The Boa Methodology

by Wilson Snyder, Digital Semiconductor, January 17, 1997

Abstract

The Synopsys methodology devised by Digital Semiconductor called the "Boa Methodology" is described. The Boa Methodology starts with a user created file which lists every major signal and the signal's expected timing. The users also provide a overall driver script, and a sub-design script file for each sub-design. These files only lists the deviations from a default methodology, rather than every minor Synopsys command, allowing the complexities of the process to be almost entirely hidden from users.

When the process starts, every sub-design is compiled stand-alone. The results are then analyzed across the whole chip, outside of Synopsys, to determine the timing slack or violation for every port in the design. Any violations are weighted by the cycle time to create different constraints for the sub-designs driving and receiving the signal. Breaking signals are also path grouped and weighted automatically.

The compile-all then re-constrain loop continues through several iterations. Leaf sub-designs are then ungrouped into their parents, and the loop continues with the larger sub-designs. This continues up to the whole chip level, or to the point which corresponds to physical layout blocks.

The timing status of the run can be viewed by looking at Verilog code in the Emacs editor with the timing of each signal back-annotated.

Descriptions of a method to run the compiles in parallel on multiple machines are also described, including some very simple scripts that others can use in their own methodologies.

This methodology was successful in meeting our cycle time goals, and approximately halved the cycle time compared to bottom-up and top-down compile methodologies.

What is Boa?

While designing core logic chips two years ago, our group tried many synthesis methodologies, but none seemed to improve our cycle time significantly. We discovered that our designs of over 150,000 gates at 10ns were too large and fast to be handled directly by Synopsys, and getting design convergence was almost impossible. Furthermore, it took a long time to trace down what signals and code were really breaking the synthesis run, as one difficult signal often caused other lower cost paths to break.

We experimented with several methodologies. Our best results came when we compiled a single sub-design stand-alone, using hand generated constraints. Not only were the results excellent, we could very easily fix timing problems in the behavioral code, because we knew exactly what timing assumptions weren't being met inside the module. We decided to automate the stand-alone compiles where possible, and in the end came up with a methodology that doesn't look stand-alone at all.

The first goal for our methodology, by far, was cycle time. Second, it had to handle ports not directly from flops, which were prevalent in our design due to custom data path blocks. We also wished to remove redundant information, such as having to specify set_input_delay and set_output_delay on the same signal in different sub-designs. We wanted to hide the internal details of the methodology. Finally, we wanted to compile the whole chip with one shell command, but still allow single sub-design recompiles for bug fixes and experimentation.

Organization

This paper will discuss our resulting methodology, called the Boa Constrictor methodology, or just Boa for short. Much of the emphasis will be on the Boa Program, a Perl script that is the center of the Boa Methodology. Rather then provide source code, this paper will discuss the features and flow of our methodology, and what we learned in developing it. Some underlying functions will be provided, though the general intent is to provide ideas for others to build upon.

This paper is divided into four major sections. The first describes the Boa Program which calculates timing and other constraints. The second describes the Boa Scripts, which give a uniform set of commands that work anywhere in a design hierarchy. The third section provides a general method of running compiles in parallel on multiple machines. Finally, the last section contains some suggestions for the improvement of the Synopsys program itself.

Overall View

Before beginning discussion of the program itself, it is useful to get a overall view of the Boa Methodology. The overall steps in the methodology are below, organized in sequential order. The step's position in the Boa Methodology, Boa Scripts or Boa Program is indicated.

1. The user first specifies the hierarchy of the design and makes a small script for each separately compiled sub-design in the design. (Boa Methodology)
2. The user creates a timing file, which includes initial timing for every port in the sub-design. (Boa Methodology)
3. The Boa Scripts start. The first phase reads in the behavioral code. (Boa Scripts)
4. Run Boa Program. Boa will take the timing file and create a constraint (.con) file for every sub-design. (Boa Program) This can further be broken into several phases:
    1. Read the timing file
    2. Characterize
    3. Calculate timing numbers
    4. Generate constraints
5. Compile each sub-design using that sub-design's .con file. (Boa Scripts)
6. Load in the top of the hierarchy, and do a characterize and write_script on each sub-design. The output of the write_scripts will contain the constraints needed if that sub-design is to make the cycle time. (Boa Scripts)
7. Run Boa Program again. Boa will take the timing file, and the write_scripts, generate new timing numbers, and create a constraint (.con) file for every sub-design. (Boa Program)
8. Repeat steps 5-7 until the critical path timing converges. (Boa Scripts)
9. Run Boa to annotate the timing back onto the behavioral code. Create a new timing file with the results of the compile. This new timing file may be used in the next Boa run as a starting point. Doing so will lead to faster convergence. (Boa Program)
10. Verify user-instantiated structural code against behavioral code with compare_design. (Boa Methodology)

Boa Program

Let's begin to look at the whole methodology at the lowest level, the Boa Program. The program is written in Perl, a excellent language for string manipulation and handling signals with associative arrays.

The Boa Program takes as input a timing file, Verilog code, and Synopsys write_script outputs. It creates constraint dc_shell scripts and annotation information.

Timing file

The main file which drives the Boa Program is the timing file. There is one timing file for the entire chip, which contains all of the simple constraints for the design. Most of the commands in the file relate to signals, which in our methodology must be named identically across hierarchical boundaries. (Thus port names and signal names are identical.) Having a single timing file greatly simplifies finding and editing the constraints in the design. (The program actually supports multiple timing files, which we experimented with, but decided wasn't worth the additional editing overhead.)

The timing file is created initially and maintained by hand. Some of our other tools also make suggestions on commands for inclusion in the timing file, but currently the user must make the changes themselves. This reduces garbage-in, garbage-out problems by insuring that bad machine-generated constraints were at least looked at once by the user.

Many new users to Boa express initial resistance to entering timing numbers. However, even for documentation reasons alone it has proven its worth. It is especially valuable at the interface signals between two designers to indicate design assumptions when coding is in progress. Later in the process it is also useful when the behavioral coder wants to indicate to a synthesis engineer what timing is expected. Without a timing file, it is hard to tell what the design assumptions were, and if the current synthesis run is meeting expectation.

What is in a timing file? The timing file is read by Boa as a sequence of commands, one per line. A command typically updates a database of signal constraints inside the program, which will then affect Boa's output files. The most interesting commands are:

timing {signal} {time}
: The timing command specifies when the signal is expected to arrive, similar to set_input_delay. Optional flags specify the clock domain, rising or falling edges, and min or max timing. The time is converted to set_output_delay by subtracting the cycle time. There is also a alias feature which allows users to use a mnemonic, such as DEFAULT_INPUT_PORT_TIME rather then a absolute timing number. This aids readability, and permits easy adjusting of the default's value.

loading {signal} /wire /port
: The loading command passes wireload and port load information to Synopsys. In our methodology, some blocks were custom, so this command allowed us to specify a specific capacitive load for those blocks. If a loading/port command is never specified for a signal, Boa will pick a typical receiver load to override Synopsys' default of no loading.

driving {signal} {cell and port}
: Driving commands specify the cell driving a signal. If a driving command is never specified for a signal, Boa will pick a typical driver to override Synopsys' default of infinite drive.

weight {signal} {weight_value}
: Weight commands create a group_path with only one signal, with the desired weights. The weight is normally just a starting value, and may be overridden if the signal is found to be more or less critical then specified. A optional flag prevents this readjustment, to allow weighting supercritical signals with

very high values.

path {signal}
: The path command defines a signal as being a fake path, such as a CSR signal. In addition to making a set_false_path in Synopsys, it also indicates to our downstream layout and timing analyzers that the signal is not critical.

Initial constraining

Before compiling for the first time, the Boa Program is run to establish initial constraints. Initial constraints are extremely important, as they make the selection between design-ware implementations, determine state machine codings, and have a large impact on the initial mapping of don't care outputs.

The initial timing constraints for a design come strait from the timing file. Hopefully, the numbers are reasonably accurate, and will be close to the final constraints when the process is complete.

When compiling a design for the first time, input driving and output loadings aren't known. Boa places defaults on both, choosing a driving cell which can handle about 8 fan-ins, and placing a loading of about 4 fan-ins. These default loadings make a huge improvement in initial results, as otherwise Synopsys would put many loads on a input net, due to Synopsys's default of infinite drive strengths. If Boa's default assumptions were vastly incorrect, they will get corrected in later iterations.

Characterizing

After compiling, the Boa Program is rerun to generate new timing constraints. To generate the new timing numbers, Boa needs to know which signals are breaking timing and which are not. Rather then using a timing report, which may only contain some of the violations, Boa wants to know the timing of each signal in the design. Since Boa can only place timing constraints on pins, only the timing of every port is required.

The key to getting port timing is the use of characterize and write_script. If for every sub-design we characterize that sub-design, then write_script, we get what that sub-design's port-timing needs. A given signal will always appear in at least two write_script output files (called .wscr files), one for the sub-design that outputs the signal, and another for the sub-design(s) that input the signal. By comparing the input and output delays that Synopsys generated for both, we get a picture of the timing requirements of that signal.

For example, let's look at a design with sub-designs OA and IB. Signal OA_SIGNAL is a output from OA and a input to IB.

OA.wscr, the wscr of the OA sub-design, contains:

- `set_output_delay 5.83 -max -rise -clock "CLK" "OA_SIGNAL"`

IB.wscr, the wscr of the IB sub-design, contains:

- `set_input_delay 3.36 -rise -clock "CLK" "OA_SIGNAL"`

What Synopsys is saying in effect is:

- To have OA meet timing, OA_SIGNAL must allow for 5.83ns of logic inside IB before the signal hits the next flop.
- To have IB meet timing, OA_SIGNAL must allow for 3.36ns of logic before the signal is generated by OA.

These two numbers aren't directly comparable, as the former number is referenced to a clock edge one clock after the latter number's reference edge. The difference in time between the reference points is the cycle time. (In this analysis, transfers between different clocks and skew issues are ignored.)

Boa adjusts both for the cycle time, to make the numbers absolute in time:

- To have OA meet timing, OA_SIGNAL must be generated before 4.17ns (10ns cycle-time - 5.83ns)
- To have IB meet timing, OA_SIGNAL must allow for 3.36ns of logic before the signal is generated by OA.

Now, these numbers were generated by the output of a actual compile run, so we know that IB's timing constraints took into account how much logic already was in OA, and vice versa. Thus:

- In the current compile of IB, OA_SIGNAL was needed by 4.17ns.
- In the current compile of OA, OA_SIGNAL was generated at 3.36ns.

The former is the needed-at time, the latter the arrival time. Comparing these two numbers gives us the slack or violation, in this case a slack of 0.81 ns, because the signal is generated that long before the input sub-design needs it. If we were to recompile, we could insert 0.81 ns of logic on this signal, and still meet the cycle time, presuming no other signals change.

Other information is also processed from the wscr files, such as set_driving_cell, and set_load constraints. These are simply recorded and used for constraints in the next compile iteration.

Calculate updated timing constraints

Once the write script files have been read, we have two numbers discussed previously for each signal; the arrival time, and the needed-by time. These are converted into a single updated timing number which will be used as a constraint the next iteration.

Each signal and edge in the chip is processed independently. (Though there are cases where one signal affects another, this simplification doesn't seem to make much difference in convergence.)

First, some arrival times or needed-by times may be hard numbers. Hard numbers are specified by the user for pins and schematics where the number cannot be changed. If either the arrival or needed-by time is hard, the updated time is identical to the hard constraint.

Otherwise, if there is slack, the slack is divided between the input sub-design and output sub-design in proportion to the amount of logic in each. This is because it is presumed that a sub-design which uses the majority of time on a signal has more opportunity to leverage the slack to improve other breaking paths.

For example, look at a signal arriving at 1ns and needed-at 6ns in a design with a 10ns cycle time. The signal has 6-1=5ns of slack. The midpoint between arrival and needed-at is (6+1)/2 = 3.5ns. Using the midpoint, the source sub-design is using 3.5ns/10ns = 35% of the total cycle time, the destination sub-design is using the remainder. Scaling the slack by 35%, the source sub-design should get 35%*5ns = 1.75ns of the slack. Adding back the original arrival time, the updated time constraint is 1ns+1.75ns = 2.75ns.

If a signal violates, a similar computation takes place. It a signal arrives at 6ns, and is needed-by 4ns, the violation is 6-4=2 ns. We scale both the arrival and needed-by time by the cycle time over path time. With the 10ns cycle time, we need to scale the whole path by (10/(10+2)) = 83%. The adjusted arrival time is 6ns * 83% = 4.98 ns. That is our adjusted timing number. (Note that the destination logic will also have to shrink by 83% to fit in the cycle time.) There is also a final window which holds the adjusted time between 1ns and

9ns, which prevents a constraint so tight that there isn't even time for a bare flop's clock-to-Q delay, or a receiving flop's setup time.

If a signal violates by more then a small amount, it is also weighted. The weight is proportional to the violation, where weight = 1 + 6 * (violation delta/cycle time). This gives the default weighting of one for paths that don't violate, and the more arbitrary weight of 6 to a signal which is doubling the cycle time. To keep compile time reasonable, we found that the number of path groups must be kept small. Thus only the strongest 100 weights over 1.5 are turned into constraints.

Generating constraints

With all of the timing numbers calculated, the constraints have to be fed to Synopsys. Boa creates a independent dc_shell script for each sub-design containing constraint commands, called the .con file. To make constraints on a sub-design the user then only has to include that sub-design's .con file, and it is ready to be compiled.

To avoid useless error messages, the .con file should only include commands for that sub-design. This is done by reading the Verilog source file for the sub-design being constrained, and looking at the input and output statements. Any signal which is a I/O to that sub-design gets the constraints for that signal. If a I/O doesn't have constraints, it's a warning. If a constraint isn't a I/O to that sub-design, it can be left out, since it won't have any effect.

Annotation

With all of the signals in a design, and various timing calculations, it becomes difficult to correlate the critical paths in a design back to the Verilog code. (HDL Advisor has since covered some of this problem.) To make it very simple, Boa annotates the original Verilog code with comments at every signal. For example, a original line of source:

```
        wire PSM_LDLADR = PSM_FRAME & PSM_ENLD
```

becomes (with spacing reformatted):

```
        wire PSM_LDLADR`3.0, 6.1, 4.6, 8.1,  3.2' =
              PSM_FRAME`4.0, 3.3, 3.7, 2.7, -1.3'
             & PSM_ENLD`2.0, 3.9, 2.4. 7.2,  4.6'
```

where the various numbers actually appear in different colors, when viewed using GNU Emacs. Each color represents a different timing number associated with that signal. Paging through a design looking for red violations can be enlightening.

Example	Color	Meaning
3.0	Gray	Original time, user entered in .timing file
6.1	Yellow	Updated time, calculated by Boa Program
4.6	Purple	Arrival time, from source logic via .wscr
8.1	Blue	Needed-by time, from destination logic via .wscr
3.2	Green	Slack (Needed-Arrival)
-1.3	Red	Violation Slack (Needed-Arrival)

Other tool connections

Any good tool grows beyond its necessary functions, and into bloat. Once we had a single timing file with the whole design's constraints, that became the convenient intermediate language for other whole chip information. Some of the features that Boa implements to talk to other tools include:

Timing analysis interface. Boa can create commands for our timing analyzer, and will automatically create the equivalent of the false_path commands used in Synopsys. Boa can also read the output of the timing analyzer and compare those numbers to Synopsys's calculations to verify the timing calculations including the wire loading information. As our timing analyzer is transistor based, this also provides verification of the cell library timing.

Global wire estimation. The timing file has simple floorplan coordinates for our major sub-designs. When coupled with port placement, Boa can calculate inter-design wire-loads by finding the Manhattan distance between all loads on a signal.

Script flow

With the Boa Program already discussed, this next section will look at the Boa Scripts. The scripts invoke the Boa Program, and provide a general method for compiling any sub-design in a hierarchy without writing additional scripts.

The scripts are organized into three levels. The lowest, and first to be discussed, is the standard sub-design script, which the user doesn't see. Next is independent sub-design script for every sub-design, which invokes the sub-design script. Finally there is a driver script which gives the commands for the whole chip.

Standard sub-design steps

The lowest level script is the standard sub-design script. Most actions that need to be performed on a sub-design are the same for every sub-design in a chip. Rather then including the same redundant commands in several scripts, some standard commands are defined in a default module compiling file, called do_standard_module.

Since Synopsys does not allow parameter passing in the scripts, the action the scripts are to take is passed as a variable called 'step'.

Each of these steps is performed on the current sub-design and every sub-sub-design below it, in top-down or bottom-up order, depending on which direction is correct for that step. So a step="compile" actually compiles bottom-up, starting at the lowest sub-designs and working up the top sub-design, all with one user command!

step = "read_verilog";
    Read -format Verilog and check_sub-design every sub-design.

step = "check_design";
    Check_design every sub-design.

step = "characterize";
    Characterize and create a write_script for every sub-design.

step = "compile";
    Set many chip default compiler variables, and perform a compile.

step = "constrain";
    Run the Boa Program to create new constraints.

step = "fsm";
Do finite state machine optimization and create state tables on every sub-design that contains a FSM.

step = "report_timing";
Make a timing report for each sub-design. Even though the top level timing report is of most interest, the lower sub-design reports are often useful for finding constraint problems and paths that are totally internal to a sub-design.

step = "ungroup";
Flatten the sub-design to the ungrouping_level. This works by clearing all don't touch attributes, putting a don't touch on the subchips that should remain, then ungrouping. Clearing the attributes first allows this step to ungroup any Design-Ware components that were created in the last compilation step.

step = "wall_to_dir";
Write the database files to a directory, marking the sub-designs as saved. Wall stands for Write-All, and is approximately equivalent to a:

```
        wall_sub-design_list = find(sub-design,"*") - find(design, structurals)
        foreach (wall_design_name, wall_design_list) {
                set_attribute (wall_design_name, compile_not_saved, false, -type boole
                write wall_design_name;
                }
```

step = "write_edif";
Write a EDIF file for each sub-design, formatted for downstream tools.

Sub-design scripts

The sub-design scripts are dc_shell scripts that the user creates for each sub-design. This script includes the hierarchy of the sub-design, and how to compile it. This is how the standard sub-design steps described above can be built to operate on entire hierarchies.

Let's look at the pieces of a sub-design script.

The first section is a header in all chips. It defines the name of the sub-design.

```
        design_name = "subwsm"          /* Name of this design */
```

The header also describes what sub-sub-designs are under this sub-design in the compile hierarchy. As a given sub-sub-design can be instantiated multiple times, the first sub_names describes the sub-sub-designs that are called, while sub_insts has the cell's instantiated names. These are used to know what sub-sub-designs have to be compiled before this sub-design. If a sub-sub-design is not to be compiled independently, but to be compiled at the same time as this sub-design, it isn't listed as a sub_name.

```
        sub_names = {"subwsm_sm"}       /* Subchips */
        sub_insts = {"sm"}              /* Instance name of subchips*/
```

A design level is assigned to this sub-design where 0 is the lowest level of hierarchy, and higher numbers are higher levels, in a arbitrary scale. The ungrouping level is a global variable so that certain scripts can change their behavior as the compile progresses. It is compared to the design_level to figure out which sub-designs to ungroup. We've found better results when little designs are compiled first, then we progressively compile larger and larger designs. To support this, ungrouping_level starts off at a zero, and as it increases, sub-designs are ungrouped and fewer scripts are executed. As mentioned above, setting sub_names to null means that the sub-sub-designs will be ungrouped. (Rather then being compiled independently with a dont_touch attribute set.)

```
        design_level = 10       /* Level this design exists at */
        if (ungrouping_level > 0 ) {
            echo design_name + ".script: Subchips are ungrouped."
            sub_names = {}
            sub_insts = {}
        }
```

Some sub-designs need special constraints that Boa doesn't support in the more generic .timing file format. This can be done by embedding any standard dc_shell command in the sub-design's script.

```
        if (step == "compile" || step=="report_timing") {
            current_design = design_name
            /* Special timing commands can go here for this sub-design */
            set_disable_timing si_lb_ioreqhint  -from GN -to D
        }
```

Since almost all sub-designs are compiled the same way, the final command in most sub-design scripts is just to call a standard alias, which invokes the standard sub-design script described in the last section.

```
        do_standard_subdesign
```

Driver scripts

With all of the sub-design scripts in place, a single driver script is used to perform the various high-level steps on the chip. Here's yet another example:

For the first compiles, we want the lowest level sub-designs to be compiled independently. Also the critical range is set somewhat low, to save compile time.

```
        /*&&&&&&&&&&&&&&&&&&&& AT LOWEST LEVEL */
        ungrouping_level = 0
        compile_default_critical_range = 1.0
```

Now, we start performing steps, first reading in the behavioral code.

```
        step = "read_verilog";                    include top.script
```

Steps are like a function call to top.script with a argument of read_verilog. Top.script will read its Verilog code, then call a script for each sub-design of top, which will read their Verilog code and call their sub-sub-designs, and so on. The result is that every sub-design in the hierarchy will have its Verilog code read in.

Since some steps take a long time, we like to checkpoint along the process by saving .dbs to a directory. This enables users to see how the block changed during the various compile steps.

```
        step = "wall_to_dir"; dir = "db_verilog";  include top.script
```

Now we place constraints on the module by running the Boa Program then including the .con files which Boa produces.

```
        step = "constrain";                       include top.script
```

Before compiling, all FSMs are reduced and optimized.

```
        step = "fsm";                             include top.script
```

Now the sub-designs are compiled. Doing four compiles, with ungroups to remove any sub-design-ware components, results in significantly better results then three or fewer compiles.

```
step = "ungroup";                            include top.script
step = "compile"; use_inc_map = "false";     include top.script
step = "ungroup";                            include top.script
step = "compile"; use_inc_map = "false";     include top.script
step = "ungroup";                            include top.script
step = "compile"; use_inc_map = "true";      include top.script
step = "ungroup";                            include top.script
step = "compile"; use_inc_map = "true";      include top.script
step = "ungroup";                            include top.script

step = "wall_to_dir"; dir = "db_comp1";      include top.script
```

Now that the first compile run is done, we want to do another run with larger sub-designs (some smaller sub-designs that existed before will be ungrouped.) Also, we'll push harder on the critical range.

```
/*&&&&&&&&&&&&&&&&&&&& AT SUBCHIP LEVEL */
ungrouping_level = 10
compile_default_critical_range = 3.0
```

Now to perform the magic of the Boa Program. Boa will characterizing the design, determining the critical paths, calculating new timing, and place the constraints on the design.

```
step = "characterize";                       include top.script
step = "constrain";                          include top.script
```

Two more incremental compiles and a write.

```
step = "ungroup";                            include top.script
step = "compile"; use_inc_map = "true";      include top.script
step = "ungroup";                            include top.script
step = "compile"; use_inc_map = "true";      include top.script
step = "ungroup";                            include top.script
step = "wall_to_dir"; dir = "db_final";      include top.script
```

Normally the last 8 steps (characterize, constrain, ungroup, compile, ungroup, compile, wall) would be repeated two or three times till the design reaches the best cycle time.

Report any timing violations and create the final edif.

```
step = "report_timing";                      include top.script
step = "write_edif";                         include top.script
```

The amazing power of the organization of the scripts is that the exact same driver will work on any level of the design. Top.script is a sub-design script just like any other, so instead of including top.script, you just include any other sub-design script, and that sub-design will be compiled.

Structural instantiations

Once a sub-design is compiled and meets timing, the final step in the Boa Methodology is to verify any structural instantiations. Although counter to the advantages of synthesis, sometimes the only way to get a path to work is to structurally instantiate cells. By naming all structural instantiated cells si_*, the Boa Scripts can automatically identify structural instantiations, and know to dont_touch them, and keep them grouped while ungrouping the behavioral instantiations.

```
// Synopsys translate_off
// Behavioral code, read by simulator, not synthesis
wire PO_PSM_PADIORDDONE = ~GIRS_PADIORDDONE_R_L | (GIRS_BYPASSCOND_R & PSM_TRDY_D1_
// Synopsys translate_on
// Verilator translate_off
// Structural instantiations, read by synthesis, not simulator
ND2P si_iorddn1_l  (.Z(PSM_IORDDN1_P_L),        .A(GIRS_BYPASSCOND_R),  .B(PSM_TRDY_
AN2P si_iorddnf_l  (.Z(PSM_PADIORDDONE_P_L),    .A(PSM_IORDDN1_P_L),    .B(GIRS_PADI
B8I  si_iorddnb1_l (.Z(PO_PSM_PADIORDDONE),     .A(PSM_PADIORDDONE_P_L));
// Verilator translate_on
```

After the complete synthesis loop is completed, the problem remains to verify if the user's instantiations implement the desired behavioral function. After the compilations are complete, this is verified by a automatic procedure which reads into Synopsys the simulator's version of the code. (Ignoring Synopsys translate_off's, and using the simulator translate_off's.) The behavioral version is then compare_design'ed with the structural version, proving equivalence.

Parallelization

In the compile loop of the Boa Scripts, each sub-design is compiled independently before any constraints are changed. This allows us to potentially compile each sub-design in parallel, each on a different machine, rather then serially on one machine. This concept was extended to general parallel compilation, which can be easily leveraged to other methodologies.

To keep things easy to maintain, the exact same script file is executed by many dc_shells each on a different machines, probably by a batch job system. (We'll just use the word machine from now on, but think of each machine as being a dc_shell running the same scripts on a different physical CPU.) To keep the log files strait, each log file has a unique name. This is done in the system .synopsys_dc.setup file with:

```
unique_user_id = get_unix_variable("USER")
                + "_" + get_unix_variable("HOST")
                + "_" + get_unix_variable("PID");
command_log_file = "./command_" + unique_user_id + ".log";
```

Each machine reads the script, and executes it sequentially. Any command in the script file will be executed at some point by every dc_shell. For example:

```
echo "Every machine will print this message."
```

That isn't particularly useful, since the same work will be done many times. We need a method to define work that only a single machine needs to do.

The scripts define a atomic unit of work as a TASK. Each task is some action that should be performed on one machine. Steps in a task are never broken up. Tasks, however, may be done in various orders with respect to previous or following tasks. In the below example, the tasks may echo messages in any order. (Task is a dc_shell alias defined in .synopsys_dc.setup.)

```
task { echo "Doing some work in task 1" }
task { echo "Doing some work in task 2" }
task { echo "Doing some work in task 3" }
```

If there is one machine running the scripts, it will execute all three echo commands in order. If there are two machines, machine one will echo two lines, and machine two will echo one line. If three machines, each machine echos one line. If using four machines, three machines each echo one line, and the fourth does nothing.

Sometimes a task needs to only be done on a single machine. Such tasks will always run it on machine 1, the primary machine. It is a primary_task. This is useful for running programs, like Boa, or report_timing, which are pointless if run more then once.

```
        primary_task { echo "This will only print on machine #1." }
```

A non_primary_task executes many times, once for each machine, EXCLUDING the primary machine. It is mainly used for synchronization commands that are not needed if the script is running on a single machine, not in parallel.

```
        non_primary_task { echo "This will print on all machines but #1." }
```

Finally, a synchronization primitive is needed to hold up work until all machines are done the previous steps. For example, all compiles must be finished before doing a report_timing on the whole design.

```
        echo "Every machine will print this message"
        sync
        echo "Before any machine prints this message"
```

All of these concepts come together to form parallel compiling:

```
        design_list = {a b c d e f}

        foreach (adesign, design_list) {
           task {
              echo "Each machine picked a design and is compiling it."
              current_design = adesign
              compile
           }
        }

        non_primary_task {
           echo "Every machine EXCEPT the primary machine is saving..."
           wall_unsaved /* Save any compiled-not-saved designs */
        }

        echo "A specific machine has finished compiling"
        sync
        echo "Every machine has finished compiling"

        primary_task {
           echo "The primary machine will now report the timing..."
           /* The primary machine doesn't have the compiled sub-designs yet. */
           remove_design find (design, "*");
           read top.db
           link
           /* Now the primary machine has the compiled sub-designs */
           report_timing
        }
```

Because so much time is spent in the compile command, our large compile runs often finished 3 times faster when run on 5 machines. Originally for example we had a 10 hour run, which you can do one of in a work day. With parallelization it takes 3 hours, which you could run four of a work day!

Synopsys Suggestion

During the development of Boa, many new Synopsys features would have greatly simplified the script development. These include things Synopsys users should look out for, and suggestions to the Synopsys programmers reading this paper.

Parallel

First, the whole parallel compilation feature should really be implemented in Synopsys itself. Several of our machines are dual or quad CPU systems, it would be excellent if compile trials could be run multi-threaded on all the CPUs in parallel. Presumably this would require multiple licenses, so it would even be to Synopsys's advantage to allow this parallel use. Even better would be multiple machines on a network, but threading first!

Associative arrays

Storing data in associative arrays, similar to perl, would have eliminated some major hacks. For example, Boa often needs to preserve some variables across a include file which wants to change those variables. This is done now by writing a disk file with the values, doing something, then executing the disk file as a script to restore the values. A most inefficient procedure.

Reading attributes

There is a strange misfeature in design compiler which requires tests of a attribute to use foreach loops instead of direct use. For example, rather then:

```
        compile -map_effort get_attribute (design_name, compile_effort)
```

You must use:

```
        eff_list = get_attribute (design_name, compile_effort)
            foreach (eff_temp_str, eff_list) {
            compile -only_design_rule -map_effort eff_temp_str
        }
```

Automatic is_structural attribute

Often we have code which only instantiates other cells, and contains no behavioral code. This is currently identified by the user in the script files, though there is no reason why it can't be detected automatically. Synopsys comes close with a is_mapped attribute, but that isn't set if structural code is read in. A is_behavioral attribute would allow automatic determination of if uncompiled behavioral code is present.

Support `ifdef

We have several programs that strip or insert comments in our Verilog code to change what Synopsys sees versus our simulator. (Such as the structural instantiations.) If Synopsys added support for 'ifdef, and a mechanism to set defines inside dc_shell, the comment hacks would no longer be required.

Better compiles

Synopsys makes different optimizations, depending on the compile switches used, as one would expect. But, you also get better results if you compile multiple times with the same switches. This may be hard to believe, but try it on your next breaking design! Currently our scripts use the sequence we found best:

```
        compile -map_effort high
        compile -map_effort high
        compile -map_effort high -incremental
        compile -map_effort high -incremental
```

This is often a overkill, and sometimes isn't even enough. Worst of all, this is only a subset of many of the switches that may help synthesis. Synopsys should implement a compile that tries many different switches, compile -so_high_that_youll_try_everything_before_giving_up, and keeps the resulting gates if a trial improves the result. If it takes a day, I don't care, just make a report to tell how to recreate the results faster the next time.

Less Verbosity

To make log files simpler to users, we strived to reduce useless messages. Synopsys echos every variable that is set, so most of our scripts redirect almost everything to /dev/null. A mode to eliminate printing variable sets would save a great deal of frustration. Errors and echo commands should of course always go to the screen and log file.

Latches

Though our designs are flop based, we have some latches in critical paths, and for race prevention. Unfortunately, Synopsys times latches incorrectly. Synopsys will time either the path from data to the gate closing, or, the path from data through Q to the next state device. The user must choose which is desired with the set_disable_timing command. This caused a tremendous headache for our project, as the process couldn't be automated, and a wrong identification could cause a real violation to be ignored.

Conclusion

The Boa Methodology provides most of the advantage of very tight hand-made constraints, omitting all of the tedious tuning and files required without Boa. Boa was successful in reaching our cycle time goals, which included a aggressive 10ns design for the core logic of the world's fastest uniprocessor system. Compiling the same chip with a bottom-up synthesis strategy, using the regular Synopsys methods, resulted in a cycle time of almost twice this methodology.

Credits

I would like to thank Paul Wasson who wrote most of the Boa Perl script, to Steve Kolecki who adapted it to the Digital 21271 chipset project, and to all who provided ideas and comments.

Appendix 1: Parallel scripts

This appendix gives the code to enable parallel compilations. It is divided into three sections:

toolkit.script
: is a dc_shell script which defines the parallel commands and is usually part of the default .synopsys_dc.setup startup file.

parallel_synchronize
: is a Perl program which is used to signal when all machines have reached a common point.

dc_script
: is a Perl program to invoke dc_shell multiple times on multiple machines.

These programs are provided as-is. Unfortunately, I do not have the time to respond to comments or support them, but hopefully they will provide ideas and a starting point for your own methodology.

This page authored by: wsnyder@wsnyder.org