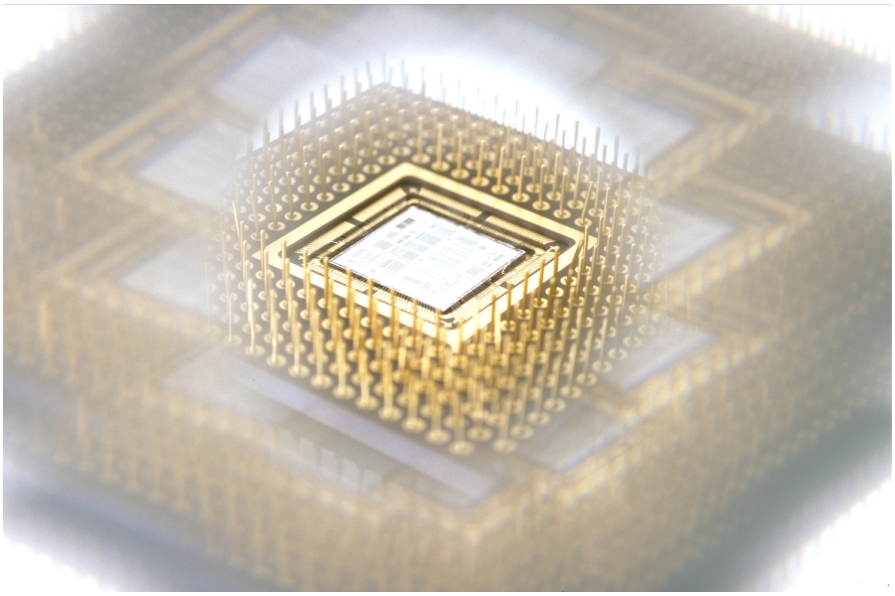
# Avertec Tools

# HITAS Tutorial



Software Release 3.4p5

June 7th, 2010

# About this Document

This document explains how to perform:

- STA at Transistor Level with HITAS
- Timing Characterization (.lib)
- Clock Handling
- Analog Blocks Handling

Documentation issued and compliant with Avertec Tools Release 3.4p5.

Please contact support@avertec.com for comments relating to this manual.

# Table of Contents

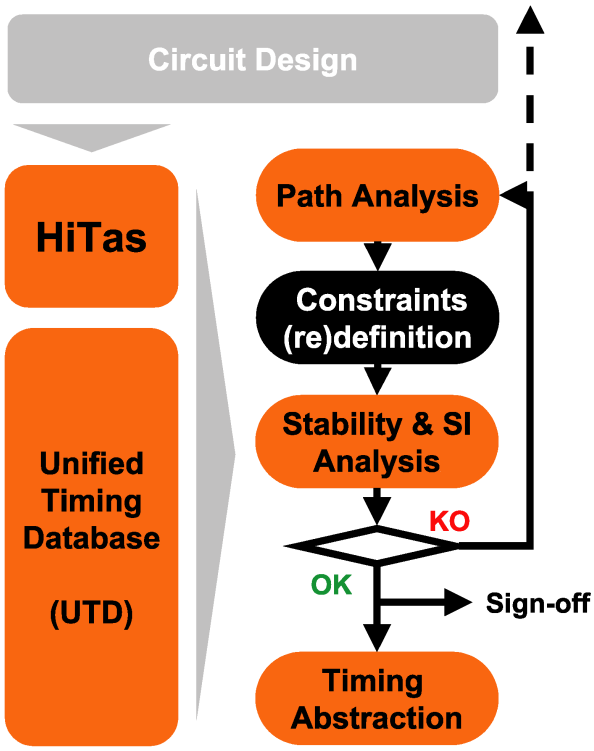
# Chapter 1. Overview

This tutorial describes the use of HITAS Static Timing Analysis and Signal Integrity Analysis platform. The main purpose of this tutorial is to show the ability of the HiTAS platform to analyze designs at transistor level.

This tutorial explains how to set-up a complete Timing and SI verification flow for each component of the design, and then for the top-level. The verification flow includes the following steps:

- Build a homogeneous database for each component
- Perform a first analysis of timing paths
- Integrate interface constraints
- Perform Timing Constraint checks (setup/hold)
- Perform a SI analysis

The verification process is detailed in the following diagram:



# Chapter 2. Static Timing Analysis

## 2.1. Timing Analysis Theory

### 2.1.1. Timing Analysis Goals

The Timing analysis should answer the following questions:

- Does the chip work? With which external timing contraints?
- What are the hold margins?
- What are the sensible paths?
- What is the sensitivity to process variations?
- What is the sensitivity to operating variations (voltage, temperature)?
- What is the chip operating frequency?
- How to improve the design in order to reach the specs

In a top-down approach, Timing Analysis is used for verification purposes. Timing Analysis must say, given the direct environment of the chip (i.e. timing constraints on the interface), if the chip will be able to work properly.

In a bottom-up approach, Timing Analysis is used for characterization purposes.

### 2.1.2. Timing Analysis in the Design Flow

As timing performance of a chip under design is one of the main concerns facing designers, it must be controlled and refined at each stage of the design flow.

In a classical top-down methodology, timing constraints are set at system-level, and synthesis and PR tools are timing-driven. A first Timing Analysis run is done after synthesis, and then after floorplanning and placement. In those cases delays are only estimated, not taking into account the parasistics induced by global routing.

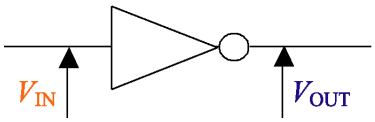
The final sign-off Timing Analysis and characterization is done after global routing, on a netlist back-annotated with extracted parasitics (a post-layout netlist).

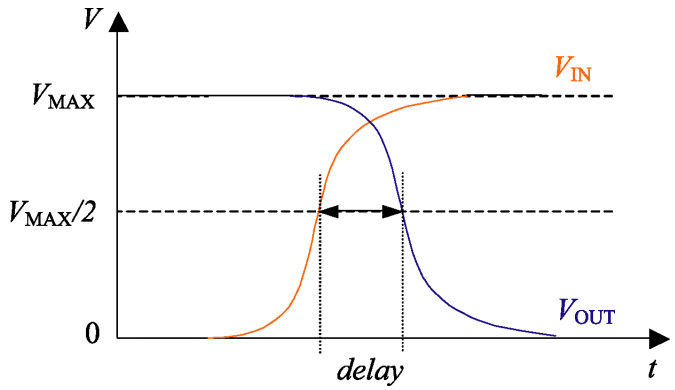
Since synthesis and PR tools are timing-driven, timing characterizations of the building blocks are also now needed. Those building blocks are sometimes large third-party IPs, with fixed timing characterizations. In such cases, timing constraints are also set by those blocks, and the methodology aquires bottom-up aspects.

## 2.2. Definitions

## 2.2.1. Delay Modeling

### Signal Propagation through a Simple Inverter




**Signal Transition**

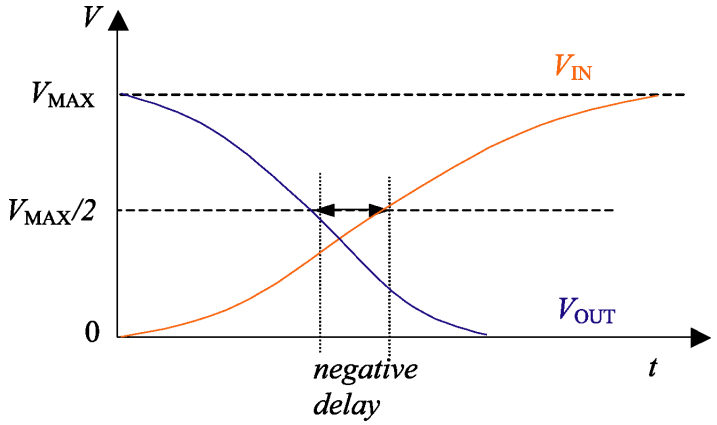
A transition is a change in the state of a signal. A rising transition occurs when the signal's voltage swings from a low level to a high level (from 0V to VMAX). A falling transistion occurs when the signal's voltage goes from a high level to a low level (from VMAX to 0V). In Avertec methodology, a signal transition is also refered to as a timing event

**Threshold**

The delay threshold is the voltage ratio where a signal is considered as having changed state. Typically, this ratio is 50%. The threshold is also the measurement point for delay calculation.

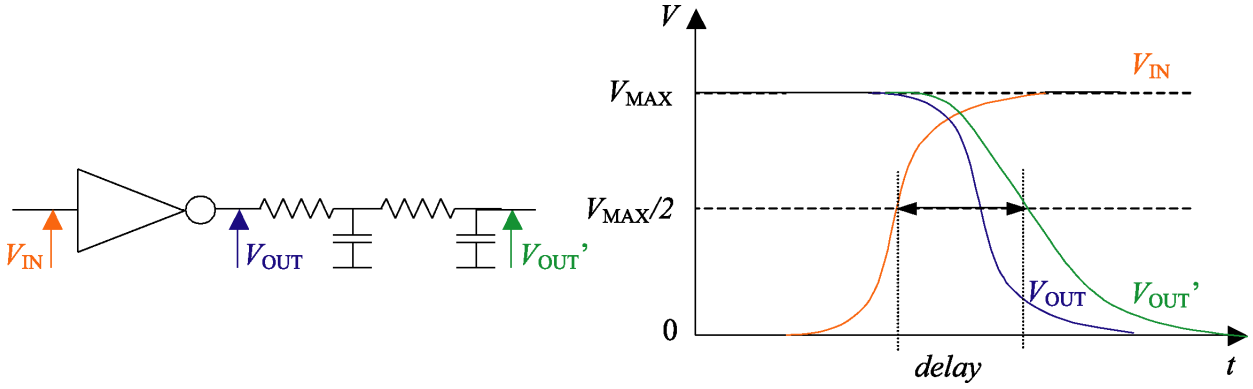
**Delay**

A delay is defined between two signal's transitions, having a causality relation (the first transition implying the second). The value of a delay is the elapsed time between the instant of the first signal's transition crossing the threshold and the instant of the second signal's transition crossing the threshold. As a result of this definition, it is possible to have negative delays (especially with a long input slope)



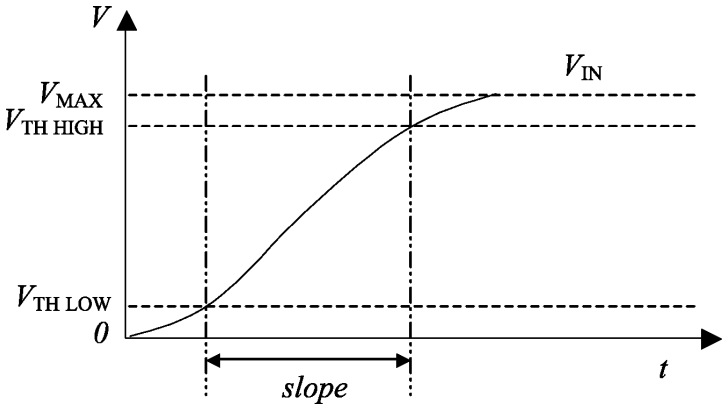

### Signal Propagation through an RC Network

Delays can be measured either on the direct output of the gate, or on any node of the RC interconnect network. Signal propagation through the RC interconnect network causes additionnal delay.




## 2.2.2. Slope Modeling

The transition of a signal is modeled by its slope:




**Slope**

A slope is defined between two thresholds: a high threshold (VTH HIGH) and a low threshold (VTH LOW). The value of the slope is the elapsed time between the instant of the

signal's transition crossing VTH LOW (VTH HIGH) and the instant of the signal's transition crossing VTH HIGH (VTH LOW).

Typically, VTH LOW varies from 5% to 40% of VMAX, and VTH HIGH varies from 60% to 95% of VMAX. A single value defined between two thresholds is a very reductive way to model slopes, as it gives no information about the shape of the slope. The most basic approach is to assume that the slope is linear. In Avertec methodology, the shape of the slope is assumed to be an hyperbolic tangent.

## 2.2.3. Delay Dependancies

The delays and slopes of a given gate depend on three different kinds of factors:

- Internal factors: the implementation of the gate itself. For example, an inverter can be designed in many ways.
- Local external factors: the immediate environment of the gate.
- Global external factors: the environment of the chip.
- Below 90nm: local internal factors: effective length, stress effect, proximity effects.

**Internal Factors**

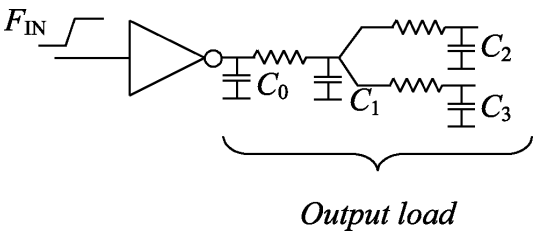
- Gate design, transistor sizes
- Transistor models (MOS9, BSIM3, BSIM4, ...)
- Foundry, technology size (0.13microns, 0.09microns ...)

**Global External Factors**

- Process: best, worst, nominal
- Voltage: global chip power supply
- Temperature

**Local External Factors**

- Input Slope
- Output Load (RC network and fanout)



# 2.3. Delay Calculation

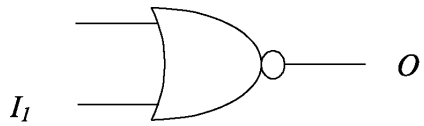
## 2.3.1. Electrical Simulation

### Simple Gates

When dealing with simple gates, delays are most often calculated by electrical simulation (SPICE simulation). The operating mode for calculating delays characterizing a gate is as follow:

- For each input of the gate: Identify (from the gate's truth table) the causality relations between possible transitions on the input and possible transitions on the output.
- For each identified relation: Set the pattern (the states of other inputs) that condition this relation.
- Simulate the design
- Measure the delay associated with the causality relation, i.e. the delay between the input transition and the resulting output transition. The measurement is performed as explained in the preceding section.

As an example, let's consider the following gate, and its associated truth table:




| $I_0$ | $I_1$ | $O$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

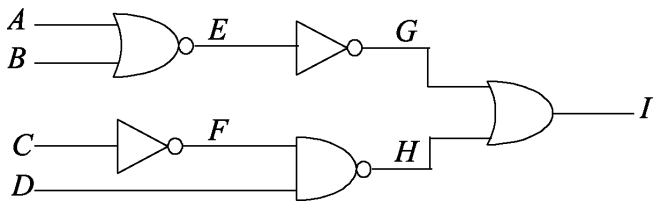
The four identified causality relations and associated delays are reported below. The state of the other input that conditions the causality relation is given between brackets.

```
delay0: I0 rising -> 0 falling (I1 = 0)
delay1: I0 falling -> 0 rising (I1 = 0)
delay2: I1 rising -> 0 falling (I0 = 0)
deIay3: 1 falling -> 0 rising (I0 = 0)
```

Four successive electrical simulations are then necessary to completely characterize the gate.

### Complex Designs

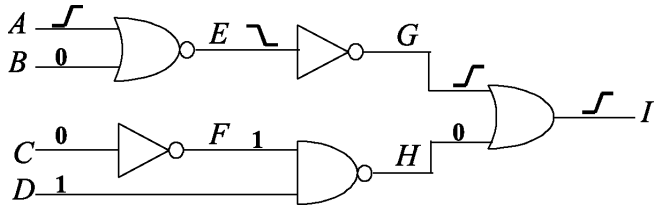
The same kind of delay calculations can be done on more complex designs. For example, let consider the following design.



We can deduce from the connectivity of the gates, and from their truth tables, causality relations between the transitions on inputs A, B, C, D and the transitions on the output I. All the possible causality relations, and the delay associated with each, are given below. The pattern conditioning each relation is given between brackets.

```
delay0: A rising -> I rising (B = 0, C = 0, D = 1)
delay1: A falling -> I falling (B = 0, C = 0, D = 1)
delay2: B rising -> I rising (A = 0, C = 0, D = 1)
delay3: B falling -> I falling (A = 0, C = 0, D = 1)
delay4: C rising -> I rising (A = 0, B = 0, D = 1)
delay5: C falling -> I falling (A = 0, B = 0, D = 1)
delay6: D rising -> I falling (A = 0, B = 0, C = 0)
delay7: D falling -> I rising (A = 0, B = 0, C = 0)
```

See below an illustration of the calculation of delay0 between A rising and H rising. A rising implies E falling if B = 0, which sets the value of input B. E falling implies G rising, which in turn implies I falling if H = 0. H = 0 if F = 1 and D = 1, which sets the value of input D. F = 1 if C = 0, which sets the value of input C.




The pattern conditioning `A rising -> I rising` is then B = 0, C = 0 and D = 1.

### Limitations

Though being quite simple, the above circuit has necessited eight simulations of the full design to completely characterize it.

Actually, for a design of n inputs and m outputs, there may exist up to 2n x 2m causality relations between input and output transistions. This can lead to a maximum of 2n x 2m electrical simulations to calculate all the delays associated with those relations, i.e. to characterize the design.

Furthermore, a causality relation is not easy to identify, and the setting of the pattern conditioning it is a very complex task.

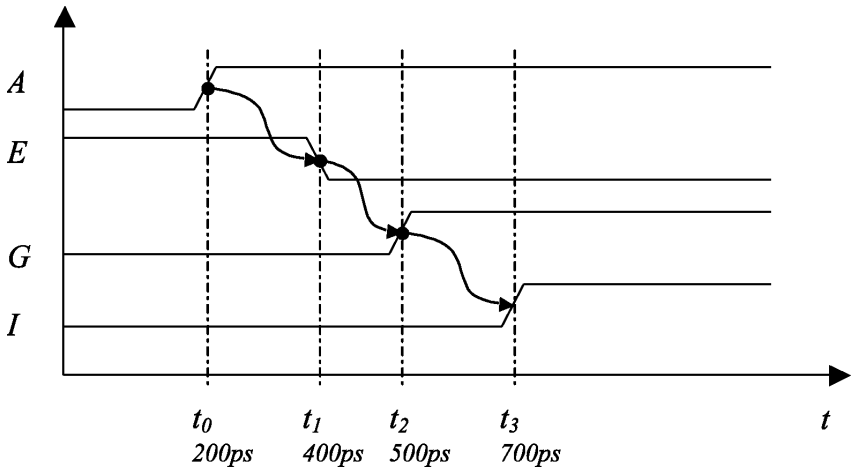
Apart of very regular designs, such as memories, where causality relations are quite simple to establish, and where simulation can be aggressively optimized, these severe drawbacks render electrical simulation impossible to apply on designs exceeding a thousand transistors.

## 2.3.2. Static Timing Analysis

### STA Basics

Static Timing Analysis has arisen from two constatations.

The first constatation was that, causality being a transitive relation, a global causality relation (from an input pin to an output pin) could be discomposed into elementary (gate) causality relations. If we take the example above, the causality relation `A rising -> I rising` can be decomposed into `A rising -> E falling -> G rising -> I rising`. A typical timing representation of such a causality relation is given by a timing diagram, as illustrated below.

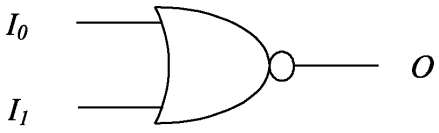


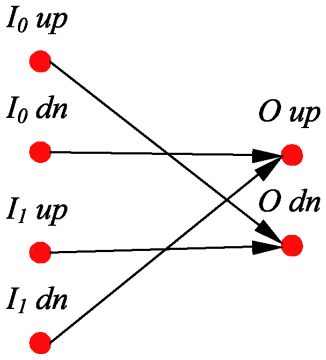

The second constatation was that, as a first approximation, delays associated with elementary causality relations could be added to get the delay of the global causality relation. From this statement we can see that it is possible to calculate (by electrical simulation) the delays associated with a gate only once, and thus achieve significant gains in calculation complexity: the delay of a global causality relation can be calculated by just adding elementary delays.

This statement supposes that delays are independent of their local environment. We have already seen that this is not really the case, and so this leads to some inaccuracy in the delay calculation. We will now see how to refine the delay modelization to attain a accuracy near the one obtained by electrical simulation.

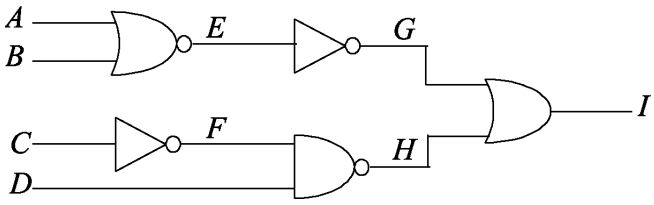
### Graph Modeling

The previous constatations allow us to model designs using weighted graphs, where an edge is a signal transition, and an arc is a causality relation. The arcs are weighted by the delay of the causality relation. The graph of a simple gate (a nor) has the following appearance:

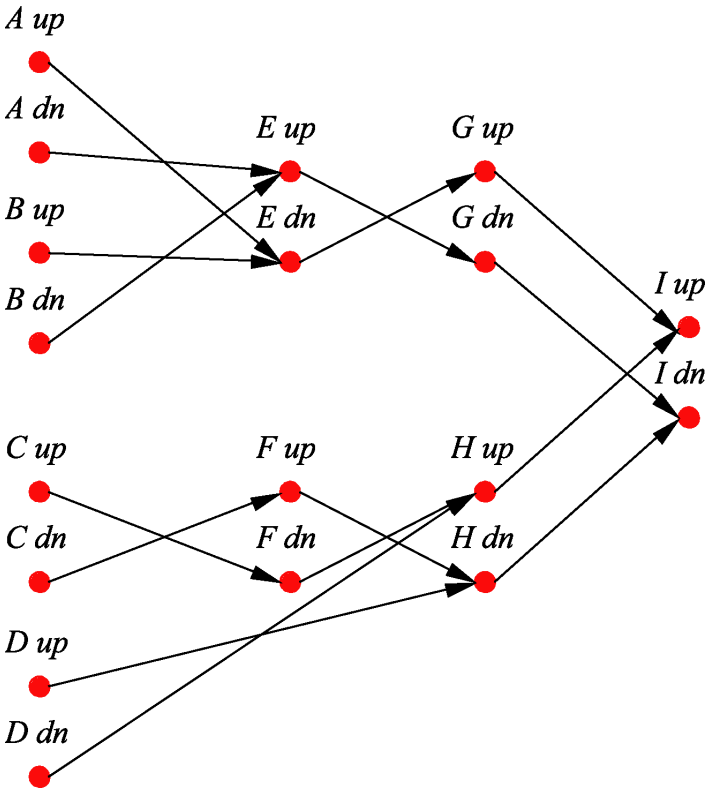



The graph of a gate-level design such as the one below is made by the connexion of the gates' graphs.

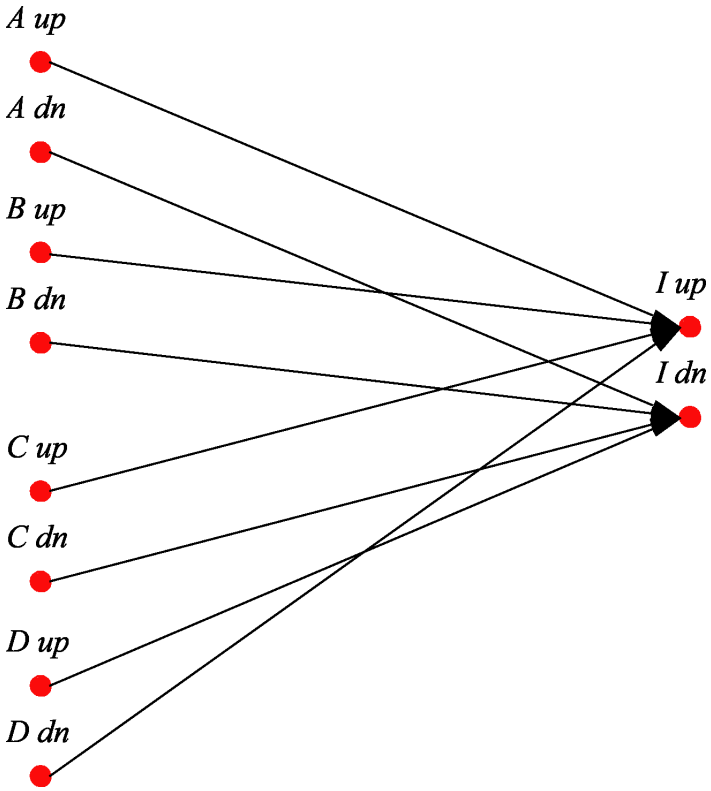



Thus, the graph of the design described above has the following appearance:




This graph is known as a causality graph. A global causality relation is represented here by what is called a path in graph theory terminology.

A graph representation allows us to apply well-known efficient algorithms, such as path searching. In a quite straightforward manner (compexity O(n)), by just following the arcs, we can identify all the timing paths of the design (the eight global causality relations described above).




## 2.3.3. Gate Characterization Methodology

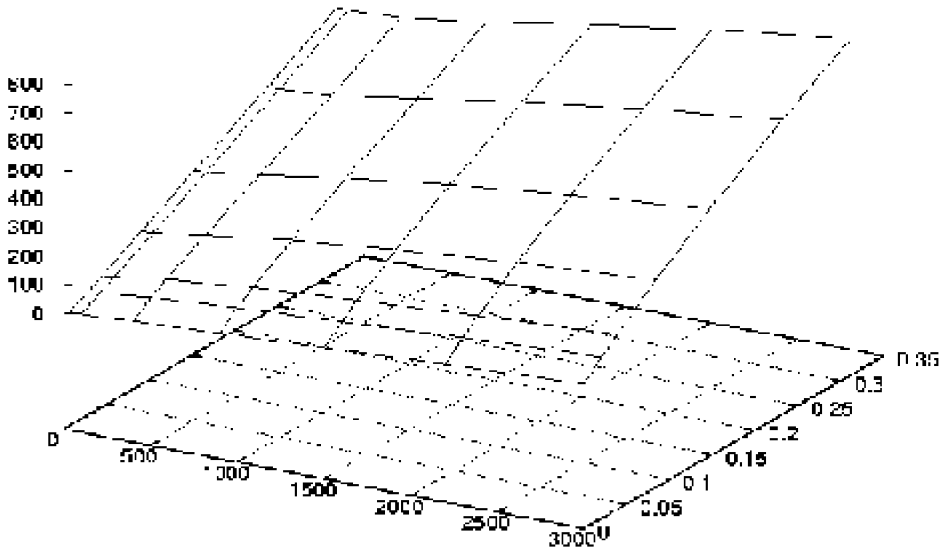
As stated in chapter 1.2.3, gate delays depend on internal factors, global external factors and local external factors. Until 90nm, internal factors don't change for a given chip, and global external factors don't change for a given timing analysis run. The only variable factors are the local external factors, i.e. the input slope and the output load of the gate.

When calculating paths delays, we sum gate delays. As a first approximation, a gate delay can be modeled by a simple value. Experience has showed that this is very unrealistic, since the local external factors can vary a lot from one instance of a gate to another. This has led to a more wide-ranging approach to gate characterization: gate delays are given for a set of input slopes and a set of output loads.

The most common way to describe this set of delay is a lookup table. A common lookup table is a 2D matrix, having for axes the input load and the output capacitance. The following figure illustrates a typical lookup-table.

Lookup table characterizations are most often provided with the gate-library itself. Since they are given for a limited range of PVT, it is often necessary to re-characterize them.

In 90nm and below, other factors may also change: local power supply due to IR-drop, instance dependant parameters (stress effect, proximity effect). This limits the acccuracy a lookup-table based characterization.




# 2.4. Timing Analysis

## 2.4.1. What Needs to be Checked?

In terms of timing, designs are made of combinational elements, and of sequential (clocked) elements.

What we called combinational elements are elements (logic gates) that just propagate signals, independantly to any clock.

Sequential elements are clocked elements. In most cases, they have a memorizing behavior controlled by clock signals (latches, flip-flops). In order to operate correctly, these elements must respect timing constraints (typically the setting of the data to memorize relative to the clock signals).

A kind of clocked element is the dynamic logic stage (precharged logic). It must also respect timing constraints.

The main purpose of the timing analysis process is:

- To verify that the design is implemented in such a way that timing constraints are met on the inner sequential elements.
- To compute the maximum frequencies of the clock signals that still allow the design to operate correctly.
- To compute constraints on the input pins, that if respected, allow the design to work in any environment (CPU, SoC, Board).

In the following sections, we will first study the timing behavior of sequential elements such as latches, flip-flops and dynamic logic gates.

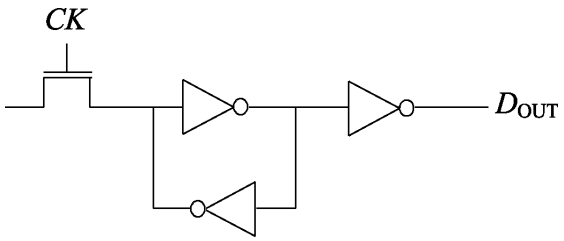
We will then discuss the constraints sequential elements set on the interface of the design (setup and hold times, access times, frequency)

Then we will study how to integrate those elements in such a way that the design can operate correctly.

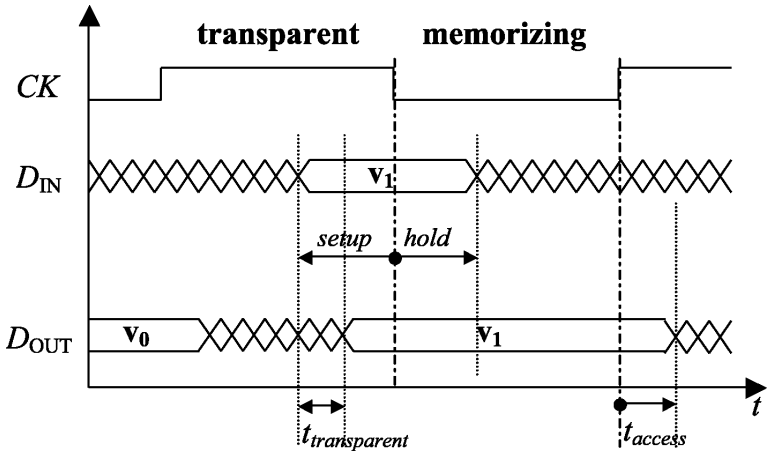
## 2.4.2. The Behavior of Sequential Elements

### Latch

Below is the schematic of a simple latch:




The following timing diagram describes the timing behavior of the latch.




When CK is high, the latch is said to be in transparant mode, i.e. the value on the input `DIN` is observable on the output `DOUT`, after the delay `Ttransparent`, also refered to as `transparancy`.

When CK goes from high to low (the latch closes), the value of `DIN` is memorized in the latch. `DIN` must be stable at the time CK falls. Actually, to ensure the stabilization of the memory loop, `DIN` must not only be stable at the time CK falls, but also for a certain amount of time before CK falls, and for a certain amount of time after CK falls. These times are refered to as `setup time` and `hold time` respectively.

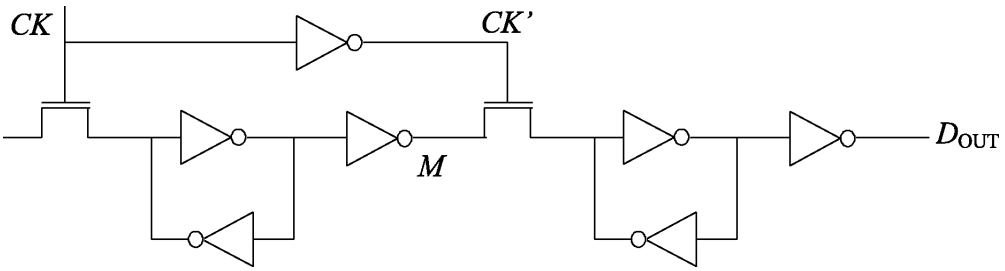
When CK is low, the latch is said to be in memorizing mode. The value observable on `DOUT` is the value memorized when the latch is closed.

When CK goes from low to high, the latch comes back in transparent mode, and a new value on the input `DIN` becomes observable on the output `DOUT` after the delay `taccess`, also refered to as `access time`.

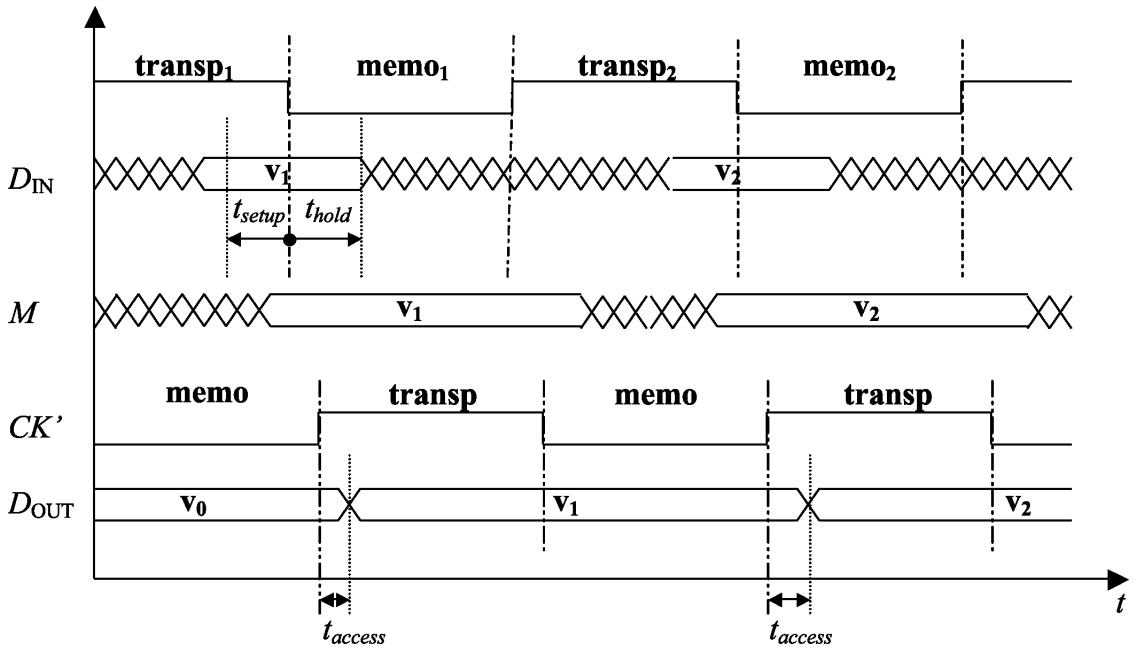
A latch is characterized by four intrinsic values: the transparency, setup, hold and access times.

### Flip-Flop

A typical flip-flop is made of two latches in series, where the clocks are inverted.




The following timing diagram describes the timing behavior of the flip-flop.




When CK is high (`transp1`):

- the first latch is transparent. The value on `DIN` propagates until M.
- the second latch is memorizing (closed)

When CK goes from high to low (`transp1 -> memo1`):

- the first latch closes, and the value on `DIN` is memorized.
- the second latch opens (becomes transparent). The value on M (the memorized value) becomes observable on `DOUT` after the delay taccess (the time taken to traverse the second latch).

When CK is low (`memo1`):

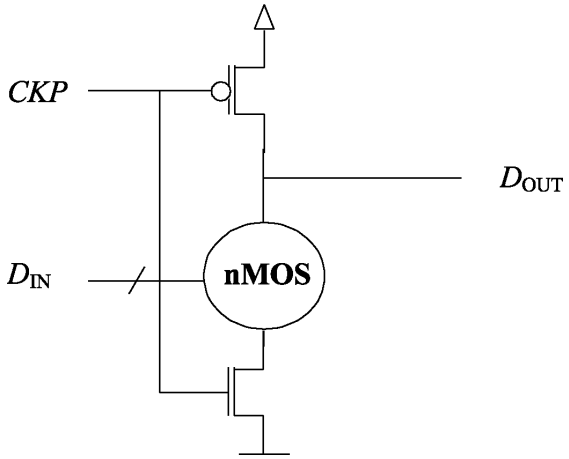
- the first latch is memorizing, and the value on M does not change
- the second is transparent, the value observable on `DOUT` is still the value on M.

When CK goes from low to high (`memo1 -> transp2`):

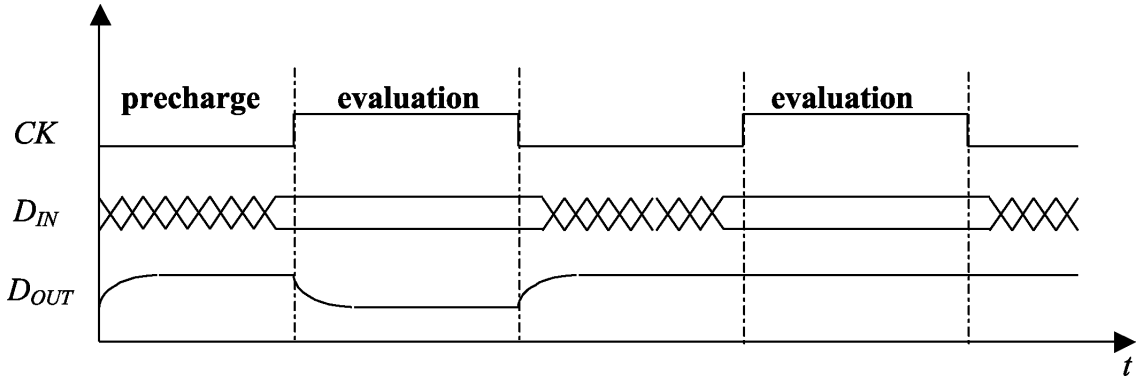
- the first latch becomes transparent, and a new value becomes observable on M
- the second latch must close before the value on M changes, i.e. `tCK->CK'` must be smaller than `tDIN->M'` ,otherwise the new value is memorized in the second latch.

### Dynamic Logic

Below is a typical implementation of Dynamic CMOS logic (precharge-evaluate logic).



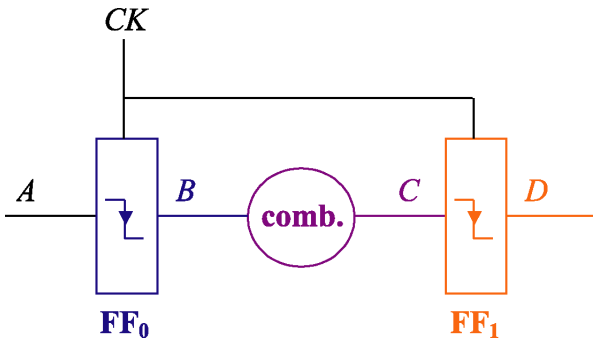

During the precharge phase, the output node of the dynamic CMOS stage is precharged to a high logic level. When the clock signal rises at the beginning of the evaluation phase, there are two possibilities: the output node of the dynamic CMOS stage is either discharged to a low level through the NMOS circuitry (falling transition), or it remains high. Regardless of the input voltages applied to the dynamic CMOS stage, it is not possible for the output node to make a rising transition during the evaluation phase. Consequently, the input configuration must have been set before the evaluation phase and must remain stable during it, otherwise an unwanted conducting path may appear through the NMOS circuitry, leading to an erroneous low-level state of the output node.



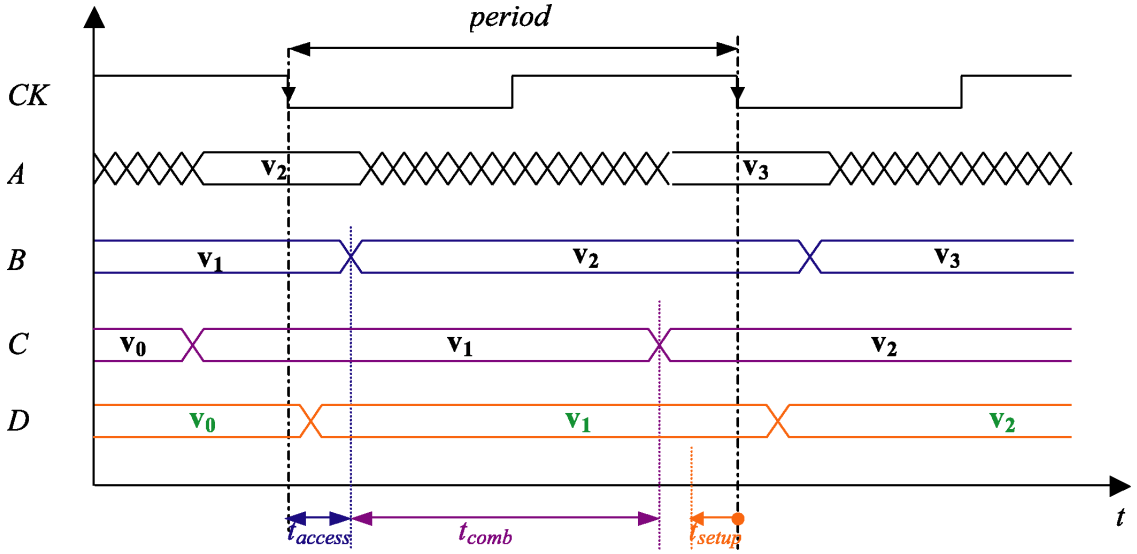
## 2.4.3. Sequential Design Analysis

### Maximum Operating Frequency in Flip-Flop Based Designs

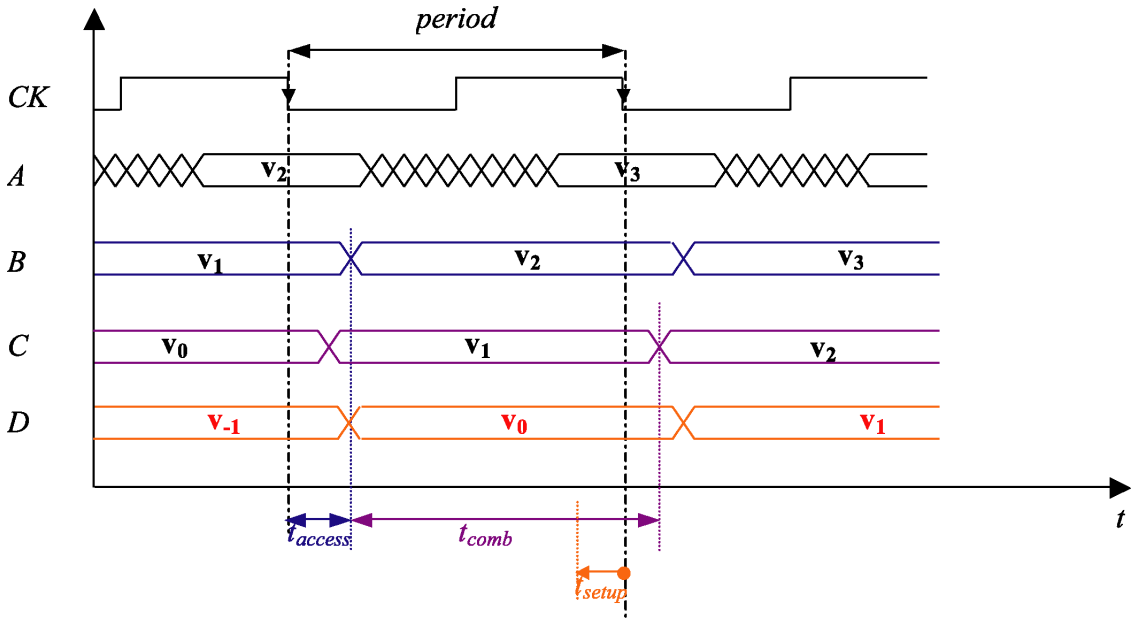
Let's consider the following design made up of two flip-flops:




The following timing diagram illustrates the correct operating mode of the design: the value v2 stored in FF0 becomes accessible on B on the first falling edge of CK, then v2 propagates through the combinational block, finally v2 is stored by FF1 on the second falling edge of CK.




The design operates correctly because `period - tsetup(FF1) > taccess(FF0) + tcomb`. Otherwise, as illustrated in the timing diagram below, if `period - tsetup(FF0) < taccess(FF1) + tcomb`, the second falling edge of CK occurs before the value v2 stored in FF0 has propagated through the combinational block. The value stored by FF1 is v1, the value stored by FF0 in the preceding phase.



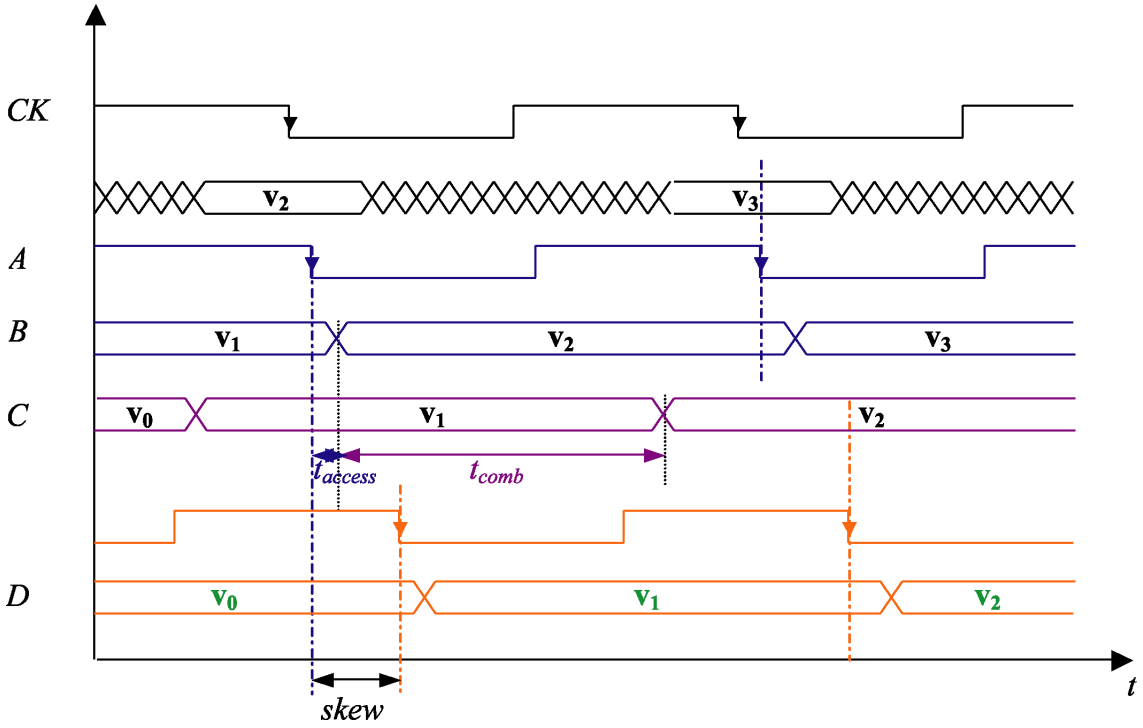

From these observations, we can deduce that there exists a minimum period (and a maximum frequency) allowing the design to operate correctly.

### Skew Impact Analysis

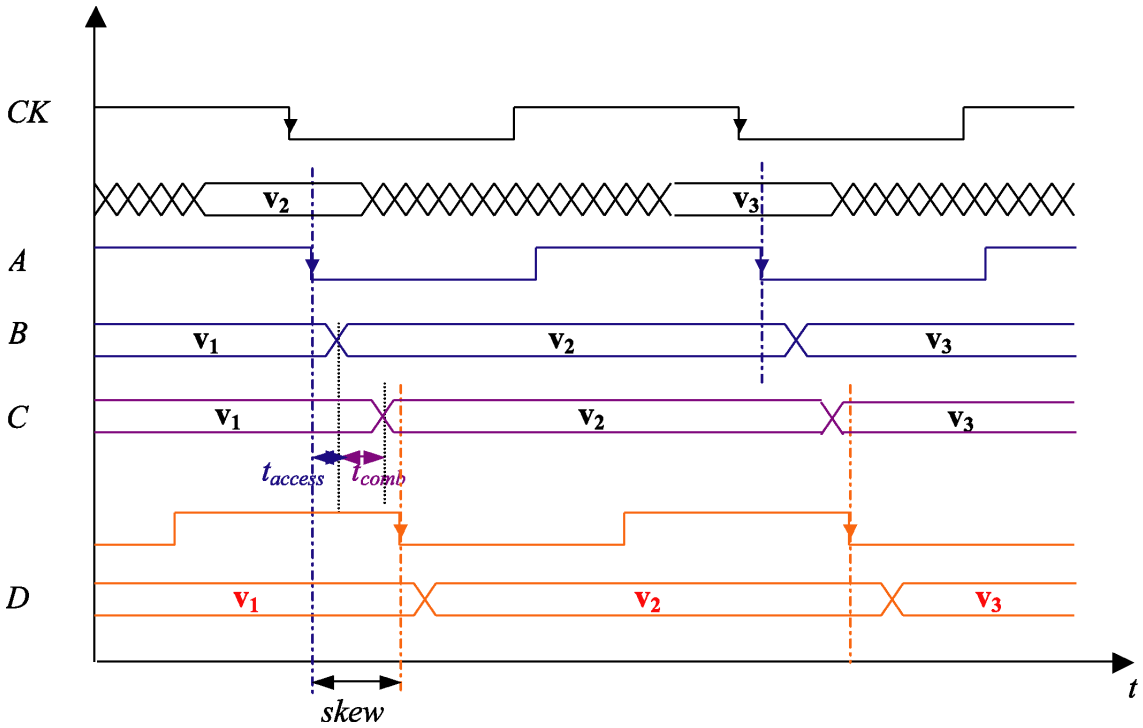
Synchronous designs are based upon the communication between memory elements, such as latches or flip-flops, this communication being controlled by the clock signal. Therefore, a single clock signal is connected to an important number of memory elements in the design, and it is very difficult to ensure that the clock signal will propagate homogenously (with the same delay) towards every memory element, even by inserting clock-tree bufferization. This phenomena is known as clock skew. The following diagram presents asymmetric clock buffering, leading to skew between the two flip-flops.

The communication between the two flip-flops, taking into account the skew, is illustrated in the following timing diagram.

if `taccess + tcomb > skew`, the design will operate correctly.




Otherwise, if `taccess + tcomb < skew`, the design will not work. Note that this timing error is independent of the period.



## 2.4.4. Global Characterization

### Global Setup and Hold Times

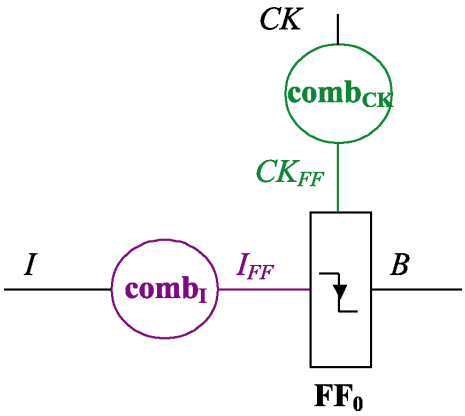
When a flip-flop input is directly connected to an input pin, or is connected through a combinational path to an input pin, the respect of setup/hold constraints depends on the stability window of the input signal itself, and on the propagation delays of the input and clock signals towards the flip-flop.

The input signal's stability window may occur too soon or too late, relative to the clock signal, to ensure the respect of the setup/hold constraints of the flip-flop.

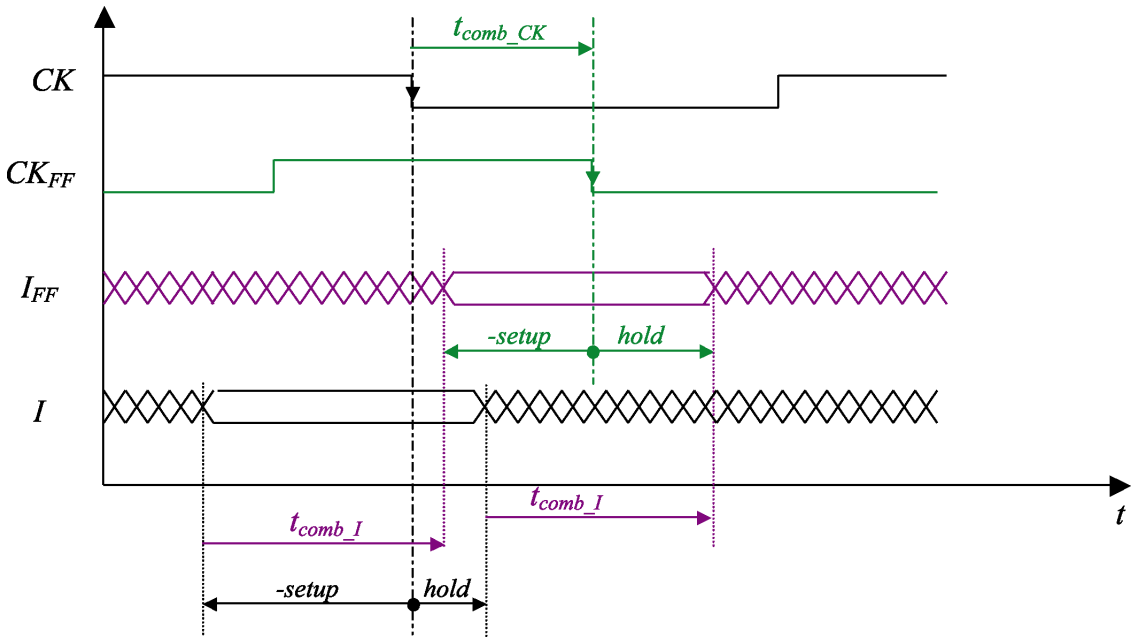
The final purpose of any design being its integration into a higher-level design, it is therefore necessary to provide information on the constraints that apply on the input pins of the design, i.e. in which timing windows input signals must be stable to ensure the respect of internal sequential elements. It is then possible to make the higher-level design in such a way that the stability windows are correctly set on the inputs of the design it integrates.

The constraints are obtained by calculating global setup and hold times.

Let's consider the following design, where I and CK are input pins.



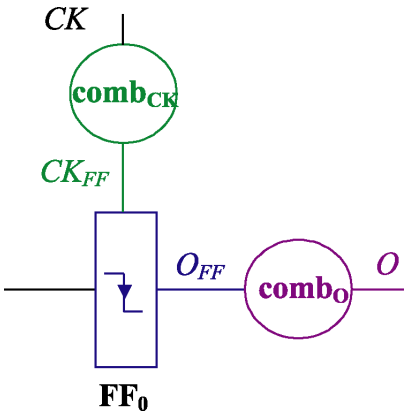

The diagram below illustrates the calculation of global setup/hold times.



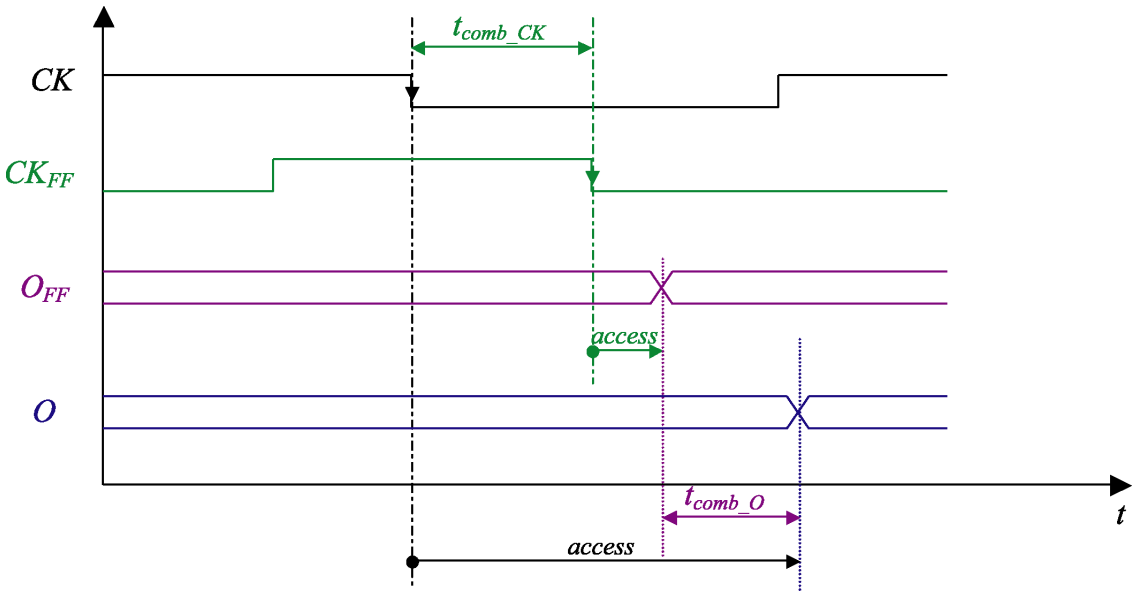

```
global_setup = setup + tcomb_I - tcomb_CK
global_hold = hold + tcomb_CK - tcomb_I
```

### Access Time

Another useful information is the access time, which tells the designer when the data on an output pin is available, relative to a clock edge. In the following design, O is an output pin.




The global access time is illustrated in the timing diagram below.




```
global_access = tcomb_CK + access +  tcomb_O
```

# Chapter 3. Introduction to Programming with Tcl

By Shyam Pather

Information and Telecommunication Technology Center, University of Kansas

## 3.1. Introduction to Tcl

Tcl was originally intended to be a reusable command language. Its developers had been creating a number of interactive tools, each requiring its own command language. Since they were more interested in the tools themselves than the command languages they would employ, these command languages were constructed quickly, without regard to proper design.

After implementing several such "quick-and-dirty" command languages and experiencing problems with each one, they decided to concentrate on implementing a general-purpose, robust command language that could easily be integrated into new applications. Thus Tcl (Tool Command Language) was born. Since that time, Tcl has been widely used as a scripting language. In most cases, Tcl is used in combination with the Tk ("Tool Kit") library, a set of commands and procedures that make it relatively easy to program graphical user interfaces in Tcl.

One of Tcl's most useful features is its extensibility. If an application requires some functionality not offered by standard Tcl, new Tcl commands can be implemented using the C language, and integrated fairly easily. Since Tcl is so easy to extend, many people have written extension packages for some common tasks, and made these freely available on the internet. (For more information, see the Tcl/Tk Information page).

## 3.2. Tcl Programming Basics

The main difference between Tcl and languages such as C, is that Tcl is an interpreted rather than a compiled language. Tcl programs are simply scripts consisting of Tcl commands that are processed by a Tcl interpreter at run time. One advantage that this offers is that Tcl programs can themselves generate Tcl scripts that can be evaluated at a later time. This can be useful, for example, when creating a graphical user interface with a command button that needs to perform different actions at different times.

The next several sections descibe the essential elements of Tcl programs. Each section is accompanied by a series of examples, and a sample Tcl interpreter that you can be use to try out the examples yourself.

## 3.2.1. Variables and Variable Substitution

Variables in Tcl, as in most other languages, can be thought of as boxes in which various kinds of data can be stored. These boxes, or variables, are given names, which are then used to access the values stored in them.

Unlike C, Tcl does not require that variables be declared before they are used. Tcl variables are simply created when they are first assigned values, using the set command. Although they do not have to be deleted, Tcl variables can be deleted using the unset command.

The value stored in a variable can be accessed by prefacing the name of the variable with a dollar sign ("$"). This is known as variable substitution, and is illustrated in the examples below.

Tcl is an example of a "weakly typed" language. This simply means that almost any type of data can be stored in any variable. For example, the same variable can be used to store a number, a date, a string, or even another Tcl script.

Example 1.0:

```
set foo "john"
puts "Hi my name is $foo"
Output: Hi my name is john
```

This example illustrates the use of variable substitution. The value "john" is assigned to the variable "foo", whose value is then substituted for "$foo". Note that variable substitution can occur within a string. The puts command (described in a later section) is used to display the string.

Example 1.1:

```
set month 2
set day 3
set year 97
set date "$month:$day:$year"
puts $date
Output: 2:3:97
```

Here variable substitution is used in several places: The values of the variables "month", "day", and "year" are substituted in the set command that assigns the value of the "date" variable, and the value of the "date" variable is then substituted in the line that displays the output.

Example 1.2:

```
set foo "puts hi"
eval $foo
Output: hi
```

In this example, the variable "foo" holds another (small) Tcl script that simply prints the word "hi". The value of the variable "foo" is subsituted into an eval command, which causes it to be evaluated by the Tcl interpreter (the eval command will be described in greater detail in a later section).

## 3.2.2. Expressions

Tcl allows several types of expressions, including mathematical expressions, and relational expressions. Tcl expressions are usually evaluated using the expr command, as illustrated in the examples below.

Example 2.1:

```
expr 0 == 1
Output: 0
```

Example 2.2:

```
expr 1 == 1
Output: 1
```

Examples 2.1 and 2.2 illustrate the use of relational expressions with the expr command. The first expression evaluates to 0 (false) since 0 does not equal 1, whereas the second expression evalutates to 1 (true), since, obviously, 1 does equal 1. The relational operator "==" is used to do the comparison.

Example 2.3:

```
expr 4 + 5
Output: 9
```

Example 2.3 shows how to use the expr statement to evaluate an arithmetic expression. Here the result is simply the sum of 4 and 5. Tcl offers a rich set of arithmetic and relational operators, each of which is described in the expr manual page.

Example 2.4:

```
expr sin(2)
Output: 0.909297
```

This example shows that the expr statement can be used to evaluate the result of a mathematical function, in this case, the sine of an angle. Tcl offers many such mathematical functions, also described on the expr manual page.

## 3.2.3. Command Substitution

Just as variable substitution is used to substitute the value of a variable into a Tcl script, command substitution can be used to replace a Tcl command with the result it returns. Consider the following example:

Example 3.1:

```
puts "I am [expr 10 * 2] years old, and my I.Q. is [expr 100 - 25]"
Output: I am 20 years old, and my I.Q. is 75
```

As this example shows, square brackets are used to achieve command substitution: The text between the square brackets is evaluated as a Tcl script, and its result is then substituted in its place. In this case, command substitution is used to place the results of two mathematical expressions into a string. Command substitution is often used in conjunction with variable substitution, as shown in Example 3.2:

Example 3.2:

```
set my_height 6.0
puts "If I was 2 inches taller, I would be [expr $my_height+(2.0/12.0)] feet tall"
Output: If I was 2 inches taller, I would be 6.16667 feet tall
```

In this example, the value of the variable "my_height" is substituted inside the angle brackets before the command is evaluated. This is a good illustration of Tcl's one-pass recursive parsing mechanism. When evaluating a statement, the Tcl interpreter, makes one pass over it, and in doing so makes all the necessary substitutions. Once this is done, the interpreter then evaluates the resulting expression. If, during its pass over the expression, the interpreter encounters square brackets (indicating that command substitution is to be performed), it recursively parses the script inside the square brackets in the same manner. For more information on one-pass parsing, refer to Matt Peters' document on the topic.

## 3.2.4. Control Flow

In all but the simplest scripts, some mechanism is needed to control the flow of execution. Tcl offers decision-making constructs (if-else and switch statements) as well as looping constructs (while, for, and foreach statements), both of which can alter the flow of execution in response to some condition. The following examples serve to illustrate these constructs.

Example 4.1:

```
set my_planet "earth"
if {$my_planet == "earth"} {
  puts "I feel right at home."
} elseif {$my_planet == "venus"} {
  puts "This is not my home."
} else {
  puts "I am neither from Earth, nor from Venus."
}
set temp 95
if {$temp < 80} {
  puts "It's a little chilly."
} else {
  puts "Warm enough for me."
}
Output:
I feel right at home.
Warm enough for me.
```

Example 4.1 makes two uses of the if-statement. It sets the value of the variable "my_planet" to "earth", and then uses an if-statement to choose which statement to print. The general syntax of the if-statementis as follows:

```
if test1 body1 ?elseif test2 body2 elseif ...? ?else bodyn?
```

If the test1 expression evaluates to a true value, then body1 is executed. If not, then if there are any elseif clauses present, their test expressions are evalutated and, if true, their bodies are executed. If any one of the tests is made successfully, after its corresponding body is executed, the if-statement terminates, and does not make any further comparisons. If there is an else clause present, its body is executed if no other test succeeds.

Another decision-making construct is the switch-statement. It is a simplification of the if-statement that is useful when one needs to take one of several actions depending on the value of a variable whose possible values are known. This is illustrated in Example 4.2, which uses a switch statement to print a sentence, depending on the value of a variable "num_legs".

Example 4.2:

```
set num_legs 4
switch $num_legs {
  2 {puts "It could be a human."}
  4 {puts "It could be a cow."}
  6 {puts "It could be an ant."}
  8 {puts "It could be a spider."}
  default {puts "It could be anything."}
}
Output:
It could be a cow.
```

The switch-statement has two general forms (both of which are described in detail in the manual page), but the form used here is as follows:

```
switch ?options? string {pattern body ?pattern body ...?}
```

Basically, the string argument is compared to each of the patterns and if a comparison succeeds, the corresponding body is executed, after which the switch statement returns. The pattern "default", if present, is always matched, and thus its body always executed if none of the earlier comparisons succeed.

It is often useful to execute parts of a program repeatedly, until some condition is met. In order to facilitate this, Tcl offers three looping constructs: the while, for, and foreach statements, each of which is shown in the examples below.

Example 4.3:

```
for {set i 0} {$i < 10} {incr i 1} {
  puts "In the for loop, and i == $i"
}
Output:
In the for loop, and i == 0
In the for loop, and i == 1
In the for loop, and i == 2
In the for loop, and i == 3
In the for loop, and i == 4
In the for loop, and i == 5
In the for loop, and i == 6
In the for loop, and i == 7
In the for loop, and i == 8
In the for loop, and i == 9
```

The general syntax for the for-loop is as follows:

```
for init test reinit body
```

The init argument is a Tcl script that initializes a looping variable. In the for-loop used in Example 4.3, the looping variable was called "i", and the init argument simply set it to 0. The test argument is a Tcl script which will be evaluated to decide whether or not to enter the body of the for-loop. Each time this script evaluates to a true value, the body of the loop is executed. The first time this script evaluates to false, the loop terminates. The reinit argument specifies a script that will be called after each time the body is executed. In Example 4.3, the reinit script increments the value of the looping variable, "i". Thus, for-loop in this example executes its body 10 times, before its test script evaluates to false, causing the loop to terminate.

Example 4.4:

```
set i 0
while {$i < 10} {
  puts "In the while loop, and i == $i"
  incr i 1
}
Output:
In the while loop, and i == 0
In the while loop, and i == 1
In the while loop, and i == 2
In the while loop, and i == 3
In the while loop, and i == 4
In the while loop, and i == 5
In the while loop, and i == 6
In the while loop, and i == 7
In the while loop, and i == 8
In the while loop, and i == 9
```

Example 4.4 illustrates the use of a while-loop, the general syntax of which follows the form:

```
while test body
```

The basic concept behind the while-loop is that while the script specified by the test argument evaluates to a true value, the script specified by the body argument is executed. The while loop in Example 4.4 accomplishes the same effect as the for-loop in Example 4.3. A looping variable, "i", is again initialized to 0 and incremented each time the loop is executed. The loop terminates when the value of "i" reaches 10. Note, that in the case of the while-loop, the initialization and re-initialization of the looping variable are not part of the while-statement itself. Therefore, the initialization of the variable is done before the while-loop, and the reinitialization is incorporated into its body. If these statements were left out, the code would probably still run, but with unexpected results.

Example 4.5:

```
foreach vowel {a e i o u} {
  puts "$vowel is a vowel"
}
Output:
a is a vowel
e is a vowel
i is a vowel
o is a vowel
u is a vowel
```

The foreach-loop, illustrated in Example 4.5, operates in a slightly different manner to the other types of Tcl loops described in this section. Whereas for-loops and while-loops execute while a particular condition is true, the foreach-loop executes once for each element of a fixed list. The general syntax for the foreachloop is:

```
foreach varName list body
```

The variable specified by varName takes on each of the values in the list in turn, and the body script is executed each time. In Example 4.5, the variable "vowel" takes on each of the values in the list "{a e i o u}" (Tcl list structure will be discussed in more detail in a later section), and for each value, the body of the loop is executed, resulting in one printed statement each time.

## 3.2.5. Procedures

Procedures in Tcl serve much the same purpose as functions in C. They may take arguments, and may return values. The basic syntax for defining a procedure is:

```
proc name argList body
```

Once a procedure is created, it is considered to be a command, just like any other built-in Tcl command. As such, it may be called using its name, followed by a value for each of its arguments. The return value from a procedure is equivalent to the result of a built-in Tcl command. Thus, command substitution can be used to substitute the return value of a procedure into another expression.

By default, the return value from a procedure is the result of the last command in its body. However, to return another value, the return command may be used. If an argument is given to the return command, then the value of this argument becomes the result of the procedure. The return command may be used anywhere in the body of the procedure, causing the procedure to exit immediately.

Example 5.1:

```
proc sum_proc {a b} {
  return [expr $a + $b]
}
proc magnitude {num} {
  if {$num > 0} {
    return $num
  }
  set num [expr $num * (-1)]
  return $num
}
set num1 12
set num2 14
set sum [sum_proc $num1 $num2]
puts "The sum is $sum"
puts "The magnitude of 3 is [magnitude 3]"
puts "The magnitude of -2 is [magnitude -2]"
Output:
The sum is 26
The magnitude of 3 is 3
The magnitude of -2 is 2
```

This example first creates two procedures, "sum_proc" and "magnitude". "sum_proc" takes two arguments, and simply returns the value of their sum. "magnitude" returns the absolute value of a number. After the procedure definitions, three global variables are created. The last of these, "sum" is assigned the return value of the procedure "sum_proc", called with the values of the variables "num1" and "num2" as arguments. The "magnitude" procedure is then called twice, first with "3" as an argument, then with "-2".

The "sum_proc" procedure uses the expr command to calculate the sum of its arguments. The result of the expr command is substituted into the return statement, making it the return value for the procedure. The "magnitude" procedure makes use of an if-statement to take different actions, depending on the sign of its argument. If the number is postive, its value is returned, and the procedure exits immediately, skipping all the rest of its code. Otherwise, the number is multiplied by -1 to obtain its magnitude, and this value is returned. The same effect could be achieved by moving the statement that multiplies the value by -1 into an else-clause, but the purpose of this example was to illustrate the use of the return statement at several locations within a procedure.

Inside the body of a procedure, new variables may be created with the set command as normal. However, these variables will be local to the procedure, and will no longer be accessible once the procedure returns. If access to global variables is needed inside a procedure, these may be accessed by means of the global keyword, as described in Example 5.2.

Example 5.2:

```
proc dumb_proc {} {
  set myvar 4
  puts "The value of the local variable is $myvar"
  global myglobalvar
  puts "The value of the global variable is $myglobalvar"
}
set myglobalvar 79
dumb_proc
Output:
The value of the local variable is 4
The value of the global variable is 79
```

The procedure "dumb_proc" achieves no special purpose, and is simply designed to illustrate the use of the global keyword to access global variables. It takes no arguments, and as such its argument list is empty. Note that even though the procedure takes no arguments, the empty list structure must still be included. The procedure first creates a local variable, "myvar", sets its value to "4", and then displays it. Then it uses the global keyword to gain access to a global variable named "myglobalvar". The value of this global variable is then printed.

After the procedure definition, a global variable "myglobalvar" is created, and assigned a value of "79". The procedure "dumb_proc" is then called, resulting in the output shown above.

## 3.2.6. Lists

Lists in Tcl provide a simple means by which to group collections of items, and deal with the collection as a single entity. When needed, the single items in the group can be accessed individually. Lists are represented in Tcl as strings with a specified format. As such, they can be used in any place where strings are normally allowed. The elements of a list are also strings, and therefore any form of data that can be represented by a string can be included in a list (allowing lists to be nested within one another). The following examples will illustrates many important list commands:

Example 6.1:

```
set simple_list "John Joe Mary Susan"
puts [lindex $simple_list 0]
puts [lindex $simple_list 2]
Output:
John
Mary
```

Example 6.1 creates a simple list of four elements, each of which consists of one word. The lindex command is then used to extract two of the elements in the list: the 0th element and the 2nd element. Note that list indexing is zero-based. It is also important to see that the lindex command, along with most other list commands, takes an actual list as its first argument, not the name of a variable containing a list. Thus the value of the variable "simple_list" is substitued into the lindex command.

Example 6.2:

```
set simple_list2 "Mike Sam Heather Jennifer"
set compound_list [list $simple_list $simple_list2]
puts $compound_list
puts [llength $compound_list]
Output:
{John Joe Mary Susan} {Mike Sam Heather Jennifer}
2
```

Example 6.2 is a continuation of Example 6.1, and assumes the variable "simple_list" (created in Example 6.1) still exists. In this example, a new variable called "simple_list2" is created, and assigned the value of another simple four-element list. A compound list is then formed by using the list command, which simply forms a list from its arguments. The list command ensures that proper list structure is observed, even when its arguments themselves are lists, or other complex structures. Displaying the value of "compound_list" shows that it is a list of two elements, each of which is itself a list of four elements. The llength command is used to obtain the length of the list, "compund_list", which is 2 in this case.

This example highlights two ways in which to create lists in Tcl: by explicitly listing the elements within quotes, and by using the list command. Explicity listing the elements works well when each of the elements is a single word. However, if the elements contain whitespaces, then maintaining proper list structure becomes a little more tricky. For these cases, the list command proves very useful.

Example 6.3:

```
set mylist "Mercury Venus Mars"
puts $mylist
set mylist [linsert $mylist 2 Earth]
```

```
puts $mylist
lappend mylist Jupiter
puts $mylist
Output:
Mercury Venus Mars
Mercury Venus Earth Mars
Mercury Venus Earth Mars Jupiter
```

In example 6.3, a simple list of 3 items is created, and assigned to the variable "mylist". The linsert command is then used to insert a new item into this list. Note that, as with the llength command, the linsert command takes an actual list as its first argument, not the name of a variable containing a list. The linsert command returns a list that is the same as the list it was passed, except that the specified item is inserted in the appropriate position. This return value needs to be assigned back to the variable "mylist" in order for the list stored in that variable to change.

One list command that does not behave in this way is the lappend command. It takes the name of a variable as its first argument, and appends its subsequent arguments onto the list stored in that variable. Thus the value of the variable is modified directly. Understanding the difference between the way the lappend command works, and the way that commands such as linsert work is fundamental to using lists correctly.

The list commands presented here are only a small subset of those available. Refer to the manual pages, or one of the other Tcl/Tk references for a complete description of all list commands.

## 3.2.7. Arrays

Another way of grouping data in Tcl is to use arrays. Arrays are simply collections of items in which each item is given a unique index by which it may be accessed. As with all other Tcl variables, arrays need not be declared before they are used, and, unlike arrays in C, their size need not be specified either. An individual element of an array may be referred to by using the array name, followed immediately by the index of the element, enclosed in parentheses. Array elements are treated much like any other Tcl variables.

They are created by means of the set command, and their values can be substituted using the dollar sign ("$"), as is the case with other variables.

Example 7.1:

```
set myarray(0) "Zero"
set myarray(1) "One"
set myarray(2) "Two"
for {set i 0} {$i < 3} {incr i 1} {
  puts $myarray($i)
}
Output:
Zero
One
Two
```

In Example 7.1, an array called "myarray" is created and initialized. Note that no special code is required to create the array because it is created by the set statement that assigns a value to its first element. The forloop simply prints out the value stored in each element of the array. Note the use of variable substitution in the array index and the array name.

Example 7.2:

```
set person_info(name) "Fred Smith"
set person_info(age) "25"
set person_info(occupation) "Plumber"
foreach thing {name age occupation} {
  puts "$thing == $person_info($thing)"
}
Output:
name == Fred Smith
age == 25
occupation == Plumber
```

Example 7.2 illustrates one of the unique features of Tcl arrays: array indices need not be integers. In fact, array indices can take on any string value. In this case, the array "person_info" is created with three elements. The indices for the elements are "name", "age", and "occupation". The foreach-loop simply displays each of the elements in the array. Using arrays with named indices is one of the ways to abstract objects in Tcl. In Example 7.2, the "person_info" array can be thought of as an "object" describing a person. Each of the elements in the array then describes a fundamental attribute of the object.

One problem with using named indices with arrays is that one needs to remember the names of all the elements in order to traverse the array. In Example 7.2, for example, the names of all the elements had to be listed explicitly. In a case such as this one, in which there are only three elements, this does not present much of a problem. However, if the array contained many more elements, explicitly listing them each time the array had to be traversed would lead to very messy code. The array Tcl command, illustrated in Example 7.3, provides a means to get around this problem.

Example 7.3:

```
set person_info(name) "Fred Smith"
set person_info(age) "25"
set person_info(occupation) "Plumber"
foreach thing [array names person_info] {
  puts "$thing == $person_info($thing)"
}
Output:
occupation == Plumber
age == 25
name == Fred Smith
```

Example 7.3 produces essentially the same result as Example 7.2, but it makes use of the array command to obtain the names of the elements in the array, instead of listing them explicitly. The array elements are displayed in a different order than they were in Example 7.2, simply because the array command returns the names of the elements in a different order than the one in which they were explicitly listed previously. The general purpose of the array command is to retrieve various pieces of information about an array (such as its size or the names of its elements), and perform other operations (such as searching) on it. The general syntax of the array command is:

```
array option arrayName ?arg arg ...?
```

The option argument specifies which array operation to perform. In the case of Example 7.3, the option argument is given the value "names", which causes the array command to return a list of the names of the elements in the array given by the arrayName argument. For a complete list of the allowed values of the option argument, and well as a description of the corresponding operations, refer to the manual page for the array command.

## 3.2.8. Strings

Since strings are the most prevalent data type in Tcl, it makes sense that Tcl provides a rich set of functions for manipulating them. Most string operations are done by means of the string command, which takes the following general form:

```
string option arg ?arg ...?
```

The string command actually performs several different functions, and the option argument is used to differentiate between them. Example 8.1 creates a string and then uses the string command to manipulate it, and obtain information about it.

Example 8.1:

```
set str "This is a string"
puts "The string is: $str"
puts "The length of the string is: [string length $str]"
puts "The character at index 3 is: [string index $str 3]"
puts "The characters from index 4 through 8 are: [string range $str 4 8]"
puts "The index of the first occurrence of letter \"i\" is: [string first i $str]"
Output:
The string is: This is a string
The length of the string is: 16
The character at index 3 is: s
The characters from index 4 through 8 are: is a
The index of the first occurrence of letter "i" is: 2
```

In Example 8.1, a variable called "str" is created, and initialized to the value, "This is a string". The string commmand is then used with various options to obtain various pieces of information about the string. Refer to the manual page for the string command for a complete listing and explanation of the various options. Also, there are several other string-related commands worth exploring, such as format, regexp, regsub, and scan.

## 3.2.9. Input/Output

Most input and output operations in Tcl are done by means of the puts and gets commands. Most of the examples in this document have made use of the puts command to display output on the console. In a similar manner, the gets command can be used to wait for input from the console, and optionally store it an a variable. Its general syntax has the following form:

```
gets channelId ?varName?
```

The first argument to gets is the name of an open channel from which to read data, and can be thought of as a file descriptor in the C sense. If the varName argument is specified, gets stores the data it reads in that variable, and returns the number of bytes read. If varName is not specified, then gets simply returns the data it read.

Example 9.1:

```
puts -nonewline "Enter your name: "
set bytesread [gets stdin name]
puts "Your name is $name, and it is $bytesread bytes long"
Output: (note that user input is shown in italics)
Enter your name: Shyam
Your name is Shyam, and it is 5 bytes long
```

Example 9.1 makes use of both the puts and gets commands. The puts command is used with the -nonewline flag to suppress the trailing newline that it normally appends to its output. A variable, "bytesread", is then assigned the result of a gets command that reads from the channel "stdin" (the standard input), and stores the data it reads in the variable, "name". Thus "bytesread" ends up storing the number of bytes of user input read from the console.

In Example 9.1, gets was used to read from the channel "stdin" (created automatically when the Tcl interpreter is started) which corresponds to the standard input. The puts command can also be used with a channel identifier to write to a specific channel. However, if no channel identifier is passed to puts, it writes to the standard output (this is the way puts has been used throughout this document). In addition to the standard input and output, channels can also be created to read from other types of files. As illustrated by Example 9.2, the open command can be used to open a channel to a file, and obtain an appropriate identifier for the channel. This identifier can then be passed to gets to read from the file, or puts to write to the file.

Example 9.2:

```
set f [open "/tmp/myfile" "w"]
puts $f "We live in Texas. It's already 110 degrees out here."
puts $f "456"
close $f
Output: (none)
```

This example uses the open command to open a channel to a file called "/tmp/myfile". The syntax of the open command can take on three forms, one of which is:

```
open name ?access?
```

The access argument specifies what type of access (for example, read-only access or read-write access) to the file given by name is desired. See the manual page for the open command for a complete description of the access modes. In this case, write-only access is desired, so the value "w" is given for the access argument.

The open command returns a channel identifier that can be used with gets and puts to read and write from the file. In Example 9.2, this identifier is stored in the variable, "f". The puts command is then used to write two strings to the file, and then the close command is used to close the file.

Example 9.3 reads the file created in Example 9.2, and displays its contents.

Example 9.3:

```
set f [open "/tmp/myfile" "r"]
set line1 [gets $f]
set len_line2 [gets $f line2]
close $f
puts "line 1: $line1"
```

```
puts "line 2: $line2"
puts "Length of line 2: $len_line2"
Output:
line 1: We live in Texas. It's already 110 degrees out here.
line 2: 456
Length of line 2: 3
```

The file, "/tmp/myfile", is opened in read-only mode with the open command. The gets command is then used with the channel identifier returned by open to read from the file. The first call to gets does not give it the name of a variable in which to store the data it reads, so this data is returned instead. Command substitution is used to store it in the variable, "line1". The second call to gets tells it to store the data it reads in the variable, "line2". Therefore, gets would return the number of bytes it read, which, by means of command substitution, is stored in the variable "len_line2". Since all the data has been read, the file is then closed.

In this case, it was known that the file contained only two lines of data. If the length of the file was not known, the eof command could be used with a while loop to read until the end of the file was reached.

## 3.2.10. Other Miscellaneous Tcl Commands

`eval`

As described earlier, Tcl uses a one-pass parsing mechanism when evaluating scripts. It is sometimes useful, however, to have the interpreter make more than one pass over a script before evaluating it. Being able to force the interpreter to parse a script more than once allows one to store Tcl scripts in variables, and have them be evaluated at a later time. This is shown in Example 10.1:

Example 10.1:

```
set foo "set a 22"
eval $foo
puts $a
Output:
22
```

The variable "foo" is set to the value "set a 22", which is itself a Tcl script. Next, the value of the variable "foo" is substituted into the eval command. The eval command simply passes its arguments through the Tcl interpreter for another round of parsing. When the interpreter encounters the statement "eval $foo", the first round of parsing simply substitutes the value of the variable "foo" in the place of "$foo", resulting in the expression "eval set a 22". The eval command then sends its arguments, "set a 22", through the interpreter again, resulting in the variable "a" being created and assigned the value "22".

One might be tempted to think that the use of the eval command could be avoided and simply replaced with the statement,

`$foo`

This does not work because, on encountering the statement "$foo", the interpreter simply replaces it with the value stored in the variable "foo", and then considers its parsing work done. So, "$foo" gets replaced by "set a 22", but the interpreter never parses "set a 22", which it needs to do to make sense of the components of the statement (it needs to realize that "set" corresponds to a built in command, and that it is being passed two arguments, "a" and "22") and evaluate it correctly .

`catch`

When an error occurs in a Tcl command, the entire script of which it is a part is halted, and an error message is displayed. However, instead of halting the whole Tcl script, it may be useful to simply display a friendly error message and continue execution of the Tcl script.

The catch command prevents Tcl's error handling mechanisms from executing (and thus halting execution) and simply returns a meaningful value when an error occurs. This allows the program to define its own behaviour in the case of an error.

Example 10.2:

```
set retval [catch {set f [open "nosuchfile" "r"]}]
if {$retval == 1} {
  puts "An error occured"
}
Output: (if there is no file "nosuchfile" in the current directory).
An error occured
```

The catch command is given a Tcl script as an argument. It evaluates this script, and if an error occurs, it returns 1, otherwise it returns 0. In Example 10.2, the script passed to catch tries to open a file named "nosuchfile". Assuming that no file with this name exists in the current directory, the open command should return an error. Since it occurs within a catch statement, the normal Tcl error handling routines do not get invoked, and the catch command simply returns 1. This return value is assigned to the variable "retval", which is checked to determine whether or not to print the error message. The catch command can be used in many different ways, only one of which is shown here. Refer to the manual page for a more complete description.

# Chapter 4. Examples

This document contains a collection of examples, each one illustrating a feature or set of features. Those examples appear in the following directories:

- `inv/`: guidelines for database construction and analysis, based upon a single inverter and a chain of inverters. Introduction to the Tcl interface.
- `adder/`: guidelines for database construction and analysis, based upon a full-adder design. Path reporting and simulation
- `ms/`: guidelines for database construction and analysis of a master-slave flip-flop. Introduction to timing checks and slack reports.
- `addaccu/`: guidelines for .lib characterization of a simple adder-accumulator design. Link with 3rd-party simulator
- `cpu2901/`: guidelines for database construction of a small microprocessor. Introduction to advanced configuration. Path reporting, timing checks and slack reports.
- `h_macro/`: guidelines for hierarchical database construction and timing analysis of a hierachical design made up of custom macros and pre-characterized blocks.
- `blackbox/`: guidelines for handling of analog blocks. Introduction to the two most simple techniques.

# Chapter 5. Inverter

## 5.1. Design Description

This example presents HITAS elementary concepts, based upon a simple inverter design and later an inverter chain.

The first example takes place in the `inv/` directory.

## 5.2. Database Generation

### 5.2.1. Principles

The database generation follows the steps below:

- Design partitioning: the algorithm creates a net-list of pseudo-gates from the flat transistor net-list. Partitions are called "cones" and have the property to be electrically independent from one to the other.
- Automatic memory components recognition: a memory-identification engine analyzes cones and loop between cones, and flags latches and pre-charged elements.
- Graph modeling: a cone is modeled as a graph, where edges are events on signals, and where arcs are possible causality relations between events. Causality relations are also called timing arcs.
- Creation of delay models: a delay model, derived from the BSIM MOS equations, is associated with each timing arc.
- Creation of all the timing paths: the successive timing arcs between connectors and memory elements are merged to create timing paths. All the possible timing paths in the design are saved into the database.

### 5.2.2. Global Configuration

The complete configuration required for the database generation takes place in the `db.tcl`. The script also launches the commands that effectively generate that database.

Configuration variables are set in the Tcl script by the mean of the `avt_config` function.

`avt_config tasGenerateConeFile yes`
: tells the tool to dump on disk the .cns file, which contains the partitions (the cones) created by the partitioning algorithm.

`avt_config avtVerboseConeFile yes`
: tells the tool to dump on disk the .cnv file, which is a more readable version.

`avt_config simVthHigh 0.8`
: High threshold of the slope.

`avt_config simVthLow 0.2`
: Low threshold of the slope.

`avt_config simSlope 20e-12`
: Transient time of the slope in second.

`avt_config simToolModel ngspice`
: tells the tool the technology file type (which simulator it is designed for)

The temperature and supplies specifications take place in the `inv.spi` file:

```
.TEMP 125
Vsupply vdd 0 DC 1.62
Vground vss 0 DC 0
```

### 5.2.3. Technology Integration

The technology file is included with a SPICE `.INCLUDE` directive in the `inv.spi` file (in the case of recursive inclusions, paths must be absolute).

In the present example, the `.INCLUDE` directive is used.

```
.INCLUDE ../techno/bsim4_dummy.ng
```

### 5.2.4. Database Generation

The generation launch is done through the command `hitas`:

```
avt_LoadFile inv.spi spice
set fig [hitas inv]
```

The `hitas` function takes as argument the name of the figure (the subckt for a SPICE netlist) to analyze. The tas function returns a pointer on the timing database newly created. This pointer can be used as an input to further steps of verification, thus avoiding costly re-reading of the timing database from the disk.

To perform the database generation, just launch the script `db.tcl`

## 5.3. Database Analysis

### 5.3.1. Database overview

At this step of the analysis process, the timing database of the adder sub-circuit consists of four files:

**DTX file**
: All the timing arcs of the sub-circuit, based upon the characterization of the "cones" created during the partitioning phase.

| | |
|---|---|
| **STM file** | The models that allow to computing the delay values for timing arcs and timing paths. |
| **RCX file** | The interconnect elements (RC) at the physical boundary of the sub-circuit. This file is used for hierarchy purposes, allowing the partial flatten of interconnections at upper levels of hierarchy. |

### 5.3.2. Database properties

The script `db.tcl` presents also Tcl access to the properties of the database:

- Temperature
- Power supply

Other properties are available. See HITAS Reference Guide.

# Chapter 6. Inverter Chain

## 6.1. Design Description

This second example (also in directory `inv/`) presents HITAS database construction and database browsing concepts, based upon a inverter-chain design (file `inv_chain.spi`).

## 6.2. Database properties

The script `db_chain.tcl` performs the database construction in the same way than the previous example. It also presents Tcl acces to the properties of the database:

- Temperature
- Power supply
- Input slope
- Output load
- Generation date

Other properties are available. See HITAS Reference Guide.

## 6.3. Path Reports

The script `report.tcl` shows a typical path report. The commands in the script are the following:

```
set fig [ttv_LoadSpecifiedTimingFigure inv_chain]
```

This command loads the timing database (`.dtx`, `.stm` and `.cns` files) into the program's memory.

```
set clist [ttv_GetPaths $fig * * rf 5 critic path max]
```

This command looks for the 5 longest paths (`5 critic path max`) in the circuit starting and ending on any terminal node (`* *`), with a rising transition on start node and a falling transition on the end node (`rf`). A terminal node is a pin or a latch.

```
set f [fopen inv_chain.paths "w+"]
```

This commands opens a file `inv_chain.paths` for further writing.

```
ttv_DisplayPathListDetail $f $clist
```

This command prints in file `inv_chain.paths` the result (`$clist`) of the previous command `ttv_GetPaths`. For results on standard output, replace `$f` by `stdout`. The output looks like:

```
     Voltage :        1.62V
     Temperature :    125 degree C

      *** Path list (unit:[ns]) ***

 Path     Start     Start      Path     Total      Data    Ending       Start
           time     slope     delay     delay       lag     slope       From_node      To_node
_____________________________________________________________________________________________
    1     0.000     0.200     0.397     0.397     0.000     0.031   (R) in         (F) out

Node type Index:
   (C) : Clock node          (L) : Latch node         (F) : Flip-flop node
   (B) : Breakpoint node     (K) : Latch command node (S) : Output connector node
   (SZ): Output HZ connector (N) : Precharge node

      *** Path details (unit:[ns]) ***

Path (1) :

        Delay
      Acc    Delta    R/F     Cap[pf]   Type   Node_Name        Net_Name     Line
  ____________________________________________________________________________________
     0.000   0.000 0.200 R      0.020          in               in
     0.070   0.070 0.094 F      0.027          1                1            inv
     0.186   0.116 0.111 R      0.027          2                2            inv
     0.258   0.072 0.074 F      0.027          3                3            inv
     0.372   0.114 0.106 R      0.027          4                4            inv
     0.397   0.025 0.031 F      0.007    (S)   out              out          inv
  ____________________________________________________________________________________
     0.397   0.397            (total)
```

There is actually only one path in this inverter chain.

# Chapter 7. Adder

This example tackles similar concepts as the ones described in the previous example. It just illustrates them on a more slightly complex design, a combinational full adder.

This example takes place in the `adder/` directory.

## 7.1. Database Generation

### 7.1.1. Global Configuration

The complete configuration required for the database generation takes place in the `db.tcl`. It is the same as in the previous example (inverters). The script also launches the commands that effectively generate that database.

The temperature and supplies specifications take place in the `adder.spi` file:

```
.TEMP 125
.GLOBAL vdd vss
Vsupply vdd 0 DC 1.62
Vground vss 0 DC 0
```

As the `adder.spi` subcircuit is not instantiated, the `vdd` and `vss` signals appear in the `.GLOBAL` statement.

The technology file is included with a SPICE `.INCLUDE` directive in the `adder.spi` file.

```
.INCLUDE ../techno/bsim4_dummy.ng
```

### 7.1.2. Database Generation

The generation launch is done through the command `hitas`:

```
avt_LoadFile adder.spi spice
set fig [hitas adder]
```

The `hitas` function takes as argument the name of the figure (the subckt for a SPICE netlist) to analyze. The tas function returns a pointer on the timing database newly created. This pointer can be used as an input to further steps of verification, thus avoiding costly re-reading of the timing database from the disk.

To perform the database generation, just launch the script `db.tcl`

## 7.2. Path Searching with the Tcl Interface

The complete configuration required for the database browsing takes place in the `report.tcl`.

The command:

```
set fig [ttv_LoadSpecifiedTimingFigure adder]
```

reads the timing database from disk (as said before, the re-reading of the database can be avoided by directly taking as an input the return value of the `hitas` function. For the sake of clarity, and because we are dealing with small timing databases, we preferred to split different verification steps into different scripts).

The command:

```
set clist [ttv_GetPaths $fig * * rr 5 critic path max]
```

gives the 5 most critical paths (`critic` and `path` arguments) of the design, that begin and end on a rising transition (`rr` argument), with no specification of signal name (`* *` arguments), in the database pointed out by `$fig`. The function returns a pointer on the newly created list.

The command:

```
ttv_DisplayPathListDetail stdout $clist
```

displays on the standard output the detail of all the paths of the path list given by the `ttv_GetPaths` function.

To get these paths, launch the script `report.tcl`.

## 7.3. Exercises

- Ex 1.1. Get the list of connectors with the Tcl interface and with the GUI.
- Ex 1.2. Get the critical paths between selected connectors, with any transition, with the Tcl interface and with the GUI
- Ex 1.3. Get all the parallel paths of the most critical path with the Tcl interface and with the GUI
- Ex 1.4 Get the detail of a parallel path and identify divergence
- Ex 1.5 Hide the column `Line Type` in the report and observe the results
- Ex 1.6 Change the unit of the report from `ns` `ps` (`ttv_SetupReport`)

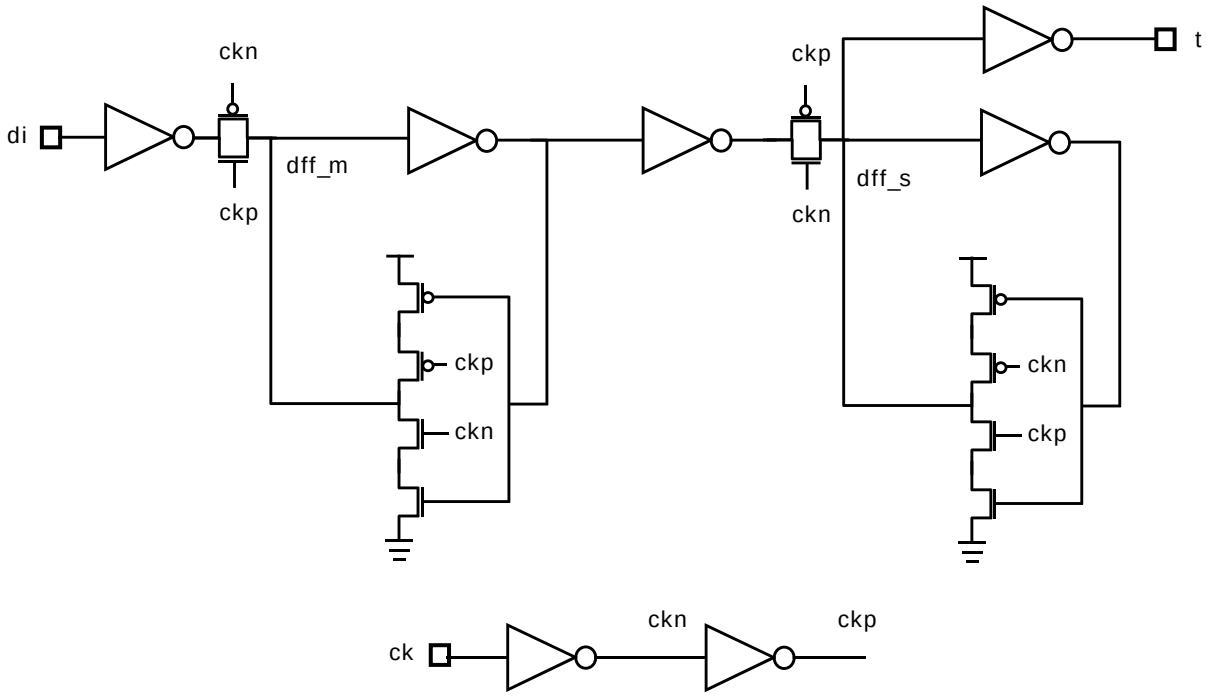
## 7.4. Solutions

```
#!/usr/bin/env avt_shell

# Ex adder.1
set fig [ttv_LoadSpecifiedTimingFigure adder]
set clist [ttv_GetTimingSignalList $fig connector interface]
foreach c $clist {
  puts "[ttv_GetTimingSignalProperty $c NAME] [ttv_GetTimingSignalProperty $c DIR]"
}

# Ex adder.2
set fig [ttv_LoadSpecifiedTimingFigure adder]
set clist [ttv_GetPaths $fig a_0 cout ?? 5 critic path max]
ttv_DisplayPathListDetail stdout $clist
```

```
# Ex adder.3 and adder.4
set fig [ttv_LoadSpecifiedTimingFigure adder]
set clist [ttv_GetPaths $fig * * rr 5 critic path max]
set plist [ttv_GetParallelPaths [lindex $clist 1] 10]
ttv_DisplayPathListDetail stdout $plist

# Ex adder.5 and adder.6
ttv_DisplayPathDetailHideColumn dt.linetype
ttv_SetupReport ps
ttv_DisplayPathListDetail stdout $plist
```

# Chapter 8. Master-Slave Flip-Flop

This example presents how HITAS performs timing checks upon a sequential design. The example given here is a simple master-slave flip-flop (`msdp2_y` diagram below). It takes place in the directory `ms/`




## 8.1. Timing Checks

### 8.1.1. Principles

Static Timing Analysis is performed by propagating interface constraints towards latch's inputs and commands, and towards output connectors. Once interface constraints have been propagated, the tool computes the setup and hold slacks.

The complete configuration required for database construction takes place in the `db.tcl`. It does not differ from previous examples

The complete configuration required for STA takes place in the `sta.tcl`.

### 8.1.2. STA with Tcl Interface

#### Timing Constraints

Timing constraints are set in SDC format (Synopsys Design Constraints). Let's review the constraints commands applied to the flip-flop:

`inf_SetFigureName msdp2_y`
: tells the tool to apply the SDC constraints to the design msdp2_y.

`create_clock -period 1000 -waveform {500 0} ck`
: Defines the clock waveform.

`set_input_delay -clock -ck -clock_fall -min 200 di`

`set_input_delay -clock -ck -clock_fall -max 300 di`
: Tells the tool that inputs signals on `di` may switch between times 200 and 300.

`set_output_delay -clock ck -clock_fall -min 200 t`

`set_output_delay -clock ck -clock_fall -max 400 t`
: Tells the tool that the delay from output connector `t` to the next memory element (hypothetical).

### Static Timing Analysis

Launch of the STA is done by invoking the following commands (file `sta.tcl`):

The command:

```
set fig [ttv_LoadSpecifiedTimingFigure msdp2_y]
```

reads the timing database from disk.

The command:

```
set stbfig [stb $fig]
```

launches the static timing analysis. The `stb` function returns a pointer on the newly created figure, which back-annotates the timing database with timing propagation information.

The function:

```
stb_DisplaySlackReport [fopen slack.rep w] $fig * * ?? 10  all 10000
```

displays a global slack report in the file `slack.rep`.

## 8.2. Timing Checks

The next sections explain how timing checks are performed. They describe the more common situations one can be faced to, i.e.:

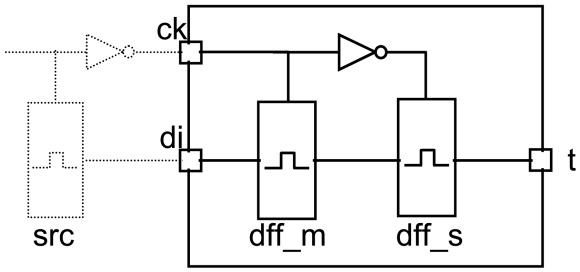
- Input to latch
- Latch to latch
- Latch to output

For each situation, an example of slack report is shown, and we explain the details of the timing checks calculation.

## 8.2.1. Input to Latch

### Inputs Specifications

Regarding input specifications, the STA engine of HITAS makes the assumption that input data is coming from a latch clocked on the opposite phase of the one the data arrives on. In our flip-flop example, `dff_m` is opened on the high state of `ck`, so `di` is supposed to come from a latch opened on the low state of `ck`.

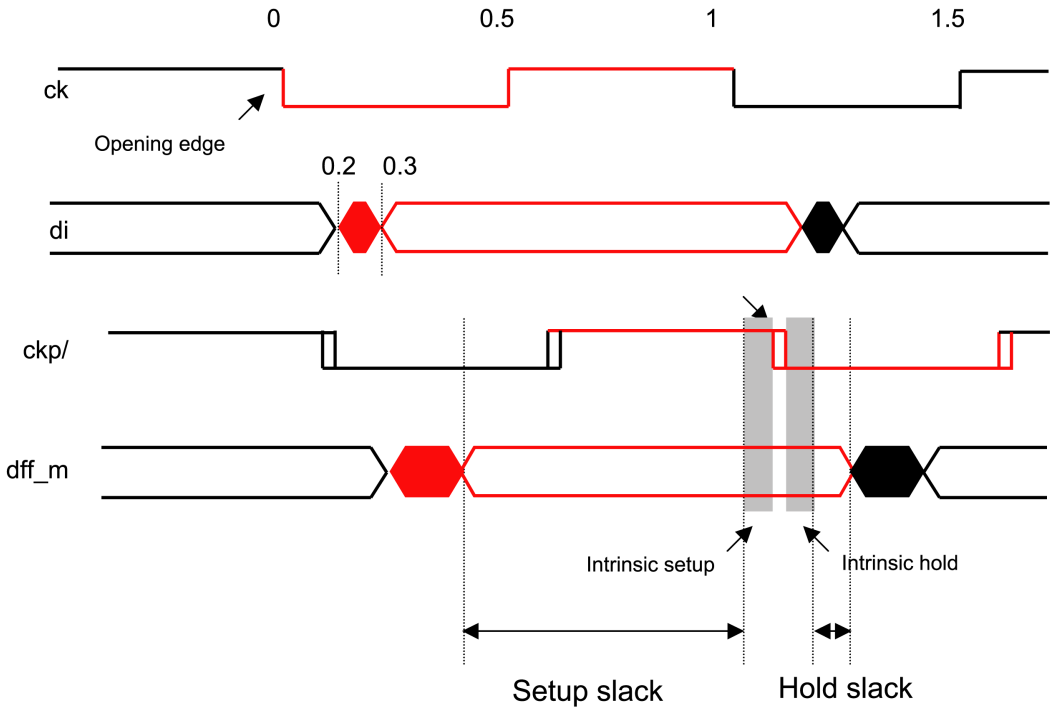



As a result, `di` should be specified as coming from `ck` falling, i.e. when the latch `src` opens. The corresponding SDC commands should look like:

```
create_clock -period 1000 -waveform {500 0} ck
set_input_delay -clock -ck -clock_fall -min 200 di
set_input_delay -clock -ck -clock_fall -max 300 di
```

### Timing Checks Description

Diagram below illustrates the way `set_input_delay` directives are propagated throughout the design, and where timing checks are performed.



### Setup Slack

Input to latch setup slack report is described in the `slack.rep` file

```
Path (4) :  Slack of 0.762
     DATA VALID:
        Delay
      Acc    Delta    R/F     Cap[pf]   Type   Node_Name          Net_Name      Line
    ------------------------------------------------------------------------------------
     0.300   0.000 0.200 R      0.034          di                 di
     0.498   0.198 0.310 F      0.028    (L)   dff_m              dff_m       master
    ------------------------------------------------------------------------------------
     0.498   0.198            (total)

     DATA REQUIRED:
        Delay
      Acc    Delta    R/F     Cap[pf]   Type   Node_Name          Net_Name    Line
    ------------------------------------------------------------------------------
     0.000   0.000 0.200 F      0.016    (C)   ck                 ck
     0.239   0.239 0.258 R      0.046   (CK)   ckn                ckn          inv
     0.340   0.101 0.140 F      0.036   (CK)   ckp                ckp          inv
     0.260  -0.081                             [INTRINSIC SETUP]
     1.260  +1.000                             [NEXT PERIOD]
    ------------------------------------------------------------------------------
     1.260   0.260            (total)
```

The value of the setup slack is given by clock_path - data_path = 1260ps - 498ps = 762ps. The intrinsic setup corresponds to an additional delay which models the amount of time required for secure memorization of the data.

### Hold Slack

Input to latch hold slack report is described in the `slack.rep` file

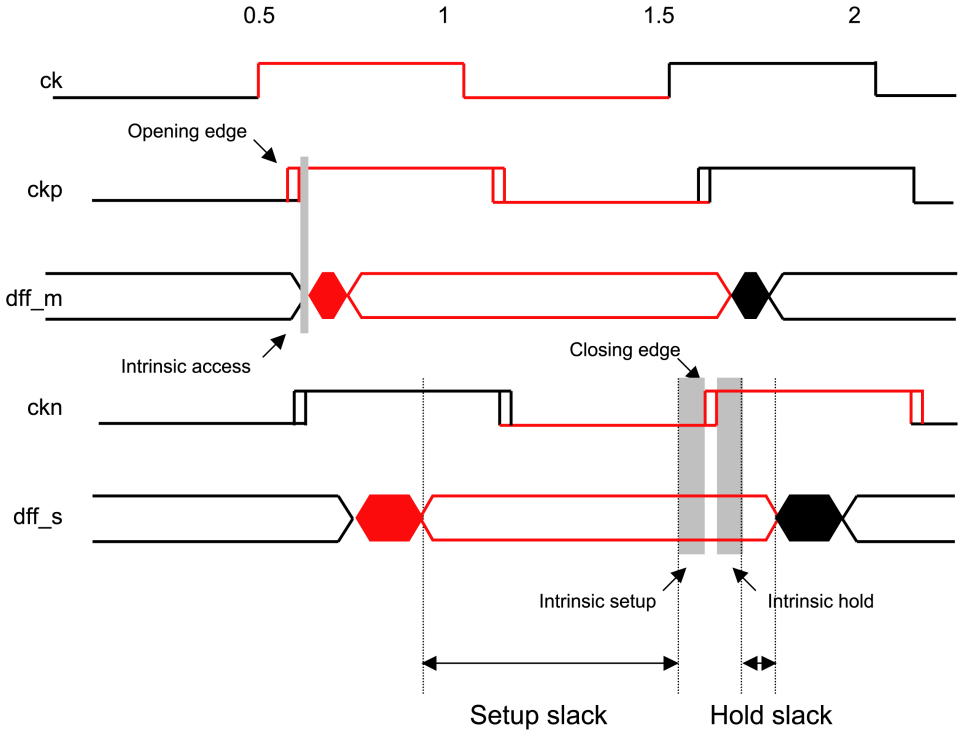
```
Path (2) :  Slack of 0.005
     DATA VALID:
        Delay
      Acc    Delta    R/F     Cap[pf]   Type   Node_Name          Net_Name      Line
    ------------------------------------------------------------------------------------
     0.200   0.000 0.200 F      0.034          di                 di
     0.542   0.342 0.508 R      0.028    (L)   dff_m              dff_m       master
    ------------------------------------------------------------------------------------
     0.542   0.342            (total)

     DATA REQUIRED:
        Delay
      Acc    Delta    R/F     Cap[pf]   Type   Node_Name          Net_Name    Line
    ------------------------------------------------------------------------------
     0.000   0.000 0.200 F      0.016    (C)   ck                 ck
     0.239   0.239 0.258 R      0.046   (CK)   ckn                ckn          inv
     0.537  +0.298                             [INTRINSIC HOLD]
    ------------------------------------------------------------------------------
     0.537   0.537            (total)
```

The value of the hold slack is given by data_path - clock_path = 542ps - 537ps = 5ps. The intrinsic hold corresponds to an additional delay which models the amount of time required for ensuring that the next cycle's data is not memorized in the current cycle.

## 8.2.2. Latch to Latch

### Timing Checks Description

Latch to latch timing checks require no additional configuration, as they are based upon the signals already propagated from inputs, and upon the clock specification. The propagation of the s.w., and corresponding timing checks are described in the following timing diagram:




### Setup Slack

Latch to latch setup slack report is described in the `slack.rep` file

```
Path (3) :  Slack of 0.284
     DATA VALID:
        Delay
      Acc     Delta     R/F      Cap[pf]    Type    Node_Name             Net_Name      Line
    ------------------------------------------------------------------------------------------
    -0.500    0.000 0.200 R        0.016     (C)    ck                    ck
    -0.399    0.101 0.128 F        0.046    (CK)    ckn                   ckn             inv
    -0.236    0.164 0.169 R        0.036    (CK)    ckp                   ckp             inv
    -0.152    0.083 0.139 F        0.028     (L)    dff_m                 dff_m        master
     0.090    0.242 0.189 R        0.040            n11                   n11             inv
     0.321    0.231 0.305 F        0.089     (L)    dff_s                 dff_s         slave
    ------------------------------------------------------------------------------------------
     0.321    0.821              (total)

     DATA REQUIRED:
        Delay
      Acc     Delta     R/F      Cap[pf]    Type    Node_Name             Net_Name    Line
    -------------------------------------------------------------------------------------
     0.500    0.000 0.200 R        0.016     (C)    ck                    ck
     0.601    0.101 0.128 F        0.046    (CK)    ckn                   ckn          inv
     0.605   +0.005                                 [INTRINSIC SETUP]
    -------------------------------------------------------------------------------------
```

```
     0.605   0.105            (total)
```

### Hold Slack

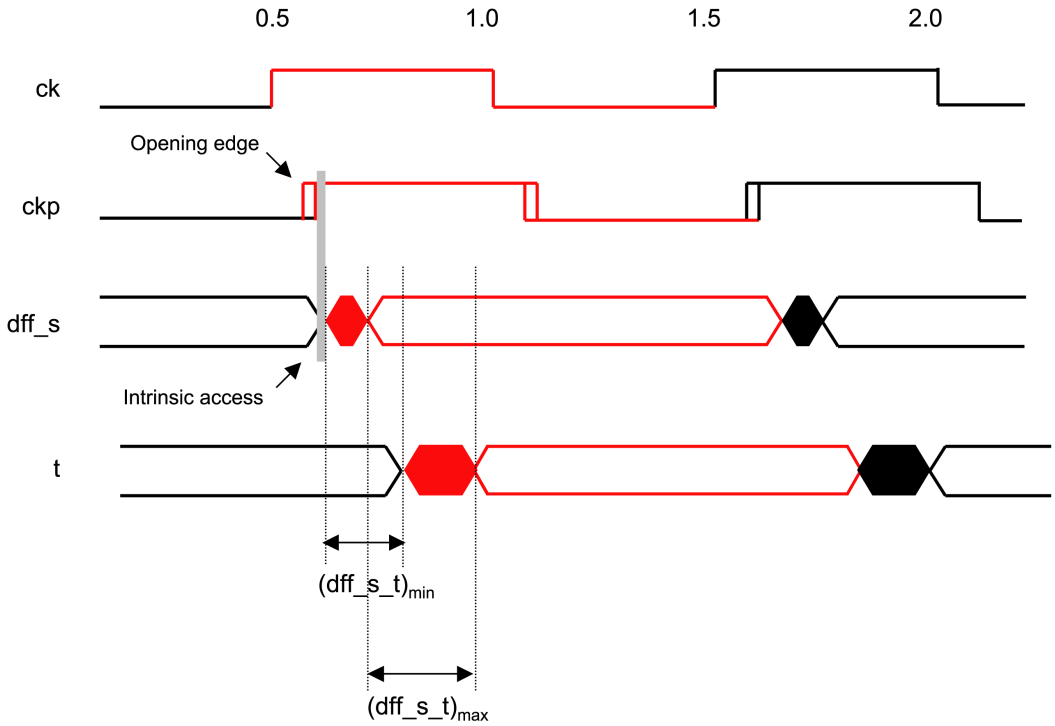
Latch to latch hold slack report is described in the `slack.rep` file

```
Path (3) :  Slack of 0.146
     DATA VALID:
        Delay
      Acc    Delta    R/F     Cap[pf]   Type   Node_Name           Net_Name     Line
 ------------------------------------------------------------------------------------
     -0.500   0.000 0.200 R      0.016    (C)   ck                  ck
     -0.399   0.101 0.128 F      0.046   (CK)   ckn                 ckn           inv
     -0.281   0.119 0.177 R      0.028    (L)   dff_m               dff_m      master
     -0.223   0.057 0.088 F      0.040          n11                 n11           inv
      0.106   0.329 0.447 R      0.089    (L)   dff_s               dff_s       slave
 ------------------------------------------------------------------------------------
      0.106   0.606            (total)

      DATA REQUIRED:
        Delay
      Acc    Delta    R/F     Cap[pf]   Type   Node_Name           Net_Name   Line
 ------------------------------------------------------------------------------------
      0.500   0.000 0.200 R      0.016    (C)   ck                  ck
      0.601   0.101 0.128 F      0.046   (CK)   ckn                 ckn         inv
      0.764   0.164 0.169 R      0.036   (CK)   ckp                 ckp         inv
      0.960  +0.196                             [INTRINSIC HOLD]
     -0.040  -1.000                             [PREVIOUS PERIOD]
 ------------------------------------------------------------------------------------
     -0.040   0.460            (total)
```

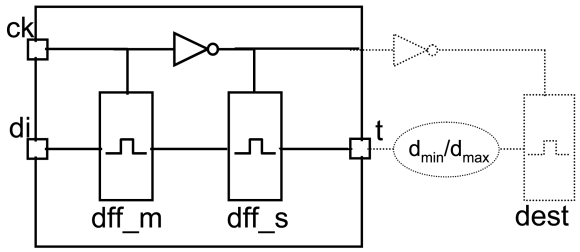
## 8.2.3. Latch to Output

### Output Constraints

Still based on the flip-flop design described above, the timing propagation on output `t` is done as follow:




In order to get setup and hold slacks on the output, one must define timing constraints on `t`. These timing constraints are defined with the `set_output_delay` SDC function. The `set_output_delay` specifies propagation delays from output connector to the next memory element latching the data. As a result, min and max delays are defined as shown in the diagram below.




One must also define the edge the data will be latched by. Here, `dff_s` is closed on the high state of `ck`. The data launched by `t` is supposed to be latched by a memory element clocked on the opposite phase, i.e. closed on low state of `ck`. Therefore, constraints on `t` should be specified relative to falling edge of `ck` (when `dst` latch closes). The `set_output_delay` functions should be used as follow:

```
set_output_delay -clock ck -clock_fall -min 200 t
set_output_delay -clock ck -clock_fall -max 400 t
```

### Setup Slack

Latch to output setup slack report is described in the `slack.rep` file

```
Path (1) :  Slack of 0.030
     DATA VALID:
        Delay
```

```
   Acc    Delta    R/F     Cap[pf]   Type   Node_Name    Net_Name      Line
 ------------------------------------------------------------------------------
  0.000   0.000 0.200 F      0.016    (C)   ck           ck
  0.239   0.239 0.258 R      0.046   (CK)   ckn          ckn            inv
  0.340   0.101 0.140 F      0.036   (CK)   ckp          ckp            inv
  0.568   0.227 0.327 R      0.089    (L)   dff_s        dff_s        slave
  0.570   0.003 0.118 F      0.011    (S)   t            t              inv
 ------------------------------------------------------------------------------
  0.570   0.570            (total)

 -> Specification: Must be stable after 0.600
```

The setup time is calculated with the maximum set_output_delay value - maximum data path - which is 400ps. As the period is 1000ps, data must arrive before time 1000 - 400 = 600ps. The setup slack is given by 600 - 570 = 30ps.

### Hold Slack

Latch to output hold slack report is described in the `slack.rep` file

```
Path (5) :  Slack of 0.635
     DATA VALID:
        Delay
    Acc    Delta    R/F     Cap[pf]   Type   Node_Name    Net_Name     Line
 ------------------------------------------------------------------------------
  0.000   0.000 0.200 F      0.016    (C)   ck           ck
  0.239   0.239 0.258 R      0.046   (CK)   ckn          ckn           inv
  0.385   0.146 0.235 F      0.089    (L)   dff_s        dff_s       slave
  0.435   0.050 0.082 R      0.011    (S)   t            t             inv
 ------------------------------------------------------------------------------
  0.435   0.435            (total)

 -> Specification: Must be stable before -0.200
```

The hold time is calculated with the minimum set_output_delay value - minimum data path - which is 200ps. The hold slack is given by data path - clock path = 435 + 200 - 0 (the clock is ideal in the set_output_delay definition) = 635ps.

# Chapter 9. Addaccu

This example describes a global timing characterization methodology. It is based upon a simple 4-bit adder-accumulator.

This example takes place in the directory `addaccu/`.

Timing characterization provides the timing properties (or constraints) of a macroblock, generally in the Liberty format. The purpose of the timing characterization is to provide other tools in the design flow - physical design, chip-level STA - with sufficient timing information about the macroblock, so that those tools can perform their task correcly. Typically, the information given in the Liberty file are:

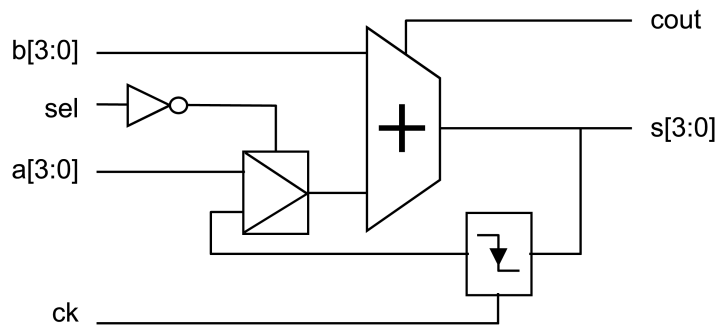
- Setup and Hold Constraints (Sequential): the maximum and minimum arrival times of data signals on input pins relative to clock signals (on clock pins)
- Access Times (Sequential): the maximum and minimum departure times of data signals on output pins relative to clock signals (on clock pins)
- Propagation Times (Combinational): the maximum and minimum path delays between inout and output pins

We describe here a methodology for secure timing characterization of macroblocks and cells. The following steps are involved:

- Construction of the macroblock timing database
- Identification of the paths involved in the timing constraints
- Validation of the paths and accuracy check by SPICE simulation
- Timing characterization for different slopes and loads
- Timing characterization by SPICE simulation for different slopes and loads

## 9.1. Design Description

The addaccu chip consists of a four-bit adder, a four-bit register, and a 2 to 1 four-bit multiplexer.



The circuit performs an addition between either the `b[3:0]` and `a[3:0]` inputs when sel is set to 0, or between `b[3:0]` and the content of the four-bit register when sel is set to 1. The content of the register is overwritten by the values of the outputs `s[3:0]` on each falling edge of the clock, ck..

## 9.2. Construction of the Timing Database

The temperature and supplies specifications take place in the `addaccu_schem.spi` file:

```
.TEMP 125
.GLOBAL vdd vss
Vsupply vdd vss DC 2.0
Vground vss 0 DC 0
```

As the `addaccu_schem.spi` subcircuit is not instantiated, the `vdd` and `vss` signals appear in the `.GLOBAL` statement.

In the present example, the `.INCLUDE` statement is used for technology file integration:

```
.INCLUDE ../techno/bsim4_dummy.ng
```

The additional configuration required for the database construction takes place in the `db.tcl` script. The script also launches the commands that effectively generate that database.

Configuration variables are set in the Tcl script by the mean of the `avt_config` function.

`avt_config tasGenerateConeFile yes`
: tells the tool to dump on disk the .cns file, which contains the partitions (the cones) created by the partitioning algorithm.

`avt_config avtVerboseConeFile yes`
: tells the tool to dump on disk the .cnv file, which is a more readable version.

`avt_config simVthLow 0.2`
: Low threshold of slope definition

`avt_config simVthHigh 0.8`
: High threshold of slope definition

`avt_config simToolModel ngspice`
: tells the tool the technology file type (which simulator it is designed for)

The construction itself is done through the command `hitas`:

```
avt_LoadFile addaccu_schem.spi
set fig [hitas addaccu]
```

## 9.3. Timing Paths Identification

The `paths.tcl` script reports the timing paths involved in the constraints described above (setup, hold, access and combinational paths).

Let's have a look at the following code sequence in the script:

```
# Setup / Hold paths
set file [fopen $figname.setuphold w]
ttv_DisplayConnectorToLatchMargin $file $fig * "split all"
fclose $file
```

The function `ttv_DisplayConnectorToLatchMargin` displays the setup and hold constraints associated with the input pins, related to the clock signal created with the `create_clock` statement (note that the frequency information is irrelevant here, as setup and hold constraints do not depend upon frequency - the syntax just requires it). Precisely, for each input pin, the `ttv_DisplayConnectorToLatchMargin` function displays all possible setup and hold values, depending on the latch involved. All information about the data paths, clock paths and latch involved is reported in the `addaccu.setuphold` file.

Now let's look at the maximum access paths detection. The related code sequence is:

```
# Max access paths
set file [fopen $figname.accessmax w]
set pathlist [ttv_GetPaths $fig * s\[*\] ?? 0 critic access max]
ttv_DisplayPathListDetail $file $pathlist
fclose $file
```

The `ttv_GetPaths` function looks for all (argument `0`) the access paths ending on signals s[0:3], using maximum path values for data and clock. The whole detail of those paths is reported in the `addaccu.accessmax`.

The next code sequence deals with minimum access paths, and is very similar to the one described above. The `ttv_GetPaths` function looks here for all the access paths ending on signals s[0:3], using minimum path values for data and clock. The whole detail of those paths is reported in the `addaccu.accessmin` file.

The final code sequence deals with combinational paths between input and output pins:

```
# Combinatorial paths
set file [fopen $figname.comb w]
set pathlist [ttv_GetPaths $fig a\[*\] s\[*\] ?? 0 critic path max]
ttv_DisplayPathListDetail $file $pathlist
set pathlist [ttv_GetPaths $fig b\[*\] s\[*\] ?? 0 critic path max]
ttv_DisplayPathListDetail $file $pathlist
set pathlist [ttv_GetPaths $fig sel s\[*\] ?? 0 critic path max]
ttv_DisplayPathListDetail $file $pathlist
set pathlist [ttv_GetPaths $fig ck s\[*\] ?? 0 critic path max]
ttv_DisplayPathListDetail $file $pathlist
fclose $file
```

The whole detail of those paths is reported in the `addaccu.comb` file.

All these paths will be the ones which will be used to characterize the design, it is therefore necessary to carefully check that there are relevant.

## 9.4. Timing Paths Validation by SPICE simulation

The `paths_simu.tcl` script performs the same task as the `paths.tcl` script, and re-simulates the paths with NG-SPICE. NG-SPICE is a freeware SPICE simulator (Berkeley license). It is provided with this tutorial. Binaries are in `../bin/Linux/` and `../bin/Solaris/`. For the sake of understanding, the set of paths - reported and simulated - has been reduced to the ones originating from some inputs only. HITAS actually generates a SPICE deck with all the stimuli allowing for signal propagation. It automatically invokes the simulator and retrieve the results, which are integrated in the reports. It just needs the following configuration:

`avt_config avtSpiceString "./bin/Solaris/ngspice -b $"`
: The command line which will be invoked by HITAS

`avt_config SimToolModel ngspice`
: Tells HITAS the SPICE format to use for the SPICE deck

`avt_config simTechnologyName ../techno/bsim4_dummy.ng`
: The technology file to include in the SPICE deck (`.INCLUDE`)

`avt_config avtSpiceOutFile $.log`
: Tells HITAS the suffix of the file containing the simulation results, required unless the simulator fixes this.

`ttv_DisplayActivateSimulation yes`
: The flag for activating the re-simulation of reported paths

Just invoke `paths_simu.tcl` to run the simulations. As before, the results are displayed in the files `addaccu.comb`, `addaccu.setuphold`, `addaccu.accessmin` and `addaccu.accessmax`. An additionnal column gives the NG-SPICE values.

The configuration is given for ngspice since the simulator is provided however another simulator can be used, for example the configuration for hspice would be something like:

`avt_config avtSpiceString "hspice $"`
: The command line which will be invoked by HITAS

`avt_config SimToolModel hspice`
: Tells HITAS the SPICE format to use for the SPICE deck

`avt_config simTechnologyName ../techno/bsim4_dummy.hsp`
: The technology file to include in the SPICE deck (`.INCLUDE`)

`ttv_DisplayActivateSimulation yes`
: The flag for activating the re-simulation of reported paths

## 9.5. Timing Characterization (.lib)

The timing abstraction configuration takes place in the `charac.tcl` script. Let's review the configuration needed:

`inf_SetFigureName addaccu`
: tells the tool to apply the SDC constraints to the design addaccu.

`create_clock -period 3000 -waveform {0 1500} ck`
: Creates a clock on signal `ck`. Period is not relevant, but required by the SDC syntax

`inf_DefineSlopeRange default {25ps 50ps 100ps 200ps 400ps} custom`
: The set of slopes to be applied on input pins

`inf_DefineCapacitanceRange default {8fF 16fF 32fF 64fF} custom`
: The set of loads to be applied on output pins

The timing abstraction is done through the command `tmabs`:

```
set abs [tmabs $fig NULL * * * -verbose -detailfile $figname.clog]
lib_drivefile [list $abs] NULL addaccu.lib max
```

A file `addaccu.clog` is issued, which contains all the paths used for characterization.

## 9.6. Timing Characterization (.lib) by SPICE simulation

The referent script is `charac_simu.tcl`. The simulation configuration is the same as in the `paths_simu.tcl` script, except for the two following lines:

```
# Simulation speed-up
avt_config simOutLoad dynamic
avt_config avtTechnologyName ../techno/bsim4_dummy.ng
```

It just tells HITAS to transform out-of-path transistors into equivalent capacitances.

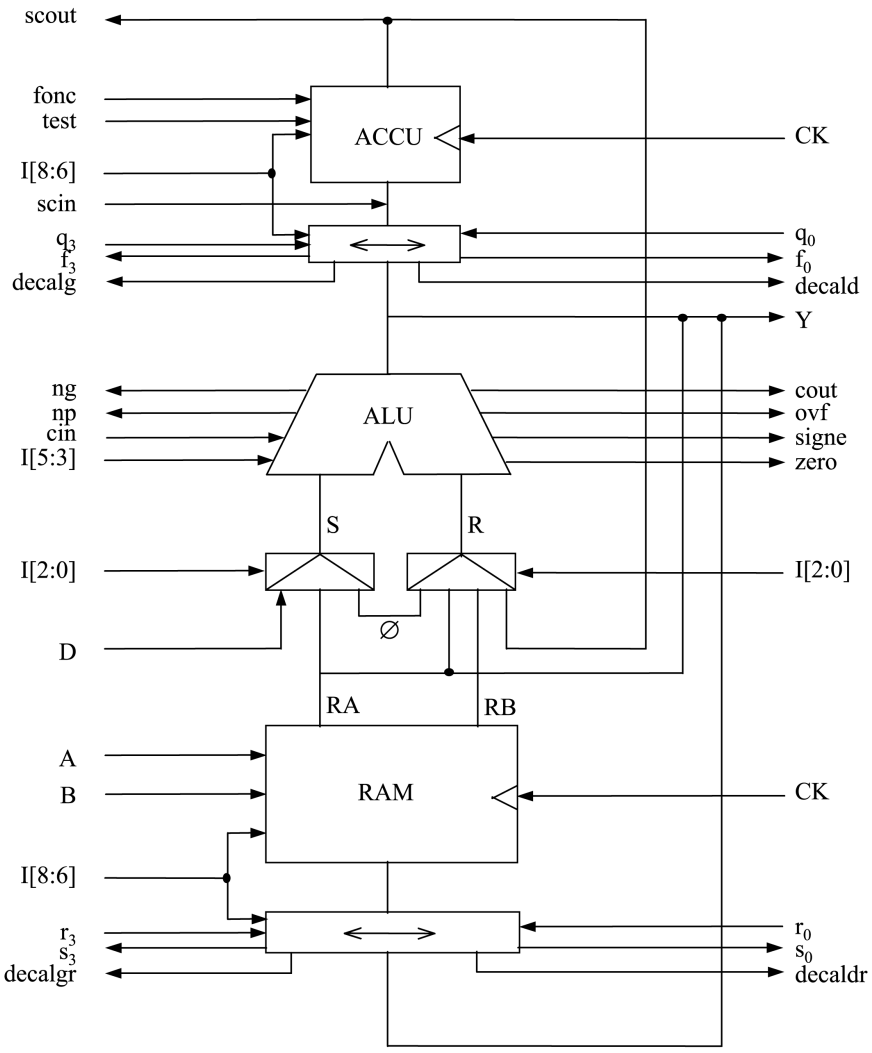
The `charac_simu.tcl` script simulates all the paths used for characterization and issues the `addaccu_golden.lib` file. It uses a cache mechanism in order to avoid resimulating several times the same path (clock paths for instance).

# Chapter 10. CPU2901

## 10.1. Design Description

This example presents HITAS database construction, case analysis, OCV, Xtalk analysis, based upon a small 4-bit microprocessor design.

It takes place in the `cpu2901/` directory.



# 10.2. Database Generation

## 10.2.1. Global Configuration

The complete configuration required for the database generation takes place in the `db.tcl`. The script also launches the commands that effectively generate that database.

Configuration variables are set in the Tcl script by the mean of the `avt_config` function.

`avt_config tasGenerateConeFile yes`
: tells the tool to dump on disk the .cns file, which contains the partitions (the cones) created by the partitioning algorithm.

`inf_SetFigureName cpu2901`
: tells the tool to apply the SDC constraints to the cpu design.

`set_case_analysis 0 test`
: Applies a 0 constraint on the pin `test`

`set_case_analysis 1 func`
: Applies a 1 constraint on the pin `func`

The temperature and supplies specifications take place in the `cpu2901.spi` file:

```
.TEMP 125
.GLOBAL vdd vss
Vsupply vdd 0 DC 1.62
Vground vss 0 DC 0
```

As the `cpu2901.spi` subcircuit is not instantiated, the `vdd` and `vss` signals appear in the `.GLOBAL` statement.

## 10.2.2. Database Generation

The generation launch is done through the command hitas:

```
avt_LoadFile cpu2901.spi
set fig [hitas cpu2901]
```

# 10.3. Database Analysis

## 10.3.1. Path Searching with the Tcl Interface

The complete configuration required for the database browsing takes place in the `report.tcl`.

The command:

```
set fig [ttv_LoadSpecifiedTimingFigure cpu2901]
```

reads the timing database from disk.

The command:

```
set clist [ttv_GetPaths $fig * * rr 5 critic path max]
```

gives the 5 most critical paths (`critic` and `path` arguments) of the design, that begin and end on a rising transition (`rr` argument), with no specification of signal name (`* *` arguments), in the database pointed out by `$fig`. The function returns a pointer on the newly created list.

The command:

```
ttv_DisplayPathListDetail $log $clist
```

displays in the log file the detail of all the paths of the path list given by the `ttv_GetPaths` function.

# 10.4. Timing Checks

The complete configuration required for stability analysis takes place in the `sta.tcl`.

## 10.4.1. Timing Constraints

Timing constraints are set in SDC format. Let's review the constraints commands applied to the cpu2901:

`inf_SetFigureName cpu2901`
: tells the tool to apply the SDC constraints to the design cpu2901.

`create_clock -period 10000 -waveform {5000 0} ck`
: Creates of clock of period 10000

`set_input_delay -min 2000 -clock ck -clock_fall [all_inputs]`

`set_input_delay -max 3000 -clock ck -clock_fall [all_inputs]`
: Defines a switching window between times 2000 and 3000 on the input connectors

## 10.4.2. STA

Launch of the static timing analysis is done by invoking the following commands:

As before, the command:

```
set fig [ttv_LoadSpecifiedTimingFigure cpu2901]
```

reads the timing database from disk.

The command:

```
set stbfig [stb $fig]
```

launches the STA

The function:

```
stb_DisplaySlackReport [fopen slack.rep w] $fig * * ?? 10  all 10000
```

displays a global slack report in the file `slack.rep`.

### 10.4.3. OCV

Comment out the command `inf_DefinePathDelayMargin` and observe the differences in the slack file. This command adds a margin of 1ns on all data paths

### 10.4.4. Crosstalk Analysis

Launch of the crosstalk analysis is done by invoking the following commands (script `xtalk.tcl`):

As before, the command:

```
set fig [ttv_LoadSpecifiedTimingFigure cpu2901]
```

reads the timing database from disk.

The crosstalk analysis is activated by switching on the following variables:

```
avt_config stbDetailedAnalysis yes
avt_config stbCrosstalkMode yes
```

The command:

```
set stbfig [stb $fig]
```

launches the crosstalk-aware STA

The function:

```
stb_DisplaySlackReport [fopen slack_xtalk.rep w] $fig * * ?? 10  all 10000
```

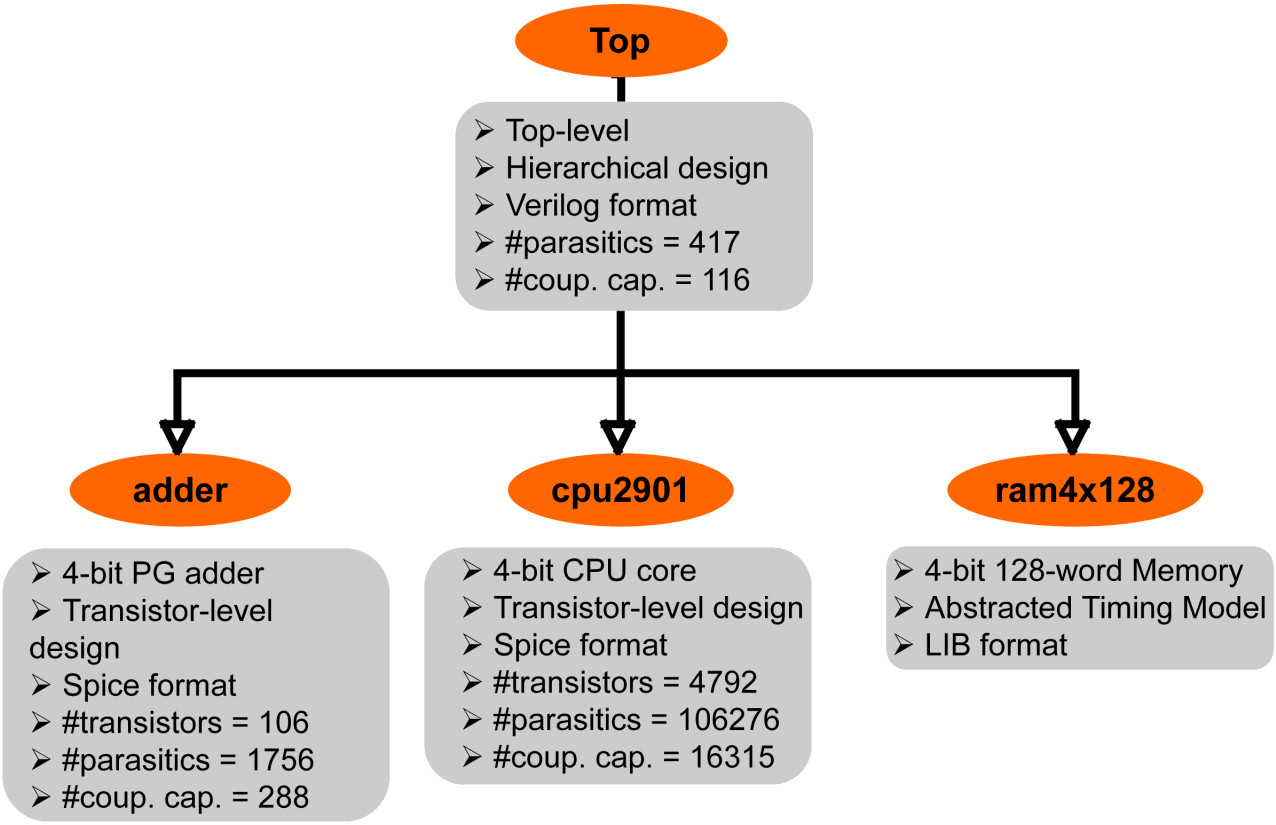
prints a global slack report in the file `slack_xtalk.rep`, displaying variations due to crosstalk effects.

# Chapter 11. Hierarchical Analysis

## 11.1. Design Description

This example illustrates the complete STA and SI of a hierarchical macro. It takes place in the `h_macro/` directory.

The design is made up of two levels of hierarchy as in the following diagram:




The two low-level blocks adder and cpu2901 are full-custom blocks (analyzed in the preceding examples), designed at transistor-level, and extracted as flat transistor net-lists with parasitics (including coupling capacitances).

The ram4x128 is an abstracted model of a 128-4bit-word memory.

The top-level instantiates these three blocks, and is back-annotated with parasitics (including coupling capacitances).

# 11.2. Database Generation

## 11.2.1. Global Configuration

The complete configuration required for the database generation takes place in the `db.tcl`. The script also launches the commands that effectively generate that database.

`avt_config avtLibraryDirs ".:../lab3_adder:../lab6_cpu"`
: tells the tool where to find timing databases for the lower levels of hierarchy.

`avt_config avtVddName vdd`
: tells the tool which signal must be considered as a power supply, necessary as netlist is not spice.

`avt_config avtVssName vss`
: idem for ground signal identification.

`avt_config tasHierarchicalMode yes`
: tells the tool to work hierarchically.

`avt_LoadFile ./ram4x128.lib lib`
: load the abstracted block ram4x128.lib

`avt_LoadFile top.v verilog`
: load the Verilog netlist of top

`avt_LoadFile top.spef spef`
: load the parasitics back-annotation in SPEF format

## 11.2.2. Database Generation

The generation launch is done through the command hitas:

```
set fig [hitas top]
```

# 11.3. Database Analysis

## 11.3.1. Path Searching with the Tcl Interface

The complete configuration required for the database browsing takes place in the `report.tcl`.

The command:

```
set fig [ttv_LoadSpecifiedTimingFigure top]
```

reads the timing database from disk.

The command:

```
set clist [ttv_GetPaths $fig * * uu 5 critic path max]
```

gives the 5 critical paths of the design.

The command:

```
ttv_DisplayPathListDetail stdout $clist
```

displays on the standard output the detail of all the paths of the path list given by the `ttv_GetPaths` function.

## 11.4. Timing Checks

The complete configuration required for stability analysis takes place in the `sta.tcl`.

### 11.4.1. Timing Constraints

Timing constraints are set in SDC format. For this example we use the same constraints specified for the cpu2901 example.

### 11.4.2. STA

Launch of the static timing analysis is done by invoking the following commands:

As before, the command:

```
set fig [ttv_LoadSpecifiedTimingFigure top]
```

reads the timing database from disk.

The command:

```
set stbfig [stb $fig]
```

launches the STA

The function:

```
stb_DisplaySlackReport [fopen slack.rep w] $fig * * ?? 10  all 10000
```

displays a global slack report in the file `slack.rep`.

# Chapter 12. Analog Blocks Handling

## 12.1. Objective

HITAS is designed to compute propagation delays in digital designs. The advantage of this restrictive target is to enable very fast computing times. The drawback is that non-digital block characterization is not directly handled by HITAS and should be supplied to 3rd-party analog simulators. However, HITAS provides various ways to link with external characterizations.

This example presents two of the simplest ways with which HITAS can deal with analog blocks. It takes place in the `blackbox/` directory.

## 12.2. Database Generation

The complete configuration required for the timing database generation takes place the `db.tcl` script. The temperature and power supplies are specified directly in the `circuit.spi` file.

## 12.3. Ignore Function

The simplest and often sufficient technique for handling analog parts of a design is to tell HITAS to explicitly ignore them so that they will be included in the timing database.

HITAS can ignore specified components with `inf_DefineIgnore` command. This directive can be used to ignore transistors, instances, resistances, capacitances and diodes by specifying them by name

It is equivalent to commenting out elements in the spice netlist

See HiTas Reference Guide for further details.

`inf_SetFigureName circuit`
: tells the tool to apply the SDC constraints to the design.

`inf_DefineIgnore resistances R1`
: tells the tool to ignore the resistance named R1 in the design.

`inf_DefineIgnore instances INV1`
: tells the tool to ignore the instance inverter named INV1 in the design.

The first ignore directive is to remove what HITAS considers to be a short circuit between the power supplies. A resistance such as this causes problems for the identification of power supply nets and so must be handled like this.

The output logging function has been activated in the `db.tcl` script for file parsing statistics (see documentation of `avtLogFile` and `avtLogEnable` in the reference guide for more details). Look at the generated log file to see the effect of the directive.

The second directive effectively leaves a hole in the netlist, however, this poses no problem for the timing database generation for the rest of the circuit. Try running the path report script (`report.tcl`) both with and without this directive to see the effect. Leave this directive commented out for the next section.

## 12.4. Integration in a Hierarchical Netlist

The second way of handling analog parts is the incorporation of timings from a `.lib` file to model the timing of a block (anolog or otherwise) instantiated within a hierarchical netlist. In order to use this method it is first of all necessary to create "analog holes" in the netlist where these blocks are instantiated. This is done with the `avt_SetBlackBoxes` function, taking as argument the list of the sub-circuits to blackbox.

The default behavior of HITAS is not to try to fill the "holes". To tell the tool to fill the holes with timing characterizations, the `tasBlackboxRequiresTimings` variable is set to `yes` in the `db.tcl` script.

The timing information for these "holes" must be provided from an external timing database, this is typically done by loading an appropriate `.lib` file.

In this example, we will be using an external `.lib` to represent the timings for the flip-flop. Although this is not really an analog circuit, the procedure would be the same for an analog block and a flip-flop is a simple example containing setup, hold and access arcs.

To try this, you should recreate the timing database with the following lines in the appropriate script:

```
avt_SetBlackBoxes {msdp2_y}
avt_config tasBlackboxRequiresTimings yes
...
avt_LoadFile ./msdp2_y.lib lib
```

The timing arcs for the instances `msdp2_y` are directly integrated in the new database. The database for `circuit` is flat and does not contain instances of `msdp2_y`.

Examine the timing database using the path report script and compare with the path reports obtained without "blackboxing" of the flip-flops.

# Chapter 13. SSTA

## 13.1. Principles

HITAS SSTA is a Monte-Carlo like analysis: it is based on a collection of STA samples. Each STA sample is based upon the creation of a timing database sample, constructed by picking up random values for the statistical parameters embedded in either the SPICE netlist or the technology files. An SSTA sample consists therefore of a timing database and a STA run. In the end, there are as many different timing databases and STA runs as SSTA samples. STA runs are of course highly configurable, in order to extract any relevant information.

## 13.2. Analysis on the ADDACCU

This example features 2 analysis: A SSTA analysis and the PATH analysis. In the SSTA analysis, slacks are computed, sorted and displayed in an efficient way. In the PATH analysis, particular paths are retrieved and their variations are displayed. For each of those analyses, 50 runs are performed on the ADDACCU design.

### 13.2.1. Generating the data for the SSTA analysis

The script to generate the SSTA data is no different from a standard STA script except for 2 TCL instructions inserted at the beginning and at the end of the script. The script used to generate those data is named `ssta.tcl`.

The first instruction is responsible for the handling of the 50 runs:

```
runStatHiTas 50 -incremental -result slacks.ssta -storedir store
```

In case the configuration variable `avtLibraryDirs` is used, this instruction must be placed after the configuration because modifications are applied to this configuration variable to get proper search paths.

Calling `runStatHiTas` will launch 50 separate runs of the `ssta.tcl` script one by one (multiprocessing is not used in the tutorial). Each run will have its data written into the file `slacks.ssta` and the required information to display the slack details will be stored in the directory `store`.

Using the option `-storedir` is not mandatory but if it is not used, only the slack summaries will be available.

The `-incremental` flag is set to enrich any previous execution of the SSTA database so 50 more runs will be added to any existing set of runs.

The second instruction is responsible for the handling of STA data:

```
ssta_SlackReport -senddata $stbfig simple
```

`ssta_SlackReport` is called after the `stb` API execution. The stability figure and a slack data output mode is given to the function. The only mode available at the moment is `simple`. Called in this form, it retrieves at most the 10000 worst negative slacks from the stability figure. If no negative slacks are found, the worst positive one is searched. The slack descriptions are then written to the file `slacks.ssta` for a future use.

## 13.2.2. Reporting the results for SSTA analysis

The script `slack_analysis.tcl` reads the `slacks.ssta` file and uses the data in the directory `store` to display some results.

### Slack occurrence

The first kind of result output is generated using the command:

```
ssta_SlackReport -display "slacks.ssta" $ofile -storedir store
```

The report is driven to the file `slack_report.log`.

At the beginning of the report, yield information is printed: the total number of runs, the number of runs with negative holds, the number of runs with negative setups, the number of runs with PVT errors and the global yield.

In a second part, each negative slack is printed with the number of occurrence the of slack, the run number where the slack is the worst, some statistical information and the slack description. As the `store` directory is given as an argument to the function, the detail of each negative slack is displayed after this summary.

Finally, at the end, a list with the different seeds used to generate each run database is displayed.

### Worst slack distributions

The second kind of result is output in a set of file through a gnuplot graphical file representation by using the command:

```
ssta_SlackReport -plot "slacks.ssta" "distrib"
```

`distrib` is a prefix that will be used to generate the gnuplot files. There are 2 plots: 1 for the setups slacks and 1 for the holds slacks.

The gnuplot command file will be named `distrib.holds.plt` and `distrib.setups.plt`. The corresponding data files are `distrib.holds.plt.dat` and `distrib.holds.plt.dat`.

The distributions can be viewed using the UNIX command:

```
gnuplot <command file>
```

## 13.2.3. Generating the data for the PATH analysis

The script to generate the PATH data is very easy. In this case there is no need for stability to be performed. The only operation to be done is to retrieve the list of desired paths to analyse. In this example all paths and accesses will be taken. As for the SSTA data generation, 2 TCL instructions are inserted at the beginning and at the end of the script. In between, the UTD is built and the path search is performed. The script used to generate those data is named `paths.tcl`.

The first instruction is responsible for the handling of the 50 runs:

```
runStatHiTas 50 -incremental -result paths.ssta -storedir store_paths
```

The instruction is placed after the configuration variable `avtLibraryDirs`

After the UTD generation, the list of accesses and paths are extracted from the UTD and merged together:

```
set paths [concat [ttv_GetPaths $fig -access] [ttv_GetPaths $fig]]
```

The last instruction is responsible for the handling of PATH data:

```
ssta_PathReport -senddata $paths simple
```

`ssta_PathReport` is called with the path list and a path data output mode. The only mode available at the moment is `simple`. The path descriptions are written into the file `paths.ssta` for a future use.

## 13.2.4. Reporting the results for SSTA analysis

The script `path_analysis.tcl` reads the `paths.ssta` file and uses the data in the directory `store_paths` to display some results.

The path result report is generated using the command:

```
ssta_PathReport -display "paths.ssta" $ofile -storedir store_paths
```

The report is driven to the file `path_report.log`.

A summary of all paths/accesses is printed at the beginning of the report. Each path has an entry in the summary with some statistical information, the minimum delay of the path and the corresponding run number, the maximum delay and the corresponding run number and finally the path description.

As the `store_paths` directory is given as an argument to the function, the detail of each path is displayed after this summary. There are 2 details for each path: the detail for the minimum path value and the detail for the maximum path value.

Finally, at the end, a list with the different seeds used to generate each run database is displayed.

# Index

**No index for this document.**