Draft Standard for Verilog Transaction Recording Extensions

This proposal has been prepared by Cadence Design Systems, Inc. for consideration by the IEEE 1364 working group for inclusion in the next revision of the IEEE 1364 standard.

1. Introduction

1.1. Scope

This proposal extends Verilog 1364-2001 to allow users to record transactions. A collection of system tasks and functions are proposed which can be used to explicitly record transactions. A method for automatic transaction recording is proposed, in which no additional code needs to be added to a design. Extensions to the VCD format are proposed so that transactions can be represented in a recording file. Finally, extensions to VPI are proposed to allow access to transaction objects in the design, and to transaction recording activity.

1.2. Purpose

The proposed extensions will allow users to raise the level of description, analysis and debugging of their designs to the transaction level. A transaction can represent for example:

- A transfer of high-level data or control information between the test bench and the design under verification (DUV) over an interface.
- Any sequence of signal transitions recorded in the simulation database as a transaction

A transaction has a begin time, an end time, and attributes. Examples of transactions include read operations, write operations, and packet transmissions. The transaction level is the level at which many users think about the design, so it is the level at which you can verify the design most effectively.

Transactions are recorded on *streams*, which users define at hierarchical positions in the design. A stream can contain *generators*, which users define to describe the kinds of transactions that are created during simulation. Each generator defines a collection of *attributes* which are defined by users, and which are meaningful to the transaction. The values of attributes are set for each transaction. Finally, transactions can be *linked* to each other. A link has a direction and a user-defined name, and specifies a relation between the two transactions.

Here’s an example:

```
module top;
        wire [7:0] addr, data;
        wire do_write, do_read;
        cpu_testbench cpu_tb_inst(addr, data, do_write, do_read);
        cpu_duv cpu_duv_inst(addr, data, do_write, do_read);
endmodule

module cpu_testbench(addr, data, do_write, do_read);
        output [7:0] addr, data;
        output do_write, do_read;
        reg [7:0] addr, data;
        reg do_write, do_read;
        integer cpu0_stream;
        integer write_generator, read_generator;
        integer transaction_handle;

        initial begin
                $tr_open(“tr_file”);

                cpu0_stream = $tr_stream(“cpu0_stream”);
                write_generator = $tr_generator(cpu0_stream, “write”);
                read_generator = $tr_generator(cpu0_stream, “read”);
                $tr_begin_attribute(read_generator, addr);
                $tr_end_attribtue(read_generator, data);
                $tr_begin_attribute(write_generator, addr, data);

                // Do a write to the DUV:
                addr = 10; data = 15; do_write = 1;
                transaction_handle = $tr_begin(write_generator);
                // wait until the write is complete
                do_write = 0;
                $tr_end(transaction_handle);

                // Do a read from the DUV:
                addr = 11; do_read = 1;
                transaction_handle = $tr_begin(read_generator);
                // wait until the read is complete
                do_read = 0;
                $tr_end(transaction_handle);
        end
endmodule
```

1. The top-level instantiates a “cpu” DUV, and a cpu test bench.
2. Transactions are recorded in “tr_file”.
3. A stream is used to represent the transaction activity on a cpu interface. The name of the stream is “cpu0_stream”, and it is positioned at “top.test.”
4. There are two transaction generators: “read” represents read transaction activity, and “write” represents write activity.
5. Each generator has two attributes: “addr” contains the value of the address, and “data” contains the value of the data. For read transactions, the addr attribute’s value is set at the beginning of the transaction and the data attribute’s value is set at the end of the transaction. For write transactions, both the addr and data attributes’ values are set at the beginning of the transaction.
6. When a write to the DUV is initiated in the example, a write transaction is begun. The values of the addr and data variables are recorded in the associated attributes of the transaction. The write transaction is ended when the write operation is completed.
7. When a read from the DUV is initiated, a read transaction is begin. The value of the “addr” attribute is set to the address that will be read.
8. When the read activity is complete, the value of the “data” attribute is set, and the transaction is ended.
9. If this activity had been caused by some other higher level activity that is also represented by a transaction, then “successor” and “predecessor” links with the other transaction can be explicitly made with the **$tr_link** task. Or, if these read and write transactions are used to provide detailed information of a larger transaction, then the transactions can be linked as “parent” and “child” relations.

1.3. Requirements

1.3.1. Allow users to explicitly express transaction activity

Users should be able to specify in their Verilog code:
1. When a transaction starts and ends
2. The names and values of attributes of each transaction
3. The stream and generator on which a transaction occurs
4. Ability to get a handle to a transaction, so this transaction can be referenced elsewhere
5. Link this transaction to another transaction
6. Ability to turn on/off recording, for the full design or for parts of it.

1.3.2. Semantics for automatic transaction recording

It should also be possible to get limited transaction recording without any additional transaction recording Verilog code.

1.3.3. Extensions to VPI to provide access to transaction objects

A VPI application should allow users to:

1. Register callbacks on transaction events, such as begin and end
2. Traverse the collection of transaction objects
3. Create transaction objects, and cause transaction activity

1.3.4. Extensions to VCD for transaction recording

Define a way in which transactions can be recorded within the current VCD format, or if not feasible then define extensions to VCD.

2. Transaction Recording System Tasks and Functions

This section lists the proposed systems tasks and functions, which will have a $tr_ prefix.

2.1. $tr_open: Open a transaction recording file

```
integer my_tr_file_handle = $tr_open(
        “transaction_file_name”,
        (list_of_argument_pairs) )
```

This function allows you to explicitly open a transaction recording database. The first argument is the name of the file that will contain the transactions. If this argument is omitted then the default name is implementation dependent.

$tr_open returns a handle to the recording file. This handle can be used to explicitly refer to a particular transaction file, which is useful if multiple files are required.

The list_of_argument pairs is optional. This proposal specifies that the standard have one kind of argument, but other argument kinds that can be tool-specific. These other argument kinds should ignored without errors or warnings by those tools that don’t support the argument.

The first argument in the list_of_argument_pairs describes the kind of argument, and the second argument contains the value.

The proposed standard argument kind is:

Argument_kind string	Value	Effect
“scope”	Levels	“scope” specifies a hierarchical scope in the design, and levels specifies the depth to which transaction recording is enabled. If the levels argument is omitted or its value is 0 then recording is enabled for all transaction activity at the specified scope and below. The “scope” argument can be included in a **$tr_open** call multiple times, and the effects are cumulative. If the “scope” argument is missing from the **$tr_open** call, then transaction recording is enabled for the entire design.

Examples:

$tr_open(“my_file”);   // Start recording all transaction activity.

my_tr_handle = **$tr_open**(“my_file”, “my_scope”, 1);
// Start Record only at the “my_scope” level.

2.2. $tr_close: Close a recording file

$tr_close(
    tr_file_handle)

End all open transactions, and close the recording database file that is associated with the tr_file_handle integer argument.

If the tr_file_handle argument is omitted, then the most recently opened file is closed.

2.3. $tr_set_recording: Enable/disable transaction recording at a specified scope

$tr_set_recording(
    “scope” ,

```
        level,
        recording_control)
```

This tasks allows you to dynamically control transaction recording in different parts of the Verilog design. If the recording_control argument is non-zero, then transactions are recorded on streams that are defined in the specified scope and below, to the depth specified by the level argument. If the recording_control argument is 0, then transaction recording is suspended for the specified scopes. Calls to this task are cumulative.

When recording is suspended, any currently open transactions are closed.

When recording is resumed, any currently open transactions are immediately opened in the recording file. The begin time of such transactions are set to the current simulation time. The values of attributes of such transactions are not defined.

Example:

```
        // Suspend recording at "my_scope" and 1 level below:
        $tr_set_recording("my_scope", 2, 0);
```

2.4. $tr_stream: Create a transaction stream

```
integer stream_handle = $tr_stream(
        "stream_name",
        "scope",
        "stream_kind",
        tr_file_handle)
```

This function returns an integer handle to the newly created transaction recording stream in the specified scope, with the specified stream name.

If the specified scope is omitted or NULL, then the current scope in which this call is made is used.

The "stream_kind" specifies the kind of stream. This argument is optional and allows further information to be associated with the stream. The default is "Transaction".

The tr_file_handle refers to a particular recording file. If omitted, then the most recently opened transaction recording file is used.

If the function fails, a warning is given and the stream_handle that is returned is 0. (Valid stream_handles are > 0.)

Example:

```
// Create a stream in the current scope:
my_stream_handle = $tr_stream("my_stream");
```

2.5. $tr_generator: Create a transaction generator in a stream

```
integer generator_handle = $tr_generator(
        stream_handle,
        "generator_name" )
```

This function returns a handle to a newly created transaction generator on the specified stream. The **$tr_begin_attribute**, **$tr_end_attribute**, and **$tr_record_attribute** tasks, which are described later, are used to specifies the attributes associated with a generator. If none of these calls are made, then the generator has no attributes.

Example:

```
// Create a generator on a stream:
$tr_generator(my_stream_handle, "my_generator");
```

2.6. $tr_begin_attribute: Define an attribute of a generator that is set at the beginning of a transaction

```
$tr_begin_attribute(
        generator_handle,
        Verilog_variable,
        "optional_name",
        ( list_of_attribute_properties ) )
```

This task defines an attribute of a transaction generator whose values are recorded at the beginning of a transaction.

The generator_handle argument specifies the generator for which this attribute is being defined.

The Verilog_variable is associated with a new attribute of this generator.

The "optional_name" is an optional argument that defines the name of this attribute. If omitted the name is the simple name of the Verilog_variable.

The list_of_attribute_properties is an optional list of pairs of argument that can specify further tool-specific information about the attribute. A tool should ignore any item in this list if it's not applicable to the tool. An example of such properties are:

The list_of_arguments is an optional list that associates Verilog variables with attributes.

In addition, each attribute can also have an optional list_of_attribute_properties, which specify further information about the attribute. Properties include:

“radix”	Specifies how the attribute’s value will be displayed.
“position”	Specifies the position of this attribute in a display

Example:

$tr_begin_attribtue(my_generator_handle, var_1);

$tr_begin_attribute(my_generator_handle, var_2, “a_better_name”);

// In this example the radix of the attribute is defined as hex, which a tool might
// interpret for transaction display purposes:
$tr_begin_attribute(my_generator_handle, var_3, , “radix”, “hex”);

2.7. $tr_end_attribute: Define an attribute of a generator that is set at the end of a transaction

$tr_end_attribute(
generator_handle,
Verilog_variable,
“optional_name”,
(list_of_attribute_properties))

This task defines an attribute of a transaction generator whose values are recorded at the beginning of a transaction.

The arguments are the same as the **$tr_begin_attribute** task, which was described above.

2.8. $tr_begin: Begin a transaction

This function has several forms, which each cause a new transaction to be created.

Integer transaction_handle = **$tr_begin**(
generator_handle,
optional_begin_time)

This form of the task begins a transaction of the specified generator_handle. The optional_begin_time argument is of type time, and specifies the time at which the transaction begins. This value can only be at the current simulation time, or in the past. If omitted, the transaction begins at the current simulation time.

```
integer transaction_handle = $tr_begin(
        generator_handle,
        "relation_name",
        other_transaction_handle,
        optional_begin_time)
```

This form of the function begins a transaction, and also establishes a relation link to the transaction given by other_transction_handle. This form combines **$tr_begin** with **$tr_link**, which is described later.

```
integer transaction_handle = $tr_begin(
        stream_handle,
        "generator_name",
        optional_begin_time)
```

```
integer transaction_handle = $tr_begin(
        stream_handle,
        "generator_name",
        "relation_name"
        other_transaction_handle,
        optional_begin_time)
```

In these forms of the function, only a transaction generator name is specified, in which case a generator is automatically created in the specified stream_handle.

2.9. $tr_end: End a transaction

```
$tr_end(
        transaction_handle,
        optional_end_time)
```

This task ends a transaction, and records the values of Verilog variables that were given in **$tr_end_attribute** calls. The optional_end_time argument is of type time, and specifies the end time of this transaction. This time must be at the current time of in the past. If omitted the current simulation time is used.

2.10. $tr_record_attribute: Record an attribute on a transaction

```
$tr_record_attribute(
        transaction_handle,
        Verilog_variable, "optional_attribute_name",
        ( list_of_attribute_properties ) )
```

This task allows you to record a transaction attribute's value after a transaction has started. The Verilog_variable and "optional_attribute_name," and list_of_attribute_properties are defined in **$tr_begin_attribute**.

For the generator of the specified transaction_handle, subsequent calls to this task for the same Verilog_variable/optional_attribute_name do not require the optional list_of_attribute_properties because the list is retained, and cannot be overridden.

2.11. $tr_get_transaction_handle: Get a transaction handle from a stream

```
integer transaction_handle = $tr_get_transaction_handle(
                    stream_handle)
```

This function returns the transaction_handle of the most recently started transaction on the specified stream_handle.

2.12. $tr_link: Link two transactions

```
$tr_link(
          transaction_handle_1,
          transaction_handle_2,
          "relation_name")
```

This task links transaction_handle_1 to transaction_handle_2, with the specified relation_name. Transactions are linked because they might have a cause and effect relation between them, and this additional information is useful if analysis is performed on transaction activity.

3. Automatic Transaction Recording

This following is a proposal to allow automatic transaction recording:

1. When a user's Verilog task or function is called, a transaction starts.
2. The input and inout arguments of the task are recorded as attributes in the transaction.
3. When the task returns, the transaction is ended, and out and inout arguments are recorded as attributes. The return value of a function is also recorded.
4. Each task defines a stream which is positioned at the hierarchical position of the module instance that contains the task.

5. Each task also defines a generator, which has attributes that are taken from the arguments of the task.

Automatic transaction recording of parts of the design can be controlled by a variant of the **$tr_set_recording** task. **$tr_set_auto_recording** refers only to automatic recording of user Verilog tasks and functions.

4. Extensions to VCD for Transaction Recording

A method for storing transaction information in a VCD file is proposed here.

1. A stream defines a new scope
2. The stream contains a variable that contains transaction activity
3. Each generator defines a new scope in the stream
4. Each generator has a variable, with a value set at the beginning of simulation to a positive unique integer ID
5. The attributes of a generator are defined as variables in the generator's scope
6. When a transaction starts, the stream activity variable is set the generator's ID
7. When the a transaction ends, the stream activity is set to the negative value of the generator's ID. Also, all the values of all attributes of the transactions are dumped.

5. Transaction Recording VPI Extensions

This section proposes to extend VPI to allow C access to transaction recording activity.

1. VPI applications can register callbacks so that they can be notified when transactions start, end, and when transactions are linked. In addition, applications can get callbacks when recording files are opened or closed, and when recording for parts of the design is enabled or disabled.
2. VPI applications can traverse transaction objects: streams, generators, attributes. Also, currently active transactions can be traversed.
3. VPI applications can create transaction objects, and can cause transaction activity.

6. Example

The following is a simple example of using the Transaction tasks and functions:

```
module top;
        integer my_stream;
        integer my_read_generator;
        integer my_transaction_handle;
        reg [7:0] addr;
        reg [7:0] data;

        initial begin
                $tr_open(“my_transaction_file”);
                my_stream = $tr_stream(“my_stream”);
                my_read_generator = $tr_generator(my_stream, “my_read_generator”);
                $tr_begin_attribute(my_read_generator, addr);
                $tr_end_attribute(my_read_generator, data);
                my_transaction_handle = $tr_begin(my_read_generator);
                // The addr attribute is recorded
                #100 …
                $tr_end(my_transaction_handle);  // The data attribute is recorded
        end
endmodule
```