A Temporal Language for Hardware Simulation, Specification and Verification

Tong Gao Tang

September, 1988

CMU-CS-88-194 2

Tong Gao Tang
Dept. of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

This research was sponsored in part by the National Science Foundation under Grant Number CCR-8722633. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation or of the U.S. Government.

A Temporal Language for Hardware Simulation, Specification and Verification

Tong Gao Tang
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213 USA
September 23, 1988

Abstract

This paper describes a temporal logic for the simulation, specification and verification of digital circuits. This language is a general purpose programming language with temporal formulas as its Boolean expressions. The temporal operators include both future-time operators and past-time operators. These past-time formulas can be used for simulation and the future-time formulas can be used for verification. In this paper, we will deal with hardware on an abstract level. For example, a logic gate is an abstraction of a concrete circuit regardless whether it is TTL or CMOS. Our temporal language is sufficiently powerful that it can be used to describe the abstract devices for many synchronous circuits. These software devices can be used to simulate the entire synchronous circuits quite easily. We can also efficiently verify some properties of the circuits using the temporal language.

Table of Contents

1. Introduction

Formal methods have already played an important role in hardware design and verification [3,4,7]. Ben Moszkowski's ITL language [7] is an outstanding example of the use of temporal logic in hardware design. It permits direct reasoning about signals, devices and circuit behavior. The intervals of time provide a unifying means for presenting various features. His logic is good for describing the initial and final states of a subinterval as well as the intervening stable behavior. However, it is not suitable for describing the precise changes that some logic circuits make. Furthermore, the time sequences in ITL are all future-time sequences, never past-time sequences. Thus, ITL can not be used to simulate logic circuits, and it is not suitable for automatic verification of Integrated Circuits without additional effort [5].

In this paper, a new temporal language is introduced which can be used for both simulation and automatic verification of logical circuits. This language is a general purpose programming language with temporal formulas as its Boolean expressions. Past-time temporal formulas such as $\bullet A$, $\blacklozenge A$, $\blacksquare A$ and *Since*(A,B) can be used for simulation, and future-time temporal formulas such as $\bigcirc A$, $\lozenge A$, $\square A$ and *Until*(A,B) can be used for verification.

A device is assumed to have the following two characteristics:

1. There is a certain logical relationship between input and output.
2. There is certain propagation delay time from a time the input gets some signals to another time the output gives out some expected signals.

A *logical circuit* consists of several logical devices and their connections. In this paper, logical devices will be modeled by software descriptions which we call *abstract devices*. These abstract devices act just like their hardware counterparts, so that we can use these abstract devices as blocks to simulate many digital circuits for design. By verifying these abstract devices, we can verify many properties of actual digital circuits.

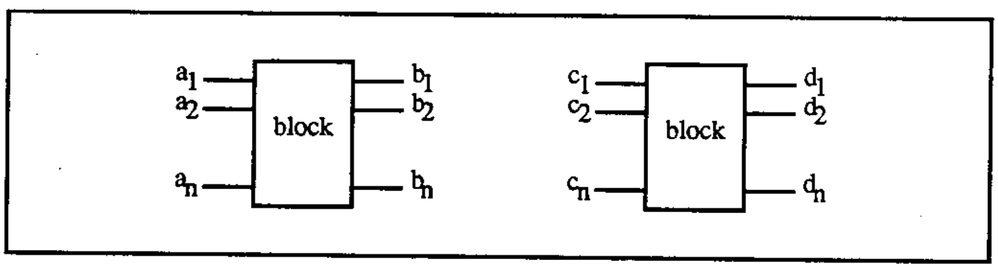


Figure 1-1: A Logical Circuit

To construct abstract devices a temporal language is introduced which adds time parameters into the normal temporal language. Simplicity and readability are still retained in the language. The temporal language introduces symbols for phrases such as **next time, sometime, always, until, last time, previous times, once, since,** and so on. We can use the temporal language to write down an easily understandable description of various devices. The temporal operators can be followed by a time parameter which specifies an exact duration of the operator. These time parameters permit a more precise and accurate description of the circuit to be made.

Since this temporal language integrates an algorithmic language with a temporal logic, it can be used for both simulation and verification without having to change the form of the description. We only need to take a different time frame of reference to satisfy the different goals. We sometimes we use an real time frame of reference (introduced in Section 3) for simulation of logical circuits; in another times we use the clock time frame of reference (i.e. the resting times discussed in Section 6). Our compiler implements both time frames very easily.

2. A Quantized Temporal Language For Hardware Design

Some temporal languages are designed for computing[8,9], while others are specially designed for verification [1,7]. Our Quantized Temporal Language has been designed as both a general purpose programming language (for simulation) and as a temporal logic (for verification).

A formula needs a truth value when it is in a *guard* G_i in a *case* statement, so we begin by describing the formulas of our language which can be either true or false. Let *expr* and *expr1* be two arithmetic expressions. Then *expr* = *expr1* is an atomic formula which is true if the two expressions have the same value. Similarly, *expr*<*expr1* is also an atomic formula that is true if *expr* is less than *expr1*. The expression *x is expr* is a special atomic formula that always is true and stores the value of *expr* in the variable *x*. *Now*(t) is an atomic formula that is true if the present time is t.

We define the set of truth formulas as follows:

- an atomic formula is a formula.
- If A and B are formulas then $\neg A$, $A \& B$, $A \vee B$ are formulas.
- If A and B are formulas then $\bigcirc A$, $\bigcirc[n]A$, $\Diamond A$, $\Diamond[n]A$, $\Box A$, $\Box[n]A$, *Until*(A,B), *Until*$[\leq n](A,B)$, *Until*$[\geq n](A,B)$ are formulas for $n \geq 0$. (The meanings of these temporal formulas are explained at the end of this section)
- If A and B are formulas then $\bullet A$, $\bullet[n]A$, $\blacklozenge A$, $\blacklozenge[n]A$, $\blacksquare A$, $\blacksquare[n]A$, *Since*(A,B), *Since*$[\leq n](A,B)$, *Since*$[\geq n](A,B)$ are formulas for $n \geq 1$. (The meanings of these temporal formulas are explained at the end of this section)

Now, we can describe the statements of our language. p := *expr* is an assignment statement that assigns the value of *expr* to the variable p. *return(expr)* is an assignment statement which sets the value of a function call.

We define statements as follows:

- An assignment statement is a statement.
- If A, B,... are statements, and G is a formula, then

  {A;B} is a statement;
  while (G) A is a statement.
  for (<expression 1>; <expression 2>; <expression 3>) A is a statement.
  ...

- If G_1, G_2,..., G_n are formulas, and A_1, A_2, ..., A_n, A_{n+1} are statements, then the following expression is a statement:

```
Case(
    G1 -: A1 ;
    G2 -: A2 ;
      ...
    Gn -: An ;
    otherwise -: An+1
)
```

The *case* statement acts like the following statements in C :

```
if (G1) A1 ;
else if (G2) A2 ;
     ...
else if (Gn) An ;
else An+1 ;
```

We allow some of the A_i to be empty. Thus, if $G_1 \vee G_2 \vee ... \vee G_n$ is true, we permit to have the statement:

```
Case(
     G1     ;
     G2 -: A2 ;
        ...
     Gn -: An ;
    )
```

As a comparison, we use the form G -: A in this language like the form $G \rightarrow A$ in Logic. The premises in statements of PROLOG appear at the back of the statements, so we have to use the symbol $\leftarrow$ in the statement $A \leftarrow G$ and we have the form A :- G in PROLOG. We save the symbol ":" for defining type. For example, for every integer variable *a* we must indicate in programs that **var** *a* : *integer*. The order of testing formulas in a program is from left to right.

In case statements, if a guard G_i is $A\&B$, we can write A,B instead. The priorities of operations are as follows:

Operator	Priority
$\wedge$	5
$\vee$	4
,	3
-:	2
;	1

We assume that there is a time frame of reference. The time sequence is t_0=0, t_1=1, t_2=2,.... For clarity we use the time sequence t_0,t_1, t_2,.... instead of a state sequence s_0, s_1, ..., s_i in this paper. For a temporal variable p there are values $p[0]$, $p[1]$, $p[2]$,... respectively at different times t_0=0, t_1=1, t_2=2,.... So, we have to use an array to actually implement the temporal variable p. If we talk about a temporal variable p at a time t_i, then p has the value $p[t_i]$, and the notation $\bigcirc p$ represents $p[t_{i+1}]$ and $\bullet p$ represents $p[t_{i-1}]$. A variable X is not a temporal variable, and it can maintain its value at different times. For the variable X there are not notations $\bigcirc X$ and $\bullet X$.

If a formula A is true at time t_i, we write $t_i \models A$. For a sequence $\tau = t_0, t_1, \ldots$, we define the truths of formulas inductively as follows:

$t_i \models Now(t)$	iff	$t = t_i$.	
$t_i \models A \& B$	iff	$t_i \models A$ first, and then $t_i \models B$.	
$t_i \models A \vee B$	iff	$t_i \models A$, otherwise $t_i \models B$.	
$t_i \models \bigcirc A$	iff	$t_{i+1} \models A$.	(next time)
$t_i \models \bigcirc[n]A$	iff	$t_{i+n} \models A$.	
$t_i \models \Diamond A$	iff	$t_i \models A$, otherwise $t_{i+1} \models \Diamond A$.	(sometime)
$t_i \models \Diamond[0]A$	iff	$t_i \models A$.	
$t_i \models \Diamond[n]A$	iff	$t_i \models A$, otherwise $t_{i+1} \models \Diamond[n-1]A$.	
$t_i \models \Box A$	iff	$t_i \models A$ first, and then $t_{i+1} \models \Box A$.	(always)
$t_i \models \Box[0]A$	iff	$t_i \models A$.	
$t_i \models \Box[n]A$	iff	$t_i \models A$ first, and then $t_{i+1} \models \Box[n-1]A$.	
$t_i \models Until(A,B)$	iff	$t_i \models B$, otherwise $t_i \models A \& \bigcirc Until(A,B)$.	(A until B)
$t_i \models Until[\geq 0](A,B)$	iff	$t_i \models Until(A,B)$.	
$t_i \models Until[\geq n+1](A,B)$	iff	$t_i \models \Box[n]A$ first, and then $t_{i+n+1} \models Until(A,B)$.	
$t_i \models Until[\leq 0](A,B)$	iff	$t_i \models B$.	
$t_i \models Until[\leq n+1](A,B)$	iff	$t_i \models B$, otherwise $t_i \models A \& \bigcirc Until[\leq n](A,B)$.	
$t_i \models \bullet A$	iff	$t_{i-1} \models A$ where $0<i$.	(last time)
$t_i \models \bullet[n]A$	iff	$t_{i-n} \models A$ where $1 \leq n \leq i$.	
$t_i \models \blacklozenge A$	iff	$t_{i-1} \models A$, otherwise $t_{i-1} \models \blacklozenge A$.	(once A)
$t_i \models \blacklozenge[1]A$	iff	$t_{i-1} \models A$.	
$t_i \models \blacklozenge[n+1]A$	iff	$t_{i-1} \models A$, otherwise $t_{i-1} \models \blacklozenge[n]A$ where $1 \leq n \leq i$.	
$t_0 \models \blacksquare A$	iff	true.	(the previous times)
$t_i \models \blacksquare A$	iff	$t_{i-1} \models A$ first, and then $t_{i-1} \models \blacksquare A$ where $0<i$.	(the previous times)
$t_i \models \blacksquare[1]A$	iff	$t_{i-1} \models A$.	
$t_i \models \blacksquare[n+1]A$	iff	$t_{i-1} \models A$ first, and then $t_{i-1} \models \blacksquare[n]A$ where $1 \leq n \leq i$.	
$t_0 \models Since(A,B)$	iff	false.	(A since B and not now)
$t_i \models Since(A,B)$	iff	$t_{i-1} \models A$ first, and then $t_{i-1} \models B \vee \bullet Since(A,B)$ where $0<i$.	(A since B and not now)

$t_i \models Since[\geq 1](A,B)$ iff $t_i \models Since(A,B)$.

$t_i \models Since[\geq n+1](A,B)$ iff $t_i \models ■[n]A$ first, and then $t_{i-n} \models Since(A,B)$ where $1 \leq n \leq i$.

$t_i \models Since[\leq 1](A,B)$ iff $t_{i-1} \models A \& B$.

$t_i \models Since[\leq n+1](A,B)$ iff $t_{i-1} \models A$ first, and then $t_{i-1} \models B \vee ● Since[\leq n](A,B)$ where $1 \leq n \leq i$.

We define the following abbreviations:

$□[m,n]A \equiv ■[m]A \;\&\; □[n]A$,

$Stable[0,n](x) \equiv □[n\text{-}1](x = ○x)$ where n>0 and x is a temporal variable.

$Stable[m,0](x) \equiv ■[m](x = ○x)$ where m>0 and x is a temporal variable.

$Stable[m,n](x) \equiv Stable[m,0](x) \& Stable[0,n](x)$ where x is a temporal variable.

$Stable[m,n](x)$ means that the values of x maintain stable in previous m times and after $n+1$ times including the current time.

3. Abstract Devices

In the operation of an electronic circuit, a small amount of time is needed for the electronic devices within the circuit to change logic levels. This effect leads to the gate characteristic called *propagation delay*. Propagation delay tells the amount of time it will take before the output of a gate switches logic levels after the input logic levels are set. For an inverter gate, it is the delay from a point on the input waveform to the same point on the output waveform. This point may typically be chosen half-way between a logical low level and a logical high level. Two delay times are specified. One, t_{plh}, is the propagation delay time when the output changes from a low state to a high state. The maximum value of the t_{plh} rating of a 7400 NAND gate is specified to be 22 nanoseconds (22 ns). While the other, t_{phl}, is the propagation delay time when the output changes from a high state to a low state. The maximum value of the t_{phl} rating of a 7400 NAND gate is specified to be 15ns. Generally, the maximum propagation delay value is specified, because it indicates the worst-case switching speed, and all 7400 NAND devices will work in 22ns or faster.

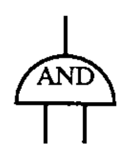
The propagation delay time of a simple gate is used as **a unit time**. For example, the gates **NOT**, **AND** and **OR** could be considered as gates that have 1 delay time approximately. For usual application , We don't need to make a distinction among them in delay time. The propagation delay of more complicated blocks will be a multiple of this unit time. This gives us our time frame of reference, with times 0, 1, 2, ..., which is called **a real time frame of reference.** We use $\bigcirc A$ to indicate that the formula A will be held at the next time (1 unit time later), and use $\bullet A$ to indicate that the formula A held at the last time (1 unit time before). Now, we introduce the following abstract devices which correspond to the appropriate logical circuits.

NOT GATE

NOT(*input,output*) ≡
Case(
  ●(*input* = 1) -: *output* := 0;
  ●(*input* = 0) -: *output* := 1;
  otherwise -: *output* := *unknown*
)

AND GATE

AND(*input1,input2,output*) ≡
Case(
  ●(*input1* = 1,*input2* = 1) -: *output* := 1;
  ●(*input1* = 0 ∨ *input2* = 0) -: *output* := 0;
  otherwise -: *output* := *unknown*
)

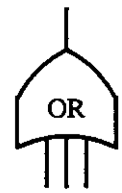


AND(*input1,input2,input3,output*) ≡
Case(
●(*input1* = 1,*input2* = 1,*input3* = 1) -: *output* := 1;
●(*input1* = 0 ∨ *input2* = 0 ∨ input3 = 0) -: *output* := 0;
otherwise -: *output* := *unknown*
)

OR GATE

OR(*input1,input2,output*) ≡
Case(
●(*input1* = 1 ∨ *input2* = 1) -: *output* := 1;
●(*input1* = 0,*input2* = 0) -: *output* := 0;
otherwise -: *output* := *unknown*
)

OR(*input1,input2,input3,output*) ≡
Case(
●(*input1* = 1 ∨ *input2* = 1 ∨ *input3* = 1) -: *output* := 1;
●(*input1* = 0,*input2* = 0,*input3* = 0) -: *output* := 0;
otherwise -: *output* := *unknown*
)

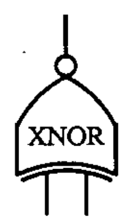


XOR GATE

XOR(*input1,input2,output*) ≡
Case(
●(*input1* = 0,*input2* = 0) -: *output* := 0;
●(*input1* = 0,*input2* = 1) -: *output* := 1;
●(*input1* = 1,*input2* = 0) -: *output* := 1;
●(*input1* = 1,*input2* = 1) -: *output* := 0;
otherwise -: *output* := *unknown*
)

XNOR GATE

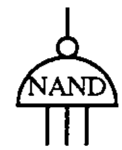
XNOR(*input1,input2,output*) ≡
Case(
●(*input1* = 0,*input2* = 0) -: *output* := 1;
●(*input1* = 0,*input2* = 1) -: *output* := 0;
●(*input1* = 1,*input2* = 0) -: *output* := 0;
●(*input1* = 1,*input2* = 1) -: *output* := 1;
otherwise -: *output* := *unknown*
)



NAND GATE

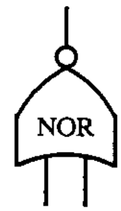
NAND(*input1,input2,output*) ≡
Case(
●(*input1* = 1,*input2* = 1) -: *output* := 0;
●(*input1* = 0 ∨ *input2* = 0) -: *output* := 1;
otherwise -: *output* := *unknown*
)

NAND(*input1,input2,input3,output*) ≡
Case(
●(*input1* = 1,*input2* = 1,*input3* = 1) -: *output* := 0;
●(*input1* = 0 ∨ *input2* = 0 ∨ *input3* = 0) -: *output* := 1;
otherwise -: *output* := *unknown*
)

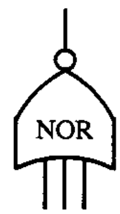


NOR GATE

NOR(*input1,input2,output*) ≡
Case(
●(*input1* = 1 ∨ *input2* = 1) -: *output* := 0;
●(*input1* = 0,*input2* = 0) -: *output* := 1;
otherwise -: *output* := *unknown*
)



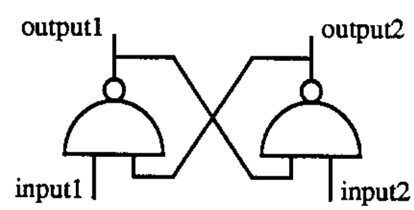
NOR(*input1,input2,input3,output*) ≡
Case(
●(*input1* = 1 ∨ *input2* = 1 ∨ *input3* = 1) -: *output* := 0;
●(*input1* = 0,*input2* = 0,*input3* = 0) -: *output* := 1;
otherwise -: *output* := *unknown*
)



RS FLIP-FLOP

RS1(*input1,input2,output2,output1*) ≡
Case(
●(*input1* = 0) -: *output1* := 1;
●(*input1* = 1),●(*X is output2*) -: *output1* := *abs(X)*;
otherwise -: *output1* := *unknown*
)

RS2(*input1,input2,output1,output2*) ≡
Case(
●(*input2* = 0) -: *output2* := 1;
●(*input2* = 1),●(*X is output1*) -: *output2* := *abs(X)*;
otherwise -: *output2* := *unknown*
)



The *abs(x)* is defined as follows:

abs(x) ≡
Case(
x = *unknown* -: *return(unknown)*;
otherwise -: *return*($|x-1|$)
)

It is a special definition of function. This function takes value from the *return(expr)*.

4. Simulations of Sequential Circuits

In a digital circuit, points connected by a wire always have the exact same voltage. Therefore, we can represent all of these points by a single point. Of course, no two representative points can be connected by a wire. These points sometimes have different voltages at different times. For instance, A point p_i gets voltages $p_i[0]$, $p_i[1]$, $p_i[2]$,..., which comprise an array, corresponding to times 0, 1, 2, ... respectively. So any point p_i in a circuit can be considered to be a temporal variable with an implicit parameter t. This time parameter must be considered when calculating the values of p_i. As an example, we now analyse a D Flip-flop constructed by six NAND chips (See Figure 4-1).

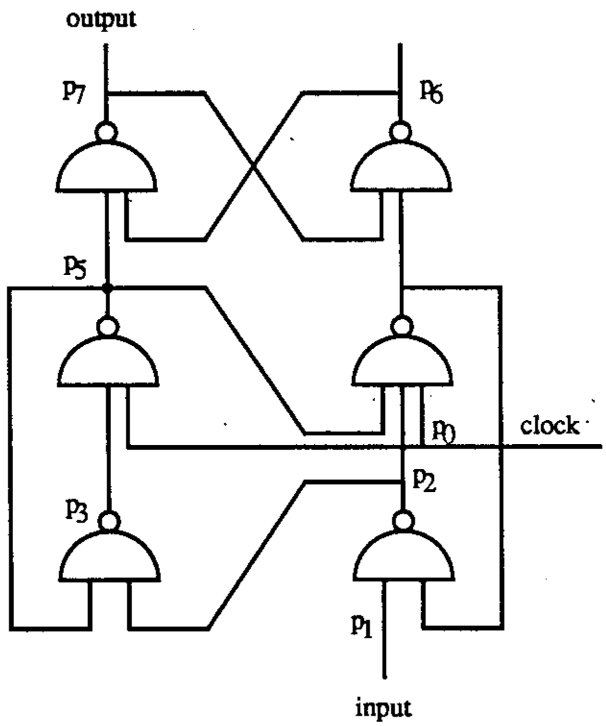


Figure 4-1: A D Flip-flop

1. If the input requirement is the following:
   a. For the point p_0, the input signal is
      CONDITION0(p_0) ≡ p_0 is 0 if the time is smaller than 10, otherwise p_0 is 1.
   b. For the point p_1, the input signal is
      CONDITION1(p_1) ≡ p_1 is 1 at any time.

The detailed input requirement will be discussed in Section 6. The 1a indicates that there is a positive-edge signal of clock pulse to be added at the point p_0. The 1b indicates that there is an input signal to be added at the point p_1. Both CONDITION0(p_0) and CONDITION1(p_1) could be treated as two additional abstract devices.

the D Flip-flop is equivalent to the following program which consists of six abstract devices, NAND, with their connections and the input conditions.

CONDITION0(p_0). { a procedure for calculating p_0 }
CONDITION1(p_1). { a procedure for calculating p_1 }
NAND(p_1,p_4,p_2). { a procedure for calculating p_2 }
NAND(p_2,p_5,p_3). { a procedure for calculating p_3 }
NAND(p_0,p_2,p_5,p_4). { a procedure for calculating p_4 }
NAND(p_0,p_3,p_5). { a procedure for calculating p_5 }
NAND(p_4,p_7,p_6). { a procedure for calculating p_6 }
NAND(p_5,p_6,p_7). { a procedure for calculating p_7 }

The above eight formulas express eight abstract devices, and they show all connections among them. So the eight abstract devices and their connections, which we call an **abstract circuit**, can be read into a computer for simulation and verification of the D Flip-flop.

We calculate the values of every point at every time from 0 to 20. We get a list as follows. The blank means a value *unknown*. If time t is less 0, the values of every point is the *unknown*.

At time 0 the p_0 is 0 (i.e. $p_0[0]$ is 0) and the p_1 is 1 (i.e. $p_1[0]$ is 1) according to the condition of input. At time 1 the p_0 is still 0 (i.e. $p_0[1]$ is 0) and the p_1 is still 1 (i.e. $p_1[1]$ is 1). At the same time 1, the p_4 is 1 (i.e. $p_4[1]$ is 1) getting from NAND(p_0,p_2,p_5,p_4), and p_5 is 1 (i.e. $p_5[1]$ is 1) getting from NAND(p_0,p_3,p_5). ....

time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p_0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
p_1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
p_2			0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p_3				1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
p_4		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
p_5		1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
p_6														0	0	0	0	0	0	0	0
p_7													1	1	1	1	1	1	1	1	1

2. If the input requirement is the following:
   a. For the point p_0, the input signal is
      CONDITION0(p_0) $\equiv$ p_0 is 0 if the time is smaller than 10, otherwise p_0 is 1.
   b. For the point p_1, the input signal is
      CONDITION1(p_1) $\equiv$ p_1 is 0 at any time.

The chips (abstract devices) in a D Flip-flop are shown as follows:

CONDITION0(p_0). { a procedure for calculating p_0 }
CONDITION1(p_1). { a procedure for calculating p_1 }
NAND(p_1,p_4,p_2). { a procedure for calculating p_2 }
NAND(p_2,p_5,p_3). { a procedure for calculating p_3 }
NAND(p_0,p_2,p_5,p_4). { a procedure for calculating p_4 }
NAND(p_0,p_3,p_5). { a procedure for calculating p_5 }
NAND(p_4,p_7,p_6). { a procedure for calculating p_6 }
NAND(p_5,p_6,p_7). { a procedure for calculating p_7 }

We calculate the values of every point at every time from 0 to 20. We get a list as follows:

time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p_0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
p_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p_2		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
p_3			0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p_4		1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
p_5		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
p_6													1	1	1	1	1	1	1	1	1
p_7														0	0	0	0	0	0	0	0

From above two lists, we get a result that the value of the input p_1 at t=0 transfers to the output p_7 at t=20 after the control p_0 goes up from 0 to 1. Let p_1 be *input*, p_7 be *output*, and p_0 be *clock*. We can check that the circuit has the following property :

Until(*clock* = 0, *Until*(*clock* = 1, *output* = *input*)) .

5. Simulations of Synchronous Circuits

The operation of the circuits in many digital systems is controlled or synchronized with a timing signal. This signal is known as the *clock*. In this paper, the clock signal is assumed to be a square wave. The amount of time during which the waveform is at the high voltage level is marked t_t and is called the *top width* of the square wave. The amount of time during which the waveform is at the low voltage level is marked t_b and is called the *bottom width*. The T_t and T_b are sufficient width for simulation of the circuits. For example, the T_t and T_b can take the largest delay time of the circuit.

The device which generates the square wave signal is a CLOCK. We use CLOCK($n,t_t,t_b,output$) to represent (to calculate) the clock in Figure 5-1 and use CLOCK($-n,t_t,t_b,output$) to represent the clock in Figure 5-2.

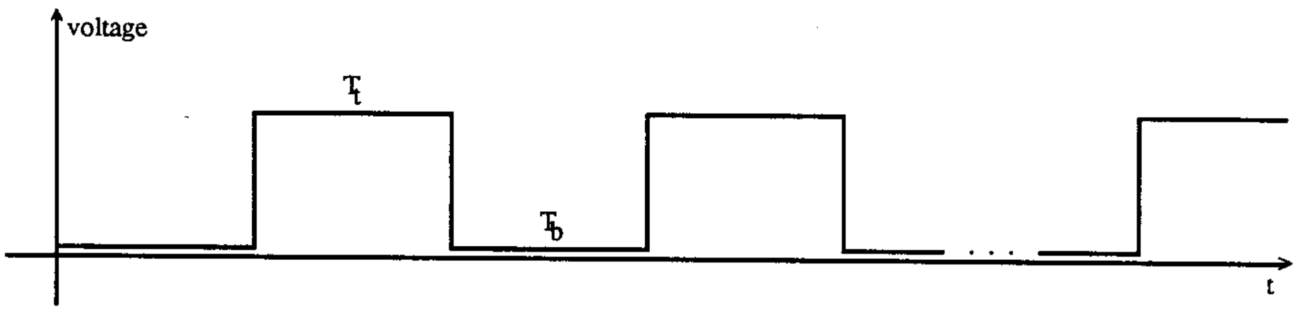


Figure 5-2: The Waveform of CLOCK($n,t_t,t_b,output$)

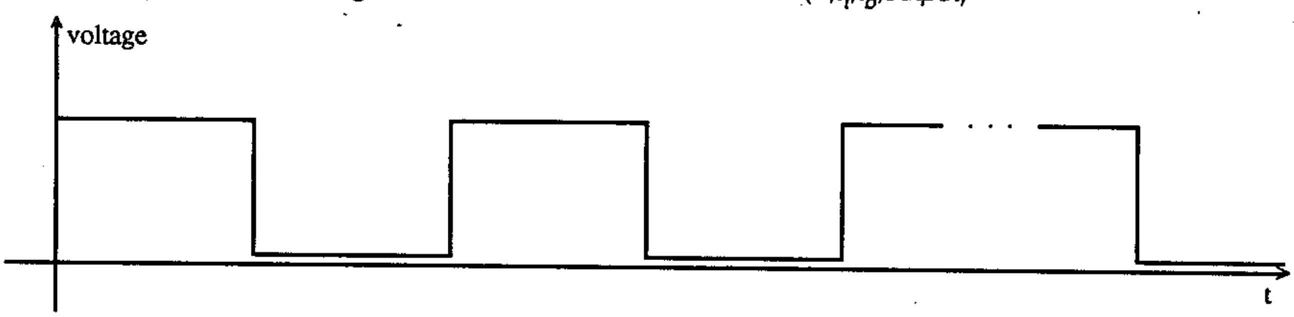


Figure 5-2: The Waveform of CLOCK($-n,t_t,t_b,output$)

In this section, we will simulate a Decade Counter circuit that consists of four D Flip-flop chips and some gates. To begin with, we need to make a abstract device for the D Flip-flop, i.e. to write a temporal logic program to simulate it. It is possible to intuitively describe the input and output properties of the D Flip-flop mentioned in the preceding section, but it is difficult to translate this description into a formal language. In figure 4-1, suppose that the p_0 is *cp* (input clock pulses), the p_1 is *input*, the p_7 is *output1* and the p_6 is *output2*. The D Flip-flop is of positive-edge triggering. We know the following properties of D Flip-flops :

(1) After Positive-edge triggering.

If the *cp* has been at level 1 for 3 or more time units before which it was at level 0 for 3 time units, and if the *input* had been at some value X for 2 units before the positive-edge transition and maintained the same value for 2 units from the transition, then the *output1* gets the value X. This property can be expressed by the following statement in our temporal language :

$Since[\geq 3](cp = 1, \blacksquare[3](cp = 0)\ \&\ Stable[2,1](input)\ \&\ (X \text{ is } input))$ -: $output1 := X$

or

$A[X]$ -: $output1 := X$

where $A[X] \equiv Since[\geq 3](cp = 1, \blacksquare[3](cp = 0)\ \&\ Stable[2,1](input)\ \&\ (X \text{ is } input))$.

(2) Output signal maintains stability after Negative-edge time.

If *cp* switches from level 1 to level 0 and maintains this level after the positive-edge triggering phenomenon $A[X]$, then the *output1* will also keep the value X. This property can be expressed by the following statement :

$Since(cp = 0, A[X])$ -: $output1 := X$

or

$Since(cp = 0, Since[\geq 3](cp = 1, \blacksquare[3](cp = 0)\ \&\ Stable[2,1](input)\ \&\ (X \text{ is } input)))$ -: $output1 := X$

(3) Output signal maintains stability at the time of Positive-edge triggering.

The output is stable if the *cp* has been at level 0 for three time units (See Figure 4-1). If *cp* switches from level 0 to level 1, it will influence output1 after 2 time units. In other words, the level of *cp* at the last time doesn't influence the current value of the output. Hence, if one unit time ago the *cp* had been at level 0 for three units, then the current value of the output is the same as at last time. This property can be expressed for *output1* by the following statement :

$\bullet(\blacksquare[3](cp = 0)\ \&\ (X \text{ is } output1))$ -: $output1 := X$;

Therefore, we use the following abstract device instead of D Flip-flop with an output *output1* :

Case(
$\bullet(\blacksquare[3](cp = 0)\ \&\ (X \text{ is } output1))$ -: $output1 := X$;
$Since[\geq 3](cp = 1, \blacksquare[3](cp = 0)\ \&\ Stable[2,1](input)\ \&\ (X \text{ is } input))$ -: $output1 := X$;
$Since(cp = 0, Since[\geq 3](cp = 1, \blacksquare[3](cp = 0)\ \&\ Stable[2,1](input)\ \&\ (X \text{ is } input)))$ -: $output1 := X$;
otherwise -: *output1* := *unknown*
)

We use the abstract devices D1(*input,cp,output2,output1*) and D2(*input,cp,output1,output2*) to express the whole D Flip-flop with two outputs.

D Flip-flop

D1(*input,cp,output2,output1*) ≡
Case(
  ●(■[3](*cp* = 0) & (*X is output1*)) -: *output1* := *X* ;
  Since[≥3](*cp* = 1, ■[3](*cp* = 0) & *Stable*[2,1](*input*) & (*X is input*)) -: *output1* := *X* ;
  Since(*cp* = 0, *Since*[≥3](*cp* = 1, ■[3](*cp* = 0) & *Stable*[2,1](*input*) & (*X is input*))) -: *output1* := *X* ;
  otherwise -: *output1* := *unknown*
)

D2(*input,cp,output1,output2*) ≡
Case(
  ●(■[3](*cp* = 0) & (*X is output2*)) -: *output2* := *X* ;
  Since[≥3](*cp* = 1, ■[3](*cp* = 0) & *Stable*[2,1](*input*) & (*X is input*)) -: *output2* := *abs(X)*;
  Since(*cp* = 0, *Since*[≥3](*cp* = 1, ■[3](*cp*=0)&*Stable*[2,1](*input*) & (*X is input*))) -: *output2* := *abs(X)*;
  otherwise -: *output1* := *unknown*
)

The Decade Counter circuit in Figure 5-3 will be analysed with an initial state a_0=0, a_1=0, a_2=0 and a_3=0. The *clk* receives a square wave from a CLOCK(10,22,22,*clk*) at time *t*=0. The Decade Counter is equivalent to the following abstract circuit which consists of seventeen abstract devices: eight devices which represent the four D Flip-flops, six AND gates, one OR gate, one CLOCK and one device for the initial state. We calculate the values of every point at every time from 0 to the end of the clock.

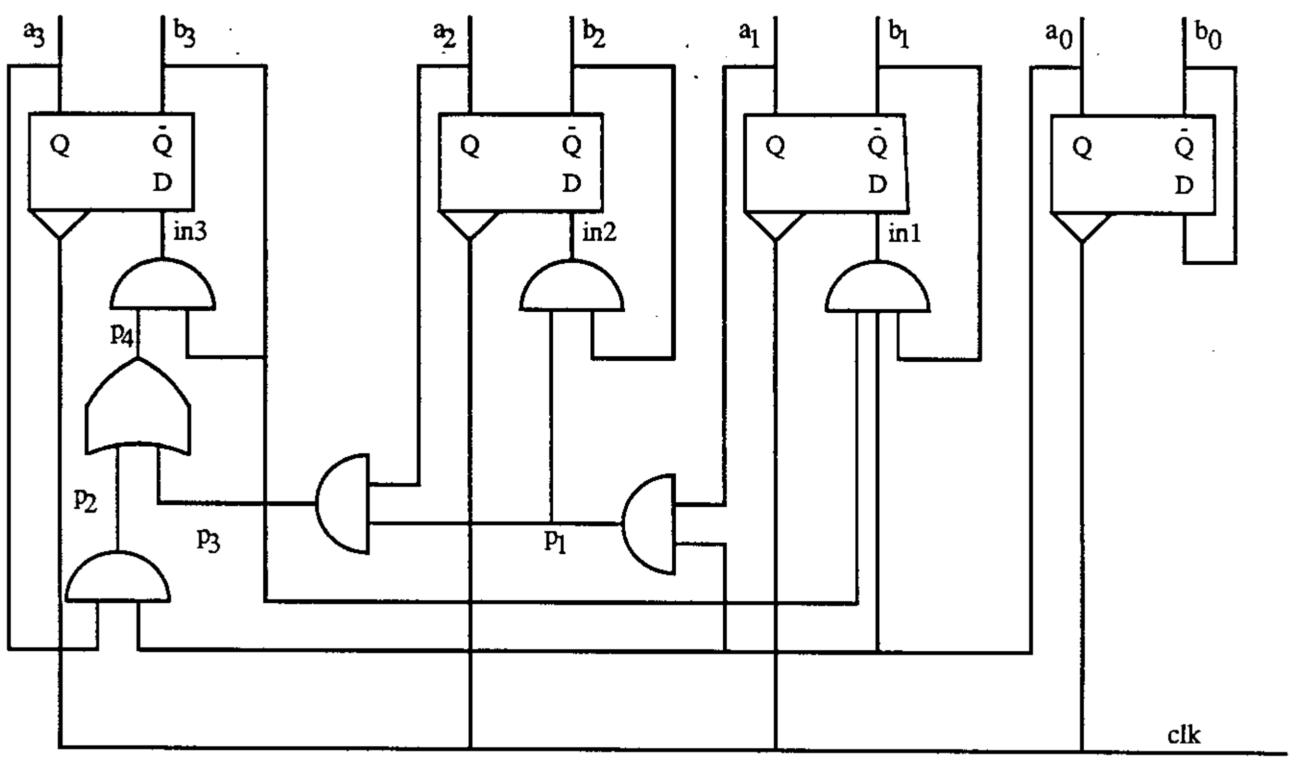


Figure 5-3: A Decade Counter

At the initial time, we let a_0, a_1, a_2 a_3 be 0, and that b_0, b_1, b_2 b_3 be 1. After the initial time, we use the following program to calculate the values of every point a_0, b_0, a_1, b_1, a_2, b_2, a_3, b_3, *in1*, *in2*, *in3*, p_1, p_2, p_3, p_4 in the circuit and the output in the decimal system.

```
D1(b0,clk,b0,a0).
D2(b0,clk,a0,b0).
D1(in1,clk,b1,a1).
D2(in1,clk,a1,b1).
D1(in2,clk,b2,a2).
D2(in2,clk,a2,b2).
D1(in3,clk,b3,a3).
D2(in3,clk,a3,b3).
AND(b1,a0,b3,in1).
AND(a1,a0,p1).
AND(p1,b2,in2).
AND(a3,a0,p2).
AND(p1,a2,p3).
OR(p2,p3,p4).
AND(p4,b3,in3).
Conversion(a0,a1,a2,a3,count).
CLOCK(10,22,22,clk).
```

where Conversion(a_0,a_1,a_2,a_3,*count*) is the following statement:

count is $a_0+a_1 \bullet 2+a_2 \bullet 2^2+a_3 \bullet 2^3$

After calculating, we get the simulation data on the above circuit. In Section 7 we will check whether the circuit satisfies the requirements of a Decade Counter.

6. Implicit Types and Programs

Objects of a given type have a representation that respects the expected properties of the data type. The representation is chosen to make it easy to perform expected operations on data objects. Conventional types are usually explicit. For example, **var** *a*: *integer* , it means the *a* is an integer. It is enough to arrange a single integer storage location for *a*. But for a temporal variable *p* , if the *p* is an integer, it is not enough to arrange only one location, otherwise the formula $p=1 \& \Diamond(p=6)$ could not be checked. Thus, if we say that a temporal variable *p* is an integer, the compiler must *implicitly* allocate an *array* of integers for *p*. For temporal variables *p* ..., we will use the following type of declarations:

time frame var $t = 0\ ..\ 440$;

$\vdots$

temporal var *p* : *integer*;

The above declaration indicates that *p* is a temporal variable which has 441 integer values corresponding to the times $t = 0,...,440$. It is assumed that any temporal variable *p* takes the value *unknown* at the time $t < 0$ or $t > 440$ (when *t* is out of the frame of reference). The **time frame** variable *t* is a special type that can be used in a **for** statement as follows :

for (t=0, t < 10, t++) *A*.

This **for** statement executes the statement *A* at every time from 0 to 9.

All QTL programs have the following general form:

PROGRAM name
   definitions
{
   instructions
}.

In the following program, temporal variables usually only take 0 or 1 as Boolean values. Hence we let the *unknown* be -1. We give a complete program for the simulation of a Decade Counter mentioned in Section 5 as follows:

PROGRAM decade (*input, output*);
time frame var $t = 0\ ..\ 440$;
var *X, Y* : integer;
var *unknown* = -1 : integer;
temporal var a_0, a_1, a_2, a_3 : integer;
temporal var b_0, b_1, b_2, b_3 : integer;
temporal var p_1, p_2, p_3 : integer;
temporal var *in1, in2, in3, cp, clk, count* : integer;

FUNCTION *abs*(var *x : integer*) : *integer*;
  Case(
        x = unknown -: return(unknown);
        otherwise -: return($|x-1|$)

```
        );
PROCEDURE AND(temporal var input1,input2,output : integer);
  Case(
        ●(input1 = 1,input2 = 1) -: output := 1;
        ●(input1 = 0 ∨ input2 = 0) -: output := 0;
        otherwise -: output := unknown
        );
PROCEDURE AND(temporal var input1,input2,input3,output : integer);
  Case(
        ●(input1 = 1,input2 = 1,input3 = 1) -: output := 1;
        ●(input1 = 0 ∨ input2 = 0 ∨ input3 = 0) -: output := 0;
        otherwise -: output := unknown
        );
PROCEDURE OR(temporal var input1,input2,output : integer);
  Case(
        ●(input1 = 1 ∨ input2 = 1) -: output := 1;
        ●(input1 = 0,input2 = 0) -: output := 0;
        otherwise -: output := unknown
        );
PROCEDURE D1(temporal var input,cp,output2,output1 : integer);
  Case(
        ●(■[3](cp = 0) & (X is output1)) -: output1 := X;
        Since[≥3](cp = 1, ■[3](cp = 0) & Stable[2,1](input) & (X is input)) -: output1 := X;
        Since(cp = 0, Since[≥3](cp = 1, ■[3](cp = 0) & Stable[2,1](input) & (X is input))) -: output1 := X;
        otherwise -: output1 := unknown
       );
PROCEDURE D2(temporal var input,cp,output1,output2 : integer);
  Case(
        ●(■[3](cp = 0) & (X is output2)) -: output2 := X;
        Since[≥3](cp = 1, ■[3](cp = 0) & Stable[2,1](input) & (X is input)) -: output2 := abs(X);
        Since(cp = 0, Since[≥3](cp = 1, ■[3](cp=0)&Stable[2,1](input) & (X is input))) -: output2 := abs(X);
        otherwise -: output1 := unknown
       );
PROCEDURE  Conversion(temporal var input0,input1,input2,input3,output : integer);
  {
    output  is  input0 + 2 • input1 + 2 • 2 • input2 + 2 • 2 • 2 • input3
  };
PROCEDURE CLOCK(var n,tt,tb : integer; temporal var output : integer);
  {
    var time = 0 : integer;
    while (notNow(time)) time++;
    Case(
          time / (tt+bb) ≥ n -: output := 0;
          time / (tt+bb) < n -:
          Case(
                time % (tt+tb) < tt -: output := 0;        /* The % is a remainder operation */
                time % (tt+tb) ≥ tt -: output := 1;
                otherwise -: output := unknown
              )
        )
  };
{
for (t=0, t<10, t++)                          /* set some expected initial values */
```

```
   {
      a0 := 0;
      a1 := 0;
      a2 := 0;
      a3 := 0;
      b0 := 1;
      b1 := 1;
      b2 := 1;
      b3 := 1;
      CLOCK(100,20,20,clk)
   };
for (t=0, t<10, t++)                          /* generate the other initial values */
   {
      in1 := unknown;
      in2 := unknown;
      in3 := unknown;
      p1 := unknown;
      p2 := unknown;
      p3 := unknown
   };
for (t=10, t<440, t++)
   {
      CLOCK(100,20,20,clk);
      D1(b0,clk,b0,a0);
      D2(b0,clk,a0,b0);
      D1(in1,clk,b1,a1);
      D2(in1,clk,a1,b1);
      D1(in2,clk,b2,a2);
      D2(in2,clk,a2,b2);
      D1(in3,clk,b3,a3);
      D2(in3,clk,a3,b3);
      AND(b1,a0,b3,in1);
      AND(a1,a0,p1);
      AND(p1,b2,in2);
      AND(a3,a0,p2);
      AND(p1,a2,p3);
      OR(p2,p3,p4);
      AND(p4,b3,in3);
   };
for (t=0, t<440, t++)                         /* calculus the output */
   {
      Conversion(a0,a1,a2,a3,count)
   }
}.
```

7. Verification of Synchronous Circuits

If we are dealing with a circuit with a clock, we are usually only interested in events that occur near a clock transition. Therefore, it is often useful to establish a new clock frame of reference. Suppose that we have a clock with a real time frame of reference t_0, t_1, t_2,.... The time at which the clock changes from 0 to 1 is called a positive edge time, and the previous time is called a negative-resting time. The time at which the clock changes from 1 to 0 is called a negative edge time , and the previous time is called a positive-resting time. Let all negative-resting times be $t'_0, t'_1, t'_2, \ldots$, and let all positive-resting times be $t''_0, t''_1, t''_2, \ldots$. All resting times are $t'_0, t''_0, t'_1, t''_1, t'_2, t''_2, \ldots$ on the clock.(See Figure 7-1)

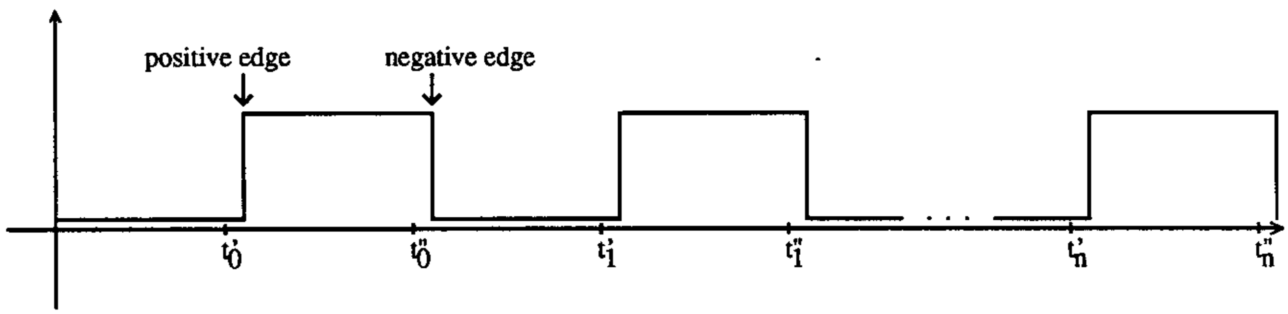


Figure 7-1: A Sequence of Resting Times

We introduce a temporal variable *clock_level*. The *clock_level* becomes 1 at times t''_0, t''_1, t''_2,..., and 0 at times t'_0, t'_1, t'_2, ... i.e. the *clock_level* is 1 at positive-resting times, and 0 at negative-resting times.

Given this sequence of all resting times, we introduce a new time frame $T_0, T_1, T_2, \ldots$ where T_0 is t'_0, T_1 is t''_0, and so on. If the clock is sufficiently slow, the circuit will be in a stable state before and at any resting times. The properties of the circuit should only be discussed at these times. Now, we can give meaning to formulas at a certain resting time T_i as follows

$T_i \models Now(t)$	iff	$t = T_i$.
$T_i \models A \& B$	iff	$T_i \models A$ first, and then $T_i \models B$.
$T_i \models A \vee B$	iff	$T_i \models A$, otherwise $T_i \models B$.
$T_i \models \bigcirc A$	iff	$T_{i+1} \models A$.
$T_i \models \bigcirc[n]A$	iff	$T_{i+n} \models A$.
$T_i \models \Diamond A$	iff	$T_i \models A$, otherwise $T_{i+1} \models \Diamond A$.
$T_i \models \Diamond[0]A$	iff	$T_i \models A$.
$T_i \models \Diamond[n]A$	iff	$T_i \models A$, otherwise $T_{i+1} \models \Diamond[n-1]A$.

$T_i \models \Box A$	iff	$T_i \models A$ first, and then $T_{i+1} \models \Box A$.
$T_i \models \Box[0]A$	iff	$T_i \models A$.
$T_i \models \Box[n]A$	iff	$T_i \models A$ first, and then $T_{i+1} \models \Box[n-1]A$.
$T_i \models Until(A,B)$	iff	$T_i \models B$, otherwise $T_i \models A \& \bigcirc Until(A,B)$.
$T_i \models Until[\geq 0](A,B)$	iff	$T_i \models Until(A,B)$.
$T_i \models Until[\geq n+1](A,B)$	iff	$T_i \models \Box[n]A$ first, and then $T_{i+n+1} \models Until(A,B)$.
$T_i \models Until[\leq 0](A,B)$	iff	$T_i \models B$.
$T_i \models Until[\leq n+1](A,B)$	iff	$T_i \models B$, otherwise $T_i \models A \& \bigcirc Until[\leq n](A,B)$.

For the Decade Counter, let *clock_level* be the *clk*. If the Decade Counter is correct, it must satisfy the following property:

```
count = 0 & □[20](
                (clk = 0, count < 9, Ocount = (count+1)) ∨
                (clk = 0, count = 9, Ocount = 0) ∨
                (clk = 1, Ocount = count)
              )
```

This can be checked by adding the following code fragment to the program. If the program sets $Y = 1$ then the Decade Counter circuit has the above property, otherwise it doesn't.

```
Case(count = 0 , □[20](
                      (clk = 0, count < 9, Ocount = (count+1)) ∨
                      (clk = 0, count = 9, Ocount = 0) ∨
                      (clk = 1, Ocount = count)
                    ) -:  Y := 1 ;
     otherwise -:  Y := 0
   )
```

8. Related Work and Discussion

In the technique described in [4] for the automatic verification of asynchronous circuits, using normal Temporal Logic can generate a huge number of states that can cause the system to run out of memory. In this case, it may be preferable to try to use this Temporal Language since it doesn't need a large memory. Suppose that the amount of delay times of all devices in an asynchronous circuit is *m*. In order to model the asynchronous circuit, we can make a abstract device for generating signal to simulate a random signal (an asynchronous signal) that occurs in the circuit. This treatment is very realistic. The abstract device will have different delay times. Generally, these delay times are smaller than the *m*. Thus, an asynchronous circuit becomes several synchronous circuits, which have different delay times. A program using this Temporal Language checks these synchronous circuits one by one independently, so it saves memory and is especially suitable for the design and verification of VLSI. If there is something wrong in an asynchronous circuit, this method can generally find it out after checking few synchronous circuits.

References

1. M. Abadi and Z. Manna, Temporal Logic Programming, Proceedings of Fourth IEEE Symposium on Logic Programming, 1987.
2. G. V. Bochmann, Hardware Specification with Temporal Logic: An example. IEEE Transactions on Computers C-30, 1, January 1981.
3. A. Camilleri, M. Gordon and T. Melham, Hardware Verification Using Higher-Order Logic, Technical Report No.91, Computer Laboratory, University of Cambridge, June 1986.
4. D. L. Dill and E. M. Clarke, Automatic Verification of Asynchronous Circuits Using Temporal Logic, IEE proceedings E: Computers and Digital Techniques, 133(5): 276-288, September 1986.
5. M. E. Leeser, Reasoning About the Function and Timing of Integrated Circuits with PROLOG and Temporal Logic, Technical Report N0.126, Computer Laboratory, University of Cambridge, February 1988.
6. A. J. Martin, Compiling Communicating Processes into Delay-Insensitive VLSI Circuits, Journal of Distributed Computing, V0l.1, N0.3, 1986.
7. B. C. Moszkowski, A Temporal Logic for Reasoning about Hardware, Proceedings Sixth International Symposium on Computer Hardware Description Languages, Carnegie-Mellon University, Pittsburgh, pages 79-90, 1983.
8. B. C. Moszkowski, Executing Temporal Logic Programs, Technical Report No. 55, Cambridge University, 1984.
9. C. T. Tang, To Unify Programming with A Temporal Logic Language System __ A Step Toward A Logic Machine, Technical Report No. 87-160, Computer Science Department, Carnegie-Mellon University, 1987.