Daisy Chaining Scan-Chains in CTL

Paul Reuter (Task Force Leader)

**LockStep:** This attribute on a ParallelPat construct defines patterns within the statement to be synchronized on a cycle by cycle basis. The synchronization is achieved with patterns calling a common protocol (Macros or Procedures). This can only be done through any indirection of protocols. That is the patterns under LockStep invoke different protocols which are limited to only invoking the common protocol. The patterns that are LockStep’ed are required to have the same number of invocations to Protocols. The set of patterns that are LockStep’ed are only allowed to be uniform with respect to their use of Macros and Procedures. That is, if Macros are being LockStep’ed there can be no Procedures in the entire ParallelPat LockStep construct.

The intent of this document is to create an example that takes two existing cores with one scan chain each and integrate them at the chip level by creating a single scan chain. The resulting chip could be a core itself and hence the CTL patterns at the chip level is of special interest.

The two cores being integrated are Core C1 and Core C2 that come with their own CTL. The final result is Chip12, and the integration task creates the new CTL for Chip12.

The non-integrated Core 1 has a scan chain with scanin si1 and scanout so1 that is controlled by the clk1 clock and the scan enable se1. Core C1 also has some broadside inputs abc. The non-integrated Core C2 has a similar configuration as Core C1 and the naming convention should be self explanatory.

Core C1 comes with CTL to define the operation of the scan chain and the vectors that use the scan chain. This example ignores all other aspects of the CTL information that would be appropriate for the system integration task. For this example, core C1 has a Macro that operates the scan chain and some patterns that use the macro. Similarly core C2 has a Macro that operates the scan chain and some patterns that use the macro.
Similarly, Core C2 has a macro that operates the scan chain and patterns that use the macro.
At the chip level we would like to have a situation where a new joint macro is written to operate the new larger scan chain and reuses the patterns that are in the files that come with the cores. The following is the outcome of the system integration task.

Signals {
    A In; B In; C In;
    X In; Y In; Z In;
    SI12 In; SO12 Out;
    SE1 In; SE2 In; clk1 In; clk2 In;
}

WBoth {
    Inh W1;
    Inh W2;
}

Variables {
    SignalVariable abc[2..0];
    SignalVariable xyz[2..0];
}

Variables core1 {
    SignalVariable scaninvals[5..0];
    SignalVariable scanoutvals[5..0];
}

Variables core2 {
    SignalVariable scaninvals[3..0];
    SignalVariable scanoutvals[3..0];
}

MacroDefs {
    M12 {
        W WBoth;
        C { SE1 = 1; SE2 = 1;}
        Shift { SI12 = 'core2::scaninvals[3..0]'
                'core1::scaninvals[5..0]';
        SO12 = 'core1::scanoutvals[5..0]'
                'core2::scanoutvals[3..0]';
        clk1=P; clk2=P}
Macro_Scan_C1 {  
    W WBoth;  
    M12 { SI1 = 'si1'; SO1 = 'so1';}
}

Macro_Scan_C2 {  
    W WBoth;  
    M12 { SI2 = 'si2'; SO2 = 'so2';}
}

// The definition of LockStep says is that the patterns are calling a common Macro.  
// LockStep requires all patterns within it to have the same number of calls to the common Macro.

PatternBurst B1 {  
    ParallelPatList LockStep {  
        P1 (Variables core1;)  
        P2 (Variables core2;)  
    }
}

Environment {  
    CTL coretests12 {  
        TestMode InternalTest;  
        DomainReferences {  
            Variables core1 core2;  
        }  
        PatternInformation {  
            PatternBurst B1 (Purpose Scan; )  
        }  
    }  
}

Include file_P1;  
Include file_P2;

NOTE: The above example is constructed to have a name-clash between variable names so that 
one can see the basic mechanism.
Now consider a new situation. This time let us add some UDL logic at the top level.

The most pertinent UDL logic to this example is some more scan cells. Let us say two scan cells are added to the top level netlist before the scan chains of the cores. The final integrated pattern would look like the following if not patterns were synchronized for the top level. No extra synchronization would be required. If the tests for the top level UDL are synchronized with that of the cores then, another set of patterns would have to be written for the UDL logic that supplies the values for the locations marked with XX. The new patterns would have to be added to the Lock-Step construct.

MacroDefs {
  M12 {
    W WBoth;
    C { SE1 = 1; SE2 = 1;}
    Shift { SI12 = 'core2::scaninvals[3..0]' \\
               'core1::scaninvals[5..0]' XX; \\
               SO12 = 'core1::scanoutvals[5..0]' \\
               'core2::scanoutvals[3..0]' XX; \\
               clk1=P; clk2=P}
  }

  Macro_Scan_C1 {
    W WBoth;
    M12 { SI1 = 'si1'; SO1 = 'so1';}
  }

  Macro_Scan_C2 {
    W WBoth;
    M12 {SI2 = 'si2'; SO2 = 'so2';}
  }
}

// The definition of LockStep says is that the patterns are calling a common Macro.  
// LockStep requires all patterns within it to have the same number of calls to the common Macro.

PatternBurst B1 {
  ParallelPatList LockStep {
    P1 {Variables core1;}
    P2 {Variables core2;}
  }
}

Environment {
  CTL coretests12 {
    TestMode InternalTest;
    DomainReferences {

Variables core1 core2;

PatternInformation {
    PatternBurst B1 { Purpose Scan; }
}

Include file_P1;
Include file_P2;
Now consider a new situation. Two Instances of the Same Core, C1.

Variables core1a {
    SignalVariable scaninvals[5..0];
    SignalVariable scanoutvals[5..0];
}

Variables core1b {
    SignalVariable scaninvals[5..0];
    SignalVariable scanoutvals[5..0];
}

MacroDefs {
    M12 {
        W WBoth;
        C { SE1 = 1;}
        Shift { SI12 = ‘core1a::scaninvals[5..0]’
            ‘core1b::scaninvals[5..0]’;
            SO12 = ‘core1a::scanoutvals[5..0]’
            ‘core1b::scanoutvals[5..0]’;
            clk1=P;}
    }
}

MacroDefs core1a {
    Macro_Scan_C1 {
        W WBoth;
        M12 { SI1 = ‘si1’; SO1 = ‘so1’;}
    }
}

MacroDefs core1b {
    Macro_Scan_C1 {
        W WBoth;
        M12 { SI2 = ‘si2’; SO2 = ‘so2’;}
    }
}

PatternBurst B1 {
    ParallelPatList LockStep {
        P1 {MacroDefs core1a; Variables core1a;}
        P1 {MacroDefs core1b; Variables core1b;}
    }
}

Environment {
    CTL coretests_1a1b {
        TestMode InternalTest;
        DomainReferences {
            CTL}
        }
    }
}
Variables corela corelb;
  MacroDefs corela corelb;
}
PatternInformation {
  PatternBurst B1 { Purpose Scan; }
}
}
}
Include file_P1;
Unequal Patterns
In this example there are fewer patterns for the core C1 than core C2. C1 has 1 pattern and C2 has 3 patterns.

Variables {
    SignalVariable scaninvals[5..0];
    SignalVariable scanoutvals[5..0];
}
Signals { a In; b In; c In; sil In; sol Out; sel In; clk1 In;)
SignalGroups { abc[2..0] = ‘a+b+c’;}
MacroDefs {
    Macro_Scan_C1 {
        W W1;
        C { sel = 1;}
        Shift { sil=’scaninvals[5..0]’;
            sol=’scanoutvals[5..0]’; clk1=P;}
        V{abc[2..0]=’abc[2..0]’;}
    }
}
Environment {
    CTL core1tests {
        PatternInformation {
            // Note CycleCount not shown but would have been used
            // to determine the number of Pp calls were needed.
            Pattern P1 { Purpose Scan; Macro Macro_Scan_C1;}
            Pattern Pp { Purpose Padding; Macro Macro_Scan_C1;}
        }
    }
}
Pattern P1 {
    M Macro_Scan_C1 { scaninvals[5..0]=101010;
        scanoutvals[5..0]=LLHHLL; abc[2..0]=000;}
}
Pattern Pp {
    M Macro_Scan_C1 { scaninvals[5..0]=111111;
        scanoutvals[5..0]=xxxxxx; abc[2..0]=000;}
}

Similarly, Core C2 has a macro that operates the scan chain and patterns that use the macro.

Variables {
    SignalVariable scaninvals[3..0];
    SignalVariable scanoutvals[3..0];
}
Signals { x In; y In; z In; si2 In; so2 Out; se2 In; clk2 In;}
SignalGroups { xyz[2..0]= ‘x+y+z’; }
MacroDefs {
    Macro_Scan_C2 {
        W W2;
        C { se2 = 1;}
        Shift { si2=’scaninvals[3..0]’;
              so2=’scanoutvals[3..0]’; clk2=P;}
        V{xyz[2..0]=’xyz[2..0]’;}
    }
}
Environment {
    CTL core2tests {
        PatternInformation {
            Pattern P2 { Purpose Scan; }
        }
    }
}
Pattern P2 {
    M Macro_Scan_C2 { scaninvals[3..0]=1111;
                     scanoutvals[3..0]=LHHH; xyz[2..0]=101;}
    M Macro_Scan_C2 { scaninvals[3..0]=1001;
                     scanoutvals[3..0]=LHLL; xyz[2..0]=100;}
    M Macro_Scan_C2 { scaninvals[3..0]=0011;
                     scanoutvals[3..0]=HHLH; xyz[2..0]=001;}
}

At the chip level we would like to have a situation where a new joint macro is written to operate the new larger scan chain and reuses the patterns that are in the files that come with the cores. The following is the outcome of the system integration task.

Signals {
    A In; B In; C In;
    X In; Y In; Z In;
    SI12 In; SO12 Out;
    SE1 In; SE2 In; clk1 In; clk2 In;
}
WBoth {
    Inh W1;
    Inh W2;
}
Variables {
    SignalVariable abc[2..0];
    SignalVariable xyz[2..0];
}
Variables core1 {
SignalVariable scaninvals[5..0];
SignalVariable scanoutvals[5..0];
}
Variables core2 {
    SignalVariable scaninvals[3..0];
    SignalVariable scanoutvals[3..0];
}
MacroDefs {
    M12 {
        W WBoth;
        C { SE1 = 1; SE2 = 1;}
        Shift { SI12 = 'core2::scaninvals[3..0]'
            'core1::scaninvals[5..0]';
            SO12 = 'core1::scanoutvals[5..0]'
            'core2::scanoutvals[3..0]';
                clk1=P; clk2=P}
    }
    Macro_Scan_C1 {
        W WBoth;
        M12 { SI1 = 'si1'; SO1 = 'so1';}
    }
    Macro_Scan_C2 {
        W WBoth;
        M12 {SI2 = 'si2'; SO2 = 'so2';}
    }
}

// The definition of LockStep says is that the patterns are calling a common Macro.
// LockStep requires all patterns within it to have the same number of calls to the common Macro.

PatternBurst B1 {
    ParallelPatList LockStep {
        PatList {
            P1 {Variables core1;}
                // would be nice to have a repeat construct here.
            Pp {Variables core1; }
            Pp {Variables core1; }
        }
        P2 {Variables core2;}
    }
}
Environment {
    CTL coretests12 {
        TestMode InternalTest;
    }
}
DomainReferences {
    Variables core1 core2;
}
PatternInformation {
    PatternBurst B1 { Purpose Scan; }
}

Include file_P1;
Include file_P2;

Mechanisms in place in CTL to make this all happen

1. Data-Protocol Separation.
2. Patterns are partitioned by protocol type.
3. Environment-PatternInfo allows for the specification of Patterns cycle-count, and the macro that it calls and Padding pattern identification.
4. LockStep is explicit.

Since Integration is not looking at pattern data everything needs to be explicit. The solution should work when multiple integrations steps are performed (hierarchical).