Automatic Correctness Proof of the Implementation of Synchronous Sequential Circuits Using an Algebraic Approach

Junji Kitamichi, Sumio Morioka, Teruo Higashino and Kenichi Taniguchi

Department of Information and Computer Sciences, Osaka University
Machikaneyama 1-3, Toyonaka, Osaka 560, Japan
Tel: +81-6-850-6607 Fax: +81-6-850-6609
E-mail: {kitamiti,morioka,higashino,taniguchi}@ics.es.osaka-u.ac.jp
WWW: http://sunfish.ics.es.osaka-u.ac.jp/

Abstract. In this paper, we propose a technique for proving the correctness of the implementations of synchronous sequential circuits automatically, where the specifications of synchronous sequential circuits are described in an algebraic language ASL, which we have designed, and the specifications are described in a restricted style. For a given abstract level's specification, we refine the specification into a synchronous sequential circuit step by step in our framework, and prove the correctness of the refinement at each design step. Using our hardware design support system, to prove the correctness of a design step, we have only to give the system some invariant assertions and theorems for primitive functions. Once they are given, the system automatically generates the logical expressions from the invariant assertions and so on, whose truth guarantees the correctness of the design step, and tries to prove those truth using a decision procedure for the prenex normal form Presburger sentences bounded by only universal quantifiers. Using the system, we have proved the correctness of the implementation of a GCD circuit, the Tamarack microprocessor, a sorting circuit and so on, in a few days. The system has determined the truth of each logical expression within a minute.

1. Introduction

Recently, many hardware description languages such as VHDL [8], Verilog HDL [17] and SFL [13] have been proposed. In order to develop reliable circuits, the specifications must be described formally, and the semantics of the specification languages and the correctness of the refinements should also be defined formally. Formal description techniques (FDT), which have those features, are studied widely [1–3, 5, 9, 14]. In the fields related with FSM, for example, the verification using Larch Prover for proving the correctness of the design of pipelined CPU's has been described in [15]. Some properties of sequential circuits have been verified in [16]. We describe the functions of a sequential circuit as a requirement specification. For example, as the requirement specification of a GCD circuit, we describe that the output of circuit must be the greatest common divisor of two inputs. Here we want to prove that synchronous sequential circuits satisfy their functional requirements, and we don't treat the verification for timing, temporal properties such as liveness property and so on. To specify and verify such properties, the higher order logic approaches or the temporal logic approaches are suitable.

In this paper, we propose a technique for proving the correctness of the implementations of synchronous sequential circuits automatically, where the specifications of synchronous sequential circuits are described in an algebraic language ASL, which we have designed [10] [1]. We have developed a hardware design support system using ASL. For a given abstract level's specification, we refine the specification into a synchronous sequential circuit step by step in our framework, and prove the correctness of the refinement at each design step using our hardware design support system automatically.

In our approach, the specifications and implementations of synchronous sequential circuits are described as sequential machine style specifications that correspond to FSMs with registers. The control flows of the circuits may depend on not only the current control state but also the current values of the registers.

We introduce a type *state* representing the abstract state of the system. Each transition corresponds to the action for changing the values of the registers and is treated as the *state transition function* which returns the next abstract state from the current abstract state. The content of each register is expressed by the *state component function* which returns the value of the register at the current abstract state. In the abstract levels' specifications, a complicated action may be specified as a transition.

At each level (denoted as level k), the relations between the current registers' values and those values after each transition is executed are described as the axioms (D_k). The order and execution conditions of the transitions are also described as the axioms (C_k) [2].

At the next level (denoted as level $k+1$), such a complicated action is refined as the execution of a sequence of some more concrete actions and its repetitions. The correspondence from the functions (state components and state transitions) in level $k+1$ to the functions in level k is also described as the axioms. The correctness of the refinement is proved by showing that each axiom in level k holds as the theorem on level $k+1$ description $(\langle D_{k+1}, C_{k+1}\rangle)$, the correspondence (M_k) and the theorems for primitive functions/predicates (PRM). We repeat those refinements until we can get a synchronous sequential circuit [3].

In this paper, we adopt a restriction for describing each level's specification. The restriction is as follows: we only use a variable s of sort *state* in the axioms for describing the relations between the current registers' values and those values

[1] The name "ASL" is also used in [18]. Of course, these two languages are different.

[2] On the concrete level's specification, the definitions of the registers (including the state register in the controller) and the combinatorial circuits are described. The connection between these components are also described.

[3] After some repetitions of the refinements, the redundant transitions may be generated in the concrete level's circuit. However, such redundant transitions are deleted using our system [11].

after the transition is executed. We don't use other variables. For example, let $MEM(T(s))$ denote the contents of a memory after the transition T is executed, and let $I1(s)$ and $I2(s)$ denote the pointers for the memory before the transition T is executed. If we want to describe the property that the contents of the memory $MEM(T(s))$ are arranged in ascending order from position $I1(s)$ to position $I2(s)$, then we describe the axiom as follows.

$$\begin{aligned}&0 \leq I1(s) \leq I2(s) \leq N \\ &\qquad \textit{imply } \textit{ordered}(MEM(T(s)), I1(T(s)), I2(T(s))) == TRUE\end{aligned}$$

where $\textit{ordered}(a, i, j)$ is a primitive predicate which represents that the contents of memory a are arranged in ascending order from position i to position j. We don't use, for example, the following description style because it includes a variable i other than $\mathtt{s}$.

$$\begin{aligned}&0 \leq I1(s) \leq i < I2(s) \leq N \\ &\qquad \textit{imply } MEM(T(s))[i] \leq MEM(T(s))[i+1] == TRUE\end{aligned}$$

Under the restriction of description style, the verification of a refinement can be done as follows.

(1) If a transition of level k is refined using some repeated executions of the transitions at level $k+1$, we use a *Nötherian induction* on transitions to prove the correctness of the implementation. In the induction, first, we assign the *invariant assertions* for some intermediate states. We don't use any variable except the variable $\mathtt{s}$ of sort *state* for describing the assertions.

(2) At the each step of the induction, we construct a logical expression P representing that the assertion (or the property to be proved) holds after a transition t of the lower level $k+1$ is executed if the assertion holds before the transition is executed. The constant $\mathtt{S}$ is substituted for the variable $\mathtt{s}$ in the assertions and so on, therefore P doesn't contain any variables.

(3) Consider the proof of the expression P. First, we make a precondition R corresponding to both the axioms AX representing the content of the transition t and the theorems for primitive functions/predicates PRM. The following is an example of theorems for the primitive predicate *ordered*:

$$\begin{aligned}&0 \leq i < j \leq N \wedge a[i] \leq a[i+1] \wedge \textit{ordered}(a, i+1, j) \\ &\qquad \textit{imply } \textit{ordered}(a, i, j) == TRUE\end{aligned}$$

The constant $\mathtt{S}$ is substituted for the variable $\mathtt{s}$ in the expressions corresponding to AX. The terms such as $f(S)$ or $f(t(S))$ representing the values of the state components at state $\mathtt{S}$ or state $\mathtt{t(S)}$ are substituted for the variables in the expressions corresponding to PRM. Therefore, the expression R doesn't contain variables. Then, we construct the expression R *imply* P.

(4) Assume that R *imply* P consists of Boolean operators, integer operators and some terms whose sorts are integers or Booleans. If R *imply* P is true regardless of the values of those terms, that is, if R *imply* P is true for any integer values of those integer terms and any Boolean values of those Boolean terms, then we conclude that P is true.

(5) If the operators among those terms are restricted to "$\wedge$, $\vee$, $\neg$, $+$, $-$, $=$, $>$", then the condition that the expression R *imply* P is true regardless

of the values of the terms can be expressed in a prenex normal form *Presburger sentence* bounded by only universal quantifiers. (The form is like $\forall v_1 \forall v_2 \cdots \forall v_n \; EXP(v_1, v_2, \cdots, v_n)$.) It is decidable whether the Presburger sentence is true [7] [4]. For example, Presburger sentence which corresponds to an expression

$$(r1(S) + r2(S) = r1(S) \times r2(S) \vee pred(r1(S)) \wedge \cdots) \; imply \; (r1(S) > 0 \vee \cdots)$$

is as follows.

$$\forall v_1 \forall v_2 \forall v_3 \forall v_4 \cdots (\; (v_1 + v_2 = v_3 \vee v_4 \wedge \cdots) \; imply \; (v_1 > 0 \vee \cdots))$$

The variables v_1, v_2, v_3 and v_4 correspond to the terms $r1(S)$, $r2(S)$, $r1(S) \times r2(S)$ and $pred(r1(S))$, respectively. Here, v_1, v_2 and v_3 are integer variables, and v_4 is a Boolean variable.

We have developed a verification support system (verifier) where the above verification method is used. The verifier has a routine to decide efficiently whether a given prenex normal form Presburger sentence bounded by only universal quantifiers is true [5].

Under this verification method, we have proved the correctness of the implementations for the GCD circuit given as the TPCD94 benchmark, the Tamarack microprocessor used as a verification example in [9] and a maxsort circuit we have designed. The design and verification of the GCD circuit have been carried out within two days which contain the time used for trial and error in the verification.

As we mentioned above, when we prove the correctness of a refinement using this verifier, one have only to give the system some invariant assertions, theorems for the primitive functions and the substitutions for the variables of the theorems. Once they are given, the verifier automatically tries to prove, and one need not send the system many commands interactively.

The paper is structured as follows. The description style of synchronous sequential circuits is explained in Section **2**. In Section **3**, we describe a stepwise refinement. In Section **4**, we define the correctness of the refinement in our framework formally and explain basic techniques for verifying the correctness of the refinements. In Sections **2**, **3** and **4**, we use the GCD circuit as an example for explanation. The experimental results of the verification of the GCD circuit are given in Section **4**, and those of the CPU and maxsort circuit are also given in Section **5**. In Section **6**, we give the concluding remarks.

2. Algebraic Language ASL and Descriptions of Synchronous Sequential Circuits

In general, the synchronous sequential circuits can be modeled as the finite state machines with registers. The specification S of a synchronous sequential

[4] Usually the Presburger sentence doesn't contain any Boolean variable. However, the truth of the sentences which contain some Boolean variables is decidable. In this paper, we also call such sentences "Presburger sentences".

[5] The routine can decide the truth of the sentences which have some Boolean variables. We have implemented the routine by integrating the tautology checking algorithm for the propositional logic into the Cooper's algorithm given in [4].

circuit consists of a pair $\langle D, C\rangle$ of a description D of the contents of transitions and a state diagram C. In D, the relations between the current registers' (memories') values and those values after the transition is executed are described. In C, the condition to execute each transition at each finite state is described, and the next finite state after the transition is executed is also specified.

Here, we describe the specifications of synchronous sequential circuits in our algebraic language ASL [10]. A specification in ASL is a tuple $t = (G, AX)$, where G is a context free grammar without a starting symbol, and AX is a set of axioms. In ASL, we assume that the (infinite) axioms for primitive functions/predicates are given as the definition tables which represent the values of functions/predicates for all input values. An ASL text (specification) can include such definition tables.

Here, we explain a requirement description (level 1's description) of the GCD calculator shown in Table 1. (The descriptions of the grammar are omitted. They are also omitted in the other tables of this paper.) We introduce `R1`, `R2` and `R4` as the registers in the level 1's circuit. These registers correspond to the registers Max, Min and $X2$ given in Fig. 14-1 of the TPCD94 benchmarks, respectively. Then we introduce the abstract transitions `calcgcd` and `nop`. For example, by executing the transition `calcgcd`, the value of the GCD of `R1` and `R2` is calculated and transferred to `R4`. We use the following primitive functions.

- $MaxMember(si)$: A function which represents the maximum value in a set si of integers. This function is defined only when si isn't empty.
- $DivisorSet(i)$: A function which represents the set of divisors of an integer i. This function is defined only when i isn't zero.
- $Intersection(si, sj)$: A function which represents the set of intersection of two sets si and sj of integers.

Table 1. Description of GCD level 1 (D_1 and C_1)

```
init1: R1(init(A,B)) == Max(A,B);
init2: R2(init(A,B)) == Min(A,B);
gcd1:
(1 <= R2(s) and R2(s) <= R1(s) and R1(s) <= N) imply
 R4(calcgcd(s))
  = MaxMember(Intersection(DivisorSet(R2(s)),DivisorSet(R1(s))))== TRUE;
nop1:   R4(nop(s))      == R4(s);
valid1:   VALID(init(A,B))    == TRUE;
valid2:   VALID(calcgcd(s))   == VALID(s) and CONTROL(s)=INIT;
valid3:   VALID(nop(s))       == VALID(s) and CONTROL(s)=END;
control1: CONTROL(init(A,B))  == INIT;
control2: CONTROL(calcgcd(s)) == if CONTROL(s) = INIT then END;
control3: CONTROL(nop(s))     == if CONTROL(s) = END  then END;
```

Using the primitives above, we describe the contents of transitions (D_1). For example, the axiom `gcd1` means that `R4(calcgcd(s))` should be the maximum number of the intersection of the divisor sets of `R1(s)` and `R2(s)` under the assumption `R1(s)` is greater than or equal to `R2(s)` [6]. The values of

[6] In this example, we use a formal parameter `N` as the maximum value of the registers. The results of our verification are valid for any positive integer value of `N`.

`R1(calcgcd(s))` and `R2(calcgcd(s))` aren't specified and hence any values of these terms are permitted. The axioms don't have any variable other than `s`.

We also describe the state diagram (C_1) as the axioms using both the predicate `VALID` and the function `CONTROL`. The predicate `VALID` represents the execution condition of each transition. The function `CONTROL` represents the state name. (At the concrete level, the function represents the value of the state register.) For example, the axiom `valid2` means that the execution condition of `calcgcd` is `CONTROL(s) = INIT`[7]. The axiom `control2` means that the value of controller after the execution of `calcgcd` at state `INIT` is `END`.

3. Stepwise Refinements

In this section we explain how to refine a specification.

Let $S_k = \langle D_k, C_k \rangle$ denote a level k's specification. In our framework, the specification $S_{k+1} = \langle D_{k+1}, C_{k+1} \rangle$ is obtained as follows.

(1) We introduce the primitive functions at level $k+1$.

(2) We introduce some state component functions at level $k+1$.

(3) We introduce some state transition functions at level $k+1$ and describe D_{k+1} using the primitive/transition functions. (In the concrete level (circuit level), we describe both the definition of the components in the circuit and the connections between the circuit's components as D_{k+1}. See Section **3.3**.)

(4) We give the correspondence M_k (mapping functions) as follows.

 (4-1) We give the correspondence from the state components functions at level $k+1$ to those at level k.

 (4-2) We give the correspondence from the state transitions functions at level $k+1$ to those at level k. Each transition at level k is implemented as the execution of a sequence of some transitions at level $k+1$ and its repetitions [8].

 We don't give the mapping explicitly if a function F in the level k corresponds to the same function F in the level $k+1$.

(5) The system can automatically synthesize the level $k+1$'s state diagram C_{k+1} from both the level k's state diagram C_k and the correspondence M_k. (In the concrete level, we describe the circuit's controller as C_{k+1}. See Section **3.3**.)

(6) For the derived level $k+1$'s specification S_{k+1}, we prove that S_{k+1} is a correct implementation of S_k under the correspondence M_k. The proof method is summarized in Section **4**.

(7) Some redundant state transitions may be included in S_{k+1}, even if S_{k+1} is a correct implementation of S_k. For example, we may be able to merge two consecutive transitions into one transition. Such an optimization is carried out using our hardware design support system.

[7] In general, the execution condition of a transition is written as a sum of products (current state and selection condition).

[8] The mapping is written by mutual recursive expressions with only tail recursions, so that the controller can be implemented by a finite state control.

The process described above is continued until a concrete circuit is derived. In the concrete level, the state component functions (including **CONTROL**) correspond to the hardware components such as the registers, memories, flip-flops and so on. The state transition functions correspond to the data calculations and transfers among those hardware components caused by one machine clock. The details are explained in Section **3.3**.

3.1 Example of refinement to level 2 of GCD circuit

In this section, we explain how we have refined the level 1's specification.

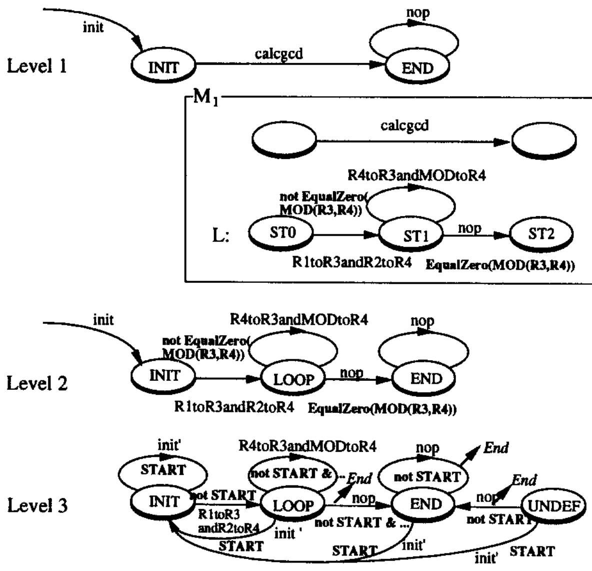


Fig. 1 State diagram of each level of GCD circuit

At first, we have decided to use the Euclid's algorithm to calculate the value of GCD. Then, we have introduced the following primitive function and predicate in the level 2.

- $MOD(i, j)$: A function which represents the value of $i\ mod\ j$ where i and j are both integers. This function is defined only when the integer j isn't zero.
- $EqualZero(i)$: A predicate whose value is *true* if and only if an integer i is equal to zero.

We have introduced **R3** as a new register in the level 2. This register corresponds to the register $X1$ in Fig.14-1 of the TPCD94 benchmarks. The circuit also has three registers **R1**, **R2** and **R4** which are the same as the those in the level 1.

We have introduced the following transitions (Table 2). We call the description of the contents of these transitions as D_2.

- **R1toR3andR2toR4** : A transition to execute parallel data transfers; the transfer from **R1** to **R3**, and that from **R2** to **R4**.
- **R4toR3andMODtoR4** : A transition to execute parallel data transfers and calculation; the transfer from **R4** to **R3**, the calculation of the value of **MOD(R3,R4)** and its transfer to **R4**.

We have described both the order of the execution of transitions and the execution conditions for their executions (Table 3). This is also a description of the correspondence from the transitions in the level 2 to the transition `calcgcd` in the level 1. We call this description M_1. All of the axioms in M_1 and D_2 don't have other variables than `s`.

Then the system automatically derived the state diagram C_2 (Fig. 1) from C_1 and M_1. Here, let L be the state diagram for `calcgcd`. The L has three states $ST0$, $ST1$ and $ST2$. $ST0$ is the initial state of L and $ST2$ is the end state of L. The states $INIT$, $LOOP$ and END in Fig. 1 correspond to $ST0$, $ST1$ and $ST2$, respectively.

Table 2. Description of the contents of the transitions in GCD level 2 (D_2)

```
R3(R1toR3andR2toR4(s))  == R1(s);
R4(R1toR3andR2toR4(s))  == R2(s);
R3(R4toR3andMODtoR4(s)) == R4(s);
R4(R4toR3andMODtoR4(s)) == MOD(R3(s),R4(s));
```

Table 3. Description of the correspondence (M_1)

```
calcgcd(s) ==   S1(R1toR3andR2toR4(s));
S1(s)      ==   if    (EqualZero(MOD(R3(s),R4(s))))
                then   nop(s)
                else   S1(R4toR3andMODtoR4(s));
```

3.2 Refinement to level 3 of GCD circuit

We have designed level 3's specification as follows.

(a) We have introduced an additional new state $UNDEF$ so that the total number of states becomes four, since we assume that the controller is implemented by using two D Flip Flops.

(b) We introduced the input signal $START$. Whenever $START$ signal becomes *true*, the transition *init'* will be executed and the circuit enters $INIT$ [9].

(c) We added the output signal *End* as follows.

$$End(s, START, A, B) ==$$
$$(\neg START \wedge CONTROL(s) = LOOP \wedge EqualZero(MOD(R3(s), R4(s))))$$
$$\vee (\neg START \wedge CONTROL(s) = END)$$
$$\vee (\neg START \wedge CONTROL(s) = UNDEF);$$

The state diagram of the level 3's specification is shown in Fig. 1.

3.3 Concrete implementation of the GCD circuit

The concrete implementation (level 4's specification) is the same as the implementation of the TPCD94 benchmark [10].

We have defined the circuit's components such as registers (including the state register of the controller) and combinatorial logic circuits (Table 4). For

[9] Since the GCD circuit in [12] uses this $START$ signal, we have also introduced it at the level 3. In our circuit, the circuit enters state END and outputs *End* signal, if $START$ signal isn't given at state $UNDEF$. These are based on [12].

[10] We have found some errors in the Tables 15-2,15-3 and 15-4 of TPCD94 benchmarks v1.0.0 during the verification using our verifier, and we have corrected the errors.

example, the axiom **reg** is the definition of the state component function **REG** (register). The state transition **CK_r** of the register corresponds to the transition caused by a clock signal.

Table 4. Definition of logical components of GCD circuit

```
reg: REG(CK_r(r_s,ctl,data)) == if ctl then data else REG(r_s);
q:   Q(CK_q(d_s,d)) == d;
mul: Mul(n,m,u)     == if u then n else m;
                :
```

The circuit has four registers and two D-FFs. (These D-FFs compose a state register of the controller.)

The state of the total circuit is described as the tuple of the state of each component. The state of the i-th component can be refered by the primitive function proj_i.

The data path between the registers is shown in Fig. 2. In the Table 5, the content of the transition **CK** under the data path shown in Fig. 2 are described. The transition **CK** is caused by a machine clock. The content of **CK** are determined by the current values of registers, the values of the control signals for the circuit's each component and the values of the circuit's input signals. The description D_4 of the content of transition in level 4 consists of the descriptions shown in Table 4 and Table 5.

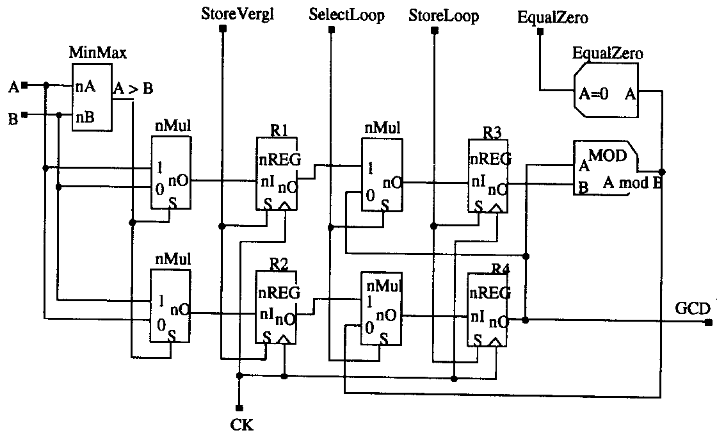


Fig. 2 Data path of GCD circuit

Table 5. Description of data path of GCD circuit

```
define 'outMod' := 'Mod(REG(proj_3(s)),REG(proj_4(s)))';
CK(s,A,B,START,StoreVergl,StoreLoop,SelectLoop,Q1,Q2)
== [ CK_r(proj_1(s),StoreVergl,Mul(A,B,GT(A,B)))
     CK_r(proj_2(s),StoreVergl,Mul(B,A,GT(A,B)))
     CK_r(proj_3(s),StoreLoop,
                        Mul(REG(proj_4(s)),REG(proj_1(s)),SelectLoop))
     CK_r(proj_4(s),StoreLoop, Mul(outMod,REG(proj_2(s)),SelectLoop))
   (* -- note -- the following line is controller *)
     CK_q(proj_5(s),Q1) CK_q(proj_6(s),Q2) ];
```

Then we gave the correspondence M_3 (Table 7). The correspondence consists of the followings: (1) correspondence from the state components in level 4 to those in level 3 (including state assignment), and (2) correspondence from state transition function in the level 4 to those in level 3.

The controller is shown in Fig. 3. The connections of the logical gates in the controller is described in Tbl 6. The description corresponds to C_4.

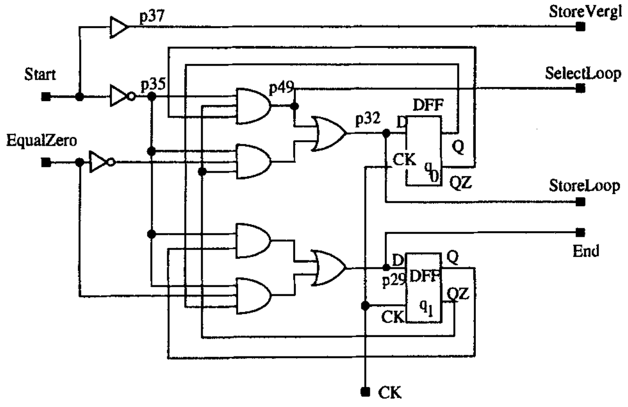


Fig. 3 Implementation of controller of GCD circuit

Table 6. Description of controller of GCD circuit

```
VALID(CK(s,A,B,START,StoreVergl,StoreLoop,SelectLoop,Q1,Q2))
== VALID(s) and StoreVergl = BUF(START)
            and StoreLoop = p32
                    :
            and Q1 = p32
            and Q2 = p29 ;
define 'p37' := 'BUF(START)'
define 'p32' := 'OR(p49,AND3(p35,NOT(eq0),QZ(proj_5(s))))';
define 'p29' := 'OR(AND(p35,Q(proj_5(s))),AND3(p35,eq0,Q(proj_6(s))))';
define 'p35' := 'NOT(START)';
```

Table 7. Correspondence from level 4 to level 3

```
R1(s) == REG(proj_1(s));
R2(s) == REG(proj_2(s));
     :
CONTROL(s) ==[ Q(proj_5(s)) , Q(proj_6(s))];
INIT   == [ FALSE , FALSE ];
LOOP   == [ FALSE , TRUE ];
     :
(* Note: "--" means " don't care" *)
      (*  A, B,START,StoreVergl,StoreLoop,SelectLoop,  Q1,  Q2 *)
R1toR3andR2toR4(s,START)
  == CK(s,--,--,START,     FALSE,      TRUE,       TRUE,FALSE,TRUE);
R4toR3andMODtoR4(s,START)
  == CK(s,--,--,START,        --,      TRUE,      FALSE,FALSE,TRUE);
nop(s,START)          == .....;
init'(s,A,B,START) == .....;
```

4. Correctness Proof of Implementations

In this section, we define the correctness of the refinements formally, and explain how to prove the correctness algebraically.

4.1 Verification techniques

Let t' denote a specification which contains all sorts and functions in a specification t. We say that t' is a correct refinement of t if and only if $\alpha \equiv_t \beta$ implies $\alpha \equiv_{t'} \beta$ where $\equiv_\tau$ denotes the congruence relation defined by specification τ. Let $\sigma(\xi)$ denote a substitution of ground terms for the variables in the term ξ. If $\sigma(\alpha)$ and $\sigma(\beta)$ are congruent for any substitution σ, then we describe as $\alpha \approx_t \beta$, and "$\alpha \approx_t \beta$" is called a *theorem*. If $l \approx_{t'} r$ holds for any axiom $l == r$ in t, then t' is a correct refinement of t [6,10]. If we can prove that all of the axioms in S_k hold as theorems on S_{k+1}, M_k and theorems for primitive functions/predicates (we call the set of these theorems PRM), then we conclude that S_{k+1} is a correct implementation of S_k.

Under our frame work, the use of the axioms in C_{k+1} is unnecessary for the proof of D_k.

The proof of C_k is unnecessary if C_{k+1} is derived from C_k and M_k automatically. So if the proof of D_k succeed, then S_{k+1} is a correct implementation of S_k. If the designer gives C_{k+1} directly, then the proof of C_k is necessary.

The followings are used to prove a theorem [6].

(A) term reduction by regarding each axiom as a rewrite rule
(B) case analysis of conditional branches
(C) the decision procedure for Presburger sentences :

Let $\xi(x_1, \cdots, x_n)$ denote a term consisting of only integers, integer (or Boolean) variables $x_1, \cdots, x_n$ and operators "$\wedge$, $\vee$, $\neg$, $+$, $-$, $=$, $>$". An algorithm to determine the truth of the following formula is given in [4].

(C-1) $\forall x_1, \cdots, x_n[\xi(x_1, \cdots, x_n)]$

If the formula (C-1) is *true* and all of the values (constructor terms) of terms $t_1, \cdots, t_n$ are defined in t, then we can conclude that $\xi(t_1, \cdots, t_n) \approx_t$ *true*.
(D) the use of the theorems for primitive functions/predicates
(E) Nötherian induction

4.2 Verification of refinement from level 1 to level 2

Now, we'll explain how to prove that the axiom **gcd1** hold as a theorem on D_2, M_1 and PRM. We should prove the followings.

- **Partial correctness** : After the execution of the state diagram L, the value of the register **R4** should be equal to the value of the GCD of **R1** and **R2** before its execution.
- **Termination** : The execution of the state diagram L should terminate.

4.2.1 Proof of partial correctness

In order to prove the partial correctness, we use the *Nötherian induction.* In the induction, first, we assign the invariant assertion IA_{ST1} (Table 8) to the intermediate state $ST1$ in L. Let *Pre* and *Post* (Table 8) be the precondition and the postcondition from the axiom **gcd1**, respectively. Then we prove that the following conditions hold under the precondition *Pre*.

- **first**: IA_{ST1} should hold immediately after reaching the state $ST1$ from the state $ST0$ by executing the transition `R1toR3andR2toR4`.
- **induc**: IA_{ST1} should hold immediately after reaching the state $ST1$ from the same state $ST1$ by executing the transition `R4toR3andMODtoR4`.
- **last**: *Post* should hold immediately after reaching the state $ST2$ from the same state $ST1$ by executing the transition **nop**.

Table 8. Assertions used for proof

```
< Precondition of gcd1: Pre(S0) >
  1 <= R1(S0) <= R2(S0) <= N
< Invariant assertion for ST1: IAST1(s,S0) >
  1 <= R3(s) <= R1(S0) and
  1 <= R4(s) <= R2(S0) and
  R4(s) <= R3(s) and
  SameSet(Intersection(DivisorSet(R1(S0)),DivisorSet(R2(S0))),
          Intersection(DivisorSet(R3(s)),DivisorSet(R4(s))))
< Postcondition of gcd1: Post(s,S0) >
  R4(s) = MaxMember(Intersection(DivisorSet(R1(S0)),
                                 DivisorSet(R2(S0))))
```

Using our verifier, the proof work is carried out as follows.

Step 1: The system displays a figure of L graphically on the X Window (see Fig. 4) automatically by analyzing the description M_1. This figure provides a facility of the user interface where the user can point the state to assign the invariant assertions by clicking the state in the figure directly.

Step 2: The system automatically generates $Pre(S0)$ and $Post(s, S0)$, which correspond to the precondition part and the postcondition (conclusion) part of the axiom **gcd1**, respectively.

Step 3: We assigned the invariant assertion $IA_{ST1}(s, S0)$ to the state $ST1$. (The proof may succeed even if another assertion is given.) This assertion expresses that the set of the common divisors of **R3** and **R4** at $ST1$ should be equal to that of the common divisors of **R1** and **R2** at $ST0$. The relation between the values of **R3** and **R4** which should hold at $ST1$ is also expressed in IA_{ST1}.

We don't use other variables except **s** in the invariant assertions.

Step 4: The system automatically generates three expressions (conditions) $P_{\mathtt{first}}$, $P_{\mathtt{induc}}$ and $P_{\mathtt{last}}$ corresponding to **first**, **induc** and **last**, respectively. The general form for these expressions is as follows. (Here, we consider a path PT from the state ST_i to ST_j through a transition T where $cond_{PT}$ denotes a condition for executing PT, IA_i denotes an assertion assigned to ST_i and IA_j denotes an assertion assigned to ST_j.)

$$Pre(S0)\ \textit{imply}$$
$$(\ IA_i(S, S0)\ \wedge\ cond_{PT}(S)\ \text{imply}\ IA_j(T(S), S0)\ )$$

For example, the expression $P_{\mathtt{induc}}$ is as follows.

$$Pre(S0)\ \textit{imply}$$
$$(IA_{ST1}(S, S0) \wedge \neg\ EqualZero(MOD(R3(S), R4(S))))$$
$$\text{imply}\ IA_{ST1}(R4toR3andMODtoR4(S), S0))$$

We should prove that the conditions $P_{\mathtt{first}}$, $P_{\mathtt{induc}}$ and $P_{\mathtt{last}}$ hold under D_2, M_1 and PRM. Now, we'll explain how to prove one of these conditions, using $P_{\mathtt{induc}}$ as an example.

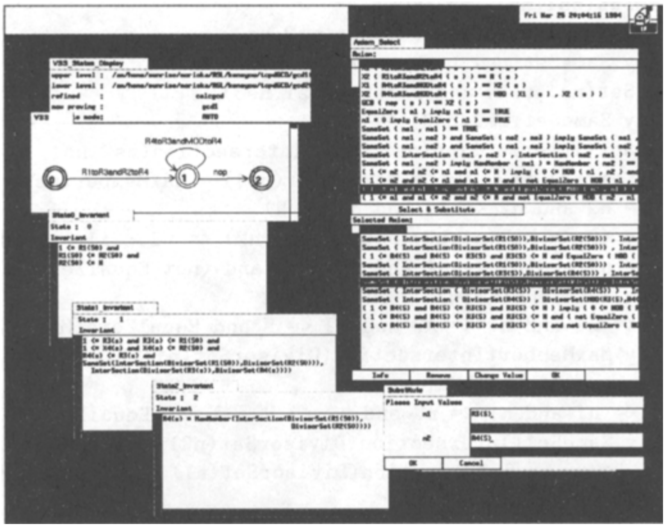


Fig. 4 Display of our verifier

Step 5: The system automatically rewrites each expression by treating the axioms of D_2 as the rewrite rules. Let $P'_{\mathtt{induc}}$ be an expression obtained from $P_{\mathtt{induc}}$ by the term rewriting.

Step 6: The system automatically selects from D_2 the axioms which describe the content of the transition `R4toR3andMODtoR4` but were not used in the term rewriting at Step 5. Then the system substitutes the constant `S` for the variable `s` in these axioms, replace "==" in the axioms with "=" and make their logical products. Let AX be an expression obtained in this way.

Step 7: We write some statements for primitive functions/predicates which are thought to be included in PRM. We call the set of the statements TH (see Table 9). Each statement is written in the same form as the axiom.

We substituted the values of the registers such as `R3(S)` and `R4(R4toR3andModtoR4(S))` for the variables of each statement in TH [11]. Then the system replace "==" in the axioms of TH with "=" and make their logical products. Let TH' be an expression obtained in this way.

Step 8: The system automatically constructs the logical expression $TH' \wedge AX$ *imply* $P'_{\mathtt{induc}}$. Let $Q_{\mathtt{induc}}$ be this expression.

Step 9: The system automatically determines the truth of each logical expression (now $Q_{\mathtt{induc}}$) as follows.

At first, the system gets, from $Q_{\mathtt{induc}}$, an expression $Q'_{\mathtt{induc}}$ consisting of "$\wedge$, $\vee$, $\neg$, $+$, $-$, $=$, $>$" and integer (Boolean) variables which satisfies the following conditions.

[11] The system can find the candidates of the substitution, that is, most general unifiers of the logical expression AX *imply* $P'_{\mathtt{induc}}$ and the statement in TH.

Table 9. Statements for primitive functions/predicates (TH)

```
prim1: EqualZero(n1) imply n1 = 0 == TRUE;
prim2: n1 = 0 imply EqualZero(n1) == TRUE;
prim3: SameSet(ns1,ns1) == TRUE;
prim4: SameSet(ns1,ns2) and SameSet(ns2,ns3)
       imply SameSet(ns1,ns3) == TRUE;
prim5: SameSet(ns1,ns2) and SameSet(ns1,ns3)
       imply SameSet(ns2,ns3) == TRUE;
prim6: SameSet(Intersection(ns1,ns2),Intersection(ns2,ns1))== TRUE;
prim7: SameSet(ns1,ns2) imply MaxMember(ns1) = MaxMember(ns2) == TRUE;
prim8: (1 <= n2 and n2 <= n1 and n1 <= N)
       imply (0 <= MOD(n1,n2) and MOD(n1,n2) <= n2 - 1) == TRUE;
prim9: (1 <= n2 and n2 <= n1 and n1 <= N and (not EqualZero(MOD(n1,n2))))
       imply 1 <= MOD(n1,n2) == TRUE;
prim10: (1 <= n1 and n1 <= n2 and n2 <= N and EqualZero(MOD(n2,n1)))
       imply MaxMember(Intersection(DivisorSet(n1),DivisorSet(n2)))
             = n1 == TRUE;
prim11: (1 <= n1 and n1 <= n2 and n2 <= N and not EqualZero(MOD(n2,n1)))
       imply SameSet(Intersection(DivisorSet(n2), DivisorSet(n1)),
                     Intersection(DivisorSet(n1), DivisorSet(MOD(n2,n1))))
       == TRUE;
```

- $Q'_{\mathtt{induc}}$ must be obtained by replacing the integer (Boolean) sub-terms in $Q_{\mathtt{induc}}$ with the integer (Boolean) variables. The same terms must be replaced with the same variable, and the different terms must be replaced with the different variables.
- The outermost operator (or function symbol) of the subterm replaced must be other than "$\wedge$, $\vee$, $\neg$, $+$, $-$, $=$, $>$".

The system gets, from $Q'_{\mathtt{induc}}$, a prenex normal form Presburger sentence $Q''_{\mathtt{induc}}$ by bounding all of the variables by the universal quantifiers.
Then the system applies the decision procedure for the prenex normal form Presburger sentences bounded by only universal quantifiers to $Q''_{\mathtt{induc}}$. If the result of the decision is "*true*", then $P_{\mathtt{induc}}$ holds on D_2, M_1 and TH[12]. Therefore, if the result is "*true*" then we conclude that $P_{\mathtt{induc}}$ holds on D_2, M_1 and PRM under the assumption TH is a subset of PRM. (Statements in TH are "correct" theorems for primitive function/predicates.)
We have implemented a decision procedure for the prenex normal form Presburger sentences bounded by only universal quantifiers. The procedure uses the transformation rule called "quantifier elimination" which is used in Cooper's algorithm [4]. For the speed-up of the algorithm, we have devised a way to decide the ordering for deleting variables depending on the form of a given expression[13].

[12] It's necessary to show that the terms replaced have their own values under the assumption that the values of the state component functions at states S0 and S are determined.

[13] For example, in the syntax tree of a given expression, a variable close to the root will be deleted first.

The truth of $Q''_{\mathtt{induc}}$ has been determined about 1 second by Sun Classic (see Table 10). In the table, the column "length of Presburger sentences" describes the total number of occurrences of the variables and operators in each Presburger sentence.

Step 10: Trial and error will be needed until the proof for all of the steps of the induction succeeds. In many cases, the proof failures have been caused by the lack of the relations of the state components in the invariant assertions, the lack of the statements for the primitive functions, and so on.

The system automatically manages which steps of the induction are already proved, and which steps aren't proved yet. And the system automatically changes the attributes of the proper steps from "Already proved" to "Not yet proved" if the user modifies an invariant assertion.

Table 10. CPU time used for deciding the truth of Presburger sentences

path	length of Presburger sentence	CPU time
ST0 → ST1	186	0.45 sec
ST1 → ST1	207	1.18 sec
ST1 → ST2	186	0.18 sec

Sun Classic

4.2.2 Proof of termination

In order to prove the termination for the axiom **gcd1**, we proved the following two conditions.

- At the state $ST1$, the value of **R4** should be always positive.
- After an execution of the transition **R4toR3andMODtoR4**, the value of **R4** must be less than the value before the execution.

To show these two conditions hold, we proved the following condition hold under D_2, M_1 and PRM.

$$\begin{aligned}&Pre(S0)\ imply\\ &\quad (IA_{ST1}(S,S0) \wedge execution\ condition\ of\ R4toR3andMODtoR4\\ &\qquad\quad imply\ 1 \leq R4(S) \wedge R4(R4toR3andMODtoR4(S)) < R4(S))\end{aligned}$$

Here, the condition $IA_{ST1}(S,S0)$ can be used in the precondition part, because it has been already proved that $IA_{ST1}(s,S0)$ is the invariant at $ST1$ under D_2, M_1, PRM and the precondition of **gcd1**.

The proof was carried out using the same verification method written in Section **4.2.1**. (We applied Step 5 $\sim$ Step 9 to the condition.) The CPU time used for the verification was 1 second (Sun Classic).

4.3 Verification of refinement from level 2 to concrete implementation

The level 3's specification satisfies the level 2's specification, because S_3 is obtained by adding some axioms to S_2. But the implementation of transition *init* doesn't obey our framework.

At the proof of the correctness of the refinement from level 3 to level 4, we proved that each axiom of S_3 holds as theorem on the S_4, M_3 and PRM [14]. The proof was carried out automatically using the term rewriting facility and decision procedure for the prenex normal form Presburger sentences bounded by only universal quantifiers in our verifier. The CPU time used for the verification was 11 seconds (Sun Classic).

5. Other Examples of Verification

5.1 The Tamarack Microprocessor

We have designed the Tamarack microprocessor in [9] and proved the correctness of the implementation. In the level 1, we described the same requirement (the same instruction set as that of [9]). In the level 3, we have used the same architecture and microprograms as those given in Fig.3 and Appendix of [9].

Our verifier has proved the correctness of each refinement automatically without trial and error, using the term rewriting and the decision procedure for the prenex normal form Presburger sentences bounded only by universal quantifiers. The CPU time used for the verification from level 1 to level 2 was 93 seconds (Sun Classic), and that from level 2 to level 3 was 55 seconds (Sun Classic).

The major reasons why the proofs were carried out without trial and error are as follows.

(1) We had no need to use any invariant assertion at the proofs, because there are no loops in the state diagram.

(2) We had no need to use the theorems for primitive functions/predicates at the proofs.

For such a case, the verification could also be carried out automatically by the symbolic simulation such as [9]. However, our verifier can treat even the case that the relations between the values of the current state component functions and those values at the next state are described as the predicates. This is a merit of our approach.

5.2 Maxsort Circuit

We have implemented a sorting circuit and proved the correctness of the implementation.

The requirement of the sorting circuit is described in Table 11. The predicate $seteq(a,i,j,b,k,l)$ represents that the set of elements between the positions i and j of array a is equal to the set of elements between the positions k and l of array b as the multi set. The predicate $arrayeq(a,i,j,b)$ represents that the integer elements between the positions i and j of array a are the same as that of array b in order.

The axiom `sort1` represents that the elements between the positions `I1(INIT)` and `I2(INIT)` (both `I1` and `I2` are registers) of the memory `MEM(sort(INIT))` after the transition `sort` are sorted where `INIT` is the initial state.

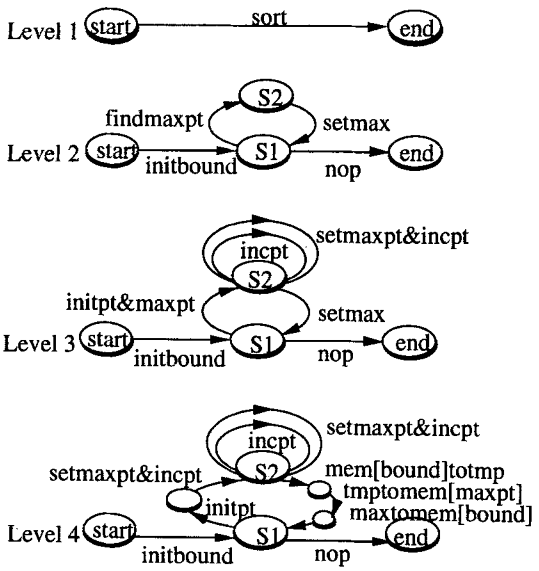
[14] At this proof we had to prove that each axiom of C_3 holds as theorem on the S_4, M_3 and PRM, since we gave C_4 (the definition of the components of the state register and the connection of the components in the circuit) directly.

The state diagram of each level is shown in Fig. 5.

We have implemented the transition **sort** using max-sort algorithm. In the level 2, the new registers **max**, **maxpt** and **bound** are introduced. The register **bound** denotes the next position of the lowest position in the sorted area.

Table 11. Requirement of sort circuit

```
sort1: 0<=I1(INIT)<=I2(INIT)<=N imply
       ordered(MEM(sort(INIT)),I1(INIT),I2(INIT))== TRUE
sort2: 0<=I1(INIT)<=I2(INIT)<=N imply
       seteq(MEM(INIT),      I1(INIT),I2(INIT),
             MEM(sort(INIT)),I1(INIT),I2(INIT))== TRUE
sort3:0<=I1(INIT)<=I2(INIT)<=N imply
      ( (0<I1(INIT)imply
             arrayeq(MEM(INIT),0,I1(INIT)-1,MEM(sort(INIT),0,I1(INIT)-1)))
        and (I2(INIT)<N imply
             arrayeq(MEM(INIT),I2(INIT)+1,N,MEM(sort(INIT),I2(INIT)+1,N)))
      ) == TRUE
```



Conditions to execute each transition are omitted

Fig. 5 State diagram of each level of maxsort circuit

The new transitions **initbound**, **findmaxpt**, **setmax** and **nop** are also introduced. By executing the transition **findmaxpt**, the maximum value in the unsorted area is stored into **max** and the position of the maximum value is stored into **maxpt**. By executing the transition **setmax**, the content at the position pointed by **maxpt** in **MEM** is replaced by the content at the position pointed by **bound**, the content at the position pointed by **bound** becomes equal to the content of **max**, and the value of **bound** is decreased by one.

In the verification of the correctness of the refinement from level 1 to level 2, we have used the Nötherian induction. To prove that the axiom **SORT1** holds, we have assigned the following invariant assertion to the state **S1**.

$$
\begin{aligned}
&I1(s) = I1(S0)\ \wedge I2(s) = I2(S0)\ \wedge \\
&\quad I1(s) \le bound(s)\ \ \le I2(s)\ \wedge \\
&\qquad (bound(s) < I2(s)\ \textit{imply} \\
&\qquad\quad ordered(MEM(s), bound(s)+1, I2(S0))\ \wedge \\
&\qquad\qquad isMaxPos(MEM(s), I1(S0), bound(s)+1, bound(s)+1))
\end{aligned}
$$

Predicate $isMaxPos(a, i, j, k)$ represents that $a[k]$ is greater than or equal to any of $a[i]$, $a[i+1]$, ..., $a[j]$.

In the level 5, we have described both the data path (Fig. 6) and the controller (Fig. 7) of the circuit.

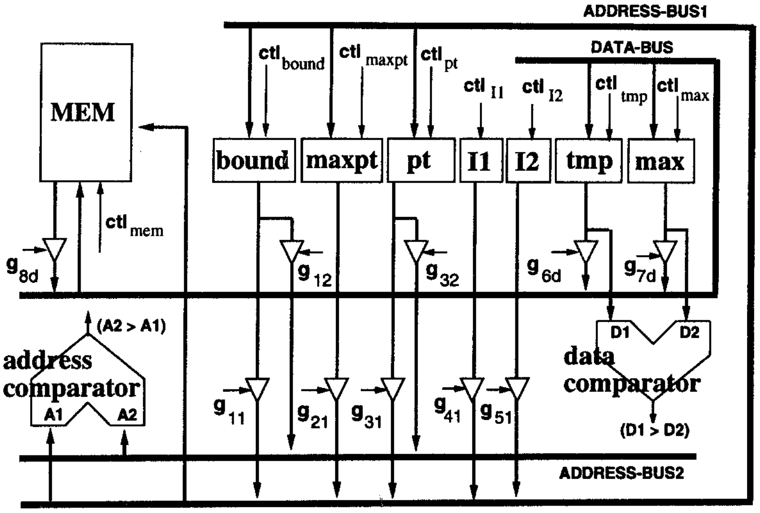
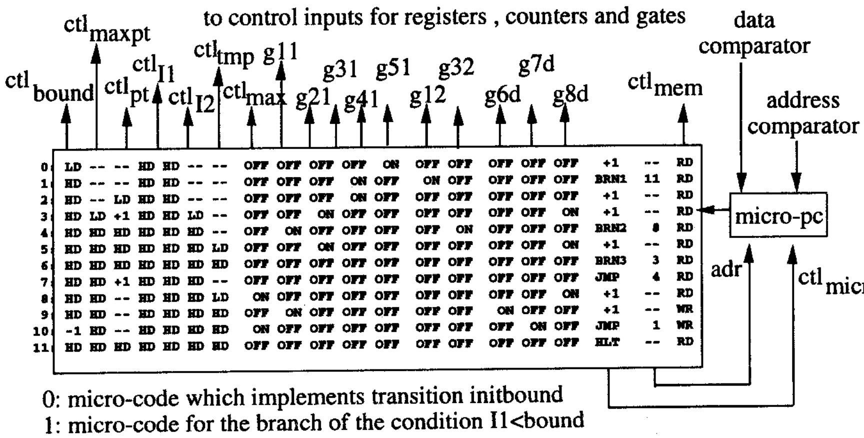


Fig. 6 Data path of maxsort circuit



	ctl bound	ctl maxpt	ctl pt	ctl I1	ctl I2	ctl tmp	ctl max	g11	g21	g31	g41	g51	g12	g32	g6d	g7d	g8d			ctl mem
0	LD	--	--	HD	HD	--	--	OFF	OFF	OFF	OFF	ON	OFF	OFF	OFF	OFF	OFF	+1	--	RD
1	HD	--	--	HD	HD	--	--	OFF	OFF	OFF	ON	OFF	ON	OFF	OFF	OFF	OFF	BRN1	11	RD
2	HD	--	LD	HD	HD	--	--	OFF	OFF	OFF	ON	OFF	OFF	OFF	OFF	OFF	OFF	+1	--	RD
3	HD	LD	+1	HD	HD	LD	--	OFF	OFF	ON	OFF	OFF	OFF	OFF	OFF	OFF	ON	+1	--	RD
4	HD	HD	HD	HD	HD	HD	--	OFF	ON	OFF	OFF	OFF	OFF	ON	OFF	OFF	OFF	BRN2	8	RD
5	HD	HD	HD	HD	HD	HD	LD	OFF	OFF	ON	OFF	OFF	OFF	OFF	OFF	OFF	ON	+1	--	RD
6	HD	HD	HD	HD	HD	HD	HD	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BRN3	3	RD
7	HD	HD	+1	HD	HD	HD	--	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	JMP	4	RD
8	HD	HD	--	HD	HD	HD	LD	ON	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	ON	+1	--	RD
9	HD	HD	--	HD	HD	HD	HD	OFF	ON	OFF	OFF	OFF	OFF	OFF	ON	OFF	OFF	+1	--	WR
10	-1	HD	--	HD	HD	HD	HD	ON	OFF	OFF	OFF	OFF	OFF	OFF	OFF	ON	OFF	JMP	1	WR
11	HD	HD	HD	HD	HD	HD	HD	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	HLT	--	RD

0: micro-code which implements transition initbound
1: micro-code for the branch of the condition I1<bound
:

Fig. 7 Implementation of controller of maxsort circuit

The experimental results for the verification are described in Table 12. In the verifications of the correctness of the refinement from the level 1 to level 2, totally 20 theorems for the primitive functions were used. Therefore the sizes of the prenex normal form Presburger sentences bounded by only universal quantifiers to be checked became rather large. (There were some sentences whose length exceeds 1000 with more than 40 different variables.) However, our verifier could decide their truth within about 1 minute. If the value of a given Presburger sentence is false, then the result '*false*' will be obtained very quickly (within a second in most case). The design and verification were carried out in three days including trial and error.

Table 12. Data of the verification of the sorting circuit

	CPU Time (Sun Classic)	Length of expression	Number of variables	Number of theorems (substitutions) of primitives
Level1 to Level2				
(proof of sort1)	64.85 sec	1093	42	16 (27)
(sort2)	28.30 sec	645	31	10 (16)
(sort3)	15.67 sec	511	29	4 (8)
(termination)	2.2 sec			
Level2 to Level3				
(partial correctness)	19.17 sec			
(termination)	1.9 sec			
Level3 to Level4	17.45 sec			
Level4 to Level5	8.80 sec			

"Length of expression" and "Number of variables" mean the length and the number of different variables of the longest Presburger sentence (that corresponds to the path $S_1 \rightarrow S_2 \rightarrow S_1$), respectively. CPU Time in Level2 to Level3, Level3 to Level4 and Level4 to Level5 means the sum of CPU time used at the verification for all axioms.

6. Conclusion

In this paper, we have proposed a method for refining a given specification to a synchronous sequential circuit and verifying the correctness of the refinements automatically. We have developed a hardware design support system based on algebraic methods. We also gave three examples of implementation verifications.

In the method proposed here, the specifications and invariant assertions must be written in a restricted style. (As the variables, only the use of a variable of sort *state* is permitted.) In the verification, it is proved that rather strong sufficient conditions hold. However, for the GCD circuit, CPU and the maxsort circuit used in this paper as examples, we succeeded in proof.

After giving the assertions and theorems for primitive functions/predicates and the substitutions for the variables of the theorems, the verification is carried out automatically although it may fail under those assertions and theorems. The design and verification of those circuits were carried out in a few days, respectively, including trial and error. This depends on the features that our verifier has many user friendly verification facilities and an efficient decision procedure for the prenex normal form Presburger sentences bounded by only universal quantifiers.

References

1. S. Bose and A. Fisher : "Automatic Verification of Synchronous Circuits Using Symbolic Logic Simulation and Temporal Logic", Proc. IMEC-IFIP Int. Workshop on Applied Formal Methods for Correct VLSI Design, pp.759-764, 1989.
2. M. Browne, E. M. Clarke, D. Dill and B. Mishra : "Automatic Verification of Sequential Circuits using Temporal Logic", IEEE Trans. on Computer, Vol. C-35, No. 12, pp.1035-1044, 1986.
3. H. Busch : "Transformational Design in a Theorem Prover", IFIP Conf. on Theorem Provers in Circuit Design, North-Holland, pp.175-196, 1992.
4. D.C.Cooper : "Theorem Proving in Arithmetic without Multiplication", Machine Intelligence, No.7, pp. 91-99, 1972.
5. M. J. C. Gordon : "HOL : A Proof Generating System for Higher Order Logic", in VLSI Specification, Verification and Synthesis, G. Birtwistle and P. A. Subrahmanyam ed-s., Kluwer Academic Publishers, pp.73-128, 1988.
6. T. Higashino and K. Taniguchi : "A System for the Refinements of Algebraic Specifications and their Efficient Executions", Proc. of the IEEE 24-th Hawaii Int. Conf. on System Sciences (HICSS-24), Vol. II, pp.186-195 (Jan. 1991).
7. J.E. Hopcroft and J.D. Ullman : "Introduction to Automata, Theory, Languages, and Computation", Addison-Wesley, 1979.
8. IEEE : "IEEE Standard VHDL Language Reference Manual", IEEE,1988.
9. J. J. Joyce : "Formal Verification and Implementation of a Microprocessor", VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, pp.129-157, 1988.
10. T. Kasami, K. Taniguchi, Y. Sugiyama and H.Seki : "Principles of Algebraic Language ASL/*", Trans. of IECE Japan, Vol.69-D, No.7, pp.1066-1074, July 1986 (in Japanese).
11. J. Kitamiti, T. Higashino, K. Taniguchi and Y. Sugiyama : "Top-Down Design Method for Synchronous Sequential Logic Circuits Based on Algebraic Technique", Trans. of IEICE Japan, Vol.77-A, No.3, March 1994 (in Japanese).
12. T. Kropf : "Benchmark-Circuits for Hardware - Verification : v.1.0.0", the Benchmark-Circuits for 2nd Conf. on Theorem Proving in Circuits Design, FTP from Univ. of Karlsruhe (129.13.18.22), Germany, 1994.
13. Y. Nakamura : "An Integrated Logic Design Environment Based on Behavioral Description", IEEE Trans. on Computer-Aided Design Integrated Circuits & Systems, Vol. 6, No. 3, pp.322-336, May 1987.
14. V. Stavridou, J. A. Goguen, A. Stevens, S. M. Eker, S. N.Aloneftis and K. M. Hobley : "FUNNEL and 2OBJ : Towards as Integrated Hardware Design Environment", IFIP Conf. on Theorem Provers in Circuit Design, North-Holland, pp.197-223, 1992.
15. J.B.Saxe S.J.Garland, J.V.Guttag and J.J.Horning, "Using Transformations and Verification in Circuits Design", Designing Correct Circuits, North-Holland, pp.1-25, 1992.
16. G. Thuau ,B.Berkane, "Using the Language LUSTRE for Sequential Circuit Verification", Designing Correct Circuits, North-Holland, pp.81-96, 1992.
17. Open Verilog International : "Verilog Hardware Description Language Reference Manual", 1991.
18. M. Wirsing : "Structured Algebraic Specifications : A Kernel Language", Tech. Report, TU Mchen, 1984.