# Formal Methods to Verify the Power Manager for an Embedded Multiprocessor Cluster

**Kesava R. Talupuru**
MIPS Technologies, Inc
Sunnyvale, CA
kesava@mips.com

## ABSTRACT

*With shrinking process geometries, static and dynamic power are increasing rapidly, forcing designers to use a variety of implementation techniques to control power. MIPS Technologies processor cores designed for low power applications use multiple power modes and employ a Power Manager to ensure correct transitions between these modes. This paper focuses on the verification of the Power Manager in the context of the – MIPS 1004K™ Coherent Processing System (CPS), in which various software and hardware events can control switching of power states. Without exhaustive verification of the Power Manager, the power management functionality of the design cannot be guaranteed. This paper discusses on how we successfully used formal methods to verify the Power Manager.*

**Categories and Subject Descriptors**

D.2.4 **[Software Engineering]:** Software/Program Verification – *assertion checkers, formal methods, validation.*

**General Terms**

Design, Verification.

**Keywords**

Power Management, Power Gating, Power Domains, MIPS Processor, 1004K™ CPS, Open Core Protocol, Assertions.

## 1. INTRODUCTION

Power consumption has become a critical issue in System-on-Chip (SoC) design. To lower power numbers, designers use a variety of techniques: clock gating, multi-voltage, dynamic voltage scaling, and power gating [1] [3]. Power-aware designs employ a Power Manager - to identify the state of the system, and ensure proper transitions to appropriate power states. The Power Manager is also needed to ensure that clocks, resets, isolation signals, etc. are all generated in an architecturally defined sequence. Typically, the Power Manager is a combination of hardware and software-based control logic.

The MIPS 1004K™ Coherent Processing System (CPS) implements low power techniques. As shown in Figure 1, the 1004K CPS consists of up to four microprocessor cores, and a Coherency Manager (CM) to route coherence traffic between the cores. The physical implementation of the 1004K CPS supports five distinct power domains - four CPUs and the CM.

The 1004K CPS Power Manager has three components as shown in Figure 2 that need to be verified. The first component is the Open Core Protocol (OCP) [5] interface which is used to communicate with other blocks within the SoC. The second component is the access control block which has global, local, and peer memory mapped registers that can be programmed by software. The third component is the finite state machine and control logic which controls the transition of power states.

Since the Power Manager is very critical to the overall functionality of the SoC, exhaustive verification of this block is a must. As power states can be altered by an array of possibilities, identifying and verifying all the corner case scenarios will be challenging with a simulation based approach. Moreover, the simulation speeds are very slow at SoC level. Formal verification fits best in this case as the power manager block is full

of control logic, and the formal proofs are very exhaustive.

The OCP interface was formally verified by reusing the OCP assertion kit described in [2], and will not be discussed in detail in this paper. All the properties in this paper are written using System Verilog Assertions (SVA) [4]. The rest of the paper will describe the following: Section 2 talks about the verification of the access control block; Section 3 talks about the verification of the Power Manager finite state machine; Section 4 talks about different bugs found; Section 5 talks about integrating formal into system level simulation verification; Section 6 talks about the results using the Synopsys Magellan formal tool.

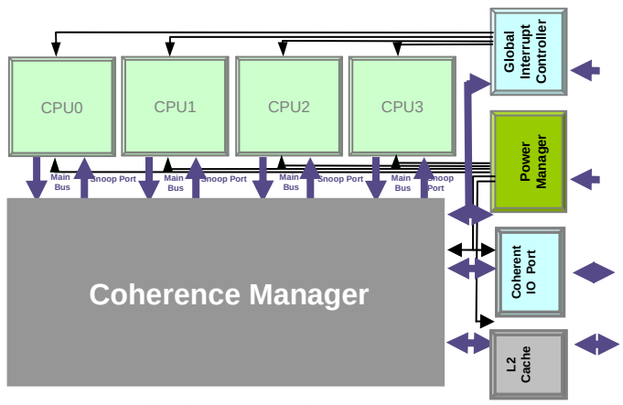


Figure 1. MIPS32® 1004K™ Coherent Processing System

## 2. Access Control Block Verification

This block consists of various 32-bit memory mapped registers which can be accessed by software. Software can read these registers to query the status of power manager, or write these registers with various power commands to control power transitions. If there are no writes from software, then the registers should hold the reset values or previously written data. This block communicates with the SoC through the OCP interface. Formal properties are deduced for this block based on the fact that the registers can be read, written, and should hold the current data if no writes occurred. The following properties written in SVA cover the verification of all the above mentioned functionality.

### 2.1 Reset Values

Some registers have read only bits which reflect some configuration choices. These bits get set immediately after reset is de-asserted. The property below formally proves that these bits are set correctly the cycle after reset. The assertion checks that in the cycle where reset goes from one to zero, the register holds the data matching to the specifications. As the assertion is disabled during reset, a one cycle delayed version of reset needs to be used.

```
s_asrt_valid_CPC_CONFIG_REG_Reset_Values
: assert property(
@ (posedge clk)
disable iff (Reset)
(Reset_dly1 == 1'b1)##1(Reset_dly1 ==
1'b0) |-> (reg_config_data[31:0] ==
32'h0F00_00FF));
```

### 2.2 Write operation

As shown in Figure 3, it takes two clock cycles to write the data into the appropriate register. The property is written from the OCP interface to the destination register. The assertion below makes sure that there is a write request, that the address is within the allowed address space, and that there are write permissions to the power domains. After two cycles, the register contents should match the data that appeared on OCP Bus two cycles earlier.

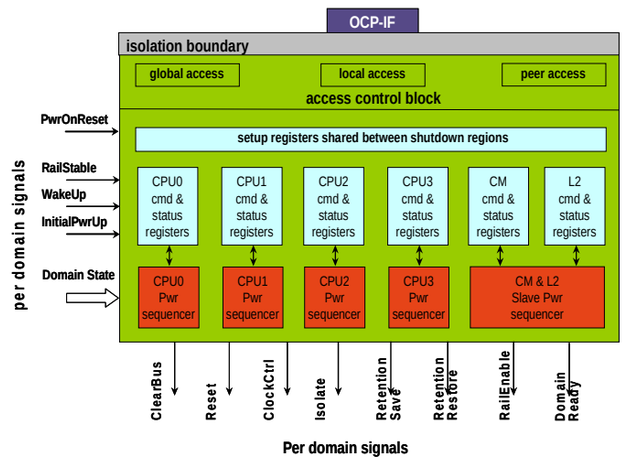


**Figure 2. Power Manager Components**

```
s_asrt_valid_CPC_GLOBAL_ACCESS_Write :
assert property(
@ (posedge clk)
disable iff (Reset)
```

```
(OCP_MCmd == `OCP_CMD_WRITE) &&
(OCP_MAddr[14:0] == 15'h0000) &&
(OCP_SCmdAccept == 1'b1)
##1 (access_permission)
|=> (reg_global_data[31:0] ==
$past($past(OCP_MData));
```

### 2.3 Read operation

This is similar to the write operation. It takes one clock cycle for the data to appear on OCP interface. The assertion below makes sure that there is a read request, and that the address is within the allowed address space. If few bits of the register are always zero, then those bits are directly compared to zero at OCP interface.

```
s_asrt_valid_CPC_GLOBAL_ACCESS_Read :
assert property(
@(posedge clk)
disable iff (Reset)
(OCP_SResp == `OCP_RESP_VALID) &&
(OCP_MAddr_dly1 == 15'h00FF)
|-> (OCP_SData[7:0] ==
reg_global_acc[7:0]) && (OCP_SData[31:8]
== 24'b0));
```

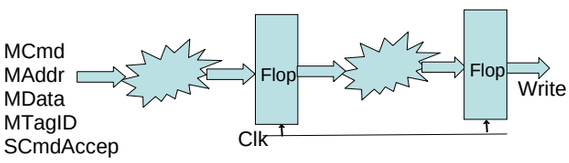


Figure 3. Write Logic Flow

### 2.4 Hold Data

If there is no write request, then the data in the registers should not change. This property is proven by saying that if the data got changed this cycle, then there should have been a write request two cycles before which matches to this register. This property needs to be checked at least two cycles after reset as we check data two cycles past.

```
s_asrt_cpc_valid_CPC_GLOBALACCESS_hold :
assert property(
@(posedge clk)
disable iff (Reset)
(r_Reset == 1'b0) ##1 (r_Reset == 1'b0)
##1(reg_global_data !=
$past(reg_global_data))
|-> ($past($past(write_req_0000))) &&
($past(_access_permission) == 1'b1));
```

### 3. Power Manager FSM

The Power Manager Finite State Machine (FSM) controls the power modes of CPUs based on the software and hardware events it receives. Typical software commands are: Power Down, Power Up, Clock Off, and Reset. The hardware events are triggered by toggling different external pins: Power Up and Interrupt pins. As shown in Figure 4, the power manager can make each individual CPU assume one of four modes of operation - coherent, non-coherent, clock-off and power-down.

**Coherent Mode**: In this mode, CPU operates as member of an L1 cache coherent domain and exchanges coherence messages with processor peers to maintain cache coherence. To avoid corruption of coherent data in the caches, power and clock must not be disabled while the CPU is in this mode.

**Non-Coherent Mode:** In this mode, CPU operates outside the coherent domain and does not exchange coherent messages with peer cores.

**Clock-Off Mode**: Any CPU operating outside of the coherent domain can assume clock-off mode. Clock distribution towards this CPU is cut at the clock generator level.

**Power-Down Mode**: In this mode, CPU is electrically isolated from its surroundings, and the supply voltage is removed. Other cores in power-up mode will continue to operate.

FSM state transitions between operating levels are depicted in Figure 4. Each transition points toward a possible command target state. However, the power manager FSM will step through a series of transitional states to maintain power integrity. Each programmable transition is outlined below:

**Coherent to Non-Coherent Mode transition:**
The core leaves the coherent domain and starts operating independently. Software must flush dirty data from data cache, and invalidate all clean data cache lines.

**Non-Coherent to Coherent Mode transition**: An independently operating core becomes a member of the coherent cluster, and the data cache lines are

invalidated as the core will be participating in coherency.

**Non-Coherent to Power Down Mode transition:** A core which is not member of the coherent domain is powered down, and the power manager will initiate the clock and power shutdown sequence.

**Non-Coherent to Clock Off Mode transition:** The clock tree root is disabled for this core, and dynamic power consumption is removed.

**Clock Off to Power Down Mode transition**: Power supply is removed for the disconnected core. Dynamic and leakage power is removed. The Power Manager initiates the power shutdown sequence.

**Clock Off to Non-Coherent Mode transition**: When a power up or reset command is given to the power manager, the FSM initiates a sequence to make the core operational.

**Power Down to Non-Coherent Mode transition**: When the power manager receives a power up command, the FSM initiates a power up sequence and the core become operational.

| Target State | PwrOff | ClkOff | Non-Coherent | Coherent |
|---|---|---|---|---|
| ClkEn | 0 | 0 | 1 | 1 |
| Isolate | 1 | 0 | 0 | 0 |
| Rail Enable | 0 | 1 | 1 | 1 |
| DomainReady | 0 | 0 | 1 | 1 |

**Table 1: FSM State Outputs**

In each mode of operation, we need to verify that conditions related to clock, reset, isolation, etc. are met. Additionally, there are a wide variety of protocols that need to be checked. Some of the important properties that need to be checked for the Power Manager are listed below.

### 3.1 Power Manager Output Signals Check

In all the different power modes, the Power Manager should make sure that isolation, clock enables, resets, etc are all driven with appropriate values as shown in Table 1. This can be verified by writing a common property as shown below and then instantiating that property for each power mode**.**

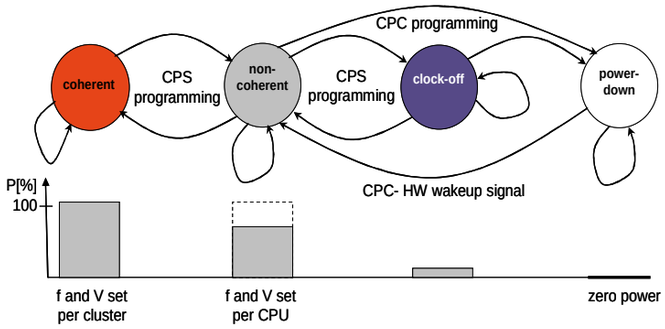


Figure 4. Cluster CPU States

The property below takes two inputs for each power manager output. One input value is the current RTL value, and the other input is the expected value passed by the user based on specification.

```
Property cpc_valid_seq_state_outputs
(clken, clken_expected, si_isolate,
si_isolate_expected, si_reset,
si_reset_expected retsave,
retsave_expected);
(si_clken == si_clken_expected) &&
(si_isolate == si_isolate_expected) &&
(si_reset == si_reset_expected) &&
(retsave == retsave_expected);
endproperty
```

The property is instantiated for each power manager state, and expected values are passed in the instance. The property below is the instance for power down.

```
asrt_vld_seq_state_outputs_in_powerdown :
assert property(
@(posedge clk)
disable iff (reset == 1'b1)
((seq_state == `PWRDN)|->
prop_cpc_valid_seq_state_outputs(
ClkEn, 1'b0,
Isolate, 1'b1,
Reset,    1'b0,
DomainReady, 1'b0
));
```

### 3.2 Power Manager FSM Valid Transitions

The Power Manager FSM should transition from the current state only to valid next states. For example, as shown in Figure 4, if the current state is Clock Off, then the valid next states are non-coherent, power down and clock off. The property below checks the valid transitions for all the power modes.

```
asrt_fsm_valid _transitions :
assert property(
@(posedge clk) disable iff (reset ==
1'b1)
(state == `PWRDN) |=>((state == `NONCOH)
|| (state == `PWRDN))
or (state == `NONCOH) |=>((state == `COH)
|| (state == `CLKOFF) || (state ==
`PWRDN) || (state == `NONCOH))
or (state == `COH) |=>((state == `NONCOH)
|| (state == `COH))
or (state == `CLKOFF) |=>((state ==
`NONCOH) || (state == `PWRDN) || (state
== `CLKOFF));
```

### 3.3 Power Manager FSM Valid Transition Triggers

Power Manager FSM will change from one power domain to another only if the valid transition trigger command is received. For example, as shown in Figure 4, the FSM will transition from power down mode to non-coherent mode only if the reset trigger is issued.

```
asrt_fsm_valid _trigger_transitions :
assert property(
@(posedge clk) disable iff (reset ==
1'b1)
(state == `PWRDN) && (command == `RESET)
|=>(state == `NONCOH));
```

### 3.4 FSM States Reachability

This property can be checked either by writing a cover point and enabling coverage in formal tool, or by writing an assertion which specifies that the state can never be reached. If the assertion passes, then that state is never reachable.

```
asrt_fsm_state_cannot_be_reached :
assert property(
@(posedge clk)
disable iff (reset == 1'b1)
(state != `COH));
```

### 3.5 Checking Power Manager States

Power modes can be changed as a result of occurrence of different events. If multiple events occur at the same time, then the power mode is switched based on priority. The property below checks that if reset, power down, and interrupt are received at the same time, and if the power manager is currently in coherent state, then all the events will be ignored and the power domain will not change.

```
asrt_fsm_state_priorities:
assert property(
@(posedge clk)
disable iff (reset == 1'b1)
Reset && (command == `PWRDN) && Interrupt
&& state == `COH)
|=>(state == `COH));
```

### 3.6 Coherent Core Shutdown

If any of the cores is participating in coherency, then that core will remain in coherent power domain to respond to peer CPU requests, and will ignore any triggers trying to change the power state. The property below checks that if current state is coherent and if the power down command is issued, the manager will still maintain the coherent mode.

```
asrt_fsm_state_hold_coh:
assert property(
@(posedge clk)
disable iff (reset == 1'b1)
(command == `PWRDN) && (state[10:0] ==
`COH) |=> (state == `COH));
```

### 3.7 Software Reset Deadlock Avoidance

When software issues a reset command for a domain, the command will be captured in the command register. Once the Power Manager resets the domain, then the command register contents should be squashed by the manager. If the reset command is not squashed, then the power manager will keep resetting the domain assuming that a new reset command has been issued.

```
asrt_fsm_state_reset_squash:
assert property(
@(posedge clk)
disable iff (reset == 1'b1)
(command == `RESET) ##1 (state[10:0] ==
`NONCOH) |=> (command == `NONCOH));
```

### 3.8 JTAG Check

The JTAG interfaces of each core are connected in a daisy chain. So, if JTAG is connected to the system, then none of the cores can be in power down mode. They should be in Clock Off mode so that the JTAG chain is not broken.

```
asrt_fsm_state_clkoff_jtag_chain_alive:
assert property(
@(posedge clk)
disable iff (reset == 1'b1)
```

```
(command == `PWRDN) && jtag_connected
|=>( state == `CLKOFF));
```

In addition to the above properties, there are some other properties that need to be checked: The CM cannot wake up if all the cores are in power down mode; If the coherent IO port is enabled, then the CM cannot power down, etc.

## 4. Bugs Uncovered

Following are the various kinds of bugs uncovered by proving the above mentioned properties using formal tools.

a) Finite state machines progresses based on target state. Target state is calculated based on power commands received. If no new command is received then the target state should hold. But, RTL was changing the target state to current intermediate state. This can create dead locks in the FSM.

b) When the coherent IO port is enabled, the CM should remain powered up even if all the cores are in power down mode. But the bug was that the CM was shutting down even when the coherent IO port was enabled.

c) Reset priority bugs – The power manager can do a cold or warm reset of power domain. For warm reset, certain status register contents will be preserved. Based on the current state, the FSM should decide whether to do warm or cold reset.

d) If Coherency is disabled for a particular core, then it should go to non-coherent mode.

e) The cores cannot shut down if there are pending transactions.

f) If the reset command is issued, the FSM should stop progressing once it reaches the reset state. But, in this case the FSM is in the reset loop

## 5. Integrating block level formal verification to system level verification

Block-level formal verification fits well into the overall system level power verification. System level simulation-based verification uncovered three bugs in the Power Manager. The bugs are mainly related to keeping the JTAG chain alive. These bugs were missed in formal verification as the specifications for this JTAG chain were not complete. With system level power aware simulations we found bugs outside of the power manager. It took 6 weeks of effort for the complete formal verification of the Power Manager, and about 20 weeks of effort for the system level verification. Formal verification environment set up took a week, but for simulation major effort was put into getting test bench environment ready.

## 6. Results

The Power Manager was formally proved by using the Synopsys Magellan [6] [7] tool. It took less than a week to get the formal setup working so that the properties could be proven. The formal tool found many bugs in the early design phases when the simulation environment was not ready. The power manager was almost bug free by the time the simulation environment was up and running. The simulation environment did find some bugs at the system level as the formal tool could not handle the system level proofs.

| Design | #Assrt | #Assum | #Proven | Time |
|---|---|---|---|---|
| One domain | 140 | 31 | 140 | 42 M |
| Two domains | 282 | 31 | 280 | 2 Hr |
| Five domains | 675 | 31 | 521 | 7 Hr |

**Table 2: Results with Magellan Tool**

The results for different numbers of power domains are listed in Table 2. For a single power domain the tool was able to prove all the properties in 42 minutues. For two power domains, the tool was able to prove all but 2 assertions in 2 hours. For all the five power domains, the tool was able to prove 521 assertions completely, whereas the remaining 141 assertions were proved partially for around 60 cycles.

## 7. Conclusions

Since the Power Manager is such a critical component of the low power SoC, exhaustive proof of correctness is desirable for this unit. Formal verification helped to exhaustively prove the correctness of the Power Manager in our

environment. With a combination of formal and power aware simulation based verification, low power designs can be made bug free.

## 8. REFERENCES


[1] Freddy Bembaron, Rudra Mukherjee, Sachin Kakkar, and Amit Srivastava . "Low Power Verification Methodology using UPF". DVCon 2009

[2] Ali Habibi, Jithendra Madala, Mandar Munishwar, and Haihui Chen. "Verifying MIPS designs using Magellan". SNUG 2008

[3] Srikanth Jadcherla, Janick Bergeron, Yoshio Inoue, and David Flynn. "Verification Methodology Manual for Low Power".

[4] IEEE, IEEE Standard for Property System Verilog, IEEE Std 1800-2005 (2005).

[5] OCP-IP Association, Open Core Protocol Specification, Release 2.2, 2006

[6] Synopsys, Discovery Visual Environment User Guide, Version Y-2006.06-SP1, April 2007.

[7] Synopsys, Magellan User Guide, Version MG_2009.03-5, March 2009