Formal Verification of OpenRISC 1200 Processor

Saqib Khan

I. INTRODUCTION

The OpenRISC 1200 (OR1200) is a 32-bit Reduced Instruction Set Computer (RISC) processor. This processor consists of all necessary components which are available in any other modern microprocessor, which is why it is widely used in the industry and academic for embedded, portable and networking applications. In this project the Verilog code for OR1200 CPU is verified formally by writing assertions, assumptions and cover properties using System Verilog. To gain a deeper understanding of how code coverage works in formal verification, we also implemented fault injection for tracing bugs based on assertion failure. The formal verification tool used in the project is Cadence Jaspergold. This report presents the procedures, analysis and detailed results of our study.

Formal verification uses static analysis based on mathematical transformations to determine the correctness of RTL implementation; compared to the dynamic verification achieved by simulation [2]. With increasing design complexity, verification teams employing simulation-based testing must come up with increased test vectors (input sequences) to test the DUT (design under test) to an acceptable degree of coverage. Formal verification is potentially very simple and fast because it doesn't have to evaluate every possible state to demonstrate that a piece of logic meets a set of properties under all conditions. However, its performance depends greatly on the type of logic on which it is deployed and the way it is applied.

II. OPENRISC 1200 ARCHITECTURE

The OR1200 processor is a 32-bit Reduced Instruction Set Computer (RISC) processor. It belongs to the OR1000 family of processors with Harvard microarchitecture. The features include 5-integer pipeline, virtual memory support (MMU), two default caches for data and instruction physically tagged together, high-resolution tick timer, power management unit, a programmable interrupt controller (PIC) and debug unit for interfacing and real-time debugging facilities as shown in Figure-1.

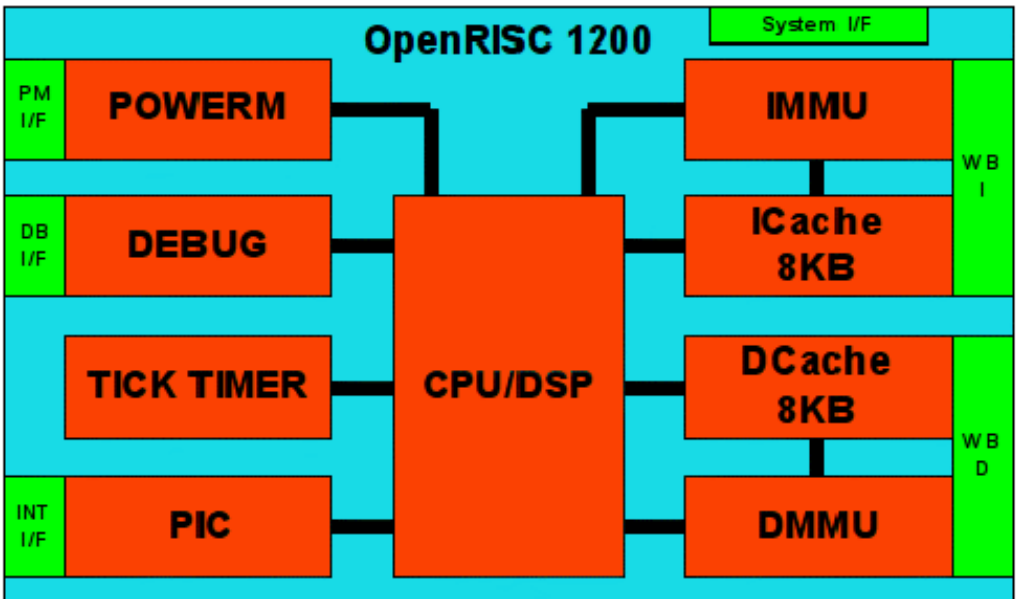

Figure-1 Block diagram for OR1200 processor architecture

This project only focuses on the verification of the CPU/DSP block; therefore, we will ignore the remaining blocks within OR1200 processor.

A. CPU/DSP

The primary and central processing part of OR1200 processor is the CPU. The CPU uses the architecture of OR1000 processor family and implements 32-bit operations while 64-bit is not realized for OR1200. Figure-2 shows the block diagram of OR1200 CPU/DSP.

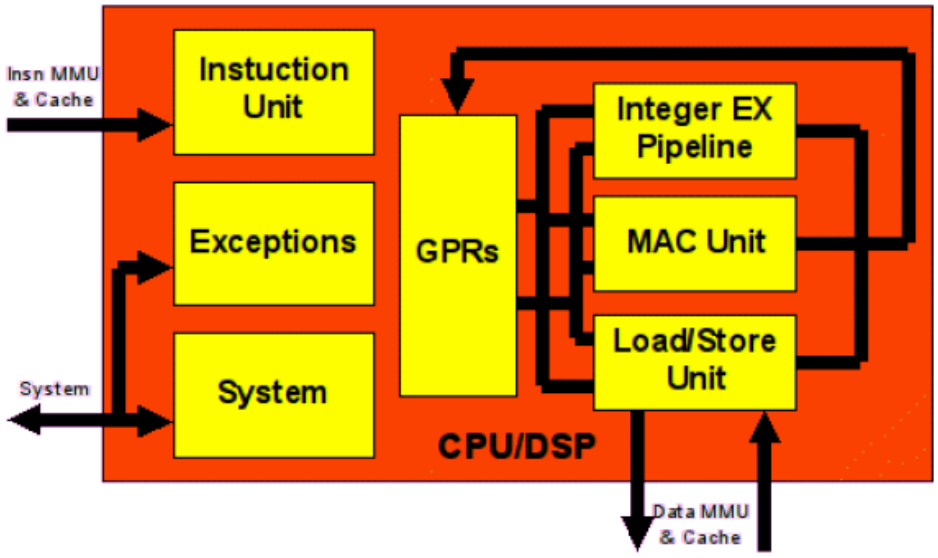

Figure-2 Block diagram of OpenRISC 1200 CPU architecture

1. **Instruction Unit**

It implements the basic pipeline instruction, fetching instructions from memory and executing them in proper order.

2. **General Purpose Registers**

There are 32 general purpose registers (GPRs). Each GPR is 32-bit wide and can be used as either source or destination registers. They are used to hold scalar data, pointers and vectors.

3. **Load/Store Unit (LSU)**

It is used to load data from memory or store data to memory. All load/store instructions are implemented in hardware. The operand may be in address register operands, source data register operand for store instructions and destination data register operands for the load instruction.

4. **Integer Execution Pipeline**

The following instructions are all 32-bit integer and implemented in this core. Most of the instructions take one cycle of time during execution.

- Arithmetic instructions
- Logical instructions
- Compare instructions
- Shift and rotate instructions

5. **System Unit**

This unit provides the interfaces to those signals to the CPU/DSP which cannot be connected through instruction and data interfaces. This unit implements the system's special purpose registers, e.g. Supervisor Registers

6. **MAC Unit**

This unit is responsible for DSP MAC operations which are 32x32 with the 48-bit accumulator. It can accept new MAC operation in each new clock cycle and is fully pipelined.

7. **Exceptions**

The core exception can be generated when exception handling occurs. In the OR1200 processor, there are some causes for exceptions to happen and given below. Exceptions take place in the supervisor mode. When it occurs, control transfers to exception handler at an offset depends on the type of encountered exceptions.

- Illegal op-codes
- External interrupt request
- System call
- Breakpoints exceptions (internal exceptions)
- Memory access conditions

III. DESIGN UNDER TEST (DUT)

The 5 stages of integer pipeline consist of Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MA) and Write Back (WB), as shown in Figure-3. These 5 stages make up the Datapath that is a collection of hardware components and their connections in a processor, which is responsible for determining the static structure of the processor. Besides the Datapath, there is a Control Path (Ctrl Path). The Ctrl Path determines the dynamic flow of data between the components of the Datapath.

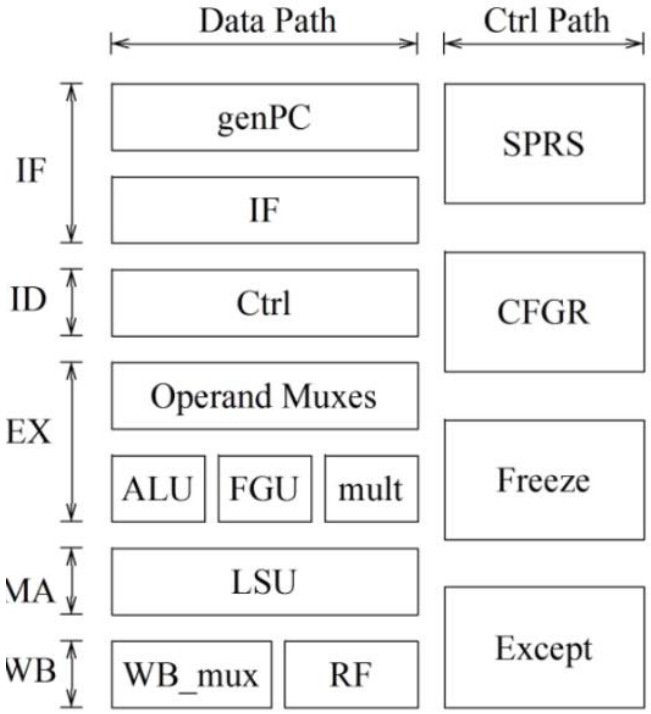

Figure-3 Five-stage integer pipeline block diagram

The Instruction Fetch (IF) stage consists of a program counter and fetches the instruction. The Control Block (Ctrl) is responsible for decoding the instruction and creating a corresponding control signal for the Execution (EX) stage. The load/store unit (LSU) in the Memory Access (MA) stage provides memory address and data alignment. Write back MUX (WB_mux) and register file (RF) are in the Write Back (WB) stage. Four control blocks, special purpose registers (SPRs), configuration registers (GFGR), Exception and Freeze, monitors the CPU's operation. In this project, we chose several blocks in both the data and control paths for developing assertions and writing cover properties.

A. INSTRUCTION FETCH (IF)

The Instruction Fetch (IF) consists of the generate Program Counter (genPC) and the instruction fetch block. The genPC is responsible for updating the Program Counter (PC) register, that holds the address of the current instruction. While the instruction fetch block is responsible for fetching an instruction from the instruction cache every cycle and uses the Program Counter (PC) to index instruction cycle. We wrote several assertions to verify the functionality of both the genPC and IF and the results for formal verification are provided in Appendix-A.

B. INSTRUCTION DECODE (ID)

The Instruction Decode (ID) consists of the control (Ctrl) block and is responsible for the following tasks:

1) Decodes opcode bits and sets up the control signals for later stages,
2) Read input operands from register file (RF) specified by decoded instruction bits, and
3) Write state to the pipeline register

The decoding process allows the CPU to determine what instruction is to be performed so that the CPU can tell how many operands it needs to fetch in order to successfully perform the instruction. The opcode fetched from the memory is decoded for the next steps and moved to appropriate registers. We wrote several assertions to verify the functionality the properties of the ID stage (or1200_ctrl). Setting the Ctrl Module as the top module, we performed formal verification and the results are given in Appendix-A

C. EXECUTE MODULE (EX)

The OR1200 has one Arithmetic Logic Unit (ALU), one Floating Point Unit (FPU) and one multiply unit. Therefore, the OperandMUX is needed to choose among different threads and provide operands to the appropriate unit. In this project we have chosen to verify only the OperandMUX in the Execute Module as the ALU and multiply units were extensively verified in LAB#03. Data forwarding is used in OR1200 to improve execution performance. The data for operand A could come from forwarding data through execution (EX) and write back (WB) stage, or the register files. While data for operand B comes from forwarding data through execution (EX) and write back (WB) stage, or the register files, or provided as an immediate number. In the

OperandMUX module, two signals “sel_a” and “sel_b” controls the assignment of both the two operands.

sel_a	Operand A	sel_b	Operand B
00	Register file	00	Immediate
01	Executing forwarding	01	Execution forwarding
10	Write back forwarding	10	Write back forwarding
11	Register file	11	Register file

Table-2 Control assignment for sel_a and sel_b

The operation of OperandMUX module is controlled by the freeze signal from each stage. Based on the different situations for the freeze signal from different stages, there are various scenarios:

1. If both the execution (EX) stage and instruction decode (ID) stage are frozen, then there are two situations
    a. If in the last cycle, instruction decode stage is not frozen, the OperandMUX module would continue assigning data to different operands
    b. If in last cycle, instruction decode stage is frozen, the OperandMUX would wait for one cycle, and in the next cycle the situation could go either 1a or 2.
2. If the execution stage is working while the instruction decode stage starts to freeze, then OperandMUX would just assign the current data to the arithmetic module and then waits for the instruction decode stage to escape from freeze mode.

We wrote several assertions to verify the functionality of the OperandMUX module (or1200_operandmuxes). Setting the OperandMUX Module as the top module, we performed formal verification and the results are given in Appendix-A.

D. EXCEPTION MODULE

The exception mechanism allows the processor to change supervisor state as a result of external signals, errors, or unusual conditions arising in the execution of instructions. When exception occur, information about the state of the processor is saved to certain registers and the processor begins execution at the address predetermined for each exception. When an instruction-caused

exception is recognized, any unexecuted instruction that appear earlier in the instruction stream are required to complete before the exception is taken. Table-1 lists different exceptions (including interrupts) along with their priorities.

Exception	Causal Conditions	Priority
Reset	Caused by software of hardware reset	1
Bus Error	Attempt to access invalid PA	2
Data Page Fault	No matching PTE found in page table for load/store	8
Instruction Page Fault	No matching PTE found in page table for instruction fetch	3
Tick Timer	Tick timer interrupt asserted	12
Alignment	Load/store access to not aligned location	6
Illegal Instruction	Illegal instruction	5
External Interrupt	External interrupt asserted	12
D-TLB Miss	No matching entry in DTLB	7
I-TLB Miss	No matching entry in ITLB	2
Range	Out of range	10
System Call	System call initiated by software	7
Floating Point	Caused by floating point instruction	11
Trap	Caused by I.trap instruction	7

Table-1 Exception types with their causal conditions and priorities

For the exception module, we are interested in verifying the following properties

1. Exception priority: when more than one exception occurs simultaneously, exception module would choose the exception with the highest priority.
2. Before going to the exception handler, the exception module would first flush the pipeline of the module. To flush the pipeline, there is a 6-stage finite state machine that must be satisfied; figure-4 shows this state machine.
3. In OR1200, there is no branch prediction. Instead, the delay slot is used to increase the performance of the program. Meanwhile, before the core flushes the pipeline and goes to the exception handler, it would first save the returned Program Counter (PC) in the Exception Program Counter Register (EPCR) register. Depending on whether the instruction is in the delay slot or not, there are two different conditions:

a. If the instruction is in the delay slot, EPCR keeps record of the address of jump instruction before it.
b. If the instruction is not in the delay slot, EPCR keeps record of the address of instruction itself.

Based on the properties mentioned above, we have developed several assertions and cover properties to test the functionality of the exception module (or1200_except). Setting the Exception Module as the top module, we performed formal verification using Cadence Jaspergold and the results are tabulated in Appendix-A.

E. CACHE MODULE

OR1200 has three levels of cache:

1. Quick embedded memory (QMEM), which works as the first level
2. Data cache and instruction cache are the second level
3. Store buffer works as the third level

The QMEM module is a small RAM. The main functionality of QMEM is to put some time critical functions into this memory and to have predictable and fast access to these functions (such as soft FPU, context switch, exception handlers, stack etc.). QMEM sits behind IMMU/DMMU, so all addresses are physical. When CPU needs to read from cache and the address is stored within QMEM, it could have a faster response compared with cache. When CPU needs to write to cache, QMEM will also be updated.

In the load/store (LSU) stage, the control signals are very important for the QMEM to communicate with other modules. For the data cache and instruction cache, since they share the same mechanism, we just take data cache as an example.

For data cache, there are two important points

1. Cache pre-fetch.
2. Cache lock

Figure-4 shows the state transition for the cache.

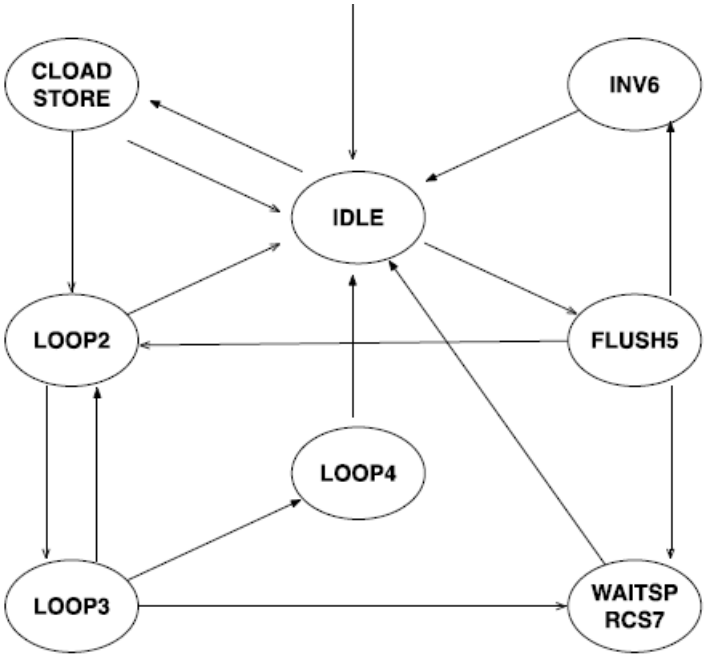

Figure-4 state transition graph for cache

The store buffer is a FIFO with two port RAM as physical device. It relates to a bus interface unit, which is the interface for data exchange for OR1200 peripherals. Setting the Cache Module as the top module, we performed formal verification and the results are presented in Appendix-A

IV. FAULT INJECTION AND DEBUGGING

Fault injection is a testing technique of introducing bugs into the code for improving code coverage and testing the validity of the defined test benches.

In this section we will introduce a bug into the Verilog code of the exception module (or1200_except.v). The highlighted code was commented out as depicted in Figure-5. By changing the flow of the Finite State Machine (FSM), we were able to force an assertion failure as shown in Figure-6.

```
`OR1200_EXCEPTFSM_IDLE:
  if (except_flushpipe) begin
     state <=  `OR1200_EXCEPTFSM_FLU1;
```

Figure-5 Fault injection in the exception module

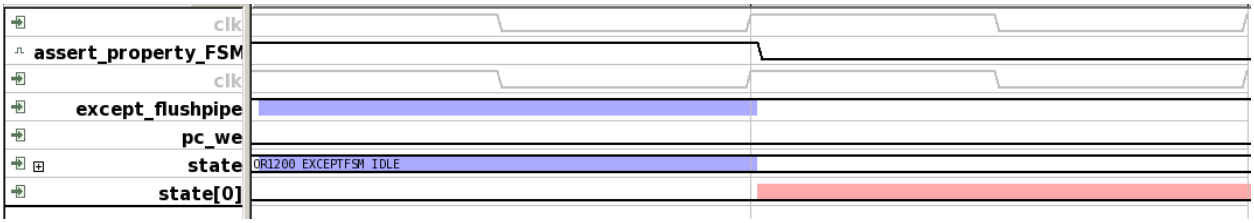

Figure-6 Failing assertion due to fault injection

As shown above, the failing assertion indicates that if the initial state of FSM state is “IDLE”, and “except_flushpipe” is true, then the FSM goes to the next state which is “FLU1”. However, due to the injected error, the FSM doesn’t change state as shown by the failing assertion.

V. CONCLUSION

This project gave us a good understanding of how formal verification works. The formal verification has been widely used in industry and is an appropriate method for final verification of a complex architecture, like the OR1200. In this project, we have learned how to use the Instruction Set Architecture (ISA) and design documents to write assertions and cover properties to test the functionality of the OR1200 core. Unfortunately, the design documents for OR1200 are poorly written and is missing several important details. This has forced us to use the RTL code to understand the base functionality of certain modules, which is a poor approach to writing tests (assertions and cover properties). This project also gave us the opportunity to work with mainstream verification tool, such as Cadence Jaspergold.

For future works, we can extend our verification effort to other modules that were not covered as part of this project, such as the ALU, FPU etc. Also, it would be helpful to understand the code coverage that resulted as part of writing these assertions.

REFERENCES

[1] [or1000_manual] Damjan Lampret et al. OpenRISC 1000 System Architecture Manual. 2004.

[2] Tsung-Han Heish and Rung-Bin Lin, "Via-configurable structured ASIC implementation of OpenRISC 1200 based SoC platform," in 2013 International Symposium on Next-Generation Electronics, vol. 1200. Kaohsiung, Taiwan: IEEE, feb 2013, pp. 21–24. [Online]. Available: http://ieeexplore.ieee.org/document/6512280/.

[3] Damjan Lampret, "OpenRISC 1200 IP Core Specification (Preliminary Draft)," Tech. Rep., 2001. [Online]. Available: www.opencores.org

[4] M. Bakiri, S. Titri, N. Izeboudjen, F. Abid, F. Louiz, and D. Lazib, "Embedded system with linux kernel based on openrisc 1200-v3," in 2012 6th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT), March 2012, pp. 177–182.

APPENDIX-A

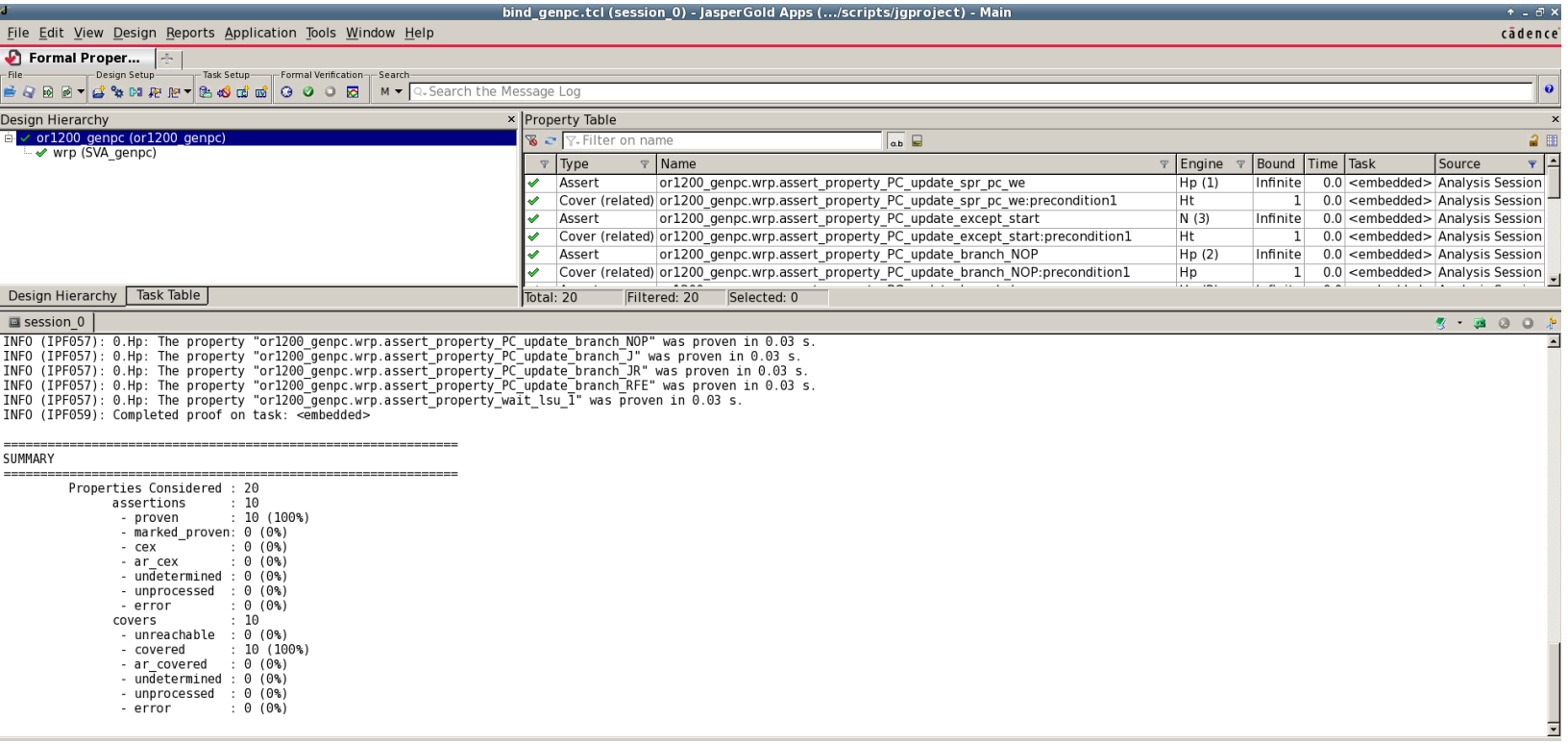

Figure-A1 Verification result for setting the or1200_genpc as the top module

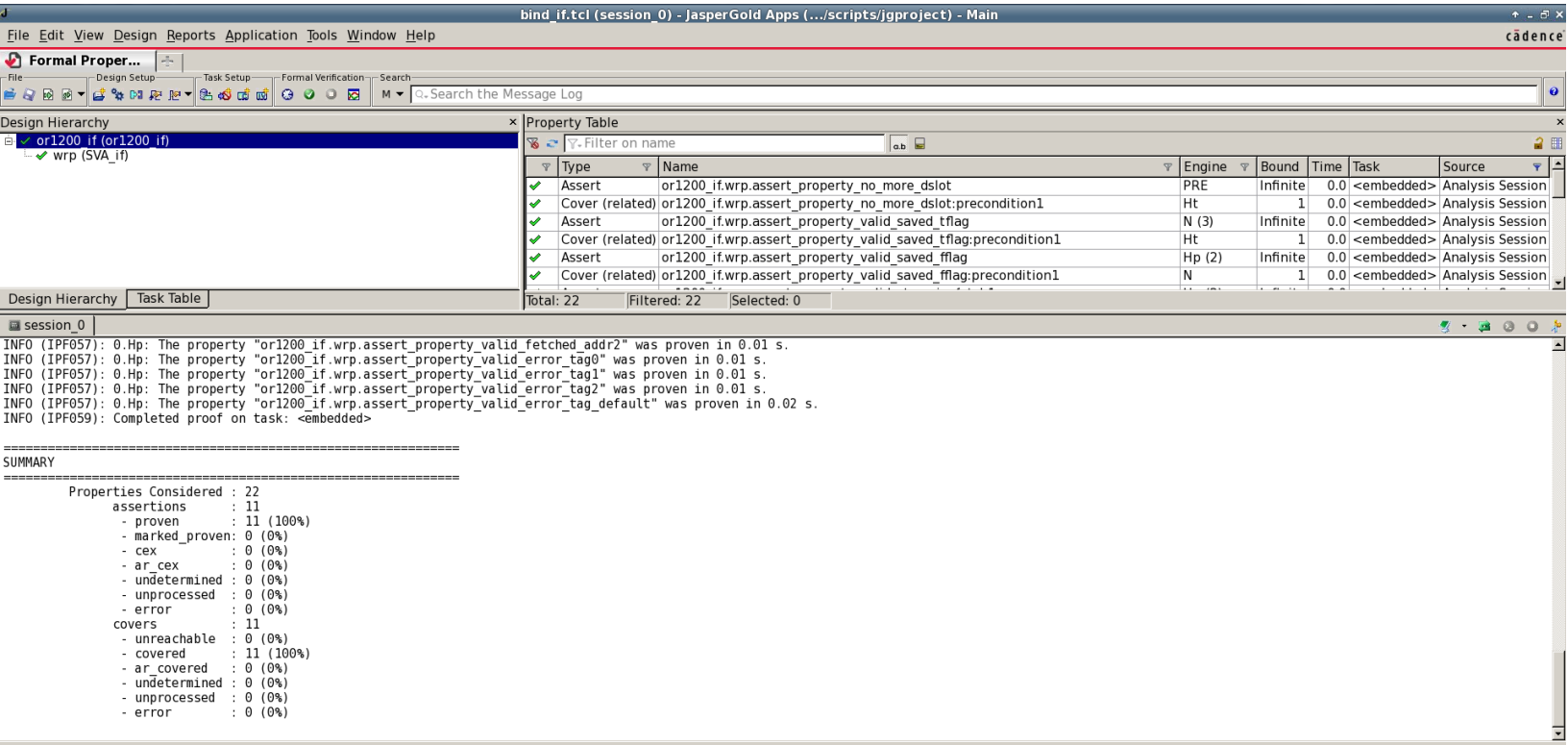

Figure-A2 Verification result for setting the or1200_if as the top module

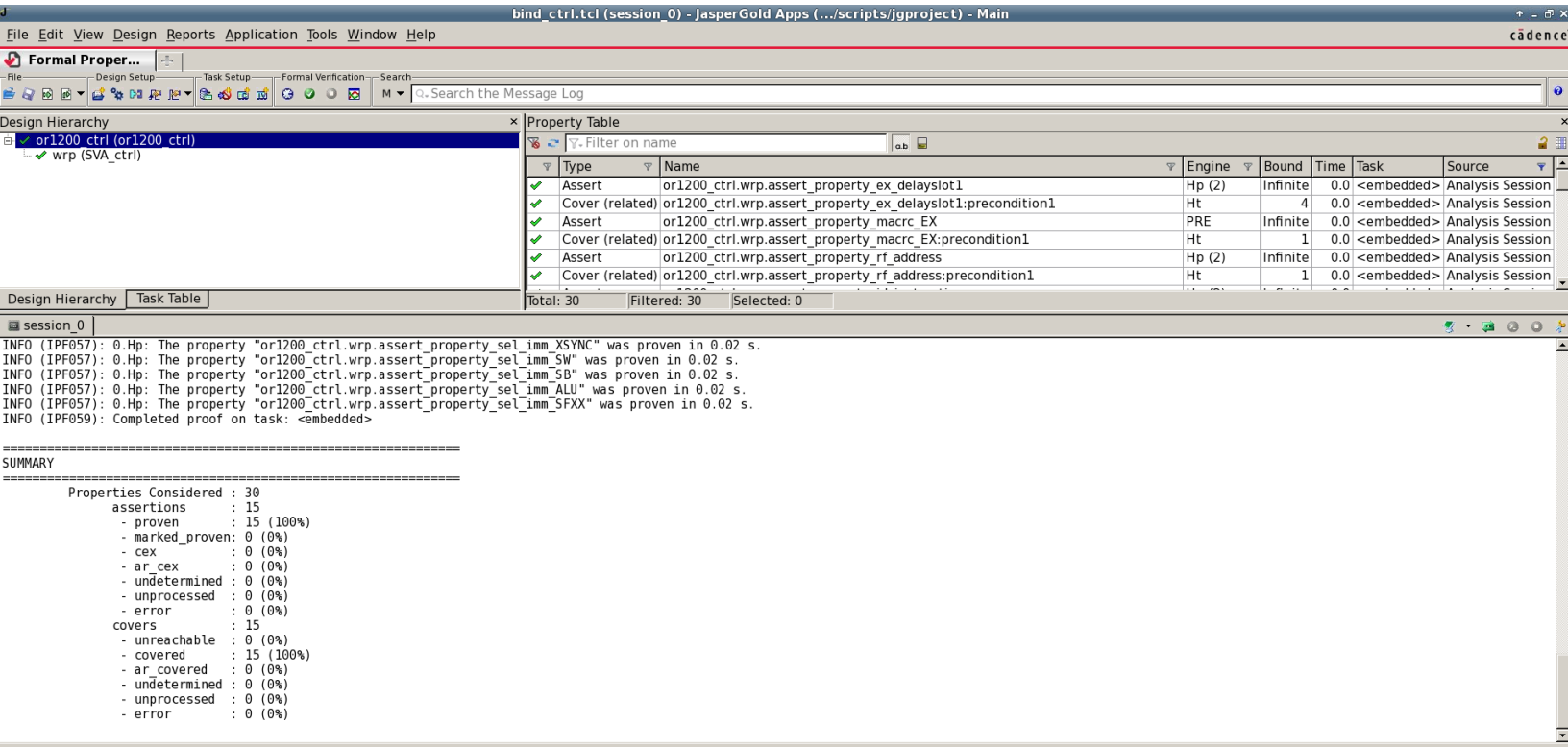

Figure-A3 Verification result for setting the or1200_ctrl as the top module

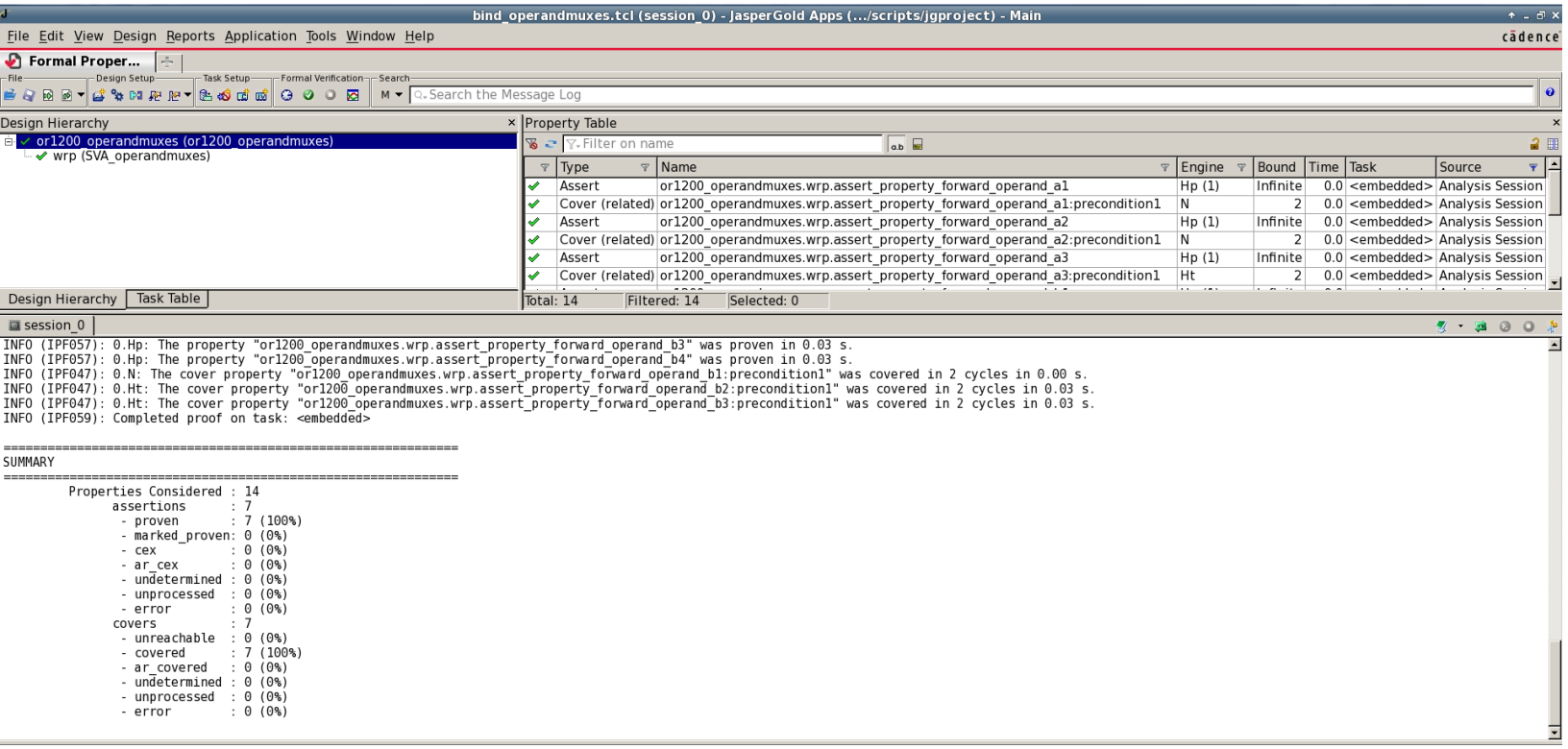

Figure-A4 Verification result for setting the or1200_operandmux as the top module

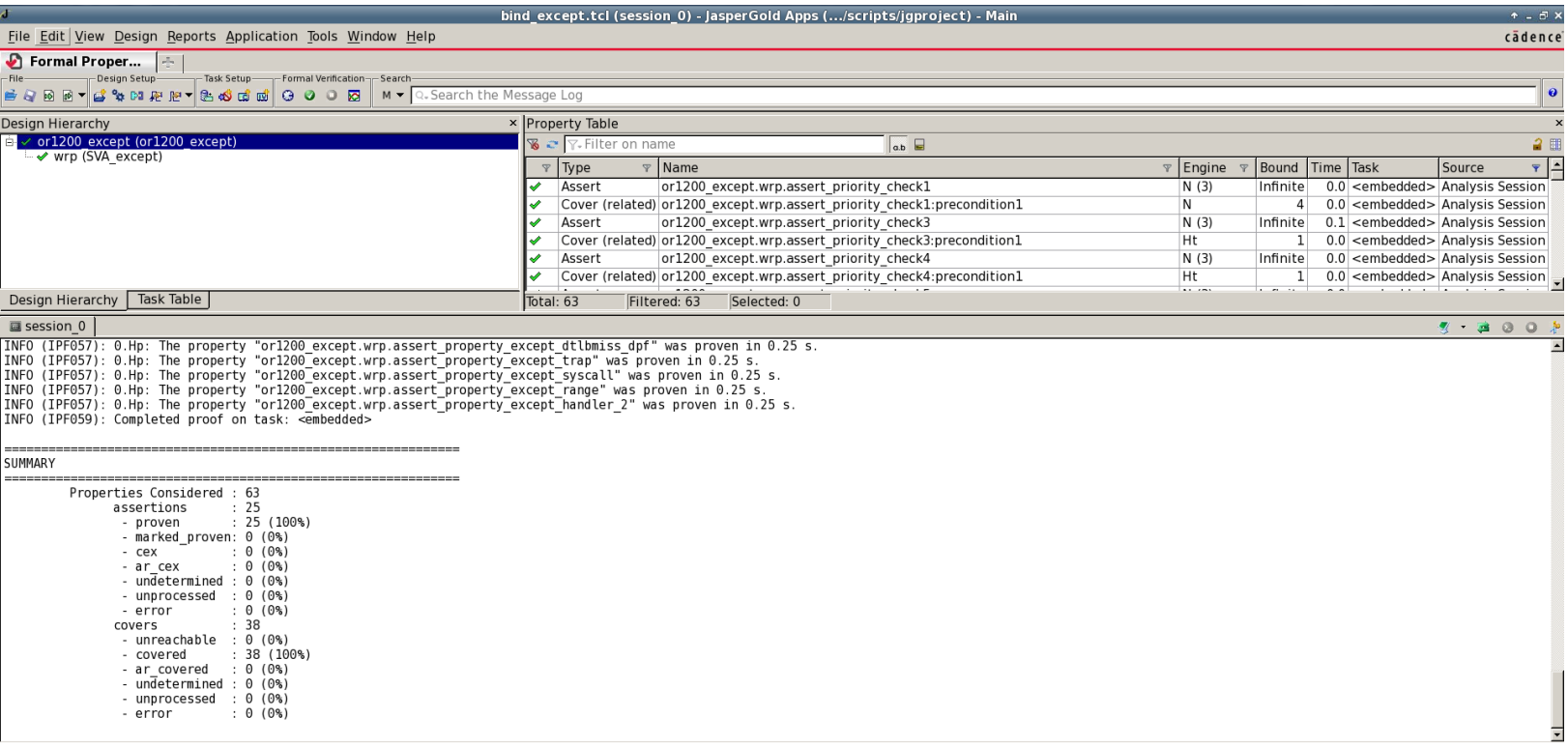

Figure-A5 Verification result for setting the or1200_except as the top module

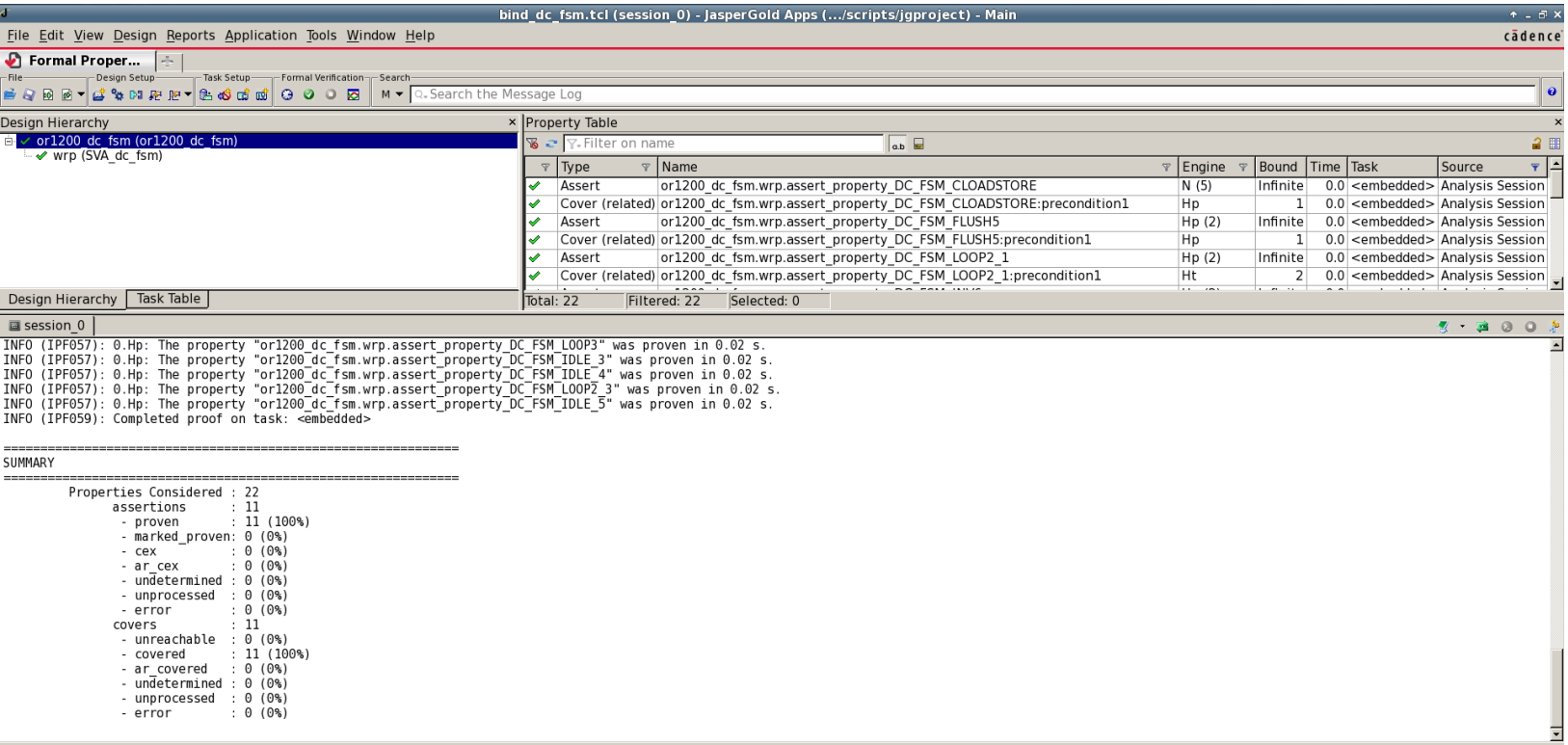

Figure-A6 Verification result for setting the or1200_dc_fsm as the top module

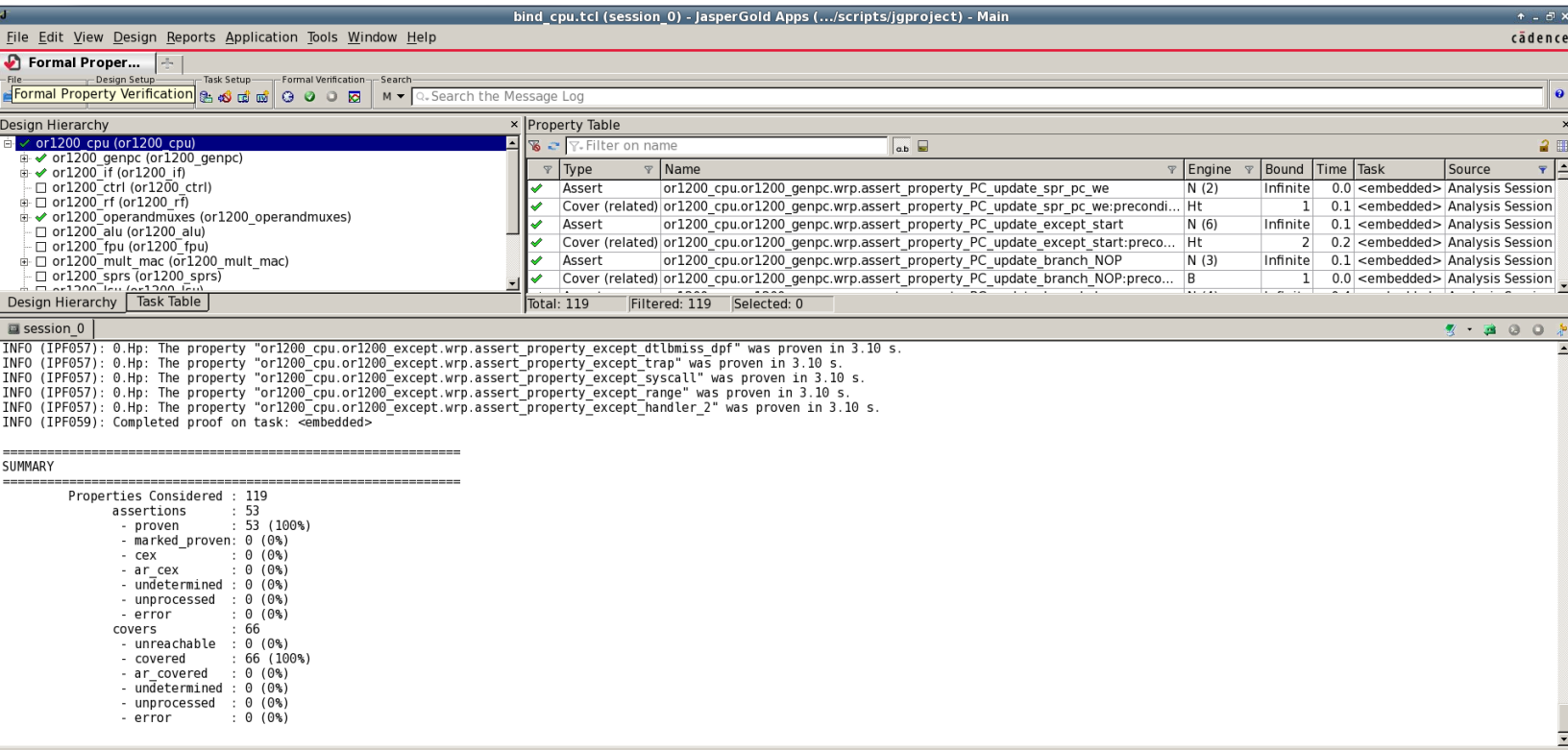

Figure-A7 Verification result for setting the or1200_cpu as the top module