A Simple, Scalable and Correct Method for Constructing Verilog Filelists

Karl W. Pfalzer (kpfalzer@gmail.com)

Introduction

This document presents a simple method for constructing Verilog filelists, aka dot-f files.
Before reading further, the reader should sit back and think a little about the similarities between Verilog source files and C/C++ (programming language) files.
Both have source code, include and define components.
When processing either, there is the need to specify how include files are found and define values… well, defined.
For those more intimately familiar with C/C++ programming/processing, think about the first step of the compilation process: the good ole C preprocessor, **cpp** (link).

Requirements

Hopefully, upon pondering the similarities mentioned above, you have a sense of where we might be headed now. But, before we pose the (obvious) solution, let's make sure it fits the problem/*givens*/requirements.

1. Verilog source file coding conventions, ideally, follow a simple, consistent rule (*best-practice*): a file called `modx.v` contains a single module definitions for module `modx`.
2. Thus, for a complex design, we would expect to see some set of modules, `m1, m2, …, mn`, consistently defined in files: `m1.v, m2.v, …, mn.v`.
3. And, let's say the (main) top module of this complex design is `mtop` (in `mtop.v`).
4. For sake of (simple) example, let's also assume any include files local to this design are contained in the same source directory as the module definitions themselves. The include file naming convention is to use suffix `.vh`.
5. And, lets assume the directory name containing the source and include files is called **/path1/rtl**. Thus, all the source and include files are in `/path1/rtl/*.{v,vh}`.
6. With these givens/assumptions (above), the simple Verilog command line to compile (for simulation) the top-level design mtop would pass the following (standard) command line options to the simulator:
   ```
   compile +incdir+/path1/rtl -y /path1/rtl +libext+.v
   /path1/rtl/mtop.v
   ```
7. Or, we could create the (familiar) dot-f, **mtop.f** as:
   ```
   +incdir+/path1/rtl
   ```

```
-y /path1/rtl +libext+.v
/path1/rtl/mtop.v
```

8. Then run:
   compile `-f /path1/rtl/mtop.f`

9. Let's assume we have a larger system (**sysx**) now comprised of several `ip1, ip2, …, ipn`, and local system integration files: `x1.v, x2.v, …, xn.v` and toplevel `xtop.v`.

10. Assume all the ip and system used the best-practice convention, so the construction of a dot-f (**sysx.f**) for the (entire) system would resemble:
```
-f /path1/rtl/mtop.f
-f /path2/rtl/ntop.f
…
-f /pathn/rtl/ztop.f
+incdir+/pathx/rtl
-y /pathx/rtl +libext+.v
/pathx/rtl/xtop.v
```

11. You may have noticed that absolute (starting with /) pathnames have been used all along here. While this makes for a great example, doesn't it greatly affect portability?
    In reality, all these files are revision controlled, so we can't possibly hardcode absolute pathnames, since they would not be portable.

12. Well, the portability problem is easily solved. Let's just use some variable to denote the actual root pathname: i.e., that part which varies depending on location.

13. In our example(s) above, we can easily see replacing at least the /pathi prefix with a variable, as in: reference to **ip1** collateral becomes `IP1/rtl/mtop.f` (or even `IP1/mtop.f`).

14. Then, we would have **mtop.f** resemble:
```
+incdir+IP1/rtl
-y IP1/rtl +libext+.v
IP1/rtl/mtop.v
```

15. But, if we try to compile that (new) **mtop.f**, we would get an error like:
    `IP1/rtl/mtop.v: file not found`.

16. So, clearly we need to expand the value of IP1 before the compiler/simulator sees the file.

17. One might immediately jump to a solution of parsing dot-f files and doing some simple text substitution. Great… but, that does not solve all the requirements. Read on!

18. In complex designs, there is typically a set of re-used, well-architected components: synchronizers, fifos, regfiles, …

19. Again, let's assume these common library components reside in a (revision controlled) directory: `/cmnlib/rtl/{fifo,sync,rf}.v` …, with the typical dot-f file `common.f`:
```
+incdir+CMNLIB/rtl
-y CMNLIB/rtl +libext+.v
```

20. And, it is/was likely the previous IP were using these, so their dot-f file actually resembled:
    //mtop.f
    ```
    +incdir+IP1/rtl
    -y IP1/rtl +libext+.v
    -f CMNLIB/rtl/common.f
    IP1/rtl/mtop.v
    ```

    //ntop.f
    ```
    +incdir+IP2/rtl
    -y IP2/rtl +libext+.v
    -f CMNLIB/rtl/common.f
    IP2/rtl/ntop.v
    ```
    …
21. Looks great. But, what happens if we now try to compile the entire system (ala **sysx.f**)?
    We would certainly get lots of warning messages about `CMNLIB/*.v` module(s) being redefined.
    Sure, the messages would be benign (since same source); but, would only compound as more IP are added (using same common lib).
22. So, lets add a requirement that some filelist processor is to mitigate the warning messages due to multiple use of (common, or any) libraries in a reasonable manner.
    Think about this requirement a bit: esp. in context of C/C++ wrt. include files. Typically, one uses a define, `ifndef/endif` guard to disable the body of an include file from being processed if already done so in the (current) compilation scope.
    Hmmm, sounds like a similar problem; and one with a solution dealt with by the C-preprocessor.
23. In fact, nowadays, C/C++ developers use a simpler method (for handling re-inclusion of same include file) with a `#pragma once` directive (link).
24. Let's add one more requirement to the mix: filelist processor should thoroughly expand nested dot-f files.
    Why?
    Readability and debugging!
    The output (of the filelist processor) should have no **-f** statements: simply inline (recursively) the contents.
    The flattening will also make the output more friendly to the variety of implementation-related compilers (like lint, cdc, synthesis, …) which may not always be able to digest **-f** statements.

A simple solution

As mentioned earlier, if you frame the Verilog filelist processing problem in the very similar light of processing C/C++ source code: you realize the latter solution/processor would work perfectly for this problem.

So, the simple solution is to construct Verilog filelists in a manner in which intent is still clear/succinct, but with a slight syntax change to take advantage of immediate C-preprocessor capabilities (builtin). No need to create scriptware to do what **cpp** (link) already does! And cpp is already available/installed on all common *nix (and macos) platforms: so no portability issues.

OK. Sounds reasonable. But, how do we actually do it?

1. Instead of using -f statements (in a dot-f), use `#include`
2. Still use the variable to qualify the root/pathname of all dot-f references (i.e., `IP1, IP2, CMNLIB, …`). **cpp** can easily do macro/variable substitution.
3. Solve the multiple include (of same file issue) with `define, ifndef/endif` guard around the entire dot-f file; or, use the simpler `#pragma once` directive (as first line).
4. Use other Verilog command line options: `+define+, +incdir+, +option+, -y, -v` as needed. But: remember, no **-f**!
5. Comment the dot-f thoroughly now. Since we are using **cpp**, `/*block style*/` and `//line-style` comments can be used throughout.
6. And, best of all, since we using **cpp**, you can also use conditional blocks (in the dot-f), such as exposing different implementations for simulation versus synthesis (i.e., `#ifdef SIMULATION #else … #endif`).

Let's see it in action.

Example

Back to our examples above, we will rewrite using the **cpp** style.

//common.f
```
#pragma once
+incdir+CMNLIB/rtl
-y CMNLIB/rtl +libext+.v
```

//mtop.f
```
#pragma once
+incdir+IP1/rtl
-y IP1/rtl +libext+.v
// Use qpath(a,b) macro to create quoted path "a/b" for #include.
#include qpath(CMNLIB,rtl/common.f)
IP1/rtl/mtop.v
```

//ntop.f
```
#pragma once
+incdir+IP2/rtl
-y IP2/rtl +libext+.v
```

```
#include qpath(CMNLIB,rtl/common.f)
IP2/rtl/ntop.v
```

//sysx.f

```
#pragma once
#include qpath(CMNLIB,rtl/common.f)
#include qpath(IP1,rtl/mtop.f)
#include qpath(IP2,rtl/ntop.f)
+incdir+SYSX/rtl
-y SYSX/rtl +libext+.v
SYSX/rtl/xtop.v
```

Then, to process **sysx.f**, for example:

```
./cpp4vflist \
     -DCMNLIB=$PWD/cmnlib \
     -DIP1=$PWD/ip1 \
     -DIP2=$PWD/ip2 \
     -DSYSX=$PWD/sysx \
     sysx/rtl/sysx.f
```

Voila!

Consider this (consistent) solution also makes it easy to capture a *bill-of-materials* (BOM): i.e., the exact versions of IP/libs being used.

To do so, simply have some project setup file define the absolute pathnames of the IP/libs, and embed the version in the pathname, such as:

```
setenv CMNLIB /path/v1.3/cmnlib
setenv IP1 /path/v2.4/ip1
…
```

Then, in the (above) invocation of **cpp4vflist**, use the `${CMNLIB}` (variable reference) style to set the -D values.

Of course, there are always other (more complicated) means to accomplish the same BOM requirement too.

Have fun!

More details

Here is the simplest **cpp4vflist** script:

```
#!/bin/csh -f
# Need to ignore stderr unless its fatal.
# Ignore exit code since it can be != 0 (even if ok).
/bin/bash -c "/bin/cpp -include ./cpp4vflist.h -x c -P ${*} 2>
>(fgrep -i 'fatal error')" || exit 0
```

Here is the `cpp4vflist.h` macro definition file (used by **cpp4vflist**) (esp: **qpath**):

```
#define _join(_a,_b) _a##/##_b
#define _si(_x) #_x
#define _s(_x) _si(_x)
#define qpath(_a,_b) _s(_join(_a,_b))
```