9 Assertions and Sequences

The testing that has been demonstrated to this point has been direct testing. *Direct testing* is a method where the designer generates an input vector with the sole purpose of testing for a specific fault or set of faults. In the combinational design, this amounted to generating all 2^n values for n inputs and checking to see that the output values were correct. In the the design of FSMs, the testbench generated a sequence of inputs that would drive the FSM through all of its state transitions. Although each of the test vectors may have uncovered more than the intended fault, the point is that the test vectors were generated with the purpose of directly testing a specific fault set.

Assertions provide a means of *indirectly* testing a general property of the design. An example of a property might be that: the value in the FSM state register will only be valid state assignments defined in an enumeration. Then as the system operates with a set of test vectors for a different fault, if that property is violated, the assertion will flag the error. Another situation may be the operation of a system bus. Read and write protocols (sequences of input values) are defined for the bus and assertions are written to check for violations of these protocols. Then a processor on the bus is simulated executing a program from the memory. Since the processor fetch and program execution need to access the memory, the assertions check if there is any violation of the bus protocols.

Notice that the test vectors in the first example, and the program's instructions in the second, weren't generated to find specific faults. Indeed, they could have been randomly chosen. Rather, the system was being run through its general functionality while the assertions watched for violations.

9.1 Introduction

There are two basic types of assertions: immediate and concurrent.

- An ***immediate assertion*** is a procedural statement and thus can be used anywhere a procedural statement is allowed. Since an immediate assertion is typically part of the

testbench, it will be in an initial block inside a program module. Its function is to check the condition given and report if an error has occurred. When completed, and assuming there is no error, execution continues with the next procedural statement.

- A ***concurrent assertion*** is an assertion that is always active. It acts as a separate (concurrent) element whose sole purpose is to check a property that is specified in the assertion. Once started it continues to execute.

Immediate assertions will be discussed in this section along with other preliminary information to provide a quick understanding of what an assertion might check, and to show how they fit into the simulation kernel. Concurrent assertions, a much larger set of topics, is left for later sections of the Chapter.

```
1  module adder;
2    logic  [3:0]  a, b, s;
3    logic  [3:0]  co;
4    logic        cIN;
5
6    full_add b0 (s[0], co[0], a[0], b[0], cIN);
7    full_add b1 (s[1], co[1], a[1], b[1], co[0]);
8    full_add b2 (s[2], co[2], a[2], b[2], co[1]);
9    full_add b3 (s[3], co[3], a[3], b[3], co[2]);
10
11   initial begin: testbench
12     a = 3; b = 2; cIN = 0;
13     #1 $display
14       ("time=%d, a=%b, b=%b, cIN=%b, s=%b, co=%b",
15          $stime, a, b, cIN, s, co);
16     checkadd0: assert (s === a + b + cIN)
17       $display ("%m works! a=%b, b=%b, cIN=%b, s=%b,\
18             co=%b", a, b, cIN, s, co);
19       else
20         $error ("%m says no cigar! a=%b, b=%b, cIN=%b,\
21             s=%b, co=%b", a, b, cIN, s, co);
22 ...  // other code omitted
23   end
24 endmodule: adder
25
26 module full_add
27   (output    sum, co,
28    input     a, b, cin);
29
30    xor (sum, a, b, cin);
31    assign co = a&b | a&cin | b&cin;
32 endmodule: full_add
33
34 time=        1, a=0011, b=0010, cIN=0, s=0101, co=0010
35 adder.testbench.checkadd0 works! a=0011, b=0010, cIN=0,
                                    s=0101, co=0010
```

Example 9.1 — Immediate Assertion and Printout

9.1.1 An Immediate Assertion Example

An example of an immediate assertion is shown in Example 9.1. The example is of a 4-bit adder made out of four instantiated 1-bit full adders. The module definition of the full adder is shown on lines 26-32; the full adder is then instantiated 4 times in the adder module to create a structural model of the 4-bit adder (lines 6-9). On lines 11-23, an initial block that sets the inputs to the adder and observes the output is shown. As a simple illustration, the initial block sets the inputs to a=3, b=2, and cIN=0. The initial block delays one time unit, allowing the values to propagate, and then $displays the results. The simulation output (line 34) of the example shows that the results $displayed are correct: 3+2 equals 5.

An immediate assertion is then executed (line 16) to check that the result is correct. The form of the immediate assertion is:

```
1  label: assert (expression)
2        pass_Statement else fail_Statement;
```

The label and pass_Statement are optional. The immediate assert acts as an "if" statement. If the expression is TRUE, the pass_Statement is executed; otherwise the fail_Statement is executed. Since most verification engineers only want to know if there was an error, the pass_Statement is normally omitted.

The simulation output of the assertion statement printing its pass_Statement is shown on line 35.

There are several important points to note from this simple example:

- An immediate assertion is exactly like an if statement but it's clear that a check of functionality is being done. There is no mistake that this is not part of the functionality of the design or testbench.
- The detailed functionality of the interconnected modules (lines 6-9) is being compared with a relatively simple statement of their functionality (line 16). This is typical when validating a design. Since all of those instantiated modules and interconnections are supposed to implement a + b + cIN, that's the check that is made. i.e., there's no reason to compare a complex design with a separate complex design — you'll specify one of them incorrectly! Probably both.

Now, let's change one character in module adder's definition: we'll change line 9 to

```
1   full_add b3 (s[3], co[3], a[3], b[3], co[1]);
```

Note that the carry in is specified incorrectly as the carry output of stage 1 (co[1]) rather than (the correct) stage 2. When the testbench is run again, the resulting printout will be

```
1   time=         1, a=0011, b=0010, cIN=0, s=1101, co=0010
2   "ImmAssertBasic.sv", 14: adder.testbench.checkadd0: started at 1s failed at 1s
3         Offending '(s === ((a + b) + cIN))'
4   Error: "ImmAssertBasic.sv", 14: adder.testbench.checkadd0: at time 1
5   adder.testbench.checkadd0 says no cigar! a=0011, b=0010, cIN=0, s=1101, co=0010
```

Line 1 is the $display statement printing but note that s=4'b1101. Clearly this is not the binary result of 2+3! Line 5 shows the immediate assertion's else clause with its $error statement printing. This statement is like a $display statement in that it prints the message given. However, in contrast to the $display, it causes a runtime error. Lines 2-4 are printed by the simulator.

Note that "%m" is used in the print string (lines 17 and 20 of Example 9.1). This prints the hierarchical context of the statement so it is easier to find what caused the printing. In this case the hierarchical context is its label checkadd0 which is inside the begin-end block named testbench which is inside module adder; thus "adder.testbench.checkadd0."

In addition to $error, there are other tasks that can be used to indicate the level of severity of an error. These are available only in an assertion statement.

- ***$fatal*** — A run-time fatal error is generated which also implicitly calls $finish which stops the simulation. It has two arguments. The first argument passed to $fatal is also passed to $finish (by default it is 1). The second argument is the string to be printed.
- ***$error*** — Called to indicate a run-time error. It's argument is a print string.
- ***$warning*** — Called to indicate a run-time warning. It's argument is a print string.
- ***$info*** — Called to indicate an issue with no specific severity. It's argument is a print string.

For all of these, the print string uses the same syntax as a $display task.

Immediate assertions can be used in other contexts. For instance, if a testbench is opening a file to print some results, the code could be:

```
1   open: assert ((filed = $fopen("fname.dat", "w")) != 0)
2     else $error ("oops! %m can't open file %s", "fname.dat");
```

Using immediate assertions in this way simplifies the coding of bothersome error conditions that need to be checked!

9.2 Introduction to Concurrent Assertions

A concurrent assertion is an independently executing element that performs checking on a design. The purpose of a concurrent assertion is to check that a property of the design holds over the time that a system is being simulated. The property is defined and then a concurrent assert statement activates the checking of the property. The assertion is clock based, following the RT timing model. It executes in the simulation kernel's observed region based on values sampled in the kernel's preponed region.

9.2.1 Defining and Asserting a Property

A *property* defines the behavior of a design to be checked. Properties can be quite complex and can be made up of several sequences of actions. An analogy would be to say that they are regular expressions that include clock edges.

A concurrent assertion is asserted in the following manner:

```
1   label: assert property (pname) pass_Statement else fail_Statement;
```

The words assert, property, and else are keywords of the language. The label is a label for the statement, pname is the name of a property, pass_Statement is the statement executed when the property passes correctly, and fail_Statement is the statement executed if the property fails. The label and pass_Statement are optional.

```
1   module simpleAssert;
2     bit  q, r, s, ck;
3     ...
4     property q1r3s;
5       @(posedge ck) q ##1 r ##3 s;
6     endproperty
7
8     assert property (q1r3s) else $error("oops");
9   ...
```

Example 9.2 — A Concurrent Assertion

We start with a very simple property example. A property named q1r3s is defined on lines 4-6 of Example 9.2. Line 4 specifies the property's name. Its action (line 5) is that at the positive edge of clock ck it will check if q is TRUE. If it is, then one clock tick later (as specified by ##1) r should be TRUE and then three clock ticks after that s should be TRUE.

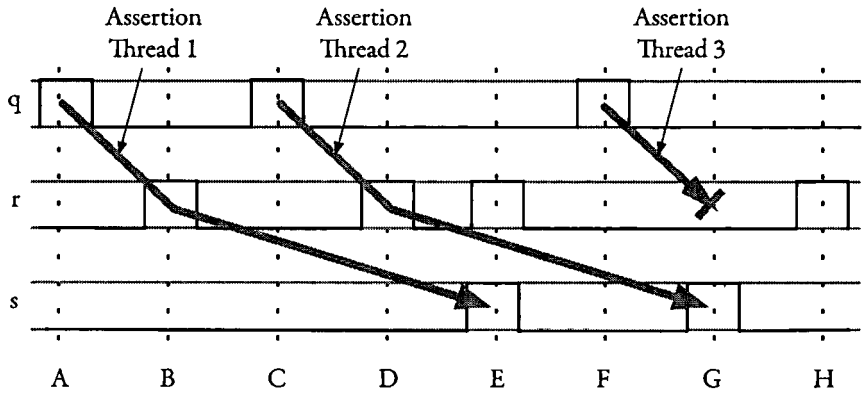

Figure 9.1 — Sequence Examples for property a1b3c

The property is asserted on line 8 using the syntax specified above. In contrast to an immediate assertion, a concurrent assertion includes the keyword "property" and the property's name. Here, only a fail-Statement is provided, specifying to print "oops". Historically, other four letter words have been used.

The property definition and assertion are defined inside a module but outside of procedural constructs such as always and initial blocks.

The property specifies the following sequence of values, shown in Figure 9.1 as waveforms. Here we see that at clock tick A that q is TRUE, r is TRUE one tick later at tick B, and s is TRUE 3 ticks after that at clock tick E. The assert statement on line 7 would follow this sequence and find that the property is successfully matched. If a pass_Statement was provided, it would be executed. Since none is specified, the assertion succeeds quietly.

The above discussion follows what is labeled "Assertion Thread 1" in the figure. The way to think about how this works is that when the assertion sees the first variable (q) TRUE, it sets off an *assertion thread* that follows the rest of the assertion. That thread executes in the observed region of the simulation kernel and eventually either succeeds or fails.

Assertion Thread 2 is also shown in the figure. When the assertion sees the start of the second sequence (at tick C), it sets off another assertion thread to follow it. These two threads overlap in time between ticks C and E but they are separate; each thread will either succeed or fail on its own.

Assertion Thread 3 starts up at clock tick F but fails at tick G where it expects to see r TRUE.

Note what the assertion is checking for in each of these cases: only that q is TRUE now, r at the next tick, and s three ticks later. It is not checking that r is FALSE at the second tick after q, or that s is FALSE two ticks after r. As illustrated in Figure 9.1, r being TRUE at clock ticks E and H have no bearing on any of the assertion threads; these values are completely ignored.

A new semantic was introduced here, the ##n clock *cycle delay operator*. Instead of delaying by a specific amount of time as would happen with a #n delay, ##n specifies the delay in terms of the number of clock ticks to delay. In a description such as Example 9.2, the clock and its active edge are inferred by the @(posedge ck) statement on line 5. Thus it would delay for n positive clock edges on ck.

The ##n cycle delay operator can also be used in procedural blocks as shown in Example 9.3. You might consider using it in program blocks instead of "@(posedge ck);" as shown there. When using the cycle delay operator in an assertion, the clock is easily inferred. The same is not typically the case for procedural blocks, so you need to specify the clock to be used. This is done with the default clocking statement on lines 4-6. This names what is called the clocking block (cName) and, in this case, specifies the clock variable and active edge on line 5. On lines 17-20, the clock ck is initialized to 0; positive edges will occur on the fives (5, 15, 25, 35,

```
1   program cycleDelay;
2     bit ck, q;
3
4     default clocking cName
5       @(posedge ck);
6     endclocking
7
8     initial begin
9       q <= 0;
10      ##2 q <= 1;
11      ##3;
12      ##1 q <= 0;
13      ##1;
14      $finish;
15    end
16
17    initial begin
18      ck = 0;
19      forever #5 ck = ~ck;
20    end
21  endprogram: cycleDelay
```

Example 9.3 — Using Cycle Delay Operators in Procedural Code

...). Following the initial block on lines 8-15, q become 0 at time 0, changes to 1 at time 15 (on the second positive edge), and then changes to 0 at time 55.

9.2.2 Properties and Sequences

The property shown above used the variable names q, r, and s which are defined as bit variables in the example. Properties can also be specified with formal parameters. Then the assertion statement can provide the actual parameters to be used. Consider the following equivalent code for Example 9.2.

```
1   property a1b3c (a, b, c);
2     @(posedge ck) a ##1 b##3 c;
3   endproperty
4
5   assert property (a1b3c (q, r, s)) else $error("oops");
```

The only difference here is that the property uses formal parameters. Then when the property is asserted on line 5, variables q, r, and s are substituted for a, b, and c respectively.

Sequences of values in a design are important enough that they can be defined and given a name using the keyword "sequence." In the case of the sequence in the above property, it could have been defined as:

```
1   sequence abxxc (a, b, c);
2     a ##1 b ##3 c;
3   endsequence
```

Then a property could have been defined to use that sequence:

```
1   property checkQRS (q, r, s);
2     @(posedge ck) abxxc (q, r, s);
3   endproperty
```

Note here that both the property and sequence use formal parameters enabling for maximum flexibility in their use.

It's possible that sequence abxxc is only part of a sequence to be checked. For instance, if variable d should be TRUE two ticks after abxxc, we could check for the property:

```
1   property checkQRSD (q, r, s, d);
2     @(posedge ck) abxxc (q, r, s) ##2 d;
3   endproperty
```

9.2.3 How to Think About the Execution of Assertions

We've stepped through an example based on Figure 9.1. Now let's formalize how it works. A concurrent assertion will try to recognize the start of its sequence at every tick of the clock. Consider property checkQRS above and the following assert statement:

```
1   assert property (checkQRS (q, r, s)) $display("Yesssss!") else $error("Noooo!");
```

This statement asserts the specified property with the actual parameters given. Based on this activation, the property will try to start and match its sequence on every clock edge, as specified in the property and its sequence. One of five things can happen at each clock tick:

- The property sees the start of its sequence. In this case q was TRUE. Nothing is printed and an assertion thread starts up to follow the sequence.
- The property doesn't see the start of its sequence because q is FALSE. In assertion-speak this is termed a vacuous success. A *vacuous success* means that it succeeded in *trying* to start an assertion thread, but the thread didn't start. The property continues trying to recognize the start of its sequence at the next clock tick.
- The assertion thread, having seen the start of its sequence earlier, didn't see one of the subsequent steps. In this case the assertion fails, the assertion thread stops, and the fail_Statement is executed. In this case it would print "Noooo!" and continue with the simulation.
- The assertion thread, having correctly seen an earlier part of its sequence, sees the next (but not last) step. It continues watching for future steps.
- The assertion thread, having correctly seen all but the last step in the sequence (sometimes called the *penultimate* step), sees the last step. The assertion "passes" and executes the pass_Statement, in this case printing "Yesssss!". At this point, the assertion threads stops as there is no more sequence to follow.

9.2.4 A State Transition Diagram View of a Sequence

Sequence abxxc above can be thought of as a state transition diagram as shown in Figure 9.2. As in any state transition diagram in logic design, the clock tick drives the change of state as per the inputs of the system. In this case, the inputs are the inputs to the sequence.

In state 1, if q is not seen, then it's a vacuous success and the sequence stays in state 1. If q is TRUE, then state 2 is entered. When in state 2, if r is TRUE then state 3 is entered. If r is FALSE in state 2, the assertion fails. Three clock ticks later when in state 5, if s is TRUE, then the assertion passes. If not, it fails.

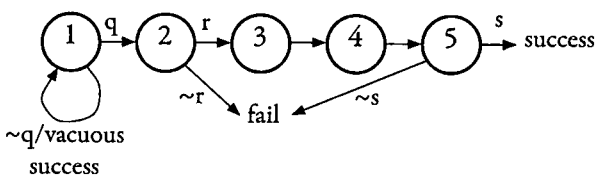

Figure 9.2 — State Transition Diagram for Sequence abxxc

As mentioned above, the assertion thread only checks for the specified variables (or expressions) to be TRUE. Thus, for instance, the assertion thread doesn't check for s to be FALSE in states 3 and 4.

An analogy for how SystemVerilog assertions and their sequences work is regular expressions; there is a one-to-one translation between regular expressions and state transition diagrams.

9.2.5 More Detail on the Previous Example

An example will show how many of these details come together in the execution of an assertion as it follows a simple waveform. The example waveform is still the one shown in Figure 9.1 where the sequence to check is (q ##1 r ##3 s).

A sequence and property written for this situation could be the following:

```
1  sequence s1 (q, r, s);
2    (q ##1 r ##3 s);
```

```
3   endsequence
4
5   property p1 (q, r, s);
6     @(posedge ck) s1(q, r, s);
7   endproperty
8
9   assert property (p1 (q, r, s)) else $error("oops");
```

In this case, we have specified the positive edge of the clock and we have used variables q, r, and s in all places to help keep the explanation clearer. Given the description of what an assertion does at each clock edge (Section 9.2.3), at every posedge of clock ck, the property tries to assert sequence s1 (line 6 above). Of course, since q isn't always TRUE at each clock tick, there will be vacuous successes that will cause the assertion to wait for the next clock tick.

```
1   module assertQRS;
2     bit       q=0, r=0, s=0;
3     bit       ck=0;
4
5     always #5 ck = ~ck;
6
7     initial begin
8       $monitor($stime,,,
9         "ck=%b, q=%b, r=%b, s=%b",
10          ck, q, r, s);
11      q <= #4 1;
12      q <= #6 0;
13      r <= #14 1;
14      r <= #16 0;
15      q <= #14 1;
16      q <= #16 0;
17      r <= #24 1;
18      r <= #26 0;
19      s <= #44 1;
20      s <= #46 0;
21      #56 $finish;
22    end
23
24    sequence s2(r, s);
25      (r ##3 s);
26    endsequence
27
28    property checkQRS (q, r, s);
29      @(posedge ck)  q |=> s2(r, s);
30    endproperty
31
32  P1a: assert property (p2(q, r, s))
33           $display("%d Yes!", $stime);
34           else $error("%d oops", $stime);
35  ...
```

Example 9.4 — Simulation of checkQRS

Most assertions of this type are started with an *implication construct* that only starts an assertion if a specific condition is TRUE. In this way, a vacuous success is avoided. An implication takes one of two forms:

```
1  i |=> j;     // i TRUE now implies that j will be TRUE at
                the next clock tick
2  i |-> j;     // i TRUE now implies that j is TRUE at the
                current clock tick (now)
```

where, in each case here, i is an expression that evaluates to TRUE or FALSE, and j is typically a sequence. If i is TRUE, the property tries to start recognizing the sequence. Implications such as this are only used within a property specification (property..endproperty). The difference between the two forms of implication is when the sequence is to start. With the first form (|=>), the first step of the sequence is checked for at the following clock tick. With the second form (|->), the first step of the sequence should be TRUE now, in the current time.

Our sequence and assertion can now be rewritten using the |=> implication construct as shown on lines 24-30 of Example 9.4.

Two changes have been made to the sequence and property statement. First, the sequence has been changed to include all but the first variable: thus instead of the sequence being (q ##1 r ##3 s), it will now be (r ##3 s) as shown on line 25. Second, the q will now be used as part of an implication in the property on line 29. The way to read line 29 is: at the positive edge of ck, if q is TRUE, then begin asserting sequence s2 at the next negative clock edge. This still results in the same sequence; the ##1 in the original sequence is now specified as part of the implication.

Example 9.4 shows the whole description; the printout obtained from simulation is shown in Example 9.5. Figure 9.3 shows the timing diagram of the variables q, r, s as generated from Example 9.4. The point of showing all of this detail is to provide the reader with a complete example with detailed timing.

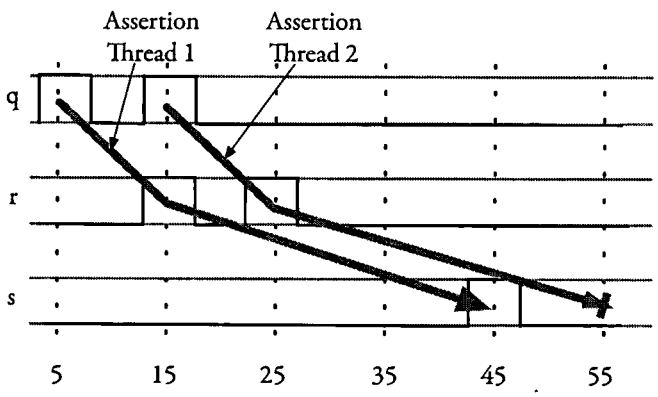

Figure 9.3 — Timing Diagram for Example 9.4

Note in the printout portion of the example that at time 45 "Yes!" is printed. This is assertion thread 1 succeeding at time 45 and printing its pass_ Statement. Assertion thread 2 starts at time 15 fails and prints "oops" at time 55 because it expects s to be TRUE.

A fine detail should not be overlooked at this point. Notice how the timing waveforms for q, r, and s were generated on lines 11-20 in Example 9.4. Specifically, q is set to 1 at time 4 and set back to 0 at time 6; this creates the first pulse on q that is recognized at time 5 by an assertion. One might ask if q could be set to 1 at time 5, when the clock edge occurs, and whether the assertion would still see it. These two situations are shown in Figure 9.4. The answer is *no* because the values used in assertions are the values sampled in the preponed part of the simulation kernel. As shown in the figure, if q changes at time 4, then the preponed value at time 5 of q is 1. However, if q changes at time 5, the preponed value of q is 0. That is, q has to be 1 *before* time 5 for it to be seen as 1 by an assertion at time 5.

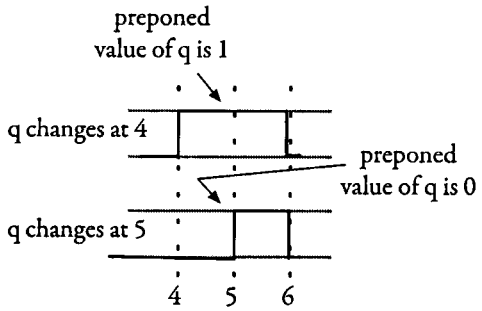

Figure 9.4 — Sampling Values in the Preponed Region

```
1          0 ck=0, q=0, r=0, s=0
2          4 ck=0, q=1, r=0, s=0
3          5 ck=1, q=1, r=0, s=0
4          6 ck=1, q=0, r=0, s=0
5         10 ck=0, q=0, r=0, s=0
6         14 ck=0, q=1, r=1, s=0
7         15 ck=1, q=1, r=1, s=0
8         16 ck=1, q=0, r=0, s=0
9         20 ck=0, q=0, r=0, s=0
10        24 ck=0, q=0, r=1, s=0
11        25 ck=1, q=0, r=1, s=0
12        26 ck=1, q=0, r=0, s=0
13        30 ck=0, q=0, r=0, s=0
14        35 ck=1, q=0, r=0, s=0
15        40 ck=0, q=0, r=0, s=0
16        44 ck=0, q=0, r=0, s=1
17        45 Yes!
18        45 ck=1, q=0, r=0, s=1
19        46 ck=1, q=0, r=0, s=0
20        50 ck=0, q=0, r=0, s=0
21 "ConcurrentAssertBasicBook.sv", 32:
22 assertQRS.P1a: started at 15s failed at 55s
23   Offending 's'
24 Error: "ConcurrentAssertBasicBook.sv", 32:
25 assertQRS.P1a: at time 55
26        55 oops
27        55 ck=1, q=0, r=0, s=0
28 $finish called from file
29 "ConcurrentAssertBasicBook.sv", line 20.
30 $finish at simulation time                56
```

Example 9.5 — Simulation Results of assertQRS

9.3 Sequences With Ranges and Repetitions

Sequence are widely use in specifying detailed protocols in digital systems and there is an extensive set of operators that can be used to specify the details of these. This section de-

scribes a simple bus protocol and then uses it to illustrate how sequences can be used to check logic models of the protocol.

9.3.1 The SimpleBus Definition

The SimpleBus protocol is shown in Figure 9.5. It connects a processor and memory across a bus with a simple protocol. This same example was used in Chapter 6.3. The processor exclusively drives the values address, start, and read. The values data and dataValid are shared bus lines that can be driven by either the processor or the memory. If it's a read, the memory drives the data lines with the value read from memory and it sets dataValid to indicate to the processor that the value is on the data lines. If it's a write, the processor drives the data lines with the value to be written into the memory. It sets dataValid to indicate to the memory that the values are on the data lines. The address bus is multiplexed with the top 8 bits of the address being sent first followed by the lower 8 bits.

A read operation starts in state S1 with the processor driving the upper address on the address bus and asserting start. In state S2, it drives the lower address on the address bus and asserts read if this is to be a read of the memory. If it's a write, then read is not asserted. The memory is initially watching for start to be asserted and when it sees the start signal, it loads the upper address from the address bus lines. In the next state (S2) it automatically reads the lower address from the address bus lines. It also sees the status of read.

Since Figure 9.5 is showing a read bus cycle, dataValid is changed from z to 0 by the memory in state S3. This is because it will be the signal for the processor to read the data lines which will be driven by the memory. This is shown happening in state S5. Note that state S4 is a place holder for many states while the processor waits for the memory to read the requested value and place it on the data lines (or where the memory waits for the processor to send the value to be written). In state S6, the data and dataValid lines become not-asserted again; the next bus cycle could start in this state with the assertion of the address lines and the start signal.

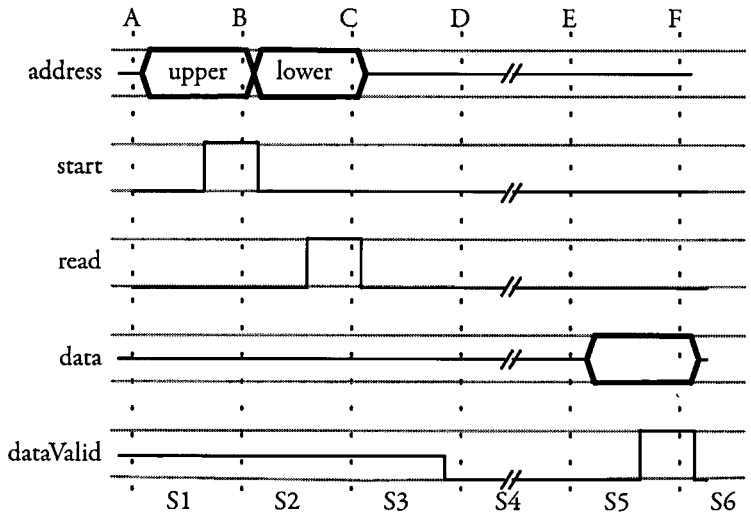

Figure 9.5 — SimpleBus Protocol Timing Diagram

The signal lines for a bus write have only one difference: read is not asserted by the processor. In the case of a write, all the rest of the lines are the same. The distinction is that in a write bus cycle, the processor drives the data and dataValid lines and the memory uses these to load the value on the data into the memory.

Figure 9.5 also shows many of the waveform details that are not specifically stated here. For instance, start and read are only asserted for one state time; otherwise they are not-asserted. Or that the earliest time that start can appear is in the state after dataValid (which would be S6 here).

9.3.2 A property and sequences for the protocol

The basic property for this bus protocol would start out as:

```
1  property x;
2    @(negedge ck) start |=> //some sequence
3  endproperty
```

The point being that the signal start is what sets off the protocol and we can use it to start up certain sequences to check various features of the protocol. These sequences can either be defined separately as sequences and then invoked on line 2, as was done in the earlier sections, or the sequence can be written on line 2 as part of the property. To make the discussion easier to follow, we'll write the sequences as part of the property.

First, let's assume that there is no delay state (S4 in Figure 9.5). The new version of the waveforms are shown in Figure 9.6. Now, read and write cycles take three clock ticks. A simple assertion to check that a read bus cycle starts and ends in three ticks of the clock would be:

```
1  property readIs3ticks;
2    @(negedge ck) start |=> read ##1 dataValid;
3  endproperty
```

This property checks that if start is valid now (clock tick B), read will be valid at the next clock tick (as specified by the |=> implication) and dataValid will be asserted one tick later.

SystemVerilog provides some useful system calls for determining if certain values are unknown. The task $isunknown(y) returns TRUE if the expression y is unknown (i.e., it contains either x or z values). Since the address bus is normally high impedance, we could check to make sure that the address lines are driven when start is asserted:

```
1  property addressBusDriven;
2    @(negedge ck) start |-> ~$isunknown(address);
3  endproperty
```

In this example a different form of implication is used. The earlier form used (|=>) specified to look for the next element of the sequence at the next clock tick, this form (|->) says to look now — when start is TRUE. The way to read this is that when start is TRUE, the address bus should be not-unknown right now. It's a standard implication form; you could read it as "start implies address bus not-unknown."

The readIs3ticks property tested to see if a read operation was done correctly. That is, after the start signal was seen, it watched for read being asserted at the next tick. If the operation on the bus was a write, then this property would fail because read wouldn't be asserted. Again, when testing a design, you don't want your properties failing even though there is no problem.

One approach to testing this would be to just test for dataValid at the third tick. This is shown here:

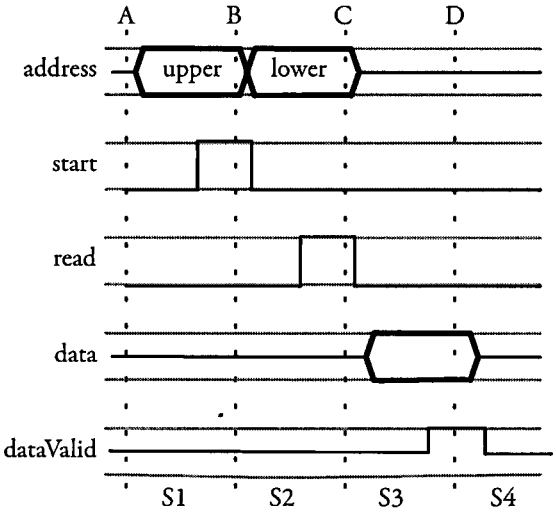

Figure 9.6 — SimpleBus Read Protocol Without Delays

```
1  property ckDataValidOnly;
2    @(negedge ck) start |=> ##1 dataValid;
3  endproperty
```

This sequence says that when start is asserted, then at the next tick begin following the sequence. The sequence says to wait one clock tick and check for dataValid. The property could have been written equivalently as:

```
1  property ckDataValidOnlyAlt;
2    @(negedge ck) start |-> ##2 dataValid;
3  endproperty
```

In this case, the implication (|->) says to start following the sequence in the same tick time as when start is TRUE. In either case, the effect is to wait for 2 clock ticks before checking for dataValid (i.e., 3 ticks in all).

With a slightly more complex firing condition for the implication, the notion of a read could be accommodated. Consider the following property:

```
1  property ckRead;
2    @(negedge ck) (start ##1 read) |=> dataValid;
3  endproperty
```

In this property, the condition for starting is that start is TRUE followed by read at the next clock tick. If that occurs, that means that a read is in progress and dataValid should be TRUE at the next tick.

The complementary check for a write would be:

```
1  property ckWrite;
2    @(negedge ck) (start ##1 ~read) |=> dataValid;
3  endproperty
```

The only difference being that ckWrite watches for read not being asserted as part of the starting condition.

9.3.3 Sequence Operators

Now we return to the full protocol specification (Figure 9.5) where there is an unspecified time delay between the start signal and the dataValid. We'll define this to be as small as 2 ticks and as large as 7 ticks for a write, and as small as 2 and as large as 10 ticks for a read; all these delays are from start. Thus the shortest time is the time checked for above where dataValid comes right after read in a read cycle (2 ticks from start).

To specify a range of tick counts in a sequence, the construct ##[n:m] is used; n is the minimum and m is the maximum. Considering only the read cycle, the property would be:

```
1  property ckReadRange;
2    @(negedge ck) (start ##1 read) |-> read ##[1:9] dataValid;
3  endproperty
```

This sequence says that when we see start at the current tick, we should see read at the next tick. This is the antecedent that starts the implication. From there, read should be TRUE in the current time and dataValid between one and nine ticks in the future.

The constants n and m can be any positive integer (including 0). m can also be the symbol "$" meaning *unbounded but finite;* sometimes read as *eventually.* $ means that the range should be satisfied by the end of simulation, no matter how far in the future that might be. If it isn't, then the assertion fails as the simulation ends.

But the design situation here is that there are really two operations that can occur and, in this case, have different possible delay times. To capture this situation, an *or* connective can be used in the sequence to specify either of the read or write conditions as shown in the following property:

```
1  property ckReadOrWrite;
2    @(negedge ck) start |=> ((read ##[1:9] dataValid) or
3                             (~read ##[1:6] dataValid));
4  endproperty
```

In this case, two sequences are specified and separated by the or operator. The first sequence to succeed causes the whole property to succeed. If neither succeeds, then the property fails.

Another aspect of the protocol that could be checked regards the start signal. The start signal is important in that it tells all the interfaces on the bus that a bus transaction (read or write) is starting. In this protocol, there can only be one transaction at a time. To check for this we need to make sure that once a transaction starts (as signaled by start), start should be FALSE until the clock tick after dataValid is TRUE. This requires checking the start signal at every clock tick. But this can be specified more efficiently with the *throughout* operator.

```
1  property ckOnlyOneStart;
2    @(negedge ck) start |=> ~start throughout
3                  ((read ##[1:9] dataValid) or
4                  (~read ##[1:6] dataValid));
5  endproperty
```

The *throughout* operator has an expression on the left that must remain TRUE throughout the sequence specified to its right. This sequence states that when start is TRUE, then at the next clock tick start will be FALSE (~start is TRUE) and it will remain FALSE until dataValid is seen in either of the read or write cycle conditions. In this way, it checks for start being TRUE for only one clock tick, and not again until the tick after dataValid is TRUE; the ending condition here is dataValid AND ~start. It could have been written more simply as:

```
1  property ckOnlyOneStartAlt;
2    @(negedge ck) start |=> ~start throughout ##[1:9] (dataValid & ~start);
3  endproperty
```

This approach only looks for dataValid regardless of whether it's a read or write. Thus, it won't detect if a write transaction takes longer than 7 clock ticks; it's just checking the start signal's value over the longest bus transaction.

Besides the sequence operators throughout and or, there are several others. Here is a full list:

- *or* — two sequences begin at the same clock tick. As soon as one succeeds, the assertion succeeds. If neither succeed, then the assertion fails.
- *and* — two sequences begin at the same clock tick. Both sequences must succeed although their ending times may be different.
- *intersect* — like and (above) but with the further restriction that both sequences end at the same time.
- *throughout* — the expression on the left is to remain TRUE throughout the sequence on its right.
- *within* — specified as "s1 within s2". s1 must start no earlier than the start of s2 and must end no later than the end of s2.
- *first_match* — whenever sequences are specified with the above operators and also have range repetitions, there can be several matches at different times. This reports the first and the others are discarded.

9.3.4 Signal/Expression Repetitions in Sequences

In some protocols there is a signal or a sequence of signals that can repeat.

Consider the ckOnlyOneStartAlt property (above) which states that once the sequence is started, start is FALSE until the tick after dataValid. This was specified using a range on the ## delay operator. Another approach would have been to specify it as a range of iterations of the start signal:

```
1  property ckOnlyOneStartRepetitions;
2    @(negedge ck) start |=> ~start [*1:8] ##1 (dataValid & ~start);
3  endproperty
```

This sequence states that one tick after start is seen, there will be between 1 and 8 consecutive ~starts followed by a dataValid AND a ~start at the same tick. There are several things to note here:

- The consecutive range repetition operator, written here as [*1:8], specifies that the element to its left, which is ~start, will repeat in consecutive clock ticks between 1 and 8 times (inclusive). The construct is left associative.
- In the last tick, the one that breaks the repetition loop of watching ~start, dataValid and ~start will both be TRUE. This assures that when dataValid occurs that ~start will still be TRUE. This is because there can't be another start until one tick after dataValid. If the ~start condition was left out of the dataValid expression, a new start could have occurred while dataValid was asserted — that's incorrect in this protocol.

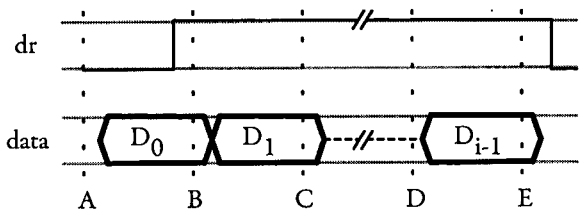

Figure 9.7 — dr Indicating When a Block of i Data Bytes Are Being Sent

Consider another protocol specification, this one shown in Figure 9.7. In this protocol, a series of data bytes is placed on the data lines by a sender at consecutive clock ticks. A data ready (dr) signal is asserted by the sender whenever there is data on the line. As long as dr is asserted, the re-

ceiver loads the data. There can be as few as 2 and as many as 73 data bytes in a row. An assertion to check the count of data bytes is shown below:

```
1  property ckDataByteCountConsec;
2    @(negedge ck) dr |=> dr [*1:72] ##1 ~dr;
3  endproperty
```

This sequence is started by the dr becoming TRUE as happens at clock tick B. After that there will be as few as 1 and as many as 72 more consecutive repetitions of dr (i.e., at successive clock ticks) before ~dr occurs. If you'd rather see the specification numbers (2 and 73) in the property, it could be written:

```
1  property ckDataByteCountConsecAlt;
2    @(negedge ck) dr |-> dr [*2:73] ##1 ~dr;
3  endproperty
```

Here, the count of consecutive repetitions starts in the same state as when dr is first seen because of the |-> implication operator.

Let's change this protocol specification to that shown in Figure 9.8. This protocol is similar to above except that the data bytes don't need to be sent consecutively. The dr signal indicates that there is a byte on the data lines. The sender tells the receiver the sequence of data bytes is done when the done signal is asserted. As shown here, it is asserted in the state following the last dr, when there is no data on the line. This corresponds to clock tick F in Figure 9.8. Again, there can be as few as 2 and as many as 73 non-consecutive data bytes sent between the first dr and the done signal.

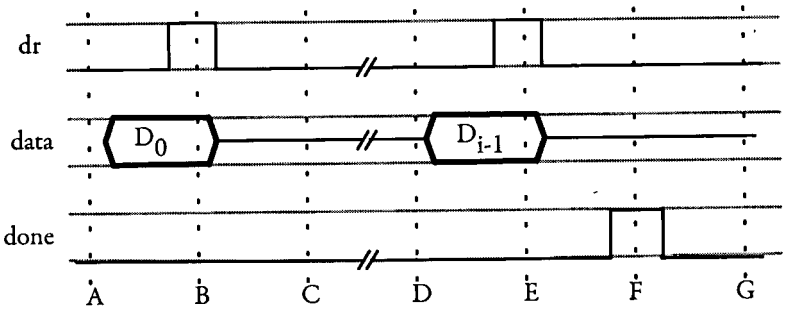

Figure 9.8 — Timing Diagram Showing the "go to" Relationship of the Final dr and done

A property to check non-consecutive iteration such as this is:

```
1  property ckDataByteCountNonConsec;
2    @(negedge ck) dr |-> dr [->2:73] ##1 done;
3  endproperty
```

The difference here is the use of the non-consecutive *goto* range repetition operator [->2:73]. It specifies two things. First, dr can be asserted between 2 and 73 times but they don't have to be at consecutive clock ticks. Thus there could be many clock ticks in between the dr signals. Second, it specifies that the done signal ending the sequence must be one clock tick after the final dr as shown in Figure 9.8.

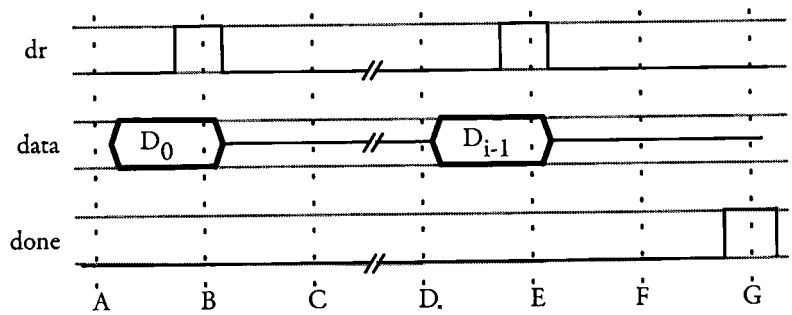

Figure 9.9 — Timing Diagram Showing a Match for the Non-Consecutive (=) Repetition Operator

There is an alternate ending to the non-consecutive repetition where the done is not required to be right after the final repetition. The sequence specification for this is shown here:

```
1  property ckDataByteCountNonConsec;
2    @(negedge ck)
3      dr |-> dr [=2:73] ##1 done;
4  endproperty
```

The timing diagram is shown in Figure 9.9 where, in this case, done is two ticks after the final dr. This still matches the sequence despite the final "##1 done" because you can think of the repetition as ending on clock tick F. There is no restriction on how long after the last tick on which dr was TRUE that done can appear. Indeed, Figure 9.8 would also match this sequence specification.

Summarizing, there are three ways to specify repetitions and ranges of repetitions of expressions:

- *Consecutive repetitions* — The expression being repeated must continuously repeat for the specified number of times. The construct "b [*8]" represents eight repetitions of b separated by ##1. The construct "b[*3:7]" represents between 3 and 7 repetitions of b separated by ##1. When a range is specified, the repetitions end when the first element of the sequence following it becomes TRUE.
- *Goto non-consecutive repetitions* — The expression being repeated can be intermittently (non-consecutively) TRUE. The construct "b[->8] ##1 done" specifies that b will be TRUE at eight possibly non-consecutive ticks and done will be TRUE strictly at the first clock tick following the last b. See Figure 9.8.
- *Non-consecutive repetitions* — This is like the goto operator except for the relationship of the last repetition in the sequence and the next part of the sequence. "b[=8] ##1 done" removes the restriction that the done must follow the last b in the next clock tick. See Figure 9.9.

The goto operator (->) is the more restrictive of the two non-consecutive types. A goto relation will only match Figure 9.8. The non-consecutive repetitions operator (=) will match both Figure 9.8 and Figure 9.9.

9.4 Calculations within Sequences

Consider the design situation where you are checking whether a sequence of data bytes is being sent correctly. For instance, you might want to insure that the sender is sending correct information on the bus. Figure 9.10 shows a protocol where this might occur.

In this protocol, data bytes are sent at consecutive clock ticks. The first data byte is on the data lines when the start signal is asserted. The last data byte, which is called the checksum is on the data lines when the done signal is asserted.

The idea behind the checksum is that it can be used to determine if the data is being received correctly. That is, the sender sends the data bytes and internally adds them up modulo 256; this sum is called the checksum. After sending the individual data bytes, the negative of the checksum byte is sent. The receiver also adds up the data bytes as they are being received. The way this works is that, in this example, the sender sends the negative

of the checksum. When it gets added into the receiver's checksum, the result should be 0. If it's not, some error in data transmission occurred.

An assertion can be written to watch the bus lines to see if the sender is correctly sending its data and checksum. To handle this design situation, sequences can be written to include local variables on which calculations can be performed as the sequence progresses. If the sequence ends up having several assertion threads active, each thread has its own copies of these local variables.

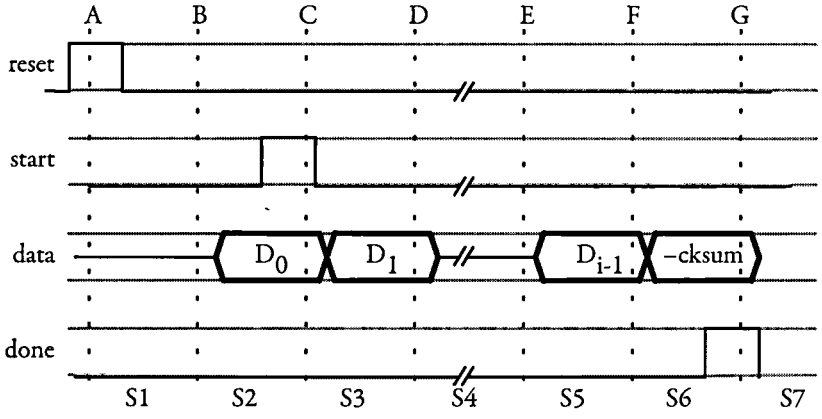

Figure 9.10 — Timing Diagram for Checksum Calculator

Consider the following sequence to calculate and check the checksum sent:

```
1  property ckCheckSumCorrect;
2    byte   cSum;
3    @(negedge ck) disable iff (reset)
4      (start, cSum = data) |=> (~done, cSum += data) [*0:19]
5                               ##1 (done && ((cSum + data) == 0));
6  endproperty
```

There are several new features in this property. First, just about all systems include a reset signal that puts a system into a specific state. In terms of using assertion in such situations, an assertion thread should not be started during reset, and assertion threads that are already in-flight should be cancelled. Line 3, with the inclusion of "disable iff (reset)," shows how to specify this. Essentially this is saying that when reset is asserted, all assertion threads with this specification are ended.

Secondly, a local variable is declared as shown on line 2.

The last new construct here couples an expression with an assignment statement separated by a comma, such as on line 4: (start, cSum = data). The first element (start) is the condition that is being checked to start the sequence. If it's TRUE, and thus the sequence is starting, then the current value of data is loaded from the bus (called data in the timing diagram) into local variable cSum. Once the sequence starts, there are between 0 and 19 consecutive states when ~done is TRUE. At each of these, data from the bus is added to cSum, calculating the assertion's checksum. done is asserted by the sender when it puts the negative of the checksum on the lines (clock tick G in Figure 9.10). When the asser-

tion runs as a result of clock G, the value being sent on the data bus is the negative of the checksum as calculated by the sender, and the value cSum is the checksum calculated by the assertion. The sequence (line 5) checks to see that when done is TRUE, that the sum of the sender's and assertion's checksums should be 0.

As a separate example, consider checking a pipelined multiply unit. The unit reads its two inputs at each active clock edge and a produces a result based solely on those two values three clocks later. The issue is how to write an assertion to check the result. The functional unit and its property would be:

```
1   module pipeMult (
2     input bit [9:0]  inA, inB,
3     input bit        ck, reset,
4     output bit [9:0] result);
5
6     bit [9:0] stage1out, stage2out;
7
8     always_ff @(posedge ck) begin
9       stage1out <= inA *  inB;
10      stage2out <= stage1out;
11      result <= stage2out;
12    end
13
14    property pipelinedFunction(inA, inB);
15      bit  [9:0]  A, B;
16      @(posedge ck) disable iff (reset)
17        (1, A = inA, B = inB) ##3 (result == A * B);
18    endproperty
19  ...
20  endmodule: pipeMult
```

The 3-stage pipelined multiplier's functionality is shown on lines 8-12 above. It's modeled as the multiply taking place during the first stage while the two inputs are still valid. (line 9). Lines 10-11 model this value being passed down the stages until it's finally on the output (result).

This property's sequence to check this uses local variables (A and B, on line 15) to remember the inputs when the assertion thread first starts. The sequence, specified on line 17, always starts because the 1 is a logic TRUE. The sequence then waits three clock ticks and compares the result output of module pipeMult with this assertion's internally calculated product. The sequence starts up a new assertion thread at every positive clock edge and thus there will be three threads in flight in steady state. Note that there will be a separate copy of local variables for each of the assertion threads.

9.5 Sampled Value Functions

Sampled value functions provide direct access to the values sampled during the preponed region of the simulation kernel.

9.5.1 Introduction

As an initial example, the function $sampled(expression) returns the value of the expression using values for the expression's variable(s) from the preponed region. This function can be used anywhere but is not allowed in synthesizable models. The following initial block illustrates how sampled value functions work.

```
1   initial begin
2     b = 0;
3     @(posedge ck);
4     b = 1;
5     $display ("%b %b", b, $sampled(b));
6   end
```

The block assumes that there is a clock that is oscillating; this clock is waited for at line 3. When the initial block is executed, the $display statement will print "1 0" because the current value of b was set on the line just previous to the $display — it prints as 1. However, since the sampled value comes from the preponed region of the kernel, the value of b is 0 because it was sampled just before the clock edge occurred.

Of course the $sampled() function isn't very interesting by itself. The whole list of sampled value functions is shown below. ce stands for clock event (i.e., something like "@(posedge ck)"):

- ***$sampled (expression)*** — The value of the expression using values sampled in the current time's preponed region.
- ***$rose (expression, ce)*** — TRUE if the least significant bit of the expression changed to 1 from the previous clock event to the current clock event. FALSE otherwise.
- ***$fell (expression, ce)*** — TRUE if the least significant bit of the expression changed to 0 from the previous clock event to the current clock event. FALSE otherwise.
- ***$stable (expression, ce)*** — TRUE if the value of the expression remained the same from the previous clock event to the current clock event. FALSE otherwise.
- ***$changed (expression, ce)*** — TRUE if the value of the expression changed from the previous clock event to the current clock event. FALSE otherwise.
- ***$past (expression, numTicks, ce)*** — The value of the expression using values sampled numTicks in the past, not counting the samples taken in the current clock time.

The clocking event (ce) specified in some of the functions is an optional specification. If these functions are used as part of evaluating an assertion, the property's clock is inferred and ce need not be specified. If these are used in other procedural blocks, such as a testbench, the clocking event needs to be specified using a clocking block as shown in Example 9.3.

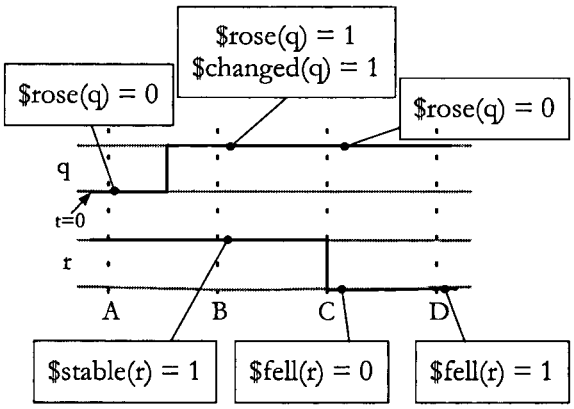

Figure 9.11 — Sampled Value Functions

Figure 9.11 shows two waveforms, q and r. It also shows several instances of the sampled value functions and their values at specific points along the waveform. Most are self explanatory. Note that $fell(r) at clock tick C is 0 because r changed to 0 right at the clock tick; its preponed value is 1 and thus $fell won't recognize the change. However, $fell(r) is TRUE at the next clock tick.

Also note that $rose(q) is 0 at clock tick A. This is a result of q being defined to be a bit variable in the simulation for this figure. If q had been defined as a logic variable, $rose(q) would be 1'bx at this point.

9.5.2 Sequences Using Sampled Value Functions

Consider first the pipelinedFunction property from Section 9.4. The example was of a pipeline multiplier that read its inputs at every clock event and produced its result 3 clock ticks later. Local variables were used to sample and remember the input values for later comparison.

Another approach to writing the property uses sampled value functions:

```
1  property pipelinedFunctionPast(inA, inB);
2    @(posedge ck) disable iff (reset)
3      result == ($past(inA, 3) * $past(inB, 3));
4  endproperty
```

Here the values of inA and inB are retrieved by the $past function looking back 3 clock ticks. The product of those two values are compared with the result generated by the pipeMult module in the current time.

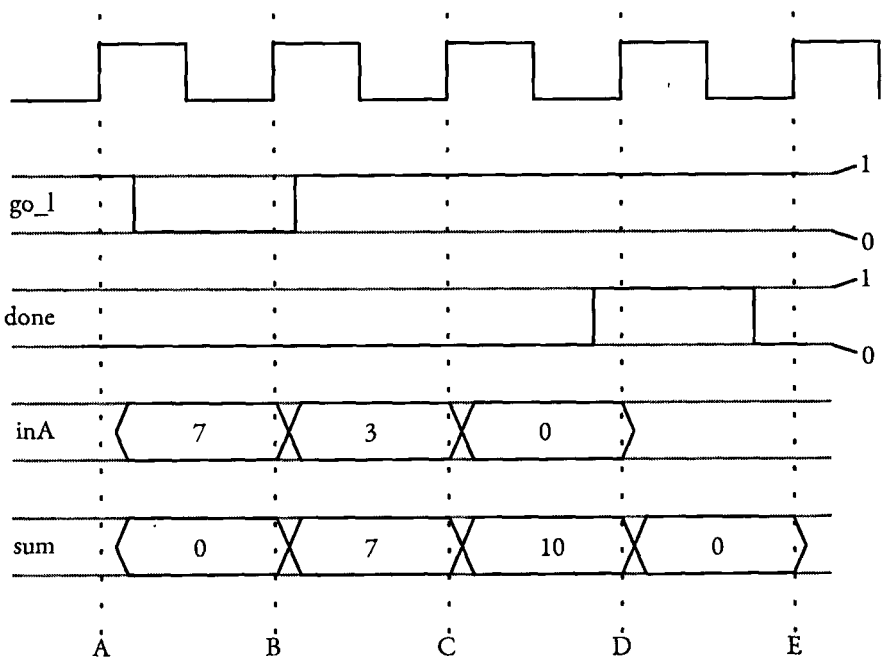

Figure 9.12 — Timing Diagram For sumItUp Thread

Since the result starts simulation with value 0, and $past returns 0 for times before the start of simulation, the first assertions pass.

For another example we return to the sumItup thread of Chapter 5.1. The protocol timing diagram of the thread is repeated here in Figure 9.12. Based on this, the following properties can be written.

```
1   property goAssertedForOneClock;
2     @(posedge ck) disable iff (~reset_l)
3       $fell(go_l) |=> $rose(go_l);
4   endproperty
```

This sequence starts when go_l has fallen. In the timing diagram, this would be detected as a result of clock edge B because preponed values of go_l are 1 at clock edge A and 0 at clock edge B. Then, at clock edge C (one clock edge later as specified by the |=> implication) we should see go_l rise to 1 again.

This will work, however, the following assertion will also check the same condition:

```
1   property goAssertedForOneClockAlt;
2     @(posedge ck) disable iff (~reset_l)
3       ~go_l |=> go_l;
4   endproperty
```

That is, if go_l is asserted now, it will be not-asserted in the next state. These sequences are equivalent because you are only checking a pulse that is one clock period wide.

The $rise, $fall, and $changed functions are useful when you want to time a sequence from when an expression originally changes; you want the sequence to be edge-triggered. Consider a signal Q that, when asserted, is only asserted for four consecutive ticks; it becomes not-asserted after that. This would be captured by the following assertion:

```
1   property fourQ;
2     @(posedge ck) disable iff (~reset_l)
3       $rose(Q) |=> Q[*3] ##1 ~Q;
4   endproperty
```

In this case, the sequence only starts when Q rises. Then starting at the next tick, Q should remain asserted for three more consecutive ticks and then it should become not-asserted. This would not work with the following replacement for line 3:

```
1       Q |=> Q[*3] ##1 ~Q;
```

This is because this sequence would start anytime that Q is TRUE. Given that it would also start on the second through fourth Qs in the sequence, there wouldn't be three more Qs following the first. Thus these sequences would fail even though there is nothing wrong with the system. Not cool.

It is important to know that done is asserted only if inA is 0. i.e., that the output signalling is working. This would be detected at clock D with the following assertion:

```
1   property doneImpliesINAis0;
2     @(posedge ck) disable iff (~reset_l)
3       done |-> (inA == 0);
4   endproperty
```

Note that the assertion wouldn't be written the other way around. That is, inA==0 does not imply that done is 1. For instance, it's possible that inA is always 0 unless specific values are being sent to be summed. Thus if there were several states between when values were being sent, inA would be 0 but done should not be asserted. Remember that done is a signalling variable indicating to the thread connected on the output that it can load the output value. This is also captured in the next-state and output logic of the FSM. This assertion will be checking all of that logic.

Consider the situation now where done is asserted which should mean that sum becomes 0. done is asserted at clock D and the value of sum will be visible as a result of clock E.

```
1  property doneImpliesSUMbecomes0;
2    @(posedge ck) disable iff (~reset_l)
3      done |=> (sum == 0);
4  endproperty
```

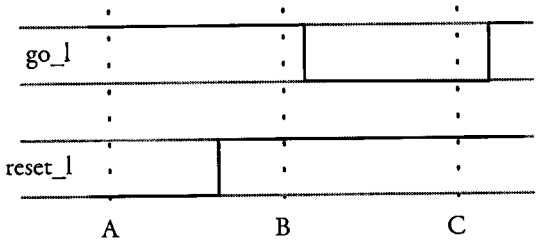

Figure 9.13 — reset_l Timing Relationship

Contrasting the last two properties, the first deals with two conditions that exist at the same time in the datapath. The second deals with two conditions that exist one clock tick apart. This can be seen in the timing diagram where the first is only active at clock D and the second spans clocks D and E.

Actions timed from reset are often specified using the deassertion (trailing) edge of the reset signal as the starting point. Figure 9.13 shows the timing of the reset_l signal with respect to the earliest timing of the go_l signal of the sumItUp thread; the go_l signal should not be asserted during a state where reset was active. The relationship of reset and go_l can be checked with the following property:

```
1  property goFollowsReset;
2    @(posedge ck)
3      $rose(reset_l) |-> (go_l == 1);
4  endproperty
```

In this case, we don't want the sumItUp thread to start during a state where reset was active. This sequence states that when we see the trailing edge of reset_l, then at clock tick B, go_l should be not-asserted (1).

Remember from the sumItUp thread implementation that reset_l is directly connected to the asynchronous reset of the sum register. The relationship of reset and sum can be checked with the following property:

```
1  property sumGetsReset;
2    @(posedge ck)
3      $rose(reset_l) |-> (sum == 0);
4  endproperty
```

In this case we check for the trailing edge of reset which is observed at clock tick B. At that point, sum should be 0 as a result of its asynchronous reset. Note that since the actions of

the reset signal are part of these sequences, the sequences must run during reset and thus "disable iff (~reset_l)" is not used.

9.6 Exercise Problems – Assertions

9.1 **Using Problem** 5.1 as a problem definition, write an assertion to check that each sequence of five bits has two and only two 1-bits.

9.2 **Using Problem** 5.2 as a problem definition, write the following assertions:

A) Assert that maxValue is 0 as a result of reset (rst).

B) Assert that after reset start precedes done.

C) Assert that after done is asserted, maxValue is 0.

D) Assert that maxValue remains 0 after reset or done until the first non-zero value is on the inputs and the input sequence has started.

E) Assert that done is not asserted while start is asserted.

F) Assert that done is asserted in the state following start becoming not-asserted.

G) Assert that maxValue is correct based on the inputs that the assertion statement sees.

9.3 **Using Problem** 5.3 as a problem definition, write the following assertions.

A) Assert that done and error are never asserted together.

B) Assert that error is asserted at the correct time.

C) Assert that error stays asserted until the next start.

9.4 **Using Problem** 5.4 as a problem definition, write the following assertions.

A) Assert that c.Free is the first signal to be asserted after reset (i.e., p.put isn't first).

B) Assert that the c.Free and p.put operate in the correct order.