29.3.2.1 Using `vpi_get_assertion_info`

CHANGE:

29.3.2.1 Using `vpi_get_assertion_info`

Static information can be obtained directly from an assertion handle by using `vpi_get_assertion_info()`, as shown below.

```
typedef struct t_vpi_source_info {
   PLI_BYTE8 *fileName;
   PLI_INT32 startLine;
   PLI_INT32 startColumn;
   PLI_INT32 endLine;
   PLI_INT32 endColumn;
} s_vpi_source_info, *p_vpi_source_info;
typedef struct t_vpi_assertion_info {
   PLI_BYTE8 *assertName; /* name of assertion */
   vpiHandle instance; /* instance containing assertion */
   PLI_BYTE8 defname; /* name of module/interface containing assertion */
   vpiHandle clock; /* clocking expression */
   PLI_INT32 assertionType; /* vpiSequenceInst, vpiAssert, vpiAssume,
                               vpiCover, */
                            /* vpiPropertyInst, vpiImmediateAssert */
   s_vpi_source_info sourceInfo;
} s_vpi_assertion_info, *p_vpi_assertion_info;
PLI_INT32 vpi_get_assertion_info (assert_handle, p_vpi_assertion_info);
```

This call obtains all the static information associated with an assertion.

The inputs are a valid handle to an assertion and a pointer to an existing `s_vpi_assertion_info` data structure. On success, the function returns `TRUE` and the `s_vpi_assertion_info` data structure is filled in as appropriate. On failure, the function returns `FALSE` and the contents of the assertion data structure are unpredictable.

Assertions can occur in modules and interfaces: for assertions defined in modules, the instance field in the `s_vpi_assertion_info` structure shall contain the handle to the appropriate module or interface instance. Note that VPI does not currently define the information model for interfaces and therefore the interface instance handle shall be implementation dependent. The clock field of that structure contains a handle to the event expression representing the clock for the assertion, as determined by 18.14.

NOTE: a single call returns all the information for efficiency reasons.

TO:

~~**29.3.2.1 Using** `vpi_get_assertion_info`~~

~~Static information can be obtained directly from an assertion handle by using `vpi_get_assertion_info()`, as shown below.~~

```
typedef struct t_vpi_source_info {
   PLI_BYTE8 *fileName;
   PLI_INT32 startLine;
   PLI_INT32 startColumn;
   PLI_INT32 endLine;
   PLI_INT32 endColumn;
} s_vpi_source_info, *p_vpi_source_info;
typedef struct t_vpi_assertion_info {
   PLI_BYTE8 *assertName; /* name of assertion */
   vpiHandle instance; /* instance containing assertion */
```

```
   PLI_BYTE8 defname; /* name of module/interface containing assertion */
   vpiHandle clock; /* clocking expression */
   PLI_INT32 assertionType; /* vpiSequenceInst, vpiAssert, vpiAssume,
                               vpiCover, */
                            /* vpiPropertyInst, vpiImmediateAssert */
   s_vpi_source_info sourceInfo;
} s_vpi_assertion_info, *p_vpi_assertion_info;
PLI_INT32 vpi_get_assertion_info (assert_handle, p_vpi_assertion_info);
```

This call obtains all the static information associated with an assertion.

The inputs are a valid handle to an assertion and a pointer to an existing `s_vpi_assertion_info` data structure. On success, the function returns `TRUE` and the `s_vpi_assertion_info` data structure is filled in as appropriate. On failure, the function returns `FALSE` and the contents of the assertion data structure are unpredictable.

Assertions can occur in modules and interfaces: for assertions defined in modules, the instance field in the `s_vpi_assertion_info` structure shall contain the handle to the appropriate module or interface instance. Note that VPI does not currently define the information model for interfaces and therefore the interface instance handle shall be implementation dependent. The clock field of that structure contains a handle to the event expression representing the clock for the assertion, as determined by 18.14.

NOTE: a single call returns all the information for efficiency reasons.

29.3.2.2 Extending `vpi_get()` and `vpi_get_str()`

CHANGE

In addition to `vpi_get_assertion_info`, the following existing VPI functions are also extended:

TO

In addition to `vpi_get_assertion_info`,tThe following existing VPI functions are also extended:

29.4.2 Placing assertions callbacks

CHANGE

Use `vpi_register_assertion_cb()` to place an assertion callback; the prototype is:

```
/* typedef for vpi_register_assertion_cb callback function */
typedef  PLI_INT32 (vpi_assertion_callback_func)(
   PLI_INT32 reason,          /* callback reason */
   p_vpi_time cb_time,        /* callback time */
   vpiHandle assertion,       /* handle to assertion */
   p_vpi_attempt_info info,   /* attempt related information */
   PLI_BYTE8 *user_data       /* user data entered upon registration */
);

vpiHandle vpi_register_assertion_cb(
   vpiHandle assertion,       /* handle to assertion */
   PLI_INT32 reason,          /* reason for which callbacks needed */
   vpi_assertion_callback_func *cb_rtn,
   PLI_BYTE8 *user_data       /* user data to be supplied to cb */
);
typedef struct t_vpi_assertion_step_info {
   PLI_INT32 matched_expression_count;
   vpiHandle *matched_exprs;  /* array of expressions */
   p_vpi_source_info *exprs_source_info; /* array of source info */
   PLI_INT32 stateFrom, stateTo;/* identify transition */
```

```
} s_vpi_assertion_step_info, *p_vpi_assertion_step_info;
typedef struct t_vpi_attempt_info {
   union {
      vpiHandle failExpr;
      p_vpi_assertion_step_info step;
   } detail;
   s_vpi_time attemptStartTime;  /* Time attempt triggered */
} s_vpi_attempt_info, *p_vpi_attempt_info;
```

TO

Use `vpi_register_assertion_cb()` to place an assertion callback; the prototype is:

```
/* typedef for vpi_register_assertion_cb callback function */
typedef  PLI_INT32 (vpi_assertion_callback_func)(
   PLI_INT32 reason,          /* callback reason */
   p_vpi_time cb_time,        /* callback time */
   vpiHandle assertion,       /* handle to assertion */
   p_vpi_attempt_info info,   /* attempt related information */
   PLI_BYTE8 *user_data       /* user data entered upon registration */
);

vpiHandle vpi_register_assertion_cb(
   vpiHandle assertion,       /* handle to assertion */
   PLI_INT32 reason,          /* reason for which callbacks needed */
   vpi_assertion_callback_func *cb_rtn,
   PLI_BYTE8 *user_data       /* user data to be supplied to cb */
);

typedef struct t_vpi_assertion_step_info {
   PLI_INT32 matched_expression_count;
   vpiHandle *matched_exprs;  /* array of expressions */
   p_vpi_source_info *exprs_source_info; /* array of source info */
   PLI_INT32 stateFrom, stateTo;/* identify transition */
} s_vpi_assertion_step_info, *p_vpi_assertion_step_info;

typedef struct t_vpi_attempt_info {
   union {
      vpiHandle failExpr;
      p_vpi_assertion_step_info step;
   } detail;
   s_vpi_time attemptStartTime;  /* Time attempt triggered */
} s_vpi_attempt_info, *p_vpi_attempt_info;
```

31.10 Reading data from multiple databases and/or different read library providers

CHANGE

The structure shall have an entry for *every* VPI routine; the order and synopsis of these entries within the structure shall exactly match the order and synopsis of function definitions in Clause 27 of the P1364 Verilog standard. After those entries the SystemVerilog VPI routine additions for assertions `vpi_get_assertion_info()` and then `vpi_register_assertion_cb()` shall be added in that order. Then all new reader routines defined in Table shall be added in exactly the order noted in the table. If a particular extension does not support a specific VPI routine, then it shall still have an entry (with the correct prototype), and a dummy body that shall always have a return (consistent with the VPI prototype) to signify failure (i.e. `NULL` or `FALSE`). The routine call must also raise the appropriate VPI error, which can be checked by `vpi_chk_error()`, and/or automatically generate

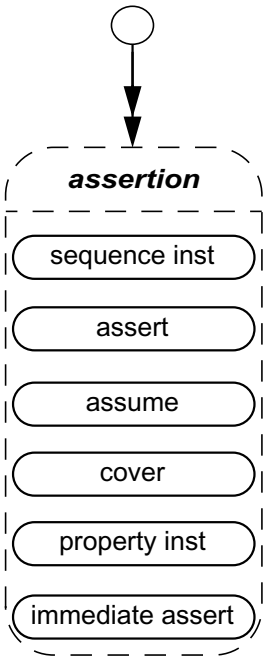
an error message in a manner consistent with the specific VPI routine.

TO

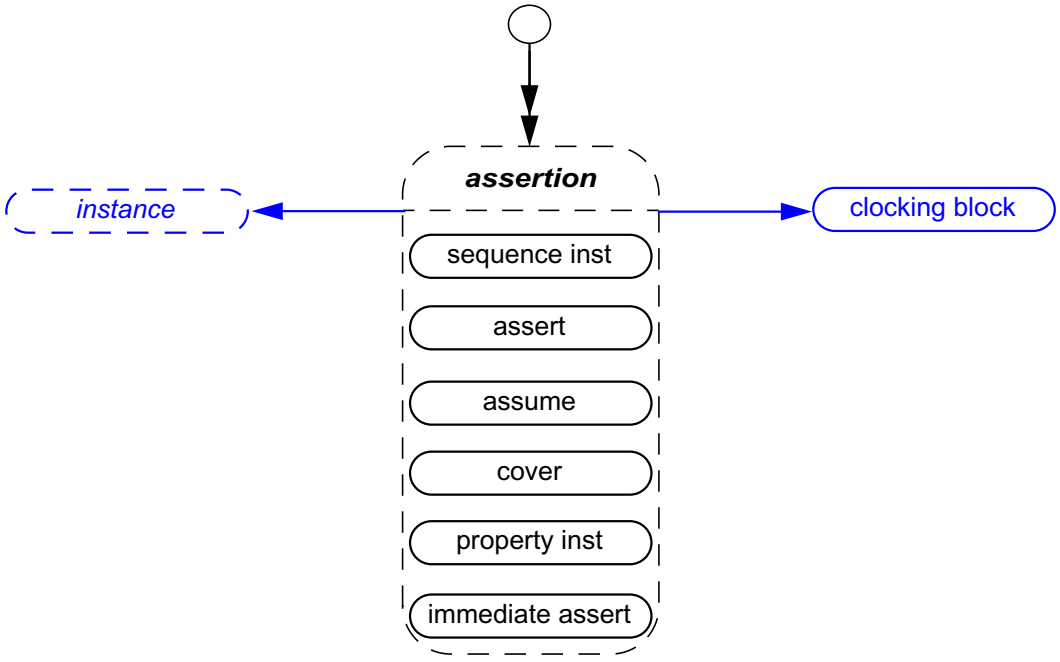
The structure shall have an entry for *every* VPI routine; the order and synopsis of these entries within the structure shall exactly match the order and synopsis of function definitions in Clause 27 of the P1364 Verilog standard. After those entries the SystemVerilog VPI routine ~~additions for assertions~~ ~~`vpi_get_assertion_info()`~~ ~~and then~~ `vpi_register_assertion_cb()` shall be added ~~in that order~~. Then all new reader routines defined in Table shall be added in exactly the order noted in the table. If a particular extension does not support a specific VPI routine, then it shall still have an entry (with the correct prototype), and a dummy body that shall always have a return (consistent with the VPI prototype) to signify failure (i.e. `NULL` or `FALSE`). The routine call must also raise the appropriate VPI error, which can be checked by `vpi_chk_error()`, and/or automatically generate an error message in a manner consistent with the specific VPI routine.

32.31 Assertion

CHANGE



TO



-> assertion type
int: vpiAssertType

-> location
str: vpiFile
int: vpiStartLine
int: vpiColumn
int: vpiEndLine
int: vpiEndColumn

-> assertion name
str: vpiName

Annex A

(normative)

sv_vpi_user.h

CHANGE

```
/* assertion related structs and types */
typedef struct t_vpi_source_info {
   PLI_BYTE8 *fileName;
   PLI_INT32 startLine;
   PLI_INT32 startColumn;
   PLI_INT32 endLine;
   PLI_INT32 endColumn;
} s_vpi_source_info, *p_vpi_source_info;

typedef struct t_vpi_assertion_info {
   PLI_BYTE8 *assertName;           /* name of assertion */
   vpiHandle instance;              /* instance containing assertion */
   PLI_BYTE8 defname;               /* name of module/interface containing
                                     * the assertion */
   vpiHandle clock;                 /* clocking expression */
   PLI_INT32 assertionType;         /* vpiSequenceInst, vpiAssert, vpiAssume,
                                     * vpiCover, vpiPropertyInst,
                                     * vpiImmediateAssert */
   s_vpi_source_info sourceInfo;
} s_vpi_assertion_info, *p_vpi_assertion_info;

typedef struct t_vpi_assertion_step_info {
   PLI_INT32 matched_expression_count;
   vpiHandle *matched_exprs;                /* array of expressions */
   p_vpi_source_info *exprs_source_info;    /* array of source info */
   PLI_INT32 stateFrom, stateTo;            /* identify transition */
} s_vpi_assertion_step_info, *p_vpi_assertion_step_info;
```

TO

```
/* assertion related structs and types */
typedef struct t_vpi_source_info {
   PLI_BYTE8 *fileName;
   PLI_INT32 startLine;
   PLI_INT32 startColumn;
   PLI_INT32 endLine;
   PLI_INT32 endColumn;
} s_vpi_source_info, *p_vpi_source_info;

typedef struct t_vpi_assertion_info {
   PLI_BYTE8 *assertName;           /* name of assertion */
   vpiHandle instance;              /* instance containing assertion */
   PLI_BYTE8 defname;               /* name of module/interface containing
                                     * the assertion */
   vpiHandle clock;                 /* clocking expression */
   PLI_INT32 assertionType;         /* vpiSequenceInst, vpiAssert, vpiAssume,
                                     * vpiCover, vpiPropertyInst,
                                     * vpiImmediateAssert */
```

```
   s_vpi_source_info sourceInfo;
} s_vpi_assertion_info, *p_vpi_assertion_info;

typedef struct t_vpi_assertion_step_info {
   PLI_INT32 matched_expression_count;
   vpiHandle *matched_exprs;                /* array of expressions */
   p_vpi_source_info *exprs_source_info;    /* array of source info */
   PLI_INT32 stateFrom, stateTo;            /* identify transition */
} s_vpi_assertion_step_info, *p_vpi_assertion_step_info;
```

CHANGE

```
/* typedef for vpi_register_assertion_cb callback function */
typedef PLI_INT32(vpi_assertion_callback_func)(
   PLI_INT32 reason,                /* callback reason */
   p_vpi_time cb_time,              /* callback time */
   vpiHandle assertion,             /* handle to assertion */
   p_vpi_attempt_info info,         /* attempt related information */
   PLI_BYTE8 *user_data             /* user data entered upon registration */
);

/* assertion specific VPI functions */
PLI_INT32 vpi_get_assertion_info(
   vpiHandle assertion,             /* handle to assertion */
   p_vpi_assertion_info info        /* assertion related information */
);

vpiHandle vpi_register_assertion_cb(
   vpiHandle assertion,             /* handle to assertion */
   PLI_INT32 reason,                /* reason for which callbacks needed */
   vpi_assertion_callback_func *cb_rtn,
   PLI_BYTE8 *user_data             /* user data to be supplied to cb */
);
```

TO

```
/* typedef for vpi_register_assertion_cb callback function */
typedef PLI_INT32(vpi_assertion_callback_func)(
   PLI_INT32 reason,                /* callback reason */
   p_vpi_time cb_time,              /* callback time */
   vpiHandle assertion,             /* handle to assertion */
   p_vpi_attempt_info info,         /* attempt related information */
   PLI_BYTE8 *user_data             /* user data entered upon registration */
);

/* assertion specific VPI functions */
PLI_INT32 vpi_get_assertion_info(
   vpiHandle assertion,             /* handle to assertion */
   p_vpi_assertion_info info        /* assertion related information */
);

vpiHandle vpi_register_assertion_cb(
   vpiHandle assertion,             /* handle to assertion */
   PLI_INT32 reason,                /* reason for which callbacks needed */
   vpi_assertion_callback_func *cb_rtn,
   PLI_BYTE8 *user_data             /* user data to be supplied to cb */
);
```