Excellent.
Now we move into **PHASE 2**, which is where many engineers *think* they are strong—but most are not.

I'll explain **Phase 2 completely, cleanly, and practically**, exactly how it must be done on a real project, without skipping reasoning steps or jumping to registers.

🧩 PHASE 2 – CORE ARCHITECTURE & PROCESSING MODEL

(How computation actually happens inside the chip)

Duration

🕑 1–2 weeks (depending on background)

Phase-2 Goal

> Build a **mental execution model** of the processor
> so that you can **predict behavior, debug failures, and reason about performance**.

This phase answers:

- *What happens from reset to first instruction?*
- *How does a core execute code?*
- *How does memory access really work?*
- *Why multicore is hard?*

1️⃣ What Phase 2 IS and IS NOT

Phase 2 IS:

- Understanding **execution flow**
- Understanding **core, cache, memory relationships**
- Understanding **privilege and control**
- Understanding **multi-core interactions**

Phase 2 is NOT:

❌ Writing drivers
❌ Tuning performance

❌ Studying every register
❌ OS internals

2️⃣ Step 1: Reset → First Instruction (The Most Important Flow)

If you understand this, **everything else becomes logical**.

Questions you must answer

- Who applies reset?
- What logic is alive after reset?
- Where is the first instruction fetched from?
- Who decides the boot source?

General processor flow

```
Power stable
 → Reset deasserted
 → Core released
 → Reset vector read
 → First instruction fetched
```

For T2080 (Power Architecture)

- Reset releases **one core**
- Core fetches instruction from **Boot ROM**
- Boot ROM behavior is influenced by **RCW**
- Everything else is disabled initially

📌 **Key insight:**

Nothing runs magically. Hardware decides first.

3️⃣ Step 2: CPU Core Basics (Without Micro-Details)

You do NOT start with pipeline stages.

What you must understand:

- Instruction fetch
- Decode

- Execute
- Write-back (conceptually)

Power Architecture essentials

- RISC-based
- Load/store architecture
- Fixed instruction width (mostly)
- Strong deterministic behavior

📌 Why this matters:

- Predictable execution
- Easier real-time guarantees
- Different from ARM/x86 assumptions

4️⃣ Step 3: Privilege Levels & Control

This is critical for debugging and OS interaction.

Typical privilege levels:

- User mode
- Supervisor / Kernel mode
- Hypervisor (in advanced SoCs)

Key questions:

- Who can touch hardware registers?
- Who configures MMU?
- Who controls cache?

For T2080:

- Early boot runs in **highest privilege**
- OS later drops privilege
- Some registers become locked

📌 **Debug insight:**

If you don't know privilege, you'll chase "permission bugs" forever.

5️⃣ Step 4: Memory Hierarchy (Core → Cache → DDR)

This is where most real bugs hide.

Memory layers:

```
Core registers
 → L1 cache
 → L2 cache
 → (L3 cache, if present)
 → DDR
```

What you must understand:

- Why cache exists
- Latency vs bandwidth
- Cache line concept
- Cache coherency

📌 **Golden rule:**

Memory is slow. Cache hides that fact.

6️⃣ Step 5: Cache Coherency (Multicore Reality)

Multicore **without coherency = chaos**.

You must understand:

- What cache coherency means
- Why hardware fabric exists
- What happens when coherency breaks

Real-world failure:

- Random crashes
- Stale data
- Non-reproducible bugs

📌 This is why networking SoCs invest heavily in coherency fabric.

7️⃣ Step 6: Endianness (Silent Killer)

Endianness errors look like “random data corruption”.

Concepts:

- Big-endian vs little-endian
- How memory is laid out
- Why networking prefers big-endian

T2080 context:

- Power Architecture traditionally big-endian
- Software must agree with hardware

📌 **Debug signal:**

Correct values in registers, wrong in memory = endianness mismatch.

8️⃣ Step 7: MMU Basics (Without OS Deep Dive)

You must know:

- Virtual vs physical memory
- Address translation concept
- Why MMU exists

Why this matters:

- Device memory vs normal memory
- Cacheable vs non-cacheable
- DMA correctness

📌 **Bring-up insight:**

Wrong MMU settings cause invisible bugs.

9️⃣ Step 8: Multicore Boot Model

Key questions:

- Which core boots first?
- How are other cores released?
- Who initializes shared resources?

Typical model:

- Core 0 boots
- Initializes memory, cache
- Releases other cores

📌 **Bug pattern:**

Two cores touching hardware too early = deadlock.

🔍 Step 9: Phase-2 Practical Exercises (Mandatory)

📝 Exercise 1: Instruction Flow Diagram

Draw reset → first instruction → memory access path.

📝 Exercise 2: Cache Thought Experiment

Explain what happens when two cores write the same variable.

📝 Exercise 3: Memory Access Path

Trace one load instruction from core to DDR.

📘 Step 10: Phase-2 Output (Non-Negotiable)

Before moving forward, the junior must produce:

📄 A clean explanation (verbal or written) of:

1. Reset-to-execution flow
2. Core → cache → memory path
3. Cache coherency importance
4. Privilege & control roles
5. Multicore startup sequence

Evaluation criteria:

- No jargon dumping
- Correct sequencing
- Real-world awareness

🚨 Phase-2 Failure Signals

Phase 2 has failed if the junior:

❌ Talks about cache as “speed booster” only

❌ Cannot explain first instruction fetch

❌ Confuses physical and virtual memory

❌ Ignores coherency
❌ Cannot explain multicore behavior

🎯 Phase-2 Completion Statement

When Phase 2 is done, the junior should be able to say:

> "I can mentally simulate how code executes on this processor, even before running it."

That's the **real milestone**.

🧭 What Comes Next

Phase 3 will bring this knowledge into **real hardware responsibility**:

- Power domains
- Reset trees
- Clocking
- Sequencing failures

This is where boards live or die.

If you want, next I can:

- Proceed to **Phase 3 – Power, Reset & Clock**
- Convert Phase 2 into a **training worksheet**
- Show **Phase-2 interview questions with ideal answers**

Just tell me 👍