Automatic Generation of Implementation Layer for Embedded System using PSS and SystemRDL

Nikita Gulliya
Sudhir Bisht

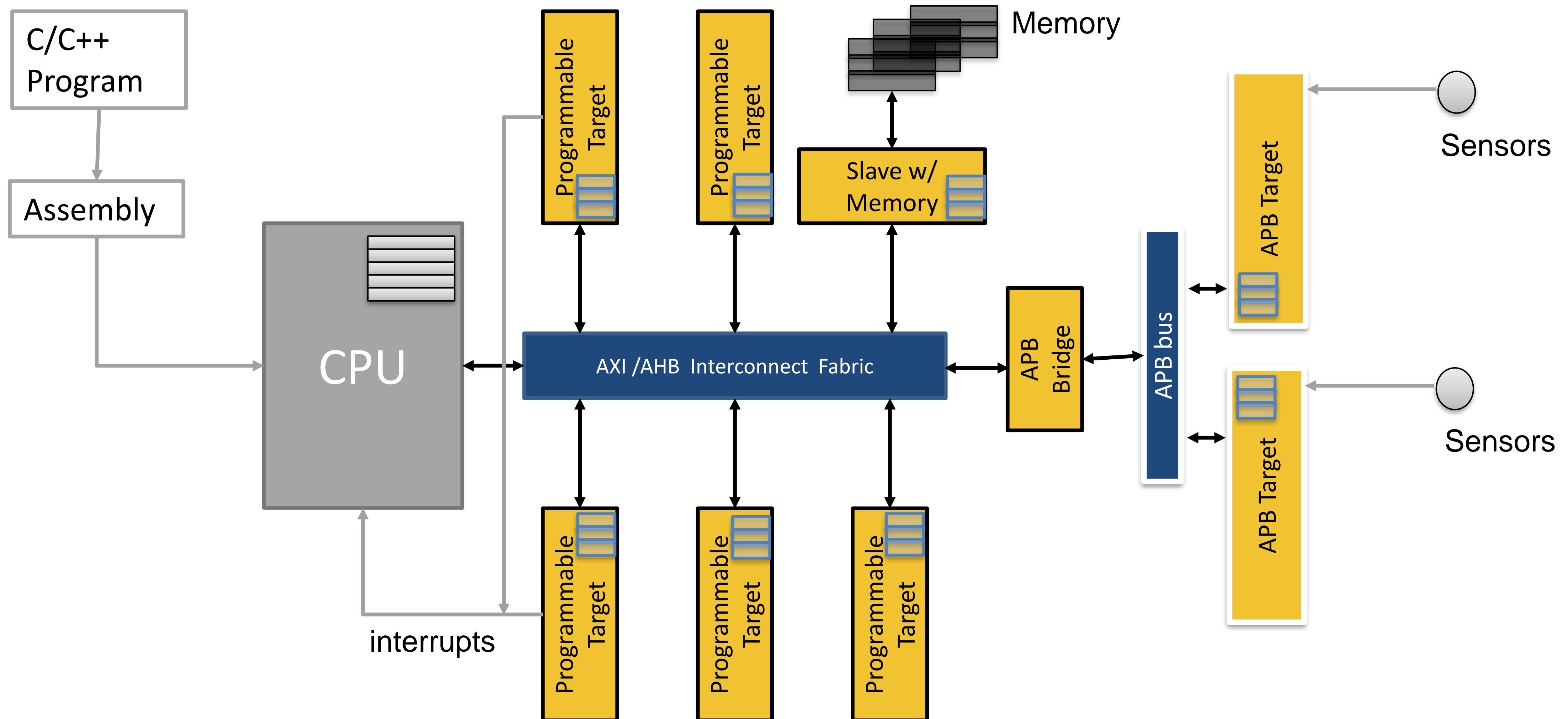
AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

accellera
SYSTEMS INITIATIVE

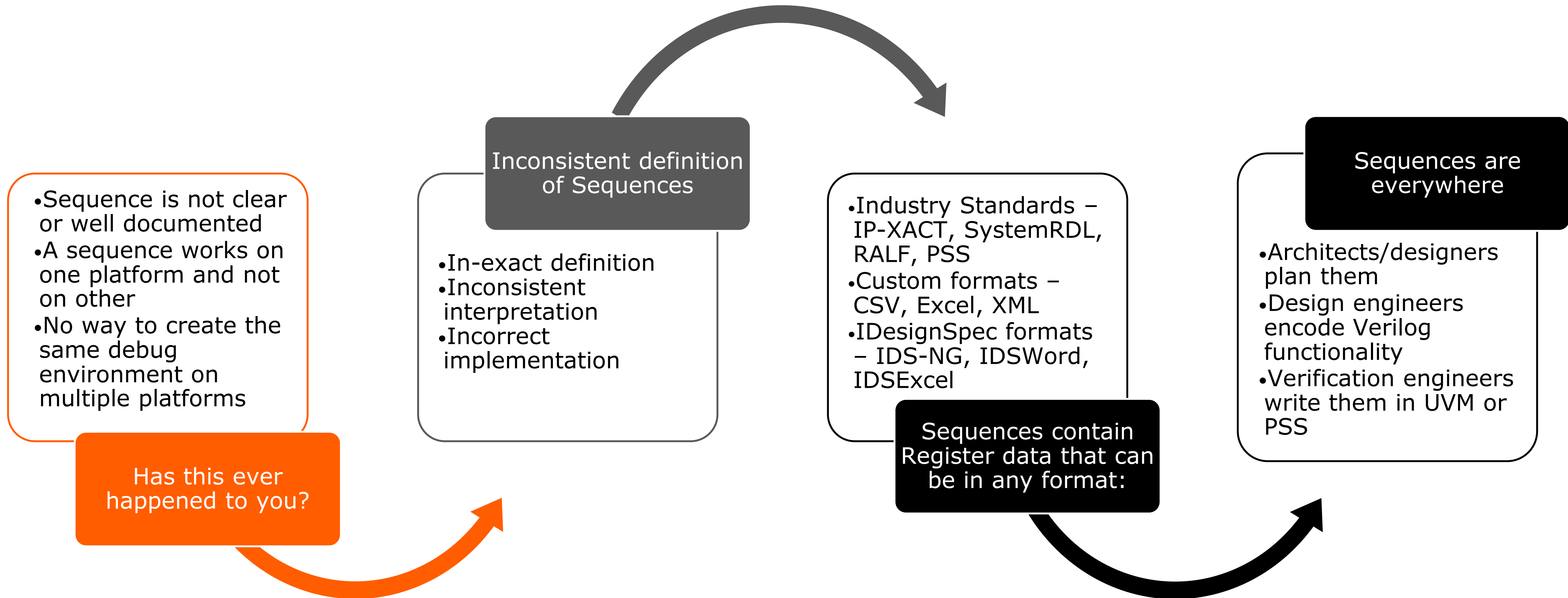
Typical Chip Design

- Hardware of the SoC is designed by HW team
  - But used by
    - Verification/Emulation team
    - Firmware team
    - Validation team
    - Software team
- How does the software interact with the IPs?
  - Through the Hardware Software Interface (HSI)
- Hardware is at the core and software API is around it
- Device drivers (part of the HSI) are tedious to create
  - They are written in C and Assembly

Introduction to a Typical SoC



Challenges Development Teams face with Sequences



Challenges Faced

- SOC design companies
  - Increasing demands of design complexity and design performance
  - Combining automation with flexibility to accommodate changes in sub-systems across applications
  - Driving down the cost of design for a better ROI
  - Shrinking market windows
  - Boosting productivity of design teams to meet shorter market windows
  - Requirement for HW/SW co-simulation to catch the bugs from the early design stage.
  - There is a lack of common set of sequences which can be shared across the teams.

An Ideal Solution

- PSS and SystemRDL will help design teams to generate unified test and programming sequences in UVM and Firmware from the specification.
- The register information can be in standard format like PSS/SystemRDL.
- Users can define the test sequences in PSS (or Excel, Python GUI (IDS-NG)), and then generate the unified test sequences from verification to validation.
- The tests generated are UVM sequences for simulation and firmware sequences for HW/SW co-simulation and post silicon validation:
    - start-up sequence, read-write operation shutdown sequence, low power mode sequence etc.

Sequences

- Sequences are a "set of steps" that involve writing/reading specific bit fields of the registers in the IP/SoC.
- These sequences can be simple, or complex involving conditional expressions, array of registers, loops, etc.
- PSS users can write a single sequence specification and a compiler has been written to generate the UVM sequences for verification, System Verilog sequences for validation, C code for firmware & Device driver development and various output formats for Automatic Test Equipment.
- Sequences can achieve a certain functionality such as:
  - Digital Programming Sequences
  - Analog and Mixed Signal Sequences
    - Power UP
    - Low Power mode
  - Functional Sequences
  - Test Sequences
  - Types
    - Simple
    - Hierarchical

Portable Stimulus Standard

- The Portable Test and Stimulus Standard defines a specification for creating a single representation of stimulus and test scenarios
- With this standard, users can specify a set of behaviors once, from which multiple implementations may be derived.
- A model consists of two types of class definitions:
  - elements of behavior, called **actions**;
  - passive entities used by actions, such as resources, states, and data flow items, collectively called **objects**.
- The behaviors associated with an action are specified as **activities**.
- All of these elements may also be encapsulated and extended in a package to allow for additional reuse and customization.

Portable Sequences and Golden Specification

- Sequences can be captured in PSS, python, spreadsheet format, or GUI(NG) and generate multiple output formats for a variety of domains:
  - UVM sequences for verification
  - SystemVerilog sequences for validation
  - C code for firmware and device driver development
  - Specialized formats for automated test equipment (ATE)
  - Hooks to the latest Portable Stimulus Standard (PSS)
  - Documentation outputs such as HTML and flowchart

Machine Power Controller

• The power controller is a discrete output device that regulates the system with guidance from the temperature controller.
• Here are some common registers and fields that may be used in a machine power controller:

1. **Power Control Register:** This register contains several fields that control the power supply, including:
    a. Power Supply Enable: This field controls whether the power supply is turned on or off.
    b. Voltage Control: This field adjusts the voltage level of the power supply.
    c. Current Limit: This field sets the maximum current that the power supply can deliver.

2. **Status Register:** This register contains fields that indicate the current status of the power supply, including:
    a. Power Supply Status: This field indicates whether the power supply is currently turned on or off.
    b. Overvoltage Status: This field indicates whether the input voltage is above the allowed range.
    c. Overcurrent Status: This field indicates whether the output current is above the allowed range.

Machine Power Controller contd..

3. Interrupt/Req Register: This register contains fields that control interrupt behavior, including:

a. Interrupt Enable: This field controls whether the system generates interrupts when certain events occur, such as overvoltage or overcurrent conditions.

b. Interrupt Status: This field indicates whether an interrupt has been generated.

4. Fault/ack Register: This register contains fields that indicate the occurrence of faults, including:

a. Overvoltage Fault: This field indicates whether the input voltage has exceeded the maximum allowed level.

b. Overcurrent Fault: This field indicates whether the output current has exceeded the maximum allowed level.

c. Temperature Fault: This field indicates whether the power supply has overheated.

PSS Register Model

- Block/register group will be defined as *component* and registers are defined as *package* and *struct* containing information of fields inside the register for a power controller.

```
package  elevator_regs_pkg {
    import addr_reg_pkg::*;
    struct UNITCONTROLLER_reg_s  : packed_s<> {
        bit[1] UPWARD_CONTROLLING_UNIT;
        bit[1] rsvd_0;
        bit[1] DOWNWARD_CONTROLLIG_UNIT;
        bit[1] rsvd_1;
        bit[1] OPEN_FLOOR;
        bit[1] rsvd_2;
        bit[1] CURRENT_FLOOR;
        bit[25] rsvd_3;
    };
```

```
struct REQUESTRESOLVER_reg_s  : packed_s<> {
       bit[1] ENTRY_PUSH_BUTTON;
       bit[1] EXIT_PUSH_BUTTON;
       bit[6] rsvd_0;
       bit[1] EMERGENCY_BUTTON;
       bit[1] UP_PUSH_BUTTON1;
       bit[2] rsvd_1;
       bit[1] UP_PUSH_BUTTON2;
       bit[2] rsvd_2;
       bit[1] DOWN_PUSH_BUTTON1;
       bit[8] rsvd_3;
       bit[1] DOWN_PUSH_BUTTON2;
       bit[7] rsvd_4;
   };
.       .       .       .       .
.       .
```

SystemRDL instead of intrinsic PSS

- SystemRDL is a language for the design and delivery of intellectual property (IP) products used in designs.
- Its semantics supports the entire life-cycle of registers from specification, model generation, and design verification to maintenance and documentation.
- Registers are not just limited to traditional configuration registers, but can also refer to register arrays and memories.

```
addrmap power_top {
addrmap elevator {
    reg UNIT_CONTROLLER {
       regwidth = 32;
      field {
        hw = r;
        sw = rw;
      } UPWARD_CONTROLLING_UNIT[0:0] = 1'h0;
      field {
        hw = r;
        sw = rw;
      } DOWNWARD_CONTROLLIG_UNIT[2:2] = 1'h0;
      field {
        hw = r;
        sw = rw;
      } OPEN_FLOOR[4:4] = 1'h1;
      field {
        hw = r;
        sw = rw;
      } CURRENT_FLOOR[6:6] = 1'h0;
    };
```

Advantages of SystemRDL

- Defining register specification in SystemRDL has an advantage over PSS in case special registers such as interrupt and counters are used.

- SystemRDL allows to define parameters within the specification, which can help to create more flexible and reusable designs.

- SystemRDL also includes a Perl preprocessor, which allows including Perl code within specification.

- This can be used to automate certain tasks, such as generating a list of registers or automatically calculating values based on other parameters.

Connecting the Register Model

- In PSS, *component* keyword stands for the block which was defined above in the register specification which contains the sequence specification.

- *extend* keyword contains the sequences it imports and instantiates the register specification, it further contains multiple *action* for each register which may contain an activity.

- A number of actions and their relative scheduling constraints is used to specify the verification intent for a given model and finally *exec blocks*, which contains *pre_solve* and *post_solve exec block*.
  - Statements in pre_solve blocks can read the values of non-random attribute fields and their non-random children.
  - Statements in post_solve blocks are evaluated after the solver has resolved values for random attribute fields and are used to set the values for non-random attribute fields based on randomly-selected values.

- Exec blocks also contains *body exec block*, it contains the actual functionality of the sequence in terms of loops such as while, conditions such as if-else etc

Connecting the Register Model cntd..

```
component elevator_controller {

extend elevator_controller {
      import elevator_regs_pkg::*;

            UNITCONTROLLER_reg_s regs;

            action display_floor {
               rand int floor_no ;

               UNITCONTROLLER_reg_s uc_reg;

                     exec post_solve {

uc_reg.CURRENT_FLOOR=1;
                       }

                      exec body {
            message("You are now on floor");
            }

          }
```

```
     action request_floor {
                 const int max_floor=10;
                       rand int desti_floor ;

                       UNITCONTROLLER_reg_s uc_reg;

                       exec body {

uc_reg.UNITCONTROLLER.write().OPEN_FLOOR==desti_floor;
                                  if (
uc_reg.UNIT_CONTROLLER.read().OPEN_FLOOR <1 ||
uc_reg.UNIT_CONTROLLER.read().OPEN_FLOOR > max_floor ) {
                                     message(" invalid floor no
.Please try again");

                                 }
        }
          }
.         .         .         .
```

Specification in IDS-NG Format

sequence name	ip	description
request_floor	elevator controller system.idsng	

arguments	value	description

constants	value	description
max_floor	10	

variables	value	description
desti_floor	0	

assign	value	description

command	step	value	description	refpath
call	display("Which floor would you like to go to")			
write	UNIT_CONTROLLER.OPEN_FLOOR	desti_floor		
if (UNIT_CONTROLLER.OPEN_FLOOR <1 || UNIT_CONTROLLER.OPEN_FLOOR > max_floor) {				
call	display(" invalid floor no .Please try again")			
}				

Portable Sequences Features

- Looping
  - For loop
  - while condition
- Condition
  - If - else condition
- Wait statement
- Switch command
- External function call
- Structs to define packets and descriptors (UVM and header)
- Optimized read/write
- Consolidated read/write
- Commenting commands

- Powerful referencing of macro sequences and IP's from any level
- IP's in different input formats (like IP-XACT, System-RDL, RALF, IDSWord, IDSExcel)
- Fork Join
- Assertions
- Concatenation
- Handling interfaces
- Copyright Headers
- Multiple Structures
- Guard Banding
- Randomization and Constraints
- Conditional wait

Results

- PSS compiler and GUI generator has been developed for generation of various outputs from above golden custom sequence specification such as:
  - SystemVerilog/MATLAB output for Validation
  - UVM output for Verification
  - C output for Firmware
  - CSV output for ATE
  - HTML/Flowchart for documentation

C Output

```
#include <stdio.h>
int display_floor( ) {
  int floor_no = 1 ;

  FIELD_WRITE(elevator_UNIT_CONTROLLER_ADDRESS,
floor_no <<
ELEVATOR_UNIT_CONTROLLER_CURRENT_FLOOR_OFFSET,
ELEVATOR_UNIT_CONTROLLER_CURRENT_FLOOR_MASK,ELEVATOR_U
NIT_CONTROLLER_CURRENT_FLOOR_OFFSET);

 // Call firmware print method

  printf("You are now on floor",);
  return 0;

}
int request_floor( ) {

  static const int max_floor = 10 ;
  int  UNIT_CONTROLLER_OPEN_FLOOR;
  int desti_floor = 0 ;

  // Call firmware print method

  printf("Which floor would you like to go to",);
```

```
 FIELD_WRITE(elevator_UNIT_CONTROLLER_ADDRESS, desti_floor <<
ELEVATOR_UNIT_CONTROLLER_OPEN_FLOOR_OFFSET,
ELEVATOR_UNIT_CONTROLLER_OPEN_FLOOR_MASK,ELEVATOR_UNIT_CONTROLLE
R_OPEN_FLOOR_OFFSET);

  UNIT_CONTROLLER_OPEN_FLOOR =
FIELD_READ(elevator_UNIT_CONTROLLER_ADDRESS,ELEVATOR_UNIT_CONTRO
LLER_OPEN_FLOOR_MASK,
ELEVATOR_UNIT_CONTROLLER_OPEN_FLOOR_OFFSET);

  if( UNIT_CONTROLLER_OPEN_FLOOR < 1 ||
UNIT_CONTROLLER_OPEN_FLOOR > max_floor){

    // Call firmware print method

    printf("invalid floor no .Please try again",);
   }
   return 0;
}

.         .         .         .         .
```

UVM Output

```
class uvm_move_elevator_seq extends
uvm_reg_sequence#(uvm_sequence#(uvm_reg_item));
    `uvm_object_utils(uvm_move_elevator_seq)

    uvm_status_e status;

    power_top_block rm ;

    function new(string name =
"uvm_move_elevator_seq") ;
        super.new(name);

    endfunction

    int Current_floor = 10 ;
    int lvar;

    task body;
```

```
 uvm_reg_data_t UNIT_CONTROLLER_OPEN_FLOOR ;
        uvm_reg_data_t UNIT_CONTROLLER_CURRENT_FLOOR ;
        uvm_reg_data_t UNIT_CONTROLLER ;

        if(!$cast(rm, model)) begin
            `uvm_error("RegModel :
power_top_block","cannot cast an object of type
uvm_reg_sequence to rm");
        end

        if (rm == null)  begin
            `uvm_error("power_top_block", "No register
model specified to run sequence on, you should specify
regmodel by using property 'uvm.regmodel' in the
sequence")
            return;
        end

        rm..read(status, UNIT_CONTROLLER_OPEN_FLOOR,
.parent(this));
        rm..read(status, UNIT_CONTROLLER_CURRENT_FLOOR,
.parent(this));
.        .        .        .        .
```

Thank You

Questions