

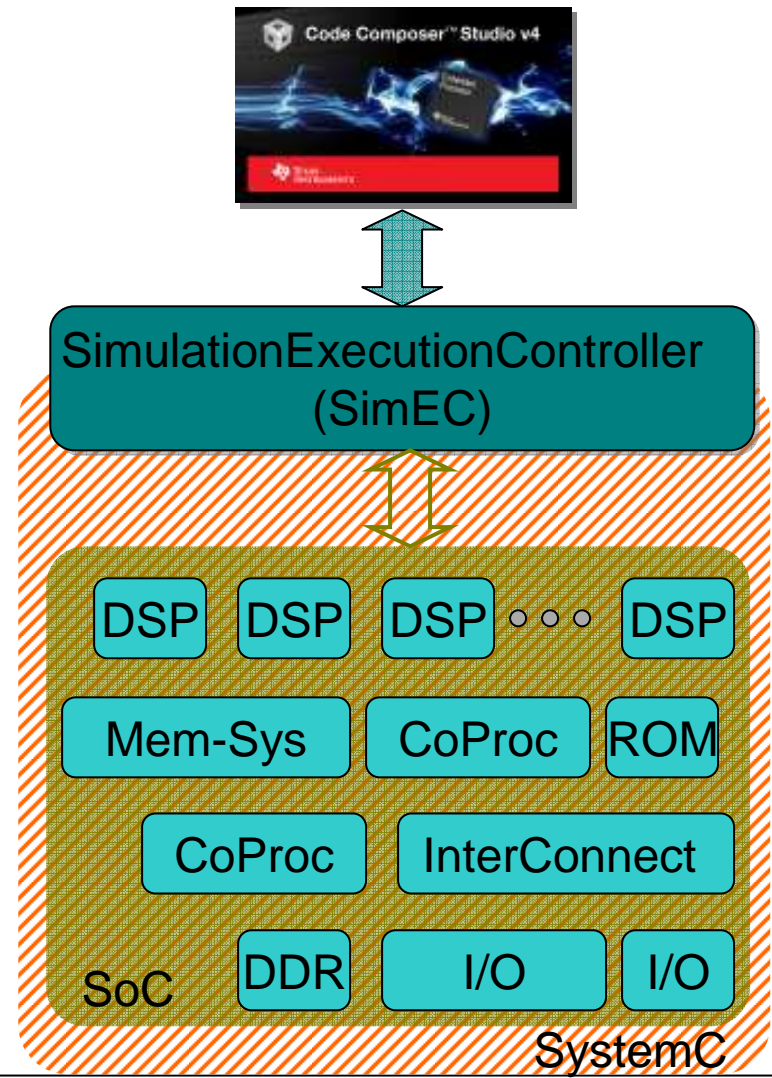
# Integrating Software-debugger on a SystemC-based Virtual-platform

Nizamudheen Ahmed[niz@ti.com]

Gulur Dwarakanath Nagendra[nagendra@ti.com]

# Our Work

- We present an approach to integrate software-debugger over a SystemC-based Virtual platform
- We integrated the **Texas Instrument's Code Composer Studio v4** debugger, over a SystemC based Multi-core SoC Virtual Platform (simulator)
  - The SoC simulator simulated is TI's next-gen high-performance multi-core SoC, targeted at LTE application
  - This SoC contains multiple TI DSP's, 10's of co-processor for LTE acceleration, DMA engines and I/O peripherals.
- Novelty:
  - The Simulation Execution Controller (SimEC) module is the key piece of our work
  - SimEC model design and implementation overcomes a number of challenges in integration a Software Debugger semantics on simulation platform



# Challenges

- The following Debugger semantics are difficult to implement over SystemC based Virtual Platform
  - Asynchronous execution mode – Running software on few DSP core's while the other put halted
    - Background: SystemC's ssc\_start API shall advance the simulation globally, and not localized to one or few DSP cores.
  - Handling breakpoints – Stops simulating all/few core's while one of the running core stops because of an instruction breakpoint
    - Background: SystemC's ssc\_start API will advance the simulation for the specified simulation time-unit. However, there is no mechanism to pre-empt or pause the simulation, before ssc\_start's time-unit completes/expires.
  - Maintaining high simulation speed, while implementing debugger features like run, halt, DSP instruction step, application source-step, breakpoints
    - Background: These features may need simulation to be advanced in fine-grain granularity, but at the cost of simulation speed.

# Introduction to SimEC and Debug API

- The debug API (API that the CCS Studio v4 debugger calls, in response to a user's command from debugger) can be classified into 2 categories
  - Visualization API – API used to view SoC registers/memories (GET\_MEM, PUT\_MEM, GET\_REG, PUT\_REG, SET\_BP, CLR\_BP...).
  - These APIs do not change simulation state, when called
  - Execution Control API – API used to control the simulation execution (SIM\_RUN, SIM\_STEP, SIM\_HALT)
  - These APIs change simulation state, when called
- The SimEC converts the Execution-Control debug API to necessary SystemC API calls. The Visualization API is handled by the individual IP models.
- The CCS Studio v4 provides a debugger-window for each DSP core being simulated.
  - Each debugger command (like run, halt...) may be issued on individual DSP core, or collectively on all, or Selected few cores
  - CCS Studio provides control to execute one assembly instruction, or one C/C++ source code statement or to run the simulation until interrupted (by a breakpoint or a halt user-command)

# User commands and Debug API

Debug API call	Description
<code>void SIM_STEP(core_id)</code>	Execute a single instruction on a given core
<code>void SIM_RUN(core_id)</code>	Execute a fixed quantity of simulation (need not be in units of instructions) on the given core
<code>stat_t SIM_STAT(core_id)</code>	Return the current state of simulation for the specified core. Return the following ■ <b>HALTED</b> (indicating a breakpoint detection or error in simulation) ■ <b>RUNNING</b> (indicating that the last SIM_RUN, SIM_STEP completed successfully).

Table: Summary of all the debug API the debugger calls to SimEC

- Few of the User commands that a user performs on the CCStudio v4 is listed below
  - Load a program into the DSP memory
  - Apply breakpoints at function, say calc
  - Run the target
  - Inspect DSP/Shared memories, MMR and DSP registers when the target halts

User command (related to Execution Control debug API)	Debug API call and expected responses
Assembly single-step on a DSP core	<code>SIM_STEP(core_id)</code> <code>SIM_STAT(core_id): HALTED</code>
Simulation-run on a DSP Core	<code>SIM_RUN(core_id)</code> <code>SIM_STAT(core_id): RUNNING</code> , <code>SIM_STAT(core_id): RUNNING</code> . . . . <code>SIM_STAT(core_id): HALTED [because of a breakpoint]</code>
Simulation run on two (or more) DSP Cores	<code>SIM_RUN(core_x)</code> , <code>SIM_RUN(core_y)</code> <code>SIM_STAT(core_x): RUNNING [both core shall run]</code> <code>SIM_STAT(core_y): RUNNING [both core shall run]</code> . . . . <code>SIM_STAT(core_x): HALTED [breakpoint]</code> <code>SIM_STAT(core_y): RUNNING [core y continues to run]</code>

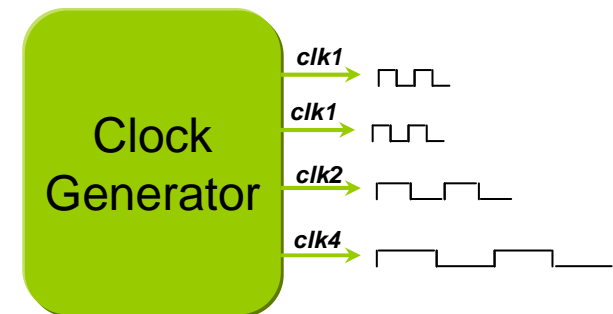
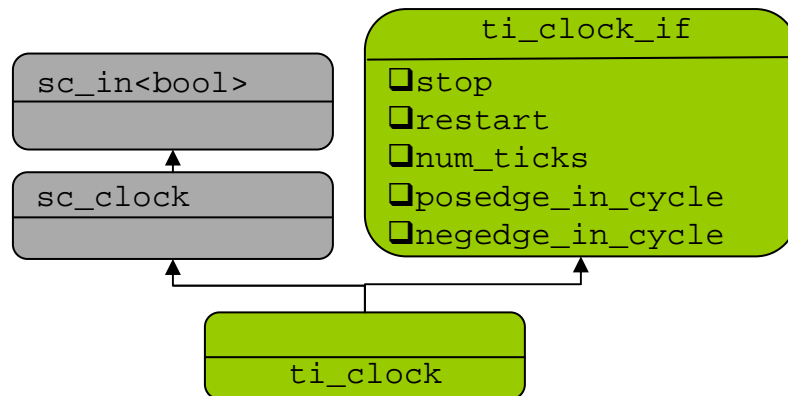
Table: Summary of Debugger commands and corresponding debug API used

# DebugAPI and SimEC semantics

- The Code Composer Studio v4, the SimEC and all the IP models simulated execute in a single OS process. In-process execution
- To keep the designs simple, and to handle scenarios arising for simulating multi-core SoC
  - SimEC maintains a global *run-list*, that is `null` initially
    - The run-list will contain a order-independent list of DSP-cores (IDs) that are ready to run.
  - `SIM_RUN` debugAPI call add its entry to the run-list. Simulation is not executed in this API call!
  - `SIM_HALT` debugAPI call removes its entry from the run-list
  - `SIM_STAT` shall execute simulation on all the DSP core in the run-list and return the `stat_t` of the requested DSP ID
  - This design of executing the simulation in `SIM_STAT` as against in `SIM_RUN` scales well in the multi-core SoC.
- Up Next: The key pieces in the SimEC module is the master clock-generator and the defined clocking-scheme that are discussed in the following slides

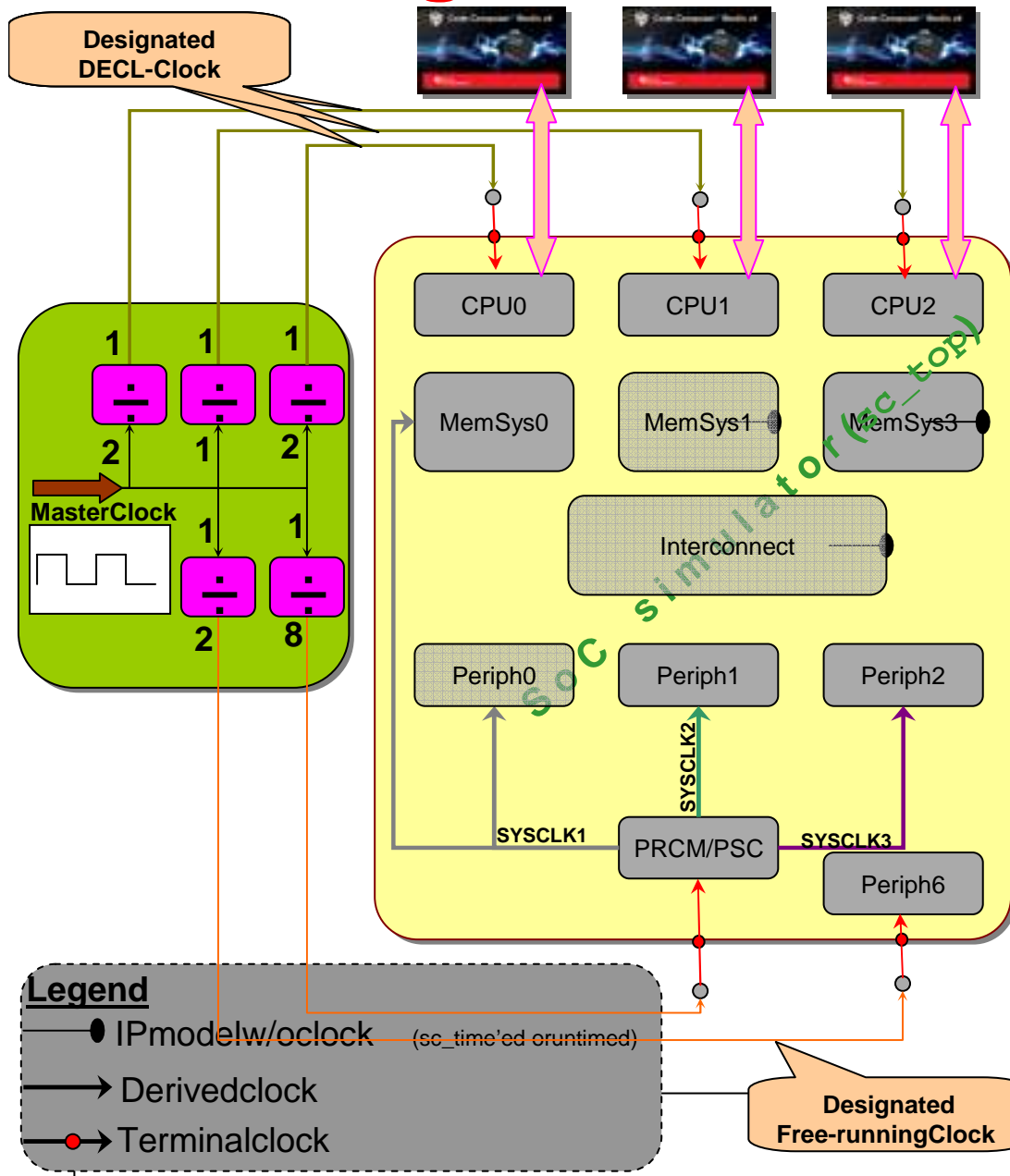
# MasterClockgenerator

- TheMasterclock-generator'skeyfeaturesare
  - Generateoneormoreclocks,basedonconfigured frequencyandduty-cycle
  - Generatedclocksare'gate-able' [ clock->stop(), clock->restart()]
- Classhierarchy



- Theclock-generatorgeneratesti\_clock,whichisbackward-compatiblewithsc\_clock.
- The `posedge_in_cycle` methodcanbeusedtoscheduleaclock-tickafter 'n' cycles(`n==1`isavailableto `posedge_event`, in `sc_in<bool>`).
- The `stop` and `restart` methodsareusedtostopandresumetheclock-ticks – Gatedclock implementation.Thisfacilitatestostop-clockingoneormoremodules/Coreswhileothercontinuetorun.
  - Ex:Allthethemodulesclockedusingclk2maybegated(`clk2->stop()`),whilethethemodulesclockedusingclk1andclk4cancontinuetorun,astheSystemEngineadvances

# Clockingscheme



- A given model in the simulation may contain one or more clocks.
- These clocks can be categorized as
  - Terminal clock
    - Clocks that are regenerated outside SoC simulator topology (by Clock-generator)
    - Terminal clocks are configured at init-time and shall not change thereon.
  - Derived clock
    - Clocks that are regenerated within SoC that may be derived from one or more terminal clocks
- A terminal clock may be designated as
  - **Free-running clock** – Clocks that are never stopped
  - **Debugger Execution Control Linked Clock (DECL)** – Clocks whose stop/restart are controlled by debugger commands (run, halt, step)

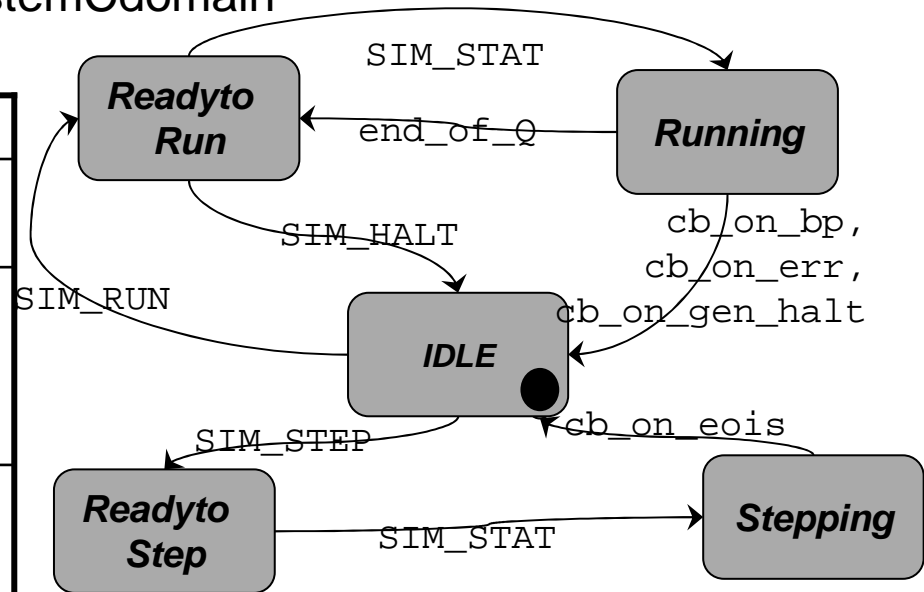
# SimECImplementation(1)

- This slide and the next give the implementation details of SimEC on converting debug API calls to SystemC calls.
- At Initialization
  - After constructing the `sc_top`, the SimEC collates all the DECL clocks in the simulation and associates each with a debugger
    - The SimEC is provided with a meta-data file (`.cfg`) to help make the set {Debugger ID, DSP id, {set of DECL clocks}}
    - Debugger ID to DSP ID holds a 1-1 mapping
    - DSP ID to DECL clock holds a 1-many mapping
    - Note: A DECL Clock should not be associated with more than one DSP ID
  - The SimEC sets up each DSP Core with few callback functions that are used to notify (to SimEC) the following
    - End of an instruction step
    - Instruction Breakpoint detection
    - Error in simulation
    - General halt
- At Runtime
  - The SimEC determines (once) the execution quantum – amount of SystemC simulation time to be advanced in call of `SIM_STAT`

# SimECImplementation(2)

- The SimEC state-transition diagram is depicted below. This state-transition diagram consolidates the mapping of debug API (transitions) to various activities carried-out in the SystemC domain

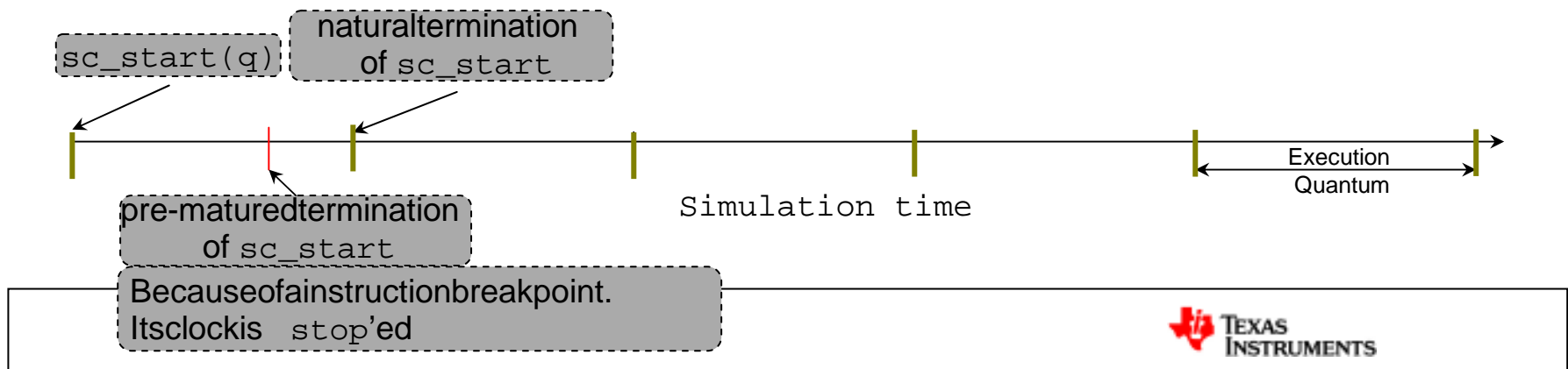
State	Description
IDLE	Initialization state. In this state, all DECL clocks are stop'ed.
ReadytoRun	In this state, the corresponding DECL clocks are restart'ed. The simulation is ready to go
ReadytoStep	
Running, Stepping	In this state, the sc_start is called with the pre-determined execution quantum



Transitions	Description
SIM_RUN, SIM_STEP, SIM_HALT, SIM_STAT	Debug API, from debugger
end_of_q	sc_start of a fixed quantum has completed
cb_on_bp, cb_on_err, cb_on_gen_halt, cb_on_eois	callback from the DSP core indicating detection of a breakpoint, error, general-halt and end-of-an-instruction respectively. This calls for a pre-mature termination of sc_start call

# About Execution Quantum

- The execution-quantum needs to be carefully chosen
  - Too less a execution quantum may lead to frequent sequence that may lead to simulation overhead
    - call of `SIM_RUN-SIM_STAT-sc_start`
  - Too high a execution quantum may lead to less-frequent call of these sequence, however, the user's feel of the debugger interactivity may worsen.
  - Care should be taken to (manually) choose a quantum that takes care of the concerns listed above.
- Handling Breakpoints/error
  - User is allowed to place breakpoints on the target -coder running on the DSP simulator model. The DSP model should stop simulation and report it to user on detecting an instruction breakpoint
  - Since SystemC does not allow pre-mature termination of clocks are `stop`'ed when the DSP core makes a call-back to pause simulation, prematurely (When it detects an instruction breakpoint).
    - This ensures the DSP is clock-gated until the user takes appropriate action when these second condition occurs.
  - However, other modules that are either 'free-running' clock'ed or not clock-gated continue to run until the `sc_start` expires.



# Results

- We were able to simulate multi-core TIDSP on Code Composer Studio v4 with LTE Layer 1, 2 and 3 coprocessors
- All the debugger semantics/commands are supported with no limitation on the expected behaviors
- We were able to simulate the entire SoC simulation at 250 KIPS on a LTE application that runs on all the DSP cores and few coprocessors.
  - The SoC simulator we simulated models fair-level of timing and models all the architecture pipeline in details.
- SimEC is implemented as an individual component that is SoC-independent and can integrate any SoC/Processor.
- We are in process of reusing the SimEC in a different TI SoC that contains few TIDSPs and ARM cores.