Applications of the **DE2** Language

Warren A. Hunt, Jr. and Erik Reeber

March 22, 2006

Abstract

We have developed a formal verification approach that permits the mechanical verification of circuit generators and hardware optimization procedures, as well as existing hardware designs. Our approach is based on deeply embedding the **DE2** HDL into the ACL2 logic [3]; we use the ACL2 theorem-proving system to verify the circuit generators. During circuit generation, a circuit generator may generate circuits based on variety of non-functional criteria. For example, a circuit generator may produce different structural circuit descriptions depending on wire lengths, circuit primitives, target technology, and circuit topology.

In this paper, we show how we have applied the **DE2** system to a simple circuit generator—the n-bit ripple-carry adder. We then show how we have applied the **DE2** system to the verification of components of the TRIPS microprocessor design.

1 Introduction

We have developed a hardware description language, **DE2**, which has a number of features that make it suitable for the verification of modern hardware designs. **DE2** has a simple semantics and includes capabilities for specifying and verifying non-functional properties, circuit generators, and hardware optimization programs.

Our verification system is based on the deep embedding of **DE2** within the ACL2 logic and theorem prover. Furthermore, we have built a fully automatic SAT-based proof engine that can verify invariants of machines designed in **DE2**. This SAT-based proof engine involves an extension to the ACL2 theorem-proving system so that it can use external SAT solvers.

In this paper, we discuss related work in Section 2. We provide some background on the ACL2 theorem prover, the **DE2** language, and our verification system, in Section 3. Next, in Section 4, we show how to apply our system to the verification of a ripple-carry adder. In Section 5, we show how we apply our system to the verification of a communication protocol used in the TRIPS processor.

2 Related Work

This work builds on our previous work with the **DE2** language [3], as well as our previous work with the verification of the FM9001 microprocessor [8]. In our earlier work, we only employed theorem-proving techniques, but our current effort also permits the use of SAT and BDD based techniques. In addition, our current approach to verifying circuit generators permits a circuit generator to make choices based on non-functional criteria. For example, a circuit generator may produce different structural circuit descriptions depending on wire lengths, circuit primitives, target technology, and circuit topology.

This work is similar in spirit to work by the functional language community to generate regular circuits using functional programs. For instance, the WIRED language has been used to improve performance of multipliers by incorporating layout information into the design of circuit generators [1].

Many model-checkers, and other automated verification tools, verify FSM properties automatically. UCLID, for example, uses SAT solvers to verify high-level FSMs with uninterpreted function symbols [5]. Another example is the FORTE tool, which has been used at Intel to verify components of processor designs [2].

3 Background

3.1 The ACL2 Theorem Prover

ACL2 stands for A Computation Logic for Applicative Common Lisp. The ACL2 language is a functional subset of Common Lisp. For a thorough description of ACL2 see Kaufmann, Manolios, and Moore's book [4].

```
(defun concatn (n a b)
  (if (zp n)
      b
    (cons (car a)
          (concatn (- n 1) (cdr a) b))))

(defun uandn (n a)
  (if (zp n)
      t
    (if (car a)
        (uandn (- n 1) (cdr a))
      nil)))

(defun bequiv (a b)
  (if a b (not b)))

(defthm example-thm
  (implies (and (not (zp x))
                (not (zp y)))
           (bequiv (uandn (+ x y) (concatn x a b))
                   (and (uandn x a) (uandn y b)))))
```

Figure 1: ACL2 Definitions and a Bit-Vector Concatenation Theorem

Figure 1 illustrates several ACL2 definitions. Here, function `concatn` concatenates two bit vectors, `uandn` returns the conjunction of the bits in a bit vector. The ACL2 function `bequiv` determines whether two ACL2 values represent the same Boolean value. We also make use of the built-in ACL2 function `(zp n)`, which returns `nil` if `n` is a positive integer and `t` otherwise.

The functions `uandn` and `concatn` are defined recursively. In order for such definitional axioms to be added to the ACL2 theory, one must first prove that the definition terminates for all inputs. In this case, the proof follows from the fact that the function argument `n` decreases on every recursive call.

Figure 1 also illustrates an ACL2 theorem. This theorem states that the `unary-and` of the concatenation of two bit vectors is equivalent to the conjunction of the `unary-and` of each individual bit vector.

3.2 The DE2 Evaluator

The semantic evaluation of a **DE2** design proceeds by binding actual (evaluated) parameters (both the inputs and the current state) to the formal parameters of the module to be evaluated; this in turn causes the evaluation of each submodule. This evaluation process is recursively repeated until a primitive module is encountered. This recursive-descent/ascent part of the evaluation can be thought of as performing all of the "wiring"; values are "routed" to appropriate modules and results are collected and passed along to other modules or become primary outputs. Finally, to evaluate a primitive, a specific primitive evaluator is then called after binding the necessary arguments. This set of definitions is composed of four (two groups of) functions (given below), and these functions contain an argument that permits different primitive evaluators to be used.

The following four functions completely define the evaluation of a netlist of modules, no matter which type of primitive evaluation is specified. The functions presented in this section constitute the entire definition of the simulator for the **DE2** language. This definition is small enough to allow us to reason with it mechanically, yet it is rich enough to permit the definition of a variety of evaluators. The `se` function evaluates a module and returns its outputs as a function of its inputs and its internal state. The `de` function evaluates a module and returns its next state; this state will be structurally identical to the module's current state, but with updated values. Both `se` and `de` have sibling functions, `se-occ` and `de-occ` respectively, that iterate through each sub-module referenced in the body of a module definition. We present the `se` and `de` evaluator functions to make clear the importance we place on making the definition compact.

The `se` and `de` functions both have a `flg` argument that permits the selection of a specific primitive evaluator. The `fn` argument identifies the name of a module to evaluate; its definition should be found in the `netlist`. The `ins` and `st` arguments provide the primary inputs and the current state of the `fn` module. The `params` argument allows for parametrized modules; that is, it is possible to define modules with wire and state sizes that are determined by this parameter. The `env` argument permits configuration or test information to be passed deep into the evaluation process.

The `se-occ` function evaluates each occurrence and returns an environment that includes values for all internal signals. The `se` function returns a list of outputs by filtering the desired outputs from this environment. To compute the outputs as functions of the inputs, only a single pass is required.

```
(defun se (flg fn params ins st env netlist)
  (if (consp fn)
      ;; Primitive Evaluation.
      (cdr (flg-eval-lambda-expr flg fn params ins env))
    ;; Evaluate submodules.
    (let ((module (assoc-eq fn netlist)))
      (if (atom module)
          nil
        (let-names
         (m-params m-ins m-outs m-sts m-occs)
         (m-body module)
         (let*
           ((new-env     (add-pairlist m-params params nil))
            (new-env     (add-pairlist (strip-cars m-ins)
                                       (flg-eval-list flg ins env)
                                       new-env))
            (new-env     (add-pairlist m-sts
                                       (flg-eval-expr flg st env)
                                       new-env))
            (new-netlist (delete-assoc-eq-netlist fn netlist)))
           (assoc-eq-list-vals
            (strip-cars m-outs)
            (se-occ flg m-occs new-env new-netlist)))))))

(defun se-occ (flg occs env netlist)
  (if (atom occs)  ;; Any more occurrences?
      env
    ;; Evaluate specific occurrence.
    (let-names
     (o-name o-outs o-call o-ins)
     (car occs)
     (se-occ flg (cdr occs)
             (add-pairlist
              (o-outs-names o-outs)
              (flg-eval-list
               flg (parse-output-list
                    o-outs
                    (se flg (o-call-fn o-call)
                        (flg-eval-list flg
                                       (o-call-params o-call)
                                       env)
                        o-ins o-name env netlist))
               env)
              env)
             netlist))))
```

Similarly, the functions `de` and `de-occ` perform the next-state computation for a module's evaluation; given values for the primary inputs and a structured state argument, these two functions compute the next state of a specified module. This result state is structured isomorphically to its input (internal) state. Note that the definition of `de` contains a reference to the function `se-occ`; this reference computes the value of all internal signals for the module whose next state is being computed. This call to `se-occ` represents the first of two passes through a module description when `DE` is computing the next state.

```
(defun de (flg fn params ins st env netlist)
  (if (consp fn)
      (car (flg-eval-lambda-expr flg fn params ins env))
    (let ((module (assoc-eq fn netlist)))
      (if (atom module)
          nil
        (let-names
         (m-params m-ins m-sts m-occs) (m-body module)
         (let*
           ((new-env     (add-pairlist m-params params nil))
           (new-env     (add-pairlist (strip-cars m-ins)
                                      (flg-eval-list flg ins env)
                                      new-env))
           (new-env     (add-pairlist m-sts
                                      (flg-eval-expr flg st env)
                                      new-env))
           (new-netlist (delete-assoc-eq-netlist fn netlist))
           (new-env     (se-occ flg m-occs new-env new-netlist)))
          (assoc-eq-list-vals
           m-sts
           (de-occ flg m-occs new-env new-netlist)))))))

(defun de-occ (flg occs env netlist)
  (if (atom occs)
      env
    (let-names
     (o-name o-call o-ins) (car occs)
     (de-occ flg (cdr occs)
             (cons
              (cons
               o-name
               (de flg (o-call-fn o-call)
                   (flg-eval-list flg (o-call-params o-call) env)
                   o-ins o-name env netlist))
              env)
             netlist))))
```

This completes the entire definition of the **DE2** evaluation semantics. This clique of functions is used for all different evaluators; the specific kind of evaluation is determined by the `flg` input. We have proved a number of lemmas that help to automate the analysis of **DE2** modules. These lemmas allow us to hierarchically verify FSMs represented as **DE2** modules. We have also defined simple functions that use `de` and `se` to simulate a **DE2** design through any number of cycles.

An important aspect of this semantics is its brevity. Furthermore, since we specify our semantics in the formal language of the ACL2 theorem prover, we can mechanically and hierarchically verify properties about any system defined using the **DE2** language.

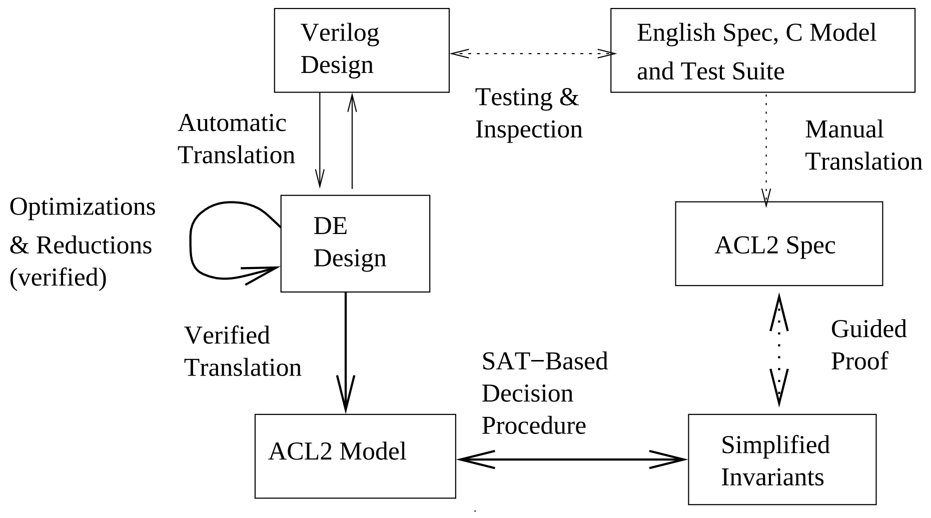

Figure 2: An overview of the **DE2** verification system

3.3 The Verification System

Having an evaluator for **DE2** written in ACL2 enables many forms of verification. In Figure 2, we illustrate our verification system, which is built around the **DE2** language.

We typically use the **DE2** verification system to verify Verilog designs. These designs are denoted in the upper left of Figure 2. Currently, our subset of Verilog includes arrays of wires (bit vectors), instantiations of modules, assignment statements, and some basic primitives (e.g. `&`, `?:` and `|`). We also allow the instantiation of memory (array) modules and vendor-defined primitives.

We have built a translator that translates a Verilog description into an equivalent **DE2** description. Our translator parses the Verilog source text into a Lisp expression, and then an ACL2 program converts this Lisp expression into a **DE2** description.

We have also built a translator that converts a **DE2** netlist into a cycle-accurate ACL2 model. This translator also provides an ACL2 proof that the **DE2** description is equivalent to the mechanical produced ACL2 model. The process of translating a **DE2** description into its corresponding ACL2 model includes a partial cone-of-influence reduction; an ACL2 function is created for each module's output and parts of the initial design which are irrelevant to that output are removed. The **DE2** to ACL2 translator allows us to enjoy both the advantages of a shallow embedding (e.g. straightforward verification) and the advantages of a deep embedding (e.g. syntax resembling Verilog).

We start with an informal specification of the design in the form of English documents, charts, graphs, C-models, and test code which is represented in the upper right of Figure 2. This information is converted manually into a formal ACL2 specification. Using the ACL2 theorem prover, these specifications are simplified into a number of invariants and equivalence properties. If these properties are simple enough to be proven by our SAT-based decision procedure, we prove them automatically; otherwise, we simplify such conjectures using the ACL2 theorem prover until we can successfully appeal to some automated decision procedure.

We also use our system to verify sets of **DE2** descriptions. This is accomplished by writing ACL2 functions that generate **DE2** descriptions, and then proving that these functions always produce circuits that satisfy their ACL2 specifications.

Since **DE2** descriptions are represented as ACL2 constants, functions that transform **DE2** descriptions can be verified using the ACL2 theorem prover. By converting from Verilog to **DE2** and from **DE2** to back into Verilog, we can use **DE2** as an intermediate language to perform verified optimizations. Another use of this feature involves performing reductions or optimizations on **DE2** specifications prior to verification. For example, one can use a decision procedure to determine that two **DE2** circuits are equivalent and then use this fact to avoid verifying properties of a less cleanly structured description.

We can also build static analysis tools, such as extended type checkers, in **DE2** by using annotations. In **DE2**, annotations are first-class objects (i.e. annotations are not embedded in comments). Such type checkers, since they are written in ACL2, can be analyzed and can also assist in the verification of **DE2** descriptions. Furthermore, annotations can be used to embed information into a **DE2** description to assist with synthesis or other post-processing tools.

4 Ripple-Carry Adder Generator Verification

In this section we present a definition of a simple parametrized ripple-carry adder to show how the **DE2** verification system is applied to verify circuit generators. The following two ACL2 functions generate the **DE2** definition of the top-level module of the ripple-carry adder:

```
(defun generate-ripple-occs (n)
  (if (zp n)
      nil
    (append (generate-ripple-occs (1- n))
            `((,(de-make-n-name 'carry n)
               ((q ,(1- n) ,(1- n)) (carry ,n ,n))
               (full-adder)
               ((g x ,(1- n) ,(1- n)) (g y ,(1- n) ,(1- n))
                (g carry ,(1- n) ,(1- n))))))))

;; Make an n-bit ripple-carry adder
(defun generate-ripple-carry (n)
  `(,(de-make-n-name 'ripple-carry n)
    (type module)
    (params )
    (outs (q ,n) (c_out 1))
    (ins (x ,n) (y ,n) (c_in 1))
    (sts )
    (wires (carry ,(1+ n)))
    (occs
     (carry_0 ((carry 0 0)) (bufn 1) ((g c_in 0 0)))
     . ,(append (generate-ripple-occs n)
                `((carry_out ((c_out 0 0))
                             (bufn 1)
                             ((g carry ,n ,n))))))))
```

The function `generate-ripple-occs` creates the occurrence list by recursively laying down one full-adder for each output bit. The function `generate-ripple-carry` then uses this occurrence list to create the top-level ripple-carry adder definition. For example, the following is the four bit ripple-carry adder produced by `(generate-ripple-carry 4)`:

```
(RIPPLE-CARRY_4
 (TYPE MODULE)
 (PARAMS)
 (OUTS (Q 4) (C_OUT 1))
 (INS (X 4) (Y 4) (C_IN 1))
(STS)
(WIRES (CARRY 5))
(OCCS (CARRY_0 ((CARRY 0 0))
               (BUFN 1)
               ((G C_IN 0 0)))
      (CARRY_1 ((Q 0 0) (CARRY 1 1))
               (FULL-ADDER)
               ((G X 0 0) (G Y 0 0) (G CARRY 0 0)))
      (CARRY_2 ((Q 1 1) (CARRY 2 2))
               (FULL-ADDER)
               ((G X 1 1) (G Y 1 1) (G CARRY 1 1)))
      (CARRY_3 ((Q 2 2) (CARRY 3 3))
               (FULL-ADDER)
               ((G X 2 2) (G Y 2 2) (G CARRY 2 2)))
      (CARRY_4 ((Q 3 3) (CARRY 4 4))
               (FULL-ADDER)
               ((G X 3 3) (G Y 3 3) (G CARRY 3 3)))
      (CARRY_OUT ((C_OUT 0 0))
                 (BUFN 1)
                 ((G CARRY 4 4)))))
```

We next define a ripple-carry adder in ACL2 which follows the same structure as the one defined in **DE2**. The following is the top-level definition of the ACL2 ripple-carry adder and the main theorem we prove about it:

```
(defun acl2-ripple-adder (n x y c_in)
  (if (zp n)
      (list nil (get-sublist c_in 0 0))
    (let* ((adder_1b
            (acl2-full-adder (get-sublist x 0 0)
                             (get-sublist y 0 0)
                             (get-sublist c_in 0 0)))
           (sub_adder (acl2-ripple-adder (1- n)
                                         (nth-cdr 1 x)
                                         (nth-cdr 1 y)
                                         (cadr adder_1b))))

      (list (append-n 1 (car adder_1b) (car sub_adder))
            (append-n 1 c_in (cadr sub_adder))))))
```

```
(defthm acl2-ripple-adder-adds
  (implies
   (and (equal n (len a))
        (equal (len b) (len a)))
   (equal (v-to-nat
           (car (acl2-ripple-adder n a b
                                    (list (bool-fix c_in)))))
          (mod-2-n (+ (if c_in 1 0)
                      (v-to-nat a)
                      (v-to-nat b))
                   n))))
```

The above theorem states that the ACL2 functional definition of the ripple-carry adder implements modular addition, as defined by ACL2's addition axioms. We prove this theorem by making use of ACL2's induction and simplification proof engines, as well as the library of lemmas that has been created to assist ACL2 users in the verification of arithmetic properties.

Next we verify the theorem below:

```
(defthm generate-ripple-se-rewrite
  (implies
   (and (not (zp n))
        (generate-ripple-carry-& n netlist))
   (equal
    (se 'bvev
        (de-make-n-name 'ripple-carry n)
        params ins st env netlist)
    (let ((x (get-value 'bvev ins env))
          (y (get-value 'bvev (cdr ins) env))
          (c_in (get-sublist (get-value 'bvev
                                        (cddr ins)
                                        env)
                             0
                             0)))
      (list (car (acl2-ripple-adder n x y c_in))
            (get-sublist (cadr (acl2-ripple-adder n
                                                  x
                                                  y
                                                  c_in))
                         n
                         n))))))
```

This theorem states that, given certain conditions, the **DE2** ripple-carry adder produces the same result as the ACL2 ripple-carry adder. The hypotheses of the theorem are that the number of bits is a positive integer and that the ripple-carry adder modules occurs in the given netlist, along with its submodules. This theorem is proven using ACL2's induction proof engine, which we use to show that each occurrence produced by a recursive step of `generate-ripple-occs` corresponds to a recursive step in `acl2-ripple-adder`.

Once we have verified `generate-ripple-se-rewrite`, we can prove the final theorem below:

```
(defthm generate-ripple-se-adds
  (implies
   (and (not (zp n))
        (generate-ripple-carry-& n netlist)
        (equal (len (get-value 'bvev ins env)) n)
        (equal (len (get-value 'bvev (cdr ins) env)) n))
   (equal
    (v-to-nat (car (se 'bvev
                       (de-make-n-name 'ripple-carry n)
                       params ins st env netlist)))
    (let ((x (get-value 'bvev ins env))
          (y (get-value 'bvev (cdr ins) env))
          (c_in (get-sublist (get-value 'bvev (cddr ins) env)
                             0
                             0)))
      (mod-2-n (+ (if (car c_in) 1 0)
                  (v-to-nat x)
                  (v-to-nat y))
               n)))))
```

This theorem states that if the `n`-bit, ripple-carry adder module is in the netlist, along with its submodules, and the first two inputs are `n` bit, bit vectors, then the natural number representation of the output of the ripple-carry adder is equal to the modular addition of its inputs.

Note we proved this theorem entirely using the standard ACL2 theorem proving techniques, without the use of SAT solvers or BDDs. That is because we completed this proof before our SAT-based proof engine was fully in place. In the next section we will show how we are verifying next-generation hardware using a mixture of SAT-solving and theorem proving.

5 Verifying TRIPS Processor Components

We are using our verification system to verify components of the TRIPS processor. The TRIPS processor is a prototype of a next-generation processor that has been designed at the University of Texas [7] and being built by IBM. One novel aspect of the TRIPS processor is that its memory is divided into four pieces; each piece has its own memory control tile, with its own cache and Load Store Queue (LSQ). We plan to verify the LSQ design, based on the design described in Sethumadhavan et. al., [6], using our verification system. In this section, we present our verification of a part of the LSQ that manages communication with other LSQs.

We first use our verification system, mentioned in Section 3.3, to "compile" the Verilog design that implements the LSQ communication protocol into a **DE2** module. We then used our automatic translation engine to compile the **DE2** description into an ACL2 model and prove their equivalence relative to the **DE2** semantics.

5.1 Verification of the Exception Protocol

One reason that the LSQ units must communicate is to conglomerate exceptions generated in various tiles into a single mask. Figure 3 presents an overview of the protocol that conglomerates exceptions. Each tile receives a four-bit input denoting the exception generated this cycle—a three-bit address plus a one-bit enable signal. The exceptions are decoded into an eight-bit mask, that each tile passes to the tile above it. Exceptions are removed when the instruction that generated the exception is flushed. The schematic of the design that implements this protocol is shown in Figure 4.

To verify the multi-tile design in Figure 3, we prove that it is equivalent to the single-tile design in Figure 5. This equivalence is broken into the following two properties:

```
(defthm exception-safety
 (implies
  (and (integerp tao)
       (<= 0 tao)
       (Tth-inputs-goodp tao input-list))
  (submaskp
   8
   (out-udt_miss_ordering_exceptions
```

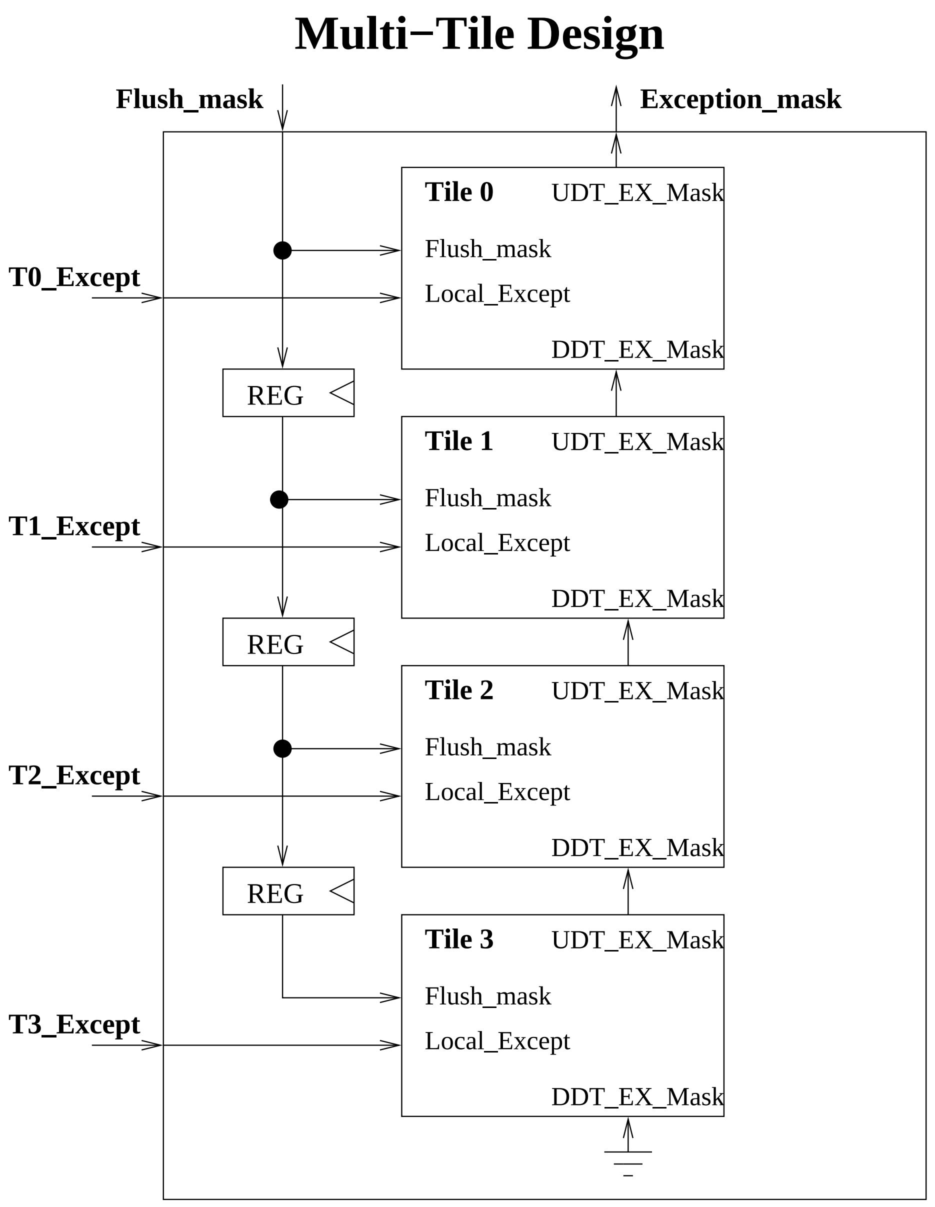

Figure 3: An overview of the four tile exception protocol design.

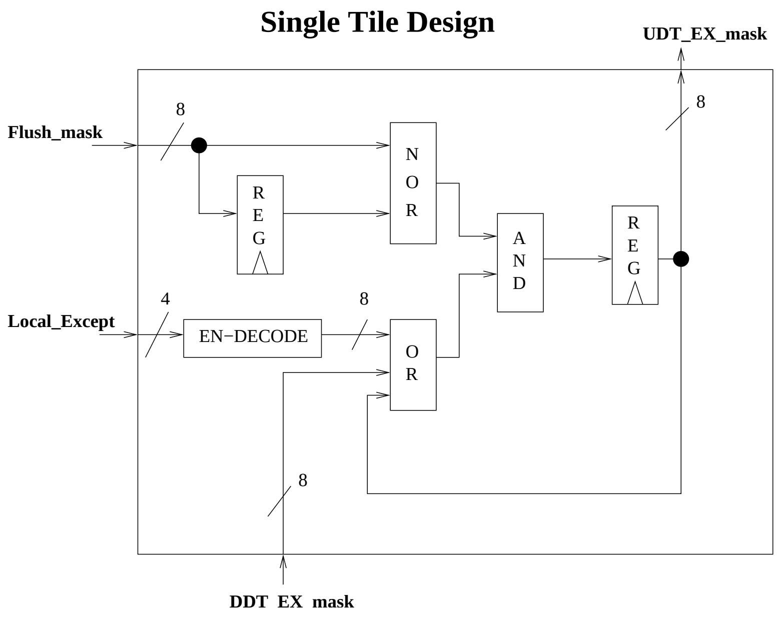

Figure 4: A look into the internals of a tile within the exception protocol.

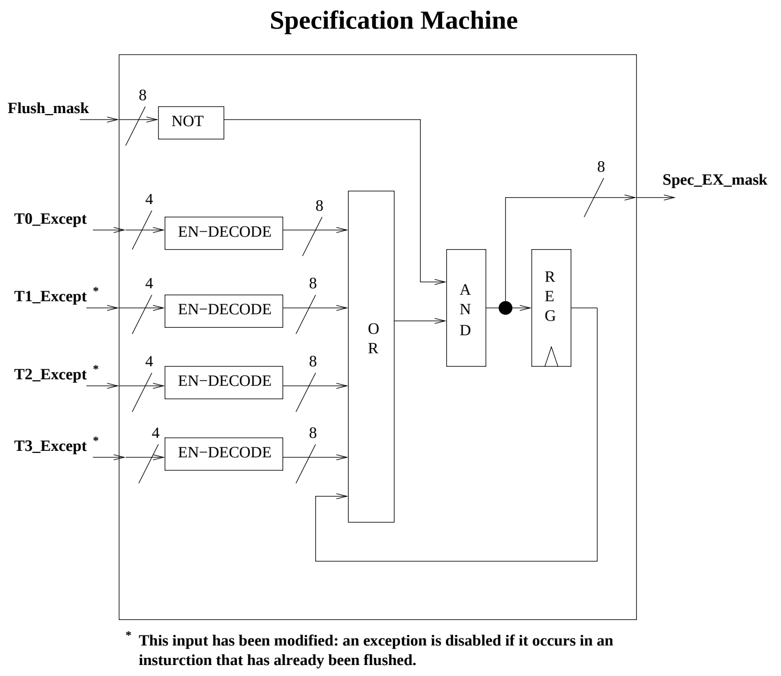

Figure 5: A simplified machine that produces the exception mask.

```
    *t0*
    (Tth-internal-state tao input-list)
    (nth tao input-list))
   (spec-miss_ordering
    (Tth-spec-state tao input-list)
    (nth tao input-list)))))

(defthm exception-liveness
 (implies
  (and (integerp tao)
       (<= 3 tao)
       (Tth-inputs-goodp tao input-list))
  (submaskp
   8
   (bv-or
    8
    (recent-flushes 3 tao *t0* input-list)
    (spec-miss_ordering
     (Tth-spec-state (- tao 3) input-list)
     (nth (- tao 3) input-list)))
   (out-udt_miss_ordering_exceptions
    *t0*
    (Tth-internal-state tao input-list)
    (nth tao input-list)))))
```

The first property proves that, for any cycle number `tao`, assuming good inputs, the exception mask generated by tile zero is a subset of the exception mask generated by the single-tile machine. The second property proves that the exception mask generated by the single tile machine is a subset of the combination of the exception mask generated by tile zero and the last three flush masks. In effect, these properties prove that our multi-tile exception design only produces exceptions produced by the specification and eventually produces all exceptions produced by the specification.

We prove these properties by reducing them to the proof of an invariant; we prove these invariants through a mixture of theorem proving and SAT solving. The following example illustrates the type of lemma that we prove with SAT. This lemma is proven by telling ACL2 to automatically call the SAT-based proof engine once its simplification rules reach a fix point.

```
(defthm sub-of-spec-mask-t0
 (implies
  (and
   (equiv-bvp
    8
    (in-ddt_miss_ordering_exceptions *t0* ins)
    (internal-st-udt_miss_ordering *t1* internal-state))
   (equiv-bvp
    8
    (in-flush_mask *t0* ins)
    (internal-st-flush_mask *t1* internal-state))
   (sub-of-spec-mask-tile *t0* spec-st internal-state)
   (sub-of-spec-mask-tile *t1* spec-st internal-state))
  (sub-of-spec-mask-tile
   *t0*
   (update-spec-st spec-st internal-state ins)
   (update-internal-state internal-state ins))))
```

5.2 Verification of an Arrived-Store Protocol

The LSQ units also communicate to create a mask of arrived stores; these are used to generate exceptions, wake deferred loads, and detect completion. Figure6 presents an overview of the arrived-store-mask protocol. This protocol is more complex than the exception protocol, because tiles send information to both the tile above and the tile below them. Also, since the arrived store mask is 256 bits, the whole mask is never sent. Instead up to three, nine-bit store signals are sent to each neighboring tile, informing each neighbor of all the new stores it has received in the last cycle.

We used the same methodology to verify the arrived-store-mask protocol as we used to verify the exception-mask protocol. We first define a single-tile design that produces the store mask. This design is shown in Figure 7. Next, we prove the equivalence of the single-tile and multi-tile designs using the following two theorems. Note that these theorems prove an equivalence over all tiles, whereas the exception mask equivalence only dealt with tile zero.

```
(defthm arrived-safety
 (implies
  (and (integerp tao)
       (<= 0 tao)
       (Tth-inputs-goodp tao input-list))
```

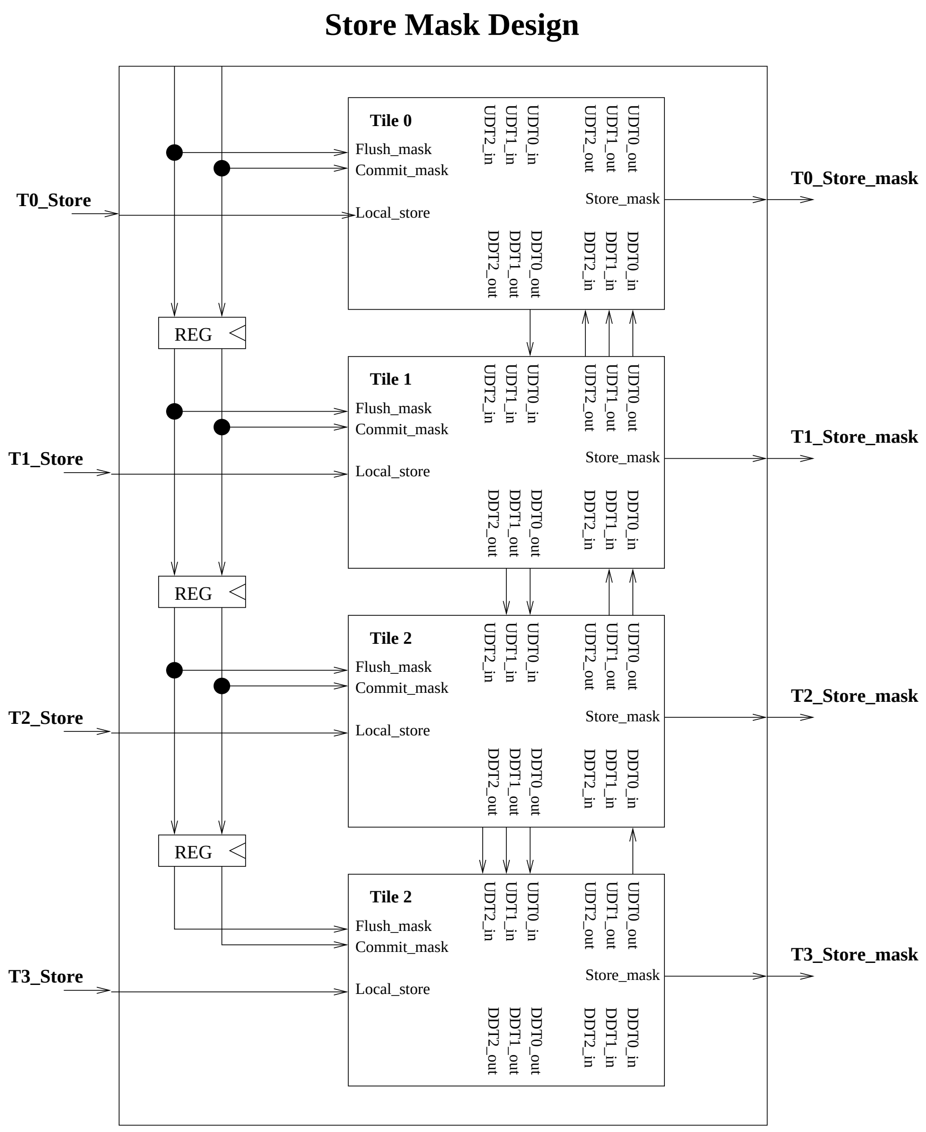

Figure 6: An overview of the protocol for generating the mask of arrived stores. Note that the tile inputs that are unconnected are either grounded or known to always be low.

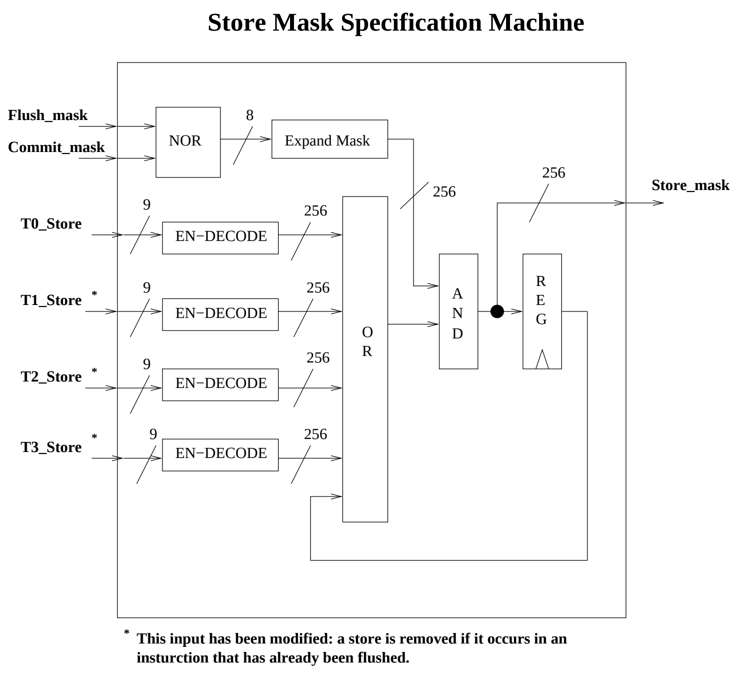

Figure 7: A simplified machine that produces the mask of arrived stores.

```
  (submaskp
   8
   (out-arrived_mask
    tile
    (Tth-internal-state tao input-list)
    (nth (- tao 3) input-list))
   (spec-arrived_mask
    (Tth-spec-state tao input-list)
    (nth tao input-list)))))

(defthm arrived-liveness
 (implies
  (and (integerp tao)
       (<= 3 tao)
       (Tth-inputs-goodp tao input-list))
  (submaskp
   8
   (bv-or
     8
     (expand-mask 8 256 (recent-flushes 3 tao tile input-list))
     (bv-or
      8
      (expand-mask 8 256 (recent-commits 3 tao tile input-list))
      (spec-arrived_mask
       (Tth-spec-state (- tao 3) input-list)
       (nth (- tao 3) input-list))))
    (out-arrived_mask
     tile
     (Tth-internal-state tao input-list)
     (nth tao input-list)))))
```

6 Conclusion

The verification of an automatically generated circuit description usually involves verifying the netlist post-synthesis. Through our ripple-carry adder example, we have shown how we can verify the correctness of the circuit generators directly, thus obviating the need to verify the resultant circuit descriptions.

To aid our verification effort, we have combined the complementary techniques of theorem proving and SAT solving. We show the usefulness of this combination through the verification of a Verilog implementation of a communication protocol used in the TRIPS processor.

An extension of our approach is to show how circuit generators can be used within the verification of the TRIPS processor. Rather than partition memory into four pieces, one could design a TRIPS processor with memory partitioned into a parametrized number of pieces. This type of verification fits well into the modular nature of the TRIPS processor design and showcases the advantages of the **DE2** language. Furthermore, this verification effort will allow us to explore the applications and limitations of fully automated verification techniques, like SAT, when used to verify large circuit generation designs.

Moving beyond circuit generators, there are many other potential applications for the **DE2** verification system. For example, we can use the **DE2** language to verify hardware optimization programs and non-functional properties. The flexibility of the **DE2** language and the ACL2 theorem proving system provides the opportunity to verify many types of applications, many of which are rarely, if ever, been formally verified.

References

[1] Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-Aware Circuit Design. In *Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *Lecture Notes in Computer Science*, pages 5–19. Springer, 2005.

[2] Robert B. Jones, John W. O'Leary, Carl-Johan H. Seger, Mark Aagaard, and Thomas F. Melham. Practical Formal Verification in Microprocessor Design. *IEEE Design & Test of Computers*, 18(4):16–25, 2001.

[3] Warren A. Hunt Jr. and Erik Reeber. Formalization of the DE2 Language. In *Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *Lecture Notes in Computer Science*, pages 20–34. Springer, 2005.

[4] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer Aided Reasoning: An Approach.* Kluwer Academic, 2000.

[5] Shuvendu K. Lahiri and Randal E. Bryant. Deductive verification of advanced out-of-order microprocessors. In *Computer Aided Verification, 15th International Conference (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*, pages 341–353. Springer, 2003.

[6] Simha Sethumadhavan, Rajagopalan Desikan, Doug Burger, Charles R. Moore, and Stephen W. Keckler. Scalable hardware memory disambiguation for high ilp processors. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO 36)*, pages 399–410. ACM/IEEE, 2003.

[7] Tera-op Reliable Intelligently adaptive Processing System, www.cs.utexas.edu/users/cart/trips.

[8] Warren A. Hunt, Jr. and Bishop C. Brock. A Formal HDL and its Use in the FM9001 Verification. In *Mechanized Reasoning and Hardware Design*, pages 35–47, Upper Saddle River, NJ, USA, 1992. Prentice-Hall, Inc.