Formal Verification of an MMU and MMU Cache

E. T. Schubert
Division of Computer Science
University of California, Davis

Abstract - **We describe the formal verification of a hardware subsystem consisting of a memory management unit and a cache. These devices are verified independently and then shown to interact correctly when composed. The MMU authorizes memory requests and translate virtual addresses to real addresses. The cache improves performance by maintaining a LRU list from the memory resident segment table.**

1 Introduction

Computers are being used in areas where no affordable level of testing is adequate. Safety and life critical systems must find a replacement for exhaustive testing to guarantee their correctness. Through a mathematical proof, hardware verification can formally demonstrate that a design satisfies its specification. However, hardware verification research has focused on device verification and has largely ignored system composition verification [1]. Our research is directed towards developing a methodology to verify a hardware base for a safety critical system. The top level hardware specification is apt to suggest a unitary implementation. This abstraction is convenient for verifying the correctness of software, however, the implementation consists of many different interacting components (CPU, memory, coprocessors, I/O devices, bus controllers, interrupt controllers, etc). This paper will describe our efforts to verify a subsystem consisting of a MMU and its cache using the HOL theorem prover [2].

The abstract MMU reported in [3] assumed a memory model where a read request was satisfied in one cycle. We extend the MMU to interact with an asynchronous memory. Additionally, the memory is more fully described; providing read and write functions. These changes required several significant changes to the abstract MMU proof script. The original proof strategy took advantage of the single cycle response time. The new strategy must use two arbitrary contents to define when memory words are returned from the memory-cache subsystem.

1.1 Related Work

Hardware verification requires that the design of a system is formally shown to satisfy its specification through a mathematical proof. Using theorem proving techniques, an expression describing the behavior of a device is proven to be equivalent in some sense to an expression describing the implementation structure of the device. These expressions concisely describe the behavior of devices in an unambiguous way. An additional benefit of hardware verification is that the behavioral semantics of the hardware are clearly defined. This provides an accurate basis for building correct software systems [5].

Hardware verification efforts thus far have focused primarily on a microprocessor as the base for computer systems [6], [7], [8], [9]. The processors verified have modeled small instruction sets and generally, have not included modern CPU features such as pipelines, multiple functional units and hardware interrupt support. Tamarack-3 [9] and AVM-1 [10] do provide sufficient interrupt support to connect with an interrupt controller. However, no system currently verified provides the memory management functions necessary to support a secure operating system.

Previous efforts to verify systems have constructed *vertically verified systems* with a microprocessor/memory as the system's base [11],[5],[1]. These efforts have aimed at illustrating how hardware verification can be used to close the semantic gap between high level languages and the computer's instruction set. However, the base for these systems (a microprocessor-memory pair) has been an unrealistic hardware platform.

1.2 HOL

The object language of HOL is a formulation of higher-order logic. Universally quantified variables are used to specify input and output device lines while internal device lines are existentially quantified. Conditional expressions are in the form: `cond` →`then-clause` | `else-clause`.

HOL provides the human verifier with a selection of tactics for use in goal-directed proofs. The tactics are very similar to the kinds of steps a human theorem prover would take in solving a goal. New tactics can be written that allow the theorem prover to be extended and customized for a particular task. New theorems can only be created in a controlled manner. All proofs can be reduced to one containing only the 8 primitive inference rules and 5 primitive axioms. High-level inference rules and tactics derived from some combination of primitive inference rules.

The following HOL expression defines an and gate implementation using an inverter and a nand gate. The existentially quantified variable p, represents an internal line which links the output of the nand gate with the input of the inverter.

$\vdash_{def}$ `andGate a b out = ∃p.nand a b p ∧ inv p out`

2 Memory Management Unit

[12] describes a number of memory management units which form a complexity hierarchy. By developing a sophisticated MMU in steps, the construction of the final proof appears to be more tractable. The simpler devices validate access to fixed length memory pages while the more complex devices authorize read, write or execute access to variable length segments and translate virtual addresses to real addresses. Many of these devices were designed and verified to the gate level. However, as the complexity increases, the emphasis of the verification shifts from gate level connections to the correctness of the operating system support features.

The device described below validates memory requests based on information maintained in a memory resident segment descriptor table. The location of the table is determined by a

segment table pointer register which is accessible only during supervisor operations. Each descriptor consists of two words: the first contains access control information (present bit, read/write/execute permissions, segment size) and the second serves as the 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. The MMU assumes the table provides an entry for all possible segment descriptors.

A generic theory for a class of MMU devices is defined where several functions and data types are left abstract. Using an abstract representation, details such as word length, can be omitted and the verification focuses only on the correctness of higher level abstraction (e.g. electronic block level rather than gate level). At a later point, the abstract representation can be instantiated with components that implement concrete behavior.

Support for generic or abstract theories is not directly provided by HOL. However, a theory about abstract representations can be defined in the object language [10]. An *abstract representation* contains a set of uninterpreted constants, types, abstract operations and a set of abstract objects. The semantics of the abstract representation is unspecified. Inside the theory, we do not know what the objects and operations mean. The abstract theory package also creates a set of selector functions [11] to extract desired functions from an abstract representation.

The abstract MMU representation generalizes traits particular to concrete implementations. Properties such as the the exact security policy and division of a virtual address into a segment identifier and offset (as well as the overall number of bits in an address), are hidden by functions which given an address, return the segment identifier or segment offset field (`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 one bit position (e.g. it adds a trailing zero bit).

The abstract functions selected by `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 which 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 bitVector comparison units defined for the more concrete units.

Additional abstract coercion functions are available to convert values between types. If the theory were instantiated, the abstract types would likely be implemented with bitVectors; leaving these functions unnecessary.

Memory is also treated abstractly. The abstract representation provides a fetch function `fetch`.

```
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)");
('val',        ":(*wordn -> num)"                );
('wordn',      ":(num->  *wordn)"                );
('address',    ":(*wordn -> *address)"           );
('fetch',      ":(*memory # *address) -> *wordn");
];;
```

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.

2.1 Specification

The specification is decomposed into several rules and ignores timing characteristics. The state and output environment of the MMU specification is a three-tuple consisting of a boolean acknowledgment, a memory address and the table pointer register value. The variable r in the definitions below is the abstract representation.

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

$\vdash_{def}$ 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) $\wedge$ (ofsLEq r) (vAddr,a))

$\vdash_{def}$ 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))

$\vdash_{def}$ superMode r vAddr rwe tblPtrADDR tblPtr data mem = ((wBIT rwe) $\wedge$ (addrEq r (vAddr,tblPtrADDR))) $\rightarrow$(T, vAddr, data) — (T, vAddr, tblPtr)

$\vdash_{def}$ userMode r vAddr rwe tblPtrADDR tblPtr data mem = legalAccess r vAddr tblPtr rwe mem $\rightarrow$(T, (vToR r vAddr tblPtr mem), tblPtr) — (F, vAddr, tblPtr)

$\vdash_{def}$ mmu_spec r vAddr rwe tblPtrADDR tblPtr data mem superv = superv $\rightarrow$superMode r vAddr rwe tblPtrADDR tblPtr data mem — userMode r vAddr rwe tblPtrADDR tblPtr data mem

---

2.2 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 straight forward with the exception of the memory and the control unit. These two units will be described in greater detail.

```
⊢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.(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 | F"
```

Memory Unit

As a first step towards composing devices, the memory specification used for the MMU verification is significantly expanded from the model used in [3]. The earlier model assumed a read-only memory that returned a value one clock cycle after a request was made. The new model defines asynchronous read and write operations. This model makes an implicit assumption that each memory request is satisfied before the next request is generated. Most of the new proof effort centered on establishing the correctness of the MMU control unit with the new memory specification.

```
⊢def memoryUnit_spec r req rwe addr data done mem =
     (done 0 = F)  ∧
     (∀ t.  (req t) →
                 (∃ t'. Next done (t, t+t')  ∧
                    (wBIT (rwe t) =>
                     ( (mem (t+t')  = store r (mem t,addr t,data t) ) ) |
                     ( (data (t+t') = fetch r (mem t,addr t) )  ∧
                       (mem (t+t') = mem t) ) ) )
                  |
                     ( (done (t+1) = F)  ∧
                       (mem  (t+1) = mem t) ) )
```

Control Unit

To process each memory request, the control unit will pass through several clocked phases. 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 state is maintained by the variable phase. 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 each phase change.

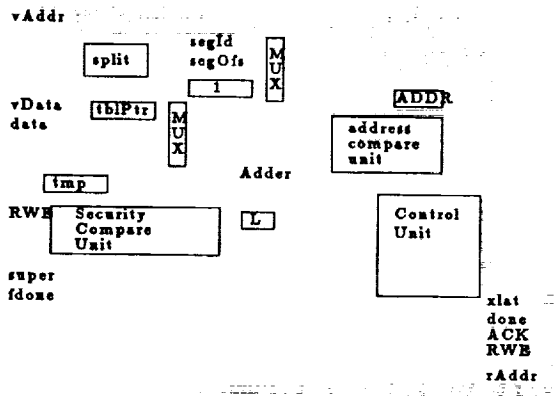

Figure 1: Abstract MMU Internal Block Diagram

The **dataPath** definition describes the interconnection between all the units other than the control unit.

```
⊢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 secData.
(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)
```

The implementation definition connects the datapath with the control unit. The state consists of the table pointer register value, the security Data register and the control unit phase (**tblPtr**, **secData**, **phase**). The input environment is provided by the system bus and the memory (**vAddr**, **vData**, **rwe**, **superv**, **reqIn**, **mem**). The output environment includes a real address and several control unit outputs (**rAddr**, **done**, **ack**, **xlat**). The memory address of the table pointer register is specified by the constant **tblPtrADDR**.

Correctness Statement

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**[9]. 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 $t1$

```
⊢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 →
            ((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))
```

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)   ∧ (tblPtrADDR t' = tblPtrADDR t1)  ∧  (mem t' = mem t1)
```

The correctness theorem states that if the implementation is in phase 0 and a memory request is made, the implementation will eventually respond (c time steps later), when the state of the implementation 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) )   ))
```

3 Memory Subsystem

An initial design integrated a FIFO cache stack inside the MMU but here we model a fully associative cache as part of the memory subsystem. The cache is described as a lookup table and implements a least recently used (LRU) replacement strategy. Each table entry consists of a key, a related data word, and a boolean indicating whether the entry is active. We will first describe the specification of the LRU replacement strategy in HOL, followed by the cache implementation.

```
TAB_ENTRY = ":bool#*address#*wordn" : type
TAB       = ":(^TAB_ENTRY)list" : type

⊢def live entry    =      (FST entry)
⊢def key entry     = (FST (SND entry))
⊢def content entry = (SND (SND entry))
```

Several auxiliary (recursive) definitions describe table operations below. When an entry is inserted into the top of table, the entry at the bottom will be lost only when the table is "full" (all entries are live). In this respect, the table acts as a queue.

```
⊢def (TAB_FULL tbl  0 = live (EL 0 tbl))  ∧
     (TAB_FULL tbl (SUC n) = (live (EL (SUC n) tbl)  ∧  TAB_FULL tbl n))
⊢def (TAB_INSERT tbl entry 0 = [entry])  ∧
     (TAB_INSERT tbl entry (SUC n) = (APPEND (TAB_INSERT tbl entry n)
            ((TAB_FULL tbl n) → [(EL n tbl)] | [(EL (SUC n) tbl)]) ) )
```

A table lookup is successful if there is a key match for one of the entries. For a table size of n, TAB_HIT returns (SUC n) if the lookup fails.

```
⊢def KEY_MATCH rep tbl sg:*address n =
     (live(EL n tbl) ∧  ((addrEq rep) (key(EL n tbl), sg)) )
⊢def (TAB_HIT rep tbl sg m 0 =
            ((KEY_MATCH rep tbl sg 0) → 0 | (SUC m)))  ∧
     (TAB_HIT rep tbl sg m (SUC n) =
            ((KEY_MATCH rep tbl sg (SUC n)) → (SUC n) | TAB_HIT rep tbl sg m n))
```

Frequently, a single matched entry must be invalidated. This can occur due to the LRU policy or a memory write operation. Occasionally, the entire cache must be invalidated at the request of the operating system. The LRU policy requires that if a key match occurs, the entry be inserted at the top of the table. By invalidating the matched entry before the insertion, a table overflow will not occur. LRU_LOOKUP returns the requested data value and the updated cache table.

```
⊢def ENTRY_INVALIDATE entry = (F ,key entry, content entry)
⊢def (TAB_INVALIDATE tbl 0 = [(ENTRY_INVALIDATE (EL 0 tbl))] )  ∧
     (TAB_INVALIDATE tbl (SUC n) =
               (APPEND (TAB_INVALIDATE tbl n) [(ENTRY_INVALIDATE (EL (SUC n) tbl))]))
⊢def (DEL_TAB_ENTRY rep tbl sg 0 =
               ((KEY_MATCH rep tbl sg 0) → [(ENTRY_INVALIDATE (EL 0 tbl))] |
                                           [(EL 0 tbl)] ))  ∧
     (DEL_TAB_ENTRY rep tbl sg (SUC n) =
               (APPEND (DEL_TAB_ENTRY rep tbl sg n)
                       ((KEY_MATCH rep tbl sg (SUC n))
                →                          [(ENTRY_INVALIDATE (EL (SUC n) tbl))]
                |                          [(EL (SUC n) tbl)] )))
```

```
⊢def  LRU_REPL rep tbl entry n = TAB_INSERT (DEL_TAB_ENTRY rep tbl (key entry) n)
                     entry n
⊢def LRU_LOOKUP rep mem tbl n addr data newTbl =
     let who = (TAB_HIT rep tbl addr n n) in
     ((who = (SUC n))
               → ( data = fetch rep( mem, addr)  ∧
                   newTbl =  TAB_INSERT tbl (T,addr,(fetch rep(mem,addr) )) n )
               |   (data = (content (EL who tbl)  ∧
                    newTbl = LRU_REPL rep tbl (EL who tbl) n)
```

Using the above definitions, the cache-memory subsystem can be defined. This definition replaces memoryUnit_spec in the MMU specification and the new system is verified in a similar manner. The proof shows that the cache/memory system is consitent with the MMU memory model requirements.

```
⊢def cache_mem_spec r req rwe addr data done mem tbl n =
      (done 0 = F)  ∧
      (∀ t.  (req t) →
         (∃ t'. Next done (t, t+t')  ∧
             (wBIT (rwe t) =>
                ( (mem (t+t')  = store r (mem t,addr t,data t) )  ∧
                  (tbl (t+t')  = DEL_TAB_ENTRY r (tbl t) (addr t) n ) |
                ( LRU_LOOKUP r (mem t) (tbl t) n (addr t)
                               (data (t+t')) (tbl (t+t')) )   ∧
                  (mem (t+t') = mem t) ) ) )
          |
              ( (done (t+1) = F)      ∧
                (mem  (t+1) = mem t)  ∧
                (tbl  (t+1) = tbl t) ) )
```

Cache Implementation

The cache implementation consists of a control unit and a stack of cache cells. Cache cells are the instantiation of the table entries described above—their state consisting of

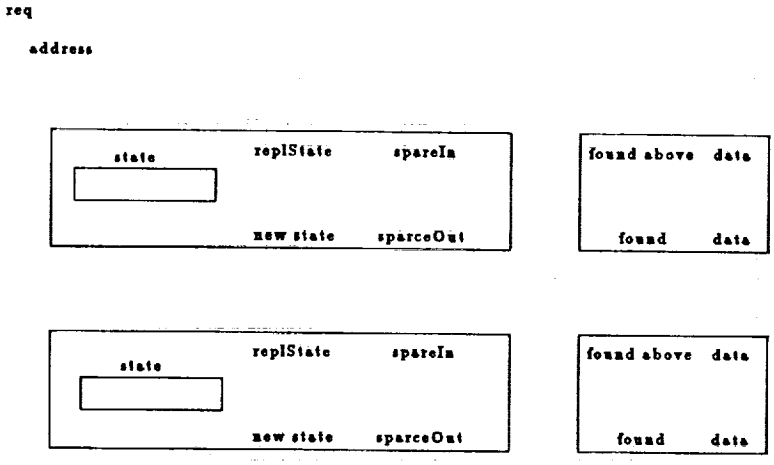

Figure 2: Cache Cell Stack

the three tuple: (valid, address key, data). The action of each cache cell is defined by a two bit function code (req) sent by the cache control unit. The stack is formed by joining the outputs of a cache unit to the inputs of the next.

```
⊢def (cache_block r state req sparceIn foundIn addr replState dataIn 0 =
      cache_cell rep 0 req addr replState (state,sparceIn,foundIn,dataIn))
   ∧
  (cache_block rep state req sparceIn foundIn addr replState dataIn (SUC n) =
     (cache_cell rep (SUC n) req addr (EL n state)
       (cache_block rep state req sparceIn foundIn addr replState dataIn n)))
```

```
⊢def cache_cell_spec rep n req addr replState (stateIn,sparceIn,foundIn,dataIn) =
  let state = (EL n stateIn) in
  let match = ( addrEq rep(addr,key state)  ∧  live state ) in
  (req = (F,F)) →  % IDLE %
        (stateIn,     foundIn, (sparceIn  ∨  ~live state),    dataIn )       |
  (req = (F,T)) →  % INVALIDATE ON MATCH %
     ( match →
        (SET_EL n stateIn(F,key state,content state), T, T, content state ) |
        (stateIn,     foundIn,  (sparceIn  ∨  ~live state),   dataIn)     )  |
  (req = (T,F)) →  % INVALIDATE %
        (SET_EL n stateIn(F,key state,content state), foundIn, T, dataIn )  |
  %req = (T,T)  →     PUSH DOWN  %
     ( sparceIn →
        (stateIn,     foundIn,     T,    content state )                     |
        (SET_EL n stateIn replState, foundIn,    F,              dataIn) )
```

When a memory request is made, the control unit signals each cache cell to invalidate its entry if its key matches the input address (F,T). Memory write requests are also passed through to memory. If a read request is pending and the value is not in the cache, the value is fetched from memory. We assume one clock cycle is needed to read a value out of the cache if it is available. After the value fetch step is completed, the control unit pushes the new value onto the cache cell stack by issuing request (T,T).

To model memory, the cache implementation uses the same memory unit specification (`memoryUnit_spec`) stated previously. We then verify that the implementation behaves as specified. The implementation also provides a means of invalidating the entire table (request (`T,F`), however, this function is not present currently in the specification.

4 Summary

We have described the formal verification of an MMU and cache/memory subsystem. The MMU has been verified to perform correctly with an asynchronous memory model. The cache specification defines an LRU replacement policy which is implemented by an electronic block level design. The cache is also demonstrated to be consitent with MMU memory model requirements.

It has been convenient to represent the behavior of devices using abstract representations. This mechanism allows the verification effort to focus on the correctness of higher level abstraction. To verify a more concrete implementation, the abstract representation can be instantiated with components that implement concrete behavior. Extending this example, we plan to demonstrate how a complete system composed of many devices can be shown to correctly implement an abstract system specification.

References

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

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

[3] E. T. Schubert and K. N. Levitt, "Verification of Memory Management Units," *2nd IFIP Working Conference on Dependable Computing for Critical Applications.*, February 1991.

[4] E. T. Schubert, "Formal Verification of an LRU Cache in Higher Order Logic," technical report CSE-91-, University of California, Davis, September 1991.

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

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

[7] W. A. Hunt, "A Verified Microprocessor," Technical Report 47, The University of Texas at Austin, Dec. 1985.

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

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

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

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

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