NASA Contractor Report 189566

Formal Verification of a Set of Memory Management Units

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

G. C. Cohen
Boeing Military Airplanes
Seattle, Washington

NASA Contract NAS1-18586
March 1992

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 Applications, Task Assignment 3. Task 3 is associated with formal verification of embedded systems. In particular, this document describes the verification of a set of memory management units (MMU). The verification effort demonstrates the use of hierarchical decomposition and abstract theories. The MMUs can be organized into a complexity hierarchy. Each new level in the hierarchy adds a few significant features or modifications to the lower level MMU. The units described included:

a. A page check translation look-aside module (TLM).

b. A page check TLM with supervisor line.

c. A base and bounds MMU.

d. A virtual address translation MMU.

e. A virtual address translation MMU with memory resident segment table.

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

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

Boeing Military Airplanes:
D. Gangsaas, Responsible Manager
T. M. Richardson, Program Manager
G. C. Cohen, Principal Investigator

University of California:
Dr. K. Levitt, Chief Researcher
Mark R. Heckman, PhD. Candidate

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 INTRODUCTION

This report describes the verification of a set of memory management units (MMU). The specification and verification were done using the HOL verification system (ref. 1). The MMUs can be organized into a complexity hierarchy. Each new level in the hierarchy adds a few significant features or modifications to the lower level MMU. The units described include:

a. A page check TLM (translation look-aside module).

b. A page check TLM with supervisor line.

c. A base and bounds MMU.

d. A virtual address translation MMU.

e. A virtual address translation MMU with memory resident segment table.

Life-critical systems are becoming increasingly dependent on computer systems. Though redundant components in fault-tolerant systems increase reliability, 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. Faults resulting from design errors are especially difficult to protect against and can compromise critical functionality (ref. 2). While simulation may discover the presence of errors, it cannot guarantee the absence of errors. Hardware verification can be used to uncover all inconsistencies between a mathematical model of the implementation and the formal specification. Hunt suggests that it is faster to verify a microprocessor design than to exhaustively test one (ref. 3).

Hardware verification requires that a system design 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. The behavioral semantics are clearly defined; providing an accurate basis for building systems (ref. 4).

1.1 MEMORY MANAGEMENT

The principle purpose of an operating system is to manage system resources. Perhaps the most fundamental resource is main memory. On behalf of a program, the operating system allocates

a section of main memory to load the program into before execution. During execution, the operating system will handle dynamic requests for additional memory. Sophisticated operating systems also support additional memory management capabilities including security and virtual memory functions.

As a minimal security function, the operating system must ensure process noninterference. Each process expects that its space will not be modified or read by other processes. Further, different portions of a process can be tagged as readable, writable, executable, or a combination of the three.

Most machines have a physical memory address space that is much smaller than the address space the processor can address. For example, a 32-bit processor may be capable of addressing 4 gigabytes of memory (2^{32}) while the machine only has 16 megabytes of actual main memory (2^{24}). When several programs are executing, each may expect access to the entire address space. Virtual memory allows the entire address space to appear available to each process.

Left to software alone, security and virtual memory capabilities cannot be completely provided. The functions demand hardware support. These functions may be present as part of the central processing unit (CPU) or as a separate chip. The MMU acts as a filter between the CPU and memory (see Figure 1.1-1).

For each CPU memory request, the MMU determines whether the request will violate security constraints. If virtual memory support is also provided, the MMU will translate a request from a virtual to a real location. When the virtual location does not map to a location presently in memory, the MMU will inform the CPU that a "fault" has occurred.

Security and virtual memory attributes are defined for blocks of contiguous memory. Access to each block can be restricted to be a combination of read, write, or execute permissions. In systems where all blocks are a fixed size, the blocks are referred to as "pages". When the blocks may be of varying size they are referred to as "segments". In many systems both types of objects are present. Segments consist of a varying number of pages. Protection attributes are established on a segment basis and the real address of a memory word is specified on a page basis.

Simple MMUs expect the information for each block to be written to MMU registers (for example, PDP-11). More sophisticated MMUs will access memory resident tables to ascertain a block's status (for example, Intel 80286, 80386 and Motorola 68851). Also a fully functional MMU would utilize a cache to speed up these table accesses. Process management functions are also frequently present. The operating system is responsible for setting up the tables and can construct

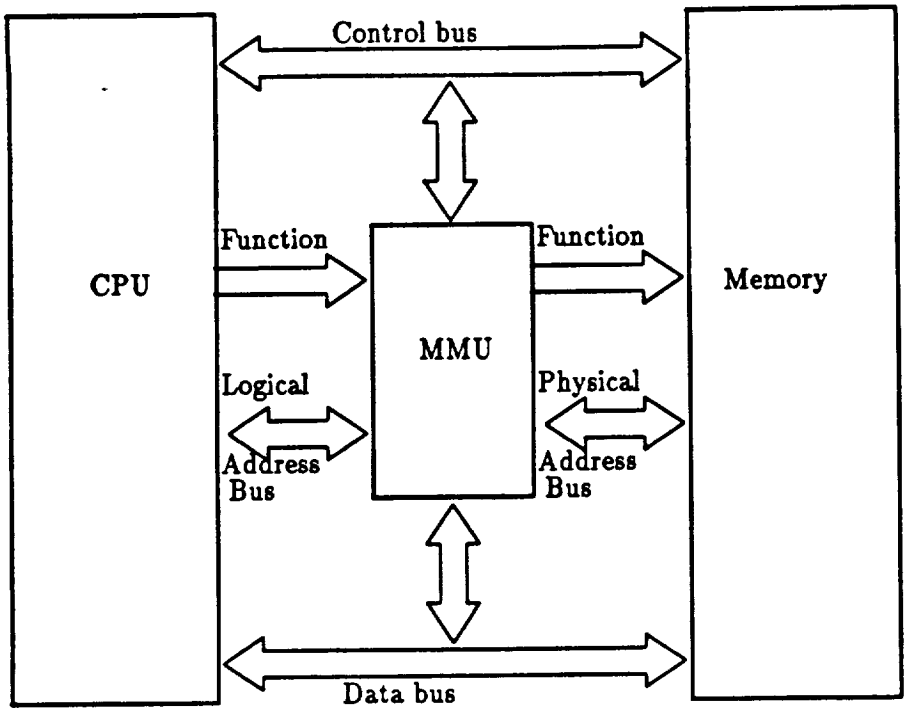


Figure 1.1-1: System Block Diagram

a distinct table for each process.

1.2 INTEGRATION

The MMU must be designed to work with other processors in a cooperative manner. The MMU must be respondent to the actions of other processors. The CPU and MMU have a codependent relationship. The MMU must know the process id (supervisor or user process), the kind of request (instruction fetch or data fetch), as well as whether the request is a read, write, or execute.

MMU exceptions (bad address, segment fault, page fault, invalid access type) are distinct from interrupts. The CPU must be prepared to handle an MMU exception during the execution of an instruction (as opposed to the standard interrupt mechanism where interrupts are handled only after the end of an executing instruction).

If the CPU performs prefetch, it is possible that the prefetch mechanism will inadvertently fetch an address that would never be executed (due to some sort of jump preceding the execution of this "instruction"). If the MMU generates one of the possible exceptions mentioned above, the CPU must postpone processing the exception until the offending value is actually used.

The MMU must also provide a means for the CPU to perform any operation regardless of possible exceptions. For example, when an external interrupt occurs, the CPU must be able to save the return address on a stack.

MMU's can also extend a CPU's instruction set. Instructions to flush its cache, search or load a translation table entry, or test the access rights of a process may be provided. To support operating system memory management, the MMU may also be responsible for setting a dirty bit within a page descriptor when the page has been modified.

The MMU must be responsive to other devices as well. For example, the activity of a direct memory access unit (DMA) can invalidate MMU cache entries. Either the MMU must watch the bus traffic or a mechanism must be available to the CPU to invalidate cached entries.

1.3 VERIFIED MEMORY MANAGEMENT UNITS

Each of the MMUs are constructed from a combination of gates, registers and word comparison units. The gates and registers were available from previous work; however, the word comparison units were designed and verified for this effort.

The simplest MMU combines a register with a word comparison unit. Addresses from a system bus can be stored in the register or compared with the register's value. An acknowledgment signal is returned to indicate whether or not the address matched the register value. Because the word comparison unit provides result output lines to indicate if the first of two inputs is greater than, less than, or equal to the other, the MMU could be trivially changed to return a different result.

While this MMU is primitive, it provides sufficient hardware support for a segmented or paged memory by combining several units and providing each with a distinct part of the address.

For minimal security, the next MMU uses input from a supervisor line. When the supervisor line is high, the MMU operates in supervisor mode. A new register value can only be stored when the MMU is in supervisor mode. Also, all accesses are authorized when in supervisor mode.

The base and bounds MMU adds two significant enhancements. First, the register is addressed as a memory location. When the supervisor line is high, the address bus value matches the register's predefined address, and the write line is high; the MMU will store the value on the data bus in its register. Also, the MMU logically divides each address into two parts: a page and an offset. The register value is divided in the same manner. For the MMU to validate a memory address, the page address must match with the stored page component and the offset must be less than or equal to

the stored bounds component.

The next MMU adds user mode virtual address translation. System information pertaining to both segment and offset validation and virtual address translation is maintained in a pair of registers. These registers can only be accessed when the MMU is operating in supervisor mode.

The last MMU validates CPU memory requests based on a memory resident segment table. Each segment-specific entry in the table defines the segment's availability, read-write-execute access rights, segment size, and real address location in memory.

The addition of these features reduces the amount of operating system software support. By developing a sophisticated MMU in steps, the construction of the final proof is much more tractable.

In the sections that follow, we briefly describe the HOL theorem prover. Then, we describe the above devices and several auxiliary theories developed to support their verification. The final section is a description of future work, including composing the MMU with a cache.

1.4 RELATED WORK

Neumann proposes a unified hierarchy that accomodates all critical requirements (ref. 5). 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 that are discharged when the level is composed (ref. 6).

There has been significant interest in formal verification as an alternative to simulation (refs. 7, 8, 9 and 10). Hardware verification efforts thus far have focused primarily on a microprocessor as the base for computer systems (refs. 3, 11, 12 and 13).

Perhaps the best known verification effort is that of the VIPER microprocessor (refs. 11, 14 and 15). VIPER is the first microprocessor intended for commercial distribution where a formal verification has been attempted. However, these processors are quite limited. Only Joyce's microprocessor, Tamarack-3, provides interrupts, and none provide memory management functions necessary to support a secure operating system.

Previous efforts to verify systems have included construction of vertically verified systems with a microprocessor/memory as the system's base. Joyce has specified and verified a compiler for the verified Tamarack-3 microprocessor (ref. 16).

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. 4). In this way, higher levels of the stack define new functionality by collecting the next lower level's functionality. The stack consists of a compiler (Micro-Gypsy), an assembler and linking loader, an operating system, and a microprocessor.

Bevier has verified a simple operating system (KIT), which ensures that tasks are isolated from one another. Implementation of the hardware base has not been verified (refs. 17 and 18). He assumes extensions to the FM8502 microprocessor to provide interrupts, asynchronous I/O, memory management, and supervisor-mode instructions.

1.5 HOL

HOL is a general theorem proving system developed at the University of Cambridge (refs. 1 and 19) that is based on Church's theory of simple types, or higher order logic (ref. 20). 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. 21) and is similar to other LCF progeny such as NUPRL (ref. 22). 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.

1.5.1 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 1.5-1 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 1.5-2 shows several of HOL's built-in binders.

Table 1.5-1: 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 1.5-2: 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 α and t has type β, the abstraction λ x. t has the type $(\alpha \rightarrow \beta)$.

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

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 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

Table 1.5-3: 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 **

(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, then $(\sigma_1, \ldots, \sigma_n)$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 1.5-3. 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 `(* -> **)`.

1.5.2 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.

1.6 DEVICE SPECIFICATION

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.

1.7 ADDITIONAL NOTATION

In the text, various fonts will be used to denote constants, `definition names` and *object types*. The turnstile symbol $\vdash$, is used to indicate that the term is a theorem which has been formally proven in the logic. When the subscript "def" is present (eg $\vdash_{def}$), the theorem is simply a definition.

2.0 AUXILIARY THEORIES

An MMU will receive as input both boolean control signals and word values. The word values are abstractly viewed as addresses into memory, but take the concrete form of an array of boolean values or bits. This sequence of bits will be referred to as a "bitVector". To support the verification of the MMUs, a theory defining how bitVectors can be ordered was constructed.

A theory describing a device that compares bitVectors was also constructed. The device accepts two bitVectors and returns a result indicating whether the first bitVector is greater than, less than or equal to the second bitVector.

2.1 BITVECTORS

BitVectors are represented by the type *:num→bool*, but are constrained to be a finite length. BitVectors are functions that, when applied to a number, return the bit at that offset. Given a bitVector *B* with length *n+1*, the term *B 0* returns the least significant bit value and the term *B n* returns the most significant bit value.

The bitVector theory contains function definitions to compare bitVectors and to compare subsequences of bitVectors. The definitions are recursive so that they may apply to bitVectors of any length. Many of the functions expect the first argument to be the offset of the most significant bit (msb) of a bitVector.

The auxiliary definitions ARB, ZEROS and ABS are defined in the box below. ARB uses the Hilbert choice operator to return an arbitrary bit (boolean) value. ZEROS serves as a bitVector of F values. The curried function expects width and bit offset number arguments and returns F for any line within the width range and an arbitrary value of type *bool* otherwise.

Signals are defined similarly to bitVectors. The concrete type is defined as *:time→bitVector* (or *:num→num→bool*). However, it is convenient for signals to appear to be of type *:num→time→bool*. The function ABS reorders arguments so that abstract signals are implemented by a function involving bitVectors.

```
⊢def ARB = ARB = ε (x:bool) . F

⊢def ZEROES w n = (n <= w) → F | ARB

⊢def ABS (w:num) (sig:num→num→bool) (t:num) (n:num)
        = n <= w → sig n t | ARB
```

Definitions bvEQUAL, bvGREATER and bvLESS correspond to the numeric comparison functions: equal, greater than and less than. These definitions reflect a twos-complement interpretation of bitVectors where the least significant bit is bit 0. T is used for the bit value 1 and F for the bit value 0. The first argument specifies the most significant bit offset and is followed by two bitVectors. The definitions, being recursive, specify a base case (where the msb offset is zero), and the inductive case. Note that bvLESS is defined as a function of bvGREATER with the bitVector arguments reversed.

```
⊢def (bvEQUAL 0 a b = (a 0 = b 0) ) ∧
     (bvEQUAL (SUC n) a b = (bvEQUAL n a b  ∧  (a (SUC n) = (b (SUC n))) ))

⊢def (bvGREATER 0 a b = ( a 0  ∧  ¬ b 0 ) )  ∧
     (bvGREATER (SUC n) a b =
       ( (a(SUC n) ∧  ¬ b(SUC n))  ∨
         ((a(SUC n)=b(SUC n))   ∧  bvGREATER n a b)
       ))

⊢def bvLESS n a b = bvGREATER n b a
```

Comparison definitions, which only consider a contiguous section of a bitVector are also defined. bvPART constructs a bitVector given a range and a bitVector. Outside the range, the new bitVector returns F, while within the range, the new bitVector returns the old bitVector's corresponding value. Definitions bvEQbit is a shorthand to compare two bits. bvPartEQUAL, and bvPartGREATER, bvPartLESS compare contiguous sections of bitVectors; from a specified top bit down to a specified bottom bit.

```
⊢def bvPART max min (sig:num→bool) (n:num)
       = (n > max) → F | (n < min) → F | sig n

⊢def bvEQbit x a b = (a x = (b (x:num)):bool)

⊢def (bvPartEQUAL 0 y a b =
     ( (y = 0) →  (bvEQbit 0 a b) | F ))  ∧
    (bvPartEQUAL (SUC x) y a b =
      ((SUC x) > y → (bvEQbit (SUC x) a b  ∧  (bvPartEQUAL x y a b)) |
                     ((SUC x) = y) → (bvEQbit (SUC x) a b) | F
      ))

⊢def (bvPartGREATER (SUC x) y a b =
      ( ((SUC x) > y) →
          ( ( a(SUC x)  ∧  ¬ b(SUC x) )  ∨
            ((a(SUC x)=b(SUC x))  ∧  bvPartGREATER x y a b) ) |
       ((SUC x) = y) → (a(SUC x) ∧  ¬ b(SUC x)) | F  )     )

⊢def bvPartLESS x y a b = bvPartGREATER x y b a
```

2.2 GATES

The devices are constructed from the gates described below. The gates inv, nor2 and nand2 are assumed to be primitive, and from these we construct and2_imp and or2_imp.

```
⊢def inv in out = (out = ¬ in)

⊢def nor2 a b out = (out = ¬ (a ∨ b))

⊢def nand2 a b out = (out = ¬( a ∧ b))

⊢def and2_imp a b out = (∃ p. nand2 a b p ∧ inv p out)

⊢def or2_imp a b out = (∃ p. nor2 a b p ∧ inv p out)
```

2.3 BITVECTOR COMPARISON UNITS

Two bitVector comparison units are constructed. The first compare unit produces three boolean results indicating either a greater than, less than or equal relation between the two input bitVectors. Frequently all that is needed is a device that recognizes two bitVectors as equal. The second unit compares two bitVectors for equality as defined by the bitVector definition bvEQUAL.

2.3.1 COMPLETE BITVECTOR COMPARISON UNIT

The bitVector comparison unit takes two words as input and produces three boolean results indicating whether the first was greater than, less than, or equal to the second bitVector. The specification and implementation definitions are constructed recursively. We begin by defining a specification bitComp_spec, and implementation bitComp_imp, for a device where the inputs (first,sec) are each a single bit rather than a bitVector. The implementation is proved to be equivalent to the specification. Note the existentially quantified variables p and q are lines internal to the device.

```
⊢def bitComp_spec first sec g l e =
          (g = ( first ∧ ¬ sec)) ∧
          (l = ( ¬ first ∧ sec)) ∧
          (e = ( first = sec ))

⊢def bitComp_imp first sec g l e =
    ∃ p q . (inv first p)     ∧ (inv sec q) ∧
             (nor2 p sec   g) ∧
             (nor2 q first l) ∧
             (nor2 g l     e)

⊢ bitComp_imp first sec g l e = bitComp_spec first sec g l e
```

Definitions for two-bit words can be constructed in a similar manner as shown below. The implementation compComb_imp is proved to be equivalent to the specification compComb_spec.

```
⊢def compComb_spec g0 g1 l0 l1 e0 e1 g l e =
      (g = (g1 ∨ (e1 ∧ g0))) ∧
      (l = (l1 ∨ (e1 ∧ l0))) ∧
      (e = (e1 ∧ e0))

⊢def compComb_imp g0 g1 l0 l1 e0 e1 g l e =
    ∃ p q. (and2_imp e1 g0 p) ∧ (or2_imp g1 p g) ∧
           (and2_imp e1 l0 q) ∧ (or2_imp l1 q l) ∧
           (and2_imp e1 e0 e)

⊢ compComb_imp g0 g1 l0 l1 e0 e1 g l e = compComb_spec g0 g1 l0 l1 e0 e1 g l e
```

Using the bitVector comparison definitions and the bitComp specification and implementation, a compare unit for an arbitrary sized bitVector is defined using recursive definitions and verified.

```
⊢def comp_spec n a b g l e =
        ( g = ( bvGREATER n a b) ) ∧
        ( l = ( bvLESS n a b) ) ∧
        ( e = ( bvEQUAL n a b) )

⊢def (comp_imp 0 a b gr ls eq = (bitComp_imp (a 0) (b 0) gr ls eq)) ∧
     (comp_imp (SUC n) a b gr ls eq =
       ∃ gn ln en gn ln en .
          (comp_imp n a b gn ln en) ∧
          (bitComp_imp (a (SUC n)) (b (SUC n)) gn ln en) ∧
          (compComb_imp gn gn ln ln en en gr ls eq) )

⊢ comp_imp n a b great less equ = comp_spec n a b great less equ
```

An example of an implementation for bitVectors of length three is in Figure 2.3-1.

2.3.2 COMPARISON OF BITVECTOR EQUALITY

Frequently, the full power of the compare unit described above is not required. For example, for a device to recognize bus requests directed to it, the device need only compare for equality the bus address with a predefined address. Note that an equality comparison unit also requires many fewer gates.

The equality comparison unit is defined in a manner similar to the full comparison unit. First, we construct a device that recognizes bit equality, and then we construct an equality unit for arbitrary sized bitVector inputs. Figure 2.3-2 shows an equality comparison unit for bitVectors of length three.

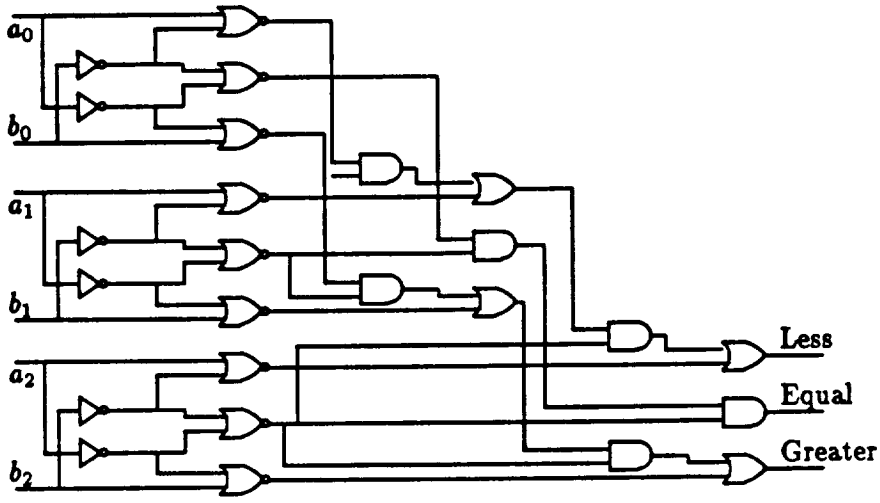


Figure 2.3-1: Compare Two BitVectors

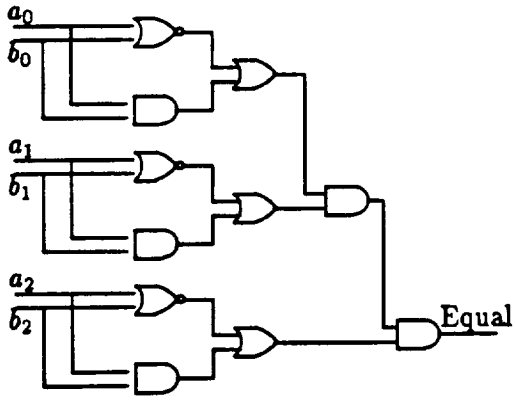


Figure 2.3-2: Compare Two Words For Equality

```
⊢def bitEq_spec first sec e =
    (e = ( first = sec ))

⊢def bitEq_imp first sec e =
    ∃ i j . (nor2   first sec i)  ∧
             (and2_imp first sec j)  ∧
             (or2_imp i j e)

⊢ bit_imp first sec e = bitEq_spec first sec e
```

```
⊢def compEq_spec n a b e = ( e = ( bvEQUAL n a b) )

⊢def (compEq_imp 0 a b eq = (bitEq_imp (a 0) (b 0) eq)) ∧
     (compEq_imp (SUC n) a b eq =
     ∃ en en .
          (compEq_imp n a b en)  ∧
          (bitEq_imp (a (SUC n)) (b (SUC n)) en)   ∧
          (and2_imp en en eq))

⊢ compEq_imp n a b e = compEq_spec n a b e
```

2.4 REGISTERS

Registers are used to store the state of an MMU over time. This theory was implemented by Phil Windley and included in this report for the sake of completeness.

Registers receive an input bitVector, and clear and load control signals. A register's output at time *t+1* depends on its input control lines `clr` and `ld` at time *t*. The output remains unchanged if both control lines are F. If both lines are high, the register is cleared. A register implementation is constructed from primitive gates, and a formal proof shows the implementation is equivalent to the specification.

```
⊢def  reg_spec w i ld clr out =
    (∀ t:num . out(t+1) = (clr t → ZEROES w | ld t → i t  | out t) )
      ∧
    (out 0 = ZEROES w)

⊢def (reg_imp 0 i ld clr out = d_ff (i 0) ld clr (out 0))
       ∧
     (reg_imp (SUC n) i ld clr out = ((reg_imp n i ld clr out)  ∧
       (d_ff (i (SUC n)) ld clr (out (SUC n)))))

⊢ reg_imp w i ld clr out =  reg_spec w (ABS w i) ld clr (ABS w out)
```

3.0 SIMPLE MEMORY MANAGEMENT UNITS

3.1 PAGE CHECK TLM

The page check TLM (translation look-aside module) is the simplest MMU. Protection is generally needed on a page or segment basis' ; rarely on a word basis [1]. Memory addresses can be decomposed into a page and a page offset descriptor. The page check TLM acts only on the page descriptor.

The device will either compare a received page descriptor[2] with another value previously stored in a register or store a new value for future comparisons. When a comparison is performed, the unit returns T when the two values are the same. The device is expected to return a result one time epoch after receiving its inputs. The units are defined using the auxiliary definitions mentioned in the previous section and are correct for all bitVector widths. To isolate the timing dependencies, the specification is divided into two parts: pgCk and pgCk_spec.

The definition pgCk_spec describes the timing details. The register and acknowledgment output values at *t+1* are a function of the input values at time *t*. The function is specified by pgCk.

The definition pgCk accepts a bitVector address, a write/compare command line and a register and returns a tuple containing the resultant register value and acknowledgment output. If the command line is T, the register is updated and the output acknowledgment is set to T (regardless of the comparison result). If the command line is F, indicating a comparison should be performed, the output acknowledgment is dependent on the result of the comparison.

The implementation pgCk_imp is constructed by composing a register, a comparison unit and an OR gate (Fig. 3.1-1 [3]). The definitions show the use of the ABS function to allow signals to take arguments out of order. The implementation is shown to imply the specification.

---

[1] Here a page is a contiguous block of memory words; each block being a fixed length. Segments are blocks of words but all segments need not be of the same length.

[2] Note that the concrete implementation of a page descriptor is a subsequence of a bitVector.

[3] The reset box in the figure is set to F in the definition.

```
⊢def pgCk n address write rgstr =
      (write = T)                → (address, T) |
      (bvEQUAL n rgstr address)→ (rgstr,   T) |  (rgstr,   F)

⊢def pgCk_spec n addr rWC reg ack =
       ∀ (t:num). (reg(t+1), ack(t+1)) =  pgCk n (addr t) (rWC t) (reg t)

⊢def pgCk_imp n addr rWC reg ack =
      ∀ t. ∃ g l e.
       (reg_imp n addr rWC bitFalse reg )  ∧
       (comp_imp n (ABS n reg t) (ABS n addr t) g l e)  ∧
       (or2_imp e (rWC t) (ack (t+1)) )

⊢ pgCk_imp n addr rWC reg ack ==>  pgCk_spec n (ABS n addr) rWC (ABS n reg) ack
```

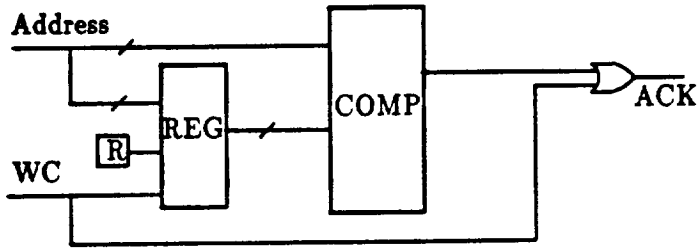


Figure 3.1-1: Page Check TLM

3.2 PAGE CHECK TLM WITH SUPERVISOR LINE

The simple page check unit cannot guarantee that processes will not interfere with one another. Processes cannot be trusted to leave the page check unit's register unmodified. The above unit cannot prevent a process from writing to the TLM unit and altering the protection scheme intended by the operating system kernel. The enhanced unit receives input from a supervisor input line. Only when the supervisor line is high, can a write to the page check register occur.

We assume that the CPU has two control states: a supervisor state intended for operating system use and a user state for use by application processes. Generally, the supervisor line status is defined by a bit in the central processing unit's program status word (PSW). Microprocessors, designed for multiprocessing, restrict access to the PSW so that process status bits (including the supervisor bit) can be modified only when the system is executing in supervisor state. This scheme assumes that nonkernel tasks execute in user state. The supervisor bit can be extended into a process identifier field or a security ring field.

The implementation requires one additional AND gate and an internal line. The proof is quite similar to the pgCk proof; it requires an additional case split to deal with the supervisor line.

```
⊢def pgCka_spec n addr rWC sup reg ack =
        ∀ (t:num). (reg(t+1), ack(t+1)) =
          pgCk n (addr t) (rWC t  ∧  sup t) (reg t)

⊢def pgCka_imp n addr rWC sup reg ack =
      ∀ t. ∃ x g l e.
       (and2_imp (rWC t) (sup t) (x t) )  ∧
       (reg_imp n addr x bitFalse reg )  ∧
       (comp_imp n (ABS n reg t) (ABS n addr t) g l e)  ∧
       (or2_imp e (x t) (ack (t+1)) )

⊢ pgCka_imp n addr rWC sup reg ack ⇒ pgCka_spec n (ABS n addr) rWC sup (ABS n reg) ack
```

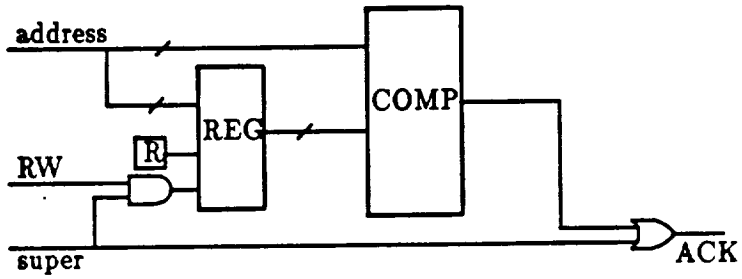


Figure 3.2-1: Page Check TLM With Supervisor Line

3.3 BASE AND BOUNDS MMU

The base and bounds MMU (bb-MMU) extends the capabilities of the page check devices. This last "simple" MMU is actually much more sophisticated than the previous devices. While the page check units left unspecified how the device's register is addressed, the bb-MMU provides a more complete interface to a system bus. The device expects inputs consisting of an address (in bitVector form), a supervisor line, a read/write line and a data value. When a request is valid, the device asserts an acknowledgment signal.

The bb-MMU is positioned between the CPU and memory and must recognize when bus requests are targeted to itself. The bb-MMU protection register is accessed as a memory location. When the supervisor line input is asserted (T) the bb-MMU will operate in supervisor mode.

In supervisor mode, the bb-MMU compares a memory request's bus address with a constant to determine whether the protection register is being accessed. If the address does match and the read/write line is T, then the protection register value will be updated. Whether the protection

register is updated or not, the acknowledgment line will be asserted.

In user mode, the bb-MMU decomposes the input address and register output into a segment and offset component. The bb-MMU verifies that the address segment matches the stored segment component (the base) and that the address offset is not greater than the stored offset (the bounds). The top bits (between n and s) of the address bitVector represent the segment identifier.

The specification is divided into parts to distinguish the supervisor and user mode behaviors. The specification **baseBoundCk_spec** is only valid when the segment offset size s is less than the bitVector size n. Note that the data and address bitVector sizes are implicitly defined to be the same length. The specification defines the resulting state as a tuple consisting of the protection register value and the acknowledgment line value. When the supervisor line is high, bbSUPERV defines the result state, otherwise, bbCOMP defines the result state.

The parameter ADDR represents an unspecified constant denoting the address of the protection register.

```
⊢def bbSUPERV n bbReg addr data ADDR rw =
      ( rw → ((bvEQUAL n addr ADDR) → (data, T:bool) | (bbReg, T))
             |   (bbReg, T)   )

⊢def bbCOMP n s bbReg addr =
    (bvEQUAL n (bvPART n s bbReg) (bvPART n s addr)  ∧  ¬(bvGREATER s addr bbReg) )
          → (bbReg, T:bool) | (bbReg, F)

⊢def bbNextState n s bbReg addr data ADDR super rw =
      ( super → bbSUPERV n bbReg addr data ADDR rw |
                | bbCOMP   n s bbReg addr )

⊢def baseBoundCk_spec n s bbReg addr data ADDR super rw ack =
       (s < n) ⇒  ∀ t. ( bbReg(t+1),ack(t+1) ) =
        bbNextState n s (bbReg t) (addr t) (data t) ADDR (super t) (rw t)
```

The implementation is defined using primitive gates, as well as the register and full comparison unit described previously. A more efficient implementation would use the equality comparison unit. The abstract function PRT is used to split off a subsection of a bitVector.

```
⊢def PRT w max min (sig:num->num->bool) (t:num) (n:num)
       = (n > max) → F |
         (n < min) → F |
         (n <= w) → (sig n t) |   ARB

⊢def baseBoundCk_imp n s bbReg addr data ADDR super rw ack =
      (s < n) ⇒  ∀ t.
     (∃ writeBB g0 g1 g2 10 11 12 e2 x addrMatch goodSeg goodOfs ok.
        (reg_imp n data writeBB bitFalse bbReg)  ∧
        (comp_imp n (ABS n addr t) ADDR g0 10 (addrMatch t))  ∧
        (and2_imp (rw t) (super t) (x t))  ∧
        (and2_imp (addrMatch t) (x t) (writeBB t))  ∧
        (comp_imp n  (PRT n n s bbReg t)
                     (PRT n n s addr  t)  g1 11 goodSeg)  ∧
        (comp_imp s  (ABS n addr  t)
                     (ABS n bbReg t)  g2 12 e2)  ∧
        (inv g2 goodOfs)  ∧
        (and2_imp goodOfs goodSeg ok)  ∧
        (or2_imp ok (super t) (ack (t+1)) )
     )
```

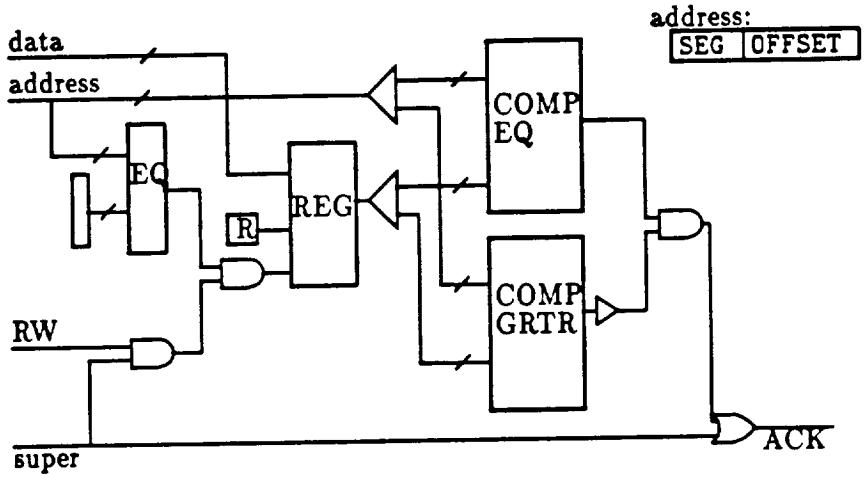


Figure 3.3-1: Base and Bounds MMU

The proof is substantially more complicated than the proofs for the page check units. In the process of verifying that the implementation implies the specification, several intermediate lemmas are useful. While they are all seemingly obvious, HOL requires a proof for each.

```
Lemma 0
⊢ (s < n) ⇒  (PRT n n s sig t) = (bvPART n s(ABS n sig t) )

Lemma 1
⊢ (bvEQUAL n(bvPART n s(ABS n bbReg t))(bvPART n s(ABS n addr t))  ∧
     ¬ bvGREATER s(ABS n addr t)(ABS n bbReg t)) =
     (¬ bvGREATER s(ABS n addr t)(ABS n bbReg t))  ∧
    (bvEQUAL n(bvPART n s(ABS n bbReg t))(bvPART n s(ABS n addr t)))

Lemma 2
⊢ (n > 0 ) ⇒  (SUC (PRE n) -1) + 1 = (SUC(PRE n))

Lemma 3
⊢ (n:num). (n > 0 ) ⇒ (SUC (PRE n)) =  n
```

Proving the final theorem required 492.7 seconds of CPU time and generated 31,227 intermediate theorems.

```
⊢ baseBoundCk_imp n s bbReg addr data ADDR super rw ack ⇒
   baseBoundCk_spec n s (ABS n bbReg) (ABS n addr) (ABS n data)  ADDR super rw ack
```

Proper management of the register's contents ensures that a process can only modify a specified address space. Although very simple, a set of these devices composed together would be sufficient to satisfy a system's security need to enforce process noninterference. While the use of multiple devices is not strictly necessary, a system with several devices might considerably reduce operating system overhead.

4.0 VIRTUAL ADDRESS TRANSLATION MMU

The MMU is programmed through two memory-mapped control registers:

a. A protection register governs the range of valid virtual memory addresses a process may access.

b. A translate address register designates the base real address accessible in memory.

Processes cannot be trusted on their own to leave the unit's registers unmodified. Only when the supervisor line is high will the unit permit a register write. This ensures that the security protection scheme intended by the operating system kernel cannot be altered intentionally or unintentionally by user processes. This scheme assumes that nonkernel tasks execute in user state. The supervisor bit can be extended into a process identifier field or a security ring field.

The protection register and virtual addresses are partitioned into a segment and an offset[4]. A request is validated if the segment address matches the stored segment component and the offset is less than or equal to the stored bounds component. When a request is validated, the MMU constructs a real address using the offset of the requested address and the translate address register. When the supervisor line is asserted, all accesses are authorized and address translation is not performed.

4.1 SPECIFICATION

The abstraction functions PRT and PRTA are used to split off a subsection of a bitVector [5]. The function definition VtoR, creates a real address by replacing the segment identifier with the real base offset; the bottom s bits of the virtual address remain unchanged.

```
⊢def PRT w max min (sig:num→num→bool) (t:num) (n:num) =
       (n > max) → F |
       (n < min) → F |
       (n <= w) → (sig n t) |   ARB

⊢def PRTA w max min (sig:num→bool) (n:num) =
       (n > max) → F |
       (n < min) → F |
       (n <= w) → (sig n) |   ARB

⊢def VtoR realA virtA s n = (n > s) →  (realA n):bool | (virtA n)
```

[4]Here a page is a contiguous block of memory words; each block being a fixed length. Segments are blocks of words but all segments need not be of the same length.

[5]Please see the appendix for a description of bitVectors and many of the device building blocks.

The specification **virtBBCk_spec** is defined as a state transition system. The specification defines the state and environment at time *t+1*, as a function of the state and environment at time *t*. The state is maintained in variables (**bbReg**, **vaReg**). The input environment consists of the address bus value, data bus value, and control bus signals (**addr**, **data**, **super**, **rw**). The output environment consists of a request validation line and a real address (**ack**, **outAddr**). The functions **vSUPERV** and **vCOMP** define the supervisor and user mode behaviors, respectively. The parameters **n**, **s** and **ADDR** serve as constants defining the most significant bitVector bit, the most significant address offset bit and the base address of the MMU registers. The size of the bitVectors must be greater than the segment offset for the specification to be meaningful.

```
⊢def  vSUPERV n bbReg vaReg addr data ADDR rw =
      ( (rw  ∧  (bvEQUAL n (bvPART n 1 addr) (bvPART n 1 ADDR) ))
   → (addr 0)  → (data,  vaReg, addr, T:bool) |
                 (bbReg, data,  addr, T:bool) |
      (bbReg, vaReg, addr, T)   )

⊢def VtoR realA virtA s n = (n > s) →  (realA n):bool | (virtA n)

⊢def vCOMP n s bbReg vaReg addr =
      (bvEQUAL n (bvPART n s bbReg)(bvPART n s addr)  ∧
        ¬ (bvGREATER s addr bbReg) )
       → (bbReg, vaReg, (VtoR vaReg addr s), T:bool)  |
         (bbReg, vaReg, addr, F)

⊢def vNextState n s bbReg vaReg addr data ADDR super rw =
      super → vSUPERV n bbReg vaReg addr data ADDR rw |
       vCOMP   n s bbReg vaReg addr

⊢def virtBBCk_spec n s bbReg vaReg addr data ADDR super rw ack outAddr =
      (s < n) ⇒
      ∀ t. ( bbReg(t+1),vaReg(t+1), outAddr(t+1), ack(t+1) ) =
       vNextState n s (bbReg t) (vaReg t) (addr t) (data t)
                  ADDR (super t) (rw t)
```

4.2 IMPLEMENTATION

The implementation **virtBBCk_imp** is defined using primitive gates, registers and the full comparison unit described previously. A more efficient implementation would use an equality comparison unit. The function **pick_imp** defines a bitVector MUX. The datapath can be seen in Figure 4.2-1.

```
⊢def pick_imp (wordA :num→bool) (wordB :num→bool) (which:bool) res
       = (which = T) → (res = wordA) | (res = wordB)

⊢def virtBBCk_imp n s ADDR bbReg vaReg addr data super rw ack outAddr=
      (s < n) ⇒  ∀ t.
      (∃ wBB wVA select x aM0 aM1 aM2 goodSeg goodOfs ok nok nxlat g l e.
         (and2_imp (rw t) (super t) (x t))  ∧
         (compEq_imp n (PRT n n 1 addr t) (PRTA n n 1 ADDR) (aM0 t))  ∧
         (and2_imp (aM0 t) (x t) (aM1 t))  ∧
         (inv (addr 0 t) (aM2 t) )  ∧
         (and2_imp (aM1 t) (addr 0 t) (wBB t))  ∧
         (and2_imp (aM1 t) (aM2  t)   (wVA t))  ∧
         (reg_imp n data wBB bitFalse bbReg)  ∧
         (reg_imp n data wVA bitFalse vaReg)  ∧
         (compEq_imp n  (PRT n n s bbReg t)
                        (PRT n n s addr  t) goodSeg)  ∧
         (comp_imp s    (ABS n addr  t)
                        (ABS n bbReg t)  g l e)  ∧
         (inv g goodOfs)  ∧
         (and2_imp goodOfs goodSeg ok)  ∧
         (or2_imp ok (super t) (ack (t+1)) )  ∧
         (inv ok nok )  ∧
         (or2_imp nok (super t) nxlat)  ∧
         (pick_imp (ABS n addr t) (ABS n vaReg t) nxlat (select t))  ∧
         ( (outAddr (t+1))= (VtoR (select t) (ABS n addr t) s ) )
```

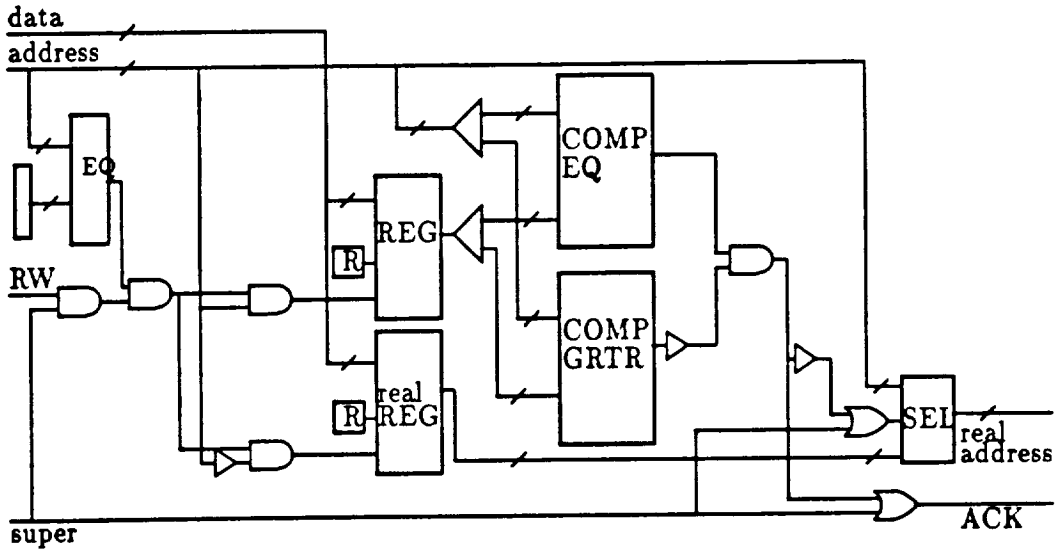


Figure 4.2-1: Base and Bounds MMU with Virtual Address Translation

4.3 VERIFICATION

Several simple intermediate lemmas were proven with the final theorem requiring 1,209 seconds of CPU time executing on a Sun SparcStation. The final proof generated 64,185 primitive inferences.

```
Lemma 0
⊢ PRT n n s sig t = bvPART n s(ABS n sig t)

Lemma 1
⊢ (bvEQUAL n(bvPART n s(ABS n bbReg t))(bvPART n s(ABS n addr t))  ∧
     ¬ bvGREATER s(ABS n addr t)(ABS n bbReg t)) =
     (¬ bvGREATER s(ABS n addr t)(ABS n bbReg t))  ∧
    (bvEQUAL n(bvPART n s(ABS n bbReg t))(bvPART n s(ABS n addr t)))

Lemma 2
⊢ VtoR a a s = a

Lemma 3
⊢ PRTA n n s sig = bvPART n s sig

Lemma 4
⊢  addr 0 t = ABS n addr t 0
```

Several of these units could be combined to provide sufficient hardware support for a segmented and paged memory. This design also supports multiple process requirements assuming the top bits of an address specify a process identifier.

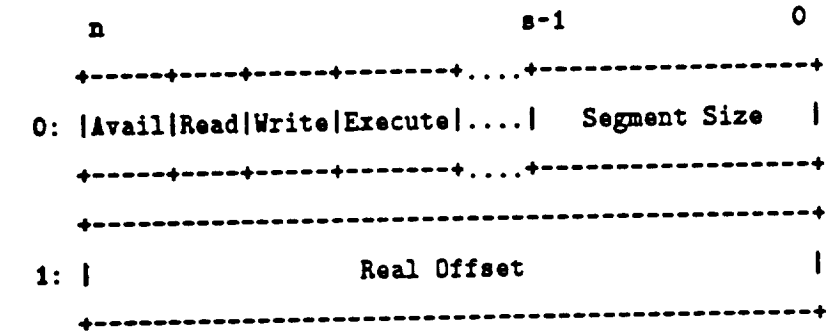
```
⊢  virtBBCk_imp n s ADDR bbReg vaReg addr data super rw ack outAddr ⇒
    virtBBCk_spec n s ADDR (ABS n bbReg) (ABS n vaReg) (ABS n addr)
                  (ABS n data) super rw ack outAddr
```

5.0 MEMORY-RESIDENT TABLE MMU

This MMU provides protection and address translation on a segment basis. These functions are only in effect when the MMU operates in user mode. When operating in supervisor mode, the memory protection mechanism is inactive and requests are passed through without address translation.

Addresses consist of a segment identifier and a segment offset. The segment identifier is used to fetch the segment descriptor. Segment descriptors are located in a memory-resident table and consist of two words. The first word specifies the segment size and read, write and execute permissions. The second word acts as a base address for the segment's real location in memory. To translate from a virtual address to a real address, the MMU adds the segment offset to the segment base address. To support segment paging, the first word also contains a bit indicating whether the segment is presently in memory. If this bit is F, the operating system is free to use the second word as a disk offset or in any other fashion.

The location of the table is determined by the MMU's segment table pointer register. This register is accessible only in supervisor mode. The MMU assumes the table provides an entry for all possible segment descriptors.



The MMU described here must fetch a descriptor from memory for each access. Initial work on a cache to speed up performance is discussed in a subsequent section.

The previous units were constructed in a bottom up manner—from the gate level up. Using the verification of these units as a model, devices that compare one bitVector with another in an arbitrary way could be specified and successfully verified. The device described in this section takes a top-down approach to the verification of a much more complicated device. The implementation level here is the electronic block level. We construct a generic theory describing an MMU where several functions are left abstract.

5.1 GENERIC THEORIES

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.

For a more complete description of abstract theories see (ref. 23).

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.

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 that manipulate these types.

Previous device theories have considered the size of the segment identifier and segment offset fields within a bitVector. The abstract representation ignores these details by providing functions that return the segment identifier or segment offset fields from an address (`segId` and `segOfs`, respectively). There is also a function `segIdshf`, which returns the offset of a segment descriptor within the memory-resident segment table for a given address. Since descriptors require two words, the implementation of this function simply shifts the segment identifier to the left 1-bit position (e.g. , adds a trailing zero bit).

The abstract functions `availBit`, `readBit`, `writeBit` and `execBit` extract a bit value from an argument of type **wordn*. These functions are applied to the first word of a segment descriptor.

Several functions that operate on two-tuples are available. Given a pair of **wordn* values, `add` returns a value of **wordn*. Functions `addrEq`, `ofsLEq` and `validAccess` replace the concrete comparison units used in previous units.

Additional abstract coercion functions are available to convert values between types. If the theory were instantiated, the concrete implementation of the abstract types would likely be the same (bitVectors) and these functions would be unnecessary.

Memory is also treated abstractly. The abstract representation provides a fetch function, and a transformation function [6].

```
new_type_abbrev ('RWE', ":bool # bool # bool");;

let mmu_abs = new_abstract_representation
 [
  ('segId',          ":(*address -> *wordn)"                 );
  ('segOfs',         ":(*address -> *wordn)"                 );
  ('segIdshf',       ":(*address -> *wordn)"                 );
% %
  ('availBit',       ":(*wordn -> bool)"                     );
  ('readBit',        ":(*wordn -> bool)"                     );
  ('writeBit',       ":(*wordn -> bool)"                     );
  ('execBit',        ":(*wordn -> bool)"                     );
% %
  ('add',            ":(*wordn # *wordn ->*wordn)"           );
% %
  ('addrEq',         ":(*address # *address  -> bool)"       );
  ('ofslEq',         ":(*address # *wordn    -> bool)"       );
  ('validAccess',    ":(*address # *wordn # RWE  -> bool)"   );
% Coercion functions %
  ('val',            ":(*wordn -> num)"                      );
  ('wordn',          ":(num->  *wordn)"                      );
  ('address',        ":(*wordn -> *address)"                 );
% Memory functions %
  ('fetch',          ":(*memory # *address) -> *wordn"       );
  ('trans',          ":*memory -> *memory"                   );
 ];;

let mmu_ty = abstract_type 'mmu_abs' 'segId';;
```

A type abbreviation *RWE* is also defined to be a three tuple of bit values. Selector functions rBIT, wBIT and eBIT access the first, second, and third bits, respectively.

```
⊢def rBIT rwe =  (FST rwe)

⊢def wBIT rwe =  (FST (SND rwe))

⊢def eBIT rwe =  (SND (SND rwe))
```

5.2 SPECIFICATION

The specification is decomposed into several rules and ignores timing details. The timing details are spelled out in the final correctness theorem. The state of the MMU specification is a three-tuple consisting of a boolean acknowledgment, a memory address and the table pointer register value.

[6] This function is included for future extensions

The definitions **superMode** and **userMode** describe the behavior of the MMU when operating in their respective modes. The definition **legalAccess** uses many of the abstract functions to fetch from memory the appropriate segment descriptor and compare it with the request's access parameters. The definition **vToR** constructs a real address from a virtual address.

The variable *r* in all definitions is the abstract representation.

```
   MMU SPECIFICATION

⊢def legalAccess r vAddr tblPtr rwe mem =
     let a = (fetch r)( mem,
           (address r)((add r) (segIdshf r vAddr,tblPtr) )) in
     ( (validAccess r) (vAddr,a,rwe)  ∧  (ofsLEq r) (vAddr,a))

⊢def vToR r vAddr tblPtr mem =
     let a = (fetch r) (mem, (address r)
        ((add r)( (wordn r 1), (add r)(segIdshf r vAddr,tblPtr) ))) in
     (address r) ((add r) (segOfs r vAddr, a))

⊢def superMode r vAddr rwe tblPtrADDR tblPtr data mem =
     ((wBIT rwe)  ∧  (addrEq r (vAddr,tblPtrADDR)) )
           → ( T, vAddr, data ) |
             ( T, vAddr, tblPtr )
⊢def userMode r vAddr rwe tblPtrADDR tblPtr data mem =
       ( legalAccess r vAddr tblPtr rwe mem
           → ( T, (vToR r vAddr tblPtr mem), tblPtr ) |
             ( F, vAddr,                     tblPtr ) )

⊢def mmu_spec r vAddr rwe tblPtrADDR tblPtr data mem superv =
     (superv → superMode r vAddr rwe tblPtrADDR tblPtr data mem |
               userMode  r vAddr rwe tblPtrADDR tblPtr data mem )
```

5.3 IMPLEMENTATION

The implementation is constructed from electronic-block model components. These are defined as specifications for the behavior of a gate-level implementation. Many of the devices specify their timing behavior as well. The building blocks consist of a security comparison unit, an address match unit, a memory fetch unit, an adder, registers, latches, muxes, and a control unit. Most of the device definitions are self-explanatory with the exception of the memory and the control unit. These two units will be described in greater detail.

The system bus provides the following to the MMU:

a. A request line.

b. A supervisor state line.

c. Read/write/execute request type lines.

d. An address bus value.

e. A data bus value.

```
⊢def secUnit_spec r a b rwe ok =
    ∀ t. ok (t+1) =
      ((validAccess r) ((a t),(b t),(rwe t))  ∧  (ofsLEq r) ((a t),(b t)))

⊢def addUnit_spec r a b c = ∀ t:num. c (t+1) = (add r ( (a t),(b t) ))

⊢def muxUnit_spec r a b out w =
    ∀ t:num. (out (t+1)) = (w (t+1)) → address r(b (t+1)) | (a t)

⊢def mux3Unit_spec a b c out w =
    ∀ t:num. (out t) = (w t = 0) → a t | (w t = 1) → b t | c t

⊢def splitUnit_spec r virt id ofs =
     ∀ t:num. ((id t)  = (segIdshf  r) (virt t))  ∧
              ((ofs t) = (segOfs r) (virt t))

⊢def latchUnit_spec r i out ctrl =
      ∀ t:num. out (t+1) = ctrl (t+1) → out t | (i (t+1))

⊢def regUnit_spec r i ld clr out =
     (∀ t:num. out (t+1) = (clr t → (wordn r 0 )  | ld t  → i t  | out t) )  ∧
     (out 0 = (wordn r 0) )

⊢def matchUnit_spec r a b m =
    ∀ (t:num). m(t+1) = ( addrEq r (a t, b t) ) → T:bool | F"

⊢def oneUnit_spec r t = (wordn r) 1

⊢def bitFalse t = F
```

5.4 MEMORY

The memory unit specification defines an interface to memory that is synchronous. If the request line *req* is high at *t*, then at *t+1*, *data* will contain the requested memory value and the *done* line will be T. If there is no request at time *t*, then *done* at *t+1* will be F. To construct an asynchronous version, this specification could be modified to state that given a request at time *t*, the next time *done* is T *data* will hold the requested value from memory.

When composing the MMU with a cache, the synchronous specification will also change. If there is a cache hit, a value would be returned much sooner (perhaps an order of magnitude) than if main memory were to be accessed.

The control unit and the final correctness statement do not rely on a synchronous memory unit specification. The proof could be easily modified to fit these other models.

```
⊢def memoryUnit_spec r req addr data done mem =
     ( (data 0 = wordn r 0)  ∧  (done 0 = F) )  ∧
     ∀ t. ( (req t) → ( (data (t+1) = fetch r (mem t, addr t) )  ∧
                        (done (t+1) = T) ) |
                      ( (data (t+1) = wordn r 0)  ∧
                        (done (t+1) = F) ) )
```

5.5 CONTROL UNIT

To process each memory request, the control unit will pass through several phases. The unit is a clocked device. At each clock tick the control unit may change its phase depending on the results computed by the other internal units and the MMU input from the system bus.

The control unit inputs include:

a. The request line (reqIn).

b. The supervisor line (super).

c. The request type (read/write/execute) lines (rwe).

d. The address compare result line (match).

e. The security unit result line (secOk).

f. The memory fetch result line (fdone).

The control unit output lines include:

a. The MUXes that control the adder's inputs (muxC).

b. The adder output latch (lC).

c. The MUX that controls the bus memory address lines (xlat).

d. The register update lines (tmpC, tblC).

e. The memory request line (rReq).

f. The MMU done line (done).

g. The MMU access acknowledgment line (ack).

There are six distinct phases; however, not all phases are executed for each request. Which phases are executed depends on the validity of the memory request. Request evaluation begins with the control unit in phase 0 and completes when phase 0 is again reached. A valid request will require five phases with a delay of at least one time unit before a phase change. Most phases

require one clock cycle; however, memory requests for a segment descriptor may take several. The control unit will busy-wait until a memory fetch completes.

```
⊢def controlUnit_spec reqIn super rwe match secOK fdone
                  muxC tmpC tblC lC rReq xlat done ack phase =
   ((muxC 0,tmpC 0,tblC 0,lC 0,rReq 0,xlat 0,done 0,ack 0, phase 0) =
    ( 0    , F   , F   , F  , F    , F   ,  F  , F  ,   0    )  )
    ∧
   (∀ t .(muxC(t+1),tmpC(t+1),tblC(t+1),lC(t+1),rReq(t+1),xlat(t+1),done(t+1),
     ack(t+1),phase(t+1) ) =                  %  M  t t l  r x d a  P  %
                                              %  U  m b a  e l o c  H  %
                                              %  X  p l t  q t n k  A  %
      (phase t = 0) →
          (reqIn t →                           ( 0, F,F,F, F,F,F,F, 1)  |
                                               ( 0, F,F,F, F,F,F,F, 0)) |
      (phase t = 1) →
          (super t →
           ((vBIT (rwe t))  ∧  match t) →      ( 0, F,T,F, F,F,F,F, 5)  |
                                               ( 0, F,F,F, F,F,T,T ,0)  |
                                               ( 2, T,F,T, T,T,F,F, 2)) |
      ((phase t = 2)  ∧  fdone t) →            ( 1, F,F,F, T,T,F,F, 3)  |
      ((phase t = 3)  ∧  fdone t) →
            (secOK t  →                        ( 0, F,F,F, F,T,F,F, 4)  |
                                               ( 0, F,F,F, F,F,T,F, 0)) |
      (phase t = 4) →                          ( 0, F,F,T, F,T,T,T, 0)  |
      (phase t = 5) →                          ( 0, F,F,F, F,F,T,T ,0)  |
      (muxC t,tmpC t,tblC t,lC t, F ,xlat t,done t,ack t,phase t))
```

The dataPath definition describes the interconnection between all the units other than the control unit. The mmu_imp joins the control unit with the data path.

```
                        Data Path

⊢def dataPath  r vAddr vData rwe mem tblPtrADDR tblPtr rAddr
               muxC tmpC tblC lC rReq xlat match secOK fdone =
    ∃ (mux1 mux2 id ofs addOut data latOut :num→*wordn)
     (secData:num→*wordn).
    (regUnit_spec     r vData tblC bitFalse tblPtr)              ∧
    (regUnit_spec     r data  tmpC bitFalse secData)             ∧
    (secUnit_spec     r vAddr secData rwe secOK)                 ∧
    (splitUnit_spec   r vAddr id ofs)                            ∧
    (mux3Unit_spec    id  ofs (oneUnit_spec r) mux1 muxC)        ∧
    (mux3Unit_spec    tblPtr data   latOut     mux2 muxC)        ∧
    (addUnit_spec     r mux1 mux2 addOut)                        ∧
    (latchUnit_spec   r addOut latOut lC)                        ∧
    (matchUnit_spec   r vAddr tblPtrADDR match)                  ∧
    (muxUnit_spec     r vAddr latOut rAddr xlat)                 ∧
    (memoryUnit_spec  r rReq rAddr data fdone mem)
```

```
⊢def mmu_imp r vAddr vData rwe superv tblPtr tblPtrADDR reqIn
             rAddr done ack xlat mem phase =
    ∃ (muxC :num→num)(tmpC tblC lC rReq match secOK fdone :num→bool) .
    (controlUnit_spec reqIn superv rwe match secOK fdone
           muxC tmpC tblC lC rReq xlat done ack phase)        ∧
    (dataPath  r vAddr vData rwe mem tblPtrADDR tblPtr rAddr
           muxC tmpC tblC lC rReq xlat match secOK fdone)
```

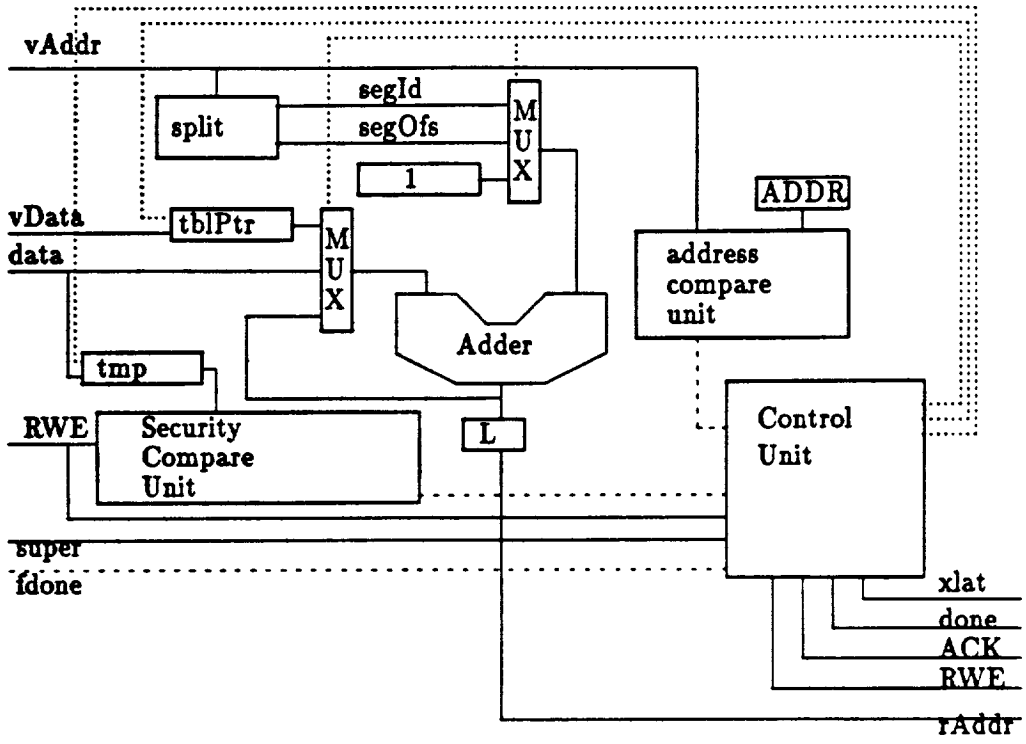


Figure 5.5-1: Abstract MMU Internal Block Diagram

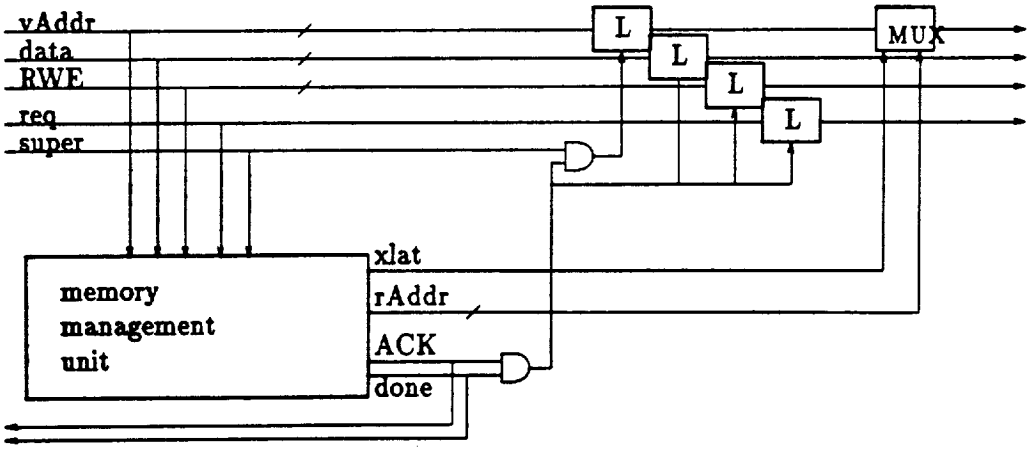


Figure 5.5-2: Abstract MMU External Block Diagram

5.6 MEMORY MANAGEMENT EXECUTION CYCLE

When the control unit is in phase 0, it will busy-wait for a request and then proceed to phase 1. During phase 0, the address comparison unit (`matchUnit_spec`) can determine whether the bus address matches the MMU's table pointer address. The result is put on the `match` line. The split unit `splitUnit_spec` divides the address into its segment table offset and segment offset components.

In phase 1, the supervisor line determines what the next phase will be. When the supervisor line is high, two results are possible. When the request is a write and the `match` line is T, the control unit will direct the table pointer register to store the value on the data bus. The control unit will set the next phase to 5. After one clock tick in phase 5, the acknowledge and done lines are asserted and the control unit returns to phase 0. This ensures the data bus value will remain constant while the register updates its store. If the request is not directed to the segment table pointer register, the done and acknowledge lines are asserted and the phase is set to 0. Since the `xlat` line remains F, the original request is effectively passed on to memory without modification.

During this time, the adder will compute the memory address of the segment descriptor using the shifted segment identifier and the segment table pointer (output from the MUXs). When the supervisor line is not high and the control unit is in phase 1, a memory fetch will be initiated using the adder output. The adder output latch control line is asserted to keep this value constant. The temporary register write control line (`tmpC`) will be asserted to capture the first word of the fetched segment descriptor. The control unit will move on to phase 2.

The control unit will remain in phase 2 until the `fdone` line is asserted indicating the memory fetch has completed. During this time, the adder will have incremented the address so that the second word of the segment descriptor can be fetched. The control unit will then move on to phase 3.

The control unit will also remain in phase 3 until the `fdone` line is asserted indicating the memory fetch has completed. If the security unit has asserted the `secOK` line, phase 4 is entered. The delay provides sufficient time for the adder to create the real address from the second word of the segment descriptor (fetched word) and the segment offset. In phase 4, the `xlat`, `done` and `ack` lines are asserted and the control unit returns to phase 0.

If the security unit does not authorize the memory request, the control unit does not enter phase 4, but, instead, returns to phase 0 asserting the `done` line, but not the `ack` line.

Note that the done line is asserted only when the MMU completes its execution cycle—and only for one clock cycle.

5.7 VERIFICATION

Several auxiliary definitions are used to express the final correctness statement. To relate the implementation to the specification, a temporal abstraction is constructed using the two predicates **Next** and **First**. The predicate **First** is true when its argument *t* is the first time that *g* is true. The predicate **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.

```
⊢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)

⊢def stable_sigs t1 t2 vAddr rwe tblPtrADDR data mem super =
    ∀ t'. t1 < t'  ∧  t' < t2 ⇒
      (super t' = super t1)  ∧
      (vAddr t' = vAddr t1)  ∧
      (rwe t' = rwe t1)      ∧
      (data t' = data t1)    ∧
      (mem t' = mem t1)      ∧
      (tblPtrADDR t' = tblPtrADDR t1)
```

The correctness theorem states that if the implementation is in phase 0 and a memory request is made, the implementation will respond *c* time steps later such that the state of the implementations matches the state defined by the specification for a set of given MMU inputs. The inputs must remain stable until the MMU responds to a request. If a memory request is not made, the acknowledgment line remains F, the phase remains 0 and the MMU table pointer register remains unchanged.

```
⊢ mmu_imp r vAddr vData rwe super tblPtr tblPtrADDR reqIn rAddr
          done ack xlat mem phase ⇒
    (∀ t.
      (phase t = 0) ⇒
        (reqIn t →
          (∃ c. Next done(t,t + c)  ∧  (phase(t + c) = 0)  ∧
          (stable_sigs t(t + c)vAddr rwe tblPtrADDR vData mem super ⇒
          (mmu_spec r (vAddr t) (rwe t) (tblPtrADDR t) (tblPtr t)
                      (vData t) (mem t) (super t) =
                ack(t + c),rAddr(t + c),tblPtr(t + c))))
           | ( (ack(t + 1) = F)   →
               (phase(t + 1) = 0) →
               (tblPtr(t + 1) = tblPtr t) )
       ))
```

Table 5.7-1: Abstract MMU Verification Script Run-Times

File name	*Time (CPU sec.)*	*Inferences*
mmu_abs	85.4	34
mmu_def	132.1	50
mmu_aux	81.6	4,385
ctrlUnit_lem	2,850.0	153,977
mmu_prf	2,665.5	122,537
	5,814.6	280,983

The correctness theorem required 2,635.2 seconds of CPU time running on a SPARCStation with 16 Mbytes of memory. HOL generated 121,858 primitive inferences to prove the theorem. Many lemmas were proven to support the final MMU correctness result. The proof effort was organized into a hierarchy of theories as presented in Table 5.7-1.

5.8 CONTROL UNIT LEMMAS

Control unit lemmas proven included the following:

a. Each phase was shown to be distinct.

b. The control unit phase state can be only one of six possible values.

c. Phase 0 can never follow phase 2.

d. During phase 0, the state of the MMU does not change.

e. A theorem showing a correct expansion of the control unit definition.

Table 5.8-1: Control Unit Theorems

Lemma	*Time (CPU sec.)*	*Inferences*
PHASE_0_UNIQUE	9.3	1,004
PHASE_1_UNIQUE	10.5	952
PHASE_2_UNIQUE	8.9	917
PHASE_3_UNIQUE	9.1	904
PHASE_4_UNIQUE	9.1	913
PHASE_5_UNIQUE	10.6	944
SIX_PHASES_ONLY	1,426.5	72,872
NOT_PHASE_2_THEN_0	112.0	6,820
PHASE_0_IDLE	1,146.5	65,672
CTRL_UNIT_EXPAND	35.5	2,774
	2,850.0	153,977

While the phase unique lemmas were trivial to prove, the other lemmas required substantial effort. A table listing the lemmas, the required CPU time to verify them and the number of intermediate theorems generated is presented in Table 5.8-1.

6.0 CONCLUDING REMARKS

Several enhancements could be made to the abstract MMU.

a. It would not be difficult to add a register that specified the number of valid entries in a segment table. The incoming segment id would be compared with this new value. When the id is greater than the stored value, the MMU could generate a segment table fault.

b. Another read-only status register could be added to indicate the type of fault that occurred.

c. A paging unit could be modeled based on the segment table unit. The device would effectively be the same as the segmentation unit. The stored real address offset might serve as the page table pointer.

d. Values were added together instead of being merged together, which is more common.

e. A cache could be added (see section on register stacks).

This research was intended to serve as a vehicle to investigate how we could reason about changes in a device under development. The compare units and the page check units demonstrate what changes to a proof are necessary for small device changes. What is of greater concern, however, is the construction of fire walls within a design; being able to recognize what effect a structural change would have and how to keep as much of an old proof as possible. The use of abstraction seems to satisfy these needs, as well as making proofs more tractable.

It also seems apparent that a generic execution tactic could be constructed to ease the pain of performing symbolic execution by hand. This would greatly simplify one of the most arduous tasks in interactive proof verification using HOL.

Abstract theories provide a mechanism to ignore many details that can be handled at lower levels of a design. For example, the abstract MMU focuses attention on the correctness of the control unit. Using the abstract theory package, abstract devices can be instantiated with verified gate level implementations of the abstracted functions.

The abstraction mechanism also permits design changes without the need for a complete reverification effort. The correctness theorem for the abstract MMU is not dependent on the layout of the segment protection descriptor or the specific protection requirements.

The basis for a secure hardware platform is a fully functional MMU. The MMU presented here serves as a model to verify a more sophisticated device, such as the hardware reference monitor

SIDEARM (ref. 24).

The MMUs verified provide sufficient hardware support for an operating system kernel to ensure process isolation and virtual memory. The device designs can be simplified to define a paging unit. Future work will investigate the composition of segmentation and paging units.

A register stack that implements a FIFO replacement strategy has also been verified. This is being enhanced to construct an MMU cache with either an LRU or LFU replacement strategy. Future work will investigate composing the MMU with the CPU and other chips to form a complete hardware base.

6.1 FUTURE WORK

One of the group's goals is to specify a set of chips that can work together as a system. The relationships between an MMU, an interrupt controller, a DMA controller, a memory, coprocessor chips (floating point processor), and the CPU were examined and several potential system integration problems were uncovered.

Further research will also examine how a set of processor specifications can be connected to create a system. A difficulty in composing independent processors occurs when they share state (e.g., memory, peripheral control registers). The proofs for each device make (legitimate) assumptions about the effects of device operations. These assumptions simplify the device proof but assume complete control over (now shared) state. We have defined some of the composition problems and are developing an interaction model based on a noninterference requirement.

REFERENCES

1. M. Gordon, "HOL: A Proof Generating System for Higher-Order Logic," in *VLSI Specification, Verification, and Synthesis* (G. Birtwhistle and P. Subrahmanyam, eds.), pp. 73–128, Kluwer Academic Press, 1988.

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

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

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

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

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

7. H. G. Barrow, "VERIFY: A Program for Proving Correctness of Digital Hardware Designs," *Artificial Intelligence*, vol. 24, 1984.

8. J. Joyce, "Formal Specification and Verification of Microprocessor Systems," *Microprocessing and Microprogramming, North-Holland*, (24)1988.

9. G. Milne and P. Subrahmanyam, *Formal Aspect of VLSI Design.* Publishers B.V., 1986.

10. D. Weise, "Functional Verification of MOS Circuits," *24th ACM/IEEE Design Automation Conference*, 1987.

11. 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.), pp. 27–71, Kluwer Academic Press, 1988.

12. W. A. Hunt, "A Verified Microprocessor," Tech. Rep. 47, The University of Texas at Austin, Dec. 1985.

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

14. W. J. Cullyer, "Implementing Safety Critical Systems: The VIPER Microprocessor," in *VLSI Specification, Verification, and Synthesis* (G. Birtwhistle and P. Subrahmanyam, eds.), pp. 1–25, Kluwer Academic Press, 1988.

15. 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.), pp. 1–91, Springer-Verlag, 1989.

16. J. J. Joyce, *Totally Verified Systems: Linking Verified Software to Verified Hardware.* Lecture Notes in Computer Science No. 408, Springer Verlag, July 1989.

17. W. R. Bevier, "A Verified Operating System Kernel," Tech. Rep. 11, Computational Logic, Inc., October 1989.

18. W. R. Bevier, "Kit and the Short Stack," *Journal of Automated Reasoning*, vol. 5, 1989.

19. 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.

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

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

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

23. P. J. Windley, "A Poor Man's Implementation of Abstract Theories," Tech. Rep. CSE-90-06, University of California, Davis, 1990.

24. W. Boebert, "The LOCK Demonstration," *11th National Computer Security Conference*, 1988.

APPENDIX A: BITVECTOR THEORY

```
system 'rm bitVector.th';;

new_theory 'bitVector';;


let ARB = new_definition
   ('ARB', "ARB = @ (x:bool) . F");;

let ZEROES = new_definition
   ('ZEROES',
    "! (w:num) (m:num) .
         ZEROES w m = (m <= w) => F | ARB");;

let ABS = new_definition
   ('ABS',
    "ABS (w:num) (sig:num->num->bool) (t:num) (n:num)
        = n <= w => sig n t | ARB");;

let bvPART = new_definition
  ('bvPART',
    "bvPART max min (sig:num->bool) (n:num)
        = (n > max) => F |
  (n < min) => F |
          sig n              ");;

let bvEQbit = new_definition
  ('bvEQbit_DEF',
   "bvEQbit x a b = a x = (b (x:num)):bool"
  );;

let bvEQUAL = new_prim_rec_definition
  ('bvEQUAL_DEF',
   "(bvEQUAL 0 a b = (a 0 = (b 0):bool)) /\
    (bvEQUAL (SUC n) a b = (bvEQUAL n a b /\ (a (SUC n) = (b (SUC n)))))"
  );;

let bvGREATER = new_prim_rec_definition
  ('bvGREATER_DEF',
   "(bvGREATER 0 a b = ( a 0 /\ ~b 0 ) ) /\
    (bvGREATER (SUC n) a b =
        ( ( a(SUC n)/\~b(SUC n) ) \/
         ( (a(SUC n)=b(SUC n)) /\ bvGREATER n a b)
        ))"
  );;

let bvLESS = new_definition
  ('bvLESS_DEF',
   "bvLESS n a b = bvGREATER n b a"
  );;


let bvPartEQUAL = new_prim_rec_definition
  ('bvPartEQUAL_DEF',
    "(bvPartEQUAL 0 y a b =
      ( (y = 0 ) =>  (bvEQbit 0 a b) | F )) /\
     (bvPartEQUAL (SUC x) y a b =
```

```
      (
       ((SUC x) > y) => (bvEQbit (SUC x) a b /\ (bvPartEQUAL x y a b)) |
       ((SUC x) = y) => (bvEQbit (SUC x) a b) | F
     )    )" );;

let bvPartGREATER = new_prim_rec_definition
  ('bvPartGREATER_DEF',
    "(bvPartGREATER (SUC x) y a b =
      (
       ((SUC x) > y) =>
          ( ( a(SUC x)/\~b(SUC x) ) \/
            ((a(SUC x)=b(SUC x)) /\ bvPartGREATER x y a b) ) |
       ((SUC x) = y) => (a(SUC x)/\~b(SUC x)) | F
     )    )" );;

let bvPartLESS = new_definition
  ('bvPartLESS_DEF',
   "bvPartLESS x y a b = bvPartGREATER x y b a"
  );;

close_theory();;
```

APPENDIX B: COMPARISON UNITS

```
loadf 'exist.tac.ml';;

system 'rm comparer.th';;

new_theory 'comparer';;

map load_parent ['gates'; 'bitVector'];;

let bitComp_spec = new_definition
  ('bitComp_spec',
   "! first sec g l e . bitComp_spec first sec g l e =
    (g = ( first /\ ~sec)) /\
    (l = ( ~first /\ sec)) /\
    (e = ( first = sec ))"
  );;

let bitComp_imp = new_definition
  ('bitComp_imp',
    "! first sec g l e . bitComp_imp first sec g l e =
     ? p q . (inv first p) /\ (inv sec q) /\
              (nor2 p sec   g) /\
              (nor2 q first l) /\
              (nor2 g l     e) "
   );;

let bitComp_correct = prove_thm
   ('bitComp_correct',
    "! first sec g l e.
bitComp_imp first sec g l e = bitComp_spec first sec g l e",
     REWRITE_TAC [ bitComp_imp; bitComp_spec; nor2; inv ]
     THEN REPEAT GEN_TAC
     THEN EXISTS_ELIM_TAC
     THEN REWRITE_TAC [DE_MORGAN_THM]
     THEN REWRITE_TAC [SPECL ["sec"; "~first"] CONJ_SYM]
     THEN EQ_TAC
     THEN STRIP_TAC
     THEN ASM_REWRITE_TAC []
     THEN MAP_EVERY BOOL_CASES_TAC ["first:bool"; "sec:bool"]
     THEN REWRITE_TAC []
    );;

%---------------------------------
bitComp_correct =
|- !first sec g l e.
    bitComp_imp first sec g l e = bitComp_spec first sec g l e
Run time: 35.5s
Intermediate theorems generated: 3470
---------------------------------%

let compComb_spec = new_definition
  ('compComb_spec',
   "! g0 g1 l0 l1 e0 e1 g l e . compComb_spec g0 g1 l0 l1 e0 e1 g l e =
    (g = (g1 \/ (e1 /\ g0))) /\
    (l = (l1 \/ (e1 /\ l0))) /\
    (e = (e1 /\ e0))"
  );;
```

```
let compComb_imp = new_definition
   ('compComb_imp',
    "! g0 g1 l0 l1 e0 e1 g l e . compComb_imp g0 g1 l0 l1 e0 e1 g l e =
     ? p q . (and2_imp e1 g0 p) /\ (or2_imp g1 p g) /\
            (and2_imp e1 l0 q) /\ (or2_imp l1 q l) /\
            (and2_imp e1 e0 e) "
   );;

let compComb_correct = prove_thm
   ('compComb_correct',
    "! g0 g1 l0 l1 e0 e1 g l e. compComb_imp g0 g1 l0 l1 e0 e1 g l e =
                                compComb_spec g0 g1 l0 l1 e0 e1 g l e",
    REWRITE_TAC [ compComb_imp; compComb_spec; and2_correct; or2_correct]
    THEN REWRITE_TAC [ and2_spec; or2_spec ]
    THEN REPEAT GEN_TAC
    THEN EXISTS_ELIM_TAC
    THEN PURE_ONCE_REWRITE_TAC
      [ SPECL [ "(l = l1 \/ e1 /\ l0)" ; "(g = g1 \/ e1 /\ g0)" ] CONJ_SYM]
    THEN PURE_ONCE_REWRITE_TAC [ SPECL [ "(e = e0 /\ e1)" ] CONJ_SYM]
    THEN REWRITE_TAC [ CONJ_ASSOC ]
   );;

%----------------------------------------
compComb_correct =
|- !g0 g1 l0 l1 e0 e1 g l e.
    compComb_imp g0 g1 l0 l1 e0 e1 g l e =
    compComb_spec g0 g1 l0 l1 e0 e1 g l e
Run time: 25.9s
Intermediate theorems generated: 2385
----------------------------------------%

let comp_spec = new_definition
   ('comp_spec',
    "! n a b g l e.
      comp_spec n a b g l e =
         ( g = ( bvGREATER n a b) ) /\
         ( l = ( bvLESS n a b) ) /\
         ( e = ( bvEQUAL n a b) )"
    );;

let comp_imp = new_prim_rec_definition
  ('comp_imp',
   "(comp_imp 0 a b gr ls eq = (bitComp_imp (a 0) (b 0) gr ls eq))/\
    (comp_imp (SUC n) a b gr ls eq =
      ? gn ln en gn ln en .
           (comp_imp n a b gn ln en) /\
           (bitComp_imp (a (SUC n)) (b (SUC n)) gn ln en)  /\
           (compComb_imp gn gn ln ln en en gr ls eq)
       )"
   );;


let compare_correct = prove_thm
   ('compare_correct',
    "!n a b great less equ.
       comp_imp n a b great less equ = comp_spec n a b great less equ",
  INDUCT_TAC
  THEN REPEAT GEN_TAC
```

```
 THEN REWRITE_TAC[comp_imp;comp_spec]
 THENL
  [% base case %
     REWRITE_TAC[bitComp_correct; bitComp_spec;
                 bvGREATER_DEF;bvLESS_DEF;bvEQUAL_DEF]
     THEN EQ_TAC THEN STRIP_TAC THEN ASM_REWRITE_TAC []
     THEN PURE_ONCE_REWRITE_TAC [ SPECL [ "~a 0" ] CONJ_SYM]
     THEN REWRITE_TAC [];
   ;% induction %
     REWRITE_TAC[comp_imp]
     THEN ASM_REWRITE_TAC[]
     THEN REWRITE_TAC[bitComp_correct;compComb_correct;
                      comp_spec; bitComp_spec; compComb_spec]
     THEN EXISTS_ELIM_TAC
     THEN REWRITE_TAC[bvGREATER_DEF;bvLESS_DEF;bvEQUAL_DEF]
     THEN EQ_TAC THEN STRIP_TAC THEN ASM_REWRITE_TAC []
     THEN PURE_ONCE_REWRITE_TAC [ SPECL [ "~a(SUC n)" ] CONJ_SYM]
     THEN PURE_ONCE_REWRITE_TAC [ SPECL [ "bvEQUAL n a b" ] CONJ_SYM]
     THEN REWRITE_TAC [] THEN EQ_TAC THEN STRIP_TAC
     THEN ASM_REWRITE_TAC []
    ]);;

%----------------------------------------------------------------------
compare_correct =
|- !n a b great less equ.
    comp_imp n a b great less equ = comp_spec n a b great less equ
Run time: 163.7s
Garbage collection time: 97.6s
Intermediate theorems generated: 13399
----------------------------------------------------------------------%

let bitEq_spec = new_definition
  ('bitEq_spec',
   "! first sec e . bitEq_spec first sec e =
    (e = ( (first:bool) = sec ))"
  );;

let bitEq_imp = new_definition
  ('bitEq_imp',
    "! first sec e . bitEq_imp first sec e =
     ? i j  . (nor2   first sec i) /\
              (and2_imp first sec j) /\
              (or2_imp i j e) "
   );;

let bitEq_correct = prove_thm
   ('bitEq_correct',
      "! first sec e.
bitEq_imp first sec e = bitEq_spec first sec e",
     REWRITE_TAC [ bitEq_imp; bitEq_spec; or2_correct;
                   nor2; and2_correct; inv; or2_spec; and2_spec]
     THEN REPEAT GEN_TAC
     THEN EXISTS_ELIM_TAC
     THEN MAP_EVERY BOOL_CASES_TAC ["first:bool"; "sec:bool"]
     THEN REWRITE_TAC []
);;


%---------------------------------
```

```
bitEq_correct =
|- !first sec e. bitEq_imp first sec e = bitEq_spec first sec e
Run time: 15.3s
Intermediate theorems generated: 1251
------------------------------------%

let compEq_spec = new_definition
   ('compEq_spec',
    "! n a b e.
      compEq_spec n a b e =
         ( e = ( bvEQUAL n a b) )"
    );;

let compEq_imp = new_prim_rec_definition
  ('compEq_imp',
   "(compEq_imp 0 a b eq = (bitEq_imp (a 0) (b 0) eq))/\
    (compEq_imp (SUC n) a b eq =
      ? en em .
           (compEq_imp n a b en) /\
           (bitEq_imp (a (SUC n)) (b (SUC n)) em)  /\
           (and2_imp en em eq)
        )"
   );;

let compEq_correct = prove_thm
  ('compEq_correct',
    "! n a b e. compEq_imp n a b e = compEq_spec n a b e",
    INDUCT_TAC
    THEN REPEAT GEN_TAC
    THEN ASM_REWRITE_TAC[compEq_imp;compEq_spec;bvEQUAL_DEF; and2_imp;
                         bitEq_correct; bitEq_spec; inv; nand2 ]
    THEN EXISTS_ELIM_TAC
    THEN PURE_ONCE_REWRITE_TAC [ SPECL [ "bvEQUAL n a b" ] CONJ_SYM]
    THEN REWRITE_TAC []
 );;

%-------------------------------------------------------
compEq_correct = |- !n a b e. compEq_imp n a b e = compEq_spec n a b e
Run time: 22.1s
Intermediate theorems generated: 1796
-------------------------------------------------------%

close_theory();;
```

APPENDIX C: PAGECHECK UNITS

```
system 'rm pgCk.th';;

loadf 'exist.tac.ml';;

new_theory 'pgCk';;

map load_parent ['gates';'bitVector';'comparer';'register'];;

let bitFalse = new_definition
   ('bitFalse', "!t . bitFalse t = F");;

%-------------------------------------------------------------
  pgCk specifies a (register/ack) pair for a (n/address/writeOp/register)
  input tuple

-------------------------------------------------------------%
let pgCk = new_definition
  ('pgCk', "!rgstr address write n. pgCk n address write rgstr =
      ((write = T)               => (address, T:bool) |
       (bvEQUAL n rgstr address) => (rgstr,   T) |
                                    (rgstr,   F)
      )"   );;

let pgCk_spec = new_definition
  ('pgCk_spec',
   "!(reg addr :num->num->bool) (rWC ack :num->bool) (n:num).
     pgCk_spec n addr rWC reg ack =
        !(t:num). (reg(t+1), ack(t+1)) =
          pgCk n (addr t) (rWC t) (reg t)"   );;

let pgCk_imp = new_definition
  ('pgCk_imp',
    "!reg addr rWC n ack. pgCk_imp n addr rWC reg ack =
      !t.
      (?g l e.
       (reg_imp n addr rWC bitFalse reg ) /\
       (comp_imp n (ABS n reg t) (ABS n addr t) g l e) /\
       (or2_imp e (rWC t) (ack (t+1)) )
      )"
   );;

let pgCk_correct = prove_thm
   ('pgCk_correct',
    "!reg addr rWC n ack . pgCk_imp n addr rWC reg ack ==>
 pgCk_spec n (ABS n addr) rWC (ABS n reg) ack",
    REPEAT GEN_TAC
    THEN REWRITE_TAC [ pgCk_imp; pgCk_spec; pgCk ]
    THEN REWRITE_TAC [ compare_correct; reg_correct ; or2_correct ]
    THEN REWRITE_TAC [ reg_spec; comp_spec; or2_spec]
    THEN REWRITE_TAC [ bitFalse ]
    THEN EXISTS_ELIM_TAC
    THEN STRIP_TAC
    THEN GEN_TAC
    THEN ASM_CASES_TAC "(rWC t):bool"
    THEN ASM_REWRITE_TAC □
    THEN ASM_CASES_TAC "(bvEQUAL n(ABS n reg t)(ABS n addr t)):bool"
```

```
   THEN ASM_REWRITE_TAC[]
  );;

%-------------NOW add a supervisor line----------%

let pgCka_spec = new_definition
  ('pgCka_spec',
   "!(reg addr :num->num->bool) (sup rWC ack :num->bool) (n:num).
     pgCka_spec n addr rWC sup reg ack =
        !(t:num). (reg(t+1), ack(t+1)) =
          pgCk n (addr t) (rWC t /\ sup t) (reg t)"   );;

let pgCka_imp = new_definition
  ('pgCka_imp',
    "!reg addr rWC sup n ack. pgCka_imp n addr rWC sup reg ack =
      !t.
      (? x g l e.
       (and2_imp (rWC t) (sup t) (x t) ) /\
       (reg_imp n addr x bitFalse reg ) /\
       (comp_imp n (ABS n reg t) (ABS n addr t) g l e) /\
       (or2_imp e (x t) (ack (t+1)) )
      )"   );;

let pgCka_correct = prove_thm
   ('pgCka_correct',
     "!reg addr rWC sup n ack . pgCka_imp n addr rWC sup reg ack ==>
 pgCka_spec n (ABS n addr) rWC sup (ABS n reg) ack" ,
    REPEAT GEN_TAC
    THEN ONCE_REWRITE_TAC [ pgCka_imp; pgCka_spec ]
    THEN ONCE_REWRITE_TAC [pgCk ]
    THEN ONCE_REWRITE_TAC
          [ compare_correct; reg_correct ; or2_correct; and2_correct ]
    THEN ONCE_REWRITE_TAC [ reg_spec; comp_spec; or2_spec; and2_spec]
    THEN REWRITE_TAC [ bitFalse ]
    THEN EXISTS_ELIM_TAC
    THEN REPEAT STRIP_TAC
    THEN POP_ASSUM(\thm. STRIP_ASSUME_TAC (SPEC_ALL thm))
    THEN ASSUM_LIST(\asl. REWRITE_TAC
        [(REWRITE_RULE [el 2 asl] (el 3 asl))])
    THEN MAP_EVERY ASM_CASES_TAC [ "(rWC t):bool"; "(sup t):bool"]
    THEN ASSUM_LIST(\thl. ASSUME_TAC  (REWRITE_RULE [
  (REWRITE_RULE [(el 1 thl); (el 2 thl)] (el 4 thl) ) ] (el 5 thl) ))
    THEN ASM_REWRITE_TAC []
    THEN ASM_CASES_TAC "(bvEQUAL n(ABS n reg t)(ABS n addr t)):bool"
    THEN ASM_REWRITE_TAC[]
 );;
```

APPENDIX D: BASE AND BOUNDS CHECK UNIT

```
system 'rm mmu.th';;

loadf 'exist.tac.ml';;

new_theory 'mmu';;

map load_parent ['gates';'bitVector';'comparer';'register'];;

let bitFalse = new_definition
   ('bitFalse', "!t . bitFalse t = F");;

%-------------------------------------------------------------
baseBounds MMU
   input:
 addr, offset, data, supervisor state, read/write request,
        ADDR of register
 s: defines number of bits defining segment size

  output:
        ack
  internal state:
        baseBounds register
-------------------------------------------------------------%

let bbSUPERV = new_definition
  ('bbSUPERV',
   "!(bbReg addr data :num->bool)
      (ADDR :num->bool) (rw:bool) (n:num).
     bbSUPERV n bbReg addr data ADDR rw =
      ( rw => ((bvEQUAL n addr ADDR) => (data, T:bool) | (bbReg, T)  )|
       (bbReg, T)   )"     );;

let bbCOMP = new_definition
  ('bbCOMP',
   "!bbReg addr n s.
     bbCOMP n s bbReg addr =
       ( (bvEQUAL n (bvPART n s bbReg)(bvPART n s addr) /\ ~(bvGREATER s addr bbReg) )
   => (bbReg, T:bool) | (bbReg, F))");;

let bbNextState = new_definition
  ('bbNextState',
   "!(bbReg addr data :num->bool)
      (ADDR :num->bool) (super rw ack :bool) (n s :num).
     bbNextState n s bbReg addr data ADDR super rw =
      ( super => bbSUPERV n bbReg addr data ADDR rw |
         bbCOMP   n s bbReg addr )"     );;

let baseBoundCk_spec = new_definition
  ('baseBoundCk_spec',
   "!(bbReg addr data :num->num->bool)(ADDR :num->bool)
    (super rw ack :num->bool) (n s:num).
     baseBoundCk_spec n s bbReg addr data ADDR super rw ack =
       (s < n) ==>
       !t. ( bbReg(t+1),ack(t+1) ) =
        bbNextState n s (bbReg t) (addr t) (data t) ADDR (super t) (rw t)");;
```

```
let PRT = new_definition
  ('PRT',
    "PRT w max min (sig:num->num->bool) (t:num) (n:num)
        = (n > max) => F |
  (n < min) => F |
          (n <=  w) => (sig n t) |   ARB ");;

let baseBoundCk_imp = new_definition
  ('baseBoundCk_imp',
   "!(bbReg addr data :num->num->bool)(ADDR :num->bool)
    (super rw ack :num->bool) (n s:num).
     baseBoundCk_imp n s bbReg addr data ADDR super rw ack =
       (s < n) ==>       !t.
      (? writeBB g0 g1 g2 10 11 12 e2 x addrMatch goodSeg goodOfs ok.
         (reg_imp n data writeBB bitFalse bbReg) /\
         (comp_imp n (ABS n addr t) ADDR g0 10 (addrMatch t)) /\
         (and2_imp (rw t) (super t) (x t)) /\
         (and2_imp (addrMatch t) (x t) (writeBB t)) /\
         (comp_imp n  (PRT n n s bbReg t)
                      (PRT n n s addr  t)  g1 11 goodSeg) /\
         (comp_imp s  (ABS n addr  t)
                      (ABS n bbReg t)  g2 12 e2) /\
         (inv g2 goodOfs) /\
 (and2_imp goodOfs goodSeg ok) /\
         (or2_imp ok (super t) (ack (t+1)) )
       )");;

%--------------------prove some lemmas--------------------%

let mmLemma0 = prove_thm
  ('mmLemma0',
    "!(n s t:num) (sig:num->num->bool). (s < n) ==>
((PRT n n s sig t) = (bvPART n s(ABS n sig t) ) )",
   INDUCT_TAC
   THEN REPEAT GEN_TAC
   THEN REWRITE_TAC [NOT_LESS_0]
   THEN STRIP_TAC
   THEN CONV_TAC(DEPTH_CONV FUN_EQ_CONV)
   THEN GEN_TAC
   THEN REWRITE_TAC [PRT;bvPART;ABS]
 );;

let mmLemma1 = prove_thm
  ('mmLemma1',
   "(bvEQUAL n(bvPART n s(ABS n bbReg t))(bvPART n s(ABS n addr t)) /\
     ~bvGREATER s(ABS n addr t)(ABS n bbReg t)) =
     (~bvGREATER s(ABS n addr t)(ABS n bbReg t)) /\
    (bvEQUAL n(bvPART n s(ABS n bbReg t))(bvPART n s(ABS n addr t)))",
  ONCE_REWRITE_TAC [
        SPEC "~bvGREATER s(ABS n addr t)(ABS n bbReg t)" CONJ_SYM]
  THEN  REFL_TAC
);;

let mmLemma2 = prove_thm
  ('mmLemma2',
   "! (n:num). (n > 0 ) ==> ( ((SUC (PRE n) -1) + 1 = (SUC(PRE n))  ))",
    GEN_TAC
    THEN ASM_CASES_TAC "n>0"
    THEN ASM_REWRITE_TAC []
```

```
    THEN REWRITE_TAC [SUC_SUB1]
    THEN REWRITE_TAC [num_CONV "1"]
    THEN REWRITE_TAC [ADD_CLAUSES]
);;

let muLemma3 = prove_thm
  ('muLemma3',
   "! (n:num). (n > 0 ) ==> ( (SUC (PRE n)) =  n )",
    GEN_TAC
    THEN ASM_CASES_TAC "n>0"
    THEN ASM_REWRITE_TAC []
    THEN  REWRITE_TAC [PRE_SUB1;ADD1]
    THEN REWRITE_TAC [num_CONV "1"]
    THEN POP_ASSUM(\thm. ASSUME_TAC
      (REWRITE_RULE [thm] (SPECL ["n"; "0"] GREATER) ))
    THEN POP_ASSUM(\thm. ASSUME_TAC
      (REWRITE_RULE [thm] (SPECL ["0"; "n"] LESS_EQ) ))
    THEN POP_ASSUM(\thm. REWRITE_TAC [
      (REWRITE_RULE [thm] (SPECL ["n";"(SUC 0)"] SUB_ADD) )])
);;


%-------------------- prove baseBoundCk_correct ----------------------%

let baseBoundCk_correct = prove_thm
  ('baseBoundCk_correct',
   "!(bbReg addr data :num->num->bool)(ADDR :num->bool)
     (super rw ack :num->bool) (n s:num).
     baseBoundCk_imp n s bbReg addr data ADDR super rw ack ==>
     baseBoundCk_spec n s (ABS n bbReg) (ABS n addr) (ABS n data)
                      ADDR super rw ack",
   REWRITE_TAC [baseBoundCk_imp; baseBoundCk_spec]
   THEN REPEAT GEN_TAC
   THEN ASM_CASES_TAC "(s < n)"
   THEN ASM_REWRITE_TAC []
   THEN ONCE_REWRITE_TAC [bbNextState]
   THEN ONCE_REWRITE_TAC [bbSUPERV; bbCOMP]
   THEN ONCE_REWRITE_TAC
        [and2_correct; reg_correct; compare_correct; or2_correct; inv]
   THEN ONCE_REWRITE_TAC [and2_spec; or2_spec; reg_spec; comp_spec]
   THEN REWRITE_TAC [ bitFalse ]
   THEN EXISTS_ELIM_TAC
   THEN REPEAT STRIP_TAC
   THEN POP_ASSUM(\thm. STRIP_ASSUME_TAC (SPEC_ALL thm))
   THEN  MAP_EVERY ASM_CASES_TAC [ "(rw t):bool"; "(super t):bool"]
   THENL [
     % 1/4 %
      ASM_CASES_TAC "bvEQUAL n(ABS n addr t)ADDR"
      THEN ASM_REWRITE_TAC []
     ;% 2/4 %
      ALL_TAC
     ;% 3/4 %
      ASSUM_LIST(\asl. REWRITE_TAC [ (el 1 asl); (el 2 asl)] )
      THEN ASSUM_LIST(\thl. ASSUME_TAC (REWRITE_RULE [(el 1 thl)] (el 5 thl) ))
      THEN ASSUM_LIST(\thl. ASSUME_TAC (REWRITE_RULE [(el 3 thl)] (el 5 thl) ))
      THEN ASSUM_LIST(\thl. ASSUME_TAC
        (REWRITE_RULE [(el 1 thl)] (SPEC "t" (el 5 thl)) ))
      THEN ASSUM_LIST(\thl. REWRITE_TAC [(el 1 thl) ; (el 3 thl)] )
     ;% 4/4 %
```

```
      ALL_TAC
   ]  % cases 2 and 4 remain %
   THEN REWRITE_TAC [mmuLemma1]
   THEN ASSUM_LIST(\asl. REWRITE_TAC [ (el 1 asl); (el 2 asl)] )
   THEN ASSUM_LIST(\thl. ASSUME_TAC (REWRITE_RULE [(el 1 thl)] (el 5 thl) ))
   THEN ASSUM_LIST(\thl. ASSUME_TAC (REWRITE_RULE [(el 2 thl)] (el 5 thl) ))
   THEN ASSUM_LIST(\thl. ASSUME_TAC
     (REWRITE_RULE [(el 1 thl)] (SPEC "t" (el 5 thl)) ))
   THEN  ASSUM_LIST(\thl. ASSUME_TAC
     (REWRITE_RULE [(el 10 thl)] (SPECL ["n";"s";"t"] mmuLemma0) ))
   THEN ASSUM_LIST(\thl. ASSUME_TAC (REWRITE_RULE [(el 1 thl)] (el 4 thl) ))
   THEN ASM_CASES_TAC "ack(t+1):bool"
   THEN ASSUM_LIST(\thl. REWRITE_TAC [(el 1 thl) ; (el 4 thl);
     (REWRITE_RULE [(el 1 thl)] (el 2 thl))  ] )
);;


%-------------------------------------------------------

baseBoundCk_correct =
|- !bbReg addr data ADDR super rw ack n s.
    baseboundck_imp n s bbreg addr data addr super rw ack ==>
    baseboundck_spec
    n
    s
    (abs n bbreg)
    (abs n addr)
    (abs n data)
    addr
    super
    rw
    ack
run time: 492.7s
garbage collection time: 347.8s
intermediate theorems generated: 31227

-------------------------------------------------------%
```

APPENDIX E: VIRTUAL ADDRESS TRANSLATION UNIT

```
set_flag('print_all_subgoals', false);;

system 'rm mmu.th';;

loadf 'exist.tac.ml';;

new_theory 'mmu';;

map load_parent ['gates';'bitVector';'comparer';'register'];;

let bitFalse = new_definition
   ('bitFalse', "!t . bitFalse t = F");;

%-------------------------------------------------------------
baseBounds MMU with virtual address translation
-------------------------------------------------------------%

let vSUPERV = new_definition
  ('vSUPERV',
   "!(bbReg vaReg addr data :num->bool)
      (ADDR :num->bool) (rw:bool) (n:num).
     vSUPERV n bbReg vaReg addr data ADDR rw =
      ( (rw /\ (bvEQUAL n (bvPART n 1 addr) (bvPART n 1 ADDR) ))
   => (addr 0)  => (data,  vaReg, addr, T:bool) |
                   (bbReg, data,  addr, T:bool) |
      (bbReg, vaReg, addr, T)   )"     );;

let VtoR = new_definition
  ('VtoR',
    "VtoR realA virtA s n
       = (n > s) =>  (realA n):bool |
                     (virtA n)"    );;

let vCOMP = new_definition
  ('vCOMP',
   "!bbReg vaReg addr n s.
     vCOMP n s bbReg vaReg addr =
      ( (bvEQUAL n (bvPART n s bbReg)(bvPART n s addr) /\
        ~(bvGREATER s addr bbReg) )
       => (bbReg, vaReg, (VtoR vaReg addr s), T:bool) |
          (bbReg, vaReg, addr, F))");;

let vNextState = new_definition
  ('vNextState',
   "!(bbReg vaReg addr data :num->bool)
      (ADDR :num->bool) (super rw ack :bool) (n s :num).
     vNextState n s bbReg vaReg addr data ADDR super rw =
      ( super => vSUPERV n bbReg vaReg addr data ADDR rw |
         vCOMP   n s bbReg vaReg addr )"     );;

let virtBBCk_spec = new_definition
  ('virtBBCk_spec',
   "!(bbReg vaReg addr data outAddr :num->num->bool)(ADDR :num->bool)
    (super rw ack :num->bool) (n s:num).
     virtBBCk_spec n s bbReg vaReg addr data ADDR super rw ack outAddr=
       (s < n) ==>
```

```
      !t. ( bbReg(t+1),vaReg(t+1), outAddr(t+1), ack(t+1) ) =
       vNextState n s (bbReg t) (vaReg t) (addr t) (data t)
                  ADDR (super t) (rw t)");;

let PRT = new_definition
  ('PRT',
    "PRT w max min (sig:num->num->bool) (t:num) (n:num)
        = (n > max) => F |
  (n < min) => F |
         (n <=  w) => (sig n t) |   ARB ");;

let PRTA = new_definition
  ('PRTA',
    "PRTA w max min (sig:num->bool) (n:num)
        = (n > max) => F |
  (n < min) => F |
         (n <=  w) => (sig n) |   ARB ");;

let pick_imp = new_definition
    ('pick_imp',
      "pick_imp (wordA :num->bool) (wordB :num->bool) (which:bool) res
       = (which = T) => (res = wordA) | (res = wordB)");;

let virtBBCk_imp = new_definition
  ('virtBBCk_imp',
   "!(bbReg vaReg addr data outAddr :num->num->bool)(ADDR :num->bool)
    (super rw ack :num->bool) (n s:num).
     virtBBCk_imp n s bbReg vaReg addr data ADDR super rw ack outAddr=
       (s < n) ==>      !t.
      (? wBB wVA select x aM0 aM1 aM2 goodSeg goodOfs ok nok nxlat g l e.
         (and2_imp (rw t) (super t) (x t)) /\
         (compEq_imp n (PRT n n 1 addr t) (PRTA n n 1 ADDR) (aM0 t)) /\
         (and2_imp (aM0 t) (x t) (aM1 t)) /\
         (inv (addr 0 t) (aM2 t) ) /\
         (and2_imp (aM1 t) (addr 0 t) (wBB t)) /\
         (and2_imp (aM1 t) (aM2  t)   (wVA t)) /\
         (reg_imp n data wBB bitFalse bbReg) /\
         (reg_imp n data wVA bitFalse vaReg) /\
         (compEq_imp n  (PRT n n s bbReg t)
                        (PRT n n s addr  t) goodSeg) /\
         (comp_imp s    (ABS n addr  t)
                        (ABS n bbReg t)  g l e) /\
         (inv g goodOfs) /\
  (and2_imp goodOfs goodSeg ok) /\
         (or2_imp ok (super t) (ack (t+1)) ) /\
  (inv ok nok ) /\
         (or2_imp nok (super t) nxlat) /\
         (pick_imp (ABS n addr t) (ABS n vaReg t) nxlat (select t)) /\
         ( (outAddr (t+1))= (VtoR (select t) (ABS n addr t) s ) )
       )");;

%--------------------prove some lemmas-------------------%

let mmuLemma0 = prove_thm
  ('mmuLemma0',
   "!(n s t:num) (sig:num->num->bool).
     (PRT n n s sig t) = (bvPART n s(ABS n sig t) )",
   CONV_TAC(DEPTH_CONV FUN_EQ_CONV)
   THEN GEN_TAC
```

```
  THEN REWRITE_TAC [PRT;bvPART;ABS]
 );;

let mmuLemma1 = prove_thm
  ('mmuLemma1',
   "(bvEQUAL n(bvPART n s(ABS n bbReg t))(bvPART n s(ABS n addr t)) /\
     ~bvGREATER s(ABS n addr t)(ABS n bbReg t)) =
     (~bvGREATER s(ABS n addr t)(ABS n bbReg t)) /\
    (bvEQUAL n(bvPART n s(ABS n bbReg t))(bvPART n s(ABS n addr t)))",
  ONCE_REWRITE_TAC [
       SPEC "~bvGREATER s(ABS n addr t)(ABS n bbReg t)" CONJ_SYM]
  THEN  REFL_TAC
);;

let mmuLemma2 = prove_thm
  ('mmuLemma2',
   "VtoR a a s = a",
   CONV_TAC(DEPTH_CONV FUN_EQ_CONV)
   THEN  REWRITE_TAC [VtoR]
   THEN GEN_TAC
   THEN BOOL_CASES_TAC "n > s"
   THEN REWRITE_TAC []
);;

let mmuLemma3 = prove_thm
  ('mmuLemma3',
   "!(n s :num) (sig:num->bool). (PRTA n n s sig) = (bvPART n s sig)",
   CONV_TAC(DEPTH_CONV FUN_EQ_CONV)
   THEN REPEAT GEN_TAC
   THEN REWRITE_TAC [PRTA;bvPART]
   THEN ASM_CASES_TAC "(n' > n)"
   THEN ASM_REWRITE_TAC []
   THEN ASSUM_LIST(\asl. ASSUME_TAC(
      REWRITE_RULE [ (SPECL ["n'";"n"] GREATER) ] (el 1 asl) ))
   THEN ASSUM_LIST(\asl. REWRITE_TAC[
      REWRITE_RULE [ (el 1 asl) ] (SPECL ["n";"n'"] LESS_CASES) ])
);;

let mmuLemma4 = prove_thm
  ('mmuLemma4',
   "addr 0 t = ABS n addr t 0",
   ONCE_REWRITE_TAC [ABS]
   THEN ONCE_REWRITE_TAC [SPECL["0";"n"] LESS_OR_EQ]
   THEN REWRITE_TAC [
     REWRITE_RULE[SPEC "(0=n)" DISJ_SYM] (SPECL["n"] LESS_0_CASES)]
);;

%-------------------- prove correct ----------------------%

let virtBB_correct = prove_thm
   ('virtBB_correct',
   "!(bbReg vaReg addr data outAddr :num->num->bool)(ADDR :num->bool)
    (super rw ack :num->bool) (n s:num).
    virtBBCk_imp n s bbReg vaReg addr data ADDR super rw ack outAddr ==>
    virtBBCk_spec n s (ABS n bbReg) (ABS n vaReg) (ABS n addr)
                      (ABS n data) ADDR super rw ack outAddr",
   REWRITE_TAC [virtBBCk_imp; virtBBCk_spec]
   THEN REPEAT GEN_TAC
   THEN ASM_CASES_TAC "(s < n)"
```

```
  THEN ASM_REWRITE_TAC []
  THEN ONCE_REWRITE_TAC [vNextState]
  THEN ONCE_REWRITE_TAC [vSUPERV; vCOMP]
  THEN ONCE_REWRITE_TAC
       [and2_correct; reg_correct; compare_correct;
        compEq_correct; or2_correct; inv]
  THEN ONCE_REWRITE_TAC [and2_spec; or2_spec; reg_spec;
                         comp_spec; compEq_spec; pick_imp]
  THEN REWRITE_TAC [ bitFalse ]
  THEN EXISTS_ELIM_TAC
  THEN REPEAT STRIP_TAC
  THEN POP_ASSUM(\thm. ASSUME_TAC (SPEC_ALL thm))
  THEN MAP_EVERY ASM_CASES_TAC ["(super t):bool"; "(rw t):bool"]
  THEN ASSUM_LIST(\asl. STRIP_ASSUME_TAC
       (REWRITE_RULE [(el 1 asl);(el 2 asl)] (el 3 asl)))
  THEN POP_ASSUM_LIST(\asl.
       MAP_EVERY ASSUME_TAC(rev( subtract asl[(el 12 asl)])))
  THENL
    [   % 1/4 (super t) (rw t) %
  ASSUM_LIST(\asl.
          REWRITE_TAC [(el 6 asl);(el 10 asl);(el 11 asl)] )
  THEN ASM_CASES_TAC "bvEQUAL n(PRT n n 1 addr t)(PRTA n n 1 ADDR)"
  THEN ASSUM_LIST(\thl. REWRITE_TAC [ REWRITE_RULE
           [(SPECL ["n";"1";"ADDR"] muLemma3);
            (SPECL ["n";"1";"t";"addr"] muLemma0)] (el 1 thl) ])
  THEN ASSUM_LIST(\thl. ASSUME_TAC
          (REWRITE_RULE [muLemma2;(el 2 thl)] (el 8 thl)))
  THENL [
ASM_CASES_TAC "(addr 0 t):bool"
       THEN ASSUM_LIST(\thl. ASSUME_TAC (REWRITE_RULE
           [(REWRITE_RULE [(el 1 thl);(el 3 thl)] (el 7 thl) )]
           (SPEC "t" (el 5 thl)) ))
       THEN ASSUM_LIST(\asl. REWRITE_TAC
           [REWRITE_RULE [muLemma4] (el 2 asl)] )
       THEN ASSUM_LIST(\thl. ASSUME_TAC (REWRITE_RULE
           [(REWRITE_RULE [(el 2 thl);(el 4 thl)] (el 9 thl) )]
           (SPEC "t" (el 7 thl)) ))
THEN ASSUM_LIST(\thl. REWRITE_TAC
           [PAIR_EQ; (el 1 thl); (el 2 thl); (el 4 thl)])
      ;
ASSUM_LIST(\thl. ASSUME_TAC (REWRITE_RULE
           [(REWRITE_RULE [(el 2 thl);(el 3 thl)] (el 7 thl) )]
    (SPEC "t" (el 5 thl)) ))
       THEN ASSUM_LIST(\thl. ASSUME_TAC (REWRITE_RULE
           [(REWRITE_RULE [(el 3 thl);(el 4 thl)] (el 7 thl) )]
            (SPEC "t" (el 5 thl)) ))
THEN ASSUM_LIST(\thl. REWRITE_TAC
           [PAIR_EQ;(el 1 thl);(el 2 thl);(el 3 thl)])
      ]
    ; % 2/4    super t /\ ~rw t %
     ASM_REWRITE_TAC[muLemma2]
    ; % 3/4   ~super t /\  rw t %
     ALL_TAC
    ; %  4/4   ~super t /\ ~rw t %
     ALL_TAC
   ]
  THEN ASSUM_LIST(\asl. (REWRITE_TAC [ (el 10 asl); (el 11 asl); muLemma1;
 (REWRITE_RULE [(el 5 asl)] (SPEC "t" (el 3 asl)));
        (REWRITE_RULE [(el 4 asl)] (SPEC "t" (el 2 asl)))]))
```

```
  THEN ASM_CASES_TAC "(~bvGREATER s(ABS n addr t)(ABS n bbReg t) /\
        bvEQUAL n(PRT n n s bbReg t)(PRT n n s addr t))"
  THEN ASSUM_LIST(\asl. REWRITE_TAC[ REWRITE_RULE [muLemma0] (el 1 asl)])
  THEN ASSUM_LIST(\asl. REWRITE_TAC[ REWRITE_RULE [(el 1 asl)] (el 7 asl)])
  THEN ASSUM_LIST(\asl. REWRITE_TAC [ muLemma2; (REWRITE_RULE
       [REWRITE_RULE [(el 1 asl)] (el 2 asl)] (el 8 asl)) ] )
);;

%-----------------------------------------------
virtBB_correct =
|- !bbReg vaReg addr data outAddr ADDR super rw ack n s.
    virtBBCk_imp n s bbReg vaReg addr data ADDR super rw ack outAddr ==>
    virtBBCk_spec
    n
    s
    (ABS n bbReg)
    (ABS n vaReg)
    (ABS n addr)
    (ABS n data)
    ADDR
    super
    rw
    ack
    outAddr
Run time: 1209.0s
Garbage collection time: 734.6s
Intermediate theorems generated: 64185
-----------------------------------------------%

close_theory();;

%-------------- work space

let muLemma0 = prove_thm
  ('muLemma0',
    "!(n s t:num) (sig:num->num->bool). (s < n) ==>
((PRT n n s sig t) = (bvPART n s(ABS n sig t) ) )",
    INDUCT_TAC
    THEN REPEAT GEN_TAC
    THEN REWRITE_TAC [NOT_LESS_0]
    THEN STRIP_TAC
    THEN CONV_TAC(DEPTH_CONV FUN_EQ_CONV)
    THEN GEN_TAC
    THEN REWRITE_TAC [PRT;bvPART;ABS]
  );;

let muLemma3 = prove_thm
  ('muLemma3',
    "!(n s :num) (sig:num->bool). (s < n) ==>
((PRTA n n s sig) = (bvPART n s sig) )",
    INDUCT_TAC
    THEN REPEAT GEN_TAC
    THEN REWRITE_TAC [NOT_LESS_0]
    THEN STRIP_TAC
    THEN CONV_TAC(DEPTH_CONV FUN_EQ_CONV)
    THEN GEN_TAC
```

```
   THEN REWRITE_TAC [PRTA;bvPART]
   THEN ASM_CASES_TAC "(n' > (SUC n))"
   THEN ASM_REWRITE_TAC []
   THEN ASSUM_LIST(\asl. ASSUME_TAC(
      REWRITE_RULE [ (SPECL ["n'";"(SUC n)"] GREATER) ] (el 1 asl) ))
   THEN ASSUM_LIST(\asl. REWRITE_TAC[
      REWRITE_RULE [ (el 1 asl) ] (SPECL ["(SUC n)";"n'"] LESS_CASES) ])
);;

------------%
```

APPENDIX F: ABSTRACT MEMORY MANAGEMENT UNIT

mmu_abs.ml

```
let Library_Root = '/epoch/d1/csgrad/schubert/hol/Library/';;

let lib_dir_list =
       (map (concat Library_Root)
    ['gates/'; 'bits/';'words/';'numbers/'; 'decimal/'; 'assoc/']);;

set_search_path (search_path() @ ['.';
                                  '/epoch/d1/csgrad/schubert/hol/tactics/';
                                  '/epoch/d1/csgrad/schubert/hol/ml/';
                                  '/epoch/d1/csgrad/schubert/hol/theories/';
                                  '/epoch/d1/csgrad/schubert/hol/lisp/vax/';
                                 ]
                               @ lib_dir_list);;

loadf('aux_defs.ml');;

system 'rm /epoch/d1/csgrad/schubert/hol/theories/mmu_abs.th';;

new_theory  'mmu_abs';;

loadf 'abstract';;

new_type_abbrev ('RWE', ":bool#bool#bool");;

let mmu_abs = new_abstract_representation
 [
  ('segId',          ":(*address -> *wordn)"                 );
  ('segOfs',         ":(*address -> *wordn)"                 );
  ('segIdshf',       ":(*address -> *wordn)"                 );
% %
  ('availBit',       ":(*wordn -> bool)"                     );
  ('readBit',        ":(*wordn -> bool)"                     );
  ('writeBit',       ":(*wordn -> bool)"                     );
  ('execBit',        ":(*wordn -> bool)"                     );
% %
  ('add',            ":(*wordn # *wordn ->*wordn)"         );
% %
  ('addrEq',         ":(*address # *address  -> bool)"       );
  ('ofsLEq',        ":(*address # *wordn    -> bool)"      );
  ('validAccess',    ":(*address # *wordn # RWE  -> bool)"  );
 % Coercion functions %
  ('val',            ":(*wordn -> num)"                      );
  ('wordn',          ":(num->  *wordn)"                      );
  ('address',        ":(*wordn -> *address)"                 );
 % Memory functions %
  ('fetch',          ":(*memory # *address) -> *wordn"       );
  ('trans',          ":*memory -> *memory"                   );
 ];;

 let mmu_ty = abstract_type 'mmu_abs' 'segId';;

 close_theory();;
```

mmu_def.ml

```
let Library_Root = '/epoch/d1/csgrad/schubert/hol/Library/';;

let lib_dir_list =
        (map (concat Library_Root)
     ['gates/'; 'bits/';'words/';'numbers/'; 'decimal/'; 'assoc/']);;

set_search_path (search_path() @ ['.';
                                  '/epoch/d1/csgrad/schubert/hol/tactics/';
                                  '/epoch/d1/csgrad/schubert/hol/ml/';
                                  '/epoch/d1/csgrad/schubert/hol/theories/';
                                  '/epoch/d1/csgrad/schubert/hol/lisp/vax/';
                                 ]
                                @ lib_dir_list);;

loadf('aux_defs.ml');;

system 'rm /epoch/d1/csgrad/schubert/hol/theories/mmu_def.th';;

new_theory 'mmu_def';;

loadf 'abstract';;

map new_parent ['mmu_abs';'time_abs'];;

let rep_ty = abstract_type 'mmu_abs' 'segId';;

%----------------------------------------------------------------------
    type definitions
----------------------------------------------------------------------%

new_type_abbrev ('RWE',":bool#bool#bool");;

let rBIT = new_definition
   ('rBIT',"!rwe:RWE. rBIT rwe =  (FST rwe)");;

let wBIT = new_definition
   ('wBIT',"!rwe:RWE. wBIT rwe =  (FST (SND rwe))");;

let eBIT = new_definition
   ('eBIT',"!rwe :RWE. eBIT rwe =  (SND (SND rwe))");;

%----------------------------------------------------------------------
    Security bit auxiliary definitions
Segment Descriptor:
     n                          s-1              0
    +-----+----+-----+-------+....+----------------+
 0: |Avail|Read|Write|Execute|....|  Segment Size  |
    +-----+----+-----+-------+....+----------------+
    +-------------------------------------------+
 1: |                Real Offset                |
    +-------------------------------------------+
----------------------------------------------------------------------%



%----------------------------------------------------------------------
    MMU SPECIFICATION
```

```
----------------------------------------------------------------------%

let legalAccess = new_definition
  ('legalAccess', "!(rwe: RWE) vAddr tblPtr mem (r:^rep_ty) .
    legalAccess r vAddr tblPtr rwe mem =
      let a = (fetch r)( mem,
             (address r)((add r) (segIdshf r vAddr,tblPtr) )) in
      ( (validAccess r) (vAddr,a,rwe) /\ (ofsLEq r) (vAddr,a))" );;

let vToR = new_definition
  ('vToR',
    "!vAddr tblPtr mem (r:^rep_ty). vToR r vAddr tblPtr mem =
     let a = (fetch r) (mem, (address r)
         ((add r)( (wordn r 1), (add r)(segIdshf r vAddr,tblPtr) ))) in
      (address r) ((add r) (segOfs r vAddr, a)) ");;


let superMode = new_definition
  ('superMode',
   "! rwe vAddr tblPtrADDR tblPtr data mem (r:^rep_ty).
      superMode r vAddr rwe tblPtrADDR tblPtr data mem =
      ((wBIT rwe) /\ (addrEq r (vAddr,tblPtrADDR)) )
            => ( T, vAddr, data ) |
               ( T, vAddr, tblPtr )");;
%
let userMode = new_definition
  ('userMode',
   "! rwe vAddr tblPtrADDR tblPtr data mem (r:^rep_ty).
      userMode r vAddr rwe tblPtrADDR tblPtr data mem =
      ((wBIT rwe) /\ (addrEq r (vAddr,tblPtrADDR)) )
        => ( F:bool, vAddr, tblPtr ) |
           ( legalAccess r vAddr tblPtr rwe mem
             => ( T, (vToR r vAddr tblPtr mem), tblPtr ) |
                ( F, vAddr,                     tblPtr ) )");;
%

let userMode = new_definition
  ('userMode',
   "! rwe vAddr tblPtrADDR tblPtr data mem (r:^rep_ty).
      userMode r vAddr rwe tblPtrADDR tblPtr data mem =
        ( legalAccess r vAddr tblPtr rwe mem
            => ( T, (vToR r vAddr tblPtr mem), tblPtr ) |
               ( F, vAddr,                     tblPtr ) )");;


let nextState = new_definition
  ('nextState',
   "! rwe superv vAddr tblPtrADDR tblPtr data mem (r:^rep_ty) .
      nextState r vAddr rwe tblPtrADDR tblPtr data mem superv=
      (superv => superMode r vAddr rwe tblPtrADDR tblPtr data mem |
                 userMode  r vAddr rwe tblPtrADDR tblPtr data mem )" );;

let mmu_beh = new_definition
  ('mmu_beh',
   "!reqIn rwe superv vAddr tblPtrADDR tblPtrIn mem data (r:^rep_ty).
    mmu_beh r reqIn rwe vAddr superv data mem tblPtrADDR tblPtrIn =
%     (reqOut , rAddr , tblPtrOut ) = %
     reqIn => nextState r vAddr rwe tblPtrADDR tblPtrIn data mem superv  |
              (F:bool, vAddr, tblPtrIn)"          );;
```

```
let mmu_spec = new_definition
  ('mmu_spec',
   "! rwe superv vAddr tblPtrADDR tblPtr data mem (r:^rep_ty) .
      mmu_spec r vAddr rwe tblPtrADDR tblPtr data mem superv=
      (superv => superMode r vAddr rwe tblPtrADDR tblPtr data mem |
                 userMode  r vAddr rwe tblPtrADDR tblPtr data mem )" );;

%----------------------------------------------------------------------
   MMU IMPLEMENTATION
----------------------------------------------------------------------%

let secUnit_spec = new_definition
  ('secUnit_spec',
   "!a b ok (r:^rep_ty)(rwe:num->RWE). secUnit_spec r a b rwe ok =
    !t. ok (t+1) =
      ((validAccess r) ((a t),(b t),(rwe t)) /\ (ofsLEq r) ((a t),(b t)))");;

let addUnit_spec = new_definition
  ('addUnit_spec',   "!(a b c :num->*wordn) (r:^rep_ty).
    addUnit_spec r a b c = !t:num. c (t+1) = (add r ( (a t),(b t) ))");;

let muxUnit_spec = new_definition
  ('muxUnit_spec',
   "!(a out:num->*address) (b :num->*wordn) (w :num->bool) (r:^rep_ty).
     muxUnit_spec r a b out w =
    !t:num. (out (t+1)) = (w (t+1)) => address r(b (t+1)) | (a t)");;

let mux3Unit_spec = new_definition
  ('mux3Unit_spec',
   "!(a b c out :num->*wordn) (w:num->num). mux3Unit_spec a b c out w =
    !t:num. (out t) = (w t = 0) => a t | (w t = 1) => b t | c t"  );;

let splitUnit_spec = new_definition
   ('splitUnit_spec',
    "!(r:^rep_ty) virt id ofs. splitUnit_spec r virt id ofs =
     !t:num. ((id t)  = (segIdshf  r) (virt t)) /\
             ((ofs t) = (segOfs r) (virt t))   ");;

let latchUnit_spec = new_definition
  ('latchUnit_spec',
   "!(i out :num->*wordn) (ctrl:num->bool) (r :^rep_ty).
   latchUnit_spec r i out ctrl =
      !t:num. out (t+1) = ctrl (t+1) => out t | (i (t+1))" );;

let regUnit_spec = new_definition
  ('regUnit_spec',
   "!(i out :num->*wordn) ld clr (r:^rep_ty). regUnit_spec r i ld clr out =
     (!t:num. out (t+1) = (clr t => (wordn r 0 )  | ld t  => i t  | out t) ) /\
     (out 0 = (wordn r 0) )");;

let matchUnit_spec = new_definition
  ('matchUnit_spec',
   "!(a b:num->*address) (m:num->bool) (r:^rep_ty). matchUnit_spec r a b m =
    !(t:num). m(t+1) = ( addrEq r (a t, b t) ) => T:bool | F");;

let oneUnit_spec = new_definition
  ('oneUnit_spec', "!t:num (r:^rep_ty). oneUnit_spec r t = (wordn r) 1");;
```

```
let bitFalse = new_definition
  ('bitFalse', "!t:num. bitFalse t = F");;

let memoryUnit_spec = new_definition
   ('memoryUnit_spec',
    "!req addr data done mem (r:^rep_ty).
    memoryUnit_spec r req addr data done mem =
      ( (data 0 = wordn r 0) /\ (done 0 = F) ) /\
      !t. ( (req t) => ( (data (t+1) = fetch r (mem t, addr t) ) /\
                         (done (t+1) = T) ) |
                       ( (data (t+1) = wordn r 0) /\
                         (done (t+1) = F) ) )");;



%-----------------------------------------------------------------
A valid request will require 4 phases required with a delay of at least 1
time unit occurs between phases.

       0:  (initial)  -wait until reqIn-
                add (shift vaddr), tblPtr into tmpReg
                compare vaddr, tblPtrADDR       (match)
       1:
         if supervisor mode
                if match and write request -> store dataIn into tblPtr
                else pass request thru (addr,data,rwe) and ack
                goto Phase 0
         else
                fetch mem (tmpReg)
  add tmpReg, 1
       2: -wait until fdone-
         fetch mem (tmpReg+1)
       3: -wait until fdone-
          if secUnit pass
       add fetched value, vaddr
       pass request thru (addr,data,rwe) and ack
          else
       FAIL
-----------------------------------------------------------------%

let controlUnit_spec = new_definition
   ('controlUnit_spec',  "!(muxC phase :num->num) (rwe: num->RWE)
     (tmpC tblC lC xlat done ack rReq reqIn super match secOK fdone:num->bool).
    controlUnit_spec reqIn super rwe match secOK fdone
                     muxC tmpC tblC lC rReq xlat done ack phase =
    ((muxC 0,tmpC 0,tblC 0,lC 0,rReq 0,xlat 0,done 0,ack 0, phase 0) =
     ( 0    ,  F   ,  F   , F  , F    ,  F   ,   F  , F  ,   0    )  )
    /\
    (!t.(muxC(t+1),tmpC(t+1),tblC(t+1),lC(t+1),rReq(t+1),xlat(t+1),done(t+1),
       ack(t+1),phase(t+1) ) =                  %  M  t t l  r x d a  P  %
                                                %  U  m b a  e l o c  H  %
                                                %  X  p l t  q t n k  A  %
        (phase t = 0) =>
            (reqIn t =>                          ( 0, F,F,F, F,F,F,F, 1)  |
                                                 ( 0, F,F,F, F,F,F,F, 0)) |

        (phase t = 1) =>
            (super t =>
             ((wBIT (rwe t)) /\ match t) =>      ( 0, F,T,F, F,F,F,F, 5)  |
                                                 ( 0, F,F,F, F,F,T,T ,0)  |
```

```
                                              ( 2, T,F,T, T,T,F,F, 2)) |
      ((phase t = 2) /\ fdone t) =>           ( 1, F,F,F, T,T,F,F, 3)  |
      ((phase t = 3) /\ fdone t) =>
            (secOK t  =>                      ( 0, F,F,F, F,T,F,F, 4)  |
            .                                 ( 0, F,F,F, F,F,T,F, 0)) |
      (phase t = 4) =>                        ( 0, F,F,T, F,T,T,T, 0)  |
      (phase t = 5) =>                        ( 0, F,F,F, F,F,T,T ,0)  |
      (muxC t,tmpC t,tblC t,lC t, F ,xlat t,done t,ack t,phase t)) " );;


let dataPath = new_definition
  ('dataPath',
   "!(r:^rep_ty) (vAddr rAddr :num->*address)(vData :num->*wordn) mem
     (rwe :num->RWE) mem (tblPtr :num->*wordn) (tblPtrADDR :num->*address)
     (muxC :num->num)(tmpC tblC lC rReq xlat match secOK fdone :num->bool).
     dataPath  r vAddr vData rwe mem tblPtrADDR tblPtr rAddr
               muxC tmpC tblC lC rReq xlat match secOK fdone =
    ?(mux1 mux2 id ofs addOut data latOut :num->*wordn)
     (secData:num->*wordn).
    (regUnit_spec     r vData tblC bitFalse tblPtr)              /\
    (regUnit_spec     r data  tmpC bitFalse secData)             /\
    (secUnit_spec     r vAddr secData rwe secOK)                 /\
    (splitUnit_spec   r vAddr id ofs)                            /\
    (mux3Unit_spec    id  ofs (oneUnit_spec r) mux1 muxC)        /\
    (mux3Unit_spec    tblPtr data   latOut     mux2 muxC)        /\
    (addUnit_spec     r mux1 mux2 addOut)                        /\
    (latchUnit_spec   r addOut latOut lC)                        /\
    (matchUnit_spec   r vAddr tblPtrADDR match)                  /\
    (muxUnit_spec     r vAddr latOut rAddr xlat)                 /\
    (memoryUnit_spec  r rReq rAddr data fdone mem)                ");;


let mmu_imp = new_definition
  ('mmu_imp',
   "!(r:^rep_ty) (vAddr rAddr :num->*address)(vData :num->*wordn)
     (rwe :num->RWE)(superv reqIn xlat ack done :num->bool) mem
     (tblPtr :num->*wordn) (tblPtrADDR :num->*address) (phase :num->num).
    mmu_imp r vAddr vData rwe superv tblPtr tblPtrADDR reqIn
             rAddr done ack xlat mem phase =
    ?(muxC :num->num)(tmpC tblC lC rReq match secOK fdone :num->bool) .
    (controlUnit_spec reqIn superv rwe match secOK fdone
           muxC tmpC tblC lC rReq xlat done ack phase)       /\
    (dataPath  r vAddr vData rwe mem tblPtrADDR tblPtr rAddr
           muxC tmpC tblC lC rReq xlat match secOK fdone)    ");;


close_theory();;
```

mmu_aux.ml

```
let Library_Root = '/epoch/d1/csgrad/schubert/hol/Library/';;

let lib_dir_list =
       (map (concat Library_Root)
    ['gates/'; 'bits/';'words/';'numbers/'; 'decimal/'; 'assoc/']);;

set_search_path (search_path() @ ['.';
                                  '/epoch/d1/csgrad/schubert/hol/tactics/';
                                  '/epoch/d1/csgrad/schubert/hol/ml/';
                                  '/epoch/d1/csgrad/schubert/hol/theories/';
                                  '/epoch/d1/csgrad/schubert/hol/lisp/vax/';
                                 ]
                               @ lib_dir_list);;

loadf('aux_defs.ml');;

system 'rm /epoch/d1/csgrad/schubert/hol/theories/mmu_aux.th';;

new_theory 'mmu_aux';;

%map load_parent ['mmu_abs';'time_abs';'mmu_def';'ctrlUnit_len'];;%

% ;'num_thms']; %

new_type_abbrev ('RWE', ":bool#bool#bool");;

%-----------------------------------------------------------------------
AUX FACTS
-----------------------------------------------------------------------%

let PLUS_ONE_TAC n =
    REWRITE_TAC [(SYM_RULE ADD1);(num_CONV n);ADD_CLAUSES];;

let T2 = prove_thm ('T2', "!t. (t + 1) + 1 = t + 2", PLUS_ONE_TAC "2" );;

let T3 = prove_thm ('T3', "!t. (t + 2) + 1 = t + 3", PLUS_ONE_TAC "3" );;

let T4 = prove_thm ('T4', "!t. (t + 3) + 1 = t + 4", PLUS_ONE_TAC "4" );;

let T5 = prove_thm ('T5', "!t. (t + 4) + 1 = t + 5", PLUS_ONE_TAC "5" );;

let T6 = prove_thm ('T6', "!t. (t + 5) + 1 = t + 6", PLUS_ONE_TAC "6" );;

let T7 = prove_thm ('T7', "!t. (t + 6) + 1 = t + 7", PLUS_ONE_TAC "7" );;


let LESS_ADD_SUC = prove_thm
 ('LESS_ADD_SUC',"!t n.  t < ( t + SUC(n) )",
   REWRITE_TAC [ADD_CLAUSES;LESS_THM]
   THEN REPEAT GEN_TAC
   THEN DISJ_CASES_TAC (SPEC "n" LESS_0_CASES)
   THENL
     [ POP_ASSUM(\thm. REWRITE_TAC [(SYM_RULE thm);ADD_CLAUSES])
     ;
      POP_ASSUM(\thm. ASSUME_TAC( REWRITE_RULE [thm]
         (SPECL ["0";"n"] LESS_NOT_EQ) ))
      THEN POP_ASSUM(\thm. REWRITE_TAC [ (REWRITE_RULE
```

```
        [(SYM_RULE thm)] (SPECL ["t";"n" ]LESS_ADD_NONZERO ))])
    ] );;


let RANGE_LEMMA = TAC_PROOF
    (([], "!t1 t2 (f:num->bool).
         (!t'. t1 < t' /\ t' < t2 ==> ~(f t')) /\ ~(f t2)
          ==> (!t'. t1 < t' /\ t' < (t2+1) ==> ~(f t'))"),
   REPEAT STRIP_TAC
   THEN ASSUM_LIST (\asl. ASSUME_TAC( SPEC "t':num" (el 5 asl)))
   THEN ASSUM_LIST (\asl. STRIP_ASSUME_TAC (
      REWRITE_RULE [SYM_RULE ADD1; LESS_THM] (el 3 asl)))
   THENL
     [ ASSUM_LIST(\asl. ASSUME_TAC ( REWRITE_RULE [(el 1 asl)] (el 3 asl)))
      ;
       ALL_TAC
     ]
   THEN RES_TAC
 );;

%
let LESS_SQUEEZE_LEMMA =
  let LESS_EQ_SUCC =
     SYM_RULE (PURE_ONCE_REWRITE_RULE [DISJ_SYM] LESS_THM) in
    PURE_ONCE_REWRITE_RULE [ADD1] (
    PURE_ONCE_REWRITE_RULE [LESS_EQ_SUC] (
    PURE_ONCE_REWRITE_RULE [LESS_OR_EQ] LESS_EQ_ANTISYM));;
%

let stable_sigs = new_definition
   ('stable_sigs',
    "!t1 t2 (rwe :num->RWE) (vAddr tblPtrADDR:num->*address)
     (data:num->*wordn) (mem:num->*memory) (super:num->bool).
     stable_sigs t1 t2 vAddr rwe tblPtrADDR data mem super =
     !t'. t1 < t' /\ t' < t2 ==>
          (super t' = super t1) /\     (vAddr t' = vAddr t1) /\
          (rwe t' = rwe t1)     /\     (data t' = data t1) /\
          (mem t' = mem t1)     /\     (tblPtrADDR t' = tblPtrADDR t1)"
    );;


%
let IMP_F_THM = prove_thm
  ('IMP_F_THM',"!f. (f ==> F) = (f = F)",
    GEN_TAC
    THEN BOOL_CASES_TAC "f"
    THEN  REWRITE_TAC []  );;

let NOT_TO_EQ_CONV =
   (PURE_REWRITE_RULE [IMP_F_THM] o
   (BETA_RULE o (ONCE_REWRITE_RULE [NOT_DEF] )));;
%

let LESS_ADD_EQ = prove_thm
   ('LESS_ADD_EQ',
    "!t x y. ((t+x) < (t+y)) = (x < y)",
    INDUCT_TAC
    THEN REWRITE_TAC [ADD_CLAUSES]
    THEN ONCE_REWRITE_TAC [CONJUNCT1(CONJUNCT2(CONJUNCT2(ADD_CLAUSES)))]
```

```
   THEN ASM_REWRITE_TAC [LESS_MONO_EQ] );;


let BETW_0_7_IS_1 = prove_thm
    ('BETW_0_7_IS_1', "0 < 1 /\ 1 < 7",
      CONV_TAC (TOP_DEPTH_CONV num_CONV)
      THEN REWRITE_TAC[LESS_0;LESS_MONO_EQ] );;

let BETW_0_7_IS_2 = prove_thm
    ('BETW_0_7_IS_2', "0 < 2 /\ 2 < 7",
      CONV_TAC (TOP_DEPTH_CONV num_CONV)
      THEN REWRITE_TAC[LESS_0;LESS_MONO_EQ] );;

let BETW_0_7_IS_4 = prove_thm
    ('BETW_0_7_IS_4', "0 < 4 /\ 4 < 7",
      CONV_TAC (TOP_DEPTH_CONV num_CONV)
      THEN REWRITE_TAC[LESS_0;LESS_MONO_EQ] );;

let BETW_0_7_IS_5 = prove_thm
    ('BETW_0_7_IS_5', "0 < 5 /\ 5 < 7",
      CONV_TAC (TOP_DEPTH_CONV num_CONV)
      THEN REWRITE_TAC[LESS_0;LESS_MONO_EQ] );;


let BETW_0_6_IS_1 = prove_thm
    ('BETW_0_6_IS_1', "0 < 1 /\ 1 < 6",
      CONV_TAC (TOP_DEPTH_CONV num_CONV)
      THEN REWRITE_TAC[LESS_0;LESS_MONO_EQ] );;

let BETW_0_6_IS_2 = prove_thm
    ('BETW_0_6_IS_2', "0 < 2 /\ 2 < 6",
      CONV_TAC (TOP_DEPTH_CONV num_CONV)
      THEN REWRITE_TAC[LESS_0;LESS_MONO_EQ] );;

let BETW_0_6_IS_4 = prove_thm
    ('BETW_0_6_IS_4', "0 < 4 /\ 4 < 6",
      CONV_TAC (TOP_DEPTH_CONV num_CONV)
      THEN REWRITE_TAC[LESS_0;LESS_MONO_EQ] );;

let BETW_0_6_IS_5 = prove_thm
    ('BETW_0_6_IS_5', "0 < 5 /\ 5 < 6",
      CONV_TAC (TOP_DEPTH_CONV num_CONV)
      THEN REWRITE_TAC[LESS_0;LESS_MONO_EQ] );;


close_theory();;
```

ctrlUnit_lem.ml

```
let Library_Root = '/epoch/d1/csgrad/schubert/hol/Library/';;

let lib_dir_list =
       (map (concat Library_Root)
    ['gates/'; 'bits/';'words/';'numbers/'; 'decimal/'; 'assoc/']);;

set_search_path (search_path() @ ['.';
                                  '/epoch/d1/csgrad/schubert/hol/tactics/';
                                  '/epoch/d1/csgrad/schubert/hol/ml/';
                                  '/epoch/d1/csgrad/schubert/hol/theories/';
                                  '/epoch/d1/csgrad/schubert/hol/lisp/vax/';
                                 ]
                               @ lib_dir_list);;

loadf('aux_defs.ml');;

system 'rm /epoch/d1/csgrad/schubert/hol/theories/ctrlUnit_lem.th';;

new_theory 'ctrlUnit_lem';;

%loadf 'abstract';;%

map load_parent ['mmu_abs';'time_abs';'mmu_def';'arithmetic'];;


%----------------------------------------------------------------------
AUX FACTS
----------------------------------------------------------------------%

let SUC_EQ_DEF = prove_thm
  ('SUC_EQ_DEF', "!m n. (SUC m = SUC n) = (m = n)",
   REPEAT GEN_TAC
   THEN EQ_TAC
   THENL
    [REWRITE_TAC [INV_SUC]
    ;
     STRIP_TAC
     THEN BOOL_CASES_TAC "m = n"
     THEN ASM_REWRITE_TAC []
    ]
  );;

let num_EQ_TAC =
   CONV_TAC (TOP_DEPTH_CONV num_CONV)
   THEN REWRITE_TAC [SUC_EQ_DEF]
   THEN REWRITE_TAC [NOT_SUC]
   THEN CONV_TAC (ONCE_DEPTH_CONV SYM_CONV )
   THEN REWRITE_TAC [NOT_SUC];;

let PHASE_0_UNIQUE = prove_thm
    ('PHASE_0_UNIQUE', "~(0 = 1) /\ ~(0 = 2) /\ ~(0 = 3)/\~(0 = 4)/\~(0 = 5)",
    REPEAT CONJ_TAC THEN num_EQ_TAC );;

let PHASE_1_UNIQUE = prove_thm
    ('PHASE_1_UNIQUE', "~(1 = 0) /\ ~(1 = 2) /\ ~(1 = 3)/\~(1 = 4)/\~(1 = 5)",
    REPEAT CONJ_TAC THEN num_EQ_TAC );;
```

```
let PHASE_2_UNIQUE = prove_thm
    ('PHASE_2_UNIQUE', "~(2 = 0) /\ ~(2 = 1) /\ ~(2 = 3)/\~(2 = 4)/\~(2 = 5)",
    REPEAT CONJ_TAC THEN num_EQ_TAC );;

let PHASE_3_UNIQUE = prove_thm
    ('PHASE_3_UNIQUE', "~(3 = 0) /\ ~(3 = 1) /\ ~(3 = 2)/\~(3 = 4)/\~(3 = 5)",
    REPEAT CONJ_TAC THEN num_EQ_TAC );;

let PHASE_4_UNIQUE = prove_thm
    ('PHASE_4_UNIQUE', "~(4 = 0) /\ ~(4 = 1) /\ ~(4 = 2)/\~(4 = 3)/\~(4 = 5)",
    REPEAT CONJ_TAC THEN num_EQ_TAC );;


let PHASE_5_UNIQUE = prove_thm
    ('PHASE_5_UNIQUE', "~(5 = 0) /\ ~(5 = 1) /\ ~(5 = 2)/\~(5 = 3)/\~(5 = 4)",
    REPEAT CONJ_TAC THEN num_EQ_TAC );;


%-----------------------------------------------------------------------
Control Unit Lemmas
-----------------------------------------------------------------------%

let SIX_PHASES_ONLY = prove_thm
   ('SIX_PHASES_ONLY',
   "! muxC phase rve tmpC tblC lC xlat done ack rReq reqIn super match
     secOK fdone.
     controlUnit_spec reqIn super rve match secOK fdone muxC tmpC tblC lC
                      rReq xlat done ack phase ==>
     (!t. (phase t = 0) \/ (phase t = 1) \/ (phase t = 2) \/
          (phase t = 3) \/ (phase t = 4) \/ (phase t = 5))",
   REPEAT GEN_TAC
   THEN PURE_ONCE_REWRITE_TAC [ controlUnit_spec ]
   THEN STRIP_TAC
   THEN INDUCT_TAC
   THENL
    [%---------------- base case ----------------%
     ASSUM_LIST(\asl. MAP_EVERY ASSUME_TAC( CONJUNCTS (
       (REWRITE_RULE [PAIR_EQ] (el 2 asl) ) )))
     THEN POP_ASSUM(\thm.  REWRITE_TAC[thm] )
     ;%---------------- induction ----------------%
     PURE_REWRITE_TAC [ADD1]
     THEN POP_ASSUM(\thm. DISJ_CASES_TAC (thm) )
     THENL
     [%----------------------- case 0 -----------------------%
      ASM_CASES_TAC "(reqIn t):bool"
      THEN POP_ASSUM_LIST(\asl. REWRITE_TAC ( CONJUNCTS (
            REWRITE_RULE ([(el 1 asl);(el 2 asl)] @ [PAIR_EQ])
                         (SPEC_ALL (el 3 asl))  )))
      ;
      POP_ASSUM(\thm. DISJ_CASES_TAC (thm) )
      THENL
       [%----------------------- case 1 -----------------------%
        ASM_CASES_TAC "(super t):bool"
        THEN ASM_CASES_TAC "(wBIT(rve t) /\ match t):bool"
        THEN POP_ASSUM_LIST(\asl. REWRITE_TAC ( CONJUNCTS (
             REWRITE_RULE ([(el 1 asl);(el 2 asl);(el 3 asl);PAIR_EQ] @
                (CONJUNCTS (PHASE_1_UNIQUE)))  (SPEC_ALL (el 4 asl)))))
        ;
        POP_ASSUM(\thm. DISJ_CASES_TAC (thm) )
```

```
      THENL
       [%---------------------- case 2 ----------------------%
        ASM_CASES_TAC "(fdone t):bool"
        THEN POP_ASSUM_LIST(\asl. REWRITE_TAC ( CONJUNCTS (
           REWRITE_RULE ((CONJUNCTS (PHASE_2_UNIQUE)) @ [PAIR_EQ])
           (REWRITE_RULE [(el 1 asl);(el 2 asl)]
                         (SPEC_ALL (el 3 asl))) )))
        ;
        POP_ASSUM(\thm. DISJ_CASES_TAC (thm) )
        THENL
         [%---------------------- case 3 ----------------------%
          ASM_CASES_TAC "(fdone t):bool"
          THEN ASM_CASES_TAC "(secOK t):bool"
          THEN POP_ASSUM_LIST(\asl. REWRITE_TAC ( CONJUNCTS (
             REWRITE_RULE ((CONJUNCTS (PHASE_3_UNIQUE)) @ [PAIR_EQ])
             (REWRITE_RULE [(el 1 asl);(el 2 asl);(el 3 asl) ]
                           (SPEC_ALL (el 4 asl)) ))))
          ;%---------------------- case 4,5 ----------------------%
          POP_ASSUM(\thm. DISJ_CASES_TAC (thm) )
          THEN POP_ASSUM_LIST(\asl. REWRITE_TAC( CONJUNCTS (
             REWRITE_RULE ((CONJUNCTS (PHASE_4_UNIQUE)) @
                           (CONJUNCTS (PHASE_5_UNIQUE)) @ [PAIR_EQ])
             (REWRITE_RULE [(el 1 asl)] (SPEC_ALL (el 2 asl))) )))
      ]]]]]  );;

%-------------------------------------------------------
SIX_PHASES_ONLY =
|- !muxC phase rve tmpC tblC lC xlat done ack rReq reqIn super match
    secOK fdone.
    controlUnit_spec
    reqIn
    super
    rve
    match
    secOK
    fdone
    muxC
    tmpC
    tblC
    lC
    rReq
    xlat
    done
    ack
    phase ==>
    (!t.
      (phase t = 0) \/
      (phase t = 1) \/
      (phase t = 2) \/
      (phase t = 3) \/
      (phase t = 4) \/
      (phase t = 5))
Run time: 1235.6s
Intermediate theorems generated: 73322

(Holly : Run time: 2728.2s)
-------------------------------------------------------%
```

```
let NOT_PHASE_2_THEN_0 = prove_thm
   ('NOT_PHASE_2_THEN_0',
   "! muxC phase rve tmpC tblC lC xlat done ack rReq reqIn super match
     secOK fdone.
     controlUnit_spec reqIn super rve match secOK fdone muxC tmpC tblC lC
                      rReq xlat done ack phase ==>
     (!t. (phase t = 2) ==> ~(phase (t+1) = 0))",
   REPEAT GEN_TAC
   THEN PURE_ONCE_REWRITE_TAC [ controlUnit_spec ]
   THEN STRIP_TAC
   THEN STRIP_TAC
   THEN STRIP_TAC
   THEN POP_ASSUM_LIST(\asl. ASSUME_TAC(
          REWRITE_RULE [(el 1 asl);PHASE_2_UNIQUE] (SPEC_ALL (el 2 asl))))
   THEN ASM_CASES_TAC "(fdone t):bool"
   THEN POP_ASSUM_LIST(\asl. MAP_EVERY ASSUME_TAC( CONJUNCTS (
          REWRITE_RULE [(el 1 asl);PAIR_EQ] (el 2 asl) )))
   THEN STRIP_TAC
   THEN POP_ASSUM_LIST(\asl. REWRITE_TAC [ (REWRITE_RULE
     ((CONJUNCTS PHASE_0_UNIQUE) @ [(el 1 asl)]) (el 2 asl))]) );;


%-------------------------------------------------------
NOT_PHASE_2_THEN_0 =
|- !muxC phase rve tmpC tblC lC xlat done ack rReq reqIn super match
    secOK fdone.
    controlUnit_spec
    reqIn
    super
    rve
    match
    secOK
    fdone
    muxC
    tmpC
    tblC
    lC
    rReq
    xlat
    done
    ack
    phase ==>
    (!t. (phase t = 2) ==> ~(phase(t + 1) = 0))
Run time: 69.5s
Intermediate theorems generated: 6905

(Holly: Run time: 233.6s)
-------------------------------------------------------%

let PHASE_0_IDLE = prove_thm
   ('PHASE_0_IDLE',
    "! muxC phase rve tmpC tblC lC xlat done ack rReq reqIn super match
     secOK fdone.
     controlUnit_spec reqIn super rve match secOK fdone muxC tmpC tblC lC
                      rReq xlat done ack phase ==>
     (!t. (phase t = 0) ==> ( (tblC t = F) /\ (muxC t = 0)) )",
   REPEAT GEN_TAC
   THEN STRIP_TAC
   THEN INDUCT_TAC
```

```
THENL
 [% base case %
  POP_ASSUM(\thm. MAP_EVERY ASSUME_TAC( CONJUNCTS (
     REWRITE_RULE [controlUnit_spec] thm )))
  THEN POP_ASSUM_LIST(\asl. REWRITE_TAC( CONJUNCTS (
       REWRITE_RULE [PAIR_EQ] (el 2 asl))))
 ;% induction case %
  REWRITE_TAC [ADD1]
  THEN ASSUM_LIST(\asl. ASSUME_TAC( SPEC_ALL(
   REWRITE_RULE [(el 2 asl)] (SPEC_ALL SIX_PHASES_ONLY))))
  THEN ASSUM_LIST(\asl. MAP_EVERY ASSUME_TAC( CONJUNCTS (
     SPEC_ALL (REWRITE_RULE [controlUnit_spec] (el 3 asl)))) )
  THEN POP_ASSUM_LIST(\asl. MAP_EVERY ASSUME_TAC(rev(subtract asl
     [(el 2 asl);(el 4 asl);(el 5 asl)])))
  THEN ASSUM_LIST(\asl. DISJ_CASES_TAC (el 2 asl) )
  THENL
    [%------------- phase 0 ------------%
     ASM_CASES_TAC "(reqIn t):bool"
     THEN POP_ASSUM_LIST(\asl. REWRITE_TAC( CONJUNCTS (
       REWRITE_RULE ([PAIR_EQ;(el 1 asl);(el 2 asl)]@
            (CONJUNCTS PHASE_0_UNIQUE)) (SPEC_ALL (el 3 asl)) )))
     ;POP_ASSUM(\thm.  DISJ_CASES_TAC (thm) )
      THENL
       [%------------- phase 1 ------------%
        ASM_CASES_TAC "(super t):bool"
        THEN ASM_CASES_TAC "((wBIT(rwe t) /\ match t)):bool"
        THEN POP_ASSUM_LIST(\asl. REWRITE_TAC(
          (CONJUNCTS PHASE_5_UNIQUE) @ (CONJUNCTS (
            REWRITE_RULE ([PAIR_EQ;(el 1 asl);(el 2 asl); (el 3 asl)] @
                (CONJUNCTS PHASE_1_UNIQUE)) (SPEC_ALL (el 4 asl)) ))))
       ;POP_ASSUM(\thm.  DISJ_CASES_TAC (thm) )
        THENL
         [%------------- phase 2 ------------%
          ASM_CASES_TAC "(fdone t):bool"
          THEN POP_ASSUM_LIST(\asl. REWRITE_TAC( (CONJUNCTS PHASE_2_UNIQUE)@
             (CONJUNCTS PHASE_3_UNIQUE) @  (CONJUNCTS ( REWRITE_RULE
             ([PAIR_EQ;(el 1 asl);(el 2 asl)]@(CONJUNCTS PHASE_2_UNIQUE))
             (SPEC_ALL (el 3 asl)) ))))
         ;POP_ASSUM(\thm.  DISJ_CASES_TAC (thm))
          THENL
           [%------------- phase 3  ------------%
            ASM_CASES_TAC "(fdone t):bool"
            THEN ASM_CASES_TAC "(secOK t):bool"
            THEN POP_ASSUM_LIST(\asl. REWRITE_TAC(
             (CONJUNCTS PHASE_3_UNIQUE) @  (CONJUNCTS ( REWRITE_RULE
             ([PAIR_EQ;(el 1 asl);(el 2 asl);(el 3 asl)]@
             (CONJUNCTS PHASE_3_UNIQUE)) (SPEC_ALL (el 4 asl)) ))))
            ;%------------- phase 4,5  ------------%
            POP_ASSUM(\thm.  DISJ_CASES_TAC (thm) )
            THEN POP_ASSUM_LIST(\asl. REWRITE_TAC(
             (CONJUNCTS PHASE_5_UNIQUE) @ (CONJUNCTS PHASE_4_UNIQUE) @
             (CONJUNCTS ( REWRITE_RULE
                 ([PAIR_EQ;(el 1 asl)]@(CONJUNCTS PHASE_4_UNIQUE)@
                 (CONJUNCTS PHASE_5_UNIQUE)) (SPEC_ALL (el 2 asl)) ))))
           ]]]] ] );;

%-------------------------------------------------------
PHASE_0_IDLE =
|- !muxC phase rwe tmpC tblC lC xlat done ack rReq reqIn super match
```

```
   secOK fdone.
   controlUnit_spec
   reqIn
   super
   rwe
   match
   secOK
   fdone
   muxC
   tmpC
   tblC
   lC
   rReq
   xlat
   done
   ack
   phase ==>
   (!t. (phase t = 0) ==> (tblC t = F) /\ (muxC t = 0))
Run time: 721.0s
Intermediate theorems generated: 66258
------------------------------------------------------------%

let CTRL_UNIT_EXPAND = prove_thm
    ('CTRL_UNIT_EXPAND',
    "controlUnit_spec reqIn super rwe match secOK fdone muxC
       tmpC tblC lC rReq xlat done ack phase ==>
     !t.
      muxC(t + 1),tmpC(t + 1),tblC(t + 1),lC(t + 1),rReq(t + 1),
      xlat(t + 1),done(t + 1),ack(t + 1),phase(t + 1) =
      ((phase t = 0) =>
       (reqIn t => (0,F,F,F,F,F,F,F,1) | (0,F,F,F,F,F,F,F,0)) |
       ((phase t = 1) =>
        (super t =>
         ((wBIT(rwe t) /\ match t) =>
          (0,F,T,F,F,F,F,F,5) |
          (0,F,F,F,F,F,T,T,0)) |
         (2,T,F,T,T,T,F,F,2)) |
        (((phase t = 2) /\ fdone t) =>
         (1,F,F,F,T,T,F,F,3) |
         (((phase t = 3) /\ fdone t) =>
          (secOK t => (0,F,F,F,F,T,F,F,4) | (0,F,F,F,F,F,T,F,0)) |
          ((phase t = 4) =>
           (0,F,F,T,F,T,T,T,0) |
           ((phase t = 5) =>
            (0,F,F,F,F,F,T,T,0) |
            (muxC t,tmpC t,tblC t,lC t, F ,xlat t,done t,ack t,
             phase t)))))))",
     STRIP_TAC
     THEN POP_ASSUM( \thm. ACCEPT_TAC (
      (CONJUNCT2( (REWRITE_RULE [controlUnit_spec] thm)))) ));;

%------------------------------------------------------------
CTRL_UNIT_EXPAND =
|- controlUnit_spec
   reqIn
   super
   rwe
   match
   secOK
```

```
   fdone
   mxC
   tmpC
   tblC
   lC
   rReq
   xlat
   done
   ack
   phase ==>
   (!t.
     mxC(t + 1),tmpC(t + 1),tblC(t + 1),lC(t + 1),rReq(t + 1),
     xlat(t + 1),done(t + 1),ack(t + 1),phase(t + 1) =
     ((phase t = 0) =>
      (reqIn t => (0,F,F,F,F,F,F,F,1) | (0,F,F,F,F,F,F,F,0)) |
      ((phase t = 1) =>
       (super t =>
        ((wBIT(rve t) /\ match t) =>
         (0,F,T,F,F,F,F,F,5) |
         (0,F,F,F,F,F,T,T,0)) |
        (2,T,F,T,T,T,F,F,2)) |
       (((phase t = 2) /\ fdone t) =>
        (1,F,F,F,T,T,F,F,3) |
        (((phase t = 3) /\ fdone t) =>
         (secOK t => (0,F,F,F,F,T,F,F,4) | (0,F,F,F,F,F,T,F,0)) |
         ((phase t = 4) =>
          (0,F,F,F,F,T,T,T,0) |
          ((phase t = 5) =>
           (0,F,F,F,F,F,T,T,0) |
           (mxC t,tmpC t,tblC t,lC t, F ,xlat t,done t,ack t,phase t))))))))
Run time: 33.7s
Intermediate theorems generated: 2782
------------------------------------------------------------%

close_theory();;
```

mmu_prf.ml

```
let Library_Root = '/epoch/d1/csgrad/schubert/hol/Library/';;

let lib_dir_list =
       (map (concat Library_Root)
    ['gates/'; 'bits/';'words/';'numbers/'; 'decimal/'; 'assoc/']);;

set_search_path (search_path() @ ['.';
                                  '/epoch/d1/csgrad/schubert/hol/tactics/';
                                  '/epoch/d1/csgrad/schubert/hol/ml/';
                                  '/epoch/d1/csgrad/schubert/hol/theories/';
                                  '/epoch/d1/csgrad/schubert/hol/lisp/vax/';
                                 ]
                               @ lib_dir_list);;

loadf('aux_defs.ml');;

system 'rm /epoch/d1/csgrad/schubert/hol/theories/mmu_prf.th';;

new_theory 'mmu_prf';;

loadf 'abstract';;

loadf 'exist.tac.ml';;

map load_parent ['mmu_abs';'time_abs';'mmu_def';'ctrlUnit_lem';'mmu_aux'];;

let rep_ty = abstract_type 'mmu_abs' 'segId';;

%----------------------------------------------------------------------
AUX FACTS AND DEFS
----------------------------------------------------------------------%

let line tok t =
   if (is_eq t)
    then (let x = fst(dest_var(rator(lhs(t))))
          in (mem x (words tok) ? false))
    else ( if (is_neg t)
           then (let y =fst(dest_var(rator(dest_neg(t))))
                 in mem y (words tok))
           else (let y = fst(dest_var(rator(t)))
                 in mem y (words tok)) )
    ? false;;

letrec lines tok t =
   if (is_conj t)
    then (let x = (dest_conj t)
          in (let b = (line tok (fst x))
              in (if b then true
                       else  (lines tok (snd x)) )))
    else (line tok t)
    ? false;;

letrec unit tok t =
  if (is_comb t)
   then     (let x = fst(dest_comb t)  in unit tok x)
   else    ((let x = fst(dest_const t) in mem x (words tok)) ? false) ;;
```

```
let FIND_ASSUM f asl = hd(filter(f o concl) asl);;

let FIND_SPEC_UNIT s u u' asl =
      (SPEC s (REWRITE_RULE [u]
       (FIND_ASSUM (unit u') asl) ));;

let FIND_ASSUM2 f asl = hd(tl(filter(f o concl) asl));;

let FIND_SPEC_UNIT2 s u u' asl =
      (SPEC s (REWRITE_RULE [u]
       (FIND_ASSUM2 (unit u') asl) ));;

let FIND_SPEC_MEM_UNIT s  asl =
      (SPEC s (CONJUNCT2 (REWRITE_RULE [memoryUnit_spec]
       (FIND_ASSUM (unit `memoryUnit_spec`) asl) )));;

let FILTER_ASSUM_TAC thml f =
 ASSUM_LIST(\asl. ASSUME_TAC( REWRITE_RULE thml
     (FIND_ASSUM f asl) ));;

let UNIT u asl =  (FIND_ASSUM (unit u) asl);;

let LINE l asl =  (FIND_ASSUM (lines l) asl);;


let LESS_CONV x  =
   REWRITE_RULE [LESS_MONO_EQ;LESS_0](
   REWRITE_RULE [ADD;ADD_SYM] ((TOP_DEPTH_CONV num_CONV) x));;

let RANGE_LEMMA = TAC_PROOF
    (([], "!t1 t2 (f:num->bool).
         (!t'. t1 < t' /\ t' < t2 ==> ~(f t')) /\ ~(f t2)
          ==> (!t'. t1 < t' /\ t' < (t2+1) ==> ~(f t'))"),
   REPEAT STRIP_TAC
   THEN ASSUM_LIST (\asl. ASSUME_TAC( SPEC "t':num" (el 5 asl)))
   THEN ASSUM_LIST (\asl. STRIP_ASSUME_TAC (
       REWRITE_RULE [SYM_RULE ADD1; LESS_THM] (el 3 asl)))
   THENL
     [ ASSUM_LIST(\asl. ASSUME_TAC ( REWRITE_RULE [(el 1 asl)] (el 3 asl)))
      ;
       ALL_TAC
     ]
   THEN RES_TAC
 );;

let RANGE_TAC hi lo =
   CONJ_TAC
   THENL
     [REWRITE_TAC [(num_CONV hi );(SPECL ["t";lo] LESS_ADD_SUC)]
      ;
      REPEAT
       (PURE_ONCE_REWRITE_TAC [(SYM_RULE T2);(SYM_RULE T3);(SYM_RULE T4);
                               (SYM_RULE T5);(SYM_RULE T6);(SYM_RULE T7)]
        THEN MATCH_MP_TAC RANGE_LEMMA
        THEN CONJ_TAC
          THENL
           [REWRITE_TAC [(SYM_RULE ADD1);LESS_LESS_SUC]
            ;
            ASM_REWRITE_TAC []
```

```
        ]
      )];;

let LESS_ADD_EQ1 =
    % "!t y. t < (t + y) = 0 < y" %
    (GEN "t"
     (REWRITE_RULE[LESS_0;(CONJUNCT1( CONJUNCT2( ADD_CLAUSES)))]
       (SPECL ["t";"0"]  LESS_ADD_EQ))
    );;

let RANGE_RULE th =
   (REWRITE_RULE [LESS_MONO_EQ;LESS_ADD_EQ;LESS_ADD_EQ1;LESS_0]
     ( CONV_RULE (TOP_DEPTH_CONV num_CONV) th ) );;


let EXPAND_TBLPTR_RULE s T asl =
         (REWRITE_RULE [(LINE 'tblC' asl);(LINE 'tblPtr' asl);T]
          (SPEC s  (CONJUNCT1( (REWRITE_RULE
          [regUnit_spec;bitFalse]
          (FIND_ASSUM2 (unit 'regUnit_spec') asl)) ))) );;

let INST_SIG_LIST t asl =
      ( ONCE_REWRITE_RULE [ADD1]
          (REWRITE_RULE [LESS_SUC_REFL; SYM_RULE ADD1;
 LESS_ADD_EQ; LESS_ADD_EQ1]
           (FIND_SPEC_UNIT t stable_sigs 'stable_sigs' asl) ));;

let NOT_FOR_TBLPTR_TAC =
   EXISTS_TAC "2"
   THEN PURE_ONCE_REWRITE_TAC [Next]
   THEN ASSUM_LIST(\asl. REWRITE_TAC [(LINE 'phase' asl);
       (LINE 'done' asl);(LINE 'ack' asl);(LINE 'tblPtr' asl)] )
   THEN  % determine rAddr(t+2) %
      ASSUM_LIST(\asl. ASSUME_TAC (
          (REWRITE_RULE [(LINE 'muxC' asl);T2;(LINE 'xlat' asl)]
          (FIND_SPEC_UNIT "t+1" muxUnit_spec 'muxUnit_spec' asl) )))
   THEN CONJ_TAC % create range and mmu_spec subgoals %
   THENL         % range subgoal %
     [RANGE_TAC "2" "1"
      ;          % mmu_spec part %
      ONCE_REWRITE_TAC [mmu_spec;stable_sigs]
      THEN STRIP_TAC
      THEN     % instantiate stable_sigs %
       POP_ASSUM(\thm. MAP_EVERY ASSUME_TAC ( CONJUNCTS (
          SYM_RULE( ONCE_REWRITE_RULE [ADD1]
          (REWRITE_RULE [LESS_SUC_REFL; SYM_RULE ADD1; SYM_RULE T2]
          (SPEC "t+1" thm))))))
      THEN FILTER_ASM_REWRITE_TAC(lines 'super')□
      THEN PURE_ONCE_REWRITE_TAC [superMode]
      THEN ALL_TAC
     ];;

 let UNPAIR_TAC l =
  POP_ASSUM_LIST(\asl.MAP_EVERY ASSUME_TAC ( (
           (rev(subtract asl[(el 1 asl)])) @
         [(REWRITE_RULE [PAIR_EQ] (el 1 asl))] )));;

 let CONTROL_LINE_TAC t len T =
       ASSUM_LIST(\asl. ASSUME_TAC(
```

```
        REWRITE_RULE ( CONJUNCTS len @
          [(LINE 'fdone' asl);T;(LINE 'phase' asl)])
            (SPEC t (MATCH_MP CTRL_UNIT_EXPAND
            (UNIT 'controlUnit_spec' asl) )) ));;

let RADDR_TAC t T =
      ASSUM_LIST(\asl. ASSUME_TAC(
       (REWRITE_RULE [(el 1 asl);T;(LINE 'xlat' asl)]
       (FIND_SPEC_UNIT t  muxUnit_spec 'muxUnit_spec' asl) )));;



%-----------------------------------------------------------------------
ABSTRACT MMU PROOF
-----------------------------------------------------------------------%

let MMU_PROOF = prove_thm
   ('MMU_PROOF',
    "!(r:*rep_ty) (vAddr rAddr :num->*address)(vData :num->*wordn)
     (rwe :num->RWE)(super reqIn xlat ack done :num->bool) mem
     (tblPtr :num->*wordn) (tblPtrADDR :num->*address) (phase :num->num).
    mmu_imp r vAddr vData rwe super tblPtr tblPtrADDR reqIn
             rAddr done ack xlat mem phase ==>
     !t. (phase t = 0) ==>
         (reqIn t) =>(?c. Next done (t,t+c) /\ ( phase (t+c) = 0 ) /\
      ((stable_sigs t (t+c) vAddr rwe tblPtrADDR vData
                                    mem super) ==>
                     (mmu_spec r (vAddr t) (rwe t) (tblPtrADDR t) (tblPtr t)
                                 (vData t)  (mem t) (super t) =
                      (ack(t+c), rAddr(t+c), tblPtr(t+c)) )))
                   | ( (ack   (t+1) = F) /\
                       (phase (t+1) = 0) /\
                       (tblPtr(t+1) = tblPtr(t) ) )",
   REPEAT GEN_TAC
   THEN PURE_REWRITE_TAC [mmu_imp;dataPath]
   THEN REPEAT STRIP_TAC
   THEN ASSUM_LIST(\asl. ASSUME_TAC( REWRITE_RULE [(el 1 asl)] (
         SPEC_ALL ( REWRITE_RULE [(UNIT 'controlUnit_spec' asl)]
                  (SPEC_ALL PHASE_0_IDLE)))))
   THEN ASSUM_LIST(\asl. ASSUME_TAC( REWRITE_RULE
        (CONJUNCTS PHASE_0_UNIQUE @ [(LINE 'phase' asl)])
        (SPEC "t" (MATCH_MP CTRL_UNIT_EXPAND
         (UNIT 'controlUnit_spec' asl) )) ))
   THEN  ASM_CASES_TAC "(reqIn t):bool"
   THEN ASSUM_LIST(\asl. REWRITE_TAC [(el 1 asl)] )
   THEN ASSUM_LIST(\asl. ASSUME_TAC(
    (REWRITE_RULE [(el 1 asl)](el 2 asl) ) ))
   THEN POP_ASSUM_LIST(\asl.MAP_EVERY
      ASSUME_TAC(rev(subtract asl[(el 3 asl)])))
   THEN
      ASSUM_LIST(\asl. ASSUME_TAC( CONJUNCT1( (REWRITE_RULE
          [regUnit_spec;bitFalse] (FIND_ASSUM2 (unit 'regUnit_spec') asl)) )))
   THEN ASSUM_LIST(\asl. ASSUME_TAC
         (REWRITE_RULE [(LINE 'tblC' asl)]
         (SPEC "t" (el 1 asl)) ) )
   THEN  % unpair control lines at (t+1) %
    ASSUM_LIST(\asl. ASSUME_TAC(
       (REWRITE_RULE [PAIR_EQ](el 3 asl)) ))
```

```
     THEN POP_ASSUM_LIST(\asl.MAP_EVERY
        ASSUME_TAC(rev(subtract asl[(el 4 asl)])))
% get rid of ~reqIn case %
   THENL
      [ ALL_TAC; ASM_REWRITE_TAC[] ]
   THEN % determine tblPtr (t+2)%
      ASSUM_LIST(\asl. ASSUME_TAC(
        (EXPAND_TBLPTR_RULE "t+1" T2 asl)))
   THEN
      ASSUM_LIST(\asl. ASSUME_TAC( (REWRITE_RULE
          (CONJUNCTS PHASE_1_UNIQUE @ [T2;(LINE 'phase' asl)])
          (SPEC "t+1" (MATCH_MP CTRL_UNIT_EXPAND
          (UNIT 'controlUnit_spec' asl) )) )))
%-------------- case analysis ----------%
   THEN  ASM_CASES_TAC "(super(t + 1)):bool"
   THENL [
     ASM_CASES_TAC "(wBIT (rwe(t + 1))):bool"
     THENL [
      ASM_CASES_TAC "(addrEq (r:~rep_ty) (wAddr t,tblPtrADDR t)):bool"
      THENL [
%--- (1.1.1) ------- super, wBIT, addrEq --------------------%
    % determine control_lines(t+2) %
    ASSUM_LIST(\asl. ASSUME_TAC( (REWRITE_RULE
      [PAIR_EQ; (LINE 'super' asl); (el 2 asl);
       (REWRITE_RULE [(el 1 asl)]
        (FIND_SPEC_UNIT "t" matchUnit_spec 'matchUnit_spec' asl) )
      ] (el 4 asl) )))
    THEN  % determine tblPtr(t+3) %
       ASSUM_LIST(\asl. ASSUME_TAC(
        (EXPAND_TBLPTR_RULE "t+2" T3 asl)))
    THEN % determine control_lines(t+3) %
       ASSUM_LIST(\asl. ASSUME_TAC( (REWRITE_RULE
           (CONJUNCTS PHASE_5_UNIQUE @ [(LINE 'phase' asl);T3;PAIR_EQ])
            (SPEC "t+2" (MATCH_MP CTRL_UNIT_EXPAND
            (UNIT 'controlUnit_spec' asl) )) )))
    THEN EXISTS_TAC "3"
    THEN PURE_ONCE_REWRITE_TAC [Next]
    THEN ASM_REWRITE_TAC []
    THEN CONJ_TAC  % create range and mmu_spec subgoals %
    THENL
      [ RANGE_TAC "3" "2"
       ;
        ONCE_REWRITE_TAC [mmu_spec]
        THEN STRIP_TAC
        THEN % expand stable_sigs for (t+1) and (t+2) %
          ASSUM_LIST(\asl. MAP_EVERY ASSUME_TAC ( CONJUNCTS (
             SYM_RULE( ONCE_REWRITE_RULE [ADD1]
              ((CONV_RULE LESS_CONV)
               (REWRITE_RULE
                [LESS_ADD_EQ;LESS_SUC_REFL; SYM_RULE ADD1; SYM_RULE T2]
              (SPEC "t+1" (REWRITE_RULE [stable_sigs] (el 1 asl))) )) ) )))
         THEN ASSUM_LIST(\asl. ASSUME_TAC
           (PURE_ONCE_REWRITE_RULE
            [(SYM_RULE ((TOP_DEPTH_CONV num_CONV) "2"))]
            (RANGE_RULE
            (SPEC "t+2" (REWRITE_RULE [stable_sigs] (el 7 asl))) ) ) )
        THEN
           FILTER_ASM_REWRITE_TAC(lines 'super')[]
           THEN PURE_ONCE_REWRITE_TAC [superMode]
```

```
        THEN ASSUM_LIST(\asl. REWRITE_TAC [ PAIR_EQ;
           (el 13 asl);(el 5 asl); % wBIT %(el 12 asl) % addrEq %])
        THEN % show vAddr t = rAddr(t+3) %
           RADDR_TAC "t+2" T3
           THEN ASSUM_LIST(\asl. REWRITE_TAC[(el 1 asl);(el 2 asl)] )
     ]
    ;
%--- (1.1.2) ------- super, wBIT, ~addrEq --------------------%
   ASSUM_LIST(\asl. ASSUME_TAC( REWRITE_RULE [(el 1 asl)]
      (FIND_SPEC_UNIT "t" matchUnit_spec 'matchUnit_spec' asl) ))
   THEN ASSUM_LIST(\asl. MAP_EVERY ASSUME_TAC( CONJUNCTS(
    REWRITE_RULE [PAIR_EQ;(el 1 asl);(el 3 asl);(el 4 asl)] (el 5 asl) )))
   THEN NOT_FOR_TBLPTR_TAC
   THEN ASSUM_LIST(\asl. REWRITE_TAC [PAIR_EQ;
         (REWRITE_RULE [SYM_RULE (LINE 'vAddr' asl)] (LINE 'rAddr' asl) );
         (el 18 asl) % addrEq %])
    ] ;
%--- (1.2) ------- super, ~wBIT  ---------------------%
   ASSUM_LIST(\asl. MAP_EVERY ASSUME_TAC( CONJUNCTS(
     REWRITE_RULE [PAIR_EQ;(el 1 asl);(el 2 asl)] (el 3 asl) )))
   THEN NOT_FOR_TBLPTR_TAC
   THEN ASSUM_LIST(\asl. REWRITE_TAC [PAIR_EQ;
         (REWRITE_RULE [SYM_RULE (LINE 'vAddr' asl)] (LINE 'rAddr' asl) );
         (REWRITE_RULE [SYM_RULE (LINE 'rwe' asl)] (el 17 asl) )
           % wBIT % ])
    ]
    ; ALL_TAC
       ] % end super cases %
%--- (2) ------- ~super  ----------------------------%
    THEN          % determine addOut(t+1) %
      ASSUM_LIST(\asl. ASSUME_TAC(
       REWRITE_RULE [(el 8 asl);
        (REWRITE_RULE [
         (FIND_SPEC_UNIT "t" splitUnit_spec 'splitUnit_spec' asl)]
         (FIND_SPEC_UNIT2 "t" mux3Unit_spec 'mux3Unit_spec' asl) )
        ;
        (REWRITE_RULE [LINE 'muxC' asl]
          (FIND_SPEC_UNIT "t" mux3Unit_spec 'mux3Unit_spec' asl) )
        ]
         (FIND_SPEC_UNIT "t" addUnit_spec 'addUnit_spec' asl) ))
   THEN  % determine latOut(t+1) %
     ASSUM_LIST(\asl. ASSUME_TAC(
       (REWRITE_RULE [(el 1 asl);(LINE '1C' asl)]
       (FIND_SPEC_UNIT "t" latchUnit_spec 'latchUnit_spec' asl) )))
   THEN  % determine fdone value at (t+2) %
     ASSUM_LIST(\asl. ASSUME_TAC( CONJUNCT2
       (REWRITE_RULE [T2; (LINE 'rReq' asl)]
       (FIND_SPEC_MEM_UNIT "t+1" asl)) ))
   THEN  % unpair control lines at (t+2) %
     POP_ASSUM_LIST(\asl.MAP_EVERY ASSUME_TAC ( (
          (rev(subtract asl[(el 5 asl)])) @
        [(REWRITE_RULE [PAIR_EQ;(LINE 'super' asl)] (el 5 asl))] )))
   THEN  % determine latOut(t+2) %
     ASSUM_LIST(\asl. ASSUME_TAC(
      (REWRITE_RULE [(el 1 asl);(LINE 'latOut' asl);T2]
      (FIND_SPEC_UNIT "t+1" latchUnit_spec 'latchUnit_spec' asl))))
   THEN  % determine rAddr(t+2) %
     RADDR_TAC "t+1" T2
% determine control lines at (t+3) %
```

```
  THEN
     CONTROL_LINE_TAC  "t+2" PHASE_2_UNIQUE T3
  THEN  % determine latOut(t+3) %
     ASSUM_LIST(\asl. ASSUME_TAC(
       (REWRITE_RULE [(LINE 'latOut' asl);T3;
       (REWRITE_RULE [PAIR_EQ](el 1 asl))]
       (FIND_SPEC_UNIT "t+2" latchUnit_spec 'latchUnit_spec' asl))))
  THEN  % determine memory value at (t+3) %
     ASSUM_LIST(\asl. ASSUME_TAC(
      (REWRITE_RULE [(LINE 'rReq' asl);T3]
      (FIND_SPEC_MEM_UNIT "t+2" asl))))
  THEN  % determine tblPtr (t+3)%
     ASSUM_LIST(\asl. ASSUME_TAC(
       (EXPAND_TBLPTR_RULE "t+2" T3 asl)))
  THEN  % unpair control lines at (t+3) %
     UNPAIR_TAC 4
% determine control lines at (t+4) %
  THEN
     CONTROL_LINE_TAC  "t+3" PHASE_2_UNIQUE T4
  THEN  % determine addOut(t+4) %
    ASSUM_LIST(\asl. ASSUME_TAC(
      REWRITE_RULE [PHASE_2_UNIQUE;T4;oneUnit_spec;
       (LINE 'latOut' asl);
       (REWRITE_RULE [(LINE 'muxC' asl);
         (FIND_SPEC_UNIT "t+3" splitUnit_spec 'splitUnit_spec' asl)]
         (FIND_SPEC_UNIT2 "t+3" mux3Unit_spec 'mux3Unit_spec' asl) );
       (REWRITE_RULE [(LINE 'muxC' asl)]
         (FIND_SPEC_UNIT "t+3" mux3Unit_spec 'mux3Unit_spec' asl) )
      ]
      (FIND_SPEC_UNIT "t+3" addUnit_spec 'addUnit_spec' asl) ))
   THEN  % determine secData reg value(t+4) %
     ASSUM_LIST(\asl. ASSUME_TAC( REWRITE_RULE
      [T4;bitFalse;(LINE 'tmpC' asl);(LINE 'data' asl)]
       (SPEC "t+3" (CONJUNCT1( (REWRITE_RULE [regUnit_spec]
       (FIND_ASSUM (unit 'regUnit_spec') asl)   ))))))
   THEN  % determine memory value at (t+4) %
     ASSUM_LIST(\asl. ASSUME_TAC(
      (REWRITE_RULE [(LINE 'rReq' asl);T4]
      (FIND_SPEC_MEM_UNIT "t+3" asl))))
   THEN  % determine tblPtr (t+4)%
     ASSUM_LIST(\asl. ASSUME_TAC(
       (EXPAND_TBLPTR_RULE "t+3" T4 asl)))
   THEN  % unpair control lines at (t+4) %
     UNPAIR_TAC 5
   THEN  % determine latOut(t+4) %
     ASSUM_LIST(\asl. ASSUME_TAC( (REWRITE_RULE
       [(el 1 asl);(LINE 'latOut' asl);(LINE 'addOut' asl);T4]
      (FIND_SPEC_UNIT "t+3" latchUnit_spec 'latchUnit_spec' asl))))
   THEN  % determine rAddr(t+4) %
     RADDR_TAC "t+3" T4
   THEN % determine securityUnit data(t+5) %
    ASSUM_LIST(\asl. ASSUME_TAC(
      (REWRITE_RULE [(LINE 'secData' asl);T5]
      (FIND_SPEC_UNIT "t+4" secUnit_spec 'secUnit_spec' asl) )))
 % determine control lines at (t+5) %
   THEN
     CONTROL_LINE_TAC  "t+4" PHASE_3_UNIQUE T5
   THEN  % determine memory value at (t+5) %
     ASSUM_LIST(\asl. ASSUME_TAC(
```

```
        (REWRITE_RULE [(LINE 'rReq' asl);T5]
        (FIND_SPEC_MEM_UNIT "t+4" asl))))
   THEN  % determine tblPtr (t+5)%
     ASSUM_LIST(\asl. ASSUME_TAC(
       (EXPAND_TBLPTR_RULE "t+4" T5 asl)))
   THEN  % unpair control lines at (t+5) %
     UNPAIR_TAC 3
   THEN   % determine addOut(t+6) %
     ASSUM_LIST(\asl. ASSUME_TAC(
       REWRITE_RULE [PHASE_1_UNIQUE;T6;
        (REWRITE_RULE [(LINE 'muxC' asl);(LINE 'data' asl);(LINE 'rAddr' asl);
         (FIND_SPEC_UNIT "t+5" splitUnit_spec 'splitUnit_spec' asl)]
         (FIND_SPEC_UNIT2 "t+5" mux3Unit_spec 'mux3Unit_spec' asl) )
        ;
        (REWRITE_RULE [LINE 'muxC' asl]
          (FIND_SPEC_UNIT "t+5" mux3Unit_spec 'mux3Unit_spec' asl) )
        ]
         (FIND_SPEC_UNIT "t+5" addUnit_spec 'addUnit_spec' asl) ))
   THEN  % determine tblPtr (t+6)%
     ASSUM_LIST(\asl. ASSUME_TAC(
       (EXPAND_TBLPTR_RULE "t+5" T6 asl)))
   THEN  % cases on validAccess %
     ASM_CASES_TAC  "validAccess (r:^rep_ty)
        (vAddr(t + 4),fetch r(mem(t + 2),rAddr(t + 2)),rwe(t + 4))
         /\
        (ofsLEq r(vAddr(t + 4),fetch r(mem(t + 2),rAddr(t + 2))))"
   THENL
     [
     ASSUM_LIST(\asl. ASSUME_TAC(
          (REWRITE_RULE [(el 1 asl)] (LINE 'secOK' asl)) ))
% determine control lines at (t+6) %
     THEN
       ASSUM_LIST(\asl. ASSUME_TAC(
         REWRITE_RULE (CONJUNCTS PHASE_3_UNIQUE @
          [(LINE 'fdone' asl);T6;(LINE 'phase' asl);PAIR_EQ;(el 1 asl)])
           (SPEC "t+5" (MATCH_MP CTRL_UNIT_EXPAND
           (UNIT 'controlUnit_spec' asl) )) ))
     THEN  % determine latOut(t+6) %
       ASSUM_LIST(\asl. ASSUME_TAC( (REWRITE_RULE
         [(el 1 asl);(LINE 'latOut' asl);(LINE 'addOut' asl);T6]
        (FIND_SPEC_UNIT "t+5" latchUnit_spec 'latchUnit_spec' asl))))
     THEN  % determine rAddr(t+6) %
       RADDR_TAC "t+5" T6
     THEN  % determine tblPtr (t+7)%
       ASSUM_LIST(\asl. ASSUME_TAC(
         (EXPAND_TBLPTR_RULE "t+6" T7 asl)))
% determine control lines at (t+7) %
     THEN
       CONTROL_LINE_TAC  "t+6" PHASE_4_UNIQUE T7
       THEN POP_ASSUM(\thm. ASSUME_TAC( REWRITE_RULE [PAIR_EQ] thm ))
     THEN  % determine latOut(t+7) %
       ASSUM_LIST(\asl. ASSUME_TAC( (REWRITE_RULE
         [(el 1 asl);(LINE 'latOut' asl);(LINE 'addOut' asl);T7]
        (FIND_SPEC_UNIT "t+6" latchUnit_spec 'latchUnit_spec' asl))))
     THEN  % determine rAddr(t+7) %
       RADDR_TAC "t+6" T7
  %          %
     THEN EXISTS_TAC "7"
     THEN PURE_ONCE_REWRITE_TAC [Next]
```

```
    THEN ASSUM_LIST(\asl. REWRITE_TAC[(LINE 'done' asl);(LINE 'phase' asl)])
    THEN CONJ_TAC  % create range and mmu_spec subgoals %
    THENL
     [ RANGE_TAC "7" "6"
      ;
       STRIP_TAC
       THEN  % write rAddr for time t %
         ASSUM_LIST(\asl. ASSUME_TAC( (REWRITE_RULE
          [(el 15 asl);(el 17 asl);
           (REWRITE_RULE [BETW_0_7_IS_5] (INST_SIG_LIST "t+5" asl) );
           (REWRITE_RULE [BETW_0_7_IS_4] (INST_SIG_LIST "t+4" asl) )]
          (LINE 'rAddr'  asl) )))
       THEN
          PURE_ONCE_REWRITE_TAC [mmu_spec]
          THEN ASSUM_LIST(\asl.  REWRITE_TAC [     (REWRITE_RULE
           [(REWRITE_RULE [BETW_0_7_IS_1] (INST_SIG_LIST "t+1" asl) )]
            (LINE 'super' asl) )])
          THEN PURE_REWRITE_TAC [userMode;legalAccess]
          THEN EXPAND_LET_TAC
       THEN  % write validAccess for time t %
          ASSUM_LIST(\asl. ASSUME_TAC ( (REWRITE_RULE
           [(REWRITE_RULE [BETW_0_7_IS_2] (INST_SIG_LIST "t+2" asl) );
            (REWRITE_RULE [BETW_0_7_IS_4] (INST_SIG_LIST "t+4" asl) );
            (el 29 asl)]
           (el 11 asl) ) ))
       THEN
          ASSUM_LIST(\asl. REWRITE_TAC [(el 1 asl);(el 2 asl);
           (LINE 'tblPtr' asl);PAIR_EQ] )
          THEN PURE_ONCE_REWRITE_TAC [vToR]
          THEN EXPAND_LET_TAC
          THEN REWRITE_TAC []
     ]
    ; % Case where  ~(validAccess ... /\ ofsLEq ... ) %
     ASSUM_LIST(\asl. ASSUME_TAC(
          (REWRITE_RULE [(el 1 asl)] (LINE 'secOK' asl)) ))
% determine control lines at (t+6) %
    THEN
       ASSUM_LIST(\asl. ASSUME_TAC(
          REWRITE_RULE (CONJUNCTS PHASE_3_UNIQUE @
           [(LINE 'fdone' asl);T6;(LINE 'phase' asl);PAIR_EQ;(el 1 asl)])
            (SPEC "t+5" (MATCH_MP CTRL_UNIT_EXPAND
            (UNIT 'controlUnit_spec' asl) )) ))
    THEN % determine latOut(t+6) %
       ASSUM_LIST(\asl. ASSUME_TAC( (REWRITE_RULE
         [(el 1 asl);(LINE 'latOut' asl);(LINE 'addOut' asl);T6]
        (FIND_SPEC_UNIT "t+5" latchUnit_spec 'latchUnit_spec' asl))))
    THEN  % determine rAddr(t+6) %
       RADDR_TAC "t+5" T6
    THEN EXISTS_TAC "6"
    THEN PURE_ONCE_REWRITE_TAC [Next]
    THEN ASSUM_LIST(\asl. REWRITE_TAC[(LINE 'done' asl);(LINE 'phase' asl)])
    THEN CONJ_TAC  % create range and mmu_spec subgoals %
    THENL
      [RANGE_TAC "6" "5"
      ;
       STRIP_TAC
       THEN  % write rAddr for time t %
          ASSUM_LIST(\asl. ASSUME_TAC( (REWRITE_RULE
           [(REWRITE_RULE [BETW_0_6_IS_5] (INST_SIG_LIST "t+5" asl) )]
```

```
          (LIKE 'rAddr'  asl) )))
        THEN
          PURE_ONCE_REWRITE_TAC [mmu_spec]
          THEN ASSUM_LIST(\asl.  REWRITE_TAC [     (REWRITE_RULE
           [(REWRITE_RULE [BETW_0_6_IS_1] (INST_SIG_LIST "t+1" asl) )]
            (LIKE 'super' asl) )])
          THEN PURE_REWRITE_TAC [userMode;legalAccess]
          THEN EXPAND_LET_TAC
        THEN  % write validAccess for time t %
          ASSUM_LIST(\asl. ASSUME_TAC ( (REWRITE_RULE
           [(REWRITE_RULE [BETW_0_6_IS_2] (INST_SIG_LIST "t+2" asl) );
            (REWRITE_RULE [BETW_0_6_IS_4] (INST_SIG_LIST "t+4" asl) );
            (el 25 asl)]
           (el 7 asl) ) ))
        THEN
          ASSUM_LIST(\asl. REWRITE_TAC [(el 1 asl);(el 2 asl);
           (LIKE 'tblPtr' asl);PAIR_EQ] )
          THEN REWRITE_TAC []
       ]
      ] % end validAccess cases %
);;


%----------------------------------------------------------------------------
MMU_PROOF =
|- !r vAddr rAddr vData rwe super reqIn xlat ack done mem tblPtr
    tblPtrADDR phase.
    mmu_imp
    r
    vAddr
    vData
    rwe
    super
    tblPtr
    tblPtrADDR
    reqIn
    rAddr
    done
    ack
    xlat
    mem
    phase ==>
    (!t.
      (phase t = 0) ==>
      (reqIn t =>
       (?c.
         Next done(t,t + c) /\
         (phase(t + c) = 0) /\
         (stable_sigs t(t + c)vAddr rwe tblPtrADDR vData mem super ==>
          (mmu_spec
           r
           (vAddr t)
           (rwe t)
           (tblPtrADDR t)
           (tblPtr t)
           (vData t)
           (mem t)
           (super t) =
           ack(t + c),rAddr(t + c),tblPtr(t + c)))) |
```

```
      ((ack(t + 1) = F) /\
       (phase(t + 1) = 0) /\
        (tblPtr(t + 1) = tblPtr t))))
Run time: 2419.4s
Intermediate theorems generated: 121858


File mmu_prf loaded
() : void
Run time: 2635.2s
Intermediate theorems generated: 122537
-------------------------------------------------------------------------------%
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1992	Contractor Report

4. TITLE AND SUBTITLE
Formal Verification of a Set of Memory Management Units

5. FUNDING NUMBERS
C NAS1-18586
WU 505-64-10-07

6. AUTHOR(S)
E. Thomas Schubert
K. Levitt
Gerald C. Cohen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Boeing Military Airplanes
P.O. Box 3707, M/S 7J-24
Seattle, WA 98124-2207

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23665-5225

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-189566

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Sally C. Johnson
Task 3 Report

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 60

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This document was generated in support of NASA contract NAS1-18586, Design and Validation of Digital Flight Control Systems Suitable for Fly-By-Wire Application, Task Assignment 3. with formal verification of embedded systems. In particular, this document describes the verification of a set of memory management units, The verification effort demonstrates the use of hierarchical decomposition and abstract theories. The MMUs can be organized into a complexity hierarchy. Each new level in the hierarchy adds a few significant features or modifications to the lower level MMU. The units described include: (1) a page check TLM (translation look-aside module); (2) a page check TLM with supervisor line; (3) a base a bounds MMU; (4) a virtual address translation MMU; and (5) a virtual address translation MMU with memory resident segment table.

14. SUBJECT TERMS
Verification, Validation, HOL, MMU, TLM, Virtual Address

15. NUMBER OF PAGES
94

16. PRICE CODE
A05

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std 239-18
298-102