12 SC

Stanford Artificial Intelligence Laboratory
Memo AIM-304

September 1977

Computer Science Department
Report No. STAN-CS-77-632

HARDWARE VERIFICATION

by

Todd Jeffry Wagner

Research sponsored by

Advanced Research Projects Agency

COMPUTER SCIENCE DEPARTMENT
Stanford University

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. ~~REPORT NUMBER~~ (14) STAN-CS-77-632, AIM-304	2. GOVT ~~ACCESSION NO.~~	3. ~~RECIPIENT'S~~ CATALOG NUMBER (9) Doctoral thesis,
4. TITLE (*and Subtitle*) (6) Hardware Verification,		5. ~~TYPE OF REPORT & PERIOD~~ COVERED Technical
		6. PERFORMING ORG. REPORT NUMBER AIM-304
7. AUTHOR(s) (10) Todd Jeffrey Wagner		8. CONTRACT OR GRANT NUMBER(s) (15) MDA903-76-C-0206,
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory, Stanford University, Stanford, California 94305		[redacted] ARPA Order-2494
11. CONTROLLING OFFICE NAME AND ADDRESS Eugene Stubbs, ARPA/PM, 1400 Wilson Blvd., Arlington, VA 22209		12. REPORT DATE (11) Sep [redacted] 77
		13. NUMBER OF PAGES ~~102~~ (12) 105p.
14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*) Philip Surra, ONR Representative, Durand Aeronautics Building Room 165, Stanford University, Stanford, California 94305		15. SECURITY CLASS. (*of this report*) 15
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (*of this Report*)

Releasable without limitations on dissemination

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*)

20. ABSTR

Methods for detecting logical errors in computer hardware designs using symbolic manipulation instead of digital simulation are discussed. A nonprocedural register transfer language is proposed that is suitable for describing how a digital circuit should perform. This language can also be used to describe each of the components used in the design. Transformations are presented which should enable the designer to either prove or disprove that the set of interconnected components correctly satisfy the specifications for the overall system.

The problem of detecting timing anomalies such as races, hazards, and oscillations is addressed. Also explored are some interesting relationships between the problems of hardware verification and program verification. Finally, the results of using an existing proof checking program on some digital circuits are presented. Although the theorem proving approach is not very efficient for simple circuits, it becomes increasingly attractive as circuits become more complex. This is because the theorem proving approach can use complicated component specifications without reducing them to the gate level.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

[illegible]

094120

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

Stanford Artificial Intelligence Laboratory
Memo AIM-304

September 1977

Computer Science Department
Report No. STAN-CS-77-632

HARDWARE VERIFICATION

by

Todd Jeffry Wagner

ABSTRACT

Methods for detecting logical errors in computer hardware designs using symbolic manipulation instead of digital simulation are discussed. A non-procedural register transfer language is proposed that is suitable for describing how a digital circuit should perform. This language can also be used to describe each of the components used in the design. Transformations are presented which should enable the designer to either prove or disprove that the set of interconnected components correctly satisfy the specifications for the overall system.

The problem of detecting timing anomalies such as races, hazards, and oscillations is addressed. Also explored are some interesting relationships between the problems of hardware verification and program verification. Finally, the results of using an existing proof checking program on some digital circuits are presented. Although the theorem proving approach is not very efficient for simple circuits, it becomes increasingly attractive as circuits become more complex. This is because the theorem proving approach can use complicated component specifications without reducing them to the gate level.

This thesis was submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2494, Contract MDA903-76-C-0206. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University or any agency of the U. S. Government.

i

(See 1473)

PREFACE

This thesis represents a first attempt at developing methods for proving the correctness of computer hardware designs. After several years of experience working with simulators I realized that some sort of symbolic manipulation program would be useful in detecting errors in hardware designs. Initially I decided to pursue hardware verification along program verification lines, but the inherent lack of sequential statement execution in hardware made this approach untenable. There is a point at which program verification techniques do apply, but not until a hardware device is a sequential machine with a well understood mechanism for determining the next instruction. To verify circuits at the gate and flip-flop level requires some very different tools.

One of my major desires was to do all of this in a context that engineers would understand and be willing to use. A very simple non-procedural register transfer language is presented that can be learned in a few minutes. I believe that it includes the minimum amount of control information needed to prove things about how a device will perform in conjunction with other devices. Naturally, the specific syntax is a function of the keyboard I am using and many equivalent languages are possible.

Transformations that can be applied to statements in this language include the basic boolean operations, definitions of arithmetic functions, and some special identities that enable us to determine the effects of signal transitions as they travel thru a circuit. The list of transformations is probably far from complete and will have to be expanded as special hardware problems are encountered. An important objective is to define the language and transformations in such a way that they can be modelled by existing proof checking programs. A major reason for developing hardware verifiers is that they provide a way to prove the correctness of complex devices for which exhaustive simulation would be impractical.

I wish to thank Lynn Quam, who first suggested the idea of a hardware verifier (in the context of a block structured design automation system) as a thesis project. Richard Weyhrauch helped me with using his FOL (First-order Logic) proof checker on hardware problems. Finally, director John McCarthy and the faculty, students, secretaries, and bureaucrats have all made working at the Stanford Artificial Intelligence Lab an exciting experience.

Table of Contents

Table of Contents

1. INTRODUCTION

This thesis discusses a method for detecting errors in digital hardware designs. A system, or module, is specified using a non-procedural register transfer language. The components that will be used to build the device are also specified in this language. A symbolic manipulation technique can then be used to determine if the interconnected components correctly satisfy the specifications for the overall system. Essentially, the process of hardware verification is based on boolean reductions and some methods for analyzing the consequences of signal transitions. The symbolic manipulation approach should be able to detect timing anomalies, such as races and hazards, in addition to logical inconsistencies.

The desirability of using proofs of correctness to detect errors in hardware designs has been discussed previously [1,4], but everyone has a different idea about what level of description should be used. Some authors think in terms of a finite state automaton that can accept or reject an input sequence depending on the algorithm it represents. Still others are interested in determining if a given algorithm correctly implements a complex mathematical function. In this paper a proof of correctness will based on showing that, for any single input change, equivalent state changes will take place in both logic descriptions. In other words, if both models of the logic are in equivalent states and the same input variable is changed on each of them, then they will change to new states that are also equivalent.

Hardware verification, as presented here, is entirely oriented to the register transfer level of system description. Problems such as validation of physical layout or design rule checking are not addressed. The circuit level, involving semiconductor physics and critical time constants, is also not discussed. Nevertheless, the register transfer level does cover a large spectrum of computer design activities. It can be used to determine if the specifications of an integrated circuit are fulfilled by the gates within the chip, or if a set of chips satisfy the definition of a complete backplane. These techniques can also be used to see if interconnected cards or backplanes correctly satisfy the requirements for whole computers.

The most important idea behind hardware verification is the concept of breaking down the specifications as little as possible. If a designer specifies the operation of binary addition in the system description and provides an adder chip in the actual circuit, then it will only be necessary to show that the correct data and control signals are sent to this chip. In this case, breaking the addition operation down to the gate level will not be necessary. This is especially useful with LSI components such as microprocessors. The microprocessor can supply the complex operations and the verifier will only have to show that the support circuitry sends it the correct control information.

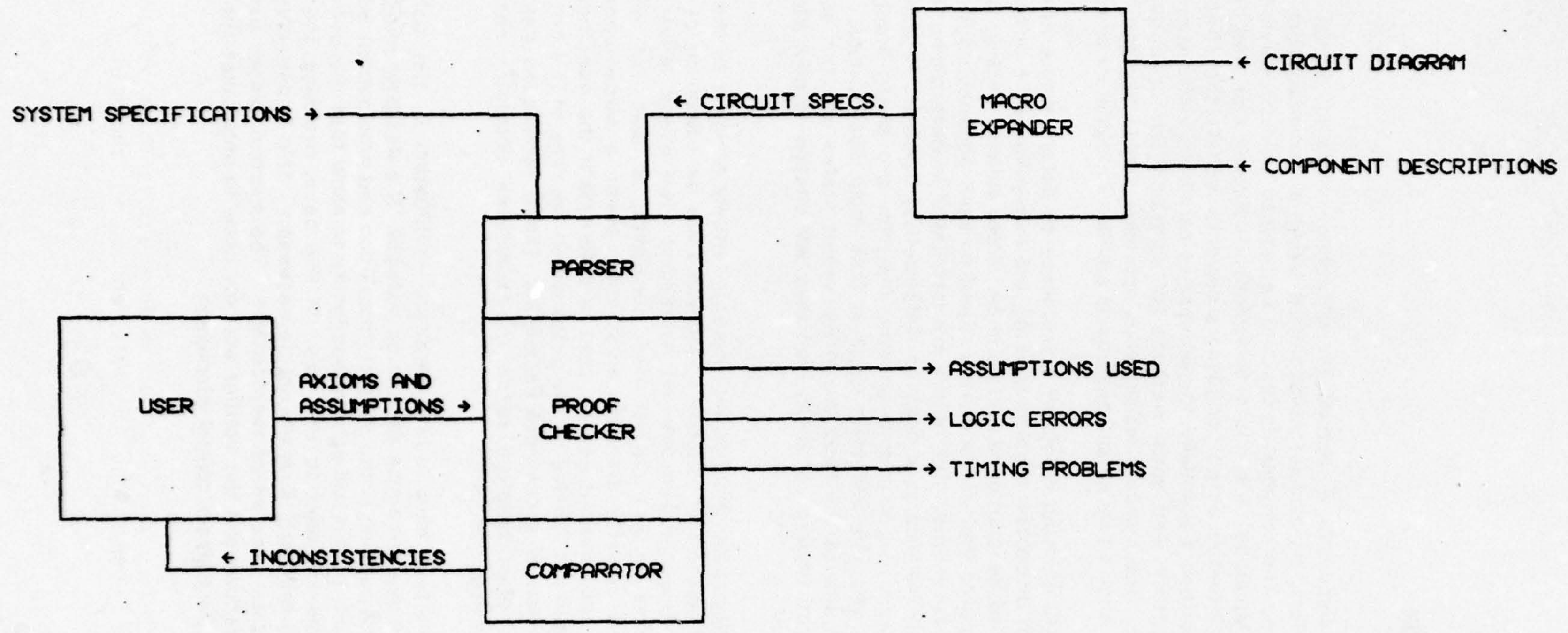


FIGURE 1.1 HARDWARE VERIFICATION

1.1 Basic assumptions

A couple of assumptions must be made regarding timing. Any circuit can be made to operate erroneously if certain external inputs are changed simultaneously, so race detection will not include these circumstances. If the circuit is to be used in conjunction with other circuits it will be possible to detect races in the composite device by verifying the overall circuit. Components will not have specific delays. When a possible race is detected a hardware verifier might be able to indicate which component(s) must be faster than which other component(s) to insure correct operation. The verification process does not say anything about how fast a circuit will operate, only that it will work correctly at some (slow enough) speed if the races and hazards are taken care of.

There is also the philosophical question of what to do about unspecified operations. The actual circuit will almost always include other operations and state variables in addition to those specified in the system description. If a designer uses an LSI device such as a microprocessor to perform only a few operations there will be a wealth of unspecified operations. A three bit counter can be implemented by using a four bit counter circuit and not connecting the high order bit. At the same time, it is easy to design a circuit in which the unspecified operations can mean disaster. It is important to be able to differentiate between additional operations that are acceptable and those that are not, since a circuit can satisfy its specifications without being equivalent to them. Some solutions to this problem are discussed in the section on feedback.

1.2 Relationship to design compilers

At present there are a number of design compilers that can take the specifications for a hardware device and generate the necessary circuitry. The resulting design will be correct and have a near minimal number of gates and flip-flops because of the use of sophisticated state reduction techniques. Although fairly successful, design compilers have several shortcomings. In much the same way that good hand coding of software can beat an optimizing compiler, circuit designers can usually find novel ways to use components that a design compiler would miss. With current LSI components it is often cheaper to throw in a complex device than to build a circuit using the minimal number of gates and flip-flops. A more serious problem is that it is virtually impossible to give a design compiler the ability to use more than a small fraction of the new components rapidly becoming available. Furthermore, the best component for a given application can depend on such varied factors as price, size, and shape as well as the usual electrical characteristics. Under these circumstances, it would seem more reasonable to use a program that lets the designer use whatever components he chooses and then checks for design errors.

A major advantage of a hardware verifier is that it can provide great flexibility when modifying a circuit. Suppose that a device has been in the field for a few years when a new component is developed that can replace several used in the original design. Using hardware verification it should be fairly simple to determine if using the new component will change the behavior of the device in any way. If the behavior is changed, the verification system can indicate what the differences are so they can be dealt with appropriately.

Incidentally, one approach to hardware verification has been to submit two specifications for the same device to a design compiler and see if it generates equivalent circuits [5]. It is unclear how this would be used when one description can satisfy the other without being equivalent to it.

1.3 Advantages over simulation

The most popular method for checking hardware designs is digital simulation. Earlier attempts at comparing high level and low level descriptions of digital devices based on simulation are discussed in [2,3]. Admittedly, digital simulation is much easier to implement on a computer than the symbolic manipulation techniques presented in this paper. Simulation methods are well understood, and are very useful for generation of fault tests that can be used to speed repairs after a device is in the field. On the other hand, simulation cannot determine if a design is correct unless it is exhaustive. It may be lucky enough to find some bugs, but others may go undetected.

The greatest disadvantage of simulation is that certain assumptions must be made about initialization, delays, and the possible values of signals. Unless the logic being designed has a reset signal it is impossible to determine the initial state of sequential elements at power-up time. Some simulators assume zero delay components, others give each component a unit delay, while still others give each component type a different delay value. These delay values may have little to do with the real world where two parts of the same type can have very different propagation delays. Simulators are generally two value or three value. Two value simulators use only zero and one, while three value simulators include an undefined state to be used during initialization and signal transitions. Three value simulators can detect races, even where they don't really exist, under certain delay assumptions.

Hardware verification can determine correctness of a circuit and produce a list of possible races and hazards, along with information on how to prevent them. There are no assumptions about circuit initialization since it is based on showing that a circuit satisfies its specifications regardless of initial state. Hardware verification will also generate results that are easier to interpret. It can show inconsistencies in the context of boolean expressions instead of the pages of ones and zeros output by most simulators. Symbolic manipulation techniques can often be much faster than simulation since they can handle arbitrarily complex devices without always breaking them down to the gate level.

2. DESIGN AUTOMATION LANGUAGES

An incredible number of hardware design languages have been proposed, many have been implemented, and several have become popular in the design community. Entire papers have been devoted to the philosophy of these languages [7,9,16,27,36,38,45,49] and to surveys comparing and contrasting some of the more popular ones [8,25,33,50,51]. Design automation languages are generally classified as structural or behavioral, although some attempt to be both. Structural languages are primarily intended to provide information such as where data paths go and where control information originates. A good example of a purely structural language is PMS [11,12,47], which is actually just a formalized notation for drawing block diagrams of computer systems.

Behavioral languages, on the other hand, attempt to describe what a system does without implying anything about the hardware structure. These languages are often used as simulation models and to help in software production. At a somewhat more detailed level behavioral languages can be organized to provide some inkling of how the system itself is organized, but they could just as easily imply an entirely different structure and still emulate the hardware correctly. Virtually all behavioral languages are register transfer languages, in which the basic unit of information is a vector of binary digits. Since they were first proposed by Reed [42] in the 1950's register transfer languages have developed into procedural and non-procedural types. Several of the procedural languages have complex control structures that permit them to operate in a non-procedural manner.

2.1 Procedural register transfer languages

Most procedural register transfer languages are programming languages modified to handle register operations. Instructions are executed sequentially, but in some languages special constructs are provided to permit parallel execution of blocks. Several procedural hardware languages are based on APL [13,26,28,30,31,32], whose advocates are quick to point out the tremendous power of the vector and matrix operations already in the language. Although very compact, hardware descriptions in APL-like languages are among the most difficult to read. Another large group of procedural hardware languages are based on Algol [9,11,12,22,29,35,46]. The block structure of Algol can often be used to help describe the system structure.

Still other procedural languages are based on state machines [14,15,23,24]. In state machine languages the logic is partitioned into blocks whose execution is controlled by state registers. Special instructions are provided to allow the state registers to be altered. Finally, there are some languages [34,39,44] that divide the logic into data manipulation blocks and a control microprogram. These are essentially the same as the state machine languages except that in the latter the control microprogram is embedded in the data manipulation blocks.

The major advantage of procedural register transfer languages is that algorithms are easy to read and understand. Many of the procedural languages provide subroutine and macro facilities. Since there is a definite order for executing statements, GOTO statements are acceptable. Procedural hardware languages are excellent for systems level simulation and are frequently used when producing software for new computers.

Unfortunately, the details of how the control and timing circuits operate are not revealed to the user. Since statements are executed more or less sequentially, it is implied that there is some control logic making sure things happen at the correct time. In many procedural languages specifying parallel operations and whether an operation is synchronous or asynchronous is very difficult. For these reasons, the hardware verification process described in this paper will be based on a non-procedural register transfer language.

2.2 Non-procedural register transfer languages

The most popular non-procedural register transfer language is called the Computer Design Language, or CDL [17,18,19,20,21]. This popularity is largely due to fact that CDL translators and simulators have been made available to the university community. Below is a CDL description of a stored program computer. First come the declarations which specify the names of all of the registers, switches, and circuit nodes.

```
Register,       PC(0-11),               $program counter
                IR(0-15),               $instruction register
                ACC(0-15),              $accumulator
                ADR(0-11),              $address register
                CYCLE(0-1),             $control register
                G,                      $run/halt flag

Subregister,    IR(OP)=IR(0-3),         $opcode part of IR
                IR(MA)=R(4-15),         $address part of IR

Memory,         M(ADR)=M(0-4095,0-15),  $memory and address register

Decoder,        C(0-3)=CYCLE,           $decode control register
                I(0-15)=IR(OP),         $decode instruction opcode

Terminal,       STOP=C(0),              $halt state
                FETCH=C(1),             $instruction fetch
                EX=C(2),                $instruction execute
                LD=I(0),                $load accumulator
                ST=I(1),                $store accumulator
                CLA=I(2),               $clear accumulator
                ADD=I(3),               $add instruction
                SUB=I(4),               $subtract instruction
                JMP=I(5),               $jump command
```

```
Switch,         RUN(ON),                $run button
                HALT(ON),               $halt button

Clock,          T(1-2),                 $two-phase clock
```

Register declarations define the size and bit ordering (msb-lsb) for each register, while subregister declarations permit a group of bits to be given a special name. The memory declaration specifies both the size of a memory and which register will be used as the address register. Decoders are used to create a vector in which only one bit is true, selected by the value of a register. In the example above, C(0) will be true if CYCLE=0, C(1) will be true if CYCLE=1, and so on. Switch statements provide variables that can be controlled internally by the hardware and externally by data supplied to the CDL simulator. Terminal statements are used to rename terminals and describe logic networks. The clock statement can provide multiple clock phases, and in this example clock phases T(1) and T(2) will be repeated continuously.

There are also CDL declarations that permit several operations to be grouped into a block, creating a sort of subroutine/macro facility. Some of the declarations, such as subregisters and terminals, are not absolutely necessary. The same logic could be described by using subscripts and the original terminal names in the logic description. However, these declarations do tend to make the description easier to understand. Below are the statements describing the actual operations of a computer using the clocks and registers declared previously.

```
Comment, run and halt controls.
/RUN(ON)/       G←1,
/HALT(ON)/      G←0,
/STOP*T(2)/     IF (G=1) THEN (CYCLE←1),
/EX*T(2)/       IF (G=0) THEN (CYCLE←0) ELSE (CYCLE←1),

Comment, instruction and operand addressing.
/FETCH*T(1)/    ADR←PC,
/FETCH*T(2)/    IR←M(ADR), CYCLE←2, PC←countup PC,
/EX*T(1)/       ADR←IR(MA),

Comment, instruction execution.
/LD*EX*T(2)/    ACC←M(ADR),
/ST*EX*T(2)/    M(ADR)←ACC,
/CLA*EX*T(2)/   ACC←0,
/ADD*EX*T(2)/   A←A add M(ADR),
/SUB*EX*T(2)/   A←A sub M(ADR),
/JMP*EX*T(2)/   PC←IR(MA),
                End
```

The expression enclosed in slashes is a conditional expression where the "*" should be interpreted as a logical AND. Most statements contain one or more terminals and a clock phase. If all of the terminals are true, then the operation indicated will occur at the specified time. Although they make the description more readable, the IF-THEN and IF-THEN-ELSE statements are not really needed. The same effects could be obtained by modifying the conditional expressions to include the information in the IF clauses. Arithmetic and logical operations are performed by predefined functions such as "shr" (shift right), "countup" and the like.

Although the specific syntax of declarations and statements may vary, virtually all non-procedural register transfer languages share the concept of permitting a conditional label to be attached to each statement. Obviously a non-procedural hardware description is more detailed than a procedural one. At the same time, algorithms become far less readable. Macro-like features can be provided, but true subroutines cannot because going to a routine and returning are sequential events. Parallelism is not a problem since any number of statements can be executed at the same time. CDL is good for describing synchronous logic but cannot be used for asynchronous circuits.

The Asynchronous Circuit Design Language, or ACDL [10], permits transitions such as $X \rightarrow 0$ or $X \rightarrow 1$ in the conditional expressions. This means that events can be described as happening on the negative or positive edge of signal transitions. ACDL is a state machine language which uses these transitions to change states. For the purpose of hardware verification we will develop a language similar to CDL that also allows transition information in the conditional expressions.

3. HARDWARE VERIFICATION LANGUAGE

Since the purpose of this paper is to discuss hardware verification techniques and not to create yet another elegant design language, the language presented here is extremely simple. Such luxuries as macros and complex operators are not included at this time. As with most register transfer languages, the actual syntax is largely a function of the keyboard available to the author. A BNF description of this language will be found in Appendix A.

3.1 Basic syntax

To begin with, there are no declarations. Hardware verification involves checking a higher level description of a device against a lower level description of the same device. In a sense, the higher level description can be thought of as providing the register declarations for the lower level description. In other computer design languages register declarations are used to allocate storage space for simulation. This is not necessary here because we are not going to simulate the logic. Subregisters and memories can be indicated by using subscripts in the actual statements. Clock declarations are not needed since any variable can be used as a clock. Every statement will be in one of the two forms shown below. The conditional expression is optional and will not be needed when describing combinational devices.

```
variable ← expression;

/conditions/ variable ← expression;
```

3.2 Variables and constants

A variable name must begin with a letter and can contain letters, digits, single quotes, and underscores. The name can be any length. Variables may also have one or two subscripts enclosed in square brackets. If there are two subscripts, they are separated by a comma. The first subscript is a bit number and can be a constant, two constants separated by a colon, or an expression. The second subscript is a memory location and may be a constant or an expression. Additionally, variables can be joined using the "&" (concatenation) operator. Some examples of variables are shown below.

```
X123'
PROGRAM_COUNTER
ACC[3]
INDEX[0:15]
REG1[BIT[0:3]]
MEMORY[0:15,PC[0:15]+BASE_REG[0:15]]
A[0]&B[2:5]
```

For the time being constants are in decimal. Binary constants can be written as ones and zeroes concatenated together.

3.3 Expressions and operators

Expressions can consist of variables, constants, and other expressions joined by various logical and arithmetic operators. The result of an expression can be one bit or a vector of bits. Below is a list of the operators in order of their precedence. Parentheses can be used to alter precedence, and operations of equal precedence are evaluated from left to right.

¬	logical complement
↑ ↓	transitions (conditional exp. only)
&	concatenation
+ -	arithmetic operations
= ≠ > < ≥ ≤	arithmetic relations
∧	logical and
∨ ⊕	logical or/exclusive or

Conditional expressions are similar to other expressions except that they must have a one-bit (true/false) result. This means that conditional expressions can have the above operations (except concatenation) performed on one-bit operands, and may include arithmetic relations when comparing bit vectors.

The transition operators are "↑" (0→1 transition) and "↓" (1→0 transition). These can only be used in conditional expressions, and must only be applied to one-bit operands or subexpressions. It is important to avoid constructs such as "¬↑" and "¬↓" except to indicate feedback states in which the variable does not change. Otherwise it would imply that state changes can happen as a result of a transition not occuring. A basic assumption in hardware verification is that events are caused by other events, and it is unclear how we should handle events that are the result of non-events.

```
/¬↑X/ A←A;                    (OK - indicates feedback)

/¬↑X/ A←Y;                    (not acceptable)
```

There is a special meta operator designated by the "*" character. It is used in conjunction with subscript expressions to mean all values not equal to that expression. In other words, the expression

```
A[*B[0:7]]
```

might be used to indicate all of the bits in register A except the one pointed to by the value in register B. The reasons for having this operator, and some special rules for how to use it, are discussed in the section of feedback.

3.4 Language examples

Here are some examples of statements in the hardware verification language. Parentheses are added for clarity in some places where the precedence of operations would have done the correct thing. Note that some logical operations, such as shifts and rotates, can be accomplished thru subscript manipulation and do not require special operators. The first three examples represent very simple gates and flip-flops, while the other examples are what might be found in computer descriptions or LSI component specifications.

```
A←¬(B∧C);                      (combinational)

/CLK/ Q←D;                     (asynchronous)
/¬CLK/ Q←Q;

/↓CLK/ Q←(J∧¬Q)∨(¬K∧Q);        (synchronous)
/¬↓CLK/ Q←Q;

/(OPCODE[0:5]=0&1&0&0)∧↑T2/ A[0:15]←A[1:15]&A[0];
/(X∧↑PHASE1)∨(Y∧↓PHASE2)/ A[0:15]&B[0:15]←B[15]&A[0:15]&B[0:14];
/FETCH∧↓(T1∧ENABLE)/ INSTR[0:15]←MEMORY[0:15,PC[0:11]];
/↑P2∨(↓P3∧CLOCK_ENAB)/ ACC[BIT[0:3]]←A[0]●OVERFLOW;
```

4. DEFINITIONS AND AXIOMS

In order to show that a lower level description of a device satisfies some higher level specification, we must develop ways to manipulate the statements in the descriptions. The identities in this section are those that the author has found useful in solving specific verification problems. This list is by no means complete, and additional transformations will have to be invented as more circuits are analyzed.

In the definitions of vector operations bit 0 is the most significant bit, although the opposite bit ordering can be implemented just as easily. Note that in most of these definitions X[i:j] and Y[i:j] are used only to make the notation simpler. These definitions are valid for arbitrary vectors (exp1&exp2&...&expn) whether they are part of the same register or not. Generally speaking, letters near the end of the alphabet such as X,Y, and Z will be used to denote arbitrary expressions, including variables. Letters at the beginning of the alphabet, namely A and B, will indicate variable names only.

Arithmetic operations are generally defined in the context of positive integers. Other arrangements, such as 2's complement or signed magnitude, can be developed from these with some extra logic in the user's circuit descriptions. It is also possible to implement definitions for other types of arithmetic in the verifier itself.

4.1 Boolean reductions

Switching algebra identities like those shown below can be found in most logic design texts. The list below is neither exhaustive nor is it minimal. A hardware verification program could use these transformations directly in a pattern matching routine, or it could use some iterative boolean minimization techniques based on only a few of them. The first few axioms define the basic nature of AND, OR, and NOT.

T1. $\neg 0 \equiv 1$
$\neg 1 \equiv 0$

T2. $X \vee 1 \equiv 1$
$X \wedge 0 \equiv 0$

T3. $X \vee 0 \equiv X$
$X \wedge 1 \equiv X$

T4. $X \vee X \equiv X$
$X \wedge X \equiv X$

T5. $\neg(\neg X) \equiv X$

T6. $X \vee \neg X \equiv 1$
$X \wedge \neg X \equiv 0$

The next few transformations involve combinations of two or more variables. They can be derived from the identities above by substituting ones and zeros for the variables.

T7. $X \vee Y \equiv Y \vee X$
$X \wedge Y \equiv Y \wedge X$

T8. $X \vee (X \wedge Y) \equiv X$
$X \wedge (X \vee Y) \equiv X$

T9. $(X \vee \neg Y) \wedge Y \equiv X \wedge Y$
$(X \wedge \neg Y) \vee Y \equiv X \vee Y$

T10. $(X \vee Y) \vee Z \equiv X \vee (Y \vee Z)$
$(X \wedge Y) \wedge Z \equiv X \wedge (Y \wedge Z)$

T11. $X \wedge (Y \vee Z) \equiv (X \wedge Y) \vee (X \wedge Z)$
$X \vee (Y \wedge Z) \equiv (X \vee Y) \wedge (X \vee Z)$

T12. $(X \vee Y) \wedge (\neg X \vee Z) \wedge (Y \vee Z) \equiv (X \vee Y) \wedge (\neg X \vee Z)$
$(X \wedge Y) \vee (\neg X \wedge Z) \vee (Y \wedge Z) \equiv (X \wedge Y) \vee (\neg X \wedge Z)$

T13. $(X \vee Y) \wedge (\neg X \vee Z) \equiv (X \wedge Z) \vee (\neg X \wedge Y)$

T14. $\neg (X \vee Y) \equiv \neg X \wedge \neg Y$
$\neg (X \wedge Y) \equiv \neg X \vee \neg Y$

The last identity, DeMorgan's law, is valid for expressions of arbitrary length. However, the two variable version may be easier to implement and can acheive the same result by being applied repeatedly. It may also be convenient to define a series of transformations for the exclusive-or function.

X1. $(X \wedge \neg Y) \vee (\neg X \wedge Y) \equiv X \bullet Y$

X2. $X \bullet 0 \equiv X$

X3. $X \bullet 1 \equiv \neg X$

X4. $X \bullet X \equiv 0$

X5. $X \bullet \neg X \equiv 1$

X6. $X \bullet Y \equiv Y \bullet X$

X7. $(X \bullet Y) \bullet Z \equiv X \bullet (Y \bullet Z)$

X8. $\neg (X \bullet Y) \equiv \neg X \bullet Y$

4.2 Definitions for subscripts and concatenation

Definitions D1 thru D4 can be used to reduce several concatenated variables into a single register variable, or vice-versa. The first two are used to change register variables with only one or two subscripts into the full three subscript format. This is to make them compatible with D3 and D4. Another approach would be to develop identities like D3 and D4 for each combination of one or two subscripts. In a hardware verification program it would probably be easiest to have the input parser convert registers into the three subscript format automatically.

```
D1.  A[i] ≡ A[i:i]

D2.  A[i:j] ≡ A[i:j,0]

D3.  A[i:j,s]&A[j+1:k,s] ≡ A[i:k,s]
     where i≤j<k

D4.  ¬A[i:j,s]&¬A[j+1:k,s] ≡ ¬A[i:k,s]
     where i≤j<k
```

The next definition points out that concatenation is associative, while the following three show that performing a logical operation on vectors (of equal length) and concatenating the result is equivalent to concatenating the vectors and then performing the logical operation. Keep in mind that these definitions are valid for arbitrary vectors which do not have to be part of the same register.

```
D5.  (X&Y)&Z ≡ X&(Y&Z)

D6.  (X[i:j]∧Y[i:j])&(X[j+1:k]∧Y[j+1:k]) ≡ X[i:k]∧Y[i:k]
     where i≤j<k

D7.  (X[i:j]∨Y[i:j])&(X[j+1:k]∨Y[j+1:k]) ≡ X[i:k]∨Y[i:k]
     where i≤j<k

D8.  (X[i:j]•Y[i:j])&(X[j+1:k]•Y[j+1:k]) ≡ X[i:k]•Y[i:k]
     where i≤j<k
```

Definitions D9 and D10 (decoder) show how to expand a variable with an expression as a subscript when it is used as the destination in a register transfer. The definitions are nearly identical, except that D9 is used when the first subscript is an expression and D10 is used to expand the second subscript. If a variable has expressions for both subscripts we can use D9 and then D10 or vice-versa. It would not be reasonable to expand a statement into several thousand using these definitions, but a theorem prover could use this information to determine the status of specific bits in a register or memory without generating statements for all possible values of n.

```
D9.  /X/ A[Y[i:j],s]←Z;
          ≡
     /X∧(Y[i:j]=n)/ A[n,s]←Z;

D10. /X/ A[s,Y[i:j]]←Z;
          ≡
     /X∧(Y[i:j]=n)/ A[s,n]←Z;
```

Similarly, D11 and D12 (multiplexor) indicate how to expand a variable with an expression as a subscript when that variable is used as part of an expression. This includes conditional expressions as well as register transfer expressions. As before, there are versions for the *first subscript* and the *second subscript.*

```
D11. A[Y[i:j],s] ≡
     (A[0,s]∧(Y[i:j]=0))v(A[1,s]∧(Y[i:j]=1))v...v
     (A[n,s]∧(Y[i:j]=n))
     when A[Y[i:j],s] is used in an expression

D12. A[s,Y[i:j]] ≡
     (A[s,0]∧(Y[i:j]=0))v(A[s,1]∧(Y[i:j]=1))v...v
     (A[s,n]∧(Y[i:j]=n))
     when A[s,Y[i:j]] is used in an expression
```

4.3 Definitions of arithmetic operators

In this subsection we have the definitions for the various arithmetic operations and relations permitted in the language. The first two definitions are for the carry operation. Definition A1 puts it in terms of ANDs and ORs, while A2 provides a way to combine carries from vectors to determine the carry function for larger vectors. The second definition will probably find more use since most adder chips provide carry-in and carry-out pins but take care of their own internal carries.

```
A1.  X[i]∧Y[i]v(X[i]vY[i])∧CI ≡ carry(X[i],Y[i],CI)
     X[i]∧Y[i]v(X[i]vY[i])∧carry(X[i+1:j],Y[i+1:j],CI) ≡
     carry(X[i:j],Y[i:j],CI)

A2.  carry(X[i:j],Y[i,j],carry(X[j+1:k],Y[j+1,k],CI) ≡
     carry(X[i:k],Y[i:k],CI)
     where i≤j<k
```

Addition is a fairly sophisticated operation, but becomes much simpler using the carry functions defined above. It can be specified on a bit by bit level, as in A3, or as a concatenation of adders as in A4. Once again, the definition which permits us to concatenate small adder chips into big adders will be the most useful *in real verification problems. The carry-in* (CI) *bit is included in the* addition definitions to facilitate working with adder chips which usually include this input.

```
A3.  (X[i]⊕Y[i]⊕CARRY(X[i+1:j],Y[i+1:j],CI)&
     (X[i+1]⊕Y[i+1]⊕CARRY(X[i+2:j],Y[i+2:j],CI)&...&
     (X[j]⊕Y[j]⊕CI) ≡ X[i:j]+Y[i:j]+CI

A4.  (X[i:j]+Y[i:j]+CARRY(X[j+1,k],Y[j+1,k],CI))&
     (X[j+1:k]+Y[j+1:k]+CI) ≡ X[i:k]+Y[i:k]+CI
     where i≤j<k
```

We will also define two special cases of addition, subtraction and increment. Subtraction can be implemented in either ones-complement or twos-complement, and the carry-in bit defined above provides an easy way to select which. Although increment could be derived from the addition definition as needed, we will make it an axiom. This can make some verification problems much shorter since increment is a very popular function.

```
A5.  X[i:j]+¬Y[i:j]+1 ≡ X[i:j]-Y[i:j] (for 2's complement)
     X[i:j]+¬Y[i:j]+0 ≡ X[i:j]-Y[i:j] (for 1's complement)

A6.  X[i]⊕(X[i+1]∧...∧X[j])&X[i+1]⊕(X[i+2]∧...∧X[j])&...&¬X[j]
     ≡ X[i:j]+1
```

Next come the arithmetic relations. Only equal-to and greater-than are defined in detail because the others can be defined in terms of these two. Single bit comparisons for these are shown in A7 and A11, while bit by bit comparisons are provided in A8 and A12. The definitions in A9 and A13 show how individual comparator chips can be connected to form larger comparators.

```
A7.  ¬X[i]⊕Y[i] ≡ X[i]=Y[i]

A8.  (X[i]=Y[i])∧(X[i+1]=Y[i+1])∧...∧(X[j]=Y[j]) ≡ X[i:j]=Y[i:j]

A9.  (X[i:j]=Y[i:j])∧(X[j+1:k]=Y[j+1:k]) ≡ X[i:k]=Y[i:k]
     where i≤j<k

A10. ¬(X[i:j]=Y[i:j]) ≡ X[i:j]≠Y[i:j]

A11. X[i]∧¬Y[i] ≡ X[i]>Y[i]

A12. (X[i]>Y[i])∨((X[i]=Y[i])∧X[i+1]>Y[i+1])∨...∨
     ((X[i:j-1]=Y[i:j-1])∧X[j]>Y[j]) ≡ X[i:j]>Y[i:j]

A13. (X[i:j]>Y[i:j])∨((X[i:j]=Y[i:j])∧X[j+1:k]>Y[j+1:k])
     ≡ X[i:k]>Y[i:k]
     where i≤j<k
```

```
A14. Y[i:j]>X[i:j] ≡ X[i:j]<Y[i:j]

A15. ¬(X[i:j]>Y[i:j]) ≡ X[i:j]≤Y[i:j]

A16. ¬(Y[i:j]>X[i:j]) ≡ X[i:j]≥Y[i:j]
```

4.4 Miscellaneous transformations

There are several transformations that permit us to combine statements or expand one statement into several others. We will consider these to be "common-sense" axioms. The first axiom changes statements that do not have a conditional part into one where the conditions are always true. This is so they can be used with the other axioms in this section. It could be invoked automatically by the verifier's input parser. Axiom M2 lets us substitute the expression part of a combinational statement for the variable when that variable appears in other statements.

```
M1.  A←Y; ≡ /1/ A←Y;

M2.  /1/ A←Y; ⊃ (A ≡ Y)
```

Axiom M3 allows two statements affecting the same variable to be combined, provided that their conditional expressions are identical except for one subexpression. This subexpression must be complemented in one statement and uncomplemented in the other. The next axiom, M4, shows how any variable or expression can be moved out of the register transfer part. This is done by creating two new statements, one showing what would happen if that variable were one and the other showing what would happen if it were zero. Axiom M5 allows us to divide statements having a logical OR in the conditional part into two separate statements, or to combine two statements with the same register transfer part. *Similarly, M6 can be used to combine statements with the same* conditional expression or to divide a statement involving vectors into two or more statements.

```
M3.  /W∧¬X/ A←Y;
     /W∧ X/ A←Z;
           ≡
     /W/ A←(¬X∧Y)∨(X∧Z);

M4.  /X/ A←...Y...;
           ≡
     /X∧ Y/ A←...1...;
     /X∧¬Y/ A←...0...;
```

```
M5.  /X∨Y/ A←Z;
        ≡
     /X/ A←Z;
     /Y/ A←Z;

M6.  /X/ A←Y;                 (note: A and Y must be the same
     /X/ B←Z;                  length, same for B and Z)
        ≡
     /X/ A&B←Y&Z;
```

The next two transformations state that if a certain variable or subexpression appears in both the conditional expression and the register transfer expression, then it may be possible to determine what its value will be when that event occurs. Axiom M9 is similar except that another statement affected by the same conditions can control the value of a variable in the register transfer expression. Since there is the possibility of a critical race under some circumstances these should be used very carefully.

```
M7.  /X/ A←...Y...;
     and (X⊃Y)
        ≡
     /X/ A←...1...;

M8.  /X/ A←..¬Y...;
     and (X⊃Y)
        ≡
     /X/ A←...0...;

M9.  /X/ A←...B...;
     /Y/ B←Z;
     and (X⊃Y)
        ⊃
     /X/ A←...Z...;
```

Axiom M10 can be used to introduce arbitrary expressions into the conditional part of a statement. This axiom will normally be used to convert statements into a form that can be used elsewhere. Finally, axiom M11 can be used when a variable appears ANDed with an expression that can control its value. It comes in two flavors, complemented and uncomplemented.

```
M10. /X/ A←Z;
        ⊃
     /X∧Y/ A←Z;

M11. /X/ A←Y;
        ⊃
     X∧A ≡ X∧Y;
     X∧¬A ≡ X∧¬Y;
```

5. FEEDBACK

A major problem with describing components in the language used here is how to separate feedback from don't cares. Most component descriptions will not contain every possible combination of conditional expressions. Those undefined conditions can either be considered don't cares, in which the state of the device may change in some random manner, or feedback states in which the component's state variables do not change. Obviously it is necessary to differentiate between the two.

5.1 Implicit feedback

This subsection represents something of a digression because, after exploring the possibilities of making feedback implicit, the author decided to use explicit feedback for the proofs in this paper. Nevertheless, it may be reasonable to implement some of these ideas depending on the structure of the proof checker being used.

If feedback is to be implicit, then statements must be added to a circuit description to specify when a variable can enter a "don't care" state. This can become very messy when using a component having several undefined operations. In a microprocessor, for example, statements such as

```
/(OPCODE[0:7]="100)∧↑T1/ ACC[0:7]← don't care;
/(OPCODE[0:7]="100)∧↑T1/ X[0:7]← don't care;
/(OPCODE[0:7]="100)∧↑T1/ Y[0:7]← don't care;
/(OPCODE[0:7]="100)∧↑T1/ PC[0:15]← don't care;
```

will have to be added for each undefined opcode. Some of this might be circumvented by developing meta operators to cover large numbers of undefined operations.

On the other hand, using implicit feedback means that a component will not change state unless specifically changed by some statement. Statements of the form

```
/X/ A←A;

/¬↑X/ A←A;
```

are not needed in component descriptions. This is especially convenient when defining registers and memories. If one is writting data into a memory it will not be necessary to show that locations other than the one being addressed remain unchanged.

Since unspecified conditions imply that the system will not change state, statements with explicit feedback can be discarded from a logic description. Here are three transformations in which statements are completely eliminated:

```
/X/ A←A; ≡ null                                   (1)

/X∧A/ A←1; ≡ null                                 (2)

/X∧¬A/ A←0; ≡ null                                (3)
```

The first one permits us to remove a statement with explicit feedback because the language assumes that the variable will remain the same unless explicitly altered. The other two statement types can be removed because they don't do anything. In the second example the variable must already be one in order for it to be set to one. The third example is similar to the second and shows a case where a variable must be zero in order to be set to zero.

We can also derive some general rules for removing the variable being changed from the conditional expression. To do this we first need the following identities:

```
/X∧A/ A←0; ≡  /X/ A←0;                            (4)

/X∧¬A/ A←1; ≡  /X/ A←1;                           (5)
```

The left side of (4) says that if the expression is true and the variable is true, then the variable goes to zero. If the variable is already zero, then it will remain that way. Therefore, whenever the expression is true, the variable will be zero. Identity (5) uses similar reasoning. Using these transformations we can now derive the following general rules:

```
/X∧A/ A←Y; ≡  /X∧¬Y/ A←0;                         (6)

/X∧¬A/ A←Y; ≡  /X∧Y/ A←1;                         (7)
```

The derivation for theorem (6) is shown below, and theorem (7) can be derived in exactly the same way with certain variables complemented.

```
/X∧A/ A←Y;                given

/X∧A∧Y/ A←1;              M4
/X∧A∧¬Y/ A←0;

/X∧A∧¬Y/ A←0;             other statement eliminated
                          using (2)

/X∧¬Y/ A←0;               (4)
```

It is apparent that implicit feedback can provide some fairly compact component descriptions and that transformations (1-7) make it easy to simplify some statements. Unfortunately, when proving the correctness of a circuit it may be necessary to show that a given variable does not change state under certain circumstances. Using implicit feedback this would amount to checking all of the statements involving that variable to make sure none of them can cause a change. With many proof checkers, including the FOL system discussed in this paper, it is much easier to make feedback explicit and then verify statements like

```
/X/ A←A;
```

the same way other register transfers are verified. For this reason we will use explicit feedback for the remainder of this paper.

5.2 Axioms for explicit feedback

By using explicit feedback the problem of specifying don't care conditions is immediately eliminated. Any set of conditions not accounted for in a component description corresponds to a don't care state. On the other hand, some component descriptions may become somewhat longer since statements must be added to indicate conditions under which the state of a variable does not change.

The only major problem is in specifying registers and memories, in which large numbers of variables remain unchanged. To solve this we have the meta operator "*", which is used to indicate all locations other than the addressed location. Using this, a typical random access memory might be defined as follows:

```
/WRITE/ MEMORY[0:7,ADDR[0:11]]←DATA[0:7];
/¬WRITE/ MEMORY[0:7,ADDR[0:11]]←MEMORY[0:7,ADDR[0:11]];
MEMORY[0:7,*ADDR[0:11]]←MEMORY[0:7,*ADDR[0:11]];
```

This would mean that the unaddressed locations represent feedback conditions whether the memory is being read or written. To completely formalize the notion of the "*" operator we need a few more axioms. Axioms F1 and F2 permit all of the unaddressed bits to be set to the same expression, while F3 and F4 have the unaddressed bits remaining at their current value.

```
F1.  /X/ A[*Y[i:j],s]←Z;
              ≡
     /X∧(Y[i:j]≠n)/ A[n,s]←Z;

F2.  /X/ A[s,*Y[i:j]]←Z;
              ≡
     /X∧(Y[i:j]≠n)/ A[s,n]←Z;
```

```
F3.  /X/ A[*Y[i:j],s]←A[*Y[i:j],s];
              ≡
     /X∧(Y[i:j]≠n)/ A[n,s]←A[n,s];

F4.  /X/ A[s,*Y[i:j]]←A[s,*Y[i:j]];
              ≡
     /X∧(Y[i:j]≠n)/ A[s,n]←A[s,n];
```

Admittedly the "*" operator is a kluge designed to solve the problem of defining memories with explicit feedback. It would be very undesirable to actually expand a memory into several thousand statements using the above axioms. They can, however, be set up in a proof checker so that the user can ask if a given set of locations change or remain the same under specific circumstances. The "*" operator should only be used in context shown above. It is unclear what we would do with expressions containing several "*" terms combined with logic operators.

6. TRANSITION ALGEBRA

In this section we will develop some identities that permit us to study the effects of signal transitions as they travel thru a circuit. The concept of a transition algebra was first proposed by Talantsev [58] in 1958. Since then, these constructs have been applied to the problem of designing circuits using edge triggered flip-flops [53,54,57].

6.1 Basic axioms

Figure 6.1 illustrates the nature of the transition operators. The expression "$\uparrow X$" refers to that incredibly short period of time during which variable X changes from 0 to 1, and is best thought of as a pulse. The fact that constants do not undergo transitions is illustrated in C1. Axiom C2 tells us that a signal cannot be changing from 0 to 1 and from 1 to 0 at the same instant. It will simplify things a great deal if we assume that two variables cannot change at exactly the same time. This assumption is reflected in axioms C3 and C4. If a transition is thought of as a pulse, going from 0 to 1 and back to 0, then it is possible to think of that pulse as having two transitions associated with it. This permits us to use axiom C5 to reduce multiple transition operators to just one (the innermost).

C1. $\uparrow 0 \equiv 0$
$\uparrow 1 \equiv 0$
$\downarrow 0 \equiv 0$
$\downarrow 1 \equiv 0$

C2. $\uparrow X \wedge \downarrow X \equiv 0$

C3. $\uparrow X \wedge \uparrow Y \equiv 0$
$\uparrow X \wedge \downarrow Y \equiv 0$
$\downarrow X \wedge \uparrow Y \equiv 0$
$\downarrow X \wedge \downarrow Y \equiv 0$

C4. $\uparrow X \bullet \uparrow Y \equiv \uparrow X \vee \uparrow Y$
$\uparrow X \bullet \downarrow Y \equiv \uparrow X \vee \downarrow Y$
$\downarrow X \bullet \uparrow Y \equiv \downarrow X \vee \uparrow Y$
$\downarrow X \bullet \downarrow Y \equiv \downarrow X \vee \downarrow Y$

C5. $\uparrow\uparrow X \equiv \uparrow X$
$\uparrow\downarrow X \equiv \downarrow X$
$\downarrow\uparrow X \equiv \uparrow X$
$\downarrow\downarrow X \equiv \downarrow X$

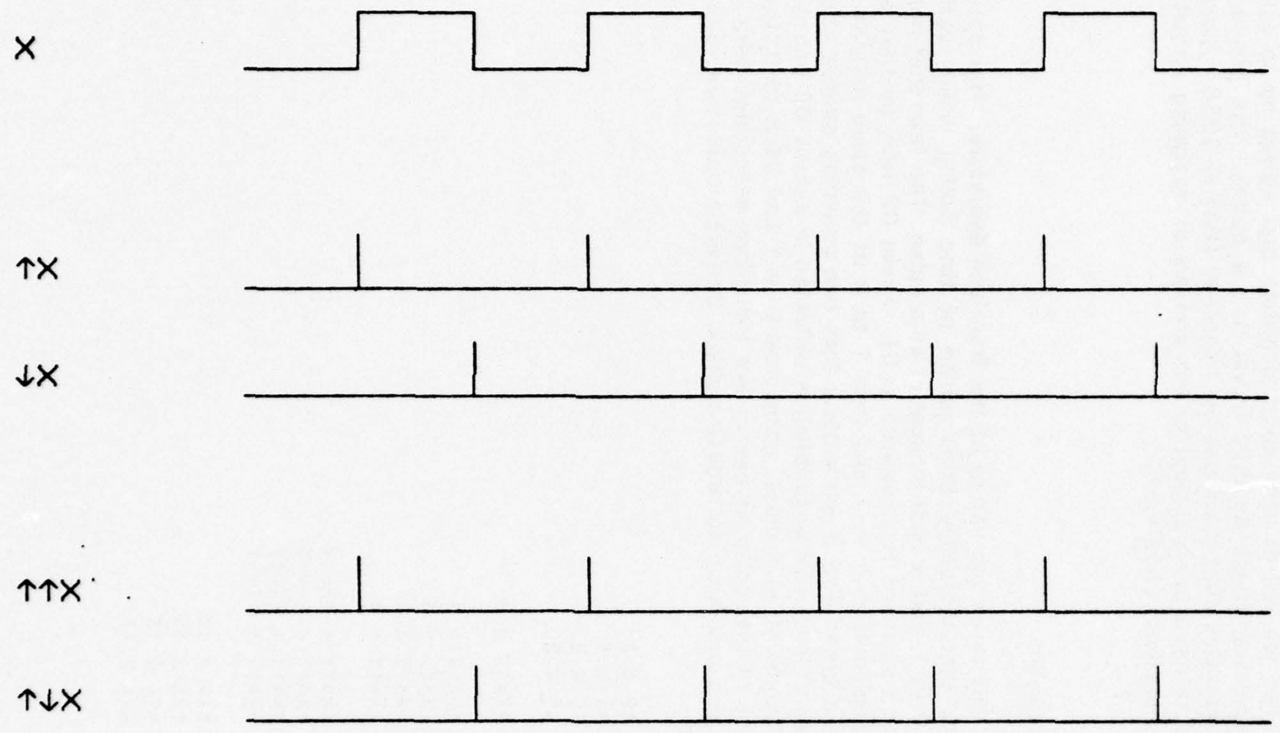


FIGURE 6.1 TRANSITION VARIABLES

6.2 Mathematical basis of transition algebra

This subsection contains the derivations that permit us to develop transition functions for switching circuits. To begin with, any variable in a logic function can be factored out by using Shannon's expansion theorem (very similar to M4). The resulting expression uses two new functions. They are the same as the original function except that the factored out variable is set to 1 in one of them and 0 in the other.

$$f(x_1,x_2,\ldots,x_n) = x_i f_{i1} \vee \bar{x}_i f_{i0} \qquad (1)$$

f_{i1} is $f(x_1,x_2,\ldots,x_n)$ with $x_i=1$

f_{i0} is $f(x_1,x_2,\ldots,x_n)$ with $x_i=0$

Whenever these two new functions can have complementary values (due to the values of other variables), a transition in the factored out variable will cause a transition in the original function. The direction of this transition depends on which of the new functions has the value of 0, which has the value 1, and the direction of the transition in the variable. The possible ways of obtaining positive or negative transitions are shown below.

$$\uparrow f = f_{i1}\bar{f}_{i0}\uparrow x_i \vee \bar{f}_{i1}f_{i0}\downarrow x_i \qquad (2)$$

$$\downarrow f = f_{i1}\bar{f}_{i0}\downarrow x_i \vee \bar{f}_{i1}f_{i0}\uparrow x_i \qquad (3)$$

To find all possible transitions for a given function it is necessary to develop expressions like (2) and (3) for every variable in the original function, and OR the results. In the equations below, the Sigma should be interpreted as a logical OR.

$$\uparrow f(x_1,x_2,\ldots,x_n) = \sum_{i=1}^{n} f_{i1}\bar{f}_{i0}\uparrow x_i \vee \bar{f}_{i1}f_{i0}\downarrow x_i \qquad (4)$$

$$\downarrow f(x_1,x_2,\ldots,x_n) = \sum_{i=1}^{n} f_{i1}\bar{f}_{i0}\downarrow x_i \vee \bar{f}_{i1}f_{i0}\uparrow x_i \qquad (5)$$

6.3 Transitions thru combinational circuits

Using (4) and (5) above we could derive transition functions for any combinational circuit. However, all we really need are the transition functions for AND, OR, and INVERT since other circuits can be defined in terms of these three. The transformations for AND and OR could be stated for arbitrary numbers of variables, but the same result will be obtained thru repeated application of the two variable versions. *Remember that X and Y can be any expression.*

C6. $\uparrow\neg X \equiv \downarrow X$
$\downarrow\neg X \equiv \uparrow X$

C7. $\uparrow(X\wedge Y) \equiv (\uparrow X\wedge Y)\vee(X\wedge\uparrow Y)$
$\downarrow(X\wedge Y) \equiv (\downarrow X\wedge Y)\vee(X\wedge\downarrow Y)$

C8. $\uparrow(X\vee Y) \equiv (\uparrow X\wedge\neg Y)\vee(\neg X\wedge\uparrow Y)$
$\downarrow(X\vee Y) \equiv (\downarrow X\wedge\neg Y)\vee(\neg X\wedge\downarrow Y)$

To show how the above transformations are applied to more complex circuits we have two examples. The circuit diagrams are shown in Figure 6.2. The example in Figure 6.2a is an exclusive-OR function.

$$\uparrow(X\bullet Y) = \uparrow((X\wedge\neg Y)\vee(\neg X\wedge Y)) =$$

$$\uparrow(X\wedge\neg Y)\wedge\neg(\neg X\wedge Y) \vee \neg(X\wedge\neg Y)\wedge\uparrow(\neg X\wedge Y) =$$

$$(\uparrow X\wedge\neg Y \vee X\wedge\downarrow Y)\wedge(X\vee\neg Y) \vee (\downarrow X\wedge Y \vee \neg X\wedge\uparrow Y)\wedge(\neg X\vee Y) =$$

$$(\uparrow X\wedge\neg Y\wedge X)\vee(\uparrow X\wedge\neg Y)\vee(X\wedge\downarrow Y)\vee(X\wedge\downarrow Y\wedge\neg Y)\vee$$
$$(\downarrow X\wedge Y\wedge\neg X)\vee(\downarrow X\wedge Y)\vee(\neg X\wedge\uparrow Y)\vee(\neg X\wedge\uparrow Y\wedge Y) =$$

$$(\uparrow X\wedge\neg Y)\vee(X\wedge\downarrow Y)\vee(\downarrow X\wedge Y)\vee(\neg X\wedge\uparrow Y)$$

Notice how terms having the same identifier as a transition variable and a non-transition variable, such as "$\uparrow X\wedge Y\wedge\neg X$", were eliminated by using T8. Unfortunately, this does not always happen. Watch what happens when we derive the transition function for the circuit in Figure 6.2b.

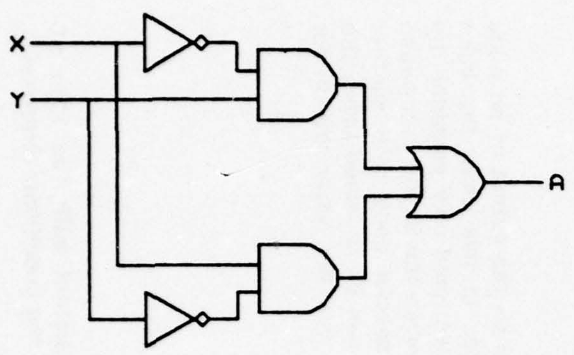
$$\uparrow(X\wedge Y \vee \neg X\wedge Z) =$$

$$\uparrow(X\wedge Y)\wedge\neg(\neg X\wedge Z) \vee \uparrow(\neg X\wedge Z)\wedge\neg(X\wedge Y) =$$

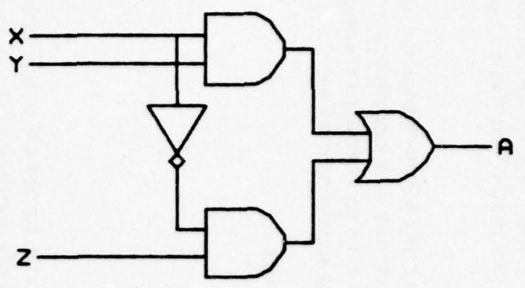
$$(\uparrow X\wedge Y \vee X\wedge\uparrow Y)\wedge(X \vee \neg Z) \vee (\downarrow X\wedge Z \vee \neg X\wedge\uparrow Z)\wedge(\neg X \vee \neg Y) \equiv$$

$$(\uparrow X\wedge Y\wedge X)\vee(\uparrow X\wedge Y\wedge\neg Z)\vee(X\wedge\uparrow Y)\vee(X\wedge\uparrow Y\wedge\neg Z)\vee$$
$$(\downarrow X\wedge Z\wedge\neg X)\vee(\downarrow X\wedge Z\wedge\neg Y)\vee(\neg X\wedge\uparrow Z)\vee(\neg X\wedge\uparrow Z\wedge\neg Y) =$$

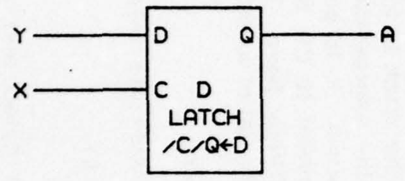
$$(\uparrow X\wedge X\wedge Y)\vee(\uparrow X\wedge Y\wedge\neg Z)\vee(X\wedge\uparrow Y)\vee(\downarrow X\wedge\neg X\wedge Z)\vee(\downarrow X\wedge\neg Y\wedge Z)\vee(\neg X\wedge\uparrow Z)$$



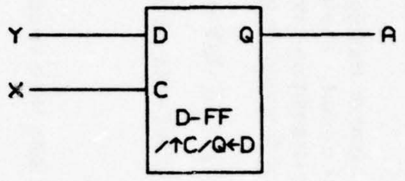
A) $\uparrow A = \uparrow(X \wedge \neg Y \vee \neg X \wedge Y) =$
$(\uparrow X \wedge Y) \vee (X \wedge \downarrow Y) \vee (\downarrow X \wedge Y) \vee (\neg X \wedge \uparrow Y)$



B) $\uparrow A = \uparrow(X \wedge Y \vee \neg X \wedge Z) =$
$(\uparrow X \wedge X \wedge Y) \vee (\uparrow X \wedge Y \wedge \neg Z) \vee (X \wedge \uparrow Y) \vee$
$(\downarrow X \wedge \neg X \wedge Z) \vee (\downarrow X \wedge \neg Y \wedge Z) \vee (\neg X \wedge \uparrow Z)$



C) $/X/\ A \leftarrow Y;$
$\uparrow A = (X \wedge \uparrow Y) \vee (\neg A \wedge Y \wedge \uparrow X)$



D) $/\uparrow X/\ A \leftarrow Y;$
$\uparrow A = (\neg A \wedge Y \wedge \uparrow X)$

FIGURE 6.2 TRANSITION EXAMPLES

Elements such as "$\uparrow X \wedge X \ldots$" indicate the possibility of a hazard. It can be shown (Appendix C) that any circuit having a hazard will have this sort of expression. The reverse is not true, and terms of this type are still present when the hazards have been corrected by consensus gates.

Another problem of this sort arises when trying to expand a non-transition thru a combinational circuit. The resulting terms seem to indicate the possibility of a hazard in a non-transition. We could add some axioms to make these extraneous terms vanish but it would no longer be true that $\neg(\neg\uparrow X)$ is equivalent to $\uparrow X$.

$$\neg\uparrow(X\wedge Y) = \neg(\uparrow X\wedge Y \vee X\wedge\uparrow Y) = (\neg\uparrow X\vee\neg Y)\wedge(\neg X\vee\neg\uparrow Y) =$$

$$(\neg\uparrow X\wedge\neg X)\vee(\neg\uparrow X\wedge\neg\uparrow Y)\vee(\neg X\wedge\neg Y)\vee(\neg\uparrow Y\wedge\neg Y)$$

6.4 Transitions thru sequential circuits

The last transformation in this group demonstrates how to obtain a transition at the output of a sequential component. Axiom C9 is easily explained if we think of a latch like the one in Figure 6.2c, in which the output follows the input as long as the clock is true. A transition will develop on the output of the latch if the data input undergoes a transition while the clock input is held true. Another way to get a transition is for the clock input to switch to true while the data inputs are set to switch the latch to a new state.

C9. $/X/\ A\leftarrow Y;$

$$\supset$$

$$\uparrow A \equiv (X\wedge\uparrow Y)\vee(\neg A\wedge Y\wedge\uparrow X)$$
$$\downarrow A \equiv (X\wedge\downarrow Y)\vee(A\wedge\neg Y\wedge\uparrow X)$$

This becomes even simpler if we want a transition on the output of an edge triggered component like the flip-flop in Figure 6.2d. In this case the input changing while the clock is true does not apply, so we need only consider the second part of the expression: the clock going true when the inputs can cause an output transition. It is possible to handle this special case with another transformation similar to C9, but the desired results can be obtained using the identities we already have. This particular problem was the major reason for including C5 in the list of axioms.

$/\uparrow X/\ A\leftarrow Y;$	given
$\uparrow A = (\uparrow X\wedge\uparrow Y)\vee(\uparrow\uparrow X\wedge Y\wedge\neg A) =$	C9
$(\uparrow X\wedge Y\wedge\neg A)$	C3,C5

It should be noted that this is completely consistent with the idea of expressing combinational circuits as statements where the conditional expression is always true.

```
/1/ A←Y;                           given
↑A = (1∧↑Y)∨(↑1∧Y∧¬A) =            C9
↑Y                                 C1
```

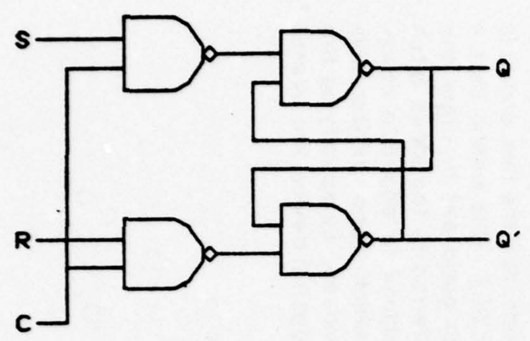
Other problems can develop if the transition variable being expanded is controlled by several statements. Generally speaking, the correct approach would be to apply the transition identities to each of the statements and then OR the results together. This will work unless some of the statements are contradictory, which is an error anyway. Contradictory statements are discussed in the section on error conditions.

6.5 Flip-flop design and modelling

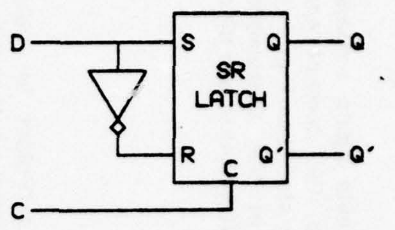
The transformations presented thus far are not sufficient to prove the correctness of some flip-flop designs. Simple latches such as those in Figure 6.3a and Figure 6.3b are easily verified. On the other hand, edge triggered devices like the ones in Figure 6.3c and Figure 6.3d cannot be verified using these methods because they depend on hazards being carefully adjusted to work correctly. The type-D flip-flop shown in Figure 6.3c, for example, will work correctly only if the delay thru the first latch is longer than the delay thru the inverter connecting the two clock inputs. The manufacturer will bias these delays to insure correct operation. Additionally, many circuits use capacitance instead of feedback to store data. For these reasons we will not attempt to verify the correctness of flip-flop designs. If a given flip-flop circuit is known to work correctly it can be added to the verification scheme as an axiom.

It is important that components be modelled carefully if the proof of correctness is to be valid. Some JK master-slave flip-flops, like the one drawn in Figure 6.3d, have the nasty habit of ones-catching [55,56]. This means that a short pulse on the J or K input while the clock is high could set or clear the master latch. This erroneous data would then be transferred to the slave latch during the clock transition. Many JK flip-flops are designed to eliminate ones-catching. At any rate, when using a JK flip-flop which does exhibit this characteristic it should be modelled as two sequential devices. By specifying the master part as a latch and the slave part as an edge triggered device the overall behavior can be properly described.

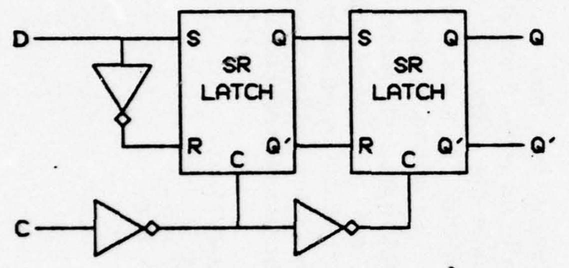
```
/C∧J∧¬Q/ X←1;
/C∧K∧ Q/ X←0;
/ ↓C/ Q←X;
/¬↓C/ Q←Q;
```



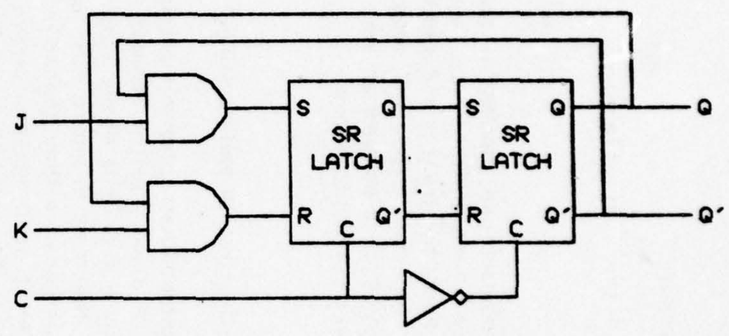
A) SET/RESET LATCH



B) TYPE-D LATCH



C) D MASTER SLAVE FLIP-FLOP .



D) JK MASTER SLAVE FLIP-FLOP

FIGURE 6.3 FLIP-FLOP DESIGNS

7. ERROR CONDITIONS

In this section we shall explore a variety of error conditions, such as races, hazards, and oscillations, that can be detected in the hardware descriptions. It is not completely clear how this fits in with the idea of comparing two descriptions of the same device since it is possible to include error conditions in higher level system specifications. If the user's system description includes a hazard, for example, then the actual circuit may need to have that hazard for the verifier to prove its correctness. Because of this conflict this section will not contain specific rules for what to do when an error is encountered. It is mainly intended to show under what conditions an error can exist, and why a verifier may have trouble proving the correctness of some circuits.

7.1 Race and hazard jargon

Now we come to the question of timing. To show correctness it is not really necessary to know specific delay times or to make any assumptions on how fast the device will go. Usually a device built from ECL will go faster than one built from MOS, but both can be logically correct. Nevertheless, it is possible to develop ways to detect timing anomalies such as races and hazards on an algebraic level without introducing specific delays.

Static and dynamic hazards [61,62,65,66,68,69,70,73,74,75] are logical conditions under which combinational circuits can produce spurious transitions. The actual transitions, created by a logic hazard and specific component delays, are called hazard pulses. Some authors use the term static-1 hazard to refer to a momentary 0 in an output that will normally be 1, and static-0 hazard to refer to a momentary 1 result. A typical example of a static-1 hazard was presented in the section on transition algebra. The term transient hazard is sometimes used to refer to a hazard that will not do any damage, while a steady state hazard can affect the final value of a state variable. A dynamic hazard is a momentary 0 and a momentary 1 occuring on an output as it changes from one state to the other. In other words, a dynamic hazard causes three transitions where only one was expected.

Some papers [59,60,63] have attacked the problem of detecting hazard conditions for specific multiple input changes. Such hazards are often refered to as M-hazards. These hazards are further divided into function hazards and logic hazards. A function hazard is inherent in the definition of the logic, while a logic hazard may exist in a particular implementation of a circuit which could otherwise be constructed without hazards. Although the problem of multiple input hazards will not be addressed in this paper, the concepts of function and logic hazards are closely related to the problem of comparing hazardous system specifications with their implementation.

An essential hazard [61,65,67,69,71,74,75] is a unique set of conditions in which the transition of an input variable causes a transition in a state variable, and the comparative delays in these transitions can determine the output of still another state variable. By Unger's definition [74] a circuit contains an essential hazard if the result after one input transition is different than the result after three input transitions. Essential hazards are not necessarily bad, and are required in counting circuits. The circuit delays must be adjusted to make sure the desired effect is achieved. A similar condition is a race [64,67,69,74,75], except that a race involves transitions of two state variables. A critical race is a race that can affect the final state of the circuit (like a steady state hazard).

Race and hazard terminology will be used somewhat differently in this paper as a matter of convenience. Since static and dynamic hazards can be thought of as a "race" between a variable and its complement (in a combinational circuit), and an essential hazard can be thought of as a "race" between an input variable and a state variable, all of these phenomena will often be refered to as races. This eliminates the problem of having to determine which variables are state variables when one of these problems is detected.

7.2 Race and hazard detection

For purposes of hardware verification we are only interested in races that can cause a permanent change in the state of the system and hazard pulses on the outputs of a device that can cause a malfunction when the circuit is used in conjunction with other equipment. Output hazards tend to take care of themselves. If the output is from state variable an output hazard will appear as a race that can affect its final state. If the output is from a combinational circuit the hazard will usually show up in the boolean expression for that circuit.

Figure 7.1a is a block diagram of a race condition. A transition in variable X passes thru circuits A and B, which might be combinational, sequential, or just wires. If the final state of circuit C can depend on whether path A is faster than path B, then we have race condition. After using the transition theorems on the circuit the statements describing circuit C will be in terms of the input variable X and the state variables (if any) of A and B. Races will then be apparent in one of the following forms:

1. A static-0 hazard in a conditional expression.

```
/...static-0 hazard.../ A←X;
```

2. *Any static or dynamic hazard in a transition variable.*

```
/...↑(static hazard).../ A←X;
/...↓(static hazard).../ A←X;
/...↑(dynamic hazard).../ A←X;
/...↓(dynamic hazard).../ A←X;
```

3. X or ¬X ANDed with ↑X or ↓X in the conditional expression.

```
/...X∧↑X.../ A←Y;
/...X∧↓X.../ A←Y;
/..¬X∧↑X.../ A←Y;
/..¬X∧↓X.../ A←Y;
```

4. X, ¬X, ↑X or ↓X in the conditional expression with X or ¬X in the register transfer part.

```
/...X.../ A←...X...;
/..¬X.../ A←...X...;
/..↑X.../ A←...X...;
/..↓X.../ A←...X...;

/...X.../ A←...¬X...;
/..¬X.../ A←...¬X...;
/..↑X.../ A←...¬X...;
/..↓X.../ A←...¬X...;
```

Some of these conditions may not mean disaster. A dynamic hazard in a conditional expression does not matter if we are merely gating some data into a register. On the other hand, if something else can happen on the opposite transition, or the register transfer part involves counting or shifting, then it can be very important.

Although we have freely indicated where static and dynamic hazards can cause problems, we have not said anything about how to detect them. Static hazards are well understood, but dynamic hazards are extremely difficult to detect. There are, however, some fairly simple conditions under which static and dynamic hazards cannot exist [65,66,68,69,70,75]. This is not to say that they necessarily will exist if these conditions are not met, but these methods are easy to implement and will probably be adequate for a verification program. A very simple rule would be to say that hazards cannot exist unless a variable appears both complemented and uncomplemented in the same expression.

It should be noted that static hazards in transition expressions will become apparent when the expression is expanded. This will generate two subexpressions of the form "↑X∧X" and "↓X∧¬X" (or "↓X∧X" and "↑X∧¬X"). Adding a consensus gate will still produce two subexpressions of this form, but they will be in terms of two different variables such as "↑Y∧¬Y" and "↑Z∧¬Z". It may be possible to develop some specific rules based on this, but for the time being it is probably easier to expand the minimal transition expression and then look for a consensus gate when a possible hazard is detected. Some results of expanding transition expressions containing static hazards are shown in Appendix C.

When a race is found a verification program may be able to indicate which portion of the circuit must be faster to insure correct operation. Another approach would be to have the user indicate which branch is faster and let the verifier determine if the circuit is correct under that assumption. Eventually it may be possible to include statistical information on component delays so that hardware verifier can determine the probability of a race or hazard.

7.3 Oscillation

Another timing anomaly, oscillation, can also be detected thru careful examination of the circuit description. Oscillation can occur by a variable being repeatedly complemented or by a continuous bit rotation. Figures 7.1b and 7.1c illustrate circuits where variable Q will be complemented as long as X is true. The only difference between the gate circuit and the latch circuit is that the latch will remain in some random state after X goes low. Figure 7.1d shows a rotate circuit that will keep rotating as long as the CLK signal is true.

Although the illustrations show very simple circuits, an oscillation can occur thru arbitrary amounts of combinational and sequential logic. The only requirement is that a variable is repeatedly complemented, or that two or more state variables are exchanged or rotated continuously. In the latter case the variables can be complemented any number of times during the rotation. Here is an example of oscillation involving a few statements:

```
/X/ A←B;
/Y/ B←C∧D;
/Z/ C←¬A;
```

In this example, oscillation can occur only if X,Y,Z, and D are all true at the same time. A simple algorithm for detecting oscillation in a given variable would be to determine the conditions under which each of the variables in the register transfer expression are transferred into that variable. Then do the same thing for each of these variables, and so on until the original variable or its complement is detected, or until the process terminates. An oscillation will be indicated anytime a variable can be continuously complemented, or when a variable is rotated (complemented or not) thru at least one other state variable.

This algorithm may not be practical in that it can grow incredibly large trees and consume lots of computer time, but it is hoped that most branches will be pruned very quickly. Anytime the process reaches a variable that is controlled by a transition it can eliminate that variable as a possible oscillation path. Branches can also be eliminated when they require conditions that contradict other conditions farther up the tree. Below are two variations on the previous example in which an oscillation will not occur. In the first we have replaced Y with ↓Y, and in the second oscillation is prevented because X and ¬X would both have to be true at the same time.

```
/X/ A←B;
/↓Y/ B←C∧D;
/Z/ C←¬A;
```

```
/X/ A←B;
/¬X/ B←C∧D;
/Z/ C←¬A;
```

7.4 Other error conditions

A more obvious problem for a hardware verifier is that of contradictory statements. By comparing statements which can change the value of the same variable one of three relationships will be discovered. The conditional expressions can be the same, they can be different but not exclusive, or they can be exclusive. An example of each is shown below.

```
1. /X/ A←Y;
   /X/ A←Z;

2. /W/ A←Y;
   /X/ A←Z;

3. /V∧X/ A←Y;
   /W∧¬X/ A←Z;
```

The first example is undoubtedly an error since the variable must be set to the results of two different expressions for the same conditions. The second case will be incorrect only if the circuit using this device can cause W and X to the true at the same time. This sort of problem can be detected by verifying the still larger circuit in which this device is used. If this larger circuit includes such things as an operator's console then there is no way to tell if the operator will push the wrong two buttons at the same time. The third condition is not problem, and can be reduced to a combinational circuit (using M3) if V and W are identical.

Many types of integrated circuit devices permit outputs to be connected together in a wired-AND or a wired-OR arrangement. An easy way to avoid getting into problems with contradictory statements on these circuits would be to include the appropriate AND or OR function in the hardware description as if there were additional gates in the circuit.

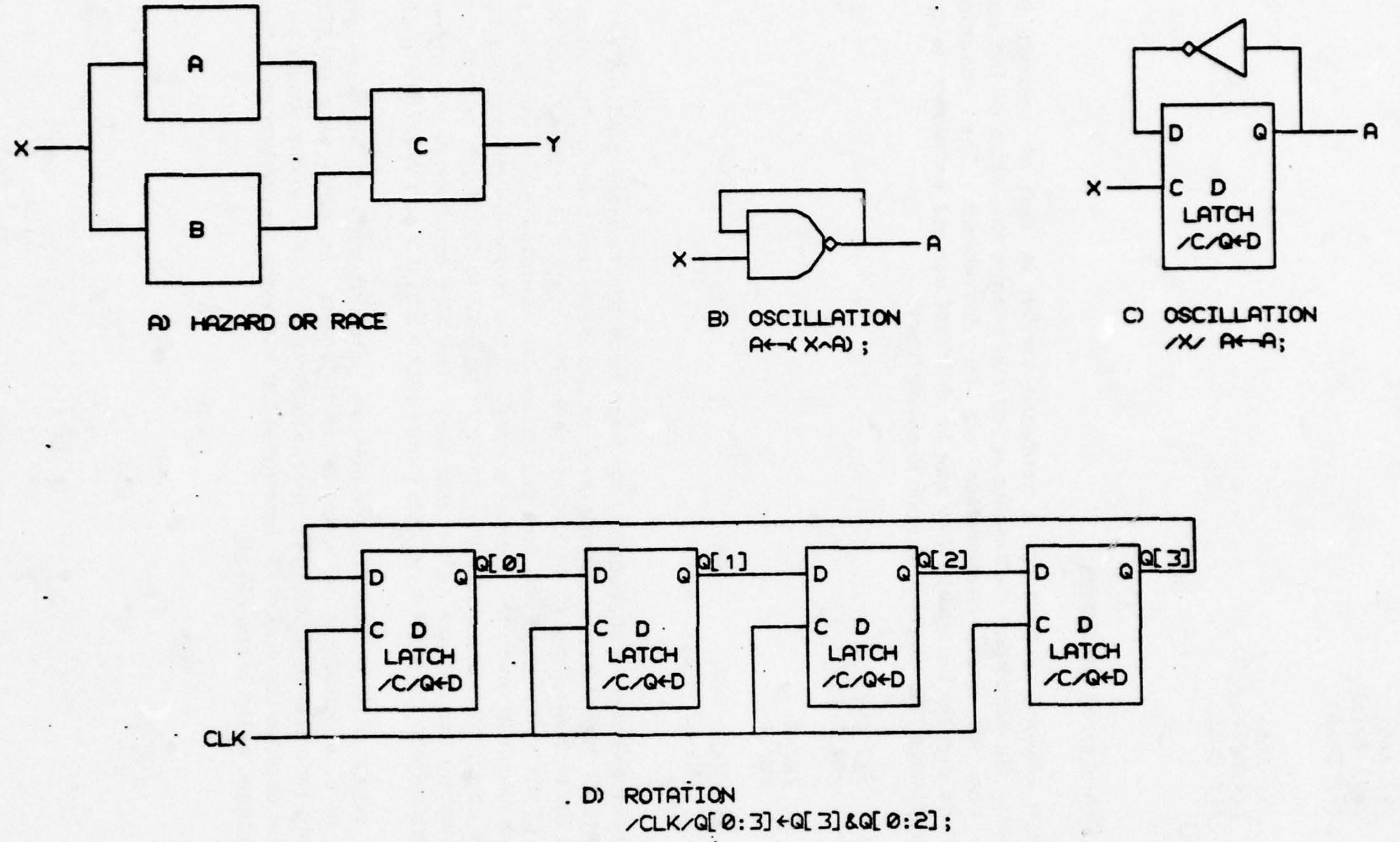


FIGURE 7.1 RACES AND OSCILLATIONS

8. MICROPROGAM VERIFICATION

This section on microprogram verification has been included for several reasons. To begin with, there are some interesting similarities between the problems of microcode verification and hardware verification. A hardware verification system should be able to handle descriptions of microprogrammed devices. Also, it should be possible to interface circuit descriptions in a hardware verification language to existing microprogram verifiers to determine the correctness of microprograms for that particular device.

8.1 Microprogram vs. program verification

Although work on program verification has been going on for about ten years, the specific problem of microprogram verification has only been discussed fairly recently. Much of this work was done at IBM using a hypothetical computer called the S-machine [77,78,81,82,83]. The S-machine is a very simple stack machine and problems involving the control structure and timing constraints were not considered.

The distinction between microprogram verification and program verification is often very hazy. One major difference is that microprogram verification concerns a program for a precisely defined piece of hardware instead of using generalized algebraic language. Details like word length and register structure are very important in microprogram verification. Additionally, each microprogram instruction usually specifies several internal operations that may happen simultaneously.

Microprograms are usually loop free or have very few loops. This makes the verification problem much simpler than for programs in general. The problem of verifying loopfree microcode is discussed in [85]. One other observation is that the desired operation of a microprogram is usually described as an algorithm rather than as a result. In program verification, for example, a sorting program might be specified by saying that it takes an array of numbers and returns them in numerical order. A microprogram verification problem, on the other hand, will usually involve showing that a specific algorithm is followed by a detailed program. For this reason most microprogram verifiers are based on proving that each statement in the algorithm description is satisfied by one or more microinstructions. For more background on microprogram verification techniques see [80,82].

Of particular interest is the STRUM (STRUctured Microprogramming language) system developed at UCLA[84]. This system uses a very popular procedural register transfer language, ISP (Instruction Set Processor), for describing the hardware to be microprogrammed. It also uses generalized program verification techniques rather than methods specifically tailored to microprogram problems. This means that loops cause fewer problems with STRUM than they do with some other microprogram verifiers.

8.2 Hardware verification for microcoded devices

Since the hardware verification language described here is non-procedural it is also loopfree. Microcoded devices can be specified in several ways. To define a circuit that fetches and executes instructions pointed to by a given register (the program counter) is fairly straightforward. Thus it is simple to prove that the hardware does the correct thing for an unspecified microprogram.

Verifying that a circuit with a specific microprogram is correct can be handled in one of two ways. Assuming that the microprogram is stored in a read-only memory it is possible to include the ROM data in the conditional parts of the overall design specification. This would result in a hardware description with statements like:

```
/(PC[0:15]=1000)∧↑T0/ A←X;
/(PC[0:15]=1001)∧↑T0/ B←Y;
/(PC[0:15]=1002)∧↑T0/ C←Z;
                 .
                 .
                 .
```

Another approach would be to describe the ROM as a combinational circuit in the higher level specifications. Although very cumbersome, this may make the hardware description more readable in those cases where the designer is actually using the ROM in place of several other combinational circuits. Facilities could be added to a hardware verifier to permit the user to specify the contents of a ROM as a bit table, combinational equations, or both.

Embedding the microprogram in the hardware description is only practical for very small control memories since the entire program must be included in the higher level description. For larger microprograms it would be much more reasonable to use other microprogram verification techniques. This would involve converting the non-procedural hardware description to a procedural one by making assertions about sequences of control signals. At the very least it would be necessary to indicate that certain clock inputs change continuously in a specific sequence and that the hardware has a well defined mechanism for determining the next instruction, such as a program counter. Adapting the hardware language used in this paper to program verification systems would be an interesting future project.

9. HARDWARE VERIFICATION USING FOL

In this section we will discuss how the FOL (First Order Logic) proof checker [86,87,88,89] can be used to prove the correctness of circuits. FOL is a manual proof checker, written in LISP, which permits the user to manipulate assumptions (component definitions) using a set of axioms and some simple commands.

9.1 Syntax modifications

Since the FOL proof checker is designed to work with formal logic it is necessary to convert the hardware language into the format of well-formed formulas (WFFs). Although the conditional expression can be related to the register transfer part by a logical implies, FOL has no understanding of dynamically changing variables. In other words, FOL cannot model register transfers. The easiest way out is to convert statements in the language used in this paper to a function of three arguments:

```
/X/ A←Y;      becomes    F(X,A,Y)

A←Y;          becomes    F(1,A,Y)
```

This way the axioms can be applied to these functions, and the results can be converted back into the hardware language on output (at the time of this writting the input conversion was being done manually and the output conversion had been implemented inside FOL).

Another minor problem involves the FOL character set. Certain logical symbols including ∧,∨,¬,≡,=, and ⊃ have special meanings in FOL. The logical AND operator, for instance, is used to combine WFFs and is somewhat different than the bitwise AND used in circuit design. To make this more clear we have the FOL representation of axiom M6, which requires that the first statement AND the second statement be true for the result to be valid.

```
M6.  /X/ A←Y;
     /X/ B←Z;
         ≡
     /X/ A&B←Y&Z;

becomes

AXIOM M6: ∀x a y b z. (F(x,a,y)∧F(x,b,z)≡F(x,a&b,y&z));;
```

The other logical connectives have the obvious meanings except for the equals operator (=). Equals works just like equivalence (≡) except that it can be used for expressions while equivalence can only be used with WFFs. Transformation T7 might look like:

```
AXIOM T7: ∀x y. x∩y=y∩x;;
```

Because the standard logical connectives are already in use, the following special symbols will be used as boolean connectives when we want a bitwise operator:

~	complement
∩	and
∪	or
⇔	equals

The FOL declarations and axioms that will be used in the examples are shown in Appendix D, and the FOL users manual [89] can explain what the syntax means. Three special functions have been provided: carry, sub, and suc. The carry function can be used in component definitions and manipulated by axioms such as A2 and A4. Sub is a function that provides a way to handle subscripts. Only the first subscript is provided for at this time since the examples do not use memories.

Q[1]	becomes	sub(Q,1,1)
Q[0:3]	becomes	sub(Q,0,3)

The successor function suc is attached to the LISP function ADD1. To illustrate its use we have the FOL version of definition D3:

```
AXIOM D3: ∀x i j k. sub(x,i,j)&sub(x,suc(j),k)=sub(x,i,k);;
```

9.2 FOL commands

FOL has a wealth of very powerful commands, but only a few are actually used in the examples that follow. How to define AXIOMs has already been amply illustrated, but there is a slight problem with axiom names. When an axiom has more than one part, like most of the T series and C series, FOL will try to append the numbers 1, 2, etc. to the axiom name for each of the different versions. This means that if we try to define both parts of T1 using only this name we will get axioms T11 and T12. This can cause problems if there are other axioms with these names. A simple solution is to give the axioms names like T1A and T1B, as is shown in Appendix D.

Variable names are declared using the DECLARE command and are set to the type INDCONST (individual constant) for signal and register names. The ASSUME command can be used to input component definitions. Assumptions are really just like axioms except that axiom names do not appear in the list of dependencies for a given proof step. Each proof step has a number or label appearing in parentheses to the left. The dependencies for each step appear in parentheses on the right. Assumptions depend on themselves.

The ∧I (AND introduction) command can combine two WFFs, and will often be used to put two statements together so that M6 can be used. Similarly, ∧E (AND elimination) can separate two WFFs. One way to generate a copy of an axiom with the correct variable names substituted is with the ∀E (FORALL elimination) command. Below is an example of this command applied to axiom D3 above.

```
∀E D3 Q 0 1 3;

gives the result

# sub(Q,0,1)&sub(Q,suc(1),3)=sub(Q,0,3)
```

The SIMPLIFY command will cause the "suc(1)" term to be evaluated and replaced by the number 2. To substitute the right side of the above equation for the left side in another statement the SUBSTR command is used. The other substitution command, SUBST, works the same way except that the left side of the expression above would be substituted for the right side. SUBSTR and SUBST work for both equals and equivalence.

One of the most powerful FOL instructions is the TAUT (tautology) command. It uses a parallel simulation routine to determine if some WFF logically follows from the axioms and assumptions. The WFF can include other WFFs connected by logical operators. For our purposes the TAUT command will be used when a ∀E produces a step containing an implies connective.

Having to use a ∀E and a substitute command every time we want to apply an axiom can become very tiresome. Fortunately, FOL has a REWRITE command that will apply axioms to expressions automatically. Since REWRITE will only use axioms in a left to right direction it will sometimes be useful to have reversed copies of some axioms (see T10B and T10BR in Appendix D). REWRITE will use the axioms repeatedly until no more substitutions can be made. For this reason the REWRITE command cannot be used with axioms like the commutative relation in T7. A REWRITE using this axiom will result in an infinite loop.

The LOGICTREE operator can be used to link several axioms together so they can be used by the REWRITE command. For the examples in the following sections we will use three axiom sets linked by LOGICTREE (see Appendix D). The REDUCE group will automatically apply axioms T1 thru T6, and to help with the commutative problem these axioms have been defined in both permutations. The TRANS group contains most of the clock transition reductions. A CONCATENATE group has been included that can reduce concatenated terms just as if simplified copies of D3 were applied repeatedly.

The only time FOL is really inconvenient is when two variables that can be combined in some way are separated by parentheses or need to be commuted before they can be reduced. Simplifying expressions like

```
(X∩Y)∩~X
```

can take several steps. It is possible to work out the transformation once and then use the ∀I (FORALL introduction) command to put the quantifiers back in. By manipulating the axioms that would be used to simplify the expression above and then using ∀I we can obtain the following:

```
# ∀x y. (x∩y)∩~x=0
```

This result can then be used with the REWRITE command whenever an expression of this sort appears. In hardware problems, where there is a tremendous amount of redundancy from one bit to the next, this can be a terrific timesaver.

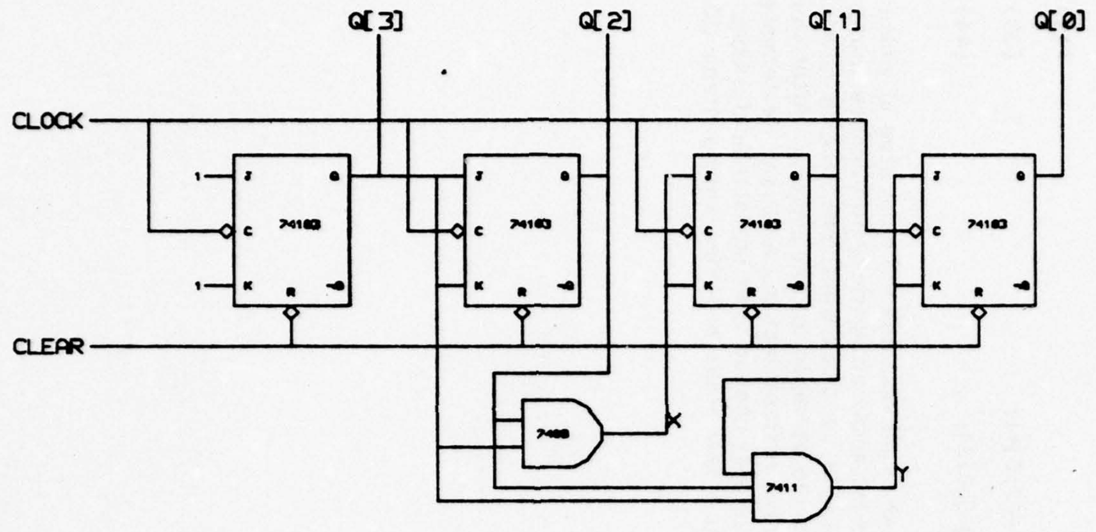
FOL also provides a substantial number of administrative features including some fairly involved file handling. A backup file of the user's typed input is kept in case the system crashes or a terrible error is made that cannot be remedied by cancelling proof steps. The CANCEL command deletes steps only from the end of the proof. Other commands let the user look at specific steps and axioms already in the proof. To reduce typing there are ways to refer to a specific expression in a given step. The second expression in step 127, for example, would be obtained when the user types 127:#2. The proofs in the following sections were printed out by the FOL print routines, which often substituted the actual expressions where the author only typed in a couple of numbers.

10. SYNCHRONOUS COUNTER

As our first example of a circuit being verified by FOL we will use the synchronous counter shown in Figure 10.1. To make the proof a little simpler we will use a JK flip-flop that does not exhibit ones-catching. Definitions for the components used in these examples are in Appendix E. The desired goal is to prove the following:

```
/~CLEAR/ Q[0:3]←0;                                    (19)

/CLEAR∩↓CLOCK/ Q[0:3]←Q[0:3]+1;                       (39)

/CLEAR∩~↓CLOCK/ Q[0:3]←Q[0:3];                        (44)
```

The number in parentheses on the right indicates the proof step at which that result was achieved. The proof is incredibly straightforward because the circuit does not do anything complicated with the clock transitions. Notice how steps 20-23 are used to create a commuted definition for exclusive-or. After working on the problem for a while the author realized that the exclusive-or terms were coming out reversed relative to what was needed to fit the increment axiom. The problem was solved by cancelling a few steps and then adding steps 20-23. Only the final result is shown here.



```
/¬CLEAR/ Q[0:3]←0;
/CLEAR∧↓CLOCK/ Q[0:3]←Q[0:3]+1;
/CLEAR∧¬↓CLOCK/ Q[0:3]←Q[0:3];
```

FIGURE 10.1 SYNCHRONOUS COUNTER

```
*****DECLARE INDCONST CLEAR,CLOCK,Q,X,Y;
*****ASSUME  /~CLEAR/Q[3]←0; ;
1  /~CLEAR/Q[3]←0;   (1)
*****ASSUME  /CLEAR∩↓CLOCK/Q[3]←(1∩~Q[3])∪(~1∩Q[3]); ;
2  /CLEAR∩↓CLOCK/Q[3]←(1∩~Q[3])∪(~1∩Q[3]);   (2)
*****ASSUME  /CLEAR∩~↓CLOCK/Q[3]←Q[3]; ;
3  /CLEAR∩~↓CLOCK/Q[3]←Q[3];   (3)
*****ASSUME  /~CLEAR/Q[2]←0; ;
4  /~CLEAR/Q[2]←0;   (4)
*****ASSUME  /CLEAR∩↓CLOCK/Q[2]←(Q[3]∩~Q[2])∪(~Q[3]∩Q[2]); ;
5  /CLEAR∩↓CLOCK/Q[2]←(Q[3]∩~Q[2])∪(~Q[3]∩Q[2]);   (5)
*****ASSUME  /CLEAR∩~↓CLOCK/Q[2]←Q[2]; ;
6  /CLEAR∩~↓CLOCK/Q[2]←Q[2];   (6)
*****ASSUME  /~CLEAR/Q[1]←0; ;
7  /~CLEAR/Q[1]←0;   (7)
*****ASSUME  /CLEAR∩↓CLOCK/Q[1]←(X∩~Q[1])∪(~X∩Q[1]); ;
8  /CLEAR∩↓CLOCK/Q[1]←(X∩~Q[1])∪(~X∩Q[1]);   (8)
*****ASSUME  /CLEAR∩~↓CLOCK/Q[1]←Q[1]; ;
9  /CLEAR∩~↓CLOCK/Q[1]←Q[1];   (9)
*****ASSUME  /~CLEAR/Q[0]←0; ;
10  /~CLEAR/Q[0]←0;   (10)
*****ASSUME  /CLEAR∩↓CLOCK/Q[0]←(Y∩~Q[0])∪(~Y∩Q[0]); ;
11  /CLEAR∩↓CLOCK/Q[0]←(Y∩~Q[0])∪(~Y∩Q[0]);   (11)
*****ASSUME  /CLEAR∩~↓CLOCK/Q[0]←Q[0]; ;
12  /CLEAR∩~↓CLOCK/Q[0]←Q[0];   (12)
*****ASSUME X←Q[2]∩Q[3]; ;
13 X←Q[2]∩Q[3];   (13)
*****ASSUME Y←(Q[1]∩Q[2])∩Q[3]; ;
14 Y←(Q[1]∩Q[2])∩Q[3];   (14)
```

```
*****∧I (10 7);

15  /~CLEAR/Q[0]←0; ∧ /~CLEAR/Q[1]←0;   (7 10)

*****∧I (15 4);

16 ( /~CLEAR/Q[0]←0; ∧ /~CLEAR/Q[1]←0; )∧ /~CLEAR/Q[2]←0;   (4 7 10)

*****∧I (16 1);

17 (( /~CLEAR/Q[0]←0; ∧ /~CLEAR/Q[1]←0; )∧ /~CLEAR/Q[2]←0; )∧ /~CLEAR/%
Q[3]←0;   (1 4 7 10)

*****REWRITE 17 BY { M6};

18  /~CLEAR/((Q[0]&Q[1])&Q[2])&Q[3]←((0&0)&0)&0;   (1 4 7 10)

*****REWRITE 18 BY CONCATENATE;

19  /~CLEAR/Q[0:3]←((0&0)&0)&0;   (1 4 7 10)

*****VE X1 x,y;

20 ((x∩~y)∪(~x∩y))=(x•y)

*****VE X6 x,y;

21 (x•y)=(y•x)

*****SUBSTR 21 IN 20;

22 ((x∩~y)∪(~x∩y))=(y•x)

*****VI 22 x y;

23 Vx y.((x∩~y)∪(~x∩y))=(y•x)

*****REWRITE 2 BY REDUCE;

24  /CLEAR∩↓CLOCK/Q[3]←~Q[3];   (2)

*****REWRITE 5 BY { 23};

25  /CLEAR∩↓CLOCK/Q[2]←Q[2]•Q[3];   (5)

*****REWRITE 8 BY { 23};

26  /CLEAR∩↓CLOCK/Q[1]←Q[1]•X;   (8)

*****VE M2 X,Q[2]∩Q[3];

27 X←Q[2]∩Q[3]; ⊃X=(Q[2]∩Q[3])

*****TAUT X=(Q[2]∩Q[3]) 13,27;

28 X=(Q[2]∩Q[3])  (13)

*****SUBSTR 28 IN 26;
```

```
29  /CLEAR∩↓CLOCK/Q[1]←Q[1]⊕(Q[2]∩Q[3]);   (8 13)

*****REWRITE 11 BY { 23};

30  /CLEAR∩↓CLOCK/Q[0]←Q[0]⊕Y;   (11)

*****∀E M2 Y,(Q[1]∩Q[2])∩Q[3];

31 Y←(Q[1]∩Q[2])∩Q[3]; ⊃Y=((Q[1]∩Q[2])∩Q[3])

*****TAUT Y=((Q[1]∩Q[2])∩Q[3]) 14,31;

32 Y=((Q[1]∩Q[2])∩Q[3])  (14)

*****SUBSTR 32 IN 30;

33  /CLEAR∩↓CLOCK/Q[0]←Q[0]⊕((Q[1]∩Q[2])∩Q[3]);   (11 14)

*****∧I (33 29);

34  /CLEAR∩↓CLOCK/Q[0]←Q[0]⊕((Q[1]∩Q[2])∩Q[3]); ∧ /CLEAR∩↓CLOCK/Q[1]←Q%
[1]⊕(Q[2]∩Q[3]);   (8 11 13 14)

*****∧I (34 25);

35 ( /CLEAR∩↓CLOCK/Q[0]←Q[0]⊕((Q[1]∩Q[2])∩Q[3]); ∧ /CLEAR∩↓CLOCK/Q[1]←%
Q[1]⊕(Q[2]∩Q[3]); )∧ /CLEAR∩↓CLOCK/Q[2]←Q[2]⊕Q[3];   (5 8 11 13 14)

*****∧I (35 24);

36 (( /CLEAR∩↓CLOCK/Q[0]←Q[0]⊕((Q[1]∩Q[2])∩Q[3]); ∧ /CLEAR∩↓CLOCK/Q[1]%
←Q[1]⊕(Q[2]∩Q[3]); )∧ /CLEAR∩↓CLOCK/Q[2]←Q[2]⊕Q[3]; )∧ /CLEAR∩↓CLOCK/Q%
[3]←~Q[3];   (2 5 8 11 13 14)

*****REWRITE 36 BY { M6};

37  /CLEAR∩↓CLOCK/((Q[0]&Q[1])&Q[2])&Q[3]←(((Q[0]⊕((Q[1]∩Q[2])∩Q[3]))&%
(Q[1]⊕(Q[2]∩Q[3])))&(Q[2]⊕Q[3]))&~Q[3];   (2 5 8 11 13 14)

*****REWRITE 37 BY { A6};

38  /CLEAR∩↓CLOCK/((Q[0]&Q[1])&Q[2])&Q[3]←(((Q[0]&Q[1])&Q[2])&Q[3])+1;%
    (2 5 8 11 13 14)

*****REWRITE 38 BY CONCATENATE;

39  /CLEAR∩↓CLOCK/Q[0:3]←Q[0:3]+1;   (2 5 8 11 13 14)

*****∧I (12 9);

40  /CLEAR∩~↓CLOCK/Q[0]←Q[0]; ∧ /CLEAR∩~↓CLOCK/Q[1]←Q[1];   (9 12)

*****∧I (40 6);

41 ( /CLEAR∩~↓CLOCK/Q[0]←Q[0]; ∧ /CLEAR∩~↓CLOCK/Q[1]←Q[1]; )∧ /CLEAR∩~%
↓CLOCK/Q[2]←Q[2];   (6 9 12)

*****∧I (41 3);
```

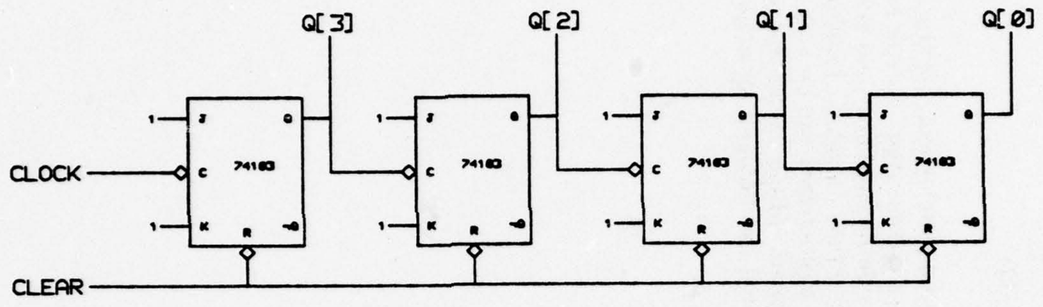
```
42 (( /CLEARn~↓CLOCK/Q[0]←Q[0]; ∧ /CLEARn~↓CLOCK/Q[1]←Q[1]; )∧ /CLEARn%
~↓CLOCK/Q[2]←Q[2]; )∧ /CLEARn~↓CLOCK/Q[3]←Q[3];   (3 6 9 12)

*****REWRITE 42 BY { M6};

43  /CLEARn~↓CLOCK/((Q[0]&Q[1])&Q[2])&Q[3]←((Q[0]&Q[1])&Q[2])&Q[3];   %
(3 6 9 12)

*****REWRITE 43 BY CONCATENATE;

44  /CLEARn~↓CLOCK/Q[0:3]←Q[0:3];   (3 6 9 12)
```

11. RIPPLE COUNTER

The next verification example will be the binary ripple counter shown in Figure 11.1. The desired goal is the same as for the last example:

```
/~CLEAR/ Q[0:3]←0;                            (17)
/CLEAR∩↓CLOCK/ Q[0:3]←Q[0:3]+1;               (96)
/CLEAR∩~↓CLOCK/ Q[0:3]←Q[0:3];                (101)
```

Although the circuit is simpler than the synchronous counter in terms of wires and components, the proof is much more complicated. This is because transition expressions for the first 3 bits must be determined and then substituted into the original assumptions. Many of the steps involve moving variables and parentheses around so that conditional expressions can be simplified. Quantifiers are used to a great advantage in this proof. Steps 26-33, for example, create a special version of transition axiom C9 that can be used for edge-triggered toggle devices with a clear input.



```
/¬CLEAR/ Q[0:3]←0;
/CLEAR∧↓CLOCK/ Q[0:3]←Q[0:3]+1;
/CLEAR∧¬↓CLOCK/ Q[0:3]←Q[0:3];
```

FIGURE 11.1   RIPPLE COUNTER

```
*****DECLARE INDCONST CLEAR,CLOCK,Q;

*****ASSUME  /~CLEAR/Q[3]←0; ;

1  /~CLEAR/Q[3]←0;   (1)

*****ASSUME  /CLEAR∩↓CLOCK/Q[3]←(1∩~Q[3])∪(~1∩Q[3]); ;

2  /CLEAR∩↓CLOCK/Q[3]←(1∩~Q[3])∪(~1∩Q[3]);   (2)

*****ASSUME  /CLEAR∩~↓CLOCK/Q[3]←Q[3]; ;

3  /CLEAR∩~↓CLOCK/Q[3]←Q[3];   (3)

*****ASSUME  /~CLEAR/Q[2]←0; ;

4  /~CLEAR/Q[2]←0;   (4)

*****ASSUME  /CLEAR∩↓Q[3]/Q[2]←(1∩~Q[2])∪(~1∩Q[2]); ;

5  /CLEAR∩↓Q[3]/Q[2]←(1∩~Q[2])∪(~1∩Q[2]);   (5)

*****ASSUME  /CLEAR∩~↓Q[3]/Q[2]←Q[2]; ;

6  /CLEAR∩~↓Q[3]/Q[2]←Q[2];   (6)

*****ASSUME  /~CLEAR/Q[1]←0; ;

7  /~CLEAR/Q[1]←0;   (7)

*****ASSUME  /CLEAR∩↓Q[2]/Q[1]←(1∩~Q[1])∪(~1∩Q[1]); ;

8  /CLEAR∩↓Q[2]/Q[1]←(1∩~Q[1])∪(~1∩Q[1]);   (8)

*****ASSUME  /CLEAR∩~↓Q[2]/Q[1]←Q[1]; ;

9  /CLEAR∩~↓Q[2]/Q[1]←Q[1];   (9)

*****ASSUME  /~CLEAR/Q[0]←0; ;

10  /~CLEAR/Q[0]←0;   (10)

*****ASSUME  /CLEAR∩↓Q[1]/Q[0]←(1∩~Q[0])∪(~1∩Q[0]); ;

11  /CLEAR∩↓Q[1]/Q[0]←(1∩~Q[0])∪(~1∩Q[0]);   (11)

*****ASSUME  /CLEAR∩~↓Q[1]/Q[0]←Q[0]; ;

12  /CLEAR∩~↓Q[1]/Q[0]←Q[0];   (12)

*****∧I (10 7);

13  /~CLEAR/Q[0]←0; ∧ /~CLEAR/Q[1]←0;   (7 10)

*****∧I (13 4);

14 ( /~CLEAR/Q[0]←0; ∧ /~CLEAR/Q[1]←0; )∧ /~CLEAR/Q[2]←0;   (4 7 10)
```

```
*****∧I (14 1);

15 (( /~CLEAR/Q[0]←0; ∧ /~CLEAR/Q[1]←0; )∧ /~CLEAR/Q[2]←0; )∧ /~CLEAR/%
Q[3]←0;   (1 4 7 10)

*****REWRITE 15 BY { M6};

16  /~CLEAR/((Q[0]&Q[1])&Q[2])&Q[3]←((0&0)&0)&0;   (1 4 7 10)

*****REWRITE 16 BY CONCATENATE;

17  /~CLEAR/Q[0:3]←((0&0)&0)&0;   (1 4 7 10)

*****VE X1 x,y;

18 ((x∩~y)∪(~x∩y))=(x●y)

*****VE X6 x,y;

19 (x●y)=(y●x)

*****SUBSTR 19 IN 18;

20 ((x∩~y)∪(~x∩y))=(y●x)

*****VI 20 x y;

21 ∀x y.((x∩~y)∪(~x∩y))=(y●x)

*****REWRITE 2 BY REDUCE;

22  /CLEAR∩↓CLOCK/Q[3]←~Q[3];   (2)

*****REWRITE 5 BY REDUCE;

23  /CLEAR∩↓Q[3]/Q[2]←~Q[2];   (5)

*****REWRITE 8 BY REDUCE;

24  /CLEAR∩↓Q[2]/Q[1]←~Q[1];   (8)

*****REWRITE 11 BY REDUCE;

25  /CLEAR∩↓Q[1]/Q[0]←~Q[0];   (11)

*****VE T10BR x,x,y;

26 (x∩(x∩y))=((x∩x)∩y)

*****REWRITE 26 BY REDUCE;

27 (x∩(x∩y))=(x∩y)

*****VE C9B CLEAR∩↓x,a,~a;

28  /CLEAR∩↓x/a←~a; ⊃↓a=(((CLEAR∩↓x)∩↓~a)∪((a∩~~a)∩↑(CLEAR∩↓x)))

*****REWRITE 28 BY REDUCE∪TRANS∪{ T10B,27};
```

```
29  /CLEAR∩↓x/a←~a; ⊃↓a=(a∩(CLEAR∩↓x))
*****VE T7B a,CLEAR∩↓x;
30 (a∩(CLEAR∩↓x))=((CLEAR∩↓x)∩a)
*****SUBSTR 30 IN 29;
31  /CLEAR∩↓x/a←~a; ⊃↓a=((CLEAR∩↓x)∩a)
*****REWRITE 31 BY { T10B};
32  /CLEAR∩↓x/a←~a; ⊃↓a=(CLEAR∩(↓x∩a))
*****VI 32 x a;
33 ∀x a.( /CLEAR∩↓x/a←~a; ⊃↓a=(CLEAR∩(↓x∩a)))
*****VE 33 CLOCK,Q[3];
34  /CLEAR∩↓CLOCK/Q[3]←~Q[3]; ⊃↓Q[3]=(CLEAR∩(↓CLOCK∩Q[3]))
*****TAUT ↓Q[3]=(CLEAR∩(↓CLOCK∩Q[3])) 22,34;
35 ↓Q[3]=(CLEAR∩(↓CLOCK∩Q[3]))  (2)
*****VE 33 Q[3],Q[2];
36  /CLEAR∩↓Q[3]/Q[2]←~Q[2]; ⊃↓Q[2]=(CLEAR∩(↓Q[3]∩Q[2]))
*****TAUT ↓Q[2]=(CLEAR∩(↓Q[3]∩Q[2])) 23,36;
37 ↓Q[2]=(CLEAR∩(↓Q[3]∩Q[2]))  (5)
*****SUBSTR 35 IN 37;
38 ↓Q[2]=(CLEAR∩((CLEAR∩(↓CLOCK∩Q[3]))∩Q[2]))  (2 5)
*****REWRITE 38 BY REDUCE∪{ T10B,27};
39 ↓Q[2]=(CLEAR∩(↓CLOCK∩(Q[3]∩Q[2])))  (2 5)
*****VE 33 Q[2],Q[1];
40  /CLEAR∩↓Q[2]/Q[1]←~Q[1]; ⊃↓Q[1]=(CLEAR∩(↓Q[2]∩Q[1]))
*****TAUT ↓Q[1]=(CLEAR∩(↓Q[2]∩Q[1])) 24,40;
41 ↓Q[1]=(CLEAR∩(↓Q[2]∩Q[1]))  (8)
*****SUBSTR 39 IN 41;
42 ↓Q[1]=(CLEAR∩((CLEAR∩(↓CLOCK∩(Q[3]∩Q[2])))∩Q[1]))  (2 5 8)
*****REWRITE 42 BY REDUCE∪{ T10B,27};
43 ↓Q[1]=(CLEAR∩(↓CLOCK∩(Q[3]∩(Q[2]∩Q[1]))))  (2 5 8)
*****SUBSTR 35 IN 23;
```

```
44  /CLEAR∩(CLEAR∩(↓CLOCK∩Q[3]))/Q[2]←~Q[2];   (2 5)

*****REWRITE 44 BY { 27};

45  /CLEAR∩(↓CLOCK∩Q[3])/Q[2]←~Q[2];   (2 5)

*****SUBSTR 35 IN 6;

46  /CLEAR∩~(CLEAR∩(↓CLOCK∩Q[3]))/Q[2]←Q[2];   (2 6)

*****REWRITE CLEAR∩~(CLEAR∩(↓CLOCK∩x)) BY REDUCE∪{ T11A,T14B};

47 (CLEAR∩~(CLEAR∩(↓CLOCK∩x)))=((CLEAR∩~↓CLOCK)∪(CLEAR∩~x))

*****REWRITE  /(CLEAR∩~↓CLOCK)∪(CLEAR∩~x)/a←a;  BY { M5};

48  /(CLEAR∩~↓CLOCK)∪(CLEAR∩~x)/a←a; ≡( /CLEAR∩~↓CLOCK/a←a; ∧ /CLEAR∩~%
x/a←a; )

*****SUBSTR 47 IN 48;

49  /(CLEAR∩~↓CLOCK)∪(CLEAR∩~x)/a←a; ≡( /CLEAR∩~↓CLOCK/a←a; ∧ /CLEAR∩~%
x/a←a; )

*****VE M10 CLEAR∩~x,↓CLOCK,a,a;

50  /CLEAR∩~x/a←a; ⊃ /(CLEAR∩~x)∩↓CLOCK/a←a;

*****REWRITE 50 BY { T10B};

51  /CLEAR∩~x/a←a; ⊃ /CLEAR∩(~x∩↓CLOCK)/a←a;

*****VE T7B ~x,↓CLOCK;

52 (~x∩↓CLOCK)=(↓CLOCK∩~x)

*****SUBSTR 52 IN 51;

53  /CLEAR∩~x/a←a; ⊃ /CLEAR∩(↓CLOCK∩~x)/a←a;

*****TAUT  /(CLEAR∩~↓CLOCK)∪(CLEAR∩~x)/a←a; ⊃( /CLEAR∩~↓CLOCK/a←a; ∧ /%
CLEAR∩(↓CLOCK∩~x)/a←a; ) 49,53;

54  /(CLEAR∩~↓CLOCK)∪(CLEAR∩~x)/a←a; ⊃( /CLEAR∩~↓CLOCK/a←a; ∧ /CLEAR∩(%
↓CLOCK∩~x)/a←a; )

*****SUBST 47 IN 54;

55  /CLEAR∩~(CLEAR∩(↓CLOCK∩x))/a←a; ⊃( /CLEAR∩~↓CLOCK/a←a; ∧ /CLEAR∩(↓%
CLOCK∩~x)/a←a; )

*****VI 55 x a;

56 ∀x a.( /CLEAR∩~(CLEAR∩(↓CLOCK∩x))/a←a; ⊃( /CLEAR∩~↓CLOCK/a←a; ∧ /CL%
EAR∩(↓CLOCK∩~x)/a←a; ))

*****VE 56 Q[3],Q[2];
```

```
57  /CLEAR∩~(CLEAR∩(↓CLOCK∩Q[3]))/Q[2]←Q[2]; ⊃( /CLEAR∩~↓CLOCK/Q[2]←Q[%
2]; ∧ /CLEAR∩(↓CLOCK∩~Q[3])/Q[2]←Q[2]; )

*****TAUT  /CLEAR∩~↓CLOCK/Q[2]←Q[2]; ∧ /CLEAR∩(↓CLOCK∩~Q[3])/Q[2]←Q[2]%
;  46,57;

58  /CLEAR∩~↓CLOCK/Q[2]←Q[2]; ∧ /CLEAR∩(↓CLOCK∩~Q[3])/Q[2]←Q[2];   (2 %
6)

*****∧E 58:#1;

59  /CLEAR∩~↓CLOCK/Q[2]←Q[2];   (2 6)

*****∧E 58:#2;

60  /CLEAR∩(↓CLOCK∩~Q[3])/Q[2]←Q[2];   (2 6)

*****SUBSTR 39 IN 24;

61  /CLEAR∩(CLEAR∩(↓CLOCK∩(Q[3]∩Q[2])))/Q[1]←~Q[1];   (2 5 8)

*****REWRITE 61 BY { 27};

62  /CLEAR∩(↓CLOCK∩(Q[3]∩Q[2]))/Q[1]←~Q[1];   (2 5 8)

*****SUBSTR 39 IN 9;

63  /CLEAR∩~(CLEAR∩(↓CLOCK∩(Q[3]∩Q[2])))/Q[1]←Q[1];   (2 5 9)

*****VE 56 Q[3]∩Q[2],Q[1];

64  /CLEAR∩~(CLEAR∩(↓CLOCK∩(Q[3]∩Q[2])))/Q[1]←Q[1]; ⊃( /CLEAR∩~↓CLOCK/%
Q[1]←Q[1]; ∧ /CLEAR∩(↓CLOCK∩~(Q[3]∩Q[2]))/Q[1]←Q[1]; )

*****TAUT  /CLEAR∩~↓CLOCK/Q[1]←Q[1]; ∧ /CLEAR∩(↓CLOCK∩~(Q[3]∩Q[2]))/Q[%
1]←Q[1];  63:64;

65  /CLEAR∩~↓CLOCK/Q[1]←Q[1]; ∧ /CLEAR∩(↓CLOCK∩~(Q[3]∩Q[2]))/Q[1]←Q[1]%
;   (2 5 9)

*****∧E 65:#1;

66  /CLEAR∩~↓CLOCK/Q[1]←Q[1];   (2 5 9)

*****∧E 65:#2;

67  /CLEAR∩(↓CLOCK∩~(Q[3]∩Q[2]))/Q[1]←Q[1];   (2 5 9)

*****SUBSTR 43 IN 25;

68  /CLEAR∩(CLEAR∩(↓CLOCK∩(Q[3]∩(Q[2]∩Q[1]))))/Q[0]←~Q[0];   (2 5 8 11%
)

*****REWRITE 68 BY { 27};

69  /CLEAR∩(↓CLOCK∩(Q[3]∩(Q[2]∩Q[1])))/Q[0]←~Q[0];   (2 5 8 11)

*****SUBSTR 43 IN 12;
```

```
70  /CLEAR∩~(CLEAR∩(↓CLOCK∩(Q[3]∩(Q[2]∩Q[1]))))/Q[0]←Q[0];   (2 5 8 12%
)

*****∀E 56 Q[3]∩(Q[2]∩Q[1]),Q[0];

71  /CLEAR∩~(CLEAR∩(↓CLOCK∩(Q[3]∩(Q[2]∩Q[1]))))/Q[0]←Q[0]; ⊃( /CLEAR∩~%
↓CLOCK/Q[0]←Q[0]; ∧ /CLEAR∩(↓CLOCK∩~(Q[3]∩(Q[2]∩Q[1])))/Q[0]←Q[0]; )

*****TAUT  /CLEAR∩~↓CLOCK/Q[0]←Q[0]; ∧ /CLEAR∩(↓CLOCK∩~(Q[3]∩(Q[2]∩Q[1%
])))/Q[0]←Q[0];  70:71;

72  /CLEAR∩~↓CLOCK/Q[0]←Q[0]; ∧ /CLEAR∩(↓CLOCK∩~(Q[3]∩(Q[2]∩Q[1])))/Q[%
0]←Q[0];   (2 5 8 12)

*****∧E 72:#1;

73  /CLEAR∩~↓CLOCK/Q[0]←Q[0];   (2 5 8 12)

*****∧E 72:#2;

74  /CLEAR∩(↓CLOCK∩~(Q[3]∩(Q[2]∩Q[1])))/Q[0]←Q[0];   (2 5 8 12)

*****∀E M3 CLEAR∩↓CLOCK,x,a,~a,a;

75 ( /(CLEAR∩↓CLOCK)∩x/a←~a; ∧ /(CLEAR∩↓CLOCK)∩~x/a←a; )≡ /CLEAR∩↓CLOC%
K/a←(x∩~a)∪(~x∩a);

*****REWRITE 75 BY { 21};

76 ( /(CLEAR∩↓CLOCK)∩x/a←~a; ∧ /(CLEAR∩↓CLOCK)∩~x/a←a; )≡ /CLEAR∩↓CLOC%
K/a←a⊕x;

*****REWRITE 76 BY { T10B};

77 ( /CLEAR∩(↓CLOCK∩x)/a←~a; ∧ /CLEAR∩(↓CLOCK∩~x)/a←a; )≡ /CLEAR∩↓CLOC%
K/a←a⊕x;

*****∀I 77 x a;

78 ∀x a.(( /CLEAR∩(↓CLOCK∩x)/a←~a; ∧ /CLEAR∩(↓CLOCK∩~x)/a←a; )≡ /CLEAR%
∩↓CLOCK/a←a⊕x; )

*****∧I (45 60);

79  /CLEAR∩(↓CLOCK∩Q[3])/Q[2]←~Q[2]; ∧ /CLEAR∩(↓CLOCK∩~Q[3])/Q[2]←Q[2]%
;   (2 5 6)

*****REWRITE 79 BY { 78};

80  /CLEAR∩↓CLOCK/Q[2]←Q[2]⊕Q[3];   (2 5 6)

*****∧I (62 67);

81  /CLEAR∩(↓CLOCK∩(Q[3]∩Q[2]))/Q[1]←~Q[1]; ∧ /CLEAR∩(↓CLOCK∩~(Q[3]∩Q[%
2]))/Q[1]←Q[1];   (2 5 8 9)

*****REWRITE 81 BY { 78};

82  /CLEAR∩↓CLOCK/Q[1]←Q[1]⊕(Q[3]∩Q[2]);   (2 5 8 9)
```

```
*****VE T7B Q[3],Q[2];

83 (Q[3]∩Q[2])=(Q[2]∩Q[3])

*****SUBSTR 83 IN 82;

84  /CLEAR∩↓CLOCK/Q[1]←Q[1]⊕(Q[2]∩Q[3]);   (2 5 8 9)

*****∧I (69 74);

85  /CLEAR∩(↓CLOCK∩(Q[3]∩(Q[2]∩Q[1])))/Q[0]←~Q[0]; ∧ /CLEAR∩(↓CLOCK∩~(%
Q[3]∩(Q[2]∩Q[1])))/Q[0]←Q[0];   (2 5 8 11 12)

*****REWRITE 85 BY { 78};

86  /CLEAR∩↓CLOCK/Q[0]←Q[0]⊕(Q[3]∩(Q[2]∩Q[1]));   (2 5 8 11 12)

*****VE T7B Q[3],Q[2]∩Q[1];

87 (Q[3]∩(Q[2]∩Q[1]))=((Q[2]∩Q[1])∩Q[3])

*****SUBSTR 87 IN 86;

88  /CLEAR∩↓CLOCK/Q[0]←Q[0]⊕((Q[2]∩Q[1])∩Q[3]);   (2 5 8 11 12)

*****VE T7B Q[2],Q[1];

89 (Q[2]∩Q[1])=(Q[1]∩Q[2])

*****SUBSTR 89 IN 88;

90  /CLEAR∩↓CLOCK/Q[0]←Q[0]⊕((Q[1]∩Q[2])∩Q[3]);   (2 5 8 11 12)

*****∧I (90 84);

91  /CLEAR∩↓CLOCK/Q[0]←Q[0]⊕((Q[1]∩Q[2])∩Q[3]); ∧ /CLEAR∩↓CLOCK/Q[1]←Q%
[1]⊕(Q[2]∩Q[3]);   (2 5 8 9 11 12)

*****∧I (91 80);

92 ( /CLEAR∩↓CLOCK/Q[0]←Q[0]⊕((Q[1]∩Q[2])∩Q[3]); ∧ /CLEAR∩↓CLOCK/Q[1]←%
Q[1]⊕(Q[2]∩Q[3]); )∧ /CLEAR∩↓CLOCK/Q[2]←Q[2]⊕Q[3];   (2 5 6 8 9 11 12)

*****∧I (92 22);

93 (( /CLEAR∩↓CLOCK/Q[0]←Q[0]⊕((Q[1]∩Q[2])∩Q[3]); ∧ /CLEAR∩↓CLOCK/Q[1]%
←Q[1]⊕(Q[2]∩Q[3]); )∧ /CLEAR∩↓CLOCK/Q[2]←Q[2]⊕Q[3]; )∧ /CLEAR∩↓CLOCK/Q%
[3]←~Q[3];   (2 5 6 8 9 11 12)

*****REWRITE 93 BY { M6};

94  /CLEAR∩↓CLOCK/((Q[0]&Q[1])&Q[2])&Q[3]←(((Q[0]⊕((Q[1]∩Q[2])∩Q[3]))&%
(Q[1]⊕(Q[2]∩Q[3])))&(Q[2]⊕Q[3]))&~Q[3];   (2 5 6 8 9 11 12)

*****REWRITE 94 BY { A6};

95  /CLEAR∩↓CLOCK/((Q[0]&Q[1])&Q[2])&Q[3]←(((Q[0]&Q[1])&Q[2])&Q[3])+1;%
   (2 5 6 8 9 11 12)
```

```
*****REWRITE 95 BY CONCATENATE;

96  /CLEARn↓CLOCK/Q[0:3]←Q[0:3]+1;   (2 5 6 8 9 11 12)

*****∧I (73 66);

97  /CLEARn~↓CLOCK/Q[0]←Q[0]; ∧ /CLEARn~↓CLOCK/Q[1]←Q[1];   (2 5 8 9 1%
2)

*****∧I (97 59);

98 ( /CLEARn~↓CLOCK/Q[0]←Q[0]; ∧ /CLEARn~↓CLOCK/Q[1]←Q[1]; )∧ /CLEARn~%
↓CLOCK/Q[2]←Q[2];   (2 5 6 8 9 12)

*****∧I (98 3);

99 (( /CLEARn~↓CLOCK/Q[0]←Q[0]; ∧ /CLEARn~↓CLOCK/Q[1]←Q[1]; )∧ /CLEARn%
~↓CLOCK/Q[2]←Q[2]; )∧ /CLEARn~↓CLOCK/Q[3]←Q[3];   (2 3 5 6 8 9 12)

*****REWRITE 99 BY { M6};

100  /CLEARn~↓CLOCK/((Q[0]&Q[1])&Q[2])&Q[3]←((Q[0]&Q[1])&Q[2])&Q[3];  %
 (2 3 5 6 8 9 12)

*****REWRITE 100 BY CONCATENATE;

101  /CLEARn~↓CLOCK/Q[0:3]←Q[0:3];   (2 3 5 6 8 9 12)
```

12. 8-BIT MULTIPLIER

For our last example we have a multiplier circuit built using some MSI components. The circuit diagram is in Figure 12.1. The 8-bit operands are inputs X and Y, and the 16-bit product will be in the Z register. When the LOAD signal is low the control counter, DONE flip-flop, and high order bits of the Z register are cleared. A downwards transition in the CLOCK at this time will put the X operand into the W register and the Y operand into the low order bits of the Z register.

The actual multiplication sequence takes place while LOAD is true. On upwards transitions in CLOCK the counter is incremented. Downwards CLOCK transitions control the adding and shifting operations. When the low order bit of the counter is true the contents of the overflow bit and the Z register will be shifted right. When it is false, the high order bits of the Z register will be set to the sum of themselves and the W register if the least significant bit of the Z register is true. The overflow bit is intended to function as the most significant bit in the Z register before shifting. After this shifting and adding goes on for 16 clock cycles, the transition from 15 to 0 in the counter will cause the DONE flag to be set. The DONE flag will inhibit the clock pulses from changing the product in the Z register.

```
/~LOAD/ OV&Z[0:7]←0;                                          (57)
/~LOAD/ Q[0:3]←0;                                             (40)
/~LOAD/ DONE←0;                                               (58)
/~LOAD∩↓CLOCK/ W[0:7]←X[0:7];                                 (70)
/~LOAD∩↓CLOCK/ Z[8:15]←Y[0:7];                                (105)

/LOAD∩↑CLOCK/ Q[0:3]←Q[0:3]+1;                                (106)
/LOAD∩↓(Q[0:3]↔15)/ DONE←1;                                   (114)
/((LOAD∩↓CLOCK)∩~DONE)∩Q[3]/ OV&Z[0:15]←0&OV&Z[0:14];         (162)
/((LOAD∩↓CLOCK)∩~DONE)∩(~Q[3]∩Z[15])/
  OV&Z[0:7]←carry(W[0:7],Z[0:7],0)&(W[0:7]+Z[0:7]+0);         (197)
/((LOAD∩↓CLOCK)∩~DONE)∩(~Q[3]∩~Z[15])/
  OV&Z[0:7]←0&Z[0:7];                                         (215)
/((LOAD∩↓CLOCK)∩~DONE)∩~Q[3]/ Z[8:15]←Z[8:15];                (227)

/~((↓LOAD∩~CLOCK)∪(~LOAD∩↓CLOCK))/W[0:7]←W[0:7];              (235)
/LOAD∩~↑CLOCK/ Q[0:3]←Q[0:3];                                 (236)
/LOAD∩~↓(Q[0:3]↔15)/ DONE←DONE;                               (239)
/LOAD∩~((↓CLOCK∩~DONE)∪(~CLOCK∩↓DONE))/
  OV&Z[0:15]←OV&Z[0:15];                                      (260)
```

In the above goals the first group represents the load sequence, the second group is the multiply sequence, and the last group has some of the more important feedback states. Although this proof is 260 steps long most of it is very straightforward. The first 51 steps were needed just to get the component definitions into the proof checker. Some of the goals were obtained in as little as one step. The second goal (step 40) was obtained directly from an input assumption.

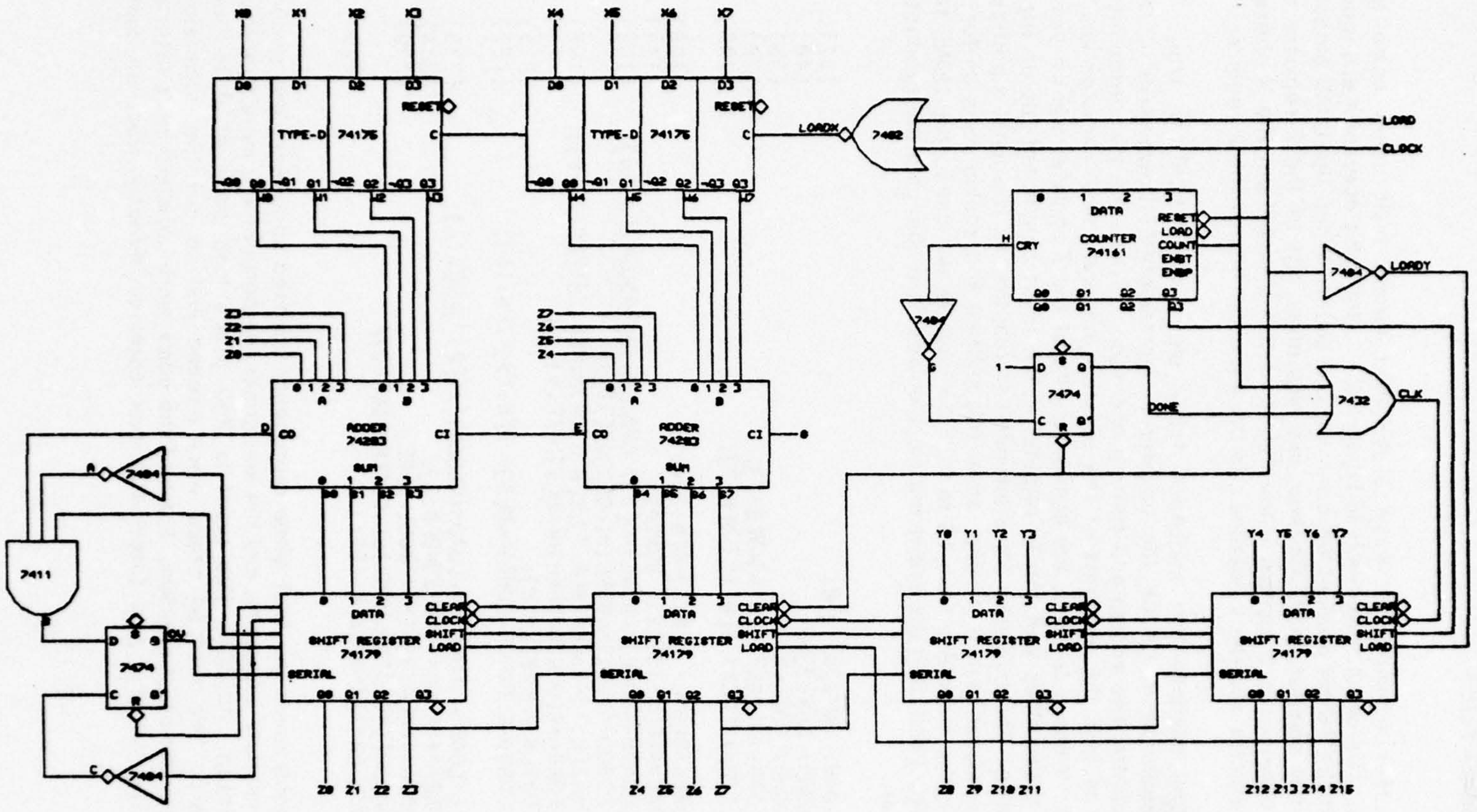


FIGURE 12.1 8-BIT MULTIPLIER

```
*****DECLARE INDCONST LOAD,CLOCK,X,Y,Z,W,S,Q,LOADX,LOADY,DONE;
*****DECLARE INDCONST A,B,C,D,E,G,H,CLK,OV;
*****ASSUME LOADX←~(LOAD∪CLOCK); ;
1 LOADX←~(LOAD∪CLOCK);   (1)
*****ASSUME LOADY←~LOAD; ;
2 LOADY←~LOAD;   (2)
*****ASSUME  /~1/W[0:3]←((0&0)&0)&0; ;
3  /~1/W[0:3]←((0&0)&0)&0;   (3)
*****ASSUME  /1∩↑LOADX/W[0:3]←X[0:3]; ;
4  /1∩↑LOADX/W[0:3]←X[0:3];   (4)
*****ASSUME  /1∩~↑LOADX/W[0:3]←W[0:3]; ;
5  /1∩~↑LOADX/W[0:3]←W[0:3];   (5)
*****ASSUME  /~1/W[4:7]←((0&0)&0)&0; ;
6  /~1/W[4:7]←((0&0)&0)&0;   (6)
*****ASSUME  /1∩↑LOADX/W[4:7]←X[4:7]; ;
7  /1∩↑LOADX/W[4:7]←X[4:7];   (7)
*****ASSUME  /1∩~↑LOADX/W[4:7]←W[4:7]; ;
8  /1∩~↑LOADX/W[4:7]←W[4:7];   (8)
*****ASSUME S[0:3]←(W[0:3]+Z[0:3])+E; ;
9 S[0:3]←(W[0:3]+Z[0:3])+E;   (9)
*****ASSUME D←carry(W[0:3],Z[0:3],E); ;
10 D←carry(W[0:3],Z[0:3],E);   (10)
*****ASSUME S[4:7]←(W[4:7]+Z[4:7])+0; ;
11 S[4:7]←(W[4:7]+Z[4:7])+0;   (11)
*****ASSUME E←carry(W[4:7],Z[4:7],0); ;
12 E←carry(W[4:7],Z[4:7],0);   (12)
*****ASSUME A←~Q[3]; ;
13 A←~Q[3];   (13)
*****ASSUME B←(D∩A)∩Z[15]; ;
```

```
14 B←(D∩A)∩Z[15];   (14)
*****ASSUME C←~CLK; ;
15 C←~CLK;   (15)
*****ASSUME  /~1/OV←1; ;
16  /~1/OV←1;   (16)
*****ASSUME  /1∩~LOAD/OV←0; ;
17  /1∩~LOAD/OV←0;   (17)
*****ASSUME  /(1∩LOAD)∩↑C/OV←B; ;
18  /(1∩LOAD)∩↑C/OV←B;   (18)
*****ASSUME  /(1∩LOAD)∩~↑C/OV←OV; ;
19  /(1∩LOAD)∩~↑C/OV←OV;   (19)
*****ASSUME  /~LOAD/Z[0:3]←((0&0)&0)&0; ;
20  /~LOAD/Z[0:3]←((0&0)&0)&0;   (20)
*****ASSUME  /(LOAD∩↓CLK)∩Q[3]/Z[0:3]←OV&Z[0:2]; ;
21  /(LOAD∩↓CLK)∩Q[3]/Z[0:3]←OV&Z[0:2];   (21)
*****ASSUME  /((LOAD∩↓CLK)∩~Q[3])∩Z[15]/Z[0:3]←S[0:3]; ;
22  /((LOAD∩↓CLK)∩~Q[3])∩Z[15]/Z[0:3]←S[0:3];   (22)
*****ASSUME  /((LOAD∩↓CLK)∩~Q[3])∩~Z[15]/Z[0:3]←Z[0:3]; ;
23  /((LOAD∩↓CLK)∩~Q[3])∩~Z[15]/Z[0:3]←Z[0:3];   (23)
*****ASSUME  /LOAD∩~↓CLK/Z[0:3]←Z[0:3]; ;
24  /LOAD∩~↓CLK/Z[0:3]←Z[0:3];   (24)
*****ASSUME  /~LOAD/Z[4:7]←((0&0)&0)&0; ;
25  /~LOAD/Z[4:7]←((0&0)&0)&0;   (25)
*****ASSUME  /(LOAD∩↓CLK)∩Q[3]/Z[4:7]←Z[3]&Z[4:6]; ;
26  /(LOAD∩↓CLK)∩Q[3]/Z[4:7]←Z[3]&Z[4:6];   (26)
*****ASSUME  /((LOAD∩↓CLK)∩~Q[3])∩Z[15]/Z[4:7]←S[4:7]; ;
27  /((LOAD∩↓CLK)∩~Q[3])∩Z[15]/Z[4:7]←S[4:7];   (27)
*****ASSUME  /((LOAD∩↓CLK)∩~Q[3])∩~Z[15]/Z[4:7]←Z[4:7]; ;
28  /((LOAD∩↓CLK)∩~Q[3])∩~Z[15]/Z[4:7]←Z[4:7];   (28)
*****ASSUME  /LOAD∩~↓CLK/Z[4:7]←Z[4:7]; ;
```

```
29  /LOAD∩~↓CLK/Z[4:7]←Z[4:7];   (29)
*****ASSUME  /~1/Z[8:11]←((0&0)&0)&0; ;
30  /~1/Z[8:11]←((0&0)&0)&0;   (30)
*****ASSUME  /(1∩↓CLK)∩Q[3]/Z[8:11]←Z[7]&Z[8:10]; ;
31  /(1∩↓CLK)∩Q[3]/Z[8:11]←Z[7]&Z[8:10];   (31)
*****ASSUME  /((1∩↓CLK)∩~Q[3])∩LOADY/Z[8:11]←Y[0:3]; ;
32  /((1∩↓CLK)∩~Q[3])∩LOADY/Z[8:11]←Y[0:3];   (32)
*****ASSUME  /((1∩↓CLK)∩~Q[3])∩~LOADY/Z[8:11]←Z[8:11]; ;
33  /((1∩↓CLK)∩~Q[3])∩~LOADY/Z[8:11]←Z[8:11];   (33)
*****ASSUME  /1∩~↓CLK/Z[8:11]←Z[8:11]; ;
34  /1∩~↓CLK/Z[8:11]←Z[8:11];   (34)
*****ASSUME  /~1/Z[12:15]←((0&0)&0)&0; ;
35  /~1/Z[12:15]←((0&0)&0)&0;   (35)
*****ASSUME  /(1∩↓CLK)∩Q[3]/Z[12:15]←Z[11]&Z[12:14]; ;
36  /(1∩↓CLK)∩Q[3]/Z[12:15]←Z[11]&Z[12:14];   (36)
*****ASSUME  /((1∩↓CLK)∩~Q[3])∩LOADY/Z[12:15]←Y[4:7]; ;
37  /((1∩↓CLK)∩~Q[3])∩LOADY/Z[12:15]←Y[4:7];   (37)
*****ASSUME  /((1∩↓CLK)∩~Q[3])∩~LOADY/Z[12:15]←Z[12:15]; ;
38  /((1∩↓CLK)∩~Q[3])∩~LOADY/Z[12:15]←Z[12:15];   (38)
*****ASSUME  /1∩~↓CLK/Z[12:15]←Z[12:15]; ;
39  /1∩~↓CLK/Z[12:15]←Z[12:15];   (39)
*****ASSUME  /~LOAD/Q[0:3]←((0&0)&0)&0; ;
40  /~LOAD/Q[0:3]←((0&0)&0)&0;   (40)
*****ASSUME  /(LOAD∩~1)∩↑CLOCK/Q[0:3]←((1&1)&1)&1; ;
41  /(LOAD∩~1)∩↑CLOCK/Q[0:3]←((1&1)&1)&1;   (41)
*****ASSUME  /(LOAD∩~1)∩~↑CLOCK/Q[0:3]←Q[0:3]; ;
42  /(LOAD∩~1)∩~↑CLOCK/Q[0:3]←Q[0:3];   (42)
*****ASSUME  /(((LOAD∩1)∩1)∩1)∩↑CLOCK/Q[0:3]←Q[0:3]+1; ;
43  /(((LOAD∩1)∩1)∩1)∩↑CLOCK/Q[0:3]←Q[0:3]+1;   (43)
```

```
*****ASSUME  /(LOAD∩1)∩~((1∩1)∩↑CLOCK)/Q[0:3]←Q[0:3]; ;

44  /(LOAD∩1)∩~((1∩1)∩↑CLOCK)/Q[0:3]←Q[0:3];   (44)

*****ASSUME H←Q[0:3]↔15; ;

45 H←Q[0:3]↔15;   (45)

*****ASSUME G←~H; ;

46 G←~H;   (46)

*****ASSUME  /~1/DONE←1; ;

47  /~1/DONE←1;   (47)

*****ASSUME  /1∩~LOAD/DONE←0; ;

48  /1∩~LOAD/DONE←0;   (48)

*****ASSUME  /(1∩LOAD)∩↑G/DONE←1; ;

49  /(1∩LOAD)∩↑G/DONE←1;   (49)

*****ASSUME  /(1∩LOAD)∩~↑G/DONE←DONE; ;

50  /(1∩LOAD)∩~↑G/DONE←DONE;   (50)

*****ASSUME CLK←CLOCK∪DONE; ;

51 CLK←CLOCK∪DONE;   (51)

*****∧I (20 25);

52  /~LOAD/Z[0:3]←((0&0)&0)&0; ∧ /~LOAD/Z[4:7]←((0&0)&0)&0;   (20 25)

*****REWRITE 52 BY { M6};

53  /~LOAD/Z[0:3]&Z[4:7]←(((0&0)&0)&0)&(((0&0)&0)&0);   (20 25)

*****REWRITE 53 BY CONCATENATE;

54  /~LOAD/Z[0:7]←(((0&0)&0)&0)&(((0&0)&0)&0);   (20 25)

*****REWRITE 17 BY REDUCE;

55  /~LOAD/OV←0;   (17)

*****∧I (55 54);

56  /~LOAD/OV←0; ∧ /~LOAD/Z[0:7]←(((0&0)&0)&0)&(((0&0)&0)&0);   (17 20%
 25)

*****REWRITE 56 BY { M6};

57  /~LOAD/OV&Z[0:7]←0&((((0&0)&0)&0)&(((0&0)&0)&0));   (17 20 25)

*****REWRITE 48 BY REDUCE;
```

```
58  /~LOAD/DONE←0;   (48)

*****VE M2 LOADX,~(LOAD∪CLOCK);

59 LOADX←~(LOAD∪CLOCK); ⊃LOADX=~(LOAD∪CLOCK)

*****TAUT LOADX=~(LOAD∪CLOCK) 1,59;

60 LOADX=~(LOAD∪CLOCK)  (1)

*****REWRITE 4 BY REDUCE;

61  /↑LOADX/W[0:3]←X[0:3];   (4)

*****REWRITE 7 BY REDUCE;

62  /↑LOADX/W[4:7]←X[4:7];   (7)

*****∧I (61 62);

63  /↑LOADX/W[0:3]←X[0:3]; ∧ /↑LOADX/W[4:7]←X[4:7];   (4 7)

*****REWRITE 63 BY { M6};

64  /↑LOADX/W[0:3]&W[4:7]←X[0:3]&X[4:7];   (4 7)

*****REWRITE 64 BY CONCATENATE;

65  /↑LOADX/W[0:7]←X[0:7];   (4 7)

*****SUBSTR 60 IN 65;

66  /↑~(LOAD∪CLOCK)/W[0:7]←X[0:7];   (1 4 7)

*****REWRITE 56 BY { T14A};

67  /↑(~LOAD∩~CLOCK)/W[0:7]←X[0:7];   (1 4 7)

*****REWRITE 67 BY TRANS;

68  /(↓LOAD∩~CLOCK)∪(~LOAD∩↓CLOCK)/W[0:7]←X[0:7];   (1 4 7)

*****REWRITE 68 BY { M5};

69  /↓LOAD∩~CLOCK/W[0:7]←X[0:7]; ∧ /~LOAD∩↓CLOCK/W[0:7]←X[0:7];   (1 4%
 7)

*****∧E 69:#2;

70  /~LOAD∩↓CLOCK/W[0:7]←X[0:7];   (1 4 7)

*****VE M2 LOADY,~LOAD;

71 LOADY←~LOAD; ⊃LOADY=~LOAD

*****TAUT LOADY=~LOAD 2,71;

72 LOADY=~LOAD  (2)
```

```
*****REWRITE 32 BY REDUCE;

73  /(↓CLK∩~Q[3])∩LOADY/Z[8:11]←Y[0:3];   (32)

*****REWRITE 37 BY REDUCE;

74  /(↓CLK∩~Q[3])∩LOADY/Z[12:15]←Y[4:7];   (37)

*****∧I (73 74);

75  /(↓CLK∩~Q[3])∩LOADY/Z[8:11]←Y[0:3]; ∧ /(↓CLK∩~Q[3])∩LOADY/Z[12:15]%
←Y[4:7];   (32 37)

*****REWRITE 75 BY { M6};

76  /(↓CLK∩~Q[3])∩LOADY/Z[8:11]&Z[12:15]←Y[0:3]&Y[4:7];   (32 37)

*****REWRITE 76 BY CONCATENATE;

77  /(↓CLK∩~Q[3])∩LOADY/Z[8:15]←Y[0:7];   (32 37)

*****SUBSTR 72 IN 77;

78  /(↓CLK∩~Q[3])∩~LOAD/Z[8:15]←Y[0:7];   (2 32 37)

*****VE M6 ~LOAD,Q[0:2],(0&0)&0,Q[3],0;

79 ( /~LOAD/Q[0:2]←(0&0)&0; ∧ /~LOAD/Q[3]←0; )≡ /~LOAD/Q[0:2]&Q[3]←((0%
&0)&0)&0;

*****REWRITE 79 BY CONCATENATE;

80 ( /~LOAD/Q[0:2]←(0&0)&0; ∧ /~LOAD/Q[3]←0; )≡ /~LOAD/Q[0:3]←((0&0)&0%
)&0;

*****SUBST 80 IN 40;

81  /~LOAD/Q[0:2]←(0&0)&0; ∧ /~LOAD/Q[3]←0;   (40)

*****∧E 81:#2;

82  /~LOAD/Q[3]←0;   (40)

*****VE M11B ~LOAD,Q[3],0;

83  /~LOAD/Q[3]←0; ⊃(~LOAD∩~Q[3])=(~LOAD∩~0)

*****TAUT (~LOAD∩~Q[3])=(~LOAD∩~0) 82:83;

84 (~LOAD∩~Q[3])=(~LOAD∩~0)  (40)

*****REWRITE 84 BY REDUCE;

85 (~LOAD∩~Q[3])=~LOAD  (40)

*****REWRITE 78 BY { T10B};

86  /↓CLK∩(~Q[3]∩~LOAD)/Z[8:15]←Y[0:7];   (2 32 37)
```

```
*****VE T7B ~Q[3],~LOAD;

87 (~Q[3]∩~LOAD)=(~LOAD∩~Q[3])

*****SUBSTR 87 IN 86;

88  /↓CLK∩(~LOAD∩~Q[3])/Z[8:15]←Y[0:7];   (2 32 37)

*****SUBSTR 85 IN 88;

89  /↓CLK∩~LOAD/Z[8:15]←Y[0:7];   (2 32 37 40)

*****VE T7B ↓CLK,~LOAD;

90 (↓CLK∩~LOAD)=(~LOAD∩↓CLK)

*****SUBSTR 90 IN 89;

91  /~LOAD∩↓CLK/Z[8:15]←Y[0:7];   (2 32 37 40)

*****VE M2 CLK,CLOCK∪DONE;

92 CLK←CLOCK∪DONE; ⊃CLK=(CLOCK∪DONE)

*****TAUT CLK=(CLOCK∪DONE) 51,92;

93 CLK=(CLOCK∪DONE)  (51)

*****SUBSTR 93 IN 91;

94  /~LOAD∩↓(CLOCK∪DONE)/Z[8:15]←Y[0:7];   (2 32 37 40 51)

*****REWRITE 94 BY TRANS;

95  /~LOAD∩((↓CLOCK∩~DONE)∪(~CLOCK∩↓DONE))/Z[8:15]←Y[0:7];   (2 32 37 %
40 51)

*****REWRITE 95 BY { T11A};

96  /(~LOAD∩(↓CLOCK∩~DONE))∪(~LOAD∩(~CLOCK∩↓DONE))/Z[8:15]←Y[0:7];   (%
2 32 37 40 51)

*****REWRITE 96 BY { M5};

97  /~LOAD∩(↓CLOCK∩~DONE)/Z[8:15]←Y[0:7]; ∧ /~LOAD∩(~CLOCK∩↓DONE)/Z[8:%
15]←Y[0:7];   (2 32 37 40 51)

*****∧E 97:#1;

98  /~LOAD∩(↓CLOCK∩~DONE)/Z[8:15]←Y[0:7];   (2 32 37 40 51)

*****VE T7B ↓CLOCK,~DONE;

99 (↓CLOCK∩~DONE)=(~DONE∩↓CLOCK)

*****SUBSTR 99 IN 98;

100  /~LOAD∩(~DONE∩↓CLOCK)/Z[8:15]←Y[0:7];   (2 32 37 40 51)
```

```
*****REWRITE 100 BY ( T10BR);

101  /(~LOAD∩~DONE)∩↓CLOCK/Z[8:15]←Y[0:7];   (2 32 37 40 51)

*****VE M11B ~LOAD,DONE,0;

102  /~LOAD/DONE←0; ⊃(~LOAD∩~DONE)=(~LOAD∩~0)

*****TAUT (~LOAD∩~DONE)=(~LOAD∩~0) 58,102;

103 (~LOAD∩~DONE)=(~LOAD∩~0)  (48)

*****REWRITE 103 BY REDUCE;

104 (~LOAD∩~DONE)=~LOAD  (48)

*****SUBSTR 104 IN 101;

105  /~LOAD∩↓CLOCK/Z[8:15]←Y[0:7];   (2 32 37 40 48 51)

*****REWRITE 43 BY REDUCE;

106  /LOAD∩↑CLOCK/Q[0:3]←Q[0:3]+1;   (43)

*****VE M2 H,Q[0:3]↔15;

107 H←Q[0:3]↔15; ⊃H=(Q[0:3]↔15)

*****TAUT H=(Q[0:3]↔15) 45,107;

108 H=(Q[0:3]↔15)  (45)

*****VE M2 G,~H;

109 G←~H; ⊃G=~H

*****TAUT G=~H 46,109;

110 G=~H  (46)

*****SUBSTR 108 IN 110;

111 G=~(Q[0:3]↔15)  (45 46)

*****REWRITE 49 BY REDUCE;

112  /LOAD∩↑G/DONE←1;   (49)

*****SUBSTR 111 IN 112;

113  /LOAD∩↑~(Q[0:3]↔15)/DONE←1;   (45 46 49)

*****REWRITE 113 BY TRANS;

114  /LOAD∩↓(Q[0:3]↔15)/DONE←1;   (45 46 49)

*****REWRITE  /LOAD∩(x∩↓(CLOCK∪DONE))/a←y;  BY TRANS;

115  /LOAD∩(x∩↓(CLOCK∪DONE))/a←y; ≡ /LOAD∩(x∩((↓CLOCK∩~DONE)∪(~CLOCK∩↓%
DONE)))/a←y;

*****REWRITE 115 BY { T11A};

116  /LOAD∩(x∩↓(CLOCK∪DONE))/a←y; ≡ /(LOAD∩(x∩(↓CLOCK∩~DONE)))∪(LOAD∩(%
x∩(~CLOCK∩↓DONE)))/a←y;

*****REWRITE 116 BY { M5};

117  /LOAD∩(x∩↓(CLOCK∪DONE))/a←y; ≡( /LOAD∩(x∩(↓CLOCK∩~DONE))/a←y; ∧ /%
LOAD∩(x∩(~CLOCK∩↓DONE))/a←y; )

*****VE T7B x,↓(CLOCK∪DONE);

118 (x∩↓(CLOCK∪DONE))=(↓(CLOCK∪DONE)∩x)

*****SUBSTR 118 IN 117;

119  /LOAD∩(↓(CLOCK∪DONE)∩x)/a←y; ≡( /LOAD∩(x∩(↓CLOCK∩~DONE))/a←y; ∧ /%
LOAD∩(x∩(~CLOCK∩↓DONE))/a←y; )

*****VE T7B x,↓CLOCK∩~DONE;

120 (x∩(↓CLOCK∩~DONE))=((↓CLOCK∩~DONE)∩x)

*****SUBSTR 120 IN 119;

121  /LOAD∩(↓(CLOCK∪DONE)∩x)/a←y; ≡( /LOAD∩((↓CLOCK∩~DONE)∩x)/a←y; ∧ /%
LOAD∩(x∩(~CLOCK∩↓DONE))/a←y; )

*****SUBST 93 IN 121;

122  /LOAD∩(↓CLK∩x)/a←y; ≡( /LOAD∩((↓CLOCK∩~DONE)∩x)/a←y; ∧ /LOAD∩(x∩(%
~CLOCK∩↓DONE))/a←y; )  (51)

*****REWRITE 122 BY { T10BR};

123  /(LOAD∩↓CLK)∩x/a←y; ≡( /((LOAD∩↓CLOCK)∩~DONE)∩x/a←y; ∧ /((LOAD∩x)%
∩~CLOCK)∩↓DONE/a←y; )  (51)

*****VI 123 x a y;

124 ∀x a y.( /(LOAD∩↓CLK)∩x/a←y; ≡( /((LOAD∩↓CLOCK)∩~DONE)∩x/a←y; ∧ /(%
(LOAD∩x)∩~CLOCK)∩↓DONE/a←y; ))  (51)

*****∧I (21 26);

125  /(LOAD∩↓CLK)∩Q[3]/Z[0:3]←OV&Z[0:2]; ∧ /(LOAD∩↓CLK)∩Q[3]/Z[4:7]←Z[%
3]&Z[4:6];   (21 26)

*****REWRITE 125 BY { M6};

126  /(LOAD∩↓CLK)∩Q[3]/Z[0:3]&Z[4:7]←(OV&Z[0:2])&(Z[3]&Z[4:6]);   (21 %
26)

*****REWRITE 126 BY { D5};

127  /(LOAD∩↓CLK)∩Q[3]/Z[0:3]&Z[4:7]←OV&(Z[0:2]&(Z[3]&Z[4:6]));   (21 %
26)
```

```
*****REWRITE 127 BY CONCATENATE;

128  /(LOAD∩↓CLK)∩Q[3]/Z[0:7]←OV&Z[0:6];   (21 26)

*****∧I (31 36);

129  /(1∩↓CLK)∩Q[3]/Z[8:11]←Z[7]&Z[8:10]; ∧ /(1∩↓CLK)∩Q[3]/Z[12:15]←Z[%
11]&Z[12:14];   (31 36)

*****REWRITE 129 BY { M6};

130  /(1∩↓CLK)∩Q[3]/Z[8:11]&Z[12:15]←(Z[7]&Z[8:10])&(Z[11]&Z[12:14]); %
  (31 36)

*****REWRITE 130 BY REDUCE;

131  /↓CLK∩Q[3]/Z[8:11]&Z[12:15]←(Z[7]&Z[8:10])&(Z[11]&Z[12:14]);   (3%
1 36)

*****REWRITE 131 BY CONCATENATE;

132  /↓CLK∩Q[3]/Z[8:15]←Z[7:14];   (31 36)

*****VE M10 ↓CLK∩Q[3],LOAD,Z[8:15],Z[7:14];

133  /↓CLK∩Q[3]/Z[8:15]←Z[7:14]; ⊃ /(↓CLK∩Q[3])∩LOAD/Z[8:15]←Z[7:14]; %


*****TAUT  /(↓CLK∩Q[3])∩LOAD/Z[8:15]←Z[7:14];  132:133;

134  /(↓CLK∩Q[3])∩LOAD/Z[8:15]←Z[7:14];   (31 36)

*****VE T7B ↓CLK∩Q[3],LOAD;

135 ((↓CLK∩Q[3])∩LOAD)=(LOAD∩(↓CLK∩Q[3]))

*****SUBSTR 135 IN 134;

136  /LOAD∩(↓CLK∩Q[3])/Z[8:15]←Z[7:14];   (31 36)

*****REWRITE 136 BY { T10BR};

137  /(LOAD∩↓CLK)∩Q[3]/Z[8:15]←Z[7:14];   (31 36)

*****∧I (128 137);

138  /(LOAD∩↓CLK)∩Q[3]/Z[0:7]←OV&Z[0:6]; ∧ /(LOAD∩↓CLK)∩Q[3]/Z[8:15]←Z%
[7:14];   (21 26 31 36)

*****REWRITE 138 BY { M6};

139  /(LOAD∩↓CLK)∩Q[3]/Z[0:7]&Z[8:15]←(OV&Z[0:6])&Z[7:14];   (21 26 31%
 36)

*****REWRITE 139 BY { D5};

140  /(LOAD∩↓CLK)∩Q[3]/Z[0:7]&Z[8:15]←OV&(Z[0:6]&Z[7:14]);   (21 26 31%
 36)
```

```
*****REWRITE 140 BY CONCATENATE;

141  /(LOADn↓CLK)nQ[3]/Z[0:15]←OV&Z[0:14];   (21 26 31 36)

*****REWRITE 18 BY REDUCE;

142  /LOADn↑C/OV←B;   (18)

*****VE M2 C,~CLK;

143 C←~CLK; ⊃C=~CLK

*****TAUT C=~CLK 15,143;

144 C=~CLK  (15)

*****SUBSTR 144 IN 142;

145  /LOADn↑~CLK/OV←B;   (15 18)

*****REWRITE 145 BY TRANS;

146  /LOADn↓CLK/OV←B;   (15 18)

*****VE M2 B,(DnA)nZ[15];

147 B←(DnA)nZ[15]; ⊃B=((DnA)nZ[15])

*****TAUT B=((DnA)nZ[15]) 14,147;

148 B=((DnA)nZ[15])  (14)

*****SUBSTR 148 IN 146;

149  /LOADn↓CLK/OV←(DnA)nZ[15];   (14 15 18)

*****VE M2 A,~Q[3];

150 A←~Q[3]; ⊃A=~Q[3]

*****TAUT A=~Q[3] 13,150;

151 A=~Q[3]  (13)

*****SUBSTR 151 IN 149;

152  /LOADn↓CLK/OV←(Dn~Q[3])nZ[15];   (13 14 15 18)

*****VE T7B D,~Q[3];

153 (Dn~Q[3])=(~Q[3]nD)

*****SUBSTR 153 IN 152;

154  /LOADn↓CLK/OV←(~Q[3]nD)nZ[15];   (13 14 15 18)

*****REWRITE 154 BY { T10B};
```

```
155  /LOAD∩↓CLK/OV←~Q[3]∩(D∩Z[15]);   (13 14 15 18)

*****VE M10 LOAD∩↓CLK,Q[3],OV,~Q[3]∩(D∩Z[15]);

156  /LOAD∩↓CLK/OV←~Q[3]∩(D∩Z[15]); ⊃ /(LOAD∩↓CLK)∩Q[3]/OV←~Q[3]∩(D∩Z[%
15]);

*****TAUT  /(LOAD∩↓CLK)∩Q[3]/OV←~Q[3]∩(D∩Z[15]);  155:156;

157  /(LOAD∩↓CLK)∩Q[3]/OV←~Q[3]∩(D∩Z[15]);   (13 14 15 18)

*****REWRITE 157 BY { M8};

158  /(LOAD∩↓CLK)∩Q[3]/OV←0;   (13 14 15 18)

*****∧I (158 141);

159  /(LOAD∩↓CLK)∩Q[3]/OV←0; ∧ /(LOAD∩↓CLK)∩Q[3]/Z[0:15]←OV&Z[0:14];  %
 (13 14 15 18 21 26 31 36)

*****REWRITE 159 BY { M6};

160  /(LOAD∩↓CLK)∩Q[3]/OV&Z[0:15]←0&(OV&Z[0:14]);   (13 14 15 18 21 26%
 31 36)

*****REWRITE 160 BY { 124};

161  /((LOAD∩↓CLOCK)∩~DONE)∩Q[3]/OV&Z[0:15]←0&(OV&Z[0:14]); ∧ /((LOAD∩%
Q[3])∩~CLOCK)∩↓DONE/OV&Z[0:15]←0&(OV&Z[0:14]);   (13 14 15 18 21 26 31%
 36 51)

*****∧E 161:#1;

162  /((LOAD∩↓CLOCK)∩~DONE)∩Q[3]/OV&Z[0:15]←0&(OV&Z[0:14]);   (13 14 1%
5 18 21 26 31 36 51)

*****∧I (9 11);

163 S[0:3]←(W[0:3]+Z[0:3])+E; ∧S[4:7]←(W[4:7]+Z[4:7])+0;   (9 11)

*****REWRITE 163 BY { M6};

164 S[0:3]&S[4:7]←((W[0:3]+Z[0:3])+E)&((W[4:7]+Z[4:7])+0);   (9 11)

*****REWRITE 164 BY CONCATENATE;

165 S[0:7]←((W[0:3]+Z[0:3])+E)&((W[4:7]+Z[4:7])+0);   (9 11)

*****VE M2 E,carry(W[4:7],Z[4:7],0);

166 E←carry(W[4:7],Z[4:7],0); ⊃E=carry(W[4:7],Z[4:7],0)

*****TAUT E=carry(W[4:7],Z[4:7],0) 12,166;

167 E=carry(W[4:7],Z[4:7],0)  (12)

*****SUBSTR 167 IN 165;

168 S[0:7]←((W[0:3]+Z[0:3])+carry(W[4:7],Z[4:7],0))&((W[4:7]+Z[4:7])+0%
);   (9 11 12)

*****VE A4 W,Z,0,0,3,7;

169 (((W[0:3]+Z[0:3])+carry(W[suc(3):7],Z[suc(3):7],0))&((W[suc(3):7]+%
Z[suc(3):7])+0))=((W[0:7]+Z[0:7])+0)

*****SIMPLIFY ;

170 (((W[0:3]+Z[0:3])+carry(W[4:7],Z[4:7],0))&((W[4:7]+Z[4:7])+0))=((W%
[0:7]+Z[0:7])+0)

*****SUBSTR 170 IN 168;

171 S[0:7]←(W[0:7]+Z[0:7])+0;   (9 11 12)

*****VE M2 S[0:7],(W[0:7]+Z[0:7])+0;

172 S[0:7]←(W[0:7]+Z[0:7])+0; ⊃S[0:7]=((W[0:7]+Z[0:7])+0)

*****TAUT S[0:7]=((W[0:7]+Z[0:7])+0) 171:172;

173 S[0:7]=((W[0:7]+Z[0:7])+0)  (9 11 12)

*****∧I (22 27);

174  /((LOAD∩↓CLK)∩~Q[3])∩Z[15]/Z[0:3]←S[0:3]; ∧ /((LOAD∩↓CLK)∩~Q[3])∩%
Z[15]/Z[4:7]←S[4:7];   (22 27)

*****REWRITE 174 BY { M6};

175  /((LOAD∩↓CLK)∩~Q[3])∩Z[15]/Z[0:3]&Z[4:7]←S[0:3]&S[4:7];   (22 27)

*****REWRITE 175 BY CONCATENATE;

176  /((LOAD∩↓CLK)∩~Q[3])∩Z[15]/Z[0:7]←S[0:7];   (22 27)

*****SUBSTR 173 IN 176;

177  /((LOAD∩↓CLK)∩~Q[3])∩Z[15]/Z[0:7]←(W[0:7]+Z[0:7])+0;   (9 11 12 2%
2 27)

*****VE M2 D,carry(W[0:3],Z[0:3],E);

178 D←carry(W[0:3],Z[0:3],E); ⊃D=carry(W[0:3],Z[0:3],E)

*****TAUT D=carry(W[0:3],Z[0:3],E) 10,178;

179 D=carry(W[0:3],Z[0:3],E)  (10)

*****SUBSTR 167 IN 179;

180 D=carry(W[0:3],Z[0:3],carry(W[4:7],Z[4:7],0))  (10 12)

*****VE A2 W,Z,0,0,3,7;

181 carry(W[0:3],Z[0:3],carry(W[suc(3):7],Z[suc(3):7],0))=carry(W[0:7]%
,Z[0:7],0)
```

```
*****SIMPLIFY ;

182 carry(W[0:3],Z[0:3],carry(W[4:7],Z[4:7],0))=carry(W[0:7],Z[0:7],0)%

*****SUBSTR 182 IN 180;

183 D=carry(W[0:7],Z[0:7],0)  (10 12)

*****REWRITE 152 BY { T10B};

184  /LOADn↓CLK/OV←Dn(~Q[3]nZ[15]);   (13 14 15 18)

*****VE T7B D;

185 ∀y.(Dny)=(ynD)

*****REWRITE 184 BY { 185};

186  /LOADn↓CLK/OV←(~Q[3]nZ[15])nD;   (13 14 15 18)

*****VE M10 LOADn↓CLK,~Q[3]nZ[15],OV,(~Q[3]nZ[15])nD;

187  /LOADn↓CLK/OV←(~Q[3]nZ[15])nD; ⊃ /(LOADn↓CLK)n(~Q[3]nZ[15])/OV←(~%
Q[3]nZ[15])nD;

*****TAUT  /(LOADn↓CLK)n(~Q[3]nZ[15])/OV←(~Q[3]nZ[15])nD;  186:187;

188  /(LOADn↓CLK)n(~Q[3]nZ[15])/OV←(~Q[3]nZ[15])nD;   (13 14 15 18)

*****REWRITE 188 BY { M7};

189  /(LOADn↓CLK)n(~Q[3]nZ[15])/OV←D;   (13 14 15 18)

*****REWRITE 189 BY { T10BR};

190  /((LOADn↓CLK)n~Q[3])nZ[15]/OV←D;   (13 14 15 18)

*****SUBSTR 183 IN 190;

191  /((LOADn↓CLK)n~Q[3])nZ[15]/OV←carry(W[0:7],Z[0:7],0);   (10 12 13%
 14 15 18)

*****∧I (191 177);

192  /((LOADn↓CLK)n~Q[3])nZ[15]/OV←carry(W[0:7],Z[0:7],0); ∧ /((LOADn↓%
CLK)n~Q[3])nZ[15]/Z[0:7]←(W[0:7]+Z[0:7])+0;   (9 10 11 12 13 14 15 18 %
22 27)

*****REWRITE 192 BY { M6};

193  /((LOADn↓CLK)n~Q[3])nZ[15]/OV&Z[0:7]←carry(W[0:7],Z[0:7],0)&((W[0%
:7]+Z[0:7])+0);   (9 10 11 12 13 14 15 18 22 27)

*****VE T10B LOADn↓CLK;

194 ∀y z.(((LOADn↓CLK)ny)nz)=((LOADn↓CLK)n(ynz))

*****REWRITE 193 BY { 194};
```

```
195  /(LOAD∩↓CLK)∩(~Q[3]∩Z[15])/OV&Z[0:7]←carry(W[0:7],Z[0:7],0)&((W[0%
:7]+Z[0:7])+0);   (9 10 11 12 13 14 15 18 22 27)

*****REWRITE 195 BY { 124};

196  /((LOAD∩↓CLOCK)∩~DONE)∩(~Q[3]∩Z[15])/OV&Z[0:7]←carry(W[0:7],Z[0:7%
],0)&((W[0:7]+Z[0:7])+0); ∧ /((LOAD∩(~Q[3]∩Z[15]))∩~CLOCK)∩↓DONE/OV&Z[%
0:7]←carry(W[0:7],Z[0:7],0)&((W[0:7]+Z[0:7])+0);   (9 10 11 12 13 14 1%
5 18 22 27 51)

*****∧E 196:#1;

197  /((LOAD∩↓CLOCK)∩~DONE)∩(~Q[3]∩Z[15])/OV&Z[0:7]←carry(W[0:7],Z[0:7%
],0)&((W[0:7]+Z[0:7])+0);   (9 10 11 12 13 14 15 18 22 27 51)

*****∧I (23 28);

198  /((LOAD∩↓CLK)∩~Q[3])∩~Z[15]/Z[0:3]←Z[0:3]; ∧ /((LOAD∩↓CLK)∩~Q[3])%
∩~Z[15]/Z[4:7]←Z[4:7];   (23 28)

*****REWRITE 198 BY { M6};

199  /((LOAD∩↓CLK)∩~Q[3])∩~Z[15]/Z[0:3]&Z[4:7]←Z[0:3]&Z[4:7];   (23 28%
)

*****REWRITE 199 BY CONCATENATE;

200  /((LOAD∩↓CLK)∩~Q[3])∩~Z[15]/Z[0:7]←Z[0:7];   (23 28)

*****VE M10 LOAD∩↓CLK,~Q[3]∩~Z[15],OV,(~Q[3]∩Z[15])∩D;

201  /LOAD∩↓CLK/OV←(~Q[3]∩Z[15])∩D; ⊃ /(LOAD∩↓CLK)∩(~Q[3]∩~Z[15])/OV←(%
~Q[3]∩Z[15])∩D;

*****TAUT  /(LOAD∩↓CLK)∩(~Q[3]∩~Z[15])/OV←(~Q[3]∩Z[15])∩D;  186,201;

202  /(LOAD∩↓CLK)∩(~Q[3]∩~Z[15])/OV←(~Q[3]∩Z[15])∩D;   (13 14 15 18)

*****REWRITE 202 BY { T108R};

203  /((LOAD∩↓CLK)∩~Q[3])∩~Z[15]/OV←(~Q[3]∩Z[15])∩D;   (13 14 15 18)

*****VE T7B ~Q[3],Z[15];

204 (~Q[3]∩Z[15])=(Z[15]∩~Q[3])

*****SUBSTR 204 IN 203;

205  /((LOAD∩↓CLK)∩~Q[3])∩~Z[15]/OV←(Z[15]∩~Q[3])∩D;   (13 14 15 18)

*****VE T10B Z[15];

206 ∀y z.((Z[15]∩y)∩z)=(Z[15]∩(y∩z))

*****REWRITE 205 BY { 206};

207  /((LOAD∩↓CLK)∩~Q[3])∩~Z[15]/OV←Z[15]∩(~Q[3]∩D);   (13 14 15 18)
```

```
*****VE M8 (LOAD∩↓CLK)∩~Q[3],OV,~Z[15],~Q[3]∩D;

208  /((LOAD∩↓CLK)∩~Q[3])∩~Z[15]/OV←~~Z[15]∩(~Q[3]∩D); ≡ /((LOAD∩↓CLK)%
∩~Q[3])∩~Z[15]/OV←0;

*****REWRITE 208 BY REDUCE;

209  /((LOAD∩↓CLK)∩~Q[3])∩~Z[15]/OV←Z[15]∩(~Q[3]∩D); ≡ /((LOAD∩↓CLK)∩~%
Q[3])∩~Z[15]/OV←0;

*****TAUT  /((LOAD∩↓CLK)∩~Q[3])∩~Z[15]/OV←0;  207,209;

210  /((LOAD∩↓CLK)∩~Q[3])∩~Z[15]/OV←0;   (13 14 15 18)

*****∧I (210 200);

211  /((LOAD∩↓CLK)∩~Q[3])∩~Z[15]/OV←0; ∧ /((LOAD∩↓CLK)∩~Q[3])∩~Z[15]/Z%
[0:7]←Z[0:7];   (13 14 15 18 23 28)

*****REWRITE 211 BY { M6};

212  /((LOAD∩↓CLK)∩~Q[3])∩~Z[15]/OV&Z[0:7]←0&Z[0:7];   (13 14 15 18 23%
 28)

*****REWRITE 212 BY { 194};

213  /(LOAD∩↓CLK)∩(~Q[3]∩~Z[15])/OV&Z[0:7]←0&Z[0:7];   (13 14 15 18 23%
 28)

*****REWRITE 213 BY { 124};

214  /((LOAD∩↓CLOCK)∩~DONE)∩(~Q[3]∩~Z[15])/OV&Z[0:7]←0&Z[0:7]; ∧ /((LO%
AD∩(~Q[3]∩~Z[15]))∩~CLOCK)∩↓DONE/OV&Z[0:7]←0&Z[0:7];   (13 14 15 18 23%
 28 51)

*****∧E 214:#1;

215  /((LOAD∩↓CLOCK)∩~DONE)∩(~Q[3]∩~Z[15])/OV&Z[0:7]←0&Z[0:7];   (13 1%
4 15 18 23 28 51)

*****REWRITE 33 BY REDUCE;

216  /(↓CLK∩~Q[3])∩~LOADY/Z[8:11]←Z[8:11];   (33)

*****REWRITE 38 BY REDUCE;

217  /(↓CLK∩~Q[3])∩~LOADY/Z[12:15]←Z[12:15];   (38)

*****∧I (216 217);

218  /(↓CLK∩~Q[3])∩~LOADY/Z[8:11]←Z[8:11]; ∧ /(↓CLK∩~Q[3])∩~LOADY/Z[12%
:15]←Z[12:15];   (33 38)

*****REWRITE 218 BY { M6};

219  /(↓CLK∩~Q[3])∩~LOADY/Z[8:11]&Z[12:15]←Z[8:11]&Z[12:15];   (33 38)

*****REWRITE 219 BY CONCATENATE;
```

```
220  /(↓CLK∩~Q[3])∩~LOADY/Z[8:15]←Z[8:15];   (33 38)

*****SUBSTR 72 IN 220;

221  /(↓CLK∩~Q[3])∩~~LOAD/Z[8:15]←Z[8:15];   (2 33 38)

*****REWRITE 221 BY REDUCE;

222  /(↓CLK∩~Q[3])∩LOAD/Z[8:15]←Z[8:15];   (2 33 38)

*****VE T7B ↓CLK∩~Q[3],LOAD;

223 ((↓CLK∩~Q[3])∩LOAD)≡(LOAD∩(↓CLK∩~Q[3]))

*****SUBSTR 223 IN 222;

224  /LOAD∩(↓CLK∩~Q[3])/Z[8:15]←Z[8:15];   (2 33 38)

*****REWRITE 224 BY { T10BR};

225  /(LOAD∩↓CLK)∩~Q[3]/Z[8:15]←Z[8:15];   (2 33 38)

*****REWRITE 225 BY { 124};

226  /((LOAD∩↓CLOCK)∩~DONE)∩~Q[3]/Z[8:15]←Z[8:15]; ∧ /((LOAD∩~Q[3])∩~C%
LOCK)∩↓DONE/Z[8:15]←Z[8:15];   (2 33 38 51)

*****∧E 226:#1;

227  /((LOAD∩↓CLOCK)∩~DONE)∩~Q[3]/Z[8:15]←Z[8:15];   (2 33 38 51)

*****REWRITE 5 BY REDUCE;

228  /~↑LOADX/W[0:3]←W[0:3];   (5)

*****REWRITE 8 BY REDUCE;

229  /~↑LOADX/W[4:7]←W[4:7];   (8)

*****∧I (228 229);

230  /~↑LOADX/W[0:3]←W[0:3]; ∧ /~↑LOADX/W[4:7]←W[4:7];   (5 8)

*****REWRITE 230 BY { M6};

231  /~↑LOADX/W[0:3]&W[4:7]←W[0:3]&W[4:7];   (5 8)

*****REWRITE 231 BY CONCATENATE;

232  /~↑LOADX/W[0:7]←W[0:7];   (5 8)

*****SUBSTR 60 IN 232;

233  /~↑~(LOAD∪CLOCK)/W[0:7]←W[0:7];   (1 5 8)

*****REWRITE 233 BY { T14A};

234  /~↑(~LOAD∩~CLOCK)/W[0:7]←W[0:7];   (1 5 8)
```

```
*****REWRITE 234 BY TRANS;
235  /~((↓LOAD∩~CLOCK)∪(~LOAD∩↓CLOCK))/W[0:7]←W[0:7];   (1 5 8)
*****REWRITE 44 BY REDUCE;
236  /LOAD∩~↑CLOCK/Q[0:3]←Q[0:3];   (44)
*****REWRITE 50 BY REDUCE;
237  /LOAD∩~↑G/DONE←DONE;   (50)
*****SUBSTR 111 IN 237;
238  /LOAD∩~↑~(Q[0:3]↔15)/DONE←DONE;   (45 46 50)
*****REWRITE 238 BY TRANS;
239  /LOAD∩~↓(Q[0:3]↔15)/DONE←DONE;   (45 46 50)
*****∧I (24 29);
240  /LOAD∩~↓CLK/Z[0:3]←Z[0:3]; ∧ /LOAD∩~↓CLK/Z[4:7]←Z[4:7];   (24 29)
*****REWRITE 240 BY { M6};
241  /LOAD∩~↓CLK/Z[0:3]&Z[4:7]←Z[0:3]&Z[4:7];   (24 29)
*****REWRITE 241 BY CONCATENATE;
242  /LOAD∩~↓CLK/Z[0:7]←Z[0:7];   (24 29)
*****∧I (34 39);
243  /1∩~↓CLK/Z[8:11]←Z[8:11]; ∧ /1∩~↓CLK/Z[12:15]←Z[12:15];   (34 39)
*****REWRITE 243 BY { M6};
244  /1∩~↓CLK/Z[8:11]&Z[12:15]←Z[8:11]&Z[12:15];   (34 39)
*****REWRITE 244 BY CONCATENATE;
245  /1∩~↓CLK/Z[8:15]←Z[8:15];   (34 39)
*****REWRITE 245 BY REDUCE;
246  /~↓CLK/Z[8:15]←Z[8:15];   (34 39)
*****VE M10 ~↓CLK,LOAD,Z[8:15],Z[8:15];
247  /~↓CLK/Z[8:15]←Z[8:15]; ⊃ /~↓CLK∩LOAD/Z[8:15]←Z[8:15];
*****TAUT  /~↓CLK∩LOAD/Z[8:15]←Z[8:15];  246:247;
248  /~↓CLK∩LOAD/Z[8:15]←Z[8:15];   (34 39)
*****VE T7B ~↓CLK,LOAD;
249 (~↓CLK∩LOAD)=(LOAD∩~↓CLK)
```

```
*****SUBSTR 249 IN 248;

250  /LOAD∩~↓CLK/Z[8:15]←Z[8:15];   (34 39)

*****∧I (242 250);

251  /LOAD∩~↓CLK/Z[0:7]←Z[0:7]; ∧ /LOAD∩~↓CLK/Z[8:15]←Z[8:15];   (24 2%
9 34 39)

*****REWRITE 251 BY { M6};

252  /LOAD∩~↓CLK/Z[0:7]&Z[8:15]←Z[0:7]&Z[8:15];   (24 29 34 39)

*****REWRITE 252 BY CONCATENATE;

253  /LOAD∩~↓CLK/Z[0:15]←Z[0:15];   (24 29 34 39)

*****REWRITE 19 BY REDUCE;

254  /LOAD∩~↑C/OV←OV;   (19)

*****SUBSTR 144 IN 254;

255  /LOAD∩~↑~CLK/OV←OV;   (15 19)

*****REWRITE 255 BY TRANS;

256  /LOAD∩~↓CLK/OV←OV;   (15 19)

*****∧I (256 253);

257  /LOAD∩~↓CLK/OV←OV; ∧ /LOAD∩~↓CLK/Z[0:15]←Z[0:15];   (15 19 24 29 %
34 39)

*****REWRITE 257 BY { M6};

258  /LOAD∩~↓CLK/OV&Z[0:15]←OV&Z[0:15];   (15 19 24 29 34 39)

*****SUBSTR 93 IN 258;

259  /LOAD∩~↓(CLOCK∪DONE)/OV&Z[0:15]←OV&Z[0:15];   (15 19 24 29 34 39 %
51)

*****REWRITE 259 BY TRANS;

260  /LOAD∩~((↓CLOCK∩~DONE)∪(~CLOCK∩↓DONE))/OV&Z[0:15]←OV&Z[0:15];   (%
15 19 24 29 34 39 51)
```

13. CONCLUSIONS

This thesis has demonstrated how theorem proving can be applied to showing the correctness of digital circuit designs. Using the FOL proof checker may, at first glance, appear to be an awkward and expensive process in terms of computer use. This is especially true for simple circuits. The correctness of the 4-bit binary counters could have been determined by any digital simulator in about 16 cycles.

On the other hand, the theorem proving approach becomes much more attractive when verifying complex circuits made up of MSI and LSI components. To exhaustively simulate the 8-bit multiplier would have required 2^{16} (65,536) multiplications at one load cycle and 16 clock cycles each. If we wanted to verify a 16-bit multiplier of similar design using simulation it would take 2^{32} (4,294,967,296) multiplications with one load cycle and 32 clock cycles each. Proving the correctness of a 16-bit multiplier using FOL might require twice as many steps as for the 8-bit unit, and possibly far less if certain quantifiers are properly manipulated.

After working with the FOL verifier for a short period it becomes surprisingly easy to use. There are a many features that could be added to the current system to make proofs of correctness for hardware much more concise. Automatic boolean minimization would help. Even better would be a subroutine similar the FOL tautology mechanism that can compare two boolean expressions and determine if they are equivalent or if one implies the other.

A considerable amount of work will have to be done to get the races and hazards information into the proof checker. It may be reasonable to develop some axioms that indicate under what conditions a circuit will be free of these problems. Another approach would be to include statistical information on component delays in the circuit description and attempt to determine the probability of a race or hazard. Some additional study of the properties of non-transitions is also needed.

Other improvements in hardware verification hinge on the adaptation of program verification techniques. It would be nice to be able to make assertions about what a device will do as a result of a sequence of input transitions. On an algorithmic level it is much easier to define processes using a procedural register transfer language. Therefore, it would be advantageous to develop a verifier based on a procedural register transfer language. Finally, there is the need to interface the language presented here with a microprogram verification system. This would make it possible for the user to prove the correctness of a microprogram in conjunction with a specific digital device.

14. REFERENCES

14.1 References on design verification

[1] Abdali,S.K. "On proving sequential machine designs.", IEEE Trans. Computers, vol. C-20, no. 4, Dec. 1971, pp. 1563-1566.

[2] Caplener,H.D. and Janku,J.A. "Improved modeling of computer hardware systems.", Computer Design, vol. 12, no. 8, Aug. 1973, pp. 59-64.

[3] Hoehne,H. and Piloty,R. "Design verification at the register transfer language level.", IEEE Trans. Computers, vol. C-24, no. 9, Sept. 1975, pp. 861-867.

[4] Losleben,P. "Design validation in hierarchical systems.", Proc. 12th Design Automation Conf., Boston, June 1975, pp. 431-438.

[5] Roth,J.P. "Verification of hardware designs at high level.", IBM Research Rept. RC 5613, Sept. 1975, 7 pp.

[6] Wagner,T.J. "Verification of hardware designs thru symbolic manipulation.", Proc. Symp. Design Automation and Microprocessors, Palo Alto, Ca., Feb. 1977, pp. 50-53.

14.2 References on design languages

[7] Baray,M.B. and Su,S.Y.H. "A digital system modelling philosophy and design language.", Proc. 8th Design Automation Workshop, Atlantic City, June 1971, pp. 1-22.

[8] Barbacci,M.R. "A comparison of register transfer languages for describing computers and digital systems.", Carnegie-Mellon Univ., Dept. of Computer Science Rept., March 1973, 42 pp.

[9] Barbacci,M.R. and Siewiorek,D.P. "Some aspects of the symbolic manipulation of computer descriptions.", Carnegie-Mellon Univ., Dept. of Computer Science Rept., July 1974, 25 pp.

[10] Bednar,G.M. and Tracey,J.H. "An asynchronous circuit design language (ACDL).", IEEE Trans. Computers, vol. C-23, no. 9, Sept. 1974, pp. 971-976.

[11] Bell,C.G. and Newell,A. "The PMS and ISP descriptive systems for computer structures.", Proc. Spring Joint Computer Conf., May 1970, pp. 351-374.

[12] Bell,C.G. and Newell,A. "Computer structures: readings and examples.", New York: McGraw Hill, 1970, 668 pp.

[13] Bingham,H.W. "Use of APL in microprogrammable machine modeling.", Proc. ACM SIGPLAN Symp. Languages for Systems Implementation, Lafayette, Ind., Oct. 1971, pp. 105-109.

[14] Bogo,G.; Guyot,A.; Lux,A.; Mermet,J. and Payan,C. "Cassandre and the computer aided logical systems design.", Proc. Int. Federation of Information Processing 1971, Ljubljana, Yugoslavia, Aug. 1971, pp. 1056-1065.

[15] Borrione,D. "LASCAR: A language for simulation of computer architecture,". Proc. 1975 Int. Symp. Computer Hardware Description Languages and their Applications, New York, Sept. 1975, pp. 143-152.

[16] Breuer,M.A. "Digital system design automation: languages, simulation, and data base.", Woodland Hills, Ca.: Computer Science Press, 1975, 417 pp.

[17] Chu,Y. "An Algol-like computer design language.", Comm. ACM, vol. 8, no. 10, Oct. 1965, pp. 607-615.

[18] Chu,Y. "Introducing the computer design language.", Digest IEEE Computer Society Conf., Sept. 1972, pp. 215-218.

[19] Chu,Y. "Computer organization and microprogramming.", Englewood Cliffs, N.J.: Prentice-Hall, 1972, 533 pp.

[20] Chu,Y. "Introducing CDL.", Computer, vol. 7, no. 12, Dec. 1974, pp. 31-33.

[21] Crall,R.F. "A formal design language for digital systems.", Computer Design, vol. 13, no. 11, Nov. 1974, pp. 103-108.

[22] Darringer,J.A. "The description, simulation, and automatic implementation of digital computer processors.", *Carnegie-Mellon Univ., Dept. of Computer Science,* Ph.D. thesis, May 1969, 328 pp.

[23] Dietmeyer,D.L. "Introducing DDL.", Computer, vol. 7, no. 12, Dec. 1974, pp. 34-38.

[24] Duley,J.R. and Dietmeyer,D.L. "A digital system design language (DDL).", IEEE Trans. Computers, vol. C-17, no.9, Sept. 1968, pp. 850-860.

[25] Figueroa,M.A. "Analysis of languages for the design of digital computers.", Univ. of Illinois, Urbana, Coordinated Science Lab., Masters thesis, May 1973, 133 pp.

[26] Franta,W.R. and Giloi,W.K. "APL*DS: A hardware description language for design and simulation,". Proc. 1975 Int. Symp. Computer Hardware Description Languages and their Applications, New York, Sept. 1975, pp. 45-52.

[27] Friedman,T.D. and Liu,C.H. "The design of a design language." Proc. Nat. Electronics Conf., vol. 26, Dec. 1970, pp. 89-93.

[28] Giloi,W.K. and Liebig,H. "A formalism for description and synthesis of logical algorithms and their hardware implementation.", IEEE Trans. Computers, vol. C-23, no. 9, Sept. 1974, pp. 897-906.

[29] Goerke,W. and Hoffman,H.J. "Simulation of switching circuits by SSM-a new hardware simulation language,". Proc. 1975 Int. Symp. Computer Hardware Description Languages and their Applications, New York, Sept. 1975, pp. 125-133.

[30] Hill,F.J. and Peterson,G.R. "Digital systems: hardware organization and design.", New York: John Wiley and Sons, 1973, 481 pp.

[31] Hill,F.J. "Introducing AHPL.", Computer, vol. 7, no. 12, Dec. 1974, pp. 28-30.

[32] Hill,F.J. "Updating AHPL.", Proc. 1975 Int. Symp. Computer Hardware Description Languages and their Applications, New York, Sept. 1975, pp. 22-29.

[33] Lee,J.A.N.; Macock,D.; Marks,P. and Wesselkamper,T.C. "The requirements for effective hardware description languages.", Virginia Polytechnic Institute and State Univ., Dept. of Computer Science, Rept. CS 75011-R, June 1975, 42 pp.

[34] Marczynski,R.W.; Pulczyn,W.T. and Sochacki,J.M. "OSM Microprogrammed hardware structure description language.", Proc. 1975 Int. Symp. Computer Hardware Description Languages and their Applications, New York, Sept. 1975, pp. 172-178.

[35] Parnas,D.L. "A language for describing the function of synchronous systems.", Comm. ACM, vol. 9, no. 2, Feb. 1966, pp. 72-76.

[36] Parnas,D.L. "More on simulation languages and design methodology for computer systems.", Proc. Spring Joint Computer Conf., May 1969, pp. 739-743.

[37] Parnas,D.L. and Darringer,J.A. "SODAS and a methodology for system design.", Proc. Fall Joint Computer Conf., Nov. 1967, pp. 449-479.

[38] Patil,S.S. and Dennis,J.B. "The description and realization of digital systems.", Digest IEEE Computer Society Conf., Sept. 1972, pp. 223-226.

[39] Potash,H. "A digital control design system", Univ. of California, Los Angeles, Dept. of Electrical Engineering, Ph.D. thesis, May 1969, 248 pp.

[40] Proctor,R.M. "A logic design translator experiment demonstrating relationships of language to systems and logic design.", IEEE Trans. Electronic Computers, vol. C-13, no. 4, Aug. 1964, pp. 422-430.

[41] Rammig,F.J. "DIGITEST II: An integrated structural and behavioral language.", Proc. 1975 Int. Symp. Computer Hardware Description Languages and their Applications, New York, Sept. 1975, pp. 38-44.

[42] Reed,I.S. "Symbolic design techniques applied to a generalized computer.", Computer, Vol. 5, no. 3, May/June 1972, pp. 47-52.

[43] Rose,C.W.; Bradshaw,F.T. and Katzke,S.W. "The LOGOS representation system.", Digest IEEE Computer Society Conf., Sept. 1972, pp. 187-190.

[44] Schlaeppi,H.P. "A formal language for describing machine logic, timing, and sequencing (LOTIS).", IEEE Trans. Electronic Computers, Vol. EC-13, no. 4, Aug. 1964, pp. 439-448.

[45] Schorr,H. "Computer-aided digital system design and analysis using a register transfer language.", IEEE Trans. Electronic Computers, vol. EC-13, no. 6, Dec. 1964, pp. 730-737.

[46] Siewiorek,D. "Introducing ISP.", Computer, vol. 7, no. 12, Dec. 1974, pp. 39-41.

[47] Siewiorek,D. "Introducing PMS.", Computer, vol. 7, no. 12, Dec. 1974, pp. 42-44.

[48] Smith,D.R. "Computer structure language (CSL).", Proc. 1975 Int. Symp. Computer Hardware Description Languages and their Applications, New York, Sept. 1975, pp. 153-160.

[49] Stabler,E.P. "System description languages.", IEEE Trans. Computers, vol. C-19, no. 12, Dec. 1970, pp. 1160-1173.

[50] Su,S.Y.H. and Carberry,R.L. "Design automation languages.", Proc. 5th Hawaii Int. Conf. on System Sciences, Honolulu, Jan. 1972, pp. 184-187.

[51] Su,S.Y.H. "A survey of computer hardware description languages in the USA.", Computer, vol. 7, no. 12, Dec. 1974, pp. 45-51.

[52] Whitney,G.E. and Tulloss,R.E. "The Best language: a language for use in simulation of digital computers.", Digest IEEE Computer Society Conf., Sept. 1972, pp. 211-214.

14.3 References on transitions

[53] Betancourt,R. and McCluskey,E.J. "Analysis of sequential circuits using clocked flip-flops.", Stanford Univ., Digital Systems Lab., Technical Note no. 82, Aug. 1975, 40 pp.

[54] Betancourt,R. "Analysis of sequential circuits using clocked flip-flops.", Stanford Univ., Dept. of Electrical Engineering, Ph.D. thesis, Oct. 1976, 109 pp.

[55] Gschwind,H.W. and McCluskey,E.J. "Design of digital computers.", (2nd revised edition), New York: Springer-Verlag, 1975, 585 pp.

[56] Morris,R.L. and Miller,J.R. "Designing with TTL integrated circuits.", New York: McGraw Hill, Texas Instruments Electronics Series, 1971, 322 pp.

[57] Smith,J.R. and Roth,C.H. "Analysis and synthesis of asynchronous sequential networks using edge-sensitive flip-flops.", IEEE Trans. Computers, vol. C-20, no. 8, Aug. 1971, pp. 847-855.

[58] Talantsev,A.D. "On the analysis and synthesis of certain electrical circuits by means of special logical operators.", Automation and Remote Control, vol. 20, no. 7, July 1959, pp. 874-883.

14.4 References on races and hazards

[59] Beister,J. "A unified approach to combinational hazards.", IEEE Trans. Computers, vol. C-23, no. 6, June 1974, pp. 566-575.

[60] Bredson,J.G. "On multiple input change hazard-free combinational switching circuits without feedback.", 14th Annual Symp. Switching Automata Theory, Iowa City, Oct. 1973, pp. 56-63.

[61] Caldwell,S.H. "Switching circuits and logical design.", New York: John Wiley and Sons, 1958, 686 pp.

[62] Chewning,D.R. and Bredt,T.H. "Hazards in asynchronous systems.", Stanford Univ., Stanford Electronics Labs., Rept. no. TR-52, Sept. 1972, 24 pp.

[63] Eichelberger,E.B. "Hazard detection in combinational and sequential switching circuits.", IBM J. Research and Development, vol. 9, no. 2, March 1965, pp. 90-99.

[64] Harrison,R.A. and Olson,D.J. "Race analysis of digital systems without logic simulation.", Proc. 8th Design Automation Workshop, Atlantic City, June 1971, pp. 82-94.

[65] Hill,F.J. and Peterson,G.R. "Introduction to switching theory and logical design.", New York: John Wiley and Sons, 1968, 212 pp.

[66] Huffman,D.A. "The design and use of hazard-free switching networks.", J. ACM, vol. 4, no. 1, Jan. 1957, pp. 47-62.

[67] Langdon,G.G. "Analysis of asynchronous circuits under different delay assumptions.", IEEE Trans. Computers, vol. C-17, no. 12, Dec. 1968, pp. 1131-1143.

[68] McCluskey,E.J. "Transients in combinational logic circuits.", in "Redundancy techniques for computing systems.", Wilcox,R.H. and Mann,W.C. editors, Washington: Spartan books, 1962, pp. 9-46.

[69] McCluskey,E.J. "Introduction to the theory of switching circuits.", New York: McGraw Hill, 1965, 318 pp.

[70] McGhee,R.B. "Some aids to the detection of hazards in combinational switching circuits.", IEEE Trans. Computers, vol. C-18, no. 6, June 1969, pp. 561-566.

[71] Meisel,W.S. and Kashee,R.S. "Hazards in asynchronous sequential circuits.", IEEE Trans. Computers, vol. C-18, no. 8, Aug. 1969, pp. 752-759.

[72] Muller,D.E. "Treatment of transition signals in electronic switching circuits by algebraic methods.", IRE Trans. Elect. Computers, vol. EC-8, no. 3, Sept. 1959, pp. 401.

[73] Rey,C. "Algebra finds logic circuit glitches.", Electronic Design, vol. 22, no. 4, Feb. 1974, pp. 90-92.

[74] Unger,S.H. "Hazards and delays in asynchronous sequential switching circuits.", IRE Trans. Circuit Theory, vol. CT-6, no. 1, 1959, pp. 12-25.

[75] Unger,S.H. "Asynchronous sequential switching circuits.", New York: John Wiley and Sons, 1969, 290 pp.

[76] Yoeli,M. and Rinon,S. "Application of ternary algebra to the study of static hazards.", Journal ACM, vol. 11, no. 1, Jan. 1964, pp. 84-97.

14.5 References on microprogram verification

[77] Birman,A. "Correctness in design: the S-machine experiment.", IBM Research Rept. RC 4193, Jan. 1973, 53 pp.

[78] Birman,A. "On proving correctness of microprograms.", IBM J. Research and Development, vol. 14, no. 3, May 1974, pp. 250-266.

[79] Birman,A. and Joyner,W.H. "MVS - A system for microprogram validation - part 1: the skeleton.", IBM Research Rept. RC 4923, July 1974, 26 pp.

[80] Bouricius,W.G. "Procedure for testing microprograms.", IBM Research Rept. RC 4905, June 1974, 19 pp.

[81] Carter,W.C. "Experiments in proving design correctness for microprogram controlled computers.", Digest Int. Symp. Fault Tolerant Computing, Champaign, June 1974, pp. 5.22-5.27.

[82] Leeman,G.B.; Carter,W.C. and Birman,A. "Some techniques for microprogram validation.", IBM Research Rept. RC 4616, April 1974, 5 pp.

[83] Leeman,G.B. "Some problems in certifying microprograms.", IEEE Trans. Computers, vol. C-24, no. 5, May 1975, pp. 545-553.

[84] Patterson,D.A. "Strum: structured microprogram development system for correct firmware.", IEEE Trans. Computers, vol. C-25, no. 10, Oct. 1976, pp. 974-985.

[85] Ramamoorthy,C.V. and Shankar,K.S. "Automatic testing for the correctness and equivalence of loopfree microprograms.", IEEE Trans. Computers, vol. C-23, no. 8, Aug. 1974, pp. 768-782.

14.6 FOL references

[86] Aiello,M. and Weyhrauch,R.W. "Checking proofs in the metamathematics of first order logic.", Stanford Univ., Computer Science Dept., Artificial Intelligence Lab. Memo AIM-222, Aug. 1974, 51 pp.

[87] Weyhrauch,R.W. and Thomas,A.J. "FOL: a proof checker for first-order logic.", Stanford Univ., Computer Science Dept., Artificial Intelligence Lab. Memo AIM-235, Sept. 1974, 56 pp.

[88] Filman,R.E. and Weyhrauch,R.W. "An FOL primer.", Stanford Univ., Computer Science Dept., Artificial Intelligence Lab. Memo AIM-288, Sept. 1976, 34 pp.

[89] Weyhrauch,R.W. "A users manual for FOL.", Stanford Univ., Computer Science Dept., Artificial Intelligence Lab. Memo AIM-277, (to appear), 68 pp.

Appendix A. BNF LANGUAGE DESCRIPTION

```
<letter>
        ::= A|B|C|...|Z|'|_

<digit>
        ::= 0|1|2|3|4|5|6|7|8|9

<transition>
        ::= ↑|↓

<unary_oper>
        ::= ¬|-

<binary_oper>
        ::= ∧|v|⊕|+|-

<arith_relation>
        ::= ≤|≥|<|>|≠|=

<identitier>
        ::= <letter>
        ::= <identifier><letter>
        ::= <identifier><digit>

<constant>
        ::= <digit>
        ::= <constant><digit>

<statement>
        ::= <var>←<exp>;
        ::= /<cond_exp>/<var>←<exp>;

<cond_exp>
        ::= <identifier>
        ::= <identifier>[<exp>]
        ::= <identifier>[<exp>,<exp>]
        ::= <transition><cond_exp>
        ::= <unary_oper><cond_exp>
        ::= <cond_exp><binary_oper><cond_exp>
        ::= <exp><arith_relation><exp>

<exp>
        ::= <constant>
        ::= <var>
        ::= <unary_oper><exp>
        ::= <exp><binary_oper><exp>
        ::= <exp><arith_relation><exp>
        ::= <exp>&<exp>
        ::= (<exp>)
        ::= (<cond_exp>)
```

```
<var>
          ::= <identifier>
          ::= <identifier>[<exp>]
          ::= <identifier>[<exp>,<exp>]
          ::= <identifier>[<constant>:<constant>]
          ::= <identifier>[<constant>:<constant>,<exp>]
          ::= <var>&<var>
          ::= <identifier>[*<exp>]
          ::= <identifier>[*<exp>,<exp>]
          ::= <identifier>[<exp>,*<exp>]
          ::= <identifier>[<constant>:<constant>,*<exp>]
```

Appendix B. LIST OF TRANSFORMATIONS

T1. $\neg 0 \equiv 1$
$\neg 1 \equiv 0$

T2. $X \vee 1 \equiv 1$
$X \wedge 0 \equiv 0$

T3. $X \vee 0 \equiv X$
$X \wedge 1 \equiv X$

T4. $X \vee X \equiv X$
$X \wedge X \equiv X$

T5. $\neg(\neg X) \equiv X$

T6. $X \vee \neg X \equiv 1$
$X \wedge \neg X \equiv 0$

T7. $X \vee Y \equiv Y \vee X$
$X \wedge Y \equiv Y \wedge X$

T8. $X \vee (X \wedge Y) \equiv X$
$X \wedge (X \vee Y) \equiv X$

T9. $(X \vee \neg Y) \wedge Y \equiv X \wedge Y$
$(X \wedge \neg Y) \vee Y \equiv X \vee Y$

T10. $(X \vee Y) \vee Z \equiv X \vee (Y \vee Z)$
$(X \wedge Y) \wedge Z \equiv X \wedge (Y \wedge Z)$

T11. $X \wedge (Y \vee Z) \equiv (X \wedge Y) \vee (X \wedge Z)$
$X \vee (Y \wedge Z) \equiv (X \vee Y) \wedge (X \vee Z)$

T12. $(X \vee Y) \wedge (\neg X \vee Z) \wedge (Y \vee Z) \equiv (X \vee Y) \wedge (\neg X \vee Z)$
$(X \wedge Y) \vee (\neg X \wedge Z) \vee (Y \wedge Z) \equiv (X \wedge Y) \vee (\neg X \wedge Z)$

T13. $(X \vee Y) \wedge (\neg X \vee Z) \equiv (X \wedge Z) \vee (\neg X \wedge Y)$

T14. $\neg(X \vee Y) \equiv \neg X \wedge \neg Y$
$\neg(X \wedge Y) \equiv \neg X \vee \neg Y$

X1. $(X \wedge \neg Y) \vee (\neg X \wedge Y) \equiv X \oplus Y$

X2. $X \oplus 0 \equiv X$

X3. $X \oplus 1 \equiv \neg X$

X4. $X \oplus X \equiv 0$

X5. $X \oplus \neg X \equiv 1$

X6. $X \oplus Y \equiv Y \oplus X$

X7. $(X \oplus Y) \oplus Z \equiv X \oplus (Y \oplus Z)$

X8. $\neg(X \oplus Y) \equiv \neg X \oplus Y$

```
D1.  A[i] ≡ A[i:i]

D2.  A[i:j] ≡ A[i:j,0]

D3.  A[i:j,s]&A[j+1:k,s] ≡ A[i:k,s]
     where i≤j<k

D4.  ¬A[i:j,s]&¬A[j+1:k,s] ≡ ¬A[i:k,s]
     where i≤j<k

D5.  (X&Y)&Z ≡ X&(Y&Z)

D6.  (X[i:j]∧Y[i:j])&(X[j+1:k]∧Y[j+1:k]) ≡ X[i:k]∧Y[i:k]
     where i≤j<k

D7.  (X[i:j]∨Y[i:j])&(X[j+1:k]∨Y[j+1:k]) ≡ X[i:k]∨Y[i:k]
     where i≤j<k

D8.  (X[i:j]●Y[i:j])&(X[j+1:k]●Y[j+1:k]) ≡ X[i:k]●Y[i:k]
     where i≤j<k

D9.  /X/ A[Y[i:j],s]←Z;
                ≡
     /X∧(Y[i:j]=n)/ A[n,s]←Z;

D10. /X/ A[s,Y[i:j]]←Z;
                ≡
     /X∧(Y[i:j]=n)/ A[s,n]←Z;

D11. A[Y[i:j],s] ≡
     (A[0,s]∧(Y[i:j]=0))∨(A[1,s]∧(Y[i:j]=1))∨...∨
     (A[n,s]∧(Y[i:j]=n))
     when A[Y[i:j],s] is used in an expression

D12. A[s,Y[i:j]] ≡
     (A[s,0]∧(Y[i:j]=0))∨(A[s,1]∧(Y[i:j]=1))∨...∨
     (A[s,n]∧(Y[i:j]=n))
     when A[s,Y[i:j]] is used in an expression


A1.  X[i]∧Y[i]∨(X[i]∨Y[i])∧CI ≡ carry(X[i],Y[i],CI)
     X[i]∧Y[i]∨(X[i]∨Y[i])∧carry(X[i+1:j],Y[i+1:j],CI) ≡
     carry(X[i:j],Y[i:j],CI)

A2.  carry(X[i:j],Y[i,j],carry(X[j+1:k],Y[j+1,k],CI) ≡
     carry(X[i:k],Y[i:k],CI)
     where i≤j<k

A3.  (X[i]●Y[i]●CARRY(X[i+1:j],Y[i+1:j],CI)&
     (X[i+1]●Y[i+1]●CARRY(X[i+2:j],Y[i+2:j],CI)&...&
     (X[j]●Y[j]●CI) ≡ X[i:j]+Y[i:j]+CI

A4.  (X[i:j]+Y[i:j]+CARRY(X[j+1,k],Y[j+1,k],CI))&
     (X[j+1:k]+Y[j+1:k]+CI) ≡ X[i:k]+Y[i:k]+CI
     where i≤j<k
```

```
A5.  X[i:j]+¬Y[i:j]+1 ≡ X[i:j]-Y[i:j] (for 2's complement)
     X[i:j]+¬Y[i:j]+0 ≡ X[i:j]-Y[i:j] (for 1's complement)

A6.  X[i]⊛(X[i+1]∧...∧X[j])&X[i+1]⊛(X[i+2]∧...∧X[j])&...&¬X[j]
     ≡ X[i:j]+1

A7.  ¬X[i]⊛Y[i] ≡ X[i]=Y[i]

A8.  (X[i]=Y[i])∧(X[i+1]=Y[i+1])∧...∧(X[j]=Y[j]) ≡ X[i:j]=Y[i:j]

A9.  (X[i:j]=Y[i:j])∧(X[j+1:k]=Y[j+1:k]) ≡ X[i:k]=Y[i:k]
     where i≤j<k

A10. ¬(X[i:j]=Y[i:j]) ≡ X[i:j]≠Y[i:j]

A11. X[i]∧¬Y[i] ≡ X[i]>Y[i]

A12. (X[i]>Y[i])∨((X[i]=Y[i])∧X[i+1]>Y[i+1])∨...∨
     ((X[i:j-1]=Y[i:j-1])∧X[j]>Y[j]) ≡ X[i:j]>Y[i:j]

A13. (X[i:j]>Y[i:j])∨((X[i:j]=Y[i:j])∧X[j+1:k]>Y[j+1:k])
     ≡ X[i:k]>Y[i:k]
     where i≤j<k

A14. Y[i:j]>X[i:j] ≡ X[i:j]<Y[i:j]

A15. ¬(X[i:j]>Y[i:j]) ≡ X[i:j]≤Y[i:j]

A16. ¬(Y[i:j]>X[i:j]) ≡ X[i:j]≥Y[i:j]


M1.  A←Y; ≡ /1/ A←Y;

M2.  /1/ A←Y; ⊃ (A ≡ Y)

M3.  /W∧¬X/ A←Y;
     /W∧ X/ A←Z;
             ≡
     /W/ A←(¬X∧Y)∨(X∧Z);

M4.  /X/ A←...Y...;
             ≡
     /X∧ Y/ A←...1...;
     /X∧¬Y/ A←...0...;

M5.  /X∨Y/ A←Z;
        ≡
     /X/ A←Z;
     /Y/ A←Z;

M6.  /X/ A←Y;             (note: A and Y must be the same
     /X/ B←Z;              length, same for B and Z)
        ≡
     /X/ A&B←Y&Z;
```

```
M7.  /X/ A←...Y...;
     and (X⊃Y)
          ≡
     /X/ A←...1...;

M8.  /X/ A←..¬Y...;
     and (X⊃Y)
          ≡
     /X/ A←...0...;

M9.  /X/ A←...B...;
     /Y/ B←Z;
     and (X⊃Y)
         ⊃
     /X/ A←...Z...;

M10. /X/ A←Z;
         ⊃
     /X∧Y/ A←Z;

M11. /X/ A←Y;
         ⊃
     X∧A ≡ X∧Y;
     X∧¬A ≡ X∧¬Y;


F1.  /X/ A[*Y[i:j],s]←Z;
              ≡
     /X∧(Y[i:j]≠n)/ A[n,s]←Z;

F2.  /X/ A[s,*Y[i:j]]←Z;
              ≡
     /X∧(Y[i:j]≠n)/ A[s,n]←Z;

F3.  /X/ A[*Y[i:j],s]←A[*Y[i:j],s];
              ≡
     /X∧(Y[i:j]≠n)/ A[n,s]←A[n,s];

F4.  /X/ A[s,*Y[i:j]]←A[s,*Y[i:j]];
              ≡
     /X∧(Y[i:j]≠n)/ A[s,n]←A[s,n];


C1.  ↑0 ≡ 0
     ↑1 ≡ 0
     ↓0 ≡ 0
     ↓1 ≡ 0

C2.  ↑X∧↓X ≡ 0

C3.  ↑X∧↑Y ≡ 0
     ↑X∧↓Y ≡ 0
     ↓X∧↑Y ≡ 0
     ↓X∧↓Y ≡ 0
```

C4. $\uparrow X \oplus \uparrow Y \equiv \uparrow X \vee \uparrow Y$
$\uparrow X \oplus \downarrow Y \equiv \uparrow X \vee \downarrow Y$
$\downarrow X \oplus \uparrow Y \equiv \downarrow X \vee \uparrow Y$
$\downarrow X \oplus \downarrow Y \equiv \downarrow X \vee \downarrow Y$

C5. $\uparrow\uparrow X \equiv \uparrow X$
$\uparrow\downarrow X \equiv \downarrow X$
$\downarrow\uparrow X \equiv \uparrow X$
$\downarrow\downarrow X \equiv \downarrow X$

C6. $\uparrow\neg X \equiv \downarrow X$
$\downarrow\neg X \equiv \uparrow X$

C7. $\uparrow(X \wedge Y) \equiv (\uparrow X \wedge Y) \vee (X \wedge \uparrow Y)$
$\downarrow(X \wedge Y) \equiv (\downarrow X \wedge Y) \vee (X \wedge \downarrow Y)$

C8. $\uparrow(X \vee Y) \equiv (\uparrow X \wedge \neg Y) \vee (\neg X \wedge \uparrow Y)$
$\downarrow(X \vee Y) \equiv (\downarrow X \wedge \neg Y) \vee (\neg X \wedge \downarrow Y)$

C9. $/X/\ A \leftarrow Y;$
$\supset$
$\uparrow A \equiv (X \wedge \uparrow Y) \vee (\neg A \wedge Y \wedge \uparrow X)$
$\downarrow A \equiv (X \wedge \downarrow Y) \vee (A \wedge \neg Y \wedge \uparrow X)$

Appendix C. TRANSITIONS THRU HAZARDS

STATIC 1 HAZARD

1. $\uparrow(X \wedge Y \vee \neg X \wedge Z) =$

$\uparrow(X \wedge Y) \wedge \neg(\neg X \wedge Z) \vee \uparrow(\neg X \wedge Z) \wedge \neg(X \wedge Y) =$

$(\uparrow X \wedge Y \vee X \wedge \uparrow Y) \wedge (X \vee \neg Z) \vee (\downarrow X \wedge Z \vee \neg X \wedge \uparrow Z) \wedge (\neg X \vee \neg Y) =$

$(\uparrow X \wedge Y \wedge X) \vee (\uparrow X \wedge Y \wedge \neg Z) \vee (X \wedge \uparrow Y) \vee (X \wedge \uparrow Y \wedge \neg Z) \vee$
$(\downarrow X \wedge Z \wedge \neg X) \vee (\downarrow X \wedge Z \wedge \neg Y) \vee (\neg X \wedge \uparrow Z) \vee (\neg X \wedge \uparrow Z \wedge \neg Y) =$

$(\uparrow X \wedge X \wedge Y) \vee (\uparrow X \wedge Y \wedge \neg Z) \vee (X \wedge \uparrow Y) \vee (\downarrow X \wedge \neg X \wedge Z) \vee (\downarrow X \wedge \neg Y \wedge Z) \vee (\neg X \wedge \uparrow Z)$

STATIC 1 HAZARD WITH CONSENSUS GATE ADDED

2. $\uparrow(X \wedge Y \vee \neg X \wedge Z \vee Y \wedge Z) =$

$\uparrow(X \wedge Y) \wedge \neg(\neg X \wedge Z) \wedge \neg(Y \wedge Z) \vee$
$\uparrow(\neg X \wedge Z) \wedge \neg(X \wedge Y) \wedge \neg(Y \wedge Z) \vee$
$\uparrow(Y \wedge Z) \wedge \neg(X \wedge Y) \wedge \neg(\neg X \wedge Z) =$

$\uparrow(X \wedge Y) \wedge (X \vee \neg Z) \wedge (\neg Y \vee \neg Z) \vee$
$\uparrow(\neg X \wedge Z) \wedge (\neg X \vee \neg Y) \wedge (\neg Y \vee \neg Z) \vee$
$\uparrow(Y \wedge Z) \wedge (\neg X \vee \neg Y) \wedge (X \vee \neg Z) =$

$(\uparrow X \wedge Y \vee X \wedge \uparrow Y) \wedge (X \wedge \neg Y \vee \neg Z) \vee$
$(\downarrow X \wedge Z \vee \neg X \wedge \uparrow Z) \wedge (\neg X \wedge \neg Z \vee \neg Y) \vee$
$(\uparrow Y \wedge Z \vee Y \wedge \uparrow Z) \wedge (\neg X \wedge \neg Z \vee X \wedge \neg Y) =$

$(\uparrow X \wedge Y \wedge \neg Z) \vee (X \wedge \uparrow Y \wedge \neg Y) \vee (X \wedge \uparrow Y \wedge \neg Z) \vee$
$(\downarrow X \wedge \neg Y \wedge Z) \vee (\neg X \wedge \uparrow Z \wedge \neg Z) \vee (\neg X \wedge \neg Y \wedge \uparrow Z) \vee$
$(X \wedge \uparrow Y \wedge \neg Y \wedge Z) \vee (\neg X \wedge Y \wedge \uparrow Z \wedge \neg Z) =$

$(\uparrow X \wedge Y \wedge \neg Z) \vee (X \wedge \uparrow Y \wedge \neg Y) \vee (X \wedge \uparrow Y \wedge \neg Z) \vee (\downarrow X \wedge \neg Y \wedge Z) \vee (\neg X \wedge \uparrow Z \wedge \neg Z) \vee (\neg X \wedge \neg Y \wedge \uparrow Z)$

STATIC 0 HAZARD

3. $\uparrow((X \vee Y) \wedge (\neg X \vee Z)) =$

$\uparrow(X \vee Y) \wedge (\neg X \vee Z) \vee \uparrow(\neg X \vee Z) \wedge (X \vee Y) =$

$(\uparrow X \wedge \neg Y \vee \neg X \wedge \uparrow Y) \wedge (\neg X \vee Z) \vee (\downarrow X \wedge \neg Z \vee X \wedge \uparrow Z) \wedge (X \vee Y) =$

$(\uparrow X \wedge \neg Y \wedge \neg X) \vee (\uparrow X \wedge \neg Y \wedge Z) \vee (\neg X \wedge \uparrow Y) \vee (\neg X \wedge \uparrow Y \wedge Z) \vee$
$(\downarrow X \wedge \neg Z \wedge X) \vee (\downarrow X \wedge \neg Z \wedge Y) \vee (X \wedge \uparrow Z) \vee (X \wedge \uparrow Z \wedge Y) =$

$(\uparrow X \vee \neg X \wedge \neg Y) \vee (\uparrow X \wedge \neg Y \wedge Z) \vee (\neg X \wedge \uparrow Y) \vee (\downarrow X \wedge X \wedge \neg Z) \vee (\downarrow X \wedge Y \wedge \neg Z) \vee (X \wedge \uparrow Z)$

```
STATIC 0 HAZARD WITH CONSENSUS GATE ADDED

4. ↑((X ∨ Y)∧(¬X ∨ Z)∧(Y ∨ Z)) =

   ↑(X ∨ Y)∧(¬X ∨ Z)∧(Y ∨ Z) ∨
   ↑(¬X ∨ Z)∧(X ∨ Y)∧(Y ∨ Z) ∨
   ↑(Y ∨ Z)∧(X ∨ Y)∧(¬X ∨ Z) =

   (↑X∧¬Y ∨ ¬X∧↑Y)∧(¬X∧Y ∨ Z) ∨
   (↓X∧¬Z ∨ X∧↑Z)∧(X∧Z ∨ Y) ∨
   (↑Y∧¬Z ∨ ¬Y∧↑Z)∧(X∧Z ∨ ¬X∧Y) =

   (↑X∧¬Y∧Z)∨(¬X∧↑Y∧Y)∨(¬X∧↑Y∧Z)∨
   (↓X∧Y∧Z)∨(X∧↑Z∧Z)∨(X∧Y∧↑Z)∨
   (¬X∧↑Y∧Y∧¬Z)∨(X∧¬Y∧↑Z∧Z) =

   (↑X∧¬Y∧Z)∨(¬X∧↑Y∧Y)∨(¬X∧↑Y∧Z)∨(↓X∧Y∧Z)∨(X∧↑Z∧Z)∨(X∧Y∧↑Z)
```

Appendix D. FOL DECLARATIONS AND AXIOMS

```
DECLARE PREDCONST BOOL 1;
DECLARE PREDCONST F(BOOL,BOOL,BOOL);
DECLARE INDVAR a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z;
DECLARE OPCONST ~(BOOL)=BOOL [R←850];
DECLARE OPCONST ↑(BOOL)=BOOL [R←750];
DECLARE OPCONST ↓(BOOL)=BOOL [R←750];
DECLARE OPCONST &(BOOL,BOOL)=BOOL [L←600,R←650];
DECLARE OPCONST +(BOOL,BOOL)=BOOL [L←500,R←550];
DECLARE OPCONST -(BOOL,BOOL)=BOOL [L←500,R←550];
DECLARE OPCONST ↔(BOOL,BOOL)=BOOL [L←400,R←450];
DECLARE OPCONST ≠(BOOL,BOOL)=BOOL [L←400,R←450];
DECLARE OPCONST >(BOOL,BOOL)=BOOL [L←400,R←450];
DECLARE OPCONST <(BOOL,BOOL)=BOOL [L←400,R←450];
DECLARE OPCONST ≥(BOOL,BOOL)=BOOL [L←400,R←450];
DECLARE OPCONST ≤(BOOL,BOOL)=BOOL [L←400,R←450];
DECLARE OPCONST ∩(BOOL,BOOL)=BOOL [L←300,R←350];
DECLARE OPCONST ∪(BOOL,BOOL)=BOOL [L←200,R←250];
DECLARE OPCONST •(BOOL,BOOL)=BOOL [L←200,R←250];
DECLARE OPCONST carry(BOOL,BOOL,BOOL)=BOOL;
DECLARE OPCONST sub(BOOL,NATNUM,NATNUM)=BOOL;
DECLARE OPCONST suc(NATNUM)=NATNUM;
REPRESENT {NATNUM} AS NATNUMREP;
ATTACH suc TO (LAMBDA (X)(ADD1 X));

AXIOM T1A: ~0=1;;
AXIOM T1B: ~1=0;;
AXIOM T2A: ∀x. x∪1=1  ∀x. 1∪x=1;;
AXIOM T2B: ∀x. x∩0=0  ∀x. 0∩x=0;;
AXIOM T3A: ∀x. x∪0=x  ∀x. 0∪x=x;;
AXIOM T3B: ∀x. x∩1=x  ∀x. 1∩x=x;;
AXIOM T4A: ∀x. x∪x=x;;
AXIOM T4B: ∀x. x∩x=x;;
AXIOM T5:  ∀x. ~(~x)=x;;
AXIOM T6A: ∀x. x∪~x=1  ∀x. ~x∪x=1;;
AXIOM T6B: ∀x. x∩~x=0  ∀x. ~x∩x=0;;
REDUCE←LOGICTREE ∪{T1A,T1B,T2A,T2B,T3A,T3B,T4A,T4B,T5,T6A,T6B};

AXIOM T7B: ∀x y. x∩y=y∩x;;
AXIOM T10B: ∀x y z. (x∩y)∩z=x∩(y∩z);;
AXIOM T10BR: ∀x y z. x∩(y∩z)=(x∩y)∩z;;
AXIOM T11A: ∀x y z. x∩(y∪z)=(x∩y)∪(x∩z);;
AXIOM T14A: ∀x y. ~(x∪y)=~x∩~y;;
AXIOM T14B: ∀x y. ~(x∩y)=~x∪~y;;

AXIOM X1: ∀x y. x∩~y∪~x∩y=x•y;;
AXIOM X6: ∀x y. x•y=y•x;;
AXIOM D3: ∀x i j k. sub(x,i,j)&sub(x,suc(j),k)=sub(x,i,k);;
AXIOM D5: ∀x y z. (x&y)&z=x&(y&z);;
AXIOM A2: ∀x y c i j k.
carry(sub(x,i,j),sub(y,i,j),carry(sub(x,suc(j),k),sub(y,suc(j),k),c))=
carry(sub(x,i,k),sub(y,i,k),c);;
AXIOM A4: ∀x y c i j k.
(sub(x,i,j)+sub(y,i,j)+carry(sub(x,suc(j),k),sub(y,suc(j),k),c))&
(sub(x,suc(j),k)+sub(y,suc(j),k)+c)=(sub(x,i,k)+sub(y,i,k)+c);;
AXIOM A6: ∀w x y z. (w•(x∩y∩z))&(x•(y∩z))&(y•z)&(~z)=w&x&y&z+1;;
AXIOM M2: ∀x y. (F(1,x,y)⊃(x≡y));;
AXIOM M3: ∀w x a y z. (F(w∩x,a,y)∧F(w∩~x,a,z)≡F(w,a,x∩y∪~x∩z));;
AXIOM M5: ∀x y a z. (F(x∪y,a,z)≡F(x,a,z)∧F(y,a,z));;
AXIOM M6: ∀x a y b z. (F(x,a,y)∧F(x,b,z)≡F(x,a&b,y&z));;
AXIOM M7: ∀x a y z. (F(x∩y,a,y∩z)≡F(x∩y,a,z));;
AXIOM M8: ∀x a y z. (F(x∩y,a,~y∩z)≡F(x∩y,a,0));;
AXIOM M10: ∀x y a z. (F(x,a,z)⊃F(x∩y,a,z));;
AXIOM M11A: ∀x a y. (F(x,a,y)⊃(x∩a=x∩y));;
AXIOM M11B: ∀x a y. (F(x,a,y)⊃(x∩~a)=(x∩~y));;

AXIOM C1A: ↑0=0;;
AXIOM C1B: ↑1=0;;
AXIOM C1C: ↓0=0;;
AXIOM C1D: ↓1=0;;
AXIOM C3A: ∀x y. ↑x∩↑y=0;;
AXIOM C3B: ∀x y. ↑x∩↓y=0;;
AXIOM C3C: ∀x y. ↓x∩↑y=0;;
AXIOM C3D: ∀x y. ↓x∩↓y=0;;
AXIOM C5A: ∀x. ↑↑x=↑x;;
AXIOM C5B: ∀x. ↑↓x=↓x;;
AXIOM C5C: ∀x. ↓↑x=↑x;;
AXIOM C5D: ∀x. ↓↓x=↓x;;
AXIOM C6A: ∀x. ↑~x=↓x;;
AXIOM C6B: ∀x. ↓~x=↑x;;
AXIOM C7A: ∀x y. ↑(x∩y)=(↑x∩y)∪(x∩↑y);;
AXIOM C7B: ∀x y. ↓(x∩y)=(↓x∩y)∪(x∩↓y);;
AXIOM C8A: ∀x y. ↑(x∪y)=(↑x∩~y)∪(~x∩↑y);;
AXIOM C8B: ∀x y. ↓(x∪y)=(↓x∩~y)∪(~x∩↓y);;
TRANS←LOGICTREE ∪{C1A,C1B,C1C,C1D,C3A,C3B,C3C,C3D,C5A,C5B,C5C,C5D,
C6A,C6B,C7A,C7B,C8A,C8B};

AXIOM C9A: ∀x a y. (F(x,a,y)⊃(↑a=(x∩↑y)∪(~a∩y∩↑x)));;
AXIOM C9B: ∀x a y. (F(x,a,y)⊃(↓a=(x∩↓y)∪(a∩~y∩↑x)));;

AXIOM CON: ∀x i j. (sub(x,i,0)&sub(x,1,j)=sub(x,i,j))
           ∀x i j. (sub(x,i,1)&sub(x,2,j)=sub(x,i,j))
           ∀x i j. (sub(x,i,2)&sub(x,3,j)=sub(x,i,j))
           ∀x i j. (sub(x,i,3)&sub(x,4,j)=sub(x,i,j))
           ∀x i j. (sub(x,i,4)&sub(x,5,j)=sub(x,i,j))
           ∀x i j. (sub(x,i,5)&sub(x,6,j)=sub(x,i,j))
           ∀x i j. (sub(x,i,6)&sub(x,7,j)=sub(x,i,j))
           ∀x i j. (sub(x,i,7)&sub(x,8,j)=sub(x,i,j))
           ∀x i j. (sub(x,i,8)&sub(x,9,j)=sub(x,i,j))
           ∀x i j. (sub(x,i,9)&sub(x,10,j)=sub(x,i,j))
           ∀x i j. (sub(x,i,10)&sub(x,11,j)=sub(x,i,j))
           ∀x i j. (sub(x,i,11)&sub(x,12,j)=sub(x,i,j))
           ∀x i j. (sub(x,i,12)&sub(x,13,j)=sub(x,i,j))
           ∀x i j. (sub(x,i,13)&sub(x,14,j)=sub(x,i,j))
           ∀x i j. (sub(x,i,14)&sub(x,15,j)=sub(x,i,j));;
CONCATENATE←LOGICTREE ∪{CON};
```

Appendix E. COMPONENT DEFINITIONS

```
Combinational devices

Q←¬(X∨Y);                   (7402)
Q←¬X;                       (7404)
Q←X∧Y;                      (7408)
Q←X∧Y∧Z;                    (7411)
Q←X∨Y;                      (7432)

Type D flip-flop            (7474)

/¬S/ Q←1;
/S∧¬R/ Q←0;
/S∧R∧↑C/ Q←D;
/S∧R∧¬↑C/ Q←Q;
/¬R/ Q'←1;
/R/ Q'←¬Q;

Edge triggered JK           (74103)

/¬R/ Q←0;
/R∧↓C/ Q←(J∧¬Q)∨(¬K∧Q);
/R∧¬↓C/ Q←Q;

4-bit binary counter        (74161)

/¬RESET/ Q[0:3]←0;
/RESET∧¬LOAD∧↑COUNT/ Q[0:3]←DATA[0:3];
/RESET∧¬LOAD∧¬↑COUNT/ Q[0:3]←Q[0:3];
/RESET∧LOAD∧ENBP∧ENBT∧↑COUNT/ Q[0:3]←Q[0:3]+1;
/RESET∧LOAD∧¬(ENBP∧ENBT∧↑COUNT)/ Q[0:3]←Q[0:3];
CRY←(Q[0:3]=15);

Quad type-D flip-flop       (74175)

/¬RESET/ Q[0:3]←0;
/RESET∧↑C/ Q[0:3]←D[0:3];
/RESET∧¬↑C/ Q[0:3]←Q[0:3];

4-bit shift register        (74179)

/¬CLEAR/ Q[0:3]←0;
/CLEAR∧↓CLOCK∧SHIFT/ Q[0:3]←SERIAL&Q[0:2];
/CLEAR∧↓CLOCK∧¬SHIFT∧LOAD/ Q[0:3]←DATA[0:3];
/CLEAR∧↓CLOCK∧¬SHIFT∧¬LOAD/ Q[0:3]←Q[0:3];
/CLEAR∧¬↓CLOCK/ Q[0:3]←Q[0:3];

4-bit binary adder          (74283)

SUM[0:3]←A[0:3]+B[0:3]+CI;
CO←CARRY(A[0:3],B[0:3],CI);
```