SystemVerilog Interfacing to Foreign Languages

Comments not intended for inclusion in the LRM are in red text.

Overview

Since the foreign language most often used is C and C linkage is well defined, it is assumed that all foreign code will use the same interface as C for direct linking of code.There is an assumption in this document that 2-state (C compatible) data types are passable by reference (pointer) and that C function prototypes are parseble in SV (since it desirable to minimize re-writing them), i.e. that C '*' and '&' pointer and reference declaration syntax can be used in SV.

Calling C From SV

To call a C routine from SV requires an external declaration in the SV source. Direct binding through code linkage requires that has a valid C name, since names in Verilog may not be valid an optional C name can be added to the external declaration. External routines can also be attributed "pure" implying that there are no side-effects, needing context, or "static" which implies dynamic binding[1]. The complete syntax is:

```
extern_decl     ::= extern (attribute (, attribute) *) ?
                    "linker name" ?
                    function_decl | task fname ( extern_func_args ? )
function_decl ::= function return-type
attribute       ::= pure | context | line | static
```

To declare a standard C function like "write" which is context free one could use the following code in SV:

```
extern int write(int, const char *buf, int);
```

If using *write* is not going to feedback into the simulation and it is to replace a differently named SV routine one could use:

```
extern pure "write" int sv_write(int, const char *buf, int);
```

The direction qualifier input is considered synonymous with "const", so the following declarations are equivalent:

```
extern void bcopy(const void *s1, void *s2, size_t n);
```

1. There is no direct C entrypoint so no symbol table entry.

```
extern void bcopy(input void *s1, void *s2, size_t n);
```

Ellipsis ("...") can be used as in C to indicate that remaining arguments are of undetermined type. Using the "line" attribute will allow passing of the all the argument types for a call instance to dynamic linking mechanisms (described below). This can be used for declaring standard C library calls:

```
extern int printf(const char *format, /* args*/ ...);
```

Functions requiring context are called with two extra arguments: a PLI handle for the calling instance (which may be $root) and a pointer to some permanently allocated memory which the callee can use for maintaining state.The structure of the permanently allocated space is described in C as:

```
typedef struct {
    union {
            void    *ptr;
            int     data[2];
            double dbl;
    }       user_context;       /* user-context, initialy zero, 64 bits (aligned) */
    int     context_version;            /* == 0, struct stops here */
    union {
        struct {
            int             call_num;           /* call on line */
            int             line_number;        /* source line number */
            const char      *file_name;         /* may be null string "" but not null*/
        } call_inst; // if (context_version & SVC_CONTEXT_LINE)[1]
    } cvu;
} svcContext;
```

This can be used by routines for reporting errors, coverage testing etc. and the callee can write its own values into the user_context union e.g. a pointer to another array of memory. The context_version field is zero for this struct, other versions which overlay it will use other values. For example, the following declaration in SV:

```
extern pure,context,line void check_val(int);
```

Will expect a C routine like this:

```
void check_val(handle instance, svcContext *p_context, int data)
{
    if (data > 911 && !p_context->user_context.data[0]) {
        if (context_version & SVC_CONTEXT_LINE) { // have line info
            fprintf(stderr,  "%s:%d(%d) / %s - Error data out of range! = %d\n",
                        p_context->cvu.call_inst.file_name,
```

1. Defined in header file(s)

```
                        p_context->cvu.call_inst.line_number,
                        p_context->cvu.call_inst.call_num,
                        tf_getinstance(),
                        data);
        } else {
            fprintf(stderr,  "%s - Error data out of range! = %d\n",
                        tf_getinstance(),
                        data);
        }
        p_context->user_context.data[0] = 1; // don't warn again
    }
}
```

If the static version of the external declaration is used, then the user code must register a locator routine before the simulation initializes so that the simulator can link up routines that are left undefined by elaboration. The registration routine is svcRegisterLocater:

```
/* template for external context calls */
typedef void (*svcExtFunc)(handle,svcContext *,...);

/* template for locater routines */
typedef svcExtFunc (*svcLocater)(   void *locater_context,
                                    char *mod_spec,
                                    char *inst_name,
                                    char *rtn_name,
                                    svcContext *svcContext,
                                    ... /* argument types */);

int svcRegisterLocator(svcLocater,void *locator_context,char *module_spec,
                        int capabilities)
```

This method allows modules to bind to C routines which are static or are overloaded in C++, the following code demonstrates how to set up binding for a C++ function whose name is not predictable[1]: The capabilities argument of the locator is reserved for future enhancements; like changing how the arguments are specified, in this version capabilities should be zero for linking by routine name only and SVC_ARGS_CSTR[2] for argument types to be given as C strings.

```
extern "C" {
#include "sv2c.h" // svcRegisterLocator etc.
}
```

1. Due to name mangling by the C++ compiler.
2. Defined as 1 in the header file(s).

```
static pointer myBinder(void *,char *,char *,char *,void **,...);
class root_cpp {
    root_cpp() {svcRegisterLocator(myBinder,this,"$root",0); }
}

static root_cpp RootCpp; // calls svcRegisterLocator when constructed

static int cpp_routine(handle inst,svcContext *p_context,int data) {
    // a routine called from SV:    extern context int cpp_routine(int);
}

static svcExtFunc myBinder( void *context,char *modname,char *inst_name,char *rtn_name,
                            svcContext *p_context,...)
{
    svcExtFunc rtn_addr = 0;

    if (0 == strcmp(rtn_name,"cpp_routine")) rtn_addr = cpp_routine;

    if (rtn_addr) p_context->user_context.ptr = context; // set context for future calls.

    return rtn_addr;
}
```

The binder function is called once for each context (module or line and position) in which SV calls the routine so that it can intialize all contexts if the extern declaration had the "line" attribute set, otherwise all calls within the module use the same context. The extra arguments to the binder function are the types of the return and arguments to the routine call in SV, and may vary from context to context; the list values are C strings and is terminated with a null pointer. If "..." was used in the declaration it will appear in the list and will be followed by the actual types if the call is being bound per line. The SV simulator will report a fatal error if no match is found or multiple matches are found with the same priority. Priority is determined by the matching of the *module_spec* argument of the locator, an exact match takes priority over a wild-card which take priority over no module specification (null), binder functions are not called if there *module_spec* does not match the instance.

Verilog has various call-back points in its compile and elaboration phases, if locator functions are not registered by construction they can be registered by the post-compile pre-elaboration call-back.

Scoping

Extern function declarations can appear in any scope where a function or task can be declared, and are selected using the same rules that would apply to an actual function or task. Extern declarations in different scopes can use different linker names for the same SystemVerilog name and the SystemVerilog name can only be used once in a given scope.

Linker Name Specification and Use

In the case of non-static external function declarations the linker name is interpreted as the name of a C

routine to be included when the simulator code is linked, or as a library and name pair in the format:

<library name>:<routine name>

The library specification can be given to the simulator in a vendor/platform specific fashion for static linking or used for dynamic linking at runtime. For example the “lvd:driver” would specify the routine named “driver” in the library “lvd”. If no prior definition of “lvd” is given the simulator will try to load the library at runtime and look for the routine “driver” in it. When a library is loaded dynamically the simulator will attempt to call the function “svcInitLib” within the library so that it can register locator functions.The only argument to the svcInitLib call is it’s dynamic linker handle[1].

On Solaris you could specify “nsl:socket” to link the standard socket call from /usr/lib/libnsl.so, see the man pages on dlopen for details.

The dynamic library mechanism is also used if the locator functions do not return a match.

Note: dynamic linking does not require re-linking the simulator for a foreign code change.

How to specify a code library to the simulator I would rather leave for later, but I would expect something like “-olib nsl=/usr/lib/libnsl.a”, the implication being that it is only added to the static link if there is an “nsl:*” used somewhere in the design.

Calling SVfrom C

To access an SV task or function from C it needs an “export” declaration. Export declarations can be placed in modules for exporting module specific tasks and functions or in $root for global tasks and functions, and module specific tasks and functions can also be exported from root using the special syntax “<module spec>::” in front of the task or function name:

export_decl	::=	export attribute (, attribute) * ?
		“linker name” ?
		[module spec::]fname
attribute	::=	static | queueable
module spec	::=	(modulename|interfacename) (::(modulename|interfacename))* ?[2]

As with external declarartions the SV name may not be a valid name so a valid C name can be provided for use by the linker. Similarly declaring the export as “static” implies only dynamic binding is used. Routines and tasks exported from $root are context free, routines and tasks exported from modules need to be called with a reference to the specific instance the call applies to. Standard PLI functions can be used to find a particular instance and the instance handle is passed by the caller as the first argument.

---

1. This platform specific
2. top::foo would refer to a module foo declared inside module top

Non-static exports are handled by the the linker, dynamic linking of context dependent SVroutines is performed after elaboration. The foreign code calls the SV simulator function svcLocateTF to obtain a pointer to the appropriate entry point:

```
/* Template for an exported context task or function */
typedef int (*svcContextTF)(handle pli_inst,...);

/* Locater function */
svcContextTF svcLocateTF(handle pli_inst,int capabilities,const char *tf_name,...);
```

The extra arguments to svcLocateTF are the types of the arguments and return the caller is expecting to use (as specified in the next section), and as with external binders the capabilities argument is reserved for future modification of that interface. The following example shows a simple dynamic binding.

```
module cboundary;
    task munge(input d);
        int d;
        ....
    endtask
    ...
endmodule

export static cboundary::munge; // create a static C entry point for task foo in module cboundary
```

The corresponding C code could be:

```
static svcContextTF cboundary_munge;
static handle cboundary_munge_pli;
...
/* find instance */
cboundary_pli = findInst("top.cboundary");

/* get task pointer */
cboundary_munge = svcLocateTF(cboundary_pli, 0,
                                        "munge",   /* function/task name */
                                        0,         /* void return */
                                        "int"      /* argument types */,
                                        0);
/* call SV task */
(*cboundary_munge)(cboundary_pli,0xFF);
```

The final argument to svcLocateTF can be “...” instead of 0 if the exported function is called with different argument lists. The return value from svcLocateTF will be null if the simulator cannot support or cannot find a matching task or function.

Queueable Calls

Only tasks can be exported with the “queueable” attribute. A call to such a task from C will execute in zero time and return a handle which can be queried for completion status. A task exported without the queuable attribute is not allowed to consume time, doing so would be a runtime error. The handle returned can be queried with the following routine:

```
bool svcCallDone(handle);
```

This call will return false until the task is finished, and then will return true once, the result from further calls is undefined.

Queueable tasks cannot be called across operating system threads unless control has been passed to foreign code, it is also illegal to make simultaneous calls into the SV simulator from foreign code, doing so will cause a runtime error.

Since tasks (in 3.0 and before) use copy-in/copy-out semantics it is easy for the simulator to copy all the arguments and create a new process to handle the call, since 3.0 introduces dynamic processes this is only a minor extension.This functionality does not require any multi-threading in the simulator kernel. The restriction that the task cannot be called across operating system threads while kernel is active makes the mechanism MT safe. In particular, a multi-threaded SystemC program running concurrently with SV would only be able to call SV tasks one-at-a-time, and only when SV has passed it control (by calling a foreign function).

Data Passing between SV and C

The sections above describe how exported and external routines are linked. The passing of data on the calls is either direct with C pass-by-value semantics (in registers or on the stack), indirect through pointers for C data types, or through an abstract interface for simulator dependent data types (packed and 4-state).

Argument Specification

The strings specifying argument types handed to the binder functions above are in a simple canonical form which includes the language and type using the form "!"<attributes>":"<type declararion> and the argument name if it is available (between “$”s[1]). Attributes are single letters:

s	SystemVerilog type
a	Abstract access only
c	Copy back, implies argument is a pointer

The default interface mode is “C”, so a simple call to C will have straightforward definitions:

```
extern context void my_write(int,const char *,int);
```

1. This required for declaring arguments like function pointers, where the type’s location in the string is non-obvious e.g.: “int (* $$)(...)” - argument is a pointer to a function returning an “int”.

Would give the arguments:

> “int $$”, “const char *$$”, “int $$” and return “void”

A more complex call with 4-state (Verilog) types would have more annotation, e.g.:

```
packed struct my_struct {
    logic l1[7:0];
    logic l2[7:0];
};
extern context logic my_func(my_struct &data[13:0]);
```

Gives:

> “!sa:packed struct #my_struct# {logic #l1#[7:0];logic #l2#[7:0];} &$data$[13:0]”
> and return "!s:logic $$"

Intermediate type names and field names are provided quoted by “#”.

It is not required that the string arguments handed to the binder functions be permanently allocated.

An argument marked “!sa:” will be handled by an abstract interface from C/C++. The run-time value that is passed for an object handled by abstract access is a VPI handle for the object, other data types are passed according to the C calling standard (as are return values). The VPI handle for a packed object can be viewed as a handle to a vector, the size of which can be calculated from the type string.

The canonical form of the argument type should be sufficient information to calculate routine names generated by C++ mangling, the code for that and the extra functionality to load/find non-context dependent routines would only need to be written once per C++ compiler - a g++ version could be added in an appendix as an example. The locator routine could dynamically load libraries if required.

The “!a:” attribute allows for the case where a simulator may have optimized C type data and it no longer exists in a directly accessible form.

Note: the binding mechanism is not necessarily dependent on PLI/VPI. Only context calls and calls with abstract types make any use of PLI/VPI.

Pass-by-Value vs Pass-by-Reference

Arguments that are of SystemVerilog type can be passed by value or reference. Pass-by-reference uses the VPI handle mechanism mentioned above, passing by value causes a C compatible copy of the data to be generated on the stack, 4-state data is converted to arrays of the svcInteger type (described in the next section), and bit vectors use arrays of unsigned int. For example the following SystemVerilog struct:

```
struct mxd {
    int         i1;
    logic [4:0] lgc;
    real        r1;
```

```
    bit  [63:0]  bit1;
};
```

Is converted to the this C struct when passed as an argument:

```
typedef struct {
    int             i1;
    svcInteger      lgc[1]; /* use only 5 least significant bits */
    double          r1;
    unsigned int    bit1[2]; /* use two 32 bit words */
} mxd;
```

The layout of the C structure can be calculated from the type information passed to binding routines.Packed structs and unions are passed as simple arrays of *svcInteger* and *unsigned int*. The difference in the exern declaration for pass-by-value rather than pass-by-reference is the presence of the '&' in the argument:

```
extern mxd_func(mxd &data_in); // pass by reference

extern mxd_func(mxd data_in); // pass by value
```

Using '*' which indicates direct pointer access is only allowed for C compatible data types, using it with any other type is illegal.

Arguments and Direction

Arguments which are normally pass-by-value can be qualified as *input*, *inout* or *output*. The use of *inout* and *output* forces copy-back (pointer) mode for extern calls since C does not directly support the Verilog copy-back mechanism. Similarly tasks called from C get passed a pointer to the data location for *inout* and *output* arguments. For example the following are matching extern and C definitions:

SystemVerilog:

```
struct my_strct {...};
extern void c_task_a(input int, input my_strct);
extern void c_task_b(input int, inout my_strct);
```

C:

```
typedef struct {...} my_strct;
void c_task_a(int in, my_strct work_tmp);
void c_task_b(int in, my_strct *work_tmp); /* type will be marked !c: */
```

Type Promotion

Arguments passed by value also follow the C promotion rules: integral types are promoted to int if

smaller, and shortreal (float) is promoted to double.

Routine Return Values

Return values can be for packed abstract types because it is up to the simulator to copy the returned value to its internal representation of the type, but the type in C is always a 2-state or 4-state vector of equivalent size. C routines that return a value for an abstract type use an array of the following struct:

```
typedef struct {
    unsigned int data;
    unsigned int control;
} svcInteger;
```

Each bit in the data field represents either 0 or 1 if it's control bit is zero and Z or X if its control bit is one. Returning multiword values requires encapsulating an array of these structs in another struct so as in the following example:

SystemVerilog code:

```
extern integer [63:0] bus_val();
```

C code:

```
typedef struct {
    svcInteger v32[2];
} vec64;

vec64 bus_val() {
    vec64 ret_val;

    ret_val.v32[0].data = C_bus64_lo32();
    ret_val.v32[1].data = C_bus64_hi32();

    ret_val.v32[0].control = ret_val.v32[0].control = 0; // never Z/X

    return ret_val;
}
```

The zeroeth element of the array returned contains the least significant bits of the value. 2-state data is returned in a similar fashion using *unsigned int*:

SystemVerilog code:

```
extern bit[63:0] bus_val();
```

C code:

```
typedef struct {
    unsigned int v32[2];
} int64;

int64 bus_val() {
    int64 ret_val;

    ret_val.v32[0] = C_bus64_lo32();
    ret_val.v32[1] = C_bus64_hi32();

    return ret_val;
}
```

Unpacked abstract types are returned using the same mechanism that pass-by-value arguments use to coerce data, but in reverse.