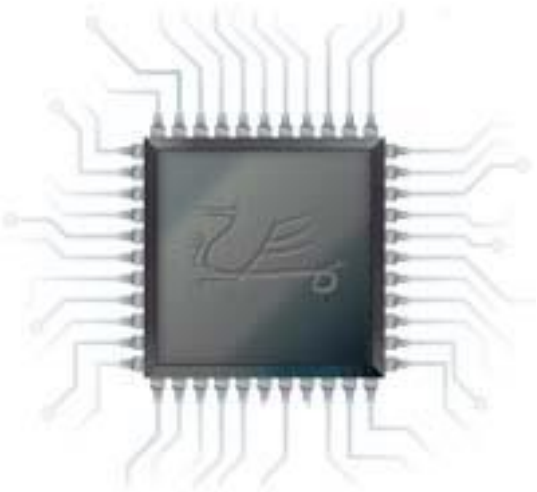
Logic Development and ‘make’



Simon Southwell

15 August 2024

Preface

This document is a PDF version of an article written in August 2024 and uploaded to LinkedIn, on the use of make in the development of Logic designs.

Simon Southwell
Cambridge, UK
15th August 2024

Contents

INTRODUCTION **4**

THE BASICS OF MAKE **5**

VARIABLES AND FUNCTIONS 7
VARIABLE SUBSTITUTION 8
CONDITIONALS 9
OUTPUT CONTROL 9
RULE ORDERING 10
OVERRIDING VARIABLES 12
INCLUSION 12

LOGIC SIMULATION AND MAKE **13**

LOGIC SYNTHESIS AND MAKE **16**

CONCLUSIONS **19**

Introduction

The ‘make’ utility has been around a long time now (since 1976), and my relationship with it isn’t much younger. Originally developed for Unix, the Linux distros include a GNU version, gmake, though this is usually aliased to ‘make’ and it is the gmake version I want to talk about here.

Originally, make was intended for automatically compiling code, particularly C, ensuring that the correct object files and executables are built when one or more source files have been updated. Of course, if everything is built regardless of whether source files have changed or not, then this would guarantee that the build outputs will be up-to-date, so why not just write a shell script to do just this? If a compilation is for a few files with, say, a single output, then this is a valid method, but in a real development environment building everything may take a long time, especially if, for example, an embedded system includes a modified Linux kernel as part of the build. Changing a comment in a single file might then kick off a big build at the next compilation. So, make can help us just re-compile (and link) those files that are affected (‘dependent’) on source files that have been updated.

The use of make is not restricted to just compiling software—indeed, this document aims to show just this in terms of logic development. However, the principles are the same whatever we are building, whether software, logic simulation or synthesis for ASICs or FPGAs. In order for make to build only the correct files, it needs to know two things; how to build the output files and what the dependencies are for each output. Note that outputs can be dependent on other outputs, for example an executable can be dependent on object files, which are dependent on source files. Thus, make needs a set of rules for building and a list of dependencies.

The antiquity of make does make it rather esoteric and not obvious how to use for someone new to its nuances. So, to start this document, we will look at make (actually gmake) in terms of compiling software, its original use, in order to see how to use its basic features before tackling how to adapt this for logic simulation and logic synthesis. It may seem a while before we start talking about logic development, but we will get there, and (hopefully) be prepared when we do, so that this step is straight forward. I’m reminded of an anecdote in one of Isaac Asimov’s essays on science, where he explained he’d put up an essay for review and built the story of the history of all the science involved to explain the topic at hand. On return of the marked-up essay from the reviewer he saw, about three quarters of the way in, that the reviewer had written ‘*at last!*’ next to the first mention of the subject of the essay.

He justified this as it is important to have the right foundations laid before jumping into the topic at hand and to perfectly be ready for this step.

The Basics of make

The make executable is a command line utility with various options available, but the main input is a 'make file'. It is in the make file that we construct the rules for specifying how to build and what the dependencies are. Open a terminal on a Linux machine (or, say, mingw-w64/MSYS on Windows) and type `make -h` to get a list of options. A `-f <file>` option (or `--file=<file>`, or `--makefile=<file>`) specifies the make file to use. However, if the directory where `make` is invoked contains a file called `makefile` or `Makefile`, then it will use this by default, and the `-f` option need not be used. Another useful option is `-C <dir>` (I'll leave you to look up any alternative option names from now on). This specifies a directory to change to before executing a make file (in that directory). This is useful if a build requires sub-directories, which have their own make builds, to be built first, under the control of a top level make file. An example of this usage is in my PCIe model, which uses the `makefile` of the VProc virtual processor to compile the software, whilst having its own `makefile` for building its local files.

So, let's put together a simple make file and look at each element:

```
file.exe: file1.c file2.c file3.c
<tab>gcc file1.c file2.c file3.c -o file.exe
```

Here we have a single rule with a 'target' output of `file.exe`. This output is dependent on three source files, `file1.c`, `file2.c` and `file3.c`, as listed after the colon for the target. Underneath this line is the command to build the output, which is just the command we would type in a terminal to build the executable. Note the '<tab>' before the command. This is compulsory before each command, and it still catches me out. Since it's a white space (like space) it can be hard to spot that a tab is missing and usually an unhelpful message is printed, such as 'missing separator'. If your editor has the means to display whitespaces, it is worth enabling these whilst developing make files. From now on, the examples will assume a tab, and not show it.

With a file called `makefile`, with the above rule, in a directory with the three dependency source files, typing `make` in that directory will execute the command and a `file.exe` will appear in the directory. Typing `make` again will produce a message "`nothing to be done for file.exe`", and the command is not executed. It knows

this because the `file.exe` exists *and* it has a timestamp that is newer than any of the dependency files. If any of the dependency files are modified, so their timestamp becomes newer than the output, then executing `make` once more will kick off running the command. Hopefully you can now see how, with the right rules, `make` will only build files that require building due to modifications to dependencies.

Some of you may have spotted that the dependency list of source files is duplicated as arguments to the `gcc` command. This is undesirable since, when developing code, it is likely that new files may be added, or files renamed etc., and the list will need to be modified in multiple places. There are various ways of solving this, but one way is the use of 'automatic variables'. These are variables (in `make` these start with a `$` to reference) that are automatically set by `make`. Below is the make rule from before, modified to use some automatic variables:

```
file.exe: file1.c file2.c file3.c
   gcc $^ -o $@
```

Here the rule replaces the list of source files in the command with `$^` And the name of the output file with `$@`. The `$^` will have been set by `make` to contain all the dependency files and `$@` will have been set to contain the target name. There are others, such as `$<`, which is the first dependency, useful if a rule has a single C source file, but is dependent also on header files (e.g., `file1.h`), when only the C file should be listed in the command. There are, of course, many other automatic variables, and a list can be found here.

Another approach to avoiding file list duplication is the use of wildcards:

```
file.exe: *.c
   gcc $^ -o $@
```

Here, the `*.c` will match any file with that suffix in the directory and make will expand this to a list (and set the `$^` automatic variable to that list). Personally, I find this method troublesome. I have, when experimenting with code, quickly renamed a file from, say, `file.c` to `file1_old.c`, and created a hacked `file1.c` to quickly try something out. Some of you may spot what will go wrong here already. Of course, with a wildcard, it will compile `file1.c` and `file1_old.c` and errors result with clashing function names etc. But perhaps I shouldn't be renaming files in the first place, and the wildcard is a valid method.

Up until now we have compiled directly from C source code to an executable, compiling and linking in a single step, but it is often prudent to separate compilation from linkage. Below is shown this separation and also serves to demonstrate the dependency hierarchy mentioned before:

```
file.exe: *.o
   gcc $^ -o $@

*.o: *.c
   gcc -c $^
```

Here we have two rules with the `file.exe` target dependent on matched object files (`*.o`). Initially these don't exist, and we provide a second rule for creating the object files (`gcc -c` compiles to object files). When run, `make` sees the `file.exe` dependencies, and looks for a rule to construct these. The second rule matches the source files with a wildcard and matches its object output with the *.o, expanding to a rule for each individual C source file. For the first rule, then, the list of objects created at the expansion now becomes the list of dependencies.

Variables and Functions

We can improve our make files further with the use of user defined variables and the provided functions. We've already encountered automatic variables, but we can set our own variables and perform transformations on them with functions.

Defining a variable is as simple `VAR = something`. To use the variable, we make a reference as `$(VAR)`. One can also use `${VAR}`, with braces and parentheses interchangeable in `gmake`. We will stick to parentheses. Let's modify our make file again:

```
CFILES = $(wildcard *.c)
file.exe: *.o
   gcc $^ -o $@

*.o: $(CFILES)
   gcc -c $^
```

Here we have set a variable, `CFILES`, using a function called `wildcard`. There are a couple of other useful variable assignment operators other than '='. If `=?` Is used, then the variable will only be set if it is not set already. As we will see later, variables

can be set from elsewhere. When assigning a variable, any references to other variables are expanded. If those variables reference other variables, then these are expanded first and so on, recursively. If this isn't the desired operation, then some ordering can be enforced with variables only expanded to the point where the assignment takes place. So, for example, if a variable `TMP` is set to `file1`, then `FILES := $(TMP)`, and `TMP` then set to `file2`, the variable `FILES` will be `file1`, and `TMP` will be `file2`. If '=' was used, both `FILES` and `TMP` would be `file2`. There is also an 'immediately expanded' assignment (`:::=`) but I've rarely seen it used and perhaps the make file is then getting too complicated. One last assignment operator to mention is +=. This appends the value to the end of the variable if it has already been assigned, or just creates it if not. So, `FILES = file1`, followed by `FILES += file2`, results in `FILES` being set to "`file1 file2`"

The `gmake` functions are all formatted as `$(<funcname> <args>)`. Here, the `wildcard` function takes a wildcard argument (`*.c`) and expands this to a list to be assigned to the variable. We can now use the `CFILES` variable for the `*.o` dependencies. This may seem like we haven't gained much here but, as we will see, having a variable with the list of source files opens up other opportunities.

Variable Substitution

Having defined a variable for the source files we can actually use this to define the list of object files using substitution

```
CFILES = $(wildcard *.c)
$(info $(CFILES))

file.exe: $(CFILES:%.c=%.o)
   $(CC) $^ -o $@
```

In the `file.exe` rule, the dependencies are now derived from the `CFILES` variable of the source code, but with some strange additional characters. After the variable name and a colon is a pattern matching and substitution specification. The `%.c` matches everything in the variable's list of files that end in `.c`, and substitutes the `.c` with `.o`, as specified by the `%.o`. The advantage of this is that a single source file list can be used for generating a list of intermediary files, and any changes to the list of source files propagates through. But wait! There is now no rule on how to generate the object files.

Well, the make file still works correctly. This is because make has some built in knowledge of how to compile C and C++ code and when only requiring vanilla

compilation these implicit rules can be invoked. I won't expand further as, basically, don't. It's confusing and lacks flexibility. I only mention it as you may encounter make files with their use and wonder what's going on.

Also shown in the above make file script is an 'implicit variable', `CC`, used for the `file.exe` rule's command. A collection of these preset variables exists (with a full list here) with default values—with `CC` being set to `cc` by default. They can, of course, be overridden by setting in the make file, in order to control things like implicit rules. Also shown in the above make file is another useful function 'info'. This will display to the console its arguments, as expanded by make. This is very useful in debugging make files. A section in the make manual on all the functions available can be found here.

Conditionals

The gmake utility also allows conditions with an if-then-else type syntax. There is an 'ifeq' and 'ifneq' for equal and not equal comparisons respectively (as well as others).

```
OSTYPE=$(shell uname)

ifeq ($(OSTYPE), Linux)
   PLILIB=libmtipli.so
else
   PLILIB=mtipli.dll
endif
```

Here a 'shell' function is used to run a command in a shell, in this case `uname`, and the returned result assigned to a variable `OSTYPE`. This variable is then compared with a fixed string 'Linux' and if equal a new variable, `PLILIB`, is assigned the name of a library file. If it is not equal, it is assigned a different library name. The PLI library of the Questa logic simulator has different names on Linux and Windows platforms, and the conditional is used to select the correct one. Note that the `shell` function is used with the assumption that a Linux like environment is where `make` has been run, so that `uname` is a valid command, and in Windows I have assumed running on mingw-64/MSYS2. The Windows Subsystem for Linux (WSL) environment is also likely to work.

Output Control

When make is run, by default, it prints out every command that it runs. So, for the make files we have been looking at, the following is printed to the terminal.

```
simon@Rioja MINGW64 ~/tmp/make
$ make
cc    -c -o file1.o file1.c
cc    -c -o file2.o file2.c
cc    -c -o file3.o file3.c
cc file1.o file2.o file3.o  -o file.exe
```

This is useful when developing a make file, but with large compilations, involving multiple make files, a very verbose output can hide things like warning which need to be addressed. There are a couple of ways of controlling output from a make build. Firstly, adding a target of `.SILENT:` (with no command after) will stop the commands being echoed to the terminal. It will silence implicit rule outputs as well as the make file rule commands, so is rather global. For more refinement, before each command the @ character can be placed to indicate that the command should not be echoed. This gives finer control and is useful when debugging to show only the commands of interest (by removing the @). The disadvantage is that implicit commands will still be echoed—another reason not to use them.

Rule Ordering

Make files can seem like scripts sometimes, with variables set and rules defined, based on those variables, as we have seen. But actually, there is no flow from start to finish as such and make loops expanding and resolving things until either it can execute commands or finds it doesn't have enough information and gives an error. The command execution flow is based on the dependencies, and it doesn't matter if a rule has a dependency on another rule's outputs, with that rule lower in the make file. Variables can also be set using variables that are defined further down the file, unless using `::=`, as we've seen. It might not be a good thing to randomly place rules and variable settings in the make file, and a structured approach is better for maintenance, but make does not care.

There is a particular exception to this. The first rule that `make` encounters is the default build target. In all the examples so far, I have assumed one simply types `make` in a terminal at the directory where the `makefile` exists. Since the examples have always had the `file.exe` target as the first rule, it attempts to build this. Where there are multiple target rules in a make file a particular target can be specified on the command line. For example, `make file2.o` will just build `file2.o`, even though this is an implicit rule.

A *convention* (not a requirement) is to have a first rule of 'all' and make this dependent on all the final target outputs (in our case this is just `file.exe`, but

perhaps a make file has multiple executables and/or libraries to build). Our make file is modified as shown below:

```
CFILES   = $(wildcard *.c)
EXECFILE =file.exe

.PHONY: all, clean
all: $(EXECFILE)

$(EXECFILE): $(CFILES:%.c=%.o)
    @$(CC) $^ -o $@

clean:
    @rm -rf $(CFILES:%.c=%.o) $(EXECFILE)
```

So now either `make` or `make all` will build everything listed as a dependency of `all`. There's also a couple of other changes made to the `makefile`. Firstly, I defined a new variable for the executable file target, `EXECFILE`. This is because I want to use it as a dependency for all *and* as a target for its build rule. This is good practice. If ever you ae typing some filename or program or whatever, more than once, then it should placed in a variable and that referenced in the rules. I try to make sure no rule has any direct reference to a file or program. The second change was a target of `.PHONY` with `all` (and `clean`) as a dependency. This now looks like the first target rule but is a special target. Since the make file is not building an output called 'all' if, for some reason, a file (or directory) was created in the directory with the `makefile`, and its date is newer than the real targets (`file.exe` in this case), then make won't build as it thinks everything is up to date. By declaring 'all' as phony, `make` knows to not check for files of that name and will always resolve that rule.

Finally, a new rule 'clean' was added. Again, this is a convention and not a requirement, but most `makefiles` you will come across will have one or more targets for cleaning up intermediate files. The point of `make` is to resolve which build files are out of date with respect to their dependencies and, all things being equal, nothing will go wrong. However, things do go wrong. For example, when developing a make file, the dependencies might be in error, with missing header file, for example. When debugging, starting from a clean initial state eases finding the faults. Another scenario I have encountered is where an intermediate file has been updated outside of the make process, without being rebuilt from modified source code (e.g., `touch file4.o`). My own projects, such as the VProc virtual processor, needs to be tested on all supported platforms, both OS (Linux and Windows) and the different

simulators. The regression tests that run all the tests for each OS/Simulator combination must start from a clean initial state as the generated files are not compatible with each other, even though source code has not been modified between tests. Therefore, a target of 'clean' is usually added, with one or more commands to remove all intermediate files that are generated during a build.

Overriding Variables

We've seen how to set variables and make the scripts more organised and maintainable with proper use, avoiding things like repetition. In addition, these variables can be set on the command line, either setting them if not set in the make file or overriding the value set in the make file. This gives us a way for a simple user interface, either on the command line, or when calling from a script or other make file.

In our example make files, we have two variables, `CFILES` and `EXECFILE`. One or both can be overridden from the command line, which might be useful to, say, change the output executable name if compiling just as a trial run. E.g.:

```
make EXECFILE=trial.exe
```

With careful choice of variables, make files can be made more generic for re-use in a variety of similar compile situations, with different lists of source files, or requiring different compiler flags for the builds etc.

Inclusion

One last feature of make and the make files that is very useful in organising make file scripts and making them generic is the use of '`include <filename>`'. This command can be used to include another file in a make file script at the point in the file with the include command, as if it was typed in at that point.

This is very handy when various flavours of compilation are needed which are all basically the same, but have different compile flags, or preprocessor definitions for compile time options, etc. As was mentioned before, the VProc virtual processor supports two OS platforms and six (with an optional seventh) logic simulators. The make files for each are very similar but also annoyingly different. There are make files for each of the different simulators in the test directory and a `makefile.common`, which is included by each of the specific make files. The common make file has all the compilation rules for the PLI co-simulation software and user code, but is highly configurable, which uses a set of variables that are meant to be set externally by the including make file (or left unset). The specific make files, for compiling the C/C++

PLI code, then just set the variables for their specific requirements and include `makefile.common`. This makes them smaller and agnostic to the details of the PLI software compilation which, if updated, automatically updates the specific make files. The rest of the specific make files are then just involved with the particular logic simulation build rules, which vary from tool to tool.

This, then, might be a good point to end the basic discussion and move to logic development use of make, starting with logic simulation (at last!).

Logic Simulation and Make

We've spent a bit of time looking at `make` in terms of compiling software, whereas this document is meant to be about `make` and logic development. Well, firstly, if you follow my article publications on LinkedIn you'll know that I'm very much a champion of co-simulation of co-design of hardware and software, and am involved in co-simulation projects, such as VProc, co-simulating software models for such things as PCIe, USB, RISC-V and TcpIp, and with adding co-simulation capabilities to the OSVVM methodology libraries. So, for me, software and logic go hand in hand to make up a complete system or test environment. So, everything we've looked at so far is relevant, and I've made a passing mention of how the make files for VProc are organised, hinting at compiling software for a logic simulator.

In a classical logic simulation, the steps to running a simulation are not so far different from compiling C/C++ code. A set of source files (Verilog, VHDL or other, possibly mixed), that make up the design under test and the test environment, are 'analysed', 'elaborated' and then executed. The steps may vary, with possibly some combined, but you get the idea. The HDL source file may be being developed and modified and, in a large design, may take some time to analyse and elaborate. If we can skip re-analysing files that are unchanged, then this is an optimisation, and `make` can be used for this, just as for software.

Many (usually commercial) EDA tools will start their manuals or user guides with an introduction to their tool via the graphical user interface (GUI). This is not unreasonable, and the tool may have a complete integrated development environment (IDE), to aid development and debugging. However, it is usually very deep in the reference manual before one finds the command line references and usually no 'quick start' guide on using these. You may ask are they needed when an advanced IDE is available. Well, it is very likely that on a project of any significant size, there will be a continuous integration/continuous deployment (CI/CD) process, using tools like Jenkins or TeamCity. This may require, for example, some or all regression

tests to be run and pass before any changes will be accepted into a main branch or release branch of the version control system. I have come across engineers who do not know how to use the batch commands of their development tools, but it is imperative to be familiar with these, so that support for CI/CD is possible. Using `make`, of course, along with the command line tools, makes CI/CD possible *and* efficient.

As has been mentioned before, simulator tools will have command line programs to analyse, elaborate and then simulate a testbench/DUT combo. The table below summarises some commands for various logic simulators, both commercial and open source.

Phase	Questa	Vivado	Icarus	Verilator	NVC	GHDL
analyse	`vlog/vcom`	`xvlog/xvhdl`	`iverilog`	`verilator`	`nvc -a`	`ghdl -a`
elaborate	`vsim`	`xelab`			`nvc -e`	`ghdl -e`
simulate		`xsim`	`vvp`	`<obj_dir>/<file>`	`nvc -r`	`ghdl -r`

Note that some of the simulators combine phases, such as analyse/elaborate, or elaborate/simulate. Verilator generates an executable which is then simply run as any normal program, and the Questa and Vivado simulators have a choice of analyser commands, depending on whether Verilog/SystemVeriilog or VHDL source code is being processed.

So now we can construct a make file that not only 'compiles' the source code, we can also run the simulation from `make`. The make file below is a fragment of VProc's test make file (`makefile.nvc`) for the NVC VHDL logic simulator.

```
# Common logic simulator flags
VPROC_TOP          = test

WAVEFILE           = waves.fst
WAVESAVEFILE       = $(WAVEFILE:%.fst=%.gtkw)

# Runtime flags for nvc
SIMFLAGS           = --ieee-warnings=off                    \
                     --format=fst --wave=$(WAVEFILE)        \
                     --load=$(VPROC_PLI)                    \
                     $(VPROC_TOP)

# --------- BUILD RULES ---------
all: vhdl

# Include common build rules
include makefile.common

# Analyse VHDL files
.PHONY: vhdl, run, rungui
vhdl: $(VPROC_PLI)
      @nvc --std=08 -a -f files_nvc.tcl -e $(VPROC_TOP)

# ------- EXECUTION RULES -------
run: vhdl
      @nvc -r $(SIMFLAGS)

rungui: vhdl
      @nvc -r  $(SIMFLAGS)
      @if [ -e $(WAVESAVEFILE) ]; then                      \
          gtkwave -A $(WAVEFILE);                           \
      else                                                  \
          gtkwave $(WAVEFILE);                              \
      fi
```

Note that the actual make file has some initialisations of variables to be used by the included `makefile.common`, which compiles the PLI co-simulation software and sets the variable `VPROC_PLI` to be the generated shared object (`VProc.so`). Using this make file, we can compile and run a simulation with `make -f makefile.nvc run`. Since the 'vhdl' target is a dependency of `run`, and `vhdl` itself has a dependency of `VProc.so` (via `VPROC_PLI`), then the PLI code will be compiled, followed by the NVC analyse command of the vhdl target. A TCL file lists all the files to be analysed by the `nvc -a` command, and the elaboration is done with the same command with the `-e` option, specifying the top level module. Finally, the command for the `run` target is executed, along with some defined flags in `SIMFLAGS` to run the simulation.

For the VProc make file an additional target of `rungui` is also added for convenience where, post simulation, GTKWave is run to display any generated wave data (`waves.fst`). The wave data is generated since `SIMFLAGS` had `--format=fst --wave=$(WAVEFILE)` options. An optimisation might be to separate out these flags into another variable, which is then referenced in `SIMFLAGS`, which would allow these

to be overridden on the command line set, say, to be blank, and not generate wave data, thus speeding up simulation—useful in regression testing. Note the 'if' of the second command for the `rungui` target. This is not a make file 'if'. All the commands are run in a shell, and so the scripting language of the shell is available as part of the commands, as if typing in a terminal. So, this 'if' is a straight BASH script command. The ifeq/ifneq, though, can be used in target commands, but they bracket commands, rather than are part of them. Useful, for example, to run one command if a variable is set one way, or a different command if not.

And that's all there is to it. The other simulators' make files have a very similar structure, with just the specific commands and flags for the particular tool. Verilator differs slightly in that it doesn't use the `VProc.so` PLI code but requires a static library to be linked. Fortunately, generating this is a step in compiling `VProc.so`, so it compiles to the sub-target of `libvproc.a` in place of `VProc.so`.

Logic Synthesis and Make

A synthesis flow takes a little more thought than a logic simulator, but the structure is the same. As a case study I want to look at a make file for Altera's Quartus Prime FPGA synthesis targeting a Cyclone V FPGA chip, with and ARM based 'hardware processor system' (HPS). This has multiple steps, with designs configured with "Platform Designer" which auto-generates infra-structure HDL to contain and connect user IP logic, before doing the synthesis in steps of analysis and synthesis, fitting, assembly and static timing.

For the auto-generation of HDL from platform designer, there is a `qsys-generate` command that takes a `.qsys` file (generated from the GUI), along with some other flags to generated the HDL. This step produces files about the configurations with the HPS, in particular a `.sopcinfo` file, and embedded software will need a header with information in it, for example regarding offsets in the memory map where logic resides. This is generated from the `.sopcinfo` file with a program `sopc-create-header-files.`

With Quartus, synthesising to a configuration file on the command line can be done in one of two ways. It can be done with a single program that does all steps (`quartus_sh`), or each step can be done individually with a series of programs (`quartus_map`, `quartus_fit`, `quartus_asm` and `quartus_sta`). The former method will do exactly what would happen if running compile from the GUI but lacks flexibility. The second method allows to selectively run each stage if necessary if, say, an intermediate stage failed, the issue fixed, and want to run from that point

onwards. It does, however, require specifying dependencies at each state, which can be difficult. Let's look at an example make file fragment:

```
TOPNAME    = top
SYSTOPNAME = $(TOPNAME)_system
HPSNAME    = hps_0
OPFILEDIR  = output_files
QPFFILE    = $(TOPNAME).qpf
QSYSFILE   = $(SYSTOPNAME).qsys
SOPCINFO   = $(SYSTOPNAME).sopcinfo
FPGATYPE   = "Cyclone V"
FPGAPART   = 5CSEBA6U23I7

.PHONY: map, fit, sta, asm, qsys, hps, check_timing, clean, synth
all: hps asm check_timing

map:  $(OPFILEDIR)/$(TOPNAME).map.rpt
fit:  $(OPFILEDIR)/$(TOPNAME).fit.rpt
sta:  $(OPFILEDIR)/$(TOPNAME).sta.rpt
asm:  $(OPFILEDIR)/$(TOPNAME).asm.rpt
qsys: $(SOPCINFO)
hps:  $(HPSNAME).h

$(SOPCINFO): $(HDLFILES)
	@qsys-generate $(QSYSFILE) --synthesis=VERILOG --output-directory=$(SYSTOPNAME) \
	                           --family=$(FPGATYPE) --part=$(FPGAPART)

$(HPSNAME).h: $(SOPCINFO)
	@bash -c "sopc-create-header-files $< --single $(HPSNAME).h --module $(HPSNAME)"

$(OPFILEDIR)/$(TOPNAME).map.rpt: $(SOPCINFO)
	@quartus_map $(TOPNAME).qpf --write_settings_files=off

$(OPFILEDIR)/$(TOPNAME).fit.rpt: $(OPFILEDIR)/$(TOPNAME).map.rpt
	@quartus_fit $(TOPNAME).qpf --write_settings_files=off

$(OPFILEDIR)/$(TOPNAME).sta.rpt: $(OPFILEDIR)/$(TOPNAME).fit.rpt
	@quartus_sta $(TOPNAME).qpf --sdc=$(TOPNAME).sdc

$(OPFILEDIR)/$(TOPNAME).asm.rpt: $(OPFILEDIR)/$(TOPNAME).sta.rpt
	@quartus_asm $(TOPNAME).qpf --write_settings_files=off

synth: $(SOPCINFO)
	@quartus_sh --flow compile $(QPFFILE)

check_timing:
	@echo "Check for timing violations. Any found will be shown below:"
	@bash -c "grep \"Info\" $(OPFILEDIR)/$(TOPNAME).sta.rpt | \
	          grep \"Worst.*slack is -\" || true"
```

Firstly, note that the variable `HDLFILES` is assumed set elsewhere with a list of all the user HDL source files. This is used just for the dependencies, as the Quartus files will have all this information already. The best way would be if a file list could be auto-generated from the Quartus files, but that's a discussion for another time.

As ever, the top of the file sets some variables with details for the commands, allowing for easy porting for other systems, or overriding on the command line. There is an 'all' target with dependencies on three sub-targets, to generate the HPS header for software, the assembled configuration data for the FPGA logic, and to

execute a timing check on the STA output, looking for reported timing violations. The next set of targets are for ease of use for the make file, where these target names can be used on the command line for a specific step. E.g., `make qsys`. The first four of these have a dependency on a report file. Each individual stage produces a report file (in `output_directory`) of the format `top.<stage name>.rpt`, as well as many, many other files. We will use the report files, though, both as our targets and as the dependency of the next stage, to control the order of execution.

The first real target is for generating the HDL from the Platform Designer file (`top_system.qsys`). This has (amongst other files) an output file of `top_system.sopcinfo`, needed to generate the HPS header, `hps_0.h`, and so this is the target for this step. It has a dependency on the HDL files, as it is the first stage of the synthesis process. Any change to the user source code will ensure the process starts from scratch.

The next target is the HPS header, with a dependency on the `.socpinfo` file from the QSYS process. It is the `hps` target that is placed in the 'all' target, as this is the end of the process for generating this file, and nothing else is dependent on it. Choosing it as an 'all' dependency ensures it, and the QSYS process are checked for building.

The actual synthesis process is, as mentioned before, four steps, with the assembly step (`asm`) the final component. Actually, the order of STA and ASM is arbitrary, but a choice was made. Thus, there is a chain of dependencies on each of the report files targets, with the next state having a dependency on the previous stage's report file. The first mapping stage, though, is, like the HPS header, dependent on the `.sopcinfo` file of QSYS, to ensure this is built first if required. The step does not use the `.sopcinfo` file itself, but the QSYS generation step generates all the files needed for the mapping step, and the `.sopcinfo` is our chosen output for determining dependency.

To show both methods of synthesizing, a `synth` target is added with the single step command to do all processes in one go. Thus, `make synth`, will build using this command. Finally, a `check_timing` target, added as a dependency to `all`, is used to extract and display any timing violations from the STA report. It is not meant to be very informative, but just show if any are present so that the full report can be inspected.

And this is all there is for the Quartus flow. At this point I was going to compare this commercial tool's flow with the open-source tool flow using YOSYS and NEXTPR, but, wouldn't you know it, someone has already done this *using make*. If you want

another example of a synthesis flow using `make`, then I highly recommend the 'learn-fpga-easily' article.

Conclusions

In this document, we've had a basic introduction to `gmake` in general, using its original intended use of compiling software for the initial examples, in order to get a feel for its usage and foibles. Once we had this basic understanding we turned our attentions to constructing make files relevant to logic development, in particular logic simulation and synthesis.

For logic simulation we identified the command line programs to free us from the GUI, and constructed an example make file for the open-source NVC simulator, with some structure to allow for ease of porting and maintenance. In particular, the use of a common make file to compile the PLI co-simulation software, configurable for the particular target simulator, made the example makefile compact and simple. The details of the PLI software compilation were glossed over, but this document is about the use of make for logic. The make file example not only compiled the HDL source code, but we saw that we can use make to build and then run the simulation, even displaying waveforms in a GUI application. This gives a single point for all steps to build and run a simulation, ensuring all components are using the latest code. In addition, the batch run target makes integration into a CI/CD system very simple—just execute `make run` within the CI/CD build process.

We next turned to synthesis and looked at an example make file for the Quartus Prime tool for Altera FPGAs. The make files are somewhat more complicated than for logic simulation, and we had to determine a suitable target output for each process step, amongst many that each step produces. Having decided on the report files, the make file constructs a hierarchy to go through each step, building any previous step as necessary. We also had some convenience targets to allow a particular step to be the target from the command line, and also added another target to use the tool's single step process, if desired. The use of the multi-step targets was shown to be more flexible to allow running from a particular step if, say, debugging failures.

This has not been a definitive look at `make`, but it is hoped that this introduction and look at its deployment in logic development is useful. Other parts of the logic development flow could easily be run via `make`, such as formal verification tools, or design rule checkers etc., making CI/CD integration easier, but also allowing simple command run in a terminal to do multi-step complex builds. The point here was to

have opened the door to generating make files and give two examples to show how these can be used to run and control logic EDA processes.