Tcl Vivado Integration – The Big Picture

Tcl-Scripts (for initialisation, batch mode, or ran via GUI …)

Text based
Feed and *Response*
allowing "intelligent" automation
(far beyond plain repetition)

(external)
Tcl Console

Keyboard
& Mouse

Visual
Feedback

`start_gui`

GUI

integrated
Tcl Console

`stop_gui`

- **Locate** Command- and
- Word-**Separators**
- Carry-out **Substitutions**:
  - Content of Variables
  - Return Value of Subroutines
  - Non-printable Characters
- Determine **First Word**:
  - lookup as command
  - handover remaining words as arguments

while obeying some simple **Quoting Rules**

Syntax Analysis
(including Substitutions)

Command Lookup Table

internal
(subroutine)
implementation
of commands

for convenience, TCL
command equivalent of
actions executed via GUI
is shown in Tcl console

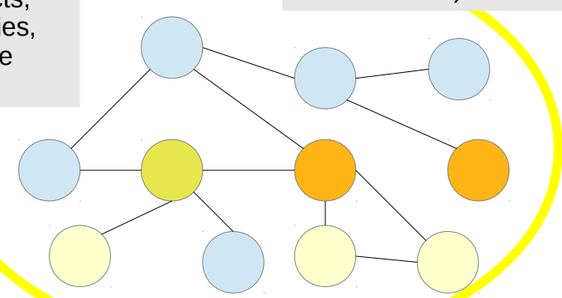
control (some)
highlighting details

Visualise design aspects of
interest (RTL source, circuit
schematics, constraints,
placement, test specification
and results ...)

enhance and transform
to final product

Query for objects,
access properties,
modify writeable
properties

In-Memory
Design
Representation



Technically a network of
- related ("interconnected") **objects**
- each from a given **class**, as e.g.
  - ***cell***, ***clock***, ***pin***, ***port***, ***net***, ...
- with a unique (hierarchical) **name**,
- and many more **properties**
(specific to the object's class)

Tcl Standard Commands, dealing with

- plain variables and higher level data structures
- flow control (including subroutines with arguments, return values, and forwarding/catching of errors)
- basic and advanced string and file processing
- … (and more) …

more or less "silently managed" by-products in project mode
prominent artefacts in non-project mode

Input Files (like HDL sources, constraints ...)

Intermediate Files (like design snapshots ...)

Output Files (like Bitstreams, Reports, …)

John Ousterhout
(started Tcl/Tk more
than 25 years ago)

The birth of Tcl
The Tcl scripting language grew out of my work on design tools for integrated circuits at the University of California at Berkeley in the early 1980's. […] In the fall of 1987, while on sabbatical at DEC's Western Research Laboratory, I got the idea of building an embeddable command language.
(Citation from: `www.tcl.tk/about/history.html`)

Tcl's Syntax Parsing Overview

Command and Word ***Separation***	sequences of ***white space*** (space or horizontal tab) separate words
	line endings and semicolons (`;`) separate commands
Variable and Return Value ***Substitution***	***variable content*** (aka. its "value") gets inserted for a dollar ($) followed by a valid variable name
	return value of a command gets inserted for parts of a command line enclosed in square brackets
Command ***Execution***	

HONOR QUOTING

A backslash (\)at the very end of a line is special, as it concatenates a line with the next one, **removing all leading white space**

Backslash-*Substitutions* – to insert non-graphic characters (e.g. `\n`, `\t`, `\a` … - like in C/C++)

Backslash-*Quoting* – to remove a special meaning from the directly following character (e.g. `\;` denotes a literal semicolon, not a *Command Separator*)

Quoting with Double Quotes (`"` … `"`) – to remove the special meaning from *Separators* and *Curly Brace Quoting*, while still allowing *Variable and Command Substitution*.

Quoting with Curly Braces (`{` … `}`) – to remove the special meaning from Everything enclosed.

For a full description in manual style see XILINX documentation ug835.

Applies ***recursively*** for nested command substitution, i.e. for ***square brackets inside square brackets***.

From the syntax viewpoint only the ***first word*** is relevant – all the following words are evaluated by the command itself!

array name must be followed by index in parentheses to access an element

unusual ("funny") names may need to be enclosed in curly braces

	Tcl (native)		Vivado
Read **T**he **F**ine **M**anual	Scarce – some hints in error messages when command execution fails … so: RTFM	Syntax Help	All commands have a `-help` option, providing extensive explanations (close to the full manual page)
	Rather weak – stronger for (historically) newer commands compared to the *"good ol' & ancient"* commands	Uniformity	very strong (especially in comparison native Tcl commands – which need to be used nevertheless)
Similar for commands `string`, `clock`, `file` …	Command Collections used for grouping, e.g. • `array set` … (get all elements of an array) • `array get` … (get all elements of an array) • `array size` … (get number of array elements) • `array names` … (get list of array indices) • …	Related Commands	Similar commands lexically grouped by common prefix, e.g.: •`get_`*`xyz`* (access *xyz* element of design model) •`set_`*`xyz`* (modify *xyz* element of design model) •`report_`*`xyz`* (generate *xyz* report) •…
list-related but creates proper list from its arguments	Prominent exception: commands for *list*-processing, e.g.: •`list` … … … •… •`llength` … •`lindex` … •`lsort` …		Similar command typically grouped under the same name, with an option to change their behaviour, e.g.: •`create_file_set` … •`create_fileset -constrset` … •`create_fileset -simset` …

commonality: first letter *ell*	*Minimalistic* approach (for economy), i.e.:
	• most commands with a "result" produce it as *return value*
	• so, to "print" it to the console use: `puts [` … `]`
	• and to save it in a file open a channel
	  • store return value of open: `set chan [open` …`]`
	  • use it as channel argument: `puts $chan [` … `]`
		Other (General)	*Regular* approach (for convenience), e.g. all `report_`*`xyz`* commands
			•by default produce their output on the console (window)
			•have a `-file` option to store the report in a *file*
			•have a `-return_string` option to deliver the report via their return value

as a "matter of taste and style" and for economy because of memory limitations in the early 1990s, when development of Tcl started

largely bloats command look-up table (but not any more an issue considering CPU power and main memory size available today)

Vivado vs. Tcl Syntax

Non-Project Mode

Non-Project Mode Design-Flow

Access inside or outside of Vivado GUI

Access via Vivado GUI **or** Tcl-Scripting

Other Tools

File System

HDL, …, Constraints

(Stored) Tcl-Scripts

Vivado GUI

Dependencies

Checkpoint Files

…

Vivado Internal Design Model

Strategies

Reports

Log-Files

Intermediate (Data-) Files

Bit-File

Session Protocol

(Journal-File)

Project Mode

Project-Mode Design Flow

Access **mainly** via Vivado GUI but …

… Tcl-Commands may be used too!

Developer

Synthesis

Optimize

Implementation

Write Bit-File

Analyse, Elaborate

Analyse, Elaborate

The Bit-File is usually considered as “final product” of a design, though during elaboration, before the target (chip) gets involved, there may be many design runs that end with **testing - not shown here**.

Besides serving as a protocol the journal may have an important role in Tcl-Scripting:

- it holds the Tcl-equivalent of all “actions” applied to the design model via the GUI;
- therefore it may serve as starting point for a Tcl-Script automating the tasks at hand.

Vivado Design Flows Overview

“Non-Project Mode” is sometimes confused with using the Vivado Tcl-Commands only.

“Project Mode” is sometimes confused with using the Vivado GUI only.

Non-Project Mode	Activity	Project Mode
Explicitely get HDL-Sources into Internal Model • `read_vhdl` • `read_verilog` • `read_xdc`	Preparing the Design Model	Manage file dependencies automatically via File Sets: • `create_project` • `add_files`
		• `current_fileset`
		• `import_…`
		• `open_project` • `current_project` • `close_project`

It is well possible to mix Vivado Tcl-Commands and using the GUI.

It is well possible to mix using the Vivado GUI and Tcl-Commands.

Elaborating the Design

There are too many command to list exhaustively, so just some typical examples are given:
- `get_cells` (select and query design objects)
- `set_property` (modify design objects)
- `report_…` (generate various kinds of reports and summaries)
- …

Run synthesis tools (various options allow to specify details) • `synth_design`	Synthesis	Select and run tools depending on changes made (various options, including some for load-sharing on compile servers): • `launch_runs` • `wait_on_run`
Run implementation tools (rich on options, especially for optimization) • `opt_design` • `place_design` • `phys_opt_design` • `route_design`	Implementation	

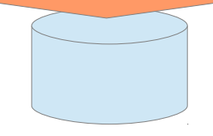
Use the Vivado GUI for any non-repetitive or mostly explorative elaboration and as a convenient way to view reports.

Use Vivado Tcl-Commands to automate anything that is systematic and repetitive.

Generating Target File

- `write_bitstream` (various options to select different formats and encoding)
- `write_verilog`
- `write_vhdl`
- `write_xdc`

Creating the ***Bitstream File*** is typically the overall goal when using Vivado.



Usually these phases overlap and are repeated, depending on reports and other kinds of checks and verifications, including simulation (not shown here). If files – especially Verilog or VHDL – are modified, they need to be read again explicitely.

Usually these phases overlap and are repeated, depending on reports and other kinds of checks and verifications. If files Verilog or VHDL Files are modified, Vivado knows about dependencies and will launch the tools as required in the right order.

Vivado Modes and Tcl Commands

Create ***new*** **Projects** with
- `create_project`

File Sets (holding file names and properties)
- list with `get_filesets`

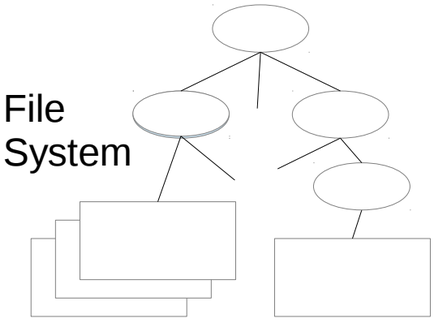
There is exactly <u>one</u> **(source) file set** per project
- get with `current_fileset`

There is exactly *<u>one</u>* **constraint file set** per project
- get with `current_fileset -constrset`

Only one **simulation file set** can be *<u>active</u>*
- get with `current_fileset -simset`
- set with `current_fileset -simset` *`other`*

Content of file set managed with
- **`create_fileset`**
- **`add_files`**
- `delete_fileset`
- `import_files`
- `move_files`
- `reimport_files`
- `remove_files`
- `reorder_files`
- `reset_project`
- `update_files`

(… not exhaustive …)



IP-Cores managed with
- `copy_ip`
- `create_ip`
- `get_ips`
- `read_ip`

(… not exhaustive …)

Multiple **Projects** can be ***open***, list with: `get_projects`

File Sets are much about
*"**Launching** the necessary **Tools** in the right Order"*
(list with: `report_compile_order`)

Convert from other **Project Formats**
- `import_synplify`
- `import_xise`
- `import_xst`

General Project Management**:**
- `archive_project`
- `refresh_design`
- `update_design`

Only one **Project** can be ***active***:
- get with `current_project`
- set with `current_project` *`other`*

(HDL) Sources
… `*.v` `*.vlog`

Constraints
… `*.xtc`

Simulation
… `*.?`

IP-Cores
… `*.?`

My Project

persistancy
via files
`*.xpr`

`open_project`
… `close_project`

Create ***New Synthesis Run*** with
- `create_run -flow {`*`Synthesis Tool`*`}`

Create ***New Implementation Run*** with
- `create_run -flow {`*`Implementation Tool`*`} -parent …`

Control Execution with
- `launch_run`
- `wait_for_runs`

Synthesis Run

Implementation Run

Get / set with
- `current_run -synthesis`
- `current_run -implementation`

Remove with
- `delete_runs`

From ***Implemented*** **Design**
- `write_bitstream`

Synthesis Defaults

Implementation Defaults

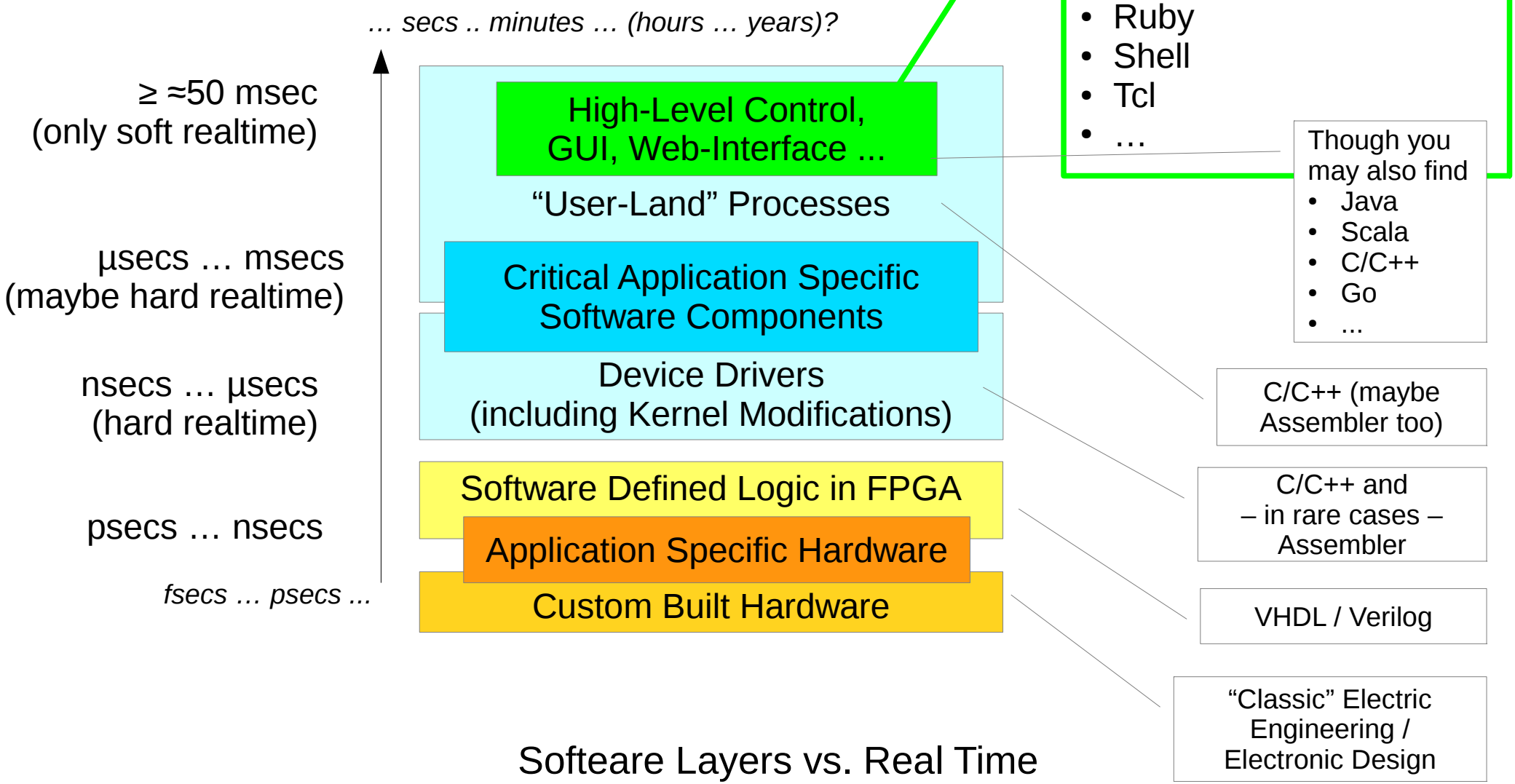
User Defined

Optimization ***Strategies***

Vivado "Project-Mode" Architecture

Concrete time values chosen for a rough orientation only (!)

- accord to technical achievments around 2010…2015
- still **widely** vary because of wide variations in technoloy



Softeare Layers vs. Real Time