I²C Master

Hareth Al-Jomaa – 214
Western Norway University of Applied Sciences

Author Note

This work was completed as part of the ELE113 course, *HW/SW System Construction*, during the Spring 2025 semester at the Western Norway University of Applied Sciences.

Table of Contents

Abstract

This report presents the design, simulation, and hardware validation of a synthesizable I²C Master controller in VHDL targeting FPGA systems. The controller implements start/stop condition handling and open-drain SDA arbitration in accordance with the I²C specification, with parameterized SCL generation via a clock enable scheme [1]. A finite state machine orchestrates single-master transactions using five handshake signals (enable, continue, ready, done, busy), supporting conditional repeated writes without repeated start sequences. Timing-compliant SDA synchronization logic and clock-gated SCL output ensure standard-mode compatibility.

Functional verification using UVVM-based and directed testbenches in Questa confirmed protocol-compliant behavior across all FSM paths. FPGA hardware tests on the DE2-115 board exposed divergence from simulation, including premature READY assertion and ACK misalignment. Despite these interface-level issues, the core module meets I²C timing margins, offering a compact, synthesizable foundation for future protocol expansion.

1. System Analysis

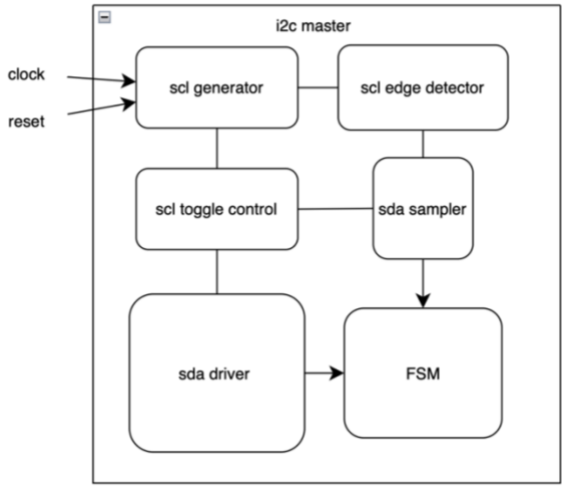
The I²C Master core is structured around four principal architectural blocks, each responsible for a specific subset of protocol logic.

I. A modular control signal block generates internal flags such as start_transaction, transaction_pending, stop_transaction, and byte_ready, with each signal synthesized in its own process to preserve isolation and deterministic behavior.
II. The SCL generation is performed by a parameterized clocking unit that produces a synchronous output via a gated enable.
III. The SDA handling splits responsibilities into two distinct units, sampling and driving.
IV. Finally, a centralized finite state machine (FSM) that governs protocol sequencing, issuing internal control signals and evaluating handshake outputs (busy, done, ack_error).

The I²C Master supports the generation of START and STOP conditions, data byte transfers with acknowledgment checking, and externally synchronized transaction continuation. The module does not support repeated START conditions. Instead, higher-level control logic manages sequencing through ready and continue signals. These signals control whether the FSM moves to stop after the wait state or loops back to write a new byte.

The I²C serial lines (SDA and SCL) adhere to the open drain signaling standard specified by the I²C protocol [1]. The SDA line is implemented as a bidirectional signal managed explicitly through two separate processes: one dedicated exclusively to driving the SDA line, based on FSM states and timing conditions, and another solely responsible for sampling incoming SDA data synchronized to SCL edges. This distinct separation of driving and sampling logic, ensures strict delineation between output control and input monitoring.

The FSM itself deliberately avoids direct manipulation of SDA and SCL signals, instead relying exclusively on internal control signals that govern the separate SDA driver and sampler processes. An optional multi-stage shift registers delay mechanism on the SDA output signal is included to ensure adequate hold time compliance



for higher speed operation, though it remains inactive under standard timing conditions.

1.2 Timing and Clock Control.

The timing of the I²C clock signal (SCL) is derived directly from the system clock using a deterministic clock divider. A parameterized constant defines the number of clock cycles required for each half-period of SCL, currently supporting operation at standard (100 kHz) I²C speed. A gated enable signal (scl_enable) is asserted once per configured period, toggling the internal SCL signal (scl_internal) synchronously with the system clock.

To support correct signal alignment and edge-sensitive transitions, rising and falling edges of SCL are explicitly detected through a sampled copy of the previous SCL value (scl_internal_prev). These detected edges (scl_risingedge, scl_fallingedge) are used throughout the design to coordinate FSM state transitions and to control the timing of SDA sampling and output driving.

This approach ensures fully synchronous clock generation and phase alignment with the I²C protocol's timing requirements [2].

1.3 Finite State Machine (FSM)

The I²C Master is controlled by a synchronous finite state machine (FSM) that governs the high-level sequencing of I²C protocol operations. The FSM transitions occur exclusively on either the rising edge or level triggers of the internally generated SCL signal, maintaining alignment with I²C timing semantics[2].

The FSM includes explicit states for each protocol phase: *idle_state, start_state, address_state, address_ack_state, write_data_state, read_data_state, wait_write_state, slave_ack_state, master_ack_state, prep_stop_state and stop_state*. Transitions are determined by control signals such as enable, continue, and stop_signal, as well as sampled line values like SDA for ACK/NACK detection.

ACK detection is performed during the SCL high phase while in the address_ack_state and slave_ack_state, with the FSM checking the SDA line directly. A failed acknowledgment sets an internal error flag (ack_error) and transitions the FSM to the stop condition. The FSM also supports both write and read paths, reconfiguring SDA direction accordingly.

1.4 System Integration

The I²C Master is encapsulated within a hierarchical top-level design, composed of three components a register interface, master and glue logic (i.e. top-level file). The entire I²C module was also integrated into an embedded SoC through Intel's Platform Designer [3], interfacing with the Nios II processor over the Simple Bus Interface (SBI). This allows the MCU to initiate and monitor I²C operations via standard register accesses[4].

The register bank, implemented in i2c_registerbank, decodes four addressable registers:

i. a control register for issuing transfer commands (enable, stop, read/write, continue),
ii. a write register combining the 7-bit slave address and 8-bit data payload,
iii. a status register reflecting handshake and error flags (busy, done, ack_error, ready),
iv. and a read register used to retrieve received data once a transaction completes.

Control signals are latched synchronously on write access to address 0x00, while the write data and slave address are stored on access to address 0x04. The read back path is fully combinational. Supporting predictable writes and immediate read-back of status. Output signals from the register bank feed directly into the i2c_master module.

This modular structure explicitly separates control plane and data plane concerns. The FSM and bit-level I²C protocol logic in i2c_master operate independently of the register interface, while the register bank acts as the software facing configuration shell. This clean partitioning improves timing analysis, simplifies integration into system on chip platforms, and ensures full traceability between software

commands and hardware actions. While conceptually similar to designs such as Intel's Avalon I²C Master IP [1], this implementation prioritizes transparency and verification tractability over configurability.

2 Testing

The verification plan was crafted to ensure comprehensive functional and temporal correctness of the I²C Master core and its surrounding system architecture. The plan applied a multi-tiered approach to validate both nominal and edge-case behavior**.** At the module level, the focus was on validating the internal finite state machine (FSM), signal sequencing, data integrity, and protocol phase handling under both standard and adverse conditions. This included assertion of START and STOP conditions, accurate sampling/driving of SDA, and proper generation of acknowledgments.

At the system level, verification extended to the integration interface between the core and its register bank, where emphasis was placed on ensuring correct interpretation of writes, status feedback, and synchronization across transaction boundaries. The interaction between software-visible control signals and hardware-driven state transitions was evaluated to ensure deterministic and observable behavior.

Beyond functional correctness, the plan incorporated strict temporal validation. Waveforms were analyzed to verify conformance with protocol requirements at 100 kHz [2]. The plan also accounted for robustness by intentionally injecting NACK conditions, simulating slave misbehavior, and evaluating the core's response to premature STOP conditions and interrupted sequences.

Collectively, the test plan established a rigorous framework, serving as the foundation for simulation and hardware testing.

2.1 Testbench Methodology and Tools

Simulation was performed in Questa Sim using a 50 MHz time base (20 ns resolution)[5]. Module-level testbenches were developed in plain VHDL to directly stimulate and observe the behavior of the i2c_master core. These testbenches instantiated minimal slave models and manually toggled control inputs (enable, continue, stop_signal) to trigger transactions and assess protocol correctness.

The second phase introduced UVVM-based testbenches for system level testing, focusing on register-level access [6]. These testbenches used reusable transaction procedures and UVVM utility to script stimulus sequences, monitor response timing, and verify synchronization between software-accessible

registers and the I²C core behavior. UVVM also enabled clean sequencing of dependent read-write operations and better modularity in coverage tracking.

Both phases used a behavioral slave model that mimicked real-world I²C slave behavior, including ACK/NACK responses and data delivery.

SCL waveform characteristics were validated by measuring tLOW and tHIGH directly in the waveform viewer. Manual assertions and waveform cursors were used to validate protocol timing (START/STOP alignment, setup and hold conditions) without relying on formal coverage tools.

2.2 Verification Scope

Coverage was tracked using a structured test matrix.

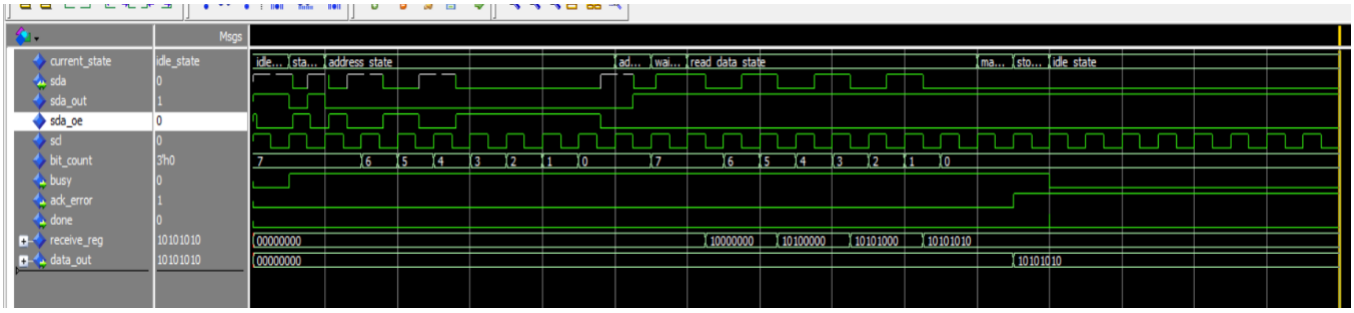
- Verifying correct generation of protocol-specific conditions, including START, STOP and repeated write sequences in accordance with the I²C standard [1].
- Validating proper handling of acknowledgment responses (ACK/NACK) from the slave during both address and data phases.
- Ensuring accurate FSM sequencing across all protocol stages, with attention to transitions triggered by enable, continue, and stop_signal control inputs.
- Observing data integrity in both transmission and reception paths, with particular focus on SDA line direction switching and SCL-aligned sampling.
- Exercising the FSM in edge-case scenarios such as back-to-back transactions, early STOP, or NACK-induced aborts.
- Monitoring SCL waveform generation at 100 kHz and verifying correct timing characteristics (tLOW, tHIGH, tSU, tHOLD) using waveform measurements [2].

Throughout all test phases, SCL and SDA outputs were reviewed to achieve glitch free behavior, high-Z release timing on SDA during ACK phases and STOP condition.

This scope ensured that the bit-level behavior of the I²C Master conformed to protocol requirements before proceeding to system integration and hardware testing.

2.3 Module-Level Testbenches

The I²C Master core was verified in isolation using three structured VHDL testbenches. These testbenches provided direct, low-level stimulation of the master's inputs and control over slave timing and behavior.

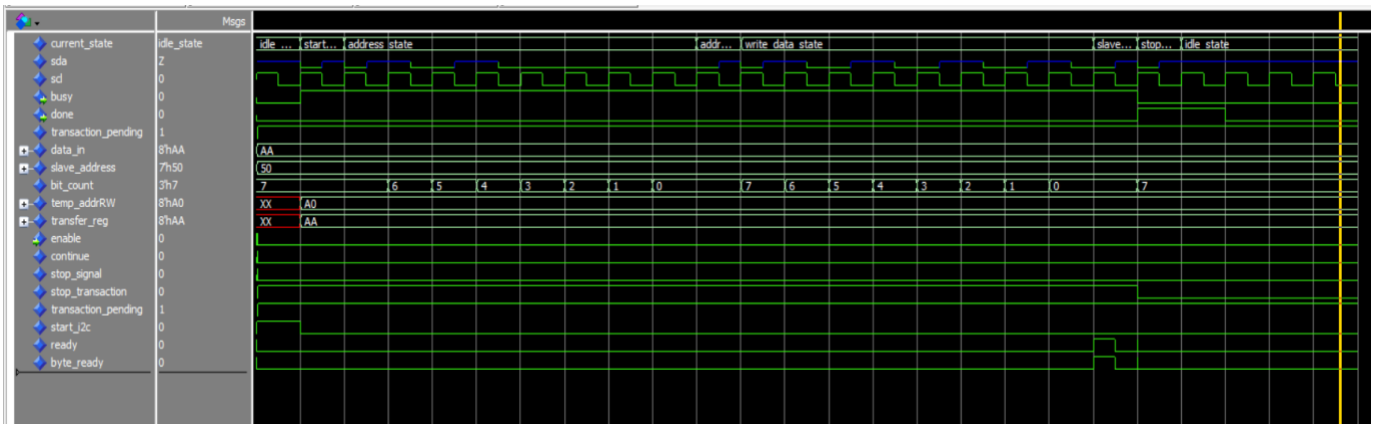


Testbench 1: Single Read Transaction with ACK

The initial testbench verified a basic I²C read operation. Upon receiving the correct address, the VHDL-based slave model responded with a fixed data byte (10101010). SCL and SDA waveforms were analyzed to confirm correct edge timing, SDA sampling during SCL high periods, and proper slave ACK detection.

The test validated FSM transition through start_state, address_state, read_data_state, master_ack_state, and stop_state. Notably, the wait_read_state was removed from the FSM. The slave model, implemented using simple VHDL constructs such as for loops and direct sda assignments, served its intended purpose. However, waveform inspection revealed non-compliant STOP conditions, unauthorized SCL toggling during the idle state and unexpected behavior from the status signal DONE. These issues were persistent across the beginning of the design and indicated flaws in signal control logic that required further debugging.

It should be noted that all testbenches used are included with the project submission. Each testbench is clearly named, commented, and structured to allow straightforward tracing of test intent, stimulus, and observed results. This enables transparent verification of the test coverage.

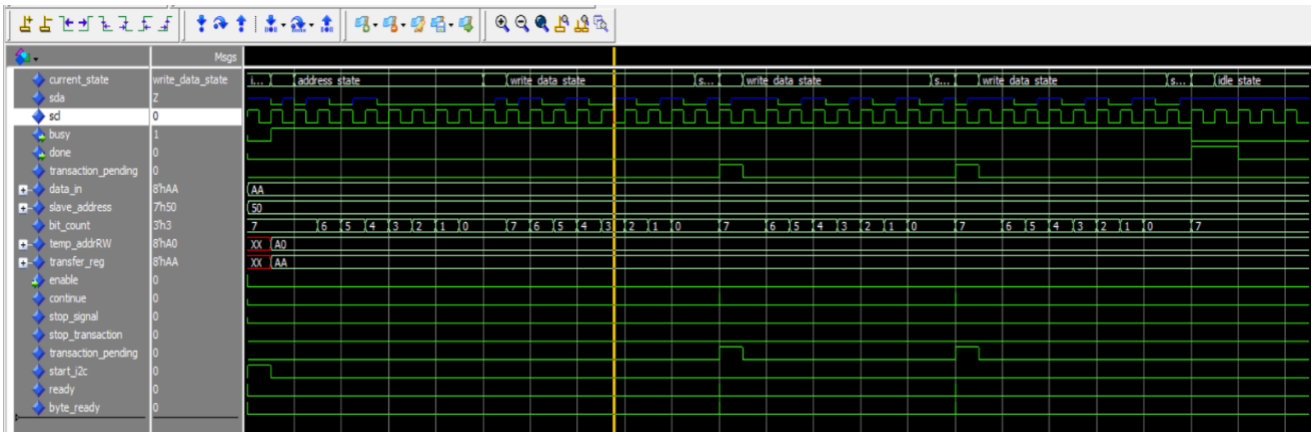


Testbench 2: Single-Byte Write Transaction with Stop Control

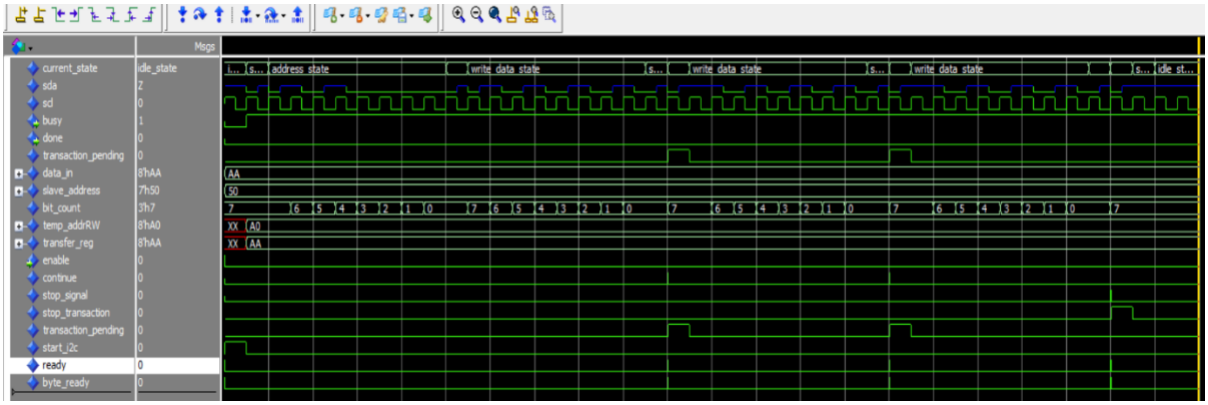
The second testbench evaluated a single byte write operation initiated concurrently with a STOP condition. Specifically, the stop_signal was asserted at the start of the transaction (latched to stop_transaction), causing the master to wait for a defined safe state namely, the slave_ack_state before terminating the sequence. Although the slave model was configured to acknowledge the transmitted byte, the FSM did not exit at the expected address_ack_state. This deviation stems from an earlier implementation ambiguity when the FSM couldn't handle both stop_transaction and the slave ACK occurring simultaneously.

Waveform analysis revealed recurring protocol violations: unauthorized toggling on the SCL line during the idle state and a non-compliant STOP condition. These issues persisted throughout the early stages of the design and are visible in all waveform outputs until the late UVVM-based validation phase.

The primary objective of this testbench was to assess the FSM's responsiveness to external control signals, particularly the newly introduced ready and byte_ready flags. Furthermore, the transaction_pending signal was observed to be latched at the assertion of enable and remained uncleared, highlighting early-stage design oversights.



Testbench 3



Testbench 4

Testbench 3 & 4: Error Injection and Back-to-Back Writes

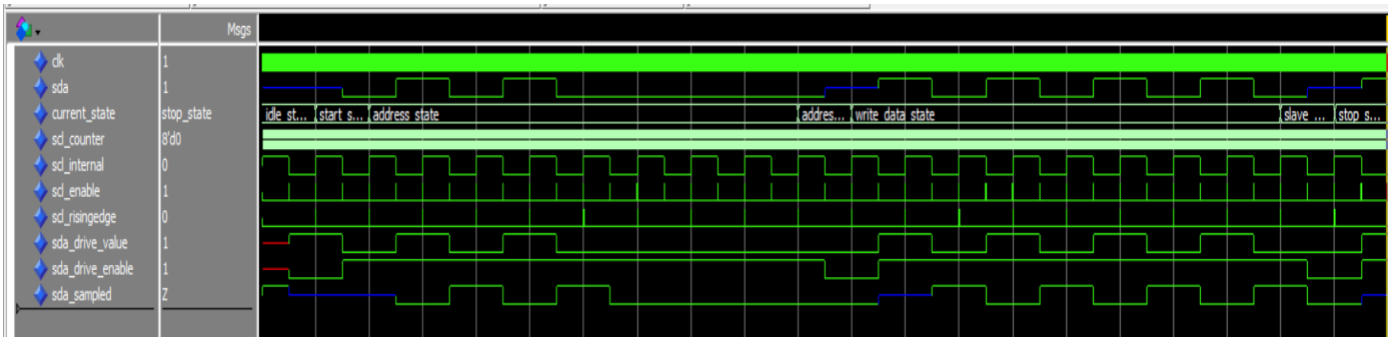
Testbenches 3 and 4 are functionally similar, with Testbench 3 serving as the initial evaluation of the continue control signal. This test focused on validating the handshake mechanism between the user logic (in this case the testbench) and the FSM. Upon entering the wait_state, the ready signal was asserted (though its visibility in the waveform is limited). The testbench actively polled ready, and upon detection, asserted the continue signal, triggering a transition from wait_state back to write_state.

At this stage of development, a new internal signal was introduced: start_i2c. Theis was used in conjunction with enable, to facilitate synchronized control of FSM transitions.

Testbench 4 exposed signal integrity issues with the busy and done flags: busy remained asserted indefinitely and done was never triggered. This behavior resulted from incorrect acknowledgment handling. In contrast, Testbench 3 did not include an ACK for the final byte, which led to

correct busy and done behavior. However, it also highlighted that the done signal had unintentionally evolved into a two-cycle pulse.

These tests were instrumental in identifying handshake timing flaws and improving control signal integration within the FSM architecture.

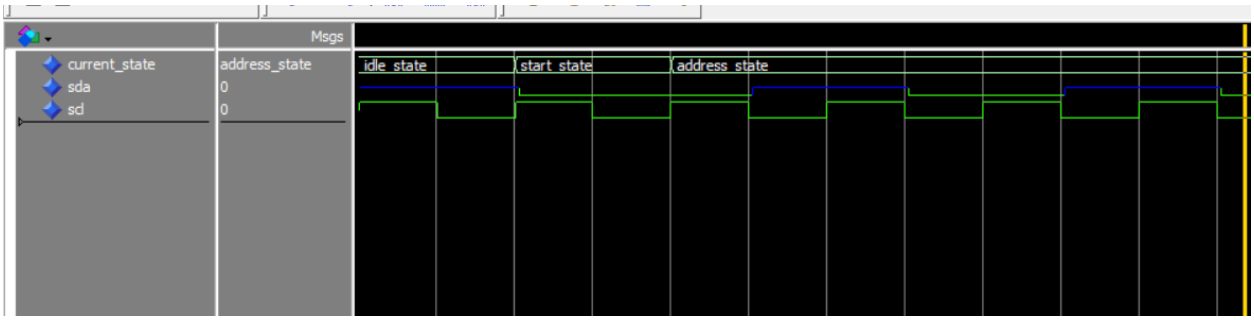


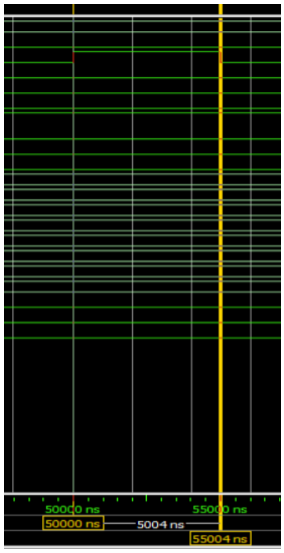
SCL and SDA Waveforms:

The waveform analysis highlights the behavior of control signals responsible for SDA line management, SCL generation, and clock edge detection. The simulation uncovered several critical design flaws. Most notably, the SDA line was incorrectly driven with direct logic levels, rather than being properly released to a high-impedance (tri-state) during idle and slave response phases, constituting a clear violation of the I²C open-drain protocol.

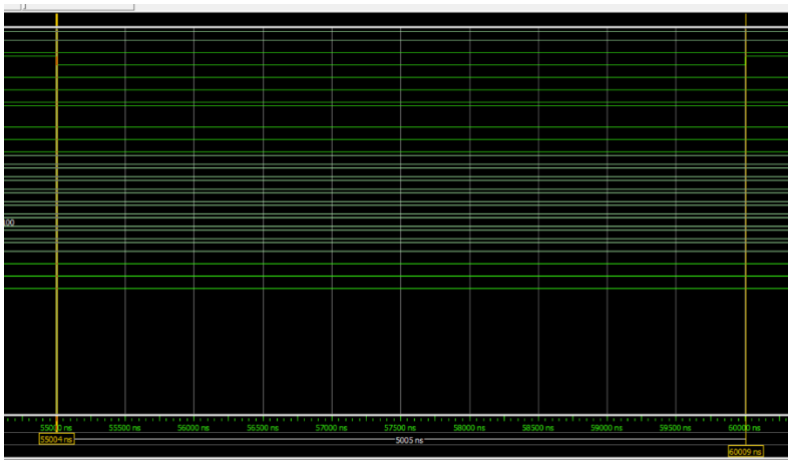
Furthermore, the behavior of the sda_sampled signal exhibited timing ambiguity, compromising the reliability of data capture and potentially leading to metastability or incorrect state transitions within the FSM.

Despite these deficiencies, the SCL control logic demonstrated correct behavior, with clean, valid edge transitions and a stable duty cycle. While this provides a minimal degree of functional assurance regarding clock generation, substantial architectural revisions were required to achieve protocol-compliant handling of SDA line transitions and sampling.





tHIGH



tLOW

Timing Waveforms:

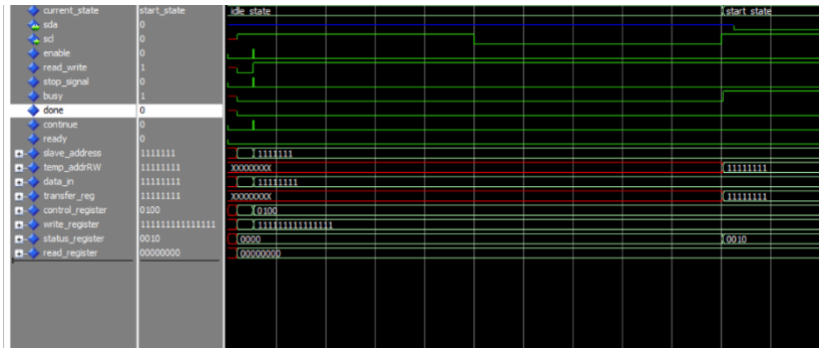
The waveform results confirm that both the HIGH and LOW phases of the SCL line measure approximately 5004 ns, corresponding to a clock frequency of 99.9kHz, which complies with Standard-mode I²C specifications (≤ 100 kHz)[2].

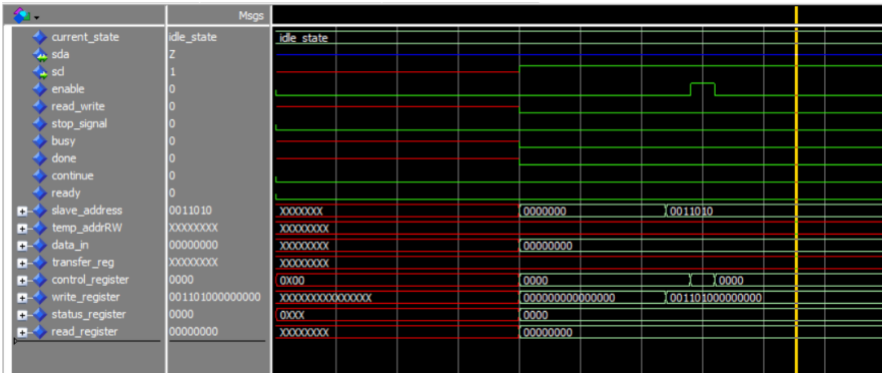
The first waveform illustrates the temporal relationship between SDA and SCL, focusing on setup and hold timing. The measured data setup time (tSU;DAT) is approximately 4984 ns, which is well above the minimum requirement of 250 ns for Standard-mode I²C [2]. While compliant, this value reflects a conservatively slow data setup and has been identified as a potential area for future timing optimization. However, due to project time constraints, performance tuning in this regard was not pursued during this iteration. The data hold time (tHD;DAT) was measured at 20 ns, satisfying the I²C specification which requires a minimum of 0 ns [2]. Although minimal, the value is considered valid and does not compromise protocol integrity.

2.4 Top-Level System Testbenches

The complete system was verified using UVVM-based top-level testbenches simulating read/write cycles.

Check 1





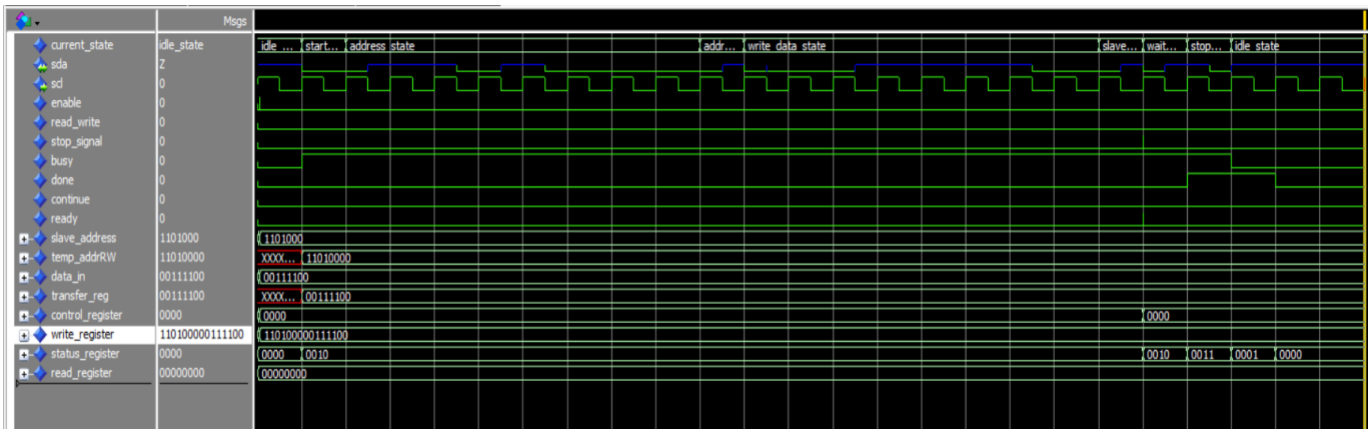
Check 2

Testbench 1: Register Check

The first validation step conducted after assembling the top-level module was a register interface test. This consisted of two nearly identical sub tests, referred to as Check 1 and Check 2, both of which employed UVVM's reset, write, and check register functions to verify correct initialization and behavior.

Check 1 was included primarily to provide enhanced signal visibility and to demonstrate that, even at this stage, SCL line behavior remained non-compliant. The simulation results confirmed that the register write operations were correctly executed and that all associated signals responded as expected. In particular, the status register contents changed appropriately in response to the issued writes, providing confidence in both the control path and register interface logic.

The use of UVVM significantly accelerated verification of the register interface by abstracting low-level signal manipulations. However, it is critical to understand the timing of checks, as improper placement relative to internal FSM transitions can lead to misleading results or missed failures.



Testbench 2: Single Byte Write Transaction with Stop

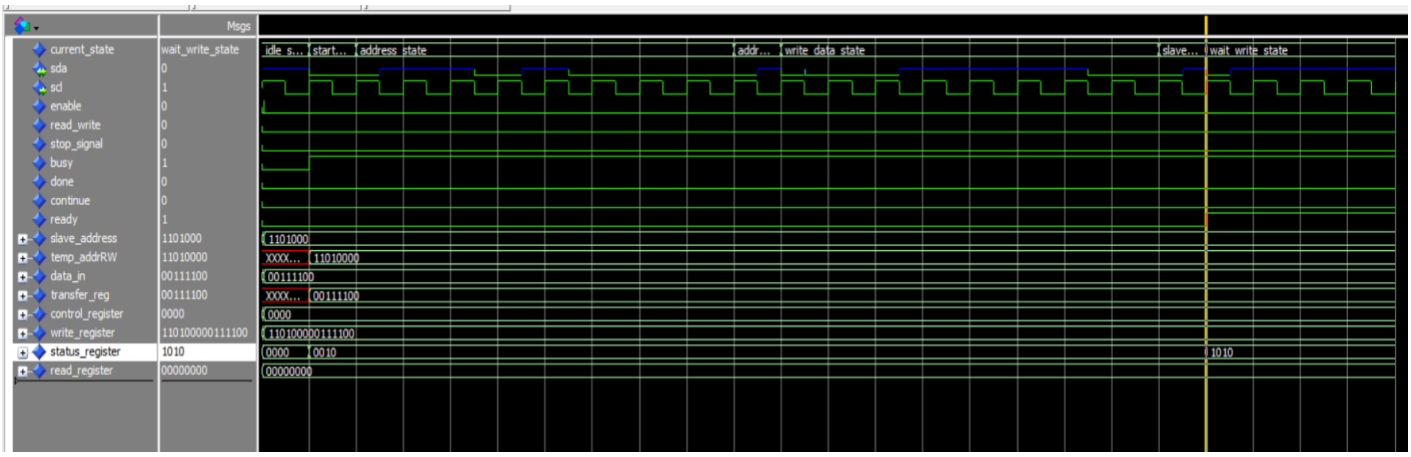
The waveform illustrates a basic test involving the transmission of a single data byte, conducted to validate register behavior mid-transfer. In this test, a poll function was employed to continuously monitor the status register. Upon entering the wait state (0010), the high bit of the control register was asserted, signaling a STOP condition and triggering the FSM to terminate the transaction and transition into the STOP state.

While this test confirmed correct status signaling and FSM responsiveness, it is important to note that consistent with prior findings , the SCL and SDA behavior during the wait and STOP sequences remained non-compliant with I²C protocol specifications [1].

It must be emphasized that these violations were not detected at first and were only observable through later waveform inspection. As this test was conducted in simulation, these issues were temporarily tolerated to allow continued development, but they underscore the importance of integrating protocol-aware validation mechanisms in future iterations.

```
# UVVM: ID_BFM                    200065.0 ns  SBI BFM                  sbi_poll_until(A:x"2", x"0000000A", 0, 1000000 ns)=> OK, read data = x"A" after 9970
# UVVM:                                                                occurrences and 199400 ns. 'polling status register'
# UVVM: ID_BFM                    200085.0 ns  TB seq.                 sbi_write(A:x"0", x"00000002") completed. 'write control register, continnue = 1'
# UVVM: ID_BFM                    200105.0 ns  TB seq.                 sbi_check(A:x"0", x"00000002")=> OK, read data = x"2". 'checking control register'
# UVVM: ID_BFM                    200125.0 ns  TB seq.                 sbi_check(A:x"1", x"0000683C")=> OK, read data = x"683C". 'checking write register'
# UVVM: ID_BFM                    200145.0 ns  TB seq.                 sbi_check(A:x"2", x"00000002")=> OK, read data = x"2". 'checking status register'
# UVVM: ID_BFM                    200165.0 ns  TB seq.                 sbi_check(A:x"3", x"00000000")=> OK, read data = x"00000000". 'checking reading register'
# UVVM:
# UVVM:      ================================================================================================================================================================
# UVVM:      *** FINAL SUMMARY OF ALL ALERTS ***
# UVVM:      ================================================================================================================================================================
# UVVM:                                  REGARDED   EXPECTED   IGNORED     Comment?
# UVVM:                NOTE         :      0          0          0        ok
# UVVM:                TB_NOTE      :      0          0          0        ok
# UVVM:                WARNING      :      0          0          0        ok
# UVVM:                TB_WARNING   :      0          0          0        ok
# UVVM:                MANUAL_CHECK :      0          0          0        ok
# UVVM:                ERROR        :      2          0          0        *** ERROR ***
# UVVM:                TB_ERROR     :      0          0          0        ok
# UVVM:                FAILURE      :      0          0          0        ok
# UVVM:                TB_FAILURE   :      0          0          0        ok
# UVVM:      ================================================================================================================================================================
# UVVM:      >> Simulation FAILED, with unexpected serious alert(s)
# UVVM:      ================================================================================================================================================================
# UVVM:
# UVVM:
# UVVM:
# UVVM:
# UVVM: ID_LOG_HDR                202165.0 ns  TB seq.                 SIMULATION COMPLETED
# UVVM: ----------------------------------------------------------------------------------------------------------------------------------------------------------------
```

The simulation summary further confirmed correct behavior in the writing to and clearing of internal registers. Two test errors, as illustrated in the preceding figure, were intentionally introduced to facilitate traceability and functional validation. By deliberately supplying incorrect expected values, it was possible to confirm that the system responded with appropriate error detection and handling.

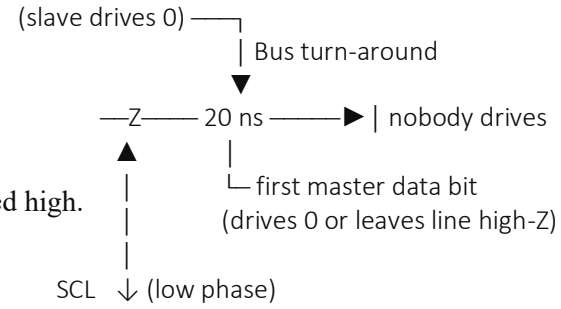


Testbench 3: Single Byte Write Transaction No Handshake

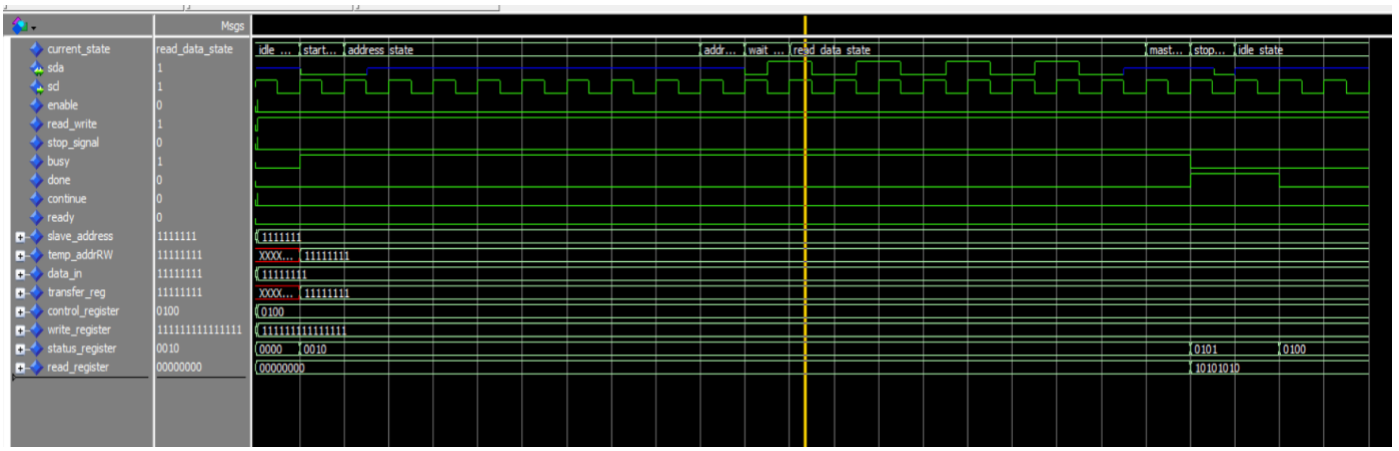
This test was conducted to verify the behavior of the ready handshake signal when routed to the register interface. As expected, the status register displayed the value (1010), indicating that the busy bit (bit 1) and the ready bit (bit 3) were both asserted. As previously described, the FSM remains indefinitely in this state until an explicit STOP or CONTINUE command is issued.

An important observation was made during the write state: immediately following the first falling edge of SCL after an ACK, a brief glitch was observed on the SDA line lasting exactly 20 ns, which corresponds to one system clock cycle at 50 MHz. This phenomenon occurs at the turnaround point, the only moment during a write cycle when the master releases control of SDA, specifically to allow the target (slave) to drive the ACK bit. As per I²C protocol[1]:

- Once SCL falls, the target releases the bus.
- The master then re-evaluates the next data bit to transmit.
- If the next bit is 0, SDA is driven low after this 20 ns window.
- If the next bit is 1, SDA is left released and will be passively pulled high.



This high-impedance (Z-state) period between ACK and bit 7 of the next byte can manifest as a short glitch in simulation. The behavior is protocol-compliant, as the I²C standard specifies a minimum data hold time (tHD;DAT) of 0 ns[2]. On real hardware, the pull-up resistor cannot raise the line significantly in 20 ns, so the subsequent low (for a '0') remains well below the receiver's VIL threshold and poses no risk of bit misinterpretation.



Testbench 4: Read Transaction

Within the testbench, a simplified but functional slave model was implemented to validate read transactions. The model was constructed exclusively using for loops and conditional SDA assignments, avoiding behavioral abstractions in favor of explicit bit-level control.

During read operations, the slave model sequentially shifted a fixed data pattern (10101010) onto the SDA line, bit by bit, synchronized with the master's SCL clocking. This was implemented via a looped bit-shifting structure that mimicked the timing behavior of an I²C slave responding to a read command.

Simulation results confirmed that the master correctly sampled the SDA line in accordance with I²C timing, and that the end-to-end signal flow from register interface through to the top-level module and testbench slave was functionally correct. The integrity of the "wiring" between the register access logic and the I²C communication module was thus validated for this scenario.

```
# UVVM:
# UVVM: ID_LOG_HDR                    420.0 ns  TB seq.                    checking regiters after reset
# UVVM: ------------------------------------------------------------------------------------------------------------------------
# UVVM: ID_BFM                        445.0 ns  TB seq.                    sbi_check(A:x"0", x"00000000")=> OK, read data = x"00000000". 'checking control register'
# UVVM: ID_BFM                        465.0 ns  TB seq.                    sbi_check(A:x"1", x"00000000")=> OK, read data = x"00000000". 'checking write register'
# UVVM: ID_BFM                        485.0 ns  TB seq.                    sbi_check(A:x"2", x"00000000")=> OK, read data = x"00000000". 'checking status register'
# UVVM: ID_BFM                        505.0 ns  TB seq.                    sbi_check(A:x"3", x"00000000")=> OK, read data = x"00000000". 'checking reading register'
# UVVM: ID_BFM                        525.0 ns  TB seq.                    sbi_write(A:x"0", x"FFFFFFFF") completed. 'writing to control register'
# UVVM: ID_BFM                        545.0 ns  TB seq.                    sbi_write(A:x"1", x"FFFFFFFF") completed. 'writing to write register'
# UVVM: ID_BFM                        565.0 ns  TB seq.                    sbi_write(A:x"2", x"FFFFFFFF") completed. 'writing to status register'
# UVVM: ID_BFM                        585.0 ns  TB seq.                    sbi_write(A:x"3", x"FFFFFFFF") completed. 'writing to reading register'
# UVVM: ID_BFM                        605.0 ns  TB seq.                    sbi_check(A:x"0", x"00000004")=> OK, read data = x"4". 'checking control register'
# UVVM: ID_BFM                        625.0 ns  TB seq.                    sbi_check(A:x"1", x"00007FFF")=> OK, read data = x"7FFF". 'checking write register'
# UVVM: ID_BFM                        645.0 ns  TB seq.                    sbi_check(A:x"2", x"00000000")=> OK, read data = x"00000000". 'checking status register'
# UVVM: ID_BFM                        665.0 ns  TB seq.                    sbi_check(A:x"3", x"00000000")=> OK, read data = x"00000000". 'checking reading register'
# UVVM:
# UVVM:      ================================================================================================================
# UVVM:      *** FINAL SUMMARY OF ALL ALERTS ***
# UVVM:      ================================================================================================================
# UVVM:                                REGARDED   EXPECTED  IGNORED     Comment?
# UVVM:                NOTE        :       0          0         0        ok
# UVVM:                TB_NOTE     :       0          0         0        ok
# UVVM:                WARNING     :       0          0         0        ok
# UVVM:                TB_WARNING  :       0          0         0        ok
# UVVM:                MANUAL_CHECK:       0          0         0        ok
# UVVM:                ERROR       :       0          0         0        ok
# UVVM:                TB_ERROR    :       0          0         0        ok
# UVVM:                FAILURE     :       0          0         0        ok
# UVVM:                TB_FAILURE  :       0          0         0        ok
# UVVM:      ================================================================================================================
# UVVM:      >> Simulation SUCCESS: No mismatch between counted and expected serious alerts
# UVVM:      ================================================================================================================
# UVVM:
# UVVM:
```

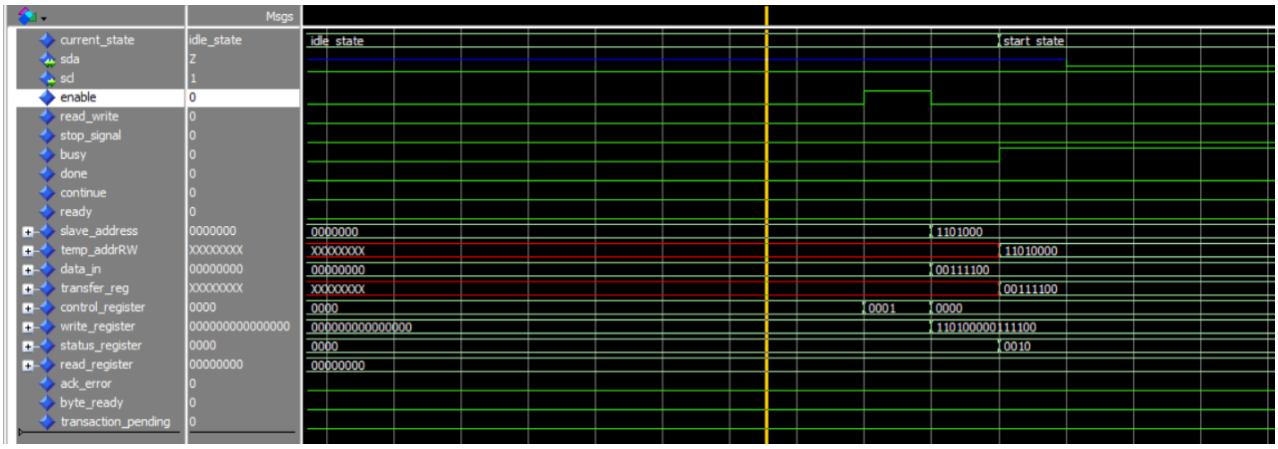
The preceding summary table presents the results of register-level verification. In this test case, register 1 represents the control register, and the value x"4" indicates that both the read/write bit and the enable bit were correctly set to 1. At the time of checking, the read data register remained empty, as no data had yet been received from the I²C slave.

2.5 Hardware Testing Outcomes and Design Revisions

Initial hardware testing yielded unsatisfactory results, which, while disappointing, were ultimately consistent with known design flaws identified during simulation. Due to time constraints, no screenshots were captured from the initial hardware phase.

Critically, as previously documented, the implementation of the IDLE, WAIT, and STOP states was non-compliant with I²C protocol requirements[1]. This directly impacted hardware behavior: the slave device failed to acknowledge (ACK) transmissions, which is fully explainable given that the SDA line was not being correctly released or driven during key protocol phases. Without a valid ACK, the master could neither complete write operations nor initiate read sequences.

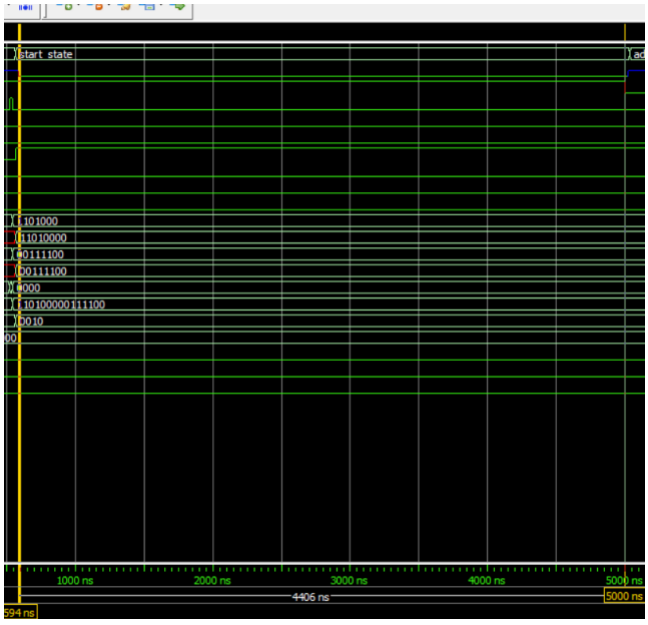
In response to these findings, development was redirected back to top-level simulation and source code correction, with a focus on restructuring the SDA control logic, refining FSM transitions, and improving protocol alignment. The remainder of this section provides a detailed account of the corrective measures taken and the functional improvements achieved through simulation.



Correction of Idle and Start Conditions:

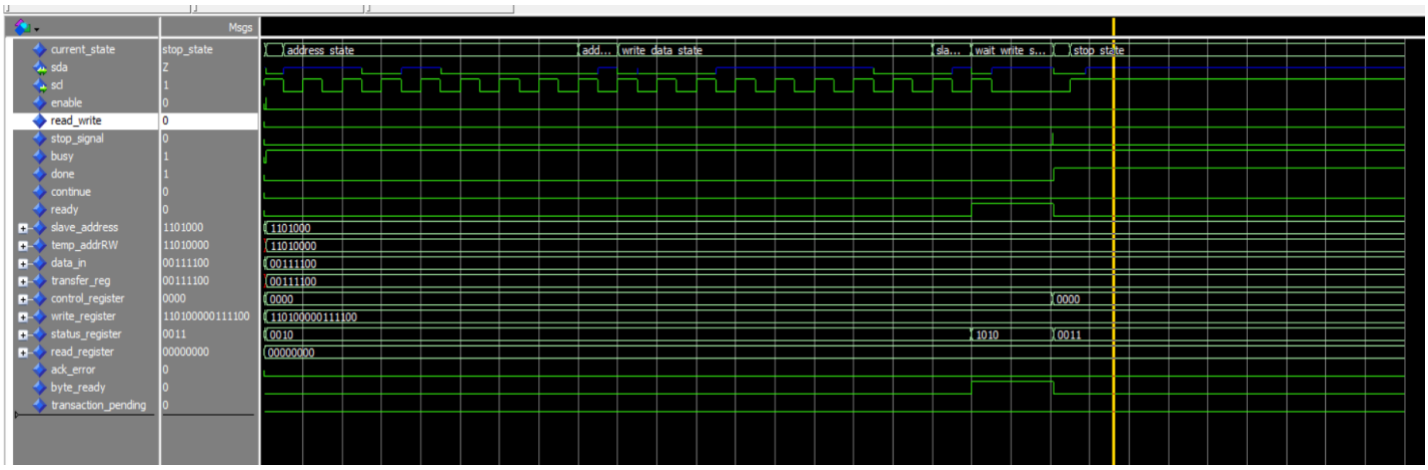
Following the identification and mapping of failures. The first corrective action was to ensure a fully I²C-compliant IDLE and START condition. To achieve this, a new process referred to as the SCL toggle controller was implemented. Its purpose was to manage the conditions under which SCL is permitted to toggle (i.e., during data transfer) versus when it must remain high (e.g., during IDLE and STOP).

With this control mechanism in place, further adjustments to the SDA driving process were made to enforce open-drain behavior and synchronize line control with protocol timing. To meet these constraints, the finite state machine (FSM) was refactored and split, a decision taken reluctantly due to design complexity but ultimately necessary for compliance.



The IDLE state was redefined as level-triggered, holding SCL and SDA high. The START condition was re-implemented as edge-triggered, transitioning to the next state on the falling edge.

Following these changes, no further structural edits were made, and the design now produces a standards compliant IDLE and START sequence. As part of validation, the START hold time (tHD;START) was measured to confirm adherence to specification limits[2].



Implementation of STOP and WAIT States with Protocol-Conscious Control:

As part of the effort to achieve protocol compliance, the STOP and WAIT states were added to the control logic managed by the SCL toggle controller. The improper SDA handling in the ACK and WAIT states could not be resolved through minor adjustments. The most effective and maintainable solution was the introduction of a new FSM state, named prep_stop_state, whose purpose is to correctly prepare the SDA line for the generation of a valid STOP condition (i.e., SDA rising while SCL is high). This design change corrected prior violations where SDA transitions occurred outside spec.

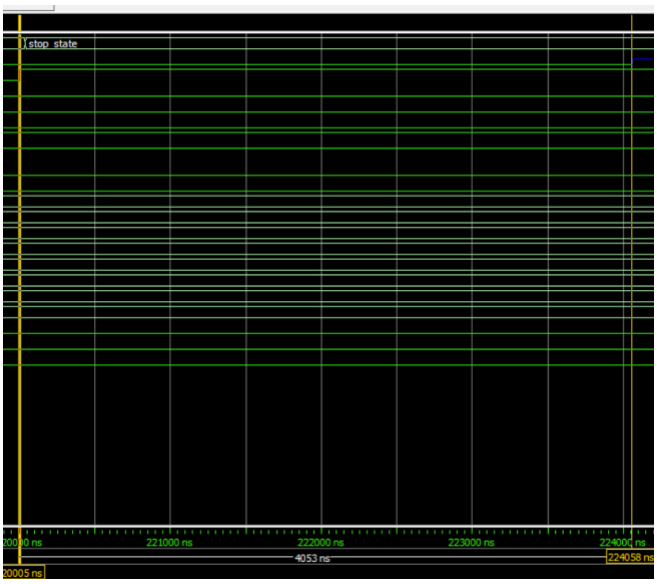
The done signal, which previously lacked synchronization with protocol state transitions, was also corrected as part of this revision. In parallel, the WAIT state was restructured to align with protocol intent, through an unconventional implementation. Specifically, I²C defines two compliant ways to indicate idle during WAIT conditions:

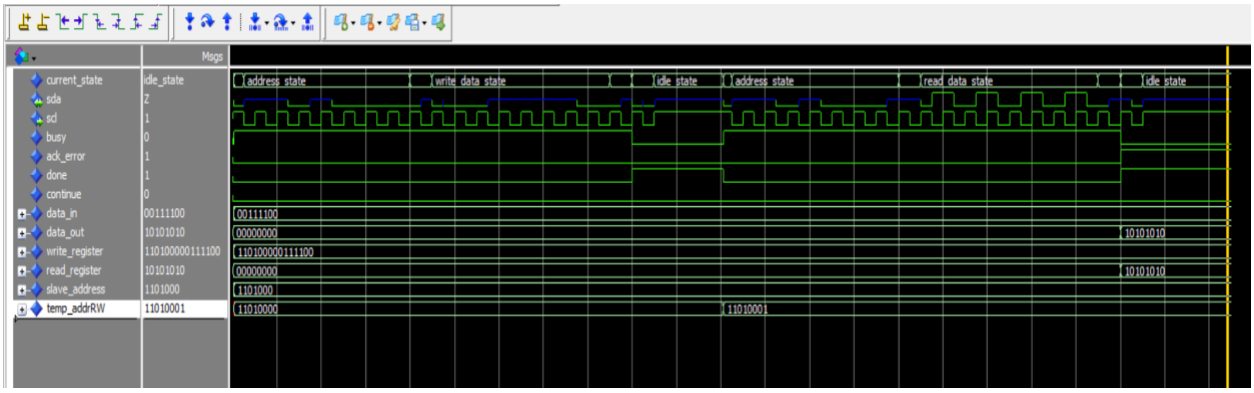
I. Release both SDA and SCL to allow them to float high (standard idle bus)[1],
II. Drive SCL low and leave SDA undefined (as SDA is not sampled while SCL is low)[1].

While the first option is theoretically cleaner, it introduced timing hazards in this design because the FSM is edge-triggered, and fully releasing both lines created ambiguous transitions and instability. To avoid further dismantling and rearchitecting of the FSM, the second approach was chosen driving SCL low during the WAIT state, which is still valid per I²C spec and operationally stable[2].

It is important to note that under this scheme, the STOP state does not transition into the IDLE state. This is due to the SCL gating logic, which constrains the transitions.

The stop setup time (tSU; STOP) was also measured to ensure complete protocol compliance 4053ns[2].





Complete Single Byte Write transactions followed by a Read transaction:

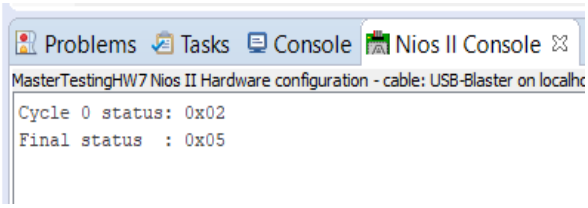
Full I2C transaction was successfully simulated, comprising the following sequence:
START → slave address (write) → register address → STOP → IDLE → START → slave address (read) → data byte → STOP → IDLE.
This test validated the integration of both write and read paths, as well as proper state transitions and bus behavior across the full transaction lifecycle.

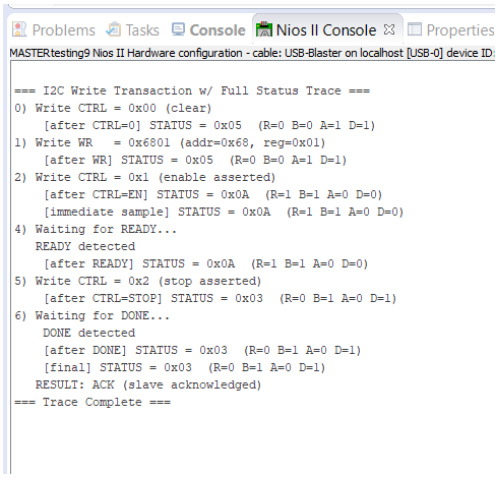
Notably, the transition from STOP to IDLE, previously non-compliant was also corrected in this iteration and now conforms to the I2C specification[1]. Although the prep_stop_state and stop_state are not directly visible in the waveform due to scale constraints, they were indeed executed. The entire transaction spanned approximately 400 µs, making finer state level transitions difficult to observe visually in the waveform window.

It is important to emphasize that simulation has limitations while timing, state sequencing, and signal behavior were confirmed to be protocol-compliant within the simulated environment, that does not guarantee correct behavior on hardware. Consequently, based on the successful simulation outcome, development proceeded to a renewed phase of hardware testing, aimed at validating the system under real electrical conditions.

3. Hardware Testing

Hardware validation was executed on a DE2-115 board interfaced to a DS3231-compatible RTC [7]. Control transactions were generated by a lightweight C driver and orchestrated via a Nios II application [4]. Two integration anomalies were observed.





First Status-Register ambiguity:

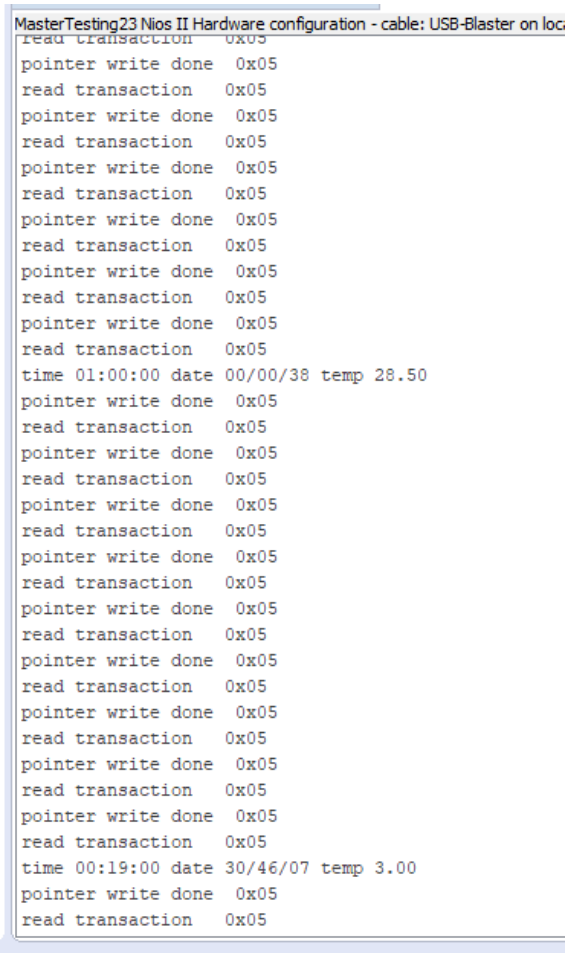
Intermittent NACK responses were observed immediately after verifying that the control register was correctly configured. In several instances, a NACK was issued before a subsequent ACK without any change in configuration. This inconsistent behavior points to a race condition between status flag updates (READY, DONE) and the FSM's transition into the STOP state. Specifically, the status register may be sampled before the internal write-back or transaction commit has finalized, leading to premature signaling of a failed transaction despite successful completion.

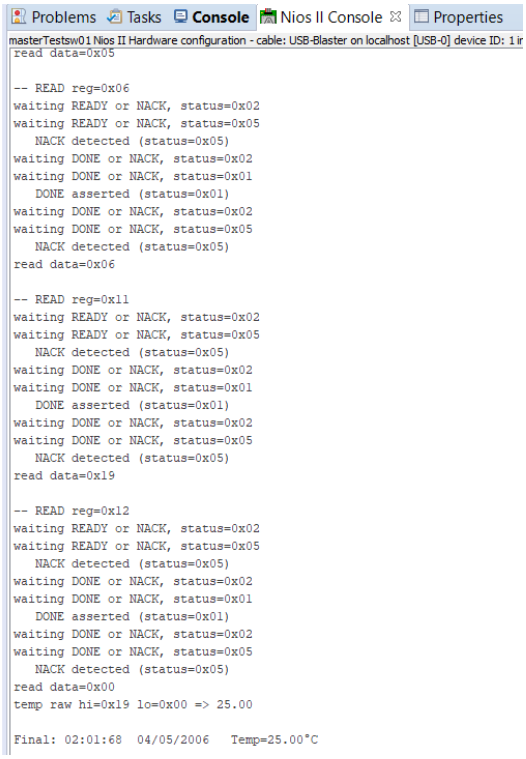
In later test iterations temporarily resolved the ACK failures, restoring functional read access to time and temperature registers. This confirmed that the root cause was not the data content or bus-level interference, but rather improper handshaking between internal control logic and bus signaling.

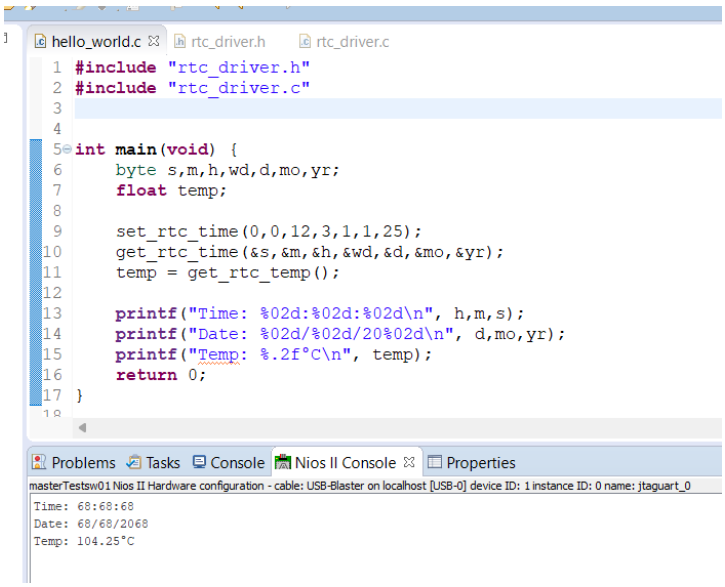
Second, Write-Back Ambiguity:

Even after successful acknowledgment sequences, read operations consistently returned values inconsistent with those written typically factory defaults or invalid data. This indicates that the pointer register was not properly latched by the RTC, likely due to a premature termination of the write cycle. Specifically, the master may have released the SDA line one SCL edge too early, violating the I²C protocol requirement for proper STOP condition timing and leaving the slave in an undefined internal state.

The persistence of incorrect read data despite corrected ACK behavior strongly supports a hypothesis of pointer sequencing failure rather than bus noise.







Due to project schedule constraints, a full architectural refactor was not feasible. Development was therefore halted at this milestone, with a defined remediation path in place: first, refine the SCL gating logic to enforce the additional half-clock period required by the RTC's tSU;WR (write setup time) specification and rebuild the SDA driver to ensure robust transmission [2].

4. Discussion

The disparities between simulation success and hardware failure are symptoms of latent methodological gaps.

Simulation abstracts away the analogue bus dynamics, deterministic time ordering, and board-level understanding. Under these idealized conditions, the controller's STOP sequence appears atomic, yet on silicon, the finite propagation through the combinational release path allows SDA to outrun SCL. This mismatch exposes an architectural assumption: that all observable timing is synchronous to the master clock. In practice, I²C timing is negotiated by the slowest edge on the bus, not by a rising-edge reference in RTL.

DONE and READY were handled poorly. The failures, therefore, is not a corner-case; it is a natural outcome of unqualified cross-domain communication.

The root issue is procedural, not purely technical. A simulation only sign-off criterion enforces functional convergence but offers no guardrail against temporal divergence. IP demands a mixed verification strategy, static timing analysis, assertion-based verification, and on-chip observability (ILA/LA probes). Without these, every "pass" in simulation risks a deferred failure on hardware.

Refinement therefore hinges on both incremental RTL tweaks and the maturation of the verification pipeline. This includes introducing the use of oscilloscopes and logic analyzers for real-time signal inspection, as well as implementing more robust gating strategies to prevent unintended transitions. Enhancing the visibility into hardware-level behavior complements improvements in RTL fidelity, ensuring that each functional block operates predictably under real-world conditions

5. Conclusion

This project established a functionally correct I²C master module, architected with protocol compliance and modular design principles. While simulation confirmed end-to-end transactional validity, hardware-level observations exposed timing-sensitive vulnerabilities requiring further refinement. These insights were not merely incidental but instrumental in identifying the temporal boundaries where simulation fidelity diverges from physical behavior.

More importantly, the project validated core architectural decisions, enabling deterministic state transitions, structured SDA/SCL control, and extensibility for future enhancements. Though currently unsuitable for direct integration into systems, the implementation forms a verified design well-positioned for evolution into hardware resilient I²C Master. The work reinforces a broader engineering axiom: correctness under simulation is necessary, but insufficient and HDL debugging is unforgiving. On-chip system readiness demands timing closure, proper signal gating, and stable external interfacing all of which will define the scope of subsequent development phases.

6. References

[1] “15. Intel FPGA Avalon® I2C (Host) Core,” Intel. Accessed: May 08, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683130/22-1/fpga-i2c-host-core.html

[2] “UM10204 I2C-Bus Specification and User Manual,” DigiKey. Accessed: May 08, 2025. [Online]. Available: https://www.digikey.com/en/pdf/n/nxp-semiconductors/um10204-i2c-bus-specification-and-user-manual

[3] “Platform Designer - Intel’s System Integration Tool,” Intel. Accessed: May 08, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/qts-platform-designer.html

[4] “5.4. Nios® II,” Intel. Accessed: May 08, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683634/21-2/nios-ii.html

[5] “Questa Verification & Simulation | Siemens Software.” Accessed: May 08, 2025. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/questa/simulation/

[6] UVVM, *UVVM/UVVM*. (Apr. 21, 2025). VHDL. Accessed: May 08, 2025. [Online]. Available: https://github.com/UVVM/UVVM

[7] Dejan, “Arduino and DS3231 Real Time Clock Tutorial,” How To Mechatronics. Accessed: May 08, 2025. [Online]. Available: https://howtomechatronics.com/tutorials/arduino/arduino-ds3231-real-time-clock-tutorial/

.