Logic Synthesis with Genus

Prof. Adam Teman

Emerging Nanoscaled
Integrated Circuits and Systems Labs

The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University

Introduction

- **In the lectures about Synthesis and Static Timing Analysis, we learned:**
  - Basic Synthesis flow and supporting algorithms
  - Library definitions, models, MMMC
  - Timing requirements and defining constraints (SDC)
  - Optimizations – pre-synthesis and post-synthesis
  - Reports and exports

- **We will now see how to run a synthesis on our example RTL block**
  - We will write a script for synthesizing our design
  - We will write supporting SDC and MMMC definitions
  - We will run the Synthesis flow and look at log files
  - We will create reports and analyze them

Cadence Genus

- **Genus is Cadence's tool for Synthesis.**
  - Genus replaced Cadence's previous tool "RTL Compiler (RC)"
  - Genus was developed for Cadence's Stylus Common-UI
- **Other popular Synthesis tools include:**
  - Synopsys "Design Compiler (DC)", which is now part of the "Fusion" suite.
  - Siemens-EDA "Oasys-RTL Synthesis"
  - The open source "Yosys" synthesizer (https://yosyshq.net/yosys/), which is provided as part of the OpenRoad project (https://theopenroadproject.org/)
  - FPGA synthesizers, e.g. "Vivado", "Quartus", "Synplify" and "Prcision FPGA"
- **In addition, high-level synthesizers are available:**
  - e.g., Cadence Stratus, Siemens-EDA "Catapult", Synopsys "Synphony"

Starting Genus

- **Go to your `workspace/` directory**

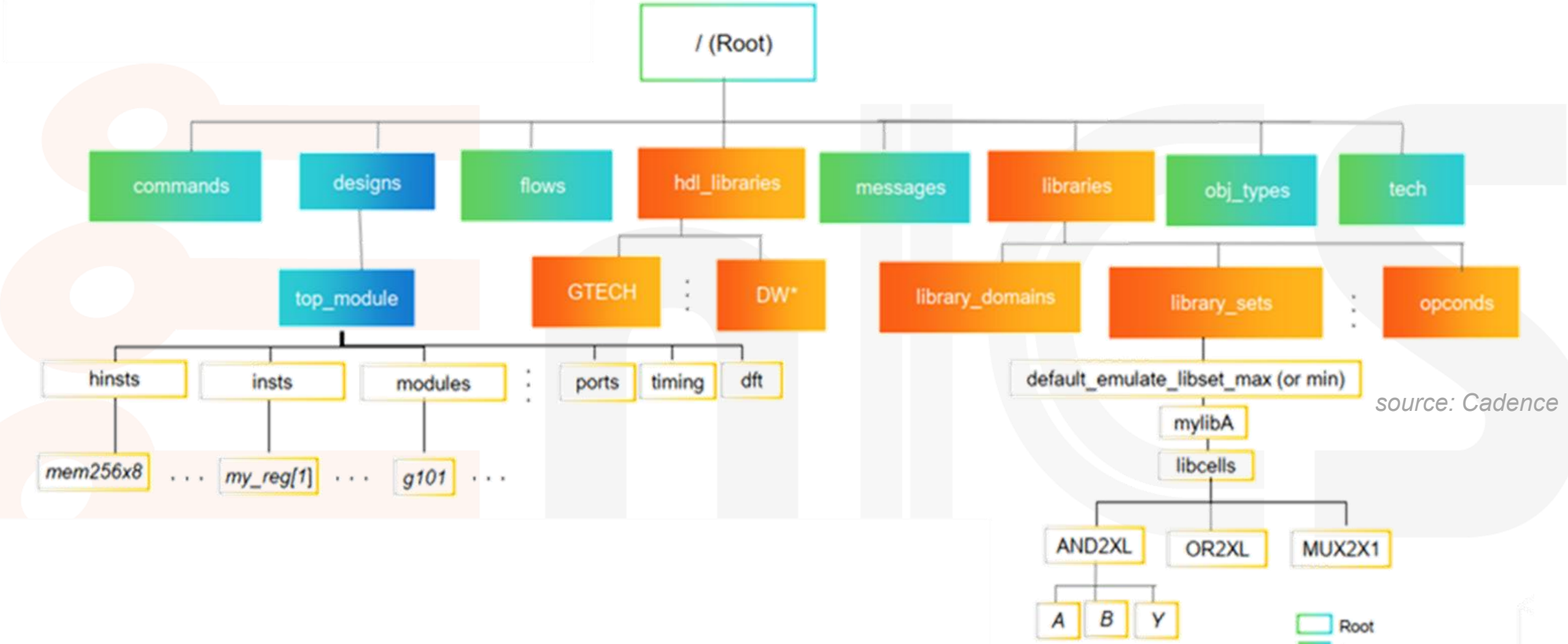
```
%> cd workspace/
```

  - Reminder: we run all tools from the `workspace/` directory
  - If you haven't yet watched the lecture about the project structure, make sure you check it out at: https://youtu.be/Wu2MuQWf8_s
- **Start Genus**

```
%> genus
```

  - By default, Genus will start in Stylus Common-UI mode
- **Genus will create two files for logging our run**
  - `genus.log` – The logfile printing out almost everything on screen.
  - `genus.cmd` – The log of commands that we ran in our session.
  - These files will get a number (i.e., `genus.log2`) if a previous run exists.
- **Now we will run our flow according to the `../scripts/genus.tcl` script**

Virtual Directory Structure



source: Cadence

```
@genus:root: 115> vls
./                      designs/                hdl_libraries/          messages/               tech/
commands/               flows/                  libraries/              obj_types/
```

Setting up our run

- **The first thing we have to do is define our `TOPLEVEL` name:**
  - The `TOPLEVEL` variable is used throughout the flow and must be defined before reading our define files.

```
@genus:root:1> set design(TOPLEVEL) "sm"
```

- **Another few run-specific variable we'll set up are:**
  - `runtype` : This helps our scripts differentiate between different tools
  - `phys_synth_type`: This tells us what type of physical synthesis to use

```
@genus:root:2> set runtype "synthesis"
@genus:root:3> set phys_synth_type "lef"
```

  - Note: I am assuming an MMMC flow.
    For a "classic" flow without MMMC, relate to Cadence documentation.

Loading Definitions

- **Now we will move on to loading all the definition files that we elaborated on in the project workspace tutorial:**
  - `procedures.tcl` : Useful procedures to use during the flow.
  - `<TOPLEVEL>.defines` : The main variable definitions for this design.
  - `settings.tcl` : Some tool-specific settings
  - `libraries.$TECHNOLOGY.tcl`: Definitions for this process technology.
  - `libraries.$SC_TECHNOLOGY.tcl`: Definitions for the standard cell library.
  - `libraries.$SRAM_TECHNOLOGY.tcl`: Definitions for compiled SRAMs.
  - `libraries.$IO_TECHNOLOGY.tcl`: Definitions for the I/O library.

- **We will then write out our debug information to the `debug.txt` file as explained in the project workspace tutorial.**

We're now ready to initialize our design

- **First step is reading our MMMC file**

```
@genus> read_mmmc $design(mmmc_view_file)
```

- **This sets up quite a few things:**
  - The paths to our `.lib` files
  - The paths to our `.sdc` files
  - The operating conditions and parasitic extraction definitions.
  - The selected Analysis Views for setup and hold timing and optimization.
  - BUT – it doesn't yet process all this stuff. That will come a bit later…
- **Next we set up the physical synthesis files:**
  - Read the `.lef` abstracts

```
@genus> read_physical -lef $tech_files(ALL_LEFS)
```

  - Optionally provide a floorplan

```
@genus> read_def $design(floorplan_def)
```

- **Finally, we'll read our RTL:**

```
@genus> read_hdl -f $design(read_hdl_list)
```

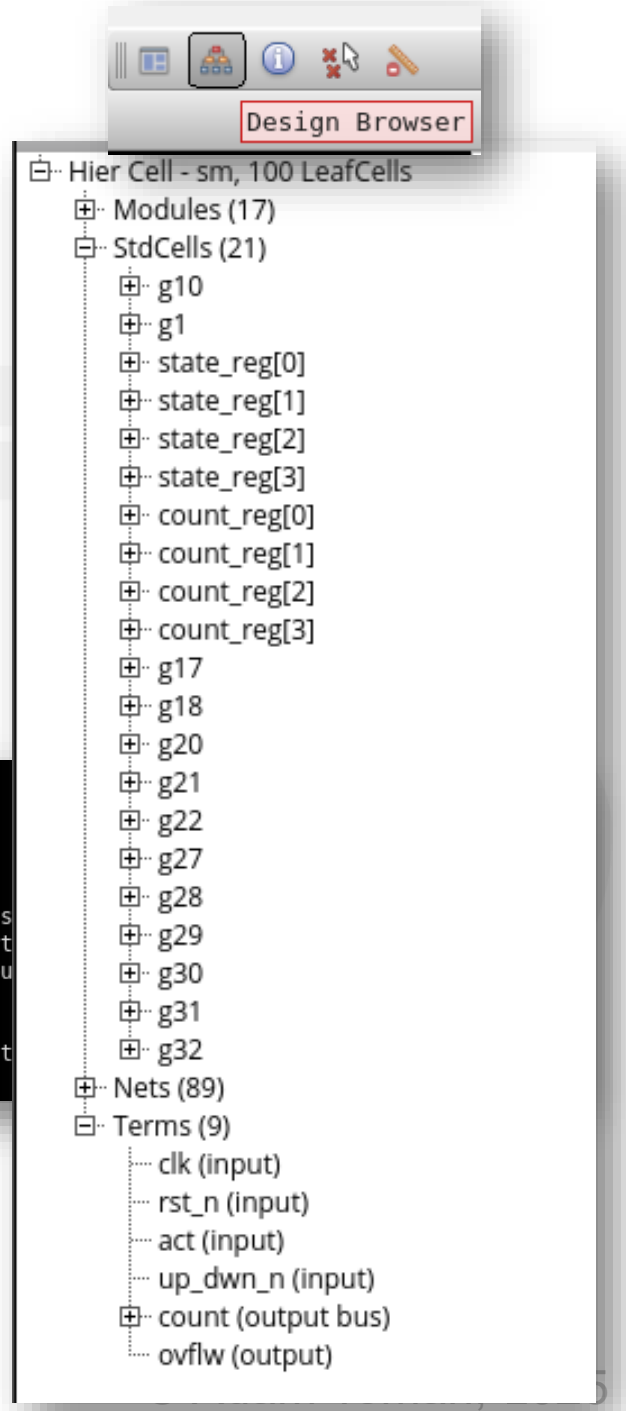
Design Elaboration

- **Now that we've read in the RTL and the library definitions, we can run design elaboration**
  - This will build our design inside genus and bind all the IPs.

```
@genus> elaborate $design(TOPLEVEL)
```

- **After Elaboration, our design is loaded into Genus, as can be seen through:**
  - The virtual directory
  - The design browser

```
@genus:root: 126> vls designs/
root:.designs:
./              sm/
@genus:root: 127> vls designs/sm
design:sm:
./                  constraint_modes/  hinsts/           modules/          pg_nets
analysis_views/     delay_corners/     hnets/            nets/             pg_port
constants/          dft/               insts/            pg_hnets/         port_bu
@genus:root: 128> vls designs/sm/ports
design:sm.ports:
./            act           clk           count[0]      count[1]      count[2]      count
@genus:root: 129>
```



- **We should now check that all IPs were bound:**

```
@genus> check_design -unresolved
```

Collections

- **At this point, we can use collections to navigate the design:**

`get_ports`	`get_clocks`	`all_inputs`	`all_registers`

- **Collections are pointers**
  - To calculate the **length** of a collection, use `sizeof_collection`
  - To **iterate** over all the objects in a collection use `foreach_in_collection`
  - To get the **name** (string) of an object, use `get_object_name`

- **However, collections are not native to Cadence tools**
  - Common-UI introduced the Dual Port Object (DPO)
  - More confusing, but more powerful within Cadence tools

Navigating the Genus Database

- The Stylus Common-UI provides a unified database across Cadence Digital Implementation tools (Genus, Innovus, Tempus, Voltus, etc.).
- Database access is through the `get_db` and `set_db` commands:

```
get_db <object> .<attribute_name>
```

```
set_db <object> .<attribute_name> <value>
```

- These commands work with Dual Port Objects (DPOs), which are Tcl strings
  - For example, to get a list of all ports:

```
get_db ports
```

  - And to check the direction of the clock port:

```
get_db port:sm/clk .direction
```

  - Or to get all the input ports

```
get_db ports -if {.direction == in }
```

Init Design

- **We can now run the `init_design` super command**
  - This will read in and process everything we loaded before: `.lib`, `.lef`, `.v`, etc.
- **Most importantly, it processes the `SDC` files**
  - We read the constraints in as part of the MMMC, but didn't have design objects and so couldn't process them.
- **The `check_timing_intent` command is a "timing linter". Find things like:**
  - Combinatorial loops
  - Multi-driven nets
  - Unconstrained inputs/outputs
- **We can now define cost groups (reg2reg, in2reg, reg2out, in2out)**

```
@genus> define_cost_group -name reg2reg -design $design(TOPLEVEL) -name reg2reg \
@genus> path_group -from [all_registers] -to [all_registers] -group reg2reg \
            -view $design(selected_setup_analysis_views)
```

It's time to Synthesize!

- **Our design is now elaborated, such that:**
  - All registers are inferred
  - All combinatorial logic is described with Boolean primitives
  - All IPs are bound to their .lib files
  - We have defined optimization constraints
- **Now we can Synthesize, starting with generic optimization**
  - Optimize datapath components
  - Insert clock gates
  - Implement mux structures

```
set_db syn_generic_effort medium
syn_generic -create_floorplan -physical
```

- **Following this, we are still not mapped to library cells!**

Technology Mapping and Optimization

- We can now apply technology mapping to connect our generic gates to standard cells from the library

```
set_db syn_map_effort medium
syn_map -physical
```

- We can now see that the cells are connected to actual standard cells

```
get_db inst:sm/state_reg[0] .base_cell
```

- And we can optimize the design

```
set_db syn_opt_effort medium
set_db opt_spatial_effort high
syn_opt -spatial
```

- And finally look at timing reports

```
report_timing
```

Reports and Export

- **Finally, we can create various reports, such as:**
  - `report_area` : Summarize the area of each component of the design
  - `report_gates` : Generate a report of all library cells
  - `report_qor` : General report of various design data

- **And we can export our design**
  - `write_design` : Generate all files needed to reload the session in Genus
  - `write_design -innovus -db` :
    Generate files needed to load the design in Innovus/Tempus/Voltus
  - `write_netlist` : Write out a structural Verilog netlist
  - `write_sdf` : Write out timing data for backannotation simulation

Summary

- **In this tutorial, we saw how to run synthesis with Cadence Genus:**
  - We set up our run by defining variables inside our Tcl arrays
  - We defined our MMMC file to set up our analysis views
  - We read in our RTL
  - We read in our `.lef` files to support physical synthesis
  - We elaborated our design
  - We initialized our design and checked that our SDC definitions were okay
  - We ran generic synthesis, technology mapping, and optimization
  - And finally, we looked through our reports and exported our design
- **In between we also saw the Genus virtual directory structure, used collections and learned how to access database objects with `get_db` and `set_db`**
- **We're now ready to go on to gate-level simulation and physical implementation.**