where memory is divided into segments of varying size and with protection unique to the segment. Note that this abstraction is presented to the CPU/operating system. However, the operating system, in turn, may present some other abstraction of memory to executing processes.

To enhance performance, a device to cache table entries is being designed. A further enhancement will add greater support for task switching. The current MMU design requires the cache to be flushed each time the segment table pointer register is changed. This significantly reduces the cache performance. Performance can be improved by adding a set of segment table pointers to the MMU and appending a process identifier to cached values. The process identifier corresponds to one of the segment table pointers. When one of the segment table pointers is modified, only those cached entries with a matching process identifier would be flushed.

## 3.0 ABSTRACTION

Abstraction is a central concept within computer science as well as mathematics. It also plays a fundamental role in hardware verification. Hardware verification can be defined as formally demonstrating that a hardware design satisfies a more abstract specification describing its behavior. The abstract specification suppresses irrelevant detail or information so that the description focuses only on the items of interest.

For example, a microcoded microprocessor specification would describe the effect of each instruction on the user visible state of the machine (registers, memory). The implementation state space is significantly larger and includes many details not pertinent to the assembly language programmer (e.g., microcode instruction pointer, memory address register, memory data register, buffer registers, latches, etc.). To demonstrate the design is correct, the microinstruction sequence corresponding to each instruction must be shown to correctly modify the user visible state.

Techniques for proving the correctness of hardware designs use abstraction mechanisms for relating formal descriptions at different levels of detail. Four such abstraction mechanisms and their formalization in higher order logic are discussed in (ref. 20). These include *structural*, *behavioral*, *data* and *temporal* abstraction.

The description of a device's implementation will contain explicit information about its structure while the specification of a device should reflect only its externally observable behavior. Structural abstraction provides a means of denoting what information is internal.

Specifications frequently only partially describe a device's behavior. The behavior in certain states or for certain input values is left unspecified. When a device will never have to operate in these states or for those inputs, the specification of its behavior is unnecessary. Behavioral abstraction relates a partial specification to an implementation that fully describes the device's behavior.

Data abstraction constructs a mapping function relating concrete data types to abstract data types. This mapping is then used to show that the operations carried out on low level data types correctly implement the desired operations on the high level types.

Temporal abstraction mechanisms are used to map between different 'grains' of discrete time. Relating the levels of temporal abstraction involves mapping points or periods of low level time to points or periods of high level time and showing that a many-step low level computation implements a one-step high level computation.

## 3.1 INTERPRETERS

The general interpreter model can be used to describe state transition systems. Interpreters can be differentiated from other models by their monolithic treatment of state and their flat control structure. The interpreter selects one of a fixed set of actions based upon the current state and returns a new state. The model incorporates four parts:

- A representation of the state **S**.
- A set of state transition functions defining the denotational semantics for each interpreter action (instruction) $\mathbf{J_i} : \mathbf{S_i} \times \mathbf{Env} \rightarrow \mathbf{S_j}$.
- A next state function which selects an appropriate state transition function given the current state $\mathbf{K} : \mathbf{S_i} \rightarrow \mathbf{J_i}$.
- A function **I**, relating the state at time $t+1$ to the state and environment at time $t$ using **K** and **J**.

$$\mathbf{I}[s, e] \equiv s(t+1) = (\mathbf{K}(s\,t))(s\,t)(e\,t)$$

Correctness is a relation between a specification and implementing interpreter such that:

$$\mathbf{I}_{impl}[s_m, e_m] \Longrightarrow \mathbf{I}_{spec}[s_n o f, e_n o f]$$

where $f$ is a temporal abstraction function (see section 3.4) relating the differing specification and implementation time granularities. The state space $s_n$ and environment $e_n$ are abstractions of the implementing state space $s_m$ and environment $e_m$, respectively.

## 3.2 HIERARCHICAL DECOMPOSITION

Systems can be decomposed into a linear hierarchy of abstract machines where each abstract machine is implemented by the next lower level machine (ref. 21). Each level depends only on the function provided by the next lower level. The system is verified by proving the implementation correctness between all levels. This can greatly reduce the verification effort as each level can assume the lower levels have been verified.

Previous hardware verification efforts have attempted to show that the structural description of a device directly implies the top level behavioral description. For example, the Royal Signals and Radar Establishment (RSRE) had initially inserted a major state level between the electronic block model and the top level. However, Cohn discarded this intermediate level when performing the formal verification. Proving the correctness of VIPER became an enormous effort and was not completed. Using case analysis, approximately 120 independent large theorems had to be proved (one for each instruction). Each theorem must reason about the structure in similar ways. For example, instructions and operands are fetched in the same manner for each instruction. This commonality was not exploited.

Windley proposed adding several intermediate abstract behavioral levels in a hierarchy to take advantage of the similarity between cases (ref. 22). Rather than the verification showing that $\mathbf{S} \Longrightarrow \mathbf{B}$, several increasingly abstract specifications of the system's behavior can be defined and the system's correctness proved by showing:

$$\mathbf{S} \Rightarrow \mathbf{B_1} \Rightarrow \ldots \Rightarrow \mathbf{B_n}.$$

The amount of effort necessary to verify a microprocessor is substantially reduced by using hierarchical decomposition. A hierarchical microprocessor verification effort can be structured as shown in figure 3.2-1 from (ref. 3). The macro level specification defines the behavioral specification as viewed by the programmer. A microcode interpreter and phase (or subcycle) behavior level have been inserted between the top level specification and the implementation (here the electronic block model).

The difficult proofs are between the electronic block model and the phase level where there are a small number of instruction cases (two to four). This reduces the overall effort by at least an order of magnitude.

The abstract behavior levels are defined as interpreters. Their similar structure allows the proofs between levels to be regular with most of the work being automatable.

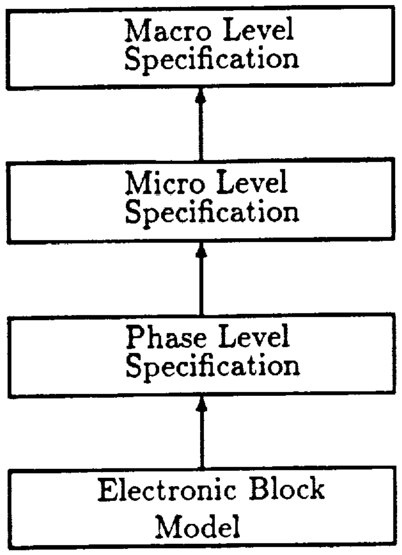

Macro Level Specification
Micro Level Specification
Phase Level Specification
Electronic Block Model

*Figure 3.2-1: A Hierarchical Microprocessor Specification*

## 3.3 GENERIC THEORIES

Generic theories are described in greater detail in (ref. 3). A generic theory consists of three parts:

a. An ***abstract representation*** of the uninterpreted constants and types in the theory. The ***abstract representation*** contains a set of abstract operations and a set of abstract objects. The semantics of the abstract representation are unspecified. Inside the theory, we don't know what the objects and operations mean.

b. A list of ***theory obligation*** predicates defining relationships between members of the abstract representation. When a theory is instantiated, these predicates must be proven about the concrete representation. Within the theory, the obligations represent axiomatic knowledge. The abstract MMU theory does not contain any theory obligations.

c. A collection of ***abstract theorems*** about the representation.

Using the abstract theory package, a set of selector functions can be created. When applied to an abstract representation, a selector function extracts the desired function.

For example, instead of dealing with concrete data types such as bitVectors with a specific length, the abstract MMU works with data values of abstract types **wordn*, **address* and **memory*. The abstract representation provides a set of functions which manipulate these types.

## 3.4 TEMPORAL ABSTRACTION

Temporal abstraction plays a significant role in this research. An implementation design typically gives more detail about how a device behaves over time than defined in its abstract specification which concentrates on "interesting" points of time. An abstraction function suppresses the implementation time points not of interest to the specification. These functions create a coarser view of time from a finer grain of discrete time.

The grain of time at the implementing level may have no direct relation to real time. The intervals may correspond to the changing of values on clock lines. However, if the clock stopped for a period of time, there would be no points of fine grain time during that interval.

For each step of high (coarse) level time, a temporal abstraction function specifies a corresponding step in low level time. In figure 3.4-1, $t_0, \ldots, t_4$ represent time points relevant to an abstract behavioral description. Points $t'_0, \ldots, t_{10}$, represent time at a lower implementing level. The function $f$ maps $t_0$ to $t'_0$, $t_1$ to $t'_3$, etc.

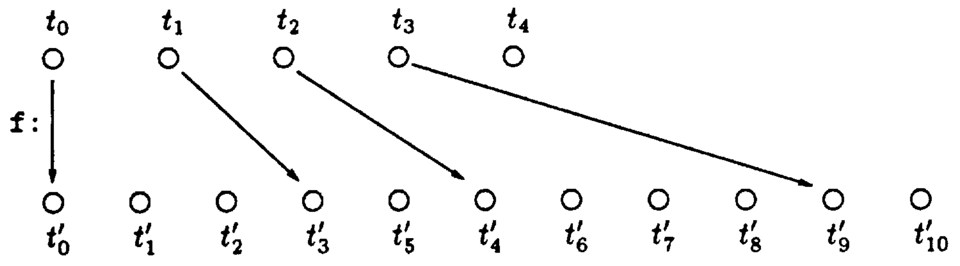


*Figure 3.4-1: Temporal Abstraction Mapping Function*

The mapping function must be increasing so that $f(t_i) > f(t_{i+1})$ for all $i$. Note that the number of low level ticks consumed for a single high level tick need not be constant or regular.

The increasing abstraction function can be used to formulate correctness statements. Let signals *sig* and *sig'* of type *:num-bool* correspond to signals at the abstract and implementing levels respectively. If $f$ is a temporal abstraction function, then:

$\forall t.\ sig(t) = sig'(f(t))$ or $sig = sig' \circ f$

Given an implementation **Imp** requiring two signals ($a$ and $b$) a specification **Spec**, and a temporal abstraction function[2] $f$, a correctness statement can be defined:

$\forall a\ b. \mathbf{Imp}(a,b) \Longrightarrow \mathbf{Spec}(a \circ \mathbf{f},\ b \circ \mathbf{f})$

The function **f** defines the sequence of low level time points that lie between each unit of high level time. The correctness statement is proved by showing that if the intermediate values of $a$ and $b$ are allowed by **Imp**, then the values of $a$ and $b$ at the abstracted time points will be allowed by **Spec**.

It is convenient to define the temporal abstraction function in terms of the lower level time.

a. The time to synchronize the two time sequences can be more easily defined in terms of the state at the lower level time. For example, the micro and macro levels of a CPU are linked when the microprogram counter has returned to the beginning of its fetch-macroinstruction phase.

b. If the correctness statement is expressed in terms of the lower level time, then several layers of structural and temporal abstraction can be linked together.

c. A proof of a correctness statement will reason at the low level time scale.

We can construct a boolean function which, given a low level time, returns true when a "map-up" occurs. Using this function **p**, a function **TimeOf** can be defined such that

[2]It is of course, not necessary that there be only one temporal abstraction function in a correctness statement

**TimeOf** **p** *n* denotes the nth time **p** is true.

If **TimeOf** is a partial function, it cannot be defined directly in the logic. **TimeOf** would be partial if there exists some *n* after which **p** is no longer true. Assuming that **p** is true at an infinite number of points, **TimeOf** is total, and **f** can be as:

$\forall$t. $f(t) = Timeof\,p\,t$ or $f = TimeOf\,p$

Two useful generic temporal abstraction predicates are **First** and **Next**. **f** can be defined using these predicates.

```
⊢def First g t =   (∀ p:time. p < t ⇒ ¬ (g p))  ∧  (g t)

⊢def Next g (t1,t2) =  (t1 < t2)  ∧
        (∀ t:time . t1 < t  ∧  t < t2 ⇒ ¬ (g t))  ∧  (g t2)
```

**First** is true when its argument *t* is the first time that *g* is true. **Next** is true when *t2* is the next time after *t1* that *g* is true. The predicate `stable_sigs` states that between *t1* and *t2* the MMU inputs will remain constant.

The definition of **f** uses Hibert's choice operator $\varepsilon$. Given a predicate $P$, $\varepsilon x.P(x)$ represents a value satisfying $P$.

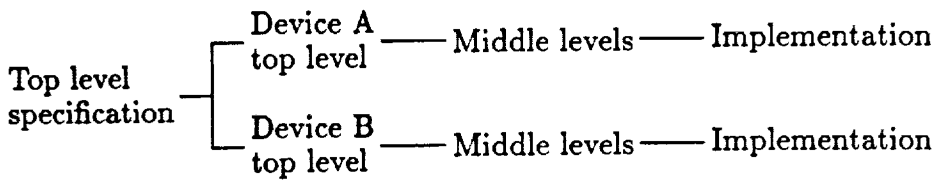
```
⊢def (f g 0 =  ε t:time. First g t)  ∧
       (f g (SUC n) = ε t:time. Next g ( (f g n), t))
```

Other forms of temporal abstraction will be necessary to prove the correctness of composed devices (see section 4.0).

# 4.0 VERIFYING COMPOSED DEVICES

Hardware verification efforts have not yet investigated horizontal integration of devices. The aim of this research is to verify a system that presents a top level interpreter describing a unified system. The top level presents a single processor view of the system while the implementation consists of many devices working in a cooperative manner. The top level abstraction conceals a great deal (perhaps all) of any concurrency in the implementation. The KIT effort characterizes the kind of assertions an operating system verification might wish to make about an underlying hardware verification and the LOCK SIDEARM serves as an example architecture which we wish to verify. In addition to a standard instruction set, the system should provide virtual memory, protected I/O, interrupts (including a timer interrupt), and security rings. The system must guarantee that a kernel could prevent a user or process from accessing these features and that executing user processes are isolated from each other.

Real systems are substantially more complex than the devices currently verified. However, a useful framework for device verification has been developed from previous efforts to verify central processing units. The interpreter hierarchy (see figure 3.2-1) demonstrates how a microprocessor specification can be decomposed into a series of implementing interpreters. Intermediate levels describing the phase and microinstruction layers are inserted between the implementation and top level specifications to facilitate the overall proof effort. This technique can also be used to verify the correctness of other system components. Unfortunately, verification of a system requires more than composition of the top level specifications.



The increasing abstraction levels make the effort of understanding a specification significantly easier. However, a substantial amount of information is lost. Much of this information is necessary to show that devices work together. Systems not only aggregate the function of the individual devices, but also formulate new function. The top level specification suppresses too much detail which is needed to reason about the composition of devices. To verify the devices work together will require properties proved about the joined intermediate or implementation levels. For example, to show that a system consisting of a floating point unit, a CPU, and a memory work together correctly, theorems must be derived to show:

a. The CPU correctly executes CPU instructions.

b. The FPU correctly executes FPU instructions.

c. The devices work together correctly. These devices pass information using a low level handshaking protocol which would not be described in the top level specifications.

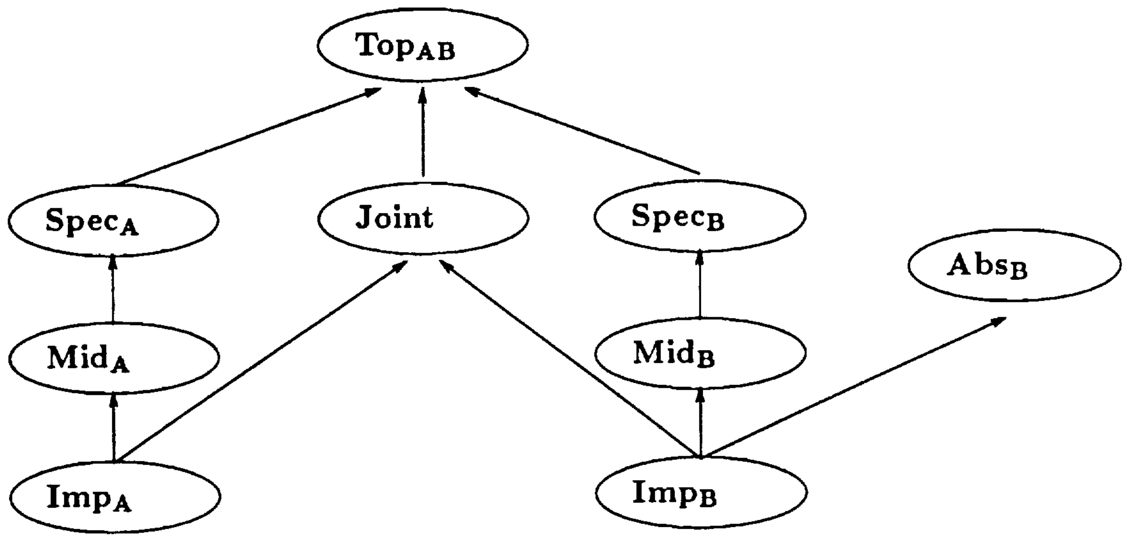


*Figure 4.0-1: Composition Lattice*

More generally, systems can be modeled as a lattice consisting of specifications and implementations (see figure 4.0-1). What makes composition interesting is the extent to which concurrency enters the picture. The models formulated previously were not designed to handle this concurrency. The interpreter hierarchy model doesn't extend to incorporate interdevice communication. A system may be decomposed into several subsystems and devices. Devices can be further decomposed and verified using current techniques such as the generic interpreter theory. When device specifications are combined, the interpreters must communicate with one another. From the implementation definitions we can derive the low level protocols used to communicate with other devices and prove that the devices can synchronize with one another to pass messages. From higher levels we can derive when these messages would be sent. This would be used to show that two devices employ a four-phase handshaking protocol when generating memory requests.[3]

Note that many abstractions based on an implementation are possible.[4] For example, the VIPER microprocessor verification effort defined a major state abstraction level between the top level and the implementation. Cohn chose to abandon the major state abstraction level as it appeared too difficult to verify it in relation to the top level (ref. 17).

Verification of a system of composed devices can be partitioned into three requirements:

a. The top level specification of each device must be correct.

b. Composed interpreters must not violate assumptions made by other interpreters about shared state.

c. The links between components must be shown to imply the correct synchronous behavior.

The first requirement can be satisfied by appropriate hierarchical decomposition strategies such as the generic interpreter theory. However, eventual solutions to the other requirements may impose new obligations on device verification that simplify the overall system proof effort.

[3]For a read request, a device using this protocol would: (1) raise the read line and address lines; (2) wait for dtack to rise and then read the data lines; (3) drop the read line; and (4) wait for dtack to fall and then drop the address lines.

[4]This is the reason why the proofs are implicative.

Windley suggests a possible solution to the shared state requirement (ref. 3). Specifications must describe memory actions with an abstract transformation function. A memory unit is then defined as:

```
⊢def MEMORY_UNIT write read memory address port =
    (read t ⟹ (port = fetch(memory,address))) ∧
    (write t → (memory(t+1) = store(memory t, address, port))
              | (memory(t+1) = trans (memory t))
```

If the **read** signal is true at time **t**, port carries the value of **memory** at **address**. If the **write** signal is true at time **t**, then **address** of **memory** is updated with the **port** value. Otherwise, the abstract function **trans**, prescribes whether **memory** is updated. The **trans** function aggregates all changes that are made by other devices during this time interval.

The shared state problem is put off to a device composition step. The uninterpreted transformation functions in both devices must then be instantiated with appropriate values. This proposal requires that a transformation function be specified for every abstraction level of the device. Additional temporal issues between levels arise as the relationship between transformation functions of different levels must be described. For example, the transformation function on the state at the micro level is a composition of the smaller transformations performed at the phase level. This places a large burden on the proof as the different functions for each level must all be discharged. Windley suggests that some of the lower levels need not always appear at the lower levels and can be introduced in the state abstraction function connecting layers (ref. 3). This technique is used in the specification of *AVM-1*.

The *AVM-1* instruction correctness lemma assumes that, during a memory fetch, no other device changes the requested location and, during a memory write, devices do not compete for access to the same memory location. These requirements are too restrictive when applied at the macro instruction level. It would not be unreasonable for an I/O device to change its status during a CPU instruction which fetches the status value.

We are pursuing an alternative solution which extends the interpreter model notion of environment. Devices treat shared locations as part of the environment rather than the state. The interpreter model does not exhibit an output environment and the treatment of an input environment can be developed further.

The third requirement involving device synchronization issues is further developed in the sections which follow. To reason about the communication between devices, the process algebras CSP and CCS are being considered. CCS appears to be clearer and easier to implement, however, CSP has recently been mechanized in HOL (ref. 23). Section 4.4.2 presents an example of how CCS may be used to describe device interaction.

Depending on the context, one type of logic is frequently more appropriate than another to describe some property. By employing several logics together the advantages of each logic can be utilized. As the different logics will be based in HOL, it is possible to reason about their combination. Researchers have proposed that several logics can be effectively expressed in HOL (refs. 2 and 24).

Section 4.2 describes an initial test case which revealed several difficulties in specifying composed systems. Section 4.3 will examine the concurrency introduced by CPU pipelining. This view can be generalized to reason about prefetch capabilities and a CPU consisting of several multiple functional units.

To compose devices we construct appropriate temporal and behavioral abstractions. The final section will present two examples of these abstractions. The first example demonstrates the temporal abstraction needed for composing coprocessors. This abstraction is also applicable to a system with a programmable I/O device such as a DMA. The second example demonstrates how a CCS notation can be used to describe how devices work together correctly.

The terms implementing level and abstract level will distinguish between two specification levels—one *relatively* more concrete than the other.

## 4.1 CPU-MMU-MEMORY SYSTEM VERIFICATION

Composing independent processors which share state (e.g., memory, peripheral control registers) raises difficulties. The proofs for each device make legitimate assumptions about the effects of device operations. These assumptions simplify the proof and without the full context in which the device is used, there is little more that the specification can express.

The independent MMU and CPU proofs define memory as providing state which determines their actions (the segment table and instruction stream, respectively). Verification of a processor often requires that its operation affects the state in a deterministic manner. For example, a CPU will expect a store instruction to change the contents of a single memory location. The semantics would also suggest that all other memory locations remain unchanged. When the state is shared, this may not be true.

In user mode, a single write to the segment table pointer register (assuming it's accessible) changes which table is used to construct real addresses from virtual addresses. This has the effect of changing the (perceived) contents of potentially all memory locations. This is inconsistent with the CPU's independent proof.

From the perspective of the CPU, the MMU cache is transparent—although memory accesses are sometimes delayed when the MMU must load a new entry. Both the segment table pointer register and the segment table(s) are memory locations read (but not modified) by the MMU. The MMU cache is modified and the behavior of memory is changed, but there is no mutually modifiable shared state. If the MMU were enhanced to manipulate a dirty segment/page bit on processor writes, these two devices would modify shared state.

The CPU and MMU act in a similar manner to concurrent processes with critical sections. A critical section is a period of execution when a process must have exclusive access to some shared resource (ref. 25). In the context of CPU-MMU interactions, a critical section is entered when either device begins to modify memory. Either low level hardware or the specification of the paired devices must guarantee that during critical sections, devices do not interfere.

## 4.2 SAMPLER-CPU

A preliminary analysis of a system combining a sampling I/O device and AVM-1, revealed several specification and verification issues. The sampling device is hypothetical, but suggests a simplified keyboard control device.

The sampler interface is through memory-mapped registers (control, status, data). The control register may be set so that the sampler is inactive, polled, or interrupt enabled. If the control register is programmed to accept new inputs, an arriving sample is placed in the data register. The status register reflects whether a value is available in the data register or if an error has occurred and is reset whenever the control register value is modified. It is the responsibility of the software (or CPU microcode) to initialize, fetch and reset the sampler appropriately.

The system specification extends the AVM-1 instruction set to include a new "instruction" (an operating system trap) which retrieves a new sample value from the input environment.[5] When the sample trap is serviced, the CPU switches to supervisor mode (allowing access to I/O space) and a short assembly language program executes and returns (in user mode) with the new sampler value in some register. We can denote this behavior as:

$$instruction\ t = TRAP_{sampler} \Longrightarrow reg_i(t+1) = sample\ t$$

The communication is performed at a finer grain of time than the top level specification. There is a delay between when the assembler routine starts and when a sample is retrieved. Figure 4.2-1 shows an example where the above expressed semantics are not satisfied. The sample arrives after $t$, but before the assembler program requests a value.

To resolve this problem, we can redefine the sampler. The new sampler would store a sample value in a latch each time the CPU began an execution cycle. This value would be written to the data register only when the appropriate trap instruction executes. This

[5]In the multilevel machine structure described by Tanenbaum, this new interpreter is a subset of the operating system machine level—above the conventional machine level and below the assembly language level (ref. 26).

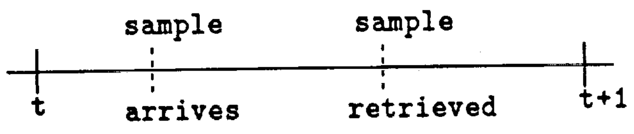


*Figure 4.2-1: Sampling Time Granularity*

simpler unit is certainly not realistic and we would like to verify the composition of the original device. This example motivated the work described in section 4.4.

## 4.3 PIPELINES

Figure 4.3-1 shows the behavioral abstraction utilized in the verification of the Mini Cayuga pipeline. Mini Cayuga is a three stage instruction pipelined RISC processor (ref. 27). Instructions pass through fetch, compute and writeback pipeline stages. The top line presents the abstracted view while the bottom line presents the implementing level view. At $t_0$, $t_1$, and $t_2$ instructions enter the pipeline.

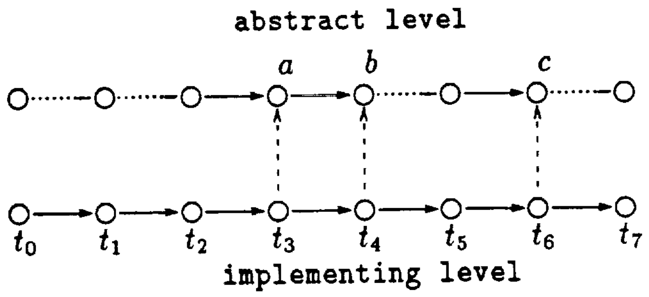


*Figure 4.3-1: Temporal Abstraction for Mini Cayuga Pipeline*

The abstract level specification declares that the effect of an instruction occurs completely in the last stage of the pipeline when values are written back to the register file. These points are labeled $a$,$b$ and $c$. The processor is described as doing work only during an interval when some instruction is in its last stage. During other periods, the CPU is considered idle. The number of cycles needed to complete an instruction varies from one instruction to another. When a memory fetch is required, the pipeline stalls until the fetch has completed (e.g., at $t_4$).

Rather than collapse these idle periods into work periods, the Mini Cayuga verification retains these periods. This appears to have been done so that the top level specification can describe the delayed branching effect caused by pipelining. If branching instructions were defined semantically as "jump after the next instruction," it would be unnecessary to describe the prefetch behavior at the abstract level.

This effort contrasts with the work described below. We collapse these idle periods and define significant events as the point where a new instruction is begun.

## 4.4 COPROCESSORS

A coprocessor is a device which shares the task of executing instructions with the CPU. Coprocessors may also provide additional functionality to the system (e.g., a memory management unit) or may only extend the system's instruction set (e.g., a floating point coprocessor). When coprocessors are present, the CPU continues to fetch all instructions, but defers to the coprocessor to execute certain instructions. The CPU may explicitly route instructions to the coprocessor (e.g., 68020, 80387) or the coprocessor may fetch instructions directly from the CPU instruction stream (e.g., 8087). These cases may be classified as "snooping" or "nonsnooping" respectively. In either case, the abstraction presented to the user is a unified system—that instruction execution is performed by two distinct units is hidden. The abstraction conceals the complex synchronization which occurs between the CPU and its coprocessors.

Some coprocessor instructions may be treated as no-ops by the CPU while others may require work by both the coprocessor and the CPU. The CPU may be required to calculate memory addresses, initiate memory operations, transfer register values or respond to a coprocessor's status flags. In the "nonsnooping" case, the CPU must also send the instruction to the coprocessor. Many of these functions could be added to the coprocessor. However, as the CPU independently requires these functional capabilities to operate, it is more space efficient not to duplicate these functions.

The CPU controls instruction sequencing and, thus, must monitor the coprocessor to synchronize actions before moving onto the next instruction. Synchronization can be achieved by the CPU either polling a coprocessor status register or reading a busy-line.

The amount of work the CPU must perform for most coprocessor instructions is quite small compared to the work the coprocessor performs to execute an instruction. This leaves the CPU idle for a potentially long period if the CPU must wait for the coprocessor to complete its task. Frequently, a significant number of instructions can be inserted between instructions destined for the same coprocessor. To enhance performance, modern coprocessor designs permit the CPU to execute instructions in parallel with the coprocessor. Either hardware or software must ensure that the concurrent instructions do not interfere with one another. The presence of instruction prefetch or CPU on-chip caches add significant complications. This will be discussed in greater detail below.

### 4.4.1 68000 COPROCESSORS

This example demonstrates a useful temporal abstraction method which supports the composition of devices.

The Motorola 68000 supports a general interface for coprocessors to receive instructions from the CPU (ref. 28). Functionality is divided so that the CPU does not have to decode coprocessor instructions and the coprocessors do not duplicate CPU functions such as address calculations.

Each coprocessor is expected to provide a set of registers (mapped to memory locations) which are used for communication.[6] The coprocessor interface includes command, response, condition, and operand registers. When the CPU fetches a coprocessor instruction, a dialogue with the appropriate coprocessor begins. The CPU writes the instruction to the command CIR register for the coprocessor to decode. The CPU then busy-waits; checking the response CIR until the coprocessor indicates either:

a. it is free to execute the next instruction.

b. an operand read or write is required.

c. or an exception has occurred.

[6]These registers are not directly available to the programmer.

While the coprocessor interface is generic, in the following discussion, we will describe the interaction with a floating point coprocessor (FPU). The description would equally apply to the memory management unit (MC68851). The MC68851 normally does not allow concurrent instruction processing since most MMU instructions change the system configuration. However, the coprocessor dialogue will take place as the MC68851 may request an effective address calculation. It is also possible that an exception will be generated during the coprocessor dialogue (e.g., a page fault).

Once communication is completed, the CPU is free to execute the next instruction. The coprocessor may continue to execute on its own while the CPU executes the next instruction. Coprocessor instructions can take much longer to complete than CPU instruction. For example a floating point coprocessor instruction may require the equivalent of several CPU instruction cycles to complete. Thus, several instructions may be executed concurrently. The parallelism may not be presented to the programmer who views the system as executing instructions sequentially. This introduces an interesting verification problem. The verification effort must show that the concurrent implementation can imply a sequential execution abstraction.

To ensure the correct execution of an instruction stream, some mechanism must be in place to ensure noninterference. If the CPU continues to execute instructions after the floating point unit has begun execution, the CPU must not reference a memory location specified in the floating point instruction. Also, assuming the floating point coprocessor can execute only one instruction at a time, the CPU must not initiate the execution of a second floating point instruction until the first has completed.

The CPU must ensure that the instruction stream does not interfere with itself. While this abstraction may seem a bit contrived, propagating the concurrency to the top level increases the difficulty of proving programs correct.

The abstraction presented can be extended to describe other situations:

a. A system with programmable I/O processors or a DMA device.

b. A CPU with multiple functional units where instructions are scheduled on several different computation units and execute in parallel.

c. A multiprocessor system where a task is divided into several parallel communicating subtasks.

#### 4.4.1.1 The System Level Abstraction

With the inclusion of a floating point coprocessor, the system provides an instruction set which contains both the CPU instruction set as well as the FPU instruction set. The register set available contains all CPU and FPU registers. At the abstraction level of the programmer, the system appears as a single unit. The programmer need not be aware that there are two distinct devices sharing the task of executing instructions. Further, at this level of abstraction, instructions appear to be executed in a sequential manner. That is, the programmer may believe that instruction $i+1$ does not begin executing until instruction $i$ has completed.

In the underlying implementation, however, the CPU may initiate the execution of a floating point instruction and then begin execution of the next instruction before the floating point unit has finished. In fact, if the floating point instruction is followed by a sequence of instructions which do not require either the FPU or the result of the initial floating point instruction, several of the instructions may complete before the FPU instruction calculates a result. This parallel execution of instructions utilizes the CPU and FPU more efficiently than the abstract specification required.

#### 4.4.1.2 Noninterference

The CPU and FPU can interfere with one another if either:

a. The CPU initiates execution of an FPU instruction when the FPU is busy executing a previous instruction.

b. The CPU attempts to read or write to the target memory location of the currently executing FPU instruction.

When the 68020 detects a coprocessor instruction, it initiates a coprocessor dialogue. If the coprocessor is already executing an instruction, the coprocessor will not complete the dialogue, causing the CPU to idle. When the coprocessor becomes free, synchronization can complete and the CPU will proceed to the fetch and execute the next instruction. This enforces the requirement that the CPU not initiate the execution of a second coprocessor instruction prematurely.

If the target of an FPU operation is a memory location, the second interference problem remains only if the CPU is released before the result is stored. The implementation does not complete the coprocessor dialogue until the result is stored. Note that the memory management coprocessor will not release the CPU until after a state change has completed. Thus, the second potential interference problem is not present.

#### 4.4.1.3 Verification

To prove the CPU-FPU system implements the system level abstraction, we must show that:

a. The independent processors work as specified.

b. The low level synchronization and message passing works correctly.

c. The devices don't interfere with one another.

The verification of the independent processors can be achieved in a manner similar to previous efforts. Passing information between memory locations or devices can be performed using a four-phase handshaking protocol as described in (ref. 29). With the exception of memory, the CPU and FPU do not share state. Memory is only changed when both the CPU and FPU cooperate to pass a value. The CPU top level interpreter is invalid only if memory were to change at a time the CPU cannot predict.

At the programmer abstraction level, each instruction completes before the subsequent instruction begins. However, the execution of a single coprocessor operation may be overlapped with the parallel execution of several CPU instructions. This contradicts the interpreter model. The interpreter model describes the resulting state based on the execution of a single instruction. This view presents an interpreter where the state at $t+1$ modifies the user visible state only as permitted by the instruction selected at $t$.

### 4.4.1.4 Concurrent to Sequential Abstraction

When the devices are composed together, the composite interpreter displays concurrency. However, given the noninterference property described above, this concurrency is an implementation detail which can be abstracted away from the programmer level. This is unlike the 8086/8087 composition where the programmer must insert guard instructions to ensure the noninterference (see section 4.4.2).

To demonstrate the abstraction technique, we will use the following possible execution sequence of instructions:

$$f_1,\ c_1,\ c_2,\ c_3,\ f_2$$

Each $f_i$ is a FPU instruction and each $c_i$ is a CPU instruction. This sequence generalizes to any sequence beginning with a floating point instruction, continuing with one or more CPU instructions and followed by a floating point operation. Instructions which transfer values from floating point registers to memory or CPU registers are classified as floating point instructions.

Figure 4.4-1 shows the independent instruction streams and the joint instruction stream at the implementing (concurrent) level. "Clock" ticks denote the points when new instructions begin to execute. It is at these clock tick times when we can describe the state of the machine. Each instruction is labeled by a start time ($c_i$) and a completion time ($\bar{c}_i$). Several possible state histories exist depending on when $f_1$ completes execution.

At $t_1$, the CPU fetches $f_1$ and begins a coprocessor dialogue (we ignore the delay between the fetch and initiation of the dialogue). The coprocessor does not release the CPU until $t_2$, when the dialogue has completed, at which point the CPU fetches $c_2$. $f_1$

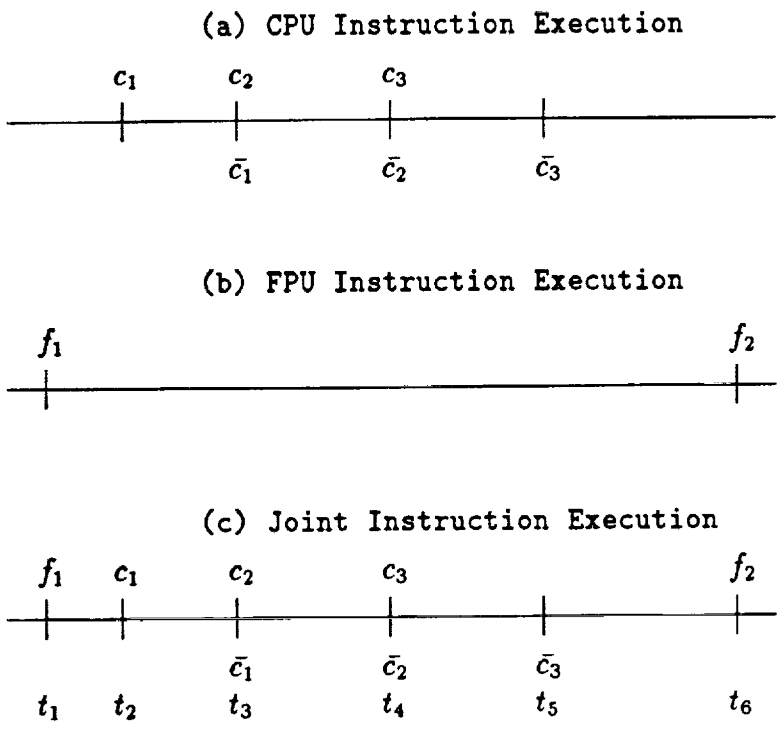

(a) CPU Instruction Execution
$c_1$ $c_2$ $c_3$
$\bar{c}_1$ $\bar{c}_2$ $\bar{c}_3$
(b) FPU Instruction Execution
$f_1$ $f_2$
(c) Joint Instruction Execution
$f_1$ $c_1$ $c_2$ $c_3$ $f_2$
$\bar{c}_1$ $\bar{c}_2$ $\bar{c}_3$
$t_1$ $t_2$ $t_3$ $t_4$ $t_5$ $t_6$

*Figure 4.4-1: CPU and FPU Concurrent Instruction Execution*

may finish anywhere between $t_2$ and $t_6$ (see figure 4.4-1c). At $t_5$, $c_3$ has completed and the CPU fetches $f_2$. The coprocessor dialogue begins, but the coprocessor will not be ready to execute $f_2$ until completing execution of $f_1$ (perhaps at $t_6$). Any time the CPU (or coprocessor) must wait can be collapsed into the previous instruction execution cycle. Thus, $t_5$ can be omitted with $c_3$ "finishing" at $t_6$.

At the higher programmer (sequential) abstraction level, the state consists of the CPU state, the FPU state, and the shared memory state. The coprocessor cannot modify the shared memory state without the assistance of the CPU. Thus, memory can be treated as part of the CPU state. The implementation guarantees that the states of the two devices are distinct. The CPU is not able to modify the FPU state nor is the FPU able to modify the FPU state.

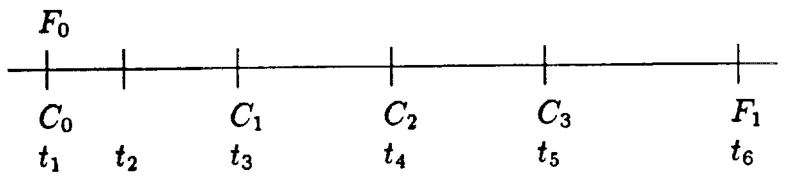


*Figure 4.4-2: CPU and FPU States*

In figure 4.4-2, $C_i$ and $F_i$ represent the CPU state and FPU state after instruction $i$. Figure 4.4-2 exhibits the distinct states relative to the time scale of figure 4.4-1. At $t_1$ the CPU is in state $C_0$ while the FPU is in state $F_0$. At the implementing level, we can construct a state pair history which represents the actual state at the indexed points:

$$\begin{array}{ccccc} C_0 & C_1 & C_2 & C_3 & C_3 \\ F_0 & F_0 & F_0 & F_0 & F_1 \end{array}$$

Abstractly we want the tuple history to reflect the sequence of instructions executed:

$$\begin{array}{ccccc} C_0 & C_0 & C_1 & C_2 & C_3 \\ F_0 & F_1 & F_1 & F_1 & F_1 \end{array}$$

The dialogue implementation guarantees that the CPU (and programmer) can not observe any FPU state change until it has completed.[7] It doesn't matter when the instruction actually completed, so the abstract view suggests it completed immediately after the coprocessor dialogue. The sequential top level interpreter then chooses to update the state pair based on whether the next instruction is a CPU or coprocessor instruction.

[7]The *Paul Mason* principle: "We will see no state until its time."

### 4.4.2 8087 COPROCESSOR

The interaction between the Intel 8086/8, the 8087 floating point coprocessor and other support chips demonstrates the complex communication which occurs between devices (ref. 30). In this section we will describe the communication between devices and describe a set of CCS *agents* which reflect the communication essential for correct system behavior. Appendix B presents a brief overview of CCS inference rules and expression construction.

The Intel 8086/8 instruction set contains several instructions for coprocessor support. The ESC instruction includes a coprocessor opCode field which is ignored by the 8086/8. The CPU may perform some work depending on the other fields. The CPU may calculate operand addresses and initiate memory fetches and stores. If multiple word memory operations are required, the coprocessor initiates subsequent transfers where the address calculation amounts to an increment operation.

To maximize concurrency, FPU and CPU instructions may be overlapped. The FPU must complete execution of each instruction before starting a new instruction. The program is obligated to synchronize the CPU and FPU. A WAIT instruction can be inserted between floating point operations to force the CPU to wait until a busy line is low. Rather than receiving its instructions from the CPU, the 8087 monitors all information passing to the 8086 (or 8088) CPU and selects instructions from the bus. For the two devices to work together correctly, they must coordinate several details:

a. 8086 or 8088 interface.

b. Instruction tracking.

c. Bus control.

d. Processor synchronization.

e. Exceptions.

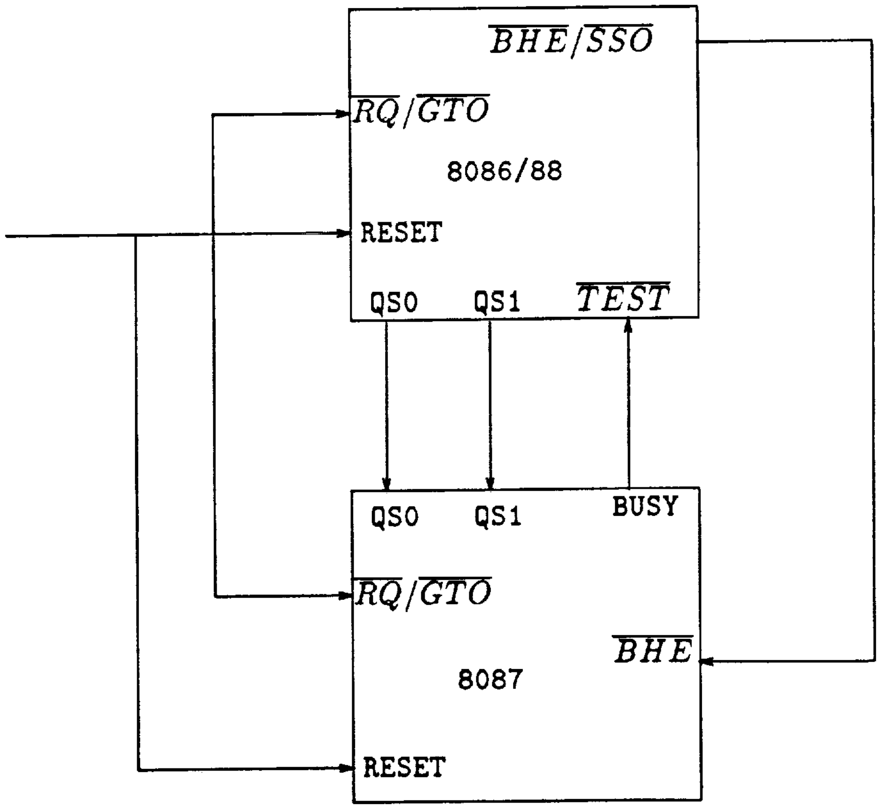
#### 4.4.2.1 8086 or 8088 Interface

The 8087 was designed to work with either the 16-bit data bus 8086 or the 8-bit data bus 8088. The 8087 detects which of the two processors is present at startup time. Upon system initialization, the RESET line becomes high and the CPU fetches a value from memory location FFFF0H. An 8086 will set $\overline{\texttt{BHE}}$ low while the 8088 will set the output from the same pin (labeled $\overline{\texttt{SSO}}$) high. By observing the RESET and $\overline{\texttt{BHE}}/\overline{\texttt{SSO}}$ lines, the 8087 can determine which CPU is present (see figure 4.4-3). These requirements can be modelled using the following CCS expressions:

$\texttt{8087} =_{def} reset.\texttt{Reset87} + \ldots$

$\texttt{Reset87} =_{def} hiBHE.\texttt{86Interface} + lowBHE.\texttt{88Interface}$

$\texttt{8086} =_{def} reset.\overline{hiBHE}.\texttt{Reset86} + \ldots$



$\overline{BHE}/\overline{SSO}$
$\overline{RQ}/\overline{GTO}$
8086/88
RESET
QS0
QS1
$\overline{TEST}$
QS0
QS1
BUSY
$\overline{RQ}/\overline{GTO}$
$\overline{BHE}$
8087
RESET

*Figure 4.4-3: 8086/8 – 8087 Interface*

### 4.4.2.2 Instruction tracking.

Many coprocessor instructions require that both processors perform portions of the work simultaneously. The two processors execute based on the same clock signal from an 8284 clock generator. The 8087 monitors all bus traffic to the CPU and must respond correctly when the CPU executes its part of the coprocessor instruction. The 8086/8 CPU prefetch mechanism and variable length instruction format complicate the synchronization.

Free bus cycles are used to fetch instruction stream bytes into a CPU resident queue. Runtime control flow behavior will determine whether these bytes are interpreted as instructions. Without duplication of the CPU decode and execution logic, the 8087 cannot directly determine which fetched bytes are instructions and which are operands. Finding `ESC` instructions is not straightforward and the 8087 has no way to (internally) determine exactly when the CPU will begin executing a coprocessor instruction.

Three lines from the CPU to the 8288 bus controller ($\overline{S_0}, \overline{S_1}, \overline{S_2}$) are monitored by the 8087 and allow the 8087 to distinguish instruction fetches from other memory operations (see figure 4.4-4).

To coordinate instruction tracking, the 8087 maintains a copy of the instruction queue and duplicates buffer management functions. The CPU informs the coprocessor of how the CPU instruction buffer is used using the `QS0` and `QS1` lines as described in table 4.4-1.

*Table 4.4-1: Abstract Level State Pair History*

| QS0 | QS1 | Condition |
|---|---|---|
| No signal | No signal | No queue change |
| No signal | Signal | Remove first byte and Interpret as instruction |
| Signal | No signal | Flush queue |
| Signal | Signal | Remove first byte |

These requirements are partially modelled using the CCS expressions:

`8086` $=_{def}$ `86Prefetch | 86Decode`

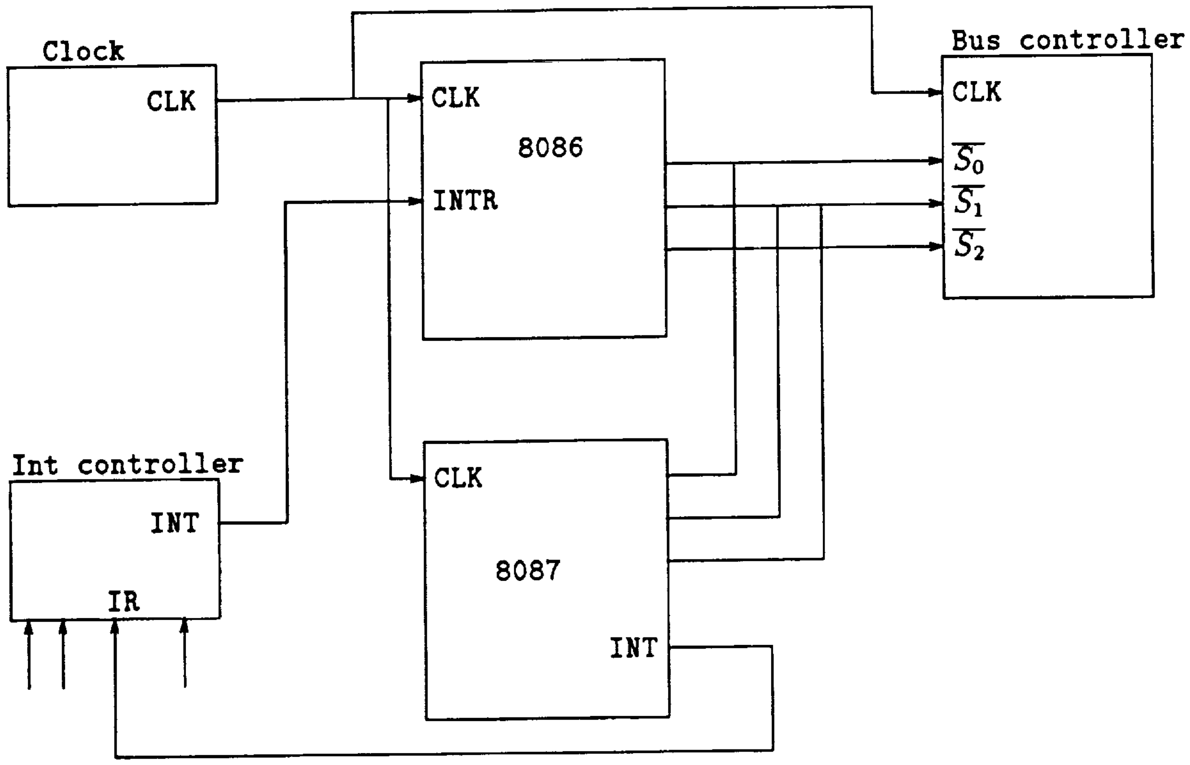


*Figure 4.4-4: 8086/8 System*

86Prefetch $=_{def}$ $\overline{\mathit{fetchCode}}$.GetCode

86Decode $=_{def}$ $\overline{\mathit{noChange}}$.86Buf + $\overline{\mathit{decode}}$.86Buf + $\overline{\mathit{flush}}$.86Buf + $\overline{\mathit{skipByte}}$.86Buf

### 4.4.2.3 Bus Control

When the 8087 requires additional bus cycles to fetch/store multiple byte memory operands, the 8087 must request bus control from the CPU. Using the $\overline{\text{RQ}}/\overline{\text{GT0}}$ line, the 8087 first makes the request, waits for a grant acknowledgment from the CPU, and issues a done acknowledgement after completing its bus activity. These requirements are partially modelled using the CCS expressions:

87NeedBus $=_{def}$ $\overline{\mathit{busReq}}.\mathit{busGrant}$.UseBus.$\overline{\mathit{busDone}}$

8086 $=_{def}$ $\ldots + \mathit{busReq}.\overline{\mathit{busGrant}}.\mathit{busDone}$.8086

#### 4.4.2.4 Processor Synchronization

When the CPU executes a WAIT instruction, it monitors its $\overline{\text{TEST}}$ pin until the value is low. This pin is connected to the 8087 BUSY pin which remains high while a coprocessor instruction executes.

$\texttt{8086} =_{def} \ldots + \mathit{waitInstr}.\mathit{notBusy}.\texttt{8086}$

$\texttt{87Busy} =_{def} \mathit{setBusy}.\mathit{setIdle}.\texttt{87Busy} + \overline{\mathit{notBusy}}.\texttt{87Busy}$

#### 4.4.2.5 Exceptions

Normally the 8087 handles its own exceptions. The 8087 may be programmed to instead assert its interrupt pin. This pin is typically connected to a 8259A interrupt controller which may pass the exception on to the interrupt controller.

$\texttt{PICidle} =_{def} \sum_{i=0}^{7} \mathit{intReq}_i.\texttt{IR}_\texttt{i}$

$\texttt{IR}_\texttt{i} =_{def} \overline{\mathit{intR}}.\texttt{ForwardIR}_\texttt{i}$

$\texttt{8086} =_{def} \ldots + \mathit{intR}.\texttt{GetIReq}$

$\texttt{87Except} =_{def} \overline{\mathit{intReq}_6}.\texttt{8087}$

## 5.0 SUMMARY OF APPROACH

As a preliminary step, we will verify a system consisting of a simple sampling device and a CPU. The sampling device will provide both polling and interrupt-driven I/O interactions. This will provide a mechanism to examine various communication interactions (rendezvous, remote procedure call, message passing). This example will also validate our proposed approach to verifying composed devices.

We will then investigate what is required to verify the composition of the verified MMU with an MMU cache. The verified MMU is inefficient as it must fetch a segment descriptor for each memory request. An initial design of a first-in-first-out (FIFO) list has been specified and designed. A more appropriate specification is being developed which describes the cache as a finite set rather than a list. The FIFO replacement strategy will be replaced with an least recently used or most frequently used replacement scheme. The ability to "lock" values into the cache may also be added.

Both the MMU and the cache will be defined as interpreters. From these interpreters, CCS device communication behaviors will be extracted. The composed abstract specification will show that the MMU subsystem is equivalent in behavior to a system without the cache and performs at least as efficiently.

Extending this example, we plan to demonstrate how a complete system composed of many devices can be shown to correctly implement an abstract system specification.

## 6.0 CONCLUSIONS

Current hardware verification methodologies do not adequately address the demands imposed by composed systems (shared state, time granularity and synchronization specification).

a. Devices treat state as a monolithic unit and assume complete control over its alteration. The state shared by composed devices does not behave as stipulated. To verify composed devices, we must define a composed view of shared state and show that devices don't simultaneously modify their shared state (memory). We propose extending the use of environments. The generic interpreter theory provides a limited output environment and does not utilize its input environment. We are investigating three alternative approaches: 1) Treat all of memory as part of the environment; 2) Define memory state as undefined at certain points; 3) Construct an abstraction allowing memory to "float between being part of state and part of the environment.

b. A new temporal abstraction mechanism has been designed to allow a sequential top level abstraction for a concurrent implementation.

c. To specify multiple device behavior dependencies we propose using CCS to specify the requirements for each device. Each device (interpreter) must be shown to imply the required behavior and the combined behavior must be shown to imply the top level specification.

The generic interpreter model is not sufficiently flexible to meet the composition requirements. The proposed transformation function does not seem to be adequate to resolve the shared state problem. Extending the generic interpreter model theory (ref. 3) to utilize an input and output environment gives promise of solving this problem. By adding a complementary method to the interpreter theory, we hope to create a methodology that takes advantage of the generic interpreter theory and solves the device synchronization verification problem. Rather than devise a new concurrency theory, we are investigating the integration of CCS with the interpreter theory.

By adding a complementary method to the interpreter theory, we will create a methodology that takes advantage of the generic interpreter theory and solves the device synchronization verification problem. Rather than devise a new concurrency theory, we are pursuing the integration of CCS with the interpreter theory.

## REFERENCES

1. W. R. Bevier, W. A. Hunt, and W. D. Young, "Toward Verified Execution Environments," *IEEE Symposium on Security and Privacy*, 1987.

2. J. J. Joyce, *Multi-Level Verification of Microprocessor-Based Systems.* PhD thesis, Cambridge University, December 1989.

3. P. J. Windley, *The Formal Verification of Generic Interpreters.* PhD thesis, University of California, Davis, 1990.

4. K. Fitzgerald, "Vulnerability exposed in AT&T's 9-hour glitch," *The Institute*, March 1990.

5. A. D. Singh and S. Murugesan, "Fault-Tolerant Systems," *Computer*, July 1990.

6. J. L. Turley, *Advanced 80386 Programming Techniques.* Osborne McGraw-Hill, 1988.

7. V. P. Nelson, "Fault-Tolerant Computing: Fundamental Concepts," *Computer*, July 1990.

8. M. Glasser, *Building A Secure Computer.* Van Nostrand Reinhold Company, 1988.

9. D. Denning, *Cryptography and Data Security.* Addision-Wesley, 1982.

10. V. D. Gilgor, "Analysis of the Hardware Verification of the Honeywell SCOMP," *IEEE Symposium on Research in Security and Privacy*, 1985.

11. J. D. Guttman and H. Ko, "Verifying A Hardware Security Architecture," *IEEE Symposium on Research in Security and Privacy*, 1990.

12. P. G. Neumann, "On Hierarchical Design of Computer Systems for Critical Applications," *IEEE Transaction on Software Engineering*, vol. SE-12, No. 9, September 1986.

13. A. Cohn, "A Proof of Correctness of the VIPER Microprocessor: the First Level," in *VLSI Specification, Verification, and Synthesis*, (G. Birtwhistle and P. Subrahmanyam, eds.), Kluwer Academic Press, 1988.

14. W. A. Hunt, "A Verified Microprocessor," technical report 47, The University of Texas at Austin, Dec. 1985.

15. W. A. Hunt, "Microprocessor Design Verification," *Journal of Automated Reasoning*, vol. 5, 1989.

16. W. J. Cullyer, "Implementing Safety Critical Systems: The VIPER Microprocessor," in *VLSI Specification, Verification, and Synthesis*, (G. Birtwhistle and P. Subrahmanyam, eds.), Kluwer Academic Press, 1988.

17. A. Cohn, "A Proof of Correctness of the VIPER Microprocessor: the Second Level," in *Current Trends in Hardware Verification and Automated Theorem Proving*, (G. Birtwhistle and P. Subrahmanyam, eds.), Springer-Verlag, 1989.

18. J. J. Joyce, "Totally Verified Systems: Linking Verified Software to Verified Hardware," *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, July 1989.

19. E. T. Schubert, "Verification of Memory Management Units using HOL," technical report CSE-90-27, University of California, Davis, August 1990.

20. T. Melham, "Abstraction Mechanisms for Hardware Verification," in *VLSI Specification, Verification, and Synthesis*, (G. Birtwhistle and P. Subrahmanyam, eds.), Kluwer Academic Press, 1988.

21. E. W. Dijkstra, "The Structure of THE—Multiprogramming System," *Communications of the ACM*, vol. 11, May 1968.

22. P. J. Windley, "A Hierarchical Methodology for Verifying Microprogrammed Microprocessors," *IEEE Symposium on Research in Security and Privacy*, 1990.

23. A. J. Camilleri, "Mechanizing CSP Trace Theory in Higher Order Logic," *IEEE Transactions on Software Engineering*, vol. 16, September 1990.

24. M. Gordon, "Mechanizing Programming Logics in Higher Order Logic," in *Current Trends in Hardware Verification and Automated Theorem Proving*, (G. Birtwhistle and P. Subrahmanyam, eds.), Springer-Verlag, 1989.

25. G. R. Andrews and F. B. Schneider, "Concepts and Notations for Concurrent Programming," *Computing Surveys*, vol. 15, March 1983.

26. A. S. Tanenbaum, *Structured Computer Organization.* Prentice Hall, 1976.

27. M. Srivas and M. Bickford, "Formal Verfication of a Pipelined Microprocessor," *IEEE Software*, vol. 7, September 1990.

28. Motorola, *MC 68851 Paged Memory Management Unit User's Manual.* Prentice Hall, 1986.

29. J. Joyce, "Formal Specification and Verification of Asynchronous Processes in Higher-Order Logic," technical report, University of Cambridge, Computer Laboratory, 1988.

30. J. F. Palmer and S. P. Morse, *The 8087 Primer.* John Wiley & Sons, 1989.

31. M. Gordon, "HOL: A Proof Generating System for Higher-Order Logic," in *VLSI Specification, Verification, and Synthesis*, (G. Birtwhistle and P. Subrahmanyam, eds.), Kluwer Academic Press, 1988.

32. A. Camilleri, M. Gordon, and T. Melham, "Hardware Verification using Higher Order Logic," in *From HDL Descriptions to Guaranteed Correct Circuit Designs*, (D. Borrione, ed.), Elsevier Scientific Publishers, 1987.

33. A. Church, "A Formulation of the Simple Theory of Types," *Journal of Symbolic Logic*, vol. 5, 1940.

34. M. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF. Lecture Notes in Computer Science No. 78*, Springer Verlag, 1979.

35. R. L. Constable, *Implementing Mathematics with the NUPRL Proof Development System.* Prentice Hall, 1986.

36. R. Milner, *Communication and Concurrency.* Prentice Hall, 1989.

# APPENDIX A: HOL

HOL is a general theorem proving system developed at the University of Cambridge (refs. 31 and 32) that is based on Church's theory of simple types, or higher order logic (ref. 33). Church developed higher order logic as a foundation for mathematics, but it can be used for describing and reasoning about computational systems of all kinds. Higher order logic is similar to the more familiar predicate logic, but allows quantification over predicates and functions, not just variables, allowing more general systems to be described.

HOL grew out of Robin Milner's LCF theorem prover (ref. 34) and is similar to other LCF progeny such as NUPRL (ref. 35). Because HOL is the theorem proving environment used in the body of this work, we will describe it in more detail.

HOL's proof style can be tailored to the individual user, but most users find it convenient to work in a goal-directed fashion. HOL is a tactic based theorem prover. A tactic breaks a goal into one or more subgoals and provides a justification for the goal reduction in the form of an inference rule. Tactics perform tasks such as induction, rewriting, and case analysis. At the same time, HOL allows forward inference and many proofs are a combination of both forward and backward proof styles. Any theorem proving strategy a user employs in connection with HOL is checked for soundness, eliminating the possibility of incorrect proofs.

HOL provides a metalanguage, ML, for programming and extending the theorem prover. Using ML, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be combined into new theories for later use. The metalanguage makes the HOL verification system extremely flexible.

In HOL, all proofs, even tactic-based proofs, are eventually reduced to the application of inference rules. Most nontrivial proofs require large numbers of inferences. Proofs of large devices such as microprocessors can take many millions of inference steps. In a proof containing millions of steps, what kind of confidence do we have that the proof is correct? One of the most important features of HOL is that it is *secure*, meaning that new theorems can only be created in a controlled manner. HOL is based on five primitive axioms and eight primitive inference rules. All high-level inference rules and tactics do their work through some combination of the primitive inference rules. Because the entire proof can be reduced to one using only eight primitive inference rules and five primitive axioms, an independent proof-checking program could check the proof syntactically.

## A.1 The Language.

The object language of HOL is described in this section. We will discuss HOL's terms and types.

**Terms.** All HOL expressions are made up of terms. There are four kinds of terms in HOL: variables, constants, function applications, and abstractions (lambda expressions). Variables and constants are denoted by any sequence of letters, digits, underlines, and primes starting with a letter. Constants are distinguished in the logic; any identifier that is not a distinguished constant is taken to be a variable. Constants and variables can have any finite arity, not just 0, and, thus ,can represent functions as well.

Function application is denoted by juxtaposition, resulting in a prefix syntax. Thus, a term of the form "`t1 t2`" is an application of the operator `t1` to the operand `t2`. The term's value is the result of applying `t1` to `t2`.

An abstraction denotes a function and has the form "`λ x. t`". An abstraction "`λ x. t`" has two parts: the bound variable `x` and the body of the abstraction `t`. It represents a function, `f`, such that "`f(x) = t`". For example, "`λ y. 2*y`" denotes a function on numbers which doubles its argument.

Constants can belong to two special syntactic classes. Constants of arity 2 can be declared to be infix. Infix operators are written "`rand1 op rand2`" instead of in the usual prefix form: "`op rand1 rand2`". Table A-1 shows several of HOL's built-in infix operators.

Constants can also belong to another special class called binders. A familiar example of a binder is ∀. If c is a binder, then the term "`c x.t`" (where `x` is a variable) is written as shorthand for the term "`c(λ x. t)`". Table A-2 shows several of HOL's built-in binders.

*Table A-1: HOL Infix Operators*

| *Operator* | *Application* | *Meaning* |
|---|---|---|
| = | t1 = t2 | t1 equals t2 |
| , | t1,t2 | the pair t1 and t2 |
| ∧ | t1 ∧ t2 | t1 and t2 |
| ∨ | t1 ∨ t2 | t1 or t2 |
| ⟹ | t1 ⟹ t2 | t1 implies t2 |

*Table A-2: HOL Binders*

| *Binder* | *Application* | *Meaning* |
|---|---|---|
| ∀ | ∀ x. t | for all x, t |
| ∃ | ∃ x. t | there exists an x such that t |
| ε | ε x. t | choose an x such that t is true |

In addition to the infix constants and binders, HOL has a conditional statement that is written a → b | c, meaning "if a, then b, else c."

**Types.** HOL is strongly typed to avoid Russell's paradox and others like it. Russell's paradox occurs in a high order logic when one can define a predicate that leads to a contradiction. Specifically, suppose that we define P as P(x) = ¬x(x) where ¬ denotes negation. P is true when its argument applied to itself is false. Applying P to itself leads to a contradiction since P(P) = ¬P(P) (i.e. , *true* = *false*). This kind of paradox can be prevented by typing since, in a typed system, the type of P would never allow it to be applied to itself.

Every term in HOL is typed according to the following recursive rules:

a. Each constant or variable has a fixed type.

b. If x has type $\alpha$ and t has type $\beta$, the abstraction $\lambda$ x. t has the type $(\alpha \to \beta)$.

c. If t has the type $(\alpha \to \beta)$ and u has the type $\alpha$, the application t u has the type $\beta$.

Types in HOL are built from type variables and type operators. Type variables are denoted by a sequence of asterisks (*) followed by a (possibly empty) sequence of letters

*Table A-3: HOL Type Operators*

| *Operator* | *Arity* | *Meaning* |
|---|---|---|
| `bool` | 0 | booleans |
| `ind` | 0 | individuals |
| `num` | 0 | natural numbers |
| `(*)list` | 1 | lists of type * |
| `(*,**)prod` | 2 | products of * and ** |
| `(*,**)sum` | 2 | coproducts of * and ** |
| `(*,**)fun` | 2 | functions from * to ** |

and digits. Thus, *, ***, and `*ab2` are all valid type variables. All type variables are universally quantified implicitly, yielding type polymorphic expressions.

Type operators construct new types from existing types. Each type operator has a name (denoted by a sequence of letters and digits beginning with a letter) and an arity. If $\sigma_1,\ldots,\sigma_n$ are types and op is a type operator of arity $n$, the $(\sigma_1,\ldots,\sigma_n)$op is a type. Note that type operators are postfix while normal function application is prefix or infix. A type operator of arity 0 is a type constant.

HOL has several built-in types which are listed in table A-3. The type operators `bool`, `ind`, and `fun` are primitive. HOL has a special syntax that allows `(*,**)prod` to be written as `(* # **)`, `(*,**)sum` to be written as `(* + **)`, and `(*,**)fun` to be written as `(* -> **)`.

## A.2 The Proof System.

HOL is not an automated theorem prover but is more than simply a proof checker, falling somewhere between these two extremes. HOL has several features that contribute to its use as a verification environment:

a. Several built-in theories, including booleans, individuals, numbers, products, sums, lists, and trees. These theories contain the five axioms that form the basis of higher order logic as well as a large number of theorems that follow from them.

b. Rules of inference for higher order logic. These rules contain not only the eight basic rules of inference from higher order logic, but also a large body of *derived* inference rules that allow proofs to proceed using larger steps. The HOL system has rules that implement the standard introduction and elimination rules for Predicate Calculus as well as specialized rules for rewriting terms.

c. A collection of tactics. Examples of tactics include: `REWRITE_TAC` which rewrites a goal according to some previously proven theorem or definition; `GEN_TAC` which removes unnecessary universally quantified variables from the front of terms; and `EQ_TAC` which says that to show two things are equivalent, we should show that they imply each other.

d. A proof management system that keeps track of the state of an interactive proof session.

e. A metalanguage, ML, for programming and extending the theorem prover. Using the metalanguage, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be aggregated to form new theories for later use. The metalanguage makes the verification system extremely flexible.

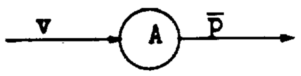
## APPENDIX B: Calculus of Communicating Systems

CCS was designed to model communication and concurrency in complex systems (ref. 36). Systems are composed of several parts acting independently of each other, but communicating with one another to achieve a mutual goal. All elements of the system are modeled as *agents*. The medium used to transmit information between a sender and receiver is also modeled as an agent. An agent's behavior is described by *actions* which may be communications with other agents or independent concurrent actions. Milner suggests that independent actions can also be modeled as (internal) communication.

The calculus provides five primitives to construct expressions describing agents' behavior: **Prefix, Summation, Composition, Restriction and Relabelling.**[8] In the pure calculus, agents do not transmit data values, but synchronize through indivisible actions where a synchronization signal is simultaneously sent by one party and received by another. The summation 'operator' can be applied to duplicate the effect of passing data values between agents. By communicating only through pure synchronizations, the calculus may ignore value variables.

An agent identifies the current state of an entity. Transitions from state to state are accomplished by an action. Actions are denoted by labels and describe either synchronization or internal transitions. Labels are taken from an infinite set of names. For two components to synchronize, they must share the same port name. The notation distinguishes send and receive synchronization labels by placing a bar over the name of send labels (e.g., $\bar{c}$). Internal transitions are labeled by the reserved label $\tau$ which has no complement.

For example, in the figure below, agent $A$ has an input synchronization port (in) and an output synchronization port ($\overline{\texttt{out}}$).



[8]Restriction and relabelling operators have the tightest binding followed by prefix, composition, and summation.

The behavior of the agent might be defined as follows:

$$A =_{def} \overline{p}.A'$$

$$A' =_{def} v.A$$

These agents describe a semaphore and show the use of the **Prefix** ( "." ) operator. Agent $A$ waits to synchronize with some other agent and upon completion, behaves as $A'$.

**Composition** hooks together two independent agents and is denoted with a vertical bar. For example, consider the following agents:

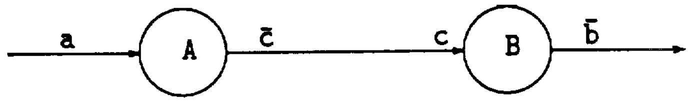
$$A =_{def} a.A'$$

$$A' =_{def} \overline{c}.A$$

$$B =_{def} c.B'$$

$$B' =_{def} \overline{b}.B$$

The composite agent $(A \mid B)$ may be connected as follows:



We can also write $A =_{def} a.A'$ as $A \xrightarrow{a} A'$. The transitional semantics of the language (described below) allow us to infer the following:

a. From $A \xrightarrow{a} A'$, we infer $A \mid B \xrightarrow{a} A' \mid B$.

b. From $A' \xrightarrow{\overline{c}} A$, we infer $A' \mid B \xrightarrow{\overline{c}} A \mid B$ (this represents communication between $A'$ and some agent other than $B$).

c. Since $A' \xrightarrow{\overline{c}} A$ and $B \xrightarrow{c} B'$ we can infer $A' \mid B \xrightarrow{\tau} A \mid B'$ (internal transitions).

d. From $B' \xrightarrow{\overline{b}} B$, we infer $A \mid B' \xrightarrow{\overline{b}} A \mid B$.

By imposing **Restriction** on the composition of $A$ and $B$ (i.e., $A \mid B \backslash c$), the action $c$ is no longer externally available. This implies item (2) would then be excluded.

In the above examples, each agent was defined as conducting only one action. The **Summation** operator $(+)$ is used to define agents which may make one of several (possibly infinite) transitions. For example, an agent similar to $A$ could be defined to either accept input signals or send output signals as follows:

$$C =_{def} a.C + \overline{c}.C$$

**Relabelling** allows agent definitions to be reused. Using relabelling we could use the same agent definition for $A$ and $B$ above. Given a relabelling function $f$, $A[f]$ denotes a copy of agent $A$ with appropriately modified action names.

## B.1 transitional semantics

CCS provides several rules describing the semantics of the expression constructors:

**Act**

$$\frac{}{a.E \xrightarrow{a} E}$$

**Sum$_\mathbf{j}$**

$$\frac{E_j \xrightarrow{a} E_j'}{\sum_{i \in I} E_i \xrightarrow{a} E_j'} (j \in I)$$

**Com$_\mathbf{1}$**

$$\frac{E \xrightarrow{a} E'}{E \mid F \xrightarrow{a} E' \mid F}$$

**Com$_\mathbf{2}$**

$$\frac{F \xrightarrow{a} F'}{E \mid F \xrightarrow{a} E \mid F'}$$

**Com$_\mathbf{3}$**

$$\frac{E \xrightarrow{a} E' \, and \, F \xrightarrow{\overline{a}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$$

**Res**

$$\frac{E \xrightarrow{a} E'}{E\backslash L \xrightarrow{a} E'\backslash L}(a, \bar{a} \ni L)$$

**Rel**

$$\frac{E \xrightarrow{a} E'}{E[f] \xrightarrow{f(a)} E'[f]}$$

**Con**

$$\frac{P \xrightarrow{a} P'}{A \xrightarrow{a} P'}(A =_{def} P)$$

# Report Documentation Page

| | | |
|---|---|---|
| 1. Report No. NASA CR 187504 | 2. Government Accession No. | 3. Recipient's Catalog No. |
| 4. Title and Subtitle: Towards Composition of Verified Hardware Devices | | 5. Report Date: November 1991 |
| | | 6. Performing Organization Code |
| 7. Author(s): E. Thomas Schubert, Dr. K. Levitt, Gerald C. Cohen | | 8. Performing Organization Report No. |
| | | 10. Work Unit No. |
| 9. Performing Organization Name and Address: Boeing Military Airplanes, P.O. Box 3707, M/S 7J-24, Seattle, WA 98124-2207 | | 11. Contract or Grant No. NAS1-18586 |
| | | 13. Type of Report and Period Covered: Contractor Report |
| 12. Sponsoring Agency Name and Address: Langley Technical Monitor: Sally C. Johnson | | 14. Sponsoring Agency Code |

15. Supplementary Notes

16. Abstract

Computers are being used in areas where no affordable level of testing is adequate. Safety and life critical systems must find a replacement for exhaustive testing to guarantee their correctness. Through a mathematical proof, hardware verification can formally demonstrate that a design satisfies its specification. However, hardware verification research has focused on device verification and has largely ignored system composition verification. To address these deficiencies, we examine how the current hardware verification methodology can be extended to verify complete systems.

| | |
|---|---|
| 17. Key Words (Suggested by Author(s)): Hardware Verification, Interpreters, Hierarchical Decomposition, General Theories, Temporal Abstraction | 18. Distribution Statement |

| | | | |
|---|---|---|---|
| 19. Security Classif. (of this report): Unclassified | 20. Security Classif. (of this page): Unclassified | 21. No. of pages | 22. Price |

NASA FORM 1626 OCT 86