Chapter 4 : Synthesis (Optional)

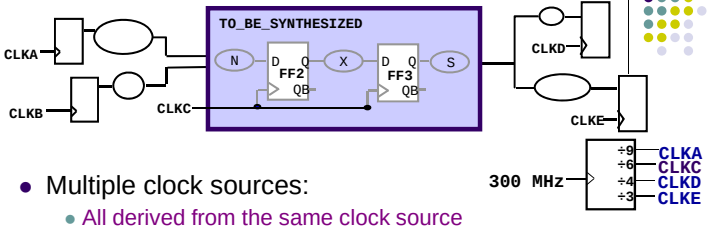
- Constraining Designs
- Design Optimization
- Compile Strategies

Timing Constrains

Multiple Clock Designs

Multiple Cycle Designs

Synchronous Multiple Clock Designs



- Multiple clock sources:
  - All derived from the same clock source
  - Some clocks do not have a corresponding clock port on our design
  - Multiple constraints on a single port
- How to constrain the design
  - Define the clock(s), using virtual clocks if necessary.
  - Specify I/O delays relative to the clock(s).

    Design Compiler will determine which clock imposes the most restrictive constraint on the design.

Virtual Clock

- What is a Virtual Clock?

  Clocks in the environment of the design to be synthesized that:
  - Are defined clock objects within Design Compiler's memory
  - Do not clock any sequential devices within the current_design
  - Serve as references for input or output delays
- How to create a virtual clock?

  It is the same as defining a clock, but you must specify a name and do not specify a clock pin or port:

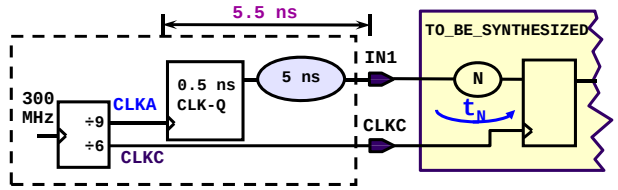
```
create_clock  -name VCLK  -period 20
```

Must be named

No source pin or port!

Multiple Clock Input Delay: Example

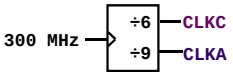
```
create_clock -period 30 -name CLKA
create_clock -period 20 [get_ports CLKC]

set_input_delay 5.5 -clock CLKA -max [get_ports IN1]
```

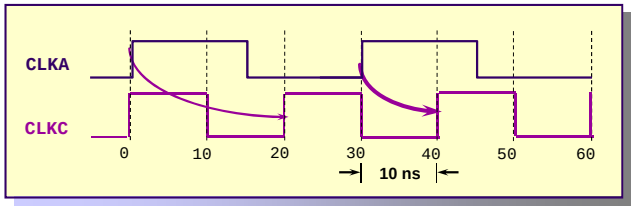


What is t_N ?

Multiple Clock Input Delay

Example -Waveforms



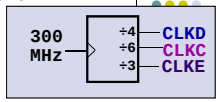


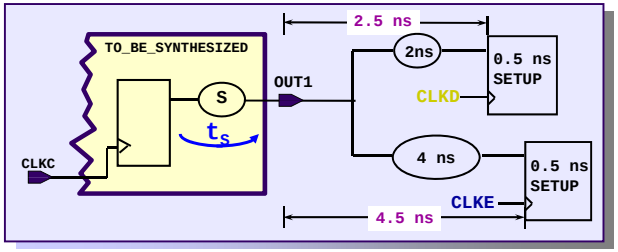
For the example shown, input logic cloud of TO_BE_SYNTHESIZED must meet:

$t_N < 10 - 5.5 - t_{setup}$

Multiple Clock Output Delay: Example

```
create_clock -period [expr 1.0/75*1000] -name CLKD
create_clock -period 10     -name CLKE
create_clock -period 20 [get_ports CLKC]

set_output_delay -max 2.5 -clock CLKD [get_ports OUT1]
set_output_delay -max 4.5 -clock CLKE -add_delay [get_ports OUT1]
```

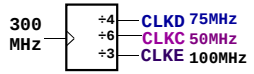


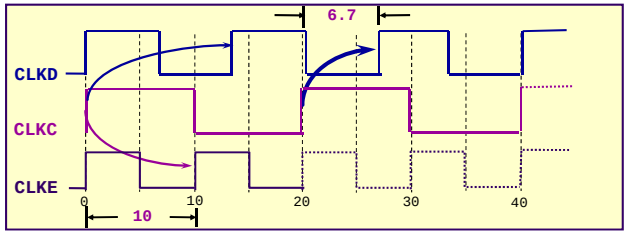


What is t_s ?

Multiple Clock Output Delay

Example -Waveforms





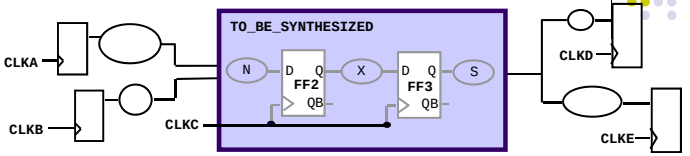
For the example shown, output logic cloud of TO_BE_SYNTHESIZED must meet:

$t_S < 10 - 4.5$ AND $t_S < 6.7 - 2.5$

Hints for Multiple Clock Designs

- By definition, all clocks used with Design Compiler are synchronous
- You cannot create asynchronous clocks with the `create_clock` command
- DC will determine every possible data launch/data capture time, and synthesize to the most conservative
- DC builds a common base period for all clocks

Asynchronous Multiple Clock Designs



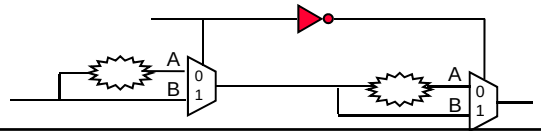
- All Asynchronous
  - Clocks do not have a corresponding clock port on our design

Synthesizing with Asynchronous Clocks

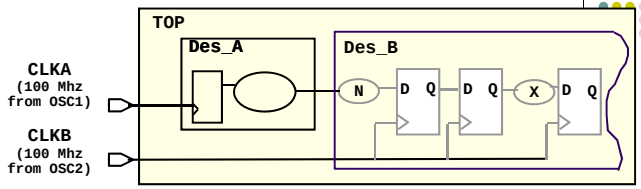
- It is your responsibility to account for the metastability:
  - Instantiate double-clocking, metastable-hard Flip-Flops
  - dual-port FIFO, etc
- You must then disable timing-based synthesis on any path which crosses an asynchronous boundary:
  - This will prevent DC from wasting time trying to get the asynchronous path to “meet timing”

The False Paths

- False paths are called timing exceptions in Design Compiler.
- A false path is a path for which you will ignore timing constraints
- Use the **`set_false_path`** command to disable timing-based synthesis on path-by-path basis
  - Useful for:
    - Constraining asynchronous paths
    - Constraining logically false paths



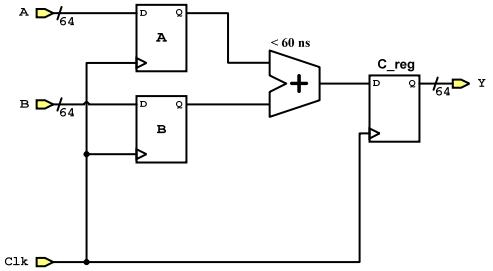
set_false_path: Example




```
current_design TOP

# Make sure register-register paths meet timing
create_clock -period 10  [get_ports CLKA]
create_clock -period 10  [get_ports CLKB]

# Don't optimize logic crossing clock domains
set_false_path -from [get_clocks CLKA] -to [get_clocks CLKB]
set_false_path -from [get_clocks CLKB] -to [get_clocks CLKA]

compile -scan
```

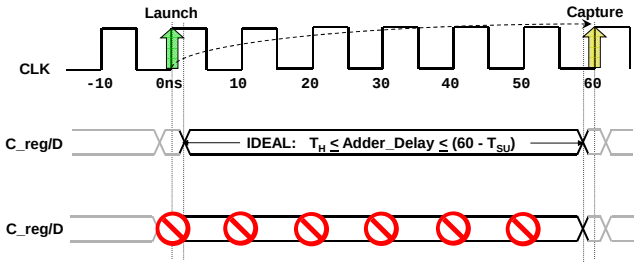
Multi-cycle Paths

Clock period is 10 ns. Per specification, the adder takes 6 clock cycles.
How do you constrain the design?



Timing with Multi-cycle Constraints

```
create_clock -period 10 [get_ports CLK]
set_multicycle_path 6 –setup -to [get_pins C_reg[*]/D]
```

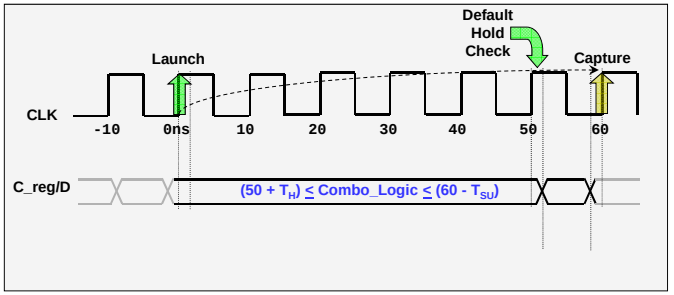


DC assumes change could occur near *any* clock edge causing metastability!

Where does DC perform hold analysis?

Default Hold Check

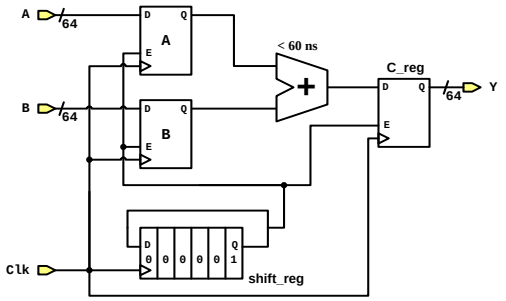
```
set_multicycle_path -setup 6 -to [get_pins C_reg[*]/D]
```



Why is hold check performed at 50 ns?

Enable Flip-Flops Prevent Metastability

Enabler allows data to change and be captured only on the desired clock edge.

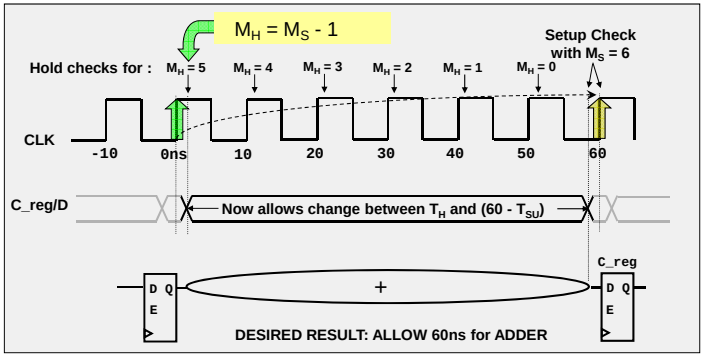


Adding enabler is the designer's responsibility!

Set the Proper Hold Constraint

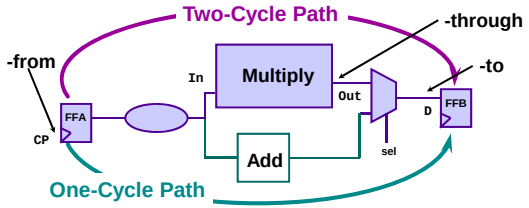
OVERRIDE!

```
set_multicycle_path  -setup 6  -to [get_pins C_reg[*]/D]
set_multicycle_path  -hold  5  -to [get_pins C_reg[*]/D]
```



Another Example

```
dc_shell-xg-t> set_multicycle_path –setup 2 -from FFA/CP \
               -through Multiply/Out -to FFB/D
dc_shell-xg-t> set_multicycle_path –hold 1 -from FFA/CP \
               -through Multiply/Out -to FFB/D
```



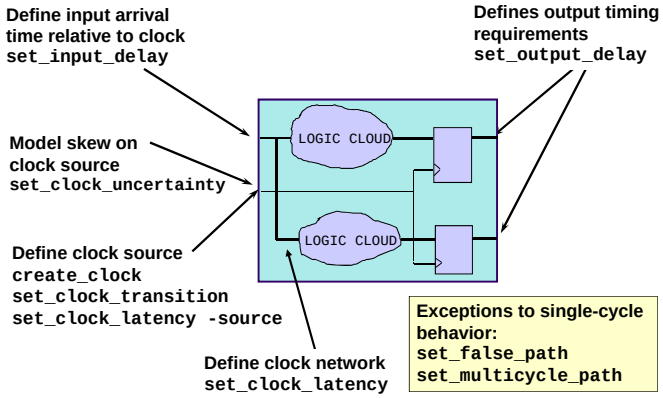
Always Check for Invalid Exceptions

No warnings are issued if an invalid exception is applied to a design.

```
dc_shell-xg-t> report_timing_requirements -ignored
```

```
Description                                Setup          Hold
-------------------------------------------------------------------
NONEXISTENT PATH                           FALSE          FALSE
        -from     { IO_PCI_CLK\
                    pclk }\
        -to       { IO_SDRAM_CLK\
                    SDRAM_CLK}
INVALID FROM OBJECT                        FALSE          FALSE
        -from     FF1/Q
```

Timing Goals Summary

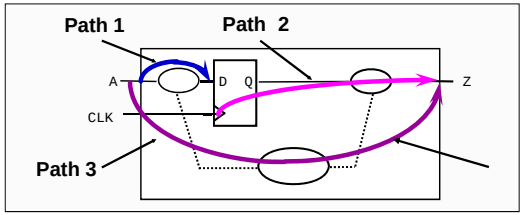


Command Summary

`set_false_path`	Remove timing constraints from particular paths
`set_multicycle_path`	Modifies the single-cycle timing relationship of a constrained path
`report_timing_requirements`	Reports timing path requirements (user attributes) and related information
`set_max_delay`	Specifies a maximum delay target for paths in the current design
`set_min_delay`	Specifies a minimum delay target for paths in the current design

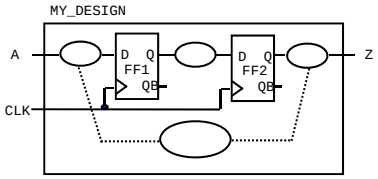
Timing Analysis

Static Timing Analysis



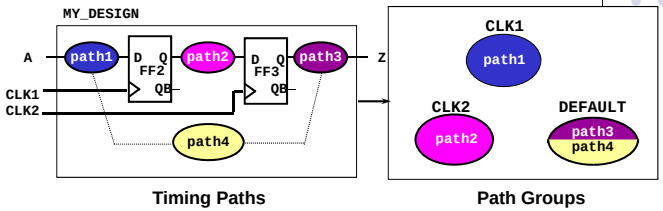
- Static Timing Analysis can determine if a circuit meets timing constraints without dynamic simulation
- This involves three main steps:
  - Design is broken down into sets of timing paths
  - The delay of each path is calculated
  - All path delays are checked to see if timing constraints have been met

Timing Paths in Design Compiler



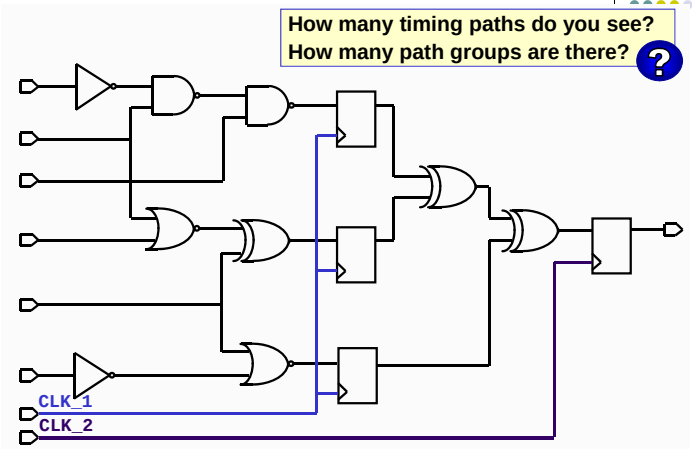
- Design Compiler breaks designs into sets of signal paths
- Each path has a startpoint and an endpoint:
  - Startpoints
    - Input ports
    - Clock pins of Flip-Flops or registers
  - Endpoints
    - Output ports
    - All input pins except clock pins of sequential devices

Organizing Timing Paths into Groups

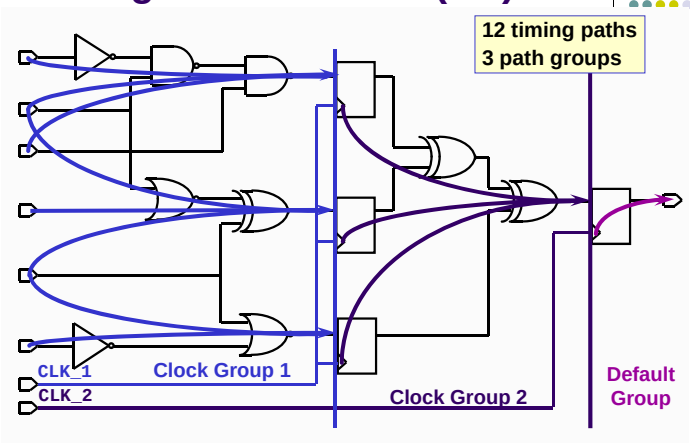


- Paths are grouped by the clocks controlling their endpoints:
  - The default path group contains all paths not captured by a clock

Timing Path Exercise (1/2)

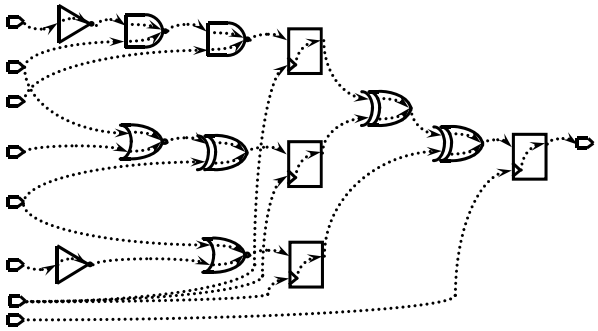


Timing Path Exercise (2/2)

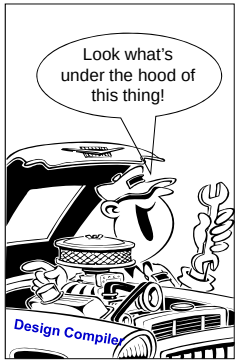


Schematic Converted to a Timing Graph

- To calculate the total delay, Design Compiler breaks each path into timing arcs:
  - Each timing arc contributes either a net or a cell delay



Components of Static Timing Analysis



What components are used in STA path delay calculations?

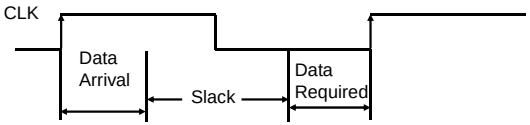
- Cell delay models:
  - Linear and Nonlinear
- Wire load models:
  - Pre-layout estimates of wire parasitics
  - How much R? How much C?
- Interconnect models: (R-C "tree type")
  - How are R and C distributed?
- Operating conditions:
  - How are the delays affected by process, voltage, and temperature?

Timing Reports

- **The `report_timing` command:**
  - Design is broken down into individual timing paths
  - Each timing path is timed out twice
    - Once for a rising edge endpoint
    - Once with a falling edge endpoint
  - The critical path (worst violator) for each clock group is found
  - A timing report for each clock group is echoed to the screen
- **A timing report has four major sections**

Major Sections in Timing Reports

- Path information section
- Path delay section
- Path required section
- Summary section. Timing margin(slack): negative indicates constraint violation.



Timing Report: Path Information Section

```
****************************************
Report : timing
         -path full
         -delay max
         -max_paths 1
Design : TT
Version: 2002.05
Date   : Fri Jun 28 16:48:52 2002
****************************************

Operating Conditions: slow_125_1.62   Library: ssc_core_slow
Wire Load Model Mode: enclosed

  Startpoint: data1 (input port clocked by clk)
  Endpoint: u4 (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  TT                 5KGATES               ssc_core_slow
```

Timing Report: Path Delay Section

Individual Contribution to Path Delay

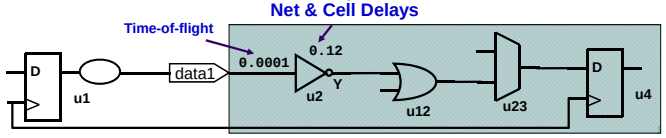
Running Total of the Path Delay

Point	Incr	Path	
clock clk (rise edge)	0.00	0.00	
clock network delay (ideal)	0.00	0.00	
input external delay	1.00	1.00	f
data1 (in)	0.00	1.00	f
u2/Y (inv1a1)	0.12	1.12	r
u12/Y (or2a1)	0.26	1.38	r
u23/Y (mx2d2)	0.23	1.61	f
u4/D (fdef1a1)	0.00	1.61	f
data arrival time		1.61	

Signal Transition

Total Delay

Net & Cell Delays



Timing Report: Path Required Section

Clock Edge

Point	Incr	Path
clock clk (rise edge)	5.00	5.00
clock network delay (ideal)	0.00	5.00
U4/CLK (fdef1a1)	0.00	5.00
library setup time	-0.19	4.81
data required time		4.81

From the Library

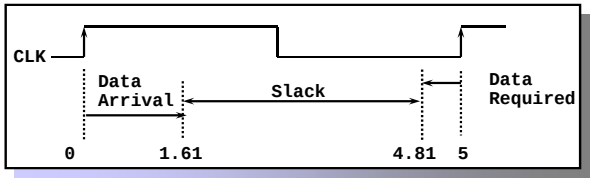
Data must be valid by this time

Timing Report: Summary Section

```
-----------------------------------------------
data required time                       4.81
data arrival time                       -1.61
-----------------------------------------------
slack (MET)                              3.20
```

Either (MET) or (VIOLATED)

Timing margin (slack): negative indicates constraint violation



Timing Report: Options

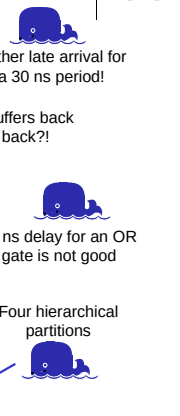
```
report_timing
            [ -delay max/min ]
            [ -to name_list ]
            [ -from name_list ]
            [ -through name_list ]
            [ -input_pins ]
            [ -max_paths path_count ]
            [ -nets ]
            [ -capacitance ]
            [ -path full_clock ]

            ...
```

Remember, the default behavior of `report_timing` is to report the path with the worst slack within each path group.

Timing Analysis: Diagnose Synthesis Results

Spot the whales in the timing report: Where are they? What are they? And why?

Point	Incr	Path
clock (input port clock) (rise edge)	0.00	0.00
input external delay	22.40	22.40 f
addr31 (in)	0.00	22.40 f
u_proc/address31 (proc)	1.08	23.48 f
u_proc/u_dcl/int_add[7] (dcl)	0.00	23.48 f
u_proc/u_dcl/U159/Q (NAND3H)	0.62	24.10 r
u_proc/u_dcl/U160/Q (NOR3F)	0.75	24.85 f
u_proc/u_dcl/U186/Q (AND3F)	1.33	26.18 f
u_proc/u_dcl/U86/Q (INVF)	0.64	26.82 r
u_proc/u_dcl/U135/Q (NOR3B)	1.36	28.17 f
u_proc/u_dcl/U136/Q (INVF)	0.49	28.67 r
u_proc/u_dcl/U100/Q (NBF)	0.87	29.54 r
u_proc/u_dcl/U95/Q (BF)	0.44	29.98 f
u_proc/u_dcl/U96/Q (BF)	0.45	30.43 r
u_proc/u_dcl/U94/Q (NBF)	0.84	31.27 r
u_proc/u_dcl/U93/Q (NBF)	0.94	32.21 r
u_proc/u_dcl/ctl_rs_N (dcl)	0.00	32.21 r
u_proc/u_ctl/ctl_rs_N (ctl)	0.00	32.21 r
u_proc/u_ctl/U126/Q (NOR3B)	1.78	33.98 f
u_proc/u_ctl/U120/Q (NAND2B)	1.07	35.06 r
u_proc/u_ctl/U99/Q (NBF)	0.88	35.94 f
u_proc/u_ctl/U122/Q (OR2B)	10.72	46.67 r
u_proc/u_ctl/read_int_N (ctl)	0.00	46.67 r
u_proc/int_cs (proc)	0.00	46.67 r
u_int/readN (int)	0.00	46.67 r
u_int/U39/Q (NBF)	1.29	47.95 r
u_int/U17/Q (INVB)	1.76	49.71 f
u_int/U16/Q (AOI21F)	2.49	52.20 r
u_int/U60/Q (AOI22B)	1.43	53.63 f
u_int/U68/Q (INVB)	1.81	55.44 r
u_int/int_flop_0/D (DFF)	0.00	55.44 r
data arrival time		55.44



Summary

Use `report_timing` to get detailed information about the critical path:

- Slack
- Setup/hold times
- Clock uncertainty
- Operating condition used
- Wire load model used
- Network delay
- Partitions
- Cell/pin/net names

Design Optimization

Pre-Compile Checklist

- ☑ Good Synthesizable HDL Code
- ☑ Good Synthesis Partitioning
- ☑ Realistic Constraints & Attributes
- ☑ False/Multicycle Paths Identified
- ☑ Wire loads Reflect Physical Placement

What Do You Do First?

1. **Satisfy the items on the checklist.**
2. **If adding margin, do not overconstrain by more than 10%.**
3. **Always, always, always (always!) start with a top-down compile.**

```
compile <-scan>
```

What Happens in a Default Compile?

- Compile stops
  - When all constrains are met
  - User interrupts – typing a Ctrl-C
  - Design Compiler reaches a point of diminishing returns – checking compiler report

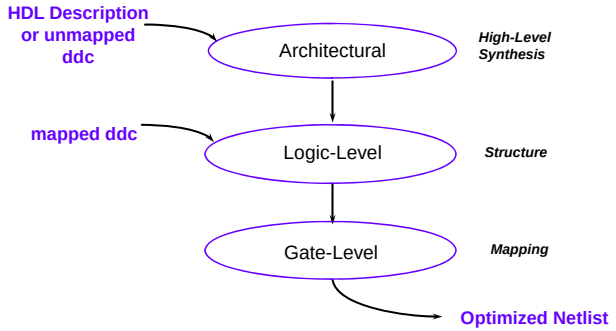
```
Beginning Delay Optimization Phase
 --------------------------------

  ELAPSED            WORST NEG  TOTAL NEG  DESIGN
    TIME      AREA     SLACK      SLACK    RULE COST      ENDPOINT
 --------- --------- ------- --------- --------- -----------------
   0:10:04   2761.7    1.38       3.20      18.1    Zro_Flag_reg/D
   0:10:05   2761.7    1.38       3.20      18.1    Zro_Flag_reg/D
   0:10:08   2761.7    1.28       3.10      18.1    Zro_Flag_reg/D
   0:10:12   2761.7    1.26       3.06      18.1    Zro_Flag_reg/D
```

Critical Path **timing violations**

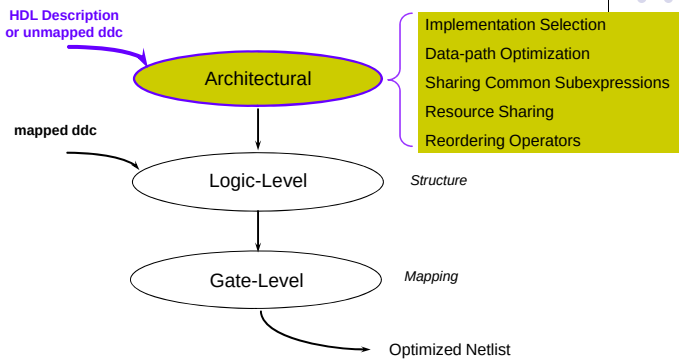
Sum of all timing violations

Three Phases of Compile

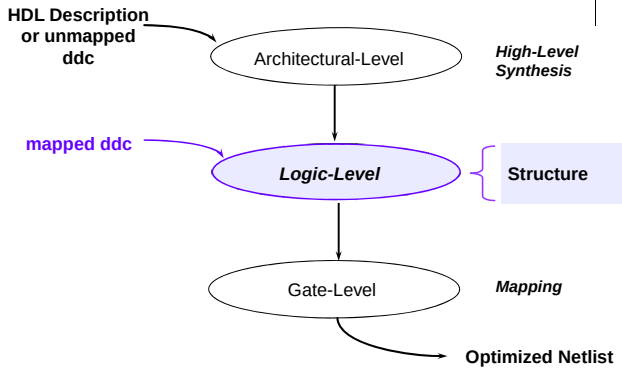
Optimization can occur at each of three levels:



Architectural Optimization



Logic-Level Optimization

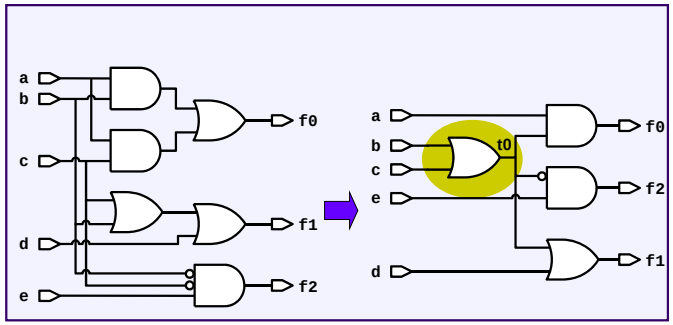


What Is Logic-Level Optimization?

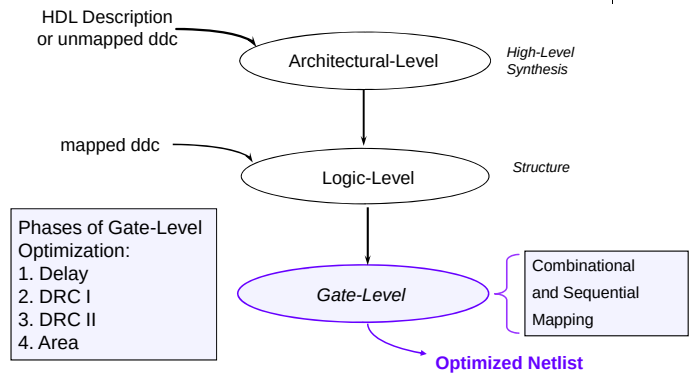
- After high-level optimization, circuit function is still represented by GTECH parts
- One optimization process occurs by default during logic-level optimization
  - Structuring
- Structuring is
  - Reducing logic using common sub-expressions
  - Useful for speed optimization as well as area optimization
  - Constraint based

What Is Structuring – A Picture?

DC's default logic-level optimization strategy

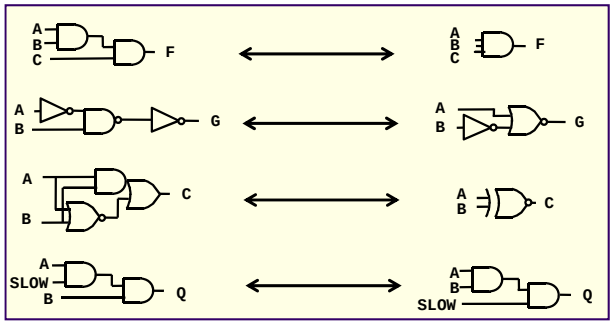


Gate-Level Optimization



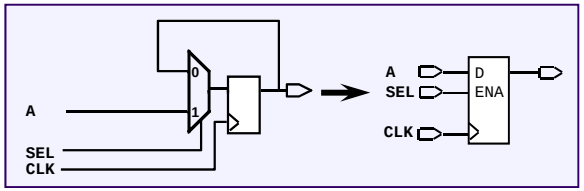
Combinational Mapping

The process of using gates from the target library to generate a design that meets timing and area goals.



Sequential Mapping

- The process by which DC maps to sequential cells from the technology library:
  - Tries to save speed and area by using a more complex sequential cell



Fixing Design Rule Violations

- Technology libraries contain vendor-specific design rules for each cell, e.g. max_capacitance
- During mapping, Design Rule Constraints (DRCs) are checked and fixed
  - Inserts buffers and resizes cells

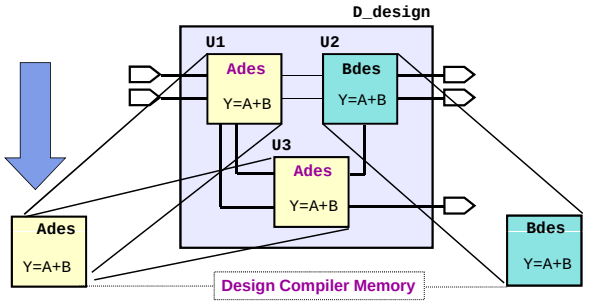
Phases of Gate-Level Optimization:

1. Delay
2. DRC I
3. DRC II
4. Area

DC tries to fix all design rule violations without affecting area or speed.

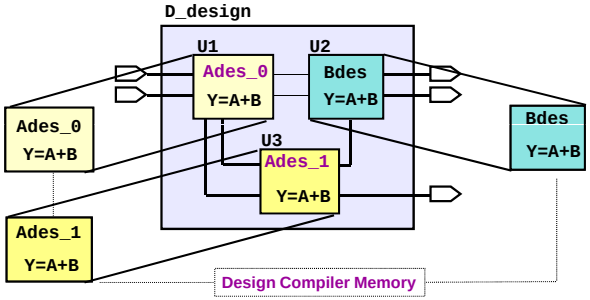
If no other way can be found, DC fixes design rule violations at the expense of timing and area.

What is a Multiple Instantiation?



Multiple Instantiations are Made Unique

- A copy of each multiply-instantiated design is made during compile
  - One copy for each instance
  - Each instance gets a unique design name
  - DC can now map each instance to its own specific environment

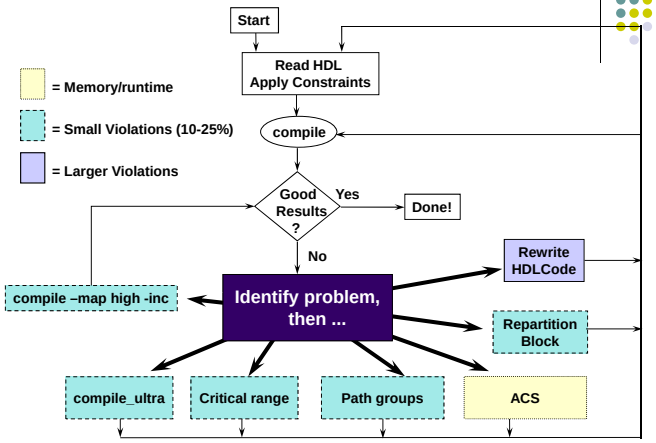


Compile Strategies

Top-Down

Bottom-Up

Analyze First, Fix the Problem Second



Compile Strategies

- **High performance design: compile_ultra**
- **Use Incremental Mapping: compile –map high –inc**
- **Path groups & critical range**
- **Automatic Chip Synthesis**

CT 1: High Performance Designs

Use `compile_ultra`:
The full strength of Design Compiler in a single command.

- A push-button solution for timing critical, high performance designs
- Significantly better delay QoR
  - High performance arithmetic optimization
- As easy-to-use as it gets:
  - All required flags and variables set automatically

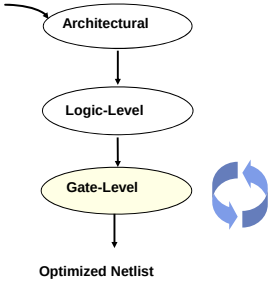
`compile_ultra` User Interface

Simple and very easy to use.

- Switches
  - `-scan` # Test ready compile
  - `-no_autoungroup` # Turn off the auto-ungrouping feature
  - `-no_boundary_optimization` # Do not run boundary optimization
  - `-no_uniquify` # Speed up runtime for multiply instantiated designs
- All DesignWare hierarchies are automatically ungrouped
  - `set compile_ultra_ungroup_dw true`
- Define the maximum block size for auto-ungroup
  - `set compile_auto_ungroup_delay_num_cells 100` (default = 500)
- DesignWare library is required for optimal QoR
  - Automatically added to `synthetic_library` variable in DC 2004.12

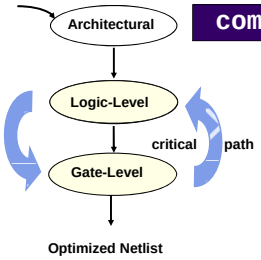
CT2: Use Incremental Mapping

`compile -scan -incremental_mapping`



- Only gate-level optimization is done
  - The design is not taken back to GTECH
  - No logic-level optimization
  - DesignWare implementations may still be changed
  - Slack will get better or stay the same
- Incremental is much faster than regular compile

Use Incremental Mapping with High Effort

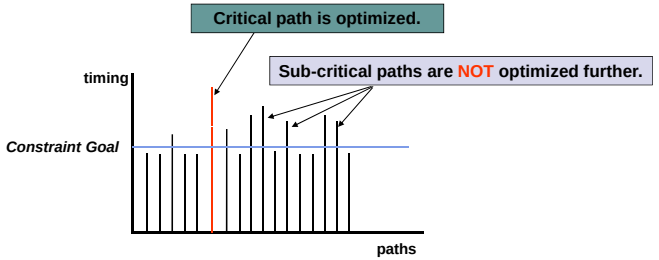



```
compile -scan -inc -map high
```

- Push DC to the extreme to achieve the design goals
- But - keep runtime reasonable with incremental compiles

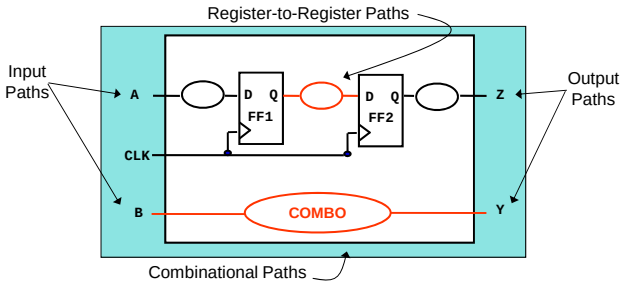
CT3: Path Groups & Critical Range

- When does a Default Compile Stop?
  - By default, `compile` stops when DC cannot find a better optimization solution for the critical path
  - Sub-critical paths are not optimized further!



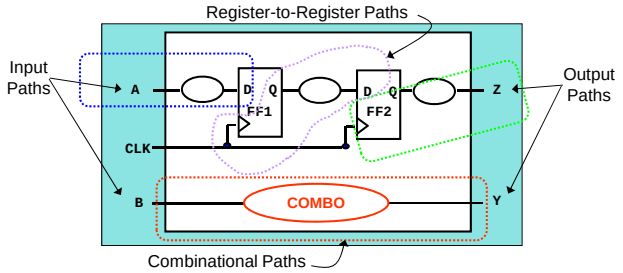
Example: Constraints are Inaccurate

- If COMBO optimization fails and contains the critical path, this prevents optimization of the relevant register-to-register paths



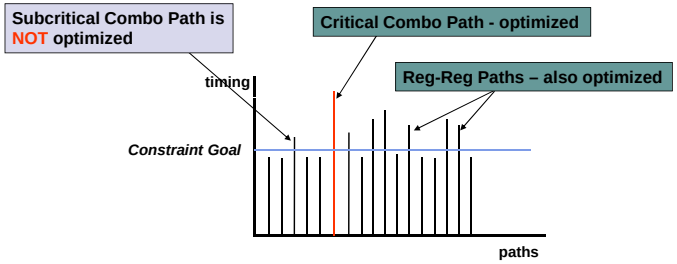
Solution A: User-Defined Path Groups

- Custom path groups allow more control over optimization
  - Each path group is optimized independently
  - Worst violator in one path group does not prevent optimization in another



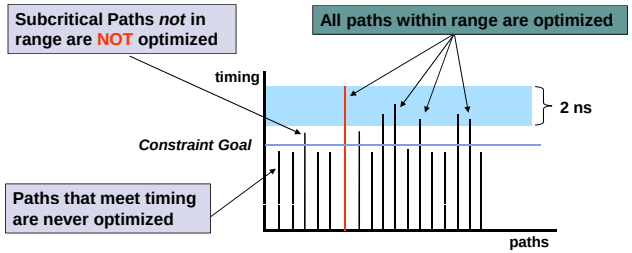
Creating Custom Path Groups

```
# Avoid getting stuck on one path in the reg-reg group
group_path -name INPUTS -from [all_inputs]
group_path -name OUTPUTS -to [all_outputs]
group_path -name COMBO -from [all_inputs] \
                       -to [all_outputs]
```



Solution B: Critical Range

```
set_critical_range 2 [current_design]
```



- Range is *within* 2 ns from critical path
- Fixing related subcritical paths may help the critical path

Solution A+B

- Path Groups + Critical Range:
  - Override the design's critical range with a specific critical range on each path group

```
# Example: Add a critical range to each path group
group_path -name CLK1 -critical_range 0.3
group_path -name CLK2 -critical_range 0.1
group_path –name INPUTS –from [all_inputs] –critical_range 0
report_path_group
```

Path Groups vs. Critical Range

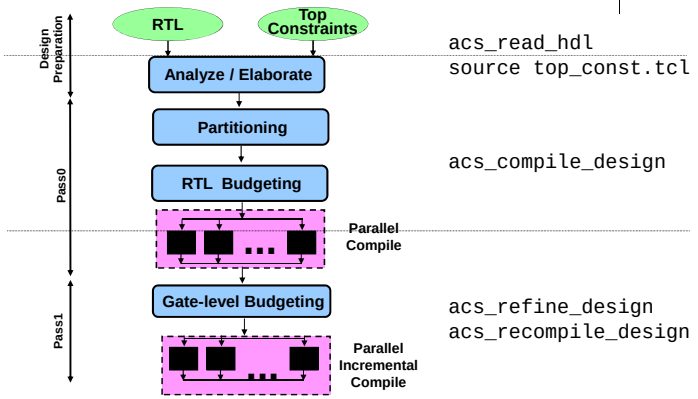
- Path Group:
  - Path Groups will allow path improvements in a given group, which degrade another group's worst violator, if the overall cost function is improved
  - Adding a path group may WORSEN the worst violator in a design
- Critical Range:
  - Critical Range will not allow improvements to near-critical paths that worsen the worst violator in the same path group

CT4: Bottom-up Design

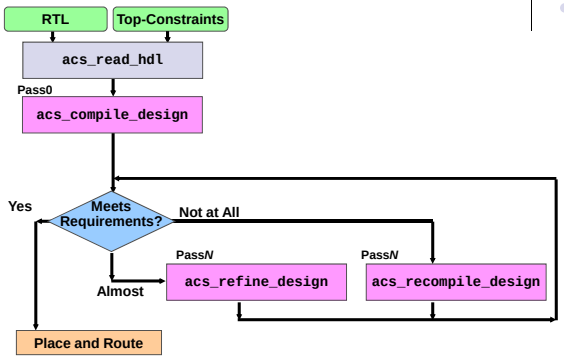
ACS -- Automated Chip Synthesis
"Divide-and-Conquer the easy way"

- Divides design into manageable subdesigns
  - Facilitates a bottom-up compile strategy
- Automates script generation and budgeting
  - ACS chooses "compile partitions"
    - Creates block-level budgets and compile scripts
- Performs single-command parallel synthesis
  - Most powerful: parallel block compiles on separate CPUs
  - If you do not have multiple CPUs: a fast, memory-efficient, serial compile on one CPU

Basic ACS Flow and Commands

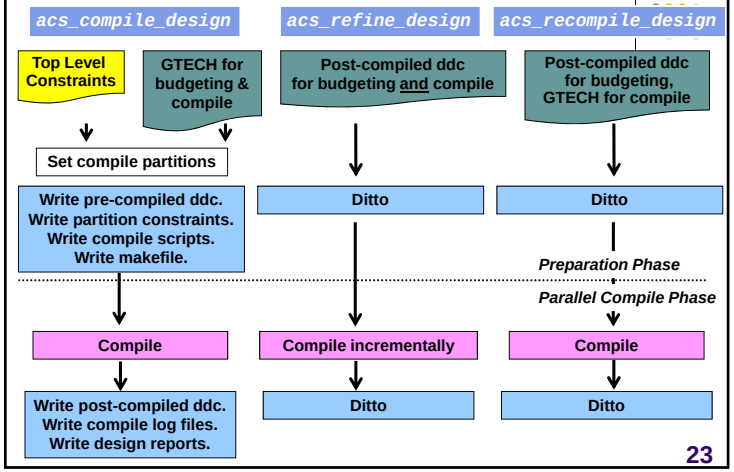


ACS Flow is Flexible



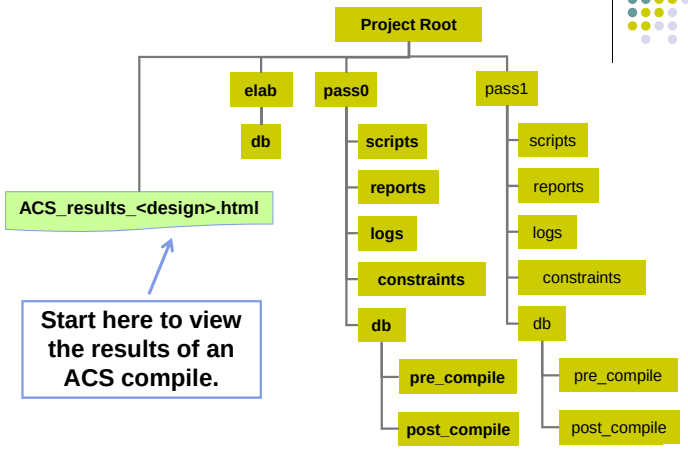
22

Compiles and their Budgets



23

ACS Directory Structure



24

View the Results of a Compile

Navigate the reports, log files, scripts and constraints from a compile using your web browser!

ACS HTML report for design RISC_CORE

Design	RISC_CORE
Destination	pass0
Date	Sat Jan 29 23:30:31 2005
Work Directory	/remote/training/projects/DesignCompiler_1/DC3_2004.12/dev_env/risc_design
No. of Partitions	9
User Defined Compile Script	
User Defined Constraint	
Environment File	env
Reports	report_area report_timing report_constraints report_qor

Partition Name	Script(T)	Constraint(T)	Log File	Errors	Warnings
RISC_CORE (top design)	partition run script	budgeted constraint	log file	0	0
ALU	partition run script	budgeted constraint	log file	0	0
CONTROL	partition run script	budgeted constraint	log file	0	0

25

Examples of Areas You Can Customize

- The directory structure and file naming conventions
- The following steps in the default flow
  - Generating the makefile
  - Resolving multiple instances
  - Identifying compile partitions
  - Generating partition constraints
  - Generating compile scripts
  - Running the compile job (i.e. how the compile job is invoked and which executable file is used)
- The default behavior of the ACS compile commands

```
help acs*
printvar acs*
```

26