EE3220 Digital Systems Design

Polar + CRC-16 Error Correction System

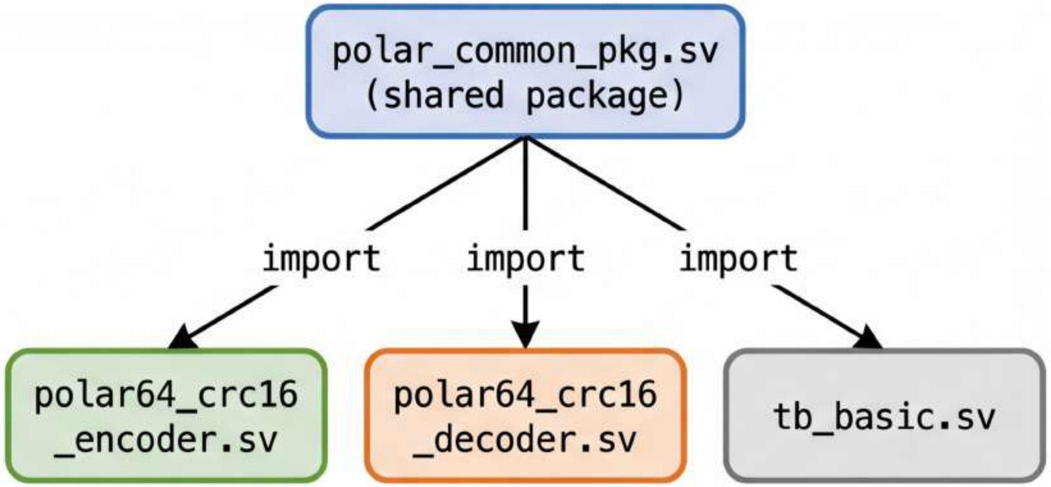
For Space-Z Mars Robot Communication

Team 4

2026

File Structure & Shared Package

Code Structure: polar_common_pkg.sv



Parameters

```
N = 64    // Codeword length
K = 40    // Info bits (24 data + 16 CRC)
F = 24    // Frozen bits
```

Bit Position Tables

0	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	24	25	36	38	38	40	41	44	48	49	50	52	52	56

INFO_POS

0	1		15		23	27	29	30	31	32	33	34	35	36	37	38	39	40	41	42	45	45	46	47	51	53	54	55	57	58	59	60	61	62	63

Blue = Information bits | Gray = Frozen bits (always 0)

Selection criteria: popcount(i) ≤ 3, ranked by Bhattacharyya parameter

Shared Functions

Function	Input → Output	Purpose
`crc16_ccitt24()`	24-bit data → 16-bit CRC	CRC-16-CCITT checksum
`build_u()`	24-bit data + 16-bit CRC → 64-bit u	Map bits to info positions
`polar_transform64()`	64-bit u → 64-bit v	Butterfly transform (self-inverse)

Encoder Workflow — Step 1: CRC-16 Computation

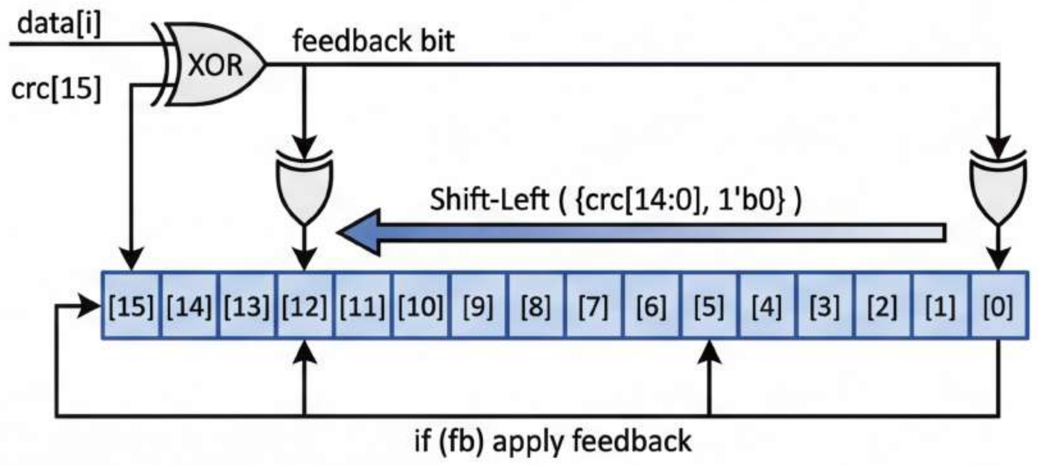
Encoding Step 1: CRC-16-CCITT Computation

```
function automatic logic [15:0]
  crc16_ccitt24(input logic [23:0] data);
  logic [15:0] crc;
  logic        fb;

  crc = 16'h0000;                    // init = 0
  for (int i = 23; i >= 0; i--) begin
    fb   = data[i] ^ crc[15];       // feedback
    crc  = {crc[14:0], 1'b0};       // shift left
    if (fb) crc = crc ^ 16'h1021; // XOR poly
  end
  return crc;
endfunction
```

Polynomial:
$x^{\wedge 16} + x^{\wedge 12} + x^5 + 1$

CRC Computation Flow Diagram

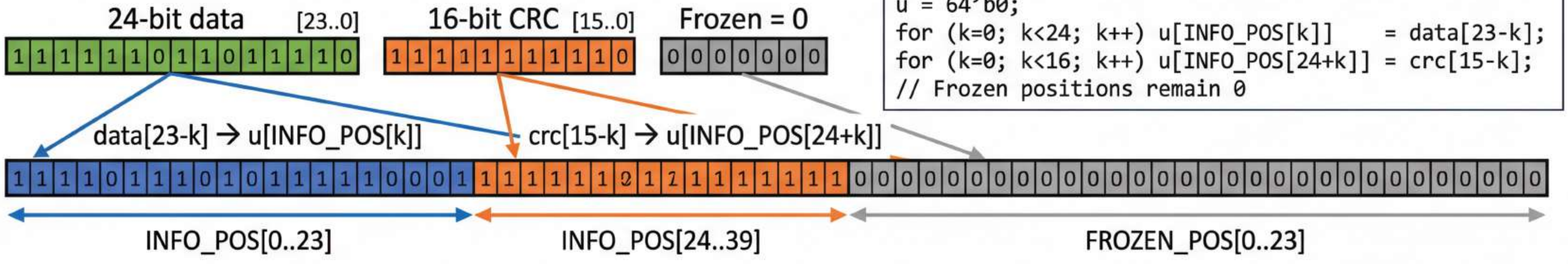


Input: data_in = 24’hABCDEF
Output: CRC-16 = crc16_ccitt24(24’hABCDEF)
Process: MSB-first, 24 iterations, no reflect, no xorout

Key Specs:
- **Polynomial:** $G(x) = x^{\wedge 16} + x^{\wedge 12} + x^{\wedge 5} + 1$
- **Init:** 0x0000
- **Processing:** MSB first (bit 23 → bit 0)
- **XOR constant:** 0x1021

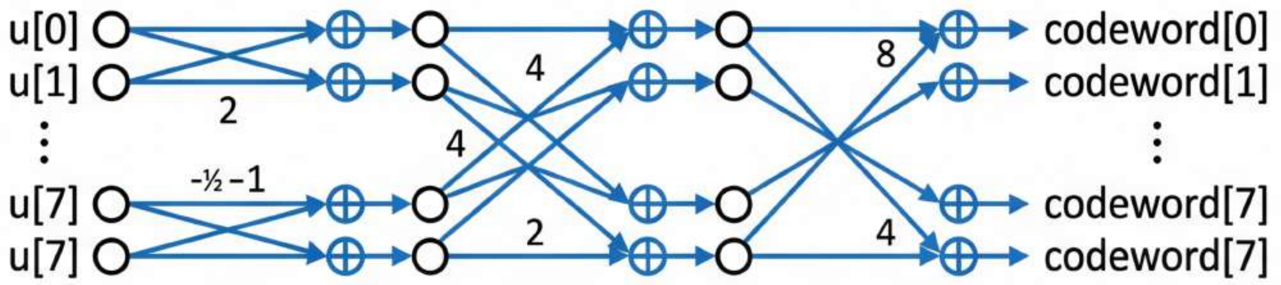
Encoding Steps 2-3: Build u-vector & Polar Transform

Build u-vector



```
u = 64’b0;
for (k=0; k<24; k++) u[INFO_POS[k]]     = data[23-k];
for (k=0; k<16; k++) u[INFO_POS[24+k]] = crc[15-k];
// Frozen positions remain 0
```

Polar Butterfly Transform



Illustrated for N=8; actual N=64 uses 6 stages

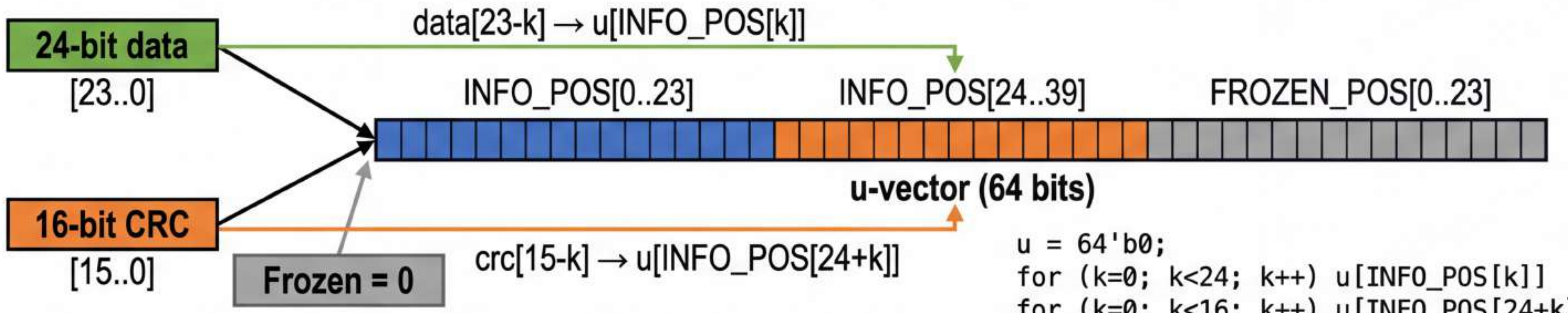
```
for s = 0..5:
    step = 2^(s+1), half = 2^s
    v[i+j+half] ^= v[i+j]
```

Self-inverse: same transform is used for decoding

$$\begin{tikzcd}
{} \arrow[r, "\text{data\_reg}", "\text{24-bit}"'] & \boxed{\text{[crc16\_ccitt24]}} \arrow[r, "\text{crc\_comb}", "\text{16-bit}"'] & \boxed{\text{[build\_u]}} \arrow[r, "\text{u\_comb}", "\text{64-bit}"'] & \boxed{\text{[polar\_transform64]}} \arrow[r, "\text{cw\_comb}", "\text{64-bit}"'] & \text{codeword}
\end{tikzcd}$$

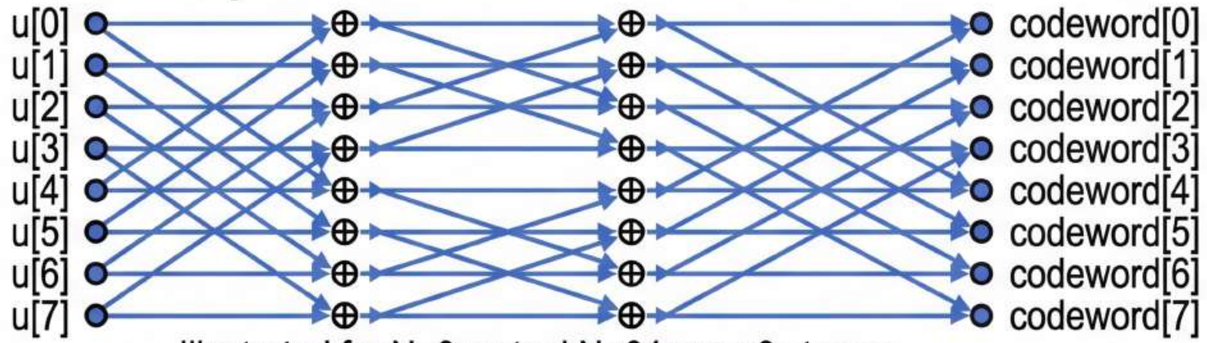
All computed in always_comb — pure combinational, registered at pipeline boundaries.

Encoding Steps 2-3: Build u-vector & Polar Transform



```
u = 64'b0;
for (k=0; k<24; k++) u[INFO_POS[k]]    = data[23-k];
for (k=0; k<16; k++) u[INFO_POS[24+k]] = crc[15-k];
// Frozen positions remain 0
```

St. 2: Polar Butterfly Transform

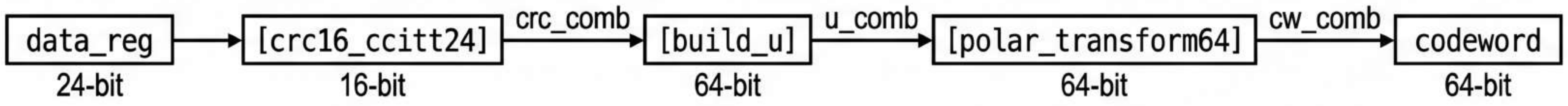


Illustrated for N=8; actual N=64 uses 6 stages

```
for s = 0..5:
  step = 2^(s+1), half = 2^s
  v[i+j+half] ^= v[i+j]
```

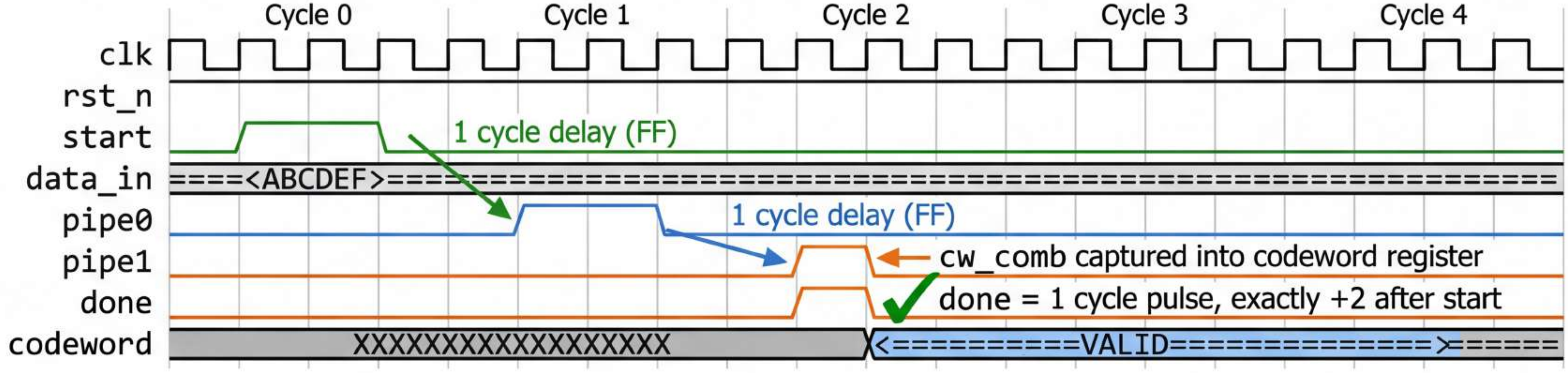
Self-inverse: same transform is used for decoding

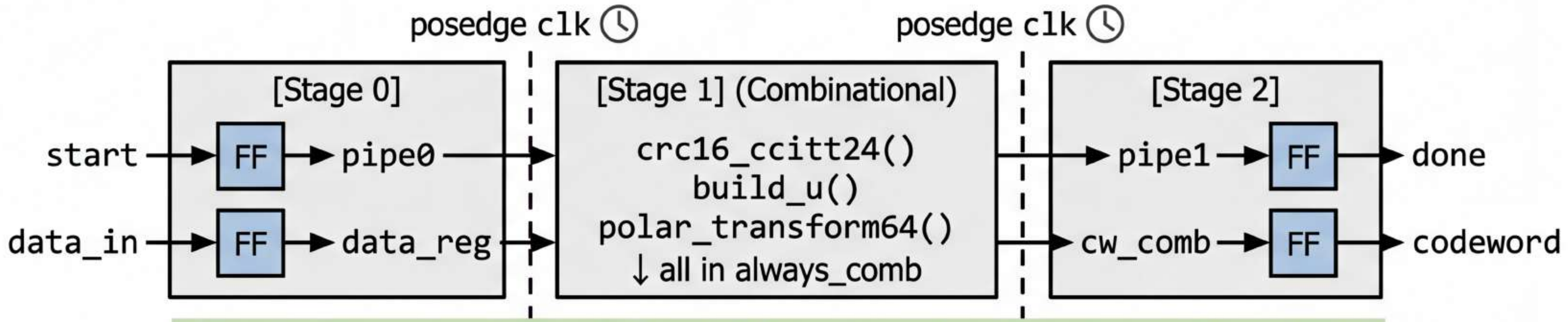
St. 3: Encoder combinational logic summary



All computed in `always_comb` — pure combinational, registered at pipeline boundaries.

Encoder Pipeline: `polar64_crc16_encoder.sv`

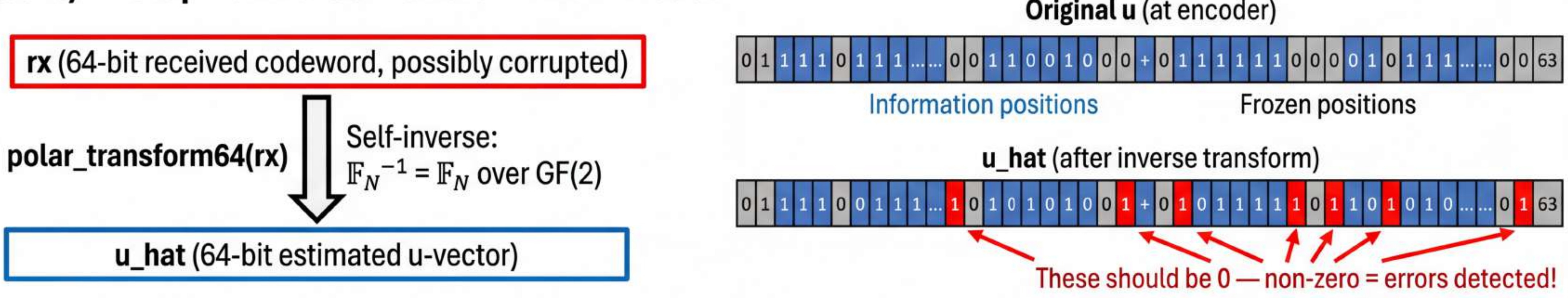




All encoding computation (CRC + **build**_u + **transform**) happens combinationally **in Stage 1.** Pipeline registers only add 2 cycles of latency for timing closure.

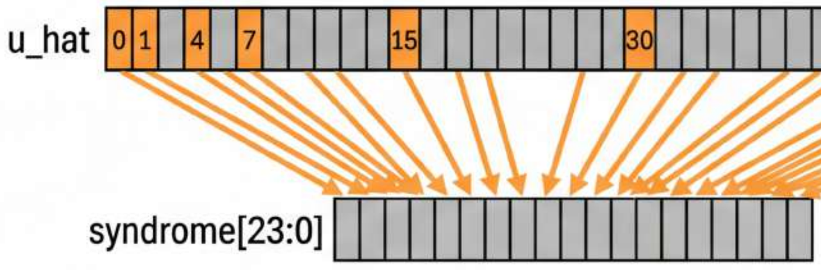
Decoding Steps 1-2: Inverse Polar Transform & Syndrome Extraction

(55%) — Step 1: Inverse Transform with visual


rx (64-bit received codeword, possibly corrupted)
polar_transform64(rx)
Self-inverse:
$\mathbb{F}_N^{-1} = \mathbb{F}_N$ over GF(2)
u_hat (64-bit estimated u-vector)
Original u (at encoder)
Information positions
Frozen positions
u_hat (after inverse transform)
63
These should be 0 — non-zero = errors detected!

(45%) — Step 2: Syndrome Extraction with code and visual

```systemverilog
// Step 2: extract 24-bit syndrome
// from frozen bit positions
for (int k = 0; k < 24; k++)
  syndrome[k] = u_hat[FROZEN_POS[k]];
```

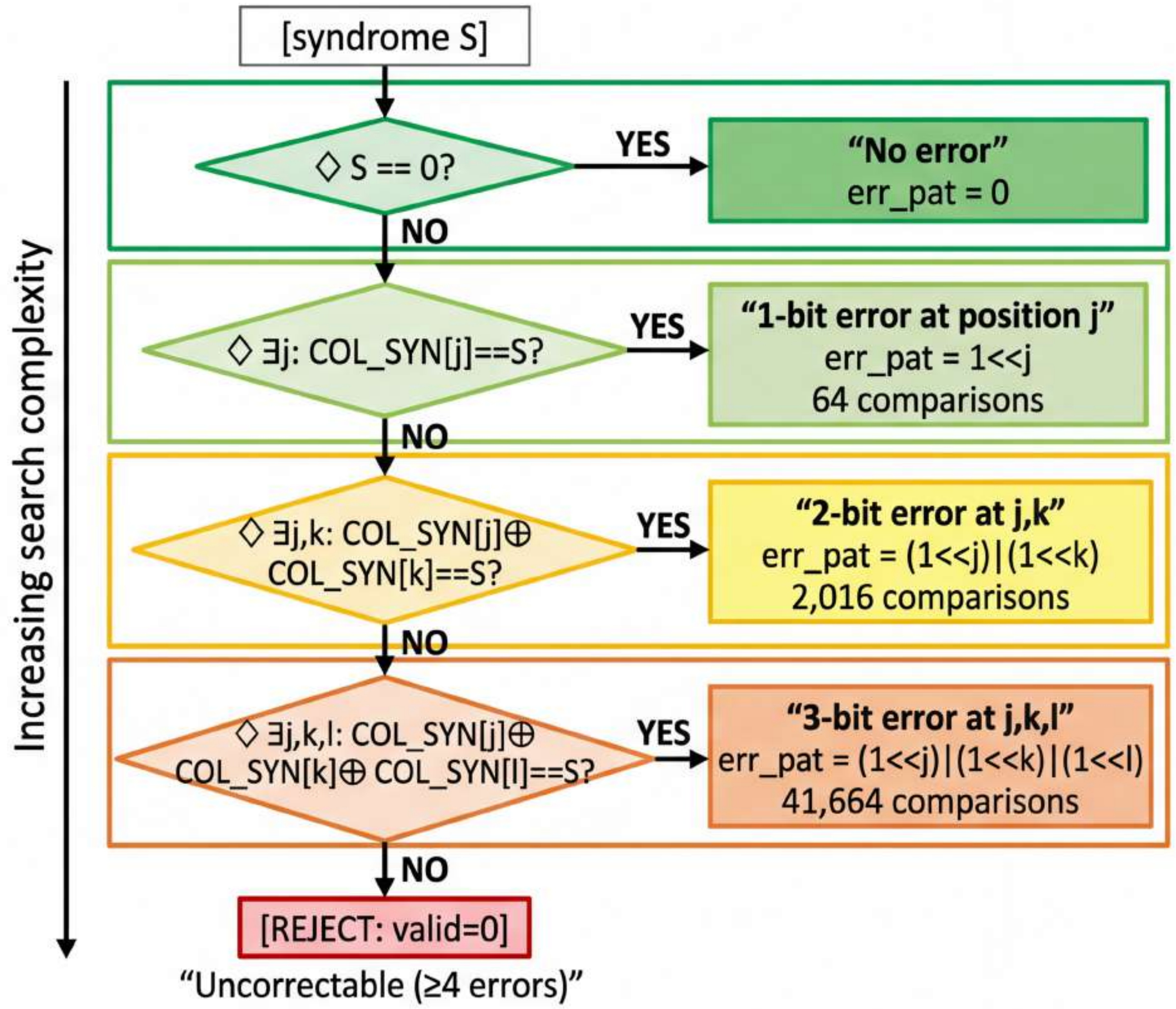


If syndrome = 24’h000000
→ No errors!

If syndrome ≠ 0
→ Errors detected
→ proceed to correction

syndrome index k	Source: u_hat position	FROZEN_POS value
0	u_hat[0]	0
1	u_hat[1]	1
2	u_hat[15]	15
...	...	...
23	u_hat[63]	63

Decoding Step 3: Syndrome Matching & Error Correction (t=3)



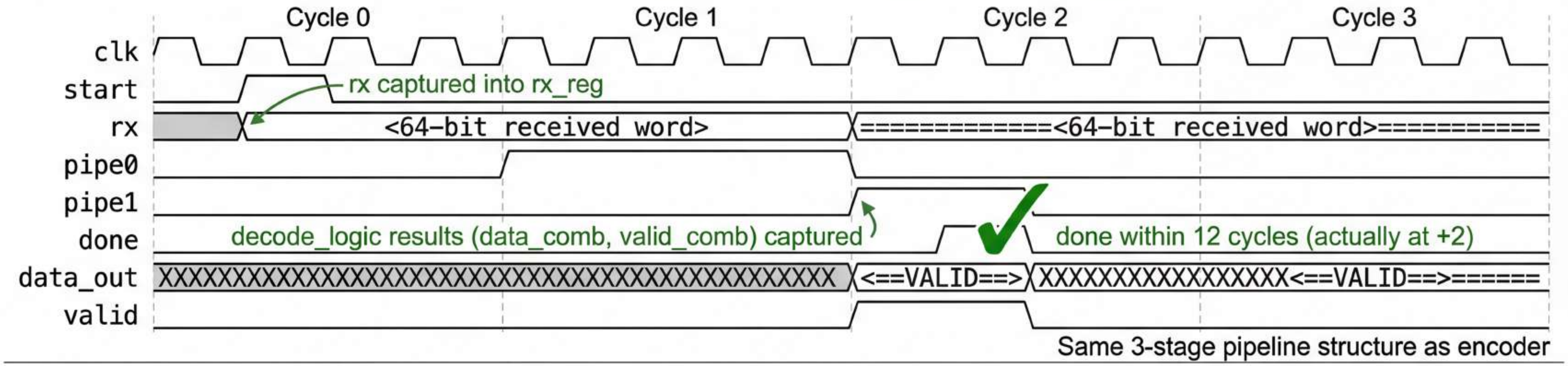
Column Syndrome Lookup Table (COL_SYN[0..63])

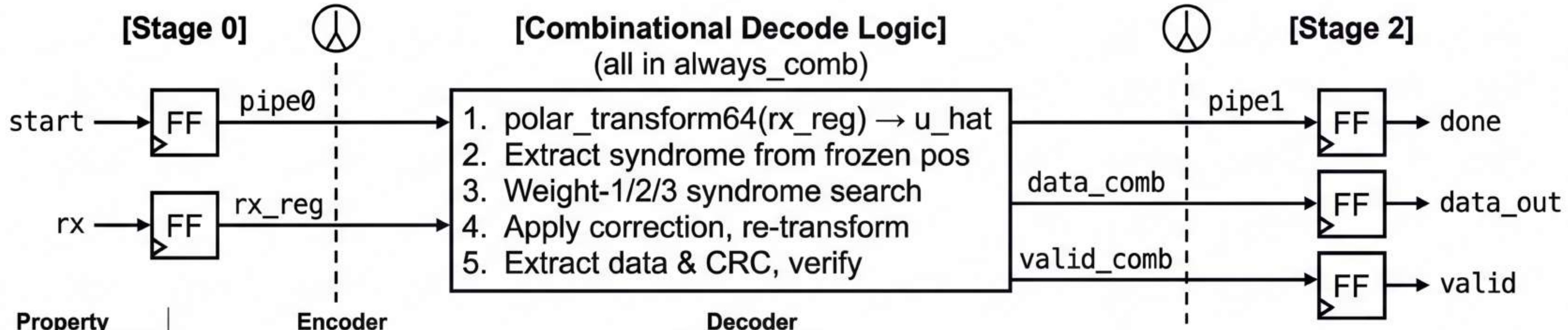
Bit Position *j*	COL_SYN[j] (24-bit)
|	`FFFFFFh`
0	`FFFFFFh`
1	`AB77BEh`
2	`CDBBDCh`
3	`89339Ch`
...	...
32	`FFFF00h`
...	...
63	`800000h`

All 64 entries are UNIQUE → guarantees unique decoding for weight ≤ 3

COL_SYN[j] = projection of **polar_transform64**(unit vector e_j) onto frozen-bit positions. Each single-bit error produces a unique 24-bit signature. Multi-bit errors: XOR the individual signatures and match.

Decoder Pipeline: polar64_crc16_decoder.sv


Cycle 0
Cycle 1
Cycle 2
Cycle 3
clk
start
rx
pipe0
pipe1
done
data_out
valid
rx captured into rx_reg
<64-bit received word>
=============<64-bit received word>===========
decode_logic results (data_comb, valid_comb) captured
done within 12 cycles (actually at +2)
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
<==VALID==>
XXXXXXXXXXXXXXXXX<==VALID==>======
Same 3-stage pipeline structure as encoder


[Stage 0]
[Combinational Decode Logic]
(all in always_comb)
[Stage 2]
start
FF
pipe0
rx
FF
rx_reg
1. polar_transform64(rx_reg) → u_hat
2. Extract syndrome from frozen pos
3. Weight-1/2/3 syndrome search
4. Apply correction, re-transform
5. Extract data & CRC, verify
pipe1
data_comb
valid_comb
FF
FF
FF
done
data_out
valid

Property	Encoder	Decoder
Pipeline stages	3 (start→pipe0→pipe1→done)	3 (start→pipe0→pipe1→done)
Done latency	Exactly 2 cycles	Exactly 2 cycles (≤12 allowed)
Comb logic	CRC + build_u + transform	Transform + syndrome search + CRC verify
Comb complexity	O(N)	O(N^3) — weight-3 search

Verification Results & Summary

Test results (tb_basic.sv)

Test Case	Error Injection	Expected	Result
Case A: 0 flips	None	valid=1, data=ABCDEF	✅ PASS
Case B-1: 1 flip (bit 5)	1 bit	valid=1, data=ABCDEF	✅ PASS
Case B-2: 2 flips (bit 0,63)	2 bits	valid=1, data=ABCDEF	✅ PASS
Case B-3: 3 flips (bit 0,1,63)	3 bits	valid=1, data=ABCDEF	✅ PASS
Case C: 4 flips (bit 0,1,2,3)	4 bits	valid=0 (reject)	✅ PASS
Fail-safe: 5 flips	5 bits	valid=0 or correct	✅ PASS

SMOKE SCORE: 30/30

tb_basic is a smoke test; full grading uses tb_hidden with randomized regression.

System Summary

Code Rate: 0.625

d_{min} = 8

Correction: ≤3 bits

Detection: 4 bits

- ✅ Shared parameter package ensures encoder/decoder consistency
- ✅ CRC-16 provides secondary error detection
- ✅ Column syndrome table enables unique error identification
- ✅ Safety-first: **valid=0** when uncertain
- ✅ 2-cycle pipelined encoder & decoder

Design Rule: Never output incorrect data with valid=1. Reject when in doubt.