The following information is provided for discussion at the 6/19/03 STIL.1 working group meeting.

18. Pattern Data

Extensions to IEEE Std. 1450-1999 clause 21

This clause defines additional formatting capabilities for defining pattern data. All statements as defined in IEEE Std. 1450-1999 clause 21 are unchanged.

Note: The following sub-clause replaces clause 6.8 - "Logical expressions with signal and group read back"

18.1 Data content of signals, variables, and constants

This clause defines additional syntax for specifying and manipulating pattern data. The additional constructs are of the form:

sigref_expr = \readback_function SIGNAL_NAME;

sigref_expr = \readback_function SIGNAL_GROUP_NAME;

sigref_expr = \readback_function SIGNAL_VARIABLE_NAME;

sigref_expr = \readback_function WFCCONSTANT_NAME;

If '\readback_function SIGNAL_NAME == *vec_data*' {}

If '\readback_function SIGNAL_NAME == *event*' {}

If '\readback_function SIGNAL_GROUP_NAME == *vec_data*' {}

If '\readback_function SIGNAL_GROUP_NAME == *event*' {}

If '\readback_function SIGNAL_VARIABLE_NAME == *vec_data*' {}

Read back functions are used to determine the current value that is defined for a signal, signal group, signal variable, or wfc constant. These read back functions are identified by the backslash escape character and a function character in front of the name. A space between the escape sequence and the name is optional.

Whenever a read back function is used on the right side of a pattern data expression, the backslash escape sequence must be used - i.e., there is no default.

In the case of the use of a read back function on the left side of a compare (i.e. 'A==B'), the backslash escape may be omitted, in which case the wfc (i.e., \W operation) shall be used.

The allowed values for *readback_function* are:

\C SIGNAL-OR-GROUP-NAME - return the last compare event/event-list that was set for the named signal or group. If no compare event has been established, then return ' ' (the space char).

\D SIGNAL-OR-GROUP-NAME - return the last drive event/event-list that was set for the named signal or group. If drive no event has been established, then return ' ' (the space char).

\E SIGNAL-OR-GROUP-NAME - return the last expect event/event-list that was set for the named signal or group. If no expect events have been established, then return ' ' (the space char).

\S SIGNAL-OR-GROUP-NAME - return the last wfc/wfc-list that was established for the named signal or group using a substitute (s or S) event in a waveform. The actual wfc is determined by the WFCMap statement. If no substitey has been established, then return ' ' (the space char).

\U SIGNAL-OR-GROUP-NAME - return the last undefined event/event-list that was set for the named signal or group. If no undefined events have been established, then return ' ' (the space char).

\W SIGNAL-OR-GROUP-OR-SIGNALVARIABLE-OR-WFCCONSTANT-NAME - return the last wfc/wfc-list that was established for the named signal or group using a V or a C statement. If the last wfc was established by a parameter using # or %, then the substituted wfc is returned. If no wfc has been established, then return ' ' (the space char). Note: this should never happen since it is required that an initial wfc be defined on the first vector of any pattern.

```
STIL 1.0 { Design D15; }
Header {
   Source "P1450.1 Working-Draft 16, May 8, 2003";
   Ann {* clause 18.1 *}
}

Signals {
   SIG Out { WFCMap S->LH; }
   SIGS[1..4] In;
}
SignalGroups {
  GRP = 'SIGS[1..4]';
}
Variables {
  SIGVAR[1..4];
}
Timing {
  WaveformTable WFT {
    SIG { LHS { '25ns' L/H/S; }
    GRP { LHS { '25ns' L/H/S; }
}}

Pattern PAT {
// Examples of  read back functions
  W WFT;
  V { SIG = L; }
  If 'SIG == L' {} // true since last wfc for this signal was L
  V { SIG = S; }
  If '\S SIG == H' {} // true if value read by the prior S event mapped to wfc H
  If '\D SIG == DriveUp' {} // true if the last drive event was drive-up
  If '\C SIG == CompareHigh' {} // true if the last compare event was compare high
  V { GRP = HHHH; }
  If 'GRP == HHHH; } // true since last wfc for this group was HHHH
  V { GRP = SSSS; }
  If '\S GRP == HLHL' {} // true if the last group substite mapped to HLHL
  If '\D GRP == DriveUp' {} // true if last drive event for all signals in GRP was 'DriveUp'

 // Examples of read back into signal variables using read back functions
  C { SIGVAR = \W GRP; }
  If 'SIGVAR == HHHH' {}
  C { SIGVAR[1] = \W SIG; }
  If 'SIGVAR[1] == H' {}
}
```

13. PatternBurst Block

Extensions to IEEE Std. 1450-1999 Clause 17

This clause defines additional statements supported within the PatternBurst block. All statements and capabilities as defined in IEEE Std. 1450-1999 Clause 17 are unchanged.

Two new pattern grouping structures are defined - the ParallelPatList and the PatSet. Also, the Fixed, Extend, and Wait statements are defined to allow the specification of how multiple patterns are to be executed.

Lastly, the If and While statements are provided to allow conditional execution of patterns within a burst.

13.1 PatternBurst Syntax

(1) **ParallelPatList** (**SyncStart** | **Independent** | **LockStep**): This block defines a set of PAT_OR_BURST_NAME that are to be executed in parallel. Execution of this set of Patterns is controlled by optional arguments **SyncStart**, **Independent**, or **LockStep**, as well as the optional statements **Wait** and **Extend**. Parallel patterns do not necessarily run synchronously or finish together. If no arguments are specified to ParallelPatList then the default operation of the Patterns is **Independent**. All of the optional statements that are defined in STIL.0 for PatList also are available in a ParallelPatList block.

SyncStart: this keyword, if present, requires that all PAT_OR_BURST_NAME present in the ParallelPatList block shall start executing at the same moment. During execution, pattern behavior may diverge if patterns contain different Vector counts or different periods in the Vectors.

Independent: this keyword, if present, allows each PAT_OR_BURST_NAME present in the ParallelPatList block to start as convenient. This option indicates that the set of patterns executing in parallel have little or no relationship between each other and can be executed independently.

LockStep: This keyword is used to specify parallel testing of cores (sub designs) that require synchronization throughout the pattern execution. A lockstep application may be used for situations where parallel cores have common access constructs that require maintaining the same state on a set of signals for the cores during test (for instance, common wrapper control logic around cores). Lockstep may be used for parallel testing of cores that have serially connected scan chains. Lockstep can also be used to map patterns onto test equipment that has limited timing flexibility that prevents true independant execution. See “LockStep Explanation and Example” on page 3.

Note1: Should the above definition of "LockStep" be enhanced to define the general nature of the following two sub-clauses? Also, should we remove the specific reference to "cores" and make the definition more general?

Note 2: It is recommended that the following two sub-clauses be removed from 1450.1. They too "tool specific" and, as such, should be documented by the producer / consumer of the data. These two sub-clauses were provided by Synopsys. There is an alternate proposal from Paul Reuter which represents thinking of the CTL working group.

13.3 LockStep Explanation and Example

This sub-clause explains in detail the semantics of the LockStep operation.

timing and sequencing - Each pattern in the ParallelPatList shall start executing at the same time, each vector in each pattern shall start executing at the same moment in time, and each vector shall have the same period. When lockstep is defined for a set of parallel patterns, these patterns shall enter and exit all data-dependent pattern constructs in parallel as long as the patterns are executing in parallel. In particular, lockstep patterns shall enter a shift block or loop data block at the same point in the execution sequence, and shall finish executing these blocks at the same time. To support this behavior, the definition of normalizing the length of data for each of these blocks is extended as follows:

-- For all shift blocks in parallel, the length of the scan data is defined as the longest data length of all signals with a "#" across all shift blocks. To facilitate this determination, it is required of the integration process that the signal/group definitions be adjusted such that the ScanIn/ScanOut-integer attribute defines the length to which each signal/group is to be padded, and this value shall be used rather than padding to the longest scan chain length of an individual pattern.

-- For all loop-data blocks in parallel, the length of the loop data is defined as the longest loop required by any of the patterns. All other patterns shall pad additional passes through the loop using the default state.

shift ordering on serial scan signals - When there are serially connected scan signals, the scan data is combined according to the following rules. The data assigned to each scan signal in common in each shift block executed in parallel across all patterns in the parallelPatList, is the concatenation of the scan data for each reference to this signal for each of the shifts executed in parallel. The data is concatenated in the order that the patterns are referenced in the ParallelPatList block. The first pattern in the list is the first to shift in and the first to shift out. Please see the example below.

shift padding on serial scan signals - When there are serially connected scan signals, the scan data padding is done according to the following rules (consistent with the rules as defined in IEEE Std. 1450-1999). Scan shift padding is done by first concatenating each serial scan signal. Then the length of each scan signal is determined by the length of each segment. Then the shorter combined scan signals are padded to the same length as the length of the largest combined scan signal. Please see the example below.

scan chain length determination - For all shift blocks in parallel, the length of the scan data is defined as the longest data length of all signals with a "#" across all shift blocks. To facilitate this determination, it is required of the integration process that the signal/group definitions be adjusted using the ScanIn/ScanOut-integer attribute that defines the length to which each signal/group is to be padded, and this value shall be used rather than padding to the longest scan chain of an individual pattern. (Note that determination of padding for LockStep is slightly different from the calculation of padding for Procedure and Macro Substitution (see STIL.0 Section 24.5 "Procedure and Macro Data substitution") which states that padding is based on the length of the longest chain.)

procedures, macros, and shifts - All patterns in the lock-step group shall have the same procedure and macro activity - i.e., procedure/macro calls shall occur at the same point in each pattern and shall be of the the same length. All procedures/macros in the lock-step group shall have the same shift-block activity - i.e., shift-blocks must occur at the same vector position in the procedure/macro.

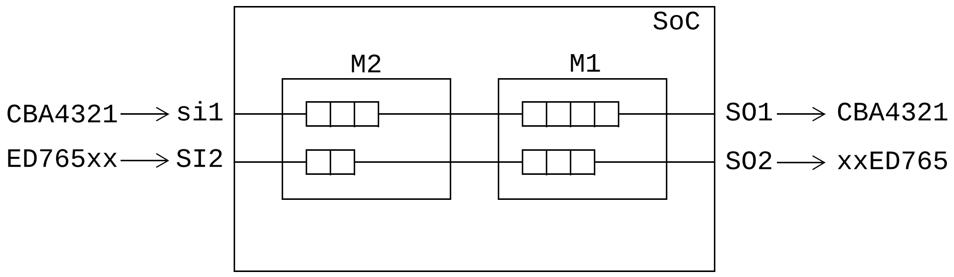
signal overlap - In general, parallel patterns should operate on separate disjoint sets of signals. However, there are cases where the same signal appears in two or more patterns. Such may be the case where two identical cores share a common input sigal. See the following sub-clause for details on this situation.

procedure considerations - In the case where parallel patterns call procedures, it is a requirement that the procedures be called in parallel. The set of signals in all of the called procures shall, in combination, define the activity for all of the signals. In general, it is expected that there be a one to one relation between the calling pattern and the called procedure, but this is not always the case. If there is overlap in the signals, then the signal overlap rules defined above apply. If there are any missing signals, then they follow the normal default to the state defined in the Signals group block, else the initial default state.

setup macro - Each pattern is required to define in the initial V or C statement the set of signals that are used by the pattern. This is typically done in a setup macro. This setup macro should limit the definition to only the signals of the pattern and not the entire signal width of the design. Note: Defining signal states that are used in other parallel patterns will preclude the use of the lock step operation, and indeed, any parallel pattern operation.

The following is an example of two pattern running in LockStep with a serially connected scan chain.

```
STIL 1.0 { Design D15; }
    Header {
    Source "P1450.1 Working-Draft 16, May 8, 2003";
    Ann {* clause 13.3 *}
}
Signals { SI1 In; SI2 In; SO1 Out; SO2 Out; }
SignalGroups G1 {
    SI1=SI1 { ScanIn 4; }
    SI2=SI2 { ScanIn 3; }
    SO1=SO1 { ScanOut 4; }
    SO2=SO2 { ScanOut 3; }
  }
SignalGroups G2 {
    SI1=SI1 { ScanIn 3; }
    SI2=SI2 { ScanIn 2; }
    SO1=SO1 { ScanOut 3; }
    SO2=SO2 { ScanOut 2; }
  }
Procedures PROCS1 {
    LOAD_UNLOAD {
     C { SI1=0; SI2=0; SO1=X; SO2=X; }
     Shift { V { SI1=#; SI2=#; SO1=#; SO2=#; } }
    }
}
Procedures PROCS2 {
    LOAD_UNLOAD {
     C { SI1=0; SI2=0; SO1=X; SO2=X; }
     Shift { V { SI1=#; SI2=#; SO1=#; SO2=#; } }
    }
}
PatternBurst {
    ParallelPatList LockStep {
     P1 { Procedures PROCS1; SignalGroups G1; }
     P2 { Procedures PROCS2; SignalGroups G2; }
    }
}
Pattern P1 {
    Call LOAD_UNLOAD { SI1=1234; SI2=567; SO1=1234; SO2=567; }
}
Pattern P2 {
    Call LOAD_UNLOAD { SI1=ABC; SI2=DE; SO1=ABC; SO2=DE; }
}
```



13.4 LockStep Pattern/Macro Alignment

Intro/Background - LockStepped parallel patterns allow a CERTAIN level of flexibility in pattern execution, however they have problems if a signal is "shared" between multiple parallel contexts (if the signal is the same in all contexts, then there is no problem). But if the signal needs to be controlled (i.e., driven or compated) from one context, and then controlled (i.e., driven or compared) from another context, there are problems with STIL semantics to allow this. The following extensions are defined to support this behavior.

These are the fundamental rules of behavior: A Signal can only be specified once in a Vector. There is no specified resolution if a Signal is driven multiple times, but there is no resolution needed if all references to that Signal specify the same WFC/WFT (redundancy is OK, but if the signal is driven to different values in the Vector there is no specified resolution; "last" doesn't win because I can't tell you what "last" is, especially in parallel contexts (next point).

This is the extended definition of a vector in parallel operation: A Vector is the sum-total of previous states and current assignments to all signals in a STIL file running a common WaveformTable. In a parallel LockStep context, this means that all Vectors in parallel are essentially treated as a single, larger Vector (it does not mean that the implementation needs to be done this way, just that the "view" of the data can be as a single Vector). This concept is critical to allow test environments without parallelism to support LockStep behavior -- and that is the ultimate goal of LockStep constraints.

LockStep Pattern/Macro Alignment: LockStepped parallel patterns enter Procedure and Macro calls at the same time, as stated above. To facilitate the alignment of entering procedure and macro calls, the BreakPoint statement may be applied in parallel lockstepped contexts, to enforce a common exit of procedures and macros across parallel patterns. The BreakPoint statement is placed at the end of a Procedure to allow synchronization across parallel procedures when the Vector count in each Procedure differs. Take for example:

```
Proc A {              Proc B {            Proc C {
  V { sA=1; }           V { sB=1; }         V { sC=1; }          (1)
  V { sA=0; }           V { sB=0; }         V { sC=0; }          (2)
  V { sA=1; }           V { sB=1; }         V { sC=1; }          (3)
  V { sA=0; }           BreakPoint {        BreakPoint {         (4)
  V { sA=1; }             V { sB=0; }         V { sC=0; }        (5)
  BreakPoint {          }                   }
    V {sA=1; }        }                   }
  }
}
```

This generates:

```
V ( sA=1; sB=1; sC=1; } // (1)
V ( sA=0; sB=0; sC=0; } // (2)
V ( sA=1; sB=1; sC=1; } // (3)
V ( sA=0; sB=0; sC=0; } // (4) - (B,C) in Break
V ( sA=1; sB=0; sC=0; } // (5) - (B,C) in Break
```

When BreakPoints are applied in this context, they shall define a single Vector in the BreakPoint block, with the Signal values necessary to maintain the current state of the portion of the design affected by this procedure.

LockStep handling of Shared Signals: Shared signals require no special handling if the signal is driven uniquely or to the same value across multiple parallel contexts AND the contexts do not rely on individual BreakPoint operations to synchronize exit. This is called "cooperative handling" of shared signals. For example:

```
Proc A {                    Proc B {                     Proc C {
  AllowInterLeave sigA;       AllowInterLeave sigA;        AllowInterLeave sigA;
  V { sigA=1; sA=1; }         V { sB=1; }                  V { sC=1; }
  V { sA=0; }                 V { sigA=0; sB=0; }          V { sC=0; }
  V { sA=1; }                 V { sB=1; }                  V { sigA=1; sC=1; }
}                           }                            }
```

This generates the composite Vectors when these three are run in parallel:

```
V ( sigA=1; sA=1; sB=1; sC=1; } // sigA from Proc A
V ( sigA=0; sA=0; sB=0; sC=0; } // sigA from Proc B
V ( sigA=1; sA=1; sB=1; sC=1; } // sigA from Proc C
```

Signals being driven from multiple parallel contexts require special handling if the parallel contexts drive those Signals to different values. The mechanism to enforce "single ownership" of a shared Signal is an interleaving process of the parallel blocks, that relies on specific BreakPoint statements before and after the shared Signal is accessed. Also, shared Signals shall be identified in each context that may assert a value on that shared Signal during the procedure execution.

Consider the situation where two procedures need to take a shared signal to different states:

```
Proc A {                        Proc B {
  AllowInterLeave sigA;           AllowInterLeave sigA;
  V { sigA=1; sA=1; }             V { sigA=0; sB=1; }
}                               }
```

As defined, this is an error if these procedures execute in parallel. However, if the procedures defined a "stable state" AROUND driving a shared signal:

```
Proc A {                         Proc B {
  AllowInterLeave sigA;            AllowInterLeave sigA;
  BreakPoint { V { sA=0; }}        BreakPoint { V { sB=0; }}
  V { sigA=1; sA=1; }              V { sigA=0; sB=1; }
  BreakPoint { V { sA=0; }}        BreakPoint { V { sB=0; }}
}                                }
```

Then these sets of BreakPoints can be applied. Let's assume Proc A gets evaluated first, then the Vector sequence is:

```
V { sigA=1; sA=1; sB=0; } // sB=0 from initial BreakPoint in B
V { sigA=0; sA=0; sB=1; } // sA=0 from second BreakPoint in A
```

If Proc B gets evaluated first, then the Vectors would be:

```
V { sigA=0; sA=0; sB=1; } // sA=0 from initial BreakPoint in A
V { sigA=1; sA=1; sB=0; } // sB=0 from second BreakPoint in B
```

There is no defined order of evaluation across procedures, so a "pre-condition BreakPoint" and a "post-condition BreakPoint" are necessary for each procedure, depending on whether that procedure has executed it's block of statements or not.

The basic construct for procedures with AllowInterLeave signals:

- either it is known that ALL parallel procedures are written to cooperate with each other,

- or each time an InterLeaved signal is assigned in that procedure, the block of vectors necessary to support that operation are enclosed in a "pre-condition" BreakPoint and a "post-condition" BreakPoint. This block of Vectors will be executed with all other procedures executing a BreakPoint condition.

It is manditory that any procedure executed in a parallel context with another procedure that has a AllowInterLeave statement, contain at least one BreakPoint at the end of the procedure to allow other procedures driving shared signals to execute.

If all procedures that execute in parallel are not written to cooperatively handle a shared signal, then the block of Vectors where the shared signals are driven in each procedure must be sandwiched between 2 BreakPoint statements. The first BreakPoint establishes a quiscent state for this procedure BEFORE this set of Vectors executes; the second BreakPoint establishes a quiscent state for this procedure AFTER this set of Vectors executes.