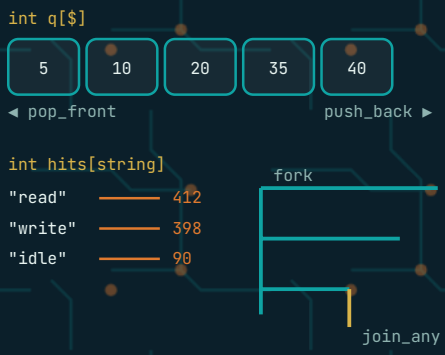
STUDY PAGE · AREA 03 OF 14

SystemVerilog Data Types & Procedural Code



03

The types and threads every testbench is built from: queues, associative arrays, structs and enums, automatic tasks, fork/join and mailboxes. Ends with a FIFO checked by a queue model.

14 h of study · **4** modules · **7** diagrams · **2** interactive · **Free** tools · Verilator 5 · **20**-question test · 70% to pass · Lab: queue-checked FIFO

1 · Types → 2 · Containers → 3 · Routines → 4 · Threads → Lab · FIFO → Area test · 70%

2/4-state · enum · struct | queue · assoc | task · function · ref | fork · mailbox | find the overflow bug | 20 questions

BEFORE YOU START · A

Who this area is for

The first SystemVerilog area. You know Verilog RTL; now you learn the language features that testbenches are written in.



EE / ECE students

Final-year and master's students preparing for VLSI internships and campus placements in design or verification.

Software engineers

Developers moving into hardware who want the foundations explained from first principles.

QA & test engineers

Test automation people heading into design verification — the same stimulus-and-check mindset applied to silicon.

FPGA & embedded engineers

Engineers who write some HDL already and want the rules and habits that verification teams expect.

RTL designers

Designers who want to move to verification and think like the person checking their code.

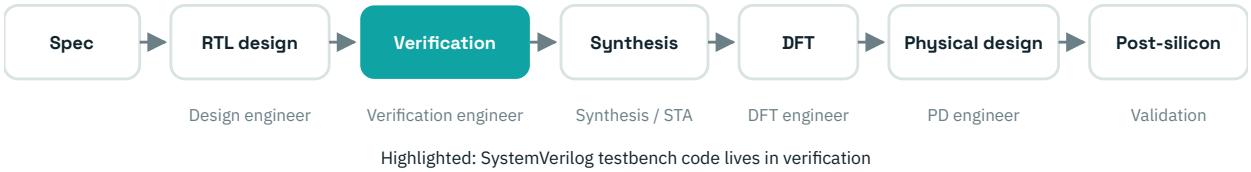
Career returners

Engineers refreshing fundamentals before interviews after time away from hands-on work.

Prerequisites

You need	Level	Where to get it
Verilog modules, always blocks, = vs <=	Comfortable	Area 02
A first testbench (driver, monitor, check)	Have written one	Area 02 · Module 4
Any programming language	Helpful: arrays, functions	—

Where this area sits in the chip flow — and the roles that use it



Highlighted: SystemVerilog testbench code lives in verification

These features are verification features. Synthesis ignores queues, mailboxes and fork — they exist to build testbenches.

Role	How you use this area on the job	Typical first title
Design verification (DV)	Write scoreboards with queues, coverage counters with associative arrays, parallel stimulus with fork	DV Engineer / Verification Engineer I
RTL design	Use enums and packed structs in RTL; read testbench code when debugging	ASIC Design Engineer / RTL Engineer
FPGA	Write SystemVerilog testbenches for FPGA blocks	FPGA Engineer
DFT	Build pattern lists and expected-response tables	DFT Engineer
Post-silicon validation	Write directed tests and models that mirror silicon tests	Validation Engineer

BEFORE YOU START · B

What you will be able to do

Every outcome is practised in the lab and checked by the area test.



- Choose between bit, logic, int and integer and avoid hiding X.
- Use typedef, enum, packed and unpacked structs.
- Pick the right container: fixed, dynamic, queue or associative.
- Use array methods instead of hand-written loops.
- Write automatic functions and tasks with ref and default arguments.
- Run parallel threads with fork/join variants and stop them safely.
- Connect threads with mailboxes and guard resources with semaphores.
- Check a FIFO with a queue model and find its overflow bug.

BEFORE YOU START · C

Tools & software

SystemVerilog testbench features need a simulator that supports them: Verilator 5 for free, or a commercial simulator on EDA Playground.



Tool	What it does	Used in this area for	Cost
Verilator (5.x)	Fast simulator that compiles Verilog/SystemVerilog to C++; also a strict linter	Every example and the lab (`make`)	Free
Icarus Verilog `iverilog` + `vvp`	Event-driven Verilog simulator: compiles, then runs	Not for this area: no associative arrays, limited SystemVerilog	Free
GTKWave or **Surfer**	Waveform viewers for `.vcd` / `.fst` dumps	Watching the FIFO's flags	Free
VS Code + Verilog extension	Editor with syntax highlighting and lint on save	Writing code	Free
EDA Playground	Browser-based simulators, including commercial ones, after a free sign-up	Commercial simulators with full SystemVerilog support	Free
make and **git**	Run builds with one command; track your changes	Lab Makefile; saving your work	Free

Commercial simulators you will meet at work

Simulator	Vendor	Notes
VCS	Synopsys	Widely used for SystemVerilog/UVM regressions
Xcelium	Cadence	Successor of Incisive / NC-Sim
Questa	Siemens EDA	Successor of ModelSim; starter editions ship with FPGA tools

i

Same language, same rules

Verilator 5 supports the features in this area (queues, associative arrays, classes, fork/join with `--timing`). Commercial simulators support the full IEEE 1800 standard.

BEFORE YOU START · D

Install step by step

Pick your computer. About 15 minutes. If anything fails, the browser option works with nothing installed.



Windows

Recommended: WSL 2 (Ubuntu inside Windows). You get the same commands as Linux and as most companies' compute farms.

1

Turn on WSL and install Ubuntu

Open **PowerShell as Administrator**, run the command, then restart.

```
wsl --install
```

After the restart an Ubuntu window opens: choose a user name and password.

2 **Install the tools**

In the Ubuntu window:

```
sudo apt update
sudo apt install -y iverilog verilator gtkwave make git
```

3 **Install VS Code and connect it to WSL**

Install VS Code from code.visualstudio.com, add the **WSL** extension, then type `code .` in Ubuntu to open the current folder.

4 **Add Verilog support**

Install the **Verilog-HDL/SystemVerilog/Bluespec SystemVerilog** extension (publisher mshr-h).

5 **Check it works**

```
iverilog -V | head -1
verilator --version
```

i **No WSL allowed on your laptop?**

Install the native Icarus Verilog for Windows build (installer from bleyer.org/icarus, which also bundles GTKWave) and tick "Add executable folder to the user PATH". Use EDA Playground for Verilator.

macOS

1 **Install Apple's command-line tools**

```
xcode-select --install
```

2 **Install Homebrew**

Follow the one-line command on brew.sh, then reopen Terminal.

3 **Install the simulators**

```
brew install icarus-verilog verilator make git
```

4 **Install a waveform viewer**

Download **Surfer** from surfer-project.org (native on Apple silicon), or try `brew install --cask gtkwave`.

5 **Editor**

Install VS Code and the **Verilog-HDL/SystemVerilog/Bluespec SystemVerilog** extension.

6 **Check it works**

```
iverilog -V | head -1
verilator --version
```

Linux

1 **Ubuntu / Debian**

```
sudo apt update
sudo apt install -y iverilog verilator gtkwave make git
```

2 **Fedora**

```
sudo dnf install -y iverilog verilator gtkwave make git
```

3 **Arch**

```
sudo pacman -S iverilog verilator gtkwave make git
```

4 **Check the versions**

```
iverilog -V | head -1      # 11 or newer
verilator --version        # 5.x recommended
```

Browser only

1. **Open EDA Playground**

   Go to edaplayground.com and create a free account (simulation needs a login).

2. **Paste the code**

   Design on the right, testbench on the left.

3. **Choose a simulator**

   Under *Tools & Simulators* pick a SystemVerilog-capable simulator (one of the commercial ones listed); tick *Open EPWave after run* for waveforms.

4. **Run**

   Press **Run**; the log shows your `$display` output.

> ! **This area needs Verilator 5**
>
> Queues, classes, mailboxes and `fork` timing need Verilator 5.0 or newer with `--timing`. Ubuntu 24.04 and current Homebrew ship 5.x; check with `verilator --version`. On an older system, use EDA Playground or build Verilator from source (verilator.org → Installation). Icarus Verilog does not support associative arrays or most class features.

BEFORE YOU START · E

Your first SystemVerilog run

A queue, an associative array and a fork in 20 lines.



1 Make a folder and a file called `first.sv`

```
module first;
  int q[$];                 // queue
  int hits[string];         // associative array
  int x;
  initial begin
    q.push_back(10); q.push_back(20); q.push_front(5);
    $display("queue size = %0d, front = %0d, back = %0d", q.size(), q[0]
    x = q.pop_front();
    $display("popped %0d, size now %0d", x, q.size());
    hits["read"]++; hits["read"]++; hits["write"]++;
    foreach (hits[k]) $display("hits[%s] = %0d", k, hits[k]);
    fork
      begin #10 $display("[%0t] slow thread done", $time); end
      begin #3  $display("[%0t] fast thread done", $time); end
    join_any
    $display("[%0t] join_any returned", $time);
    disable fork;
    $finish;
  end
endmodule
```

2 Compile it

```
verilator --binary --timing first.sv
```

3

Run it

```
./obj_dir/Vfirst
```

Expected output:

```
queue size = 3, front = 5, back = 20
popped 5, size now 2
hits[read] = 2
hits[write] = 1
[3] fast thread done
[3] join_any returned
- first.sv:18: Verilog $finish
```

4

What to notice

`join_any` returned at time 3, when the fast thread finished; `disable fork` killed the slow one, so its message never prints. Associative keys print in sorted order.

BEFORE YOU START · F

Study plan

14 hours at about 7 hours a week.

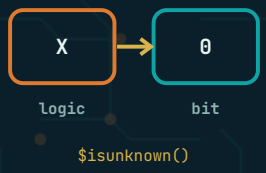


Session	Do this	Time
Day 1	Install Verilator 5 · first run · Module 1 (types, enum, struct)	3 h
Day 2	Module 2 (containers); queue playground; rewrite a loop with array methods	3 h
Day 3	Module 3 (functions, tasks, automatic, ref)	2 h
Day 4	Module 4 (fork variants, events, semaphores, mailboxes)	3 h
Day 5	Lab: complete the queue model, find the FIFO bug · area test	3 h

MODULE 1 OF 4

Types that replace bit-twiddling

Two-state vs four-state, enums, structs and strings — and the X trap.



- bit vs logic, int vs integer
- typedef and enum methods
- Packed and unpacked structs
- Strings and $sformatf

⏱ 3 hours · Area 03 · SystemVerilog

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

STEP 1

Why SystemVerilog

≈ **In plain words**	SystemVerilog is Verilog plus a lot of extra features for writing testbenches: better types, containers, classes, threads and random stimulus.
☀ **Think of it like**	The same car as before, with a full toolbox in the boot.
✎ **Worked example**	Everything you wrote in Area 02 still works. You now add tools that make testbenches shorter and safer.
Aa **The word for it**	**SystemVerilog (IEEE 1800)**. Files usually end in `.sv`.
! **Why it matters**	Every verification job description asks for it — UVM in later areas is built on it.

2

STEP 2

Two-state and four-state types

≈ **In plain words**

`logic` can hold 0, 1, X and Z. `bit` can hold only 0 and 1 — any X or Z becomes 0.

☀ **Think of it like**

A camera that records everything (logic) versus one that turns every blurry photo into a black frame (bit).

✎ **Worked example**

1. `logic [3:0] a = 4'b1x0z;` → a shows `1x0z`.
2. `bit [3:0] b = 4'b1x0z;` → b shows `1000`: the X and Z were silently turned into 0.

Aa **The word for it**

Four-state (`logic`, `integer`) and **two-state** (`bit`, `int`, `byte`).

! **Why it matters**

Use `logic` for anything connected to the DUT, so X bugs stay visible. Use `int` / `bit` for counters inside the testbench.

PAUSE & TRY **`bit [3:0] x = 4'b1z1x;` What does x hold?**

Answer

Z and X become 0 → `1010`.

3

STEP 3

Integer types at a glance

≈ **In plain words**

SystemVerilog has ready-made integer types of different widths, some signed.

☀ **Think of it like**

Boxes of standard sizes: small, medium, large.

✎ Worked example

Type	Bits	States	Signed
`bit`	1	2	no
`byte`	8	2	yes
`shortint`	16	2	yes
`int`	32	2	yes
`longint`	64	2	yes
`logic`	1	4	no
`integer`	32	4	yes

Aa **The word for it**

Signed types read the top bit as negative (Area 01's two's complement).

! **Why it matters**

A `byte` holding 200 reads as −56 — a classic surprise when you store data bytes in it. Use `bit [7:0]` for raw data.

4 STEP 4 typedef and enum: names instead of numbers

≈ **In plain words**

`typedef` gives a type a name. `enum` gives a list of values names, and can print them.

✳ **Think of it like**

Naming the dial positions on a washing machine instead of numbering them.

✎ Worked example

```
typedef enum logic [1:0] {IDLE, BUSY, DONE} state_t;
state_t s = BUSY;
$display("%s", s.name());   // prints BUSY
s = s.next();                  // DONE
```

Aa The word for it — **typedef**, **enum**, and enum methods (`name()`, `next()`, `first()`).

! Why it matters — Logs that say “state = BUSY” are far faster to debug than “state = 1”.

5 STEP 5 struct: group related fields

≈ In plain words — A `struct` bundles several fields under one name.

☀ Think of it like — A form with boxes for name, age and city — one form, many fields.

✎ Worked example

```
typedef struct packed {
  logic [3:0] kind;
  logic [3:0] len;
} hdr_t;
hdr_t h = 8'h52;   // packed: kind = 5, len = 2 — the fiel
```

Aa The word for it — **Packed struct** (a bit vector with named fields) versus **unpacked** (separate variables).

! Why it matters — Packet headers and register layouts are natural structs — your checks read `h.len` instead of `bus[3:0]`.

6 STEP 6 Strings and formatted messages

≈ In plain words — `string` holds text of any length; `$sformatf` builds text like `$display` but returns it.

☀ Think of it like — Filling in a message template: “Packet %0d had length %0d”.

Worked example

```
string msg = $sformatf("pkt %0d len %0d", id, len);
$display(msg);   // pkt 7 len 12
```

The word for it — **string**, **$sformatf**.

Why it matters — Clear, consistent error messages are half of debugging.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

Four-state type
Each bit can be 0, 1, X or Z: `logic`, `integer`, `reg`, `wire`.

Two-state type
Each bit is only 0 or 1: `bit`, `byte`, `int`, `longint`. Faster; X becomes 0.

typedef
Gives a new name to a type so it can be reused: `typedef logic [7:0] byte_t;`

enum
A type whose values are named constants; `.name()` prints the label.

Packed struct
Fields laid out as one contiguous bit vector — can be sliced and assigned to bits.

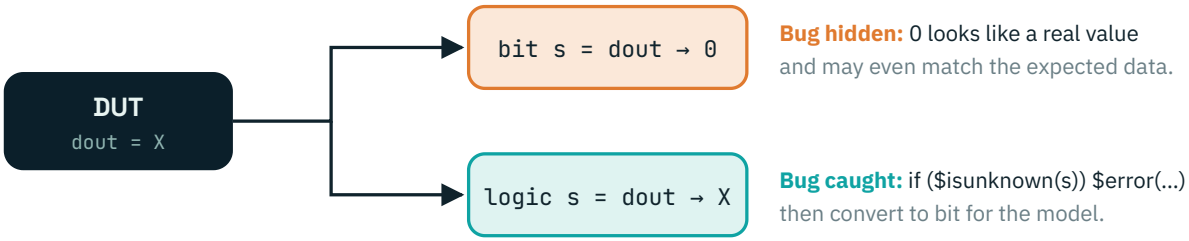
Unpacked struct
A record of fields that may include strings, queues or class handles; not a bit vector.

Cast
Converting a value to another type: `op_e'(2)`, `int'(x)`, `$cast`.

string
A dynamic text type with methods: `len`, `toupper`, `substr`, comparison with `==`.

Two-state and four-state types

Type	States	Width	Signed	Use for
`logic`	4	1 (or as declared)	no	DUT signals; anything that can be X
`bit`	2	1	no	Testbench data, flags
`byte`	2	8	yes	Byte streams, characters
`shortint` / `int` / `longint`	2	16 / 32 / 64	yes	Counters, loop variables, models
`integer`	4	32	yes	Legacy Verilog code
`time`	4	64	no	Simulation time



Never let a two-state variable swallow an X. Sample DUT outputs into `logic`, check for unknowns, then convert.

typedef and enum

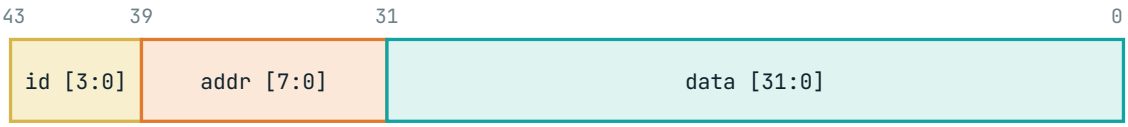
```
typedef enum logic [1:0] {READ = 2'b00, WRITE = 2'b01, IDLE = 2'b11} op_e;
op_e op = WRITE;
$display("%s", op.name());   // WRITE
op = op.next();               // IDLE
op = op_e'(0);               // explicit cast needed from an integer
```

Enum method	Returns
`.name()`	The label as a string
`.first()` / `.last()`	First / last declared value
`.next()` / `.prev()`	Neighbouring value (wraps)
`.num()`	How many values

struct and union

```
typedef struct packed {
  logic [3:0]  id;
  logic [7:0]  addr;
  logic [31:0] data;
} bus_req_t;

bus_req_t req = '{id: 4'h3, addr: 8'h10, data: 32'hCAFE_F00D};
```




```
logic [43:0] flat = req; // packed structs are just bit vectors
```

Packed means contiguous. The first field sits at the top bits. Use packed structs for things that travel on wires, unpacked ones for testbench records.

Strings

Code	Result
`string s = "axi";`	declare
`s = {s, "_lite"};`	"axi_lite"
`s.len()`	8
`s.toupper()`	"AXI_LITE"
`s.substr(0, 2)`	"axi"
`$sformatf("addr=%0h", a)`	formatted string — the workhorse for messages

Common mistakes

Assigning DUT outputs to `bit` or `int` · assigning an integer to an enum without a cast · expecting `%p` to print in decimal · putting a queue inside a packed struct (not allowed).

INTERVIEW CORNER

Q. Difference between `logic` and `bit`? **A.** `logic` is four-state, `bit` two-state; X/Z become 0 in a bit.

Q. Packed vs unpacked struct? **A.** Packed is a contiguous bit vector usable in expressions; unpacked is a record that can hold dynamic types.

Key takeaways

1. Use `logic` for anything that touches the DUT, two-state types for testbench maths.
2. Enums give names in logs and waves, and catch illegal values at compile time.
3. Packed structs are bit vectors with field names.

MODULE 2 OF 4

Arrays & containers

Fixed, dynamic, queue, associative — and the methods that make them fast to use.



- Choosing a container
- Queue operations
- Associative arrays as memories and counters
- Array methods with `with`

⏱ 3 hours | Area 03 · SystemVerilog

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

1 STEP 1 — Fixed arrays

≈ In plain words — A fixed array has a set number of elements decided when you write the code.

☀ Think of it like — An egg box with exactly 12 slots.

✎ Worked example

```
int scores[4] = '{10, 20, 30, 40};
foreach (scores[i]) $display(i, scores[i]);
```

Aa The word for it — **Fixed-size (static) array**.

! Why it matters — Fine for things with a known count, like 4 channels or 8 registers.

2 STEP 2 — Dynamic arrays

≈ **In plain words**

A dynamic array's size is chosen while the test runs, with `new[n]`.

☀ **Think of it like**

Setting out chairs once you know how many guests came.

✎ **Worked example**

```
byte payload[];
payload = new[len];   // len decided at run time
```

Aa **The word for it**

Dynamic array.

! **Why it matters**

Packet payloads have random lengths — dynamic arrays fit them exactly.

3 STEP 3 — Queues: first in, first out

≈ **In plain words**

A queue is a list you add to and take from at either end. Most often: add at the back, remove from the front.

☀ **Think of it like**

A line at a shop counter: the first person in is the first served.

✎ **Worked example**



`q = {}; push_back(5); push_back(8); push_back(3); pop_front() → q = {8, 3}, q.size() = 2`

push_back adds at the back; pop_front takes from the front.

Aa **The word for it**

Queue `int q[$];` with `push_back`, `pop_front`, `size()`.

! **Why it matters**

A queue is the perfect model of a FIFO — exactly what Lab 03 uses as its reference.

PAUSE & TRY **Start with `q = {}`. `q.push_back(4), push_back(7), pop_front(), push_back(1)`. What is q?** Answer

After pushes: {4, 7}. pop_front removes 4 → {7}. push_back(1) → **{7, 1}**.

4 STEP 4

Associative arrays: look up by key

≈ **In plain words**

An associative array stores values under keys you choose, and only uses memory for keys you actually write.

☀ **Think of it like**

A phone book: look up by name; empty pages take no space.

✎ **Worked example**

```
bit [31:0] mem [bit [31:0]];   // key = address
mem[32'h1000_0000] = 32'hCAFE;
if (mem.exists(addr)) data = mem[addr];
```

Aa **The word for it**

Associative array; `exists()`, `num()`, `delete()`.

! **Why it matters**

Modelling a 4 GB memory with only the addresses the test touched — a daily scoreboard trick.

5 STEP 5

Array methods do the loops for you

≈ **In plain words**

Arrays and queues have built-in methods to sum, search and sort.

☀ **Think of it like**

A spreadsheet's SUM and FILTER buttons instead of adding by hand.

✎ **Worked example**

1. `int a[] = '{3, 9, 4, 12};`
2. `a.sum()` → 28
3. `a.find(x) with (x > 5)` → {9, 12}
4. `a.sort()` → {3, 4, 9, 12}

Aa **The word for it**

Array reduction, **locator** and **ordering** methods, with a `with` clause.

! **Why it matters**

Coverage of “which packet lengths did we send” and checksum checks become one line.

6 STEP 6 Choosing the right container

≈ **In plain words**

Pick by how you use the data.

☀ **Think of it like**

Egg box, folding chairs, shop line, phone book.

✎ **Worked example**

Need	Use
Known, fixed count	fixed array
Size known at run time, then fixed	dynamic array
Add/remove at the ends (FIFO, expected-results list)	queue
Sparse, look up by key (memory, counters per ID)	associative array

Aa **The word for it**

Containers.

! **Why it matters**

The right container makes a scoreboard ten lines instead of fifty.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

Fixed-size array
Size set in the declaration: `int a[8]`.

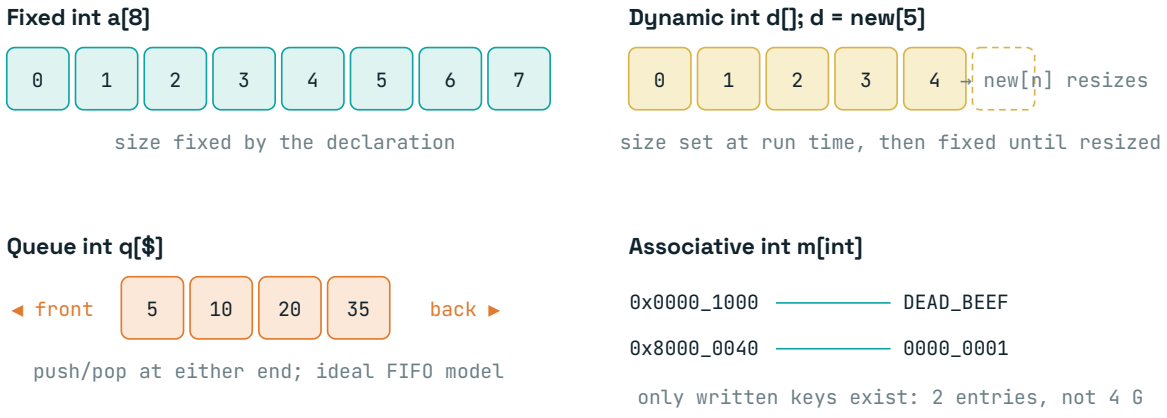
Dynamic array
Declared with `[]`, sized at run time with `new[n]`.

Queue
Declared with `[$]`; grows and shrinks at both ends.

Associative array
Indexed by any key type; stores only the entries written.

Array method
Built-in operation such as `sum()`, `find`, `sort()`.

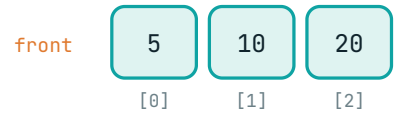
`with` clause
An expression applied to each `item` in methods like `find`.



Pick the container by how the size behaves. Fixed for tables, dynamic for random lengths, queues for FIFOs and expected lists, associative for sparse memories and counters.

Try it: queue playground

7




```
int q[$] = '{5, 10, 20};
q.size() = 3   q[0] = 5   q[$] = 20
```

Queue operations

Call	Effect	Starting from '{1, 2, 3}
`q.push_back(4)`	Add at the back	1 2 3 4
`q.push_front(0)`	Add at the front	0 1 2 3
`x = q.pop_front()`	Remove and return the first	x = 1, q = 2 3
`x = q.pop_back()`	Remove and return the last	x = 3, q = 1 2
`q.insert(1, 9)`	Insert at index	1 9 2 3
`q.delete(0)`	Delete at index	2 3
`q.size()`, `q[$]`	Length; last element	3; 3

Associative arrays

```
bit [31:0] mem [bit [31:0]];        // sparse 4 GB memory model
mem[32'h1000] = 32'hDEAD_BEEF;
if (mem.exists(32'h2000)) ...
foreach (mem[addr]) $display("%h: %h", addr, mem[addr]);
int hits[string];  hits["read"]++;          // counters by name
```

Method	Does
`exists(k)`	1 if key present
`delete(k)` / `delete()`	Remove one / all
`num()` / `size()`	Number of entries
`first(k)`, `next(k)`, `last(k)`, `prev(k)`	Walk keys in order

Array methods worth memorising

Method	Returns	Example
`sum()`, `product()`	a value	`payload.sum()`
`min()`, `max()`, `unique()`	a queue	`q.max()`
`find with (item > 3)`	queue of matches	`q.find with (item > 3)`
`find_index with (...)`	queue of indexes	`q.find_index with (item == 0)`
`sort()`, `rsort()`, `reverse()`, `shuffle()`	reorders in place	`q.sort()`

Reading a key that isn't there

Reading a missing associative key returns the default value (0) and, on most simulators, a warning. In a memory model always check `exists()` first, or decide deliberately what an unwritten address returns.

Common mistakes

Indexing past the end of a queue (returns default, no error on many tools) · forgetting `new[n]` before using a dynamic array · iterating a queue while popping from it · using an associative array where order matters without `first/next`.

INTERVIEW CORNER

Q. Which container models a FIFO's expected data? **A.** A queue: `push_back` on write, `pop_front` on read.

Q. Why an associative array for a memory model? **A.** Only written addresses consume storage — a 4 GB space with a few hundred entries.

Key takeaways

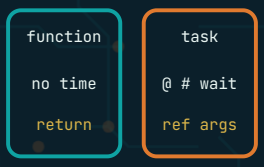
1. Queue = FIFO model; associative = sparse memory or counters.
2. Dynamic arrays need `new[n]` .
3. Array methods replace hand-written loops for sums, searches and sorts.

MODULE 3 OF 4

Tasks, functions & procedural code

Routines that scale: automatic lifetime, ref arguments, defaults and loops.

- Function vs task
- automatic vs static
- ref and const ref
- Loops and jumps



2 hours | Area 03 · SystemVerilog

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

STEP 1

Function or task?

In plain words

A **function** calculates and returns a value in zero simulation time. A **task** can wait — for a clock edge or a delay.

Think of it like

A function is a calculator: instant answer. A task is an errand: it takes time.

Worked example

```
function int parity(bit [7:0] b); return ^b; endfunction
task wait_cycles(int n); repeat (n) @(posedge clk); endtas
```

The word for it

function (no time, returns a value) and **task** (may consume time).

Why it matters

Reference models are functions; drivers and monitors are tasks.

2

STEP 2

Passing arguments: copy or link

In plain words

By default an argument is **copied** in. With `ref` the routine works on the caller's own variable.

Think of it like

Emailing a copy of a document versus sharing a link to the original.

Worked example

```
function void clear(ref int q[$]); q.delete(); endfunction
```

The word for it

Directions `input`, `output`, `inout`, `ref`, and `const ref` (link, read-only).

Why it matters

`const ref` avoids copying big arrays; plain `ref` lets a task see values change while it waits.

3

STEP 3

automatic versus static

In plain words

An `automatic` routine gives every call its own local variables. A `static` one shares a single set between all calls.

Think of it like

Everyone getting their own notepad (automatic) versus everyone writing on one whiteboard (static).

Worked example

1. Two threads call the same static task at once.
2. Both write the same local variable — one overwrites the other.
3. Make it `automatic` and each call has its own copy ✓

The word for it

automatic and **static** lifetime. Class methods are automatic by default.

Why it matters — Testbenches run many threads; static locals create bugs that appear only sometimes.

STEP 4

4 Loops and jumps

In plain words — SystemVerilog has `for`, `foreach`, `repeat`, `while`, and `break` / `continue`.

Think of it like — Walking down a row of lockers, skipping the empty ones and stopping at the first full one.

Worked example

```
foreach (q[i]) begin
  if (q[i] == 0) continue;   // skip zeros
  if (q[i] < 0)  break;      // stop at a negative
  total += q[i];
end
```

The word for it — **foreach** walks every index without you writing the bounds.

Why it matters — Off-by-one loop bounds are a common testbench bug; `foreach` removes them.

PAUSE & TRY

For `q = {2, 0, 5, -1, 7}`, what `total` is after the loop above?

Answer

2 + (skip 0) + 5, then −1 stops the loop → **7**.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

Function
Returns a value (or `void`) and may not consume simulation time.

Task
May consume time (`@`, `#`, `wait`); returns results through output arguments.

automatic
Each call gets its own copy of local variables — required for recursion and parallel calls.

static
One shared copy of local variables for all calls — the default inside modules.

`ref` argument
Passed by reference: no copy, and the routine sees changes made elsewhere.

`const ref`
By reference but read-only: speed of `ref` without the risk.

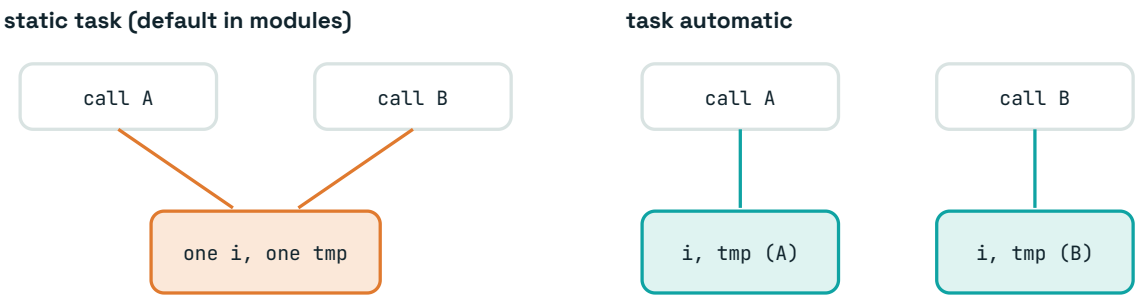
	function	task
Consumes time	no	yes
Returns	a value or void	via `output` / `ref`
Can call	functions	functions and tasks
Typical use	parity, models, formatting	drive, wait, sample

```
function automatic int parity(input bit [31:0] v);
  return ^v;
endfunction

task automatic drive(ref logic [7:0] bus, input byte val, input int cycles = 1);
  bus = val;
  repeat (cycles) @(posedge clk);
endtask

drive(.bus(d), .val(8'h5A));          // named arguments; cycles uses its defaul
```

Why every testbench routine is automatic



Static storage is shared. Two threads calling a static task overwrite each other's variables; automatic gives each call its own frame.

Loops and jumps

Construct	Example
`foreach`	`foreach (arr[i]) sum += arr[i];` · 2-D: `foreach (m[i, j])`
`for`	`for (int i = 0; i < n; i++)`
`repeat`	`repeat (8) @(posedge clk);`
`while` / `do … while`	`while (!done) @(posedge clk);`
`break` / `continue` / `return`	Leave a loop / skip to next / leave a routine

× **Common mistakes**

Omitting `automatic` on testbench tasks · calling a task from a function (compile error) · passing large arrays by value in hot loops · relying on `ref` in a static task.

INTERVIEW CORNER

Q. Can a function call a task? **A.** No — a function can't consume time, and a task might.

Q. What does `ref` give you? **A.** Pass by reference: no copy and the callee sees live changes; `const ref` makes it read-only.

Key takeaways

1. Functions compute, tasks take time.
2. Write `automatic` on every testbench routine.
3. Use `const ref` for large read-only arguments.

MODULE 4 OF 4

Threads & communication

Running many things at once — and keeping them in step.

- fork with join, join_any, join_none
- disable fork and wait fork
- events and semaphores
- Mailboxes between components



⏱ 3 hours | Area 03 · SystemVerilog

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

1 STEP 1 — Doing several things at once

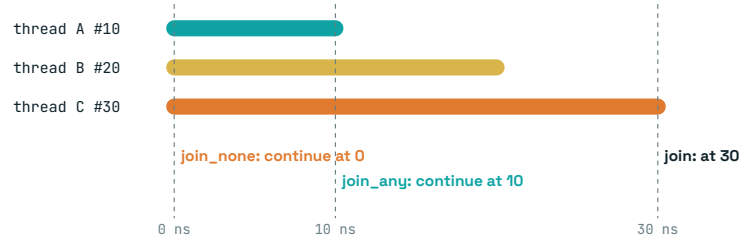
≈ In plain words	A testbench must drive inputs, watch outputs and check results at the same time. Each of those runs as a separate **thread**.
☀ Think of it like	Cooks in a kitchen: one chops, one stirs, one plates — all at once.
✎ Worked example	A driver thread sends packets while a monitor thread collects outputs and a checker thread compares them.
Aa The word for it	**Thread** (a process).
! Why it matters	Every UVM component you'll write later runs in its own thread.

2 STEP 2 — fork … join, join_any, join_none

In plain words — `fork` starts several threads. The ending decides when the parent continues.

Think of it like — Waiting for all your friends to arrive (join), for the first one (join_any), or not waiting at all (join_none).

Worked example



Three threads that take 10, 20 and 30 ns.

The word for it — **join** (wait for all), **join_any** (first), **join_none** (don't wait).

Why it matters — Timeouts are written as “fork the test and a timer, join_any” — whichever finishes first wins.

PAUSE & TRY **Threads of #5, #15 and #40 inside `fork` … `join_any`, started at time 0. When does the parent continue?** Answer

When the first one finishes: at **5 ns**.

3 STEP 3 Events: a doorbell between threads

In plain words — One thread triggers an event; others waiting for it wake up.

Think of it like — A doorbell: someone presses it, everyone listening reacts.

✎ **Worked example**

```
event done;
initial begin #50 -> done; end       // ring
initial begin @(done); $display("woke at %0t", $time); end
```

Aa **The word for it** — **event**, trigger `->`, wait `@` or `wait(e.triggered)`.

! **Why it matters** — Simple synchronisation: “start checking once reset has finished”.

4 STEP 4 — Semaphores: one key for a shared room

≈ **In plain words** — A semaphore holds a number of keys. A thread takes a key before using a shared resource and gives it back afterwards.

✳ **Think of it like** — One key to a meeting room: whoever holds it uses the room; others wait.

✎ **Worked example**

```
semaphore bus_key = new(1);
bus_key.get();  drive_bus();  bus_key.put();
```

Aa **The word for it** — **semaphore**; `get`, `put`, `try_get`.

! **Why it matters** — Stops two drivers from driving the same bus at once.

5 STEP 5 — Mailboxes: a postbox between components

≈ **In plain words** — A mailbox passes data from one thread to another, in order. `get` waits if it's empty.

✳ **Think of it like** — A postbox: the sender drops letters in; the receiver collects them one by one, oldest first.

✎ **Worked example**

```
mailbox #(int) mb = new();
initial repeat (3) mb.put($urandom_range(1, 9));          //
initial forever begin int v; mb.get(v); drive(v); end     //
```

Aa **The word for it**

mailbox; `put`, `get`, `try_get`, bounded size.

! **Why it matters**

This generator → driver hand-off is the same idea as UVM's sequencer and driver.

6 STEP 6 — Putting it together: a FIFO checked by a queue

≈ **In plain words**

To check a FIFO, keep a queue as the model: push on every write the FIFO accepts, pop and compare on every read.

☀ **Think of it like**

Keeping your own list of what you put into a locker, and checking each item as it comes out.

✎ **Worked example**

1. Write 5 → model {5}. Write 8 → model {5, 8}.
2. Read → FIFO gives 5; model pops 5 ✓.
3. FIFO full and a write is attempted → the FIFO must **not** accept it, so the model must not push.

Aa **The word for it**

A **scoreboard** with a **queue model**.

! **Why it matters**

Lab 03's planted bug writes when the FIFO is full — the queue model is what exposes it.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

Thread / process

An independent flow of execution; `initial`, `always` and `fork` branches are threads.

fork … join

Start several threads; the parent waits for all of them.

join_any / join_none

Parent continues after the first child finishes / immediately.

disable fork

Kill all child threads still running from the current thread.

event

A named trigger: `-> e` fires it, `@e` waits for it.

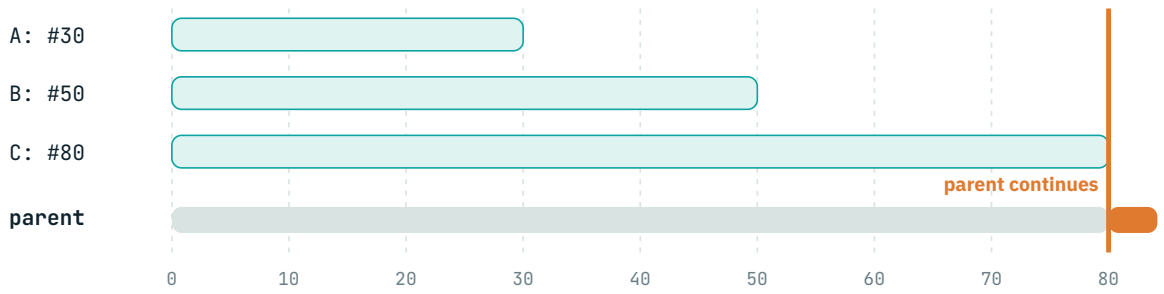
semaphore

A bucket of keys that limits how many threads use a resource.

mailbox

A thread-safe FIFO for passing messages (transactions) between threads.

Try it: fork variants



join — the parent waits until every child finishes: it continues at t = 80.

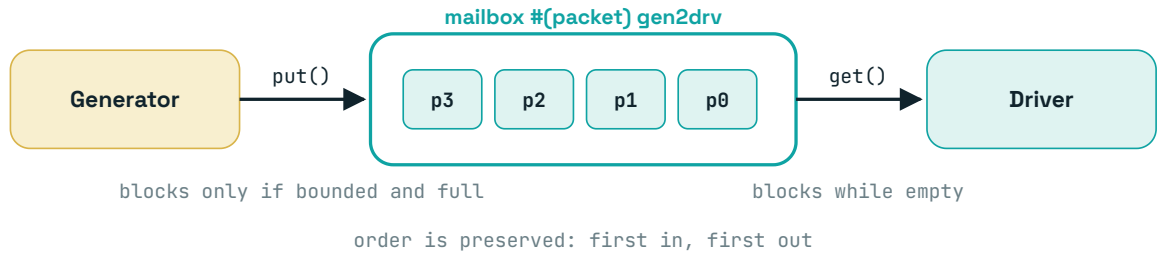
```
fork
  begin wait (done); end
  begin repeat (1000) @(posedge clk); $error("timeout"); end
join_any
disable fork;                      // stop whichever is still running

for (int i = 0; i < 4; i++) begin
  automatic int k = i;             // capture the loop variable!
  fork drive_port(k); join_none
end
wait fork;                         // wait for every spawned child
```

The loop-variable trap

Without `automatic int k = i;` every `join_none` thread starts after the loop has finished and sees `i == 4`.

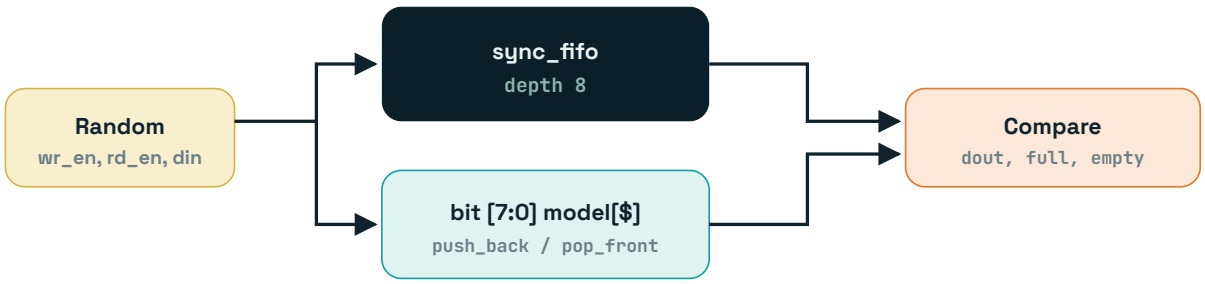
Events, semaphores, mailboxes



A mailbox is the pre-UVM connection between components. UVM's TLM FIFOs do the same job from Area 08.

Primitive	Create	Key calls
event	`event e;`	`-> e;` · `@e;` · `wait (e.triggered);`
semaphore	`semaphore s = new(1);`	`s.get(1)` · `s.put(1)` · `s.try_get(1)`
mailbox	`mailbox #(packet) m = new();`	`put` · `get` (blocks) · `try_get` · `peek` · `num`

The lab: a FIFO checked by a queue



Mirror every accepted operation. Accepted write → `push_back`; accepted read → `pop_front` and compare; flags checked against `model.size()` every cycle.

Common mistakes

Forgetting `disable fork` after a watchdog (threads leak into later tests) · `@e` missing an event that fired in the same time step (use `wait (e.triggered)`) · unbounded mailboxes hiding a stuck consumer.

INTERVIEW CORNER

Q. join_any + disable fork vs join_none + wait fork? **A.** The first waits for the first finisher then kills the rest; the second starts everything and later waits for all to finish.

Q. Mailbox vs queue? **A.** A mailbox is thread-safe and `get` blocks until data arrives; a queue is just storage.

Key takeaways

1. `join` waits for all, `join_any` for one, `join_none` for none.
2. Capture loop variables before forking.
3. Mailboxes connect generator to driver; semaphores guard shared resources.

PRACTICE

Check yourself

Answer in your head, then open each card. The area test has 20 more.

A DUT output is X and you assign it to `bit b`. What is b?

0 — two-state types turn X and Z into 0.

Which container models a sparse 4 GB memory?

An associative array keyed by address.

What does `q.pop_front()` return for '{5, 7, 9}?

5; the queue becomes '{7, 9}.

Can a function contain `@(posedge clk)`?

No — only tasks may consume time.

Why must testbench tasks be automatic?

So parallel or recursive calls get their own variables.

After `fork … join_any`, how do you stop the remaining threads?

```
disable fork;
```

What does `mailbox.get()` do when the mailbox is empty?

It blocks until an item arrives.

In the FIFO lab, what must `full` equal every cycle?

`model.size() == DEPTH`.

REFERENCE

Interview questions

Asked in DV and RTL interviews. Try an answer out loud before opening.



Explain the four SystemVerilog array types and when to use each.

Fixed (static tables), dynamic (run-time length, e.g. payloads), queue (FIFO models, expected lists), associative (sparse memories, counters by key).

What's the difference between `==` and `===` on logic?

== returns X when any bit is X/Z; === compares X and Z literally and returns 0 or 1.

Why might a test pass on one simulator and fail on another?

Races, reliance on X turning into 0 in two-state variables, or unspecified ordering such as reading an associative array without ordered iteration.

How do you implement a timeout in a testbench?

fork the wait and a delay, `join_any`, then `disable fork`; report an error if the delay won.

What is a semaphore used for in a testbench?

Serialising access to a shared resource, such as one bus driven by several sequences.

How do you pass a large array to a task efficiently and safely?

`const ref`.

What does `wait fork` wait for?

All child threads spawned by the current thread.

Enum assigned an out-of-range integer — what happens?

Compile error without a cast; with a cast the value is stored but `.name()` returns an empty string.

REFERENCE

Glossary

Every term used on this page, A to Z.

A a

Associative array
Sparse array indexed by any key type.

automatic
Per-call storage for routine variables.

bit
Two-state 1-bit type.

const ref
Read-only pass by reference.

disable fork
Kills child threads.

Dynamic array
Array sized at run time with new[n].

enum
Type with named values.

event
Named trigger between threads.

fork / join
Parallel threads with different waiting rules.

Function
Zero-time routine that returns a value.

logic
Four-state type for DUT signals.

Mailbox
Thread-safe message FIFO.

Packed struct
Struct stored as one bit vector.

Queue
Variable-size array with front/back operations.

ref
Pass by reference.

Semaphore
Key bucket guarding a resource.

static
Shared storage for routine variables.

string
Dynamic text type.

Task
Routine that may consume time.

typedef
Names a type.

Unpacked struct
Record that may hold dynamic types.

wait fork
Waits for all child threads.

REFERENCE

FAQ & troubleshooting

The problems learners hit most in this area, and the fix for each.



You see	Why	Fix
`Unsupported: associative array` (Icarus)	Icarus has limited SystemVerilog support	Use Verilator 5 (`make`) or EDA Playground
Verilator: `Need --timing`	Delays and fork timing need the timing option	Compile with `verilator --binary --timing`
All forked threads print the same index	Loop variable not captured	`automatic int k = i;` before `fork`
Simulation hangs at the end	A thread waits on an empty mailbox or an event that never fires	Add a watchdog; check producer counts
Lab says `FAIL: nothing was checked`	Model TODOs still empty	Complete TODO 1–4 in `tb_sync_fifo.sv`

REFERENCE

Further reading

Go deeper when you are ready.



Resource	Why read it
IEEE 1800 SystemVerilog standard — data types, arrays and processes chapters	Definitive behaviour of every feature on this page
Chris Spear & Greg Tumbush, *SystemVerilog for Verification*	The standard text for testbench SystemVerilog
Stuart Sutherland, Simon Davidmann, Peter Flake, *SystemVerilog for Design*	Types, structs and enums from the design side
Verilator manual — SystemVerilog support and `--timing`	What the free simulator supports

HANDS-ON LAB · ABOUT 3 HOURS

Lab 03 · Queue-checked FIFO

You get `sync_fifo.v` (depth 8, one planted bug), `tb_sync_fifo.sv` with biased random push/pop traffic and an empty reference model, and a Makefile for Verilator.

1. Run `make`: with the model empty it reports `FAIL: nothing was checked`.
2. TODO 1–2: check `full` and `empty` against `model.size()`; on an accepted read pop the expected value and compare with `!==`.
3. TODO 3–4: push accepted writes; count operations in an associative array.
4. Run again: the first error points at the bug — note what traffic was happening. Fix the RTL and re-run.

Done when you see `PASS: … reads checked, 2000 cycles`.