The Andrew and Erna Viterbi Faculty of
ELECTRICAL & COMPUTER
ENGINEERING

Technion
Israel Institute of Technology

Hardware Implementation of a Real-time Genetic Algorithm for Adaptive Filtering Applications

Final Report

Submitted by:

May Buzaglo Judit Ben Ami

Supervised by:

Shahar Gino

Submitted on: 20-January-2022

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1. Adaptive Filters

What is an Adaptive Filter? [1]

An adaptive filter is a computational device that iteratively models the relationship between the input and output signals of the filter. The classical usages of adaptive filtering are system identification, prediction, noise cancellation, and inverse modeling.

An adaptive filter self-adjusts the filter coefficients according to an adaptive algorithm, meaning that coefficients change with an objective to make the filter converge to an optimal state. The optimization criterion is a cost function, which is most commonly the mean square of the error signal between the output of the adaptive filter and the desired signal. As the filter adapts its coefficients, the mean square error (MSE) converges to its minimal value. At this state, the filter is adapted, and the coefficients have converged to a solution.

The Adaptive Filtering Problem

Given the input signal $x[n]$ to an adaptive filter at time n, the corresponding filter output signal sample at time n is $y[n]$. For the moment, the structure of the adaptive filter is not important, except for the fact that it contains adjustable parameters whose values affect how $y[n]$ is computed. The output signal is compared to a second signal $d[n]$, called the desired response signal, by subtracting the two samples at time n.

This difference signal, which is known as the error signal , is given by:

$$e[n] = d[n] - y[n] \quad \textbf{(1)}$$

The error signal is fed into a procedure which alters or adapts the parameters of the filter from time n to time n+1 in a well-defined manner.
Over time, i.e. the time index n increasing, it is hoped that the adaptive filter outputs is becoming a closer fit to the desired signal, such that the magnitude of $e[n]$ decreasing over time.
A scheme of an adaptive filter is shown in figure 1.

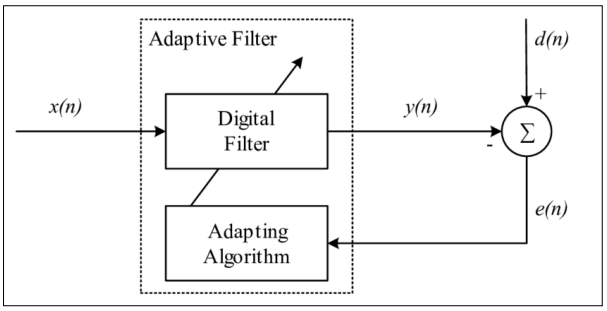


Figure 1 : *Diagram of a typical Adaptive Filter*

Filter Structures

The filter input vector is given by:

$$\textbf{(2)} \quad X[n] = (x[n], x[n-1], ..., x[n-M+1])^T$$

where M is the filter length.

The coefficient vector $W[n]$ is:

$$\textbf{(3)} \quad W[n] = (w_1[n], w_2[n], ..., w_M[n])^T$$

where $w_k[n]$ refers to the k[th] weight at time n. This vector represents the set or vector of weights, which controls the filter at sample time n.

Finally, the output is expressed as:

$$\textbf{(4)} \quad y[n] = W[n]X[n]$$

1.2. Genetic Algorithms

Different optimization problems have different algorithms for solution, depending on the nature of the problem. Some problems are "easy enough" and they have algorithm who guarantees to provide the optimal solution in a reasonable time, and some problems don't have a known efficient algorithm for solving them. For those types of problems different heuristics have been developed, one of them is the family of genetic algorithms. **Genetic algorithm is a heuristic for solving optimization problems, inspired by the process of natural selection and evolution**.

The genetic algorithm [2] starts with an initial set of possible solutions, known as "generation 0". Each set of solutions is called ***population***, and each solution in the population is called an ***individual***. The solution is usually represented as a sequence of bits, and each bit is called a ***gene***. Every iteration a new ***generation*** (population) is formed by performing the actions of selection, crossover and mutation on the previous generation, and the evolution process (iterations) continuous until reaching a certain stopping criterion, e.g. until reaching the G^{th} generation when G is set in advance.

A flowchart of genetic algorithm with stopping criteria of reaching to G^{th} generation is shown in figure 2.

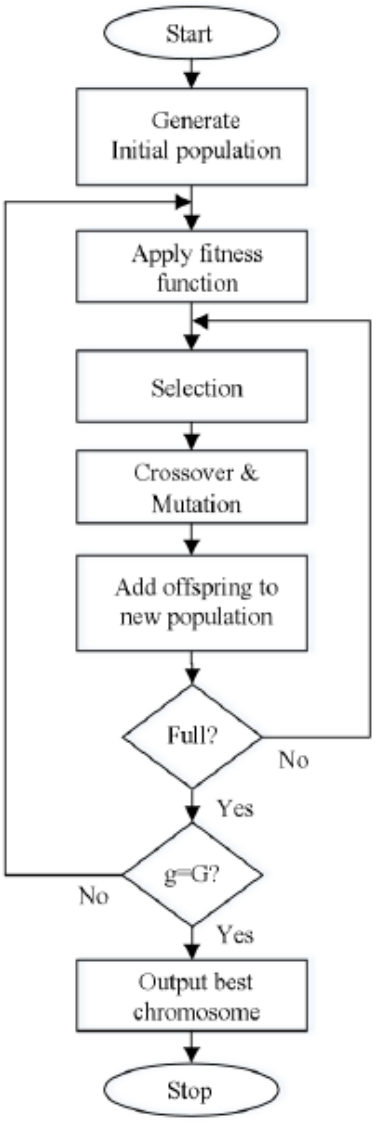


Figure 2 *: Genetic algorithm flowchart*

Description for each phase:

1. Create initial population:
   This can be done in several methods. The most common one is to generate random individuals. The number of individuals in a generation , N, is constant for all generations.
2. Calculate fitness function:
   The fitness function is applied for each individual separately, and it quantifies "how much" a solution is optimal.
3. Selection
   In this phase pair of individuals are selected. Overall, there will be N pairs, and each individual can be in more than one pair. The more fit an individual is, the higher the probability he will be selected for a pair.
4. Crossover
   For each pair a crossover between the two individuals is performed in order to create a new individual (two parents from generation i creates a child that is part of generation i+1). For each pair a crossover point is chosen at random from within the sequence of gene, and then the child's gene sequence will be from one parent up to the crossover point and from the other parent from the crossover point and onward.
5. Mutation
   In a pre-defined ratio, some of the children will go through mutation phase, i.e. their genes will be subjected to mutations (changes, e.g. flips of bits) with low random probability.

An example of all phases is presented in figure 3 below.

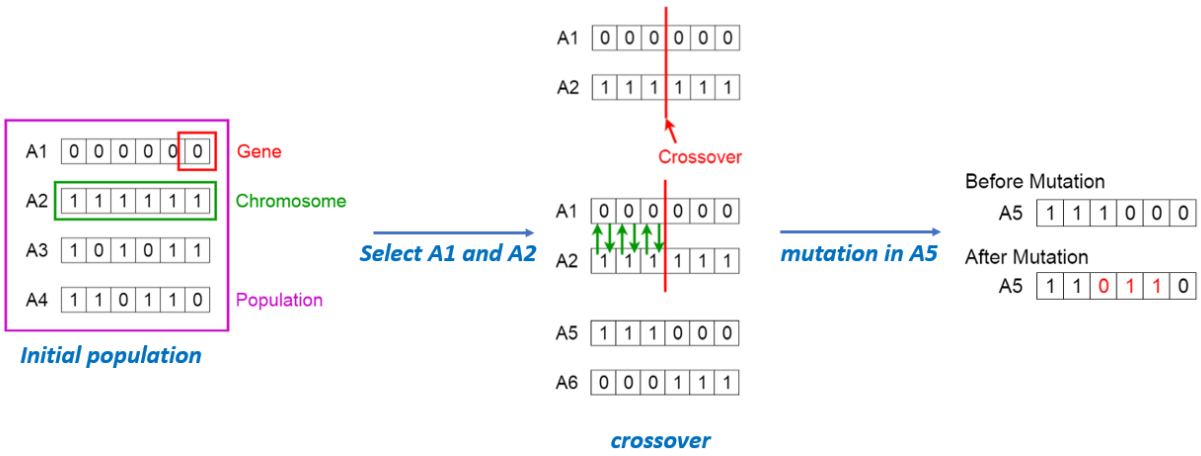


Figure 3 : *Genetic algorithm phases example*

1.3. ARMA models

General speaking, a lot of signals (time series) can be described by 3 components:

1. **Trend** – a general systematic linear or nonlinear component that changes over time and does not repeat.
2. **Seasonality** – a general systematic linear or nonlinear component that changes over time and does repeat.
3. **Noise** – a non-systematic component that is nor Trend/Seasonality within the data.

A visualization of the above can be shown in figure 4 [3].

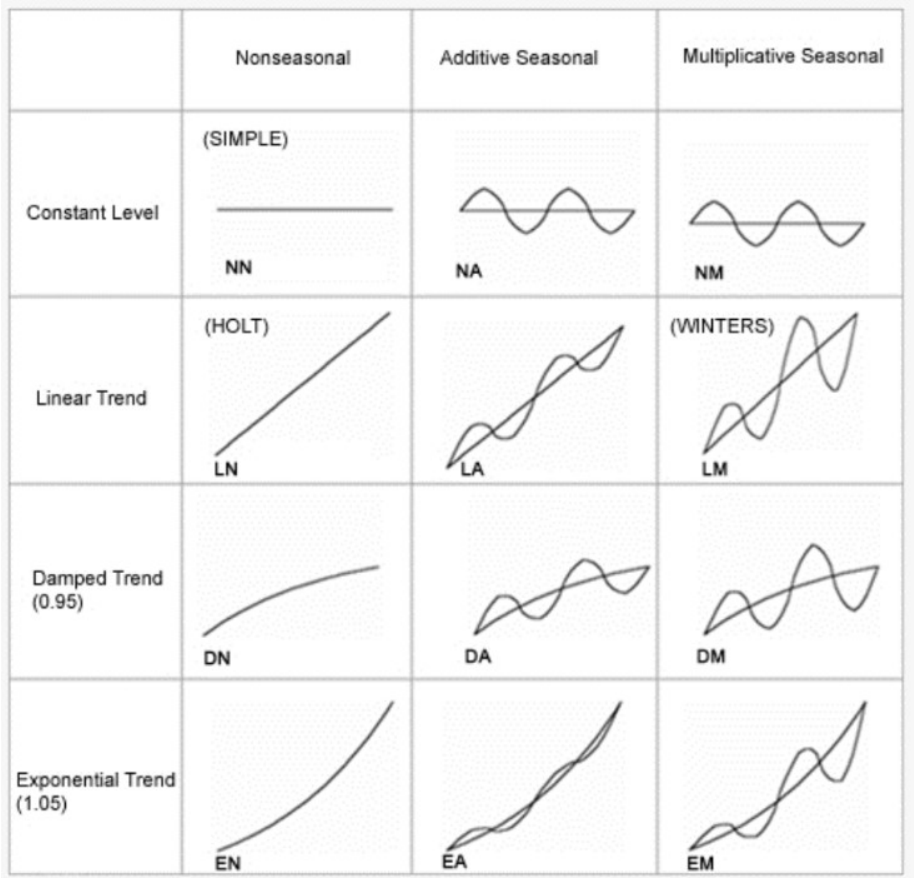


***Figure 4**: Trend, seasonality, and noise of signals*

ARMA (AutoRegressive Moving Average) models is a family of linear estimators for stochastic signals that are **stationary**, i.e., have a constant trend of 0, have no seasonality, and that their standard-deviation is constant over time. In other words, those signals can be assumed to be the originated from a **filtered white Gaussian noise**.

Trend and seasonality can be estimated and take into account by different models that will not be discussed.

ARMA models are parametric models, and they are divided into 3:

1. Moving average (MA)
2. Autoregressive (AR)
3. A combination of MA and AR – this is called ARMA

There are also generalizations of ARMA module that will not be covered in this scope.

As mentioned above, the signal is assumed to be filtered white Gaussian noise. Mathematically, the signal is denoted as y[n], and the ARMA model suggests that is a linear combination of its previous K samples and L previous noise values (the white Gaussian noise that is the input to the filter):

$$\textbf{(5)} \quad y[n] = -\sum_{k=1}^{K} a[k]y[n-k] + \sum_{l=0}^{L} b[l]u[n-l]$$

$y[n]$ is the signal, $u[n]$ is white Gaussian noise, and $\{a\}_{k=1}^{K}, \{b\}_{k=0}^{L}$ are parameters.

- MA model assumes that the signal is only a function of the white noise, i.e., it assumes that $a[k] = 0$ for every k:

  $$\textbf{(6)} \quad y[n] = \sum_{l=0}^{L} b[l]u[n-l]$$

  Note that in this case the filter that generates the signal from the white Gaussian noise is FIR filter.

- AR model assumes that the signal is only a function of its previous samples, i.e. ,it assumes that $b[l] = 0$ for every l:

  $$\textbf{(7)} \quad y[n] = -\sum_{k=1}^{K} a[k]y[n-k]$$

  Note that in this case the filter that generates the signal from the white Gaussian noise is IIR filter.

Given samples of a signal, ARMA estimator will provide the estimations for the coefficients $\{a\}_{k=1}^{K}, \{b\}_{k=0}^{L}$ so that a predication of the next sample can be made:

$$\textbf{(8)} \quad \hat{y}[n] = -\sum_{k=1}^{K} a[k]y[n-k] + \sum_{l=0}^{L} b[l]u[n-k]$$

1.4. AMBA APB

The APB (Advanced Peripheral Bus) is part of the AMBA 3 (Advanced Microcontroller Bus Architecture) protocol family [4] . It is communication protocol between single master and single slave, which allows read or write transactions. The protocol is synchronous and has single clk and single rstN, and a single transaction duration is at least 2 clk cycles, because a transaction has 2 phases: setup phase and access phase.

In addition, the protocol supports wait state (slave not ready) and slave error signal option. **Those features will not be discussed and are not in use in the project**.

Basic write transaction:

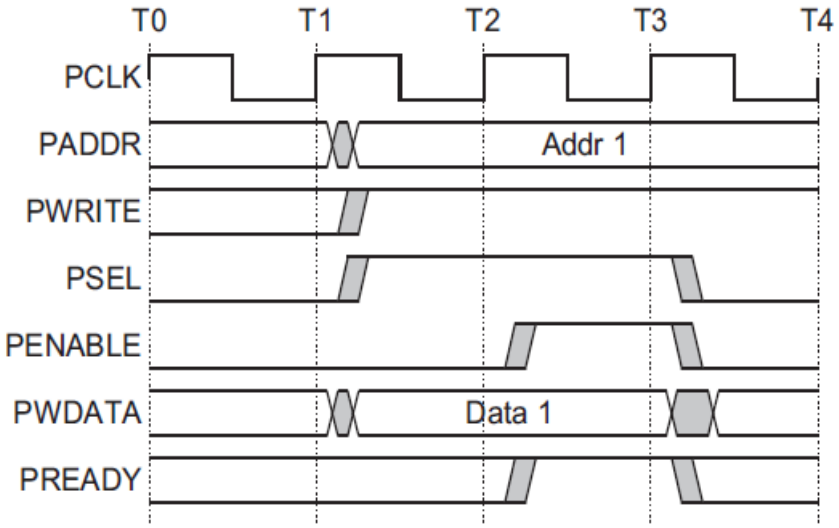


***Figure 5** : AMBA APB basic write transaction waveform*

The write transfer starts with the address, write data, write signal and select signal, all changing after the rising edge of the clock. The first clock cycle of the transfer is called the Setup phase. After the following clock edge the enable signal is asserted, PENABLE, and this indicates that the Access phase is taking place. The address, data and control signals all remain valid throughout the Access phase. The transfer completes at the end of this cycle.

The enable signal, PENABLE, is deasserted at the end of the transfer. The select signal, PSEL, also goes LOW unless the transfer is to be followed immediately by another transfer.

Basic read transaction:

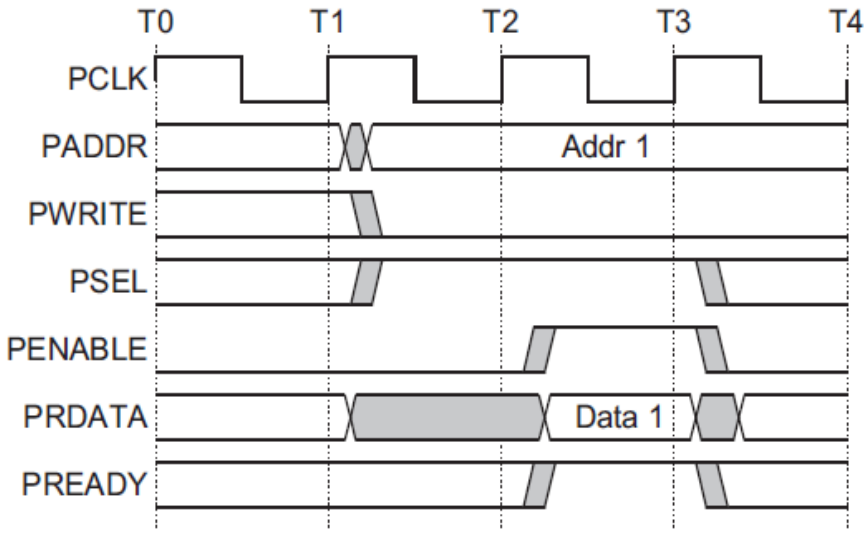


Figure 6 : AMBA APB basic read transaction waveform

The timing of the address, write, select, and enable signals are as described in the basic write transfers section above. The slave must provide the data before the end of the read transfer.

Interface description:

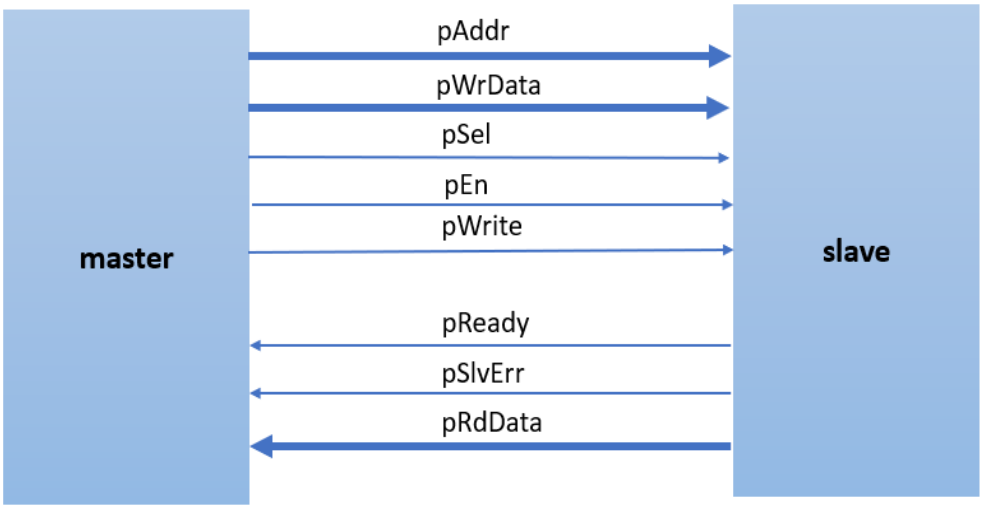


Figure 7: AMBA APB Interface signals scheme

Master's signals:

- **pAddr** – the slave's address that the master requests to read from/write to.
- **pWrData** – the data the master requests to write into pAddr. Relevant only for write transactions (pWrite HIGH).

- **pWrite** – indicates if the transaction is read transaction (pWrite LOW) or write transaction (pWrite HIGH).
- **pSel** – HIGH only when there is an active transaction. This signal is asserted at the entrance to the setup phase and deasserted when leaving the access phase.
- **pEn** – HIGH only when there is an active transaction that is in the access phase. This signal is asserted at the entrance to the access phase and deasserted when leaving the access phase.

Slave's signals:

- **pRdData** – the data the slave returns to the master as a result of read request. Valid only when pEnable and pReady are both asserted.
- **pReady** – Not in use in this scope. Stuck at HIGH (slave is always ready for requests).
- **pSlvErr** – Not in use in this scope.

1.5. Paper study

The paper "*Hardware Implementation of a Real-time Genetic Algorithm for Adaptive Filtering Applications*" [5] proposes a hardware solution for estimate ARAM models coefficients by performing genetic algorithm.

This dedicated hardware communicates with a main CPU. Inputs to the filter, V[n] and d[n], are fed to the genetic algorithm accelerator, and the hardware outputs as a response the filter's coefficients, W, as it can be shown in figure 8.

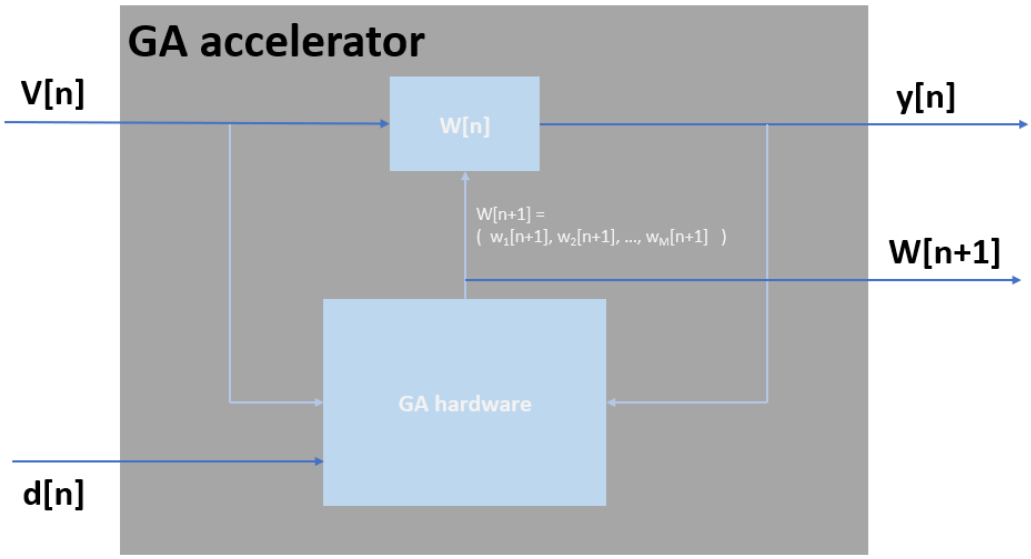


***Figure 8** : GA based adaptive filter*

The number of coefficients M is configured by the software, and each coefficient is Q=6 bits in fixed-point representation: the MSB is sign bit and the other 5 bits are fraction bits, such that the decimal value for each coefficient ranges in the interval $\left[-1, \frac{31}{32}\right]$.

The process starts with the software configuration of system's parameters, for example M (more parameters are mentioned ahead). Afterwards, every time the software provides a new valid V[n] and d[n], the GA hardware estimates the weights for the next sample, i.e. estimates W[n+1], by preforming G iterations of the genetic algorithm (fitness, selection, crossover, mutation).

The coefficients estimation, as mentioned, is done by genetic algorithm, and it's flowchart is shown in figure 2. An FSM controls the state of the system and the progress in the pipe, which is consists of the genetic algorithm stages.

The hardware architecture for each step's main block is presented in the article. An individual is a sequence of $Q \cdot M$ bits, which represents the following concatenation of all the weights:

(9) $$\{ \underbrace{w_1}_{Q-bits}, \underbrace{w_2}_{Q-bits}, ..., \underbrace{w_M}_{Q-bits} \}$$

The number of individuals in a population, P, is configured in the beginning by software and is constant for every generation. The number of generations, G, is also configured by software.

Block A – Random number generator:

The random number generator architecture is shown in figure 9. It is implemented using linear feedback shift register (LFSR) with 42 bits. It is used to generate the initial population, and random variables for crossover and mutation.

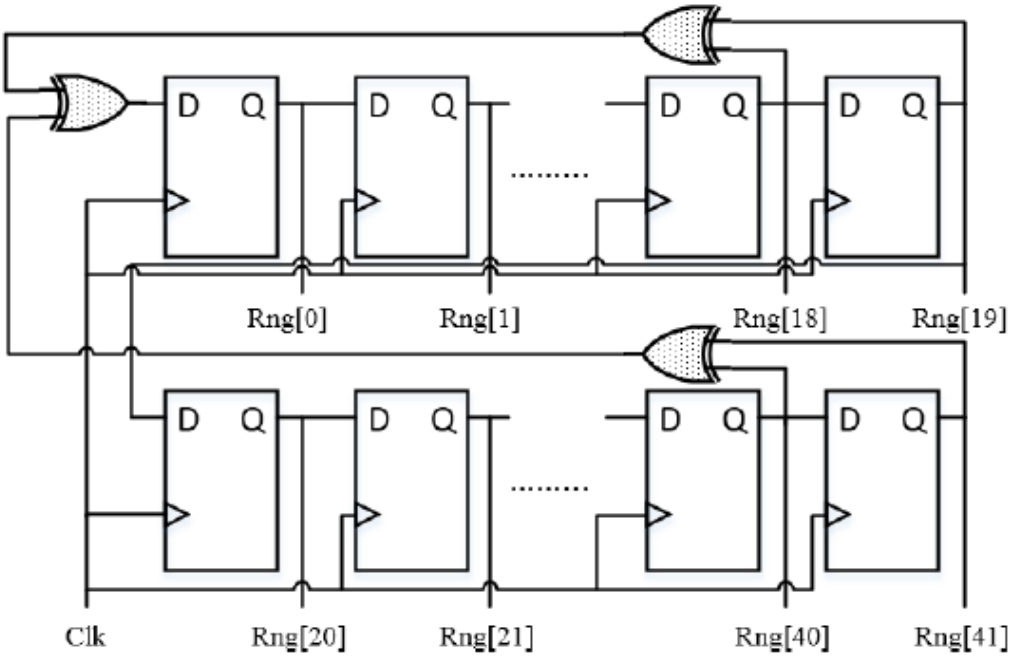


Figure 9 : *GA hardware – random number generator architecture*

Block B – Fitness block:

The step performed at the beginning of each iteration is applying the fitness function over every individual in the population. The fitness function applied is given by:

$$\textbf{(10)} \qquad f(n) = \sum_{n-B+1}^{n} |d(n) - y(n)|$$

Where B is configured by software and is the number of previous outputs y[n] to take into account. Only after B inputs have been inserted to the system the fitness calculation can starts.

For linear systems, as mentioned before, y[n] is given by equation (4), i.e. linear combination of the current coefficients and the current and previous inputs. But since nonlinear systems are taking into account, a hyperbolic tangent function is applied (mapped in a lookup table (LUT)).

Overall, the architecture of the fitness block is shown in figure 10. It gets the set of individuals from the previous step, and outputs them with an attached fitness function result.

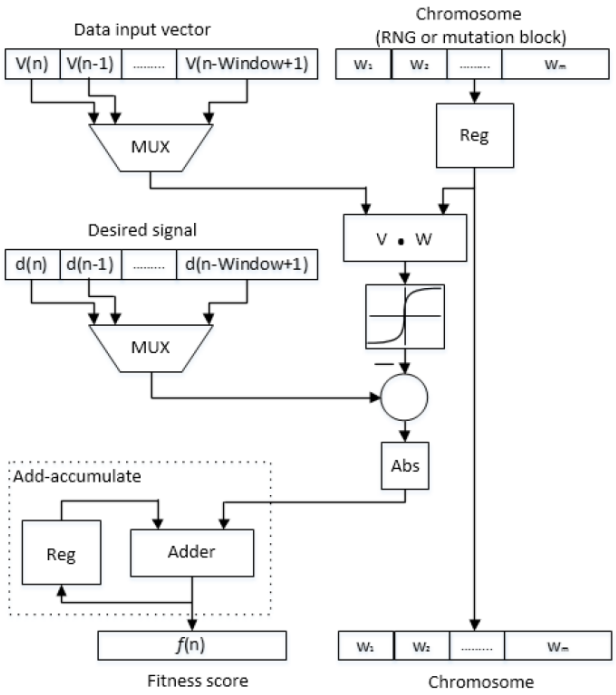


***Figure 10** : GA hardware – fitness block architecture*

Block C – Selection:

After the fitness stage, the selection step takes place and it outputs P pairs of parents, that will later produce the children of the next generation.

The fitness block architecture is shown in figure 11.

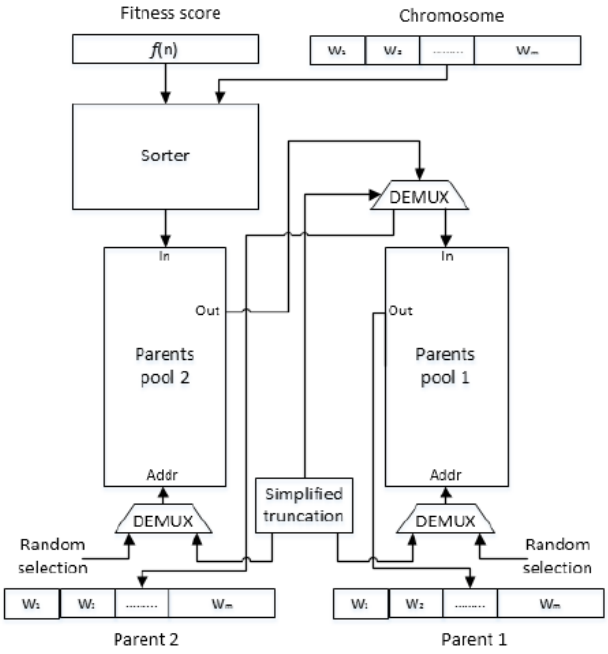


***Figure 11** : GA hardware – selection block architecture*

Block D – Crossover:

The crossover step is performed after the selection stage and is performed separately for each pair of parents selected. The crossover rate is 1, i.e. only one weight, that is chosen randomly, will be subjected to the crossover operation. All other weights will be transformed from parent 1 directly to the child. The chosen weight will have a crossover point that is set according to the fittest individual in the population (the fitter it is, the crossover point will be closer to the LSB).

Overall, the architecture of the crossover block is shown in figure 12, and it passes the P children created to the next step.

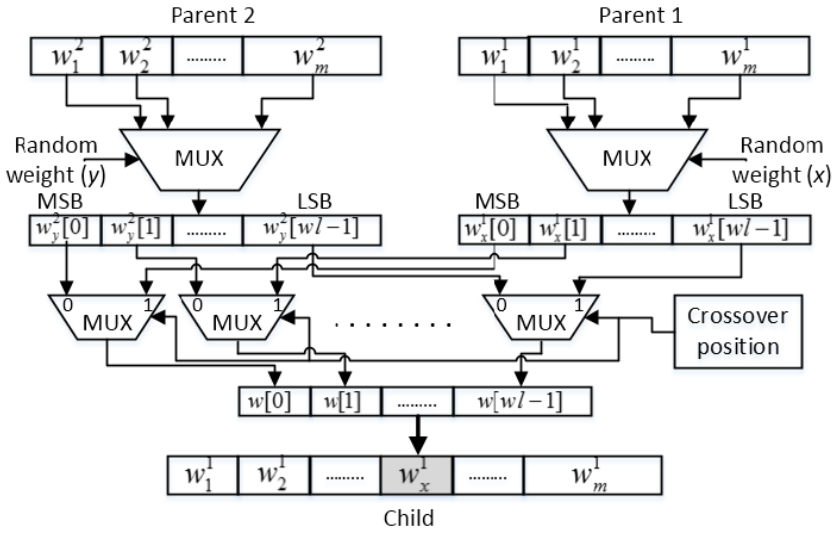


***Figure 12** : GA hardware – crossover block architecture*

Block E – Mutation:

The mutation stage is performed after the crossover stage, and it is the last stage in the pipe and its output will be the next generation final population.

The mutation is achieved by performing a XOR operation between a randomly selected individual at a randomly selected weight, and a random generated binary word. A variable mutation rate based on the best fitness score of the population is adopted, and it gradually increases from 0.1 to 0.5 when the fitness score varies from its highest to the lowest value.

Overall, the architecture of the mutation block is shown in figure 13.

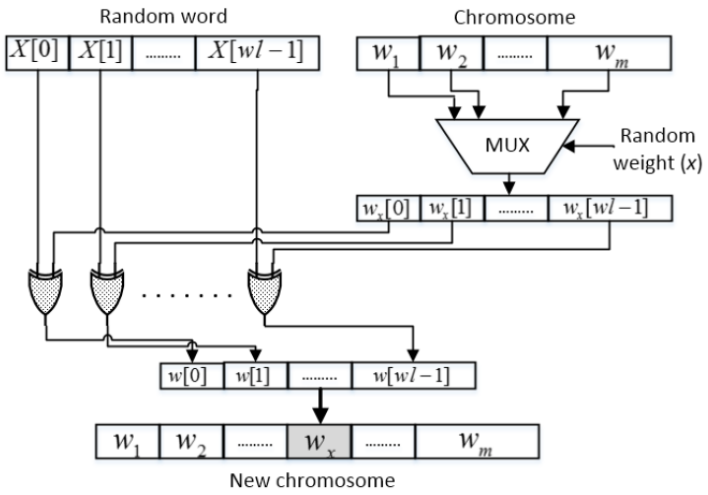


***Figure 13** : GA hardware – mutation block architecture*

After the mutation block there is a population of P individuals that belong to the next generation, and after G iterations the best (fit) from all is selected to be the solution, i.e. to represent W[n+1].

Finally, the paper shows a study case of identification the parameters of ARMA model with M=7 weights. The chosen parameters were B=16 sampled data points and P=16 individuals in a generation.

The implementation shows high signal processing performances and low resources cost.

1.6. Summary

In section 1.1 the concept of adaptive filtering has been introduced, and as mentioned the parameters of the filter are its set of weights, through them an input signal is being filtered to create an output signal, hopefully very close to a certain desired signal. Adaptive filtering is a very useful tool for signal processing objectives, and thus finding an efficient way to estimate its set of weight is a matter of interest. A type of adaptive filter is ARMA filter, that was introduced in section 1.3.

Overall, finding the weights is an optimization problem because the goal is to generate an output signal similar to a given desired signal. Different approaches can be applied, and the paper that was represented in section 1.5 approach the problem via genetic algorithm, that was introduced in section 1.2.

The solution in the paper suggest that the estimation of the weights will be done in a dedicated hardware, which recreating it is the goal of this project. As every hardware, a communication protocol should be set in order to communicate with the main CPU. The chosen protocol is AMBA APB, which is presented in section 1.4.

2. GA Accelerator Implementation

2.1. Block diagram

2.1.1. Top level diagram

Top level diagram of the GA accelerator implemented in the project is shown in figure 14.

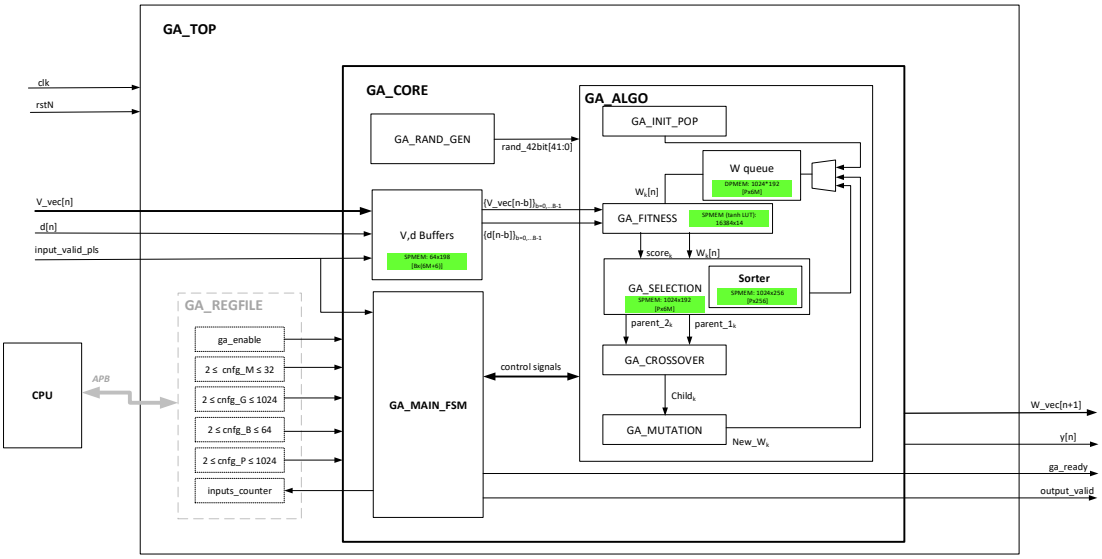


Figure 14 *: GA accelerator – top level block diagram*

The goal of GA accelerator hardware is to receive over time values of V_vec[n] and d[n] signals, and at each timestamp to output:

1. **The output y[n]=W_vec[n]*V_vec[n]**
2. **The estimated weights for the next timestamp: W_vec[n+1]**

At first, all the genetic algorithm configurations (parameters) should be set by CPU and are saved in registers (GA_REGFILE). After they are all set, 1 bit enable register (ga_enable) value should rise and should remain high. After enable rises, every valid V[n] and d[n] inputs are being processed in the GA accelerator, and the GA accelerator outputs y[n]=V[n]W[n] and the estimation for W[n+1]. Additional information regarding the data interface commounication is at section 2.4.3.

The GA accelerator implements genetic algorithm, with an FSM based control unit. At the rise of enable an initial population is generated by GA_INIT_POP, and after that for every valid set of V[n] and d[n] the ready signal (ga_ready) is deasserted and the calculation of W[n+1] takes place:

1. GA_FITNESS calculates the fitness for every individual, and pass it to the next stage: selection.
2. GA_SELECTION sorts the individuals by the result of the fitness function, and selects pairs of parents, and pass them to the next stage: crossover.
3. GA_CROSSOVER generates a new child from the two parents, and pass them to the next stage: mutation.
4. GA_MUTATION gets a child and if decided preforms a mutation on it, and generates the final new individual for the next generation.
5. Stages 1-4 repeats themselves until reaching the G^{th} iteration (generation). When g=G the best individual is output as W[n+1] and the ready signal is asserted (GA accelerator is done and ready for the next valid V[n],d[n]).

The design includes registers file to allow flexibility in genetic algorithm parameters:

1. **cnfg_M register** – number of elements in the filter, i.e. number of elements in V_vec/W_vec/chromosome. Range is [2,M_max] whereas **M_max=32**.
2. **cnfg_G register** – number of generations for the algorithm to run per {V,d} input. Range is [2,G_max] whereas **G_max=1024**.
3. **cnfg_B register** – number of elements in the smoothing window. Range is [2,B_max] whereas **B_max=64**.
4. **cnfg_P register** – number of individuals per generation. Range is [2,P_max] whereas **P_max=1024**.

Additional information regarding the registers and the size limitations appears in the programmer's guide in section 2.9.

Memories:

As shown in figure 14 (in green), ga accelerator contains 5 memories: 4 SRAMs and 1 DRAM. Those memories are listed below (with B,M,P parameters referring to B_max, M_max, P_max and DATA_W referring to Q which is one weight width) :

1. **V&d buffer SRAM**: at all times saves the B latest samples of V_vec and d for the usage of GA_FITNESS.
   Hence, the size of the memory is:
   $$V\ \&\ d\_buffer\_mem\_size = B \times DATA\_W(M+1)\ [\text{bits}]$$
   And for default parameters values:
   $$V\ \&\ d\_buffer\_mem\_size = 64 \times 6(32+1)\ [\text{bits}] = 64 \times 198\ [\text{bits}]$$
2. **W queue (chromosomes queue) DRAM**: DRAM is used to allow parallel push and pop to the queue. The queue stores chromosomes for GA_FITNESS to use. Required space is the maximum allowed number of chromosomes.
   Hence, the size of the memory is:
   $$chromosomes\_queue\_mem\_size = P \times DATA\_W \cdot M\ [\text{bits}]$$
   And for default parameters values:
   $$chromosomes\_queue\_mem\_size = 1024 \times 6 \cdot 32\ [\text{bits}] = 1024 \times 192\ [\text{bits}]$$

3. **tanh LUT SRAM**: GA_FITNESS calculation consists of few stages, one of them is calculation of tanh. For reasons specified in GA_FITNESS selection (section 2.5.3.2), the size of the memory is:
$$tanh\_lut\_mem\_size = 16384 \times 14 \text{ [bits]}$$
4. **W sorted pool (chromosomes sorted pool) SRAM**: All the chromosomes of a generation eventually arrive to this memory, which is placed in GA_SELECTION, and from there they are paired to create the next generation.
Hence, the size of the memory is the number of individuals per generation over the space needed for a single chromosome:
$$chromosomes\_sorted\_pool\_mem\_size = P \times DATA\_W \cdot M \text{ [bits]}$$
And for default parameters values:
$$chromosomes\_sorted\_pool\_mem\_size = 1024 \times 6 \cdot 32 \text{ [bits]} = 1024 \times 192 \text{ [bits]}$$
5. **Sorter memory SRAM**: in GA_SELECTION a sorter is placed, and its job is to sort the chromosomes by their fitness score (FIT_SCORE_W). As explained in GA_SELECTION section (section 2.5.3.3) it requires a memory in the size of:
$$sorter\_mem\_size = 1 + FIT\_SCORE\_W + DATA\_W \cdot M + 4 \cdot (1 + c\log 2(P)) + 1 [\text{bits}]$$
And for default parameters values:
$$sorter\_mem\_size = 1 + 18 + 192 + 4 \cdot (1 + c\log 2(1024)) + 1 = 256[bits]$$

2.1.2. Pipeline diagram

Top pipeline diagram is shown in figure 15. The diagram presents the main steps and time ratios at each step, from the moment enable rises until the first W[n+1] is calculated and valid (and the accelerator is ready for V[n+1] and d[n+1]).

The following configurations are considered:

1. Seven weights (M=7) – so that creating a new random chromosome takes 1clk cycle.
    a. Deeper understanding regarding timing of this stage is in section 2.5.3.1.
2. Smoothing window of size 4 (B=4) – so that only the 4th set of valid V and d (V[3] and d[3]) deasserts ga_ready and triggers the calculation of W[4].
3. Population of 8 chromosomes (P=8).
4. Two generations (G=2).

In the diagram:

- Each column represents a stage in algorithmic pipe.
- Each row represents one clock cycle, except the first row that is the title: each stage name and clock cycles it requires in a parametric manner (according to the registers configuration).

- The small grey-blue circles are the 8 chromosomes of the first generation and the small orange-red squares are the 8 chromosomes of the second generation.
- In each generation the darker the shade of the chromosome the higher its fitness score.
- At each row only the units that are in use are marked in purple.

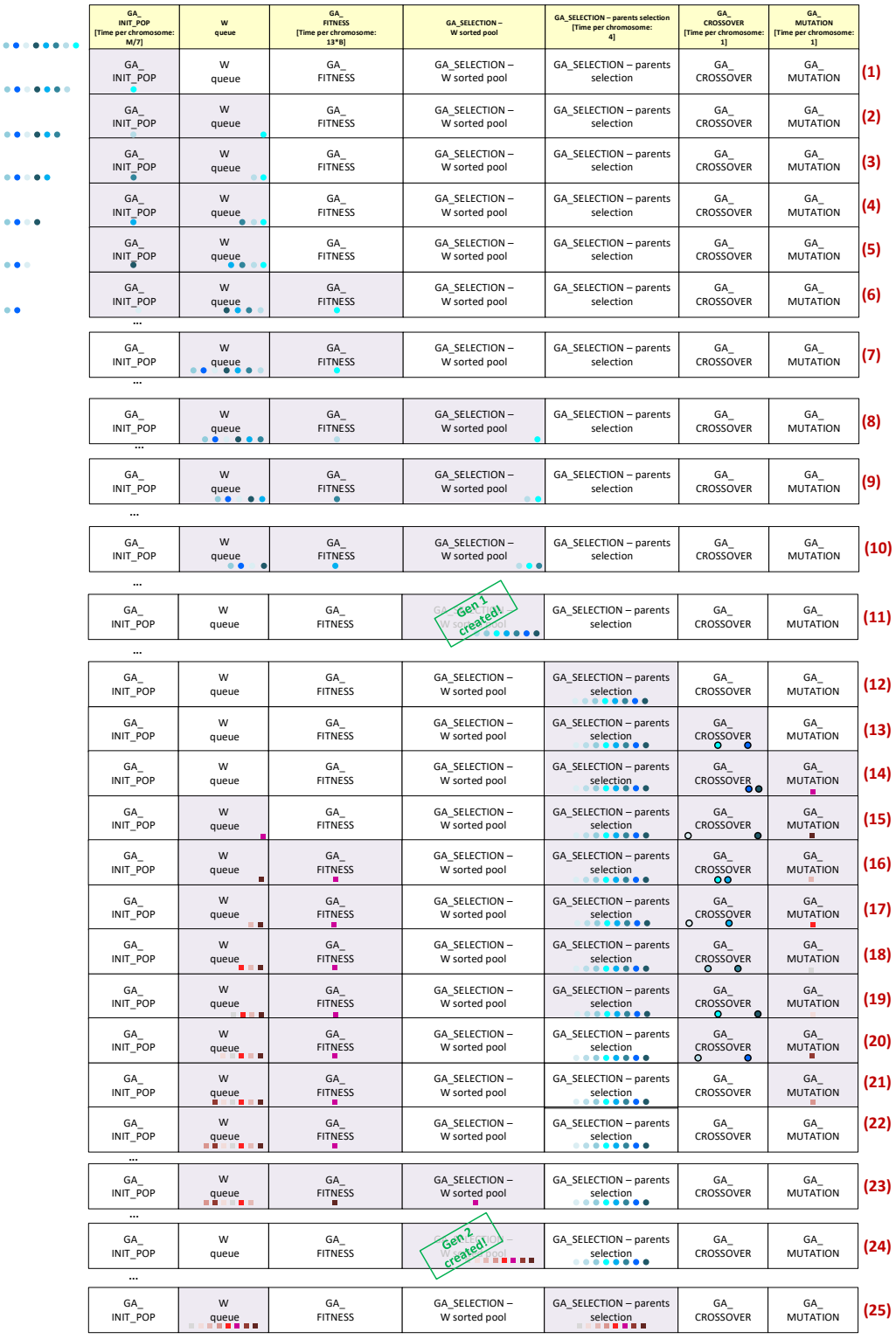


Figure 15 *: GA accelerator – top level pipeline diagram*

Detailed description for each stage in the diagram:

line (1) is the immediate response after enable (ga_enable) was asserted: random chromosomes are being created. In line (1) one can see that the first chromosome is created.

In line (2) the first chromosome is pushed to the chromosomes queue (W queue) and the second chromosome is being created.

Simultaneously sets of valid V and d arrive and fill the smoothing window (B). Until the 4^{th} (B=4) valid won't arrive, the fitness block will not start to pop elements from the queue, hence the chromosomes that are being created are waiting in the queue.

In lines (3),(4),(5) the 3^{rd}-5^{th} chromosomes of the first generation are created, and in parallel additional sets of valid V and d arrive, such that in line (5) the B^{th} (4^{th}) set arrives.

In line (6) ga_ready is deasserted and the fitness unit starts to pull elements from the queue and pops the head of the queue. The fitness unit calculates the fitness score for each chromosome separately and the calculation takes 13*B clock cycles per chromosome. Hence, between lines (6) and (8) there are three dots in order to emphasize the time gap. Line (7) is 3 clock cycles after line (6), and at this point GA_INIT_POP is done, and all the chromosomes are ready and waiting in the queue.

Line (8) is the one clock after last clock cycle of the first chromosome fitness calculation: one can see that the first chromosome has passed to the selection stage and the second chromosome is inside the fitness calculation unit.

The selection stage is divided into two parts: the first part is sort phase in which the chromosomes of the current generation are sorted. The second stage is the parent selection stage that starts to generate the next generation, and this occurs only after all the chromosomes of the current generation were sorted (Hence, the sorter is a bottle neck. This bottle neck is mandatory since fair selection of parents can be done only after all the chromosomes are ordered).

In lines (9),(10),(11) the chromosomes are inserted one-by-one, according to arrival order, to the fitness unit and afterwards to the sorted pool. The insertion to the sorted pool is ordered according to fitness score, such that eventually in line (11) one can see all the 8 chromosomes of the first generation are sorted in the sorted pool and at this point generation 1 is done.

Since G=2, generation number two should be created. Hence, the data from the sorter is copied into W sorted pool. The coping process takes time (read and write actions from single port memories), and after P clock cycles line (12) occurs: all the chromosomes are entering the parent selection phase.

In line (13) the first pair of parents enters to the crossover stage in order to create next generation child. One clock cycle later, in line (14), this child enters to the mutation stage and in parallel the second pair of parents enters the crossover stage.

One clock cycle after that, in line (15), the first child enters the W queue, the second new child enters the mutation unit, and a third pair of parents is selected and enters the crossover unit.

The process continues in lines (16)-(24): overall 8 pairs are generated (because P=8), and each pair creates a child that goes throw crossover, mutation, W queue, fitness and sort stages (in this order).

In line (24) the second generation is done, sorted and ready, and since G=2 this is the last generation. The darkest square is outputted as the best chromosome (as W[n+1], and for this example W[4]), ga_ready is asserted and the accelerator waits for new valid set of V and d (to V[4] and d[4]). Meanwhile, the chromosomes of the last generation are transferred to W queue and after P clock cycles are all inside, as shown in line (25). The transition of the chromosomes from GA_SELECTION is important because when the new V and d will arrive a new fitness score should be calculated for them.

Note that in the scenario presented above the new set of V and d doesn't arrive, but there can be a scenario in which it arrives immediately and then the process of fitness calculation to the first chromosome and the transfer of the other chromosomes to the W queue happen simultaneously.

2.2. Pins description

signal name	input/output	Bits	pls/lvl	description
Clk	Input	1		
Rstn	Input	1	Lvl	active low
Data IF				
V_vec[n]	Input	M_max x DATA_W	Pls	M_max elements, each is signed fixed-point at DATA_W.
d[n]	Input	DATA_W	Pls	Reference signal, signed fixed-point.
input_valid_pls	Input	1	Pls	active high
W_vec[n+1]	Output	M_max x DATA_W	Lvl	M_max elements, each is signed fixed-point at DATA_W.
y[n]	Output	DATA_W	Lvl	signed fixed-point.
ga_ready	Output	1	Lvl	active high: high when the block is ready for new V[n],d[n]
output_valid	Output	1	Lvl	active high: high when the outputs W_vec[n+1] and y[n] are valid.
APB IF				
pAddr	Input	REG_ADDR_W		
pWrData	Input	REG_DATA_W		
pWrite	Input	1		
pSel	Input	1		
pEn	Input	1		
pRdData	Output	REG_DATA_W		
pReady	Output	1		

***Table 1**: ga accelerator interface*

- V_vec, d and W_vec are all with wordlength (DATA_W) of 6bit in fixed point representation: 1 sign bit and 5 fractional bits. Hence, the range of value they can have is $\left[-1, \frac{31}{32}\right]$.
- The values of the parameters are:
  - DATA_W = 6 [DATA_INT_W =1, DATA_FRACT_W=5]
  - M_max = 32
  - REG_ADDR_W = 8 (6 registers, described in section 2.9)
  - REG_DATA_W = 32

2.3. Clock and Resets

The design is a synchronized design with positive-edge clock. There is one clock domain hence there is no need for CDC (clock domain crossing). The clock's frequency is 125MHz (T=8ns).

The design has an active low asynchronous reset.

2.4. Interfaces description

2.4.1. APB

APB interface is used for registers access. Detailed description regarding APB protocol is displayed in section 1.4.

In practice, the APB interface will not be implemented in this project, and the GA algorithm parameters will be set at GA_TOP (However, the design is implemented under the consideration that those parameters are configurations from registers).

2.4.2. Clock and Reset

The design has single clock and single asynchronous reset. Additional information regarding clock and reset can be found in section 2.3.

2.4.3. Data interface

At first, all the genetic algorithm configurations (parameters) should be set by CPU and are saved in registers (GA_REGFILE). After they are all set enable register value (1bit) should rise and should remain high. Only at this point the data interface can activates the hardware.

The GA accelerator hardware gets the data in a serial manner: The data inputs are V_vec and d, and they have an appropriate valid_pls to indicate that signals values are valid in the current clock cycle.

V_vec and d values are transferred to buffers, each in size B, when both valid_pls input and ready output are asserted (high) and ignored otherwise. At this point the ready output signal is deasserted and will be asserted again only when the GA accelerator will be ready to accept new valid V_vec and d. At the first few valid

V_vec and d the ready output signal remains high because no calculation takes place. At the B^{th} valid V_vec and d the ready signal is deasserted and genetic algorithm calculation is performed and estimates W_vec. When W_vec (and y) is ready, both ready and output_valid outputs are asserted, and remains high until getting new valid inputs.

Wave diagram of the above is shown in figure 16.

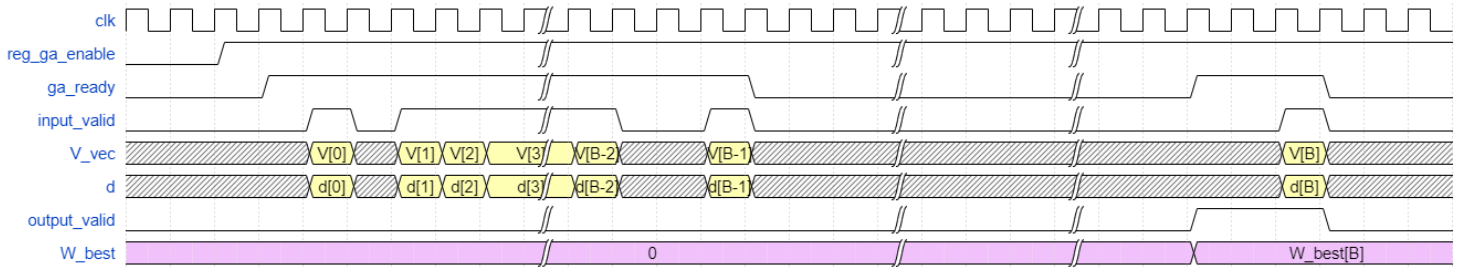


Figure 16 *: GA accelerator – top data interface wave diagram*

2.5. Sub-units description

All the sub-unites described below are sub-units of GA_CORE.
GA_CORE interface:

signal name	input/output	bits	pls/lvl	description
Clk	input	1		
Rstn	input	1	lvl	active low
Data IF				
V_vec[n]	input	M_max x DATA_W		M_max elements, each is signed fixed point with DATA_W.
d[n]	input	DATA_W		Signed fixed point
input_valid_pls	input	1	pls	active high
W_vec[n+1]	output	M_max x DATA_W	lvl	M_max elements, each is signed fixed point with DATA_W.
y[n]	output	DATA_W		Signed fixed point
ga_ready	output	1	lvl	active high: high when the block is ready for new V[n],d[n]
output_valid	output	1	lvl	active high: high when the outputs W_vec[n+1] and y[n] are valid.
Registers IF				
ga_enable	input	1	lvl	Enable for ga_accelerator. All cnfg_* registers should be set before the rise of ga_enable and stay steady until ga_enable falls. Every fall of ga_enable restart the system (sw_rst), and every rise of ga_enable triggers random formation of initial chromosome generation.
cnfg_M	input	M_MAX_W= clog2(M_max+1)= clog2(32+1)=6	lvl	M is the number of weights in the filter (number of elements in V_vec/W_vec). **Range: [2,32]** (M_max=32)
cnfg_P	input	P_MAX_W= clog2(P_max+1)= clog2(1024+1)=11	lvl	P is the number of chromosomes in each generation. **Range: [2,1024]** (P_max=1024)
cnfg_B	input	B_MAX_W= clog2(B_max+1)= clog2(64+1)=7	lvl	B is the size of the smoothing window, i.e. the number of timestamps in it. **Range: [2,64]** (B_max=64)
cnfg_G	input	G_MAX_W= clog2(G_max+1)= clog2(1024+1)=11	lvl	G is the number of generations to estimate single W_vec[n+1]. **Range: [2,1024]** (G_max=1024)
inputs_counter	output	32	lvl	The inputs counter counts how many valid sets of {V[n],d[n]} has been received since ga_enable rose. (Valid sets means sets in which input_valid_pls&ga_ready is TRUE)

***Table 2**: GA_CORE interface*

2.5.1. GA_MAIN_FSM

GA_MAIN_FSM is the main FSM that controls the ga_accelerator.

GA_MAIN_FSM controls the start and the stop of the algorithm, the generation counting, and set the push source to GA_ALGO queue (the mux selector for the chromosome queue (a.k.a W queue)).

In addition, GA_MAIN_FSM is in charge to calculate y[n] and to output it with W_vec[n+1] when its ready.

Interface:

signal name	input/output	Bits	pls/lvl	Description
Clk	Input	1		
Rstn	Input	1	Lvl	active low
Registers IF				
ga_enable	Input	1	Lvl	Enable from register files. Rises by SW after all the configurations parameters are set and remains high throughout the entire time the GA accelerator is used.
cnfg_B	Input	B_MAX_W= clog2(B_max+1)= clog2(64+1)=7	lvl	B is the size of the smoothing window, i.e. the number of timestamps in it. **Range: [2,64]** (B_max=64)
cnfg_G	Input	G_MAX_W= clog2(G_max+1)= clog2(1024+1)=11	lvl	G is the number of generations to estimate single W_vec[n+1]. **Range: [2,1024]** (G_max=1024)
inputs_counter	Output	32	Lvl	Counts the valid inputs (V_vec and d set) that were accepted by GA accelerator, from the rise to the fall of ga_enable.
Data IF: I/O form GA_ACCELERATOR				
input_valid_pls	Input	1	Pls	Indicates there is a new valid input (set of V_Vec and d).
V_vec[n]	Input	M_max x DATA_W		M_max elements, each is signed fixed point with DATA_W.
ga_ready	Output	1	Lvl	active high: high when the block is ready for new V[n],d[n]
output_valid	Output	1	Lvl	active high: high when the outputs W_vec[n+1] and y[n] are valid.
W_vec[n+1]	Output	DATA_W*M_max	Lvl	Relevant only when ga_ready is high. M_max elements, each is signed fixed point with DATA_W.
y[n]	Output	DATA_W	Lvl	Relevant only when ga_ready is high. Signed fixed point.
Data IF: GA_ALGO				
gen_created_pls	Input	1	Pls	Received from GA_ALGO (GA_SELECTION) and indicates that a single generation has been generated and sorted.

best_chrom	Input	DATA_W*M_max	Pls	Received from GA_ALGO (GA_SELECTION) and is valid when gen_created_pls is high. This is the best chromosome at the generation that was created. M_max elements, each is signed fixed point with DATA_W.
init_pop_start	Output	1	Pls	Indicates GA_INIT_POP should start to create cnfg_P random chromosomes.
fit_enable	Output	1	lvl	Activate GA_FITNESS block (allows it to pull data from the queue)
create_new_gen_req_pls	Output	1	Pls	Send to GA_SELECTION block as a response to gen_created_pls, when a new generation needs to be created (start signal for parent selection process).
stop_create_gens_req_pls	Output	1	Pls	Send to GA_SELECTION block as a response to gen_created_pls when it was the last generation created (i.e. when no new generation needs to be created). (Start signal for sending chromosomes from GA_SELECTION to chromosomes queue).
queue_mux_sel	Output	2	Lvl	Select to the mux placed before the queue in GA_ALGO (chromosomes queue for GA_FITNESS). Selects who is the source of the data that will be pushed. Has 3 options: 2'b00 – push source: ga_init_pop 2'b01 – push source: ga_mutation 2'b10 – push source: ga_selection

***Table 3**: GA_MAIN_FSM interface*

Diagram:

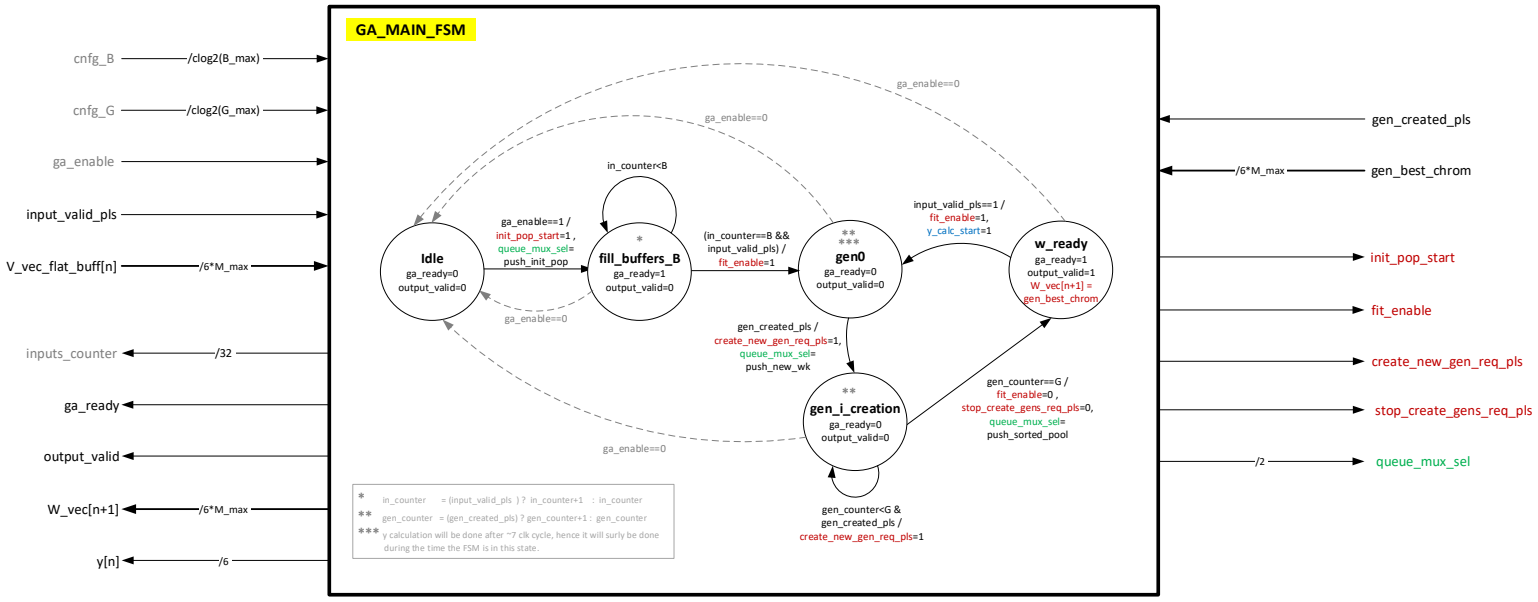


***Figure 17** : GA accelerator – GA_MAIN_FSM microarchitecture*

Detailed description:

1. At reset the FSM is at idle. Once ga_enable rises the FSM goes out of Idle, and in the future in any case ga_enable will fall the FSM will get back to idle.
2. When getting out of idle state (rising of ga_enable) the FSM goes to fill_buffers_B state. At this state the input buffers are filling up and no calculation takes place so ga_ready is high. Simultaneously, to speed up the process, init_pop_start rises so that the creation of initial random chromosomes will starts.
3. For B (cnfg_B) valid inputs the FSM remains in this state (inputs_counter) and when the buffers get full, i.e. at the B^{th} valid input, the FSM goes to gen0 state and the calculation, i.e. the genetic algorithm, starts, by enabling GA_FITNESS (fit_enable rises).
4. At gen0 state, and at all previous states (Idle, fill_buffers_B), queue_mux_sel is set to push_init_pop, i.e. the push source to queue is GA_INIT_POP. One by one GA_FITNESS pops out the chromosomes from the queue, calculates their fitness score, and transfers them to GA_SELECTION that sorts them. After the sort of all the P (cnfg_P) chromosomes in the generation is done GA_SELECTION asserts gen_created_pls that indicates the first generation is done and sends the best chromosome of this generation (best_W). As a result, the FSM rises create_new_gen_req_pls to allow parents creation by GA_SELECTION block and goes to gen_i_creation state.
5. At the transition to gen_i_creation state the mux selector, queue_mux_sel, changes to push_new_wk, i.e. the push source is the GA_MUTATION block. For most of the time the genetic algorithm takes place, the FSM is in this state (gen_i_creation). By an inner counter G (cnfg_G) generation are being counted (the counter increases every time gen_created_pls arrives), when the trigger to GA_ALGO to create a new generation is sending create_new_gen_req_pls). Eventually, when gen_created_pls rises for the G^{th} time the genetic algorithm is done and the FSM goes to w_ready state.
6. Arrival to w_ready state indicates the G^{th} generation is done and sorted in W sorted pool, and the best fit (best_chrom) is outputted as W_vec[n+1], and the ga_ready and output_valid rises. Note that at this point y[n]=0 since this is the first iteration and there is no valid W_vec[B-1].
   In addition, fit_enable is deasserted and stop_create_gens_req_pls is asserted in order to make sure no unnecessary actions are taking place and in order to save power. Also, queue_mux_sel is set to push_sorted_pool, i.e. push source is the GA_SELECTION which starts to push the sorted chromosomes to the queue, in order to prepare them to the next valid V_vec and d, which will require additional G iteration, and so on.
   The FSM will remain in this state until input_valid rises again, and when it does it will increase in_counter by 1 and will go to gen0 state. In addition, it will trigger the calculation of y[n]=V[n]*W_vec[n].

6.1. The calculation of y[n] is done by a simple inner product unit (see section 2.5.4) that is placed inside the FSM and takes 7 clock cycles. Hence y[n] will be ready while the FSM is still in gen0 state. However, output_valid will not rise until W_vec[n+1] will be ready.

7. The FSM will remain in the loop gen0→gen_i_creation→w_ready until ga_enable is deasserted or until reset is asserted.

2.5.2. GA_42BIT_RAND_GEN

GA_42BIT_RAND_GEN is a block that every clock cycle generates 42 random bits.

Those 42 bits are then wired in different places in the design in order to create randomization: in initial population creation, in parent selection, and in the crossover and mutation operations.

Interface:

signal name	input/output	Bits	pls/lvl	description
Clk	Input	1		
Rstn	Input	1	Lvl	active low
sw_rst	Input	1	Pls	
rand_data	Input	42	Pls	42 bit of random data

***Table 4**: GA_42BIT_RAND_GEN interface*

Diagram:

GA_42BIT_RAND_GEN microarchitecture is identical to the article's implementation and is presented in figure 9.

Detailed description:

The implementation is done using linear feedback shift register (LFSR) with 42 bits. Every clock cycle new set of 42bits is generated.

The block supports rstn and software reset, and the reset value of the LFSR is 42'b1.

Usage short description:

Those 42bits are routed to several places inside GA_ALGO main blocks: GA_INIT_POP, GA_SELECTION, GA_CROSSOVER, GA_MUTATION.

GA_INIT_POP works separately from all the other block mentioned, and its work is done before all the other blocks needs to use any random data. Hence, to GA_INIT_POP all of the 42bits are connected, so that it could generate chromosomes in the highest possible rate.

All of the other 3 units (GA_SELECTION,GA_CROSSOVER,GA_MUTATION) shares the 42 random bits in order to maximize randomization and to make them independent.

The split is done in the following way (detailed explanations in each unit section):

1. GA_SELECTION needs the following number of random bits:
   SELECTION_RAND_W = clog2(P_max)+clog2(P_max/2)
   And for the chosen parameters:
   SELECTION_RAND_W = clog2(1024)+clog2(512) = 19
   Hence, GA_SELECTION gets the 19LSBs, i.e. rand_42bit[18:0].

2. GA_CROSSOVER needs the following number of random bits:
   CROSSOVER_RAND_W = 2*clog2(M_max)
   And for the chosen parameters:
   CROSSOVER_RAND_W = 2*clog2(32) = 10
   Hence, GA_CROSSOVER gets the next 10 bits, i.e. rand_42bit[28:19].

3. GA_MUTATION needs the following number of random bits:
   MUTAION_RAND_W = clog2(M_max)+DATA_W
   And for the chosen parameters:
   MUTATION_RAND_W = clog2(32)+6 = 11
   Hence, GA_MUTATION gets the next 11 bits, i.e. rand_42bit[39:29].

2.5.3. GA ALGO BLOCKS

All the algo blocks are instantiated inside ga_algo_top, that is inside ga_core (see section 2.1.1).

The main function of each algo block and the relations between them are shown in figure 18.

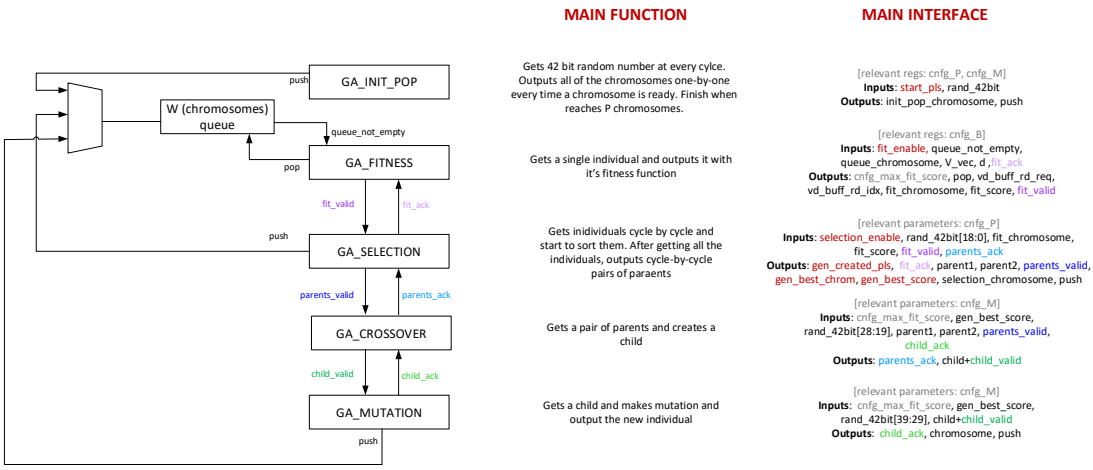


***Figure 18** : GA accelerator – algo block high level description*

Each algo blocks performs a stage in the genetic algorithm pipe, and the different blocks are activated and de-activated by ga_main_fsm.

At the beginning GA_INIT_POP is activated to create the initial population once, and pushes the chromosomes to the chromosomes queue. At its turn, after it has been activated by ga_main_fsm, GA_FITNESS start to pop chromosomes from the queue and calculate their fitness score one by one: At each clock cycle there is no more than one chromosome in GA_FITNESS, and only after the calculation of the first chromosome ends the second chromosome is popped into GA_FITNESS and so on.

After having its fitness score calculated, each chromosome enters GA_SELECTION block. GA_SELECTION block is divided into two steps: sort and select. The first step is sort and this is the step chromosomes from the fitness block arrives to: they are being sorted one-by-one at their arrival and placed inside W sorted pool, at a form that at all times W sorted pool is sorted.

Only after W sorted pool is full and all the chromosomes have been entered GA_SELECTION is in charge to send GA_MAIN_FSM that the current generation is done and ready. If there is a need for another generation then GA_MAIN_FSM will activate GA_SELECTION selection step.

Once GA_SELECTION step has been activated, P (cnfg_P) pairs of parents start to be created randomly one by one, and every pair is transferred into GA_CROSSOVER in order to create a new chromosome for the next generation. The new chromosome is transferred from GA_CROSSOVER to GA_MUTATION for a final transformation, and after the mutation block is done the new chromosome is ready.

From GA_MUTATION the chromosomes are transferred into the chromosomes queue so that they can have their fitness score calculated and to be sorted.

After all the new chromosomes have been sorted and are in W sorted pool, GA_SELCTION block informs GA_MAIN_FSM that another generation is sorted and ready, and as long as the parent selection is activated (by GA_MAIN_FSM) the above process/cycle will continue
(selection→crossover→mutation→queue→fitness→sorter).

After the last generation created GA_MAIN_FSM will deactivate GA_SELECTION parent selection step and also GA_FITNESS, and at this point the chromosomes in W sorted pool will be transferred to the chromosomes queue and wait there for GA_MAIN_FSM to activate GA_FITNESS (which will happen when a valid new set of {V,d} will arrive).

An important note is that as described above and as shown in figure 18, there are three origins to chromosomes queue (three sources that push chromosomes: GA_INIT_POP, GA_MUTATION, GA_SELECTION). Since they will not output push requests in parallel, a simple mux between the requests is enough (and no complex arbiter is required).

In addition, the fall of ga_eanble register is handled as software reset and sends all the algo block back to IDLE state.

2.5.3.1 GA_INIT_POP

GA_INIT_POP is the algo block that is in charge on the creation of the initial population.

The block is activated after the rise of ga_enable register by GA_MAIN_FSM who sends start signal for GA_INIT_POP. GA_INIT_POP generates the initial population of chromosomes which is completely random, and then goes back to IDLE until the next rise of ga_enable.

Interface:

signal name	input/output	Bits	pls/lvl	Description
Clk	Input	1		
Rstn	Input	1	Lvl	active low
Registers/Configurations IF				
sw_rst	Input	1	Pls	Falling edge of enable register (ga_enable)
cnfg_M	Input	M_MAX_W= clog2(M_max+1)= clog2(32+1)=6	Lvl	M is the number of weights in the filter (number of elements in V_vec/W_vec). **Range: [2,32]** (M_max=32)
cnfg_P	Input	P_MAX_W= clog2(P_max+1)= clog2(1024+1)=11	lvl	P is the number of chromosomes in each generation. **Range: [2,1024]** (P_max=1024)
Data IF: TOP/GA_MAIN_FSM				
start_pls	Input	1	Pls	Start signal from GA_MAIN_FSM. At rise of start_pls GA_INIT_POP starts to create random chromosomes.
rand_data	Input	42	Pls	42 bit of random data from GA_42BIT_RAND_GEN. Every clock cycle new random data is created.
Data IF: CHROMOSOME QUEUE				
queue_push	Output	1	Pls	Push command to the chromosomes queue. Active high.
queue_chromosome	Output	DATA_W*M_max	Pls	Value is valid only when queue_push is high. The random generated chromosome of the initial population.

***Table 5**: GA_INIT_POP interface*

Diagram:

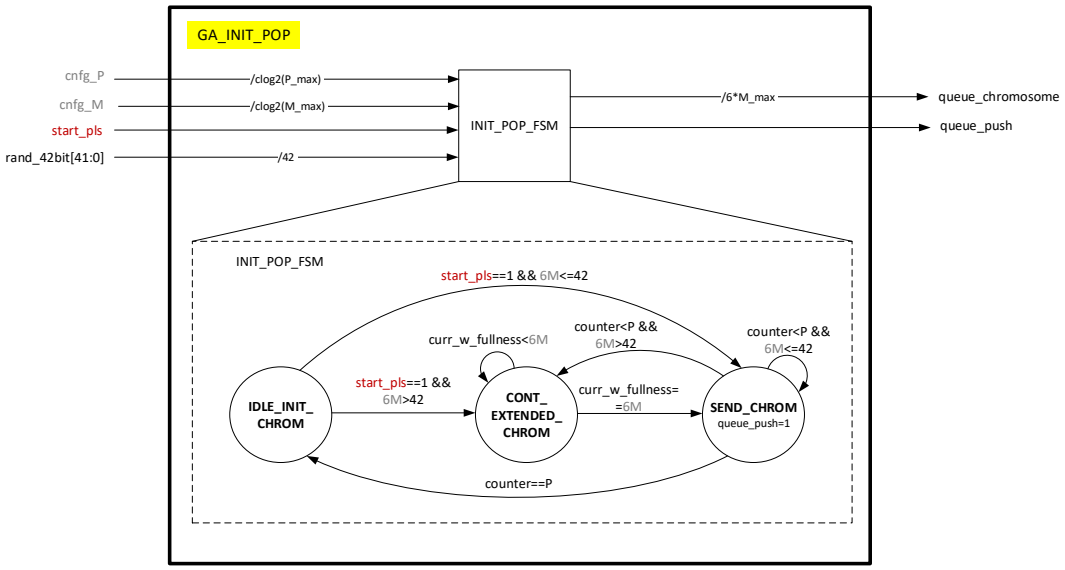


***Figure 19** : GA accelerator – GA_INIT_POP microarchitecture*

Detailed description:

GA_INIT_POP starts at IDLE state.

When start_pls rises GA_INIT_POP starts to create the P (cnfg_P) chromosomes one by one: it aggregates chunks of 42 bits from the random 42 bit input and counts until it reaches chromosome length (DATA_W*cnfg_M). Every time it reaches chromosome length it sends the new chromosome to the chromosomes queue, and continues to create next chromosome.

Overall, depends on the configurations there are two modes:

1. Mode of short chromosomes:
   Short chromosomes are chromosomes that their length is no more than 42bits, which allows to create new chromosome every clock cycle and to send push commands to the queue back-to-back.
   Since chromosome length is DATA_W*cnfg_M, and DATA_W=6, this mode is for cnfg_M≤7.
   In this mode after the rise of start_pls, the FSM goes to sent_chrom state and every clock cycle takes the 42bit random input and arrange them to be chromosome and send them to the queue. The FSM remains in this state until sending all of the P (cnfg_P) chromosomes, and then goes back to idle state until new start_pls will arrive.

2. Mode of extended (long) chromosomes:
   Extended (long) chromosomes are chromosomes that their length is larger than 42bits, which requires more than one clock cycle to create a single chromosome.
   Since chromosome length is DATA_W*cnfg_M, and DATA_W=6, this mode is for cnfg_M≥8.
   In this mode at the rise of start_pls the FSM starts to aggerate the first 42 bits for the chromosome and goes to cont_extended_chrom state. Every additional clock cycle in this state it will continue to aggregate chunks of additional 42bits until getting to the last chunk of 42 bits. At this point it will concatenate the required bits to the previous bits that were saved, finish to create the new chromosome, and will transit to send_chrom state in which it will send the push command to the queue. Afterwards it will move to cont_extended_chrom and create the next chromosome, and when it is done, it will go to send_chrom state again.
   Overall, the FSM will go from send_chrom state to cont_extended_chrom state interchangeably, until it will create all the P (cnfg_P) chromosomes. At this point it will go back to idle state and will stay there until new start_pls will arrive.

Time analysis:

The time needed per chromosome depends on the configuration of M, and at a single clock cycle $\frac{42}{DATA\_W} = \frac{42}{6} = 7$ new weights can be created. Thus, the number of clock cycles required to create a single chromosome is $\left\lceil \frac{cnfg\_M}{7} \right\rceil$.

2.5.3.2 GA_FITNESS

GA_FITNESS block is in charge of calculating fitness scores for the chromosomes.

It is the only client that pops chromosomes from the chromosomes queue. At all times that GA_FITNESS is activated (by GA_MAIN_FSM) it checks if the chromosomes queue is not empty. If the chromosomes queue is not empty, GA_FITNESS pops out a chromosome from the head of the queue and calculates its fitness score. When it's done it outputs the chromosome along with its fitness score, and when GA_SELECTION approves that it got the chromosome GA_FITNESS goes back to check if the chromosomes queue is not empty, and so on.

Interface:

signal name	input/output	Bits	pls/lvl	description
Clk	Input	1		
Rstn	Input	1	Lvl	active low
Registers/Configurations IF				
sw_rst	Input	1	Pls	Falling edge of enable register (ga_enable)
cnfg_B	Input	B_MAX_W= clog2(B_max+1)= clog2(64+1)=7	lvl	B is the size of the smoothing window, i.e. the number of timestamps in it. **Range: [2,64]** (B_max=64)
cnfg_max_fit_score	Output	FIT_SCORE_W	Lvl	Maximal fitness score possible for current registers configuration (worst fitness score). Unsigned fixed point.
Data IF: TOP/GA_MAIN_FSM				
fit_enable	Input	1	Lvl	Enable signal from the GA_MAIN_FSM. Active high. Only when fit_enable is high GA_FITNESS is activated, and when fit_enable falls GA_FITNESS goes back to IDLE.
vd_buff_d	Input	DATA_W	Pls	d[cnfg_B-1-vd_buff_rd_idx]. Value valid only at the clock cycle following vd_buff_rd_req assertion. Signed fixed point with DATA_W.
vd_buff_V_vec	Input	DATA_W*M_max	Pls	V_vec[cnfg_B-1-vd_buff_rd_idx]. Value valid only at the clock cycle following vd_buff_rd_req assertion. M_max elements, each is signed fixed point with DATA_W.
vd_buff_rd_req	Output	1	Pls	Read request for v&d buffer. Active high. The read data is guaranteed to arrive in 1 clock delay after the read request is asserted.
vd_buff_rd_idx	Output	clog2(B_max)= clog2(64)=6	Pls	Value valid only when vd_buff_rd_req is high. The index in v&d buffer to read from. Range: [0,cnfg_B-1]. Index 0 is the oldest set {V[cnfg_B-1],d[cnfg_B-1]}, and index cnfg_B-1 is the newest set {V[0],d[0]}.

Data IF: CHROMOSOME QUEUE				
queue_not_empty	Input	1	Lvl	At all times indicates about the status of the queue: empty or not. Active high. Only when queue_not_empty is asserted pop requests can be performed.
queue_chromosome	Input	DATA_W*M_max	Pls	Value is valid only one clock cycle after queue_pop has been asserted. This is the chromosome to calculates its fitness score. M_max elements, each is signed fixed point with DATA_W.
queue_pop	Output	1	Pls	Pop command to the chromosomes queue. Active high. The read data is guaranteed to arrive in 1 clock delay after the read request is asserted.
Data IF: GA_SELECTION				
fit_ack	Input	1	Pls	Active high. Acknowledgment signal from GA_SELECTION that it has received fit_chrom and fit_score from GA_FITNESS. Value is relevant only when fit_valid is high.
fit_valid	Output	1	Lvl	Active high. Rises at the end of a fitness calculation for a single chromosome and indicates there are valid chromosome and score for GA_SELECTION to use. Remains high and steady along with the data it validates until receiving fit_ack from GA_SELECTION, and falls when receiving the ack.
fit_chrom	Output	DATA_W*M_max	Lvl	Valid only when fit_valid is asserted. This is the chromosome itself. M_max elements, each is signed fixed point with DATA_W.
fit_score	Output	FIT_SCORE_W	Lvl	Valid only when fit_valid is asserted. This is the fitness score of the chromosome in fit_chrom. unsigned fixed point.

***Table 6**: GA_FITNESS interface*

- For default values FIT_SCORE_W=18, and it's a function of DATA_W, B_max, and the precision required for tanh.
  Overall, the formula is (will be clarified in the detailed description segment):

  FIT_SCORE_W =

  = clog2(2*B_max+1)+2*DATA_FRACT_W = clog2(2*64+1)+2*5 = 18

  When:

  FIT_SCORE_INT_W = clog2(2*B_max+1) = 8 [No sign bit]

  FIT_SCORE_FRACT_W = 2*DATA_FRACT_W = 10

Diagram:

In figure 20 a top view of GA_FITNESS is presented.

In figures 21-23 each one of the parts is zoomed-in: top fitness, fitness FSM and fitness algo.

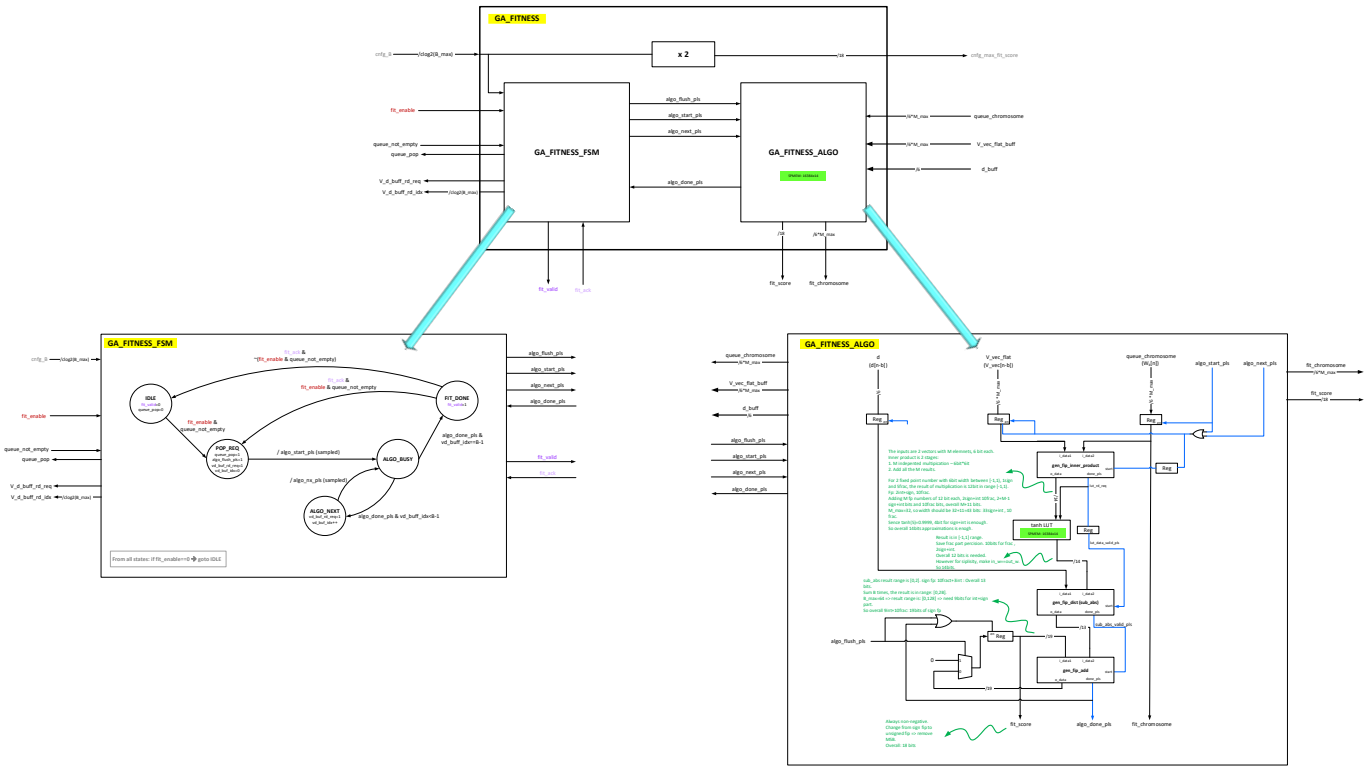


***Figure 20** : GA accelerator – GA_FITNESS microarchitecture – top view*

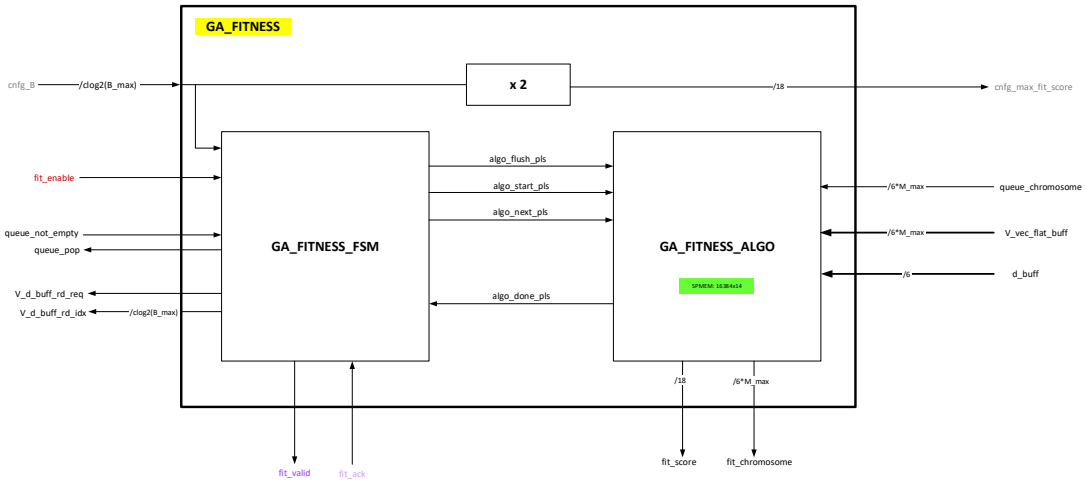


***Figure 21** : GA accelerator – GA_FITNESS microarchitecture – top*

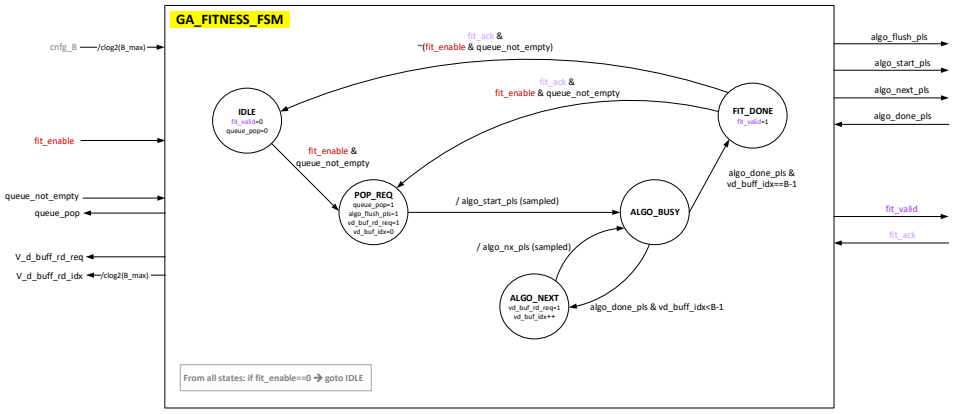


Figure 22 *: GA accelerator – GA_FITNESS microarchitecture – FSM (GA_FITNESS_FSM)*

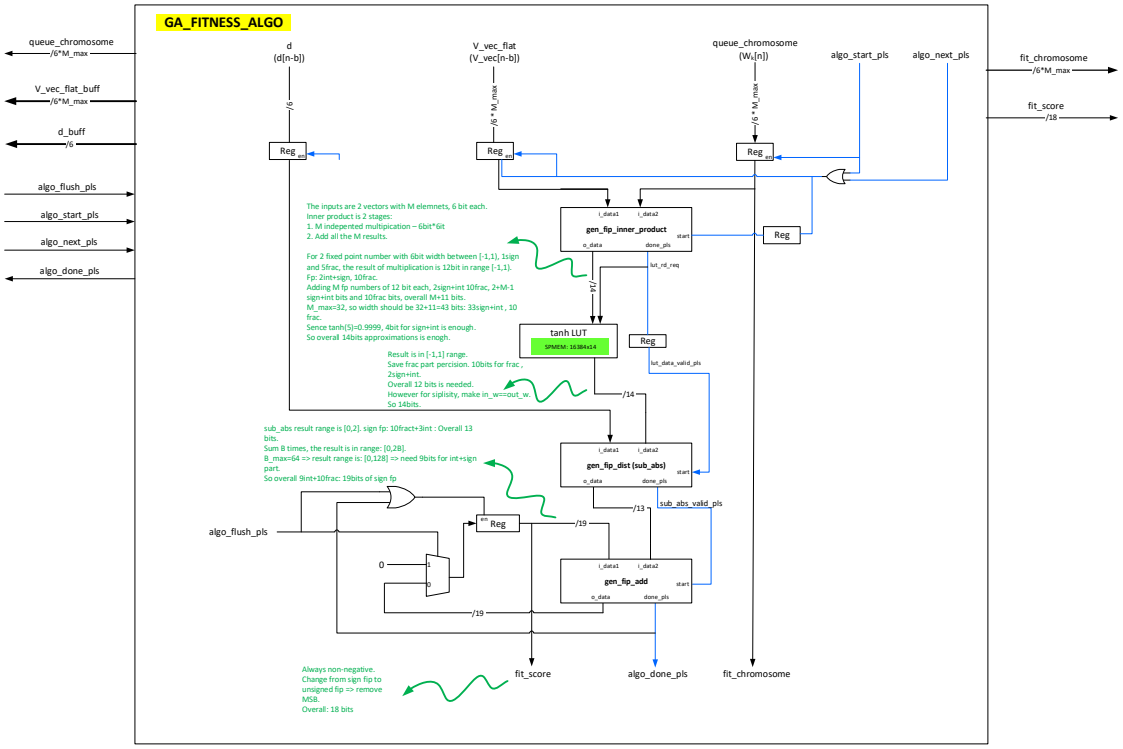


Figure 23 *: GA accelerator – GA_FITNESS microarchitecture – algo (GA_FITNESS_ALGO)*

Pipeline diagram:

GA_FITNESS_ALGO consists of a pipe in which the chromosome goes throw.

For simplicity the pipe is unorthodox and at each clock cycle only one element of the pipe is active (like un-piped system). However, in the future ideas section a proposal for a real pipe per chromosome and a pipe that will allow more than one chromosome inside GA_FITNESS is offered for higher efficiency/throughput.

The walk-throw for a single chromosome in the GA_FITNESS algo pipe is presented in figure 24.

	FETCH {V[b],D[b]} FROM VD BUFF (1clk)	SAMPLE (1clk)	INNER PRODUCT: V dot chromosome (8clk)	Tanh LUT (1clk)	DISTANCE TO d(b) (1clk)	ADD TO ALL PREVIOUS VD (1clk)
(1)	FETCH VD	Sample	INNER PRODUCT	LUT	DIST	ADD
(2)	FETCH VD	Sample	INNER PRODUCT	LUT	DIST	ADD
(3)	FETCH VD	Sample	INNER PRODUCT	LUT	DIST	ADD
(4)	FETCH VD	Sample	INNER PRODUCT	LUT	DIST	ADD
(5)	FETCH VD	Sample	INNER PRODUCT	LUT	DIST	ADD
(6)	FETCH VD	Sample	INNER PRODUCT	LUT	DIST	ADD
(7)	FETCH VD	Sample	INNER PRODUCT	LUT	DIST	ADD
(8)	FETCH VD	Sample	INNER PRODUCT	LUT	DIST	ADD
(9)	FETCH VD	Sample	INNER PRODUCT	LUT	DIST	ADD
(10)	FETCH VD	Sample	INNER PRODUCT	LUT	DIST	ADD
(11)	FETCH VD	Sample	INNER PRODUCT	LUT	DIST	ADD
(12)	FETCH VD	Sample	INNER PRODUCT	LUT	DIST	ADD
(13)	FETCH VD	Sample	INNER PRODUCT	LUT	DIST	ADD

***Figure 24** : GA accelerator – GA_FITNESS pipeline diagram*

Overall per chromosome the algo pipe is performed B (cnfg_B) times, each time for different {V_vec,d} set. The result of the calculation is presented in equation 10 from section 1, and in GA_FITNESS names terminology for a single chromosome:

$$\textbf{(11)} \qquad fit\_score = \sum_{b=0}^{B-1} \left| \tanh[queue\_chromosome \cdot V\_vec(b)] - d(b) \right|$$

The walk-through of one iteration for a single {V_vec(b),d(b)} is shown in figure 24: At first there is a read request from v&d buffer, which takes 1 clock cycle, and the cycle after it they are sampled into GA_FITNNES_ALGO because they need to stay stable for more than one clock cycle (time optimization could be made here but for simplicity did not). Afterwards, inner product computation occurs (V dot chromosome) which takes 8 clock cycles for the default parameter M_MAX = 32 (see section 2.5.4), and then the result acts as an address for tanh LUT, which is implemented with SRAM so the output is valid in one clock latency. Afterwards the result from the LUT is compared to d[b] and the calculation of the distance between them takes one clock cycle. Eventually the result is added to all the previous results, which takes additional clock cycle.

Overall, per {V_vec(b),d(b)} it takes 13 clock cycles. For a single chromosome full calculation (all of the smoothing window), it takes 13*cnfg_B clock cycles.

Detailed description:

GA_FITNESS communicates with several units:

1. GA_MAIN_FSM – in charge of enable signal to GA_FITNESS.
2. CHROMOSOMES QUEUE – GA_FTINESS pulls chromosomes from it.
3. VD BUFFERS – GA_FITNESS reads sets of {V_vec,d} from it.
4. GA_SELECTION – GA_FITNESS sends to GA_SELECTION chromosomes and their fitness score.

GA_FITNESS consists of two functional units: FSM and algo pipe. GA_FITNESS_FSM is in charge of controlling when actions will take place, to send requests to chromosomes queue and to vd buffers, and to send valid to GA_SELECTION once the calculation is done. GA_FITNESS_ALGO is in charge of the calculation itself: with control signals from GA_FITNESS_FSM it starts a single calculation for single {V_vec,d} set.

Overall, the process from GA_FITNESS_FSM point of view is as follow:

1. GA_FITNESS_FSM starts at idle state. No requests to chromosomes queue/vd buffers are sent, and fit_valid is deasserted.
2. Once fit_enable from GA_MAIN_FAIM rises GA_FITNESS_FSM starts to check whether or not chromosomes queue is not empty. Once GA_FITNESS_FSM detects there are elements in the queue it goes out of IDLE state and moves to POP_REQ state.
3. At POP_REQ state the pop request to the chromosomes queue is high and in addition read request to vd buffers is also high, to get the first set of {V_vec,d}.
   3.1. The rise of fit_enable guarantees that vd buffers data is ready and will be stable for the entire calculation, hence there is no need for a special check from GA_FITNESS and the read request will always succeed.
4. At the next clock cycle the data will arrive from the queue and the buffer and GA_FITNESS_FSM will send start pulse to GA_FITNESS_ALGO and go to ALGO_BUSY state.
5. GA_FITNESS_FSM will stay at ALGO_BUSY state until GA_FITNESS_ALGO will send the FSM that it is finished, i.e. until algo_done_pls is asserted.
6. The rise of algo_done_pls means that GA_FITNESS_ALGO has finished to calculate $|\tanh[queue\_chromosome \cdot V\_vec(b)] - d(b)|$ for the current chromosome and {V_vec,d}. As a result GA_FITNESS_FSM goes to ALGO_NEXT state: it adds "+1" to vd index counter and sends read request to vd buffer with the new index.
7. At the next clock cycle the data from vd buffer will arrive and simultaneously GA_FITNESS_FSM will go to ALGO_BUSY state and send next pulse to GA_FITNESS_ALGO.

7.1. The difference between algo_start_pls and algo_nx_pls is that algo_start_pls indicates there is a new chromosome valid (with the first {V_vec,d} valid for the calculation), and algo_nx_pls indicates that the current chromosome is still not done and that there is a new set of {V_vec,d} to continue the calculation with.

8. Again, GA_FITNESS_FSM will remain at ALGO_BUSY state until GA_FITNESS_ALGO will send algo_done_pls.
9. Once algo_done_pls will rise again GA_FITNESS_FSM will check vd buffer index counter, and according to it will decide if to go again to ALGO_NEXT state and do additional read request with the next index, or to finish and to go to FIT_DONE state.
10. Overall the loop between ALGO_BUSY and ALGO_NEXT will be done B (cnfg_B) times, and B (cnfg_B) read requests will be done to vd buffer to cover all of the smoothing window.
    10.1. As mentioned under the pipeline diagram (figure 24) one loop takes 13 clock cycles (for the default parameter M_MAX = 32).
11. After GA_FITNESS_ALGO will finish the calculation for the last {V_vec,d} set, GA_FITNESS_FSM will go to FIT_DONE state.
12. At FIT_DONE state valid signal is send to GA_SELECTION and GA_FITNESS_FSM will stay at this state with fit_valid high until it will receive ack from GA_SELECTION.
13. After receiving ack from GA_SELECTION the FSM will get out of FIT_DONE state: chromosomes queue is empty it will go back to IDLE, and if it is not empty it will go to POP_REQ state.
14. Like mentioned before, at POP_REQ state pop and read requests are sent to the queue and to the buffer. In addition, flush request is sent to GA_FITNESS_ALGO in order to reset the fitness score register to make the sum start from 0 since it's a new chromosome.

In addition, at each state if fit_enable falls then GA_FITNESS_FSM goes back to IDLE state.

GA_FITNESS_ALGO point of view:

1. Once GA_FITNESS ALGO receives algo_start_pls it knows there are valid chromosome, V_vec and d at its interface and it samples them.
2. The first step in the calculation is calculating the inner product between the chromosome and V_vec: $queue\_chromosome \cdot V\_vec(current\_b)$
    2.1. V_vec and the chromosome are at the same width: $DATA\_W \cdot M\_MAX$ bits, and for default parameters $DATA\_W \cdot M\_MAX = 6 \cdot 32 = 192$ bits.

2.2. The inner product calculation requires M_max multiplications and then sum all the results.

2.3. Every element for the multiplication is signed fixed point with DATA_W bits: DATA_INT_W bits for sign and int part and DATA_FRACT_W for fraction part. Thus, a result of one multiplication requires $2 \cdot DATA\_W$ bits: $2 \cdot DATA\_INT\_W$ for sign and integer part and $2 \cdot DATA\_FRACT\_W$ bits for fraction part.
Afterward, sum M_max numbers requires M_max-1 sum operations and the results consists of the same fraction part width ($2 \cdot DATA\_FRACT\_W$) and additional M_max-1 bits for int part, over all the result is $2 \cdot DATA\_W + M\_MAX - 1$ bits: $2 \cdot DATA\_INT\_W + M\_MAX - 1$ bits for sign+int part and $2 \cdot DATA\_FRACT\_W$ bits for fraction part.

3. As explained, the required width for the inner product result is $2 \cdot DATA\_W + M\_MAX - 1$ bits which for default parameters is $2*6+32-1=43$ bits: 33 bits for sign+int part and 10 for fraction part. Since the result is used later as argument to tanh LUT and 43 bits input for a LUT required large memory, and because tanh(7)=0.9999983 an approximation is made: 4 bits for sign+int part are used: every result higher than 7 or smaller than -8 will be rounded to 7 (or -8, respectively) and the result of their tanh will be the accurate enough.
So, overall the result of the inner product is chosen to be 14bits.
4. When the result of the inner product is done, its hyperbolic tangent is calculated: $\tanh[queue\_chromosome \cdot V\_vec(current\_b)]$.
The calculation is done using LUT which consists of SRAM with address width of 14 bits and line width of 14 bits, hence memory in the size of $2^{14} \times 14 = 16384 \times 14$ bits. The LUTs table appears in appendix A.
The result (data in memory) is 14 bits as mentioned, and is also signed fixed point: 4 bits for sign and int part, and 10 bits for fraction part.
Since the LUT uses SRAM the result is valid one clock cycle after the read request (after inner product result is valid).

4.1. Note that an optimization could have been made here: the memory size could be reduced to half due to tanh symmetry. However, for implementation simplicity it has been decided not to do so.

5. The next step is to calculate the distance between d(b) and the result of the hyperbolic tangent:
$|\tanh[queue\_chromosome \cdot V\_vec(current\_b)] - d(current\_b)|$
For uniformity and simplicity, the result of the distance is also signed fixed point. Since by definition $|d| \le 1$ and also $\forall x : |\tanh(x)| \le 1$, the maximum distance is 2 hence it requires 3 bits for sign+int part. For fractional part it requires $\max(10, 2 \cdot DATA\_FRACT\_W)$, and since for default parameters $2 \cdot DATA\_FRACT\_W = 10$ values overall the distance result is 13 bits: 3 for sign+int and 10 for fraction.

6. The final step is the sum operation: to sum the new result to all the previous results: $\sum_{b=0}^{current-b} |\tanh[queue\_chromosome \cdot V\_vec(b)] - d(b)|$. The sum of the previous results is saved in a register (a.k.a accumulative sum register), and the new result will be added to it.
   6.1. Since overall the maximum number of elements that will be summed together is B_MAX, and the distance result is in range [0,2], the sum result will eventually be in the range of [0,2B], i.e. the output cnfg_max_fit_socre=2*cnfg_B_max. Hence, the total number of bits required for sign+int part is 1+clog2(2B+1). In order to save the precision the fraction part will remain untouched and will be $\max(10, 2 \cdot DATA\_FRACT\_W)$.

   So, overall the sum result in singed fixed point is: $1 + c\log 2(2 \cdot B\_\max + 1) + \max(10, 2 \cdot DATA\_FRACT\_W)$ bits. For default values it is $1 + c\log 2(2 \cdot 64 + 1) + \max(10, 2 \cdot 5) = 9 + 10 = 19$ bits.
7. After the above calculation is done (steps 2-6) GA_FITNESS_ALGO sends to GA_FITNESS_FSM that it's done: GA_FITNESS_ALGO rises algo_done_pls and waits for the next instruction from GA_FITNESS_FSM.
   7.1. Eventually the interface towards GA_SELECTION is the chromosome that has been sampled in the last algo_start_pls and its fitness score. The fitness score, fit_score, is unsigned fixed point and is the value in the accumulate sum register not including the MSB (sign bit), hence for default values its 19-1=18 bits.
8. At most times the next instruction will be algo_nx_pls: the meaning of it, unlike algo_start_pls, that a new chromosome shouldn't be sampled but only new V_vec and d. After the sampling stage is done, steps 2-7 will happen the same way.
9. Third and last kind of signal GA_FITNESS_ALGO can receive from GA_FITNESS_FSM is algo_flush_pls that implies that a new chromosome is about to arrive. In this case the result in the accumulative sum register will be changed by force to zero, in order to clean the pipe for the new chromosome.
   9.1. Note that until algo_flush_pls is not asserted the result in the accumulative sum register stays stable between activations of GA_FITNESS_ALGO, hence when GA_MAIN_FSM goes to FIT_DONE state and outputs fit_valid towards GA_SELECTION, fit_chromosome and fit_score remains stable (and hence valid) as well.

Time analysis:

The time-consuming unit is GA_FITNESS_ALGO which takes 13 clock cycles per set of {V,d} and 13*cnfg_B clock cycles for a single chromosome.

2.5.3.3 GA_SELECTION

GA_SELECTION block is the largest of all the algo units (area) and the most time consuming of all the algo units. GA_SELECTION is in charge of both sorting the chromosomes and select random pairs of parents that will be transferred to GA_CROSSOVER for creation of the next generation.

GA_SELECTION communicates with the following units: GA_FITNESS – gets chromosomes from it, GA_CROSSOVER – gives him pairs of parents, and GA_MAIN_FSM – informs it when a generation is done and sorted (GA_SELECTION is the main algo block to communicate with GA_MAIN_FSM). In addition, GA_SELECTION is also a source that pushes data to the chromosomes queue after the last generation was created.

Interface:

signal name	input/output	Bits	pls/lvl	description
Clk	Input	1		
Rstn	Input	1	Lvl	active low
Registers/Configurations IF				
sw_rst	Input	1	Pls	Falling edge of enable register (ga_enable)
cnfg_P	Input	P_MAX_W= clog2(P_max+1)= clog2(1024+1)=11	lvl	P is the number of chromosomes in each generation. **Range: [2,1024]** (P_max=1024)
Data IF: TOP/GA_MAIN_FSM				
rand_data	Input	RAND_ W	Pls	Random bits from GA_42BIT_RAND_GEN.
create_new_gen_req_pls	Input	1	Pls	Sent from GA_MAIN_FSM to GA_SELECTION block as a response to gen_created_pls, when a new generation needs to be created (Start signal for parent selection process).
stop_create_gens_req_pls	Input	1	Pls	Sent from GA_MAIN_FSM to GA_SELECTION block as a response to gen_created_pls when it was the last generation created (i.e. when no new generation needs to be created). (Start signal for sending chromosomes from GA_SELECTION to chromosomes queue).
gen_created_pls	Output	1	Pls	Sent from GA_SELECTION to GA_MAIN_FSM and indicates that a single generation has been generated and sorted.
gen_best_chrom	Output	DATA_W*M_max	Pls	Sent from GA_SELECTION to GA_MAIN_FSM and is valid when gen_created_pls is high. This is the best chromosome at the generation that was created. M_max elements, each is signed fixed point with DATA_W.

Data IF: ALGO_TOP,CHROMOSOME QUEUE				
gen_best_score	Output	FIT_SCORE_W	Lvl	Sent from GA_SELECTION to GA_CROSSOVER and GA_MUTATION and represents the current generation best score. Unsigned fixed point.
queue_push	Output	1	Pls	Push command to the chromosomes queue. Active high.
queue_chromosome	Output	DATA_W*M_max	Pls	Value is valid only when queue_push is high. The last generation chromosomes that are outputted from the sorter.
Data IF: GA_FITNESS				
fit_valid	Input	1	Lvl	Active high. From GA_FITNESS to GA_SELECTION. Rises at the end of a fitness calculation for a single chromosome and indicates there are valid chromosome and score for GA_SELECTION to use. Remains high and steady along with the data it validates until receiving fit_ack from GA_SELECTION, and falls when receiving the ack.
fit_chrom	Input	DATA_W*M_max	Lvl	Valid only when fit_valid is asserted. This is the chromosome itself. M_max elements, each is signed fixed point with DATA_W.
fit_score	Input	FIT_SCORE_W	Lvl	Valid only when fit_valid is assreted. This is the fitness score of the chromosome in fit_chrom. unsigned fixed point.
fit_ack	Output	1	Pls	Active high. Acknowledgment signal from GA_SELECTION to GA_FITNESS that it has received fit_chrom and fit_score from GA_FITNESS. Value is relevant only when fit_valid is high.
Data IF: GA_CROSSOVER				
parents_ack	Input	1	Pls	Active high. Acknowledgment signal from GA_CROSSOVER to GA_SELECTION that it has received parent1 and parent2 from GA_SELECTION. Value is relevant only when parents_valid is high.
parents_valid	Output	1	Lvl	Active high. From GA_SELECTION to GA_CROSSOVER. Rises once a pair of two chromosomes are chosen to be parents and are ready to go to next stage (crossover). Remains high and steady along with the data it validates until receiving parents_ack from GA_CROSSOVER, and falls when receiving the ack.

parent1	Output	DATA_W*M_max	Lvl	Valid only when parents_valid is asserted. This is the parent that have been selected from the better half of the chromosomes sorted pool. M_max elements, each is signed fixed point with DATA_W.
parent2	Output	DATA_W*M_max	Lvl	Valid only when parents_valid is asserted. This is the parent that have been selected from the entire chromosomes sorted pool. M_max elements, each is signed fixed point with DATA_W.

***Table 7**: GA_SELECTION interface*

- For default values RAND_W=19, and it's a function of P_max.
  Overall, the formula is (will be clarified in the detailed description segment):

  RAND_W =

  = clog2(P_max)+ clog2(P_max/2) = clog2(1024)+ clog2(512) = 19

Diagram:

In figure 25 a top view of GA_SELECTION is presented.

In figures 26-29 each one of the parts is zoomed-in: top selection, selection FSM, selection sorter and selection parents.

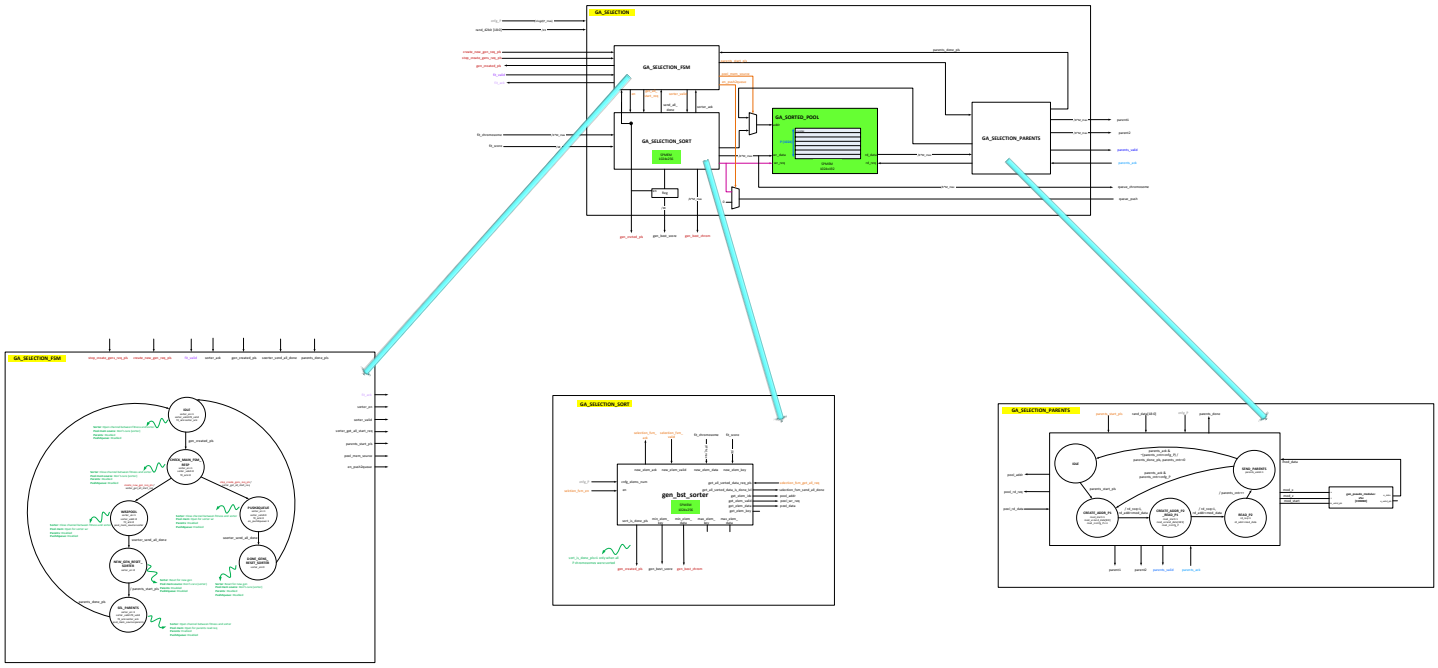


***Figure 25** : GA accelerator – GA_SELECTION microarchitecture – top view*

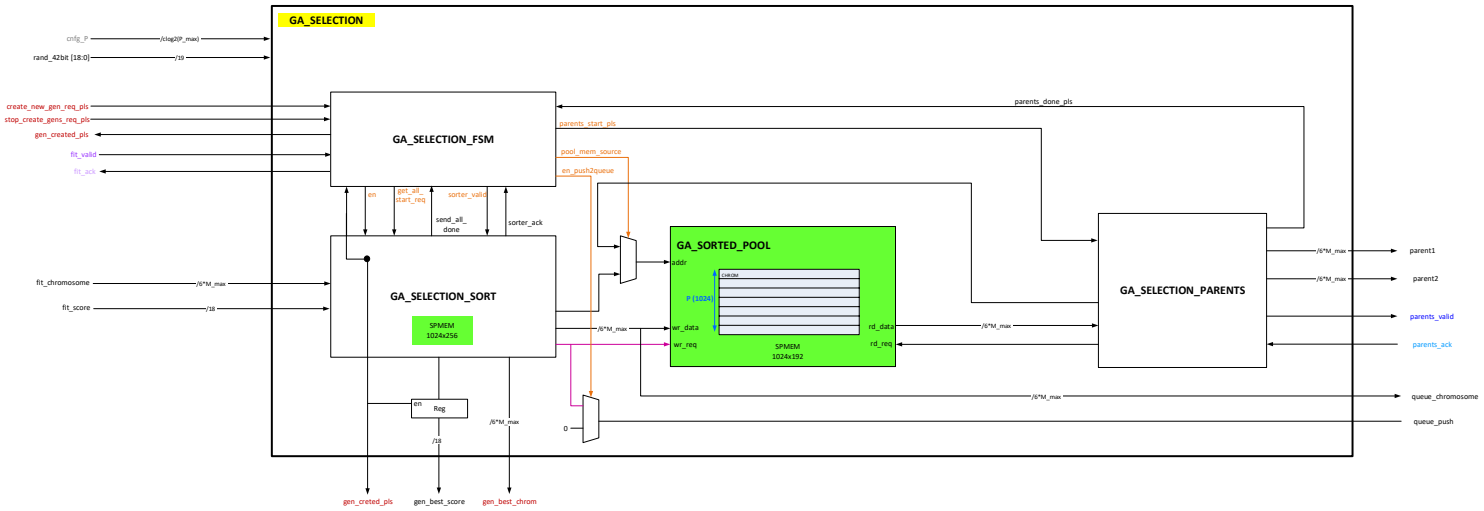


Figure 26 *: GA accelerator – GA_SELECTION microarchitecture – top*

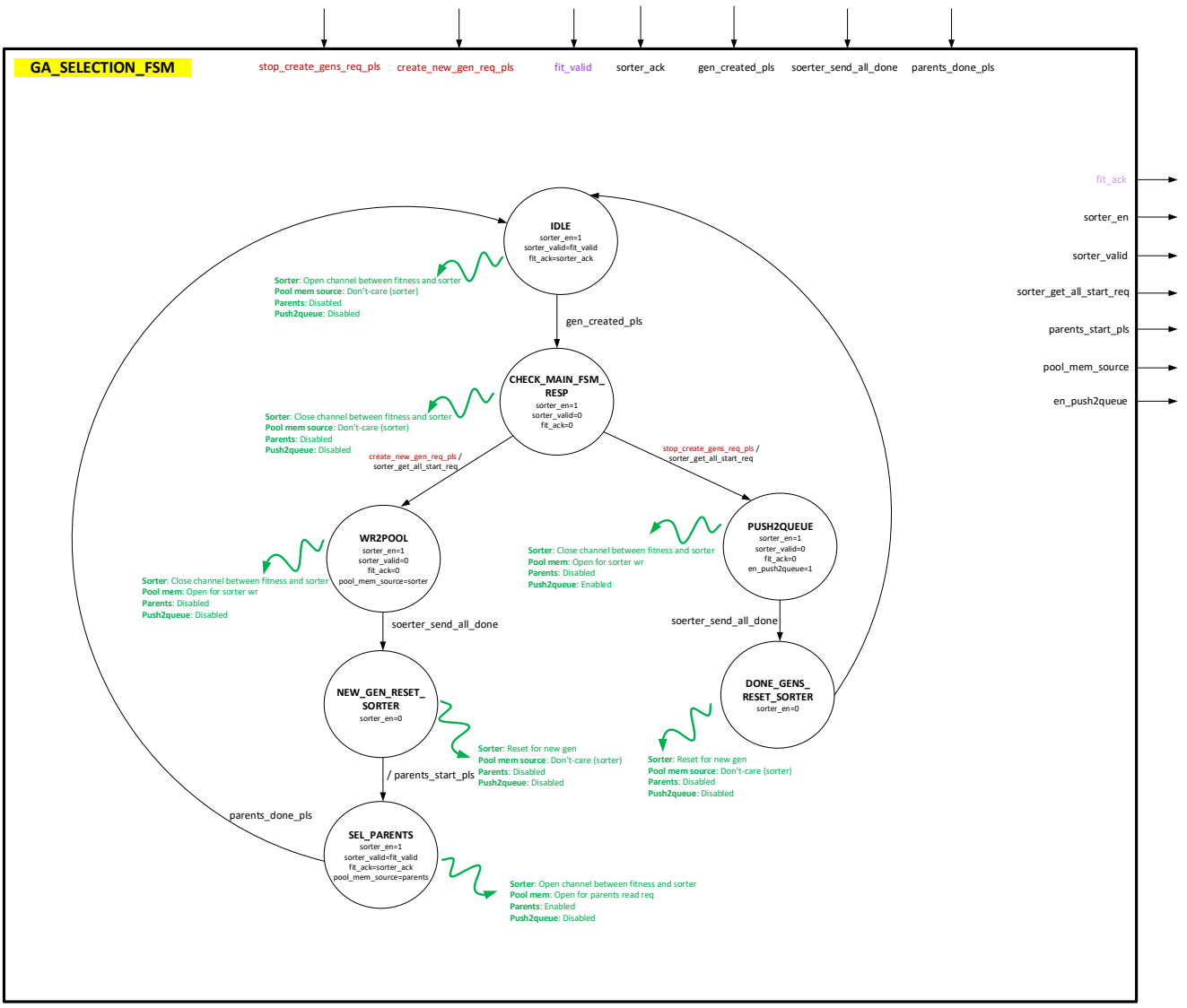


Figure 27 *: GA accelerator – GA_SELECTION microarchitecture – fsm (GA_SELECTION_FSM)*

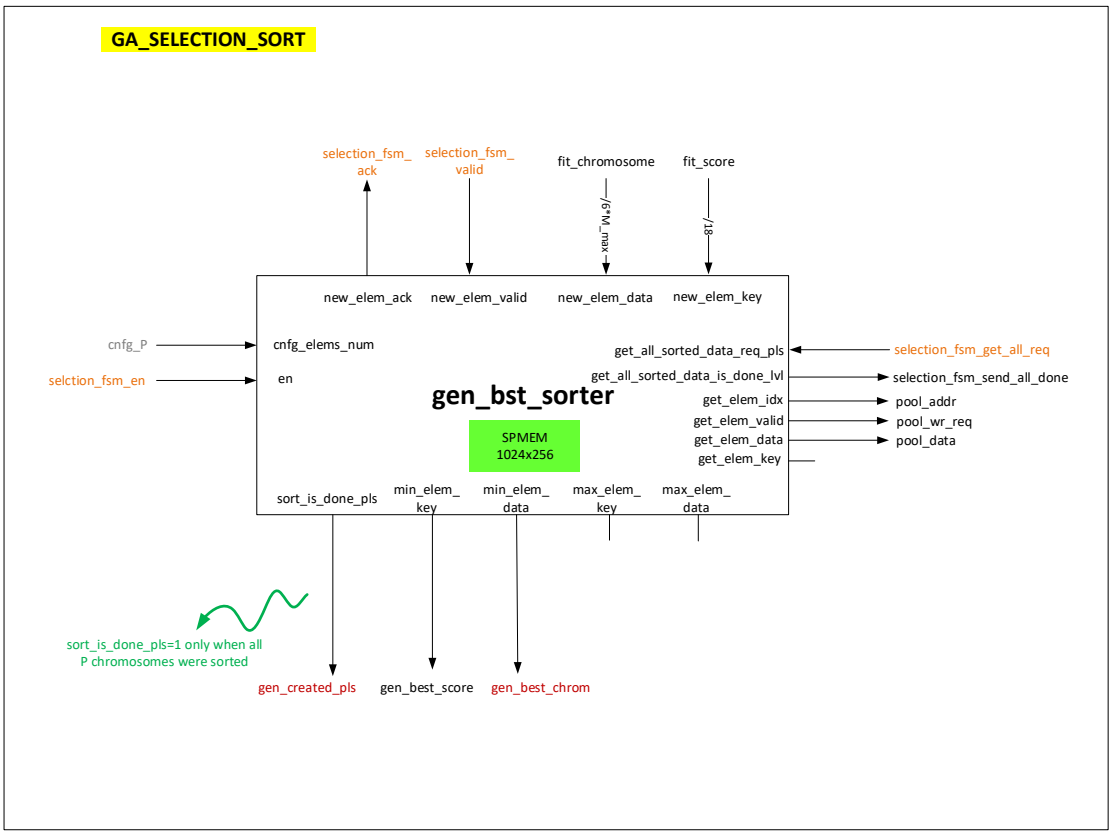


***Figure 28** : GA accelerator – GA_SELECTION microarchitecture – sort (GA_SELECTION_SORT)*

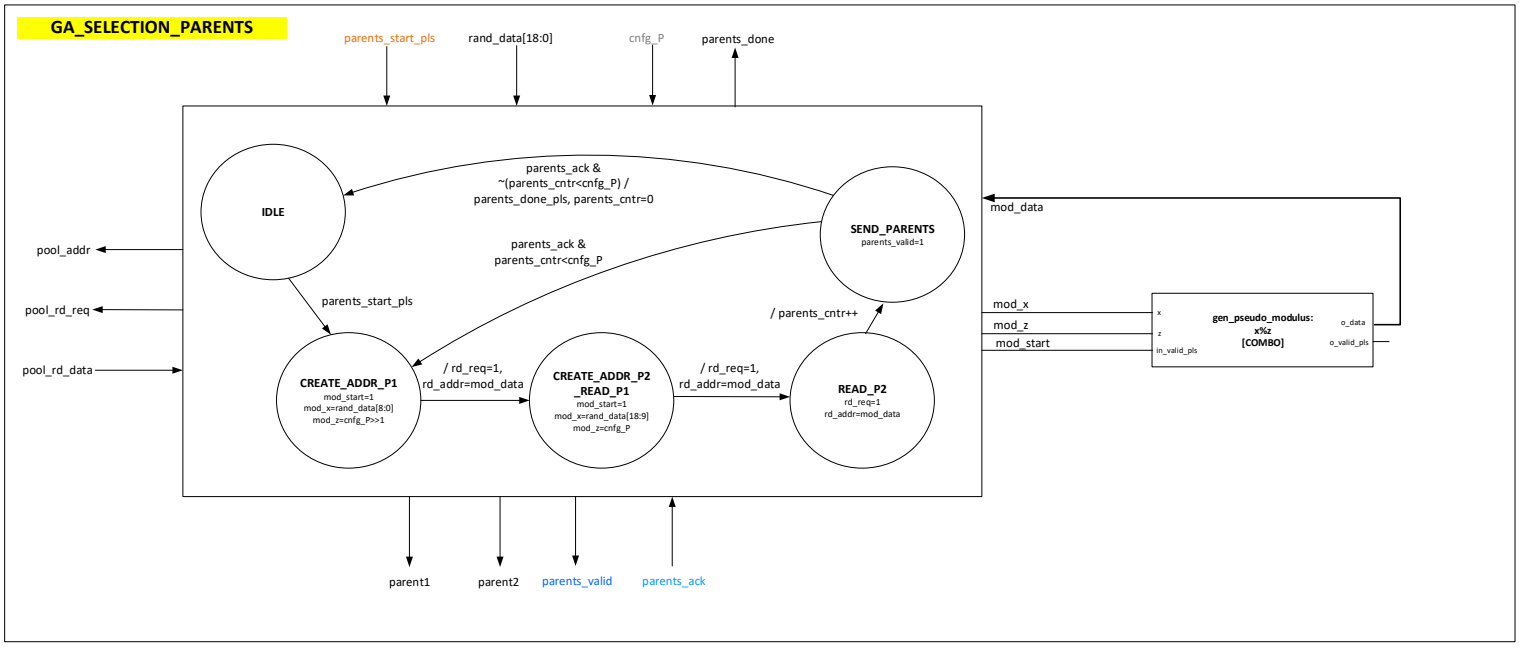


***Figure 29** : GA accelerator – GA_SELECTION microarchitecture – parent selection (GA_SELECTION_PARENTS)*

Detailed description:

As mentioned before, GA_SELECTION has two main stages:

1. Sort current generation chromosomes. This is done by GEN_BST_SORTER (see section 2.5.5), and its inputs are controlled by GA_SELECTION_FSM. Once done, inform GA_MAIN_FSM and send it the best chromosome.
2. Depends on GA_MAIN_FSM response, start to select random pairs of parents for the next generation, or start to push the chromosomes to W queue for the next time the calculation of W[n+1] will take place.

The transitions between those phases are controlled by GA_SELECTION_FSM (figure 27).

GA_SELECTION_SORTER instance:

Full description of the sorter is in section 2.5.5.

The sorter's elements are the chromosomes, when the value is the chromosome itself and the fitness score is the key by which the sort order is set. The sorter needs to be configured in advanced with the number of elements to be sorted, in this case its cnfg_P elements, and it has an enable signal that comes from GA_SELECTION_FSM.

The sorter is a parametric block and the instance parameters are as follow:

- **KEY_W** = FIT_SCORE_W = 18
- **VALUE_W** = CHROM_MAX_W = DATA_W*M_max = 192
- **MAX_ELEM_NUM** – P_MAX = 1024

In addition, GEN_BST_SORTER has an internal SRAM in which the sorted chromosomes are placed, with size MAX_ELEM_NUM x ROW_W, and in current case 1024 x 256

- The formula for row width size of the SRAM is presented in section 2.5.5 and it is:
  $$1 + KEY\_W + VALUE\_W + 4 \cdot (1 + c\log 2(MAX\_ELEM\_NUM)) + 1$$
  So for this instance parameters:
  $$1 + KEY\_W + VALUE\_W + 4 \cdot (1 + c\log 2(MAX\_ELEM\_NUM)) + 1 =$$
  $$= 1 + FIT\_SCORE\_W + DATA\_W \cdot M + 4 \cdot (1 + c\log 2(P)) + 1 =$$
  $$= 1 + 18 + 192 + 4 \cdot (1 + c\log 2(1024)) + 1 = 1 + 18 + 192 + 4 \cdot (1 + 10) + 1 = 256$$

Once the sorter has been enabled, it works in two phases:

1. **First phase is the sort phase**: gets elements in serial manner, from GA_FITNESS, and counts them (as mentioned, fit_score is the key and fit_chrom is the value). Once the last element arrives (the P^{th} element) the sorter will not receive new elements and after the sorter finds the element's proper place it sends done pulse to GA_MAIN_FSM to indicate that the current generation was created. In addition, it sends it the best chromosome, i.e. the chromosome with the lowest fitness score, which for the last generation will be outputted by GA_MAIN_FSM as W[n+1].
   Lastly, for GA_CROSSOVE and GA_MUTATION future usage (will be explained in their sections), it outputs the best score (minimal) that have been received, which is saved in a register that will stay steady until the next generation will be done, i.e. for the entire time GA_CROSSOVER and GA_MUTATION will handle the current generation.
2. **Second phase is extraction phase**: will start only after the sort phase is done, by control pulse signal from GA_SELECTION_FSM. Once this signal is asserted all the sorted chromosomes will be copied one by one to the sorted pool which is an SRAM in the size of P_MAX x CHROMOSOME_MAX_W (1024x192). The SRAM needs to be available for requests with the sorter as it solely master at this stage.
   Once all the cnfg_P elements have been written (by order) inside the sorted pool SRAM the sorter sends done indication to GA_SELECTION_FSM and stops working until the next time GA_SELECTION_FSM will deassert the enable signal (acts as sw_rst) and will assert it again. At this point the sorter will return to the first stage for new set of elements.

<u>The process from GA_SELECTION_FSM point of view:</u>

At idle state, i.e. by default, the sorter is enabled and the channel between the sorter and GA_FITNESS is open: fit_valid from GA_FITNESS arrives to the sorter and ack from the sorter returns to GA_FITNESS. The FSM will remain at this state until the sorter will send it that all the chromosomes of this generation have been received (the sorter is in charge of the counting). At this point the FSM will go to CHECK_MAIN_FSM_RESP since in parallel this information has been send to GA_MAIN_FSM, and the next step for GA_SELECTION depends at GA_MAIN_FSM decision. The response of GA_MAIN_FSM can be one of the following two: request for a new generation (left branch in figure 27) or request to finish the creation of the generations (right branch in figure 27).

The right branch is the short branch that will take place only at the last (cnfg_G^{th}) generation. In this case the FSM will go to PUSH2QUEUE state and will both active

the extraction phase of the sorter and open the channel of push request to the chromosomes queue. In this case the chromosomes that are outputted from the sorter in the extraction phase will be pushed to the chromosomes queue. Once the sorter is done, the FSM will go to DONE_GENS_RESET_SORTER state and will close the channel to the queue. In addition, it will reset the sorter by deasserting its enable for one clock cycle, and then will go back to IDLE state. As remembered, at idle state the channel between the sorter and GA_FITNESS is open, hence GA_SELECTION_FSM has enabled the new generation chromosomes to enter the sorter the next time a new set of {V,d} will arrive, as required.

- Important thing to notice here that in case {V,d} arrives before the extraction phase of the sorter is done, the first chromosome will be entered to GA_FITNESS but fit_valid will not be acked by the sorter until GA_SELECTION_FSM will go back to idle (i.e. the sorter will be reset and enable again and the channel to it will be opened).

The left branch is the common one and it will take place every time a new generation should be created (generations 1 to cnfg_G-1). If GA_MAIN_FSM sends request for a new generation GA_SELECTION_FSM will go from CHECK_MAIN_FSM_RESP state to WR2POOL state and will send extraction request to the sorter, and will set the master of the sorted-pool SRAM to be the sorter. Once the sorter is done GA_SELECTION_FSM goes to NEW_GEN_RESER_SORTER and deasserts sorter enable signal in order to reset it. This is allowed because all of the data is now in sorted pool memory and will not be lost, and this is done at this point to increase performance (explained below).
The next clock cycle the FSM sends to the selection parents unit, GA_SELECTION_PARENTS, start pulse and goes to SEL_PARENTS state. At this state the parent selection unit is active and is the solely master of the sorted-pool SRAM. It is in charge of choosing cnfg_P pairs of parents from the sorted pool SRAM and send them to GA_CROSSOVER. In parallel, at this state the channel between the sorter and GA_FITNESS is open again and this is the performance enhancement: the next generation chromosomes that are created by GA_CROSSOVER/MUTATION will be able to be sorted while the current generation is still in the sorted-pool and generated additional new chromosome, so that GA_ALGO unit will handle two generations in parallel.
Once GA_SELECTION_PARETNS unit is done it sends done pulse to GA_SELECTION_FSM that as a response returns to IDLE. Again, in IDLE the channel between the sorter and GA_FITNESS is still open and the FSM waits once again for the sorter to be done.

The process from GA_SELECTION_PARENTS point of view:

GA_SELECTION_PARENTS consists mainly of an FSM that randomly selects pairs of parents by calculating random addresses to read from the chromosome sorted pool, reads the parents, and sends them as a couple to the next stage: GA_CROSSOVER.

The FSM in GA_SELECTION_PARENTS starts at IDLE state. Once getting start pulse (from GA_SELECTION_FSM) it goes to CREATE_ADDR_P1 state and starts creating the first pair of parents to output to GA_CROSSOVER by looping throw the states CREATE_ADDR_P1→CREATE_ADDR_P2_READ_P1→READ_P2→SEND_PARENTS→CREATE_ADDR_P1→… etc. GA_SELECTION_PARENTS FSM is in charge of counting the pairs of parents it has created, and knows to stop after cnfg_P pairs, send that its done to GA_SELECTION_FSM and goes back to IDLE state, there it will wait for new start pulse command from GA_SELECTION_FSM.

Overall, the loop to create a single pair of parents takes 4 clock cycles as follows:

1. At CREATE_ADDR_P1 state (first clock cycle) the random address for the first address is calculated. The first parent (parent1) is chosen from the better half of the sorted pool. Hence, the range for the address it can be in is [0,cnfg_P/2-1]. The number of random bits (input from GA_42BIT_RAND_GEN) needed for this is the number of bits representing numbers in [0,P_max/2-1] range, which is clog2(P_max/2) bits. In order to ensure those bits are in the allowed range, the following pseudo-modulus is calculated:

   random_data[clog2(P_max/2)-1:0] % (cnfg_P/2)

   - The pseudo-modulus operation is performed by a generic unit explained in section 2.5.6, and it is combinatoric.
   - For the parameters chosen for the design (P_max=1024), the calculation is random_data[8:0] % (cnfg_P/2)

   The FSM outputs the result unsampled to the sorted pool memory as address along with a read request, and goes to CREATE_ADDR_P2_READ_P1 state.
2. At CREATE_ADDR_P2_READ_P1 state (clock cycle number 2) the FSM gets back the read data from the memory, chromosome, which will be sampled in the next clock cycle to parent1 register. Simultaneously, at the second clock cycle, the FSM calculates the second parent (parent2) address in a similar way as for parent1, but this time the parent can be chosen from anywhere in the pool, therefore the range of the address is [0,cnfg_P-1] and the number of bits required from GA_42BIT_RAND_GEN is clog2(P_max). So overall , the following pseudo-modulus calculation will take place:
   random_data[clog2(P_max)-1+ clog2(P_max/2):0+ clog2(P_max/2)] % cnfg_P
   - For the parameters chosen for the design (P_max=1024), the calculation is random_data[18:9] % cnfg_P

The FSM outputs the result unsampled to the sorted pool memory as address along with a read request, and goes to READ_P2 state.

3. At READ_P2 state (third clock cycle) parent1 is already sampled, and parent2 data arrives back from the memory. The FSM is prepared to sample it into parent2 register in the next clock cycle and goes to SEND_PARENTS state along with asserting parents_valid signal, since both parent1 and parent2 are ready.
   In addition, at this point the parents counter increases by 1, since another pair of parents has been created.
4. At SEND_PARENTS state, the 4^{th} clock cycle, both parent1 and parent2 are ready and outputted to GA_CROSSOVER along with parents_valid signal (all sampled). The FSM remains at this state (SEND_PARENTS) until it gets back parents_ack from GA_CROSSOVER.
5. Once parents_ack arrives from GA_CROSSOVER the FSM goes to CREATE_ADDR_P1 state to create another pair of parents, unless this was the $cnfg\_P^{th}$ pair and in this case it sends done pulse to GA_SELECTION_FSM and goes back to IDLE state.

Time analysis:

The time for the sorter operations can not be determined exactly because it depends on the order in which the chromosomes are arriving to the sorter, which is random, and in {V,d} samples which are unknows. However the time for the sorter can be estimated in average: the insertion phase of the sorter takes in average log(cnfg_P) clock cycles per chromosome, and cnfg_P*log(cnfg_P) clock cycles for the entire population (according to the sorter performance analysis). The extraction phase takes in average 2*cnfg_P clock cycles. So, overall the sorter requires cnfg_P*(2+log(cnfg_P)) clock cycles for the entire generation: from the first chromosome who enters the sorter until the last chromosome who is written to the chromosomes sorted pool.

This point is a bottle neck in the total GA_ALGO pipe, since the parent selection stage can not start until all the chromosomes are in the sorted pool. However, in average, the $(i+1)^{th}$ generation chromosomes will start to enter the sorter quite fast while pairs of parents from the i^{th} generation are still selected, such that when all the i^{th} generation pairs of parents are done, there is a small amount of chromosomes form the $(i+1)^{th}$ generation that are not in the sorter yet. This ability is meaningful and leads to a smaller bottleneck – only the extraction phase (that can take place only after all the chromosomes have been inserted to the sorter).

Afterwards, the last stage in GA_SELECTION is parent selection, which as mentioned above, take 4 clock cycles per pair of parents, and thus 4*cnfg_P clock cycles in order to create all the parents of a single generation.

2.5.3.4 GA_CROSSOVER

GA_CROSSOVER block is in charge of performing crossover operation between two chromosomes and output a single modified chromosome which is a mix of them. GA_CROSSOVER is mostly combinatoric and it gets a pair of parents, and at one clock latency outputs their child.

Interface:

signal name	input/output	Bits	pls/lvl	Description
clk	Input	1		
rstn	Input	1	Lvl	active low
Registers/Configurations IF				
sw_rst	Input	1	Pls	Falling edge of enable register (ga_enable)
cnfg_M	Input	M_MAX_W= clog2(M_max+1)= clog2(32+1)=6	Lvl	M is the number of weights in the filter (number of elements in V_vec/W_vec). **Range: [2,32]** (M_max=32)
cnfg_max_fit_score	Input	FIT_SCORE_W	Lvl	Maximal fitness score possible for current registers configuration (worst fitness score). Unsigned fixed point.
Data IF: TOP , ALGO_TOP				
rand_data	Input	RAND_ W	Pls	Random bits from GA_42BIT_RAND_GEN.
gen_best_score	Input	FIT_SCORE_W	Lvl	From GA_SELECTION. Represents the current generation best fitness score (minimal score). Unsigned fixed point.
Data IF: GA_SELECTION				
parents_valid	Input	1	Lvl	Active high. From GA_SELECTION to GA_CROSSOVER. Rises once a pair of two chromosomes are chosen to be parents and are ready to go to next stage (crossover). Remains high and steady along with the data it validates until receiving parents_ack from GA_CROSSOVER, and falls when receiving it.
parent1	Input	DATA_W*M_max	Lvl	Valid only when parents_valid is asserted. This is the parent that have been selected from the better half of the chromosomes sorted pool. M_max elements, each is signed fixed point with DATA_W.
parent2	Input	DATA_W*M_max	Lvl	Valid only when parents_valid is asserted. This is the parent that have been selected from the entire chromosomes sorted pool. M_max elements, each is signed fixed point with DATA_W.
parents_ack	Output	1	Pls	Active high. Acknowledgment from GA_CROSSOVER to GA_SELECTION that it has received parent1 and parent2 from GA_SELECTION. Value is relevant only when parents_valid is high.

Data IF: GA_MUTATION				
child_ack	Input	1	Pls	Active high. Acknowledgment signal from GA_MUTATION to GA_CROSSOVER that it has received the child from GA_CROSSOVER. Value is relevant only when child_valid is high.
child_valid	Output	1	Lvl	Active high. From GA_CROSSOVER to GA_MUTATION. Rises once a child for the next generation is form and ready to go to next stage (mutation). Remains high and steady along with the data it validates until receiving child_ack from GA_MUTATION, and falls when receiving the ack.
child	Output	DATA_W*M_max	Lvl	Valid only when child_valid is asserted. This is the child that was created from the two parents. M_max elements, each is signed fixed point with DATA_W.

***Table 8**: GA_CROSSOVER interface*

- For default values RAND_W=10, and it's a function of M_max.
  Overall, the formula is (will be clarified in the detailed description segment):
  RAND_W = 2*clog2(M_max) = 2*clog2(32) = 10

Diagram:

In figure 30 a top view of GA_CROSSOVER is presented.

In figures 31-33 each one of the parts is zoomed-in: top crossover, crossover find thresholds, crossover algo.

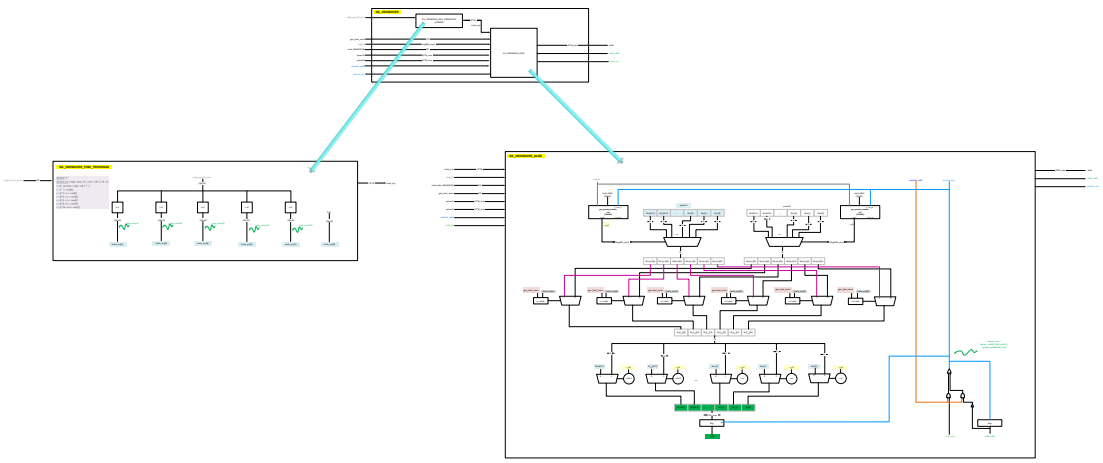


***Figure 30** : GA accelerator – GA_CROSSOVER microarchitecture – top view*

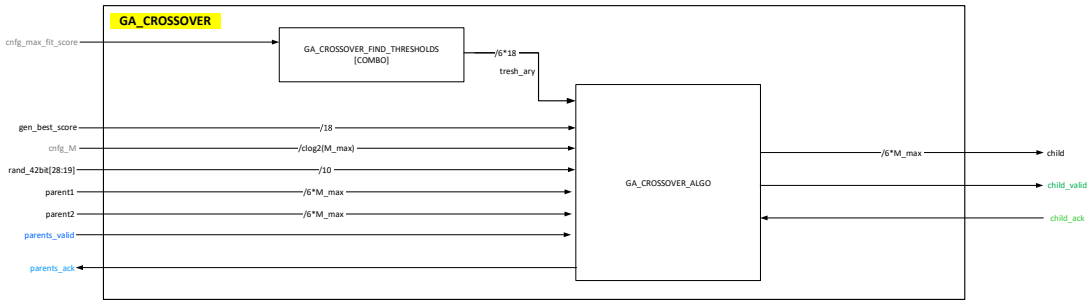


Figure 31 *: GA accelerator – GA_CROSSOVER microarchitecture – top*

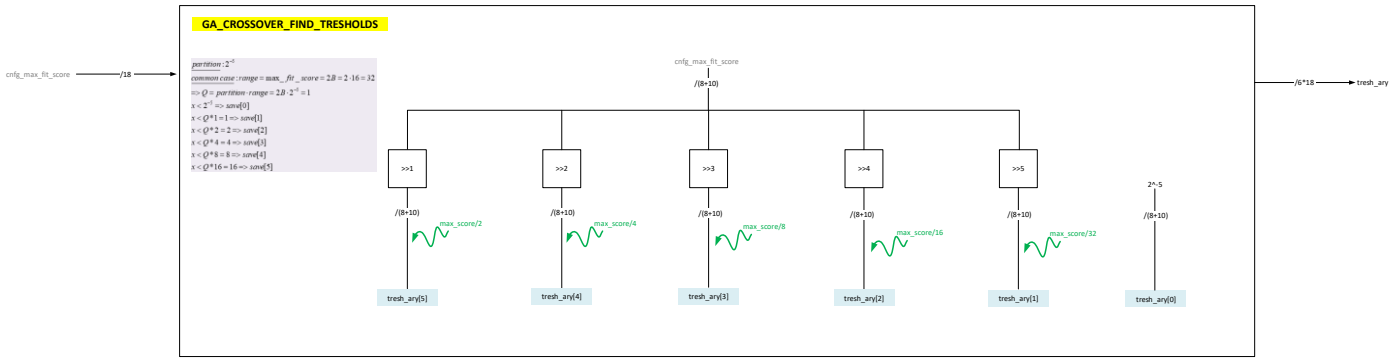


Figure 32 *: GA accelerator – GA_CROSSOVER microarchitecture – find thresholds (GA_CROSSOVER_FIND_THRESHOLDS)*

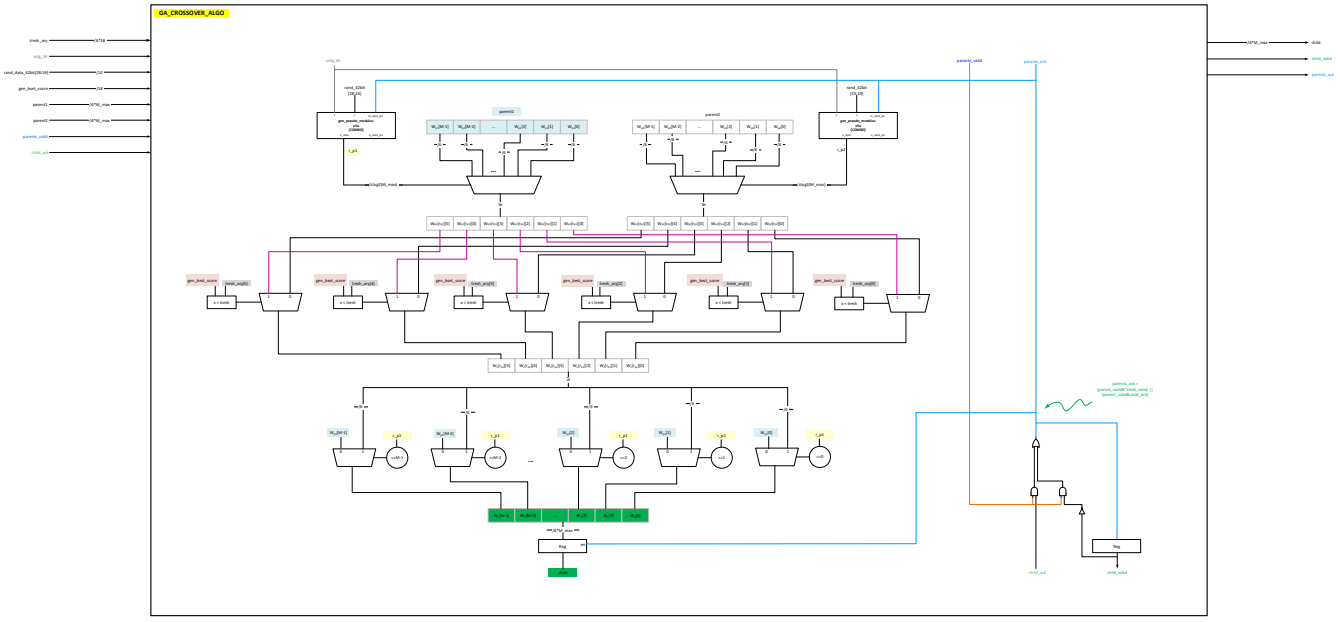


Figure 33 *: GA accelerator – GA_CROSSOVER microarchitecture – algo (GA_CROSSOER_ALGO)*

Detailed description:

Once GA_CROSSOVER gets a valid pair of parents, a few combinatorial steps are taking place in order to create their child. The child is a duplication of parent1 except a single weight that is chosen in random. A few LSBs of this weight are replaced with bits from another random weight of parent2, and the crossover position (number of LSBs) is chosen according to the best score in the current generation.

The stages are as follow: (this are the stages of GA_CROSSOVER_ALGO is shown in figure 33 and it is very similar to the block described in the article (figure 12))

1. At first step, random weights are chosen from parent1 and parent2. Since there are at most M_max weights, it requires clog2(M_max) random bits to represent the index of the weight for each parent, so overall 2*clog2(M_max) random bits are required, which is for current parameters 2*clog2(32)=10 bits.
   Since the actual number of bits is set by register cnfg_M, and as a result the range of the selected index is [0,cnfg_M-1], pseudo-modulus operation is needed in order to ensure the random index is in the allowed range.
   The following pseudo-modulus is calculated:
   random_data[clog2(M_max)-1:0] % cnfg_M
   - The pseudo-modulus operation is performed by a generic unit explained in section 2.5.6, and it is combinatoric.
   - For the parameters chosen for the design (M_max=1024), the calculation is random_data[4:0] % cnfg_M
2. After the random weights were chosen, a crossover operation is performed between them. A single point crossover is preformed, and the crossover position is set deterministically according to the best fitness score of the generation. In general, the better the fitness score is the less bits from parent1 weight will be replaced and the one replaced will be less significant, i.e. the crossover position will be smaller. (Crossover position calculation is detailed below.)
3. After the crossover have been made, the new weight is inserted to the index of parent1 in the child, and besides this the child is a copy of parent1 (which is the parent that was chosen from the better half of the sorted-pool in GA_SELECTION, and hence have a greater probability to be with a higher fitness score).
4. Finally, the child is sampled and outputted with a valid signal to GA_MUTATION. The valid and the child will stay high and steady until ack will be received from GA_MUTATION, as it can be showed in figure 33.
5. After the ack from GA_MUTATION will be received a new chromosome will be able to enter GA_CROSSOVER (i.e.: GA_CROSSOVER won't return ack to GA_SELECTION if its not free to handle it).

Crossover position is calculated as follow:

Since DATA_W=6, 6 comparators are placed, one for each bit of the chosen weight from parent1. For each bit the result of the comparator acts as a selector to a mux that choses if the corresponding bit of the new weight will be from parent1 current weight or from parent2 other weight.

The comparators compare between current generation best fitness score and a configured set of thresholds: thresh_ary. This thresholds array has as mentioned 6 elements, since DATA_W=6, and are in increasing order, such that thresh_ary[0] is the smallest number. The values are a function of the maximal possible fitness score for the current registers configuration, in order to use all of the dynamic range. Overall:

- thresh_ary[5] = cnfg_max_fit_score / 2
- thresh_ary[4] = cnfg_max_fit_score / 2^2
- thresh_ary[3] = cnfg_max_fit_score / 2^3
- thresh_ary[2] = cnfg_max_fit_score / 2^4
- thresh_ary[1] = cnfg_max_fit_score / 2^5
- thresh_ary[0] = 2^{-5} – which is the guaranteed to be smaller than thresh_ary[1] for all values of cnfg_max_fit_score.
    - Explanation: cnfg_max_fit_score comes from GA_FITNESS and its value (as explained in section 2.5.3.2) is 2*cnfg_B. Since cnfg_B minimal value is 2, the minimal value for cnfg_max_fit_score is 4. In this case thresh_ary[1]=$4/2^5=2^{-3}$, which is two orders of magnitude larger them thresh_ary[0].
    - The meaning of that if there's a generation in which the best fitness score is smaller then thresh_ary[0], there will be no replacements and for all chromosomes child=parent1.
    - Since bits represent powers of 2 the following partition was chosen and not a linear/uniform one, because the affect of replacing bits is not linear. In addition, this partition is very efficient area wise and time wise.

Time analysis:

GA_CROSSOVER is combinatorial with sampled outputs, hence the block takes 1 clock cycle per chromosome.

2.5.3.5 GA_MUTATION

GA_MUTATION block is in charge of performing the last modification to the new generation chromosomes, and by this finish its creation.
GA_MUTATION is mostly combinatoric and it gets a single child from GA_CROSSOVER, decided if its need mutation or not, and at one clock latency outputs its final version to the chromosomes queue.

Interface:

signal name	input/output	Bits	pls/lvl	Description
clk	Input	1		
rstn	Input	1	Lvl	active low
Registers/Configurations IF				
sw_rst	Input	1	Pls	Falling edge of enable register (ga_enable)
cnfg_M	Input	M_MAX_W= clog2(M_max+1)= clog2(32+1)=6	Lvl	M is the number of weights in the filter (number of elements in V_vec/W_vec). **Range: [2,32]** (M_max=32)
cnfg_max_fit_score	Input	FIT_SCORE_W	Lvl	Maximal fitness score possible for current registers configuration (worst fitness score). Unsigned fixed point.
Data IF: TOP , ALGO_TOP				
rand_data	Input	RAND_ W	Pls	Random bits from GA_42BIT_RAND_GEN.
gen_best_score	Input	FIT_SCORE_W	Lvl	From GA_SELECTION. Represents the current generation best fitness score (minimal score). Unsigned fixed point.
Data IF: GA_MUTATION				
child_valid	Output	1	Lvl	Active high. From GA_CROSSOVER to GA_MUTATION. Rises once a child for the next generation is formed and ready to go to next stage (mutation). Remains high and steady along with the data it validates until receiving child_ack from GA_MUTATION, and falls when receiving the ack.
child	Output	DATA_W*M_max	Lvl	Valid only when child_valid is asserted. This is the child that was created from the two parents. M_max elements, each is signed fixed point with DATA_W.
child_ack	Input	1	Pls	Active high. Acknowledgment signal from GA_MUTATION to GA_CROSSOVER that it has received the child from GA_CROSSOVER. Value is relevant only when child_valid is high.

Data IF: CHROMOSOMES QUEUE				
queue_push	Output	1	Pls	Push command to the chromosomes queue. Active high.
queue_chromosome	Output	DATA_W*M_max	Pls	Value is valid only when queue_push is high. The final version of the chromosome for the new generation.

***Table 9**: GA_MUTATION interface*

- For default values RAND_W=11, and it's a function of M_max and DATA_W. Overall, the formula is (will be clarified in the detailed description segment):

  RAND_W = clog2(M_max)+DATA_W = clog2(32)+6 = 11

Diagram:

In figure 34 a top view of GA_MUTATION is presented.

In figures 35-38 each one of the parts is zoomed-in: top mutation, mutation find ratio, mutation FSM, mutation algo.

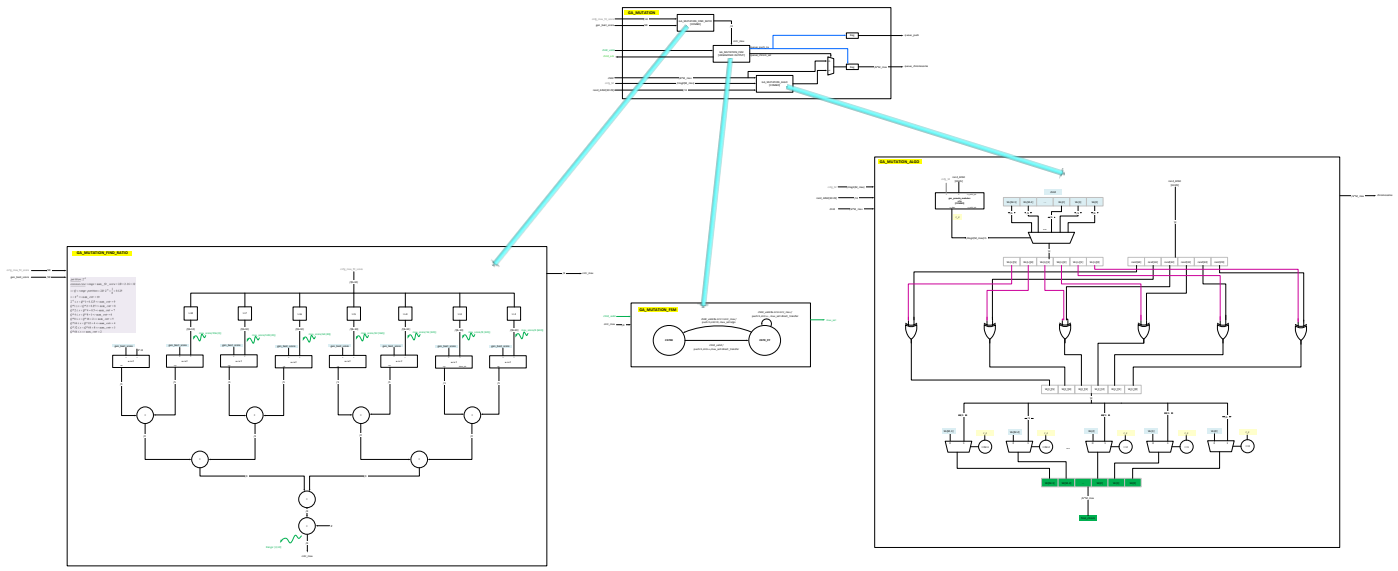


***Figure 34** : GA accelerator – GA_MUTATION microarchitecture – top view*

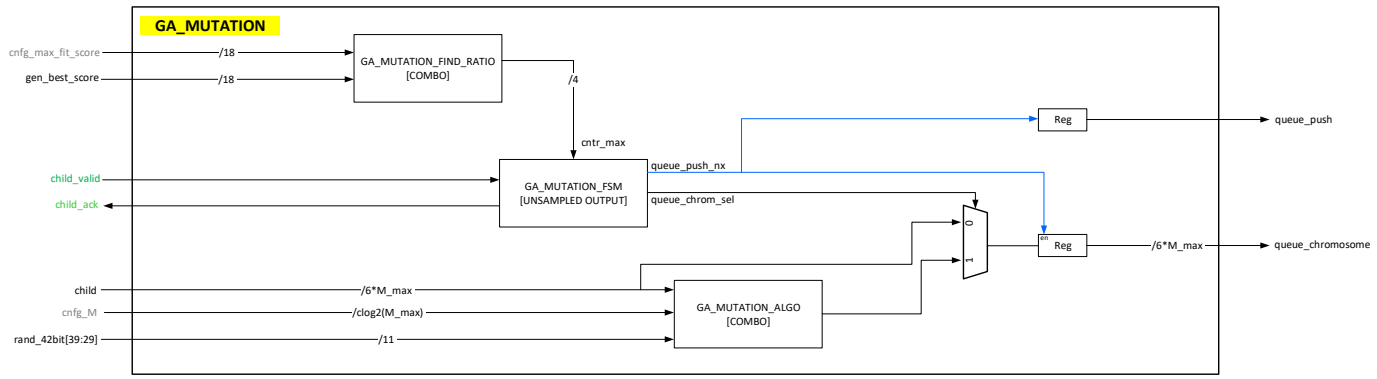


Figure 35 *: GA accelerator – GA_MUTATION microarchitecture – top*

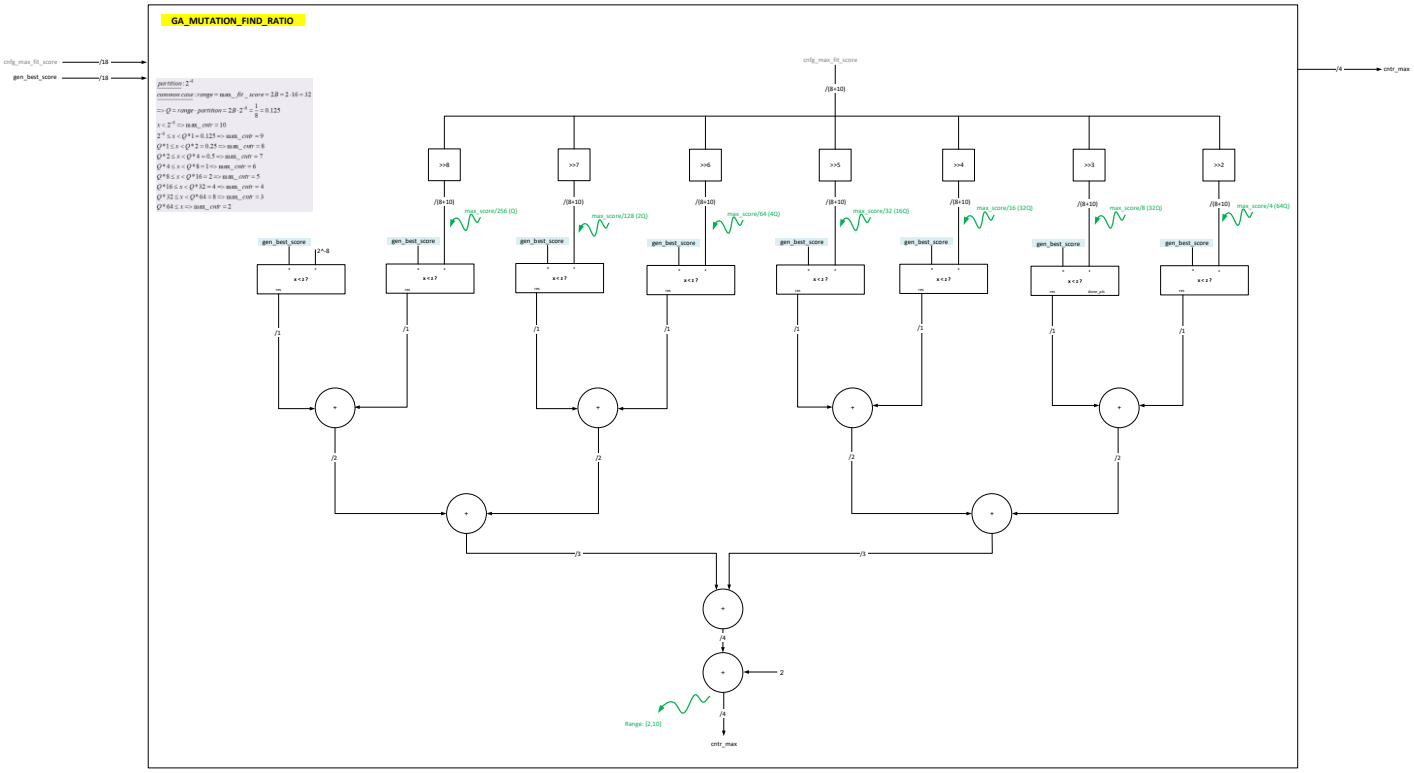


Figure 36 *: GA accelerator – GA_MUTATION microarchitecture – find ratio (GA_MUTATION_FIND_RATIO)*

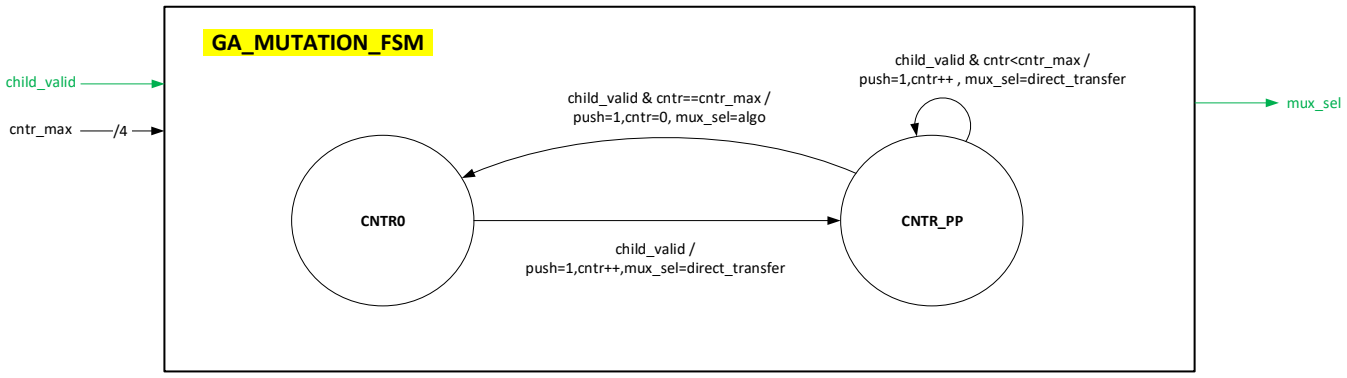


Figure 37 *: GA accelerator – GA_MUTATION microarchitecture – fsm (GA_MUTATION_FSM)*

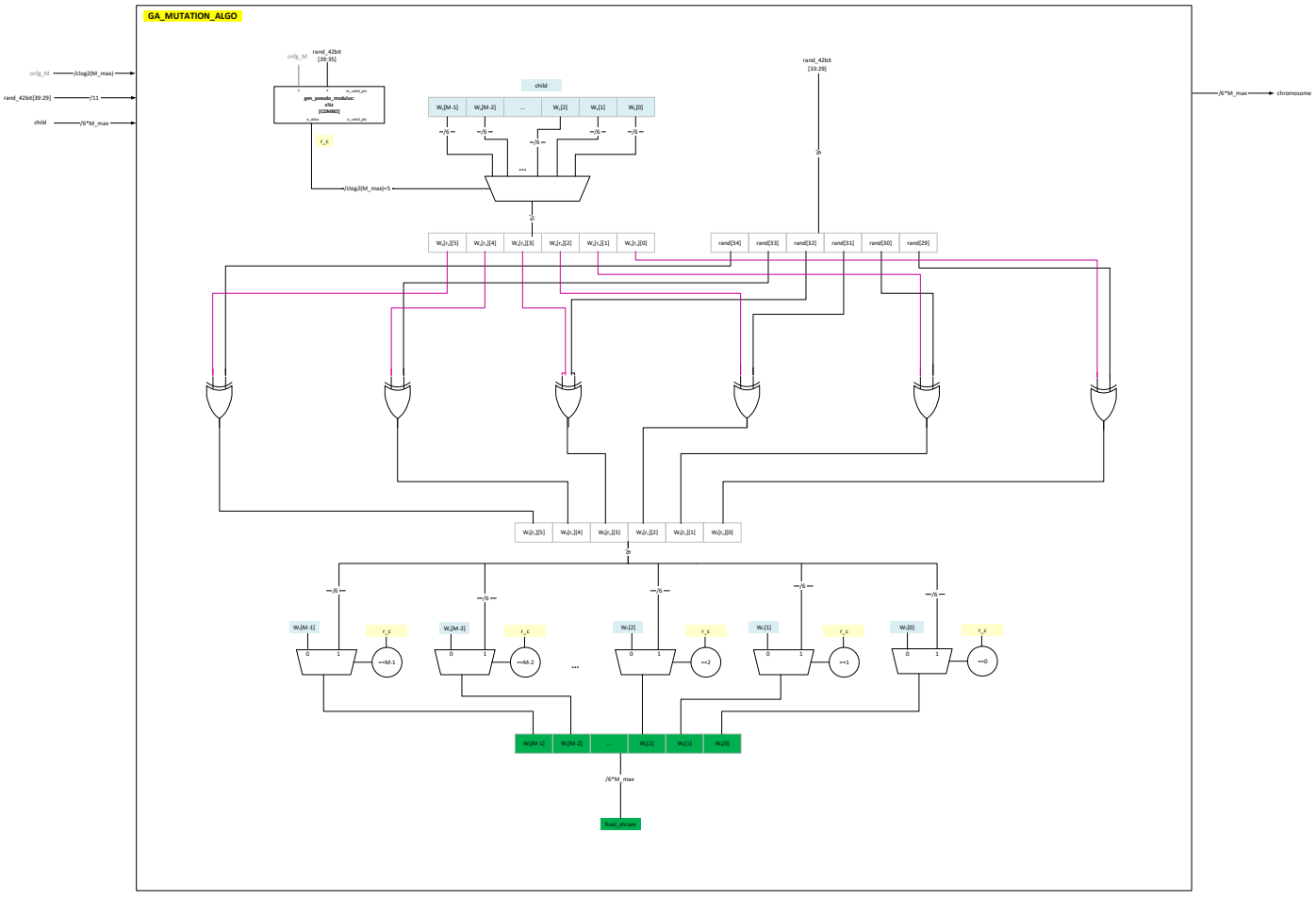


Figure 38 *: GA accelerator – GA_MUTATION microarchitecture – algo (GA_MUTATION_ALGO)*

Detailed description:

GA_MUTATION is a simple block that handles one chromosome at a time. GA_MUTATION gets a child+valid from GA_CROSSOVER, and when its free to handle it returns ack to GA_CROSSOVER and performs its action on the child and one clock later outputs the final child, i.e. the final chromosome, to the chromosomes queue. It is assumed by GA_MUTATION that when it sends its push commands to the chromosomes queue they are received immediately.

- This is indeed guaranteed by GA_MAIN_FSM by setting the chromosomes queue mux to choose data from GA_MUTATION, and by the fact that the queue allows push and pop to be done at the same clock cycle, so GA_FITNESS requests pop won't interrupt GA_MUTATION request to push.

Once GA_MUTATION gets a valid child from GA_CROSSOVER, the first step is to decide whether or not this child needs to be mutated: only a part of the received chromosomes will go out after mutation, and most of them will go out to the chromosomes queue unchanged. Mutation ratio is between 10% to 50%, according to the current generation best fitness score: the better the fitness score there will be less mutations (lower mutation ratio).

Mutation rate is handled by a simple counter (GA_MUTATION_FSM) and the maximal value for this counter ranges between 2 to 10, when 2 means that every second chromosome will be mutated (50% mutation rate), and 10 means that every 10^{th} chromosome will be mutated (10% mutation rate).

As mentioned, the maximal value of the counter is set according to the current generation best fitness score, and more specifically in its place inside the dynamic range of possible fitness score according to configuration: the lowest the best score of the current generation, the higher the counter will get. This is calculated in GA_MUTATION_FIND_RATIO: this unit gets gen_best_score and cnfg_max_fit_score, and outputs (combinatorically) the maximal value of the counter which will be used by GA_MUTATION_FSM that handles the counting itself.

In figure 36 the structure of GA_MUTATION_FIND_RATIO is presented, and it is consist of a set of shifters, comparators and substructures. Same as in GA_CROSSOVER the range is not split uniformly but according to powers of two, and this is done here in order to create greater penalty for worst fitness score (and in addition the usage of shifters instead of complex divisors saves area and time).

Once the maximal value of the counter has been set (range is [2,10]), it is used by GA_MUTATION_FSM that holds the counter itself. The counter starts at 0 (IDLE

state) and the first time a valid child arrives it increases by one and goes to CTNR_PP state (the first child is transferred to the chromosomes queue unchanged). The next time a valid child will arrive the counter will increase by one and if it has reached its maximal value the child will be mutated and only after this will be pushed to the chromosomes queue, and the FSM will go to CNTR0 state and start to count from 0 again. Otherwise, meaning if at CNRT_PP state the counter hasn't reached its maximal value yet, the FSM will stay in this state and increase the counter by one every time a new valid chromosome will arrive, until it will reach its maximal value. In this case the child will go over mutation and the above process will happen again and again.

Overall the decision to go throw mutation or not is represented by a mux selector to the chromosomes queue data that GA_MUTATION_FSM governs, and it chooses whether to transfer the original child or the output of GA_MUTATION_ALGO unit, that inside it the mutation itself occurs combinatorically.

GA_MUTATION_ALGO is shown in figure 38 and it is very similar to the block described in the article (figure 13). The stages of mutation are as follow:

1. A random weight of the child is selected. From same considerations as in the random weight selection in GA_CROSSOVER, clog2(M_max) random bits are required for this, and pseudo-modulus operation (see section 2.5.6) should be made with cnfg_M value, i.e. the index of the weight selected will be the result of:

   random_data[clog2(M_max)-1:0] % cnfg_M

   - For current parameters that were set its:

     random_data[4:0] % cnfg_M

2. A bitwise XOR operation is done between the chosen weight and random word with worldlength of DATA_W bits, in order to create a new weight. This is the mutation.
   - For current parameters that were set DATA_W=6, hence the random word is 6bits long: random_data[10:5].

3. The new weight is inserted to the same place (index) in the final chromosomes, and all of the other weights remain untouched. The chromosome created is the final chromosome after mutation.

Time analysis:

GA_CROSSOVER is combinatorial with sampled outputs, hence the block takes 1 clock cycle per chromosome.

2.5.4. GEN_INNER_PRODUCT

The gen_inner_product block is a synchronic unit with parametric design, that calculates the inner product between two vectors:

$$result = \sum_{i=0}^{N-1} vec1[i] \cdot vec2[i]$$

When N is the number of elements in each vector

The vectors elements are signed fixed point and the result is signed fixed point as well.

The block contains multipliers and adders and is a pipelined unit, so back-to-back requests can be received.

In addition, the unit supports approximation of the result if the wanted number of bits by the user (parameter) is different than the needed number of bits by arithmetic rules.

Parameters:

As mentioned, the block is a generic designed component. Its parameters are:

- **VEC_ELEMS_NUM** – Number of elements in each vector.
  - A.k.a N ($N \equiv VEC\_ELEMS\_NUM$).
- **ONE_ELEM_INT_W** – Each element is represented in signed fixed point representation. ONE_ELEM_INT_W is the number of bits for sign+int part.
- **ONE_ELEM_FRACT_W** – Each element is represented in signed fixed point representation. ONE_ELEM_FRACT_W is the number of bits for fraction part.
  - So overall the number of bits in one element is:
    ONE_ELEM_W = ONE_ELEM_INT_W + ONE_ELEM_FRACT_W
    - A.k.a M ($M \equiv ONE\_ELEM\_W$)
- **RES_INT_W** – The result is given in signed fixed point representation. RES_INT_W is the number of bits for sign+int part.
- **RES_FRACT_W** – The result is given in signed fixed point representation. RES_FRACT_W is the number of bits for fraction part.
  - So overall the number of bits in the result is:
    RES_W = RES_INT_W + RES_FRACT_W

Interface:

signal name	input/output	Bits	pls/lvl	description
clk	Input	1		
rstn	Input	1	Lvl	active low
sw_rst	Input	1	Pls	active high
DATA IF				
i_valid_pls	Input	1	Pls	Active high. Start pulse per calculation, that indicates that the values in i_vec1 and i_vec2 are valid and is a request for an inner product calculation between them.
i_vec1	Input	VEC_ELEMS_NUM* ONE_ELEM_W	Pls	Value relevant only when i_valid_pls is high. One of the vectors to calculate the inner product with. Contains VEC_ELEMS_NUM elements, each in width of ONE_ELEM_W, and the vector is flatten, i.e. the elements are concatenated in the following form: {[N-1],[N-2],…,[1],[0]}.
i_vec1	Input	VEC_ELEMS_NUM* ONE_ELEM_W	Pls	Value relevant only when i_valid_pls is high. One of the vectors to calculate the inner product with. Contains VEC_ELEMS_NUM elements, each in width of ONE_ELEM_W, and the vector is flatten, i.e. the elements are concatenated in the following form: {[N-1],[N-2],…,[1],[0]}.
o_valid_pls	Output	1	Pls	Active high. Done pulse per calculation, that indicates that the values in o_res are valid. The o_valid_pls are in-order to i_valid_pls, and the first o_valid_pls will address to the first i_valid_pls, and so on (pipe).
o_res	Output	RES_W	Pls	Value relevant only when o_valid_pls is high. The inner product result.

***Table 10**: GEN_INNER_PRODUCT interface*

Diagram:

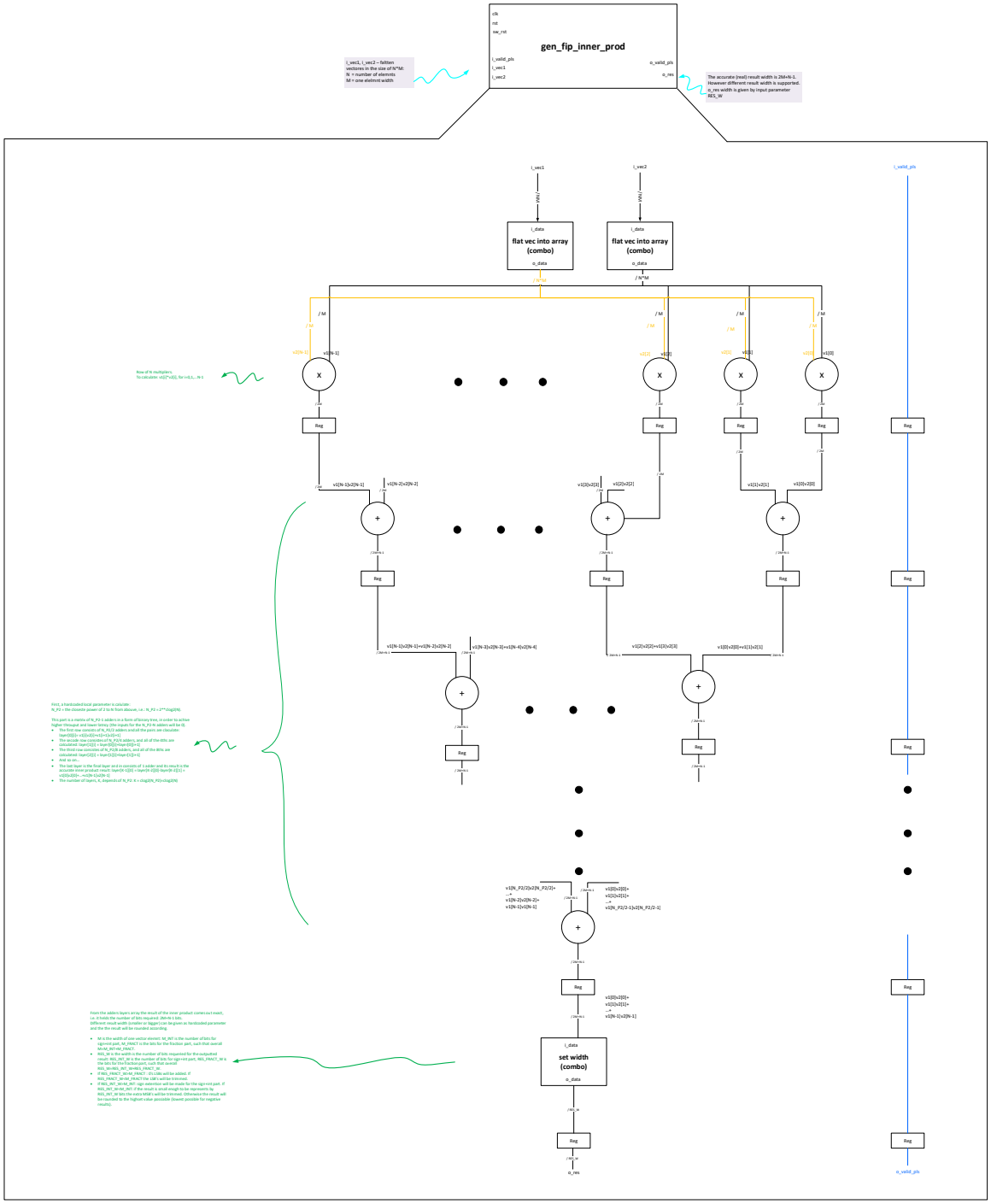


Figure 39 *: generic inner product block microarchitecture*

Detailed description:

As shown in figure 39 above, the pipe consists of 1 layer of multipliers, clog2(N) layers of adders, and 1 layer for result width adjustments, so over all the number of stages in the pipe is 2+clog2(N).

- Full explanations are written inside figure 39: the function of each layer, the calculation of the number of adders layers, the width of each bus, etc.
- Note that an optimization could be made here and the width adjustment layer could be spared in cases RES_W larger or equal to the accurate result width, however for simplicity it is present either way.

Performance analysis:

Since the pipe consists of 2+clog2(N) layers, each takes 1 clock cycle, the latency of gen_inner_product is $2+clog2(VEC\_ELEMS\_W)$ clock cycles, and the throughput $\frac{1}{2+clog2(VEC\_ELEMS\_W)}$ results per clock cycle.

- So for example for N=VEC_ELEMS_NUM=32, the latency is 2+clog2(32)=7 clock cycles latency.
    - Hence, from the rise of i_valid_pls until the rise of o_valid_pls there are 8 clock cycles.

2.5.5. GEN_BST_SORTER

The sorter is a generic component that gets data in a serial manner and afterwards outputs it in a serial manner as well. The data are elements consists of key and value:

- The key is the field according to it the elements are sorted, and it must be non-negative. Can be integer or fixed point.
- The value is a nonfunctional field, data one wants to save with the element.

The sorter has internal memory (SRMA) in order to reduce area, which results in one operation (read or write) per clock cycle. The algorithmic data structure used is binary search tree in order to make the sorter fast under this restriction. This is the source of the name of the sorter: BST stands for binary search tree.

Short behavioral description:

GEN_BST_SORTER is for serialized data (gets one element at a time). The sorter has internal memory which contains all the data, in binary-search-tree data structure.

- Each element has key and value, and the order is set by the key.
- Key can be integer or fixed-point number, and can not be negative.
- The number of elements to sort, N, should be known in advanced by configuration, and should be set before the rise of enable.
- The sorter start to work at the rise of enable and has 2 phases:
    - first phase is insert phase: Until reaching N elements, the sorter waits for new elements to arrive. Each element that arrives is inserted by its key to its appropriate location in the tree.
    - second phase is extract phase: After "sort_is_done_pls" rises the sorter waits for the rise of "get_all_sorted_data_req_pls" which triggers full extractions of the elements: the elements are outputted by in-order traversal, from min key to max key. After the extraction is done (N elements has been outputted) the signal "get_all_sorted_data_is_done_lvl" rises.
- After the 2nd phase ends, the sorter is done, and for new data to be inserted enable must rise again. (So between every 2 runs enable must re-asserted).

Parameters:

As mentioned, the sorter is a generic designed component. Its parameters are:

- **KEY_W** – Number of bits for key.
- **VALUE_W** – Number of bits for value.
- **MAX_ELEM_NUM** – The maximum number of elements the hardware supports.

The SRAM size is described in figure 42:

- Number of rows: MAX_ELEM_NUM (one row per element)
- Width of row: a row consists of the element's key, value and some additional fields required for the operation: valid bit, left and right sons address and valid bits, same for parent, and twins list information. Overall ,the width of a row, i.e. the number of bits the data structure requires per element is given by:

$$1 + KEY\_W + VALUE\_W + 4 \cdot (1 + c\log 2(MAX\_ELEM\_NUM)) + 1$$

Interface:

signal name	input/output	Bits	pls/lvl	description
clk	Input	1		
rstn	Input	1	Lvl	active low
sw_rst	Input	1	Pls	active high
CONFIGURATION AND STATUS				
i_cnfg_elems_num	Input	clog2(MAX_ELEM_NUM+1)	Pls	The number of elements to sort. Must be set before the rise of i_enable, and stay stable until i_enable falls/reset. **Range: [1,MAX_ELEM_NUM]**
i_enable	Input	1	Lvl	Active high. Enable signal to the sorter. Only when i_enable is asserted (high) the sorter is activated. When i_enable falls the sorter goes back to IDLE. **Must be deasserted and asserted again before every new sort missions.**
o_sorter_phase	Output	1	lvl	The sorter has two phases in its life cycle: insert phase, in which it gets elements and sorts them, and after all elements has been sorted it goes to the extract phase and by order outputs the elements that were sorted. o_sorter_phase indicates in which phase the sorter is in: 1'b0 – insert phase 1'b1 – extract phase
Data IF: INSERT				
new_elem_valid	Input	1		Active high. Indicates that there is a new element to sort. Must stay stable until the new_elem_ack is asserted (high).
new_elem_key	Input	KEY_W		Value relevant only when new_elem_valid is asserted (high). Must be non-negative and integer or fixed-point (same format to all elements). Must stay stable until the new_elem_ack is asserted (high).

new_elem_data	Input	VALUE_W		Value relevant only when new_elem_valid is asserted (high). Data that is saved along with the elements and will be outputted later at the extract phase. However the sorter doesn't use it (except saving it). Must stay stable until the new_elem_ack is asserted (high).
new_elem_ack	Output	1	Pls	Active high and relevant only when new_elem_valid is high. Indicates that the new element has been received by the sorter and that the new_elem_* interface can change.
sort_is_done_pls	Output	1	Pls	Active high. Indicates that i_cnfg_elems_num has been received and sorted, the sort phase is done, and no new elements can be sorted.
min_elem_key	Output	KEY_W	Lvl	The key of the minimal element that has been sorted so far (from the rise of i_enable).
min_elem_value	Output	VALUE_W	Lvl	The value of the minimal element that has been sorted so far (from the rise of i_enable).
max_elem_key	Output	KEY_W	Lvl	The key of the maximal element that has been sorted so far (from the rise of i_enable).
max_elem_key	Output	VALU_W	Lvl	The value of the maximal element that has been sorted so far (from the rise of i_enable).
Data IF: EXTRACT				
get_all_sorted_data_req_pls	Input	1	Pls	Active high. Can be asserted only when sort_is_done_lvl is high. Rise of this signal will trigger an extraction of all elements in the sorter: all of the i_cnfg_elems_num will be outputted in order from min to max one by one. (No ack back is required so the slave asserting this signal must be ready to get all the elements.)
get_all_sorted_data_done_lvl	Output	1	Lvl	Active high. Indicates that i_cnfg_elems_num has been outputted, the extract phase is done.
get_elem_idx	Output	clog2(MAX_ELEM_NUM+1)	Pls	Value relevant only when get_elem_valid is asserted (high). The outputted elements are being counted from 0 (min elem) to i_cnfg_elems_num-1 (max elem), and this signal is their index (count).

get_elem_valid	Output	1	Pls	Active high. Indicates on another valid element that is outputted. Overall during the extract phase there will be i_cnfg_elems_num clock cycles in which this signal is asserted (high).
get_elem_key	Output	KEY_W	Pls	Value relevant only when get_elem_valid is asserted (high). The key of the current element that is outputted.
get_elem_value	Output	VALUE_W	Pls	Value relevant only when get_elem_valid is asserted (high). The value of the current element that is outputted.

***Table 11**: GEN_BST_SORTER interface*

Diagram:

The sorter consists of three main components: memory in which it stores all the elements, FSM for the insert phase and FSM for the extract phase.

In figure 40 a top view of GA_BST_SORTER is presented.

In figures 41-44 each one of the parts is zoomed-in: top sorter, memory, sort FSM and extract FSM.

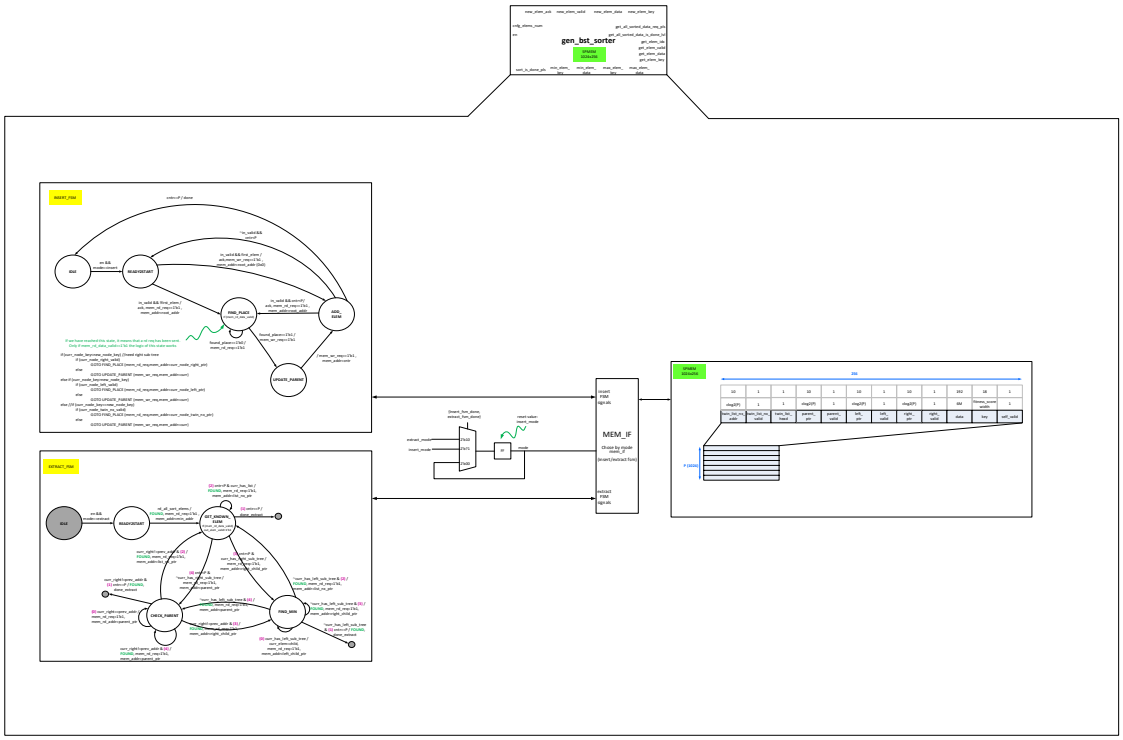


***Figure 40** : GEN_BST_SORTER microarchitecture – top view*

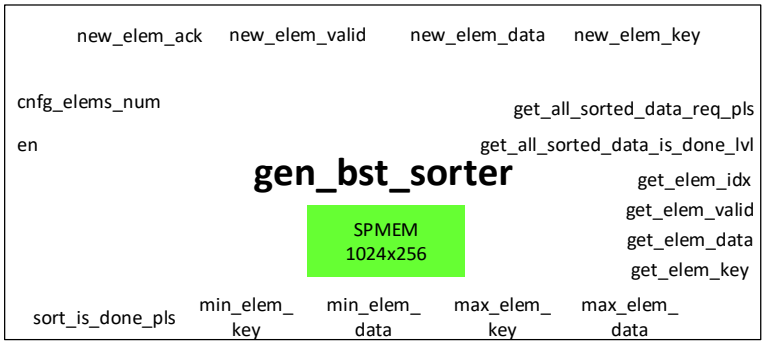


***Figure 41** : GEN_BST_SORTER microarchitecture – interface*

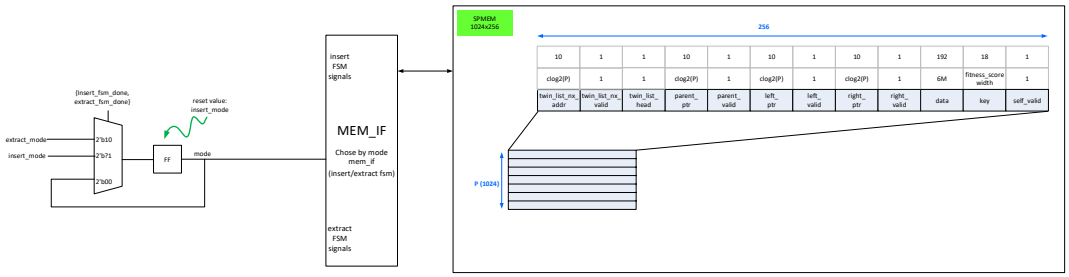


***Figure 42** : GEN_BST_SORTER microarchitecture – memory*

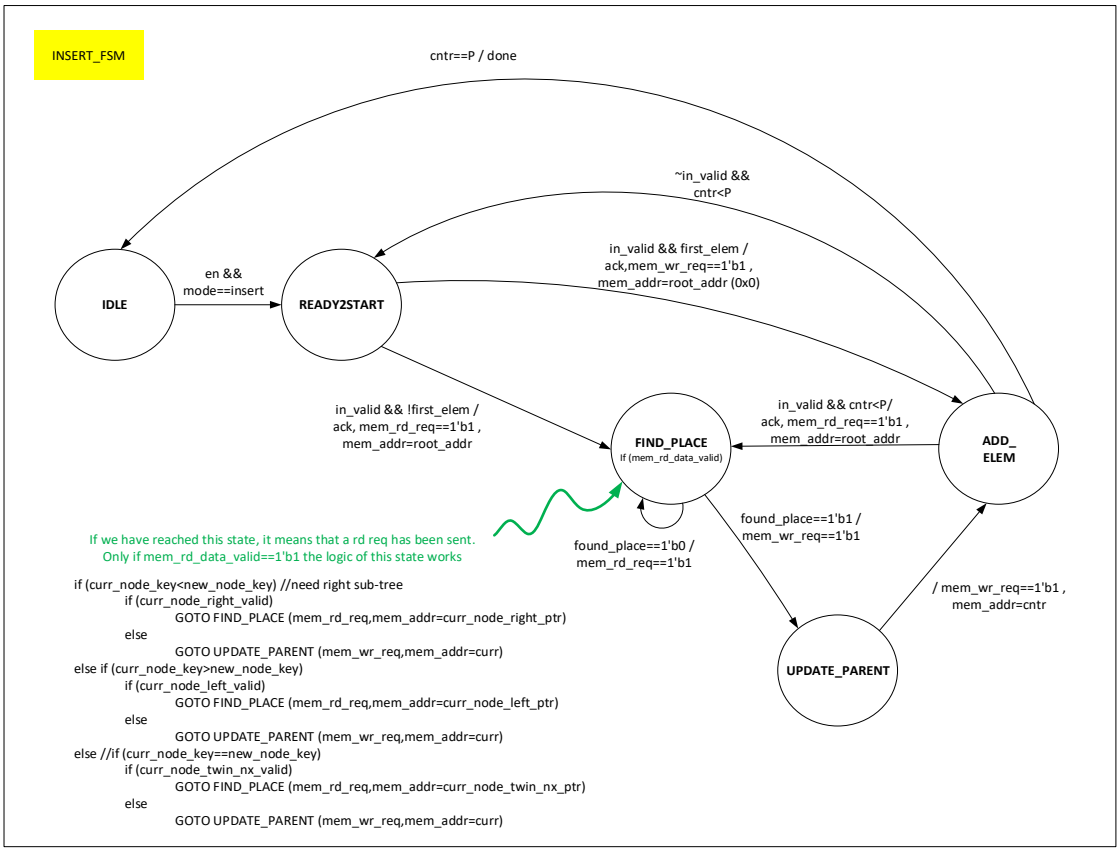


***Figure 43** : GEN_BST_SORTER microarchitecture – insert FSM*

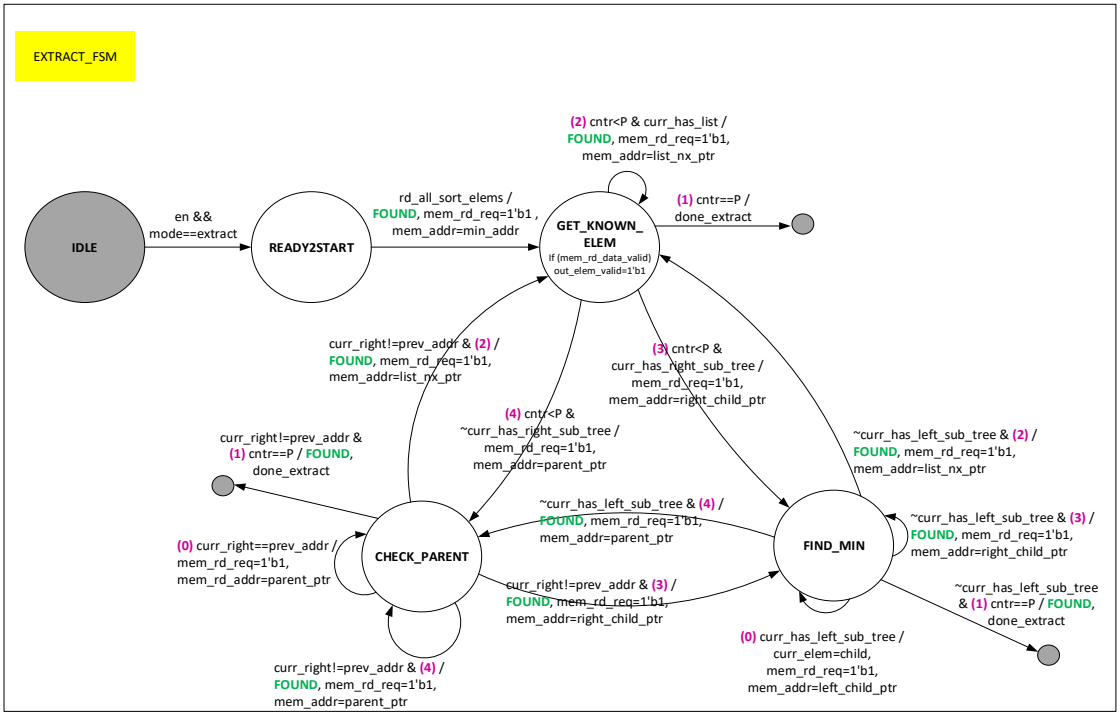


***Figure 44** : GEN_BST_SORTER microarchitecture – extract FSM*

Software pseudo code for BST (insertion and in-order traversal):

BST (binary search tree) is a known data structure that allows sorting of elements and in-order traversal (extract elements in order).

BST is defined as a binary tree in which for every node the left subtree contains elements that are smaller, and the right subtree contains elements that are larger:

$$\max\_key(left\_sub\_tree(node)) < key(node) < \min\_key(right\_sub\_tree(node))$$

This property forces the keys in tree to be unique. As a result in order to support several elements with the same key a linked-list is present in each node and only the first unique key that will arrive will be part of the BST, and the others with the same key will be inserted to a linked-list, in which the first one to arrive is the head of the list.

Software algorithm pseudo-code for the tree:

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// ~~~~~~~~~~~~~~~~ THE DATA STRUCTRE IS BINARY-SEARCH-TREE, IN WHICH ~~~~~~~~~~~~~~~~
// ~~~~~~~~~~~~~~~~ EVERY NODE IS A HEAD OF A LINKED LIST OF ELEMNTS ~~~~~~~~~~~~~~~~~
// ~~~~~~~~~~~~~~~~ WITH THE SAME KEY. ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


// ###################################################################################################
// ###################################################################################################
// INSERT PHASE
// ###################################################################################################
// ###################################################################################################

// ========================================================
// Function inputs are the tree and a new node.
// The operation is go down the tree, find the proper place
// for the new node and insert it.
// Have no return value.
// ========================================================
InsertNewNodeToTree(tree,new_node)
{
        tmp_curr_node = tree->head;
        done              = false;
        while (!done)
        {
                if (tmp_curr_node->key < new_node->key) //new in right sub-tree
                {
                        if (tmp_curr_node->right != null)
                        {
                        tmp_curr_node = tmp_curr_node->right;
                        }
                        else //found place
                        {
                                tmp_curr_node->right = new_node;
                                done = true;
                        }
                }
                else if (tmp_curr_node->key > new_node->key) //new in left sub-tree
                {
                        if (tmp_curr_node->left != null)
                        {
                        tmp_curr_node = tmp_curr_node->left;
                        }
                        else //found place
                        {
                                tmp_curr_node->left = new_node;
                                done = true;
                        }
                }
                else //if (tmp_curr_node->key == new_node->key) - new in linked list
                {
                        while (tmp_curr_node->link_list_nx != null)
                        {
                        tmp_curr_node = tmp_curr_node->link_list_nx;

                        }
                        tmp_curr_node->link_list_nx = new_node;
                        done = true;
                }
        }
    return;
}

// ###################################################################################################
// ###################################################################################################
// EXTRACT PHASE
// ###################################################################################################
// ###################################################################################################

// ========================================================
// Function inputs are the tree and a current node.
// The operation is to find its successive element in the
// tree, and this what the function returns.
// ========================================================
findNextNodeInTree(tree,curr_node)
{
        tmp_curr_node = curr_node;
        if (curr_node == max(tree))
        {
                nx_node = null; //No next element
        }
        else if (curr_node->right != null)  // Go to right subtree
        {
```

```
                nx_node = findMinOfSubTree(curr_node->right);
        }
        else
        {
                while (tmp_curr_node->parent->right == tmp_curr_node)
                {
                        tmp_curr_node = tmp_curr_node->parent;
                }
                nx_node = tmp_curr_node->parent; // We are left child of our parent
        }
        return nx_node;
}

// ========================================================
// Function input is a node.
// The operation is to find the minimal element in the
// subtree it spans, and this what the function returns.
// ========================================================
findMinOfSubTree(subtree_head_node)
{
        tmp_curr_node = subtree_head_node;
        while (tmp_curr_node->left != null)
        {
                tmp_curr_node = tmp_curr_node->left;
        }
        min_node = tmp_curr_node; // We are left-most leaf in the tree
        return min_node;
}
```

Visual example: Insert new node to BST:

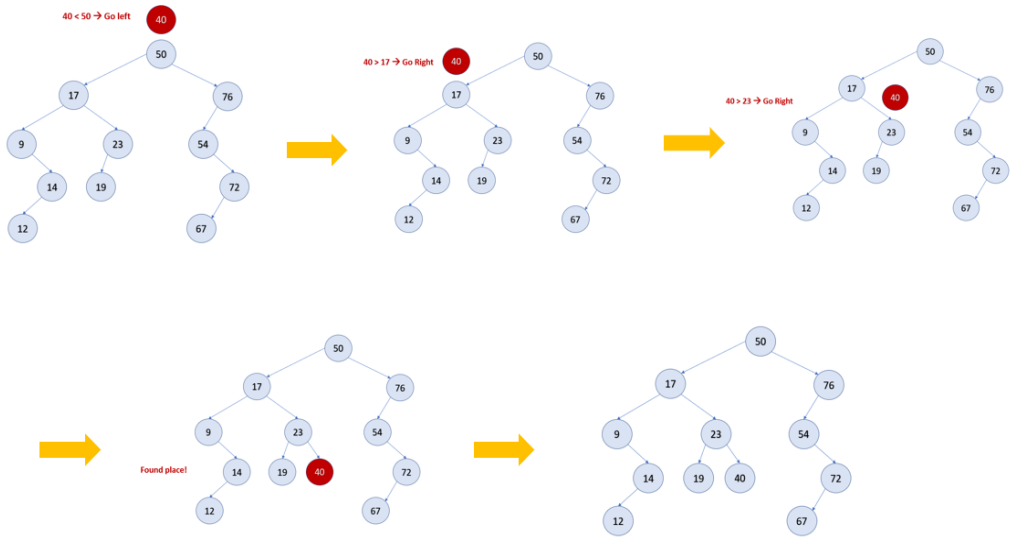
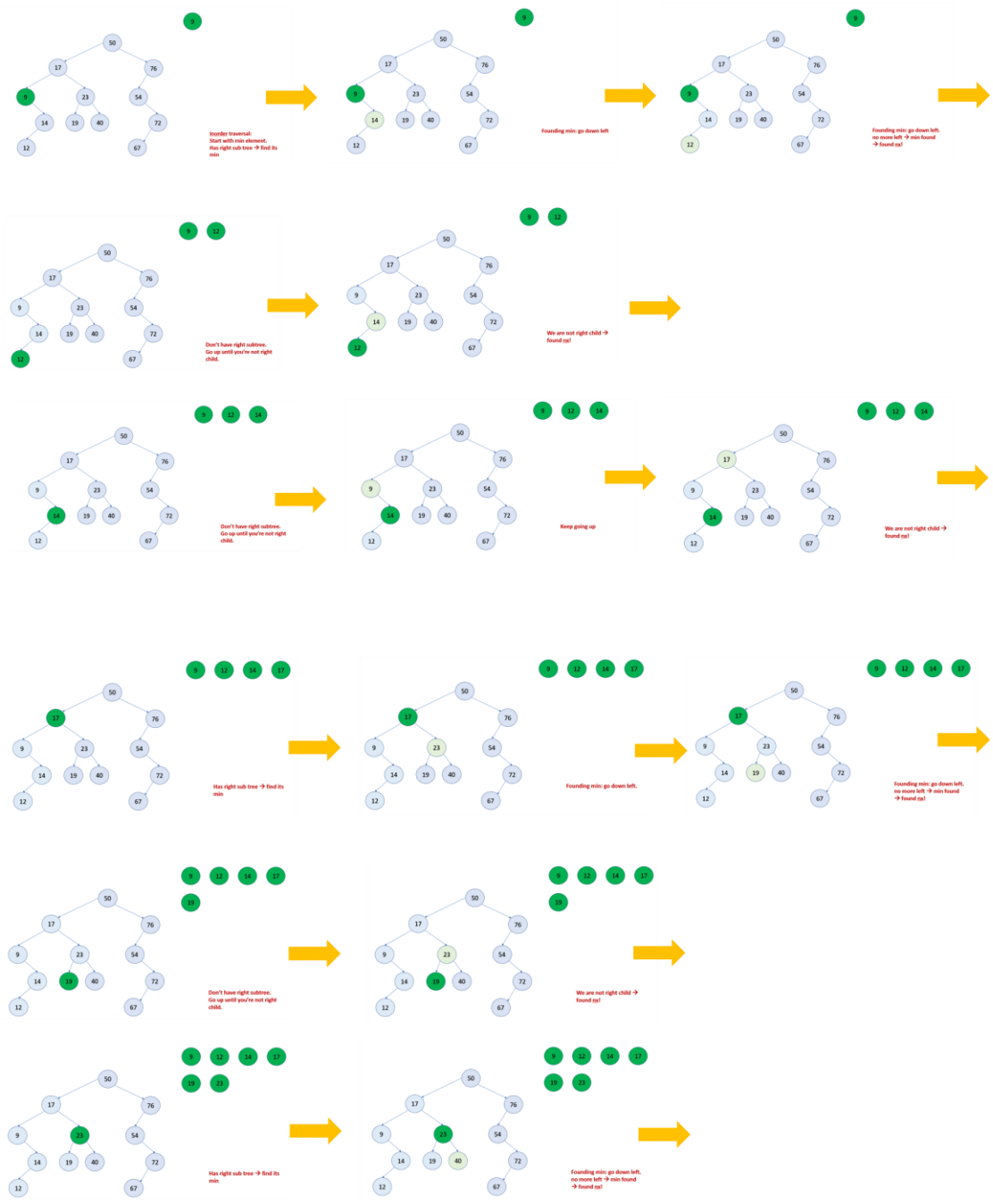


Figure 45 *: GEN_BST_SORTER algorithm example – insert mew node*

Visual example: BST inorder traversal:



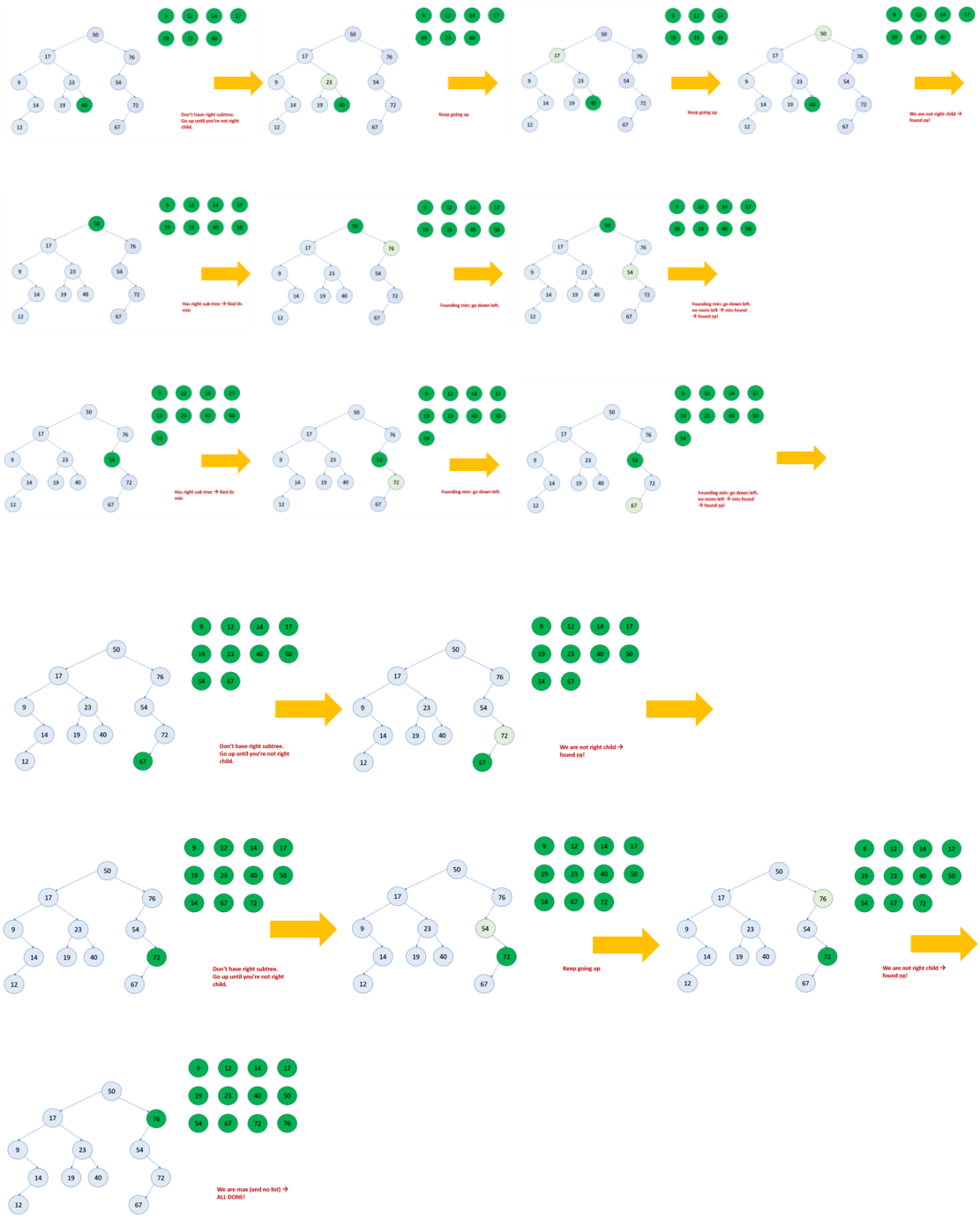


Figure 46 *: GEN_BST_SORTER algorithm example – inorder traversal*

Performance analysis:

The insertion stage of all the N elements combined takes in average $N \cdot \log(N)$ clock cycles, and in worst case can takes N^2 clock cycles.

The extraction stage, which is in-order traversal, takes in average $2N$ clock cycles, and in worst case takes $6N$ clock cycles.

2.5.6. GEN_PSEUDO_MODULUS

GEN_PSEUDO_MODULUS is an asynchronies parametric block, which its goal is to roughly calculate the modulus between two unsigned integers inputs, x%z. Whilst the result is guaranteed to be in the rage [0,z-1] as expected from modulus, the result in not necessary the real modulus between the numbers.

Motivation:

For this project, ga accelerator, there are some places random selections are required: random addresses of parents and random weights inside chromosomes (parent selection, crossover, mutation). Simply using some N random bits can cause problems because they might exceed the allowed range to select from, which is dictated by registers: the range to select from is between 0 to registers value minus 1 (cnfg_P for parent index, cnfg_M for weight index). So, the N random bits should be handled in some way in order to ensure they are in the allowed range. In addition, it should be taken into account that the result should be fairly distributed inside the allowed range in order to get the proper diversity in the population. Hence, solutions like rounding to the highest allowed number or constantly remove some MSBs are unacceptable.

From all above, the obvious solution is to use the modulus operation: N random bits % register, which will guarantee a result in the allowed range and will also don't compromise the random N bits distribution.

However, calculating the exact modulus between two unsigned integers can be challenging hardware task: it can take a lot of clock cycles such that in each cycle a single subtraction action will take place until reaching the result. Additional subtractors can be added in order to save clock cycles, but its a trade-off for more area that will be required. Alternately a LUT can be implemented, but this solution requires a lot of area or alternately limit very fast the range of the numbers (smaller LUT).

Hence, a creative solution that will not take much time or space was required. Since the origin of the data is random, then a pseudo-modulus operation was invented: its asynchronies hence doesn't consume time, it uses minimal area, it has minimal effect of the distribution of the input, and most importantly as the name suggests, its result is in the allowed range.

Parameters:

As mentioned, the block is parametric. It has a single parameter:

- **DATA_W** – Number of bits in each of the inputs

Interface:

signal name	input/output	Bits	pls/lvl	Description
Data IF: TOP/GA_MAIN_FSM				
i_valid_pls	Input	1	Pls	Uses for control friendly interface, has no real effect on the block. Represent a request for calculation.
i_x	Input	DATA_W	Pls	The dividend. Unsigned integer.
i_z	Input	DATA_W	Pls	The divisor. Unsigned integer.
o_res_valid_pls	Input	1	Pls	Uses for control friendly interface, has no real effect on the block. Immediate to i_valid_pls and represent that the result is ready.
o_res	Output	DATA_W	Pls	The pseudo-modulus result: o_res = i_x % i_z . in rage [0,i_z-1]. Unsigned integer.

***Table 12**: GEN_PSEUDO_MODULUS interface*

Diagram:

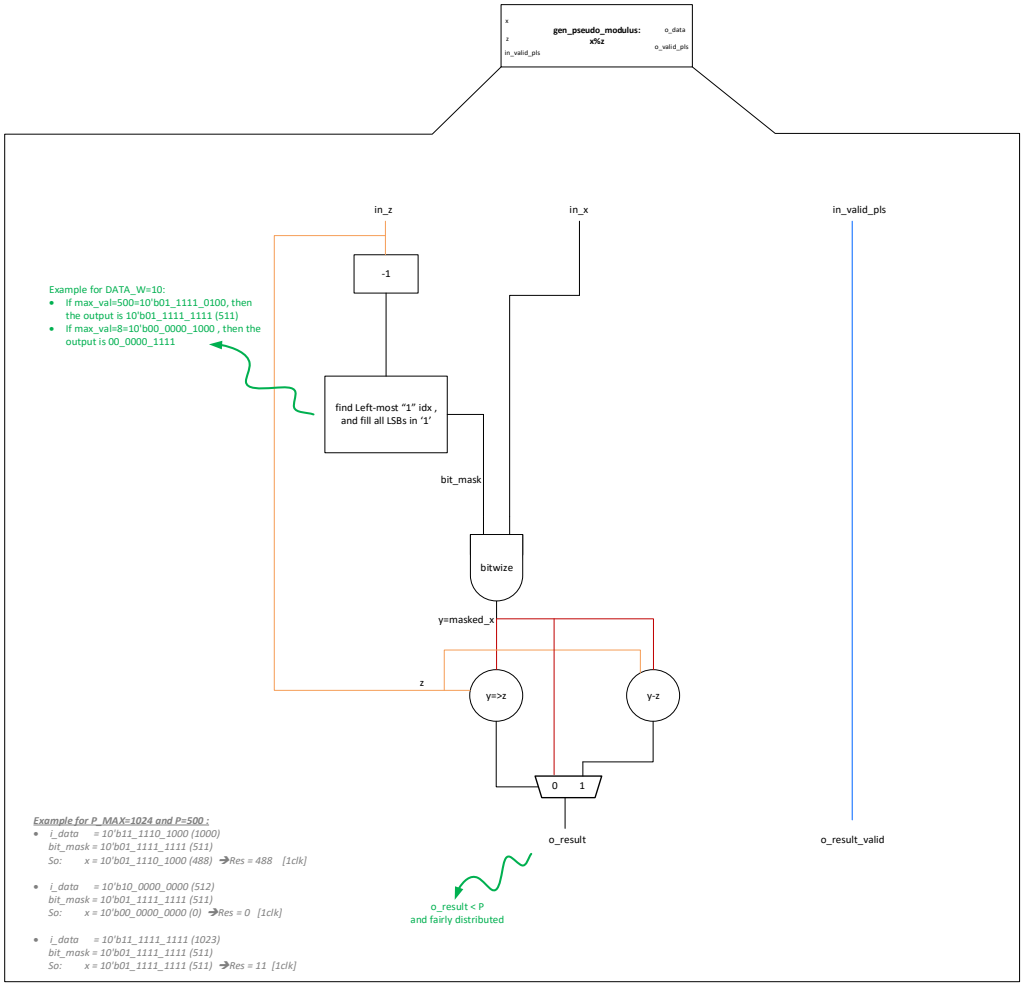


***Figure 47** : GEN_PSEUDO_MODULUS – microarchitecture*

Detailed description:

The calculation of the pseudo-modulus consists of few steps:

1. Subtract 1 from i_z in order to get the max allowed value of the result (i_z-1).
2. Find the MSB and create bit_mask by putting "1" at all the LSBs up to the MSB (including).
3. Create a masked_x: truncate i_x by doing bitwise AND with the mask.
4. The result (maksed_x) is either smaller than i_z, in which case masked_x is the result, or its one subtraction away from it, in which case this subtraction is done (masked_x-i_z) and its result is the pseudo-modulus result.

Some examples:

1. For DATA_W=10, i_z=500, i_x=488:

i_z	500=10'b01_1111_0100
max_res=i_z-1	499=10'b01_1111_0011
Mask	10'b01_1111_1111
i_x	488=10'b01_1110_1000
masked_x=i_x&mask	10'b01_1110_1000=488
o_res	masked_x is smaller than i_z => o_res=masked_x=488

2. For DATA_W=10, i_z=500, i_x=500:

i_z	500=10'b01_1111_0100
max_res=i_z-1	499=10'b01_1111_0011
Mask	10'b01_1111_1111
i_x	500=10'b01_1111_0100
masked_x=i_x&mask	10'b01_1111_0100=500
o_res	masked_x is NOT smaller than i_z => o_res=masked_x-i_z=0

3. For DATA_W=10, i_z=500, i_x=1000:

i_z	500=10'b01_1111_0100
max_res=i_z-1	499=10'b01_1111_0011
Mask	10'b01_1111_1111
i_x	1000=10'b11_1110_1000
masked_x=i_x&mask	10'b01_1110_1000=488
o_res	masked_x is smaller than i_z => o_res=masked_x=488

2.6. Synthesis

2.6.1. Technology and Constrains (SDC)

The synthesis was made in Tower 180um technology, with Design vision tool of Synopsys.

The floorplan was made by Innovus tool of Cadence.

The memories which are placed in the design are standard cell RAM memories of Tower 180um technology. Single clock and single reset (active low) were used, where as the clock frequency taken is 125MHz due to memories limitations.

No other constrains were applied, and no multi cycle paths were defined.

2.6.2. Floorplan

Floorplan snapshot from Innovus tool:

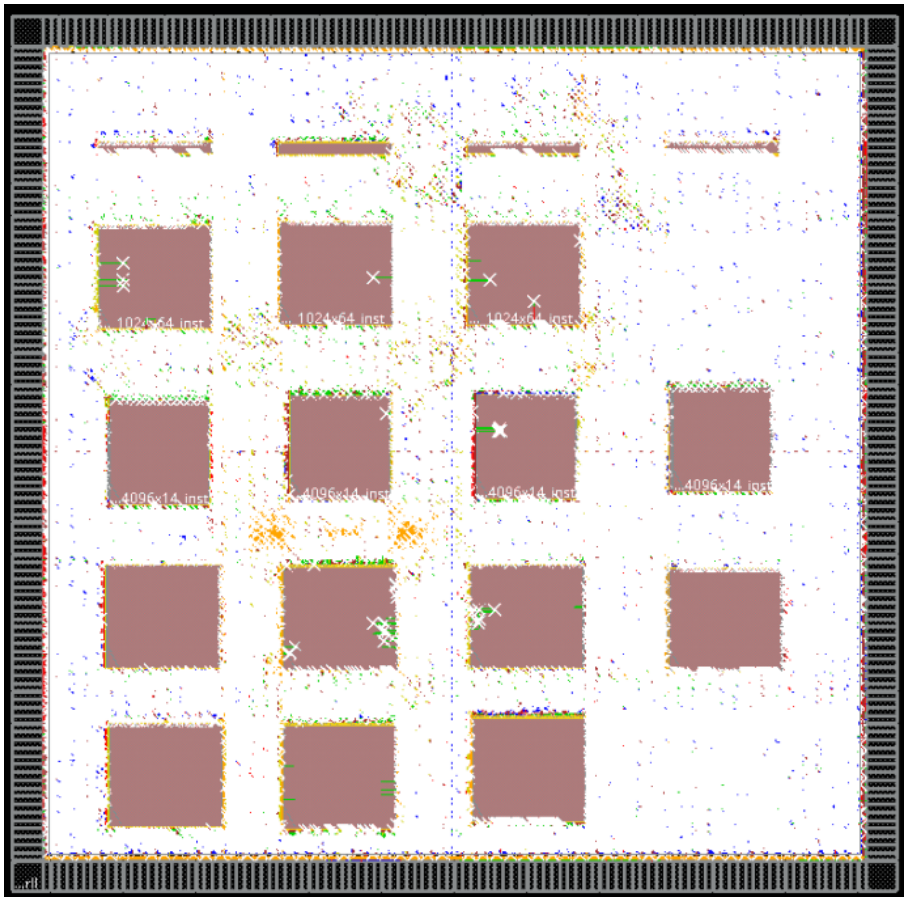


Figure 48 *: GA accelerator floorplan (snapshot from Innovus tool)*

- The pink rectangles in the floorplan snapshoot are the memories. As explained in section 2.1.1 there are 5 memories needed:
    - V&d buffer memory: single port, 64x198 bits (64 rows, each 198 bits).
    - Chromosomes queue memory: dual port, 1024x192 bits (1024 rows, each 192 bits).
    - Tanh LUT memory: single port, 16384x14 bits (16384 rows, each 14 bits).
    - Sorter internal memory: single port, 1024x256 bits (1024 rows, each 256 bits).
    - Sorted pool memory: single port, 1024x192 bits (1024 rows, each 192 bits).

  However, one can see that there are 18 memories in the above floorplan (figure 48), because in practice only memories of 3 sized were in use: 1024x64, 64x64, 4096x14. By using mux-demux structures the given memories were concatenated to each other to simulate bigger memories at the required size.

  1. **V&d buffer memory**:
     Location: 1st row in the floorplan image (figure 48).
     Needed memory (logical wrapper memory): 64x198 bits (64 rows, each 198 bits).
     Used memories: 4 memories of 64x64, i.e. 64x(4x64).

  2. **Chromosomes queue memory**:
     Location: 2nd row in the floorplan image (figure 48).
     Needed memory (logical wrapper memory): 1024x192 bits (1024 rows, each 192 bits).
     Used memories: 3 memories of 1024x64, i.e. 1024x(3x64).

  3. **Tanh LUT memory**:
     Location: 3rd row in the floorplan image (figure 48).
     Needed memory (logical wrapper memory): 16384x14 bits (16384 rows, each 14 bits).
     Used memories: 4 memories of 4096x14, i.e. (4x4096)x14.

  4. **Sorter internal memory**:
     Location: 4th row in the floorplan image (figure 48).
     Needed memory (logical wrapper memory): 1024x256 bits (1024 rows, each 256 bits).
     Used memories: 4 memories of 1024x64, i.e. 1024x(4x64).

  5. **Sorted pool memory**:
     Location: 5th row in the floorplan image (figure 48).
     Needed memory (logical wrapper memory): 1024x192 bits (1024 rows, each 192 bits).
     Used memories: 3 memories of 1024x64, i.e. 1024x(3x64).

  Overall:

  $$4+3+4+4+3=18$$

  Units of memories

2.6.3. Area and StaticPower

Area report:

```
****************************************
Report : area
Design : ga_top
Version: R-2020.09-SP2
Date   : Fri Dec 24 00:40:35 2021
****************************************

Library(s) Used:
        tsl18fs120_typ (File:
        /tools/kits/tower/PDK_TS18SL/FS120_STD_Cells_0_18um_2005_12/DW_TOWER_tsl18fs120/2005.12/sy
        nopsys/2004.12/models/tsl18fs120_typ.db)
        dpram1024x64_CB_typ (File: /users/epjuma/memories/db/dpram1024x64_CB_typ.db)
        dpram4096x14_cb_typ (File: /users/epjuma/memories/db/dpram4096x14_cb_typ.db)
        dpram64x64_cb_typ (File: /users/epjuma/memories/db/dpram64x64_cb_typ.db)

Number of ports:                        87320
Number of nets:                        285452
Number of cells:                       199677
Number of combinational cells:         184782
Number of sequential cells:              7476
Number of macros/black boxes:              18
Number of buf/inv:                      61846
Number of references:                      13

Combinational area:             228375.500000
Buf/Inv area:                    35312.750000
Noncombinational area:           39262.750000
Macro/Black Box area:           924284.539062
Net Interconnect area:          106521.037405

Total cell area:               1191922.789062
Total area:                    1298443.826467
```

StaticPower report summary:

```
****************************************
Report : power
        -analysis_effort low
Design : ga_top
Version: R-2020.09-SP2
Date   : Fri Dec 24 00:41:01 2021
****************************************


Library(s) Used:

     tsl18fs120_typ (File:
     /tools/kits/tower/PDK_TS18SL/FS120_STD_Cells_0_18um_2005_12/DW_TOWER_tsl18fs120/2005.12/synops
     ys/2004.12/models/tsl18fs120_typ.db)
    dpram1024x64_CB_typ (File: /users/epjuma/memories/db/dpram1024x64_CB_typ.db)
    dpram4096x14_cb_typ (File: /users/epjuma/memories/db/dpram4096x14_cb_typ.db)
    dpram64x64_cb_typ (File: /users/epjuma/memories/db/dpram64x64_cb_typ.db)


Operating Conditions: tsl18fs120_typ   Library: tsl18fs120_typ

Wire Load Model Mode: enclosed



Global Operating Voltage = 1.8
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  =  96.0478 mW   (95%)
  Net Switching Power  =   4.7708 mW    (5%)
                         ---------
Total Dynamic Power    = 100.8185 mW  (100%)

Cell Leakage Power     =  63.6161 uW
```

```
                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000            0.0000             0.0000  (   0.00%)
memory            17.1883           0.1540        5.7940e+07            17.4003  (  17.25%)
black_box          0.0000           0.0000            0.0000             0.0000  (   0.00%)
clock_network      0.0000           0.0000            0.0000             0.0000  (   0.00%)
register          77.7388           0.2184        9.3608e+05            77.9573  (  77.28%)
sequential         0.0000           0.0000            0.0000             0.0000  (   0.00%)
combinational      1.1201           4.3984        4.7404e+06             5.5229  (   5.47%)
--------------------------------------------------------------------------------------------------
Total             96.0472 mW        4.7708 mW     6.3616e+07 pW       100.8805 mW
```

2.6.4. Worst slack analysis

Critical path (worst slack) report:

```
Information: Updating design information... (UID-85)
Warning: Design 'ga_top' contains 1 high-fanout nets. A fanout number of 1000 will be used for
delay calculations involving these nets. (TIM-134)

****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
        -sort_by group
Design : ga_top
Version: R-2020.09-SP2
Date   : Fri Dec 24 00:40:09 2021
****************************************

 # A fanout number of 1000 was used for high fanout net computations.

Operating Conditions: tsl18fs120_typ   Library: tsl18fs120_typ
Wire Load Model Mode: enclosed

  Startpoint: cnfg_p_reg[4]
              (rising edge-triggered flip-flop clocked by clk)
  Endpoint:
u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_pool_inst/MEM_IF_DEPTH_GT_64_LTE_1024.INST_4W
IDTH_LOOP[0].u_mem_1024x64_inst
             (falling edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  ga_top             2000000               tsl18fs120_typ
  ga_core_top        2000000               tsl18fs120_typ
  ga_selection_parents
                     4000                  tsl18fs120_typ
  gen_pseudo_modulus_x_mod_z_DATA_W11
                     4000                  tsl18fs120_typ
  gen_pseudo_modulus_x_mod_z_DATA_W11_DW01_sub_1
                     ForQA                 tsl18fs120_typ
  ga_selection       540000                tsl18fs120_typ

  Point                                                   Incr       Path
  --------------------------------------------------------------------------
  clock clk (rise edge)                                   0.00       0.00
  clock network delay (ideal)                             0.00       0.00
  cnfg_p_reg[4]/CP (dfcrq4)                               0.00 #     0.00 r
  cnfg_p_reg[4]/Q (dfcrq4)                                0.32       0.32 f
  u_ga_core_inst/cnfg_p[4] (ga_core_top)                  0.00       0.32 f
  u_ga_core_inst/U8/ZN (invbd4)                           0.05       0.37 r
  u_ga_core_inst/U19/ZN (inv0da)                          0.05       0.42 f
  u_ga_core_inst/u_ga_algo_top_inst/cnfg_p[4] (ga_algo_top)
                                                          0.00       0.42 f
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/cnfg_p[4] (ga_selection)
                                                          0.00       0.42 f
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/cnfg_p[4]
(ga_selection_parents)
                                                          0.00       0.42 f
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/U6/Z (mx02d1)
                                                          0.21       0.63 f
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/i_z[4]
(gen_pseudo_modulus_x_mod_z_DATA_W11)
                                                          0.00       0.63 f
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U33/ZN (inv0d1)
                                                          0.12       0.74 r
```

```
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U82/ZN (nd02d2)
                                                  0.07       0.81 f
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U93/ZN (inv0d2)
                                                  0.06       0.87 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U40/ZN (nd02d2)
                                                  0.06       0.93 f
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U94/ZN (inv0d2)
                                                  0.05       0.98 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U81/ZN (nd02d2)
                                                  0.05       1.03 f
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U95/ZN (inv0d2)
                                                  0.05       1.09 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U84/ZN (nd02d2)
                                                  0.05       1.14 f
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U96/ZN (inv0d2)
                                                  0.06       1.20 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U83/ZN (nd02d2)
                                                  0.07       1.27 f
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U35/ZN (nr02d2)
                                                  0.08       1.35 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U39/Z (xr02d2)
                                                  0.29       1.64 f
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U38/ZN (nd02d2)
                                                  0.08       1.72 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U37/ZN (inv0d2)
                                                  0.02       1.74 f
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U100/ZN
(oai211d1)
                                                  0.10       1.84 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U28/ZN (inv0d2)
                                                  0.03       1.87 f
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U101/ZN
(oai211d1)
                                                  0.11       1.98 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U34/ZN (inv0d2)
                                                  0.03       2.01 f
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U102/ZN
(oai211d1)
                                                  0.09       2.10 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U89/ZN (nd02d0)
                                                  0.11       2.22 f
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U17/ZN (inv0d0)
                                                  0.43       2.65 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/sub_111/A[6]
(gen_pseudo_modulus_x_mod_z_DATA_W11_DW01_sub_1)
                                                  0.00       2.65 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/sub_111/U47/ZN
(nd02d1)
                                                  0.19       2.84 f
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/sub_111/U96/ZN
(inv0d0)
                                                  0.29       3.13 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/sub_111/U29/ZN
(aoi21d1)
                                                  0.09       3.22 f
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/sub_111/U27/ZN
(oai21d1)
                                                  0.10       3.32 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/sub_111/U90/ZN
(xn02d2)
                                                  0.26       3.58 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/sub_111/DIFF[8]
(gen_pseudo_modulus_x_mod_z_DATA_W11_DW01_sub_1)
                                                  0.00       3.58 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/U130/ZN
(oaim22d1)
                                                  0.14       3.72 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/u_x_mod_z_inst/o_res[8]
(gen_pseudo_modulus_x_mod_z_DATA_W11)
                                                  0.00       3.72 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/U891/Z (an02d1)
                                                  0.11       3.83 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_parents_inst/pool_mem_rd_addr[8]
(ga_selection_parents)
                                                  0.00       3.83 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/U195/Z (mx02d4)
                                                  0.17       4.00 r
  u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_pool_inst/addr[8]
(custom_spmem_wrapper_DATA_W192_DEPTH1024_SIM_DLY1)
                                                  0.00       4.00 r

u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_pool_inst/MEM_IF_DEPTH_GT_64_LTE_1024.INST_4W
IDTH_LOOP[0].u_mem_1024x64_inst/A1[8] (dpram1024x64_CB)
                                                  0.00       4.00 r
  data arrival time                                          4.00

  clock clk (fall edge)                           4.00       4.00
  clock network delay (ideal)                     0.00       4.00
```

```
u_ga_core_inst/u_ga_algo_top_inst/u_selection_inst/u_pool_inst/MEM_IF_DEPTH_GT_64_LTE_1024.INST_4W
IDTH_LOOP[0].u_mem_1024x64_inst/CEB1 (dpram1024x64_CB)
                                                          0.00       4.00 f
  library setup time                                      0.00       4.00
  data required time                                                 4.00
  --------------------------------------------------------------------------
  data required time                                                 4.00
  data arrival time                                                 -4.00
  --------------------------------------------------------------------------
  slack (MET)                                                        0.00
```

Analysis:

Timing requirements were met.

The technology of the memories requires 8ns clock cycle, thus a path of read or write request to the memories is expected to be a critical path. Most of the read/write requests to the memories were sampled and only in the selection block there were a few unsampled paths of requests to the memories that were unsampled due to performance considerations. Hence, overall one of these paths was expected to be a critical path, which indeed happened according to the report above.

The critical path is at the parent selection phase, a path that chose the address of a parents to read from the sorted pool.

The path starts at the rising edge of the clock at cnfg_P register. After propagation delay D→Q in the register the value goes to ga_selection_parents block, the block that calculates address to read parents from and sends them to sorted pool mem. At ga_selection_parents the calculation is done: the decision of which address to read from includes getting a random number and calculating its pseudo-modulus with cnfg_P register, in order to a legal memory address (number between 0 and cnfg_P-1). The pseudo-modulus calculation is done in x_mod_z block, which takes most of the path's time (since it contains comparators and subtractors). Once the result is ready the output from x_mod_z goes to the memory and arrives at 4ns: the exact time needed (slack is 0) for the memory to get the request, since the memories samples the requests at negative-edge of the clock.

2.7. Performance

Pipeline diagram of GA accelerator is presented in figure 15.

In this chapter the average steady state latency is analyzed, i.e. after the first B^{th} {V,d} sets were received and after GA_INIT_POP is done, and for average timing of the sorter.

Every {V,d} input is calculated separately and the next {V,d} cannot be received until the previous is finished. Hence, the latency will be calculated per {V,d}. In addition, for a single {V,d} cnfg_G generations are performed in order to get the result, and every generation starts with all the chromosomes inside GA_SELECTION sorted-pool and ends with all the new chromosomes sorted in the sorter. Between generations a transition of data form inside the sorter to the sorted-pool is occurred, during this time no other action is performed in the pipe.

The latency in clock cycles per chromosomes for each algo block:

- $Latency(GA\_SELECTION - parents) = 4$
- $Latency(GA\_CROSSOVER) = 1$
- $Latency(GA\_MUTATION) = 1$
- $Latency(GA\_FITNESS) = 13 \cdot cnfg\_B$
- $Average\ \ Latency(GA\_SELECTION - sorter) = \log_2(cnfg\_P)$

Hence, for one chromosome the total latency is:

$$Average\ \ Latency(one\ chromosome) = 6 + 13 \cdot cnfg\_B + \log_2(cnfg\_P)\ [clock\ \ cycles]$$

The full life cycle of a generation is partially piped:

The first new chromosome takes $6 + 13 \cdot cnfg\_B + \log_2(cnfg\_P)$ to get into its place in the sorter, and while it was in GA_FITNESS and entered into the sorter, other chromosomes were accumulated in the chromosomes queue, waiting for the sorter at the rate of 1 chromosome every 6 clock cycles. Because cnfg_P maximal value is 1024, the average number of clock cycles to get into the sorter is 10, which makes GA_FITNESS the unit that will set the rate. Overall, one generation full transition from GA_SELECTION sorted pool into the sorter takes:

$$Average\ \ Latency(one\ generation) =$$
$$= 6 + 13 \cdot cnfg\_B + \log_2(cnfg\_P) + 13 \cdot cnfg\_B \cdot (cnfg\_P - 1)\ [clock\ \ cycles]$$

And including the transition between the sorter to the sorted-pool:

$$Average\ \ Latency(one\ generation) =$$
$$= 6 + 13 \cdot cnfg\_B + \log_2(cnfg\_P) + 13 \cdot cnfg\_B \cdot (cnfg\_P - 1) + cnfg\_P \cdot \log_2(cnfg\_P)[clock\ cycles] =$$
$$= 6 + cnfg\_P \cdot 13 \cdot cnfg\_B + (1 + cnfg\_P) \cdot \log_2(cnfg\_P)[clock\ cycles]$$

In conclusion, since there are cnfg_G generations, **GA accelerator average steady state latency and throughput**, in respect to valid results (per {V,d} set):

$$Latency = cnfg\_G \cdot (6 + cnfg\_P \cdot 13 \cdot cnfg\_B + (1 + cnfg\_P) \cdot \log_2(cnfg\_P)) \left[\frac{clock\ cycles}{result}\right]$$

$$Throughput = \frac{1}{Latency} = \frac{1}{cnfg\_G \cdot (6 + cnfg\_P \cdot 13 \cdot cnfg\_B + (1 + cnfg\_P) \cdot \log_2(cnfg\_P))} \left[\frac{result}{clock\ cycles}\right]$$

And in bits units, since the result is both w_vec and o_y:

$$Latency = \frac{cnfg\_G \cdot (6 + cnfg\_P \cdot 13 \cdot cnfg\_B + (1 + cnfg\_P) \cdot \log_2(cnfg\_P))}{(cnfg\_M + 1) \cdot DATA\_W} \left[\frac{clock\ cycles}{bit}\right]$$

$$Throughput = \frac{1}{Latency} = \frac{(cnfg\_M + 1) \cdot DATA\_W}{cnfg\_G \cdot (6 + cnfg\_P \cdot 13 \cdot cnfg\_B + (1 + cnfg\_P) \cdot \log_2(cnfg\_P))} \left[\frac{bit}{clock\ cycles}\right]$$

- For clock cycle of 8ns, DATA_W=6 and :

  cnfg_M = 7 cnfg_P = 16 cnfg_B = 16 cnfg_G = 10

$$Latency = 34020\left[\frac{clock\ cycles}{result}\right] = 272160\left[\frac{ns}{result}\right] = 708.75\left[\frac{clock\ cycles}{bit}\right] = 5670\left[\frac{ns}{bit}\right]$$

$$Throughput = \frac{1}{Latency} = 2.94 \cdot 10^{-5}\left[\frac{result}{clock\ cycles}\right] = 3.67 \cdot 10^{-5}\left[\frac{result}{ns}\right] = 1.41 \cdot 10^{-5}\left[\frac{bit}{clock\ cycles}\right] = 1.76 \cdot 10^{-4}\left[\frac{bit}{ns}\right]$$

2.8. Zero-order ("aliveness") verification

Zero-order verification was made to GA accelerator in top level and in block level for main functional units: all 5 GA_ALGO blocks, GA_MAIN_FSM and the three generic modules: GEN_INNER_PRODUCT, GEN_BST_SORTER and GEN_PSEUDO_MODULUS.

Important notes:

- **All of the numerical signals are in decimal notation, unless mentioned otherwise.**
- **No checkers were used and the results were examined visually.**

2.8.1. GA Accelerator – Top Level Test

In the following sub section (2.8.1) GA accelerator top waveforms are presented and also the waveforms of all the GA units mentioned above **as part of the top level test**, i.e. the same top-level-test in different places in the design. Additional explanations regarding the scenario simulated are under figure 50 (section 2.8.1.1).

2.8.1.1 GA Accelerator Top Level Waveform

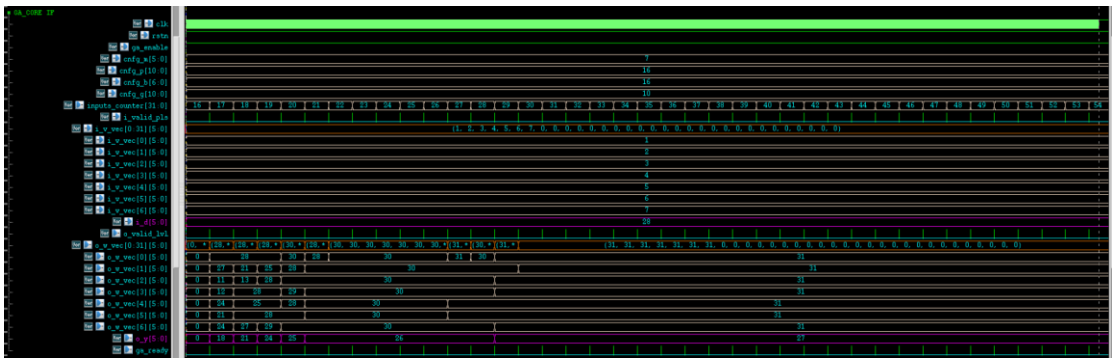


***Figure 49** : GA accelerator top level waveform – all test zoom-out*

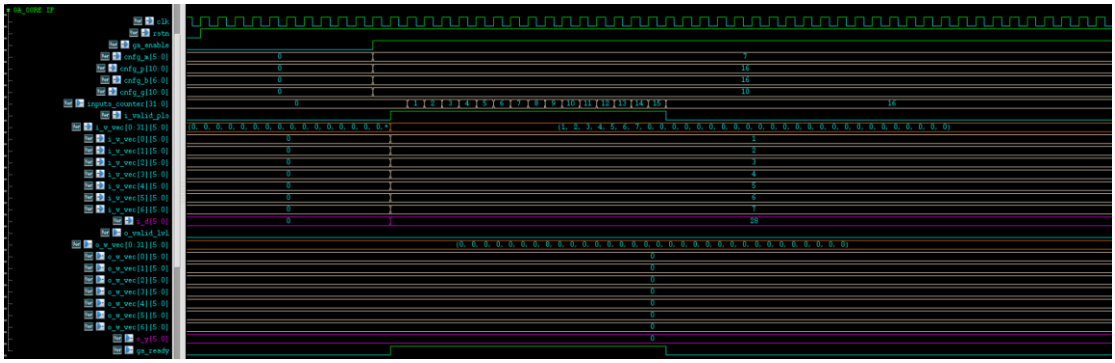


***Figure 50** : GA accelerator top level waveform – start of test zoom-in*

The scenario described in figure 49 is the top test scenario, and as mentioned is common to all the waveforms presented in this report.

As one can see, the registers configurations are:

cnfg_M = 7 cnfg_P = 16 cnfg_B = 16 cnfg_G = 10

The top level test simulates one activation of GA accelerator, i.e. one rise of ga_enable with a sequence of {V_vec,d} sets with the same real W_vec (same W_vec for which d = V_vec · real_W_vec). Multiple activations, i.e. few rises of ga_enable for different filters (different W vectors) or different configurations, were not tested.

The test includes constant inputs (both V_vec and d) when the relation between them is the sum operation, i.e. W_vec elements are expected to be all 1s, since all of the valid inputs throughout the entire simulation are:

$$i\_vec[0] = 6'b0.0\_0001 = 0\cdot 2^{0} + 0\cdot 2^{-1} + 0\cdot 2^{-2} + 0\cdot 2^{-3} + 0\cdot 2^{-4} + 1\cdot 2^{-5} = \frac{1}{32} \xrightarrow{\text{in signed int}} 6'd1$$

$$i\_vec[1] = 6'b0.0\_0010 = 0\cdot 2^{0} + 0\cdot 2^{-1} + 0\cdot 2^{-2} + 0\cdot 2^{-3} + 1\cdot 2^{-4} + 0\cdot 2^{-5} = \frac{2}{32} \xrightarrow{\text{in signed int}} 6'd2$$

$$i\_vec[2] = 6'b0.0\_0011 = 0\cdot 2^{0} + 0\cdot 2^{-1} + 0\cdot 2^{-2} + 0\cdot 2^{-3} + 1\cdot 2^{-4} + 1\cdot 2^{-5} = \frac{3}{32} \xrightarrow{\text{in signed int}} 6'd3$$

$$i\_vec[3] = 6'b0.0\_0100 = 0\cdot 2^{0} + 0\cdot 2^{-1} + 0\cdot 2^{-2} + 1\cdot 2^{-3} + 0\cdot 2^{-4} + 0\cdot 2^{-5} = \frac{4}{32} \xrightarrow{\text{in signed int}} 6'd4$$

$$i\_vec[4] = 6'b0.0\_0101 = 0\cdot 2^{0} + 0\cdot 2^{-1} + 0\cdot 2^{-2} + 1\cdot 2^{-3} + 0\cdot 2^{-4} + 1\cdot 2^{-5} = \frac{5}{32} \xrightarrow{\text{in signed int}} 6'd5$$

$$i\_vec[5] = 6'b0.0\_0110 = 0\cdot 2^{0} + 0\cdot 2^{-1} + 0\cdot 2^{-2} + 1\cdot 2^{-3} + 1\cdot 2^{-4} + 0\cdot 2^{-5} = \frac{6}{32} \xrightarrow{\text{in signed int}} 6'd6$$

$$i\_vec[6] = 6'b0.0\_0111 = 0\cdot 2^{0} + 0\cdot 2^{-1} + 0\cdot 2^{-2} + 1\cdot 2^{-3} + 1\cdot 2^{-4} + 1\cdot 2^{-5} = \frac{7}{32} \xrightarrow{\text{in signed int}} 6'd7$$

$$i\_d = real\_W \cdot i\_vec = \sum_{n=0}^{6} i\_vec[n] = \frac{1+2+3+4+5+6+7}{32} = \frac{28}{32} \xrightarrow{\text{in signed int}} 6'd28$$

So overall in 6bits signed-fixed-point terminology, with MSB as sign bit and the rest fractional, each element of W_vec is expected to be 6'b01_1111 (which is the closest value to 1 possible in this format) – 31 in signed decimal and $\frac{31}{32} = 0.96875$ in unsigned fixed-point with 1 bit for sign and 5 bits for fractional part.

- Explanation:

  $$i\_d = real\_W \cdot i\_vec = \sum_{n=0}^{6} i\_vec[n] = \frac{1+2+3+4+5+6+7}{32} = \frac{28}{32} \xrightarrow{\text{in signed int}} 6'd28$$

  $$\Rightarrow real\_W \sim 6'b0.1\_1111 \xrightarrow{\text{in signed int}} 6'd31$$

  $$indeed:$$

  $$real\_W \cdot i\_vec = \frac{1\cdot 31 + 2\cdot 31 + 3\cdot 31 + 4\cdot 31 + 5\cdot 31 + 6\cdot 31 + 7\cdot 31}{32\cdot 32} = \frac{28\cdot 31}{32\cdot 32} = 0.8477 - closest\ to\ 1\ possiable$$

GA accelerator outputs value over time are in figures 51-52 below:

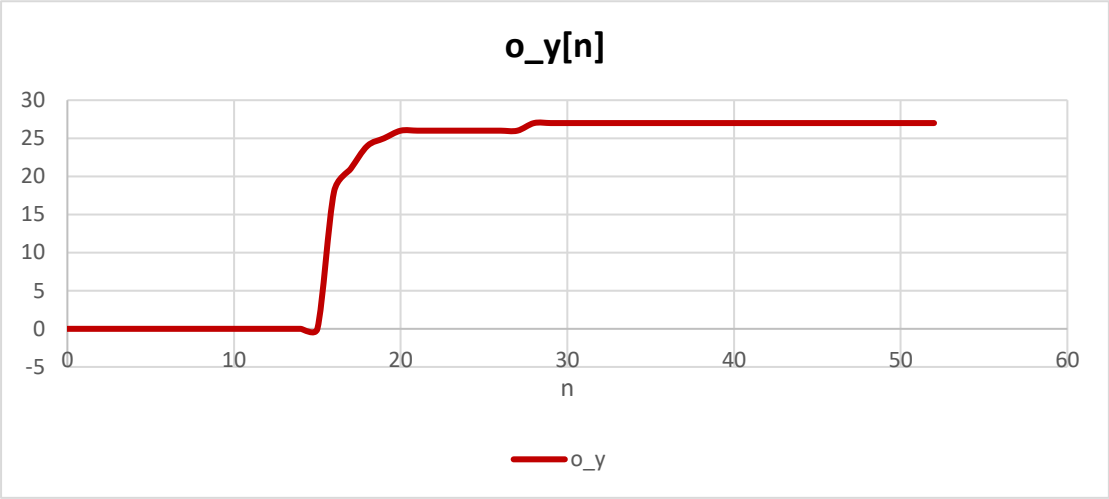


***Figure 51** : GA accelerator top level test – o_y over time*

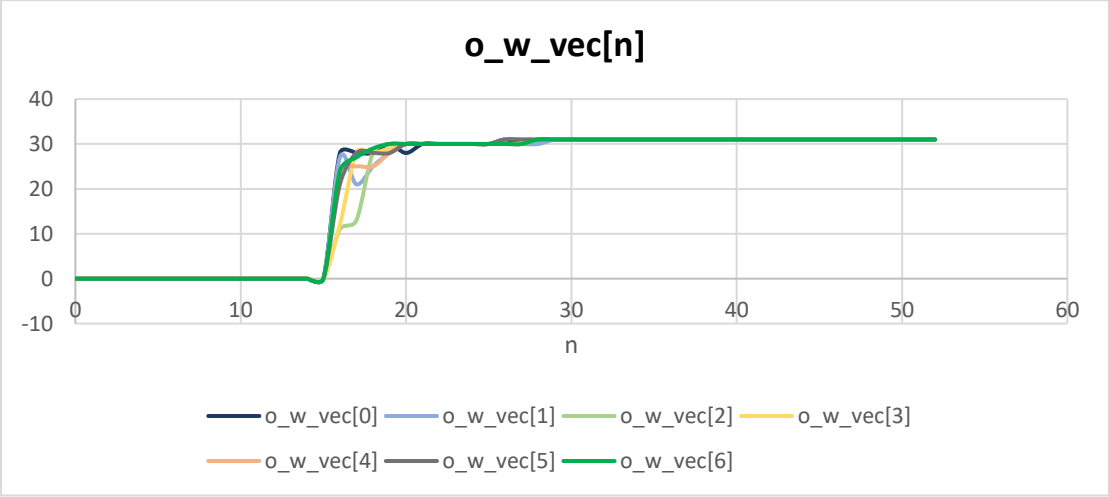


***Figure 52** : GA accelerator top level test – o_w_vec over time*

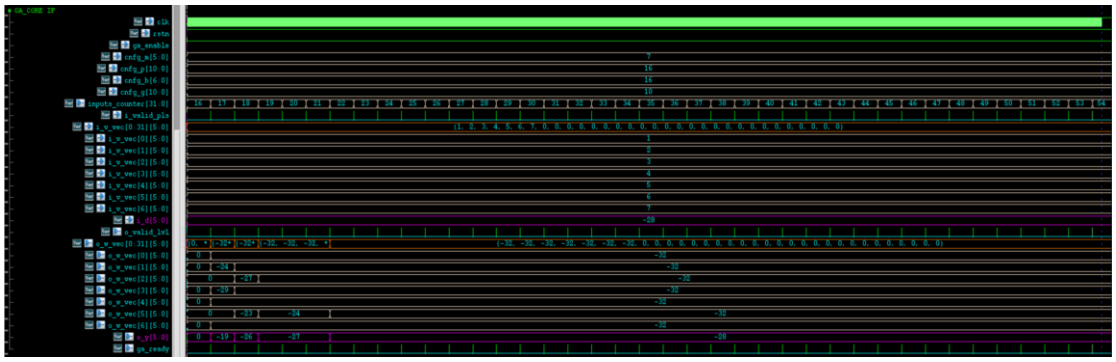
From the waveform in figure 49 and from the graphs above one can see that:

- 54 inputs have been received by GA accelerator, and as mentioned they are all with the same data (i_vec,i_d).
- Over time o_y output of the GA accelerator is approaching the reference signal i_d.
- Over time o_w_vec elements are getting closer to each other, i.e. uniformed filter is created, and the values increasing towards 31, which is their desired value.

To summarize, one can see that GA accelerator pass aliveness test and seems to perform well for a basic case, i.e. level zero pass.

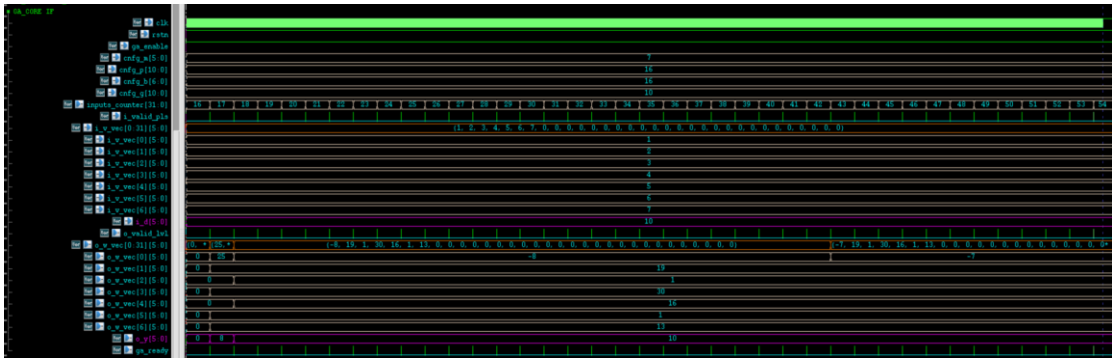
- Additional simple scenarios were checked in top level and got good results as well:
  - ***Constant i_v_vec and i_d:***

$$\left.\begin{array}{l} i\_v\_vec = \{1,2,3,4,5,6,7\} \\ i\_d = -1 \cdot i\_v\_vec = -28 \\ real\_W = \{-1,-1,-1,-1,-1,-1,-1\} \cdot 32 \end{array}\right\} \rightarrow \begin{array}{c} o\_y = -28 \\ o\_w\_vec = \{-32,-32,-32,-32,-32,-32,-32\} \end{array}$$



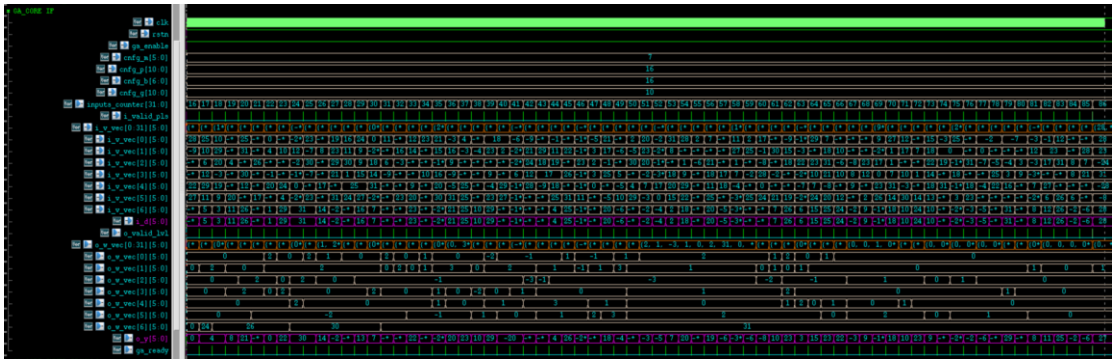
  - ***Constant i_v_vec and i_d:***

$$\left.\begin{array}{l} i\_v\_vec = \{1,2,3,4,5,6,7\} \\ i\_d = 10 \\ real\_W = \{1,1,1,1,0,0,0\} \cdot 32 \end{array}\right\} \rightarrow \begin{array}{c} o\_y = 10 \\ o\_w\_vec = \{-7,19,1,30,16,1,13\} \end{array}$$



  - ***Only last element:***

$$\left.\begin{array}{l} i\_v\_vec = random \\ i\_d = i\_v\_vec[6] \\ real\_W = \{0,0,0,0,0,0,1\} \cdot 32 \end{array}\right\} \rightarrow o\_w\_vec = \{\sim 0/1, \sim 0/1, \sim 0/1, \sim 0/1, \sim 0/1, \sim 0/1, 31\}$$



In the second scenario It's interesting to see that o_y result is correct or very close to it, but the o_w_vec creating is not. The reason for it is multiple degrees of freedom in this example (in contrast to the original scenario and the first and third example here). Thus, this outcome should not be surprising, and future idea can be to add penalty for longer filters (more elements in o_w_vec that are not 0s). However, this feature concerns algorithmic level and not a result of the implementation.

Additional thing to notice is the data interface communication – behaves as expected: once ga_enable rose ga_ready stayed steady until the 16th input, and in its arrival ga_ready dropped and rose again only at the end of the calculation, along with o_valid.

2.8.1.2 GA_MAIN_FSM Waveform

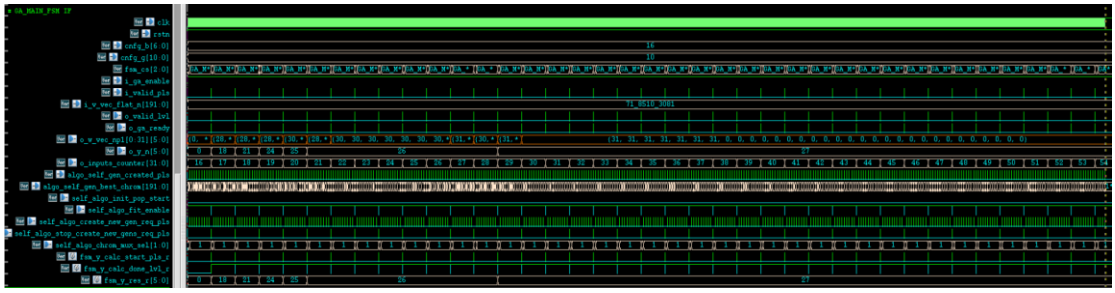


***Figure 53** : GA_MAIN_FSM waveform – full*

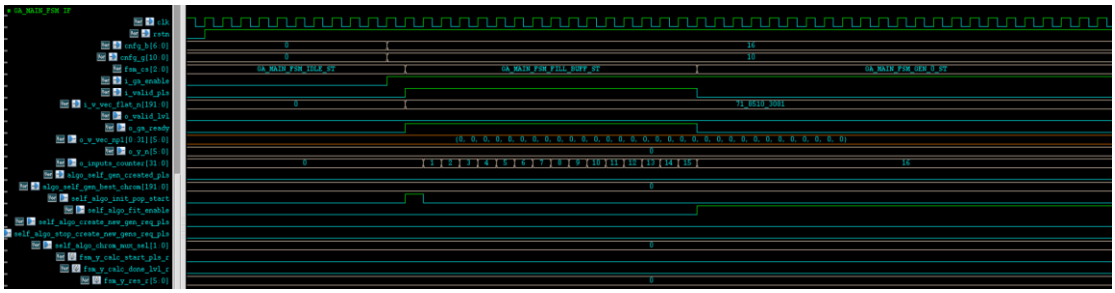


***Figure 54** : GA_MAIN_FSM waveform – zoom-in: start of simulation*

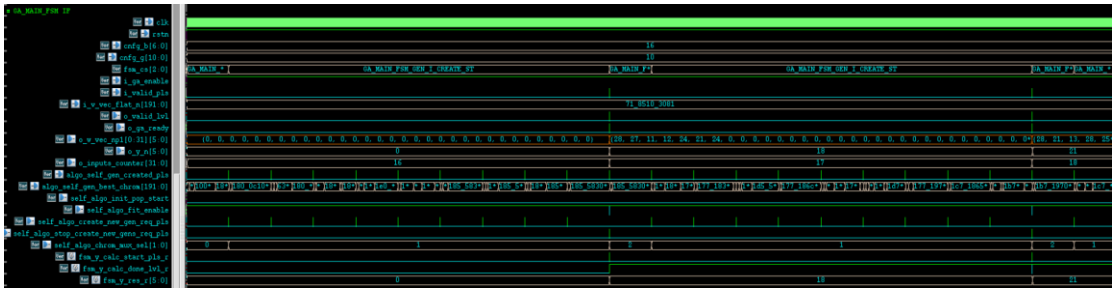


***Figure 55** : GA_MAIN_FSM waveform – zoom-in: first two sets of {V,d} that gets o_valid*

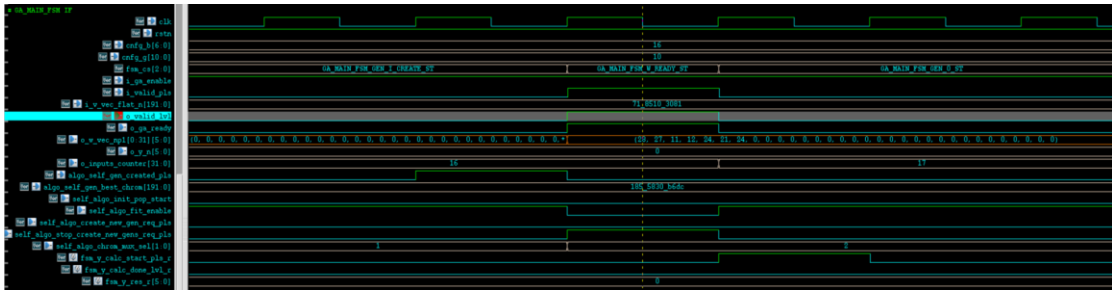


***Figure 56** : GA_MAIN_FSM waveform – zoom-in: first set of {V,d} that gets o_valid – rise of o_valid*

In figure 53 the complete waveform of GA_MAIN_FSM is presented, and in figures 54-56 zoom-in for chosen signals and events are presented.

In figure 54 the start of the simulation is zoomed-in and fsm_cs is the register that saves GA_MAIN_FSM state, and the behavior is as expected: GA_MAIN_FSM starts at idle state, once ga_enable rises it send GA_INIT_POP start signal and goes to FILL_BUFF state. There it waits to more inputs to arrive while keeping ga_ready high, and at the 16th i_valid desserts it and goes to GEN0 state, there it waits until GA_SELECTION sends it gen_created_pls signal, as seen in figure 55. At this point GA_MAIN_FSM sends to GA_SELECTION the response of create another generation and goes to GEN_I_CREATE state, as expected. At GEN_I_CREATE state GA_MAIN_FSM counts the generations, and when the Gth (10th) generation is created GA_MAIN_FSM disables GA_FITNESS so it won't pull chromosomes from the queue, rises o_valid and ga_ready and goes to W_READY state as shown in the zoom-in in figure 56.

Overall one can see that the behavior of GA_MAIN_FSM is as expected and as described in section 2.5.1.

2.8.1.3 GA_INIT_POP Waveform

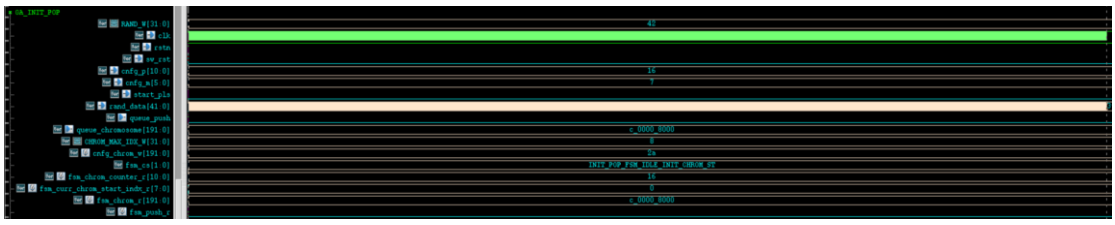


***Figure 57** : GA_INIT_POP waveform - full*

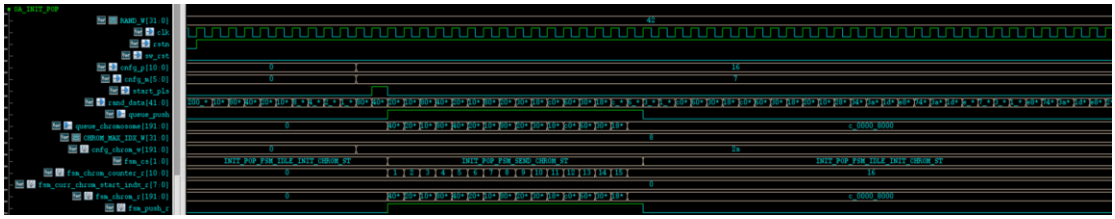


***Figure 58** : GA_INIT_POP waveform – zoom-in to start of test*

In figure 57 one can see GA_INIT_POP chosen signals throughout the entire simulation. As expected, they seem constant and GA_INIT_POP FSM is at idle state and doesn't send push requests to chromosomes queue. GA_INIT_POP is active only one time at the start of the simulation, as expected and as showed in figure 58 in zoom-in: since cnfg_M=7 the chromosomes are short and contains 42 bits each, which allows back-to-back push requests to the chromosomes queue. Once start_pls rises the FSM goes to SEND_CHROM state and sends a burst of 16 chromosomes (cnfg_P) to the chromosomes queue, and afterwards goes back to IDLE state.

- In block level simulations both short and extended chromosomes configurations were checked, and also a sequence of giving few start_pls pulses (that simulates a few rises of ga_enable) was checked.

2.8.1.4 GA_FITNESS Waveform

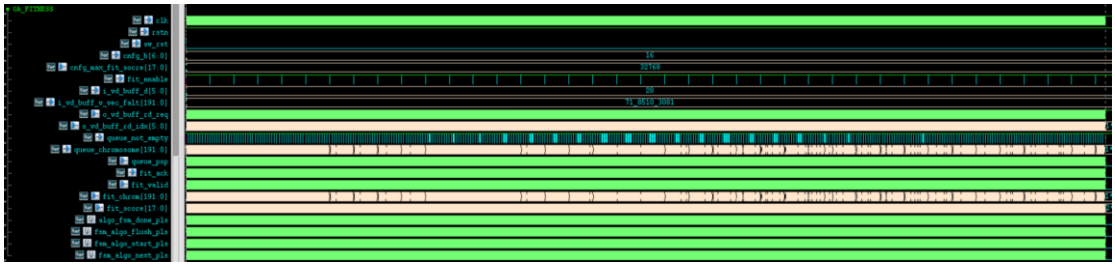


***Figure 59** : GA_FITNESS waveform – full simulation*

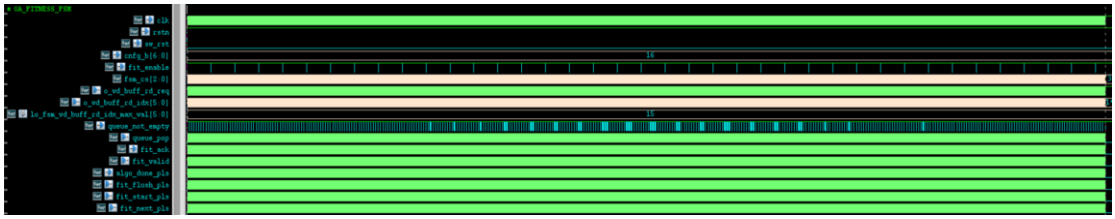


***Figure 60** : GA_FITNESS_FSM waveform – full simulation*

In figures 59-60 above some chosen signals from GA_FITNESS unit are presented as they toggle throughout the entire time of the simulation, as expect. GA_FITNESS pass aliveness check.

To exam basic logic, zoom-in for the beginning of the simulation for GA_FITNESS_FSM is shown in figures 61-63:

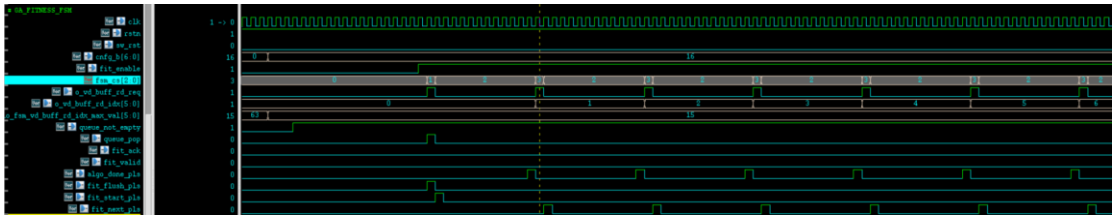


***Figure 61** : GA_FITNESS_FSM waveform – zoom-in: start of simulation*

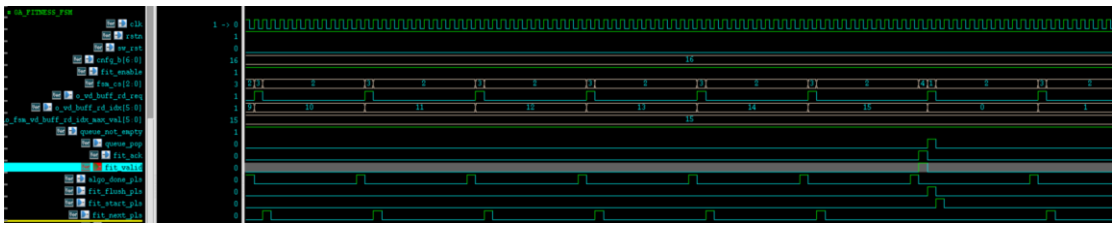


***Figure 62** : GA_FITNESS_FSM waveform – zoom-in: first fit_valid*

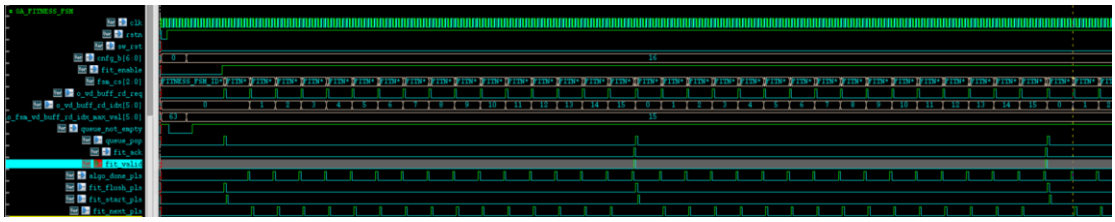


***Figure 63** : GA_FITNESS_FSM waveform – zoom-in: two first chromosomes calculation*

GA_FITNESS_FSM state is represented by the signal fsm_cs. The states encoding is as follows:

- IDLE = 0
- POP_REQ = 1
- ALGO_BUSY = 2
- ALGO_NEXT = 3
- FIT_DONE = 4

At figure 61 the beginning of the simulation is presented. GA_FITNESS_FSM starts at IDLE state and once fit_enable rises starts to work. At this point it goes to POP_REQ state (queue is not empty) and the next cycle gets the data both for V&d buffer and the chromosomes queue. It sends fit_start_pls to GA_FITNESS_ALGO to trigger it to start working on the first iteration for the current chromosomes, and then goes to ALGO_BUSY state to wait for GA_FITNESS_ALGO to finish. Once ALGO unit sends algo_done to the FSM, the FSM goes to ALGO_NEXT state and sends read request to V&d buffer with the next index. The next clock it gets the data and sends fit_next to GA_FITNESS_ALGO and goes to ALGO_BUSY again and so on, until all the 16 V&d samples were read. As it shown in figure 62, once this is done and GA_FITNESS_ALGO sends algo_done for the 16th time, GA_FITNESS_FSM rises fit_valid and goes to FIT_DONE state, there it waits for fit_ack. Since this is the first chromosome the sorter in GA_SELECTION is free and returns immediate ack, and

GA_FITNESS_FSM continues to POP_REQ state again and performs the same process for the next chromosome, as shown in figure 63.

Performance measurements has been done, and the distance between two adjacent algo_done, i.e. the time required for a single {V_vec,d} is 13 clock cycles as expected. The total time for one chromosome full calculation is 13*cnfg_B+1=209 clock cycles, as expected as well.

- Important observation: in the current test, as a result of the ratio between cnfg_B and cnfg_P, the sorter is always faster than GA_FITNESS and hence fit_ack is immediate. In block level simulation that was made non-immediate acks were checked as well.

GA_FITNESS_ALGO dataflow for random chromosome is presented in figures 64 below:

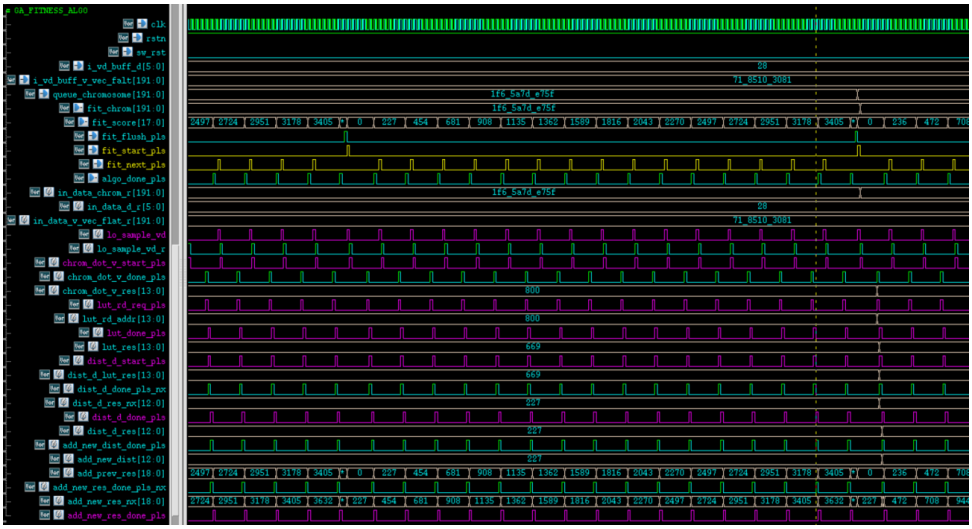


Figure 64 *: GA_FITNESS_ALGO waveform – zoom-in: single chromosome flow*

In the above figure, zoom-in between two fit_start pulses is presented. The number of fit_next signals between them is 15 as expected, and the result data increases every algo_done as it should. Also, in high level one can see the propagation of the indicators for every operation (start and done pulses for every operation).

In order to observe them even more closely, a deeper zoom-in is presented in figure 65, in this case zoom-in between two random fit_next signals:

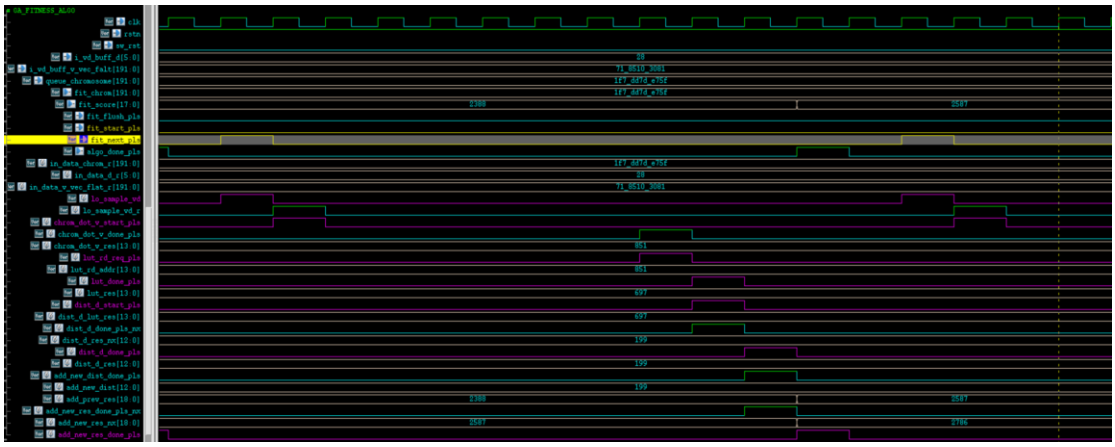


Figure 65 *: GA_FITNESS_ALGO waveform – zoom-in: single {V_vec,d} flow*

At first the new {V,d} set is sampled. Afterwards start trigger is sent to GEN_INNER_PRODUCT module who calculates the inner product of the chromosomes and V_vec. After 7 clocks it is done and the result is passed as an address to the tanh LUT, which at one clock latency outputs the result (memory). Then, the distance between the result from the LUT to d is calculated, which takes another clock cycle, and eventually it adds up to the previous result and done_algo pulse is sent to GA_FITNESS_FSM.

As mentioned and seen again here (figure 65), this process takes 13 clock cycles.

Overall, aliveness, zero order logic and timing were all checked in a GA_FITNESS at random times visually and seem to work as expected.

- Smaller tests were made for each one of the fixed-point computational units (adder, find distance), and the overall fitness score values were checked and pass at block level simulation that was made for GA_FITNESS_ALGO.

2.8.1.5 GA_SELECTION Waveform

In figures 66-67 below some chosen signals from GA_SELECTION unit are presented as they toggle throughout the entire time of the simulation, as expect.
GA_SELECTION pass aliveness check:

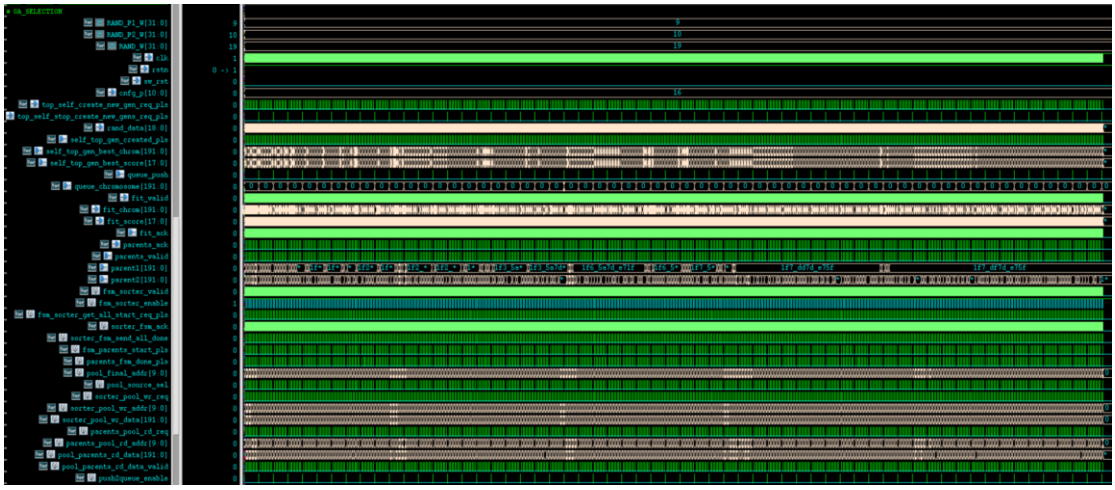


***Figure 66** : GA_SELECTION waveform – full simulation*

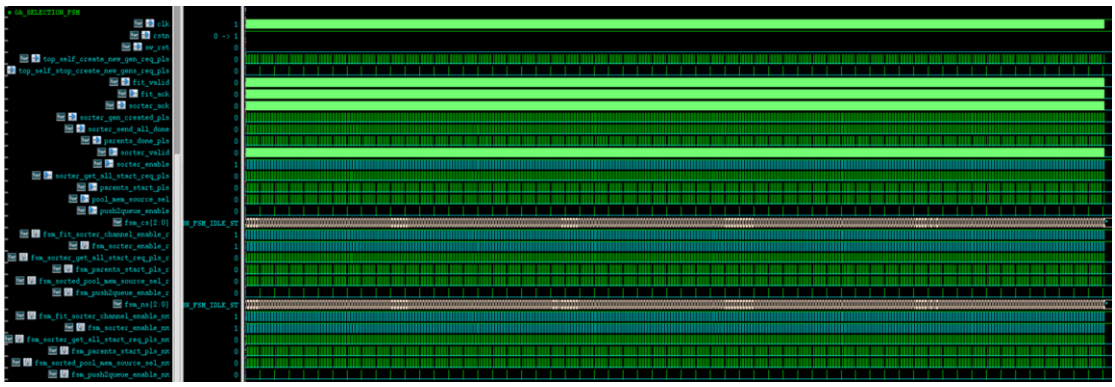


***Figure 67** : GA_SELECTION_FSM waveform – full simulation*

To exam basic logic, zoom-in snapshoots from the beginning of the simulation for GA_SELECTION_FSM and GA_SELECTION_SORTER are shown in figures 68-71:

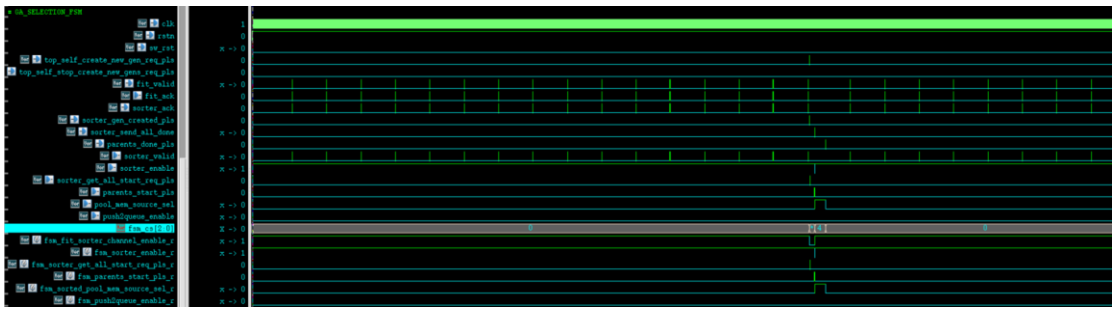


***Figure 68** : GA_SELECTION_FSM waveform – zoom-in: first generation sort*

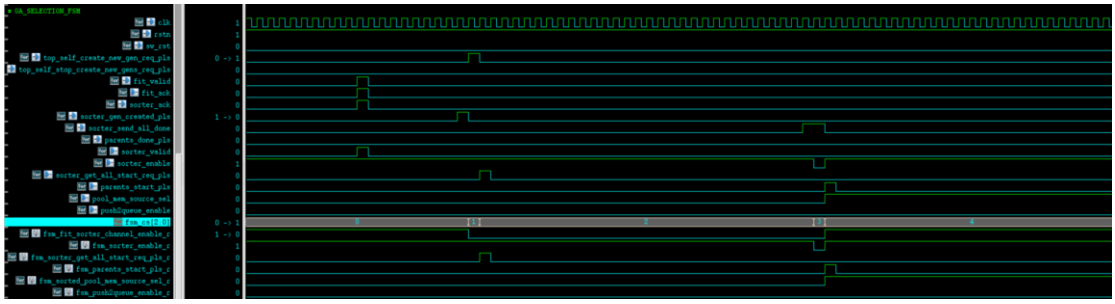


Figure 69 *: GA_SELECTION_FSM waveform – zoom-in: first generation end of sort*

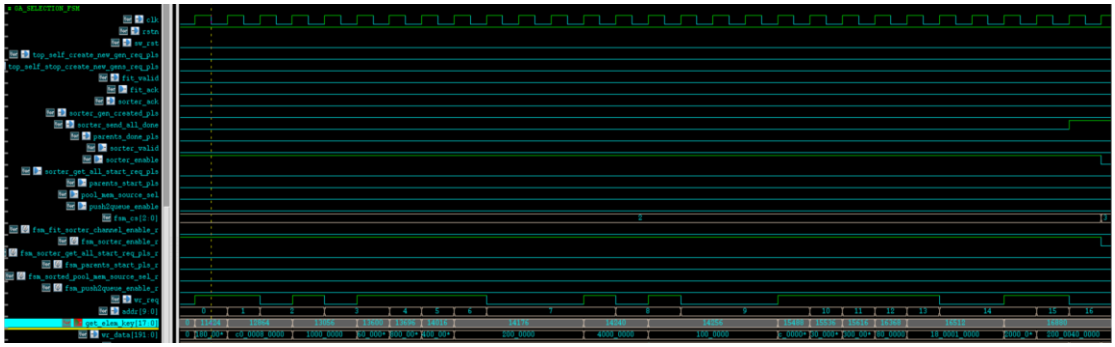


Figure 70 *: GA_SELECTION_FSM waveform – zoom-in: first generation write to sorted-pool*

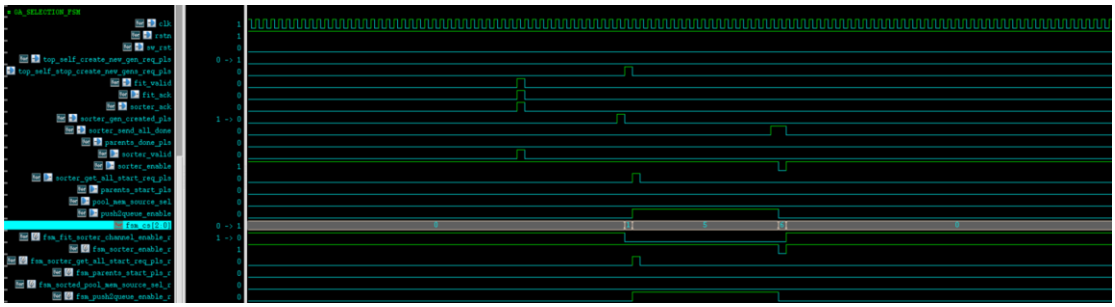


Figure 71 *: GA_SELECTION_FSM waveform – zoom-in: end of last generation*

In figures 68-71 GA_SELECTION_FSM chosen signals for the first handled {V,d} set (i_v_vec[15],d[15]) are presented.

GA_SELECTION_FSM state is represented by the signal fsm_cs. The states encoding is as follows:

- IDLE = 0
- CHECK_MAIN_FSM_RESP = 1
- WR2POOL = 2
- NEW_GEN_RESET_SORTER = 3
- SELECT_PARENTS = 4
- PUSH2QUEUE = 5
- DONE_GENS_RESET_SORTER = 6

In figure 68 top view of the first generation is presented. Most of the time the FSM is in IDLE state as expected, with the channel between GA_FITNESS and the sorter opened and the sorter enabled. Pulses of fit_valid and fit_ack are seen, until the sorter sends gen_created_pls. At this point another zoom-in is shown in figure 69 and one can see that as expected the FSM closes the GA_FITNESS-sorter channel and goes to CHECK_RESP state, there it gets the response of creating a new generation. As a result, in the next clock cycle the FSM goes to WR2POOL state and sends the sorter a request to get all the data in order. Note that sorter-pool source selector (fsm_sorted_pool_mem_source_sel_r) is low which represent that the source is the sorter. The FSM stays in this state for a few clock cycles until all the chromosomes are written to the pool and the sorter sends a notification that it happened.

- In figure 70 greater zoom-in is presented, that shows that indeed in those cycle where the FSM is at WR2POOL state there are write requests to the memory. Note that there are exactly 16 (cnfg_P) write requests, with ascending fitness_score, i.e. to the lowest address the best chromosome is written as required, and after the 16th write request the sorter sends that the process was complete (asserting sorter_send_all_done signal).
    - Block level simulation for the sorter is at section 2.8.2.2.

Back to figure 69, after the sort is done GA_SELECTION_FSM goes to NEW_GEN_RESET_SORTER state for one clock cycle and disables the sorter. After that, it goes to SELECT_PARENTS state and sends start pulse to GA_SEOECTION_PARENTS as expected, enables the sorter again including opening the channel between it and GA_FITNESS, and changes the memory source selector to 1 (GA_SELECTION_PARENTS).

The FSM remains in SELECTION_PARENTS state for few clock cycles, and in figure 68 its shown that it stays there until it gets done signal from GA_SELECTION_PARENTS (parents_done_pls). Once that happens, it goes back to IDLE state and waits again for the sorter to notify that a full generation has been created and sorted.

The above process continues again and again until the last generation. In the case of the last generation when GA_SELECTION_FSM is in CHECK_RESP state GA_MAIN_FSM sends stop_create_new_gens signal, as shown in figure 71. After getting this signal GA_SELECTION_FSM goes to PUSH2QUEUE state and it stays there until sorter finishes to push the chromosomes to the queue. Afterwards the FSM goes to DONE_GENS_RESET_SORTER in order to reset the sorter, and eventually one cycle later goes back to IDLE, to wait again for the next generation to be fully inside the sorter.

As mentioned, once GA_SELECTION_FSM is at SELECT_PERETS state the unit GA_SELECTION_PARENTS is active. Random operation for a single generation is displayed in figure 72 below:

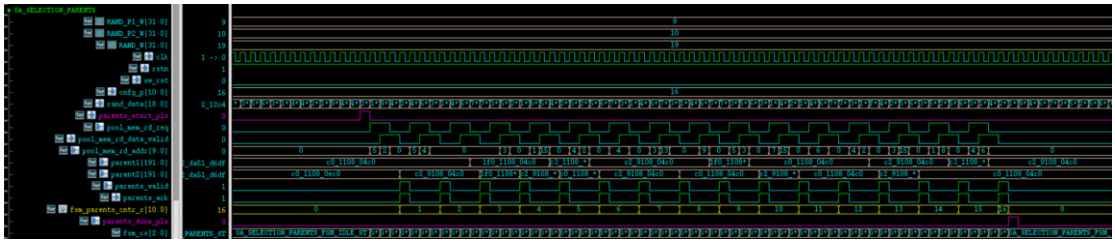


***Figure 72** : GA_SELECTION_PARENTS waveform – zoom-in: single generation*

In figure 72 a single full iteration of GA_SELECTION_PARENTS is presented, i.e. for a single generation the process of selecting 16 pairs of parents (cnfg_P). Parents counter register (fsm_parents_cntr_r) is counting from 1 to 16 each rise of parents_valid, and once reaching the last pair it goes back to zero after getting ack (parents_ack). The ack is immediate due to the fact GA_CROSSOVER is always opened to receive new parents since its operation is one clock cycle. Memory read requests can be observed as well, including the address (decimal): as required, the addresses are in the legal range of [0,16-1] and parent1 address (first request at each two adjacent requests) is always smaller or equal to 7.

Regarding timing performance one can see from figure 72 that the time required for a single pair of parents is 4 clock cycles, as required.

- The sorter time, both sort phase and extract phase, is irrelevant because it cannot be set deterministically due to reasons mentioned in section 2.5.3.3.

2.8.1.6 GA_CROSSOVER Waveform

In figure 73 below some chosen signals from GA_CROSSOVER unit are presented as they toggle throughout the entire time of the simulation, as expect. GA_CROSSOVER pass aliveness check:

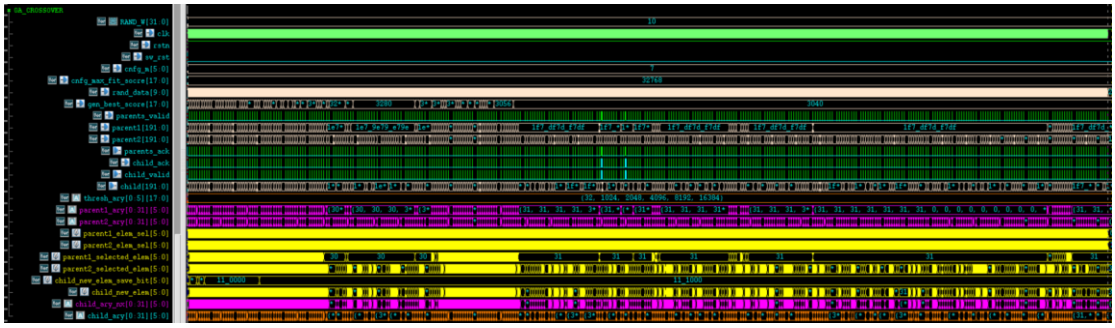


***Figure 73** : GA_CROSSOVER waveform – full simulation*

In the above signals the maximal possible fitness score for the current configurations is shown: 32768 in unsigned integer or in unsigned fixed-point with 8 int bits and 10 fractional bits is $2^5 = 32 = 2 \cdot cnfg\_b$, which is as expected. The thresholds array that is created as a result is presented as well, and also achieve the desired values.

To exam basic logic, zoom-in snapshots of different {V,d} from few random places throughout the simulation are shown in figures 74-76:

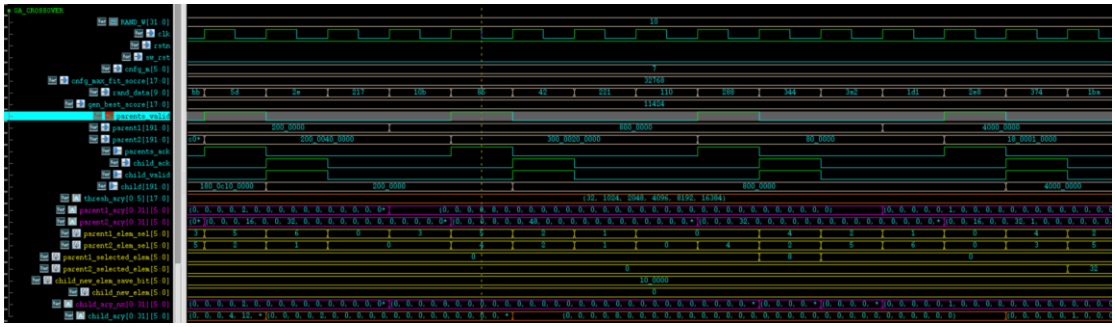


***Figure 74** : GA_CROSSOVER waveform – zoom-in: early {V_vec,d}*

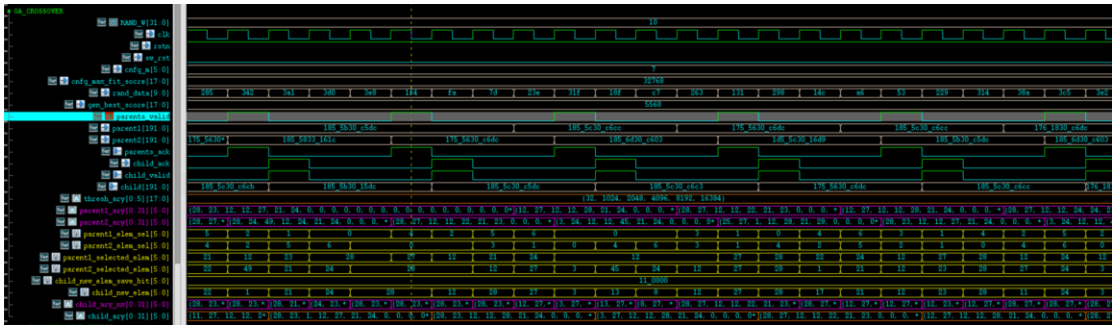


***Figure 75** : GA_CROSSOVER waveform – zoom-in: intermediate {V_vec,d}*

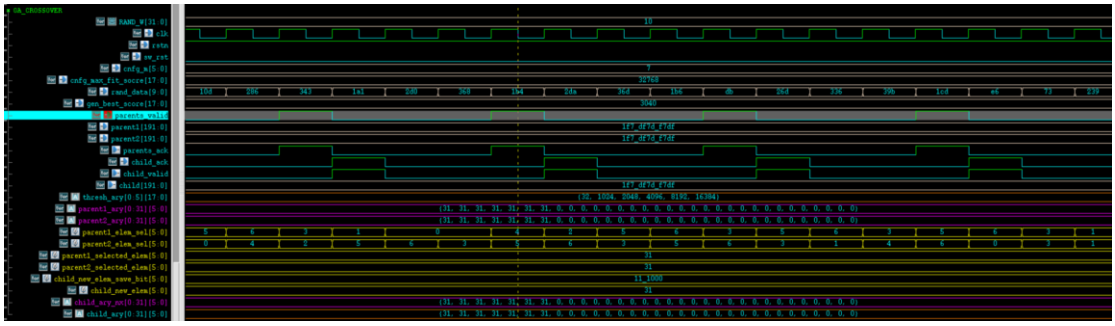


***Figure 76** : GA_CROSSOVER waveform – zoom-in: lasts {V_vec,d}*

In figure 74 a random crossover operation is presented form one of the first {V_vec,d} sets. The generation best result is high, i.e. not very well. In unsigned int the best result is 11424. It is smaller only than the highest threshold in thresh_ary. Hence, as expected, only the MSB of the chosen weight will be saved (child_new_elem_save_bit in binary is 10_0000, meaning only bit 5 will be saved). At a random parent_valid one can see that the weights selector from each parent is in the allowed range ([0,cnfg_M-1]=[0,6]), and that the proper bit exchanged was made.

All of this is done in one clock cycle and the next clock cycle the modified child is outputted (child_valid).

Similarly, all the crossovers zoomed-in in figures 75-76 are acting the same way as expected, except the generation's best score that is getting smaller over time and according to it more bits are saved.

2.8.1.7 GA_MUTATION Waveform

In figure 77 below some chosen signals from GA_MUTATION unit are presented as they toggle throughout the entire time of the simulation, as expect. GA_MUTATION pass aliveness check:

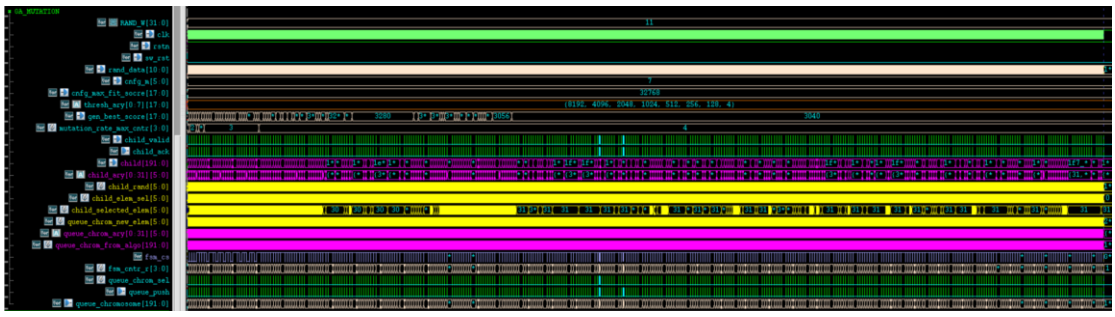


***Figure 77** : GA_MUTATION waveform – full simulation*

In the above signals the maximal possible fitness score for the current configurations is shown: 32768 in unsigned integer or in unsigned fixed-point with 8 int bits and 10 fractional bits is $2^5 = 32 = 2 \cdot cnfg\_b$, which is as expected. The thresholds array that is created as a result is presented as well, and also achieve the desired values.

To exam basic logic, zoom-in snapshots of different {V,d} from few random places throughout the simulation are shown in figures 78-79:

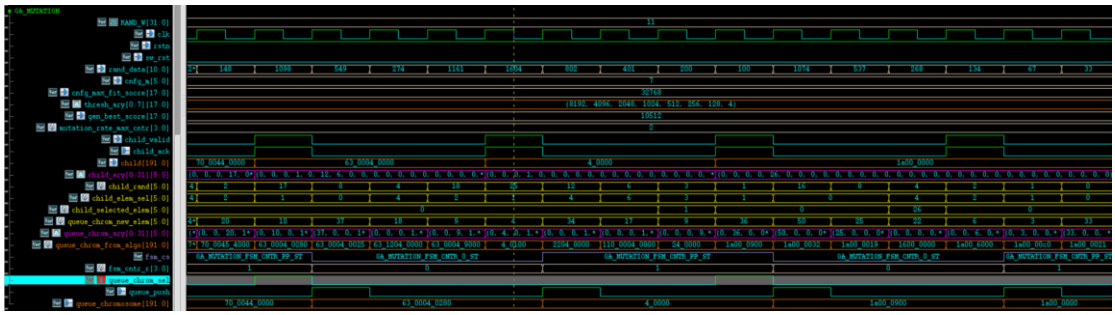


Figure 78 *: GA_MUTATION waveform – zoom-in: early {V_vec,d}*

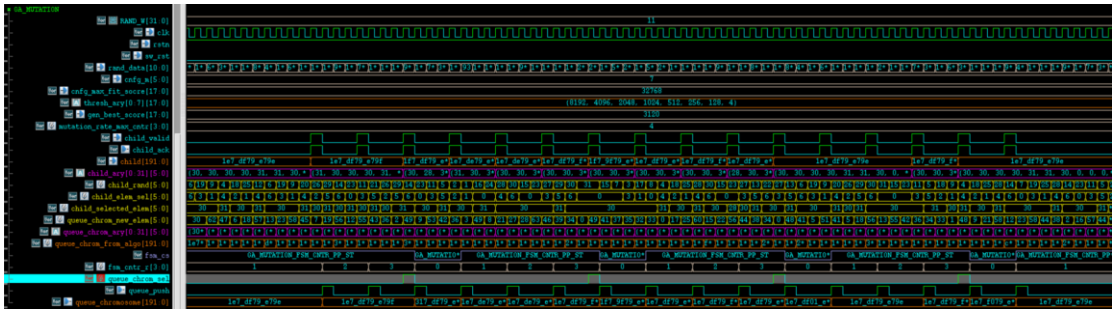


Figure 79 *: GA_ MUTATION waveform – zoom-in: intermediate {V_vec,d}*

In figure 78 mutation operations from a relative early stage are presented. The generation's best score is higher than each one of the thresholds, thus the mutation rate is maximal, and the counter gets the value of 2, as expected. From the 4 parents presented in the snapshoot (figure 78) one can see that the chromosomes that are pushed to the queue are one from the mutation algo block and one is the child directly, alternatively. Hence, the rate is as expected. More information that can be seen is the selection of a random index for weight in the parent, and its transformation to new weight, according to XOR with a random wight.

In figure 79 a random generation from the middle of the simulation was chosen. Its best result is better and as a result the counter maximal value is higher: 4. One can see that from all the 16 chromosomes of the new generation (pushed chromosomes) only 4 comes from algo block and the rest are the child directly, as expected.

Timing is as expected as well and indeed in one clock latency since the rise of child_valid the final chromosome is pushed to queue.

2.8.2. GA Accelerator – Generic Components Tests

2.8.2.1 GEN_INNER_PRODUCT Waveform

Block level simulation with basic inputs was made.

Chosen parameters:

- **VEC_ELEMS_NUM** = 32
- **ONE_ELEM_INT_W** = 1
- **ONE_ELEM_FRACT_W** = 5
- **RES_INT_W** = 4
- **RES_FRACT_W** = 10

Test cases:

Test num	i_vec1	i_vec2	Expected result
1	$in\ decimal : \{2, 2, 10\}$ $in\ fixed\text{-}point : \{0.625, 0.625, 0.3125\}$	$in\ decimal : \{-3, -3, 21\}$ $in\ fixed\text{-}point : \{-0.09375, -0.09375, 0.65625\}$	$in\ decimal : 198$ $in\ fixed\text{-}point : \frac{99}{512}$
2	$in\ decimal : \{31, 31, 31, 31\}$ $in\ fixed\text{-}point : \{0.96875, 0.96875, 0.96875, 0.96875\}$	$in\ decimal : \{31, 31, 31, 31\}$ $in\ fixed\text{-}point : \{0.96875, 0.96875, 0.96875, 0.96875\}$	$in\ decimal : 3844$ $in\ fixed\text{-}point : \frac{3844}{1024}$
3	$in\ decimal : \{-32, -32, -32, -32\}$ $in\ fixed\text{-}point : \{-1, -1, -1, -1\}$	$in\ decimal : \{-32, -32, -32, -32\}$ $in\ fixed\text{-}point : \{-1, -1, -1, -1\}$	$in\ decimal : 4096$ $in\ fixed\text{-}point : \frac{4096}{1024}$
4	$in\ decimal : \{-32, -32, -32, -32\}$ $in\ fixed\text{-}point : \{-1, -1, -1, -1\}$	$in\ decimal : \{31, 31, 31, 31\}$ $in\ fixed\text{-}point : \{0.96875, 0.96875, 0.96875, 0.96875\}$	$in\ decimal : -3968$ $in\ fixed\text{-}point : -3.875$
5	$in\ decimal : 10\ elems, each\ 31$ $in\ fixed\text{-}point : 10\ elems, each\ 0.96875$	$in\ decimal : 10\ elems, each\ 31$ $in\ fixed\text{-}point : 10\ elems, each\ 0.96875$	Overflow, so maximal positive value: $in\ decimal : 8191$ $in\ fixed\text{-}point : \frac{8191}{1024}$
6	$in\ decimal : 10\ elems, each\ -32$ $in\ fixed\text{-}point : 10\ elems, each\ -1$	$in\ decimal : 10\ elems, each\ 31$ $in\ fixed\text{-}point : 10\ elems, each\ 0.96875$	Overflow, so minimal positive value: $in\ decimal : -8192$ $in\ fixed\text{-}point : -\frac{8192}{1024}$

In figure 80 below the simulation is presented:

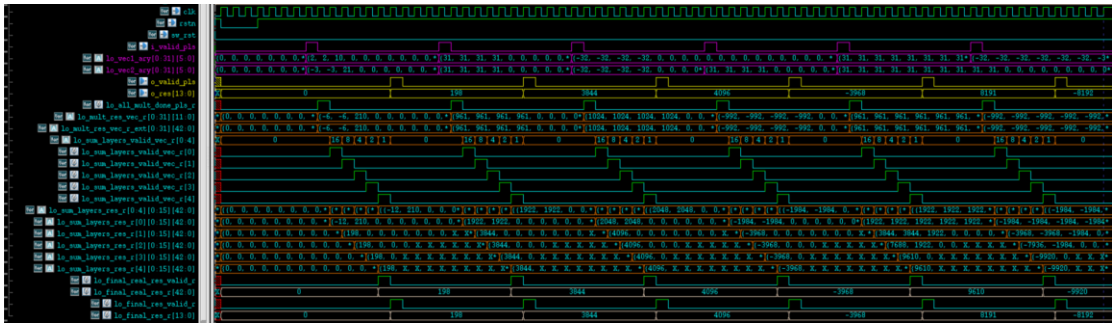


***Figure 80** : GEN_INNER_PRODUCT waveform*

One can see the results for each input are as expected, and the propagation of data inside the block.

In addition, timing can be checked and indeed there are 7 clock cycles from i_valid to o_valid.

2.8.2.2 GEN_ BST_SORTER Waveform

Block level simulation for one example was made.

Chosen parameters:

- **KEY_W** = 8
- **VALUE_W** = 16
- **MAX_ELEM_NUM** = 32

In figure 81 below the simulation is presented:

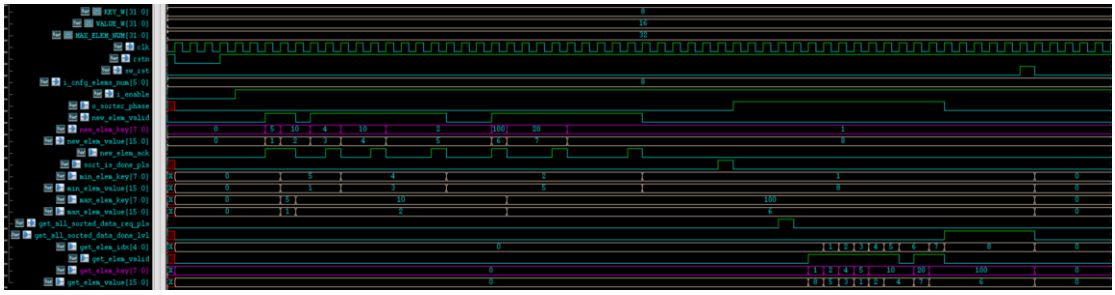


***Figure 81** : GEN_BST_SORTER waveform*

As mentioned only one example was tested, i.e. one rise of enable. The configurable number of elements for the sorter is 8 and they are entered in a random order to the sorter (ack response is different for each element according to its place in the tree). Once the 8th element is inserted to the tree sort_is_done_pls rises and sorter phase goes to 1, i.e. extraction phase.

The tree that was created:

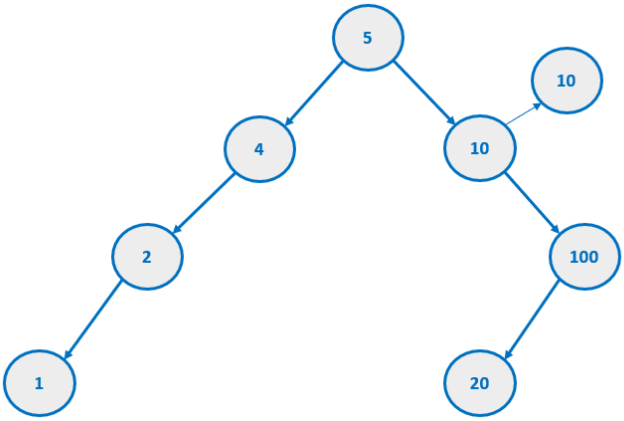


***Figure 82** : GEN_INNER_PRODUCT – tree created in simulation*

Once the sorter gets "get_all_sorted_data_req_pls" the sorter starts to output the elements one by one in order. As shown in the figure the elements are outputted from low to high as expected, with increasing index order, and at the end, after all of the 8 elements are outputted, "get_all_sorter_data_done_lvl" rises and the sorter operation is completed.

Overall, the sorter passes zero-order verification.

2.8.2.3 GEN_ PSEUDO_MODULUS Waveform

The combinatorial block gen_pseudo_modulus was tested with a few basic examples, as shown in figure 83:

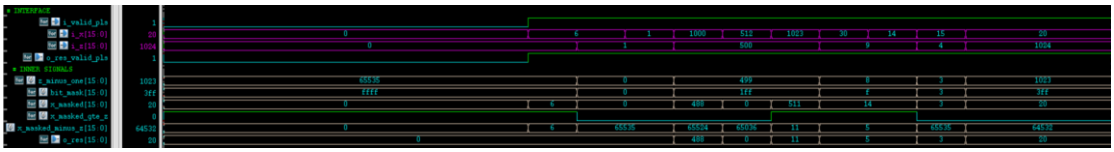


***Figure 83** : GEN_PSEUDO_MODULUS waveform*

The signals are presented in decimal, except bit_mask that is in hexadecimal. The parameter DATA_W was chosen to be 16.

The block calculates i_x "%" i_z, whereas "%" is pseudo modulus, and one can see that for each i_z the result is smaller than i_z, expect for i_z=0 in which case the result is zero (but in fact is not defined). In addition, one can see that all the outputs are as expected according to the explanations is section 2.5.6.

2.9. Programmer's Guide

GA accelerator registers file contains 6 registers. They are described in the following table:

Register name	Type (RO/RW)	bits	Address (byte address)	description
ga_enable	RW	1	0x0	Enable for ga_accelerator. All cnfg_* registers should be set before the rise of ga_enable and stay steady until ga_enable falls. Every fall of ga_enable restart the system (sw_rst), and every rise of ga_enable triggers random formation of initial chromosome generation.
cnfg_M	RW	M_MAX_W= clog2(M_max+1)= clog2(32+1)=6	0x4	M is the number of weights in the filter (number of elements in V_vec/W_vec). **Range: [2,32]** (M_max=32)
cnfg_P	RW	P_MAX_W= clog2(P_max+1)= clog2(1024+1)=11	0x8	P is the number of chromosomes in each generation. **Range: [2,1024]** (P_max=1024)
cnfg_B	RW	B_MAX_W= clog2(B_max+1)= clog2(64+1)=7	0xC	B is the size of the smoothing window, i.e. the number of timestamps in it. **Range: [2,64]** (B_max=64)
cnfg_G	RW	G_MAX_W= clog2(G_max+1)= clog2(1024+1)=11	0x10	G is the number of generations to estimate single W_vec[n+1]. **Range: [2,1024]** (G_max=1024)
inputs_counter	RO	32	0x14	The inputs counter counts how many valid sets of {V[n],d[n]} has been received since ga_enable rose. (Valid sets means sets in which input_valid_pls&ga_ready is TRUE)

***Table 13**: GA accelerator registers*

The minimal and maximal values of the configuration registers (cnfg_M, cnfg_P, cnfg_B, cnfg_G) were set according to algorithmic reasonable needs and according to hardware reasonable limitations.

3. Summary

3.1. Project's summary

In this project hardware accelerator of a genetic algorithm was implemented, based on the article "Hardware Implementation of a Real-time Genetic Algorithm for Adaptive Filtering Applications". The approach of using genetic algorithm for filters prediction is due to the fact that it tends to reach global maximum unlike the popular least squares method who tends to find local minimum.

The hardware accelerator was designed to provide full solution for the prediction of filters fixed-point weights with simple data interface and high flexibility to software, which is achieved by using configurations registers in order to set the algorithm's main parameters.

The article presents the main algorithmic idea for each step: initial population, fitness, selection, crossover and mutation. The full hardware design that was implemented in this project added on top of it communications signals between the units, internal FSMs for most of the units in order to control their action, and completed a lot of details that did not appear in the article. In addition, as mentioned, registers were added in order to allow software flexibility and main FSM was designed in order to control the entire process and to manage system inputs and outputs.

The hardware accelerator is parametric for flexible hardware needs, and in order to minimize area memories were used in places with large amounts of data. In addition to area, performance was an important objective and parallelism between and inside units was extensively used, with a trade-off between timing, area and in few places simplicity.

There were two main original contributions:

1. Area efficient combinatorial block to calculate pseudo-modulus between two numbers, which its main goal is to create a number between 0 and a given reference signal which is given as an additional number, and preserving the original number distribution.
2. Original sorter that stores the data in binary search tree data structure. This structure allows saving the data in a memory and hence saving area, and also enables high performance unit with an average logarithmic complexity.

Synthesis with Tower 180um technology and 125MHz clock frequency was made and passed successfully, with all the timing requirements met. The main limitation to clock frequency are the memories which requires at least 8ns clock cycle.

Zero-order verification was also made and passed successfully. Few simple test cases were checked and in all of them the accelerator was able to predict w_vec and y with high accuracy, except in cases that provide high degrees of freedom to w_vec in which case y was correct or very close but w_vec not (as expected in this case).

3.2. Take message home

Overall, the project was consists of microarchitecture design, implementation in SystemVerilog and simple zero-order verification, and eventually synthesis and floorplan. The hardware design process included a lot of tradeoffs between area, timing and power, and a lot of effort was made in order to make the design parametric so it can be scalability.

Genetic algorithm is an efficient way to predict weights of different filters. The usage in dedicated hardware to do so allows much faster results and opens an ability for real time application.

3.3. Next steps

A significant improvement in performance can be achieved by creating a pipe for chromosomes inside GA_FITNESS, so that B chromosomes could be handled in parallel.

A possible improvement in results and weights estimation might be achieved by adding penalty for long filters, this way with multiple degrees of freedom a shorter and more likely filter will be predicated.

In addition, additional verification needs to be made, with automatic checkers that are not used currently, different configurations need to be checked, random tests to cover more cases should be made as well, and also the check of additional edge cases, coverage, throughput and stress tests should be performed in the next steps.

Finally, because usually CPU frequency is relatively slow, performance might be improved by using two clocks instead of one: one for the core and the other for the CPU interface.

4. References

[1] Douglas, S. C., "Introduction to adaptive filters," in *Digital Signal Processing Handbook*, 1999.

[2] V. Mallawaarachchi, "Introduction to Genetic Algorithms — Including Example Code," 2017. [Online]. Available: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3.

[3] J. D. Seo, "Trend, Seasonality, Moving Average, Auto Regressive Model : My Journey to Time Series Data with Interactive Code," 2018. [Online]. Available: https://towardsdatascience.com/trend-seasonality-moving-average-auto-regressive-model-my-journey-to-time-series-data-with-edc4c0c8284b.

[4] *AMBA 3 APB Protocol Specification (v1.0),* ARM Inc., 2004.

[5] H. Merabti and m. Daniel , "Hardware Implementation of a Real-time Genetic Algorithm for Adaptive Filtering Applications," *IEEE 27th Canadian Conference on Electrical and Computer Engineering,* pp. 1-5, 2014.

5. Appendix

5.1. Appendix A – tanh LUT

14 bits input and 14 bits output tanh LUT:

tanh LUT.xlsx