Portland State UNIVERSITY

ECE-560, Fall 2024

Assertion Based Verification

Asst. Prof of Practice: Venkatesh Patil

Maseeh College of Engineering and Computer Science

Final Project Report

Assertion Based Verification of the I2C Communication Protocol

Team Number - 6

Kumar Durga Manohar Karna	kkarna@pdx.edu
Nivedita Boyina	nivedita@pdx.edu
Pooja Satpute	poojas@pdx.edu
Srikar Varma Datla	srikard@pdx.edu

Table of Contents

- Introduction
- Working of I2C communication protocol
- Block Diagrams
- Finite State Diagrams
- Dynamic Simulation and Results
- Formal Verification using AEP and FXP apps
- Formal Verification using FPV app
- Formal Verification test plan
- Contributions
- Challenges
- References

Introduction

The I2C (Inter-Integrated Circuit) communication protocol is a prevalent standard in embedded systems which facilitates for low-speed, short-distance data exchange between integrated circuits. It is a crucial protocol in numerous applications across industries since it allows multiple devices, including sensors, microcontrollers, and memory modules, to communicate across a two-wire bus. I2C's simplicity, combined with its flexibility, makes it particularly useful in circumstances that require minimum wiring and dependable data delivery.

As embedded systems are wrought in more complex forms, the need to validate commercial lines robustly becomes even more imperative. For instance, I2C applications would have to hinge strongly on various parameters, among them synchronisation between the master and slave devices, integrity of the data-to-be-transmitted, and effective error detection. Most traditional simulation techniques hardly catch the minute violations of protocol that happen in a complex design system.

Assertion-based verification (ABV) would prove effective in addressing all the above problems. This allows engineers to embed assertions for compliance of the logical requirements to simulate a system's behavior during the simulation process. A full-fledged verification and hardware description language, SystemVerilog affords robust constructs to realize such assertions in a format integrating them into an automatic testbenches verification environment.

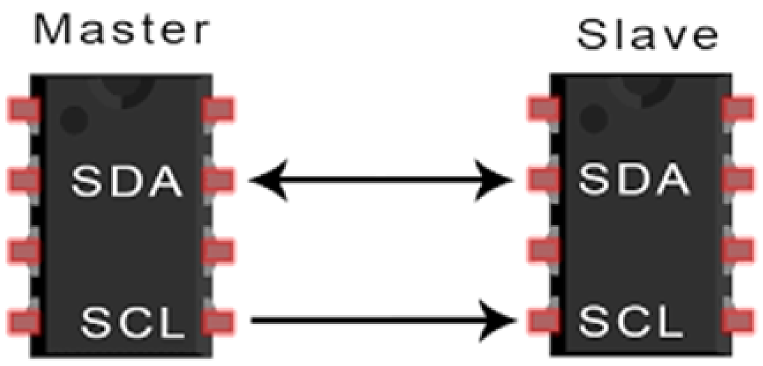
This project will focus on the assertion-based verification of the I2C communication protocol using SystemVerilog. The aim is mainly to develop sets of SystemVerilog assertions to enforce rigorous monitoring on critical aspects of the I2C communication with regard to data integrity, clock synchronization, and error detection in order to ensure that the communication is free of errors and behaves as expected in all possible practical scenarios.

The subsequent sections of this report will detail the methodology employed, the tools used in the verification process, the results obtained, and the conclusions drawn, highlighting the advantages of assertion-based verification in ensuring the correctness and reliability of I2C communication.

I2C Communication Protocol

The I2C (Inter-Integrated Circuit) protocol is a widely-used serial communication protocol that facilitates data transfer between integrated circuits in embedded systems. Developed by Philips, it uses only two wires: Serial Data (SDA) for data transfer and Serial Clock (SCL) for synchronization.

The I2C protocol represents master-slave communication since it comprises command from the master (typically a microcontroller) to slave devices for specific responses. They are designed for communication in short distances at low speed and are usually used to interface sensors, EEPROMs, RTCs, and related devices with a microcontroller.



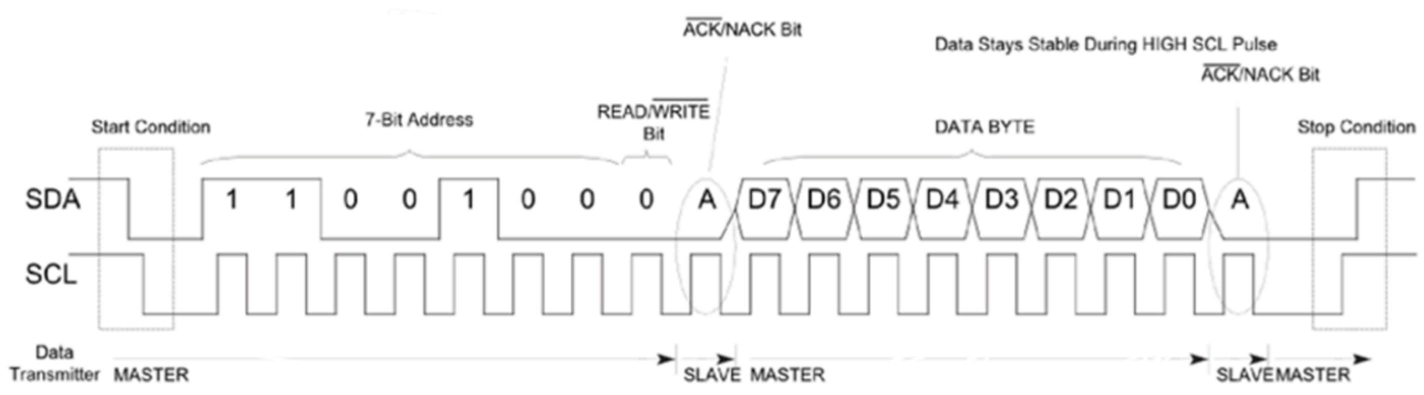
SDA (Serial Data)

The SDA line is a serial data wire that enables two-way communication of data between master and slave devices: sending and receiving from both the master and slave. It also carries the actual data being transferred in 8-bits (bytes) during the process of the communication.

SCL (Serial Clock)

The SCL line is a serial clock wire that conveys the clock signal generated from the master device. The clock carries synchronization of all the data interfaces between devices on the bus. For every clock pulse generated in the SCL line, retrieved data from the SDA line sends every bit consecutively to ensure both sending and receiving data is synchronized.

Bit Field Mapping for I2C



Address Bits

A 7-bit unique address is used to identify the slave device on the bus. The address is sent by the master when it intends to begin communication with a slave device. To the master, there is a unique address for every slave device connected to the bus which enables it to communicate with a specific device.

Read/Write Bits

The master sends this Read/Write bit to indicate to the slave whether the master is going to send data to it or receive data from it. If the bit happens to be 0, this indicates that the master would like to send data to the slave (write operation). If the bit is 1, the master intends to receive data from the slave (read operation). This bit thus defines for the master and slave whether data will be sent or received.

ACK/NACK bits

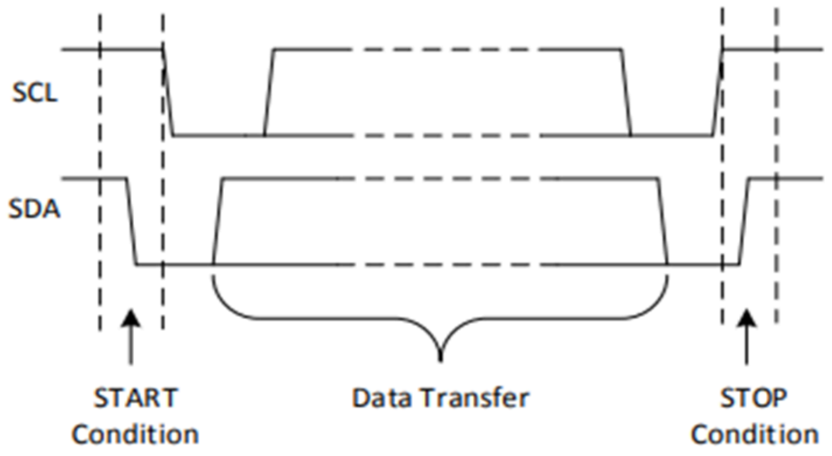
The Acknowledge (ACK) bit is for the purpose of confirming with the sending device (master/slave) that one byte of data has been received. For every byte that is transmitted, the receiver pulls SDA low to acknowledge that the byte has been received. In cases where no acknowledgment takes place (perhaps because of some communication error or the device is not responding), a No Acknowledge (NACK) bit is sent.

Data bits

Data bits are the 8-bit sequence carrying actual information passed between the master and the slave, sending or receiving. The bits refer to the data byte being sent or received. After receiving every byte, the receiver sends an ACK bit for confirmation.

START and STOP conditions for I2C

I2C communications are initiated and terminated with distinct signals called Start condition and Stop conditions. These conditions indicate the beginning and the end of data transmission.



START Condition

A Start condition is initiated by the master when it pulls down the SDA while keeping SCL high. This special action indicates to all devices on the bus that they are about to initiate some kind of communication between them. It indicates that the master is about to initiate a transaction and is signaling all the devices connected to him that it is time to prepare themselves for either receiving or sending data.

STOP Condition

When the master leaves SDA high with SCL still high, this is a Stop condition. This indicates that the communication session is over. In essence, the master has given a signal that he has completed the transfer of data and now other devices can take over the bus for their communications.

Overall, I2C, or Inter-Integrated Circuit, is a communication protocol that allows devices to exchange data using only two wires: SDA (Serial Data), a bi-directional line for data transfer, and SCL (Serial Clock), which synchronizes the communication. The master initiates communication by generating a Start condition, sending the 7-bit slave address along with a Read/Write bit to indicate the data direction. Data is transmitted in 8-bit packets, each followed by an ACK/NACK bit for acknowledgment. The transmission concludes with a Stop condition. This simple yet efficient protocol enables multiple devices to share a bus and reliably transfer data in embedded systems.

Block Diagrams

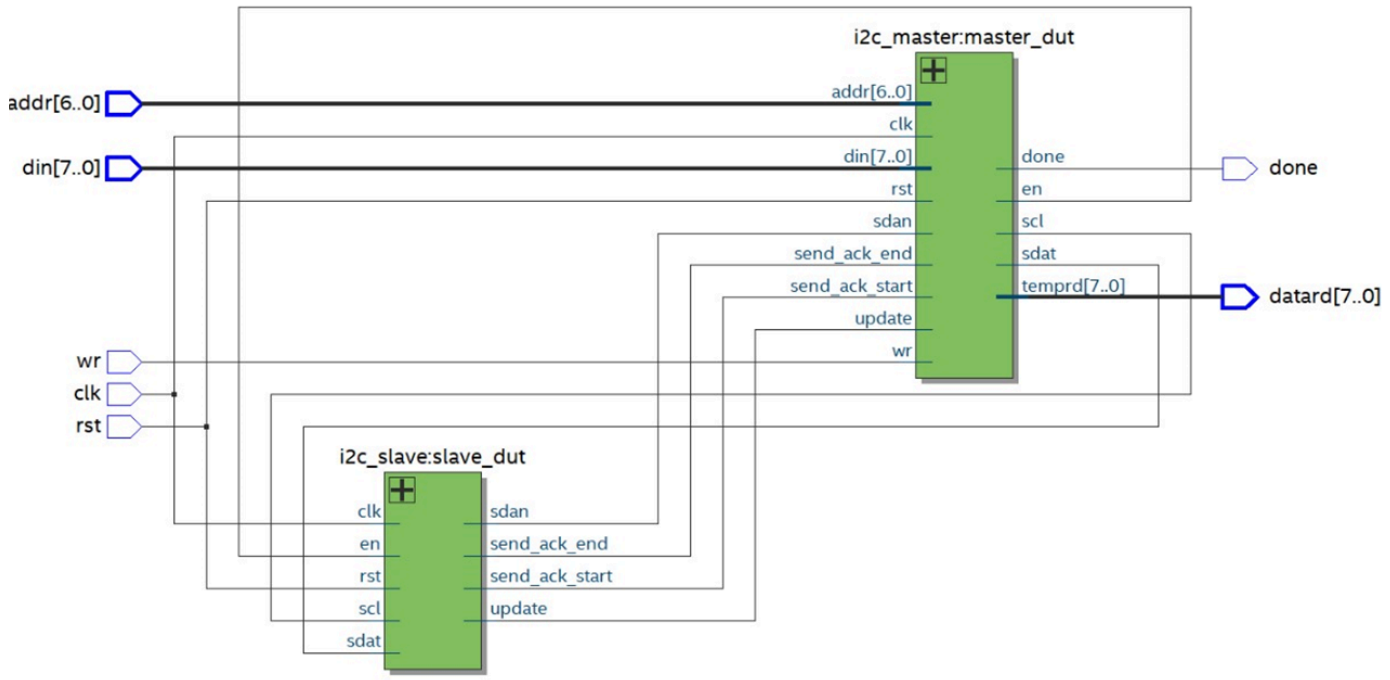


Fig: I2C Protocol - Block Diagram of Implemented Design

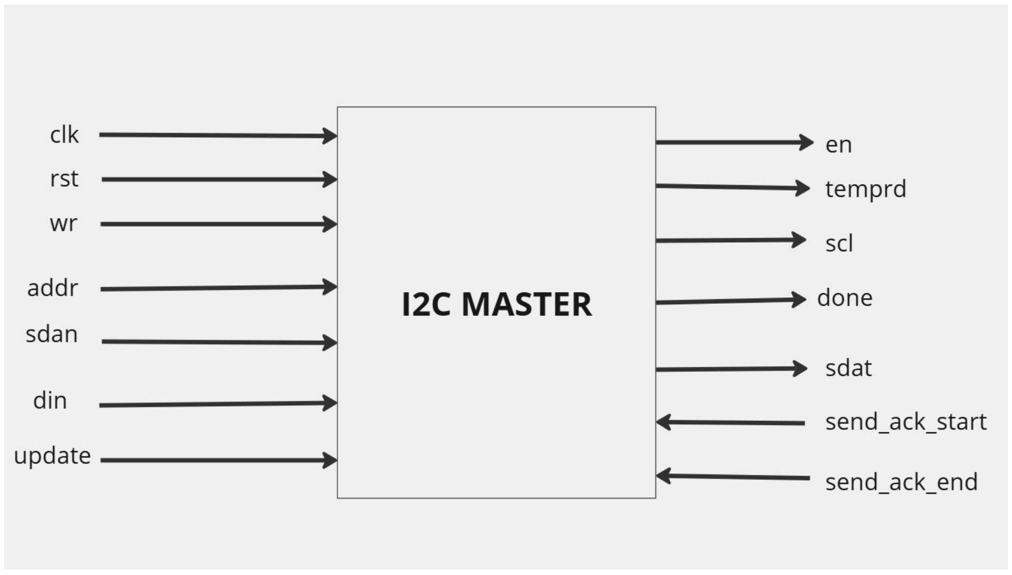


Fig: I2C master block diagram

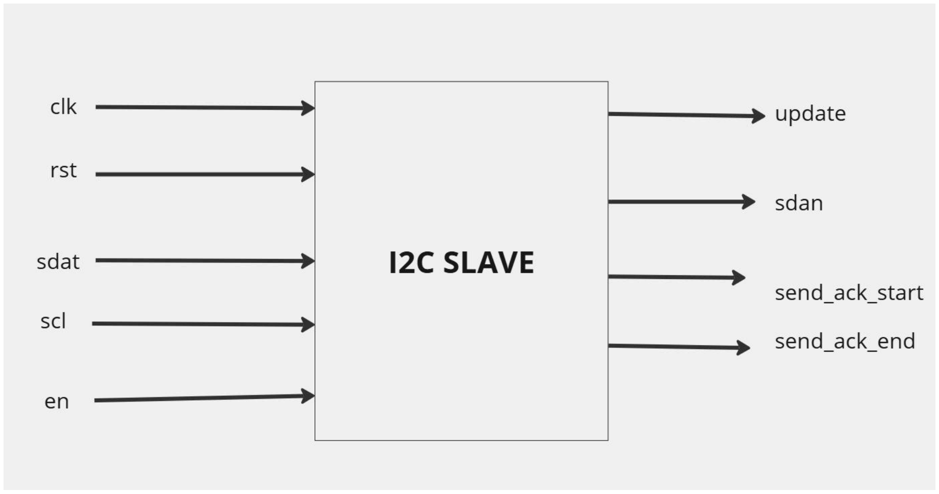


Fig: I2C slave block diagram

Finite State Diagrams

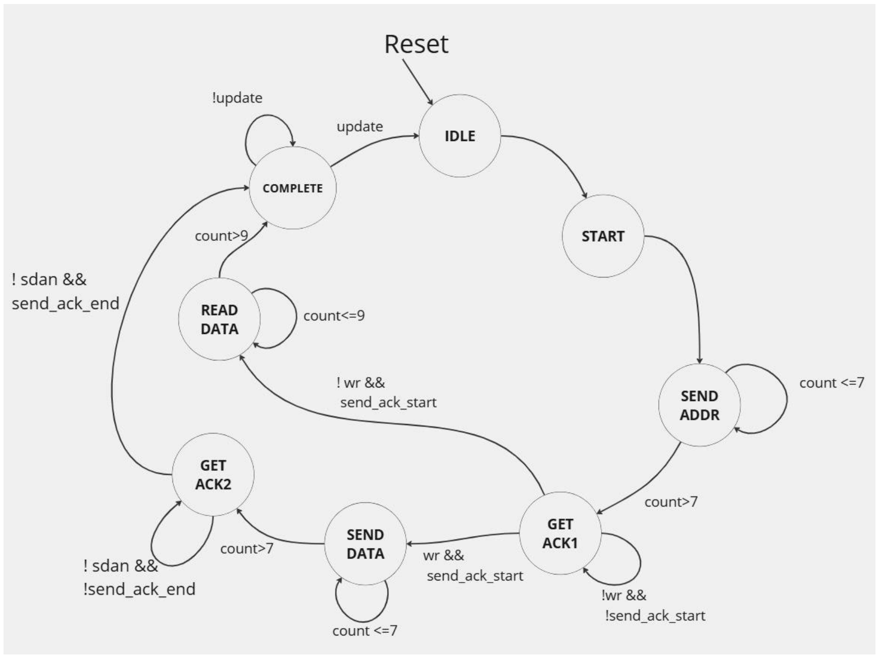


Fig: I2C Master FSM diagram

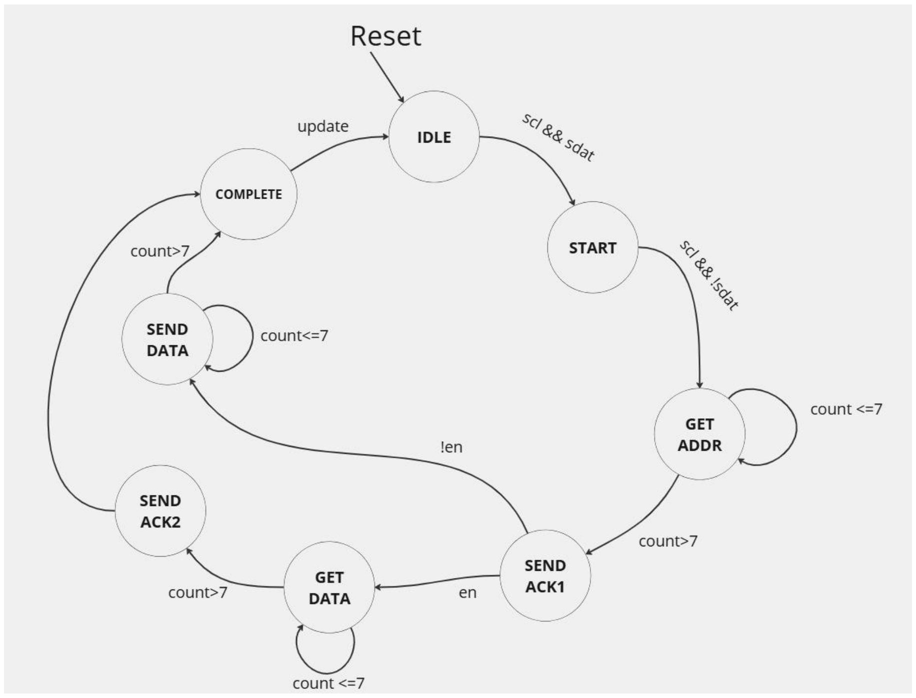


Fig: I2C Slave FSM diagram

Dynamic Simulation

We implemented the design code for the I2C communication protocol and created a simplified testbench covering various test cases to verify its functionality. The I2C communication was successfully executed, and the expected results were obtained. Below are the details of the results achieved during the simulation:

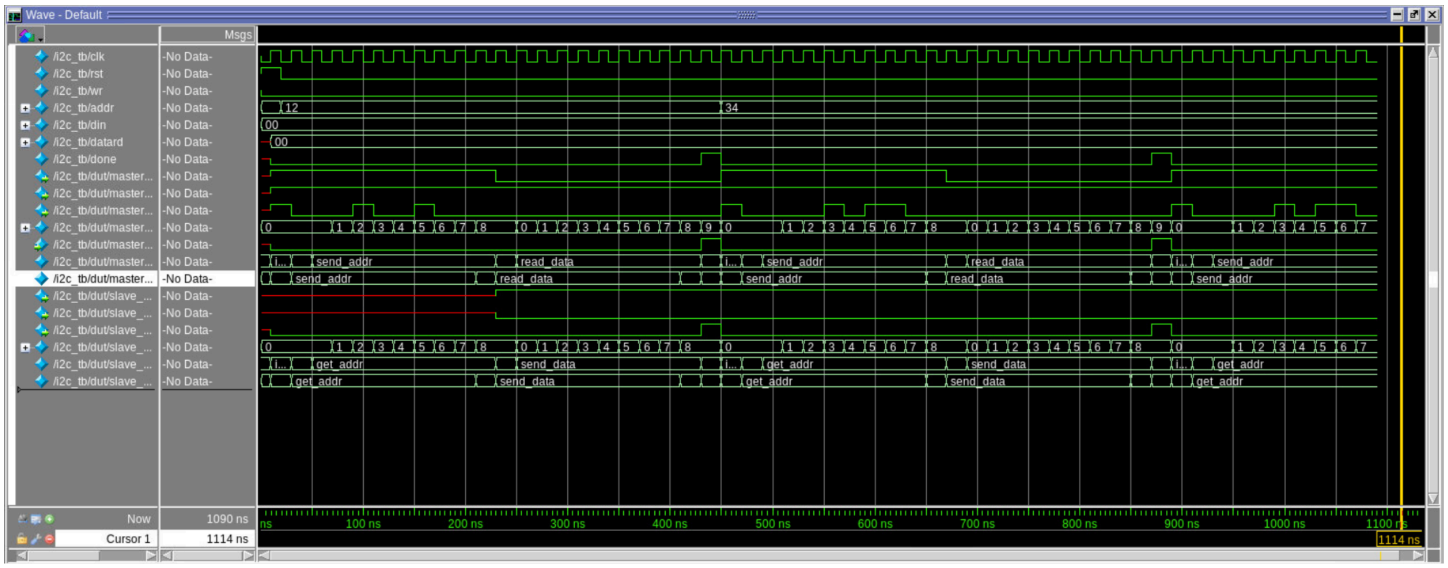


Fig: I2C read operation waveform

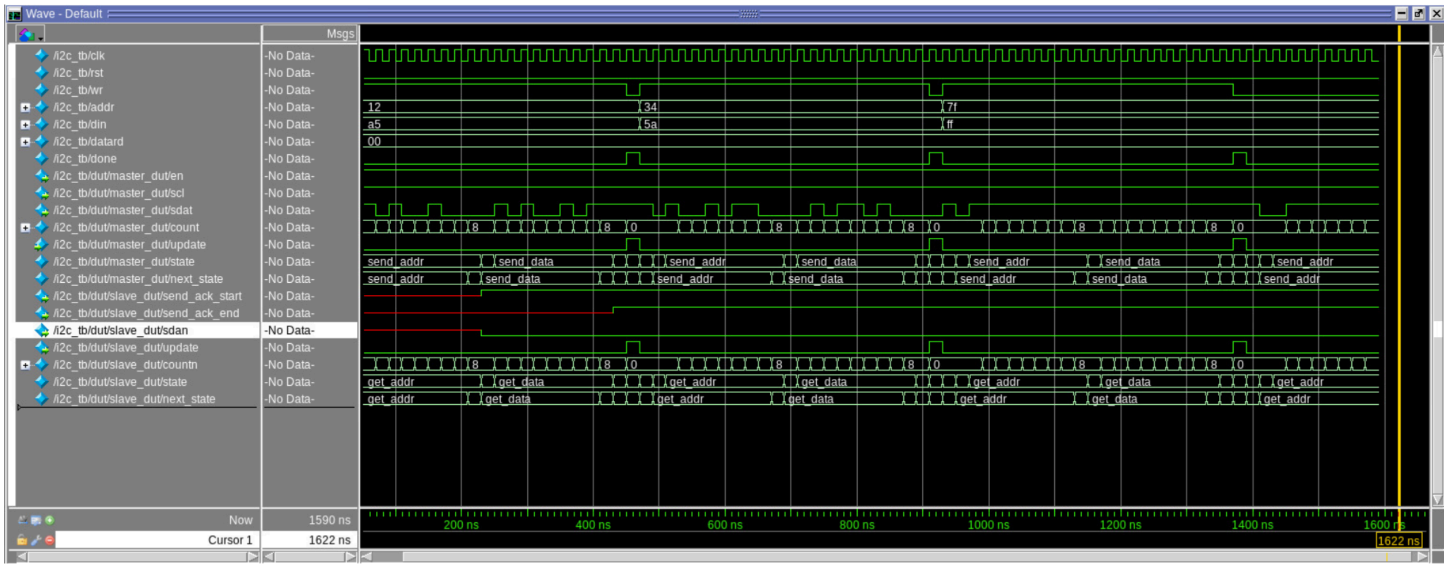

Wave - Default
Msgs
/i2c_tb/clk
/i2c_tb/rst
/i2c_tb/wr
/i2c_tb/addr
/i2c_tb/din
/i2c_tb/datard
/i2c_tb/done
/i2c_tb/dut/master_dut/en
/i2c_tb/dut/master_dut/scl
/i2c_tb/dut/master_dut/sdat
/i2c_tb/dut/master_dut/count
/i2c_tb/dut/master_dut/update
/i2c_tb/dut/master_dut/state
/i2c_tb/dut/master_dut/next_state
/i2c_tb/dut/slave_dut/send_ack_start
/i2c_tb/dut/slave_dut/send_ack_end
/i2c_tb/dut/slave_dut/sdan
/i2c_tb/dut/slave_dut/update
/i2c_tb/dut/slave_dut/countn
/i2c_tb/dut/slave_dut/state
/i2c_tb/dut/slave_dut/next_state
-No Data-
12
34
7f
a5
5a
ff
00
send_addr
send_data
get_addr
get_data
Now 1590 ns
Cursor 1 1622 ns
200 ns
400 ns
600 ns
800 ns
1000 ns
1200 ns
1400 ns
1600 ns

Fig: I2C write operation waveform

To implement the write operation for the verification of I2C protocol, we have initialized the memory contents in the design though it is not synthesizable.

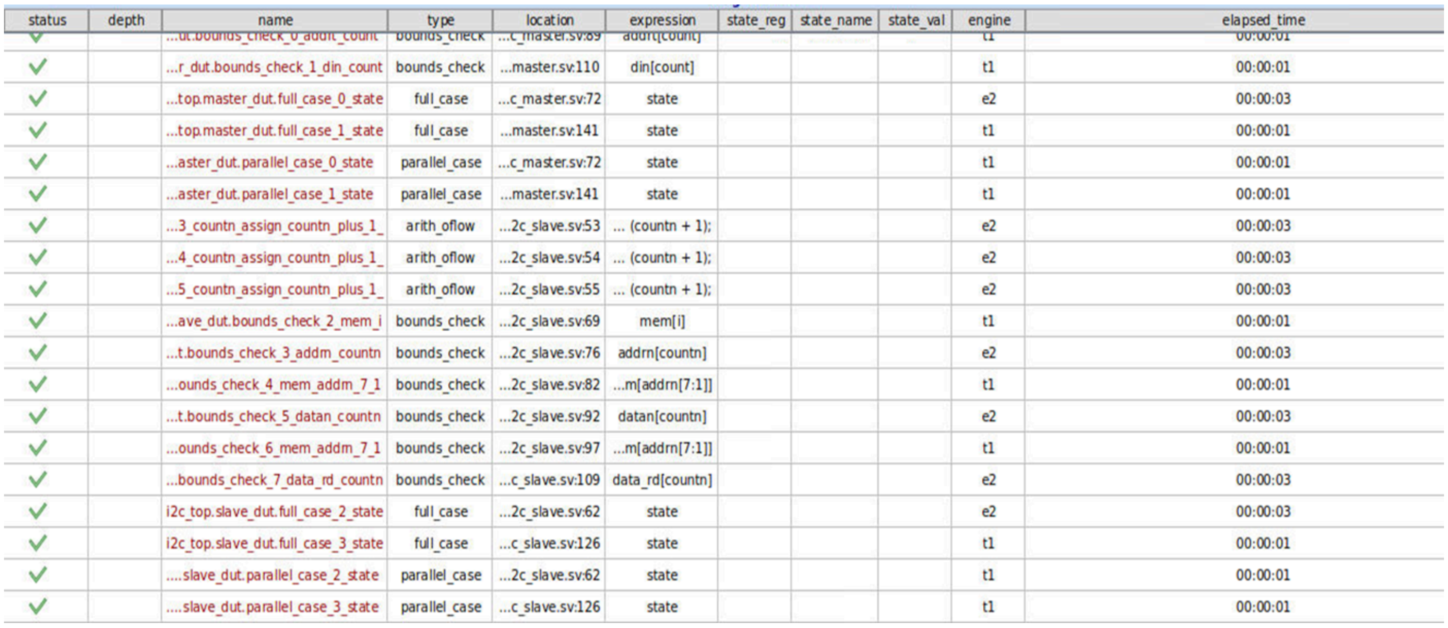
Formal Verification

Formal verification is performed to ensure the correctness and reliability of a design by mathematically proving that it satisfies its specified properties under all possible conditions. Unlike traditional testing methods, which rely on specific test cases, formal verification exhaustively checks every possible state and behavior of a system. This is particularly important for safety-critical applications, such as in
aerospace, medical devices, or security systems, where errors can have severe consequences. By using formal methods, designers can guarantee that their system will always behave as intended, offering a higher level of confidence in its correctness.

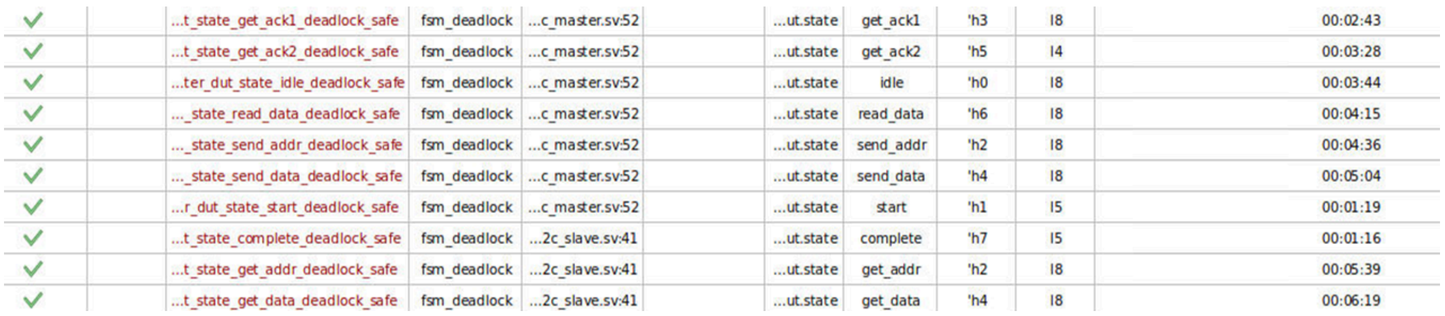
AEP Application

The AEP app in VC Formal is used as an analysis tool for auto-checking a design for out-of-bound arrays, arithmetic overflow, X-assignments, simultaneous set/reset, full case, parallel case, muti-driver and conflicting bus, and floating bus checks. Without the AEP app, these checks would each require their own dedicated simulation tests, consuming lots of time and energy.

We have run the formal verification using the AEP application and here are the following results:



status	depth	name	type	location	expression	state_reg	state_name	state_val	engine	elapsed_time
✓		...ut.bounds_check_0_addr_count	bounds_check	...c_master.sv:89	addr[count]				t1	00:00:01
✓		...r_dut.bounds_check_1_din_count	bounds_check	...master.sv:110	din[count]				t1	00:00:01
✓		...top.master_dut.full_case_0_state	full_case	...c_master.sv:72	state				e2	00:00:03
✓		...top.master_dut.full_case_1_state	full_case	...master.sv:141	state				t1	00:00:01
✓		...aster_dut.parallel_case_0_state	parallel_case	...c_master.sv:72	state				t1	00:00:01
✓		...aster_dut.parallel_case_1_state	parallel_case	...master.sv:141	state				t1	00:00:01
✓		...3_countn_assign_countn_plus_1_	arith_oflow	...2c_slave.sv:53	... (countn + 1);				e2	00:00:03
✓		...4_countn_assign_countn_plus_1_	arith_oflow	...2c_slave.sv:54	... (countn + 1);				e2	00:00:03
✓		...5_countn_assign_countn_plus_1_	arith_oflow	...2c_slave.sv:55	... (countn + 1);				e2	00:00:03
✓		...ave_dut.bounds_check_2_mem_i	bounds_check	...2c_slave.sv:69	mem[i]				t1	00:00:01
✓		...t.bounds_check_3_addrn_countn	bounds_check	...2c_slave.sv:76	addrn[countn]				e2	00:00:03
✓		...ounds_check_4_mem_addrn_7_1	bounds_check	...2c_slave.sv:82	...m[addrn[7:1]]				t1	00:00:01
✓		...t.bounds_check_5_datan_countn	bounds_check	...2c_slave.sv:92	datan[countn]				e2	00:00:03
✓		...ounds_check_6_mem_addrn_7_1	bounds_check	...2c_slave.sv:97	...m[addrn[7:1]]				t1	00:00:01
✓		...bounds_check_7_data_rd_countn	bounds_check	...c_slave.sv:109	data_rd[countn]				e2	00:00:03
✓		i2c_top.slave_dut.full_case_2_state	full_case	...2c_slave.sv:62	state				e2	00:00:03
✓		i2c_top.slave_dut.full_case_3_state	full_case	...c_slave.sv:126	state				t1	00:00:01
✓		....slave_dut.parallel_case_2_state	parallel_case	...2c_slave.sv:62	state				t1	00:00:01
✓		....slave_dut.parallel_case_3_state	parallel_case	...c_slave.sv:126	state				t1	00:00:01

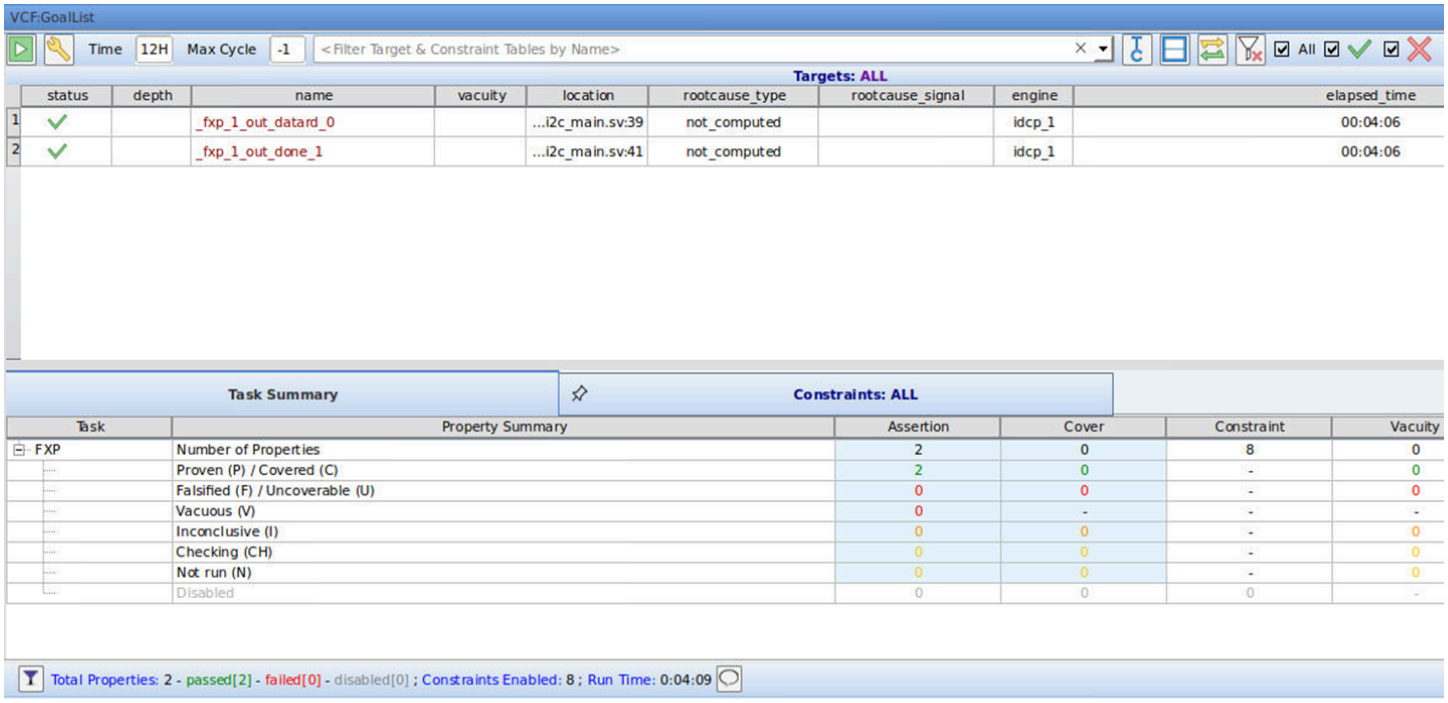


✓		...t_state_get_ack1_deadlock_safe	fsm_deadlock	...c_master.sv:52		...ut.state	get_ack1	'h3	l8	00:02:43
✓		...t_state_get_ack2_deadlock_safe	fsm_deadlock	...c_master.sv:52		...ut.state	get_ack2	'h5	l4	00:03:28
✓		...ter_dut_state_idle_deadlock_safe	fsm_deadlock	...c_master.sv:52		...ut.state	idle	'h0	l8	00:03:44
✓		..._state_read_data_deadlock_safe	fsm_deadlock	...c_master.sv:52		...ut.state	read_data	'h6	l8	00:04:15
✓		..._state_send_addr_deadlock_safe	fsm_deadlock	...c_master.sv:52		...ut.state	send_addr	'h2	l8	00:04:36
✓		..._state_send_data_deadlock_safe	fsm_deadlock	...c_master.sv:52		...ut.state	send_data	'h4	l8	00:05:04
✓		...r_dut_state_start_deadlock_safe	fsm_deadlock	...c_master.sv:52		...ut.state	start	'h1	l5	00:01:19
✓		...t_state_complete_deadlock_safe	fsm_deadlock	...2c_slave.sv:41		...ut.state	complete	'h7	l5	00:01:16
✓		...t_state_get_addr_deadlock_safe	fsm_deadlock	...2c_slave.sv:41		...ut.state	get_addr	'h2	l8	00:05:39
✓		...t_state_get_data_deadlock_safe	fsm_deadlock	...2c_slave.sv:41		...ut.state	get_data	'h4	l8	00:06:19

FXP Application

The Formal X-propagation (FXP) app in VC Formal is used to check for and trace back an unknown signal value (X). The app detects X propagation through a design and guides you to the failed property that is the source of this signal by using the Verdi schematic and waveform within VC Formal.

We started exploring the design by running the completely automated apps, AEP and FXP, we ran them on the top module with the filelist, as well as on the individual modules for better signal visibility. In both cases, we didn't get much information using those automated techniques.


VCF:GoalList

Time 12H Max Cycle -1 <Filter Target & Constraint Tables by Name>

Targets: ALL

	status	depth	name	vacuity	location	rootcause_type	rootcause_signal	engine	elapsed_time
1	✓		_fxp_1_out_datard_0		...i2c_main.sv:39	not_computed		idcp_1	00:04:06
2	✓		_fxp_1_out_done_1		...i2c_main.sv:41	not_computed		idcp_1	00:04:06

Task Summary | Constraints: ALL

Task	Property Summary	Assertion	Cover	Constraint	Vacuity
FXP	Number of Properties	2	0	8	0
	Proven (P) / Covered (C)	2	0	-	0
	Falsified (F) / Uncoverable (U)	0	0	-	0
	Vacuous (V)	0	-	-	-
	Inconclusive (I)	0	0	-	0
	Checking (CH)	0	0	-	0
	Not run (N)	0	0	-	0
	Disabled	0	0	0	-

Total Properties: 2 - passed[2] - failed[0] - disabled[0] ; Constraints Enabled: 8 ; Run Time: 0:04:09

FPV Application

The FPV application is utilized for verifying a DUT by proving properties. These properties can be created by users or provided by commercial AIPs for interface protocol or function blocks. The app uses a range of powerful VC Formal engines to thoroughly prove or disprove a property. If an assertion fails, the FPV app will generate a counter-example to demonstrate a violating trace. However, there may be cases where it is impossible to definitively prove or disprove a property. The FPV app would then output the status of properties such as assertion status, assumption status, and cover status. The assertion status could be proven, falsified, vacuous, witness-coverable or inconclusive.

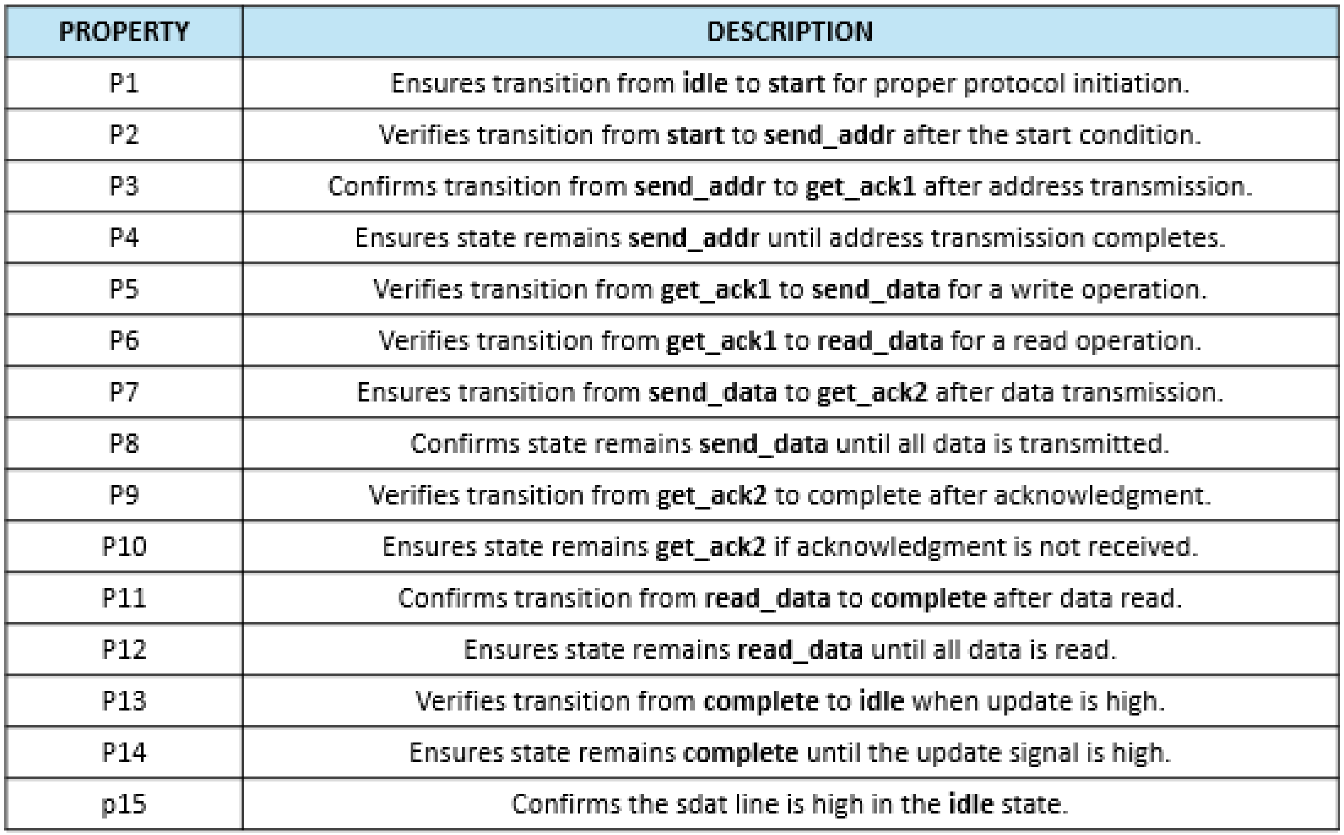
Assumptions for I2C master

Property	Description
p1	When the system is in the start state, sdat (SDA line) must be pulled low, and scl (SCL line) must remain high.
p2	During the send_addr state, the signal wr (write control signal) must remain stable (unchanged).
p3	During the send_data state, the signal din (data input) must remain stable (unchanged).

Assumptions for I2C slave

Property	Description
p1	Ensures that during the states get_data and send_data, the values of datan and data_rd remain within the 8-bit range (<= 8'hFF).
p2	Ensures that whenever the system is in the start state, the next_state must transition to get_addr.
p3	Ensures that during the get_addr and get_data states, the signal addrn (address register or variable) remains stable (unchanged).

Assertions for I2C master

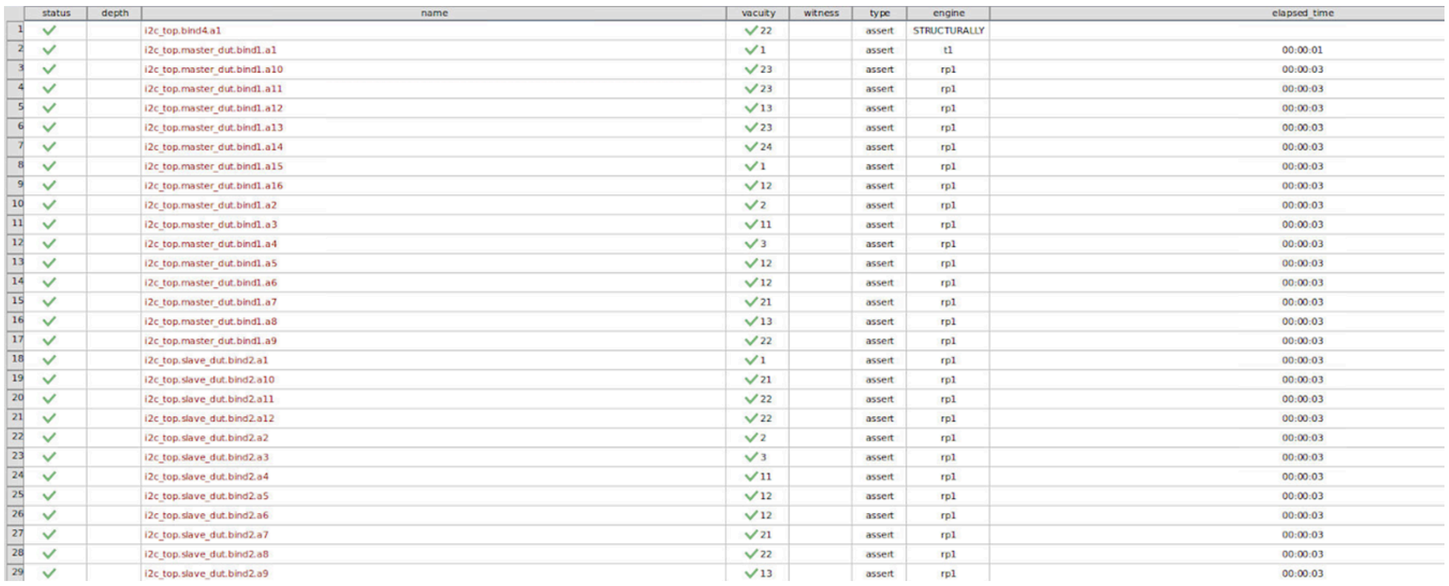


PROPERTY	DESCRIPTION
P1	Ensures transition from **idle** to **start** for proper protocol initiation.
P2	Verifies transition from **start** to **send_addr** after the start condition.
P3	Confirms transition from **send_addr** to **get_ack1** after address transmission.
P4	Ensures state remains **send_addr** until address transmission completes.
P5	Verifies transition from **get_ack1** to **send_data** for a write operation.
P6	Verifies transition from **get_ack1** to **read_data** for a read operation.
P7	Ensures transition from **send_data** to **get_ack2** after data transmission.
P8	Confirms state remains **send_data** until all data is transmitted.
P9	Verifies transition from **get_ack2** to complete after acknowledgment.
P10	Ensures state remains **get_ack2** if acknowledgment is not received.
P11	Confirms transition from **read_data** to **complete** after data read.
P12	Ensures state remains **read_data** until all data is read.
P13	Verifies transition from **complete** to **idle** when update is high.
P14	Ensures state remains **complete** until the update signal is high.
p15	Confirms the sdat line is high in the **idle** state.

Assertions for I2C slave

PROPERTY	DESCRIPTION
P1	Ensures transition from **idle** to **start** for proper protocol initiation.
P2	Ensures transition from **start** to **get_addr** when a start condition is detected.
P3	Ensures the state remains **get_addr** while the address bits are being received (count ≤ 7).
P4	Ensures transition from **get_addr** to **send_ack1** after all address bits are received (count > 7).
P5	Ensures transition from **send_ack1** to **get_data** when enabled (en signal is high).
P6	Ensures transition from **send_ack1** to **send_data** when disabled (en signal is low).
P7	Ensures transition from **get_data** to **send_ack2** after all data bits are received (count > 7).
P8	Ensures transition from **send_ack2** to complete after sending the acknowledgment.
P9	Ensures the state remains **send_data** while data bits are being sent (count ≤ 7).
P10	Ensures transition from **send_data** to **complete** after all data bits are sent (count > 7).
P11	Ensures transition from **complete** to **idle** for resetting the transaction.
P12	Ensures the update signal is active only when the slave is in the **complete** state.

All the Assertions properties check:



	status	depth	name	vacuity	witness	type	engine	elapsed_time
1	✓		i2c_top.bind4.a1	✓22		assert	STRUCTURALLY	
2	✓		i2c_top.master_dut.bind1.a1	✓1		assert	t1	00:00:01
3	✓		i2c_top.master_dut.bind1.a10	✓23		assert	rp1	00:00:03
4	✓		i2c_top.master_dut.bind1.a11	✓23		assert	rp1	00:00:03
5	✓		i2c_top.master_dut.bind1.a12	✓13		assert	rp1	00:00:03
6	✓		i2c_top.master_dut.bind1.a13	✓23		assert	rp1	00:00:03
7	✓		i2c_top.master_dut.bind1.a14	✓24		assert	rp1	00:00:03
8	✓		i2c_top.master_dut.bind1.a15	✓1		assert	rp1	00:00:03
9	✓		i2c_top.master_dut.bind1.a16	✓12		assert	rp1	00:00:03
10	✓		i2c_top.master_dut.bind1.a2	✓2		assert	rp1	00:00:03
11	✓		i2c_top.master_dut.bind1.a3	✓11		assert	rp1	00:00:03
12	✓		i2c_top.master_dut.bind1.a4	✓3		assert	rp1	00:00:03
13	✓		i2c_top.master_dut.bind1.a5	✓12		assert	rp1	00:00:03
14	✓		i2c_top.master_dut.bind1.a6	✓12		assert	rp1	00:00:03
15	✓		i2c_top.master_dut.bind1.a7	✓21		assert	rp1	00:00:03
16	✓		i2c_top.master_dut.bind1.a8	✓13		assert	rp1	00:00:03
17	✓		i2c_top.master_dut.bind1.a9	✓22		assert	rp1	00:00:03
18	✓		i2c_top.slave_dut.bind2.a1	✓1		assert	rp1	00:00:03
19	✓		i2c_top.slave_dut.bind2.a10	✓21		assert	rp1	00:00:03
20	✓		i2c_top.slave_dut.bind2.a11	✓22		assert	rp1	00:00:03
21	✓		i2c_top.slave_dut.bind2.a12	✓22		assert	rp1	00:00:03
22	✓		i2c_top.slave_dut.bind2.a2	✓2		assert	rp1	00:00:03
23	✓		i2c_top.slave_dut.bind2.a3	✓3		assert	rp1	00:00:03
24	✓		i2c_top.slave_dut.bind2.a4	✓11		assert	rp1	00:00:03
25	✓		i2c_top.slave_dut.bind2.a5	✓12		assert	rp1	00:00:03
26	✓		i2c_top.slave_dut.bind2.a6	✓12		assert	rp1	00:00:03
27	✓		i2c_top.slave_dut.bind2.a7	✓21		assert	rp1	00:00:03
28	✓		i2c_top.slave_dut.bind2.a8	✓22		assert	rp1	00:00:03
29	✓		i2c_top.slave_dut.bind2.a9	✓13		assert	rp1	00:00:03

Formal verification test plan

Functions to be validated	Methodology used	Further description	signals
no x propagation on data read	FXP	all test cases for the signal temprd are covered and edge cases are tested to validate that no ‘X’ values propagated	assign datard = temprd
no x propagation on done signal	FXP	all test cases for the signal temp_done are covered and edge cases are tested to validate that no ‘X’ values propagated	assign done = temp_done
i2c_master fsm no deadlock	AEP	fsm_deadlock conditions checked	master_dut.state
i2c_slave fsm no deadlock	AEP	fsm_deadlock conditions checked	slave_dut.state
i2c_master arithmetic overflow check and i2c_slave arithmetic overflow	AEP	arithmetic overflow checked	count <= count +1
i2c_master bounds check	AEP	bounds checked	addrt(count), din(count), mem[i],
i2c_master full case	AEP	full case checked	state
i2c_master parallel case and i2c_slave parallel case	AEP	parallel case checked	state
i2c_master and i2c_slave fsm	FPV	Verified the fsm states and transitions	
output logic based on the states	FPV	verified that state produces the expected output according to specs	

Contributions

Team members	Contribution
Srikar Varma Datla	Dynamic simulation, Formal verification(AEP app), Report
Kumar Durga Manohar Karna	Dynamic simulation, Formal verification(AEP app), Report
Nivedita Boyina	Formal verification(FXP, FPV app), Writing assertions and assumptions, ppt
Pooja Satpute	Formal verification(FXP, FPV app), Writing assertions and assumptions, ppt

Challenges Faced

- **Debugging Assertion Failures:**
  Identifying root causes of assertion violations required extensive signal tracing and analysis.
- **Synchronization Issues:**
  Ensuring proper timing between SDA and SCL signals was critical for accurate data transfer.
- **Formal Verification Complexity:**
  Managing state explosion in formal verification was challenging, especially with multiple properties.
- **State Transition Errors:**
  Validating seamless transitions between states required careful design and debugging.

References

- A Basic Guide to I2C – Texas Instruments https://www.ti.com/lit/pdf/sbaa565#:~:text=I2C%20is%20a%20two%2Dwire,and%20receive%20commands%20and%20data.
- I2C Protocol Verification –EDA Playground https://www.edaplayground.com/x/rSCn
- CummingsSNUG2016SV_SVA_Best_Practices - http://www.sunburstdesign.com/papers/CummingsSNUG2016SV_SVA_Best_Practices.pdf
- Verification Academy – Siemens https://verificationacademy.com/topics/assertions/
- ECE 560 Professor Venkatesh Patil Lecture Slides
- Mohamed Ghonim Tutorial for Synopsys VC Formal Tutorials
- ChatGPT, Perplexity AI