1. Project Overview

You are working for Space-Z, a leading space exploration company founded by Gen-Z scientists, responsible for the interplanetary robotic communication for the Mars Robotic Mission. Commands sent to the robot travel over a noisy deep-space channel where bit flips can occur due to radiation and weak signal conditions. Your team will design an Error Correction Code (ECC) subsystem using a Polar code plus an embedded CRC so the receiver can both:

(a) correct some errors, and

(b) detect when errors are likely uncorrectable and declare the command invalid.

You will implement:

- a Polar encoder (in Verilog or SystemVerilog),
- a Polar decoder (in Verilog or SystemVerilog),
- a CRC module (in Verilog or SystemVerilog).

You will be provided with a basic SystemVerilog testbench, and an elaborate hidden testbench will be used for final grading.

2. What is Team-Based Learning (TBL)?

Team-Based Learning (TBL) is a structured active-learning method where students work in competing teams to solve authentic engineering tasks under time constraints.

A typical TBL experience includes:

- competing teams (here: ~20 teams × 6 students/team)
- preparation and readiness (short pre-reading / brief lecture or quiz)
- team application exercise (the main build-and-verify task)
- immediate feedback (simulation results, checkoffs, test outcomes)
- peer contribution expectations (role assignment, accountability)

In this project, TBL is implemented by having each team:

- assign clear roles (including an AI role),
- parallelize encoder/decoder/verification work,
- integrate quickly,
- demonstrate working simulation behavior in Vivado xsim.

3. Project Background

An Error Correction Code (ECC) adds structured redundancy to data so that a receiver can detect and/or correct errors introduced by a communication channel.

Key terms

- Block length (N): number of transmitted bits per codeword
- Information bits (K): number of useful payload bits (may include CRC)
- Code rate (R): $R = K/N$. Lower rate ⇒ more redundancy ⇒ typically better protection.
- Error detection vs. correction:
    - detection: receiver can tell something is wrong (e.g., CRC fails)
    - correction: receiver can recover the original message despite errors

CRC inside ECC (fail-safe design)

A CRC (Cyclic Redundancy Check) is an error-detection code. In this project, CRC-16 is included *inside* the ECC payload so that after decoding, the receiver can verify integrity. If CRC-16 fails, the command is marked invalid and must not be used. While CRC-16 check is not a 100% guarantee, it is used extensively in the commercial world because of its high reliability. The probability of incorrect CRC-16 check is `2^-16 ≈ 0.0015%`. This supports the robotic-control safety rule: prefer no action to wrong action.

4. Team Organization and Role Assignment

Each team of 6 must assign at least the following roles. One person may hold more than one role, but every role must be clearly owned.

(a) Team Leader

- coordinates task split, integration schedule, and final submission quality
- ensures the team meets interface specs and deadlines

(b) Encoder Engineers

- implement CRC generation + Polar transform encoding
- produce known-good vectors for verification

(c) Decoder Engineers

- implement the decoding strategy and CRC check
- ensure fail-safe `valid` behavior

(d) Verification Engineers

- use the provided `tb_basic.sv` to run Vivado xsim, check waveforms, create regression scripts

- maintain a small set of directed tests (no-error, 1-3bit flips, 4-bit flips, 5+-bit flips)
- own top-level integration, README, and the 2-page report summary
- ensure a clean, reproducible `xsim` command line / Makefile target

(e) AI Engineers (Prompt & Code Auditor)

- use AI tools to accelerate boilerplate and algorithm translation
- maintain a prompt log and ensure generated code matches the spec
- review for common AI mistakes (bit ordering, CRC conventions, interface timing)

5. AI Usage Policy

AI tools are permitted to improve productivity (boilerplate RTL, CRC logic, scaffolding, comments). However:

- Your team is responsible for correctness, integration, and understanding.
- You must submit `ai_log.txt`:
  - prompts used,
  - what code was adopted,
  - what was modified and why.
- Teams may be asked brief oral questions about:
  - CRC bit ordering,
  - frozen-bit mapping,
  - encoder transform definition,
  - validity rules.

Using AI does not reduce the expectation that you can explain your design.

6. Polar Code Specification

This project uses a fixed parameter set so every team can build a similar system.

6.1 Parameters

- Code length (N) = `64`
- Payload data bits = `24`
- CRC bits = `16`
- Total information bits (K) = `40` (`24` data + `16` CRC)
- Total ECC parity bits = `24`
- ECC code rate (R) 40/64 = `40/64` (`0.625`)

6.2 Information and Parity Bit Positions

Each codeword contains a 24-bit data payload, 16 CRC bits, and 24 ECC parity bits. Each team needs to design the positions of the information bits (`INFO_POS[0..39]`) and the parity bits (`FROZEN_POS[0..23]`). Specifically, the information bits and parity bits need not be contiguous. The objective is to maximize the minimum distance between any two polar ECC codewords. In this project, a minimum distance (`dmin`) of 8 allows correction of up to 3-bit flips and detection of 4-bit flips without using CRC check.

6.3 Polar transform definition (no bit-reversal)

Let `u[63:0]` denote the polar encoder input vector, where information bits (data and CRC) are placed at `INFO_POS` and frozen bits at `FROZEN_POS` are set to `0`. Encoding produces `codeword[63:0]` from `u[63:0]` using the standard polar butterfly transform, with no bit-reversal permutation.

```
v = u
for s = 0..5:
   step = 2^(s+1)
   half = 2^s
   for i = 0..63 in increments of step:
      for j = 0..half-1:
         v[i+j] = v[i+j] XOR v[i+j+half]
codeword = v
```

7. CRC Specification

We use CRC-16-CCITT generator polynomial:

- `G(x) = x^16 + x^12 + x^5 + 1`
- Binary (including `x^16` term): `0x11021`
- Feedback XOR constant (without `x^16` term): `0x1021`
- Note: `0x11021` is a 17-bit representation; in Verilog write `17'h11021`. Do not write `16'h11021` in Verilog. The feedback XOR constant is `16'h1021`.

CRC conventions:

- Initial remainder: `16'h0000`
- Bit processing order: MSB-first (`data_in[23] down to data_in[0]`)
- No reflection
- No final XOR (`xorout = 0`)

Reference bit-serial algorithm:

```
crc = 0
for i = 23 downto 0:
      feedback = data_in[i] XOR crc[15]
crc = (crc << 1) & 0xFFFF
if feedback == 1:
      crc = crc XOR 0x1021
```

The encoder places `crc[15:0]` into `u[INFO_POS[24..39]]`. The decoder extracts CRC bits from `u_hat[INFO_POS[24..39]]`, recomputes CRC from `data_out[23:0]`, and compares.

8. Functional Requirements

8.1 Encoder behavior

The encoder implementation:

- Must set `u[FROZEN_POS[23:0]] = 0`.
- must generate `codeword[63:0]` from `data_in[23:0]` using the mapping and transform above.
- must compute CRC-16-CCITT as specified and embed it in the encoded block.

8.2 Decoder behavior

The decoder implementation:

- attempts to decode:
  - `rx[63:0]` into `u_hat[63:0]`.
- enforces frozen bits:
  - `for k=0..23:  u_hat[FROZEN_POS[k]] = 0`
- extracts data bits using `INFO_POS`:
  - `for k=0..23:  data_out[23-k] = u_hat[INFO_POS[k]]`
- extracts CRC bits using `INFO_POS`:
  - `for k=0..15:  crc_rx[15-k] = u_hat[INFO_POS[24+k]]`
- recompute CRC on `data_out` and compare to `crc_rx`

Polar-only correction/detection rule

- The decoder must implement bounded-distance decoding with radius 3:

- If there exists a unique polar-valid codeword within Hamming distance ≤ 3 of `rx`, output `valid=1` and the corresponding `data_out`.
- Otherwise output `valid=0`.

Guarantee from code design

With `dmin = 8`, this guarantees:

- correction of all 1–3 bit flips, and
- detection (reject) of any 4-bit flip pattern (no mis-correction to a wrong codeword).

CRC is an additional integrity check; if CRC fails, `valid` must be 0.

8.3 Fail-safe rule (grading)

- If `valid==1`, then `data_out` must match the originally transmitted data.
- If uncertain, prefer `valid==0`.

9. Module Interfaces

9.1 Encoder module

Module name: polar64_crc16_encoder

- Inputs: `clk`, `rst_n`, `start` (1-cycle pulse), `data_in[23:0]`
- Outputs: `done` (1-cycle pulse), `codeword[63:0]`
- Timing: `done` must assert exactly 2 cycles after start is sampled.

9.2 Decoder module

Module name: polar64_crc16_decoder

- Inputs: `clk`, `rst_n`, `start` (1-cycle pulse), `rx[63:0]`
- Outputs: `done` (1-cycle pulse), `data_out[23:0]`, `valid`
- Timing: `done` must assert within 12 cycles after start.

10. Verification Expectations

The student-visible testbench should include, at minimum:

- Reset sequence
- Encode a known value (example: `data_in = 24'hABCDEF`)
- Channel model cases:

- Case A (0-bit flips): decoder outputs `valid=1` and correct `data_out`.
- Case B (1-bit to 3-bit flips): decoder must correct → `valid=1` and correct `data_out`.
- Case C (4-bit flips): decoder must detect and reject → `valid=0`.

Recommended Vivado xsim compile/run (SystemVerilog enabled)

```
xvlog -sv polar_common_pkg.sv polar64_crc16_encoder.sv \
polar64_crc16_decoder.sv tb_basic.sv
xelab tb_basic -debug typical -s sim_snapshot
xsim sim_snapshot -runall
```

11. Deliverables

Submit a single zip:

```
team_<XX>_submission.zip
├── polar64_crc16_encoder.sv
├── polar64_crc16_decoder.sv
├── crc.sv                # optional if integrated elsewhere
├── tb_basic.sv           # submit if you have extended it
├── README.md
├── report.pdf            # max 2 pages
└── ai_log.txt            # required: prompts + brief notes on what was used
```

README.md must include:

- exact Vivado xsim commands to compile and run
- module descriptions and latency behavior
- division of labor (who did what)

report.pdf (2 pages max) should include:

- architecture summary (encoder/decoder)
- CRC integration description
- verification strategy and key tests
- one short paragraph reflecting on robustness/fail-safe behavior

12. The Challenge

The first three teams that can complete the ECC modules correctly will receive an award and extra credits. Make sure you have fixed all the bugs before submission. Each team has only two chances to submit its design. Good luck!

13. Assessment Rubric (100 points + Bonus)

- Correct CRC implementation and integration: 15
- Correct Polar encoder (matches spec): 20
- Decoder correctness on clean channel (0 flips): 15
- Fail-safe behavior (avoid wrong valid outputs): 20
- Meets interface/timing requirements: 10
- Testbench quality + reproducibility in Vivado xsim: 10
- Code quality + documentation: 10

Bonus:

- Decoder `done` ≤ 8 cycles: +5
- Pipelined / higher-throughput architecture +5
- Clear synthesis-oriented RTL style: +5