

Formality Jumpstart Training 2008

Agenda

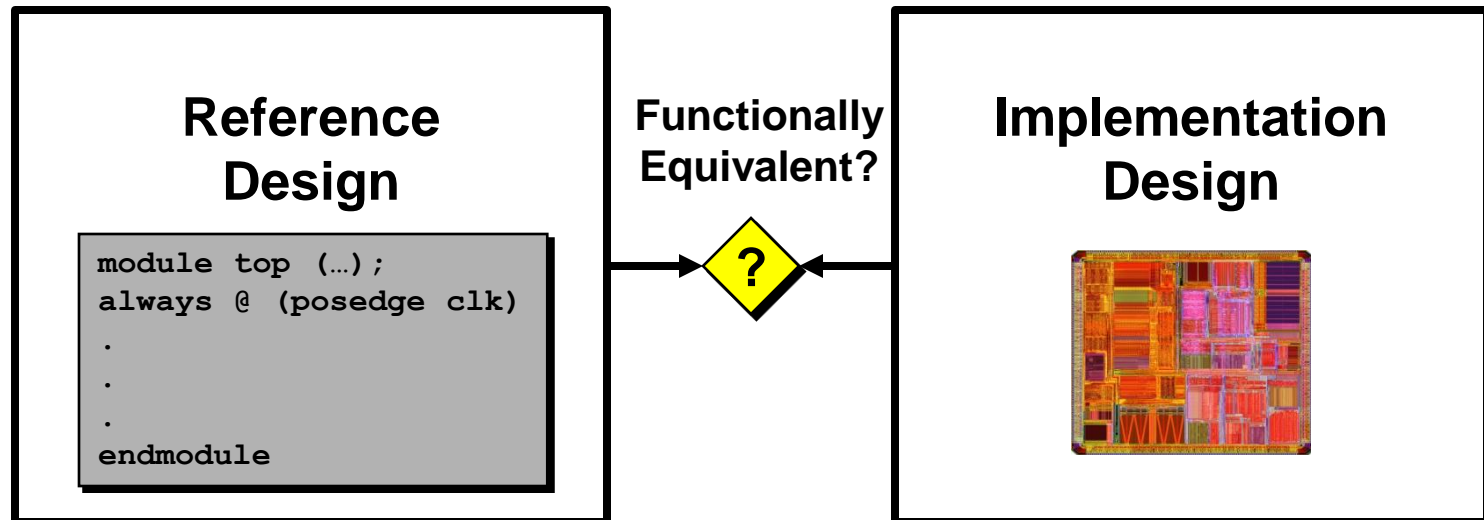
- Introduction to Equivalence Checking
- Using Formality
 - Flow Overview
 - Guidance
 - Read
 - Setup
 - Match
 - Verify
 - Debug
- Documentation and Help

Formality Terminology

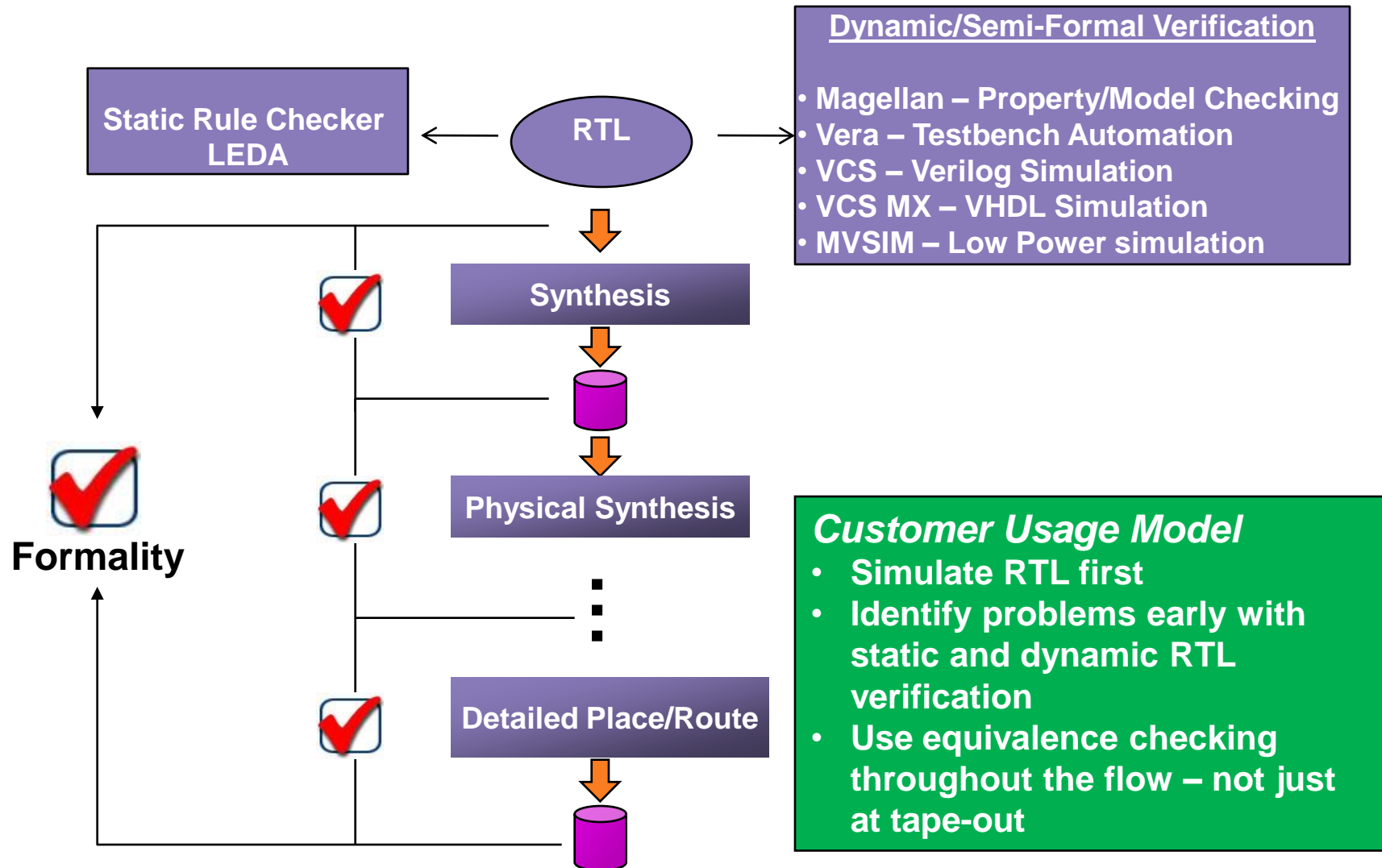
- Reference Design
 - The “golden” design under test
 - Frequently RTL (Verilog, SystemVerilog, VHDL)
 - Simulated and known to be good
- Implementation Design
 - The modified design being checked against the golden reference
- Containers
 - Formality database for designs and libraries
 - Default reference container is named “r”
 - Default implementation container is named “i”
 - Can be saved and read using any version of Formality

Formality is an Equivalence Checker

- Assumes that the reference design is functionally correct
- Determines if the implementation design is functionally equivalent to the reference
 - Provides counter-examples if designs are functionally different
- Is mathematically exhaustive with no missing corner cases
- Does not require test vectors



Equivalence Checking in Your Flow

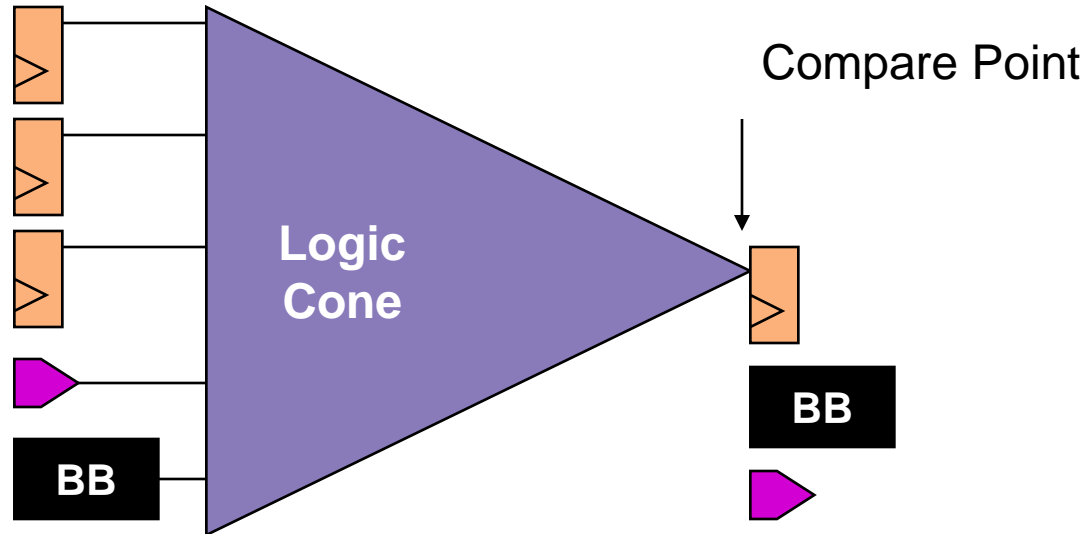


Key Equivalence Checking Concepts

Logic Cones and Compare Points

- Common Compare Points
 - Primary output
 - Register or latch
 - Input of a black-box
- Less Common Compare Points
 - Multiply-driven net
 - Loop
 - Cutpoint
- Logic Cone
 - A block of combinational logic which drives a compare point

Key Concepts – Logic Cone



Inputs

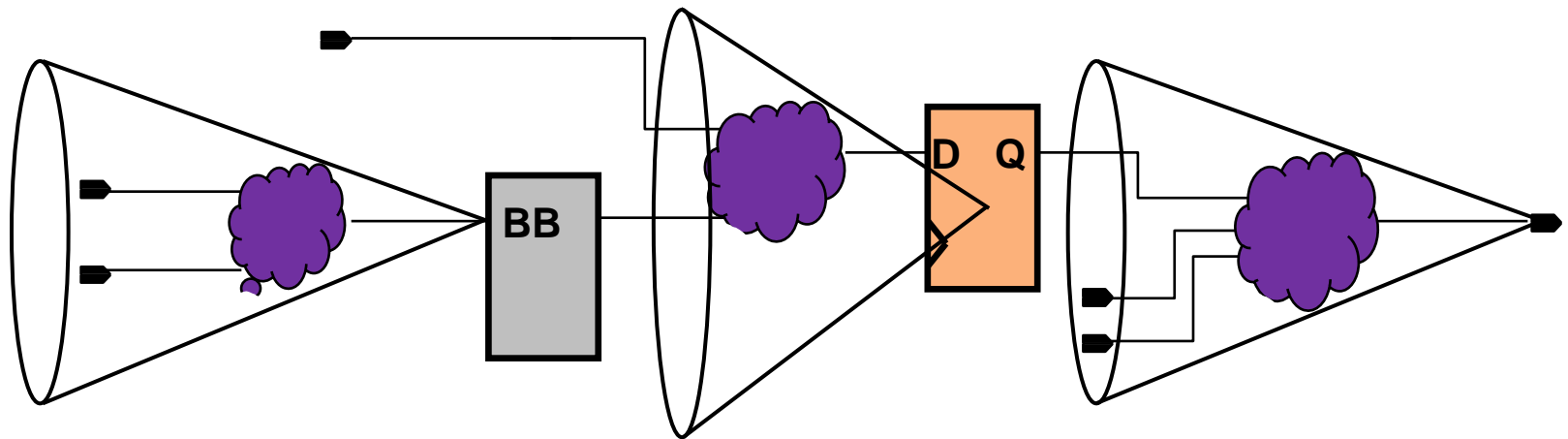
- Registers
- Primary Input Ports
- Black Box Output Pins

Compare Points

- Registers
- Primary Output Ports
- Black Boxes Input Pins

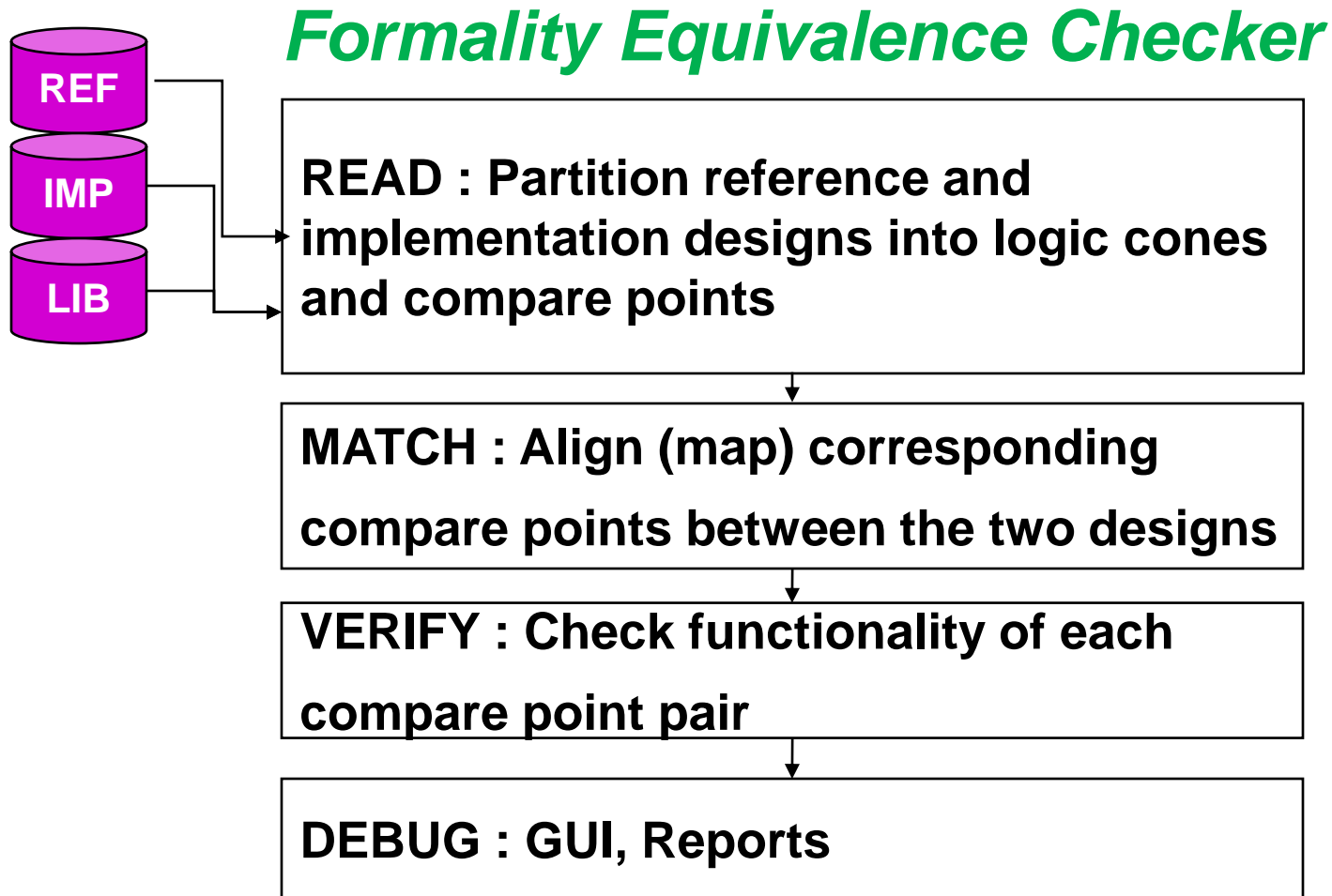
Key Concept – Designs Consist of Logic Cones and Compare Points

- Formality breaks the reference and implementation designs up into compare points, each with its own logic cone



Determining Compare Points

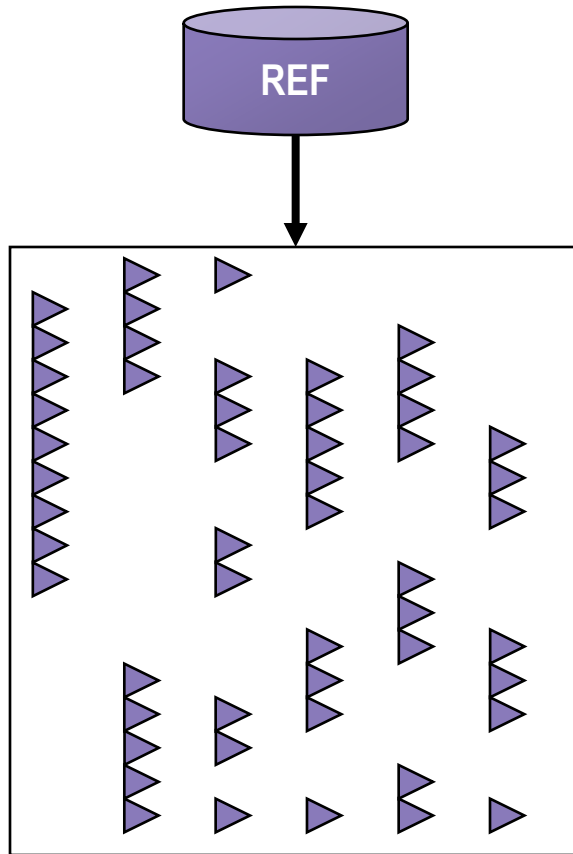
Formality Flow Overview



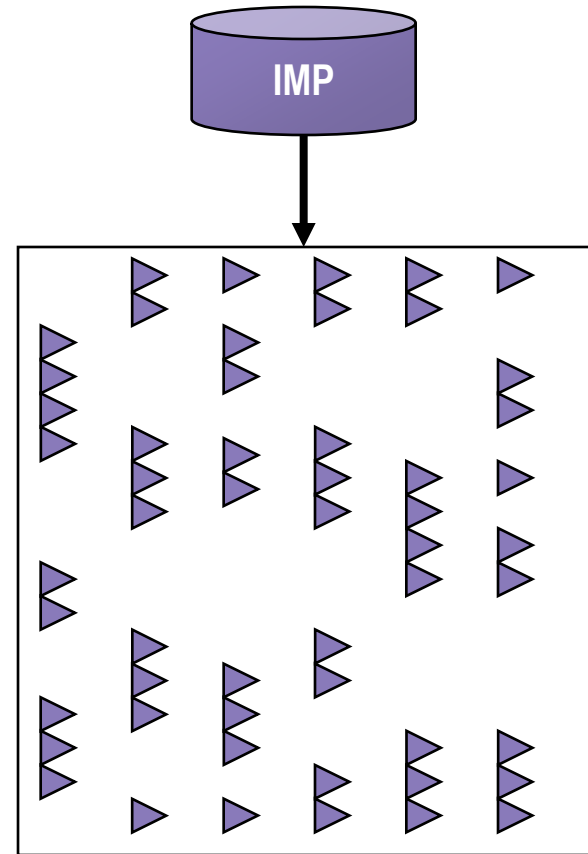
The Design Read Cycle

Breaks Designs into Logic Cones

Reference Design



Implementation Design

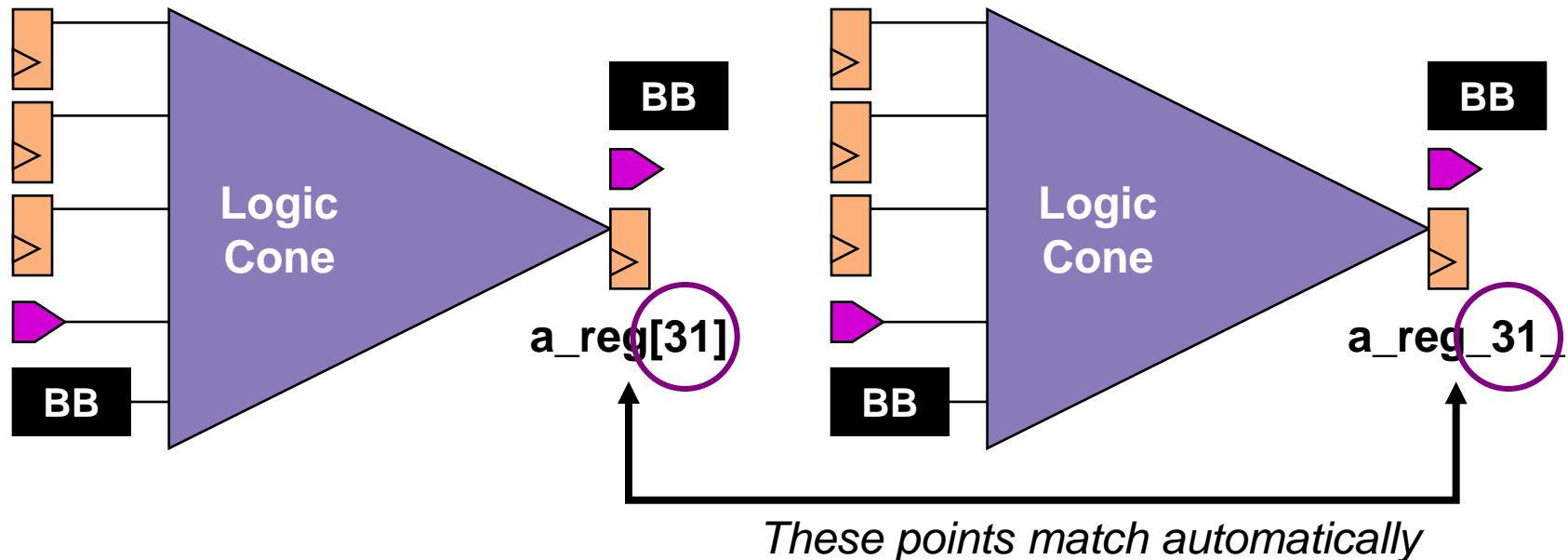


The Matching Cycle

Matches Corresponding Points within Designs

Reference Design

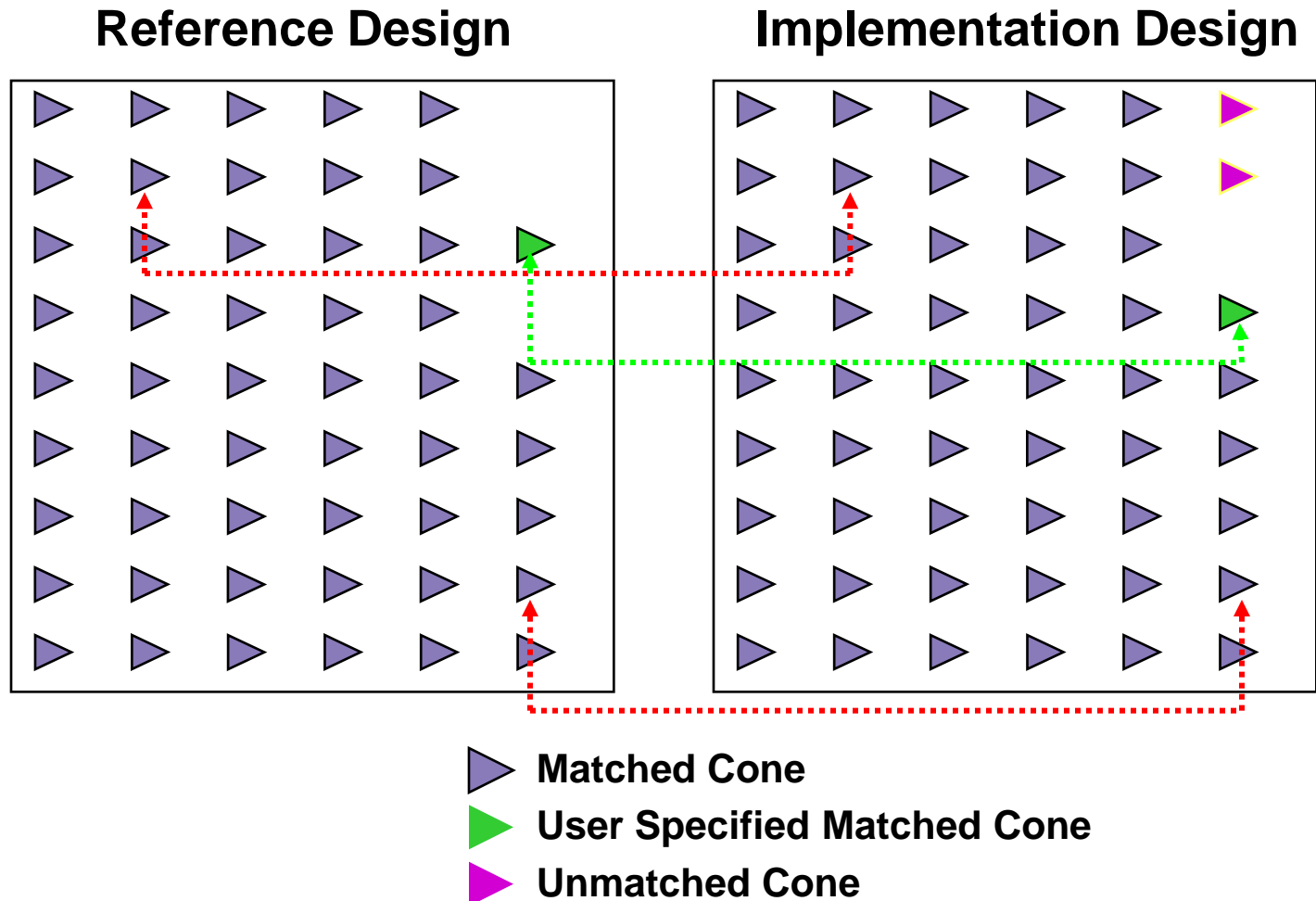
Implementation Design



**Most compare points match by name;
otherwise, need guidance information,
manual matching, or compare rules**

The Matching Cycle

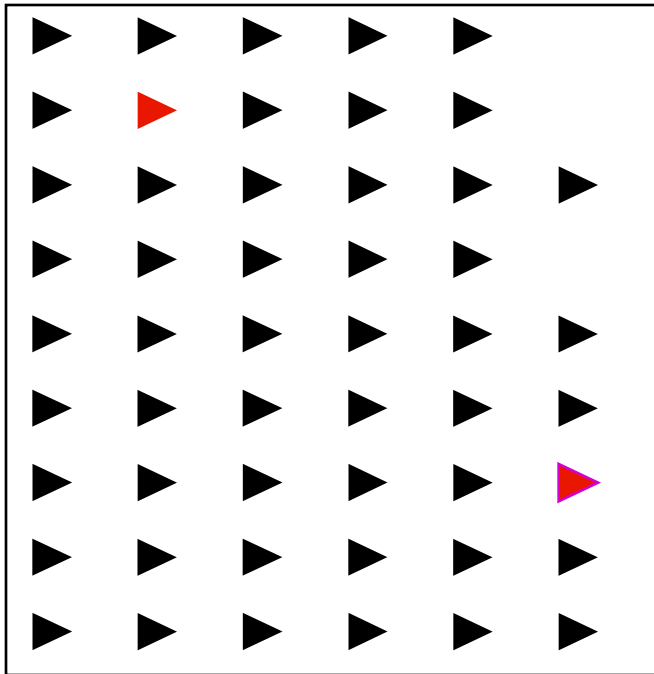
Matches Corresponding Points within Designs



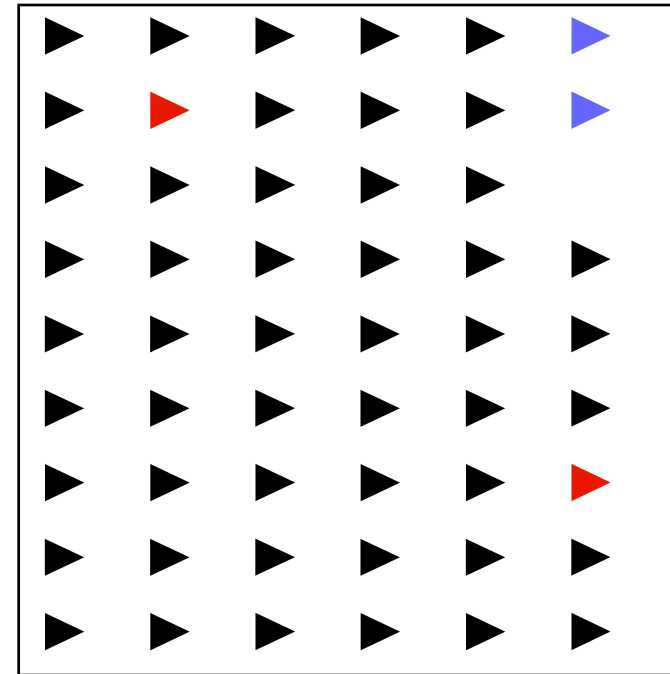
The Verification Cycle

Verifies Logical Equivalence for Each Logic Cone

Reference Design



Implementation Design

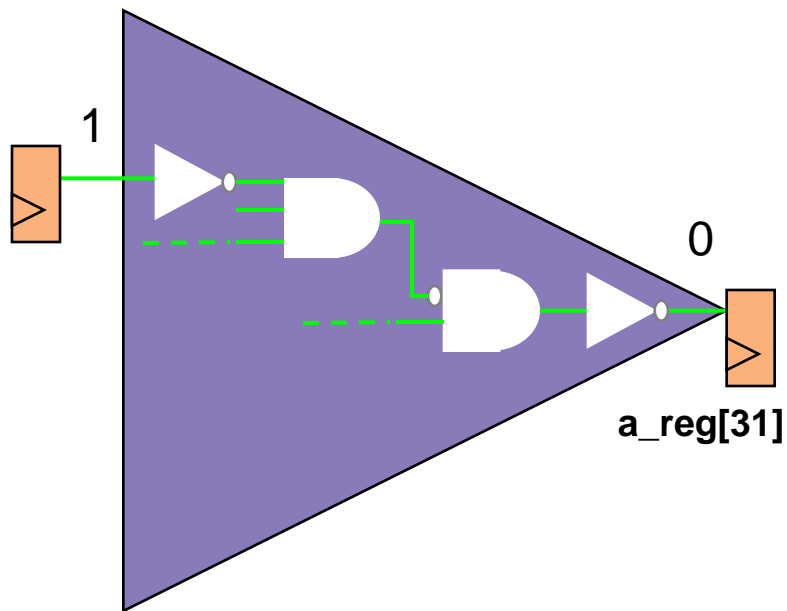


- ▶ Passing Cone
- ▶ Failing Cone
- ▶ Unmatched Cone

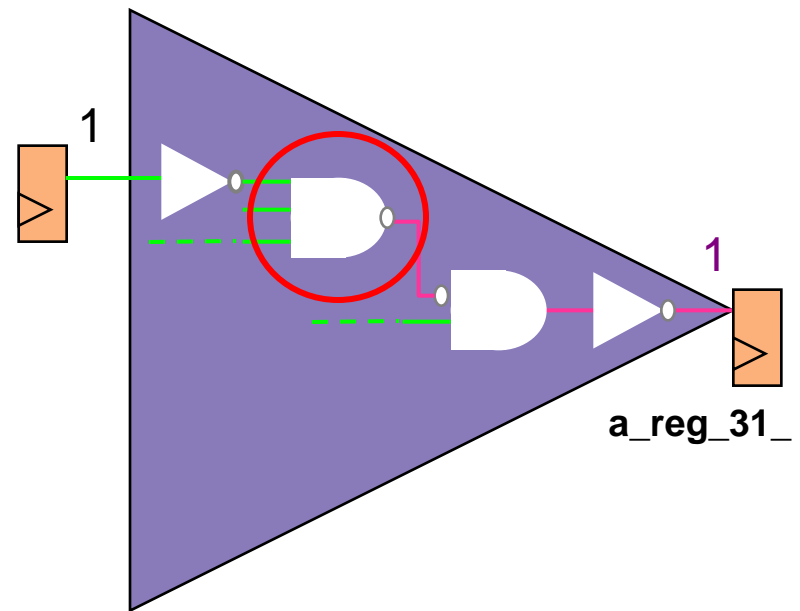
The Debug Cycle

Isolation of Implementation Errors

Reference Design Cone



Implementation Design Cone



Agenda

- Introduction to Equivalence Checking
- Using Formality
 - Flow Overview
 - Guidance
 - Read
 - Setup
 - Match
 - Verify
 - Debug
- Documentation and Help

Invoking Formality

- Typical Formality TCL script execution:

```
% fm_shell -f runme.fms |tee runme.log
```

- Starting the GUI from UNIX:

```
% formality
```

or

```
% fm_shell -gui -f runme.fms |tee runme.log
```

- Starting the GUI within a batch session:

```
fm_shell (setup)> start_gui
```

- To view other invocation options:

```
% fm_shell -help
```


Files that Formality Generates

- Record of commands issued
 - fm_shell_command.log
- Logfile of informational messages
 - formality.log
- Working files
 - FM_WORK directory
 - fm_shell_command.lck and formality.lck
 - Formality automatically deletes all working files when you exit (gracefully)

Formality Setup File

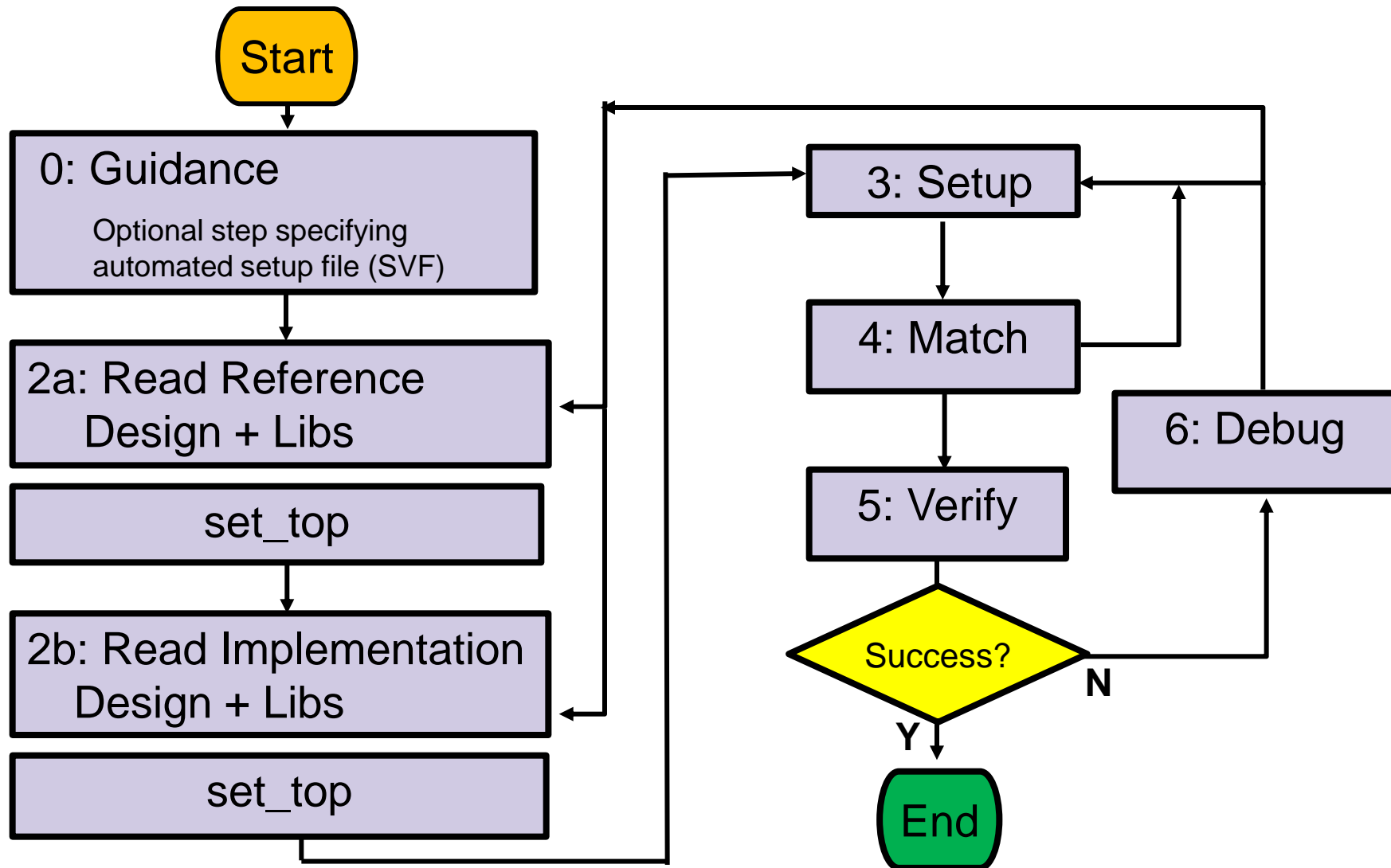
- Formality reads `.synopsys_fm.setup` when invoked
- Typical setup file contains commands like:

```
set search_path `.` ./lib ./netlists ./rtl"  
alias h history
```
- Reads from all of the following locations:
 1. Formality installation directory:
 `<formality_root>/admin/setup/.synopsys_fm.setup`
 2. Your home directory
 3. Current working directory
- Cumulative effect of all three files

Agenda

- Introduction to Equivalence Checking
- Using Formality
 - Flow Overview
 - Guidance
 - Read
 - Setup
 - Match
 - Verify
 - Debug
- Documentation and Help

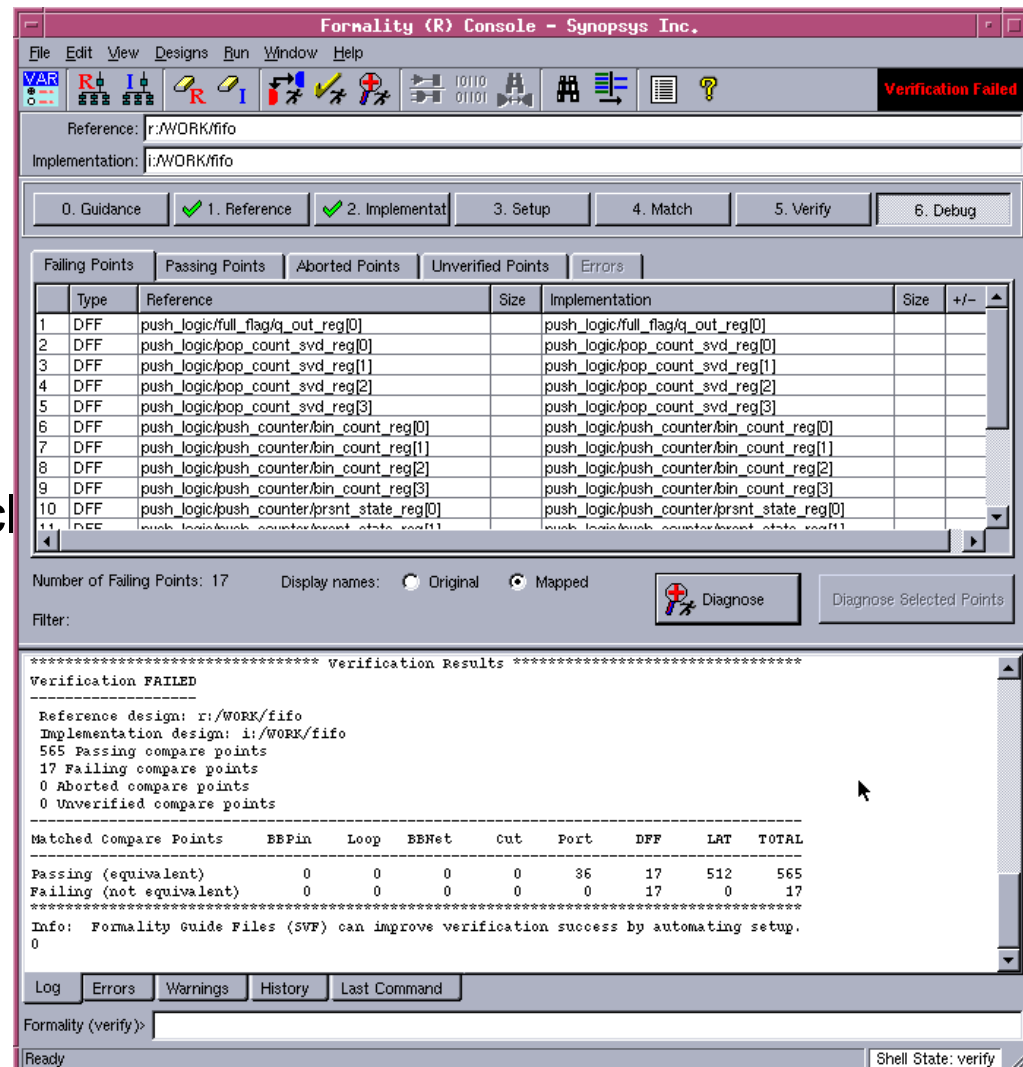
Formality Flow Overview



The Formality GUI

Recommended for New Users

- Guides you through the flow
- Context sensitive help
- Tabs for each step of the flow
- If you skip a step, you are informed
- You do not have to remember Tcl syntax
- Translates mouse clicks to Tcl for you
- GUI Preferences stored in `~/.synopsys.fmg`



Basic Formality Script

#Step 1: Guidance

```
set_svf default.svf
```

#Step 2a: Read Reference Design

```
read_verilog -r alu.v
```

```
set_top alu
```

#Step 2b: Read Implementation Design

```
read_db -i lsi_10k.db
```

```
read_verilog -i alu.fast.vg
```

```
set_top -auto
```

#Step 3: Setup

```
#No setup required here
```

#Steps 4 & 5: Match and Verify

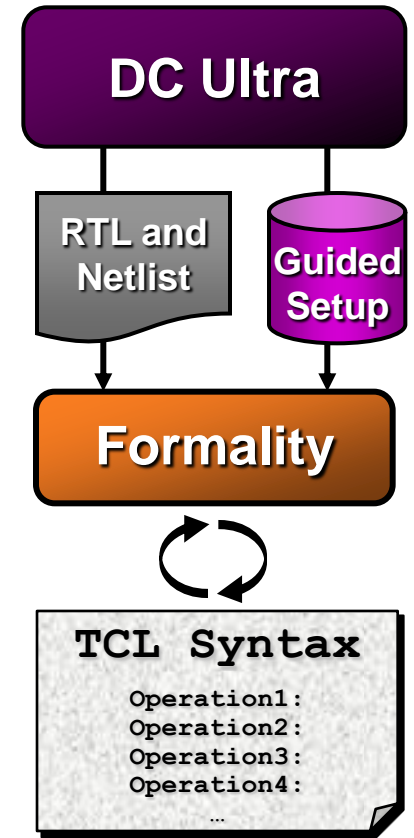
```
verify
```

Agenda

- Introduction to Equivalence Checking
- Using Formality
 - Flow Overview
 - Guidance
 - Read
 - Setup
 - Match
 - Verify
 - Debug
- Documentation and Help

What is Guidance?

- Guided Setup file (SVF)
- Hint file passed from DC to Formality
 - Automatically generated in DC
 - Contains both setup and guidance information
 - Reduces user setup effort and errors
 - Removes unnecessary verification iterations
- All SVF data is implicitly or explicitly proven in Formality, or it will not be used
- Using the SVF flow is recommended (but optional)



SVF Whitepaper:

http://synopsys.com/products/verification/formality_guided_setup.pdf

Guidance File Contents

- SVF contains these types of information:
 - Object name changes
 - Constant register optimizations
 - Duplicate and merged registers
 - Multiplier and divider architecture types
 - Datapath transformations
 - FSM re-encoding (must be enabled in Formality to be used)
 - Retiming
 - Register phase inversion

Using SVF with DC and Formality

- DC creates “default.svf” file automatically
 - Optional DC command usage `set_svf file.svf`
- Formality uses same command reading-in SVF information: `set_svf file.svf`
 - Can read-in one file, multiple files, or directory
 - SVF guidance commands specified by design name
 - Automatically determines multiple SVF file processing order
 - Places “formality_svf” in working directory
 - Creates ASCII text version “svf.txt” inside

Using Feature: Auto Setup Mode

- Variable `set synopsys_auto_setup true`
 - Assumptions made in DC will also be made in FM
 - Increases out-of-the-box (OOTB) verification success rate
 - Set auto setup variable before doing `set_svf file.svf`
- Works with or without SVF, does more with SVF
 - Handles undriven signals
 - RTL interpretation
 - Auto-enable clock-gating and auto-disable scan (requires SVF)
- All SVF passed variables and commands can be overwritten by user
 - Transcript summary shows variable settings
 - Variables will take the last value that was set

What Auto Setup Mode Does

- Performs all of the following automatically:
`set hdlin_ignore_parallel_case false`
`set hdlin_ignore_full_case false`
`set verification_set_undriven_signals 0:X`
`set hdlin_error_on_mismatch_message false`
`set svf_ignore_unqualified_fsm_information false`
`set verification_verify_directly_undriven_output false`
- DC places additional setup information in SVF
 - Clock-gating notification
 - Disabling scan mode

Benefits of Auto Setup Mode

- Enhances Formality's ease-of-use
- Reduces the need for debugging
 - A large percentage of failing verifications are “false failures” caused by incorrect or missing setup in Formality
- Improves user productivity
 - Dramatically reduces manual setup
- Simplifies overall verification effort

Agenda

- Introduction to Equivalence Checking
- Using Formality
 - Flow Overview
 - Guidance
 - Read
 - Setup
 - Match
 - Verify
 - Debug
- Documentation and Help

Read Commands

- Formality input formats:
 - Verilog (synthesizable subset) - `read_verilog`
 - VHDL (synthesizable subset) - `read_vhdl`
 - SystemVerilog (synthesizable subset) - `read_sverilog`
 - Synopsys Milkyway - `read_milkyway`
 - Synopsys binary files - `read_db, read_ddc`
- Designs are read into containers
 - `r` # default reference container
 - `i` # default implementation container
 - `container containerID` # Other container name
- Link top-level of design with `set_top`
 - Load all required designs and libraries prior to executing `set_top`
 - Must complete elaboration of each container before loading subsequent containers

Reading in Technology Libraries

- Verilog Simulation Libraries
 - Use “vcs” option with read_verilog command
 - Example: `read_verilog -i top.gv -vcs "-y ./lib +libext+.v"`
 - Older command: `read_simulation_library`
 - Example: `read_sim -i -lib mylib {./udps/*.v}`
- Synopsys binary files (db)
 - Use `read_db`
 - Example: `read_db -i lsi_10k.db`
 - Shared technology libraries
 - Subsequent containers will have access to this library
- Pure RTL does not require a library
- Instantiated DesignWare
 - Set variable `hdlin_dwroot` to top-level of DC software tree

Linking and Referencing Designs

- After reading in the source files, use `set_top` to elaborate or link the design and designate the top-level module
 - Use `set_top` only one time per container
 - When using default container (“r” and “i”) the `set_top` command automatically designates which design is reference or implementation
 - When using non-default names, specify which container is ref or impl
 - `set_reference_design`
 - `set_implementation_design`
- After `set_top` has been completed
 - TCL variable `$ref` specifies the reference design
 - TCL variable `$impl` specifies the implementation design
- Format of `$ref` and `$impl` is:
 - ContainerName:/Library/Design
 - Examples:
r:/DESIGN/chip
i:/WORK/alu_0

Reference Design

```
fm_shell (setup)> read_verilog -r alu.v  
fm_shell (setup)> set_top -auto
```

- `read_verilog` loads design into container
 - The “-r” signifies the (default) reference container
- This script does not load a technology library into “r”
 - The file `alu.v` is pure RTL (no mapped logic)
- `set_top -auto` finds and links the top-level module
 - `set_top` uses the current container (“r”)
 - The top-level module found by Formality is “alu”
 - Since the current container is “r” Formality automatically sets the `set_reference_design` variable (`$ref`) to `r:/WORK/alu`
 - WORK is the default library name

Implementation Design

```
fm_shell (setup)> read_verilog -i alu.vg  
fm_shell (setup)> read_db -i class.db  
fm_shell (setup)> set_top alu_0
```

- **read_verilog** loads the implementation design
 - The “-i” signifies the (default) implementation container
- **read_db** loads the technology library class.db
 - Since “-i” specified this library is visible only in the container “i”
- **set_top** links top-level module “alu_0”
 - Script reads both design and technology library before set_top
 - **set_top** uses the current container (“i”)
 - Since the current design is “i” Formality automatically sets the implementation design variable (**\$impl**) to i:/WORK/alu_0
 - WORK is the default library name
 - The script specifies that the top level module is “alu_0”

Simulation-Style Verilog Read

- `read_verilog` supports VCS style switches
- `read_verilog -r top.v -vcs "switches"`

where `"switches"` include:

`-y <directory_name>`

Search `<directory_name>` for unresolved modules

`-v <file_name>`

Search `<file_name>` for unresolved modules

`+libext+<extension>`

Look at files with this extension, typically `".v"` or `".h"`

`+define+ :` Define values for Verilog parameters

`+incdir <dirname> :` Directory containing ``include` files

`-f <file_name>` VCS option file supported

- Can use `"-vcs"` only once per container

Common Errors during Read

- Simulation/synthesis mismatch messages:
*Warning: /global/wwas/training/lab1/rtl/DCT8_final.vhd line 1059
Default initial value of signal will be ignored (FMR_VHDL-1002)
Error: RTL interpretation messages were produced during read.
Verification results may disagree with a logic simulator. (FM-089)*
- By default, Formality is conservative in RTL interpretation
- Stops processing when there is a difference between simulation and synthesis
- Continue processing by converting these error messages into warning messages:
 - > `set hdlin_warn_on_mismatch_message "FMR_VHDL-1002 ..."`
 - > `hdlin_report_messages`
- Use this variable before reading in the RTL into a container
- Note: If these mismatch issues are in your designs, make sure that these conditions are investigated before taping-out your design
- Auto Setup Mode
 - `set hdlin_error_on_mismatch_message false`

Instantiated DesignWare

- Set variable `hdlin_dwroot` to top-level of DC installation
 - Example of transcript when variable is not set:

```
fm_shell (setup)> printvar hdlin_dwroot
hdlin_dwroot      = ""
fm_shell (setup)> read_ver -r chip.v -vcs "-v tsc6000.v"
Loading verilog file '/users/boston/brandall/labs/fm_2005.09_working/fm/lab7a/chip.v '
No target library specified, default is WORK
Current container set to 'r'
1
fm_shell (setup)> set_top chip
Setting top design to 'r:/WORK/chip'
Status:   Elaborating design chip    ...
Status:   Elaborating design pll     ...
Status:   Elaborating design ff      ...
Status:   Elaborating design dp      ...
Error: Design 'DW01_add' is not defined but cell '/WORK/dp/u0' references it. (FM-234)
Error: Design 'DW01_add' is not defined but cell '/WORK/dp/u1' references it. (FM-234)
Error: Design 'DW01_add' is not defined but cell '/WORK/dp/u2' references it. (FM-234)
Error: Design 'DW01_add' is not defined but cell '/WORK/dp/u3' references it. (FM-234)
Status:   Elaborating design cntrl   ...
Error: Failed to set top design to 'r:/WORK/chip' (FM-156)
0
fm_shell (setup)>
```

Reading and Writing Containers

- Command: **write_container**
 - Saves all design information in the current “elaborated” state (including libraries) to a file
 - Must run **set_top** before saving a container
- Command: **read_container**
 - Restores a design that is fully elaborated
 - No need to run **set_top** on restored containers
- Recommend to save containers before running **match**
- Containers can be used with any version of Formality

```
fm_shell> write_container -replace -r ref
fm_shell> read_container -r ref.fsc
```

Save and Restore Session

- Use after verification to save the current state of Formality
- Commonly used to debug failing verification in a separate Formality run
- Saved sessions are NOT portable across Formality releases

```
fm_shell> save_session -replace mysession_file  
fm_shell> restore_session mysession_file.fss
```


LAB TIME

- Lab1: Basic Formality Flow

Agenda

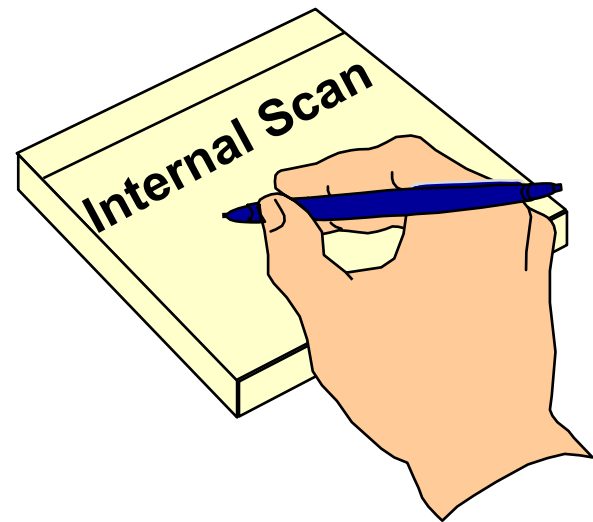
- Introduction to Equivalence Checking
- Using Formality
 - Flow Overview
 - Guidance
 - Read
 - Setup
 - Match
 - Verify
 - Debug
- Documentation and Help

Setup Needed for Verification

- Need to guide matching and verification
 - Recommended: Use SVF file
 - Essential for retiming and register inversion
- Design transformations that may need setup:
 - Internal Scan
 - Boundary Scan
 - Clock-gating
 - Clock Tree Buffering
 - Finite State Machine (FSM) Re-encoding
 - Black-boxes
- Auto Setup Mode handles most setup automatically
`set synopsys_auto_setup true`

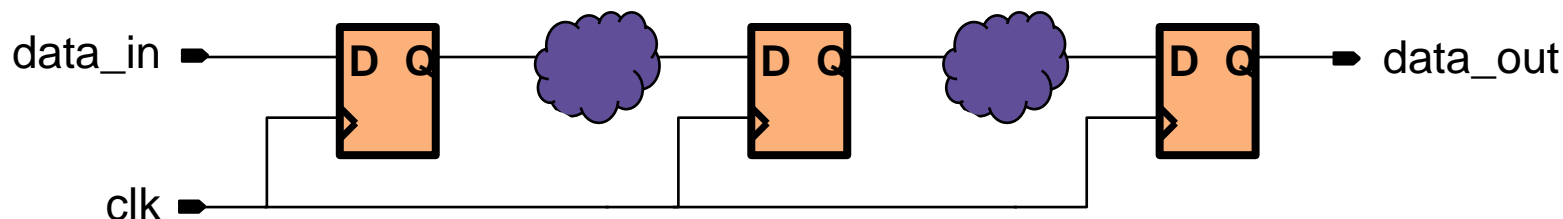
Internal Scan: What Is It?

- Implemented by DFT Compiler
 - Replaces flip-flops with scan flops
 - Connects scan flops into shift registers or “scan chains”
- The scan chains make it easier to set and observe the state of registers internal to a design for manufacturing test



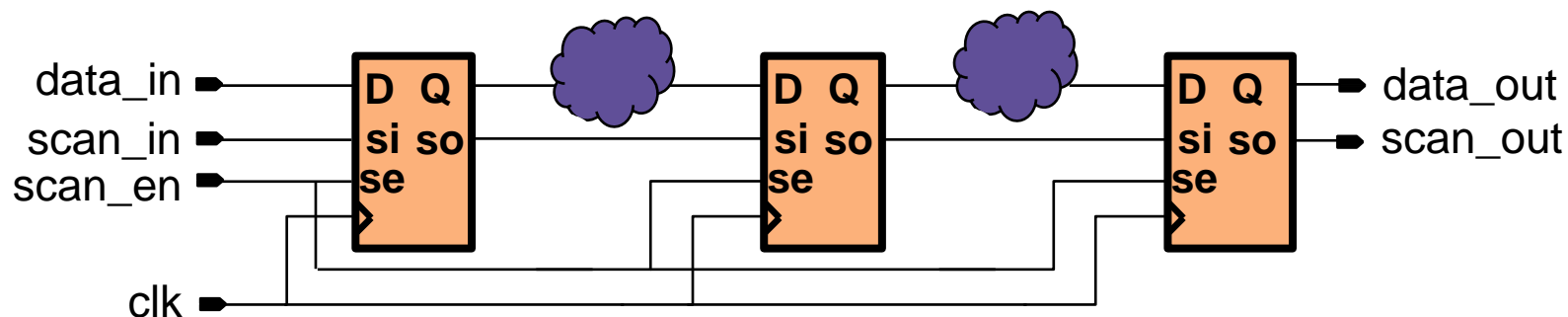
Internal Scan: Why It Requires Attention

The additional logic added during scan insertion changes the combinational function



Pre-Scan

Post-Scan



Internal Scan: How to Deal With It

- Determine which ports disable the scan circuitry
 - Default for DFT Compiler is test_se
- Set those ports to the inactive state using the `set_constant` command

```
fm_shell (setup) > set_constant i:/WORK/TOP/test_se 0
```

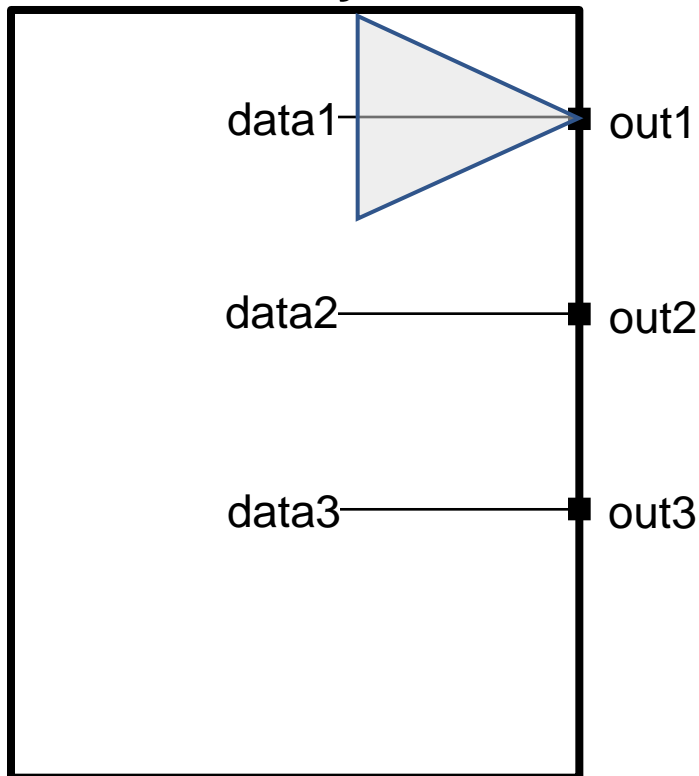
Boundary Scan: What Is It?

- Boundary scan is similar to internal scan in that it involves the addition of logic to a design:
 - The added logic makes it possible to set and or observe the logic values at the primary inputs and outputs (the boundaries) of a chip
 - Used in manufacturing test at board and system level
 - Added by BSD Compiler
- Boundary scan is also referred to as
 - The IEEE 1149.1 specification
 - JTAG

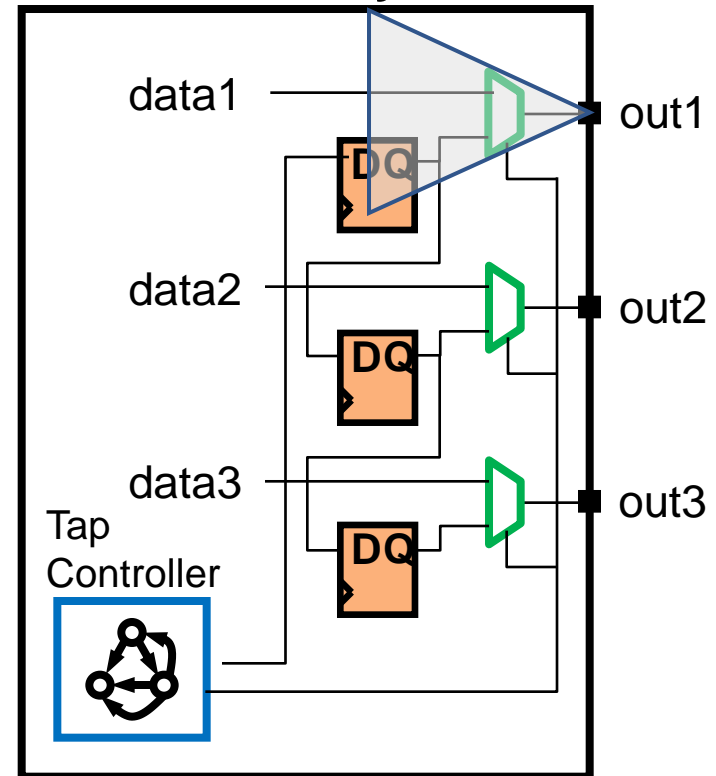
Boundary Scan: Why It Requires Attention

- The logic cones at the primary outputs are different
- The logic cones driven by primary inputs are different
- The design has extra state holding elements

Pre-Boundary Scan



Post-Boundary Scan



Boundary Scan: How to Deal With It

- Disable the Boundary scan:
 - If the design has an optional asynchronous TAP reset pin (such as TRSTZ or TRSTN), use `set_constant` on the pin to disable the scan cells
 - If the design has only the 4 mandatory TAP inputs (TMS, TCK, TDI and TDO), then force an internal net of the design using the `set_constant` command

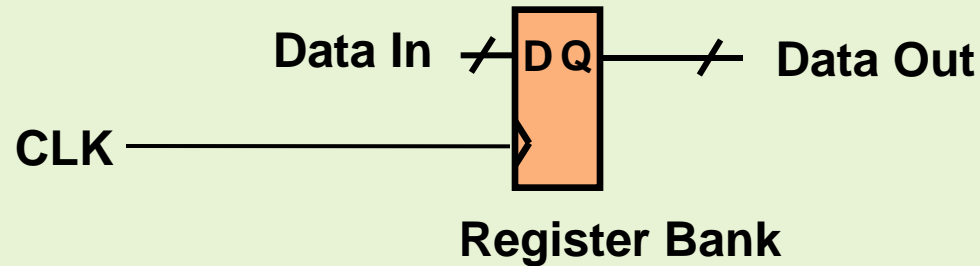
```
fm_shell (setup) > set_constant i:/WORK/TOP/TRSTZ 0  
fm_shell (setup) > set_constant i:/WORK/alu/somenet 0
```

Clock-Gating: What Is It?

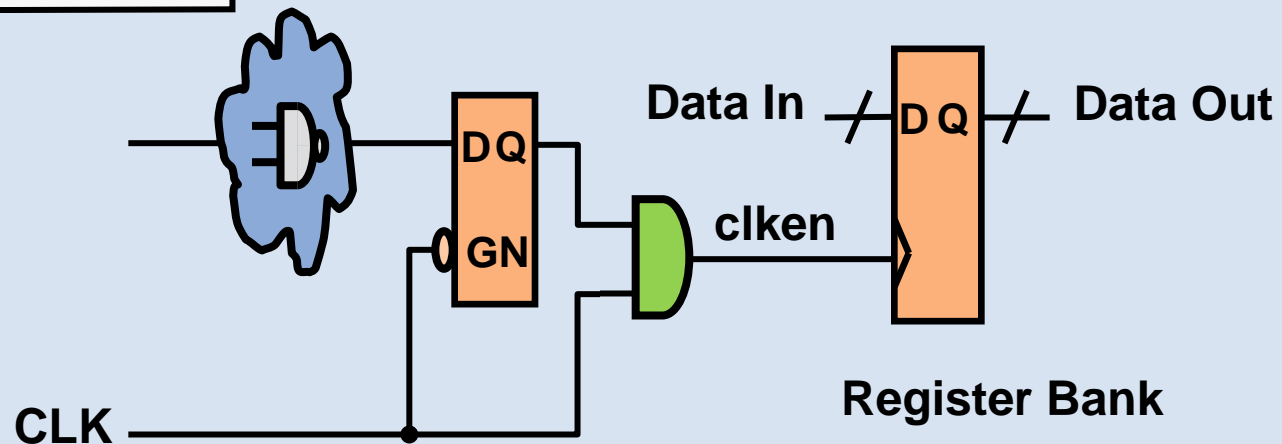
- Added by Power Compiler
- Clock-gating is the addition of logic in a register's clock path which disables the clock when the register output is not changing
- The purpose of clock-gating is to save power by not clocking register cells unnecessarily

Clock-Gating

Before Clock-Gating



After Clock-Gating



Clock-Gating: Why Is It an Issue?

- Without intervention failing compare points will result
 - A compare point will be created for the clock-gating latch
 - This compare point does not have a matching point in the other design and will fail
 - The logic feeding the clock input of the register bank has changed
 - The register bank compare points will fail

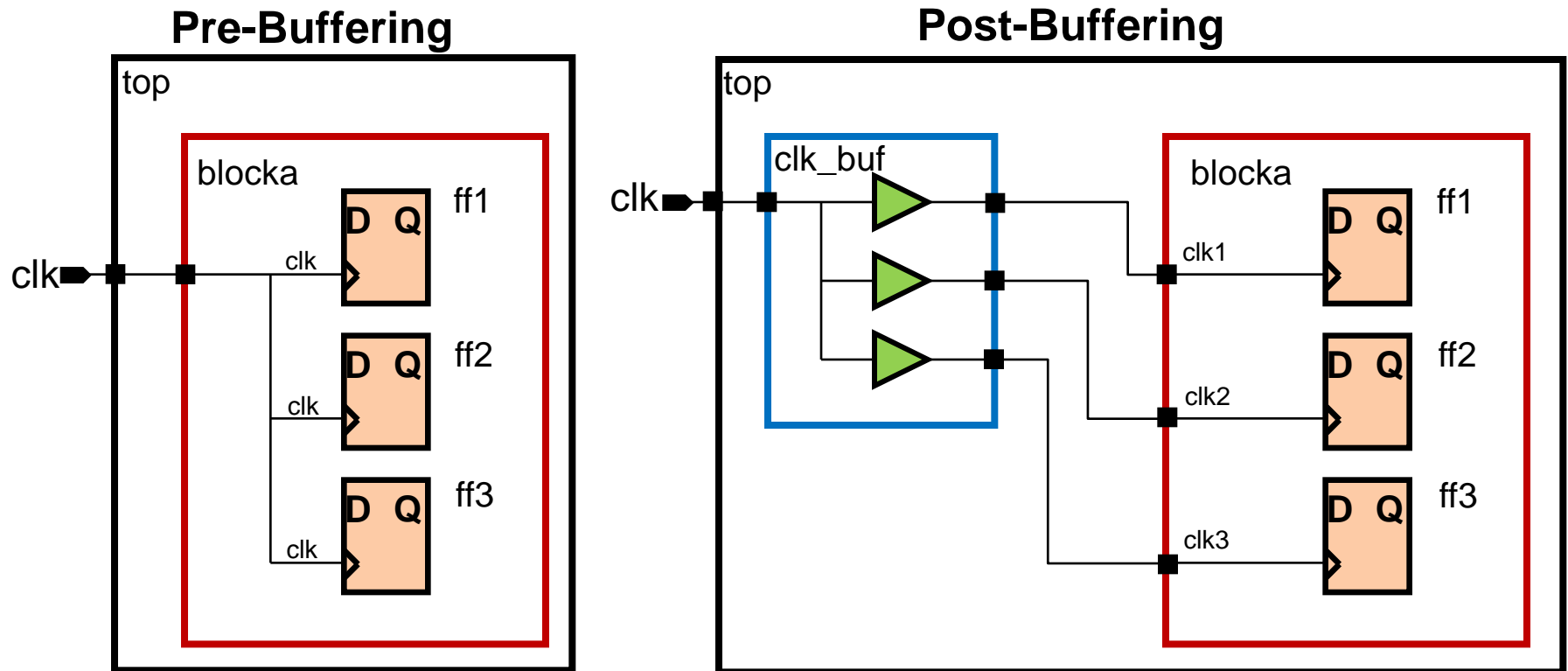
Clock-Gating: How to Deal with It

- Use variable `set verification_clock_gate_hold_mode any`
 - Typical setting that allows for both low and high clock-gating in the same design
- If clock-gating net also drives primary outputs or black-box inputs use option `"collapse_all_cg_cells"`
 - Use `set_clock` command to identify the primary input clock net if clock-gating cells do not drive any clk-pin of a DFF

```
fm_shell (setup)> set verification_clock_gate_hold_mode any
```

Clock Tree Buffering

Clock tree buffering is the addition of buffers in the clock path to allow the clock signal to drive large loads



Clock Tree Buffering: How to Deal With It

- Verification at the top level requires no setup
- When verifying at “blocka” sub-block level use `set_user_match` command to show buffered clock pins are equivalent

```
fm_shell (setup)> set_reference_design r:/WORK/blocka
fm_shell (setup)> set_implementation_design i:/WORK/blocka
fm_shell (setup)> set_user_match r:/WORK/blocka/clk \
i:/WORK/blocka/clk1 \
i:/WORK/blocka/clk2 \
i:/WORK/blocka/clk3
fm_shell (setup)> verify
```

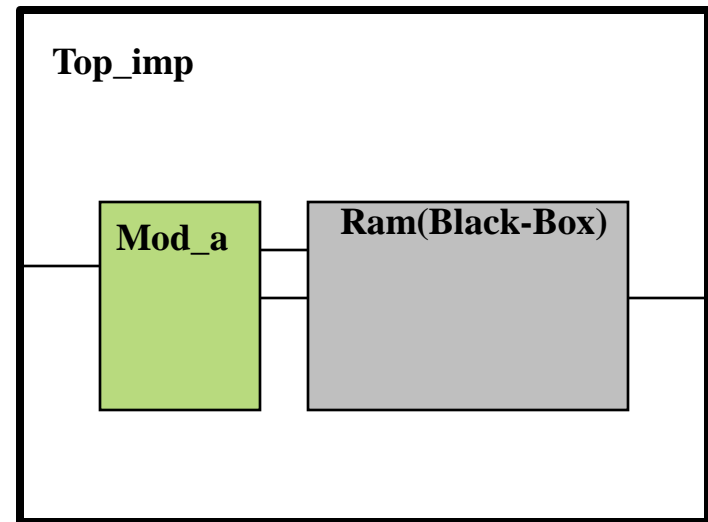
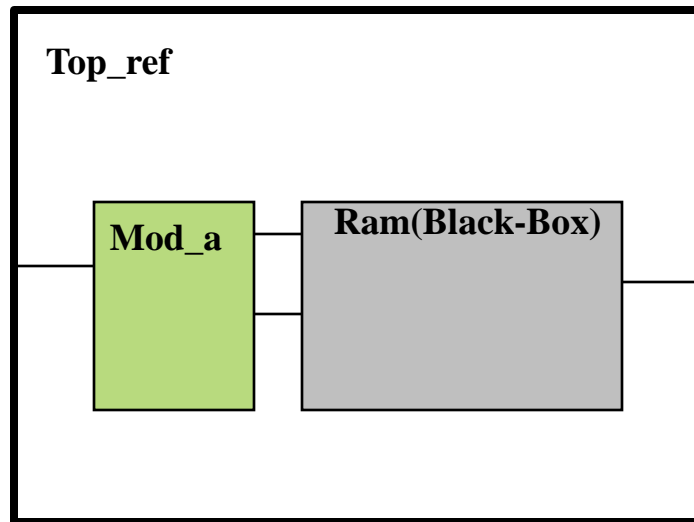
Finite State Machine Re-encoding

- Verify re-encoding in the automated setup file (SVF) is correct
 - View the ASCII file: `./formality_svf/svf.txt`
- Enable the use of this setup information in Formality

```
fm_shell> set svf_ignore_unqualified_fsm_information false
```


Black-Boxes

- A Black-Box is a module or entity which contains no logic
 - These are modules that are not verified
 - Analog circuitry
 - Memory devices
 - Need to match up between reference and implementation



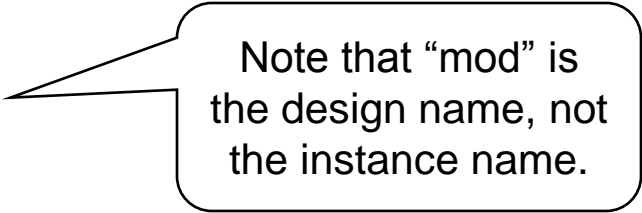
How Are Black-Boxes Created?

- Modules read in that have only I/O port declarations
- Library .db cells that contain port and timing arcs only
 - Typically a memory
- Missing a piece of design when using this variable
`set hdlin_unresolved_modules black_box`
- Usage of other variable when reading in designs
`set hdlin_interface_only "SRAM* dram16x8"`
 - Any module beginning with SRAM and the dram16x8 module will become a black-box
- Declare a sub-design as a black-box
`set_black_box designID`
- Command `report_black_boxes` shows list of black-boxes

Setting a Black-Box Property

- Can manually set or unset the black_box property on any reference or implementation sub-design
 - Black-box inputs will become new compare points
 - Black-box outputs will become inputs to logic cone
- Commands:

```
set_black_box r:/WORK/mod  
remove_black_box r:/WORK/mod  
report_black_boxes
```



Note that “mod” is the design name, not the instance name.

Finding Black-Boxes using fm_shell

```
fm_shell (setup)> report_black_box
Information: Reporting black boxes for current reference and implementation designs.
(FM-184)
```

```
|
| Legend:
|
|     Black Box Attributes
|         s = Set with set_black_box command
|         i = Module read with -interface_only
|         u = Unresolved design module
|         e = Empty design module
|         * = Unlinked design module
|
```

```
#####
```

```
####      DESIGN LIBRARY - i:/WORK
```

```
#####
```

```
Design Name                               Attributes
```

```
-----
```

```
sRAM01                                   s
```

```
#####
```

```
####      DESIGN LIBRARY - r:/WORK
```

```
#####
```

```
Design Name                               Attributes
```

```
-----
```

```
sRAM01                                   i
```

LAB TIME

- Lab2: Recognizing Simulation/Synthesis Mismatch Errors

Agenda

- Introduction to Equivalence Checking
- Using Formality
 - Flow Overview
 - Guidance
 - Read
 - Setup
 - Match
 - Verify
 - Debug
- Documentation and Help

Matching Compare Points

```
fm_shell (setup)> match
```

- **match** is an optional command
 - The **verify** command will run matching
 - Recommendation:
 - For interactive work use the explicit match command for feedback
 - Omit the match command from scripts to reduce runtime
- Name matching algorithms are used first
- Remaining unmatched points matched by signature analysis
 - Includes structural techniques
 - May be turned off (but not recommend)
- Any remaining unmatched points then reported
 - User can specify compare rules or explicit matches
- Automated setup flow (SVF) improves name matching performance and completion
 - Match points by name without user intervention

GUI Unmatched Point Report

Formality (R) Console - Synopsys Inc.

File Edit View Designs Run Window Help

Verification Failed

Reference: r:/WORK/tv80s
Implementation: i:/WORK/tv80s

0. Guidance 1. Reference 2. Implementation 3. Setup 4. Match 5. Verify 6. Debug

Compare Rule Setup User Match Setup Matched Points Unmatched Points Summary

	Reference Object	Type
1	i_tv80_core/ACC_reg[0]	DFF
2	i_tv80_core/ACC_reg[1]	DFF
3	i_tv80_core/ACC_reg[2]	DFF
4	i_tv80_core/ACC_reg[3]	DFF
5	i_tv80_core/ACC_reg[4]	DFF
6	i_tv80_core/ACC_reg[5]	DFF
7	i_tv80_core/ACC_reg[6]	DFF
8	i_tv80_core/ACC_reg[7]	DFF
9	i_tv80_core/ALU_Op_r_reg[0]	DFF
10	i_tv80_core/ALU_Op_r_reg[1]	DFF
11	i_tv80_core/ALU_Op_r_reg[2]	DFF

Create a compare rule

Set User Match

	Implementation Object	Type
1	ACC_reg_0_	DFF
2	ACC_reg_1_	DFF
3	ACC_reg_2_	DFF
4	ACC_reg_3_	DFF
5	ACC_reg_4_	DFF
6	ACC_reg_5_	DFF
7	ACC_reg_6_	DFF
8	ACC_reg_7_	DFF
9	ALU_Op_r_reg_0_	DFF
10	ALU_Op_r_reg_1_	DFF
11	ALU_Op_r_reg_2_	DFF

Select two points and set match

Number of unmatched reference points: 314 Number of unmatched implementation points: 351 Display names: Original Mapped

Filter on reference list: Filter on implementation list:

Run Matching

Compare Rules

- When names change in predictable ways write a compare rule
- Use SED syntax to translate names in one design to the corresponding names in the other design:

```
fm_shell (match)> set_compare_rule $ref \
                    -from {i_tv80_core} -to {}
fm_shell (match)> match
```

Agenda

- Introduction to Equivalence Checking
- Using Formality
 - Flow Overview
 - Guidance
 - Read
 - Setup
 - Match
 - **Verify**
 - Debug
- Documentation and Help

Verify Implementation Design

- Runs Formality's verification algorithms on compare points
 - Formality deploys many different solvers
 - Each solver uses a different algorithm to prove equivalence or non-equivalence
- Four possible results:
 - Succeeded: implementation is equivalent to the reference
 - Failed: implementation is not equivalent to the reference
 - Logic difference or setup problem
 - Inconclusive: no points failed, but analysis is incomplete
 - May be due to timeout or complexity
 - Not run: a problem earlier in the flow prevented verification from running at all

Verify Implementation Design (cont)

- For each matched pair of compare points Formality
 - Confirms same functionality of logic cones
 - Marks point as “passed”

Or,

- Determines that the functionality is different between logic cones
 - Finds one or more “counter examples” that shows different response at compare point
 - Marks the compare point as “failed”
- By default all valid compare points are verified
 - Constant registers are not verified
 - “Unread” compare points are not verified by default

Verify Example

```
fm_shell (match)> verify
```

- Verification is incremental
 - Verification can continue again after being stopped
 - You may match additional compare points manually and continue with verification
 - To force verification of entire design: **verify -restart**
- Options:
 - Verification of single compare point
 - Verification against a constant
 - Use **set_dont_verify** to exclude points from verification

Controlling Verification Runtimes

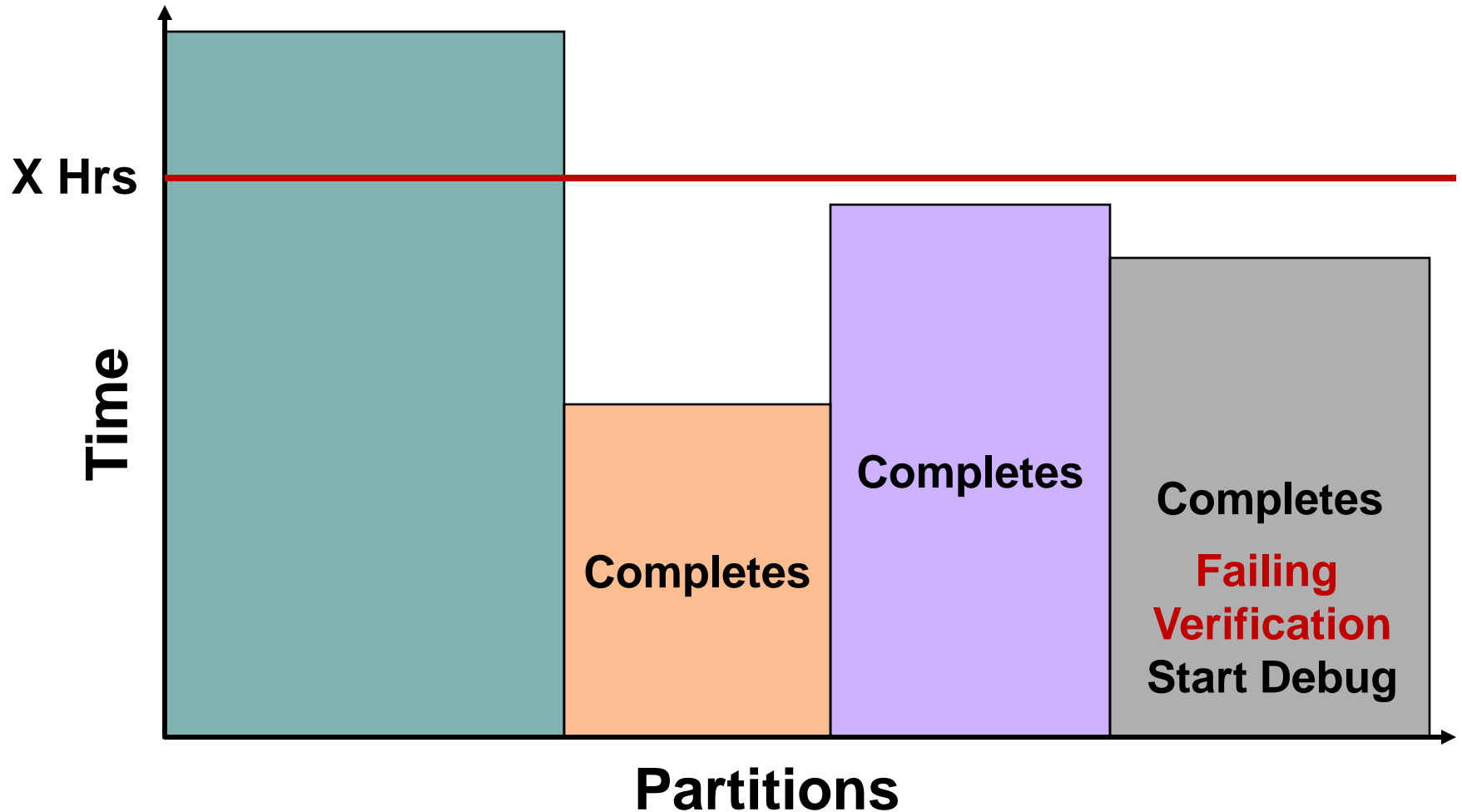
- `set verification_timeout_limit hrs:min:sec`
 - Halts entire verification after a specified time
 - No default limit (0 means no timeout)
 - Remaining unverified compare points are not attempted
- `set verification_failing_point_limit number`
 - Halts verification after specified number of compare points fail (default is 20 failing points)
 - Enables you to correct for missing setup
 - Enables you to begin debug on failing points

Controlling Verification Runtimes

- `set verification_effort_level [super_low | low | medium | high]`
 - Specifies amount of effort spent solving a compare point (default level is high)
 - Enables you to quickly get a status on the majority of your design
 - Enables you to begin debug on failing points
- `set verification_partition_timeout_limit hrs:min:sec`
 - A partition is a collection of similar logic cones
 - Verification of a partition stops after allotted time and verification moves onto next partition

Partition Time Out Limit

Find Errors Faster



Hierarchical Verification

- Command `write_hierarchical_verification_script`
 - Formality generates TCL script that performs hierarchical verification on current reference and implementation designs
 - Helpful for debugging large and hard-to-verify designs
 - Usage:

```
...  
set_top i:/WORK/top  
set_constant $impl/test_se 0  
write_hier -replace -level 3 myhierscript  
source myhierscript.tcl  
quit
```

- View results in file `fm_myhierscript.log`
- Formality will create one session file, by default, if verification fails on a sub-design

Distributed Verification

- Time-to-results advantage for long runs
 - Use on designs greater than 250K gates and
 - Taking over 2 hours to verify
- How it works:
 - Formality divides the design into partitions (groupings of compare points) and distributes the verification workload
- Does not require a license
 - Up to four distributed verifications allowed
 - No additional purchase of licenses
- Example:

```
add_distributed_processors frodo bilbo gandalf*2
```

- Support for LSF and GRD

Agenda

- Introduction to Equivalence Checking
- Using Formality
 - Flow Overview
 - Guidance
 - Read
 - Setup
 - Match
 - Verify
 - Debug
- Documentation and Help

Typical Formality Script

```
set search_path ". ./rtl ./lib ./netlist"
set synopsys_auto_setup true
set hdlin_dwroot /tools/syn/A-2007.12-SP2

set_svf default.svf

read_verilog -r "fifo.v gray_counter.v \
               pop_ctrl.v push_ctrl.v rs_flop.v"
set_top fifo

read_db -i tcb013ghpwc.db
read_verilog -i fifo.vg
set_top fifo

# set_constant $impl/test_se 0

verify
```

Debugging Problem 1

Find the problem in this script:

```
read_verilog -r alu.v
set_top alu

read_verilog -i alu.fast.vg
set_top alu
read_db -i class.db

verify
```

Debugging Problem 2

Find the problem in this script:

```
read_verilog -r alu.v

read_db -i class.db
read_verilog -i alu.fast.vg

set_top r:/WORK/alu
set_top i:/WORK/alu

verify
```

Debugging

Check for Warning Signs

- Check RTL interpretation messages in transcript
 - Were `full_case` or `parallel_cases` pragmas interpreted?
- Check for “unmatched input” error messages
- Check for black-box warnings in the transcript
- Check for unmatched compare points
 - Same number of unmatched compare points in reference and implementation?
 - Unmatched compare points only in implementation? Clock-gating latches?
- Is there a setup problem?
 - Not accounting for an intention design difference
 - Example: Did you disable scan?
- Try using Auto Setup Mode

```
set synopsys_auto_setup true
```

Graphical Debugging

Display Schematic Logic Cones

Formality (R) Console - Synopsys Inc.

File Edit View Designs Run Window Help

Reference: r:/WORK/mR4000
Implementation: i:/WORK/mR4000

0. Guidance 1. Reference 2. Implementation 3. Setup 4. M

Failing Points	Passing Points	Aborted Points	Unverified Points	Errors
Type	Reference	Size	Implementation	
28 DFF	Instruction_reg_8_	3	Instruction_reg_8_	
29 DFF	Instruction_reg_9_	3	Instruction_reg_9_	
30 DFF	Instruction_reg_7_	3	Instruction_reg_7_	
31 DFF	Instruction_reg_0_	3	Instruction_reg_0_	
32 DFF	Instruction_reg_3_	3	Instruction_reg_3_	
33 DFF	ALUOp_reg_1_	195	ALUOp_reg_1_	
34 DFF	ALUSelB_reg_0_	195	ALUSelB_reg_0_	
35 DFF	IRWrite_reg	195	IRWrite_reg	
36 DFF	ALUSelB_reg_1_	195	ALUSelB_reg_1_	36

Number of Failing Points: 252 Display names: ☐ Original ☒ Mapped

Filter:

Implementation design: i:/WORK/mR4000
98 Passing compare points
252 Failing compare points
0 Aborted compare points

Log Errors Warnings History Last Command

Formality (verify)>

Ready Shell State: verify

- Show Logic Cones
- Show Selected Cone Sizes
- Show All Cone Sizes
- Show Patterns
- Show Matching Tool
- View Reference Object
- View Implementation Object
- View Reference Source
- View Implementation Source
- Set Don't Verify
- Diagnose
- Diagnose Selected Points
- Copy Reference Name
- Copy Implementation Name

Failing Patterns

- Formality automatically creates set of vectors to illustrate failure
 - These counter examples are called failing patterns
 - “Proof” of non-equivalence is done mathematically
- Failing patterns
 - Cones are annotated with these vectors

Failing Pattern Window

Required vector bit values shown in red

Patterns - q_out_reg[0]/q_out_reg[0]

File Edit View Window Help

Compare point values for vector 1

R q_out_reg[0] (DFF, Loading 1) SL 1 Const SD 1 CLK 0

I q_out_reg[0] (DFF, Loading 0) D 1 CP 0 TI 0 TE 1

Reference	Implementation	+/-
1	test_se	
2	pop	
3	pop_clk	
4	... logic/empty_flag/q_out_reg[0]	
5	pop_logic/push_count_svd_reg[0]	
6	pop_logic/push_count_svd_reg[1]	
7	pop_logic/push_count_svd_reg[2]	
8	pop_logic/push_count_svd_reg[3]	

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	0	0	0	0	0	0	0	0	0	0	0	1
3	0	0	0	0	0	0	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0	0	0	0
5	0	1	0	1	0	0	1	1	0	0	0	0	0
6	1	1	1	0	1	0	0	1	1	1	0	0	0
7	1	1	0	1	1	1	0	1	1	0	0	0	0
8	1	0	0	1	1	0	0	1	0	0	0	0	0

Ready

Non-required bit values shown in black

- Allows quick identification of issues with setup and matching
 - This example shows scan enable signal which makes verification fail when it has a “1” value
 - Try using “set_constant \$impl/test_se 0” to get a successful verification

Pattern and Cone Windows

Failing Compare Point values annotated

The image shows two windows from a Synopsys design tool. The top window, titled "Patterns - q_out_reg[0]/q_out_reg[0]", displays a comparison of point values for vector 1. It shows a table with Reference and Implementation columns, and a grid of values (1s and 0s) for various signals. The bottom window, titled "Cone Schematics - q_out_reg[0]/q_out_reg[0]", shows a logic cone schematic for the same signal. A red box highlights a specific point in the schematic, and a callout box provides details about the point, including its name, type, and loading status.

Patterns - q_out_reg[0]/q_out_reg[0]

File Edit View Window Help

Compare point values for vector 1

R q_out_reg[0] (DFF, Loading 1) SL 1 Const SD 1 CLK 0

I q_out_reg[0] (DFF, Loading 0) D 1 CP 0 TI 0 TE 1

Reference	Implementation	+/-
1	test_se	
2	pop	
3	pop_clk	
4	..._logic/empty_flag/q_out_reg[0]	
5	pop_logic/push_count_svd_reg[0]	
6	pop_logic/push_count_svd_reg[1]	
7	pop_logic/push_count_svd_reg[2]	
8	pop_logic/push_count_svd_reg[3]	

Ready

Cone Schematics - q_out_reg[0]/q_out_reg[0]

Schematic Edit View Window Help

Color Mode Error Candidates

Reference>>> pop_logic/empty_flag/q_out_reg[0]

pop

pop_logic/empty_flag/q_out_reg[0]

SL 0 1 pop_logic/empty_flag/q_out[0]

SD 1

CLK 0

pop_logic/empty_flag/q_out_reg[0] (SEQ) Loading 1

SL = synchronous load (enable)

SD = synchronous data

Ready

Vector annotated in schematic (logic cone view)

Running Diagnosis: Error-ID

What is it?

- Use “Diagnose” or “Diagnose Selected Points”
- Reports the logic that needs to be changed to correct the error
- Report example:

Single error detected in implementation

Recommended error candidate:

Cell X/Y/Z/prop_99

Alternate error candidates:

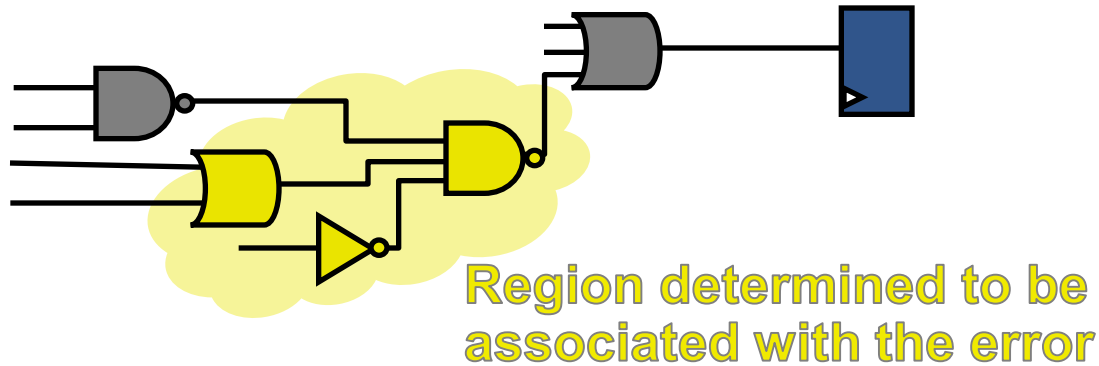
Cell X/Y/Z/procmon_out

Cell X/Y/Z/BW2_INV_D65327

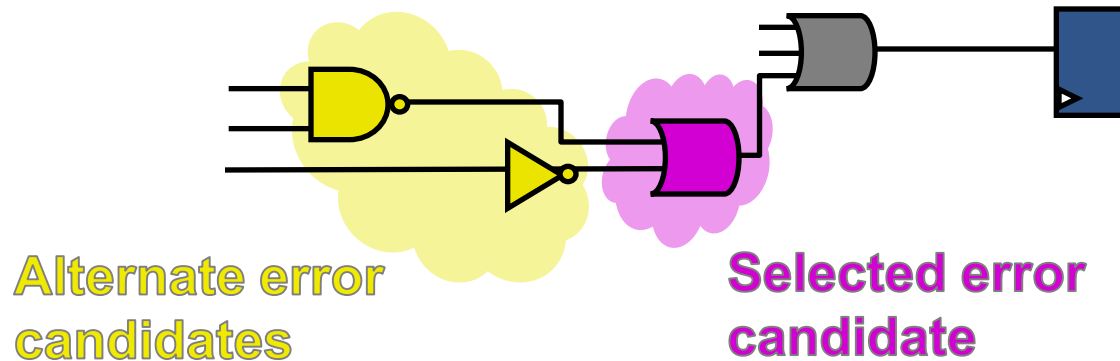
Error-ID Graphical View

Implementation Analyzed

Reference



Implementation



Viewing RTL Source From Schematics

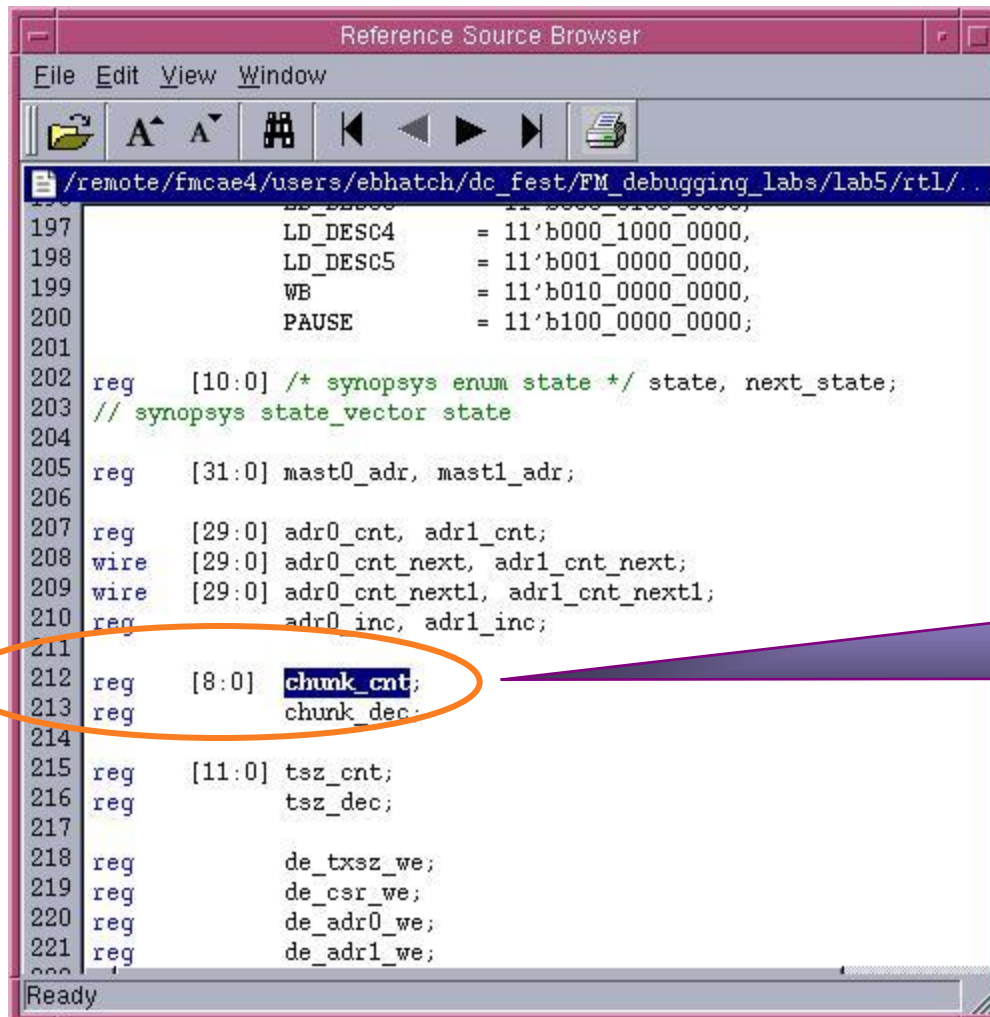
The screenshot displays the Synopsys design environment. The main window shows a schematic diagram with various logic components. A specific cell, `u2/chunk cnt reg[0]`, is highlighted with a white box. A green callout box with the text "Select Cell, Popup Menu, and View Source" points to this cell. A context menu is open on the right side of the screen, listing various actions. The "View Source" option is highlighted in blue. The menu also includes options like "View Object", "Show Logic Cones", and several "Find" and "Copy" actions. The bottom status bar shows "Ready".

Select Cell, Popup Menu, and View Source

Context Menu Options:

- Zoom Full (F)
- View Source**
- View Object
- Show Logic Cones
- Find Compare Point
- Find Error Candidates (E)
- Find Diagnosed Matching Region (R)
- Find Net Driver (D)
- Find Net Load (L)
- Find By Name (F3)
- Find Matching (M)
- Find X Sources (X)
- Copy Name
- Remove Non-Controlling (F5)
- Remove Subcone (F6)
- Isolate Subcone (F7)
- Isolate Error Candidates (F8)
- Return Subcone (Ctrl+F6)
- Group All By Parent (Ctrl+G)
- Group Selected By Parent (G)
- Ungroup Selected (U)
- Undo Last (Z)
- Revert (Shift+Z)

Source Code Browser



```
Reference Source Browser
File Edit View Window
[Icons]
/remote/fmcae4/users/ebhatch/dc_fest/FM_debugging_labs/lab5/rtl/...
197 LD_DESC4 = 11'b000_1000_0000,
198 LD_DESC5 = 11'b001_0000_0000,
199 WB = 11'b010_0000_0000,
200 PAUSE = 11'b100_0000_0000;
201
202 reg [10:0] /* synopsys enum state */ state, next_state;
203 // synopsys state_vector state
204
205 reg [31:0] mast0_adr, mast1_adr;
206
207 reg [29:0] adr0_cnt, adr1_cnt;
208 wire [29:0] adr0_cnt_next, adr1_cnt_next;
209 wire [29:0] adr0_cnt_next1, adr1_cnt_next1;
210 reg adr0_inc, adr1_inc;
211
212 reg [8:0] chunk_cnt;
213 reg chunk_dec;
214
215 reg [11:0] tsz_cnt;
216 reg tsz_dec;
217
218 reg de_txsz_we;
219 reg de_csr_we;
220 reg de_adr0_we;
221 reg de_adr1_we;
222
Ready
```

Gate and line number highlighted

Agenda

- Introduction to Equivalence Checking
- Using Formality
 - Flow Overview
 - Guidance
 - Read
 - Setup
 - Match
 - Verify
 - Debug
- Documentation and Help

Help For Commands and Variables

- Three important commands for getting help:

`printvar`

- Displays the value of a Tcl variable
- Accepts wildcards

`help`

- Displays brief description of a Formality command
- Accepts wildcards

`man`

- Displays detailed information about a Formality command, Tcl variable, warning, or error message
- Does not accept wildcards

Help Examples

```
fm_shell (setup)> help report_con*
```

```
report_constants          # Report user specified constants
report_constraint         # Reports on the defined constraints
```

```
fm_shell (setup)> read_verilog -r r400.v
```

```
Error: Can't open file r400.v (FM-016)
```

```
0
```

```
fm_shell (setup)> man FM-016
```

messages	N. Messages	Command
Reference		

NAME

FM-016 (error) Can't open file %s.

DESCRIPTION

The specified file does not exist or cannot be created.

WHAT NEXT

Verify that you specified the correct filename and that you have permission to open and create files.

Command Editing and Completion

- The TCL shell supports powerful command editing and completion capabilities
 - Command completion with “Tab”
 - Use up and down arrow keys for moving through command stack

```
fm_shell (setup)> read_v  
  read_verilog read_vhdl  
fm_shell (setup)> read_verilog
```



Hit Tab key



Enter “e” and hit
Tab key

Other Help Sources

- Outside of Formality:
 - Formality documentation (release notes and user guides) on the web
 - <http://www.synopsys.com>
 - Login to SolvNet
 - Documentation and Media->Documentation on the Web
 - Formality News
 - Receive emails highlighting release content or topics of general interest
 - Subscribe through SolvNet

Reference Methodology Guides

- DC reference scripts show you the latest recommended synthesis and verification methodology
- Top-down and hierarchical synthesis flow covering: DCT, Power Compiler, DFT Compiler, IC Compiler, and Formality
- Download from SolvNet:
<https://solvnet.synopsys.com/retrieve/021023.html>
(Search for “DC-RM”)

Verilog RTL Interpretation

- Paper: “full_case parallel_case”, the Evil Twins of Verilog Synthesis, by Cliff Cummings

http://www.sunburst-design.com/papers/CummingsSNUG1999Boston_FullParallelCase.pdf

- Important information about using synthesis pragmas in your Verilog RTL for CASE statements

Links to Formality Information

- Subscribe to Formality Tech Bulletin
Go to “My preferences” on SOLVNET, select the subscriptions tab
- Retiming Verification Whitepaper
<http://www.synopsys.com/cgi-bin/verification/pdfr1.cgi?retimingwp.pdf>
- Datapath Verification Whitepaper
http://www.synopsys.com/cgi-bin/verification/pdfr1.cgi?formalityeq_wp.pdf
- Error-ID Whitepaper
<http://www.synopsys.com/cgi-bin/verification/pdfr1.cgi>
- Guide File Whitepaper
http://www.synopsys.com/products/verification/formality_guided_setup.pdf
- Hier-IQ Whitepaper
http://www.synopsys.com/products/verification/hier-iq_fs.html
- Datasheet
http://www.synopsys.com/products/verification/formality_ds.html

Command Summary – 1

fm_shell [-gui]	Launch Formality from Unix command line
formality	Launch Formality GUI from Unix command line
start_gui	Launch Formality GUI from Formality shell
printvar	Display current value of a Tcl variable
set	Set value of a Tcl variable
help [-verbose]	Get brief help on a Formality command
man	Get detailed help on a Formality command, variable, or error message.

Command Summary - 2

set_svf	Specify guidance file
read_verilog	Read Verilog source files
read_sverilog	Read SystemVerilog source files
read_vhdl	Read VHDL source files
read_db, read_ddc, read_mdb	Read Synopsys binary formats
set_top	Link the design
write_container	Save current container
read_container	Reads container file created by write_container

Command Summary - 3

match	Match compare points
verify	Check designs for equivalence
save_session	Save “snapshot” of current work
restore_session	Restore session file created by save_session

LAB TIME

- Lab3: Missing Part of the Design
- Lab4: Beginning Debug

Quiz

- a) Name three types of objects that can be compare points?
- b) How do you load a guidance file from Design Compiler into Formality?
- c) After reading in the reference design, what sub-step must you perform before reading in the implementation design?
- d) How would you load the class.db as a shared technology library?
- e) What Tcl variable must you set before reading a RTL design containing instantiated DesignWare?
- f) How would you find black-boxes from the command line?
- g) Explain how to override simulation/synthesis mismatch warnings in Formality.
- h) How would you manually match individual points from GUI or from command line?
- i) Which variable would you use to speed up matching for regular name changes (Ex: all_reg[12] -> all_regx12x)?
- j) How would you generate a list of failing (passing) points?
- k) What is an “unread” point?
- l) How would you exclude compare points from verification?
- m) Which variable specifies the number of failing points at which Formality will stop verification?
- n) How does Formality handle undriven signals by default?

Answers:

- a) Name three types of objects that can be compare points?
Primary outputs, registers/latches, black-box input
- b) How do you load a guidance file from Design Compiler into Formality?
set_svf default.svf
- c) After reading in the reference design, what sub-step must you perform before reading in the implementation design?
set_top design_name
- d) How would you load the class.db as a shared technology library?
read_db class.db
- e) What Tcl variable must you set before reading a RTL design containing instantiated DesignWare?
hdlin_dwroot
- f) How would you find black-boxes from the command line?
report_black_boxes
- g) Explain how to override simulation/synthesis mismatch warnings in Formality.
set hdlin_warn_on_mismatch_message "list_of_error_codes"

Answers:

h) How would you manually match individual points from GUI or from command line?

`set_user_match r:/WORK/top/regA123 i:/WORK/top/regApple_1_2_3`

i) Which variable would you use to speed up matching for regular name changes (Ex: all_reg[12] -> all_regx12x)?

`append name_match_filter_chars "x"`

j) How would you generate a list of failing (passing) points?

`report_passing_points, report_failing_points`

k) What is an “unread” point?

An unread point does not affect other compare points or outputs. These could be spares, or compare points which are blocked by constants.

l) How would you exclude compare points from verification?

`set_dont_verify ...`

m) Which variable specifies the number of failing points at which Formality will stop verification?

`verification_failing_point_limit`

n) Which variable deals with handling undriven signals?

`verification_set_undriven_signals` – the default is “Binary:X”

SYNOPSYS®

Predictable Success