1. Issues list

This is a list of issues or questions that I have with regard to the current state of dot-1 (tony).

1. See attached info on variables and expressions

2. (page 17) Request that WFCMap be moved from Signals/SignalGroups to Timing for the following reasons:
   - if WFCMap is associated with Signals it can never be adjusted according to domain
   - the Timing block is where the wfc's get defined on for each signal for a given pat-exec and hence is the appropriate place to define mapping
   - the Signal/SignalGroups block has no way of knowing if wfc chars are being used that don't actually exist. The Timing block would have this knowledge.

```
Timing name {
   SignalGroups name;
   WCFMap {
      sigref_expr {
         { z->x; 01->x; }
      }
   }
   WaveformTable W { }
}
```

2. New Variable and Expression Definitions

Table 1: Variable and Expression Usage

expr-type	variable types	where defined	where used	syntax examples
time_expr	time integer real @label	Spec SignalGroups Spec Timing	Timing	‘23.0ns/2+16.5e-9-t2’ ‘t1/i + r - t2’
real_expr	time integer real @label	Spec SignalGroups Spec Timing	Pattern	same as a timing expressions except that it may be used in a non-timing context
integer_expr	integer *other?*	SignalGroups	Pattern CTL	PatternCharacteristics -> NumberVectors ‘max * mode’;
sigvar_expr	SignalVariable	PatternGroups	Pattern	V { grp = ‘sv1’; } V { grp = ‘sv2[1..5]’; } V { grp = ‘sv3[5 4 2 3 1]’; }
logical_expr	integer time real SignalVariable Signal SignalGroup	SignalGroups Spec Spec SignalGroups SignalGroups SignalGroups	Pattern	If ‘i == 0’ {} If ‘period > 23.0ns’ {} If ‘value < 3e-6’ {} If ‘sv2[1..5] == 11000’ {} If ‘(s1==H) | (s2==H)’ {} If ‘grp != HHHH’ {}

The following variable types are to be defined in sec 10 - SignalGroups

1. Integer - *According to dot-0 an integer can be used in a time_expr, however there is no way to define a spec variable of type integer. If dot-1 defines an integer variable, can it then be used in a time_expr:*

```
SignalGroups {
   k Integer;
}
Timing -> WaveformTable -> Waveforms -> sigref_expr ->
   01 { ‘2ns+k*(0.5ns)’ U/D; }
```

2. SignalVariable - *New type in dot-1. Was previously name WFC type. Name is changed to reflect that it follows similar rules to Signals and SignalGroups.*

3. Enumeration - *New type in dot-1*

4. Real - *This type is already allowed in the Spec block. Why do we want it to be in a SignalGroup block as well? I suggest that it be removed from dot-1.*

The following expression types will be defined in sec 5.n

1. logical_expr - *new type in dot-1*

2. sigvar_expr - *new type in dot-1*

A sigvar_expr is an ordered list of elements that operates like a sigref_expr but is not associated with any signal names. Its application is to hold signal data and to pass signal data between Patterns and Procedures/Macros. The output function of a sigvar_expr is an ordered string of wfc's. SignalVariable expressions are enclosed in single quotes. A SignalVariable expression may be assigned list of wfc values to a signal-variable. The signal-variable may be used to transfer the wfc values to either another signal-variable or to signal or signal-group.

3. cellref_expr - *new type in dot-1. As defined in dot-1.*

4. integer_expr - *new type in dot-1*
   Is an "integer_expr" the same thing as "logical_expr"? Should we allow reference to "integer_expr" to indicate that the only allowed result is an integer?
   Also, should we have a reference type of "boolean_expr" which is a logical expression that is being used as a boolean?

5. integer_list - *An integer_list is NOT a referenced type in the language. Whenever it is to be used it should be referenced as in the syntax definition as " (INTEGER)+ ". This signifies a set of space separated integer numbers. There should not be any usage of comma separated integers. Do we put this non-definition into the language? I think not - tony. Example usage is:*

```
V { signame[5 4 3 2 1] = 11001; }

BistRegister reg {
   TapPositions 0 2 4 6 8 10 12;
}
```

6. real_expr - *New type in dot-1. "real" variables are already allowed in a spec block. What is not defined is "real_expr". If we want to reference such a term in dot-1 then we need to add a definition of it.*
   A real_expr has the exact syntax as a timing expression. The difference is that this form of reference is used when the real number is not being used to represent a timing value (i.e., in a pattern).

7. sigref_expr, time_expr - *These expression types are already defined in 1450-1999 and need not be elaborated.*

3. Operators and Functions

Two changes have been made to the table currently in dot-1. The unary operators are deleted, and the extra columns showing what operators are used by expression type have been added.

Table 2—Operators and functions allowed in expressions

Op	Definition	time	real	int	logical
min()	minimum value	YES	YES	YES	YES
max()	maximum value	YES	YES	YES	YES
()	parenthesis	YES	YES	YES	YES
table 3, table 4 of IEEE Std. 1450-1999	SI units & prefixes	YES			
/	divide	YES	YES	YES	YES
*	multiply	YES	YES	YES	YES
+	add	YES	YES	YES	YES
-	subtract	YES	YES	YES	YES
%	**modulus**			YES	YES
<	less than (boolean value)	YES	YES	YES	YES
>	greater than (boolean value)	YES	YES	YES	YES
<=	less or equal (boolean value)	YES	YES	YES	YES
>=	greater or equal (boolean value)	YES	YES	YES	YES
!	**negation (boolean value)**			YES	YES
&&	**and (boolean value)**			YES	YES
||	**or (boolean value)**			YES	YES
==	equal (boolean value)				YES
!=	not equal (boolean value)				YES
~	**bit-wise negation**			YES	YES
&	**bit-wise and**			YES	YES
|	**bit-wise inclusive or**			YES	YES
^	**bit-wise exclusive or**			YES	YES
^~, ~^	**bit-wise equivalence**			YES	YES
~~**&**~~	~~**reduction and**~~				

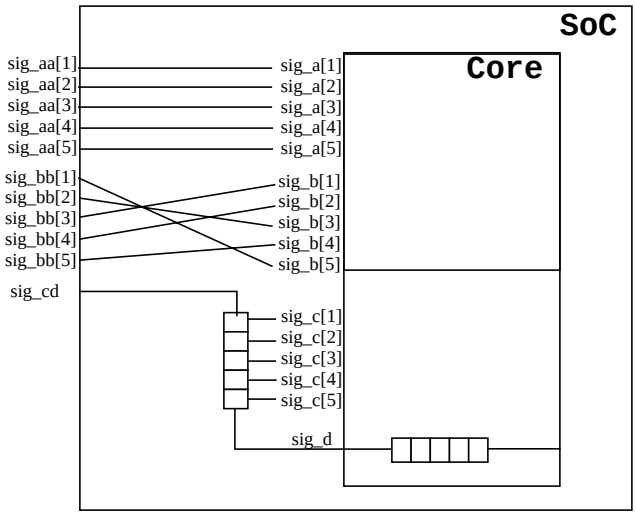
Table 2—Operators and functions allowed in expressions

Op	Definition	time	real	int	logical
~~&~~	~~**reduction nand (not-and)**~~				
~~|~~	~~**reduction or**~~				
~~|~~	~~**reduction nor (not-or)**~~				
~~^~~	~~**reduction xor**~~				
~~^, ^~~	~~**reduction xnor**~~				
~~<<~~	~~**left shift**~~				
~~>>~~	~~**right shift**~~				
?:	conditional expression			YES	YES
=	assignment	YES	YES	YES	YES

4. Signal Mapping Using SignalVariable and Expressions (sigvar_expr)

This example illustrates the following capabilities:

1. definition of SignalVariables in a SignalGroups block
2. use of parameters in a Macro (or Procedure call)
3. use of # to update the parameter values
4. application of parameters in a Macro (or Procedure)
5. use of logic expressions (logic_expr) with signal variables in a Macro (or Procedure)
6. process of re-using signal groups (un-mapped) as signal variables (mapped)



========== pattern ==========

```
Pattern pat_a {
   Macro mac_a {
      sig_a[1..5] = 10101;
      sig_b[1..5] = 11011;
      grp_c = 01010;
      sig_d = 00100;
   }
}
```

========== un-mapped ==========

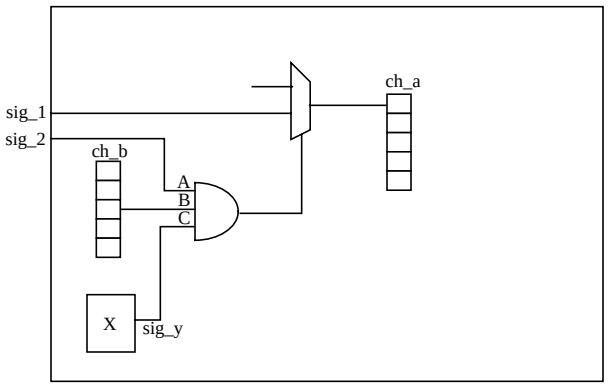
```
Signals {
   sig_a[1..5] In;
   sig_b[1..5] In;
   sig_c[1..5] In;
   sig_d In { Length 5; ScanIn; }
}

SignalGroups {
   grp_c = 'sig_c[1..5]';
}

MacroDefs {
   mac_a {
      V { sig_a[1..5] = #; }
```

```
      V { sig_b[1..5] = #; }
      V { grp_c = #; }
      Shift {
         V { sig_d = #; }
      }
   }
}

===================== mapped =====================
Signals {
   sig_aa[1..5] In;
   sig_bb[1..5] In;
   sig_cd In { Length 10; ScanIn; }
}

SignalGroups {
   sig_a[1..5}    SignalVariable;
   sig_b[1..5]    SignalVariable;
   grp_c          SignalVariable { Length 5; }
   grp_c[1..5]    SignalVariable;
   sig_d          SignalVariable { Length 5; }
}

MacroDefs {
   mac_a {
      C { sig_a[1..5] = #; sig_b[1..5] = #; grp_c = #; sig_d = #;}
      C { grp_c[5 4 3 2 1] = ‘grp_c’; }
      V { sig_aa[1..5] = 'sig_a[1..5]'; }
      V { sig_bb[1..5] = 'sig_b[5 3 1 2 4]'; }
//    V { sig_bb[3 4 2 5 1] = 'sig_a[]'; } // alternate stmt
      Shift {
         V { sig_cd = 'grp_d[1..5]' 'grp_c[1..5]' ; }
//       V { sig_cd = 'grp_d[1..5] + grp_c[1..5]' ; } // alternate?
//       V { sig_cd = 'grp_d[1..5]' 00000 ; } // alternate
//       V { sig_cd = 00000 'grp_c[1..5]' ; } // alternate
      }
   }
}
```

5. Using Logic Expressions (logic_expr) in CTL

This example illustrates the following capabilities:

1. logic expression using & (and) operation
2. logic expression made up of signals, scan signals, and core signals
3. application in CTL of logic expression to define enable logic

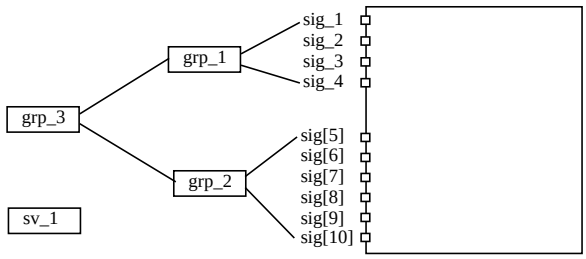



```
Signals {
   sig_1 In;
   sig_2 In;
}

Environment {
   CTL {
      Internal {
         sig_1 {
            IsConnected In {
               StateElement Scan cell_a1;
               IsEnabledBy Logic 'A & B & C' {
                  Signal A sig_2;
                  ScanSignal B cell_b3;
                  CoreSignal C core_x.sig_y;
               }
            }
         }
      }
   }
```

6. Using Boolean Expressions (boolean_expr) in Patterns, Macros, and Procedures

This example illustrates the following capabilities:

1. use of parameters on a Macro (or Procedure)
2. use of logic expression in conditional-If/Else statements in a pattern
3. use of parameters in logic expression to create wfc data in a vector




```
Signals {
   sig_1 In; sig_2 In; sig_3 In; sig_4 In;
   sig[5..10] In;
}
SignalGroups {
   grp_1 = ‘sig_1 + sig_2 + sig_3 + sig_4’;
   grp_2 = ‘sig[5..10]’;
   grp_3 = ‘grp_1 + grp_2’;
   sv_1 = SignalVariable { Length 4; }
}

Pattern pat_1 {
   Macro mac_1 { sig_1 = 1; sig_2 = 0; }
   Macro mac_2 { grp_1 = 1100; sv_1 = 0011; }
}

MacroDefs {
   mac_1 {
      C { sig_1 = #; sig_2 = #; }
      If ‘sig_1 == 1’ { V { sig_1 = A; }}
      If ‘(sig_1 == 1) & (sig_2 == 0)’ { V { sig_1 = B; }}
   }
   mac_2 {
      C ( grp_1 = #; sv_1 = #; }
      If ‘grp_1 == 1100’ {
         V { grp_1 = ‘sv_1’; grp_2 = 111111; }
      }
      Else {
         V { grp_3 =  ‘sv_1’ 000000;
      }
   }
}
```