CareerSwami · VLSI & Embedded Hardware

FREE TWO-WEEK INTERNSHIP

Project 1 — Design a water-tank controller that handles a broken sensor

Two sensors, one pump, four possible sensor readings — and one of those four is impossible. A controller that only handles the three sensible cases is the one that floods a building when a wire comes loose.

2 weeks	1 project	6–8 hrs
START TO SUBMIT	ONE SUBMISSION	PER WEEK

WHAT IS IN THIS BRIEF

1. The situation — the problem you are being handed
2. The four behaviours that must be exactly right — the cases your work must survive
3. What you are given — your data and starting files
4. What finished looks like — the two things you hand in
5. How the two weeks run — week by week
6. Before you start — rules, tools and your starter kit
7. How the work is judged — the marking scheme
8. How to submit — one link, pasted into your dashboard

Read 1 to 4 before you do anything. They take five minutes and they are what the marking scheme is built on. Sections 5 to 8 are the how — come back to them once you know what you are solving.

1 The situation

Treat this as a real client brief. The tank is simulated, but the design discipline — specify, then verify against the specification rather than against the code — is exactly how hardware is actually built.

A water tank has two level sensors. The low sensor reads one when water reaches the low mark; the high sensor reads one when water reaches the high mark. A pump fills the tank. There is a clock and a reset.

Both sensors at zero means empty: start the pump. Low at one and high at zero means mid: keep pumping. Both at one means full: stop. The fourth combination — low at zero and high at one — says the water is above the high mark and below the low mark at the same time. That is physically impossible, so it means a sensor or a wire has failed.

When that happens the controller must latch a fault, stop the pump, and stay that way until it is reset. Not recover if the reading flickers back, not keep pumping while it thinks about it. Latch, stop, wait — because the thing you are protecting against is a sensor you can no longer trust.

WHAT YOU ARE BEING ASKED TO DO

Specify the controller as a state machine, write ten test sequences that prove it behaves correctly — including the invalid and reset cases — then implement it and show the waveforms that confirm the specification was met.

2 The four behaviours that must be exactly right

Three of these are ordinary operation. The fourth is the one this project exists for, and your ten planned sequences must cover all of them.

Empty to full, pump staying on

Start at 00, continue through 10, stop at 11.

Where designs fail — A design that decides purely from the current inputs will switch the pump off the instant the level dips below the low mark on the way up, then on again, chattering the relay. The controller needs state, not just a combinational decision.

Full, draining to mid

Stay off until empty.

Where designs fail — Turning the pump on the moment the high sensor drops gives you a pump cycling on every ripple. The specification says pumping starts at empty, so mid-on-the-way-down is a different state from mid-on-the-way-up — which is exactly why a truth table is not enough here.

The invalid reading, 01

Latch fault, stop the pump, stay there.

Where designs fail — Treating it as a don't-care leaves the behaviour to whatever the synthesis tool chose, which is unpredictable and untestable. Treating it as transient and recovering when it clears defeats the purpose — the sensor is not to be trusted any more.

Reset

Clears the fault and stops the pump.

Where designs fail — A reset that clears the fault but leaves the pump output in its previous state is a real and dangerous bug. Define every output's value in reset explicitly rather than leaving any of them to inference.

Timing matters too: inputs are sampled on the clock edge and the state updates after it. Writing a testbench that changes an input on the same edge the design samples it produces results that depend on simulator

ordering, and you will chase that for a day.

3 What you are given

The interface and the behaviour are fully specified. The state encoding, the implementation and the verification strategy are yours.

The interface

`in: clk` `in: reset` `in: low_sensor` `in: high_sensor` `out: pump_on` `out: fault`

The sensor encoding, so there is no ambiguity

`low=0, high=0`	Empty — start the pump
`low=1, high=0`	Mid — keep pumping if already on; stay off if draining
`low=1, high=1`	Full — stop the pump
`low=0, high=1`	Impossible — latch fault, stop the pump
`While faulted`	Pump stays off regardless of sensor readings
`Reset`	Clears fault, stops the pump, returns to the start state
`Timing`	Inputs sampled on the clock edge; state updates after it

Four behaviours to verify

`Fill` `Hold` `Fault` `Reset`

Sensors are simulated; no physical hardware, water or electrical work is involved in this project. Where the specification leaves something open — your state encoding, whether mid-while-draining is its own state — decide, write it into the specification, and verify it. A behaviour you did not specify cannot be verified, only observed.

4 What finished looks like

At the end of two weeks someone should be able to read your specification, run your simulation, and confirm the design matches what you wrote.

- **A specification, a state diagram and ten planned sequences** — every state, every transition and every output written down in words; a diagram showing the states with their transition conditions labelled; and ten input sequences planned *before* implementation, each with the output you expect at every clock tick.
- **The source, a documented simulator setup and annotated waveforms** — the implementation; exact instructions for running the simulation with versions, so someone else gets your results; and waveform

captures or logs annotated to show which part of the specification each one demonstrates.

Write the ten sequences before the code. Written afterwards they test what you built, not what was specified — and the difference between those two is where every hardware bug lives.

5 How the two weeks run

Work through it in two parts — but you submit **once**, at the end, not once per week.

PART 1 · WEEK 1

Specify it completely, then plan how you will prove it

Write the state machine out in full — every state, every transition, every output in every state — and draw it. Then plan ten input sequences with expected outputs, before any code exists.

Produce

- A written specification: states, transitions, outputs in each state
- A state diagram with every transition condition labelled
- Ten planned input sequences with the expected output at each tick

> Every state must define both outputs, pump_on and fault — no implicit values. Your sequences must include the invalid 01 reading, a reset while faulted, and a full fill-then-drain cycle.

PART 2 · WEEK 2

Implement it and show the waveforms

Build the design, get the simulation running reproducibly, and capture waveforms annotated against the specification clauses they demonstrate.

Produce

- The source, with the state encoding and reset behaviour explicit
- A documented simulator setup: tool, version, exact commands
- Annotated waveforms or logs, each tied to a specification clause

> Sample inputs on the clock edge and update state after it. Show the fault latching and surviving a return to a valid reading — a waveform where the fault clears by itself is a failed design, not a quirk.

6 Before you start

- Plan the ten sequences before writing any code. This is the single most important instruction in the brief.
- Define both outputs in every state, including reset. Leave nothing to inference.

- The invalid reading must latch. A fault that clears when the reading returns to normal has not understood the requirement.
- Do not change testbench inputs on the same clock edge the design samples. Drive them on the opposite edge or with a small offset.
- Mid on the way up and mid on the way down may or may not be the same state in your design. Decide, specify it, and test both paths.

Your tools

Verilog or VHDL with Icarus Verilog and GTKWave, or Verilator, or EDA Playground in a browser. All free. A Python state-machine model is an acceptable substitute if hardware tooling is genuinely unavailable — say so if you take that route. No paid tool is required, no physical hardware is required, and no ID document is ever required.

CS_vlsi-embedded-hardware_Starter_Kit.zip — planned-sequences.csv, a specification template, a state diagram template, a testbench skeleton, an evidence log, and a file index.
https://careerswami.com/wp-content/uploads/careerswami-projects/CS_vlsi-embedded-hardware_Starter_Kit.zip

7 How the work is judged

WHAT IS ASSESSED	WEIGHT
Problem and requirements — did you understand what you were solving	**20%**
The core artefact — the thing you actually made	**40%**
Validation and evidence — how you know it works	**25%**
Clarity and reproducibility — can someone else open and follow it	**15%**

Strong	Complete, correct, evidence-backed and easy to inspect.
Developing	A useful partial result, with the gaps explained.
Needs revision	Core deliverable missing, claims unverified, or the work cannot be opened.

Credit reflects demonstrated work, not visual polish. A small, honest result beats a large, unsupported one.

What counts as evidence

expected result → actual result → the annotated waveform or simulation log → conclusion. Then say what the evidence does not establish.

Why this one is worth doing. Hardware is unforgiving because you cannot patch it after it ships, and the thing that separates an engineer from a student is a specification written before the code and tests written against the specification rather than the implementation. The invalid sensor state is in this brief for one reason: in the real world, wires come loose, and the controller that had no plan for it is the one that caused the damage.

8 How to submit

One link, pasted into your dashboard

1. Put your work somewhere with a link — Google Drive, Figma, GitHub, Notion, Loom. Whatever fits the work.
2. **Set it so anyone with the link can view.** A locked link is the single most common reason a submission comes back.
3. Include a short README or cover page saying what to open first, and where each deliverable lives.
4. Open your dashboard, paste the link into the submit box, and add a line or two on anything you want the reviewer to know.

Name files `CS_vlsi-embedded-hardware_YourName_v1`. Keep your own copy. A screenshot on its own is not enough for anything interactive. A video is optional — annotated screenshots or narrated slides are fine.

Submitted and heard nothing, or the link came back? Email **support@careerswami.com** with the email address you enrolled with.

Work honestly. Use fictional data and your own work. Credit anything you reused, and disclose AI assistance — including what you checked yourself. Never invent interviews, users, tests or results. Nothing here requires you to spend money, contact real organisations, or deploy anything publicly. Never upload identity documents, real customer records or credentials.

Stuck, confused, or something is broken? Write to us.

`support@careerswami.com`

Or WhatsApp **+91 7383138418**. We answer on working days. Use it for anything at all — a part of this brief that does not make sense, a starter file that will not open, a dashboard link that has stopped working, a deadline you need moved, or a submission you are not sure about.

To get a useful answer fast, spend twenty minutes narrowing the problem down first, then tell us four things: what you expected, what actually happened, what you already tried, and your one specific question. Include your track name and the email you enrolled with.

Asking is not penalised. Submitting partial work with a clearly stated blocker always scores better than a finished answer you copied, and better than a silent no-show.

Optional stretch

Optional, and only once the main work is done — cap it at an hour. Add assertions that check the invariants continuously, or a debounce stage on the sensor inputs. Label it clearly as a stretch.
