6.1 Loops in RTL Design

To help understand how loops are used in behavioral descriptions, it is useful to review how loops can be used in RTL descriptions. Many loop structures imply a state machine that is required to control the operation of the loop. For example, when the number of times a loop executes depends on the input data, some kind of a state machine is required. Historically, RTL synthesis tools have not automatically created state machines.

Loops do not appear very often in RTL descriptions. This is largely because loops have generally only supported in a very limited manner for RTL synthesis—usually only FOR loops with constant bounds can be specified. FOR loops with non-constant bounds, WHILE loops, and infinite loops have not been supported or supported only in a limited manner for RTL synthesis.

Many RTL designers are used to writing RTL code based on this limitation. Some examples of loops that illustrate these restrictions are shown in Figure 6-1.

![Figure 6-1: Examples of loops](image-url)
During RTL synthesis, **FOR loops are unrolled**. To unroll a FOR loop, the statements inside the loop are copied as many times as the loop would be executed. For each copy of the loop, the iteration variable is replaced with the appropriate constant for that iteration of the loop.

Figure 6-2 shows a FOR loop with constant bounds and the equivalent code after the loop has been unrolled.

![Unrolling a FOR loop with fixed bounds](image)

This transformation implies that in an RTL synthesis tool that doesn't perform resource sharing, an unrolled FOR loop always results in the generation of all of the hardware that is required to perform the execution of every iteration of the loop in the resulting design. A few lines of code can generate significant amounts of hardware. For example, if a FOR loop in an RTL description that is executed 100 times contains a statement with a 16-bit add operation, the circuit generated by synthesis will contain 100 adder components! If an RTL designer wishes to use a single adder and schedule the calculation over 100 clock cycles, he would have to write code that describes a state machine and the appropriate surrounding circuitry to multiplex data through the adder and to store intermediate values.

Consider the FOR loop in Figure 6-3. When unrolled, this loop not only implies 7 adders, it also introduces significant delay into the synthesized design.

![Hardware inferred from a FOR loop with fixed bounds](image)
The main use of loops in RTL design is to provide a convenient way to code simple repetitive structures as shown in Figure 6-2.

In addition to limiting the use of loops to FOR loops with fixed bounds, most RTL synthesis tools do not support NEXT and EXIT statements. For these reasons, loops rarely appear in RTL descriptions.

6.2 Loop Constructs and State Machines

In order to support more complex looping structures, a synthesis tool must be able to generate state machines. RTL tools have not traditionally done this. Some of today's RTL synthesis tools support the use of multiple WAIT statements, but only in a very simple way.

Chapter 5 discussed how a process implies a state machine — WAIT statements can be thought of as corresponding to states and the statements between the WAIT statements can be thought of as occurring on the transition between states.

Figure 6-4 illustrates the relationship between WAIT statements in a PROCESS and an implied state machine.
In VHDL simulation, when the last statement in a process has been executed, simulation continues at the first statement in the process. In the state machine that corresponds to this description, there is a transition from the last state (s5) back to the first state (s1). A process infers a “circular” state machine. This means that any design that is scheduled will perform a series of operations over and over again.

An infinite loop also implies a “circular” state machine. Figure 6-5 shows VHDL code that is almost identical to the code in Figure 6-4. However, the repetitive nature of this state machine is inferred by a loop construct, rather than a process.

```
LOOP
  state s1
    WAIT UNTIL (clk'EVENT AND clk = '1');
    s1 + s2
  state s2
    WAIT UNTIL (clk'EVENT AND clk = '1');
    s2 + s3
  state s3
    WAIT UNTIL (clk'EVENT AND clk = '1');
    s3 + s4
  state s4
    WAIT UNTIL (clk'EVENT AND clk = '1');
    s4 + s5
  state s5
    WAIT UNTIL (clk'EVENT AND clk = '1');
    s5 + s1
END LOOP;
```

Figure 6-5: Relationship between WAIT statements and a state machine in an infinite loop

This loop is called an infinite loop, because the loop construct itself does not describe any conditions for exiting the loop. An infinite loop implies a circular state machine. Branches can be added to a state machine using NEXT statements (discussed in Section 6.5) and EXIT statements (discussed in Section 6.3). In addition to infinite loops, there are FOR loops and WHILE loops, which describe conditions for exiting the loop.

The loop construct is fundamental to writing behavioral descriptions. Loops provide a powerful mechanism for describing repetitive operations that occur in an algorithm. A goal of behavioral synthesis is to allow designers to describe at a higher level of abstraction that is closer to the algorithm. This would be very difficult if the use of loop structures were disallowed since algorithms often use looping structures to describe their function.
6.3 The EXIT Statement

The EXIT statement transfers control to the statement that follows the end of a loop. Any statements between the EXIT statement and the end of the loop are not executed.

There are a number of different ways to use an EXIT statement. An exit from a loop can be conditional or unconditional. In addition, an EXIT statement can specify the name of the loop to exit -- this provides a mechanism for exiting an outer loop from within an inner loop. If a loop name is not specified, an EXIT statement will result in exiting the innermost loop. All of these forms can be used in behavioral descriptions. Figure 6-6 shows examples of code fragments that include EXIT statements.

![Figure 6-6: Example uses of the EXIT statement](image)

Figure 6-7 shows two nested infinite loops with an EXIT statement that conditionally exits the inner loop. The figure also shows the state machine that corresponds to this code.

![Figure 6-7: Two embedded loops with an exit from the inner loop](image)
To understand this diagram, think of the WAIT statements as representing states and the code between WAIT statements as the transitions between states.

To construct the state diagram that this code represents, every possible state-to-state transition must be considered. For example, the statements that immediately follow the start of the loop called "internal_loop" are executed under two possible conditions:

... when transitioning from state s2 to s3 when entering the loop
... when transitioning from state s6 to s3 on successive iterations of the inner loop

The EXIT statement creates branches in the state machine. The transition from state s4 to s5 is taken when the condition that controls the EXIT statement is false. The transition from state s4 to state s7 is taken if that condition is true.

Loops with EXIT statements can be used in behavioral descriptions to affect the state machine that is created during synthesis. An EXIT statement will correspond to a branch in the generated state machine.

6.3.1 Specifying a Synchronous Reset

An important consideration when designing a circuit is providing a method for bringing the circuit into a known state. This is typically done using a reset signal that is made active for a short period of time when power is applied to the circuit. Since in most technologies the power-up state of registers is not predictable, resetting a circuit usually involves setting output and internal registers to either a '0' or a '1'.

Any scheduled design that implies a state machine must contain a reset signal. This is because the reset signal is used to initialize the state machine. The reset signal can also be used to initialize other portions of the circuit, but without a reset, the gate-level implementation of the state machine cannot be set to a known state.

In an RTL description, a synchronous reset is described by testing a reset condition at the clock edge. A reset condition is a comparison between a reset signal and the value '0' or '1' (for an active-low or active-high reset, respectively). If the reset condition is true, then the assignments that comprise the reset action are performed. A reset action consists of assigning constant values (zeros and/or ones) to one or more signals or variables. If the reset condition is false, the behavior of the circuit continues unaffected.

The same behavior must be specified in a behavioral description: at the clock edge, the reset condition must be tested. The difference between an RTL clocked process and a behavioral process is that the behavioral process can describe multiple clock edges. So in a behavioral description, it is necessary to test the reset condition at every clock edge. A description with multiple WAIT statements implies a state machine with multiple states. A design with a synchronous reset must have a transition from each of these states back
to the first state. Since behavioral synthesis tools can create Mealy-style state machines, the reset actions can be thought of as occurring on the transitions from each state back to the first state. Such transitions can be created with EXIT statements.

Similar to an RTL clocked process, the reset action in a behavioral description must be separated from the normal operation of the circuit. After a circuit has been reset, normal operation should resume. The circuit should not perform the reset action again unless the reset condition is once again true.

To define a synchronous reset for a behavioral design, an EXIT statement must follow every WAIT statement. These EXIT statements must all have the same exit condition and should exit an infinite loop that describes the normal (non-reset) behavior of the design. The reset behavior is described outside the main loop.

6.3.2 Simulating the Reset Condition

The VHDL code to describe a synchronous reset is shown in Figure 6-8.

![Figure 6-8: Synchronous reset described with EXIT statements](image-url)
Consider how this description will simulate when the reset condition becomes true (i.e. the value of the signal reset is ‘1’). Assume that the active statement is the WAIT statement that corresponds to state s3. The next time there is a rising clock edge, execution will proceed to the EXIT statement which follows the WAIT statement. Since the reset condition is now true, execution will continue with the first statement that follows the loop labeled ‘main_loop’. But there are no statements after the loop, so the process repeats from the top. This is where the reset action is defined.

In this example, the reset action assigns zeros to the output signal y. Execution continues to the next statement, which is the first WAIT statement inside the main loop. If the reset action is still true at the next rising clock edge, the main loop is exited once again. As long as the reset action is true, only the assignments in the reset action will be performed. The circuit is ‘stuck’ in state s1. When the reset action becomes false, the main loop continues executing normally. Since the main loop is an infinite loop, when the last statement in the main loop is reached, execution continues at the top of that loop.

Now consider how the VHDL code in Figure 6-8 will simulate in general. Simulation begins at the first statement in the process, which is the reset action. The reset action is performed and the loop that describes the normal operation of the circuit is entered. This means that the reset action is always performed at the start of VHDL simulation – even if the reset condition was never set to true!

A better behavioral model would require that the reset condition be true in order for the reset action to be performed. This can be accomplished by moving the reset action from before the main loop to after the main loop. The two methods of specifying a synchronous reset are shown in Figure 6-9.

```
PROCESS
  -- Reset action before the main loop is performed even if reset condition is NEVER activated!
  main_loop: LOOP
    -- Normal (non-reset) operation
    WAIT UNTIL (clk'EVENT AND clk = '1');
    EXIT main_loop WHEN (reset = '1');
    -- Algorithm goes here
  END LOOP main_loop;
END PROCESS;

PROCESS
  main_loop: LOOP
    -- Normal (non-reset) operation
    WAIT UNTIL (clk'EVENT AND clk = '1');
    EXIT main_loop WHEN (reset = '1');
    -- Algorithm goes here
  END LOOP main_loop;
END PROCESS;
```

Figure 6-9: Alternate positions for reset action — before (left) and after (right) the main loop

The structure on the right requires that the reset condition be true in order for the reset action to be performed. This means that a test bench must exercise the reset condition even when simulating the behavioral model. This is important since a synthesized design will contain a state machine that must be reset. Placing the reset action at the end of a behavioral model helps ensure that a test bench or any other model that is interfacing with the description will perform an appropriate reset.
6.3.3 Complex Reset Sequences

A reset action can only contain simple assignments of constants – there can not be any expressions that require actual time to compute (such as an add). But some circuits will require a more complex initialization sequence during reset. For example, a description may include a variable that has been mapped to memory (how to do this is discussed in Chapter 9). If every word in the memory needs to be loaded with zeros when the circuit is reset, a multi-cycle initialization sequence is needed. Writing to a memory requires at least one clock cycle, so initializing an entire memory to zeros may take a significant number of clock cycles.

Figure 6-10 shows the general structure of a design that contains both a reset action and an initialization sequence.

In this code, an extra loop has been added to contain the initialization sequence. In addition, the EXIT statements that specify the synchronous reset exit the initialization loop, not the main loop. The reset action now appears after the initialization loop.
The state machine that corresponds to this code is also shown in Figure 6-10. The states that represent the initialization sequence start with the letter "I". Note that the initialization sequence can potentially take many clock cycles to complete. When the initialization is complete, control is transferred to the first state in the main loop. The states that represent the main loop start with the letter "M". When the last state of the main loop is reached, control is passed again to the first state of the main loop.

From every state there is a possible transition to the first state in the initialization sequence. The EXIT statements that represent the synchronous reset imply these transitions.

As long as the reset condition is true, the circuit is held in state I1. The initialization sequence only begins when the reset condition becomes false. Other modules must account for the delay between the reset condition going to false and the start of the main loop when interfacing to a design with an initialization sequence.

Unlike the reset action, the initialization sequence is always performed during behavioral simulation, whether or not a reset is performed. However, since the reset action is after the initialization loop, the reset assignment is still only performed when the reset condition is true.

The code that describes the initialization sequence is scheduled in the same manner as code in the main loop. Thus, unlike the reset action, the code that describes an initialization sequence is not restricted — an initialization sequence can contain multiple WAIT statements, or even additional loops.

6.3.4 Using an Attribute to Specify a Reset Signal

Most behavioral synthesis tools do not require that the reset sequence be explicitly described using EXIT statements in the behavioral description. The EXIT statements can be omitted from the code and the reset signal and phase (active high or active low) can be specified using an attribute or a tool command.

The advantage of omitting the EXIT statements from the behavioral description is that it may make the code appear simpler. To specify a reset in a behavioral description, an EXIT statement must follow every WAIT statement. When a behavioral description models the reset behavior, each clock edge is defined using two statements (the WAIT statement and the EXIT statement).

The disadvantage of omitting the EXIT statements is that it is not possible to simulate the reset behavior in the behavioral description, prior to scheduling. But even a design that does not explicitly describe the reset behavior must still contain a reset signal — the reset signal is required to bring the state machine in the synthesized design into a known state.

The most reliable design process includes simulation of the reset behavior. This requires that the reset behavior be explicitly modeled in the behavioral description.
6.3.5 Specifying an Asynchronous Reset

Some designers prefer to use asynchronous resets instead of synchronous resets. Unlike a synchronous reset that only tests the reset condition on a clock edge, an asynchronous reset immediately changes the state of the circuit when the reset condition becomes true.

This requires a change in the WAIT statement. Under normal operation, the WAIT statement must wait for the next clock edge. However, if the reset condition becomes true while waiting for the clock edge, control must transfer immediately to the reset action. This behavior is accomplished by adding the reset condition to the WAIT statement. The EXIT statement that follows the WAIT statement has the same form as for a synchronous reset. An example of a WAIT / EXIT pair that describes asynchronous behavior is shown in Figure 6-11.

![Figure 6-11: Example of a WAIT / EXIT pair that describes asynchronous reset behavior](image)

Aside from the modification to the WAIT statement, a design that describes an asynchronous reset has the same overall structure as a design that describes a synchronous reset. The simulation issues associated with placing the reset action before or after the main loop also apply to asynchronous resets.
6.4 Types of Loops

Loops can be grouped into three categories: infinite loops, WHILE loops, and FOR loops. Unlike infinite loops, which require an EXIT statement to terminate execution of the loop, WHILE loops and FOR loops describe in the loop declaration the conditions under which the loop is terminated.

All three kinds of loops can be used in behavioral descriptions and can use EXIT and NEXT statements to alter the flow of control through the loop.

6.4.1 Infinite Loops

An infinite loop is a loop that, in its declaration, does not describe any conditions for exiting the loop. An EXIT statement is required to terminate an infinite loop. NEXT statements can also appear in an infinite loop (see Section 6.5).

An infinite loop implies a circular state machine. When the last statement in the loop is reached, execution again continues at the statement at the start of the loop. Branches can be added to the implied state machine using NEXT and EXIT statements.

The beginning of this chapter discussed how an infinite loop implies a "circular" state machine. Figure 6-12 shows the state machine that is inferred by an infinite loop.

Figure 6-12: An infinite loop implies a "circular" state machine
6.4.2 WHILE Loops

A WHILE loop is a kind of unbounded loop. This means that at compile time, the number of times the loop will be performed is not known. The declaration of a WHILE loop includes an exit condition. When the condition of the loop becomes false, control is transferred to the statement that follows the end of the WHILE loop.

Every WHILE loop can be represented as an infinite loop with an EXIT statement. Often, behavioral synthesis tools translate WHILE loops into infinite loops prior to scheduling. The hardware that is implied by behavioral synthesis is based on the infinite loop, so understanding this translation will simplify the analysis of scheduled results.

Figure 6-13 shows a simple design with a WHILE loop and how that loop can be translated into an infinite loop with an EXIT statement. This design samples the value of the signal din. The design then continues to sample this signal until the sum of those values equals or exceeds the value 240.

```vhdl
User Code
main_loop : LOOP
    WAIT UNTIL (clk'EVENT AND clk='1');
    EXIT main_loop WHEN not rst='1';
    sum := din;
    dout <= others => '0';
    WHILE (sum < 240) LOOP
    WAIT UNTIL (clk'EVENT AND clk='1');
    EXIT main_loop WHEN not rst='1';
    sum := sum + din;
    END LOOP;
    WAIT UNTIL (clk'EVENT AND clk='1');
    EXIT main_loop WHEN rst='1';
    dout <= sum;
END LOOP main_loop;
```

Translated Code
```vhdl
main_loop : LOOP
    WAIT UNTIL (clk'EVENT AND clk='1');
    EXIT main_loop WHEN not rst='1';
    sum := din;
    dout <= others => '0';
    exit_condition := not( sum < 240 );
    LOOP
    EXIT WHEN exit_condition;
    WAIT UNTIL (clk'EVENT AND clk='1');
    EXIT main_loop WHEN not rst='1';
    sum := sum + din;
    exit_condition := not( sum < 240 );
    END LOOP;
    WAIT UNTIL (clk'EVENT AND clk='1');
    EXIT main_loop WHEN not rst='1';
    dout <= sum;
END LOOP main_loop;
```

Figure 6-13: Translation of a WHILE loop into an infinite loop with an EXIT statement

In the translated code the exit condition is calculated in two locations: immediately prior to entering the loop and immediately prior to performing the next iteration of the loop. The duplication of this statement corresponds to the state machine that is constructed for a WHILE loop. Consider how this design simulates. If the value of the input signal din is greater than 240 prior to the execution of the WHILE loop, the loop is never executed. This corresponds to a branch in the state machine. However, the exit condition must have been calculated prior to making the branch decision. The exit condition is
duplicated outside of the loop so that the exit condition (in this case the "<" operator) can be scheduled in a state before the start of the loop. Of course the calculation of this condition must also be calculated before subsequent iterations of the loop, so the condition is also calculated at the end of the loop. These relationships are illustrated in Figure 6-14.

There are branches from state s1 to states s2 and s3. If the exit condition is true, the branch to state s3 is taken, which bypasses the loop. If the exit condition is false, the branch to state s2 is taken, entering the loop.

The exit condition is duplicated during the translation of a WHILE loop to an infinite loop so that this decision can be made. It is necessary to calculate the exit condition prior to entering the loop. The first occurrence of the exit condition is calculated in state s1 to decide if the loop is executed or bypassed. The second occurrence of the exit condition is calculated in state s2 (inside the loop) to decide if the loop has completed.
6.4.3 FOR Loops

Like WHILE loops, every FOR loop can be represented as an infinite loop. Behavioral synthesis tools also translate FOR loops into infinite loops prior to scheduling.

Figure 6-15 shows a simple design with a FOR loop and how that loop can be translated into an infinite loop. This design samples the value of the signal din "n" times.

![User Code vs Translated Code](image)

The translation of a FOR loop into an infinite loop is slightly different than the translation for a WHILE loop. In particular, a variable must be created to represent the loop iterator.

The variable that represents the iterator is initialized just before the infinite loop. It is set to the starting value of the iterator, as declared in the FOR loop. In this example, the starting value is 1. Then the exit condition is calculated, similar to a WHILE loop. It is necessary to calculate the exit condition prior to the start of the loop because it is possible that the contents of the loop are not executed at all. For example, if the value of n is 0, the loop is not executed.

The exit condition is calculated using the greater-than or less-than operators (> or <), depending upon the direction of the loop (TO and DOWNTO, respectively). These operators are used to account for starting values that are out of range (like in the example when "n" has the value 0).
The translated loop begins with the entire contents of the original loop. At the end of the loop, the next value of the iterator is calculated. The iterator is either incremented or decremented, depending on the direction of the loop (TO or DOWNTO, respectively). Thus, a FOR loop implies either an incremeneter or decremeneter (in hardware), depending on the direction of the loop.

After the iterator has been incremented (or decremented), its value is tested. If the value is greater-than or less-than (depending upon the direction of the loop) the ending value of the iterator, the loop is complete and is exited. In this example, the ending value of the iterator is "n".

The EXIT statements in the translated code correspond to branching in a state machine in the same manner as for a WHILE loop. Figure 6-16 shows the state machine that corresponds to the translated FOR loop.

**Translated Code**

```
main_loop : LOOP

 state s1
   WAIT UNTIL (clk'EVENT AND clk='1');
   EXIT main_loop WHEN ret='1';
   sum := din;
   dout <= { OTHERS => '0' };
   i := 1;
   exit_condition := ( i > n );
   LOOP
     EXIT WHEN exit_condition;
   END LOOP;

 state s2
   WAIT UNTIL (clk'EVENT AND clk='1');
   EXIT main_loop WHEN ret='1';
   sum := sum + din;
   i := i + 1;
   exit_condition := ( i > n );
   END LOOP;

 state s3
   WAIT UNTIL (clk'EVENT AND clk='1');
   EXIT main_loop WHEN ret='1';
   dout <= sum;
   END LOOP main_loop;
```

Figure 6-16: Translated FOR loop and corresponding state machine

### 6.5 The NEXT Statement

The NEXT statement is similar to the EXIT statement in that it transfers control when inside a loop. The NEXT statement, however, transfers control to the start of the next iteration of the loop as opposed the statement after the end of the loop. Any statements between the NEXT statement and the end of the loop are not executed.
When a NEXT statement appears in a FOR loop, the iterator is assigned its next value (incremented or decremented) prior to the transfer of control.

The NEXT statement can be used at the start of a loop to wait until a signal obtains a particular value. For example, a handshaking signal called start could be added to the simple algorithm that was introduced in Chapter 2 to indicate that valid data is available at the inputs and that the computation of the algorithm should begin.

Figure 6-17 shows a loop with a NEXT statement and the corresponding state machine.

Figure 6-17: NEXT statement used for input handshaking

Note that the condition being tested is \( \text{start} \neq '1' \). This means that if the value of the signal start is not ‘1’, control should be transferred to the next iteration of the loop, in which case the design waits for a clock cycle and tests the value of the signal start again. In other words, the value ‘1’ indicates that the input data is valid.
6.6 Scheduling Loops

The following section describes how behavioral synthesis tools schedule loops that appear in behavioral descriptions. The section depicts the corresponding state machines that are automatically constructed by such tools.

A loop is represented by one or more states in the state machine that is synthesized for the design. The general structure of the state machine that is implied by a loop is shown in Figure 6-18.

![Figure 6-18: The general structure of the state machine implied by a loop](image)

The operations that are used to make the decision to enter a loop or to bypass a loop are scheduled in the state prior to the start of the loop. In the case of a WHILE loop, the exit condition is calculated outside the loop. If the condition is false, the state machine transitions to the first state of the loop (L_1). If the condition is true, the state machine branches to the first state after the end of the loop (S_n+1).

Entering a loop corresponds to entering the first state that represents the loop (L_1). When the end of the loop is reached (represented by the state L_n), the exit condition is calculated once again. If the condition is now true, the loop is complete and the state machine exits the loop by transitioning to the first state after the loop (S_n+1). If the condition is false, the state machine branches back to the first state of the loop (L_1) to perform another iteration of the loop.
6.6.1 Clock Cycles and Control Steps

A loop is represented on a Gantt chart in a manner that represents this behavior. Figure 6-19 shows a loop and a Gantt chart that represents a schedule for that loop.

The loop is delineated from other operations in the chart so that the start and end of the loop are easily recognizable. The loop itself is displayed in a Gantt chart as requiring one or more control steps (or c-steps).
Until now, this book has referred to clock cycles when discussing the scheduling of operations. A more correct term is c-steps, or control steps. This distinction is made because the term clock cycles implies a particular delay.

The source code indicates that this design might take 3 clock cycles to compute the value of dout, or it might take 20 clock cycles to compute – the number of clock cycles required to compute the result depends upon the value of the input signal n.

For loops, the Gantt chart indicates the number of clock cycles required to perform one iteration of the loop. This information is based on the schedule of the design and is not dependent upon the input data.

Thus, this design is said to be scheduled in 3 c-steps, even though it may require many more than 3 clock cycles to compute the value of dout. It could in fact take less than 3 clock cycles if the loop is not executed at all.

6.6.2 Minimum Loop Execution Time

A loop is represented by one or more states in the state machine that is synthesized for the design. The operations that are used to make the decision to enter a loop or to bypass a loop are scheduled in the state prior to the start of the loop. The loop contents are scheduled onto one or more states. In the last state representing the body of the loop, the decision is made to either exit the loop or perform another iteration of the loop. This general structure was shown in Figure 6-18.

Consider the two pieces of code in Figure 6-20 which might be used to wait for a start signal to go high before reading other input signals. Assume that the signals a and b are valid only in the clock cycle when the signal start is ‘1’.

The code on the left was introduced in a previous section. This code uses a NEXT statement to wait until the signal start goes to the value ‘1’. The code waits for a clock cycle and then tests the value of start. If the value is ‘0’, the code begins again at the start of the loop, waiting a clock cycle and re-testing the value.
The code on the right uses a WHILE statement. In this code, the WHILE loop is executed as long as the value of start remains at '0'. The loop is exited as soon as the value changes to '1'.

At first glance, these two pieces of code may appear functionally equivalent, but they are not. In the code on the left, the values of a and b are read in the same clock cycle as when the signal start goes to '1'. In the code on the right, the values of a and b are read in the next clock cycle after the signal start goes to '1'. This is shown in Figure 6-21.

It may seem that the way to “fix” the behavior of the code with the WHILE loop is to remove the WAIT statement that is highlighted in the diagram. But depending on the tool used, the scheduled design may still behave as if the WAIT statement was there — the signals a and b will be read in the following cycle.

There are often many ways to express the same or similar behavior. If a particular interface protocol is required then care must be taken to ensure that the description of the algorithm conforms to the desired protocol and that it is feasible for the behavioral synthesis tool to generate a suitable schedule.
6.7 Loop Unrolling

Loop unrolling is a transformation that can be applied to a loop construct. When a loop is unrolled, the statements inside the loop are copied as many times as the loop is being unrolled. When a loop is unrolled, the control structure of the loop is modified to maintain the same functionality – unrolling a loop does not modify the behavior of the loop. There are two types of unrolling: complete unrolling and partial unrolling.

6.7.1 Complete Unrolling

If the number of times a loop will be executed is known during synthesis, the loop can be completely unrolled. FOR loops with constant bounds can be completely unrolled. Since the bounds of the loop are constant the number of iterations is known.

To completely unroll a FOR loop, the statements inside the loop are copied as many times as the loop would be executed. For each copy of the loop statements, the iteration variable is replaced with the appropriate constant for that iteration of the loop. The loop construct is completely replaced by the copied statements. Recall that this is how FOR loops are processed by RTL synthesis tools.

Infinite loops, WHILE loops, and FOR loops with non-constant bounds can not be completely unrolled because the number of times the loop will be performed is data-dependent and thus not known when the design is being synthesized.

Figure 6-22 shows a FOR loop with constant bounds and the equivalent code after the loop has been completely unrolled.

\[\text{FOR } i \text{ IN 0 TO 7 LOOP} \]
\[\text{tmp} := \text{tmp} + s(i);\]
\[\text{END LOOP};\]

\[\text{tmp} := \text{tmp} + s(0);\]
\[\text{tmp} := \text{tmp} + s(1);\]
\[\text{tmp} := \text{tmp} + s(2);\]
\[\text{tmp} := \text{tmp} + s(3);\]
\[\text{tmp} := \text{tmp} + s(4);\]
\[\text{tmp} := \text{tmp} + s(5);\]
\[\text{tmp} := \text{tmp} + s(6);\]
\[\text{tmp} := \text{tmp} + s(7);\]

Figure 6-22: Completely unrolling a FOR loop with constant bounds

Some behavioral synthesis tools unroll constant-bound FOR loops by default, others do not. Whatever the default behavior, there is an appropriate mechanism to direct the behavior of the synthesis tool.

It is not always easy to decide whether or not it is best to completely unroll a loop. It is necessary to evaluate the particular loop to make this determination.

A loop should only be unrolled if it can reduce the number of cycles required to perform the algorithm.
Although not required, it is generally best to unroll certain kinds of loops. When a loop has been used simply as a coding convenience, the loop should be unrolled. Consider the loop in Figure 6-23.

```
FOR i IN 0 TO 6 LOOP
  s(i) := s(i^2 + 1);
END LOOP;
```

Figure 6-23: Unrolling a FOR loop with fixed bounds

There is no benefit to keeping this loop rolled. This loop is only used to simplify coding. When the loop is left rolled, 7 clock cycles are required to execute the loop. In addition, hardware is required to control the loop, such as an incrementer and a comparator. Furthermore, the indexed read and indexed write to the array s both imply significant hardware (see Section 4-8). But when completely unrolled, the loop only infers wires in the synthesized design.

When a loop contains more substantive operations (such as add and multiply operations), the decision to unroll is more difficult. If a loop is entirely unrolled, there is maximum opportunity to share or chain operations between iterations of the loop. This will reduce the number of clock cycles required to execute the loop. But this usually requires additional hardware.

Consider again the loop shown in Figure 6-22. Assume that the loop is left rolled. If the clock cycle is 50 ns and an add component has a delay of only 3 ns, it will still take 8 clock cycles (400 ns) to execute the loop. The state machine that controls a loop requires at least one clock cycle per iteration of the loop. So when left rolled, the loop implies that one (and only one) addition is performed per clock cycle.

But the number of clock cycles can be reduced if the loop is unrolled. In fact, if the loop is completely unrolled, the loop functionality can be performed in a single clock cycle (50 ns)! Unfortunately, this requires 8 adders in the synthesized hardware.

```
Completely unrolling a loop can increase parallelism but the parallelism comes at a cost of increased hardware. The decision to unroll a loop should be based on the relative cost and benefit of these two factors.
```

Constant-bound loops that perform a large number of iterations should not be completely unrolled. Unrolling a loop usually results in a greater number of operations that must be scheduled (remember that all the statements inside of the loop are duplicated). Unrolling a loop a large number of times will almost always result in longer scheduling run times in the synthesis tool.
6.7.2 Partial Unrolling

Some behavioral synthesis tools support the partial unrolling of loops. To partially unroll a loop, the statements inside the loop are copied as many times as the loop is to be unrolled. The control structure of the loop is appropriately modified to maintain the functionality of the loop.

The loop that was completely unrolled in Figure 6-22 can be partially unrolled. Consider unrolling such a loop 2 times. The translated code that results from the unrolling is shown in Figure 6-24. The translated code shows some obvious “optimization” — the exit condition is set to “false” prior to the start of the loop, as that is the evaluation of the expression “0 > 7”.

There is now an opportunity to create a more parallel implementation. If enough hardware is allocated to perform the operations in the loop in the time of a single clock cycle, the entire loop can be performed in 4 clock cycles instead of 8 clock cycles.

The partial unrolling of the loop does not eliminate the hardware required to perform the indexed read and write to the array s. This hardware is not needed only if the indexed references to s are always constant. This is only true if the loop is entirely unrolled.
But the loop functionality can also be achieved in 4 clock cycles by completely unrolling
the loop, but limiting the number of adders that can be used in the design. Remember
that if the loop is completely unrolled, the loop functionality could be performed in one
clock cycle, given 8 adders can be used. If only 2 adders are available, the calculation
must be spread over 4 clock cycles. This eliminates the hardware required to index s but
requires the same number of clock cycles as unrolling the loop 2 times.

The previous example focused on partially unrolling a loop with constant bounds. It is
also possible to partially unroll infinite loops, WHILE loops, or FOR loops with non-
constant bounds. For these loops, the number of times the loop will execute is data
dependent.

When a loop with non-constant bounds is partially unrolled, the exit condition is
generated and tested between each copy of the loop contents. This is necessary since
the loop may complete at the end of any copy of the loop contents. Figure 6-25 shows
an example of unrolling a WHILE loop 3 times.

6.7.3 When to Unroll

It is not always easy to decide whether or not it is best to unroll a loop. It is necessary to
understand the design to make this determination.

First, unrolling a loop results in a greater number of operations that must be scheduled.
Unrolling the loop may lead to superior results in terms of schedule length or required
resources, but unrolling a loop will almost certainly result in longer scheduling run times.
If a loop is entirely *unrolled*, there is maximum opportunity to share or chain operations between iterations of the loop. When entirely unrolled, the loop no longer exists in the design.

If a loop is left *rolled*, there is no opportunity to share or chain operations between different iterations of the loop.

Figure 6-25 shows a loop that is scheduled with the loop left rolled, with the loop completely unrolled, and with the loop unrolled 2 times.

```
FOR i IN 0 TO 7 LOOP
  tmp := tmp + s(i);
END LOOP;
```

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Rolled</th>
<th>Completely Unrolled</th>
<th>Unrolled 2 Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resources</td>
<td>1 adder</td>
<td>4 adders</td>
<td>2 adders</td>
</tr>
<tr>
<td></td>
<td>1 comparator</td>
<td></td>
<td>1 comparator</td>
</tr>
<tr>
<td></td>
<td>1 incrementer</td>
<td></td>
<td>2 incrementers</td>
</tr>
<tr>
<td></td>
<td>indexing logic</td>
<td></td>
<td>indexing logic</td>
</tr>
<tr>
<td>Latency</td>
<td>8 clock cycles</td>
<td>2 clock cycles</td>
<td>4 clock cycles</td>
</tr>
</tbody>
</table>

Figure 6-25: Simple loop and three schedules (with required resources and latency)

The schedules assume an unlimited number of resources, and a clock cycle large enough to schedule 4 add operations. The resources required to implement the schedules and the number of clock cycles required to perform the loop calculation (latency) are also shown.
6.8 Summary

Loops are fundamental building blocks of behavioral descriptions. Loop structures provide a powerful mechanism for describing the flow of control in an algorithm.

A central feature of behavioral synthesis is the ability to automatically construct a state machine to control the use of resources in a design. The state machine that is implied by a loop structure is also automatically generated.

Unlike RTL synthesis tools, which support only constant-bound FOR loops, behavioral synthesis tools support FOR loops with both constant and non-constant bounds, WHILE loops, and infinite loops. In an RTL design methodology, the use of loop structures is discouraged. Behavioral synthesis methodologies encourage the use of loops to allow designers to describe algorithms at a higher level of abstraction.

Loops can be left rolled or unrolled entirely or partially. Unrolling can dramatically influence the final schedule, but will also impact run-time.

Today's behavioral synthesis tools do not automatically determine when and how loops should be unrolled. This discussion highlights the issues the designer must carefully consider when making these decisions.