VC Formal Lab
Memory Abstraction

Learning Objectives

In this VC Formal lab, you will use an ALU example to learn to do the following:

- Perform memory abstraction
- Find a solution to converge the properties

Lab Duration:
40 minutes

Familiarity with the SystemVerilog Assertion (SVA) language and knowledge of basic formal verification concepts are required for this lab.

Files Location

All files for this VC Formal lab are in directory:
$VC_STATIC_HOME/doc/vcst/examples/FPV/Abstraction/Memory_Abstraction

Directory Structure	
FPV/Abstraction/Memory_Abstraction	Lab main directory
README_VCFormal_Memory_Abstraction.pdf	Lab instructions
design/	Verilog RTL code of the Device Under Test (DUT)
sva/	SVA properties to check functionality of the DUT
run/	Run directory
solution/	Solution directory

Resources

The following resources are available for in-depth guidance regarding VC Formal usage, commands, and variables.

VC Formal User Guide:
$VC_STATIC_HOME/doc/vcst/VC_Formal_Docs/VC_Formal_UG.pdf

VC Formal Apps Quick References Guides:
$VC_STATIC_HOME/doc/vcst/VC_Formal_Docs/Quick_Reference_Guides/

VC Formal Apps Tcl Templates:
$VC_STATIC_HOME/doc/vcst/VC_Formal_Docs/Quick_Reference_Guides/vcf_tcl_templates/

Prepare your Environment

1. Set environment variable pointing to your VC Formal installation directory:

```
    %setenv VC_STATIC_HOME /tools/synopsys/vcstatic
```

2. Add path $VC_STATIC_HOME/bin to the PATH environment variable.

3. Change your working directory to FPV/Abstraction/Memory_Abstraction/run:

```
    %cd FPV/Abstraction/Memory_Abstraction/run
```

Now you are ready to begin the lab.

Create a run.tcl Setup File

VC Formal has a Tcl-based command interface. It is common to start with a Tcl file to set up and compile a design. In this step, you will create a VC Formal Tcl file for the DUT, an ALU, used in this lab.

4. Open file run.tcl (any arbitrary name is ok to use) using any text editor:

```
    %vi run.tcl
```

5. Add command to enable FPV App mode (default when starting VC Formal):

```
    set_fml_appmode FPV
```

6. Specify Formal TB top level module name as Tcl variable:

```
    set design tb_alu
```

7. Add command to compile DUT and SVA properties:

   The DUT files and filelist are located under directory FPV/Abstraction/Memory_Abstraction/design. The assertion and bind files are located under directory FPV/Abstraction/Memory_Abstraction/sva.

```
   read_file -sva -format sverilog -top $design \
        -vcs {-sverilog ../design/alu.sv ../sva/tb_alu.sv}
```

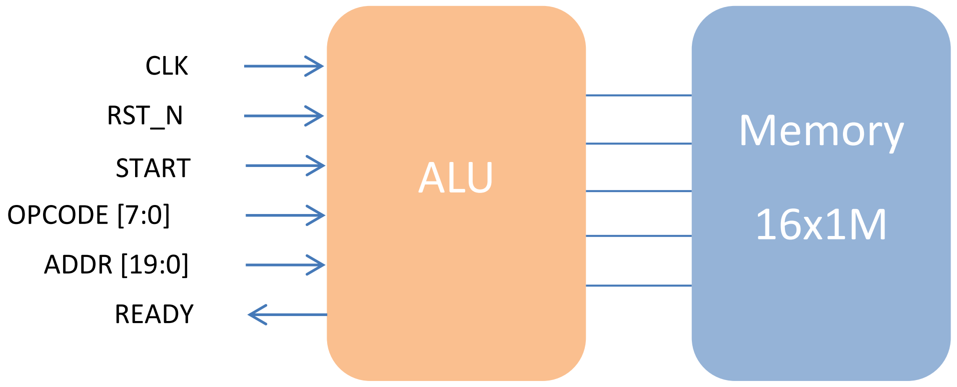
   Note: To use unified usage model to compile design, use these commands instead of `read_file` to compile design and SVA properties:

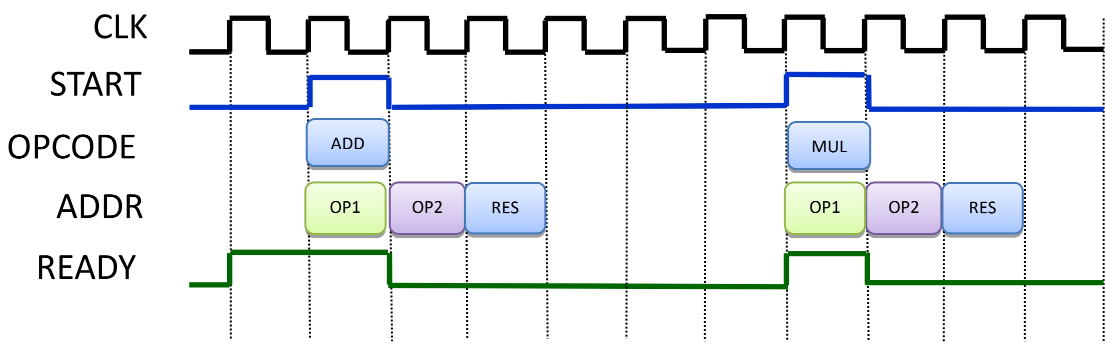
```
    analyze -format sverilog \
      -vcs {-sverilog ../design/alu.sv ../sva/tb_alu.sv}
    elaborate $design -sva
```

8. Save run.tcl file and exit editor.

ALU Design Implementation

- ALU implements memory-to-memory operations based on OPCODE and ADDR inputs when START and READY are asserted.
- Some opcodes require multiple addresses that are provided on consecutive cycles.





- When OPCODE == 8'h05, the ALU performs addition of the values at the two operand addresses and stores the result back in the result address. It takes three clock cycles to complete the operation.
- The addresses of the operands and result do not need to be unique.
- There is a property in the ALU testbench provided that verifies that the correct value is written to the result address. The large size of the memory prevents convergence.
- The design is triggered on posedge of CLK and implements an asynchronous negedge triggered RSTN input.

Formal Testbench for ALU

- op1_data stores operand1
- op2_data stores operand2
- res_addr stores the result address
- res_data stores the result

```
case (alu_state)
           IDLE : begin
              if (start) begin
                 op1_data  <= dut.mem_out;
                 alu_state <= OP2;
              end
           end
           OP2 : begin
              op2_data  <= dut.mem_out;
              alu_state <= RES;
           end
           RES : begin
              res_addr  <= random_addr;
              alu_state <= IDLE;
           end
           default : begin
              alu_state <= IDLE;
           end
         endcase // case (alu_state)
…
assign res_data = dut.mem.core[res_addr];
```

- start to be deasserted for next 2 cycles once ready and start are asserted

```
asm_start_interval : assume property (@(posedge clk) disable iff
(~rst_n)(random_start && ready) |=>(!random_start)[*2]);
```

- Constraint to set opcode to 8'h05 or 8'h06

```
asm_opcode_valid : assume property (@(posedge clk) disable iff
(~rst_n) random_start |-> (random_opcode==8'h05 ||
random_opcode==8'h06));
```

- Assertion to check add (opcode==8'h05)

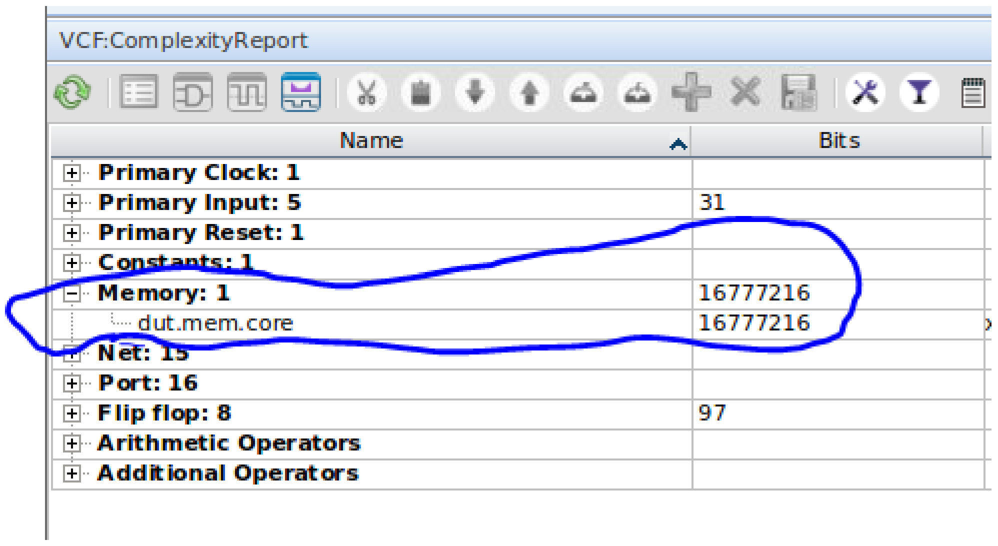
```
ast_add_check: assert property (@(posedge clk) disable iff
(~rst_n)(start && random_opcode==8'h05) |-> ##3 (res_data ==
(op1_data + op2_data)));
```

9. Start the tool in Verdi GUI mode:

```
    %vcf -f run.tcl -verdi
```

VC Formal starts in the Verdi GUI mode, with icons, tables, tabs, and windows especially designed for property verification with the FPV App. The App mode is set to FPV by default.

Assertion is run and “ast_add_check” do not converge. Property complexity report shows the memory to be hindering convergence.



Name	Bits
Primary Clock: 1	
Primary Input: 5	31
Primary Reset: 1	
Constants: 1	
Memory: 1	16777216
dut.mem.core	16777216
Net: 15	
Port: 16	
Flip flop: 8	97
Arithmetic Operators	
Additional Operators	

Lab Solution

- Abstract memory by blackboxing it

```
    set_blackbox -cells {dut.mem}
```

- Create an abstract model:
    - Add FFs to save `addr` and `mem_in` when `we` is high
    - Drive `mem_out` with saved data when `addr` matches write address
    - Drive TB `mem_res` with saved data

```
always @(posedge clk)
     if (we) begin
        res_addr <= addr;
        mem_data <= mem_in;
     end

assign mem_out = (addr==res_addr)? mem_data : free_data;

assign mem_res = mem_data;
```