Use Palladium to debug your design

Outline

Overview

- Two Ways to Run/Debug
  - IRUN
    - Command line or Simvision GUI
    - Limited support for interactive debugging of TB/DUT on Palladium
    - Better for regression runs
  - xeDebug
    - Full interactive debugging including hotswap
    - Smart command interpreter that looks at object of command to determine which engine to use
    - Access to both Simvision and Debussy GUI
    - Full set of XEL commands for emulator control

Overview: xeDebug

- xeDebug
  - debugging tool which supports both text and graphical mode

- xeDebug Text mode

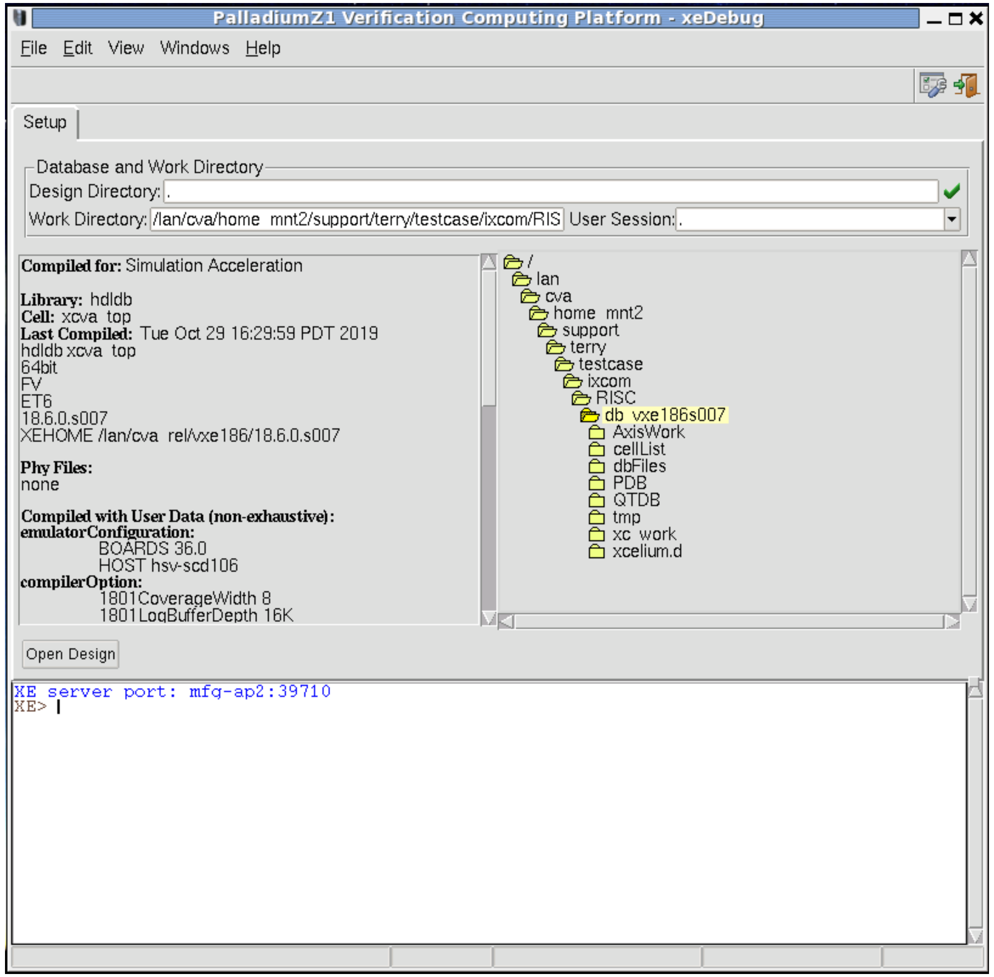
```
xeDebug
```

```
xeDebug - VXE, V18.6.0.s007

(c) 1991-2019 Cadence Design Systems, Inc. All rights reserved worldwide.
See files in <rootdir>/share/vxe/install/Copyrights
#source init.tcl
XE> debug .
XE> host $env(XE_HOST)
INFO (legacy-51751): Setting emulator to hsv-scd106 from host command.  DBEngine host is mfg-ap2.
INFO (legacy-51832): An InfiniBand connection was verified from host mfg-ap2 to hsv-scd106.
WARNING (legacy-52246): xeDebug version of '18.6.0.s007' is not supported with configmgr version of '18.5.0.195' running on 'hsv-scd106'.
xrun -R -disxedebug -64bit -nolog +ignoreNCVerCheck -tcl -input /lan/cva_rel/vxe186/18.6.0.s007/tools.lnx86/etc/qel/ncrun.qel
INFO (legacy-51578): Opened xeConnect port 49891 on host mfg-ap2 (158.140.43.247) for xeDebug process 14565.
INFO (legacy-0): 1096 cells have been restored
INFO (legacy-0): 96073 smTrNets have been restored
INFO (legacy-0): 68306 smTrPrims have been restored
DBEngine_rpcbind - VXE, V18.6.0.s007

INFO (legacy-47055): Running in design directory: '/lan/cva/home_mnt2/support/terry/testcase/ixcom/RISC/db_vxe186s007'
xrun(64): 18.09-s017: (c) Copyright 1995-2019 Cadence Design Systems, Inc.
TOOL:   xrun(64)        18.09-s017: Started on Oct 29, 2019 at 16:35:59 PDT
xmsim(64): 18.09-s017: (c) Copyright 1995-2019 Cadence Design Systems, Inc.
Loading snapshot xc_ncwork.cpu_test:module .................... Done
Log started on host: mfg-ap2 at: Tue Oct 29 16:36:04 2019

libxcrt - VXE, V18.6.0.s007
xcelium> source /lan/cva_rel/vxe186/18.6.0.s007/tools.lnx86/etc/ixcom/xc.tcl
xcelium> source [file join $env(QTHOME) etc qel ncloadrun.qel]
Connecting to dc:mfg-ap2:44793...

XEsim>
```


Overview: xeDebug

- xeDebug Graphical Mode

```
xeDebug -gui
```



Overview: Debug Capabilities

- Full Vision
  - View waveforms of any signal in the design without need to specify signals to be viewed before run
- Waveform Trace Depth
  - Up to 1M cycles with Full Vision mode, 42M cycles using Dynamic Probe mode
  - Infinitrace feature allows unlimited trace depth and random access
- Dynamic Events
  - can trigger any signal or set of signals without recompile
- State Description Language (SDL)
  - easy-to-use language for controlling triggers

cadence®

Overview: Debug Capabilities - Monitors

- Monitors
  - Create/use monitors to get protocol or bus listings
  - Especially SW engineers prefer listings over waveforms
  - Easier and more complete overview of DUT activities
- Multiple options
  - SDL display based
    - Palladium supports parallel SDL instances / parallel triggers.
    - Cadence has a few monitors for standard interfaces
      e.g. related to PCIe SpeedBridge
  - Assertion based (ixcom only)
    - Assertions can be used in a similar way as SDL, either monitoring some activities, or for triggering error conditions (and off course to collects metrics)
    - Cadence offers a set of ABVIP for standard interfaces
  - Verilog based (ixcom only)
    - You can bind Verilog ($display) based monitors just like any other side top
  - DRTL
    - Dynamic RTL, new Z1 feature, combination of SDL and Verilog:
    - Verilog syntax
    - Compiled and bound to the design at runtime, just like SDL

Outline

cadence

Run Modes

- **Static Target testBench (STB) Mode**
  - FCLK and tracing stops automatically when trigger occurs, or after specified time or number of cycles
  - Normally, design clocks are generated within Palladium, derived from FCLK
  - Use this mode for targetless emulation, or with a target which can tolerate emulator stopping
- **Logic Analyzer (LA) Mode**
  - When trigger occurs, waveform tracing stops but FCLK keeps running
  - Design clocks usually generated within Palladium, but can also come from target
  - Use this mode if target cannot tolerate the emulator stopping.
- **Vector Debug (VD) Mode**
  - Design runs from a vector file
  - One FCLK per vector
  - Use this mode for specialized purposes. For example, validating silicon test vectors

Vision Modes: Overview

- **Full Vision (FV)**
  - This mode gives maximum flexibility in debugging
    - You can iteratively view more and more signals as debug progresses, without needing to re-run the test
    - Disadvantage: uploading waveforms can take a long time, if trace depth and number of signals are both large
- **Dynamic Probes (DYNP)**
  - Specialized Mode with following advantages
    - Very fast waveform upload
    - Larger trace depth if desired number of signals is small
    - In some designs, uses less emulator capacity
  - Disadvantage: You must decide before running, what signals to probe
- **Specify the Vision Mode when you compile:**

  compilerOption –add {visionMode FV} ; this is the default

  compilerOption –add {visionMode DYNP}

Vision Mode Differences

	DYNP	FV
Max trace depth (Z1)	128M with 512 probes/ domain	Usually 4M (can exceed 4M if emulator utilization is very low)
Unlimited number of probes	N	Y
Trade off between #probes and depth	Y	N
Upload speed	Fast	Can be slow – specify multiple workstations to speed it up
Multiple workstations for upload supported	N	Y
Can specify probes after trace capture	N	Y
Conditional Acquire support	Y	Y, but with 32-sample granularity
Additional probes during Offline Debug	N	Y

Offline Debug

- Purpose is to allow iteration of “select more signals”, “view their waveforms” over and over while debugging, without occupying the emulator hardware during this debug.
- During the online session, upload only enough signals to confirm that the trigger condition is what you intended. Then, switch to offline mode.
- As the source of waveform data, offline debug uses a directory (trace.phy) created by upload operation(s) during an online session.
- Fixed time window(s) (determined by trace.phy)
  – But you can select a sub-window using database -tracewindow
- Here is one way to invoke offline debug:
  – xeDebug –offline trace.phy -gui
- In IXCOM flow, offline debug only understands the DUT
  – No testbench probes
- xeDebug –offline only works with visionMode FV
  - HINT wrt DYNP: You can also view DYNP-generated waveforms after Palladium is released, just not via xeDebug –offline and not by adding more probes than you had uploaded at runtime

Trace Depth

- Maximum Trace Depth
  - Determined at compile time.
    - In FV, the maximum trace depth is normally determined by the compiler.
    - In DYNP, user can set max trace depth using compilerOption –add {traceDepth <n>}. Specifying large trace depth reduces the number of signals that can be probed.
  - Reported in xe.msg near end of compile

    `INFO (db2util-1108): Physical data capture trace depth is 4418688.`
- Actual Trace Depth
  - Set during run
  - xeset traceMemSize <n> [<units>]
  - Smaller trace depth means faster upload
- HINT
  - In ixcom flow, due to possible behavioral cycles, filling up the trace buffer without advancing time, it usually is impossible for the user or tool to predict the available maximum trace depth

Infinitrace

- Infinitrace feature allows
  - Unlimited trace depth
  - Run part of a test over and over, with different triggers

- Infinitrace limitation
  - If using external target, the target must tolerate the emulator stopping periodically – i.e. “dynamic targets” are not supported with infiniTrace

Outline

Overview

Modes of Operation

Run-time Logistics

Getting Waveforms

Viewing Waveforms in SimVision

SDL Language to Control Triggering

DRTL

Infinitrace

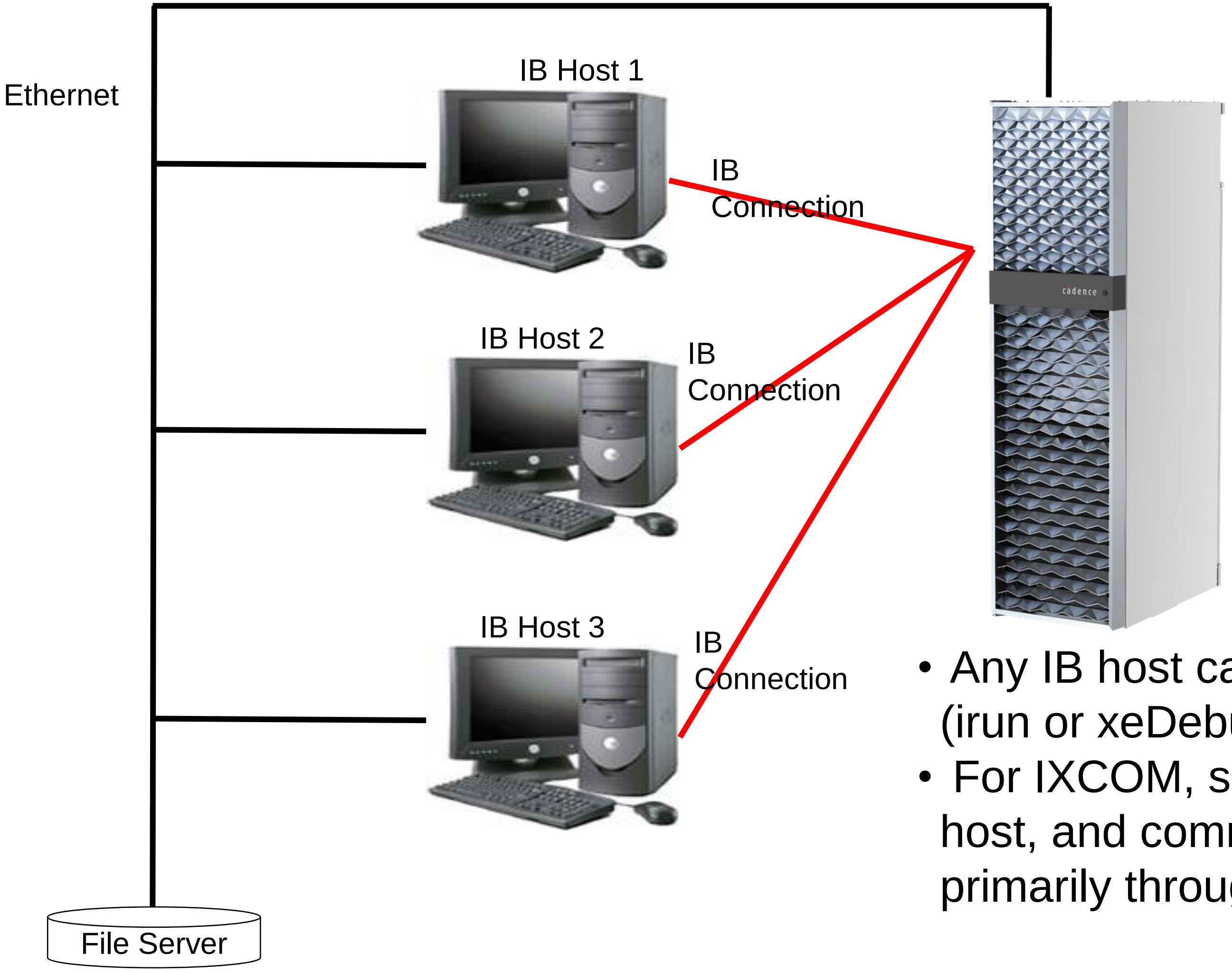
QEL commands for Debug

xeDebug GUI

Regression Mode

Log Files

Where to run xeDebug?



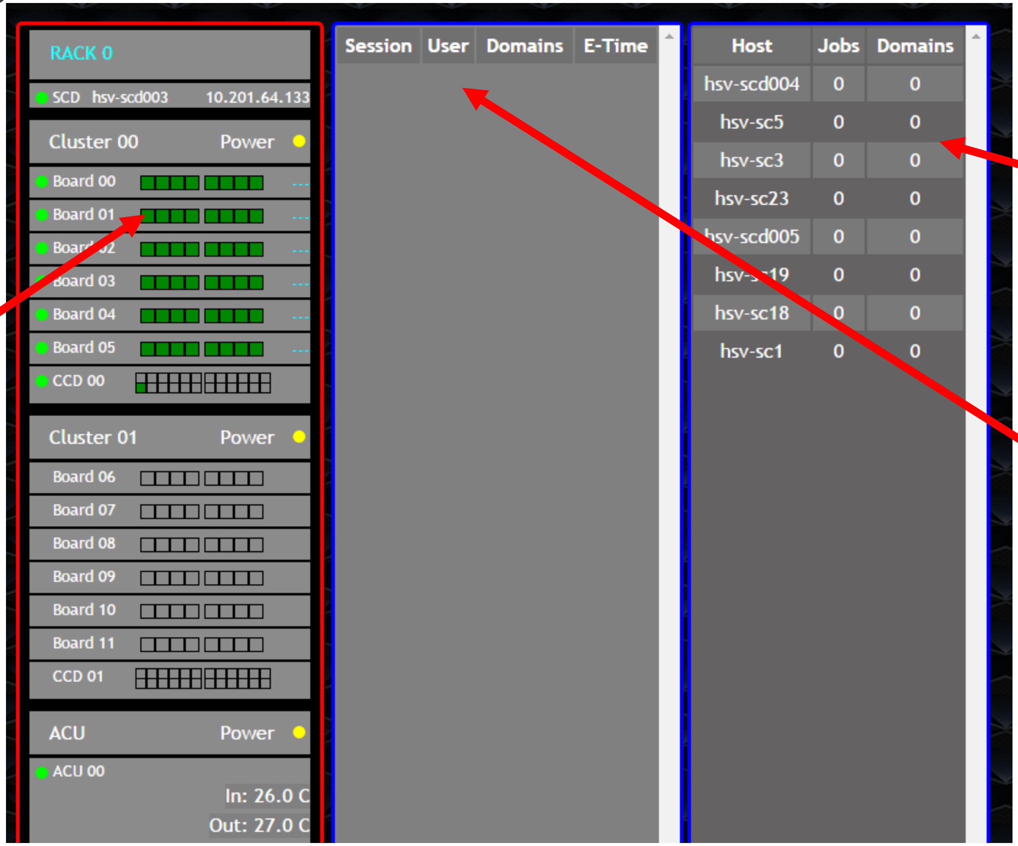
- Any IB host can run a Palladium session (irun or xeDebug)
- For IXCOM, simulator will run on the IB host, and communicates with DUT primarily through the IB connection.

Checking Emulator Availability

- Give the command test_server (at unix prompt) on the IB host to check availability of the emulator
  - test_server –listEmu: shows all emulators connected to the IB host
  - test_server –listIb: shows all IB hosts connected to the network
- Command test_server by itself shows available boards and domains on an emulator
  - test_server <top_cell> -location: identifies best location for downloading <top_cell>

Web-Based Status Display

- **Found at scd_host_name.company_url.com**
- **Shows current status of racks, clusters, boards, and power supply within the emulator**
- **Displays IB hosts connected to the emulator**



status of racks, clusters, boards

ib_hosts connected to the emulator

users logged in to scd_host

Invoking xeDebug

- To run and debug designs, use the xeDebug program
  - Use the –gui option if you want GUI
    - Default is to run in command line mode
- Per default waveforms are stored in shm enabling SimVision for waveform viewing
  - Use –fsdb option if you'll use Synopsys debug tools (Verdi, etc)
- See docs for other options

Outline

Overview

Modes of Operation

Run-time Logistics

Getting Waveforms

Viewing Waveforms in SimVision

SDL Language to Control Triggering

DRTL

Infinitrace

QEL commands for Debug

xeDebug GUI

Regression Mode

Log Files

Getting waveforms

- Getting waveforms involves three steps:
  - Probe signals – specify the signals whose waveforms you want
    - XEL command: probe –create (many options)
    - Probe button in GUI –probe signals currently selected in a browser
    - Several other ways in xeDebug Probe tab
    - Fast probe: probe –fast enables you to bypass fullvision computation for faster waveform generation
  - Upload waveforms into a waveform database (trace.fsdb or trace.shm)
    - XEL command: database –upload
    - Upload button in GUI
  - Display waveforms in waveform viewer
    - Display button in GUI
    - Can use SimVision to view waveforms
- Some GUI buttons combine two or more of these steps

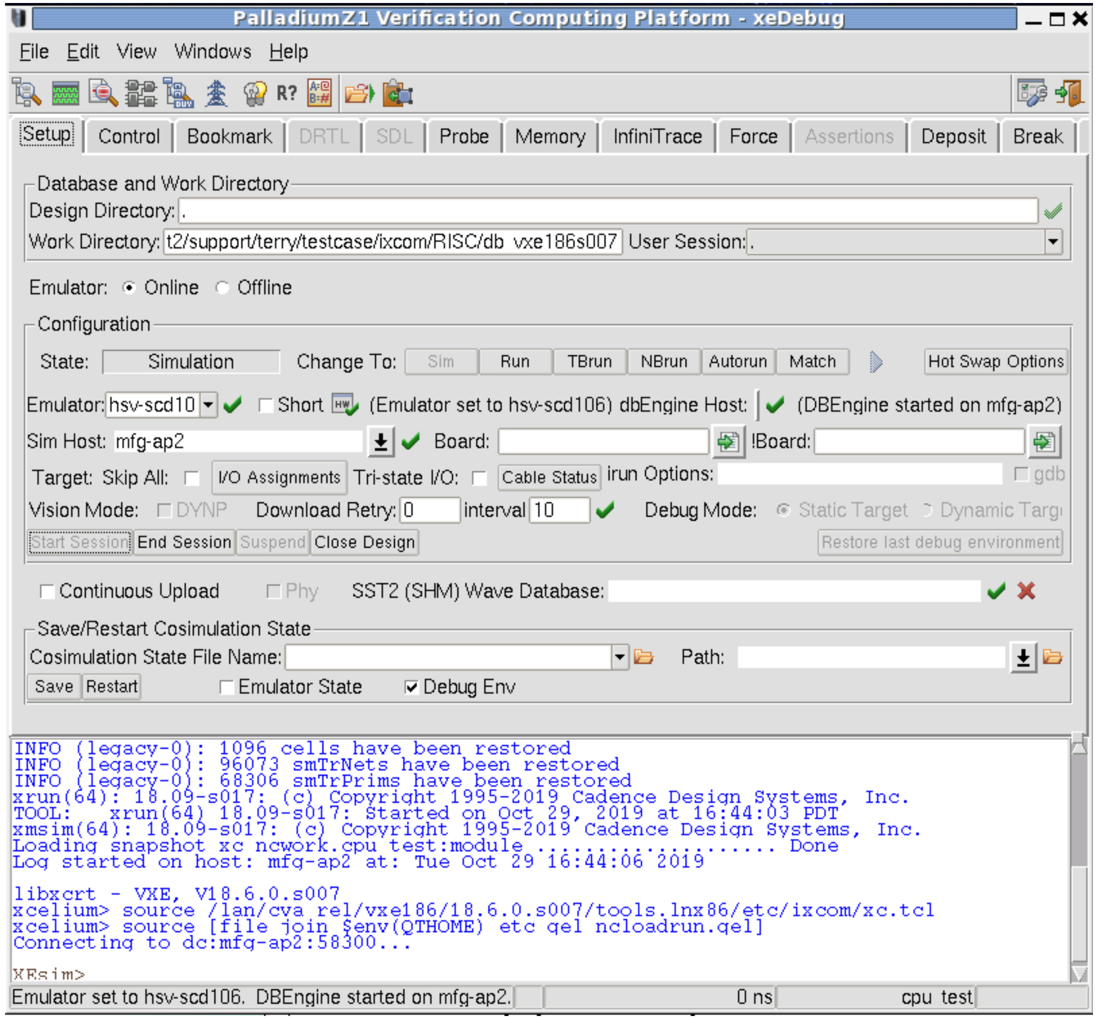
Waveforms: Using Commands

- use probe command
- database must be opened (no default at startup)

```
database -open databasename
probe -create cpu_test.RISC.control -all -depth 0
run 100ns
database -upload
```

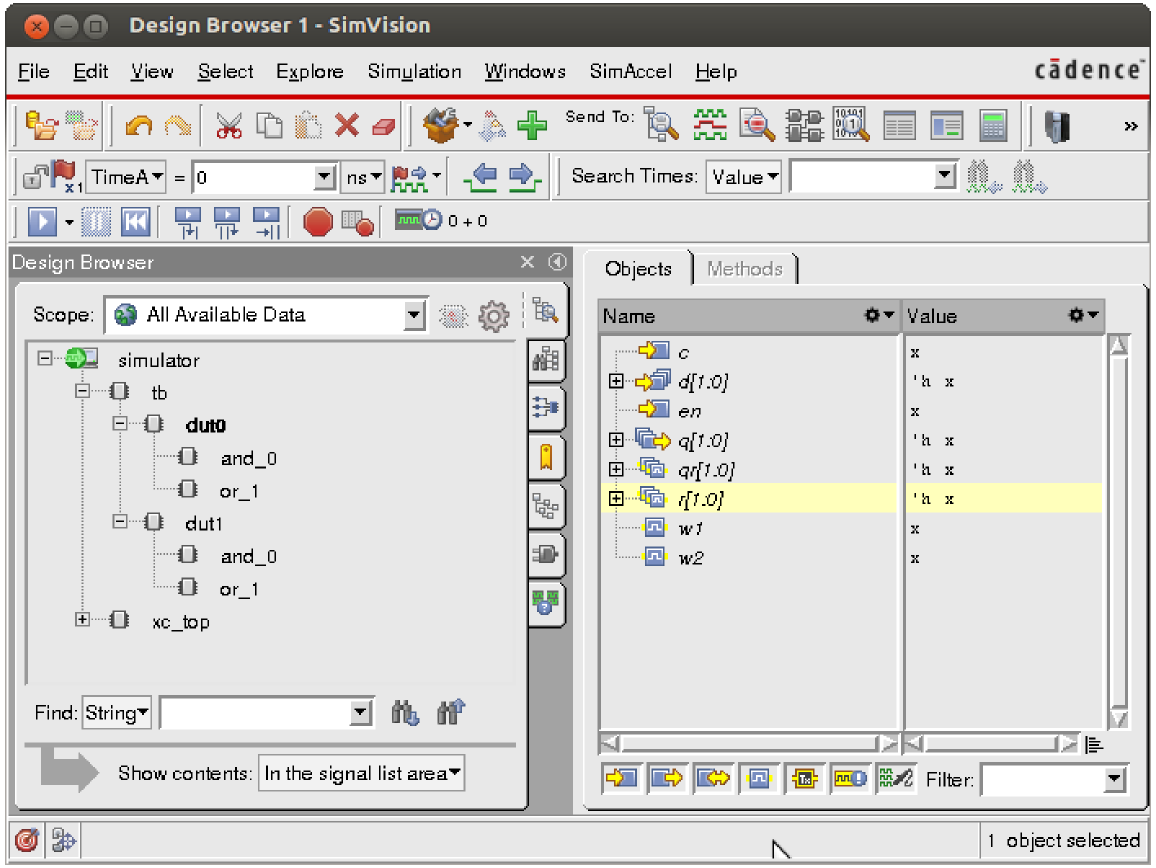

Waveforms: GUI Mode

- STEP #1: Launch xeDebug GUI Tool
  - This will create the xeDebug main window
  - SimVision window may also appear, if not use SimVision buttons in upper left corner of xeDebug or simply proceed (starting to view waveforms will open SimVision automatically anyway)



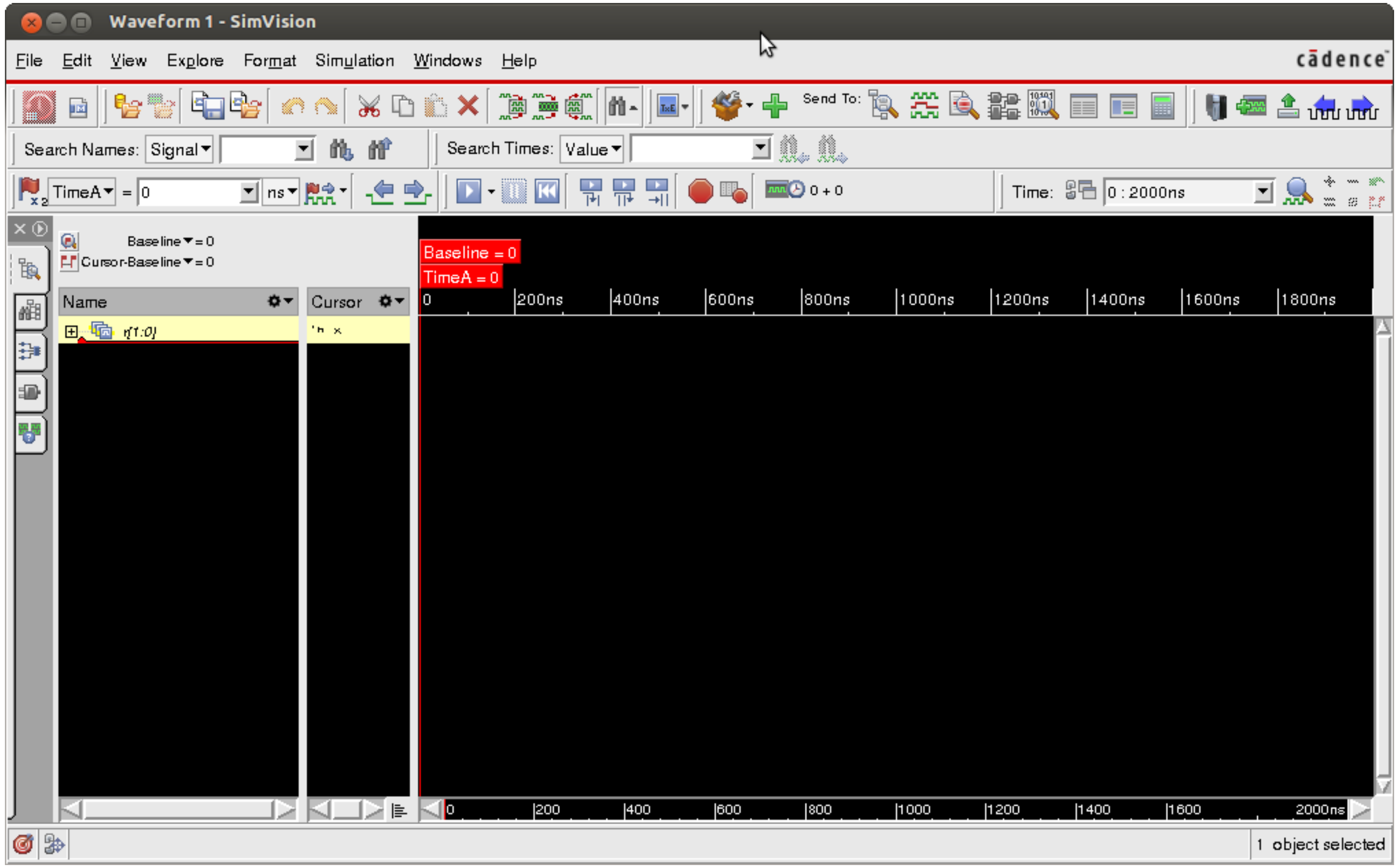
Waveforms: GUI Mode

- STEP #2: Set Probes by “send to waveform”
  - browse the design hierarchy and select signals
  - right click and select “send to waveform”



Waveforms: GUI Mode

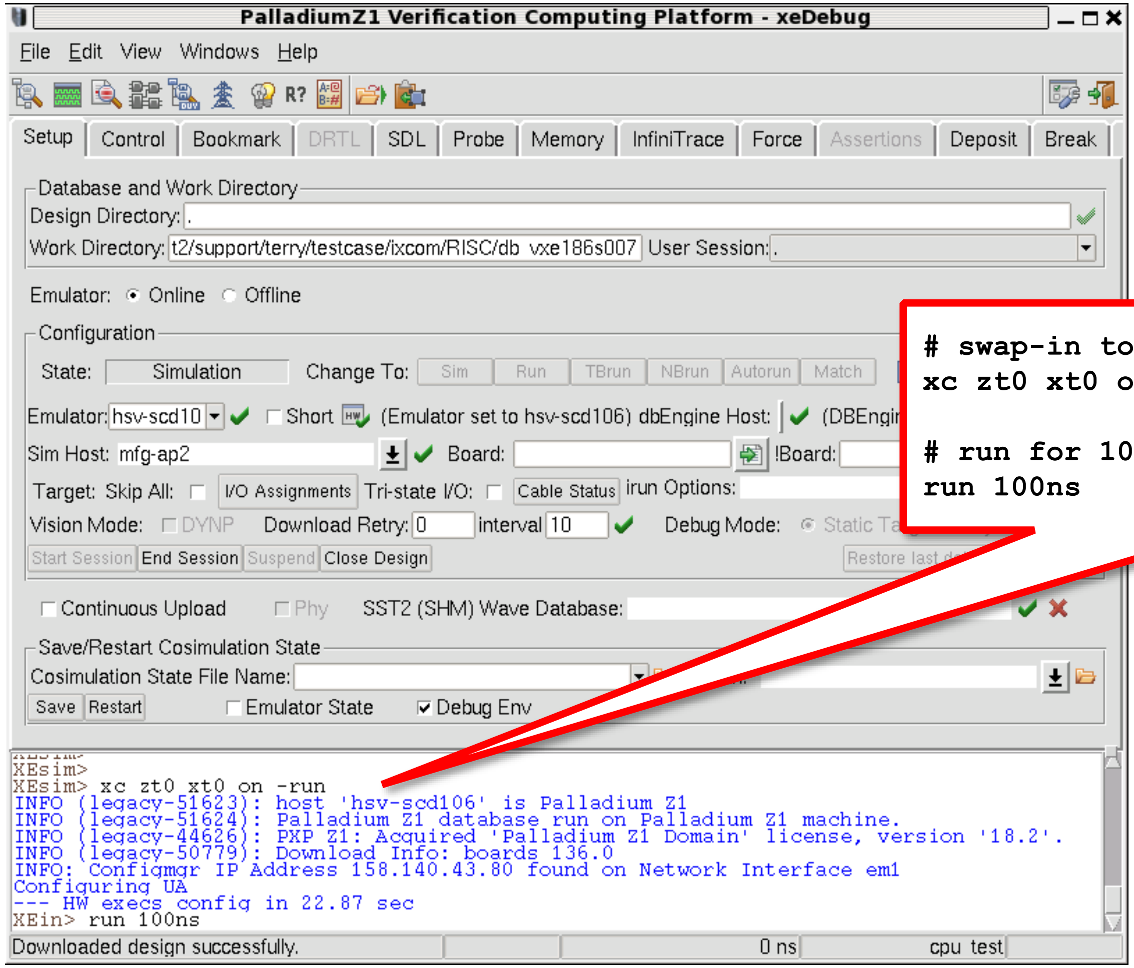
- STEP #3: Set Probes by “send to waveform”
  - simvision window will show the probed signal



cadence

Waveforms: GUI Mode

- STEP #4: Swap in to HW and run
  - use shell for flexible control

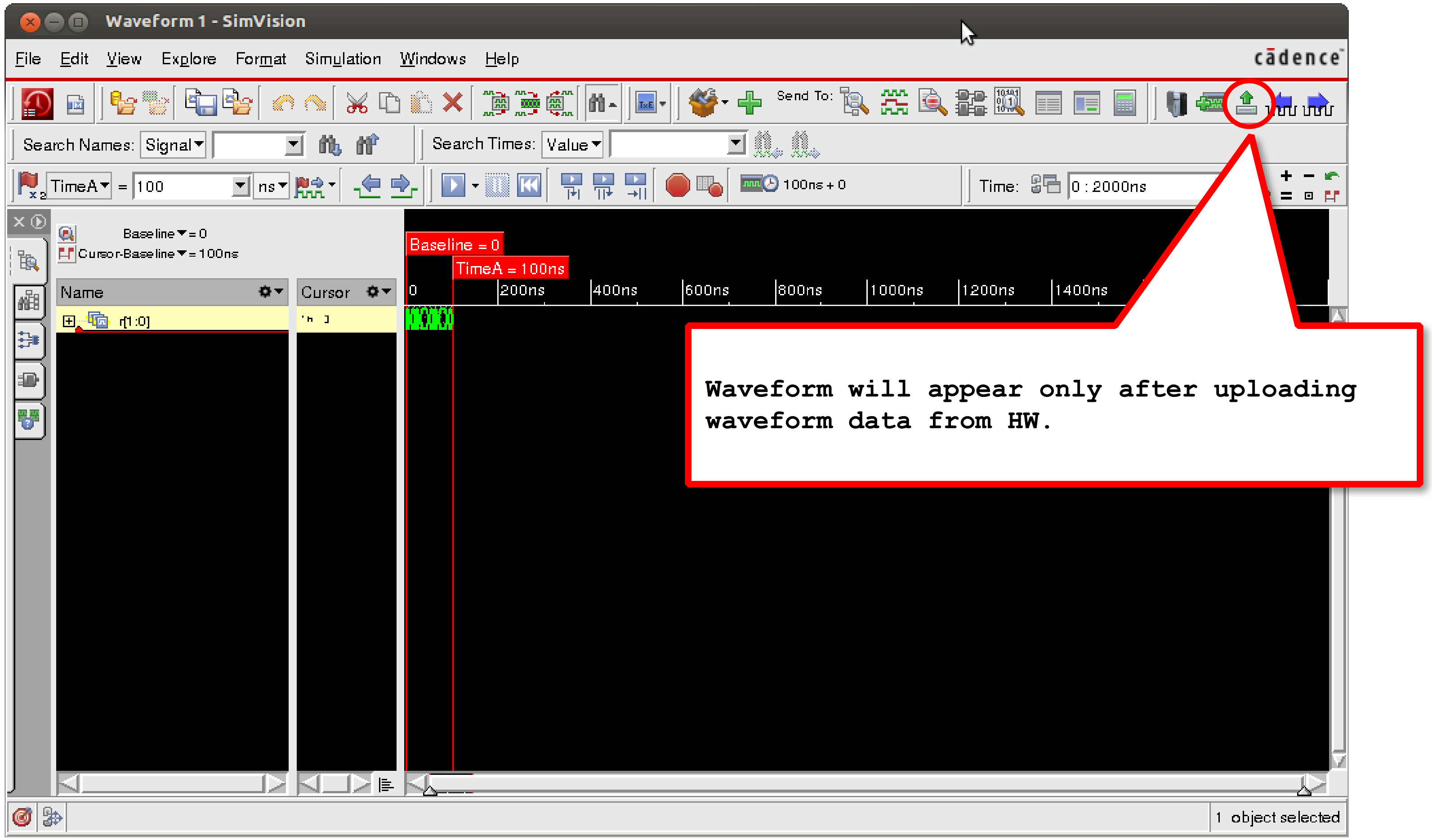



```
# swap-in to HW
xc zt0 xt0 on -run

# run for 100ns
run 100ns
```


cādence®

Waveforms: GUI Mode

- STEP #5: Use “Upload” button in simvision to show HW waveform



“Upload” operation in Full Vision

- **In full vision, the upload operation itself involves three steps:**

1. Upload data from Emulator.
   - This step uploads enough data to generate waveforms for all signals in the design. The data is stored in trace.phy. The actual waveforms can be generated later, without needing the emulator. (That’s called offline debug).
   - You can perform this step by itself, using database -prepareOffline
2. Reconstruct register data for entire design.
   - This step constructs waveform data for all registers in the design. The data is in trace.phy, but is not yet in a form that can be displayed.
3. Generate waveform data for user-requested signals
   - This step generates the waveform data (in trace.shm or trace.fsdb) to be displayed in the waveform viewer. If you specify millions of signals, this step can take a long time.
   - The first database –upload command after a run does steps 1 and 2 for the whole design, and does step 3 for all the requested signals. Subsequent database –upload commands do only step 3, for newly requested signals.

“Continuous Upload” operation

- This feature automatically detects DCC overflow and pauses the run to upload the waveform before the DCC memories overflow.
- A continuous upload is done automatically when you:
  – execute the run command later;
  – issue the stop command if running in the non-blocking mode.
- Use the database -close command to close the database and stop the continuousupload.
- Sample command:
  – database -open -continuousupload <database_name>

“Multi host DCC upload” operation

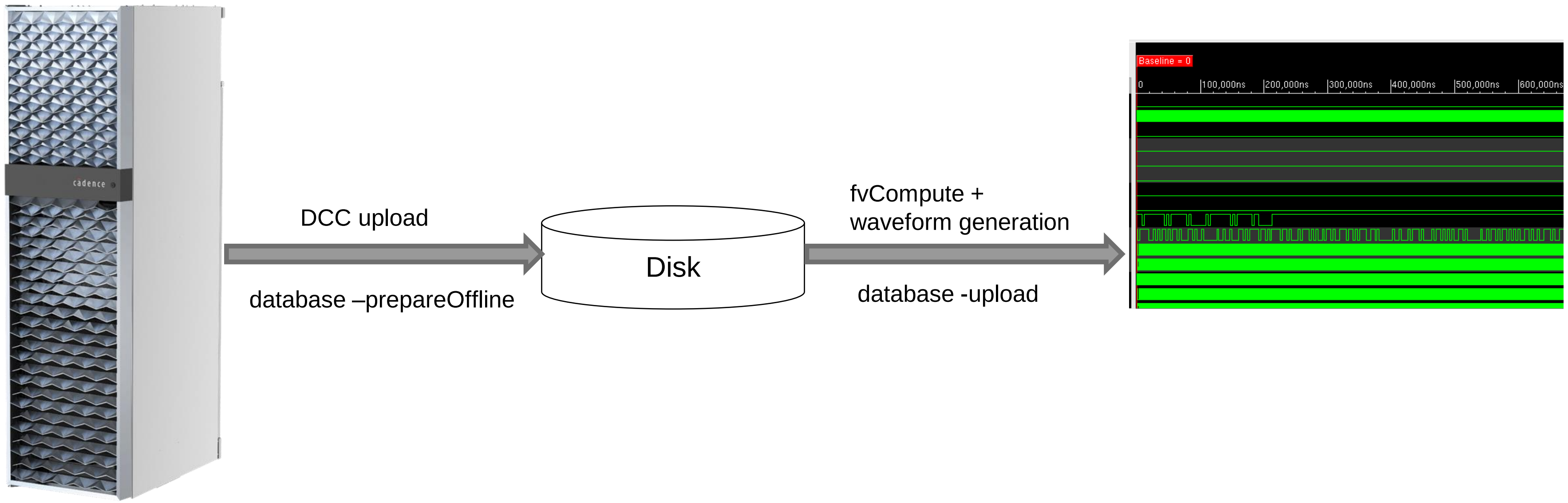
- The Data Capture Cards (DCCs) are used to capture design data during a run. By default, the DBEngine host reads all DCC raw data from all used Logic Drawers, processes the data, and writes the data to the disk, into the PHY1 file.
- The multi host DCC upload feature distributes DCC uploads to multiple hosts and thereby improves the performance over the single host DCC upload architecture by speeding up the PHY1 generation for big designs.
- DCC upload job distribution is evenly spread across the given hosts.
- To enable this feature:
  - setenv CDN_MT_IB_HOST host1:host2:…:hostN
    - Use rsh/ssh to launch on N IB hosts
  - setenv CDN_MT_IB_HOST cmd:N:script_name.sh
    - Use script_name.sh to spawn using LSF

Optimizing Full Vision Performance

- With Full Vision, you specify a set of signals to upload
- You can specify all signals in the design
  - But if you do that, upload will be slow
- It's usually better to upload waveforms for only the instances likely to be needed to debug a given problem
  - If, during debug, you realize more signals are needed, specify more instances and upload their waveforms. You can do this as many times as needed, without re-running the test.
- Upload time is proportional to trace depth
  - Upload huge trace depth only if you need it

Runtime Partial FullVision
Introduction

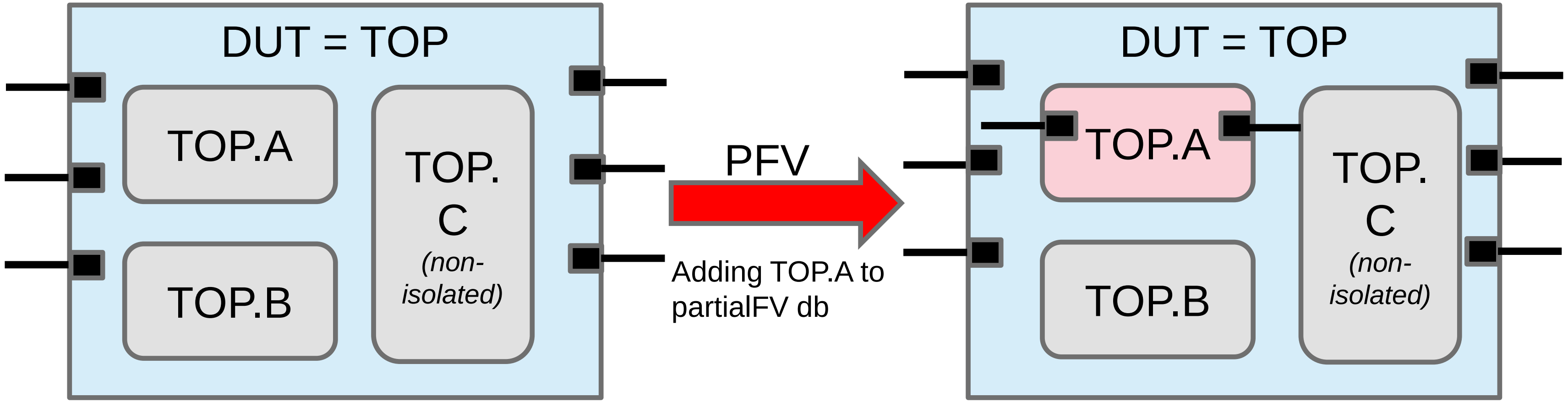
- The FullVision operation consists of 3 steps
  1. DCC upload
  2. FullVision compute
  3. On Demand waveform generation



Runtime Partial FullVision
Introduction

- Time consumption
  - The second step (fvCompute) is most time consuming among the three
  - As design size grows, the time consumption for FvCompute can increase significantly

- Runtime Partial FullVision
  - Allows user to add isolation in DUT on the emulator
  - Physical probes then can be added to boundary nets of the said isolated instance
  - The resultant waveform generation is thus faster for the isolated part



Runtime Partial FullVision
3 steps to use partialFV

1) Adding Isolation

- Add isolation of instance(s) in online session before running emulation

  database –partialfv –add <instance_name>
- Verify that the instance has been added to isolation list

  database –partialfv –list

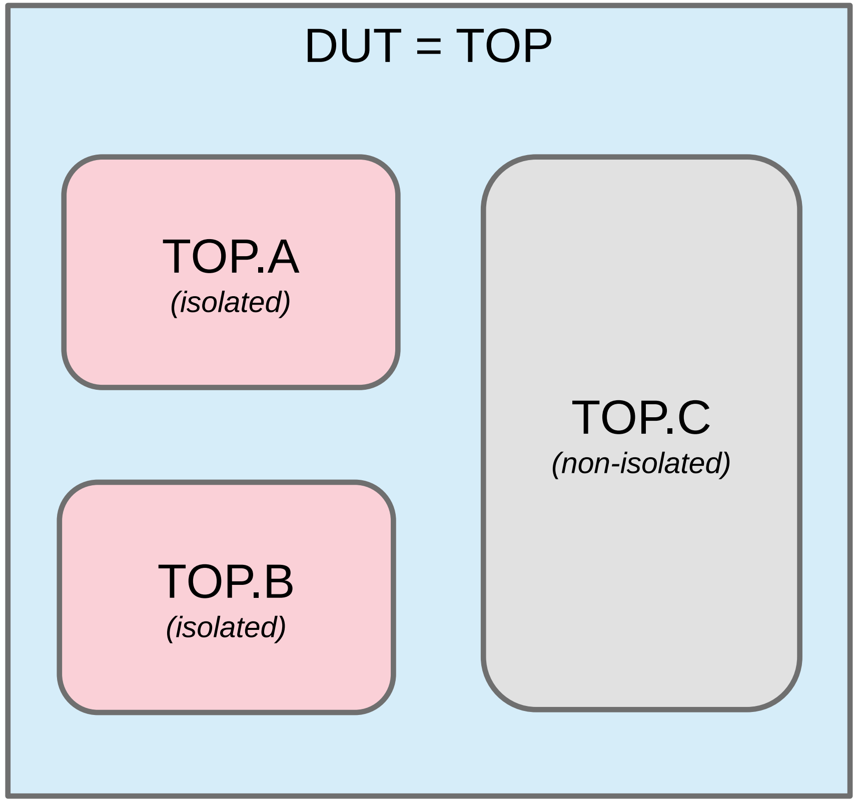
2) Adding Probes

- User can then add probes for the instance added to isolation
- SW will know which partition to compute based on the set of probes. Smaller the isolation, faster the FV compute

3) Generating Waveform

- At the end of the run, use *database –upload* to generate waveform
- The isolation information is stored in .phy file. In Offline session, user can open .phy file, add probe and upload waveform

Runtime Partial FullVision
Working



fvCompute stand alone command:

fvCompute <file>.phy

The above command will compute TOP.A and TOP.B

- Cases without partialFV benefits
  - Probing A, B and C
    - Same as FV because we are probing entire design – inside and outside isolation both
- Cases with partialFV benefits
  - Probing A & B
    - Faster than FV since TOP.A and TOP.B are added to isolation. Only TWO partitions are calculated
  - Probing A or B
    - Faster than above case because now only ONE partition will be calculated – A OR B
  - Probing A or B and C
    - Faster because only ONE out of TOP.A and TOP.B will be computed along with TOP.C
  - Probing only C
    - FASTER! with partialFV
    - Probing only C is faster because partialFV will not have to compute TOP.A and TOP.B since they are isolated. It only has to compute TOP.C

Runtime Partial FulllVision
Guidelines and Limitations

- Guidelines
  - Partial FV doesn't affect or improve DCC upload time
  - Boundary signals of instance added to partialFV are added as physical probes
  - The number of instances we can add to isolation are limited by the available physical probes. We can recompile using the following compiler option

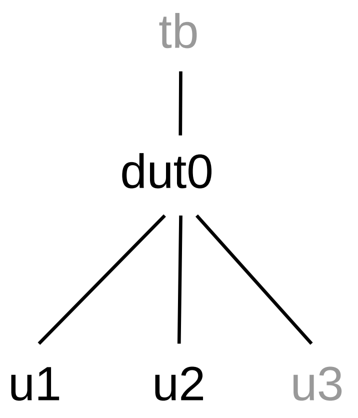
    compilerOption –add {numExtraCapturedNetsPerDomain <num>}

  - Above command will reserve <num> extra probes at compile time
  - The performance improvement depends on the size of instance we isolate and probe.

- Limitations
  - PartialFV isolations are not saved with *save* command. Therefore on *restore*, the isolations will not be retained

probe –depth all in IXCOM flow

- Example:



Suppose that

- U3 is behavioral module
- When swap into hardware, tb and u3 stay in simulator

Recommended probing just for a small design where you wish to probe all signals:

- probe -depth all tb
  - tb module is probed in simulator
  - probe is able to cross SW/HW boundary, dut is probed as well
  - dut probes are “smart probes”: automatically probed in emulator when running in hardware

Probing guidelines – IXCOM flow

- Probes in the simulator are continually uploaded while running
  - This can slow down the simulation
- Hardware probes are uploaded only when you give the database –upload command
  - Only database –upload time, not run time, depends on the number of probes
  - If you swap from hardware back to software, you must first do database –upload, if you wish to see waveforms from the hardware

Partial swfvDB / Parallel FSDB
Overview

- What is swfvDB?
  - swfvdb, a.k.a. software fullvision database, is the design data read in by xeDebug along with the PHY file in order to generate a waveform or SAIF file.

- Partial swfvDB allows the user to create multiple swfvDB partitions and work on them in parallel instead of the traditional one big swfvDB
  - The swfvDB partitions are built on top of PPC partitions – hence PPC is a pre-requisite

- Breaking down the swfvDB into partitions results in 2 key advantages:
  - Smaller memory footprint for fvCompute hosts and the local host
  - Multiple XeDebug sessions in parallel = faster throughput

cadence®

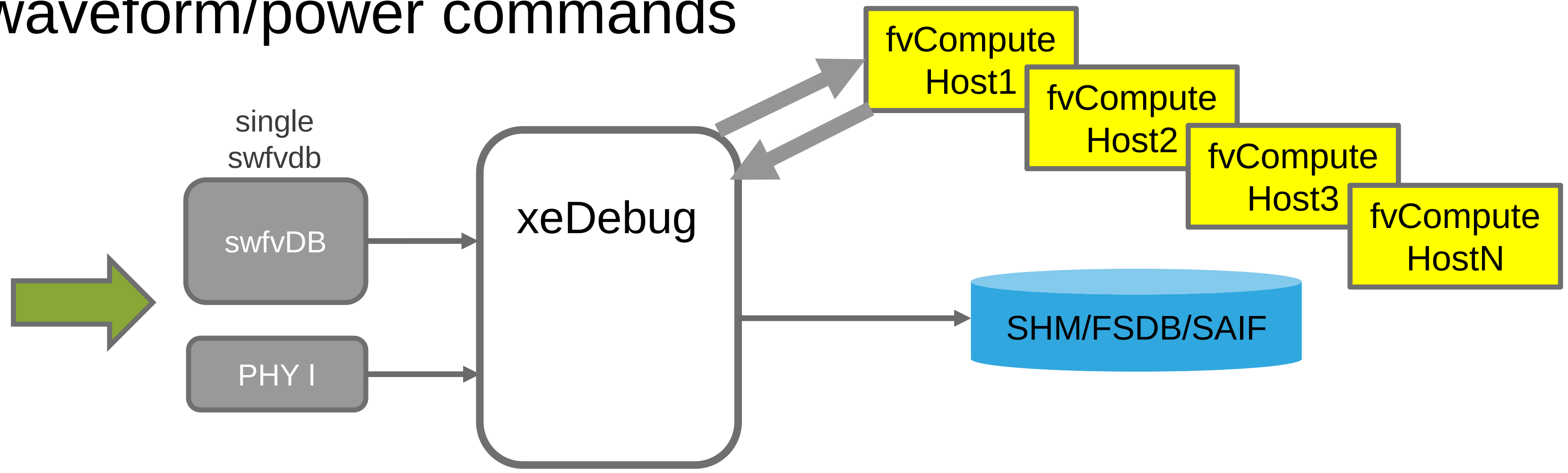
Partial swfvDB / Parallel FSDB
Using a PHY file

- Having generated a PHY1 file from the run, user can use the PHY file to:
  - Generate an SHM/FSDB waveform
  - Generate a SAIF file for power analysis

- In all cases, user needs to open the design in xeDebug offline mode and read in the PHY file followed by waveform/power commands

Generic commands

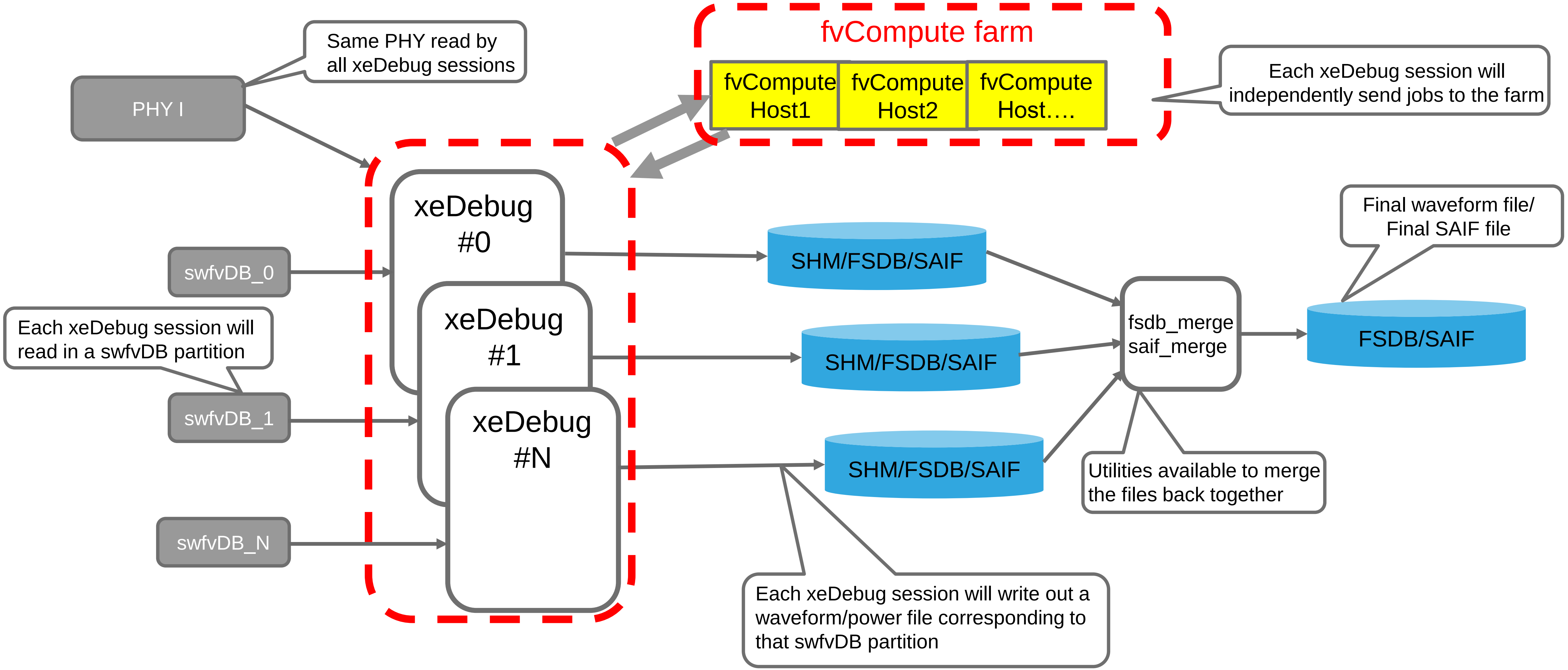
>debug .
>host –offline <phy_file>
>dpa/probe commands
>dpa/database –upload
>exit



cadence

Partial swfvDB / Parallel FSDB

Partial swfvDB flow



Partial swfvDB / Parallel FSDB
Enabling the feature

- To enable partial swfvDB creation, user needs to add the following option

```
compilerOption -add {fvPartition on}
```

  - This will create swfvDB partitions (swfvDB_0, swfvDB_1, swfvDB_2, …), in addition to creating the full swfvDB just as with the regular flow
  - These swfvDB partitions can be found in the design directory under dbFiles/
  - swfvDB: original full swfvDB
    - Since the full swfvDB is present, user always has the option to generate waveform/SAIF with traditional flow

Partial swfvDB
SAIF flow

Sample script:

```
debug . -partition 0
host -offline ../traceMockTop_32k.phy
dpa -addinst -depth 0 .
dpa -outfile -saif partition0 -instance .
dpa -upload
```

<name>_0.saif.gz

- A session directory is created with name partition_0.session
- A SAIF file is created with name partition0_0.saif.gz

```
debug . -partition 1
host -offline ../traceMockTop_32k.phy
dpa -addinst -depth 0 .
dpa -outfile -saif partition1 -instance .
dpa -upload

exit
```

<name>_1.saif.gz

- A session directory is created with name partition_1.session
- A SAIF file is created with name partition1_1.saif.gz

Contents of partition_0.session directory:

```
AxisWork  dbFiles              partition0_0.saif_segment.log  QTDB  xcelium.d  xeDebug.key  xe.msg
cellList  partition0_0.saif.gz  PDB                           tmp   xc_work    xeDebug.log  xmls.log
```

- 8 swfvDB partitions = 8 xeDebug sessions = 8 partition_<n>.session directories
- Each session directory with a corresponding SAIF file
- These SAIF files can be merged together using saif_merge utility

cadence

Partial swfvDB
Parallel FSDB flow

Sample script:

```
debug . -partition 1
host -offline ../traceMockTop_32k.phy
database partition1_mtsfdb -batch -numFile 10
probe . -depth all
database -upload
exit
```

- A session directory is created with name partition_1.session
- An FSDB file is created with name partition1_mtsfdb.fsdb
- A virtual file is also created by default for the partition:partition1_mtsfdb.fsdb.vf

Contents of partition_1.session directory:

```
AxisWork  dbFiles                partition1_mtsfdb.fsdb.vf  QTDB  traceMockTop_32k.phy  xc_work       xeDebug.log  xmls.log
cellList  partition1_mtsfdb.fsdb  PDB                        tmp   xcelium.d             xeDebug.key   xe.msg
```

- Number of waveform files = Number of waveform threads * Number of swfvDB partitions
  - For current example, 10 FSDB threads * 8 partitions = 80 FSDB files
- A virtual file for each partition is created in the session directory
  - For current example, 8 partitions = 8 virtual files
- To stitch the waveform back together, user needs to create a master virtual file that includes the 8 virtual files generated by default

Partial swfvDB / Parallel FSDB
Key takeaways

- PPC is required in order to leverage this feature
  – autoPart and advancedPart are supported
- Writing a PHY2
  – PHY2 file will not be generated with this flow
  – This implies that a user can NOT write to the PHY file to convert it from PHY1 -> PHY2.
- Probing specific instances
  – User doesn’t know which swfvDB has the data of the instance he’s interested in. So, a user must iteratively probe through all the partitions to ensure he gets the full waveform/dpa data.
- Less than 256G memory footprint guaranteed on fvCompute hosts
  – Since swfvDB partitions are built on top of PPC partitions (given that PPC partitions are smaller 1 cluster), it is guaranteed that no fvCompute job will take more than 256G of memory
- Serial vs Parallel xeDebug sessions
  – Use serial: When the priority is to use less resources (fvCompute hosts with smaller memory)
  – Use parallel: When the priority is a faster throughput (fvCompute hosts are big enough to take up multiple jobs)

Generating Partitioned Waveforms Using genWave Utility
Background

- In PPC flow, as mentioned previously, user has the option to generate partial swfvDB and then use the parallel FSDB/SHM flow to speed up waveform generation. It requires the following steps:
  - Launching offline xeDebug sessions for various partitions
  - Generating multiple waveform files
  - Executing makeup session
  - Merging all waveforms to create a single waveform file
- User can avoid executing the above steps and generate a unified waveform file from different swfvDB partitions by using the genWave utility. This utility internally executes all the above mentioned steps. The genWave utility supports both SHM and FSDB waveforms.

Generating Partitioned Waveforms Using genWave Utility
Usage

- By default, the genWave utility is used to automate the partitioned waveform generation for PPC flow. This is called *Double-Parallel Waveform Generation* mode. To enable this mode:
  - Compile the design with PPC
  - Partition the FullVision database using compilerOption -add {fvPartition ON}
- genWave can also be used for waveform generation in non-PPC flow. This is called Single-*Parallel Waveform Generation* mode.
  - To enable this mode, specify -partition NONE for the genWave command
- Example Commands:
  - Double-Parallel mode: genWave -phyFile trace.phy -partition ALL -probeCmdFile probe.xel
  - Single-Parallel mode: genWave -phyFile trace.phy -partition NONE

cadence

Generating Partitioned Waveforms Using genWave Utility
Limitations

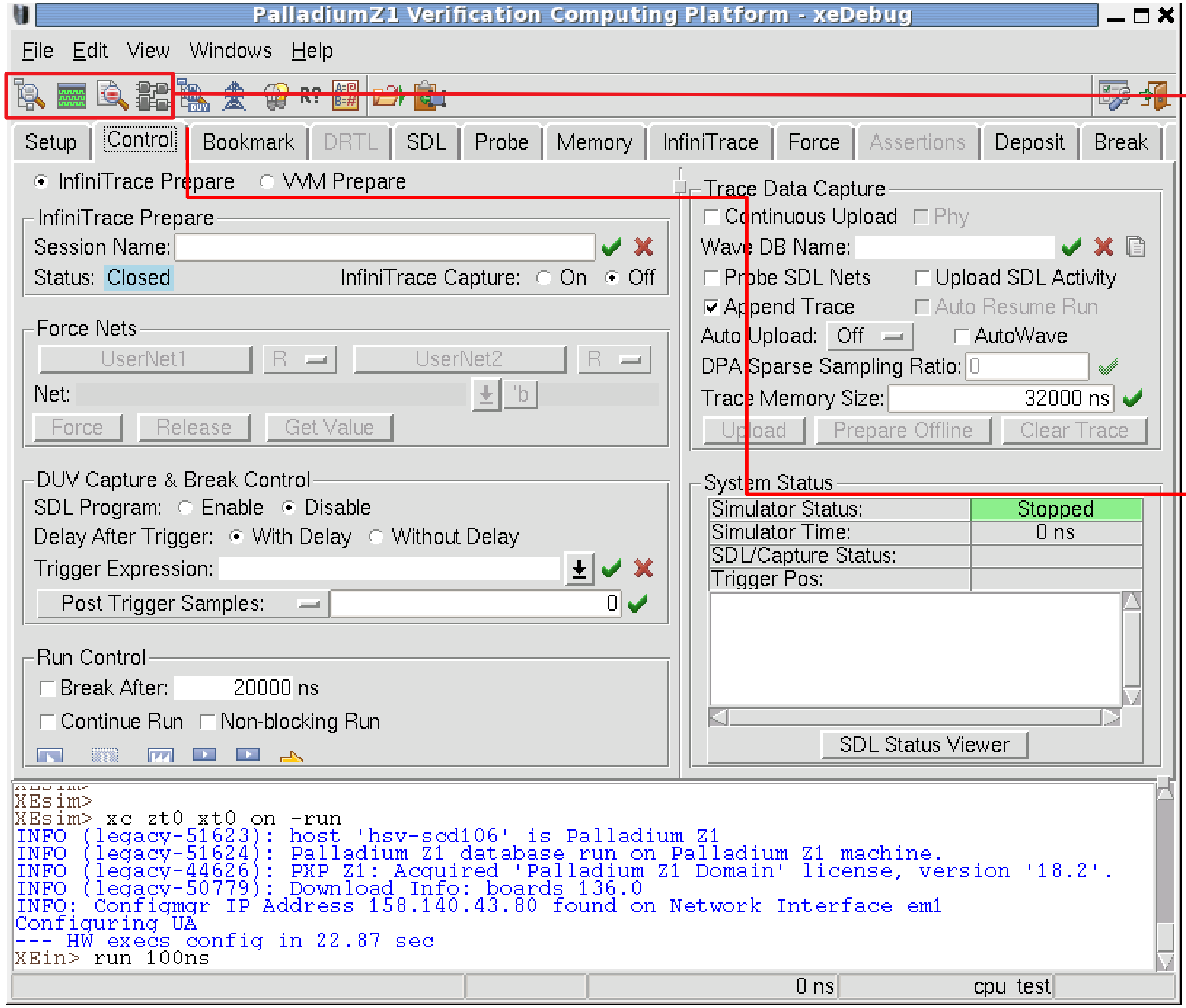
- The performance of the genWave utility depends on the LSF settings and execution of the LSF jobs. Ensure that the LSF settings are specified properly. Otherwise, the genWave utility might become unresponsive.
- Failure of LSF xeDebug jobs impact the functioning of the genWave utility.
- The genWave utility does not verify the value of the following environment variables:
  – CDN_XED_HOST, CDN_MT_HOST, CDN_XED_HOST2 and CDN_MT_HOST2
- User needs to set the following environment variables to ensure proper functionality:
  – CDN_XED_HOST = “bsub –q linux–R “rusage[mem=500000]”
  – oCDN_MT_HOST = “LSF:5: bsub –q linux–R “rusage[mem=300000]”

Outline

Set up simulation snapshot for the RTL - IXCOM flow

- In Simulation Acceleration flow, an simulation snapshot is created automatically for testbench and DUT, and xeDebug knows its location. This snapshot has the “instrumented” RTL created by IXCOM. It is similar to the original RTL, but with some extra logic and signals added.
- HINT: If you want an additional reference to your original unmodified (non-instrumented) RTL, use +xref_browser at compile time

Invoke Browsers



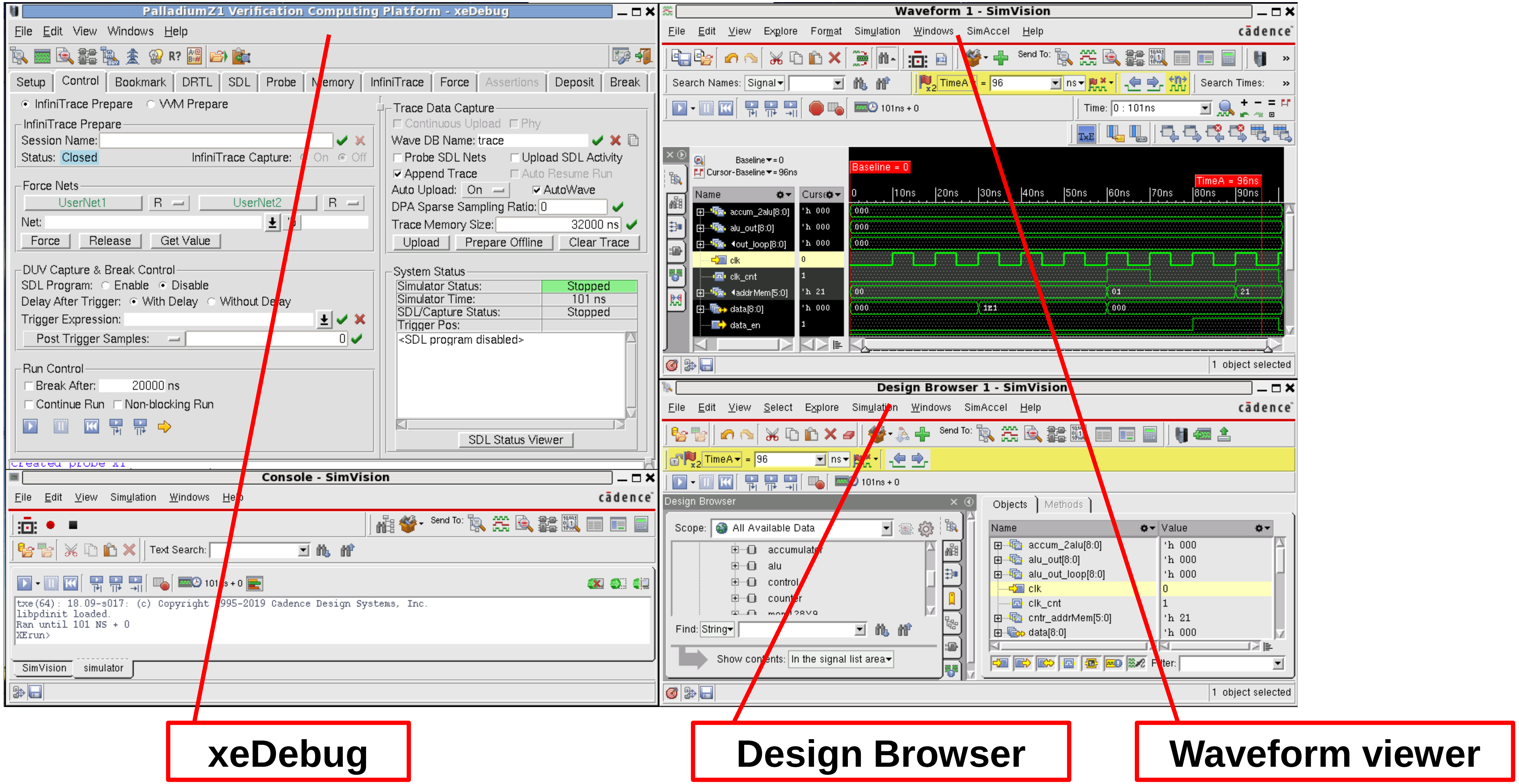
SimVision:

- Schematic Viewer
- Source Browser
- Waveform
- Design Browser

Note: Source Browser shows instrumentation code after ixcom and not original RTL

DUV Browser

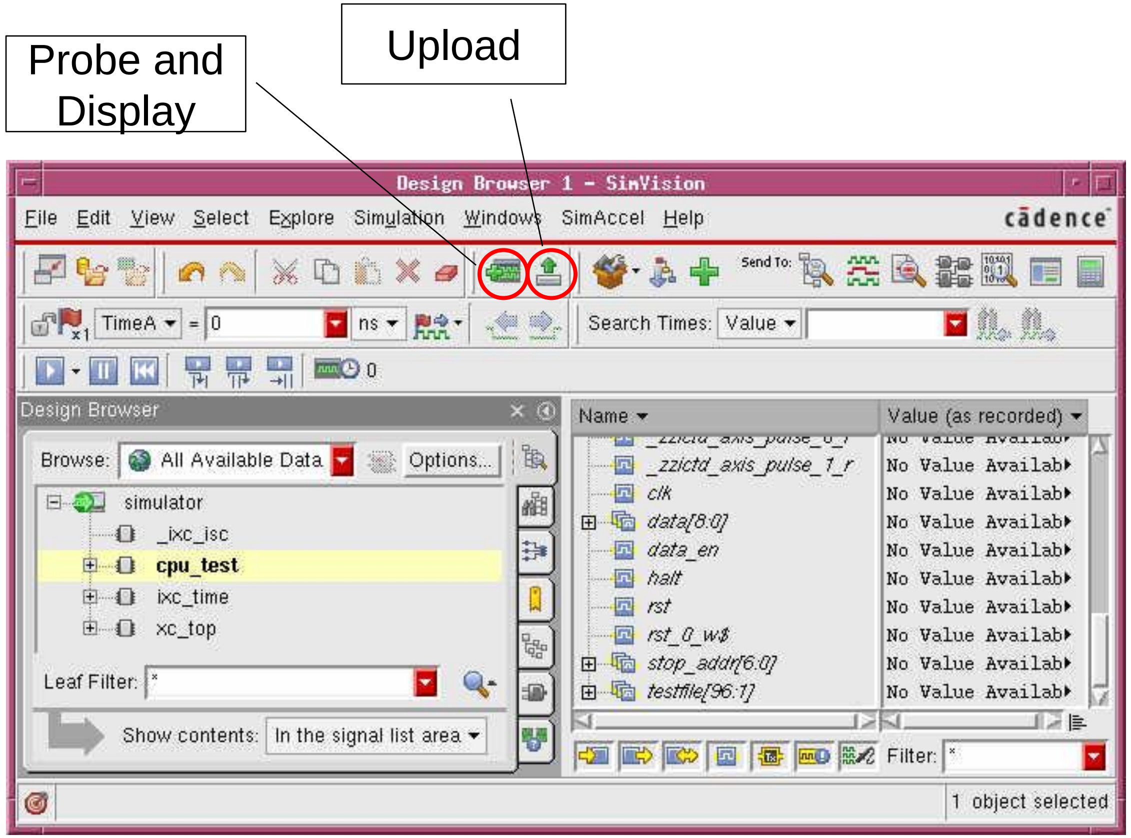
xeDebug & SimVision Environment



  Cadence Confidential

cādence®

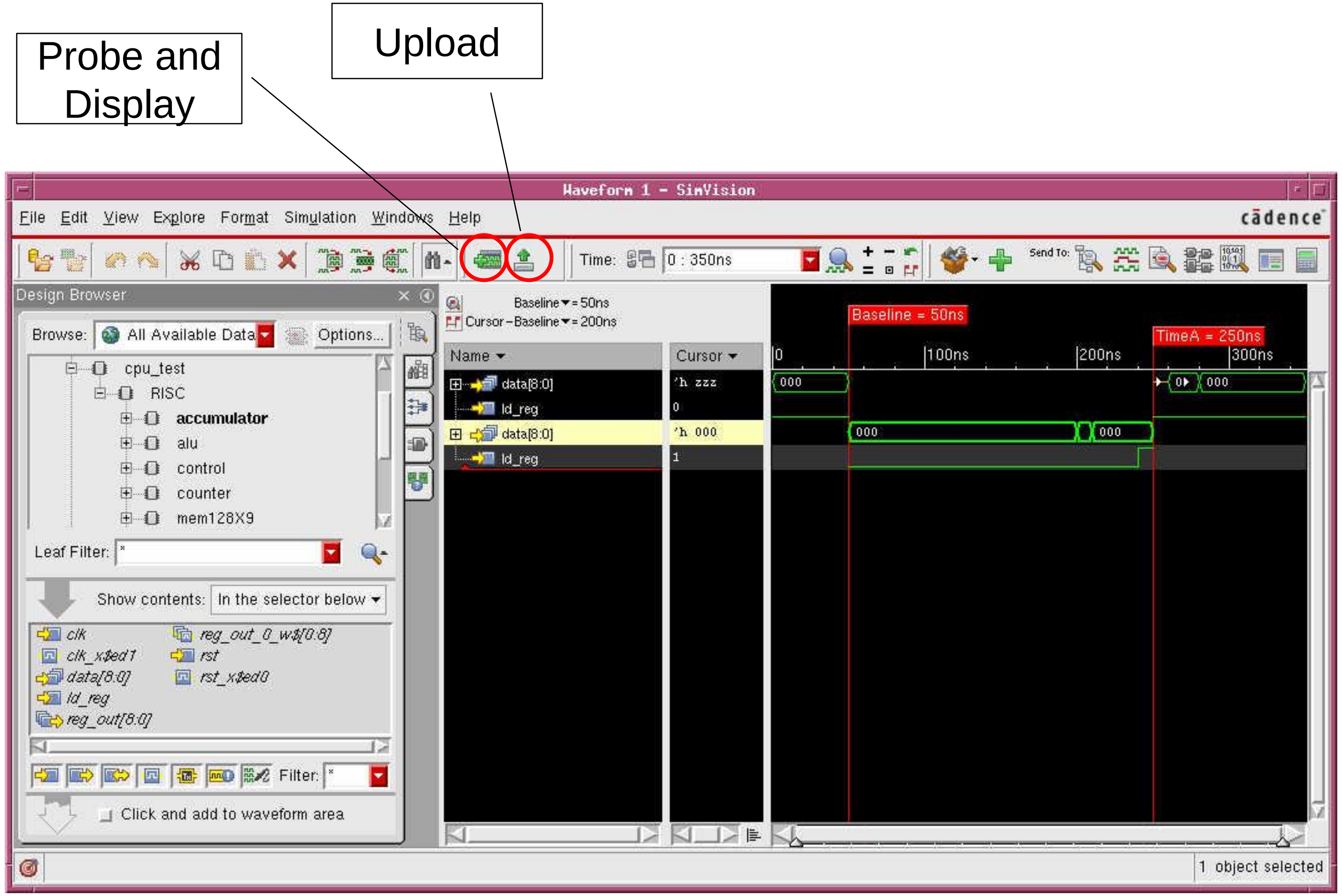
SimVision Design Browser



- **Probe and Display** adds signal names to the waveform viewer but does not upload waveform data
- **Upload** uploads waveform data to the waveform database (trace.shm) and to the waveform viewer

cādence

SimVision Waveform



Outline

SDL - Introduction

- SDL is the language you use to define a Trigger State machine. It has all the capabilities of commercial logic analyzers – plus more.
- When user-defined logic conditions are met, logic analyzer will “trigger”
  - In all modes, stop collection of trace data
  - In all modes except LA mode, stop the running design
    - In LA mode, trace data collection stops but design keeps running
- Trigger is like a simulation breakpoint
  - But can be more powerful, because triggering can be determined by a state machine that you define during debug
  - Trigger is sometimes less powerful, because you can’t break on a source line
- Trigger state machine can be changed dynamically during a debug session, all signals available to SDL without recompiling the design
- In SA, SDL is available only for signals visible in the DUT, and only when running in hardware. However, you can have simulation breakpoints in the testbench, and you can automatically convert some types of simulation breakpoint in the DUT to equivalent SDL.
  - sdl –ncbreakpoints -import

SDL – Basic Properties

- SDL tracks sequences of events by monitoring design objects such as signals, assertions, CPF/UPF objects using a state machine description
- Multiple instances of SDL can be used to track multiple independent sequences of events
- Each SDL instance has its own hardware resources:
  - One state machine
  - Expression evaluators (can be used inside state machines, or independently)
  - 2 general purpose counters (for counting events)
- Each SDL instance can perform, on a cycle by cycle basis, any of the following actions:
  - ACQUIRE: decide whether an individual probe sample should be acquired or rejected
  - TRIGGER: stop design clocks and/or waveform acquisition (depends on settings)
  - EXEC: Execute a TCL/XEL command/proc
  - DISPLAY: print out a formatted message, including time and signal values
  - Control internal SDL resources (go to a different state, increment/decrement/load counters, etc.)

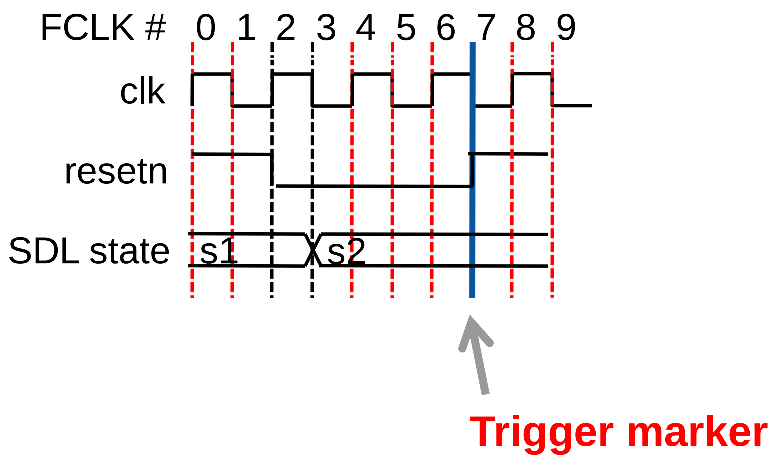
cadence

SDL Example

Trigger when resetn goes low, and then goes high:

```
State  s1
{
    if ( resetn == ‘b0 ) {
        goto s2;
    }
}

State  s2
{
    if ( resetn == ‘b1 ) {
       Trigger;
    }
}
```

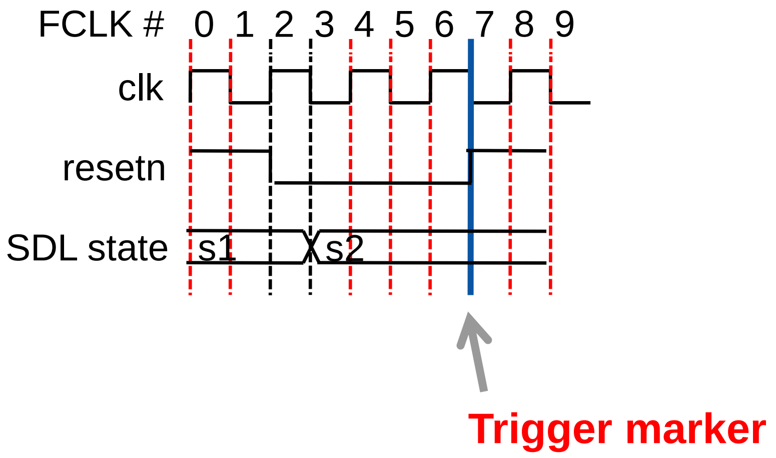
State s1: wait ‘til resetn==0, then goto s2.
State s2: wait ‘til resetn==1, then trigger.



SDL – Basic Properties

- At the beginning of the run we are in the first state of the SDL program
- At each FCLK, SDL program can only be in one state
- If in a certain FCLK we are in state S1 and we execute “Goto S2”, then in the next FCLK we will be in state S2
- At each FCLK,
  - First, all signals in the design are updated
  - Then, all the tests in the SDL for current state are evaluated concurrently
  - Then, (depending on the test results) 0 or more actions are executed concurrently

```
State  s1
{
    if ( resetn == ‘b0 ) {
        goto s2;
    }
}
State  s2
{
    if ( resetn == ‘b1 ) {
       Trigger;
    }
}
```



Running SDL from Command Line

The script in this example does the following:

- Set the design database directory, select emulator, and download the design.
- Request to probe all the signals that are used by SDL.
- Load the SDL program in file mySdlFile.tdf into the emulator and enable it.
- run until trigger.
- Print status of SDL at end of run.
- Write a trace report on SDL activity during the run into the file sdldump.txt.

```
debug .
host .
download
sdl -autoprobe on
sdl -load mySdlFile.tdf
sdl -enable
run
puts [sdl -report]
sdl -tracedump sdldump.txt
```

SDL Expressions

- Based on Verilog Expression Syntax:
  - Variety of boolean and logical operators.
  - Unsigned magnitude comparison.
  - Concatenation.
  - Follows Verilog's precedence and association rules.
- Special syntax constructs for detecting signal transitions.
- Operands may include:
  - constants.
  - signal values.
  - assertion states.
  - states of UPF/CPF objects.
  - values of SDL's own state machine's general purpose counters (counter1 and counter2)

SDL Expressions

Testing value of SDL's general purpose counters:

- Only > and <= comparison operators are allowed within SDL internal counters
- The right operand must be a non negative integer number (up to 40 bits)

```
state1 {
  if(counter1 <= 5)
    trigger;
  if(counter2 > 50000)
    goto state2;
}
```

SDL Expressions

- Constants and Special Values
  - numeric Constants:
    - binary `'b11111110` or `8'b11111110`
    - octal `'o376` or `8'o376`
    - decimal `254` or `d'254` or `8'd254`
    - hexadecimal `'hFE` or `8'hFE`

- Special values (based on values from current and previous cycle)
  - positive edge on signal A `A == 'bP`
  - negative edge on signal A `A == 'bN`
  - any transition on signal A `A == 'bT`
  - no transition on signal A `A == 'bS`
  - signal A stays high `A == 'bH`
  - signal A stays low `A == 'bL`

SDL Expressions

- Detecting Transitions
  - Detect any positive or negative transition on any of the 4 bits using the transition() operator

```
if(transition(C[3:0]))
  trigger;
```

SDL Actions

- Basic Actions
  - trigger
  - exec
  - display
  - goto
- Trace Control Actions
  - acquire
  - no_acquire
- General Purpose Counters Control
  - load
  - decrement
  - increment
  - start
  - stop

cādence

SDL Actions: Trigger

```
if (expr)
  trigger;
```

- A trigger may stop one or more of the following:
  - Design clock(s).
  - Waveform capture.
  - Software simulator (during co-simulation under IXCOM).
- Exact behavior depends on run mode and run time settings

cadence

SDL Actions: Exec

```
if (expr)
  exec ”force SIG [3:0] 15”;
```

- Executes the TCL or XEL command inside the "..."
- The command is executed with some delay relative to the expression becoming true.
  - The delay is no more than one simulation time step when running in co-simulation mode in IXCOM flow

SDL Actions: Display

```
if (expr)
 display("time=%t  A=%h B=%d", A[127:0], B[63:32]);
```

- Print out formatted messages to the console and or file(s).
- Uses syntax similar to Verilog's `$display` system task.
- For improved performance the messages are streamed through a memory buffer in the emulator while the clocks are running.
- Design clocks are paused only if the buffer becomes full (in run modes that allow to pause the clocks).

SDL Actions: Acquire

```
if (expr)
  acquire;
```

- A waveform sample is acquired into the trace buffer when `expr` is true.
- The presence of `acquire` anywhere in the SDL program implies that by default, data is not acquired into the trace buffer.
- Any number of `acquire` actions may appear in the SDL program.

SDL Actions: No_Acquire

```
if (expr)
  no_acquire;
```

- Works similar to `acquire`, but with opposite semantics.
- A waveform sample is prevented from being acquired into the trace buffer when `expr` is true.
- The presence of `no_acquire` anywhere in the SDL program implies that by default, data is acquired into the trace buffer.
- Any number of `no_acquire` actions may appear in the SDL program.
- The actions `acquire` and `no_acquire` may not exist together in the same SDL program.

More SDL Features

- Up to 64 instances of SDL can execute concurrently
  - You must specify the number of SDL instances at compile time
  - Each instance identified by a name and has its own set of resources (states, counters, etc)

```
instance i1;
state s1 { … }
state s2 { … }
instance i2;
state s1 { … }
state s2 { … }
```

  - Example: Trigger the first time at least one of the following is true:
    - Signal u1.A has been high for at least 5 consecutive FCLKs
    - Signal u2.A has been high for at least 5 consecutive FCLKs
    - Signal u3.A has been high for at least 5 consecutive FCLKs
  - How would you do it?

More SDL Features

- Up to 64 instances of SDL can execute concurrently
  - You must specify the number of SDL instances at compile time
  - Each instance identified by a name and has its own set of resources (states, counters, etc)

```
instance i1;
state s1 { … }
state s2 { … }
instance i2;
state s1 { … }
state s2 { … }
```

  - Example: Trigger the first time at least one of the following is true:
    - Signal u1.A has been high for at least 5 consecutive FCLKs
    - Signal u2.A has been high for at least 5 consecutive FCLKs
    - Signal u3.A has been high for at least 5 consecutive FCLKs
  - Possible solution: use three SDL instances. Each instance is the same as our first class problem, but using one of the signals u1.A, u2.A, u3.A

Using Multiple SDL Instances

Example of SDL program with two SDL instances:

```
// First SDL instance
Instance I1;
state first {
  if (top.I1.A[7:0] < 13) goto second;
}
state second {
  if (top.I1.A[7:0] == 3) trigger;
}
// second SDL instance
Instance I2;
state first {
  if (top.I1.B[7:0] < 13) goto second;
}
state second {
  if (top.I1.B[7:0] == 3) trigger;
}
```

By default, the compiler allocates hardware for a single SDL instance.

The following requests the compiler to allocate hardware for 10 SDL instances:

```
compilerOption -add {sdlInstances 10}
```

Other SDL Facts

- **You can modify/verify the SDL program at any time, even while the emulator is running. The modified SDL program is only loaded upon execution of the *run* command**
- **SDL Trace Dump facility can aid debugging:**
  - **sdl –traceDump <file.txt> will dump the SDL trace to specified file**

// Time-stamp	State	ACTIONS
1027	init	GOTO st1 ; START CNT2 ; LOAD CNT2 1024 ;
8322	st2	GOTO st3 ; NO_ACQUIRE ;
8335	st3	TRIGGER ;

How would you do it?

- **We wish to trigger the first time signal A is high for at least 5 consecutive FCLKs**
- **Use the following SDL features**
  - **Two counters are available: counter1 and counter2. For this exercise, counter1 is enough.**
  - **You can load a counter with a specific value:**

    load counter1 5;
  - **You can decrement a counter:**

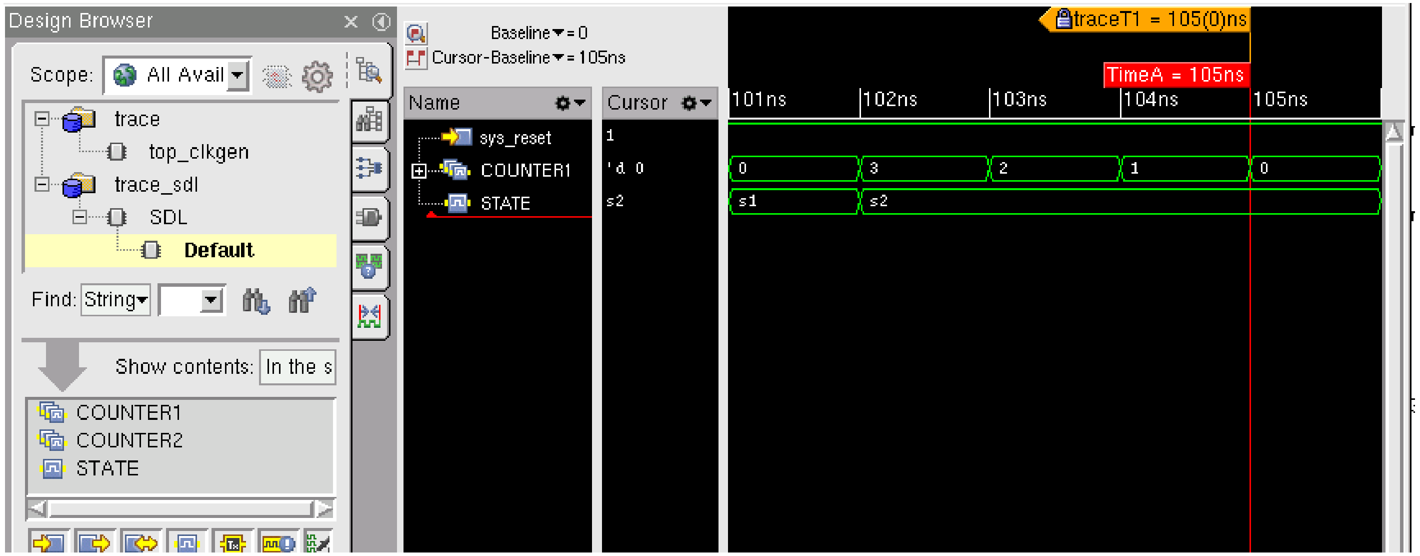
    decrement counter1;
  - **You can test if a counter is less than or equal to 0**

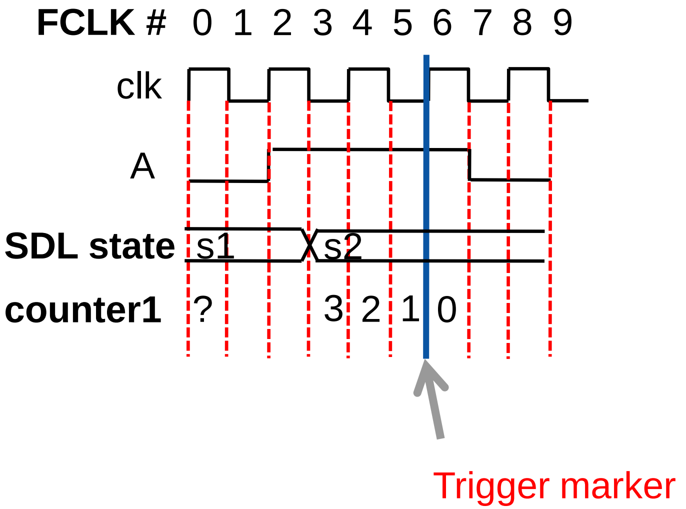
    if(counter1<=0){ … }

How would you do it? (answer)

```
State  s1
{
      if ( A == ‘b1 ) {
        load counter1 3;
        Goto s2;
      }
}
State  s2
{
     if ( A == ‘b0) {
        goto s1;
     } else if (counter1<=0) {
        trigger;
     } else {
        decrement counter1;
     }
}
```

triggers the first time signal A remains high for at least 5 consecutive FCLK cycles.





cadence

Outline

Overview

Modes of Operation

Run-time Logistics

Getting Waveforms

Viewing Waveforms in SimVision

SDL Language to Control Triggering

DRTL

Infinitrace

QEL commands for Debug

xeDebug GUI

Regression Mode

Log Files

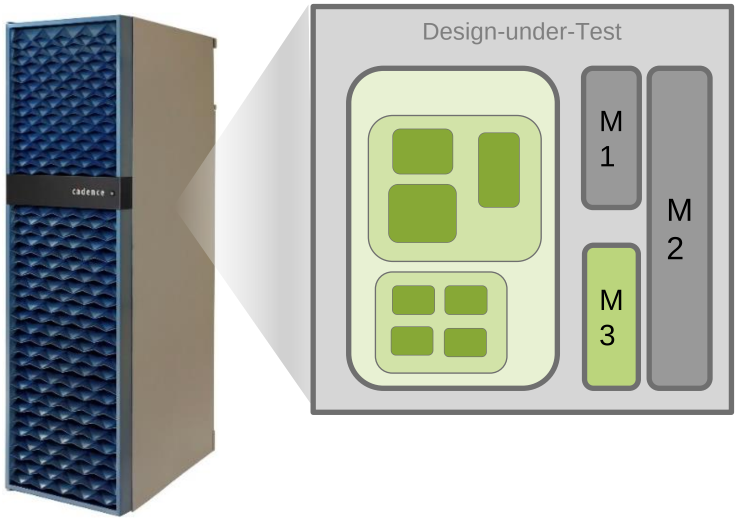
Dynamic RTL (DRTL)

- Constructed from Verilog/VHDL RTL language
  - Loaded and instantiated at runtime
  - Can monitor, trigger and provide runtime control
- Advantages of DRTL are
  - User can easily code state machines or copy one from the DUT written in HDL
  - Able to Save and Load DRTL from precompiled files
    - Allows user to create a library of DRTLs for future use
  - Flexible, single module can be instantiated multiple times
    - User only needs to change nets connected to DRTL ports

Dynamic RTL

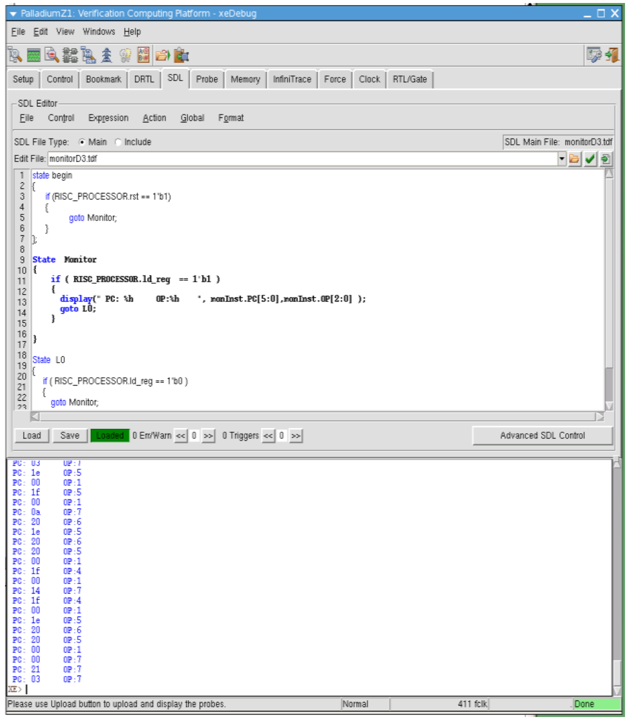
- Use model is similar to SDL
  - Loaded at runtime
  - Execution semantics follows that of DUT code (unlike SDL)
  - Code runs on existing available emulator resources
  - No impact on runtime performance
- Must be a synthesizable code
  - uses HDL-ICE under the hood for synthesis
- Compliments SDL
  - Easier to write complex logic, complex state machines
  - Uses SDL to provide control of the runtime session

Dynamic RTL




```
module riscMon(clk, rst, data, ld, PC,OP);
   input clk, rst;
   input [8:0] data;
   input       ld;
   output [5:0] PC;
   output [2:0] OP;

   reg  [5:0] rPC;
   reg  [2:0] rOP;
   reg [3:0]  currentState;
   reg [3:0]  nextState;

   localparam STATE_INIT = 4'b0000;
   localparam STATE_LOAD = 4'b0001;
   localparam STATE_FETCH = 4'b0010;
   localparam STATE_EXECUTE= 4'b0011;

   assign PC = rPC;
   assign OP = rOP;

always @(posedge clk or negedge rst)
  begin
    if (rst == 1'b0 )
      currentState <= STATE_INIT;
    else
      currentState <= nextState;
  end

always @( * )
begin
   case (currentState)
     STATE_INIT: begin
        if ( ld == 1'b1) begin
           nextState =  STATE_LOAD;
        end
     end
     STATE_LOAD: begin
        rPC = data[5:0];
        rOP = data[8:6];
        nextState = STATE_FETCH;
     end
     STATE_FETCH: begin
        if ( ld == 1'b0) begin
           nextState =  STATE_INIT;
        end
     end
   endcase // case (currentState)
End
endmodule
```

```
state begin
{
    if (RISC_PROCESSOR.rst == 1'b1)
    {
        goto Monitor;
    }
};

State  Monitor
{
   if ( RISC_PROCESSOR.ld_reg  == 1'b1 )
   {
      display(" PC: %h     OP:%h     ",
monInst.PC[5:0],monInst.OP[2:0] );
      goto L0;
   }

}

State  L0
{
   if ( RISC_PROCESSOR.ld_reg == 1'b0 )
   {
      goto Monitor;
   }
}
```



DRTL code
- State machine monitoring design
- Read-in / compiled at runtime (similar to SDL)

SDL code
- Legacy control mechanism for emulator
- Accesses outputs from DRTL state machine

Dynamic RTL

- Using DRTL - There are 3 steps involved in using DRTL

  1. Define a DRTL module
  2. Instantiate a DRTL module
  3. Using DRTL module

Dynamic RTL
Defining a DRTL module

- DRTL must be written in HDL-ICE compliant RTL code
- Must be self-contained, all modules must be defined (no Blackbox)
  - NO search paths support, NO libraries
- Can be hierarchical in structures, with a single top module
- Cannot mix Verilog and VHDL in files passed to HDL-ICE
- Memories blasted to registers. Big memories are not allowed

Dynamic RTL
Defining a DRTL module

- Writing a Verilog/VHDL source file for DRTL (cont.)

  – DRTL source file can have Input/Output ports
    – Input ports are connected to signals in the DUT
    – Output ports can be input to other DRTL modules or SDL program
    – InOut ports are not supported

  – DRTL CANNOT update / provide feedback to the DUT – monitoring only

  – Supports System tasks $display, $qel
    – *$display* works the same way as in a Verilog code
    – *$qel* takes a single string as argument which is the XEL command to be executed

  – Required to run in one of two execution modes
    – Use output ports so data is passed from DRTL to other sources
    – If no output ports, then must contain either *$display* or *$qel*
    – No more than 1024 $qel and $display in total

Dynamic RTL
Defining a DRTL module

- Can take place before the design is downloaded on the emulator

- Able to pass Verilog and VHDL file on the same command line
  - Does NOT support mixed language in single file

- Complete syntax is:
  drtl -definemodule <DRTL_module_name> [-verilog | -vhdl | -sv] <file_name1> [file_name2] [-topmodule <DRTL_topmodule_name>]

- *<DRTL_module_name>* and *<top_module_name>* CANNOT be escaped

cādence

Dynamic RTL
Defining a DRTL module

- Once defined, the DRTL module can be saved to a file for later use and loaded when required
  - Allows re-use of DRTL, creating pre-compiled libraries of DRTL
  - Loading of pre-compiled DRTL module is very fast

- To save defined DRTL module(s) into a file, call:

  drtl –save [-overwrite] <file_name> [<DRTL_module_name>]

- To load a/some DRTL module(s) from a file, call:

  drtl -load [-overwrite | -ignore] <file_name> [<DRTL_module_name>]

Dynamic RTL
Instantiating a DRTL module

- Instantiation connects DRTL ports with DUT signals
  - Must occur after the design has been downloaded on the emulator

- Three ways to instantiate port connections

  drtl –addInst drtlA my_inst (.clk(dut.clk), .in(dut.signalA [3:0]))
  drtl –addinst drtlA my_inst (dut.clk, dut.signalA[3:0])
  drtl –addinst drtlA my_inst (.clk(dut.clk), .in(4b’1010))

- During instantiation, the DRTL squeezes itself in unused logic
  - Size of DRTL code can be found in log files located at ./tmp/drtl

Dynamic RTL
Instantiating a DRTL module

- To get more resources for DRTL that does not fit, you can recompile the design
  - If DRTL is too large to fit in, user would see error message when instantiating
  - Recompile of design doesn't guarantee DRTL will fit

- In VXE 16.5, large DRTL instances were not guaranteed to schedule

- With VXE 18.1/18.5, the scheduling of DRTL instance is guaranteed no matter the size with the following compilerOption
  - *compilerOption –add {DynamicNetlist <N>}*
  - User must make sure <N> is larger than size of DRTL

Dynamic RTL
Using DRTL

- One of the two following use models must be implemented in a DRTL module
  - Use DRTL output port
  - Use System tasks in DRTL code ($display to print or $qel to execute XEL commands)

- DRTL is fully functional for a design compiled with CAKE1X mode with VXE 18.1/18.5

Dynamic RTL
Using DRTL

- Use DRTL output port example: for module drtlA, instance my_inst, there is an output port out1.
  - It can be referenced to as my_inst.out1
    drtl –definemodule drtlA drtl_source.v –topmodule drtl_top
    drtl -addinst drtlA my_inst (.clk(clk), .in(DUT.in))
  - The output of DRTL can be used in an SDL program

```
Instance TrackRiscInstr;
state begin
{
    if( my_inst.out1 )
                goto NEXT_STATE;
};
State NEXT_STATE
{
    if( my_inst.out2)
                EXEC("force reset 1");
};
```

```
-------------------------------drtl_souce.v----------------------------

module drtl_top (clk, in, out1, out2 );
input clk;
output reg out1, out2;
input [3:0] in ;
always @(*)
   begin
      if(clk && in  == 4'b1010)
         begin
                out1 <= 1'b1;
                out2 <= 1'b0;
         end
    end
 endmodule
```


cādence

Dynamic RTL
Using DRTL

- Use System tasks in DRTL code
  - Use $display to print out statements
  - Use $qel to execute XEL commands

```
--------------------------------drtl_souce.v-----------------------------

module drtl_top (clk, in);
input clk;
input [3:0] in;
always @(*)
    begin
        if(clk && in  == 4'b1010)
            begin
                    $display(“%h”, in[3:0]);
                    $qel(“trigger”);
            end
     end
 endmodule
```

Outline

Overview

Modes of Operation

Run-time Logistics

Getting Waveforms

Viewing Waveforms in SimVision

SDL Language to Control Triggering

DRTL

Infinitrace

QEL commands for Debug

xeDebug GUI

Regression Mode

Log Files

InfiniTrace

- **InfiniTrace is a special mode in which certain trace data is continually saved during a run**
  - Emulator state saved periodically; inputs saved continuously
  - This run is called the "prepare session"
  - There is no limit on the length of the run
- **At any later time, you can enter an "observe session" using the saved data in observe session, you can**
  - "Go to" any time during the run, and upload a waveform for a time window before this time
  - Replay the emulation going forward from this time, with any trigger condition
- **Observe session uses the emulator but not the target system**

InfiniTrace Performance Cost

- **There is an emulation speed penalty during the prepare session**
- **The slowdown is the product of two factors: ST and PI**
  - **ST (slowdown due to periodic state dump) depends on the amount of state and frequency of dumping the state**
  - **PI (slowdown due to dumping primary inputs) depends on the number of target primary inputs and bidis, and on FCLK speed.**

Note: You can turn prepare session tracing on/off. Performance penalty only occurs while tracing is on.

InfiniTrace in IXCOM flow

- **Simulator is treated like an external target**
  - **DUT state saved periodically; DUT inputs saved continuously**
- **During observe session**
  - **Simulator does not re-run**
  - **Only DUT signal are visible**
- **In IXCOM flow, InfiniTrace has a performance penalty, but the penalty affects only the emulator’s share of the total time**

Using InfiniTrace to get unlimited trace depth

- **The most common use of InfiniTrace is to run a small section of a long run over and over with a different trigger condition each time**
- **A small section of the design can be run over and over with a different trigger condition each time, provided the trigger condition only uses signals available in the DUT**
- **InfiniTrace can also be used to get unlimited trace depth. In an observe session, you can set the trace depth to any value. However, if you set it very large, the upload will take a long time, especially if the number of signals uploaded is large.**
  - **In such a case, consider using triggering to find the event of interest, and use a small trace depth for fast upload**

Outline

Run Modes

- **xc run**
  - This mode gives the slowest performance, but is good for initial design bringup
- **xc tbrun**
  - This mode gives faster performance, because multiple DUT clock cycles can be run without interacting with the testbench
  - To use this mode, one or more clocks (usually the fastest ones) must be generated in the DUT
  - Possible pitfall: testbench could miss events in HW, unless $export_event, tbcall, or tbcall_region are used
- **xc [autorun|nbrun]**
  - Alternatives to tbrun with even less synchronizations, expert level
- **xc match**
  - Run in both software and hardware, compare
  - Use this mode for debugging when a design runs correctly in software mode, but fails in hardware mode
- **The above commands prepare to swap the design into hardware**
  - To actually swap in, use “run –swap”.
  - To swap back to software, use “xc off” followed by run –swap.
  - To release the emulator, use “xc free”.

Run Modes

- **C-TestBench Run Mode (CTB Run Mode)**
  - CTB run mode supports C-only testbench tasks/functions to and from hardware.
  - The C hardware scheduler to xmsim context-switching is completely avoided until the end of the run.
  - All transactions between C-only testbench tasks/functions and hardware are done in the C space and the control does not switch back to HDL.
  - This run mode is categorized into CTB TBrun and CTB NBrun modes. These modes are similar to the TBrun Mode and NBrun Mode.
- **CTB TBrun Mode**
  - The same as TBrun mode except that it only supports C tasks/functions to/from hardware.
  - xc on -ctb –tbrun
- **CTB NBrun Mode**
  - The same as the CTB TBrun mode except that the events that are scheduled by concurrent GFIFO calls, are executed immediately while the hardware clock is still running
  - xc on -ctb -nbrun 0

Symbol & Value Commands

- `symbol` **creates a symbol with multiple signals.**
  - **Example:**

    symbol –add my_symbol RAS CAS WE CS
  - **You can then force, deposit, or get the value of the entire symbol as a unit.**
- `value` **finds value of a signal, bus, or symbol at any time during emulation. Emulator need not be stopped to get the value**
  - **Examples:**

    value RAS

    value my_symbol

    redirect {puts “value of my_symbol is [value my_symbol]” } >> myfile

Force/Deposit

- **For testbench signals, and for DUT signals when running in software, you can**
  - Force any signal using force command
  - Deposit a value to any register using deposit command
- **For DUT signals when running in hardware,**
  - Force or deposit is not necessarily allowed
  - To guarantee a DUT signal can be forced, or register output can be deposited,
    - Declare it as a keepNet during compile, in compilerOptions.qel
    - keepNet –add <signal>
    - IO between testbench and DUT are automatically treated as keepNet
    - Or, use $export_frcrel(<signal>) in a testbench module
    - This gives fast force through SA channel but consumes resources
  - If you force a DUT signal when running in software, and then swap to hardware,
    - If the signal cannot be forced in hardware you should get an error
- **Force command in DUT RTL**
  - In “initial” statement: will be implemented automatically
  - In “always” statement: will be ignored unless you use ixcom +hwfrc

XEL Commands for Force

- Forcing/Releasing Values to DUT wires/variables

```
XEL commands:
   force <object_name> <value>
   release <object_name>
```

- In IXCOM flow, to dynamically force/release signals, do either of the following

  - add $export_frcrel(signal) before compiling the design

  - at run time, execute the following command

```
XEL command:
  xc export_frcrel <object_name>
```

  - Hint: keepnet may be needed

XEL Commands for Deposit

Depositing Values to DUT wires/variables

```
XEL commands:
   deposit <object_name> <value>
```

- In IXCOM flow, to dynamically deposit value to a DUT variable, do either of the following

  - add $export_deposit(variable) before compiling the design

  - at run time, execute the following command

```
XEL command:
  xc export_deposit <object_name>
```

  - Hint: keepnet may be needed

Deposit / Force / Release / Value Commands

- **Any net specified at compile time as a `keepNet` can be set, forced or released**
- **Forced value remains until XEL `release` command is issued**
- **`deposit` operation on flipflop or latch output sets the state of the output and remains until the design changes its state**
- **You can `deposit` or `force` a signal, bus or symbol with a hex, octal, binary, or decimal value.**
- **When forcing a bus or symbol, if all bits must change at the same time, then you must make sure FCLK is stopped when you apply the force**
- **`value` returns the current value of the signal or symbol**

cādence®

Memory Commands (Basic options)

- **Load or dump at any time**
- **Load or dump faster if clocks are stopped**
- **memory –load [<format>] <memory_name> -file <file_name>**
  - Writes to the memory instance specified by memory_name from the file specified by file_name
  - Specifying the format is optional
- **memory –dump <format> <memory_name> -file <file_name>**
  - Dumps memory to the file specified by file_name
  - Reads from memory instance specified by memory name
  - User must specify the format of the output file

Access Signal Values: Memory Access

Memory load

```
XEL command:
   memory -load [<format>] <memory_name> -file <file_name>
```

- Writes to the memory instance specified by the memory_name from the file specified by file_name

Memory dump

```
XEL command:
   memory -dump <format> <memory_name> -file <file_name>
```

- Dumps memory to the file specified by file_name

Memory Command (additional options)

- **memory –list**
  - Displays the names of all the memory instances in the design
- **memory –set -all | memory_instance_name**
  - Sets all or specified memory instances to 1
- **memory –reset –all | memory_instance_name**
  - Resets all or specified memory instances to 0
- **Other options available for streaming data from/into files**

Memory File Formats

- **ASCII Formats:**
  - %pd_b : Palladium ASCII format; radix bin
  - %pd_o : Palladium ASCII format; radix octal
  - %pd_d: Palladium ASCII format; radix dec
  - %pd_h: Palladium ASCII format; radix hex
- **Verilog ASCII Formats:**
  - %readmemb : readmemb format
  - %readmemh : readmemh format
- **Binary Formats**:
  - %pd_memtran: memtran format
  - %pd_raw: raw format
  - %pd_raw2 : raw2 format

Note: Binary file formats speed up memory load and dump

Outline

Overview

Modes of Operation

Run-time Logistics

Getting Waveforms

Viewing Waveforms in SimVision

SDL Language to Control Triggering

DRTL

Infinitrace

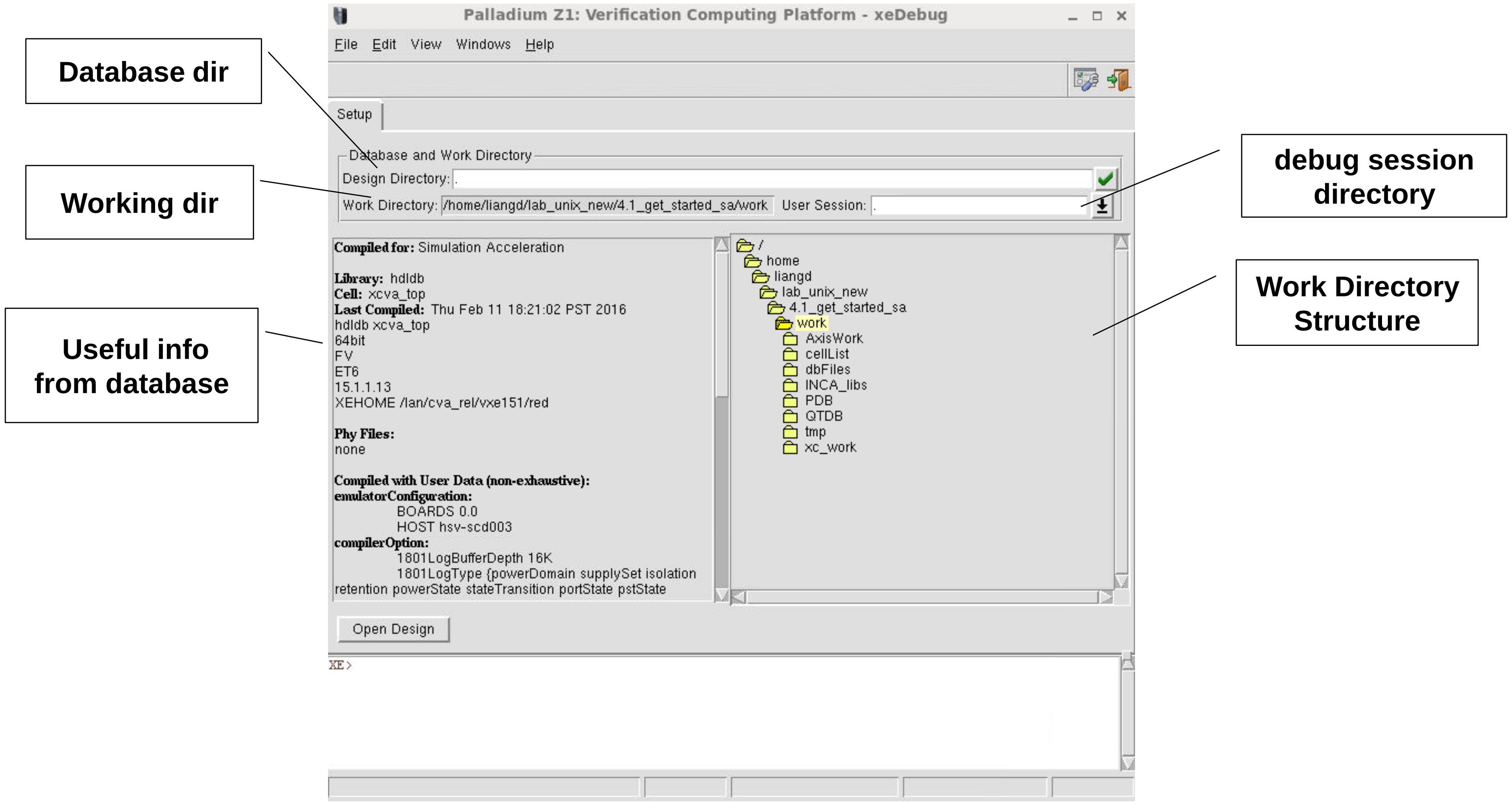
QEL commands for Debug

xeDebug GUI

Regression Mode

Log Files

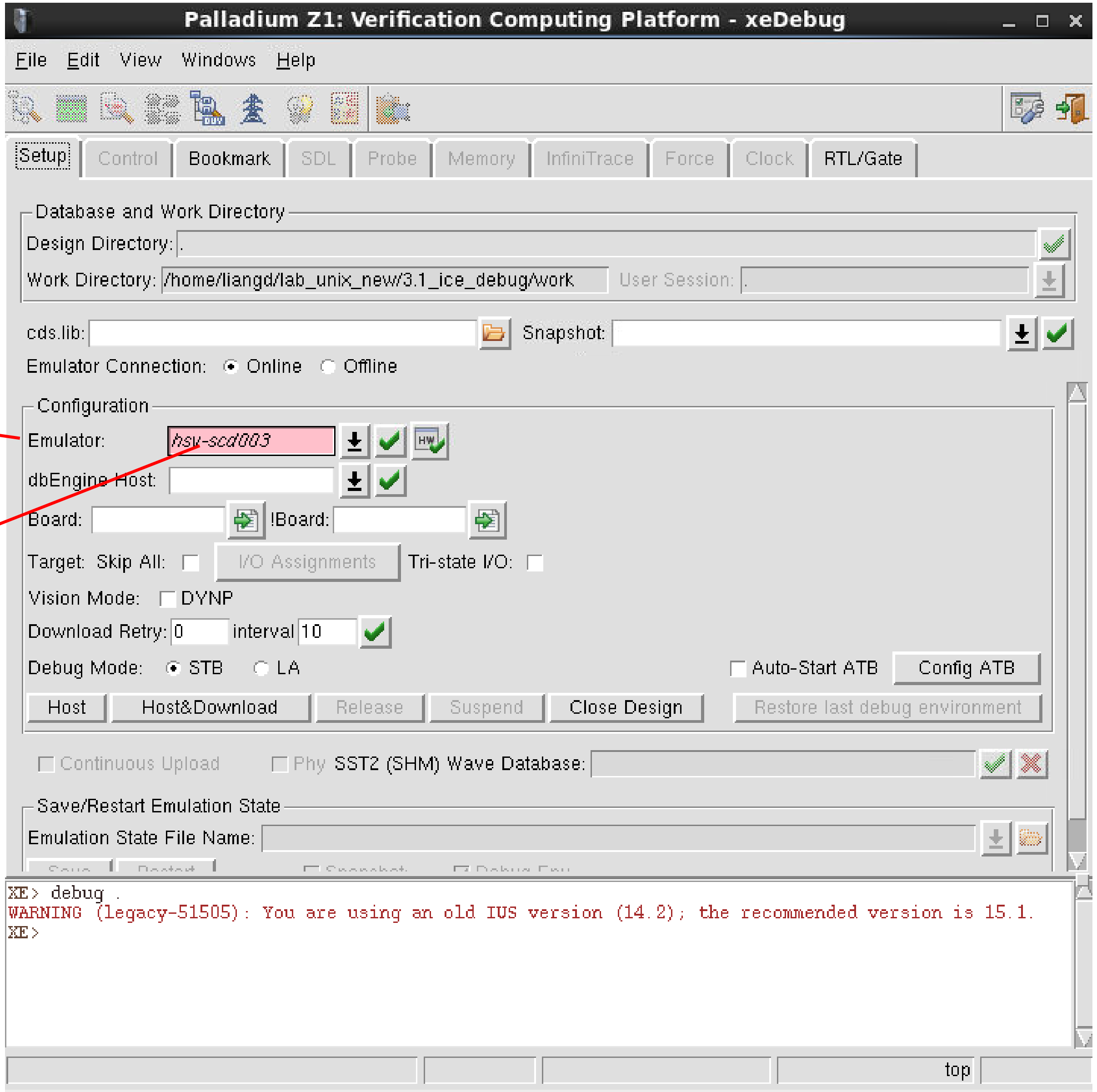
Setup Tab – Before Open



Setup Tab – After open, before download

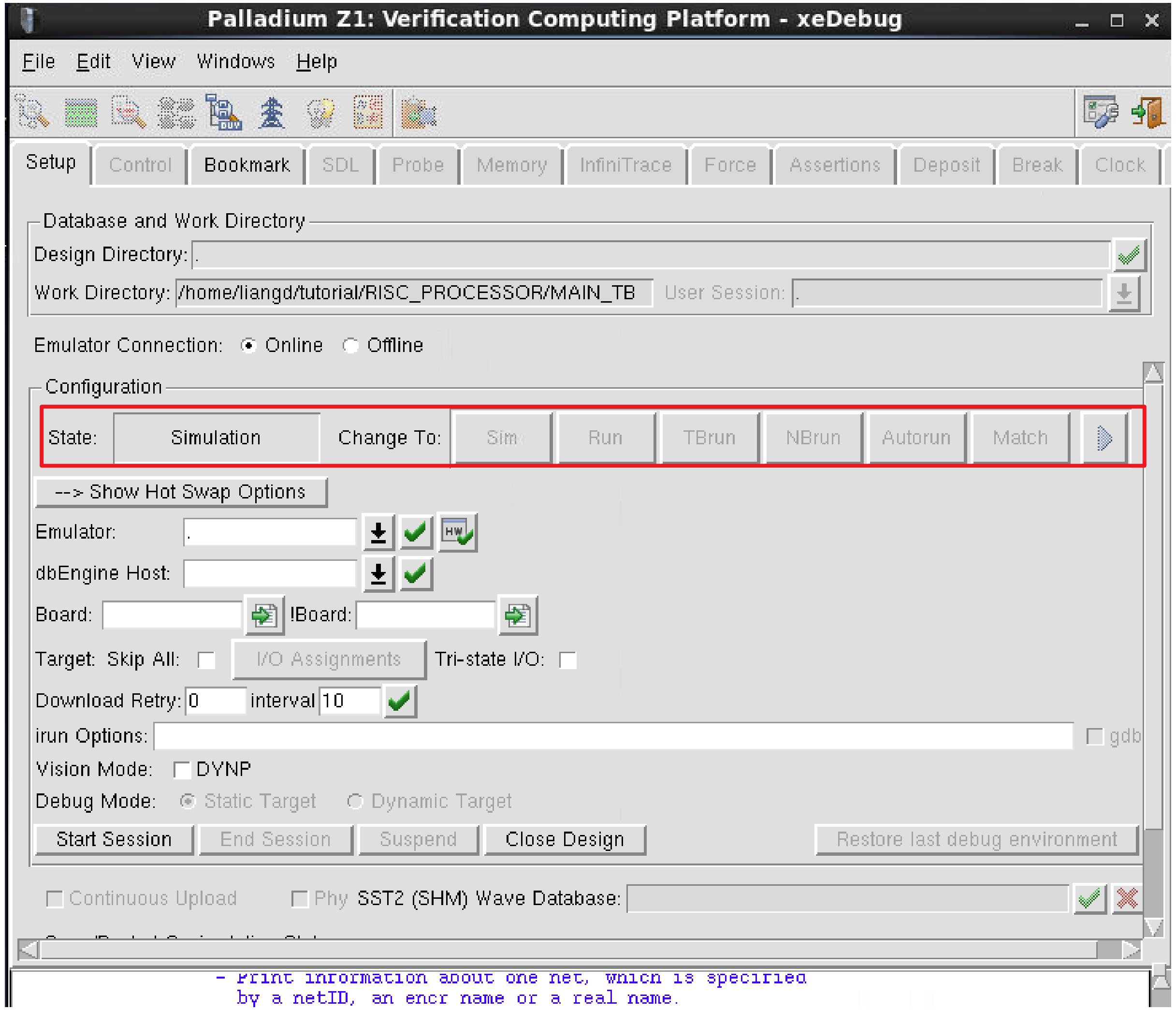
Where to download

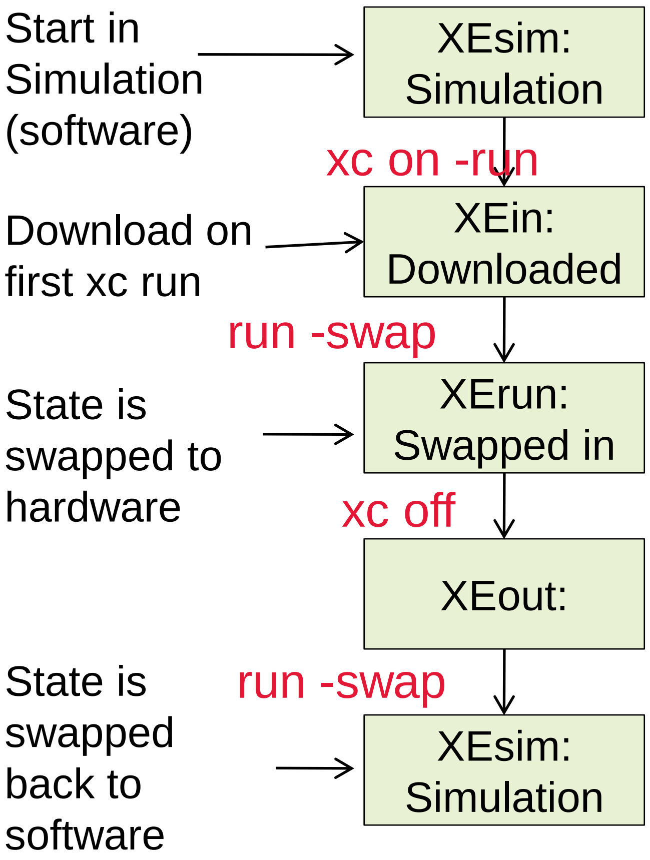
Important: if a field is pink, you must click the green check-mark to confirm your changes!



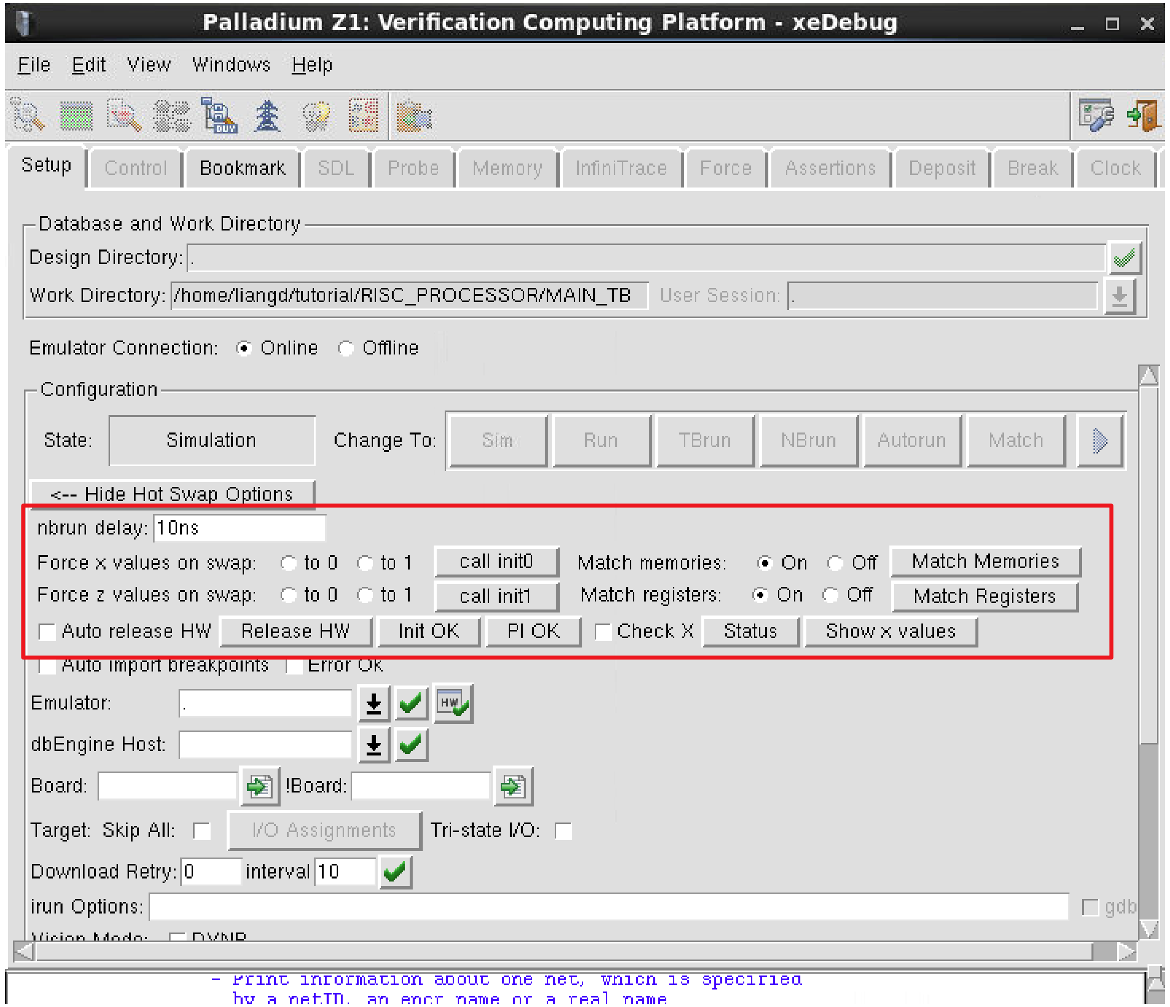
cādence®

Hot Swap Control



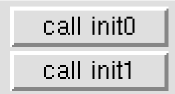


Hot Swap Options



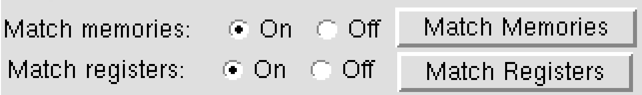
Forces x & z values to 0 or 1 when swapping into the emulator : “xc zt0 xt0” or “xc zt1 xt1”

Release emulator: “xc free”



Forces simulation values of x and z to 0 or 1: “ncsim call init0” or “ncsim call init1”

Allows swap to the emulator even if DUT initial blocks have not finished : “xc initok”



Options for Match mode

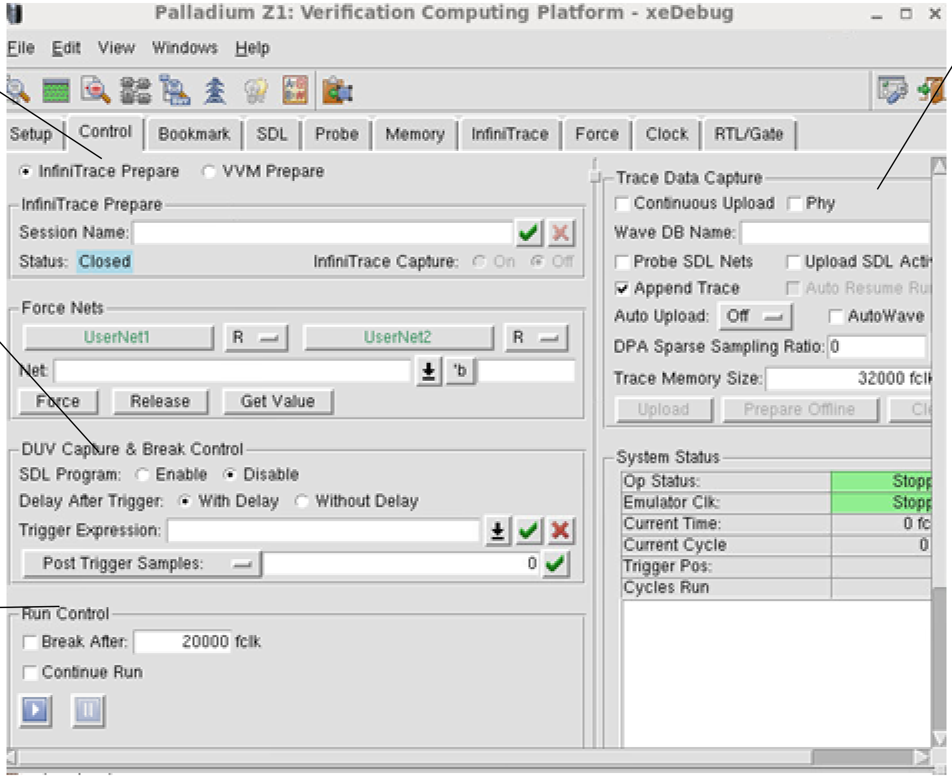
Shows actual emulation status: “xc status”

Control Tab

InfiniTrace Prepare Session

Trigger using SDL, or Trigger by simple expr

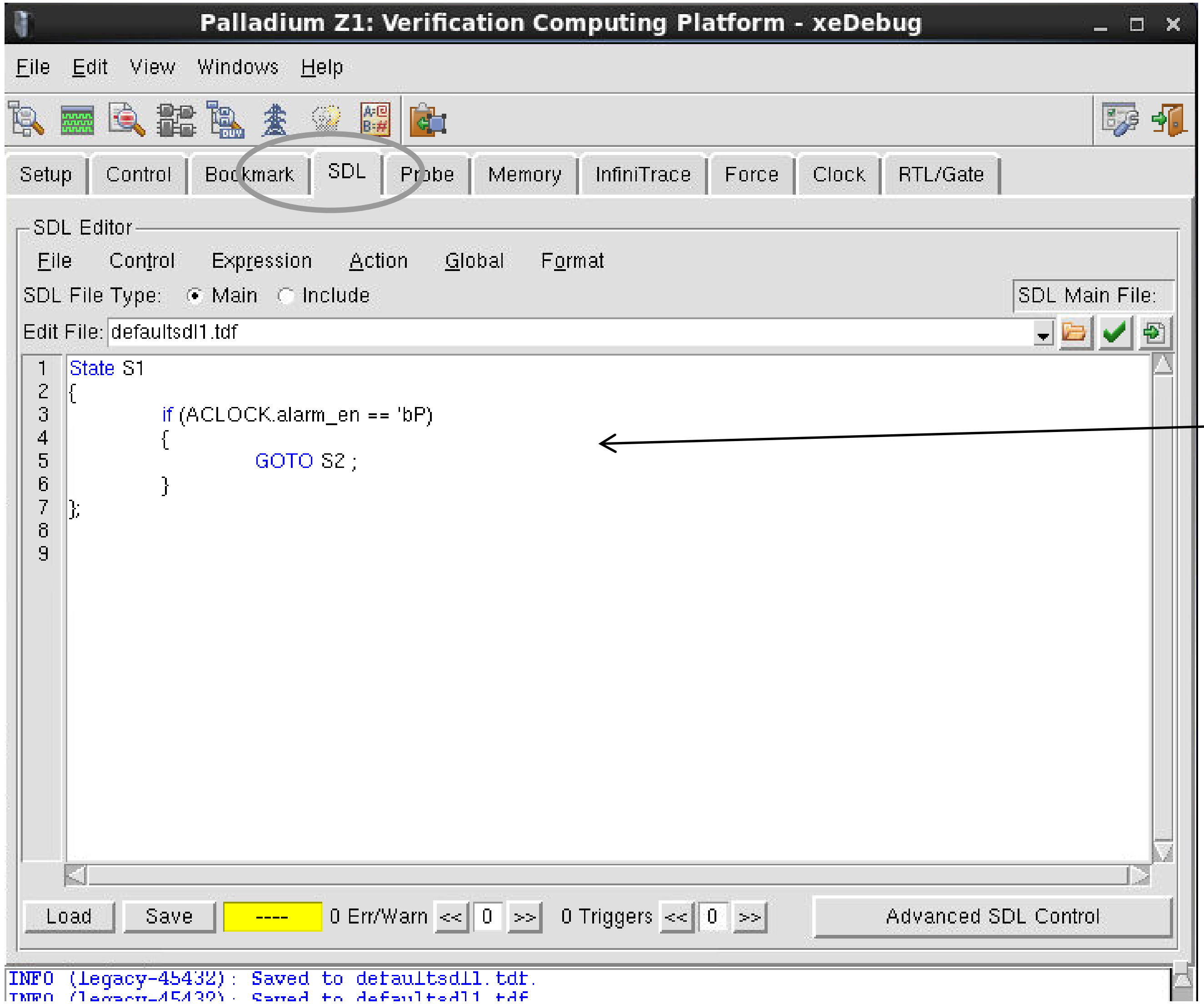
Run control



Trace Upload Options

- **Probe SDL Nets**
  - **Adds nets in SDL to waves**
- **Upload SDL Activity**
  - **Adds counters and states to waves**
- **Append Trace**
  - **Previous Upload data retained after new upload**
- **AutoUpload**
  - **Database –upload**
- **AutoWave**
  - **Start waveform viewer**

SDL Tab

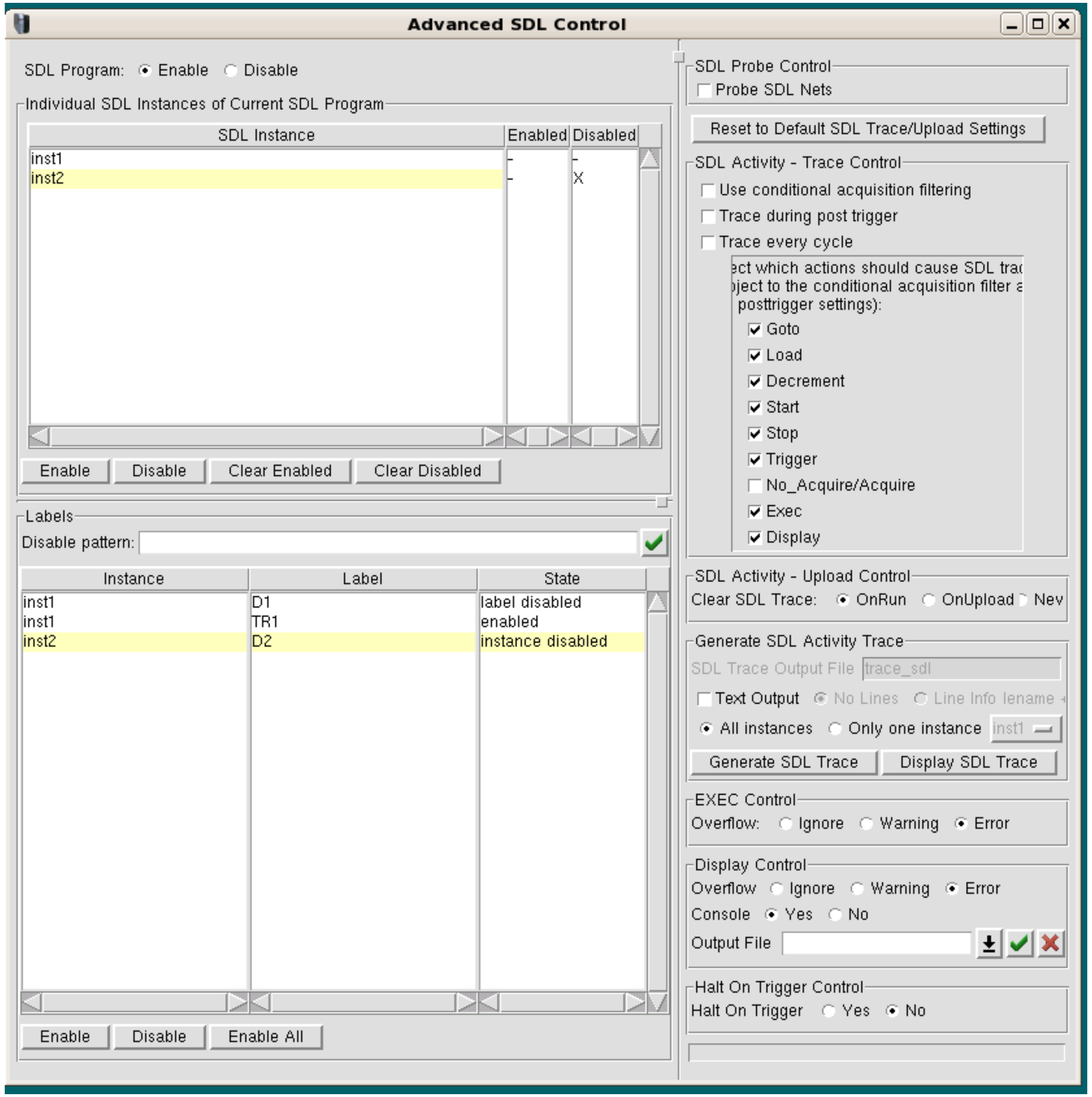


SDL Editor helps you create your state machine

  Cadence Confidential

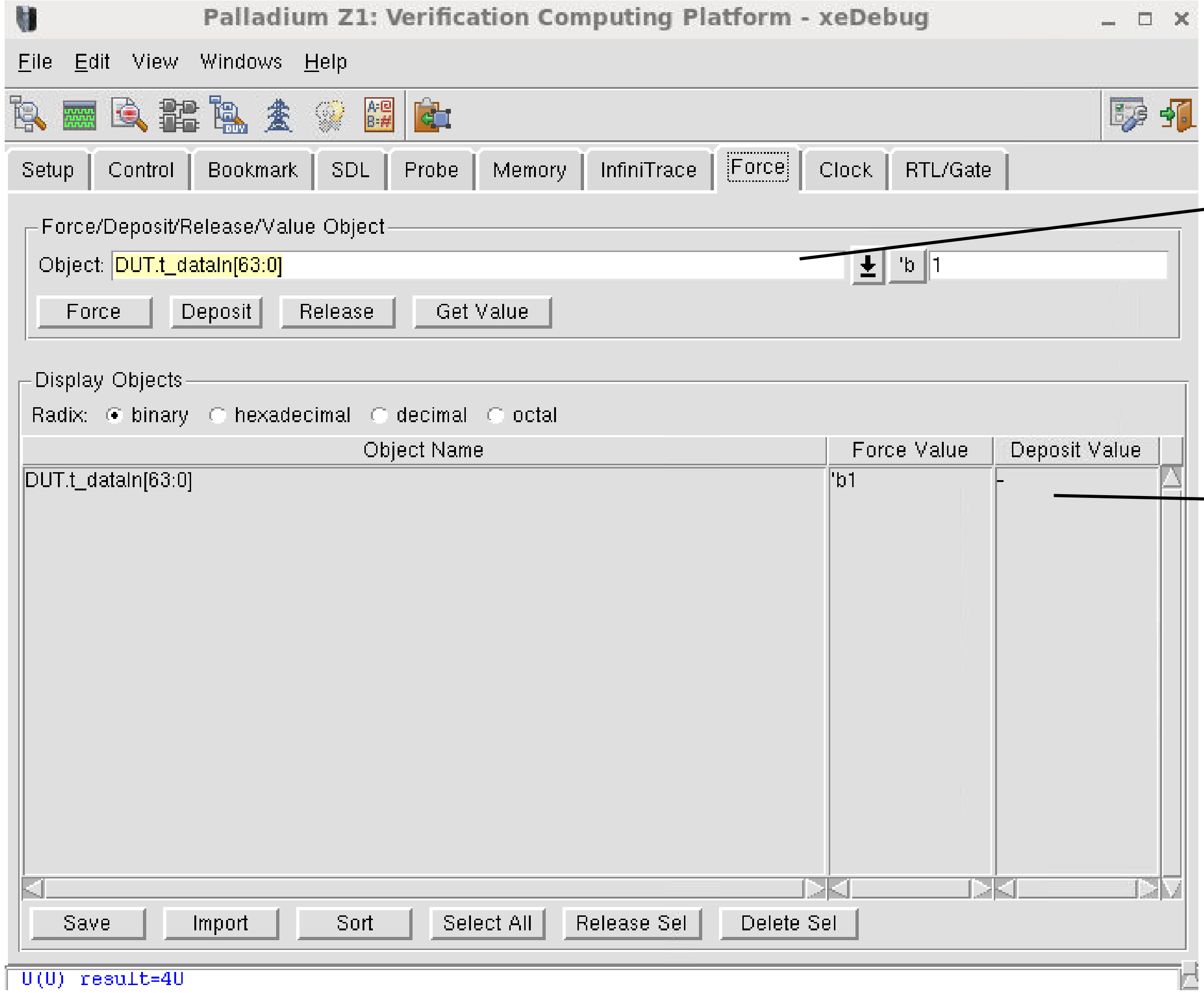
cadence

Advanced SDL Control



Force Tab

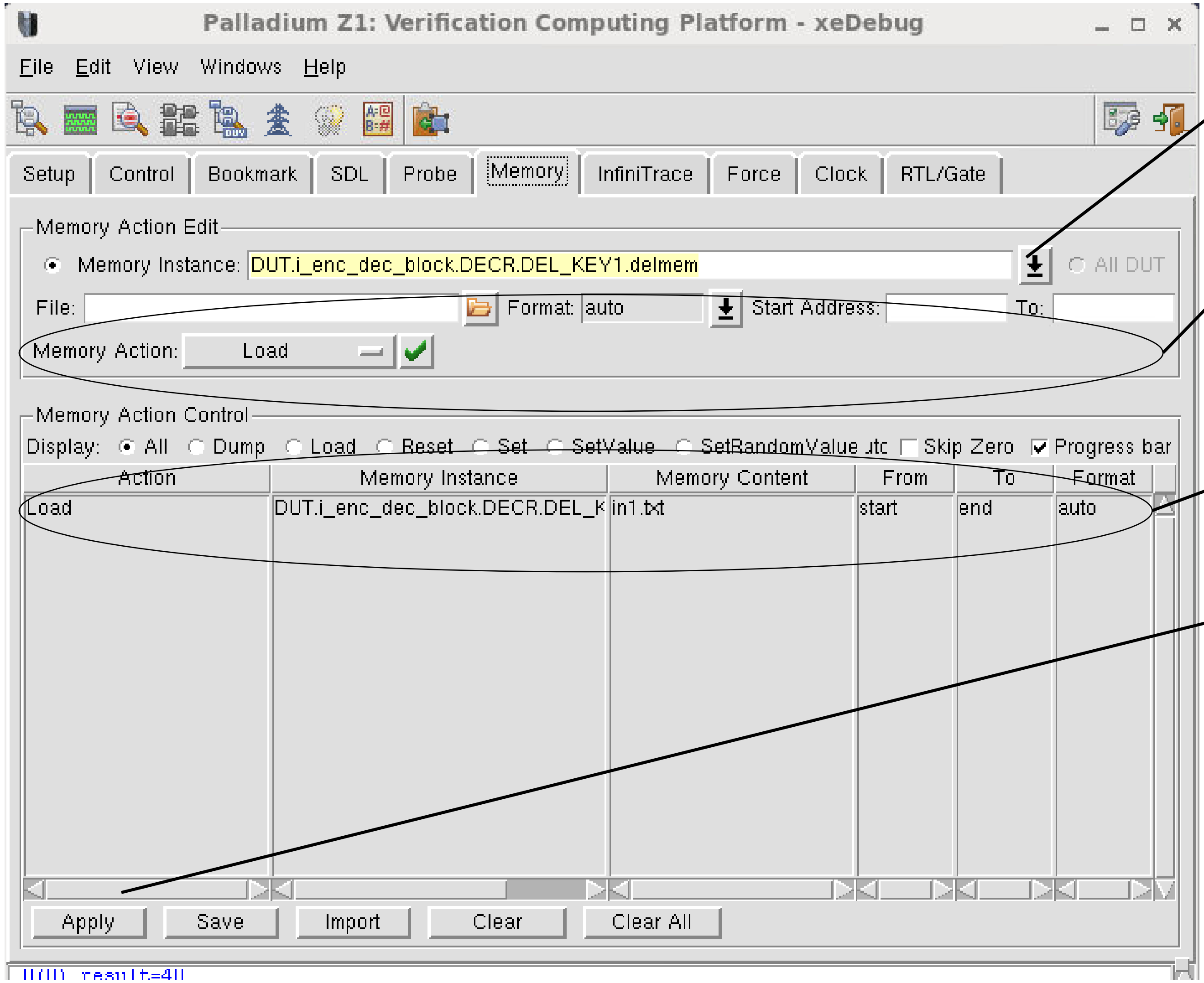
Force Tab can help you keep track of forced signals



Deposit, Force, or Release a signal, bus, or symbol

Status display

Memory Tab



Pulldown Lists all memories

Define an Action
e.g. load instance i1 from file f1

List of Actions you've defined.

To perform an action, select it, and then press Apply

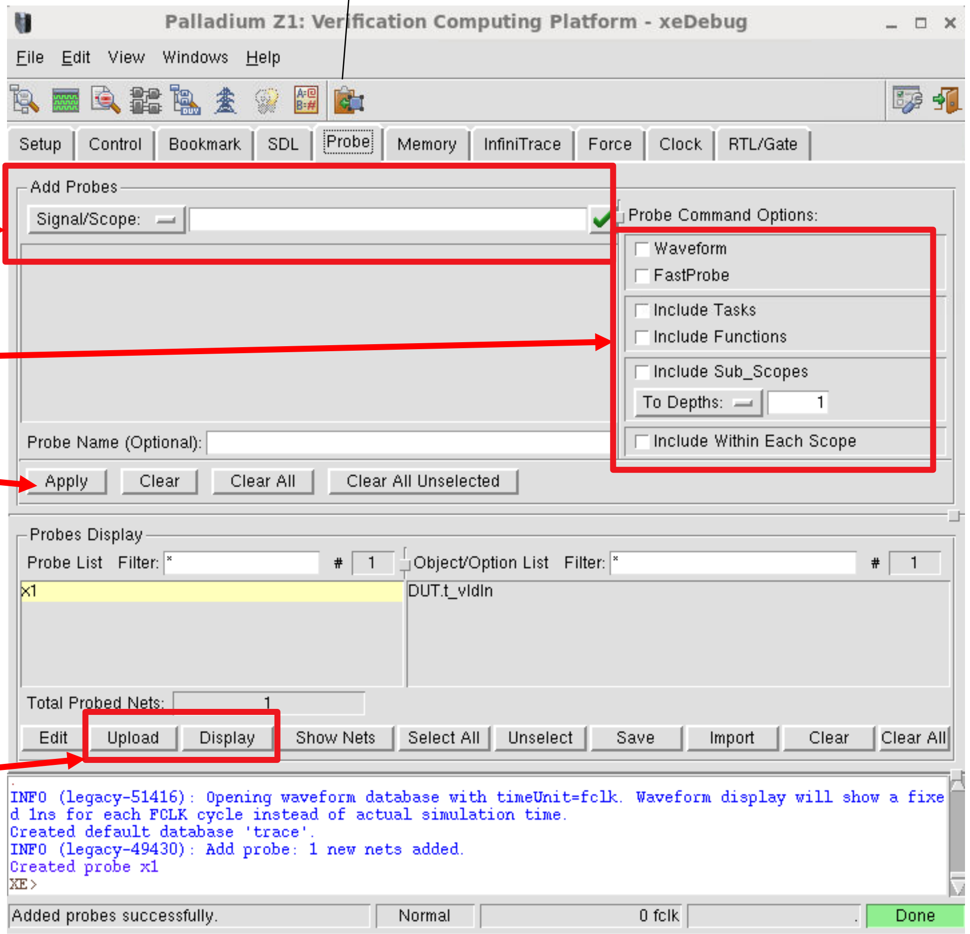
Probe Tab

TIP: you can use this "Get Object" button to bring in items you've selected in browser (next slide)

To probe signals,

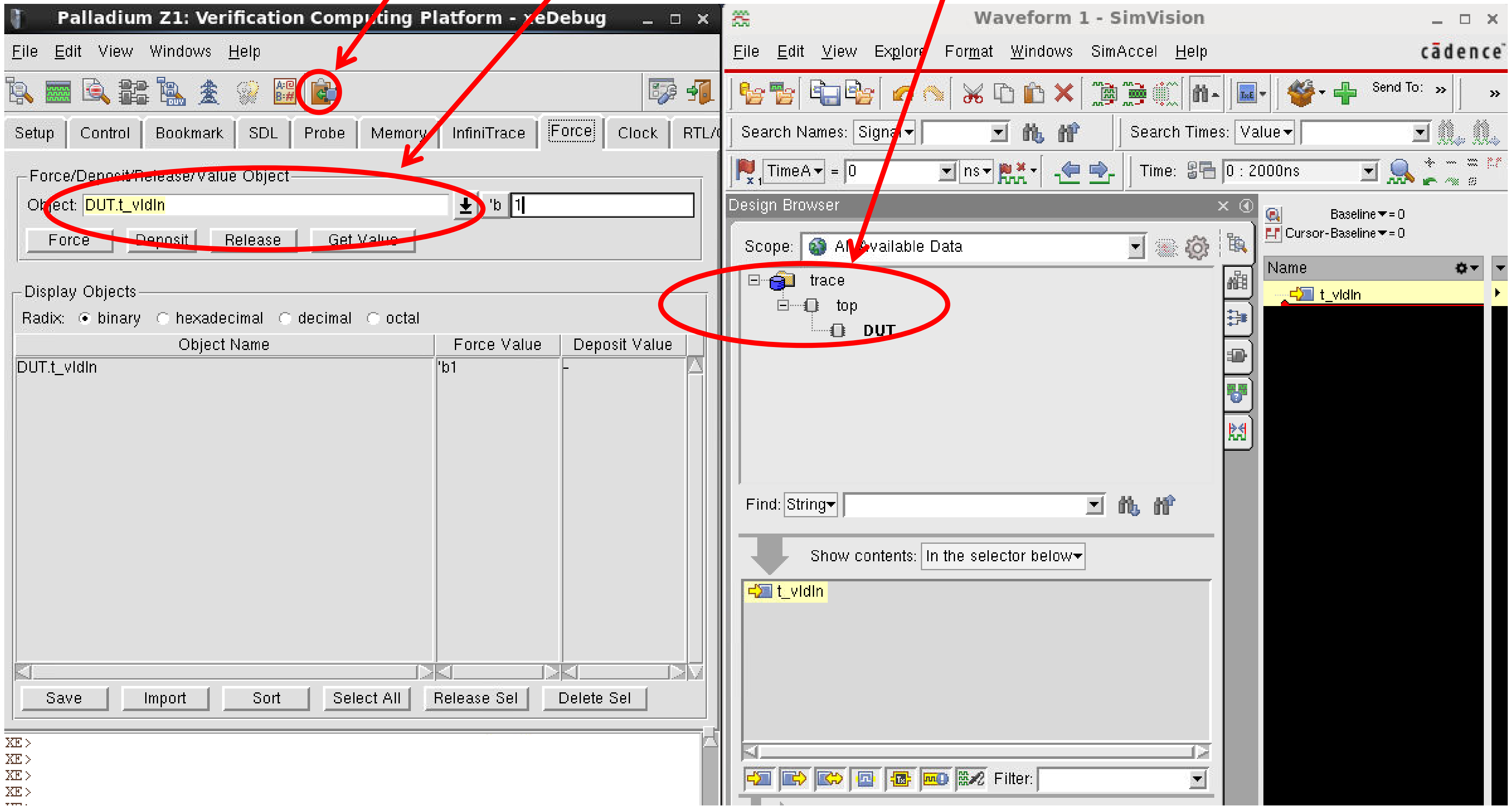
1) Enter one or more names
2) Select option:
   signal, scope, cone or path
3) Press Apply

To upload and display waveforms, use Upload / Display buttons



Using the Get Object Button

1. Select a net or instance in Design Browser or Waveform Viewer
2. Place cursor in a xeDebug text entry window , left click
3. Click Get Object



  Cadence Confidential

cadence

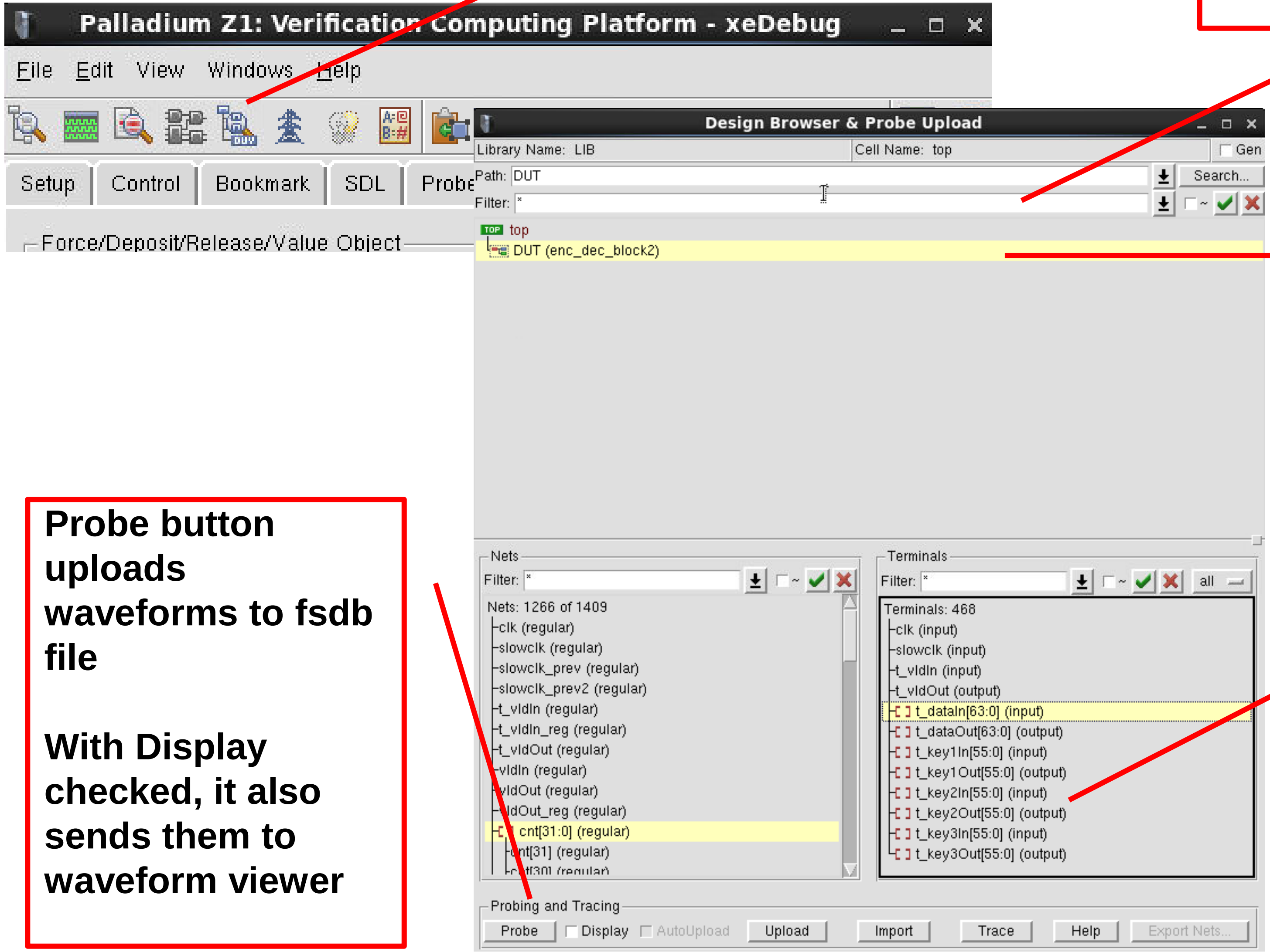
DUV Browser

Shows hierarchy and signals, but not RTL

Invoke DUV browser (fifth button)

Filter (positive or negative)

Hierarchy browser



Probe button uploads waveforms to fsdb file

With Display checked, it also sends them to waveform viewer

Trace traces connectivity in gate-level netlist

Saving and Reloading Debug Settings

- **Following forms allow data to be saved and restored:**
  - Clock
  - Set/Force
  - SDL/Events
  - Symbol & Value
  - Probe Data Upload
  - Memory
  - Design Browser
- **In all the above forms, data can be saved using the Save button and the saved file can be imported in later sessions using the Import button**

Outline

Regression Mode: Highlights

- xeDebug sometimes consume long initialization time
  - Opening the Palladium database impacts time and memory. In certain cases, only a simple validation of the design is required, and only if the validation fails, a full-scale debugging is used.
- What is Regression Mode
  - User can skip loading the database and instead, load a smaller version of the database while opening a debug session in xeDebug. This mode is referred to as the regression mode.
  - Can be used for any design in any compilation (PPC, non-PPC, regular or modular compile flow)
  - Provides fast and light validation ability
  - lower memory footprints
- Enabling Regression Mode in xeDebug mode
  - debug . -regression
  - Or, xeDebug -regression <other options>
- Enabling Regression Mode in xrun mode
  - setenv XE_DEBUG_ARGS “. -re”
  - Or, env XE_DEBUG_ARGS=“. -re”

Regression Mode: Supported Features

- Supported
  - InfiniTrace
  - Save/Restart
  - Memory commands
  - CAKE 1x, 2x
  - xeset commands
  - suspend/resume
  - Offline mode
- Supported with keepNet
  - DRTL
  - Probe -stream
  - Force/Release
  - Value command
  - SDL

Regression Mode: Restrictions

- Not supported
  - DPA/UPF
  - database -upload
  - host -dynp
  - probe commands (in FV or partial FV or DYNP)
  - Wild cards with streaming probes
  - ATB constructs
  - Suspend in regression mode and resume in normal mode or vice versa

- For a complete list of supported and not supported run-time commands, please refer to *Table 39-2 Support and Restrictions for Run-Time Commands in Regression Mode* in VXE User Guide.

Outline

Overview

Modes of Operation

Run-time Logistics

Getting Waveforms

Viewing Waveforms in SimVision

SDL Language to Control Triggering

DRTL

Infinitrace

QEL commands for Debug

xeDebug GUI

Regression Mode

Log Files

cadence

IXCOM Log Files

- **ixcom.log, xe.msg: shows results of compilation**
- **xrun.log, xeDebug.log, xe.msg: show basic runtime statistics and the results of the run**
- **Assertion results can be viewed in Incisive Metrics Center (IMC)**

Debugging Runtime Performance

Basic runtime statistics are printed in xrun.log or xeDebug.log after each run

--- HW execs 12212 tbcall syncs

➔ The number includes user-written tbcalls, DPI import function/task calls,

➔ DPI export function/task ends, imported testbench function/task calls.

➔ exported DUT function/task ends.

--- HW execs 2470 memory read 8878 write

➔ HW memory read/writes, could be memories instrumented for scemi pipes

--- HW execs 268490363 evals 22034 bevals in 474.58 sec (1.77us/eval; 565746.24 eval/sec)

➔ HW evals, usually number of fastest DUT clock edges

➔ bevals, behavioral evalsdate

--- HW execs 34470 force/deposit 0 input events

➔ User force/deposits or dpi task arguments. For values to be transferred from SW side

➔ into the hardware

0 input events:

➔ number of DUT input changes from IUS to HW,

➔counts individual bits of a bus separately

--- HW execs 32303 output events

➔ number of DUT output changes from HW to IUS,

➔ counts vectors as one

--- Accessed Memory 11348 times, transferred 29300112 bytes in 1.177sec (24.9MB/sec)

➔ user or instrumented memory access

IXCOM Profiling

Basic Performance Numbers – xrun.log

```
--- HW execs 20 tbcall syncs
--- HW execs 21 evals 0 bevals(0 gfifo) 21 cfclks 21 tbsyncs
--- HW execs 0 force/deposit 20 input events
--- HW execs 60 output events
--- HW execs emulator command line session time 28.94 sec (28.44 CPU sec)
--- xc Profile: (%)
            64.76        emu (Elapsed: 18.74 sec; CPU: 18.42 sec)
            29.58        IUS (Elapsed: 8.56 sec; CPU: 8.41 sec)
             4.29     decode (Elapsed: 1.24 sec; CPU: 1.22 sec)
             1.37 xc_runtime (Elapsed: 0.40 sec; CPU: 0.39 sec)
```

- tbcall syncs: number of HW-SW syncs due to TBCALLs
- evals: number of evaluations
- bevals: number of behavioral evaluations;
  - one evaluation may include multiple bevals
  - if this number is high compared to evals, need investigation
- force/deposit/input/output events

cadence

IXCOM Profiling

Basic Performance Numbers – xrun.log

```
--- HW execs 20 tbcall syncs
--- HW execs 21 evals 0 bevals(0 gfifo) 21 cfclks 21 tbsyncs
--- HW execs 0 force/deposit 20 input events
--- HW execs 60 output events
--- HW execs emulator command line session time 28.94 sec (28.44 CPU sec)
--- xc Profile: (%)
            64.76        emu (Elapsed: 18.74 sec; CPU: 18.42 sec)
            29.58        IUS (Elapsed: 8.56 sec; CPU: 8.41 sec)
             4.29     decode (Elapsed: 1.24 sec; CPU: 1.22 sec)
             1.37 xc_runtime (Elapsed: 0.40 sec; CPU: 0.39 sec)
```

- emu: time spent by Palladium for execution of DUT
- IUS: time spent by IES for simulation of TB
- decode: time used by IXCOM runtime sending output to TB
- xc_runtime: time used by IXCOM runtime for TB-DUT sync
- deposit: time spent in force/deposit/DPI task arguments
- mem: time spent for memory read/write

cādence

IXCOM Profiling

IES SW Simulation Profile – NCPROF.OUT

```
-----------------------------------------------------------
Stream Counts (37 hits total)
-----------------------------------------------------------
%hits #hits  #inst  name
 37.8    14 [     ] tcl_functions
 27.0    10 [     ] IXCOM (Emulator + Overhead)
 21.6     8 [    1] System Task Enable (file: ./xc_work/v/1n.sv, line: 68
in xc_ncwork.tb [module])
  5.4     2 [     ] outside engine
  5.4     2 [     ] /vobs/ua/tools.lnx86/uxe/lib/64bit/libxcrt.so
  2.7     1 [    2] Task set_count (file: ./xc_work/v/3n.sv, line: 212 in
xc_ncwork.dut [module])
```

- time spent used by tcl_functions, xc_top.v, libxcrt.so are caused by Palladium or IXCOM runtime
- outside engine: mostly for execution of C code (e.g. import DPI C code)

Runtime Profiling

- Runtime profiling option: irun +xcprof or   xeDebug --ncsim +xcprof --
- Generates detailed profiling report with default name: xcprof.out
- Option to modify location & name of output file: +xcprofoutput=<filename>
- Profiled time does not include download time or swap to/from HW time
- Overhead from profiling should not exceed 1-2%
- Report divided into sections
  - IXCOM Performance and Design Profile – operating speed, elapsed run time, synch info, xc Profile table with breakdown
  - Detailed statistics categories: reported only if their count in run is non-zero

Behavioral evals statistics	# of behavioral evaluations done per module.
TB/DPI/SVA/SysTask calls	# of TBCalls, DPI-Calls, SVA & System Task calls & line/file info for these calls
Output event statistics	#of events on the DUT outputs
Input event statistics	#of events on the DUT Inputs
Force Release statistics	# of force-release on DUT signals during HW run
Deposit statistics	# of deposits on DUT signals during HW run
Memory Read statistics	# of memory read transactions issued to HW
Memory Write statistics	# of memory write transactions issued to HW

xcprof.out

```
        Log started on host: hsv-sc23 at: Wed Jan 20 18:36:00 2016
libxcrt - VXE, V15.1.1.7 (compiled with IES-15.10-s002, loaded with  IES-14.20-s004)
--- DPI Profiling Sample count set to default 100. Use option +dpiprofsamplecnt=<N> to change.
--- xc status: @ sim-time = 0 FS, IN SOFTWARE MODE going to be IN RUN MODE, xt0, zt0.
--- xc status: @ sim-time = 0 FS, IN RUN MODE, xt0, zt0.
--- xc status: @ sim-time = 8211920 NS (End-Of-Simulation), IN RUN MODE, xt0, zt0.

======================================================================
IXCOM Performance and Design Profile
----------------------------------------------------------------------
--- Maximum HW operating speed (compiler fclk freq): 3144.00 KHz
--- Clocking Mode: Default (2X).
--- HW execution command line session time: 85.69 sec (43.80 CPU sec)
--- HW execution wall clock time          : 84.28 sec (43.75 CPU sec)
--- HW execution emulator busy time       : 2.61 sec (3.10%)
--- Simulation acceleration speed achieved: 97.43 KHz (97432.18 evals/sec)
--- HW executed
   --    8211921 ECM controlled fclk cycles      @ 97.43 KHz (97432.18 cfclks/sec)
   --    8211921 Eval (or sim-timestep) cycles   @ 97.43 KHz (97432.18 evals/sec)

--- Total number of HW-SW Synchronizations(tbSyncs) : 8211920 (1.00 evals/tbSync)
   -- Synchronizations due to tbcalls(tbcallSyncs) : 0

--- xc Profile: (%)
        83.76     HW-EMU (Elapsed: 70.60 sec; CPU: 36.65 sec)
        14.95     SW-SIM (Elapsed: 12.60 sec; CPU: 6.54 sec)
         1.29     SYNC-OH (Elapsed: 1.09 sec; CPU: 0.56 sec)

     --- HW-EMU : HW evaluations and Synchronization Latency.
     --- SW-SIM : TB, TBCalls, DPICalls, VPI/VHPI/PLI/E/SystemC.
     --- SYNC-OH: Synchronization Overhead of IXCOM Runtime
```

Time spent in HW-SW Synchronization =
# of HW-SW Synchs *
Channel Latency per Synch

% of Total time spent in HW, SW, & Overhead

Hardware-Emulation time has two components:
- time in Synchronization (input, output)
- time in DUT evaluation

Detailed statistics listed by category below this area

cādence

I/O Signal Profiling

Checking I/O Signals from xcprof.out

```
--- Primary Inputs  (bits): INPUT(4)  FORCED(0)
                            DEPOSIT(0 : dut=0 + ixcom=0).
--- Primary Outputs (bits): OUTPUT(4 : dut=4 + dpi=0 + ixcom=0).
```

- same information obtained from ixcom.log (logs from UASA) but contains more detailed information
- DEPOSIT
  - dut: DUT variable marked by $export_deposit
  - ixcom: DEPOSIT created by IXCOM instrumentation
- OUTPUT
  - dut: normal DUT port signals
  - dpi: POs created by IXCOM instrumentation
  - ixcom: TBCALL POs (either user-generated or internally-generated)