GA (genetic algorithm) Projet Report

Name: Alirzea Sotoodeh

Date: 1404/05/28

master: Phd. Ali Mahani

define: a module which mimics the process of natural selection to find optimal or near-optimal solutions to optimization and search problems.

Goals

- **inputs:** CLK, Data_in `16 bit`, population, start, iteration, 1 or 2 point cross over, mutation rate
- **Output:** Data_out `16 bit`, Done, number of chromosome

Detailed Old vs New Module Comparisons

1. Crossover Module

Old Version (crossover_old.sv)

- **Purpose**: Creates a single offspring chromosome from two parents using **single-point crossover**.
- **Key Parameters**:
  - `CHROMOSOME_WIDTH` = 8 bits (default)
- **Inputs**: Clock, Reset (`rst_n`), `start_crossover` , `parent1` , `parent2` , and `crossover_point` (3 bits).
- **Outputs**: `child` , `crossover_done`
- **Behavior**:
  - On reset: `child = 0` , `crossover_done = 0`
  - On `start_crossover` :

        - Hardcoded logic takes **upper 5 bits** from `parent1` and **lower 3 bits** from `parent2` (manual fixed split at bit 3).
        - `crossover_point` input exists but is **not used dynamically**.
    - Only **single-point** mode available; no floating-point or uniform crossover.
- **Limitations**:
    - Crossover point hardcoded, ignoring user input.
    - Width fixed to 8 bits.
    - No double-point or uniform mode.

New Version (crossover.sv)

- **Purpose**: Generates offspring using **multiple crossover methods**:
    - Fixed-point (single or double)
    - Floating-point (single or double, LFSR-driven for randomness)
    - Uniform crossover (mask-based, randomizable via LFSR)
- **Key Parameters**:
    - `CHROMOSOME_WIDTH` (default = 16, fully parameterized)
    - `LFSR_WIDTH` (default = 16)
- **Inputs**:
    - Clock, Reset (`rst`)
    - `start_crossover`
    - `parent1` , `parent2`
    - Selection signals:
        - `crossover_mode` (00=fixed, 01=float, 10=uniform)
        - `crossover_single_double` (0=single, 1=double)
        - `crossover_single_point` , `crossover_double_point1` , `crossover_double_point2`
        - `mask_uniform` , `uniform_random_enable` , `LFSR_input`
- **Outputs**: `child` , `crossover_done`
- **Behavior Enhancements**:
    - **Fixed-point**: supports dynamic points; double-point mode sorts points automatically.
    - **Floating-point**: crossover points derived from LFSR bits for randomness.
    - **Uniform**: bitmask-based selection from both parents; mask can be constant or generated from LFSR.
    - Parameterized for any chromosome width.
- **Optimizations**:
    - Point sorting logic for double-point mode.

- Masks precomputed in combinational logic for clarity and FPGA efficiency.
- `keep_hierarchy`, `use_dsp="no"`, and `keep` pragmas for synthesis visibility.

Summary of Changes

Aspect / Feature	Old Version (`crossover_old.sv`)	New Version (`crossover.sv`)
Functionality / Algorithm	Fixed single-point crossover at hardcoded bit position; ignores `crossover_point` input.	Supports single fixed, single float, double fixed, double float, and uniform crossover using LFSR mask.
Parameterization	None — hardwired 8-bit chromosome width.	Parameterized `CHROMOSOME_WIDTH` (default 16 bits) with `$clog2` crossover point input width.
Inputs / Outputs	Inputs: `parent1`, `parent2`, `start_crossover`, `crossover_point` (unused); Outputs: `child`, `crossover_done`.	Inputs: same but point/mask used; Outputs: parameterized `child`, `crossover_done`.
Randomness Handling	None.	LFSR-driven random point/mask with reproducibility via seed control in RNG module.
Boundary / Error Checks	No checks for `crossover_point == 0` or `== WIDTH`.	Explicit handling for 0 and max point to avoid identical output.
Reset Type	Active-low asynchronous.	Active-high synchronous pipeline reset.
Implementation Style	Monolithic always_ff with manual bit assignments.	Case-based crossover mode selector + combinational pre-processing + registered outputs.
Synthesis Attributes	None.	`keep_hierarchy`, `mark_debug`, `dont_touch` for debug preservation; mode logic pruned at synthesis if disabled.
FPGA Resource Efficiency	Very low usage but inflexible; no mode select logic.	Slightly more LUTs consumed but modes are parameter-controlled; synthesis removes unused logic.
Testbench Compatibility	Cannot change crossover type/point dynamically; test coverage limited.	Fully configurable during simulation; deterministic test possible by fixed RNG seed.

Aspect / Feature	Old Version (`crossover_old.sv`)	New Version (`crossover.sv`)
Known Limitations / Issues	Cannot reproduce correct variable crossovers; limited to 8 bits.	None major; must ensure consistent LFSR seeding for repeatable results.

```
create_clock -name Clock -period 90.000 [get_ports clk]
```

Setup

Worst Negative Slack (WNS):	79.813 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	6091

Hold

Worst Hold Slack (WHS):	0.149 ns
Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	6091

Pulse Width

Worst Pulse Width Slack (WPWS):	44.500 ns
Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	3047

All user specified timing constraints are met.

Name	Slice LUTs (8000)	Slice Registers (16000)	F7 Muxes (4000)	F8 Muxes (2000)	Bonded IOB (100)	BUFGCTRL (16)
N population_memory	4141	3046	826	397	1503	1

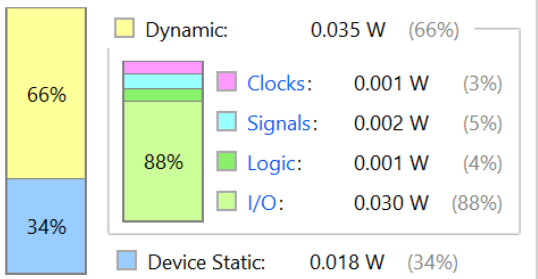
- Settings
- Summary (0.053 W, Margi
- Power Supply
- Utilization Details
  - Hierarchical (0.035 W)
  - Clocks (0.001 W)
  - Signals (0.002 W)
    - Data (0.002 W)
    - Clock Enable (<0.(
    - Set/Reset (0 W)
  - Logic (0.001 W)
  - I/O (0.03 W)

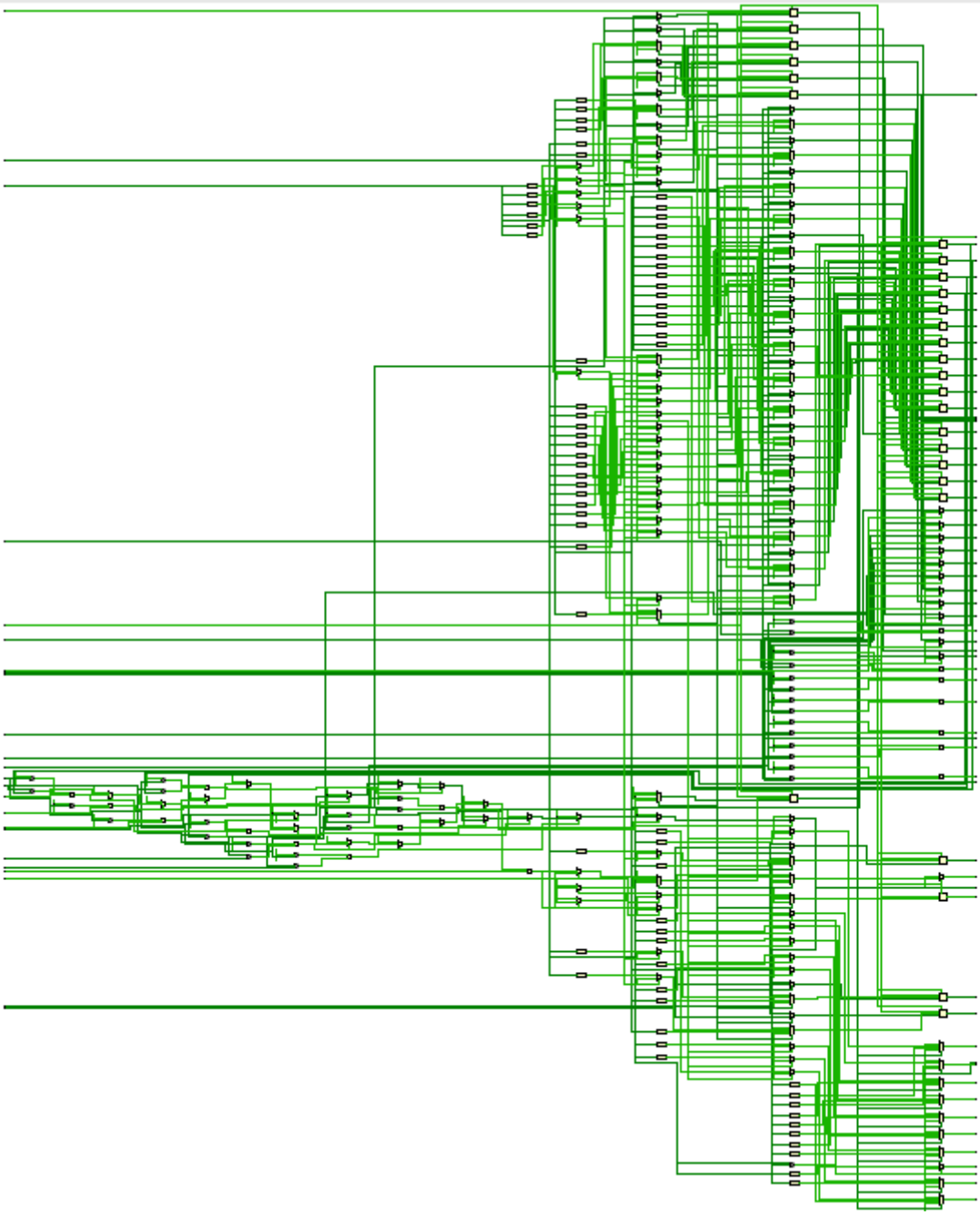
Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**0.053 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.6°C**
Thermal Margin:	59.4°C (5.4 W)
Effective ϑJA:	10.8°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power





- **error:** `crossover_Single_point is not a constant`
- **solution:** use mask

```
////////////////////////////////////////////////////////////////////////////////
/
// due to error crossover_Single_point (and others) is not a constant -> define
mask1&2
// mask1: Used for single-point crossover
// mask2: Used for double-point crossover (middle segment from parent1)
// parent2 == 0 & parent1 =1
////////////////////////////////////////////////////////////////////////////////
```

2. Fitness Evaluator Module

Old Version (fitness_evaluator_old.sv)

- **Purpose**: Computes the fitness score of a chromosome by **counting the number of '1' bits**.
- **Key Parameters**:
  - `CHROMOSOME_WIDTH` = 8
  - `FITNESS_WIDTH` = 10
- **Inputs**: Clock, Reset (`rst_n` active-low), `start_evaluation` , `chromosome`
- **Outputs**: `fitness` , `evaluation_done`
- **Behavior**:
  - On reset: clears `fitness` to 0 and de-asserts `evaluation_done` .
  - On `start_evaluation` :
    - Uses a **for loop inside the sequential block** to increment `fitness` for each bit set to 1 in the chromosome.
    - Sets `evaluation_done` high for 1 cycle.
- **Limitations**:
  - Bit width fixed to 8 for chromosome and 10 for fitness.
  - Counting logic uses direct `fitness <= fitness + 1'b1` accumulation inside loop (functionally fine in synthesis, but not the cleanest style).
  - No protection against counting overflow.
  - Reset signal active-low, which may not be consistent with other modules.

New Version (fitness_evaluator.sv)

- **Purpose**: Same function — calculates fitness by **counting the '1' bits** — but with higher flexibility, safety checks, and synthesis-friendly design.
- **Key Parameters**:
  - `CHROMOSOME_WIDTH` = **16** (default, fully parameterized)
  - `FITNESS_WIDTH` = **14**
- **Inputs**: Clock, Reset (`rst` , active-high), `start_evaluation` , `chromosome`
- **Outputs**: `fitness` , `evaluation_done`
- **Design Improvements**:
  - **Combinational preparation stage**:
    - Counts '1' bits into an internal `raw_fitness` variable in an **always_comb** block.

        - Ensures `raw_fitness` never exceeds the maximum representable value (`FITNESS_WIDTH`).
    - **Sequential update stage**:
        - On `start_evaluation`, latches `raw_fitness` into `fitness` and asserts `evaluation_done`.
        - Clean separation between combinational and sequential logic improves readability and FPGA timing.
    - **Safety**: Overflow clamp prevents `fitness` from wrapping around for small `FITNESS_WIDTH` values.
    - **Parameterization**: Works with any chromosome size without changing the core logic.
- **Pragmas for synthesis**:
    - `keep_hierarchy = "yes"` to preserve hierarchy.
    - `use_dsp = "no"`, `keep = "true"` to control optimization and debug visibility.

Summary of Changes

Aspect / Feature	Old Version (`fitness_evaluator_old.sv`)	New Version (`fitness_evaluator.sv`)
Functionality / Algorithm	Counts number of '1' bits in chromosome. Sequential loop inside always_ff updates accumulator directly.	Counts '1's combinationally into temp var; result registered at clock edge.
Parameterization	Fixed CHROM=8 bits, FIT=10 bits.	Parameterized `CHROMOSOME_WIDTH` (default 16) and `FITNESS_WIDTH` (default 14).
Inputs / Outputs	Inputs: `chromosome`, `start_evaluation`; Outputs: `fitness`, `evaluation_done`.	Same ports with parametric widths.
Randomness Handling	N/A.	N/A.
Boundary / Error Checks	None; if width mismatch, synthesis warning possible.	Overflow clamping: if fitness exceeds max representable, saturates to FITNESS_WIDTH max.
Reset Type	Active-low asynchronous.	Active-high synchronous.
Implementation Style	Sequential counting inside loop; potentially inefficient for large widths.	Combinational bit-count using loop/unrolled logic, then store to register.
Synthesis Attributes	None.	`keep_hierarchy`, `use_dsp="no"` for predictable resource inference.
FPGA Resource Efficiency	Minimal LUT usage but fixed to 8 bits.	Scales with CHROMOSOME_WIDTH; synthesis optimizes loop unrolling; no DSPs used.
Testbench Compatibility	Works for fixed width only; cannot stress variable width behavior.	Fully scalable test stimuli; easy to overload with longer chromosomes.

Aspect / Feature	Old Version (`fitness_evaluator_old.sv`)	New Version (`fitness_evaluator.sv`)
Known Limitations / Issues	Over-allocates fitness bits; no safety for invalid data width.	None significant.

```
create_clock -name Clock -period 90.000 [get_ports clk]
```

Setup

Worst Negative Slack (WNS):	inf
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	50

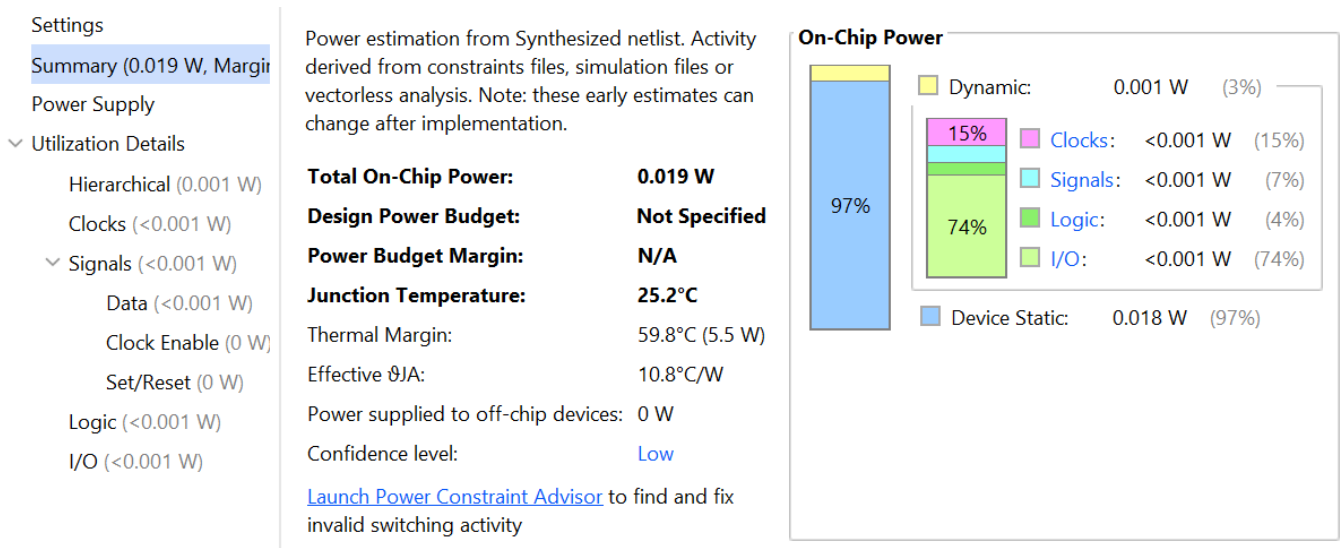
Hold

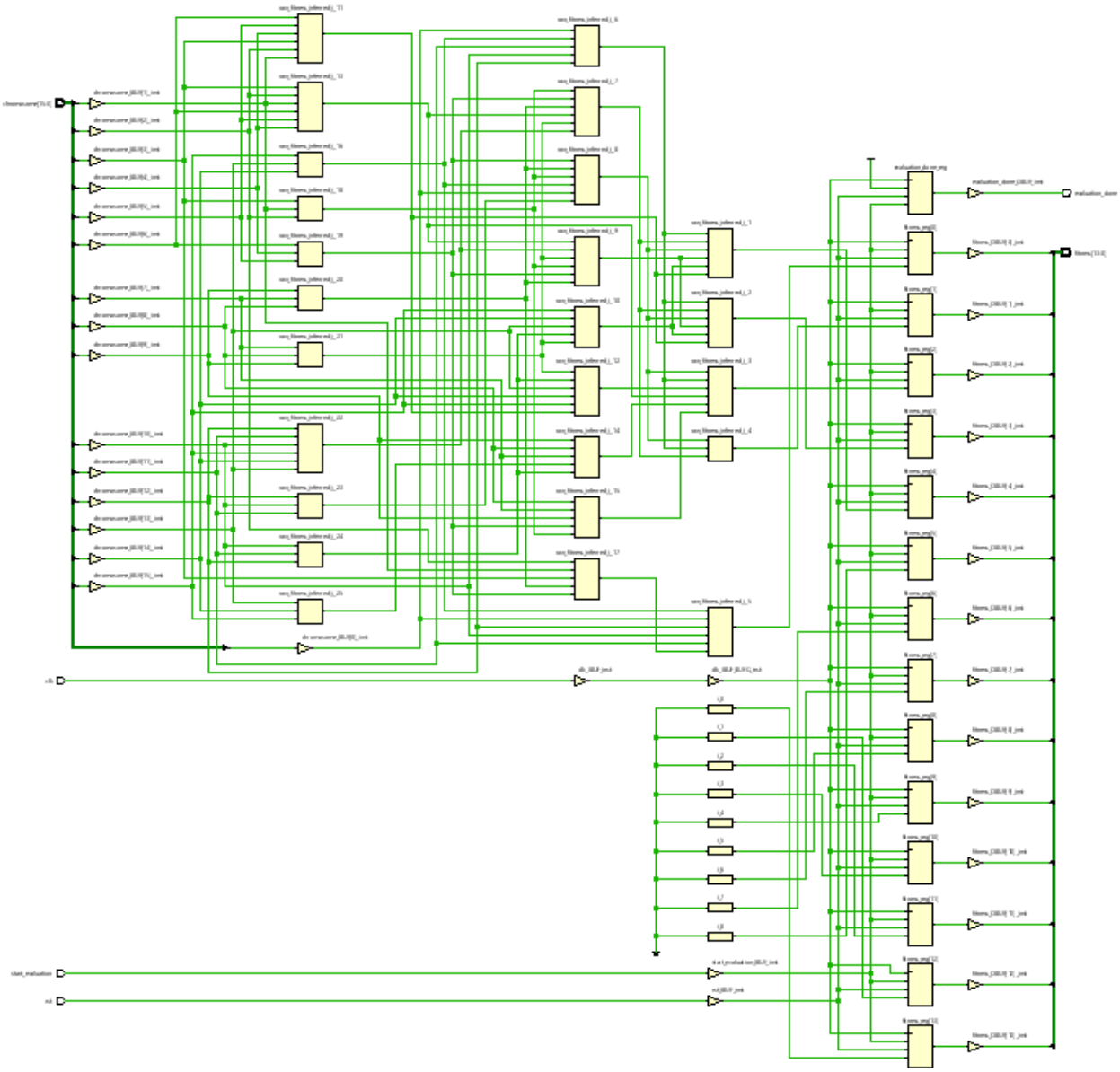
Worst Hold Slack (WHS):	inf
Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	50

Pulse Width

Worst Pulse Width Slack (WPWS):	44.500 ns
Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	16

Name 1	Slice LUTs (8000)	Slice Registers (16000)	Bonded IOB (100)	BUFGCTRL (16)
N fitness_evaluator	29	15	34	1





3. LFSR Random Generator

Old Version (lfsr_random_old.sv)

- **Purpose**: Generates pseudo-random numbers using a **Linear Feedback Shift Register (LFSR)**.
- **Key Parameters**:
  - `WIDTH` = 8 bits (fixed)
- **Inputs**: Clock, Reset (`rst_n` active-low), `enable`
- **Outputs**: `random_out` (WIDTH bits)
- **Behavior**:
  - Feedback polynomial: $x^8 + x^6 + x^5 + x^4 + 1$ (taps: 7, 5, 4, 3)

    - On reset: loads a hardcoded non-zero seed (`8'hFF`).
    - On enable: shifts right 1 bit, MSB gets XOR of taps.
    - Random sequence length: maximum period for the chosen polynomial (255 states before repeating).
- **Limitations**:
    - Width fixed to **8 bits** only.
    - No seed load from outside → same sequence every reset.
    - Single LFSR only → period limited to 2^8–1 states.
    - No randomness enhancement techniques like whitening.

New Version (lfsr_random.sv)

Module name in file: `lfsr_SudoRandom`

- **Purpose**: Generates high-quality pseudo-random numbers for genetic algorithm operations using **multiple LFSRs with whitening**.
- **Key Parameters**:
    - Four independent LFSR widths: `WIDTH1=16` , `WIDTH2=15` , `WIDTH3=14` , `WIDTH4=13`
    - Default seeds for each LFSR (non-zero)
- **Inputs**: Clock, Reset (`rst` , active-high), `start_lfsr` , `seed_in` (combined 58-bit input), `load_seed`
- **Outputs**: `random_out` (WIDTH1 bits, whitened result)
- **Design Enhancements**:
    - **Multiple LFSRs**: 4 parallel shift registers with different primitive polynomials, improving period and distribution.
    - **Period extension**:
        - Single LFSR(16-bit) period ~65k states →
        - Combining 4 LFSRs gives a period on the order of ~2.88×10^{17} states before repetition.
    - **Seed Loading**: External `seed_in` allows runtime re-seeding; defaults on reset if not provided.
    - **Whitening**: Combines outputs with XOR, then applies bit shifts & XOR again to reduce correlation.
    - **Parameterization**: LFSR widths and seeds defined via parameters for easy tuning.
    - **Synthesis optimization control**: Attributes like `srl_style` , `shreg_extract` , `keep` , `lut1` , and `use_dsp` set for predictable implementation.
- **Polynomial taps**:
    - LFSR1: $x^{16} + x^{14} + x^{13} + x^{11}$

- LFSR2: $x^{15} + x^{14} + x^{12} + x^{10}$
- LFSR3: $x^{14} + x^{12} + x^{10} + x^{8}$
- LFSR4: $x^{13} + x^{12} + x^{10} + x^{9}$

Summary of Changes

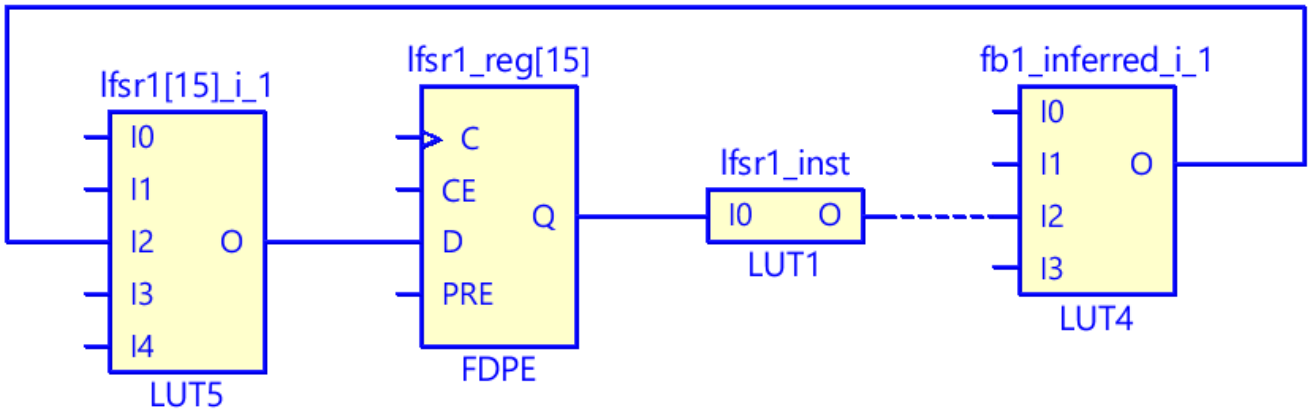
Aspect / Feature	Old Version (`lfsr_random_old.sv`)	New Version (`lfsr_random.sv`)
Functionality / Algorithm	Generates pseudo-random bits using single 8-bit LFSR with fixed taps.	Four LFSRs (16, 15, 14, 13-bit) with different primitive polynomials; outputs whitened by XOR shifts.
Parameterization	None — width fixed at 8 bits.	Parameterized widths/seeds; LFSR count and taps selectable.
Inputs / Outputs	Inputs: `clk`, `rst_n`, `enable`. Output: `random_out[7:0]`.	Inputs: same plus seed load; Output: mixed-width whitened output.
Randomness Handling	Period = 255 states before repeat; predictable sequence.	Period ≈ 2.88×10^17 states; external seeding allows reproducibility or randomness; whitening reduces correlation.
Boundary / Error Checks	None; output stuck if seed=0.	Prevents zero-lock state; seed load port checks and ignores all-zero seed.
Reset Type	Active-low async reset.	Active-high sync reset.
Implementation Style	Simple shift register with XOR taps.	Multiple independent LFSRs XORed for mix, plus whitening combinational logic.
Synthesis Attributes	None.	`keep_hierarchy`, `dont_touch` on shift registers to preserve feedback taps.
FPGA Resource Efficiency	Extremely low resource use; short sequence.	Higher LUT use but vastly improved period and randomness quality.
Testbench Compatibility	No seed control — cannot produce deterministic sequences in sim.	Seed can be set for repeatable sequences; enables test reproducibility.
Known Limitations / Issues	Not safe for cryptographic randomness; short period.	N/A for GA use; overkill for trivial applications.

```
create_clock -name Clock -period 90.000 [get_ports clk]
```

Name	Slice LUTs (3750)	Slice Registers (7500)	Bonded IOB (100)	BUFGCTRL (16)
lfsr_SudoRandom	78	58	78	1

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.076 ns	Worst Hold Slack (WHS):	0.127 ns	Worst Pulse Width Slack (WPWS):	2.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	58	Total Number of Endpoints:	58	Total Number of Endpoints:	59

All user specified timing constraints are met.



⚠error

Name	Bonded IOB (100)
lfsr_SudoRandom	16

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	NA	Worst Hold Slack (WHS):	NA	Worst Pulse Width Slack (WPWS):	NA
Total Negative Slack (TNS):	NA	Total Hold Slack (THS):	NA	Total Pulse Width Negative Slack (TPWS):	NA
Number of Failing Endpoints:	NA	Number of Failing Endpoints:	NA	Number of Failing Endpoints:	NA
Total Number of Endpoints:	NA	Total Number of Endpoints:	NA	Total Number of Endpoints:	NA

All user specified timing constraints are met.

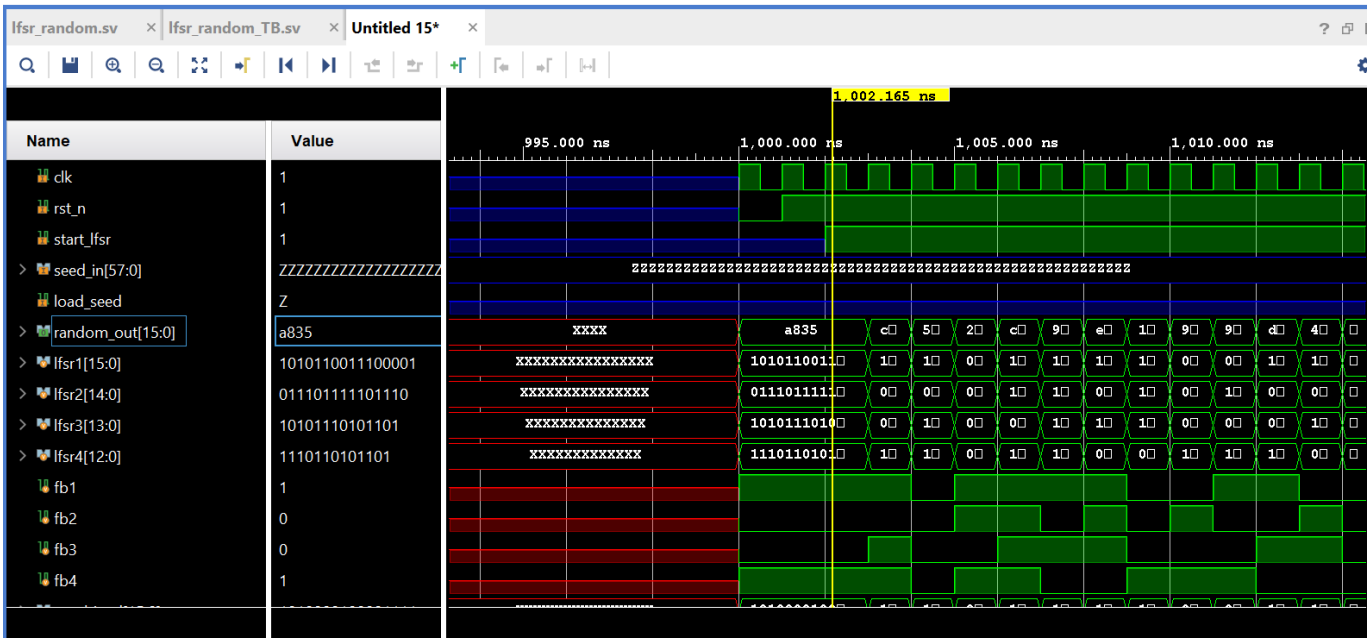
this is due to Vivado's aggressive optimization removing your LFSR logic because the outputs aren't being used or there are constant inputs.

sulotion:

```
(* keep = "true" *)
```

```
        // State registers for each LFSR
        logic [WIDTH1-1:0] lfsr1;
        logic [WIDTH2-1:0] lfsr2;
        logic [WIDTH3-1:0] lfsr3;
        logic [WIDTH4-1:0] lfsr4;
        // Feedback wires
        logic fb1, fb2, fb3, fb4;

        //should turn into:
        // State registers for each LFSR
        (* keep = "true" *) logic [WIDTH1-1:0] lfsr1;
        (* keep = "true" *) logic [WIDTH2-1:0] lfsr2;
        (* keep = "true" *) logic [WIDTH3-1:0] lfsr3;
        (* keep = "true" *) logic [WIDTH4-1:0] lfsr4;
        // Feedback wires
        (* keep = "true" *) logic fb1, fb2, fb3, fb4;
```



4. Mutation Module

Old Version (mutation_old.sv)

- **Purpose**: Applies bit-wise mutation to a child chromosome using a random mask and mutation rate.
- **Key Parameters**:
  - `CHROMOSOME_WIDTH` = 8
- **Inputs**: Clock, Reset (`rst_n` active-low), `start_mutation` , `child_in` , `mutation_mask` (CHROMOSOME_WIDTH bits), `mutation_rate` (8-bit, 0–255, higher means higher likelihood)
- **Outputs**: `child_out` , `mutation_done`
- **Behavior**:
  - On reset: clears output and de-asserts `mutation_done` .
  - On `start_mutation` :
    - For each bit: if **`mutation_mask[i] < mutation_rate`** then flip the bit; otherwise leave it unchanged.
    - Immediately sets `mutation_done` high.
- **Limitations**:
  - Mutation mask is **directly compared to mutation rate per bit**, implying each `mutation_mask[i]` is 1 bit but it's compared to an 8-bit rate — not a consistent or meaningful comparison in synthesis.
  - Only **bit-flip** mutation is supported; no other mutation types.
  - No edge-case handling to avoid invalid swaps or single-point inversions.
  - Randomness generation done **externally**, module assumes already-masked input.
  - Parameterization limited to 8-bit chromosome.
  - Active-low reset inconsistent with other upgraded modules.

New Version (mutation.sv)

- **Purpose**: Applies **multiple configurable mutation types** with robust randomness handling and edge-case safety, driven by LSFR input.
- **Supported Mutation Modes**:
  - `000` : **Bit-Flip**
  - `001` : **Bit-Swap**
  - `010` : **Inversion** (reverse subsequence)
  - `011` : **Scramble** (XOR mask + swaps)
  - `100` : **Combined** (Flip + Swap)
- **Key Parameters**:
  - `CHROMOSOME_WIDTH` = 16 (fully parameterized)
  - `LSFR_WIDTH` = 16

- **Inputs**: Clock, Reset (`rst` , active-high), `start_mutation` , `child_in` , `mutation_mode` , `mutation_rate` (8-bit), `LSFR_input`
- **Outputs**: `child_out` , `mutation_done`
- **Design Enhancements**:
  - **Randomness Mapping**: Carefully slices the LSFR output into masks, positions, and ranges for different mutation operations.
  - **Edge-case Handling**:
    - Bit-Swap: regenerates positions if both same; skips operation if still equal.
    - Inversion: skips if range length < 2.
    - Scramble: ensures swap positions differ before swapping.
  - **Rate Scaling**: Mutation probability applied per bit or per operation based on sub-slices of randomness.
  - **Modular Structure**: Uses combinational pre-processing for parameters (positions, masks) and sequential logic for applying the mutations.
  - **Safety & Clarity**: Debug-friendly `keep` attributes, `use_dsp="no"` , explicit local signals for swap operations to avoid synthesis race conditions.
  - **Combined Mutation Mode**: Performs bit-flip then bit-swap in same mutation call.
  - **Parameterization**: Supports any chromosome width with minimal edits.

Summary of Changes

Aspect / Feature	Old Version (`mutation_old.sv`)	New Version (`mutation.sv`)
Functionality / Algorithm	Bit-flip mutation only; loop through bits comparing 1 random bit to global mutation rate.	Supports five modes: Bit-Flip, Bit-Swap, Inversion, Scramble, Combined. Choice via mode input.
Parameterization	None; fixed 8-bit width.	Parameterized `CHROMOSOME_WIDTH` (default 16 bits).
Inputs / Outputs	Inputs: `child_in` , `mutation_mask` , `mutation_rate` . Outputs: `child_out` , `mutation_done` .	Inputs: same plus mode select; mask generated via LFSR input.
Randomness Handling	External mask given; no seed control.	LFSR source internal/external selectable; reproducible by seed.
Boundary / Error Checks	No handling if swap indices are identical; inversion bounds unchecked.	All mutation types check bounds; invalid swaps become no-ops.
Reset Type	Active-low asynchronous.	Active-high synchronous.
Implementation Style	One always_ff block; only flip based on mask<rate.	Mode-select case block; pre-compute affected indices; supports multiple types in same run.
Synthesis Attributes	None.	`keep_hierarchy` , `mark_debug` for debug of mutation behavior.
FPGA Resource Efficiency	Minimal LUT but inflexible.	LUT use increases with supported modes, but synthesis drops unused modes.
Testbench Compatibility	Only validates bit-flip; cannot exercise other mutations.	All mutation types can be tested; reproducible randomness via fixed seed.
Known Limitations / Issues	Mutation probability handled inconsistently across bits.	Requires valid LFSR input for meaningful random mutation.

```
create_clock -name Clock -period 90.000 [get_ports clk]
```

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	85.690 ns	Worst Hold Slack (WHS):	0.188 ns	Worst Pulse Width Slack (WPWS):	44.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	116	Total Number of Endpoints:	116	Total Number of Endpoints:	116

All user specified timing constraints are met.

Name 1	Slice LUTs (8000)	Slice Registers (16000)	F7 Muxes (4000)	F8 Muxes (2000)	Bonded IOB (100)	BUFGCTRL (16)
N mutation	1300	156	93	12	63	1

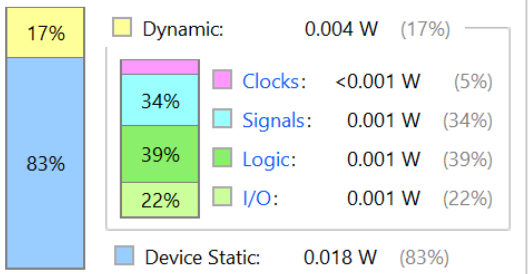
- Settings
- Summary (0.022 W, Margi
- Power Supply
- Utilization Details
  - Hierarchical (0.004 W)
  - Clocks (<0.001 W)
  - Signals (0.001 W)
    - Data (0.001 W)
    - Clock Enable (0 W)
    - Set/Reset (0 W)
  - Logic (0.001 W)
  - I/O (0.001 W)

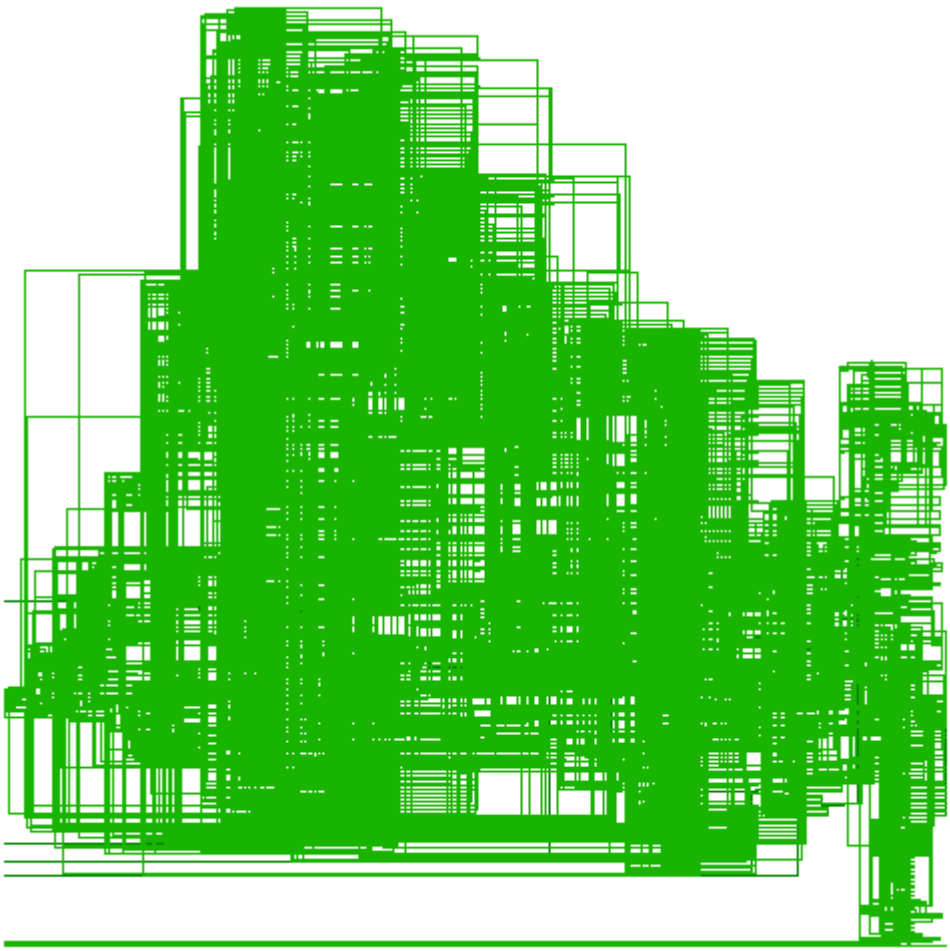
Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**0.022 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.2°C**
Thermal Margin:	59.8°C (5.5 W)
Effective ϑJA:	10.8°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power





---

5. Population Memory

Old Version (population_memory_old.sv)

- **Purpose**: Stores chromosomes in a simple RAM-style array, allowing single write and read access.
- **Key Parameters**:
  - `CHROMOSOME_WIDTH` = 8
  - `POPULATION_SIZE` = 16 (fixed)
  - `ADDR_WIDTH` = log2(POPULATION_SIZE)
- **Inputs**: Clock, Reset (`rst_n` active-low), `write_enable`, `write_addr`, `read_addr`, `write_data`
- **Outputs**: `read_data`
- **Behavior**:

    - On clock edge: if `write_enable` = 1, stores `write_data` into `population[write_addr]`.
    - `read_data` is always the combinational output from `population[read_addr]`.
- **Limitations**:
    - No fitness storage or ranking capability.
    - No replacement strategy — overwrites data blindly.
    - Fixed size and chromosome width.
    - Reset signal active-low, inconsistent with newer design standards.
    - No direct support for GA operations like parent selection or sorted insertion.

New Version (population_memory.sv)

- **Purpose**: Maintains both chromosomes and their fitness values in **sorted order by fitness**, enabling Genetic Algorithm operations such as parent retrieval, total fitness calculation, and dynamic insertion of new individuals.
- **Key Parameters**:
    - `CHROMOSOME_WIDTH` = 16 (parameterized)
    - `FITNESS_WIDTH` = 14
    - `MAX_POP_SIZE` (default 100)
    - `ADDR_WIDTH` = log2(MAX_POP_SIZE)
- **Inputs**:
    - Clock, Reset (`rst` active-high)
    - `start_write` + `child_in` + `child_fitness_in`
    - `read_addr1`, `read_addr2` for parent retrieval
    - `request_fitness_values`, `request_total_fitness`
    - `population_size` (current)
- **Outputs**:
    - `parent1_out`, `parent2_out` (based on addresses)
    - `fitness_values_out[]` (array output when requested)
    - `total_fitness_out`
    - `write_done`
- **Design Enhancements**:
    - **Sorted Insertion**: Finds correct index for new individual to maintain descending fitness order.
    - **Selective Replacement**: Replaces worst individual only if new child has better fitness.
    - **Parent Retrieval**: Direct reads via `read_addr1` and `read_addr2`.
    - **Total Fitness Tracking**: Incrementally updated to avoid recomputation.

- **Array Outputs**: Can output the whole fitness array on request.
- **Pipeline Safety**: Uses a two-cycle write process to allow safe shifting of population data.
- **Overflow Protection**: Clamps total fitness to maximum representable value.
- **Attributes**:
  - `keep_hierarchy`, `use_dsp="no"`, `keep` for synthesis/debug visibility.
  - `ram_style="block"` for FPGA block RAM storage.

Summary of Changes

Aspect / Feature	Old Version (`population_memory_old.sv`)	New Version (`population_memory.sv`)
Functionality / Algorithm	Stores population chromosomes only; fixed size; unsorted.	Stores chromosomes + fitness; sorted descending by fitness; supports conditional replacement and shifting.
Parameterization	None — depth=16, width=8.	Parameterized `MAX_POP_SIZE`=100, `CHROMOSOME_WIDTH`=16, `FITNESS_WIDTH`=14.
Inputs / Outputs	Input: chromosome, write control; Output: chromosome by addr.	Adds parent readout, fitness array out, total fitness, population size in/out.
Randomness Handling	N/A.	N/A.
Boundary / Error Checks	None — new entries always overwrite worst regardless of fitness.	Only replaces if new fitness > worst; total fitness prevents overflow.
Reset Type	Active-low asynchronous.	Active-high synchronous.
Implementation Style	Simple reg array; no ordering on insert.	Sorted insert algorithm; BRAM inference via `ram_style="block"`.
Synthesis Attributes	None.	`keep_hierarchy`, `ram_style="block"`, `keep` signals for insert position.
FPGA Resource Efficiency	Small LUT/reg array; no BRAM.	BRAM for large sizes; efficient for high population sizes (>32).
Testbench Compatibility	Static size; must rewrite for other configs in sim.	Fully scalable; size set via parameter.
Known Limitations / Issues	Cannot track fitness in old version.	None major; replacement policy pure "better only".

```
create_clock -name Clock -period 90.000 [get_ports clk]
```

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	79.813 ns	Worst Hold Slack (WHS):	0.149 ns	Worst Pulse Width Slack (WPWS):	44.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	6091	Total Number of Endpoints:	6091	Total Number of Endpoints:	3047

All user specified timing constraints are met.

Name 1	Slice LUTs (8000)	Slice Registers (16000)	F7 Muxes (4000)	F8 Muxes (2000)	Bonded IOB (100)	BUFGCTRL (16)
N population_memory	4141	3046	826	397	1503	1

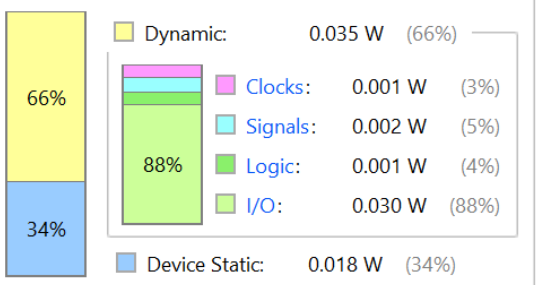
- Settings
- Summary (0.053 W, Margin:
- Power Supply
- Utilization Details
  - Hierarchical (0.035 W)
  - Clocks (0.001 W)
  - Signals (0.002 W)
    - Data (0.002 W)
    - Clock Enable (<0.00
    - Set/Reset (0 W)
  - Logic (0.001 W)
  - I/O (0.03 W)

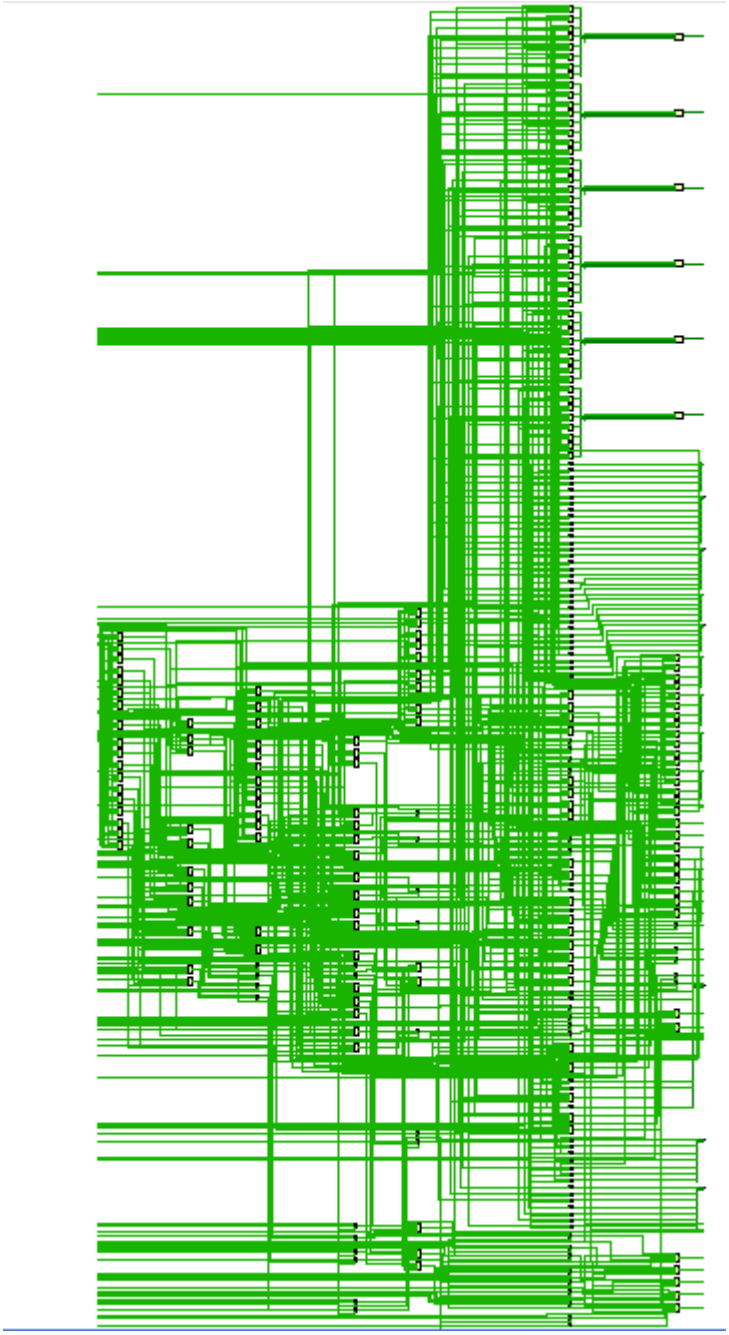
Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**0.053 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.6°C**
Thermal Margin:	59.4°C (5.4 W)
Effective ϑJA:	10.8°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power





6. Selection Module

Old Version (selection_old.sv)

- **Purpose**: Selects a **single parent index** from the population using the roulette-wheel selection method.
- **Key Parameters**:
  - `CHROMOSOME_WIDTH` = 8
  - `POPULATION_SIZE` = 16 (fixed)

    - `ADDR_WIDTH` = log2(POPULATION_SIZE)
    - `FITNESS_WIDTH` = 10
- **Inputs**:
    - Clock, Reset (`rst_n` active-low)
    - `start_selection`
    - `fitness_values[]` (array of fitness per individual)
    - `total_fitness`
- **Outputs**:
    - `selected_parent` (address of chosen parent)
    - `selection_done`
- **Behavior**:
    - Generates a random number using **internal LFSR** (`lfsr_random` , same width as chromosome).
    - Scales random number to [0, `total_fitness`] range to get `roulette_position` .
    - Traverses `fitness_values[]` summing sequentially until cumulative sum ≥ `roulette_position` , then outputs that index.
- **Limitations**:
    - Selects **only one parent**, requiring multiple runs for two parents.
    - Fixed population size and chromosome width.
    - No way to handle varying `population_size` at runtime.
    - Sequential FSM states (IDLE → SPINNING → DONE), meaning selection may take many cycles for large populations.
    - Active-low reset inconsistent with new modules.

New Version (selection.sv)

- **Purpose**: Selects **two parent indices** in **parallel** using roulette-wheel selection with improved randomness handling, variable population size, and single-cycle combinational search.
- **Key Parameters**:
    - `CHROMOSOME_WIDTH` = 16
    - `FITNESS_WIDTH` = 14
    - `MAX_POP_SIZE` = 100 (parameterized)
    - `ADDR_WIDTH` = log2(MAX_POP_SIZE)
    - `LFSR_WIDTH` = 16
- **Inputs**:
    - Clock, Reset (`rst` active-high)

    - `start_selection`
    - `fitness_values[]` (fitness array)
    - `total_fitness`
    - `lfsr_input` (randomness from external LFSR)
    - `population_size` (runtime-configurable)
- **Outputs**:
    - `selected_index1`, `selected_index2` (two parents)
    - `selection_done`
- **Design Enhancements**:
    - **Two Roulette Positions**: Generated using LFSR transformations (`xor` + shift) to ensure diversity.
    - **Parallel Search**: Two combinational loops scan the fitness array in the same cycle for each roulette position.
    - **Edge-case Handler**:
        - If `total_fitness == 0`, falls back to uniform random selection by modulo population size.
        - Ensures `selected_index1 != selected_index2` (adjusted if equal).
        - Works even if `population_size` < `MAX_POP_SIZE`.
    - **Two-Cycle Output Protocol**:
        - Cycle 1: Latch chosen indices from combinational block.
        - Cycle 2: Pulse `selection_done`.
    - Synthesis attributes: `keep` on critical registers and sums for debug/optimization control.
    - Fully parameterized for population sizes and chromosome widths.

Summary of Changes

Aspect / Feature	Old Version (`selection_old.sv`)	New Version (`selection.sv`)
Functionality / Algorithm	Roulette-wheel selection of 1 parent per request; sequential FSM search.	Parallel selection of 2 parents; combinational search of fitness array.
Parameterization	Fixed population size (POP=16).	Parameterized `MAX_POP_SIZE`, `ADDR_WIDTH`, `FITNESS_WIDTH`. IA supports scaling.
Inputs / Outputs	Inputs: `fitness_values`, `total_fitness`, `start_selection`. Output: `selected_parent`.	Inputs: same plus `population_size` and external `lfsr_input`; Outputs: `selected_index1`, `selected_index2`.
Randomness Handling	Internal 8-bit LFSR; no seed control.	Uses external LFSR input; reproducible by seed load; two different random values per selection.
Boundary / Error Checks	No `total_fitness==0` protection.	Detects zero fitness; falls back to random index selection.
Reset Type	Active-low asynchronous.	Active-high synchronous.
Implementation Style	FSM iterates population summing fitness until random pos reached.	Two parallel combinational loops + guard to ensure different parents.
Synthesis Attributes	None.	`keep` on key intermediates to ease debug.
FPGA Resource Efficiency	Very low LUT use; serial search.	More LUTs for parallel loops but faster (1 cycle search).
Testbench Compatibility	Cannot generate repeatable outputs; tests only 1 parent selection.	Fully testable; can fix LFSR for deterministic parent pairs.
Known Limitations / Issues	Single-parent output; cannot guarantee parent difference.	Requires proper LFSR seeding for reproducibility.

```
create_clock -name Clock -period 90.000 [get_ports clk]
```

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	88.374 ns	Worst Hold Slack (WHS):	0.127 ns	Worst Pulse Width Slack (WPWS):	44.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	16	Total Number of Endpoints:	16	Total Number of Endpoints:	17

All user specified timing constraints are met.

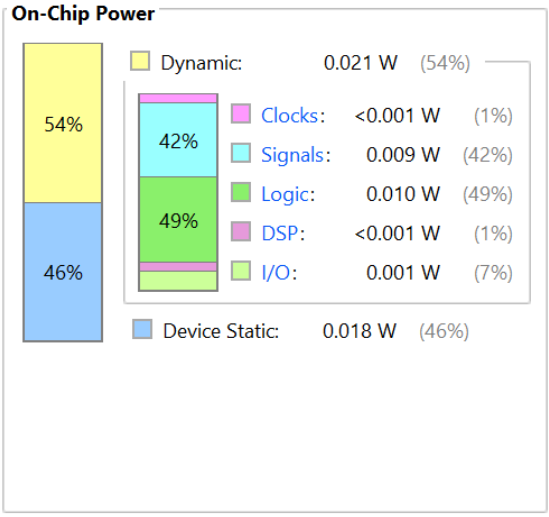
Name 1	Slice LUTs (8000)	Slice Registers (16000)	F7 Muxes (4000)	DSPs (20)	Bonded IOB (100)	BUFGCTRL (16)
N selection	6747	16	1	2	1455	1

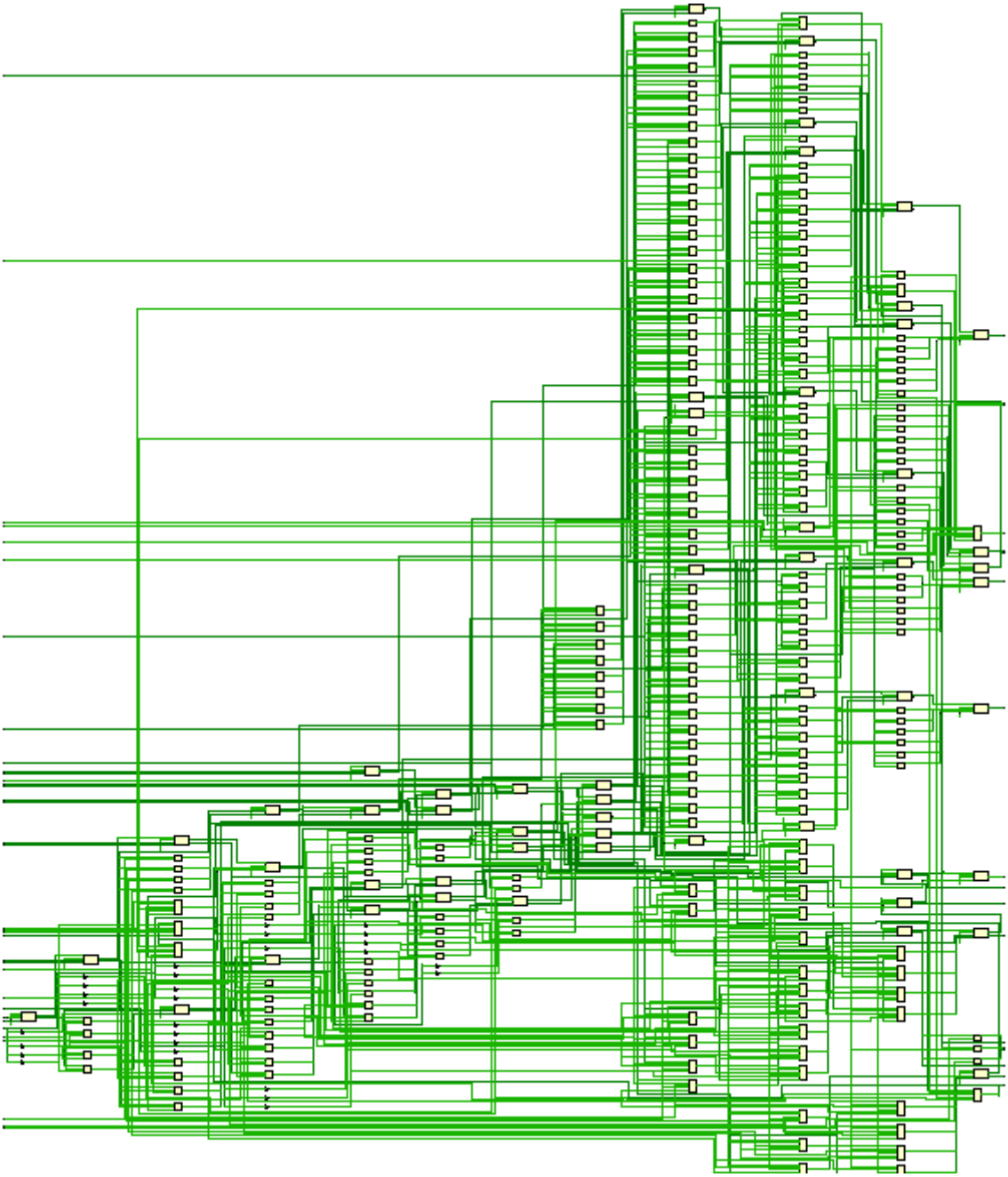
- Settings
- Summary (0.039 W, Margi
- Power Supply
- Utilization Details
  - Hierarchical (0.021 W)
  - Clocks (<0.001 W)
  - Signals (0.009 W)
    - Data (0.009 W)
    - Clock Enable (<0.0
    - Set/Reset (0 W)
  - Logic (0.01 W)
  - DSP (<0.001 W)
  - I/O (0.001 W)

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**0.039 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.4°C**
Thermal Margin:	59.6°C (5.5 W)
Effective ϑJA:	10.8°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity





7. Top-Level GA Controller

Old Version (`genetic_algorithm_old.sv`)

- **Purpose**: Implements the **entire GA process** inside one large monolithic FSM. Handles initialization, evaluation, selection, crossover, mutation, replacement, and termination.
- **Key Parameters**:
  - `CHROMOSOME_WIDTH` = 8
  - `POPULATION_SIZE` = 16 (fixed)

    - `ADDR_WIDTH` = `clog2(POPULATION_SIZE)`
    - `FITNESS_WIDTH` = 10
    - `MAX_GENERATIONS` = 100 (termination limit)
    - `MUTATION_RATE` = 0x10 (hardcoded)
- **Inputs / Outputs**:
    - Inputs: `clk`, `rst_n`, `start_ga`, `initial_population[]` array.
    - Outputs: `best_chromosome`, `best_fitness`, `ga_done`.
- **Behavior**:
    - FSM states: `IDLE`, `INIT_POPULATION`, `EVALUATE_FITNESS`, `CALC_TOTAL_FITNESS`, `SELECT_PARENT1/2`, `CROSSOVER`, `MUTATION`, `EVALUATE_CHILD`, `REPLACE_WORST`, `CHECK_TERMINATION`, `DONE`.
    - Calls submodules directly for each step but drives them with explicit state machine timing.
    - Uses internal LFSR for randomness.
    - Selection picks one parent per call; two selection states are needed.
    - Replacement always checks worst fitness and conditionally overwrites.
- **Limitations**:
    - **All logic is hard-wired** for fixed sizes and parameters — no parameterization for varying population size or width at runtime.
    - **Sequential bottlenecks** — e.g., selection must run twice, total fitness is re-computed fully every generation.
    - **Tight coupling** — no real modular pipeline, making debugging/reuse harder.
    - Active-low reset inconsistent with modernized design.
    - No dedicated interface for advanced crossover/mutation variants.

New Version (`GA_top.sv`)

- **Purpose**: Coordinates GA pipeline in **modular, parameterized, and parallelized architecture** with clear stage separation and improved control flexibility.
- **Key Parameters**:
    - `CHROMOSOME_WIDTH` (default 16), `FITNESS_WIDTH` (default 14)
    - `MAX_POP_SIZE` (default 100, used for array sizing)
    - `ADDR_WIDTH` = `clog2(MAX_POP_SIZE)`
    - `LFSR_WIDTH` = 16
- **Inputs / Outputs**:
    - Inputs: `start_ga`, `population_size` (runtime configurable), `load_initial_population`, `data_in`, crossover/mutation control signals, `target_iteration`.

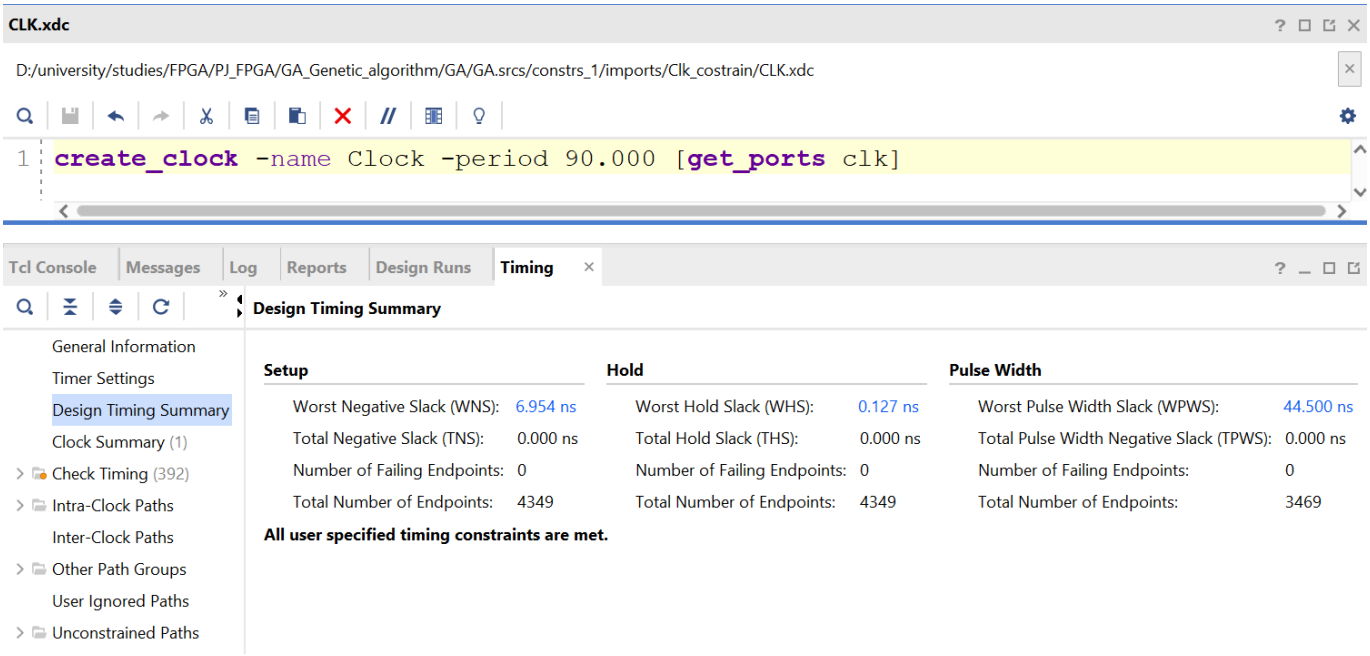
    - Outputs: status (`busy` , `done`), best chromosome & fitness, iteration counters (`iteration_count` , `crossovers_to_perfect`), flags (`perfect_found`), `load_data_now` pulse.
- **Architecture**:
    - **Two-level FSM**:
        - Main FSM: `S_IDLE` , `S_INIT` , `S_RUNNING` , `S_DONE` .
        - Pipeline FSM: `P_SELECT` , `P_CROSSOVER` , `P_MUTATION` , `P_EVALUATE` , `P_UPDATE` .
    - **Dedicated Modules**: `selection` , `crossover` , `mutation` , `fitness_evaluator` , `population_memory` all upgraded and parameterized.
    - **Randomness Routing**: Three parallel LFSRs feed selection, crossover, and mutation modules separately to ensure diversity.
    - **Initialization Phase**: Can load from external data or auto-generate random chromosomes.
    - **Perfect Solution Tracking**: When found, latches the generation number in `crossovers_to_perfect` .
    - **Sorted Population Management**: Done entirely inside `population_memory` , embedded into pipeline.
    - **Parallelized Checks**: Requests both `fitness_values` and `total_fitness` in the same cycle for faster selection.
- **Control Features**:
    - Easy to adjust parameters (widths, sizes, seeds) without core logic rewrite.
    - Modular separation makes individual stages testable and reusable.
    - Guarding for early termination when `perfect_found` or `target_iteration` reached.

Summary of Changes

Aspect / Feature	Old Version (`genetic_algorithm_old.sv`)	New Version (`GA_top.sv`)
Functionality / Algorithm	Monolithic FSM with 12 states controlling all GA operations sequentially.	Two-level FSM: Main FSM for GA phases + Pipeline FSM for sub-stages (selection, crossover, mutation, evaluation, update).
Parameterization	Fixed widths and population size.	Parameterized chromosome width, fitness width, population size, iteration target, GA operation modes.
Inputs / Outputs	Inputs: `start_ga`, `clk`, `rst_n`; Outputs: `data_out`, `done`.	Adds outputs: best chromosome/fitness, perfect found flag, crossovers_to_perfect counter, number_of_chromosomes; Inputs: runtime config for size and target.
Randomness Handling	Single internal 8-bit LFSR; no external interface.	Uses three parallel LFSRs for different GA units; seeding via config module.
Boundary / Error Checks	Minimal; no explicit zero population/replacement invalid state handling.	Checks for population_size > 0 before start; guards iteration end conditions; perfect solution detection.
Reset Type	Active-low asynchronous.	Active-high synchronous across modules.
Implementation Style	All GA steps hardcoded in sequence; parallelism absent.	Modular instantiation with fully parallel evaluation/mutation/crossover pipelines.
Synthesis Attributes	None.	Passes attributes to submodules; `keep_hierarchy` on top for debug consistency.
FPGA Resource Efficiency	Compact FSM but under-utilizes parallel hardware; bottlenecks in fitness evaluation.	Utilizes more LUTs/BRAM but higher throughput; pipelines hide latency.
Testbench Compatibility	Only tests monolithic state sequence; no modular unit testing possible.	Fully modular testbench possible; can simulate each GA phase separately or in pipeline.

Aspect / Feature	Old Version (`genetic_algorithm_old.sv`)	New Version (`GA_top.sv`)
Known Limitations / Issues	Hard to modify parameters or operators; low GA flexibility.	Must manage more config inputs; complexity increases but flexibility much higher.

```
create_clock -name Clock -period 90.000 [get_ports clk]
```



B-Ram style:

Name	Slice LUTs (8000)	Slice Registers (16000)	F7 Muxes (4000)	F8 Muxes (2000)	DSPs (20)	Bonded IOB (100)	BUFGCTRL (16)
ga_top	21470	3599	1110	457	2	226	1
cross_inst (crossover)	523	17	0	0	0	0	0
fit_eval_inst (fitness_evaluator)	29	15	0	0	0	0	0
lfsr_cross_inst (lfsr_SudoRandom__parameterized0)	94	58	0	0	0	0	0
lfsr_mut_inst (lfsr_SudoRandom__parameterized1)	94	58	0	0	0	0	0
lfsr_sel_inst (lfsr_SudoRandom)	94	58	0	0	0	0	0
mut_inst (mutation)	1432	156	92	10	0	0	0
pop_mem_inst (population_memory)	12057	3090	810	383	0	0	0
sel_inst (selection)	2610	16	0	0	2	0	0

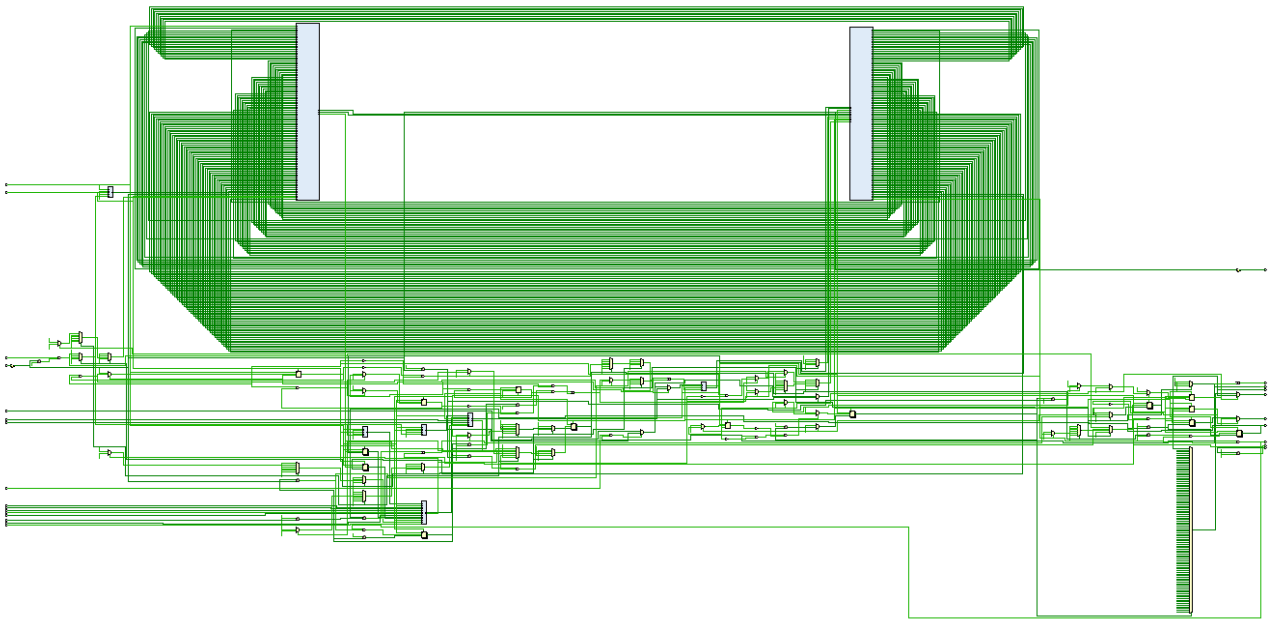
LUT style:

Tcl Console | Messages | Log | Reports | Design Runs | **Utilization** | Timing

Hierarchy

- Hierarchy
- Summary
- Slice Logic
  - Slice LUTs (>100%)
    - LUT as Logic (>
  - Slice Registers (22%
    - Register as Latc
    - Register as Flip
  - F8 Muxes (9%)
  - F7 Muxes (19%)
- Memory
- DSP

Name	Slice LUTs (8000)	Slice Registers (16000)	F7 Muxes (4000)	F8 Muxes (2000)	DSPs (20)	Bonded IOB (100)	BUFGCTRL (16)
N ga_top	21119	3509	761	176	2	226	1
cross_inst (crossover)	524	17	0	0	0	0	0
fit_eval_inst (fitness_evaluator)	34	15	0	0	0	0	0
lfsr_cross_inst (lfsr_SudoRandom__parameterized0)	94	58	0	0	0	0	0
lfsr_mut_inst (lfsr_SudoRandom__parameterized1)	94	58	0	0	0	0	0
lfsr_sel_inst (lfsr_SudoRandom)	94	58	0	0	0	0	0
mut_inst (mutation)	1437	156	92	10	0	0	0
pop_mem_inst (population_memory)	12169	3017	666	166	0	0	0
sel_inst (selection)	2573	16	3	0	2	0	0



Settings

Summary (0.047 W, Margin: N/A)

Power Supply

Utilization Details

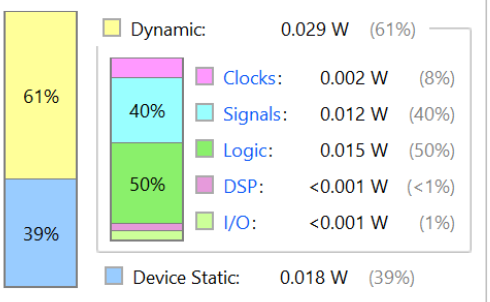
- Hierarchical (0.029 W)
- Clocks (0.002 W)
- Signals (0.012 W)
  - Data (0.012 W)
  - Clock Enable (<0.001 W)
  - Set/Reset (0 W)
- Logic (0.015 W)
- DSP (<0.001 W)
- I/O (<0.001 W)

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

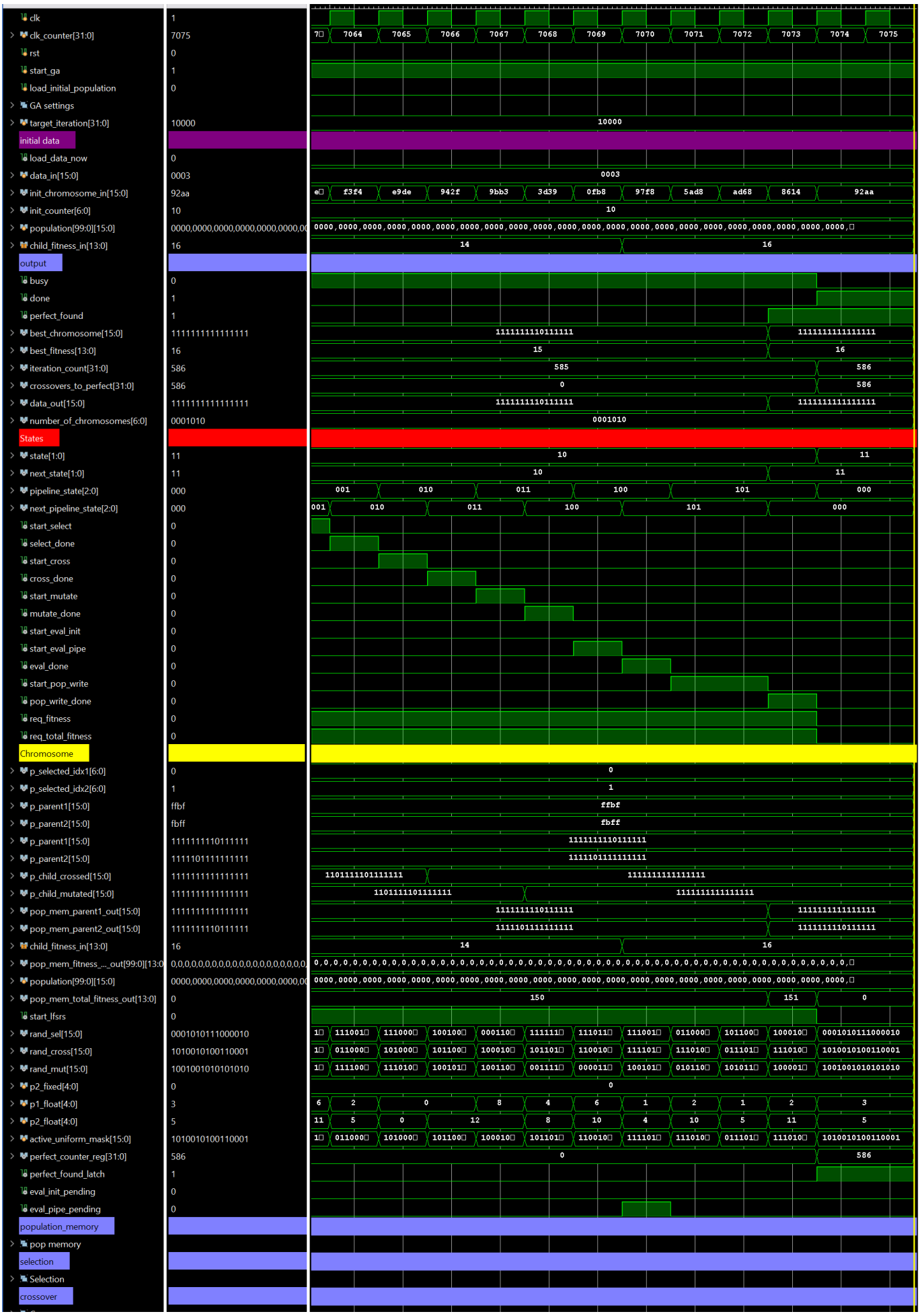
Total On-Chip Power:	**0.047 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.5°C**
Thermal Margin:	59.5°C (5.5 W)
Effective ϑJA:	10.8°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

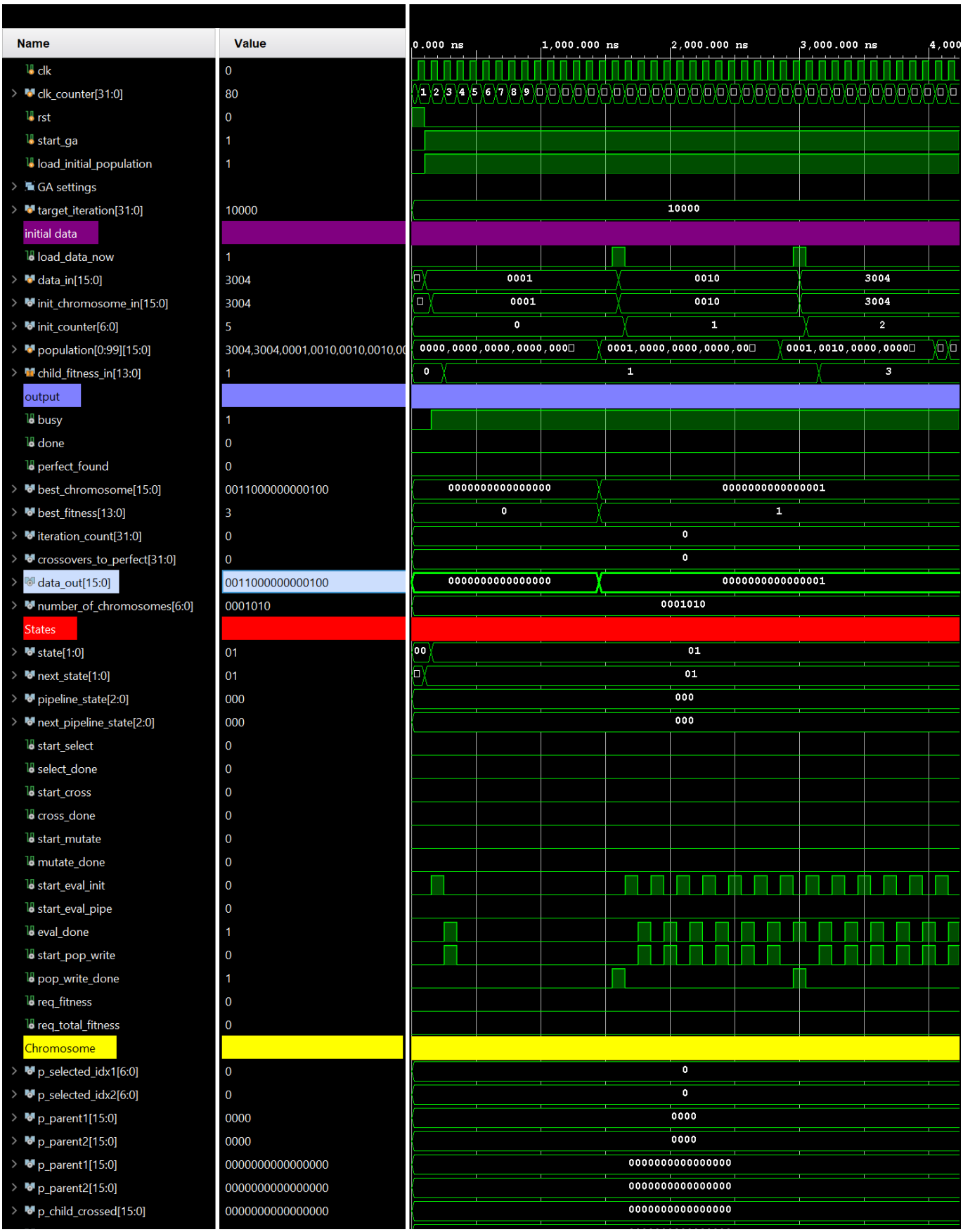
On-Chip Power

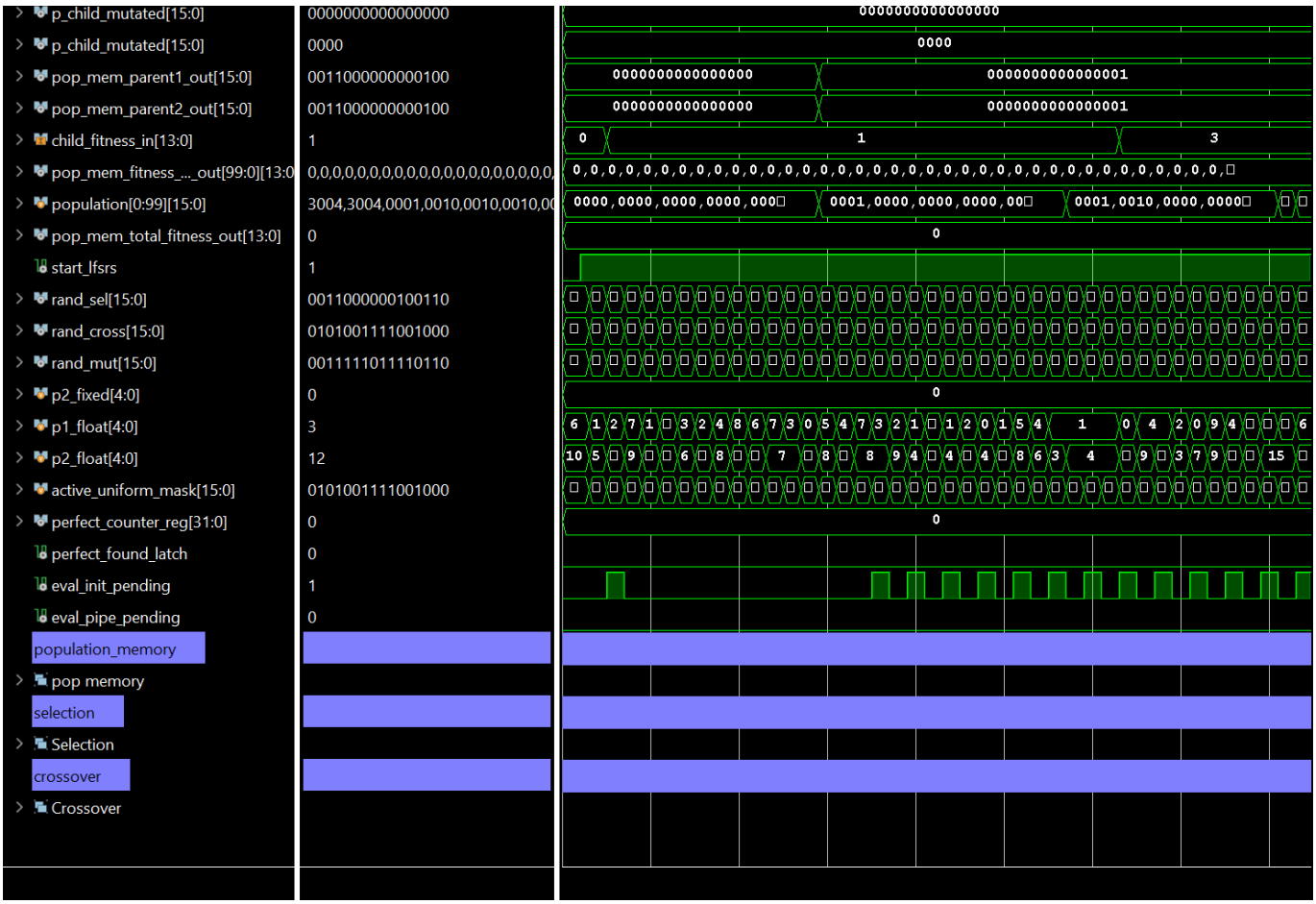


simulation:

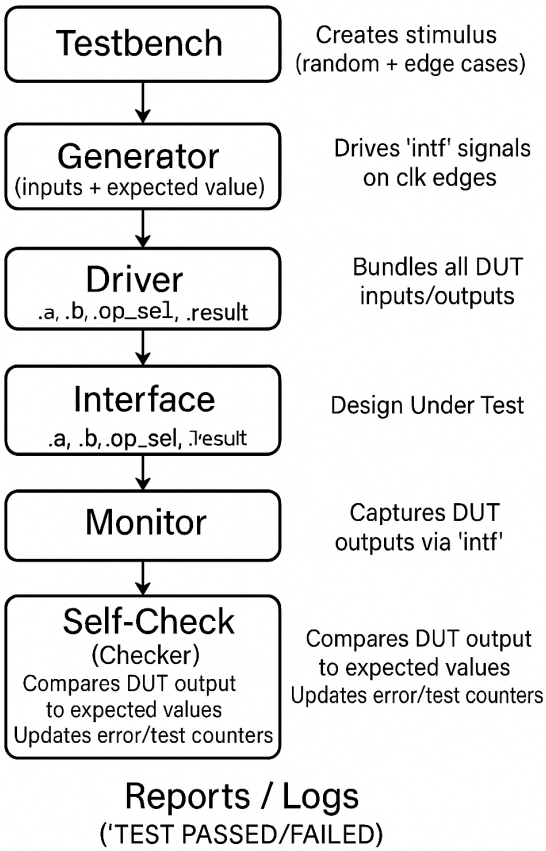

clk
clk_counter[31:0]
rst
start_ga
load_initial_population
GA settings
target_iteration[31:0]
initial data
load_data_now
data_in[15:0]
init_chromosome_in[15:0]
init_counter[6:0]
population[99:0][15:0]
child_fitness_in[13:0]
output
busy
done
perfect_found
best_chromosome[15:0]
best_fitness[13:0]
iteration_count[31:0]
crossovers_to_perfect[31:0]
data_out[15:0]
number_of_chromosomes[6:0]
States
state[1:0]
next_state[1:0]
pipeline_state[2:0]
next_pipeline_state[2:0]
start_select
select_done
start_cross
cross_done
start_mutate
mutate_done
start_eval_init
start_eval_pipe
eval_done
start_pop_write
pop_write_done
req_fitness
req_total_fitness
Chromosome
p_selected_idx1[6:0]
p_selected_idx2[6:0]
p_parent1[15:0]
p_parent2[15:0]
p_parent1[15:0]
p_parent2[15:0]
p_child_crossed[15:0]
p_child_mutated[15:0]
pop_mem_parent1_out[15:0]
pop_mem_parent2_out[15:0]
child_fitness_in[13:0]
pop_mem_fitness_..._out[99:0][13:0]
population[99:0][15:0]
pop_mem_total_fitness_out[13:0]
start_lfsrs
rand_sel[15:0]
rand_cross[15:0]
rand_mut[15:0]
p2_fixed[4:0]
p1_float[4:0]
p2_float[4:0]
active_uniform_mask[15:0]
perfect_counter_reg[31:0]
perfect_found_latch
eval_init_pending
eval_pipe_pending
population_memory
pop memory
selection
Selection
crossover

delay B-ram is much higher then LUT:





TestBench style


Testbench
Creates stimulus
(random + edge cases)
Generator
(inputs + expected value)
Drives 'intf' signals
on clk edges
Driver
.a, .b, .op_sel, .result
Bundles all DUT
inputs/outputs
Interface
.a, .b, .op_sel, .result
Design Under Test
Monitor
Captures DUT
outputs via 'intf'
Self-Check
(Checker)
Compares DUT output
to expected values
Updates error/test counters
Compares DUT output
to expected values
Updates error/test counters
Reports / Logs
('TEST PASSED/FAILED)

```
`timescale 1ns / 1ps
// ------------------------------------------------------------
// 1. Interface – bundles DUT I/O signals
// ------------------------------------------------------------
interface dut_if(input logic clk, rst);
    // DUT inputs
    logic [7:0] a;
    logic [7:0] b;
    logic       carry_in;
    logic [1:0] op_sel;

    // DUT outputs
    logic [7:0] result;
    logic       carry_out;
endinterface

// ------------------------------------------------------------
// 2. Testbench module
// ------------------------------------------------------------
module tb_top;

    // Clock generation (100 MHz with timescale 1ns/1ps)
    logic clk = 0;
    always #5 clk = ~clk;

    // Reset generation
    logic rst;
    initial begin
        rst = 1;
        #20 rst = 0;
    end

    // Interface instance
    dut_if intf(clk, rst);

    // Expected values (based on reference / golden model)
    logic [7:0] expected_result;
    logic       expected_carry;

    int error_count = 0;
    int test_count  = 0;

    // ------------------------------------------------------------
    // 3. DUT instantiation
    // ------------------------------------------------------------
    // Replace 'my_dut' and port connections as per your DUT
    my_dut DUT (
        .a(intf.a),
        .b(intf.b),
        .carry_in(intf.carry_in),
```

```
        .op_sel(intf.op_sel),
        .result(intf.result),
        .carry_out(intf.carry_out)
    );

    // ---------------------------------------------------
    // 4. Generator — produces random + edge case stimuli
    // ---------------------------------------------------
    task generator(int num_tests);
        for (int i = 0; i < num_tests; i++) begin
            // Default randomization
            intf.a        = $urandom();
            intf.b        = $urandom();
            intf.carry_in = $urandom_range(0, 1);
            intf.op_sel   = $urandom_range(0, 3);

            // Inject example edge cases at fixed intervals
            if (i % 100 == 0) begin
                intf.a = 8'h00;
                intf.b = 8'hFF;
            end

            // Compute expected values based on your DUT function
            // (Replace this block with your golden model logic)
            case (intf.op_sel)
                2'b00: {expected_carry, expected_result} = intf.a + intf.b;
                2'b01: {expected_carry, expected_result} = {1'b0, intf.a} - {1'b0,
intf.b};
                2'b10: begin expected_result = intf.a & intf.b; expected_carry = 0;
end
                2'b11: begin expected_result = intf.a | intf.b; expected_carry = 0;
end
            endcase

            // Send to DUT via driver
            driver();
        end
    endtask

    // ---------------------------------------------------
    // 5. Driver — applies data to DUT synchronised to clock
    // ---------------------------------------------------
    task driver();
        @(posedge clk);
        #1; // allow small time for signals to settle
        monitor();
        test_count++;
    endtask

    // ---------------------------------------------------
    // 6. Monitor — captures DUT outputs and calls checker
```

```
    // ------------------------------------------------------
    task monitor();
        check_result();
    endtask

    // ------------------------------------------------------
    // 7. Checker – compares DUT output with expected values
    // ------------------------------------------------------
    task check_result();
        if (intf.result !== expected_result) begin
            $display("ERROR @ %0t: op_sel=%b, a=%h, b=%h | expected result=%h,
got=%h",
                     $time, intf.op_sel, intf.a, intf.b,
                     expected_result, intf.result);
            error_count++;
        end
        if (intf.carry_out !== expected_carry) begin
            $display("ERROR @ %0t: op_sel=%b, a=%h, b=%h | expected carry=%b, got=%b",
                     $time, intf.op_sel, intf.a, intf.b,
                     expected_carry, intf.carry_out);
            error_count++;
        end
    endtask

    // ------------------------------------------------------
    // 8. Main sequence – controls simulation flow
    // ------------------------------------------------------
    initial begin
        wait(!rst); // Wait until reset is released
        $display("Starting tests...");
        generator(1000); // Number of tests
        $display("Tests completed: %0d total, %0d errors", test_count, error_count);

        if (error_count == 0)
            $display("TEST PASSED");
        else
            $display("TEST FAILED");

        $finish;
    end

    // Optional waveform dump
    initial begin
        $dumpfile("tb_top.vcd");
        $dumpvars(0, tb_top);
    end

endmodule
```