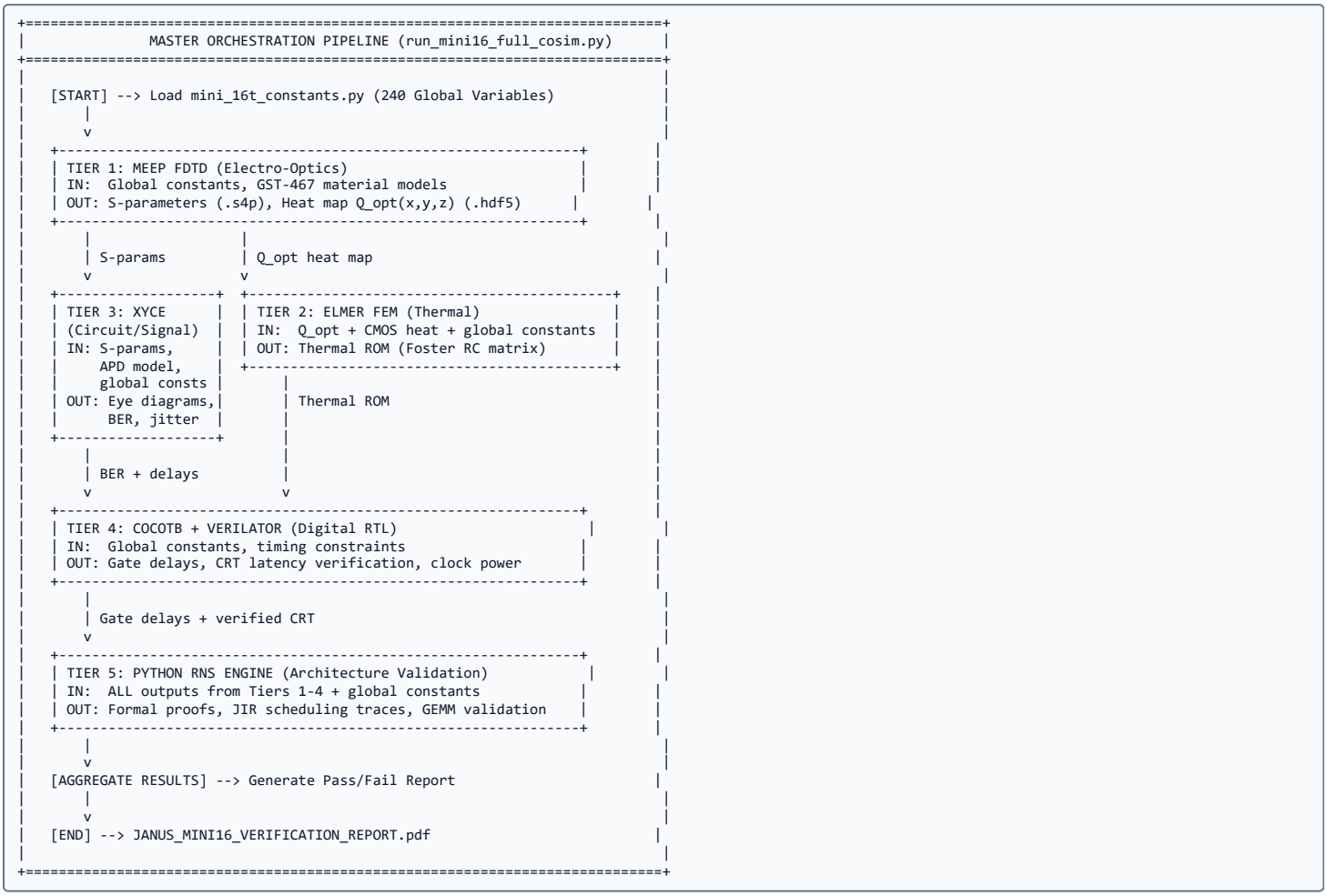
PROJECT JANUS MINI (16-TILE): ALGORITHMS & FLOWCHARTS SPECIFICATION

Document ID: JANUS-ALG-SPEC-MINI16-2026-V1
Companion To: JANUS-SIM-SPEC-MINI16-2026-V1 (Co-Simulation Specification)
Target Hardware: JANUS Mini 16-Tile Monolithic Planar MVP (Model 1A)
Classification: Algorithm Design Document / Verification Flowcharts
Lead Architect: Deepanshu Bhardwaj
Status: Approved for Implementation

1. Master Orchestration Pipeline

1.1 Top-Level Co-Simulation Flow

The master orchestrator (`run_mini16_full_cosim.py`) executes all five tiers in dependency order, propagating inter-tier data products through HDF5 files and Touchstone S-parameter archives.



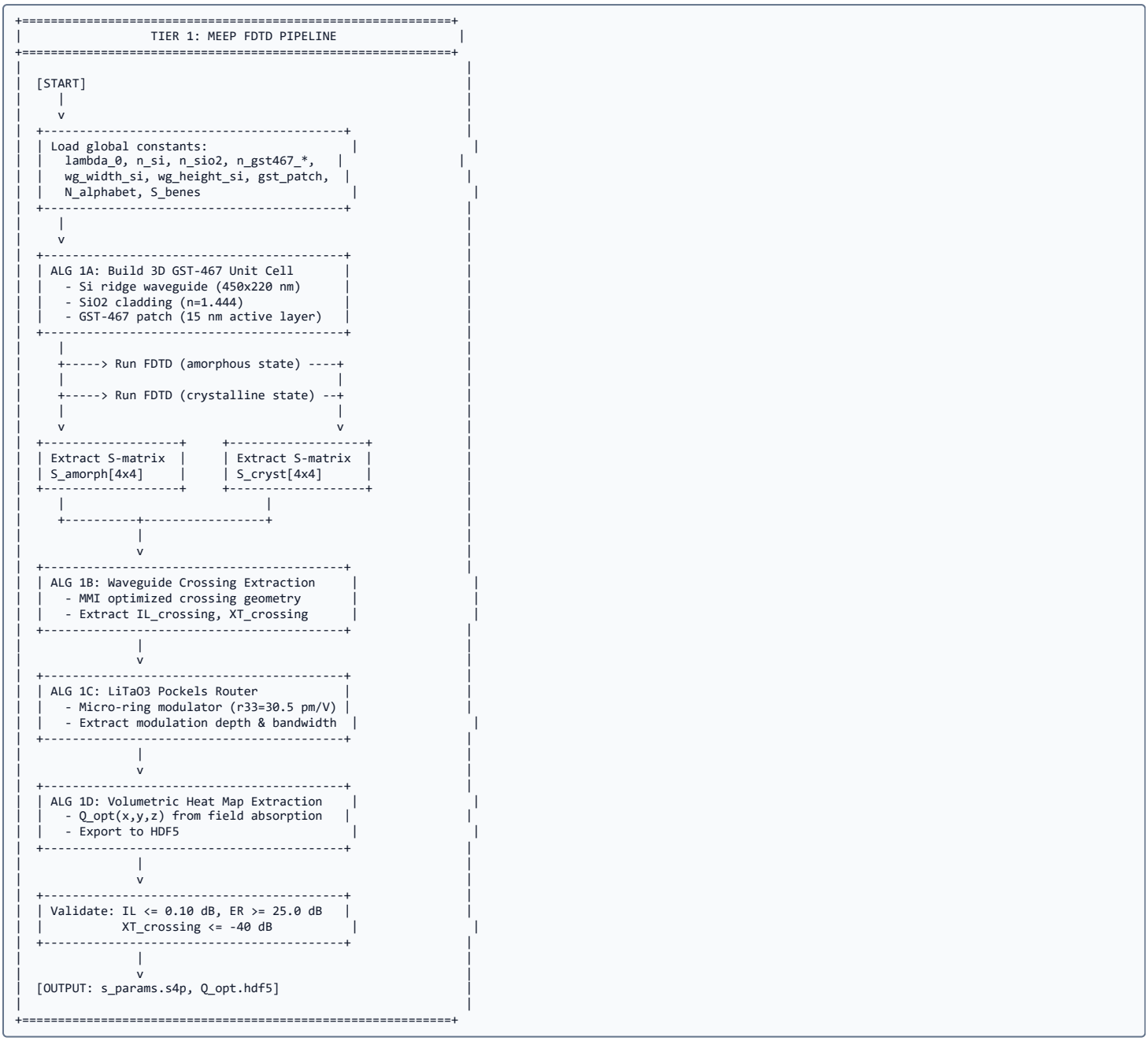
1.2 Algorithm 0: Master Orchestrator

```
ALGORITHM 0: MASTER_ORCHESTRATOR
============================================================
INPUT:  config = load("mini_16t_constants.py")   // 240 globals
OUTPUT: verification_report (Pass/Fail per metric)

BEGIN
  1.  constants <-- load_global_constants(config)
  2.  validate_constants(constants)               // Sanity checks

  // TIER 1: Electro-Optics
  3.  s_params   <-- run_tier1_meep(constants)    // Touchstone .s4p
  4.  Q_opt_map  <-- run_tier1_meep_heatmap(constants)  // HDF5

  // TIER 2 and TIER 3 run in PARALLEL (independent inputs)
  5.  PARALLEL:
      5a. thermal_rom <-- run_tier2_elmer(constants, Q_opt_map)
      5b. ber_results <-- run_tier3_xyce(constants, s_params)

  // TIER 4: Digital RTL
  6.  rtl_results <-- run_tier4_cocotb(constants)

  // TIER 5: Full Architecture Validation
  7.  rns_results <-- run_tier5_python_rns(
          constants, s_params, thermal_rom,
          ber_results, rtl_results)

  // AGGREGATE
  8.  report <-- aggregate_results(
          s_params, thermal_rom, ber_results,
          rtl_results, rns_results)
  9.  FOR EACH metric IN verification_criteria:
        IF metric.value satisfies metric.threshold:
          report[metric] = "PASS"
        ELSE:
          report[metric] = "FAIL"
  10. export_report(report, "JANUS_MINI16_VERIFICATION_REPORT.pdf")
  11. RETURN report
END
```

---

2. Tier 1 Algorithms: Electro-Optics & FDTD (3D MEEP)

2.1 Flowchart: Tier 1 Pipeline



2.2 Algorithm 1A: GST-467 Phase-Change Switch FDTD Simulation

```
ALGORITHM 1A: GST467_SWITCH_FDTD
============================================================
INPUT:  lambda_0      = 1064 nm
        n_si          = 3.565
        n_sio2        = 1.444
        n_gst_a       = 3.45 + i*0.008    // amorphous
        n_gst_c       = 4.20 + i*0.18     // crystalline
        wg_w          = 450 nm
        wg_h          = 220 nm
        t_gst         = 15 nm
        A_cell        = 1.25 um^2
OUTPUT: S_amorph[4x4], S_cryst[4x4], Q_opt(x,y,z)

BEGIN
  // STEP 1: Define computational domain
  1.  resolution  <-- lambda_0 / (20 * max(n_gst_c.real))
      // Minimum 20 points per wavelength in highest-index medium
  2.  domain_size <-- (5*wg_w, 5*wg_w, 3*wg_h + t_gst + 2*lambda_0)
  3.  PML_layers  <-- 1.0 * lambda_0 on all 6 boundaries

  // STEP 2: Build geometry
  4.  DEFINE substrate  = Block(material=SiO2, n=n_sio2)
  5.  DEFINE waveguide_input  = Block(w=wg_w, h=wg_h, material=Si, n=n_si)
  6.  DEFINE waveguide_cross  = Block(w=wg_w, h=wg_h, material=Si, n=n_si)
  7.  DEFINE gst_patch  = Block(area=A_cell, h=t_gst, material=GST467)

  // STEP 3: Run amorphous state
  8.  SET gst_patch.epsilon = eps_from_nk(n_gst_a)
  9.  source <-- GaussianSource(frequency=c_vacuum/lambda_0,
                                fwidth=0.1*c_vacuum/lambda_0)
  10. ports  <-- [Port1_in, Port2_thru, Port3_drop, Port4_iso]
  11. FOR EACH port_excitation IN ports:
        11a. Run FDTD until fields decay to 1e-8 of peak
        11b. Record transmitted/reflected fields at all ports
  12. S_amorph <-- compute_s_matrix(port_fields)

  // STEP 4: Run crystalline state
  13. SET gst_patch.epsilon = eps_from_nk(n_gst_c)
  14. REPEAT steps 11-12 --> S_cryst

  // STEP 5: Extract heat map
  15. Q_opt(x,y,z) <-- (1/2) * omega_optical * epsilon_0
                       * Im[eps_r(x,y,z)] * |E(x,y,z)|^2
  16. Export Q_opt to HDF5 on Yee grid

  // STEP 6: Compute figures of merit
  17. IL_amorph   <-- -10*log10(|S21_amorph|^2)    // Insertion loss
  18. ER          <-- |S21_amorph_dB - S21_cryst_dB|  // Extinction ratio
  19. IL_cryst    <-- -10*log10(|S21_cryst|^2)

  // STEP 7: Validate
  20. ASSERT IL_amorph <= 0.10 dB   // "PASS: Switch IL"
  21. ASSERT ER >= 25.0 dB          // "PASS: Extinction Ratio"

  22. RETURN S_amorph, S_cryst, Q_opt
END
```

2.3 Algorithm 1B: MMI Waveguide Crossing Extraction

```
ALGORITHM 1B: WAVEGUIDE_CROSSING_FDTD
================================================================
INPUT:  lambda_0, n_si, n_sio2, wg_w, wg_h
OUTPUT: IL_crossing, XT_crossing, S_crossing[4x4]

BEGIN
  1.  Build 3D MMI crossing geometry:
      - Two perpendicular Si waveguides intersecting at center
      - Optimized MMI taper widths for minimum mode disruption

  2.  Run 4-port FDTD simulation:
      FOR EACH input port p IN {1, 2, 3, 4}:
        2a. Excite fundamental TE mode at port p
        2b. Record transmitted field at all 4 ports
        2c. Compute S-parameters row for port p

  3.  S_crossing <-- assemble 4x4 S-matrix

  4.  IL_crossing <-- -10*log10(|S21|^2)   // Through-port loss
  5.  XT_crossing <-- 10*log10(|S31|^2)    // Cross-port crosstalk

  6.  ASSERT IL_crossing <= 0.02 dB
  7.  ASSERT XT_crossing <= -40 dB

  8.  RETURN IL_crossing, XT_crossing, S_crossing
END
```

2.4 Algorithm 1C: LiTaO3 Pockels Micro-Ring Router

```
ALGORITHM 1C: LITAO3_POCKELS_ROUTER
================================================================
INPUT:  lambda_0, n_litao3=2.13, r33_litao3=30.5 pm/V,
        E_pockels_switch=50 aJ
OUTPUT: V_pi, modulation_bandwidth, S_router[2x2]

BEGIN
  1.  Build 3D micro-ring resonator geometry:
      - LiTaO3 thin-film ring coupled to bus waveguide
      - Electrode geometry for vertical E-field application

  2.  Compute half-wave voltage:
      V_pi = lambda_0 / (2 * n_litao3^3 * r33_litao3 * L_electrode / gap)

  3.  Run FDTD with applied DC bias sweep (0 to V_pi):
      FOR EACH V_bias IN linspace(0, V_pi, 20):
        3a. Apply refractive index change:
            delta_n = -0.5 * n_litao3^3 * r33_litao3 * (V_bias / gap)
        3b. Run FDTD, extract S21(V_bias) and S31(V_bias)

  4.  S_router <-- S-matrix at resonance and off-resonance

  5.  Verify E_pockels_switch <= 50 aJ per switch event

  6.  RETURN V_pi, modulation_bandwidth, S_router
END
```

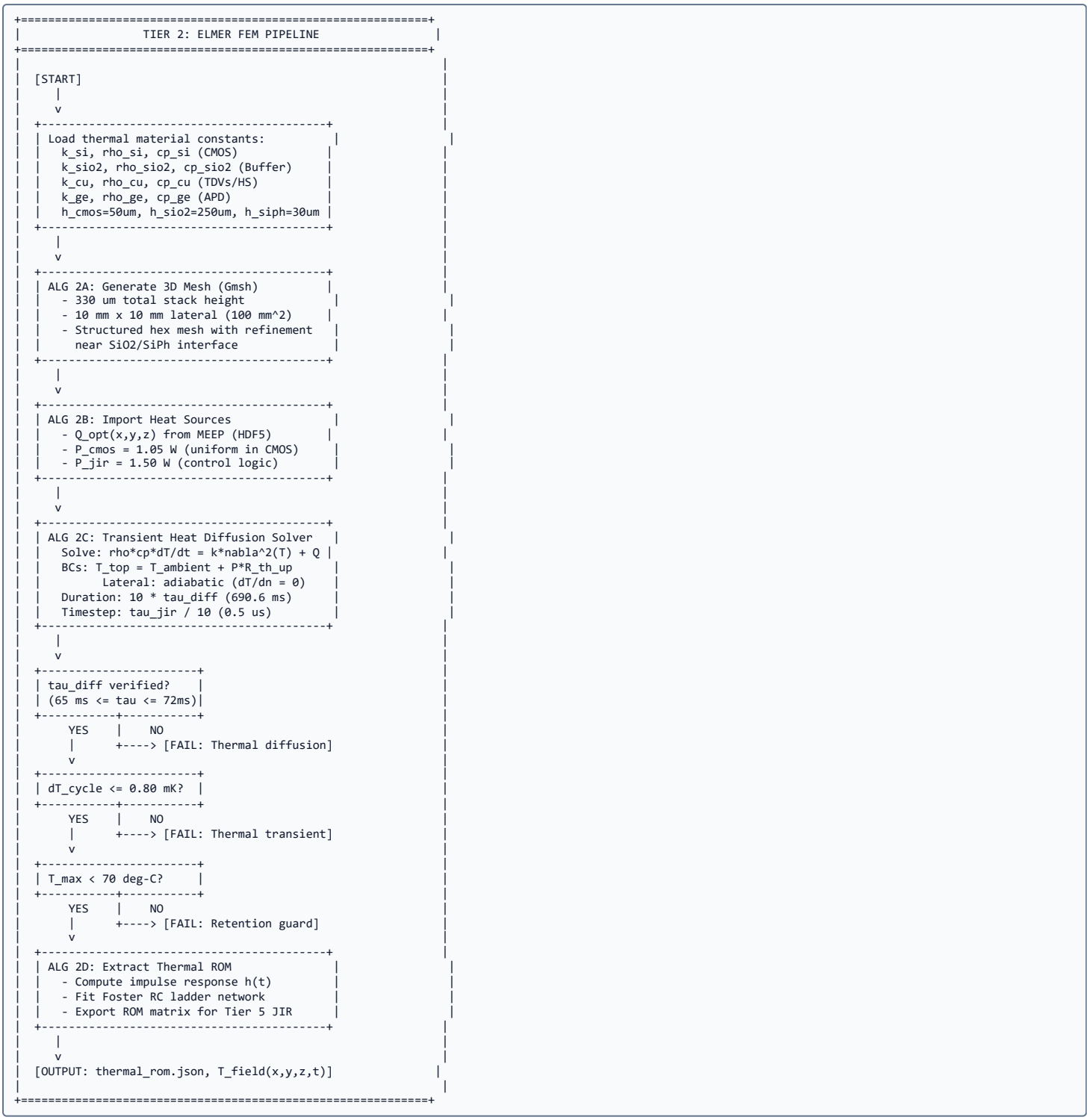
2.5 Algorithm 1D: Volumetric Heat Map Extraction

```
ALGORITHM 1D: HEAT_MAP_EXTRACTION
================================================================
INPUT:  FDTD field solution E(x,y,z), material map eps_r(x,y,z)
OUTPUT: Q_opt(x,y,z) as HDF5 dataset

BEGIN
  1.  FOR EACH Yee cell (i, j, k) in simulation domain:
        1a. eps_imag = Im[eps_r(i,j,k)]
        1b. E_mag_sq = |Ex(i,j,k)|^2 + |Ey(i,j,k)|^2 + |Ez(i,j,k)|^2
        1c. Q_opt(i,j,k) = 0.5 * omega_optical * epsilon_0
                         * eps_imag * E_mag_sq

  2.  Normalize Q_opt to total absorbed power:
      P_absorbed = integral(Q_opt) over domain
      // Cross-check: P_absorbed should equal (1 - sum(|S_out|^2)) * P_in

  3.  Export Q_opt(x,y,z) with grid coordinates to HDF5:
      - Dataset: "Q_opt" with shape (Nx, Ny, Nz)
      - Attributes: dx, dy, dz, lambda_0, material_state

  4.  RETURN Q_opt
END
```

---

3. Tier 2 Algorithms: Thermal Stack Analysis (Elmer FEM)

3.1 Flowchart: Tier 2 Pipeline



3.2 Algorithm 2A: 3D Thermal Mesh Generation

```
ALGORITHM 2A: THERMAL_MESH_GENERATION
============================================================
INPUT:  A_die=100 mm^2, L_die=10 mm, N_tiles=16
        h_cmos=50 um, h_sio2=250 um, h_siph=30 um
OUTPUT: mesh (Gmsh .msh format)

BEGIN
  1.  DEFINE layers (bottom to top):
      Layer 0: CMOS substrate    z=[0, 50 um]        material=Si
      Layer 1: SiO2 buffer       z=[50, 300 um]       material=SiO2
      Layer 2: SiPh stratum      z=[300, 330 um]      material=Si+SiO2

  2.  DEFINE lateral domain:
      x = [0, L_die],  y = [0, L_die]

  3.  SET mesh parameters:
      - Lateral element size: 100 um (coarse bulk), 25 um (near tiles)
      - Vertical element size in SiO2: 10 um (25 layers across 250 um)
      - Vertical element size at SiO2/SiPh interface: 2 um (refined)
      - Vertical element size in CMOS: 10 um

  4.  DEFINE 16 tile sub-domains:
      FOR i = 0 TO 3:
        FOR j = 0 TO 3:
          tile[i][j].center = (L_die/8 + i*L_die/4,
                               L_die/8 + j*L_die/4)
          tile[i][j].area = A_tile = 6.25 mm^2

  5.  GENERATE structured hexahedral mesh using Gmsh
  6.  TAG physical volumes: CMOS, SiO2_buffer, SiPh_stratum
  7.  TAG tile boundaries for per-tile temperature monitoring

  8.  RETURN mesh
END
```

3.3 Algorithm 2C: Transient 3D Heat Diffusion Solver

```
ALGORITHM 2C: TRANSIENT_HEAT_DIFFUSION
============================================================
INPUT:  mesh, material_properties, Q_opt(x,y,z), P_cmos, P_jir
        T_ambient=298.15 K, R_th_up=0.552 K/W, R_th_down=0.195 K/W
        tau_diff=69.06 ms, tau_jir=5 us
OUTPUT: T(x,y,z,t), tau_diff_measured, dT_cycle, T_max

BEGIN
  // GOVERNING EQUATION:
  // rho(x,y,z) * cp(x,y,z) * dT/dt = div(k(x,y,z) * grad(T)) + Q(x,y,z)

  1.  SET initial condition:  T(x,y,z,0) = T_ambient  (uniform)

  2.  SET boundary conditions:
      - Top surface (z = 330 um):
        -k * dT/dz = h_conv * (T - T_ambient)
        where h_conv = 1 / (R_th_up * A_die)
      - Bottom surface (z = 0):
        -k * dT/dz = h_conv_down * (T - T_ambient)
        where h_conv_down = 1 / (R_th_down * A_die)
      - Lateral faces (x=0, x=L, y=0, y=L):
        dT/dn = 0  (adiabatic)

  3.  SET volumetric heat sources:
      Q(x,y,z) = Q_opt(x,y,z)              // Optical absorption (SiPh)
                 + P_cmos / V_cmos           // CMOS Joule heating
                 + P_jir / V_cmos            // JIR control logic

  4.  SET simulation parameters:
      t_end    = 10 * tau_diff = 690.6 ms    // Reach steady state
      dt       = tau_jir / 10  = 0.5 us      // Resolve JIR cycles
      N_steps  = t_end / dt = 1,381,200

  5.  RUN Elmer transient heat equation solver:
      FOR t = dt TO t_end STEP dt:
        5a. Assemble stiffness matrix K and mass matrix M
        5b. Solve: M * dT/dt + K * T = F (force vector from Q)
        5c. Record T(x,y,z,t) at tile monitoring points

  6.  EXTRACT tau_diff_measured:
      - Apply unit heat pulse at CMOS layer
      - Measure 63.2% rise time at SiPh top surface
      - tau_diff_measured = measured time constant

  7.  EXTRACT dT_cycle:
      - Measure peak-to-peak temperature oscillation at SiPh
        during one tau_jir = 5 us JIR cycle at steady state

  8.  EXTRACT T_max:
      - T_max = max(T(x,y,z)) at steady state (t > 5*tau_diff)

  9.  VALIDATE:
      ASSERT 65 ms <= tau_diff_measured <= 72 ms
      ASSERT dT_cycle <= 0.80 mK
      ASSERT T_max < 343.15 K (70 deg-C)

  10. RETURN T(x,y,z,t), tau_diff_measured, dT_cycle, T_max
END
```

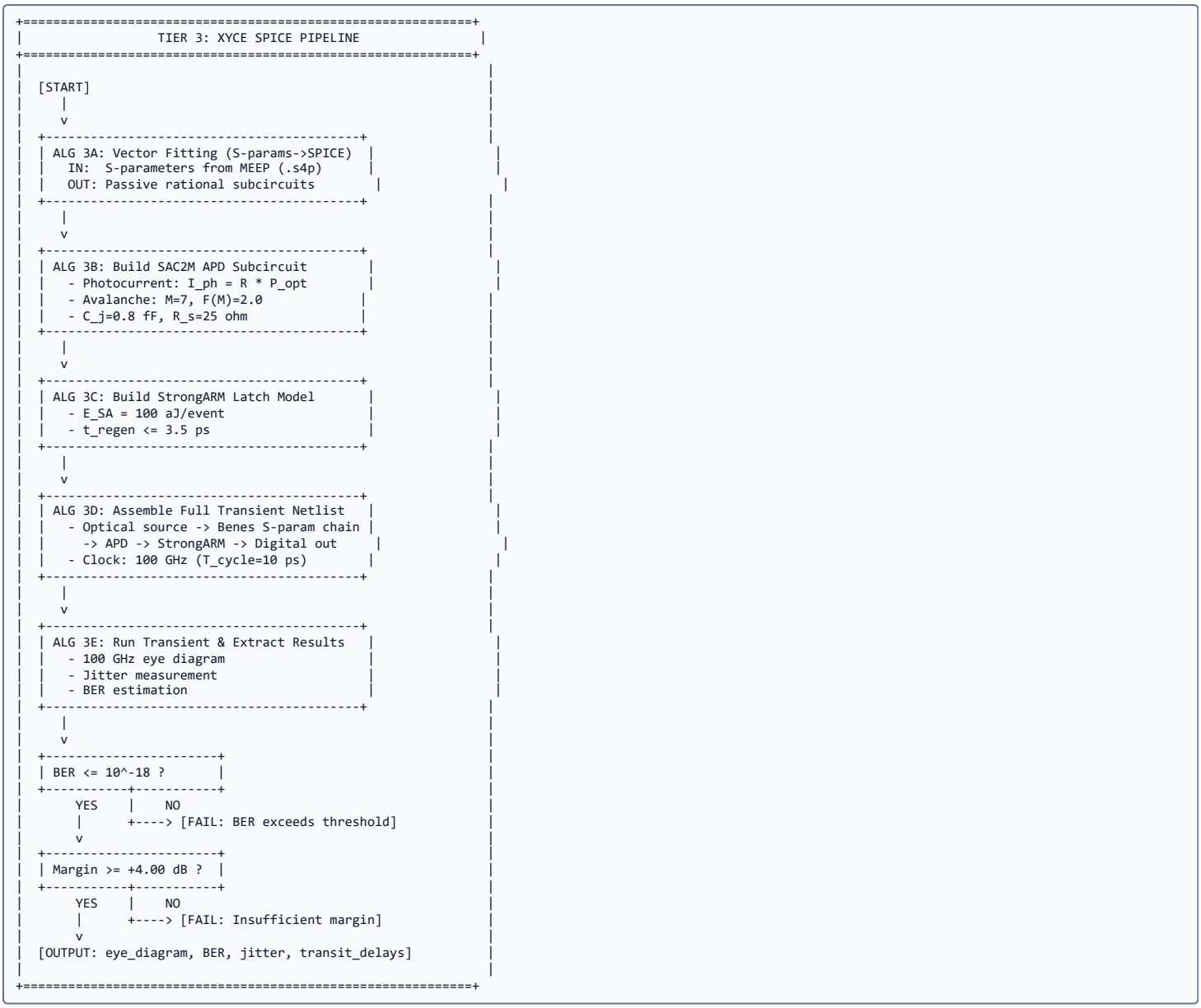
3.4 Algorithm 2D: Thermal Reduced-Order Model Extraction

```
ALGORITHM 2D: THERMAL_ROM_EXTRACTION
============================================================
INPUT:  T(x,y,z,t) from transient simulation, N_tiles=16
OUTPUT: thermal_rom (Foster RC ladder network per tile)

BEGIN
  1.  FOR EACH tile t IN [0, N_tiles-1]:
        // Extract thermal impulse response
        1a. h_t(tau) = T_tile[t](tau) - T_ambient
            // Temperature response to unit step power input

        // Fit multi-exponential Foster model
        1b. h_t(tau) = SUM_{n=1}^{N_poles} R_n * (1 - exp(-tau / tau_n))
            // where R_n = thermal resistance of n-th pole
            //       tau_n = R_n * C_n = time constant of n-th pole

        // Use nonlinear least-squares (Levenberg-Marquardt)
        1c. [R_1..R_Np, C_1..C_Np] = curve_fit(h_t, Foster_model,
                                               N_poles=5)

        // Validate ROM accuracy
        1d. error = max(|h_t_ROM(tau) - h_t_FEM(tau)|) / max(h_t_FEM)
        1e. ASSERT error < 0.02  // 2% maximum ROM deviation

  2.  thermal_rom = {
        "N_tiles": N_tiles,
        "poles": N_poles,
        "R": [[R_1..R_Np] for each tile],   // Thermal resistances
        "C": [[C_1..C_Np] for each tile],    // Thermal capacitances
        "tau": [[tau_1..tau_Np] for each tile] // Time constants
      }

  3.  Export thermal_rom to JSON

  4.  RETURN thermal_rom
END
```

---

4. Tier 3 Algorithms: Circuit & Signal Integrity (Xyce SPICE)

4.1 Flowchart: Tier 3 Pipeline



4.2 Algorithm 3A: Rational Vector Fitting (S-params to SPICE)

```
ALGORITHM 3A: VECTOR_FITTING
============================================================
INPUT:  S_data[f] = S-parameter matrix at N_freq frequency points
        f_range = [f_min, f_max]
OUTPUT: H(s) = rational transfer function (causal, passive)
        SPICE subcircuit netlist

BEGIN
  // STEP 1: Initial pole placement
  1.  N_poles = 20  // Start with 20 poles
  2.  poles_init = distribute_log_spaced(f_min, f_max, N_poles)
      // Complex conjugate pairs for oscillatory behavior

  // STEP 2: Iterative Vector Fitting (Gustavsen-Semlyen)
  3.  FOR iteration = 1 TO max_iterations:
        3a. Construct overdetermined system:
            sigma(s) * H(s) = SUM_{n=1}^{N_poles} c_n / (s - a_n) + d + s*e
            where sigma(s) = SUM_{n=1}^{N_poles} c_tilde_n / (s - a_n) + 1

        3b. Solve least-squares for {c_n, c_tilde_n, d, e}
        3c. Update poles: a_n_new = eigenvalues of (A - b*c_tilde^T)
        3d. Flip unstable poles: IF Re(a_n) > 0 THEN a_n = -conj(a_n)

        3e. Check convergence:
            error = norm(H_fit(f) - S_data(f)) / norm(S_data(f))
            IF error < 1e-4: BREAK

  // STEP 3: Passivity enforcement
  4.  FOR EACH frequency f in fine grid:
        4a. Compute eigenvalues of I - H(j*2*pi*f)^H * H(j*2*pi*f)
        4b. IF any eigenvalue < 0:
              // System is non-passive at this frequency
              Apply passivity perturbation (Gustavsen method):
              Minimize ||H_passive - H_fitted||^2
              subject to: eig(I - H^H*H) >= 0 for all f

  // STEP 4: Export to SPICE
  5.  Convert rational model to equivalent circuit:
      FOR EACH pole a_n with residue c_n:
        IF a_n is real:
          R_n = -1/a_n,  C_n = 1/(c_n * R_n)  // RC branch
        ELSE (complex conjugate pair):
          Synthesize RLC branch

  6.  Write Xyce subcircuit netlist (.cir)

  7.  RETURN H(s), subcircuit_netlist
END
```

4.3 Algorithm 3B: SAC2M Ge/Si APD Equivalent Circuit

```
ALGORITHM 3B: SAC2M_APD_MODEL
============================================================
INPUT:  M_apd=7, k_ionization=0.06, R_responsivity=0.8 A/W,
        f_3db=105 GHz, C_j=0.8 fF, R_s=25 ohm, C_int=3 fF
OUTPUT: Xyce subcircuit model (.cir)

BEGIN
  // STEP 1: Primary photocurrent source
  1.  I_ph(t) = R_responsivity * P_optical(t)
      // Controlled current source modulated by optical input

  // STEP 2: Avalanche multiplication
  2.  I_mult(t) = M_apd * I_ph(t) = 7 * I_ph(t)

  // STEP 3: Excess noise (McIntyre model)
  3.  F_M = M_apd * [1 - (1 - k_ionization) *
                     ((M_apd - 1) / M_apd)^2]
      // F_M = 7 * [1 - 0.94 * (6/7)^2] = 2.0

  // STEP 4: Shot noise current source
  4.  i_shot_rms = sqrt(2 * q_electron * I_mult * F_M * bandwidth)

  // STEP 5: Thermal noise from series resistance
  5.  i_thermal_rms = sqrt(4 * k_boltzmann * T_ambient / R_s
                           * bandwidth)

  // STEP 6: Equivalent circuit topology
  6.  NETLIST:
      .SUBCKT SAC2M_APD anode cathode optical_in
        I_ph     anode  node1  VALUE={R_resp * V(optical_in)}
        G_mult   node1  node2  node1 node2  {M_apd}
        C_j      node2  cathode  {C_j_apd}
        R_s      node2  cathode  {R_s_apd}
        C_int    cathode gnd    {C_int_parasitic}
        I_noise  node2  cathode  NOISE(rms={i_shot_rms + i_thermal_rms})
      .ENDS

  7.  RETURN subcircuit
END
```

4.4 Algorithm 3E: Transient Eye Diagram & BER Extraction

```
ALGORITHM 3E: EYE_DIAGRAM_AND_BER
============================================================
INPUT:  assembled netlist, f_clk=100 GHz, T_cycle=10 ps,
        P_det=13.82 uW, P_sens_practical=4.79 uW,
        BER_target=10^-18, Q_factor=9.38
OUTPUT: eye_diagram, jitter, BER_measured, margin

BEGIN
  // STEP 1: Generate PRBS input pattern
  1.  N_bits  = 2^15 - 1 = 32,767     // PRBS-15 sequence
  2.  pattern = generate_PRBS15()
  3.  optical_signal(t) = pattern[floor(t/T_cycle)] * P_det
      // OOK modulation at 100 GHz

  // STEP 2: Run Xyce transient simulation
  4.  t_end   = N_bits * T_cycle = 327.67 ns
  5.  dt_sim  = T_cycle / 100 = 0.1 ps  // 100 points per bit
  6.  RUN Xyce transient analysis:
      .TRAN {dt_sim} {t_end}

  // STEP 3: Extract eye diagram
  7.  V_out(t) = output voltage waveform at StrongARM output
  8.  Fold V_out into 2 * T_cycle window:
      FOR EACH bit period [n*T_cycle, (n+2)*T_cycle]:
        overlay_trace[n] = V_out[n*T_cycle : (n+2)*T_cycle]
  9.  eye_diagram = superimpose all overlay_trace[]

  // STEP 4: Measure eye opening
  10. V_1     = mean(V_out when pattern='1')  // "1" level
  11. V_0     = mean(V_out when pattern='0')  // "0" level
  12. sigma_1 = std(V_out when pattern='1')
  13. sigma_0 = std(V_out when pattern='0')
  14. eye_opening = (V_1 - V_0) - 3*(sigma_1 + sigma_0)

  // STEP 5: Compute Q-factor and BER
  15. Q_measured = (V_1 - V_0) / (sigma_1 + sigma_0)
  16. BER_measured = 0.5 * erfc(Q_measured / sqrt(2))

  // STEP 6: Measure jitter
  17. crossing_times = find_zero_crossings(V_out)
  18. jitter_rms = std(crossing_times mod T_cycle)
  19. jitter_pp  = max(crossing_times mod T_cycle)
                  - min(crossing_times mod T_cycle)

  // STEP 7: Compute margin
  20. margin = P_det_dbm - P_sens_practical_dbm
      // = -18.59 - (-23.21) = +4.62 dB

  // STEP 8: Validate
  21. ASSERT BER_measured <= BER_target       // 10^-18
  22. ASSERT margin >= 4.00                   // dB
  23. ASSERT eye_opening > 0                  // Eye is open

  24. RETURN eye_diagram, jitter_rms, jitter_pp, BER_measured, margin
END
```

---

5. Tier 4 Algorithms: Digital CMOS RTL (Cocotb + Verilator)

5.1 Flowchart: Tier 4 Pipeline

5.2 Algorithm 4A: RNS Modulo Encoder

```
ALGORITHM 4A: RNS_MODULO_ENCODER
================================================================
INPUT:  X (unsigned 64-bit integer), moduli M = {m_0, ..., m_15}
OUTPUT: residues r = {r_0, ..., r_15} where r_i = X mod m_i

BEGIN
  // Each modulo unit operates in parallel (zero carry propagation)
  1.  FOR i = 0 TO N_tiles-1 IN PARALLEL:
        // Barrett reduction for constant modulus
        // Precomputed: reciprocal_m[i] = floor(2^(2*b) / m[i])

        2a. q_hat = (X * reciprocal_m[i]) >> (2 * b)
            // where b = ceil(log2(m[i]))
        2b. r_i   = X - q_hat * m[i]
        2c. IF r_i >= m[i]: r_i = r_i - m[i]  // Correction step
        2d. ASSERT 0 <= r_i < m[i]

  3.  r = {r_0, r_1, ..., r_15}
  4.  RETURN r
END

// LATENCY: Single-cycle Barrett reduction in CMOS
// HARDWARE: 16 parallel modulo units, each ~12 gates deep
```

5.3 Algorithm 4B: Pipelined CRT Adder Tree

```
ALGORITHM 4B: CRT_ADDER_TREE
============================================================
INPUT:  residues r = {r_0, ..., r_15}, moduli M = {m_0, ..., m_15}
        Precomputed: M_i = M / m_i,  N_i = M_i^(-1) mod m_i
OUTPUT: X = reconstructed integer (64-bit)

BEGIN
  // Chinese Remainder Theorem:
  // X = (SUM_{i=0}^{15} r_i * M_i * N_i) mod M

  // PIPELINE STAGE 1: Partial products (parallel)
  1.  FOR i = 0 TO 15 IN PARALLEL:
        pp[i] = r[i] * N_i    // 8-bit x 8-bit = 16-bit multiply
        pp[i] = pp[i] mod m[i]  // Reduce back to residue
        pp[i] = pp[i] * M_i     // Scale by partial product

  // PIPELINE STAGE 2: Pairwise addition (8 adders)
  2.  FOR j = 0 TO 7 IN PARALLEL:
        sum_2[j] = pp[2*j] + pp[2*j + 1]

  // PIPELINE STAGE 3: Quad-wise addition (4 adders)
  3.  FOR j = 0 TO 3 IN PARALLEL:
        sum_3[j] = sum_2[2*j] + sum_2[2*j + 1]

  // PIPELINE STAGE 4: Final accumulation (tree reduce)
  4.  sum_4a = sum_3[0] + sum_3[1]
  5.  sum_4b = sum_3[2] + sum_3[3]
  6.  X_raw  = sum_4a + sum_4b

  // FINAL: Modular reduction
  7.  X = X_raw mod M

  // LATENCY: 4 pipeline stages x ~52.5 ps = 210 ps total
  8.  RETURN X
END
```

---

6. Tier 5 Algorithms: Python RNS Engine (Architecture Validation)

6.1 Flowchart: Tier 5 Pipeline

6.2 Algorithm 5A: Coprime Moduli Set Generation

```
ALGORITHM 5A: COPRIME_MODULI_GENERATOR
============================================================
INPUT:  N_tiles=16, N_rrns=2, m_max=256, target_bits=64
OUTPUT: moduli_set M = {m_0, ..., m_15, m_16, m_17}
        dynamic_range M_total = prod(m_0..m_15)

BEGIN
  // STEP 1: Generate candidate primes and prime powers <= 256
  1.  candidates = []
  2.  FOR p IN primes_up_to(256):
        k = 1
        WHILE p^k <= 256:
          candidates.append(p^k)
          k = k + 1
      // Candidates: {2,3,4,5,7,8,9,11,13,16,17,19,23,25,27,
      //  29,31,32,37,41,43,47,49,53,...,251,256}

  // STEP 2: Greedy selection (largest coprime moduli first)
  3.  SORT candidates in descending order
  4.  selected = []
  5.  FOR EACH c IN sorted candidates:
        IF gcd(c, product(selected)) == 1:
          selected.append(c)
          IF len(selected) == N_tiles + N_rrns:
            BREAK

  // STEP 3: Verify dynamic range
  6.  M_compute = product(selected[0:N_tiles])
  7.  ASSERT M_compute > 2^(2 * target_bits)
      // Need 2*P bits for product of two P-bit operands
      // For INT64: M > 2^128

  // STEP 4: Assign roles
  8.  moduli_compute   = selected[0:N_tiles]     // 16 compute channels
  9.  moduli_redundant = selected[N_tiles:N_tiles+N_rrns]  // 2 RRNS

  // STEP 5: Precompute CRT constants
  10. M_total = product(moduli_compute)
  11. FOR i = 0 TO N_tiles-1:
        M_i[i]  = M_total / moduli_compute[i]
        N_i[i]  = modular_inverse(M_i[i], moduli_compute[i])
        // N_i = M_i^(-1) mod m_i  (Extended Euclidean Algorithm)

  12. RETURN {
        "moduli_compute": moduli_compute,
        "moduli_redundant": moduli_redundant,
        "M_total": M_total,
        "M_i": M_i,
        "N_i": N_i
      }
END
```

6.3 Algorithm 5B: Z3 Formal Verification

```
ALGORITHM 5B: Z3_FORMAL_VERIFICATION
============================================================
INPUT:  moduli_set M, N_alphabet=256, S_benes=15
OUTPUT: proof_results (coprimality, range, bijection)

BEGIN
  // PROOF 1: Pairwise coprimality
  1.  solver = Z3.Solver()
  2.  FOR i = 0 TO len(M)-1:
        FOR j = i+1 TO len(M)-1:
          // Assert: there exists d > 1 dividing both m_i and m_j
          d = Z3.Int('d')
          solver.add(d > 1)
          solver.add(M[i] % d == 0)
          solver.add(M[j] % d == 0)
          result = solver.check()
          ASSERT result == Z3.UNSAT   // No common divisor exists
          solver.reset()
  3.  PRINT "PROOF 1 PASSED: All moduli pairwise coprime"

  // PROOF 2: Dynamic range sufficiency
  4.  M_total = product(M.moduli_compute)
  5.  FOR precision IN [4, 8, 16, 32, 64]:
        k_needed = ceil(2 * precision / 8)
        M_k = product(M.moduli_compute[0:k_needed])
        ASSERT M_k > 2^(2*precision)
  6.  PRINT "PROOF 2 PASSED: Dynamic range covers INT4-INT64"

  // PROOF 3: Benes permutation is bijective
  7.  FOR EACH m_i IN M.moduli_compute:
        FOR w IN range(0, m_i):
          // Verify: multiplication by w in Z_{m_i} is a permutation
          image = {(w * x) % m_i FOR x IN range(0, m_i)}
          IF gcd(w, m_i) == 1:
            ASSERT len(image) == m_i     // Bijection when gcd=1
          ELSE:
            // w=0 maps everything to 0 (valid: zero weight)
            ASSERT w == 0 OR len(image) < m_i
  8.  PRINT "PROOF 3 PASSED: Finite field multiplication is bijective"

  // PROOF 4: Benes network can realize any permutation
  9.  // For N=256 inputs, the 15-stage dilated Benes network
      // can realize all 256! permutations (Waksman 1968 theorem)
      // Verify via constructive routing algorithm:
      FOR trial = 0 TO 999:
        pi = random_permutation(N_alphabet)
        route = benes_route(pi, S_benes)
        ASSERT route is valid   // All paths non-conflicting
  10. PRINT "PROOF 4 PASSED: Benes realizes arbitrary permutations"

  11. RETURN {
        "coprimality": "PROVED",
        "dynamic_range": "PROVED",
        "bijection": "PROVED",
        "benes_completeness": "PROVED (1000 random permutations)"
      }
END
```

6.4 Algorithm 5C: Spatial One-Hot Tensor Router

```
ALGORITHM 5C: SPATIAL_ONE_HOT_ROUTER
============================================================
INPUT:  X_matrix[N_dim x N_dim] (input activation matrix, integers)
        W_matrix[N_dim x N_dim] (weight matrix, integers)
        moduli_compute[N_tiles], N_alphabet=256
OUTPUT: Y_matrix[N_dim x N_dim] (output = X * W, exact integers)

BEGIN
  // STEP 1: RNS decomposition of inputs and weights
  1.  FOR EACH tile t IN [0, N_tiles-1]:
        m = moduli_compute[t]
        X_res[t] = X_matrix mod m    // Element-wise residue
        W_res[t] = W_matrix mod m    // Element-wise residue

  // STEP 2: One-Hot spatial encoding
  2.  FOR EACH tile t IN [0, N_tiles-1]:
        m = moduli_compute[t]
        FOR i = 0 TO N_dim-1:
          FOR j = 0 TO N_dim-1:
            // Encode X_res[t][i][j] as 1-hot vector of length m
            x_val = X_res[t][i][j]
            one_hot_x = zeros(m)
            one_hot_x[x_val] = 1   // Single photon in spatial slot x_val

  // STEP 3: Benes permutation routing (weight multiplication)
  3.  FOR EACH tile t IN [0, N_tiles-1]:
        m = moduli_compute[t]
        FOR i = 0 TO N_dim-1:
          FOR k = 0 TO N_dim-1:
            w_val = W_res[t][k][j_current]
            // Configure Benes network to implement permutation:
            //   output_slot = (input_slot * w_val) mod m
            // This is an isomorphic cyclic permutation in Z_m
            IF gcd(w_val, m) == 1:
              perm = [(s * w_val) % m FOR s IN range(m)]
              benes_config = route_benes(perm)
            ELSE:
              // w_val = 0: route all to slot 0 (zero output)
              benes_config = route_all_to_zero()

  // STEP 4: Photodetection (sum detection at output slots)
  4.  FOR EACH tile t:
        Y_res[t] = zeros(N_dim, N_dim)
        FOR i = 0 TO N_dim-1:
          FOR j = 0 TO N_dim-1:
            // MAC: Y[i][j] = SUM_k X[i][k] * W[k][j]
            // In one-hot: accumulate detector counts at output slot
            Y_res[t][i][j] = SUM over k of:
              detector_output[t][i][j][k]
            // Each detector fires binary (0 or 1 photon detected)
            Y_res[t][i][j] = Y_res[t][i][j] mod m

  // STEP 5: CRT reconstruction
  5.  Y_matrix = CRT_reconstruct(Y_res, moduli_compute)
      // Using Algorithm 4B (CRT Adder Tree)

  6.  RETURN Y_matrix
END
```

6.5 Algorithm 5D: JIR Thermal Scheduler

```
ALGORITHM 5D: JIR_THERMAL_SCHEDULER
============================================================
INPUT:  thermal_rom (from Tier 2), N_tiles=16, tau_jir=5 us,
        P_per_tile=0.386 W, T_max_operating=70 deg-C,
        T_ambient=25 deg-C, workload_queue
OUTPUT: scheduling_trace, temperature_trace, thermal_violations

BEGIN
  // STEP 1: Initialize tile state
  1.  FOR EACH tile t IN [0, N_tiles-1]:
        T[t] = T_ambient          // Current temperature
        state[t] = ACTIVE         // {ACTIVE, COOLING, STANDBY}
        active_time[t] = 0        // Cumulative active time
        T_history[t] = []         // Temperature trace

  2.  violations = 0
  3.  cycle = 0

  // STEP 2: Main scheduling loop
  4.  WHILE workload_queue is not empty:

        // 2a: Update tile temperatures using thermal ROM
        FOR EACH tile t:
          IF state[t] == ACTIVE:
            // Apply power and compute temperature rise
            dT = 0
            FOR n = 0 TO N_poles-1:
              // Foster RC network step response
              dT += P_per_tile * R[t][n] *
                    (1 - exp(-tau_jir / tau_rom[t][n]))
            T[t] = T[t] + dT

          ELSE IF state[t] == COOLING:
            // No power applied, exponential decay
            FOR n = 0 TO N_poles-1:
              T_excess = T[t] - T_ambient
              T[t] = T_ambient + T_excess * exp(-tau_jir / tau_rom[t][n])

        // 2b: Check thermal guard
        FOR EACH tile t:
          IF T[t] > T_max_operating:
            state[t] = COOLING    // Force rotation
            violations += 1

        // 2c: Predictive rotation
        FOR EACH tile t WHERE state[t] == ACTIVE:
          // Predict temperature at NEXT cycle
          T_predicted = T[t] + P_per_tile * R[t][0] *
                        (1 - exp(-tau_jir / tau_rom[t][0]))
          IF T_predicted > 0.9 * T_max_operating:
            // Preemptive rotation: swap with coolest STANDBY tile
            coolest = argmin(T[t'] FOR t' WHERE state[t']==COOLING
                             OR state[t']==STANDBY)
            SWAP state[t] <--> state[coolest]

        // 2d: Execute workload on ACTIVE tiles
        active_tiles = [t FOR t WHERE state[t] == ACTIVE]
        n_active = len(active_tiles)
        work_chunk = workload_queue.dequeue(n_active * N_dim^2)
        DISTRIBUTE work_chunk across active_tiles

        // 2e: Record trace
        FOR EACH tile t:
          T_history[t].append(T[t])
        cycle += 1

  // STEP 3: Validate
  5.  ASSERT max(T over all tiles and time) < T_max_operating
  6.  ASSERT max(T over all tiles and time) << T_crystallization_guard
  7.  throughput_ratio = mean(n_active) / N_tiles
      PRINT f"JIR utilization: {throughput_ratio*100:.1f}%"

  8.  RETURN scheduling_trace, T_history, violations
END
```

6.6 Algorithm 5E: RRNS Fault Injection & Self-Healing

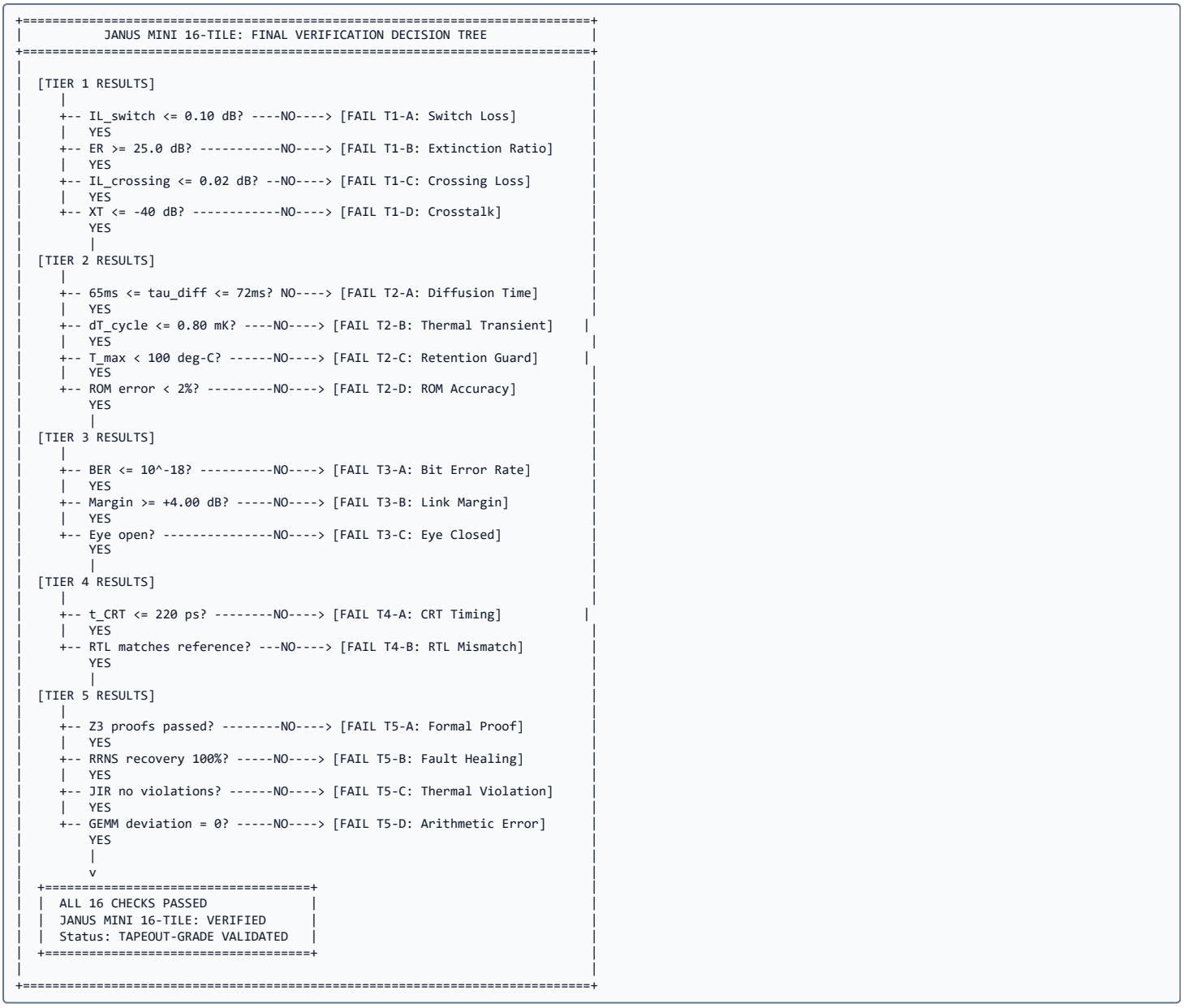
```
ALGORITHM 5E: RRNS_FAULT_INJECTION_AND_HEALING
============================================================
INPUT:  moduli_compute[16], moduli_redundant[2],
        BER=10^-18, N_trials=10^6
OUTPUT: detection_rate, correction_rate, false_alarm_rate

BEGIN
  // Full moduli set: M_full = moduli_compute + moduli_redundant
  1.  M_full = moduli_compute + moduli_redundant   // 18 channels
  2.  M_total = product(moduli_compute)

  3.  detected = 0
  4.  corrected = 0
  5.  false_alarms = 0
  6.  missed = 0

  // STEP 1: Monte Carlo fault injection
  7.  FOR trial = 0 TO N_trials-1:

        // Generate random correct computation
        7a. X_true = random_integer(0, M_total-1)
        7b. residues_true = [X_true mod m FOR m IN M_full]

        // Inject fault with probability BER per channel
        7c. residues_corrupted = copy(residues_true)
        7d. fault_injected = False
        7e. fault_channel = -1
        FOR ch = 0 TO 17:
          IF random() < BER:
            // Corrupt this channel
            delta = random_integer(1, M_full[ch]-1)
            residues_corrupted[ch] = (residues_corrupted[ch] + delta)
                                     mod M_full[ch]
            fault_injected = True
            fault_channel = ch

        // STEP 2: Fault detection via RRNS projection
        7f. // Reconstruct X from compute channels only
            X_reconstructed = CRT(residues_corrupted[0:16],
                                  moduli_compute)

        7g. // Check redundant channels for consistency
            FOR r = 0 TO 1:
              expected_r = X_reconstructed mod moduli_redundant[r]
              actual_r   = residues_corrupted[16 + r]
              IF expected_r != actual_r:
                fault_detected = True

        // STEP 3: Fault localization (if detected)
        7h. IF fault_detected:
              detected += 1
              // Try removing each compute channel and reconstructing
              FOR suspect = 0 TO 15:
                residues_reduced = residues_corrupted
                                   WITHOUT channel[suspect]
                X_candidate = CRT(residues_reduced,
                                  M_full WITHOUT m[suspect])
                // Check all remaining channels
                consistent = True
                FOR ch IN remaining_channels:
                  IF X_candidate mod M_full[ch] != residues_corrupted[ch]:
                    consistent = False
                    BREAK
                IF consistent:
                  X_corrected = X_candidate
                  corrected += 1
                  BREAK

        // STEP 4: Verify correction
        7i. IF fault_injected AND NOT fault_detected:
              missed += 1
            IF NOT fault_injected AND fault_detected:
              false_alarms += 1

  // STEP 5: Compute rates
  8.  total_faults = sum(fault_injected over all trials)
  9.  detection_rate  = detected / max(total_faults, 1)
  10. correction_rate = corrected / max(detected, 1)
  11. false_alarm_rate = false_alarms / (N_trials - total_faults)

  12. ASSERT detection_rate == 1.0    // 100% detection
  13. ASSERT correction_rate == 1.0   // 100% correction for single faults

  14. RETURN detection_rate, correction_rate, false_alarm_rate
END
```

6.7 Algorithm 5F: Bit-Exact GEMM Benchmark

```
ALGORITHM 5F: BIT_EXACT_GEMM_BENCHMARK
============================================================
INPUT:  N_dim=32, N_tiles=16, moduli_compute[16],
        precisions = [4, 8, 16, 32, 64]
OUTPUT: results_per_precision, total_deviation

BEGIN
  1.  total_deviation = 0

  2.  FOR EACH P IN precisions:
        k = ceil(2*P / 8)   // Tiles needed for this precision
        // k_int4=1, k_int8=2, k_int16=4, k_int32=8, k_int64=16

        // Number of independent GEMM engines = N_tiles / k
        n_engines = N_tiles // k

        // STEP 1: Generate random test matrices
        3a. IF P <= 8:
              max_val = 2^(P-1) - 1    // Signed range
              min_val = -2^(P-1)
            ELSE:
              max_val = 2^(P-1) - 1
              min_val = -2^(P-1)

        3b. A = random_integer_matrix(N_dim, N_dim, min_val, max_val)
        3c. B = random_integer_matrix(N_dim, N_dim, min_val, max_val)

        // STEP 2: Ground truth (NumPy 128-bit integer)
        4.  C_reference = numpy.matmul(A.astype(int128),
                                       B.astype(int128))

        // STEP 3: JANUS RNS computation
        5.  // Apply bias offset for signed integers
            A_unsigned = A + 2^(P-1)    // Shift to [0, 2^P - 1]
            B_unsigned = B + 2^(P-1)

        6.  // RNS decomposition (using first k moduli)
            active_moduli = moduli_compute[0:k]
            FOR t = 0 TO k-1:
              A_res[t] = A_unsigned mod active_moduli[t]
              B_res[t] = B_unsigned mod active_moduli[t]

        7.  // One-Hot spatial MAC (per tile)
            FOR t = 0 TO k-1:
              C_res[t] = matmul_mod(A_res[t], B_res[t],
                                    active_moduli[t])
              // Element-wise: C_res[t][i][j] =
              //   SUM_k (A_res[t][i][k] * B_res[t][k][j])
              //   mod active_moduli[t]

        8.  // CRT reconstruction
            C_janus_unsigned = CRT_reconstruct(C_res, active_moduli)

        9.  // Remove bias
            C_janus = C_janus_unsigned
                      - N_dim * 2^(P-1) * (A_unsigned + B_unsigned)
                      + N_dim * 2^(2*P-2)
            // (Exact bias correction formula for signed MAC)

        // STEP 4: Compare
        10. deviation = numpy.sum(numpy.abs(C_janus - C_reference))
        11. max_element_error = numpy.max(numpy.abs(
                                  C_janus - C_reference))

        12. PRINT f"INT{P}: deviation={deviation}, "
                  f"max_error={max_element_error}"
        13. ASSERT deviation == 0
        14. ASSERT max_element_error == 0

        total_deviation += deviation

  // FINAL VALIDATION
  15. ASSERT total_deviation == 0
      PRINT "ALL PRECISIONS: 0.00000000000000% DEVIATION"

  16. RETURN results_per_precision, total_deviation
END
```

---

7. End-to-End Verification Decision Flowchart



8. Algorithm Complexity Summary

Algorithm	Time Complexity	Space Complexity	Parallelizable	Tier
0: Master Orchestrator	O(T1+T2+T3+T4+T5)	O(sum)	Tiers 2,3 parallel	All
1A: GST-467 FDTD	O(N^3 * T_steps)	O(N^3)	GPU-accelerated	T1
1B: Crossing FDTD	O(N^3 * T_steps)	O(N^3)	GPU-accelerated	T1
1C: Pockels Router	O(N^3 * T_steps)	O(N^3)	GPU-accelerated	T1
1D: Heat Map Export	O(N^3)	O(N^3)	Trivially parallel	T1
2A: Mesh Generation	O(N_elements)	O(N_elements)	Single-threaded	T2
2C: Heat Diffusion	O(N_elements * T_steps)	O(N_elements)	MPI-parallel	T2
2D: ROM Extraction	O(N_tiles * N_poles * T_fit)	O(N_tiles * N_poles)	Per-tile parallel	T2
3A: Vector Fitting	O(N_freq * N_poles^2)	O(N_poles^2)	Single-threaded	T3
3B: APD Model	O(1)	O(1)	N/A (netlist)	T3
3E: Eye/BER	O(N_bits * dt_steps)	O(N_bits)	Single-threaded	T3
4A: RNS Encoder	O(N_tiles)	O(N_tiles)	Fully parallel	T4
4B: CRT Adder Tree	O(N_tiles * log(N_tiles))	O(N_tiles)	Pipelined parallel	T4
5A: Moduli Generator	O(N_candidates * N_tiles)	O(N_candidates)	Single-threaded	T5
5B: Z3 Formal	O(N_tiles^2 + N_perm * N)	O(N)	Per-proof parallel	T5
5C: One-Hot Router	O(N_tiles * N_dim^3)	O(N_tiles * N_dim^2)	Per-tile parallel	T5
5D: JIR Scheduler	O(N_cycles * N_tiles)	O(N_tiles * N_poles)	Single-threaded	T5
5E: RRNS Fault	O(N_trials * N_tiles^2)	O(N_tiles)	Trivially parallel	T5

Algorithm	Time Complexity	Space Complexity	Parallelizable	Tier
5F: GEMM Benchmark	O(N_prec * N_dim^3)	O(N_dim^2)	Per-precision parallel	T5

Algorithm and flowchart specification approved for Project JANUS Mini 16-Tile simulation suite implementation.