SystemVerilog Interfacing to Foreign Languages

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).

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) *) ?
                        function_decl | task fname ( extern_func_args ? )
                        “C name” ?;
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 the using write is not going to feedback into the simulation and it is to replace a differently name SV routine one could use:

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

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 {
```

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

```
                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]) {
            assert(context_version & SVC_CONTEXT_LINE);
            fprintf(stderr,  ”%s:%d(%d) / %s - Error data out of range! = %d\n”
                                p_context->cvu.call_inst.file_name,
                                p_context->cvu.call_inst.line_number,
                                p_context->cvu.call_inst.call_num,
                                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 */
```

1. Defined in header file(s)

```
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.
}

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;
```

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

```
        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 (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. 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.

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) * ?
                               [module spec::]fname “C name” ?;
attribute      ::=   static
module spec    ::=   (modulename|interfacename) (::(modulename|interfacename))* ?[1]
```

As with external declarartions the SV name may not be a valid name so a valid C name can be provided. 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.

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,...);
```

1. top.foo would refer to a module foo declared inside module top

```
/* 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_munge_pli = findInst("top.cboundary");

/* get task pointer */
cboundary_munge = svcLocateTF(cboundary_munge_pli, 0,
                                        0,      /* void return */
                                        "foo",
                                        "int"   /* argument types */,
                                        0);
/* call SV task */
(*cboundary_munge)(cboundary_munge_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.

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 (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 declarartion> and the argument name if it is available (between “$”s[1]). Attributes are single letters:

s	SystemVerilog
a	Abstract access only

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);
```

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:

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

Intermediate type names are provided quoted by “#”.

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

An argument marked “!s:” will be handled by an abstract interface from C/C++. The run-time value for an object handled by abstract access is an opaque pointer (void *), other data types are passed according to the C calling standard (as are return values).

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”.