Threads & Interprocess Communication in SystemVerilog

Introduction:

With the completion of the earlier phases:

- **Day 1** explained *why modern verification needs structure*
- **Day 2** explained *how SystemVerilog models data and execution*
- **Day 3** explained *how behavior is abstracted and reused over time*
- **Day 4** explained *safe, race-free testbench–DUT interaction*
- **Day 5** introduced *object-oriented transaction modeling*
- **Day 6** completed *OOP reuse using inheritance and polymorphism*

Day 7 marks the final and most critical behavioral milestone.

At this stage, verification moves from *single-threaded reasoning* to **concurrent execution and synchronization**. Modern verification environments do not execute sequentially. They execute **multiple independent processes in parallel**, all interacting with shared resources, data, and time.

This phase introduces **threads and interprocess communication (IPC)** - the mechanisms that make **real-world verification environments possible**.

Why Threads & IPC Are Critical in Verification

In real verification environments:

- A **driver** continuously sends stimulus
- A **monitor** continuously observes DUT behavior
- A **scoreboard** compares expected vs actual results
- A **checker** watches protocol rules
- A **coverage collector** tracks functional progress

All of these run **simultaneously**.
If concurrency is not controlled:

- Data is corrupted
- Race conditions appear randomly
- Tests pass on one simulator and fail on another
- Debugging becomes impossible

Concurrency bugs are verification bugs, not design bugs.

What Is a Thread in SystemVerilog?

A **thread** is an independently executing process.

In SystemVerilog:

- Every initial block is a thread
- Every always block is a thread
- Every fork block creates multiple threads
-

Threads execute **in parallel**, sharing:

- Simulation time
- Variables
- Objects
- Resources

1. fork...join — Creating Parallel Threads

1.1 Why fork...join Exists

Without fork...join, code executes **sequentially**:

```
task A();
  #10;
endtask

task B();
  #10;
endtask

initial begin
  A();
  B();
end
```

Execution time = **20 time units**

But verification components must run **in parallel**, not one after another.
This is where fork...join is required.

1.2 Basic fork...join Syntax

```
fork
  A();
  B();
join
```

Now:

- A and B start at the **same simulation time**
- Total time = **10 time units**

1.3 Simple fork...join Example

```
module fork_basic;

  task task1();
    #5;
    $display("Task1 finished at %0t", $time);
  endtask

  task task2();
    #10;
    $display("Task2 finished at %0t", $time);
  endtask

  initial begin
    fork
      task1();
      task2();
    join
    $display("Both tasks completed at %0t", $time);
  end

endmodule
```

Execution Explanation

- Both tasks start at time 0
- task1 finishes at time 5
- task2 finishes at time 10
- Parent resumes after both complete

Verification Meaning

This models:

- Driver and monitor running together
- Multiple protocol activities in parallel

1.4 fork Variants

fork...join

- Waits for **all threads**

fork...join_any

- Continues when **any one thread finishes**
-

fork...join_none

- Parent continues immediately
- Threads run in background
-

Example:

```
fork
  task1();
  task2();
join_none
$display("Parent continues immediately");
```

Used for:

- Background monitors
- Watchdog threads
- Timeout logic

2. Events - Synchronization Without Data

2.1 Why Events Are Needed

Consider two threads:

- One generates stimulus
- Another waits until stimulus is done

Using delays is unsafe:

- Timing changes break behavior
- Tests become fragile

Events provide **explicit synchronization**.

2.2 What Is an Event?

An event is a **notification mechanism**:

- One thread triggers the event
- Another thread waits for it

Events:

- Do NOT carry data
- Carry **meaning**

2.3 Basic Event Example

```
module event_example;

  event done;

  initial begin
    #10;
    $display("Stimulus completed at %0t", $time);
    -> done;
  end

  initial begin
    $display("Waiting for event");
    @done;
    $display("Event received at %0t", $time);
  end

endmodule
```

Execution Explanation

- Second thread blocks at @done
- First thread triggers event at time 10
- Waiting thread resumes

2.4 Verification Use of Events

Events are used for:

- Phase synchronization
- End-of-test signaling
- Handshake between components

Example:

- Driver signals “transaction sent”
- Scoreboard starts checking

3. Semaphores — Protecting Shared Resources

3.1 The Problem Semaphores Solve

When multiple threads access shared data:

expected_queue.push_back(pkt);

If two threads do this simultaneously:

- Data corruption occurs
- Debug becomes impossible

3.2 What Is a Semaphore?

A semaphore is a **token-based lock**.

Rules:

- Thread must acquire token before entering critical section
- Token must be released after use

3.3 Semaphore Example

```
module semaphore_example;

  semaphore sem = new(1);

  task access_resource(string name);
    sem.get(1);
    $display("%s entered at %0t", name, $time);
    #5;
    $display("%s exiting at %0t", name, $time);
    sem.put(1);
  endtask

  initial begin
    fork
      access_resource("Thread1");
      access_resource("Thread2");
    join
  end

endmodule
```

Execution Explanation

- Only one thread enters at a time
- Second thread waits for token
- Resource is accessed safely

3.4 Verification Use of Semaphores

Semaphores protect:

- Scoreboards
- Shared queues
- Coverage databases
- Global counters

Without semaphores:

- Results are non-deterministic

4. Mailboxes — Safe Data Transfer Between Threads

4.1 Why Mailboxes Are Needed

Events synchronize **execution**, but cannot pass data.
Mailboxes:

- Synchronize
- Transfer data safely

4.2 What Is a Mailbox?

A mailbox is a **thread-safe communication channel**.
Features:

- Carries data
- Blocks automatically
- Prevents races

4.3 Basic Mailbox Example

```
module mailbox_example;

  mailbox mb = new();

  initial begin
    int data = 42;
    $display("Sending data at %0t", $time);
    mb.put(data);
  end

  initial begin
    int received;
    mb.get(received);
    $display("Received %0d at %0t", received, $time);
  end

endmodule
```

4.4 Verification-Oriented Mailbox Example

```
mailbox #(packet) mb = new();

task driver();
  packet pkt = new();
  pkt.addr = 8'h10;
  mb.put(pkt);
endtask

task monitor();
  packet pkt;
  mb.get(pkt);
  pkt.display();
endtask
```

Verification Meaning

- Driver sends transaction
- Monitor receives transaction
- No race conditions
- No timing hacks

This is **transaction-level communication**.

5. Comparison of IPC Mechanisms

Mechanism	Purpose	Data	Blocking	Verification Use
fork...join	Parallel execution	No	No	Concurrent components
Event	Notification	No	Yes	Phase sync
Semaphore	Resource locking	No	Yes	Protect shared data
Mailbox	Data + sync	Yes	Yes	Transaction passing

Key Takeaways — Day 7

- Verification is inherently concurrent
- Threads execute in parallel
- Synchronization is mandatory
- fork...join enables concurrency
- Events synchronize execution
- Semaphores protect shared resources
- Mailboxes enable safe data transfer
- IPC bugs are verification bugs

Completion of Functional Verification Journey

With Day 7, the **functional verification foundation is complete**.
You now understand:

- Structure
- Execution semantics
- Behavior reuse
- Safe synchronization
- Object-oriented modeling
- Parallel execution

This is the **exact conceptual base required for UVM**.

Using all the concepts learned across the previous phases, a **full functional verification of a 2-bit adder** was implemented. This example shows **how a real verification flow is built**, from stimulus generation to checking, and has been maintained in GitHub as a complete reference.

Github link for 2 bit Adder Verification

Github link for UART Transmitter Protocol Verification

--The End --