technische universität
dortmund

Masterarbeit

# Evolutionäre Optimierung pseudoboolescher Funktionen auf FPGAs

Sascha Mücke
März 2019

Gutachter:
Prof. Dr. Katharina Morik
Dr. Nico Piatkowski

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Künstliche Intelligenz (LS8)
http://www-ai.cs.tu-dortmund.de

# Inhaltsverzeichnis

# Abkürzungen

**EA** Evolutionärer Algorithmus.

**FPGA** Field Programmable Gate Array.

**LFSR** Linear-Feedback Shift Register.

**LUT** Lookup Table.

**MRF** Markov-Random-Field.

**PBF** Pseudoboolesche Funktion.

**QUBO** Quadratic Unconstrained Binary Optimization.

**RAM** Random-Access Memory.

**UART** Universal Asynchronous Receiver Transmitter.

**VHDL** Very High Speed Integrated Circuit Hardware Description Language.

# Kapitel 1

# Einleitung

Das Konzept des Quantencomputers, dessen *Qubits* sich mithilfe des quantenmechanischen Effekts der Superposition in mehreren Zuständen zugleich befinden können, hat in den letzten Jahrzehnten die effiziente Lösung NP-schwieriger Probleme in greifbare Nähe gerückt und das weite Feld der Forschung an Quanten-Algorithmen eröffnet [31, 27]. Ein Optimierungsproblem, das sich besonders für die Lösung mithilfe von Quanten-Algorithmen eignet, ist die Quadratic Unconstrained Binary Optimization (QUBO), die auf einer Menge von binärwertigen Variablen eine polynomielle Zielfunktion zweiten Grades zu minimieren sucht; die Zielfunktion wird dabei so interpretiert, dass das globale Optimum dem angestrebten Quantenzustand minimaler Energie entspricht. Ein eng verwandtes Optimierungsproblem ist das Ising-Modell (auch Ising-Spin-Glas), dessen Variablen die Werte $+1$ und $-1$ annehmen können. Zahlreiche Probleme aus verschiedenen Bereichen des Maschinellen Lernens wie der Klassifikation [19, 9, 21] und der Bilderkennung [20] lassen sich mithilfe von QUBO formulieren und folglich mit einem entsprechenden Quanten-Optimierer effizient lösen. Da die Optimierung des Ising-Modells weiterhin NP-vollständig ist, lassen sich sämtliche NP-schweren Probleme darauf reduzieren [17].

Während es an Anwendungen für Quanten-Algorithmen also nicht mangelt, steckt die physische Umsetzung von Quantenrechnern noch immer weitgehend in den Kinderschuhen. Vorreiter in der Kommerzialisierung von Quantenrechnern ist das kanadische Unternehmen D-Wave, das mit seinem System *D-Wave 2000Q* den ersten frei käuflichen Quantencomputer der Welt produziert. Der enthaltene QPU-Kern besitzt rund zweitausend Qubits auf einem Chip von der Größe eines Daumennagels und ist von einem drei Meter langen, zwei Meter breiten und drei Meter hohen Gehäuse umgeben, das u. a. ein Kühlsystem beherbergt, welches den Kern auf die notwendige Betriebstemperatur von 15 Millikelvin über dem absoluten Nullpunkt bringt, und das außerdem das Erdmagnetfeld abschirmt und so einen feldfreien Raum mit einer magnetischen Flussdichte von unter einem Nano-

tesla erzeugt[1]. Dass diese Technologie mit einem Preis von 15 Millionen US-Dollar nicht erschwinglich ist, überrascht daher nicht. Überraschend hingegen sind Resultate vergleichender Benchmarks von herkömmlichen Computern und Quanten-Computern aus dem Hause D-Wave, die keinen Hinweis auf eine erhebliche Geschwindigkeitssteigerung durch die Verwendung von Qubits fanden [24].

Obwohl QUBO und Ising-Modelle besonders gut für Quantenrechner geeinet sind, lassen sie sich mit anderen, herkömmlichen Verfahren optimieren. Eine vielversprechende Klasse von Optimierverfahren sind die *Evolutionären Algorithmen*, die sich die natürliche Selektion zum Vorbild nehmen und auf Populationen von Lösungen Mutation und Rekombination anwenden, um zu immer besseren Lösungen zu gelangen. Eine Methode, um solche Verfahren möglichst effizient und schnell auszuführen besteht in einer hardwarenahen Implementierung anstelle einer üblichen Implementierung in einer Hochsprache. Besonders Field Programmable Gate Arrays (FPGAs), die zum einfachen und schnellen Prototyping von Mikrochips entwickelt wurden, sind das ideale Medium, um die positiven Effekte von Mikroparallelisierung auf die Geschwindigkeit von Algorithmen auszunutzen. Durch ihre hohe Taktrate und die fehlenden Abstraktionsebenen, die die meisten Hochsprachen auszeichnen und zu einer hohen Zahl von CPU-Operationen führen, erreichen sie Geschwindigkeiten, die herkömmliche Software-Implementierungen häufig um das Tausendfache übertreffen. Im Vergleich zum D-Wave 2000Q sind sie zudem mit einer Preisspanne von unter 100 bis einigen Tausend Euro vergleichsweise erschwinglich.

In dieser Arbeit soll das Problem der QUBO-Optimierung mithilfe eines evolutionären Algorithmus' auf einem FPGA gelöst werden. Die resultierende Implementierung soll auf das Zusammenspiel der problemspezifischen und algorithmenspezifischen Parameter und ihre Auswirkung auf den Verbrauch von FPGA-Elementen (Lookup Tables (LUTs), Random-Access Memory (RAM) usw.) und die Geschwindigkeit hin untersucht werden. Anschließend soll die Leistungsfähigkeit anhand einiger Experimente mit Anwendungen aus dem Bereich des Maschinellen Lernens demonstriert werden.

Ähnliche Arbeiten über die Implementierung Evolutionärer Algorithmen auf FPGAs verwenden entweder feste Systemparameter, die sich nicht auf ein gegebenes Optimierungsproblem anpassen lassen [29, 32, 30] oder erlauben häufig nur eine kleine Zahl von Variablen [12, 33], üblicherweise einige wenige 32-Bit-Zahlen. Die im Rahmen dieser Arbeit entwickelte Implementierung ist sowohl hinsichtlich der Populationsgröße, der Anzahl Variablen sowie der Größe der Zielfunktionsparameter voll anpassbar. Weiterhin ist sie ohne Neuprogrammierung des FPGAs dazu imstande, den initialen Seed für die Zufallszahlengenerierung zu ändern und zwischen der Optimierung eines QUBO-Problems und eines Ising-Modells zu wechseln, und arbeitet mit mit einer hohen Taktfrequenz von 100 MHz.

[1] `https://www.dwavesys.com/sites/default/files/D-Wave%202000Q%20Tech%20Collateral_1029F.pdf`

# Kapitel 2

# Grundlagen

## 2.1 Evolutionäre Optimierung

Die Idee der Optimierung mithilfe von durch die Natur inspirierten Mechanismen der darwinschen Evolutionstheorie reicht bis in die 1950er-Jahre zurück. Die Forschung auf diesem Gebiet führte zur Entwicklung mehrerer "Schulen", die dem natürlichen Vorbild mehr oder weniger detailliert folgen, vor allem die der genetischen Algorithmen [7] und der Evolutionsstrategien [13]. Allen Ansätzen gemein ist das Vorgehen, eine Zielfunktion $L : \mathcal{X} \to \mathbb{R}$ (je nach Interpretation auch *Fitness-* oder *Kosten-* bzw. *Verlustfunktion*) auf einem Lösungsraum $\mathcal{X}$ zu definieren, die die Güte dieser Lösung numerisch bewertet, um anschließend mit dieser Zielfunktion auf einer *Population* von Lösungskandidaten $\boldsymbol{x} \in \mathcal{X}$ (*Individuen*) eine Erzeugung von Nachkommen durch *Rekombination* zweier oder mehrerer Elternteile und anschließender zufälliger *Mutation* zu simulieren. Da auch in der Natur jeder Lebensraum mit all seinen Resourcen begrenzt ist, wird die Population durch *Selektion* auf Basis der Zielfunktion wieder verkleinert, beispielsweise indem die Individuen mit den (im Falle einer Minimierung) höchsten Werten – also solche, die am schlechtesten angepasst sind – "sterben" und aus der Population entfernt werden. Wiederholt sich dieser Vorgang häufig, passt sich die Population immer besser der Zielfunktion an und bewegt sich – so die Hoffnung – in Richtung eines globalen Optimums. Die genaue Beschaffenheit dieses virtuellen Lebensraums und seiner "Bewohner" sowie die Art und Weise, wie Rekombination, Mutation und Selektion ablaufen, lässt sich auf vielfältige Weise in Algorithmen übersetzen, von denen die wichtigsten und für diese Arbeit relevantesten hier vorgestellt werden sollen.

### 2.1.1 Komma- und Plus-EA

Sei $P^{(t)} = \left(\boldsymbol{x}_1^{(t)}, \ldots, \boldsymbol{x}_\mu^{(t)}\right)$ eine *Population* der Größe $\mu \in \mathbb{N}$ von Lösungskandidaten oder *Individuen* $\boldsymbol{x}_i \in \mathcal{X}$ zu einem Zeitpunkt $t \in \mathbb{N}_0$, wobei Individuen auch mehrfach in einer Population vorkommen dürfen. $P^{(t)}$ ist somit eigentlich eine Multimenge oder Liste, wird im Folgenden aber vereinfachend als Vektor aus $\mathcal{X}^\mu$ interpretiert. Die Population $P^{(0)}$

ist die *Startpopulation*. Weiterhin sei rec : $\mathcal{X}^\mu \mapsto \mathcal{X}$ der *Rekombinationsoperator* und mut : $\mathcal{X} \mapsto \mathcal{X}$ der Mutationsoperator. Der Rekombinationsoperator kann weiter unterteilt werden in eine zufällige Auswahl von $\rho$ Eltern aus der aktuellen Population (mit $1 \leq \rho \leq \mu$) und der Kombination verschiedener Teile der Elternindividuen zu einem neuen Individuum,

$$\text{rec} = \text{re} \circ \text{co},$$

mit co : $\mathcal{X}^\mu \mapsto \mathcal{X}^\rho$ und re : $\mathcal{X}^\rho \mapsto \mathcal{X}$. Der *Mutationsoperator* mut : $\mathcal{X} \mapsto \mathcal{X}$ erzeugt ein neues Individuum, indem er ein gegebenes Individuum zufällig mit einer Mutationswahrscheinlichkeit $p_m$ variiert. Aus einer Elternpopulation $P^{(t)}$ wird mithilfe dieser Operatoren eine Nachkommenpopulation $\tilde{P}^{(t)}$ bestehend aus $\lambda$ Nachkommen erzeugt, wobei $\mu \leq \lambda$:

$$\tilde{P}^{(t)} = \left(\text{mut}(\text{rec}(P^{(t)}))_i\right)_{i \in \{1,\ldots,\lambda\}} = \left(\tilde{\boldsymbol{x}}_1^{(t)}, \ldots, \tilde{\boldsymbol{x}}_\lambda^{(t)}\right)$$

Aus der Nachkommenpopulation wird mithilfe des *Selektionsoperators* sel : $\mathcal{X}^\lambda \mapsto \mathcal{X}^\mu$ eine neue Elterngeneration ausgewählt. Die Selektion erfolgt über die Zielfunktion, sodass die $\mu$ Individuen mit den besten (sprich *kleinsten* im Falle einer Minimierung) Zielfunktionswerten in die neue Elternpopulation übernommen werden:

$$\text{sel}\left(\tilde{P}^{(t)}\right) = \left(\tilde{\boldsymbol{x}}_i^{(t)} \in \tilde{P}^{(t)}.\ |\{\tilde{\boldsymbol{x}}_j^{(t)} \in \tilde{P}^{(t)}.\ L(\tilde{\boldsymbol{x}}_j^{(t)}) \leq L(\tilde{\boldsymbol{x}}_i^{(t)})\}| \leq \mu\right)$$

Liegen die Individuen der Nachkommenpopulation o. B. d. A. aufsteigend nach Zielfunktionswert sortiert vor, vereinfacht sich die Selektion zu

$$\text{sel}\left(\tilde{P}^{(t)}\right) = \left(\tilde{\boldsymbol{x}}_1^{(t)}, \ldots, \tilde{\boldsymbol{x}}_\mu^{(t)}\right)$$

Diese Art der Selektions wird auch als *Turnierselektion* bezeichnet, da alle Individuen der Nachkommenpopulation "gegeneinander kämpfen" und die stärksten, also solche mit den besten Zielfunktionswerten, überleben. Eine Iteration der Evolutionsstrategie lässt sich somit zusammenfassen als

$$P^{(t+1)} = \text{sel}\left(\tilde{P}^{(t)}\right) = \text{sel}\left(\left(\text{mut}(\text{rec}(P^{(t)}))_i\right)_{i \in \{1,\ldots,\lambda\}}\right)$$

Diese Evolutionsstrategie wird als $(\mu, \lambda)$-EA bezeichnet. Ein Unterscheidungsmerkmal verschiedener EA-Verfahren ist das Konzept des *Elitismus*: Demnach ist ein EA inhärent elitistisch, wenn die Individuen mit den besten Zielfunktionswerten stets überleben, sich das beste Individuum einer Population also nie verschlechtern kann. Der $(\mu, \lambda)$-EA ist nicht inhärent elitistisch, da durch die zufällige Rekombination und Mutation die Situation entstehen kann, dass für alle $\tilde{\boldsymbol{x}}_i^{(t)} \in \tilde{P}^{(t)}$ gilt, dass $L(\tilde{\boldsymbol{x}}_i^{(t)}) > L(\boldsymbol{x}^{(t)*})$, wobei $\boldsymbol{x}^{(t)*}$ das Individuum aus $P^{(t)}$ mit bestem Zielfunktionswert zum Zeitpunkt $t$ ist. Da die Elternindividuen für die folgende Generation ausschließlich aus den Nachkommen ausgewählt werden, verschlechtert sich somit das Optimum zwangsläufig im nächsten Schritt, und das bessere Individuum geht verloren.

Um Elitismus zu gewährleisten und somit die beste jemals erreichte Lösung zu bewahren, gibt es verschiedene Ansätze. Der wohl einfachste besteht darin, das beste jemals beobachtete Individuum separat von der Population zu speichern, beispielsweise während der Zielfunktionsauswertung (*forced elitism*, "erzwungener Elitismus"). Ein weiterer Ansatz ist eine Abwandlung des Selektionsschritt hin zu einem Selektionsoperator $\mathrm{sel}_+ : \mathcal{X}^{\mu+\lambda} \mapsto \mathcal{X}^{\mu}$, der die besten Individuen sowohl aus den Nachkommen als auch aus den Eltern selbst auswählt:

$$P^{(t+1)} = \mathrm{sel}_+ \left( \tilde{P}^{(t)} \sqcup P^{(t)} \right)$$

mit $\sqcup$ als Vereinigung auf Multimengen. Diese Evolutionsstrategie ist der $(\mu + \lambda)$-EA. Da bei dieser Variante das vorherige Optimum mit in die Selektion eingeschlossen wird, kann sich das Optimum nie verschlechtern, da es selbst bei einer Verschlechterung aller Nachkommen wieder mit in die nächste Elterngeneration einfließen würde.

Neben diesen grundlegenden EA-Varianten gibt es weitere Verfeinerungen, die über zahlreiche weitere Strategieparameter die verschiedenen Schritte des evolutionären Verfahrens zu kontrollieren suchen, beispielsweise über eine begrenzte Lebensdauer von $\kappa$ Iterationen pro Individuum, eine feste Turniergröße $\zeta$ [28] oder die schrittweise Anpassung der Schrittweite bei der Mutation [14].

Während die Konvergenz zu einem globalen Optimum für elitistische Verfahren wie den $(\mu + \lambda)$-EA, bei denen sich das Optimum immer schrittweise monoton verbessert und durch Mutation jedes $\boldsymbol{x} \in \mathcal{X}$ mit einer positiven Wahrscheinlichkeit erreicht werden kann, intuitiv und einfach zu beweisen ist (siehe z. B. [1]), lässt sich auch für nicht-elitistische Verfahren wie den $(\mu, \lambda)$-EA unter bestimmten Vorraussetzungen zeigen, dass ein globales Optimum garantiert erreicht wird [25].

Wie [10] zeigt, liegt die Konvergenzrate des $(1 + 1)$-EAs für lineare Zielfunktionen bei $\mathcal{O}(n \log n)$, wenn als Mutationsrate $\Theta(1/n)$ gewählt wird. Polynome höheren Grades und andere Zielfunktionen konvergieren hingegen in der Regel mit exponentieller Zeit, die genaue erwartete Laufzeit ist allerdings stark problemabhängig. Die Konvergenzraten des $(1+1)$-EA können als untere Schranken für die Konvergenzrate des $(\mu+\lambda)$-EAs gelten, da durch die (parallele) Auswertung mehrerer Individuen statt nur eines einzigen die Durchsuchung des Lösungsraums nur beschleunigt werden kann.

### 2.1.2 Rekombination und Mutation

Die genaue Beschaffenheit der Rekombinations- und Mutationsoperatoren ist stark abhängig vom jeweiligen Optimierungsproblem. Für $\mathcal{X} = \mathbb{R}^n$ kann die Rekombination von $\rho$ Elternindividuen beispielsweise durch eine (möglicherweise zufällig) gewichtete Interpolation erfolgen:

$$\mathrm{rec}_{\mathbb{R}^n}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_\rho) = \sum_{i=1}^{\rho} w_i \boldsymbol{x}_i \quad \text{mit} \quad \sum_{i=1}^{\rho} w_i = 1$$

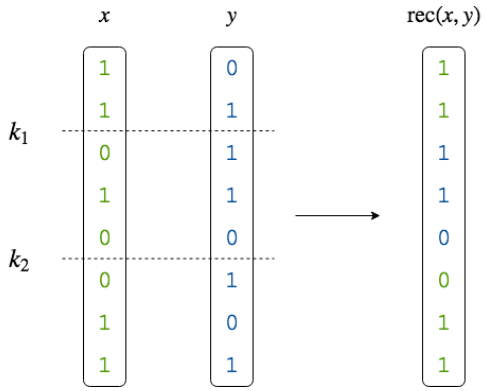



**Abbildung 2.1:** Beispiel für *2-point crossover* mit $n = 8$ und $\rho = 2$

Die Mutation für den gleichen Lösungsraum kann über eine normalverteilte zufällige Verschiebung der einzelnen Vektorkomponenten umgesetzt werden:

$$\text{mut}_{\mathbb{R}^n}(\boldsymbol{x}) = (x_1 + z_1, \ldots, x_n + z_n)^\top \quad \text{mit } z_i \sim N(0, \sigma^2) \text{ für alle } i \in \{1, \ldots, n\}$$

Auf diskreten vektorartigen Lösungsräumen gibt es Ansätze für den Rekombinationsoperator, die eher den entsprechenden Vorgängen in der Natur ähneln, z. B. bei der Kombination von DNS-Sequenzen: Ist $\mathcal{X} = \mathbb{B}^n = \{0, 1\}^n$, sind die Individuen also Bitvektoren der Länge $n$, so kann eine Rekombination zweier Bitvektoren beispielsweise durchgeführt werden, indem eine zufällige Schnittstelle $k \sim \mathcal{U}\{1, n-1\}$ gewählt und der Nachkomme aus den ersten $k$ Stellen des ersten und den letzten $n-k$ Stellen des zweiten Elternvektors zusammengesetzt wird (*1-point crossover*), was auf eine bliebige Anzahl $K$ von Schnittstellen erweitert werden kann, sodass jedes Segment des Nachkommens abwechselnd (oder bei mehr als zwei Eltern reihum oder zufällig) den teilnehmenden Elternvektoren entnommen werden (*K-point crossover*, siehe Abb. 2.1). Eine weitere Möglichkeit besteht darin, jede einzelne Stelle des Vektors zufällig gleichverteilt einem der Elternvektoren zu entnehmen, was einem randomisierten *K-point crossover* mit $K = n-1$ entspricht (*uniform crossover*).

Die Mutation von Bitvektoren funktioniert analog zu $\text{mut}_{\mathbb{R}^n}$, nur dass es lediglich zwei mögliche Zustände pro Dimension gibt, weshalb statt einer Normalverteilung eine Bernoulli-Verteilung ausreicht: Hat eine Bernoulli-verteilte Variable $z_i$ den Wert 1, wird das Bit an Stelle $i$ des Vektors invertiert, andernfalls unverändert übernommen:

$$\text{mut}_{\mathbb{B}^n}(\boldsymbol{x}) = (x_1 \oplus z_1, \ldots, x_n \oplus z_n)^\top \quad \text{mit } z_i \sim \mathcal{B}(p) \text{ für alle } i \in \{1, \ldots, n\} \tag{2.1}$$

Die Operation $\oplus : \mathbb{B} \times \mathbb{B} \mapsto \mathbb{B}$ bezeichnet dabei den XOR-Operator und $\mathcal{B}(p)$ die Bernoulli-Verteilung mit Erfolgswahrscheinlichkeit $p$. Der Erwartungswert der Variable $z_i$ ist $\mathbb{E}[z_i] = p$, und somit auch $\mathbb{E}[x_i \text{ wird invertiert}] = p$. Die Wahrscheinlichkeit einer Mutation von einem Vektor $\boldsymbol{x}$ zu einem beliebigen anderen Vektor $\tilde{\boldsymbol{x}} \in \mathbb{B}^n$ ist daher gegeben durch

$$q(\boldsymbol{x}, \tilde{\boldsymbol{x}}) = p^{H(\boldsymbol{x}, \tilde{\boldsymbol{x}})}(1-p)^{H(\boldsymbol{x}, \tilde{\boldsymbol{x}})},$$

wobei $H(\boldsymbol{x}, \tilde{\boldsymbol{x}})$ die Hamming-Distanz der beiden Vektoren bezeichnet. Da $0 \leq H(\boldsymbol{x}, \tilde{\boldsymbol{x}}) \leq n$ und $p > 0$, ist $q(\boldsymbol{x}, \tilde{\boldsymbol{x}})$ offenbar immer größer als 0, sodass der Support der Übergangswahrscheinlichkeitsfunktion für einen festen Ausgangsvektor $\boldsymbol{x}$ gleich der gesamten Menge $\mathbb{B}^n$ ist. Diese wichtige Eigenschaft sorgt dafür, dass jedes Individuum in einer einzigen Mutation das globale Optimum erreichen kann, was natürlich für großes $n$ zunehmend unwahrscheinlich wird.

## 2.2 QUBO

### 2.2.1 Pseudoboolesche Funktionen

Funktionen der Form $f : \mathbb{B}^n \mapsto \mathbb{R}$ heißen *pseudoboolesche Funktionen* (kurz PBF). Da ihre Urbildmenge mit $|\mathbb{B}^n| = 2^n$ endlich ist, kann auch die Bildmenge nur aus höchstens $2^n$ verschiedenen Werten bestehen. Somit lässt sich jede solche Funktion theoretisch als Liste von Paaren $(\boldsymbol{x}, f(\boldsymbol{x}))$ für alle $\boldsymbol{x} \in \mathbb{B}^n$ darstellen. Da diese Darstellung für große $n$ äußerst ineffizient ist, bietet sich die Repräsentation in Form von Polynomen an:

$$f(\boldsymbol{x}) = \alpha_0 + \sum_{i=1}^{n} \alpha_i x_i + \sum_{i=1}^{n} \sum_{j=1}^{i} \beta_{ij} x_i x_j + \sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=1}^{j} \gamma_{ijk} x_i x_j x_k + \dots$$

Dabei sind $\alpha_i, \beta_{ij}, \dots \in \mathbb{R}$. Eine wichtige Beobachtung ist, dass die Multiplikation von Binärwerten einer Verundung entspricht, das Produkt also nur gleich 1 ist, wenn *alle* $x_i$ gleich 1 sind. Die Definition des Polynoms ist demnach äquivalent zu einer Zuordnung von Koeffizienten zu allen möglichen Untermengen von Bits, die in $\boldsymbol{x}$ enthalten sind. Die Koeffizienten, deren zugehörige Untermenge von Bits verundet gleich 1 ist, werden aufsummiert. Dadurch ergibt sich folgende alternative, kompaktere Darstellung des Polynoms:

$$f(\boldsymbol{x}) = \sum_{I \subseteq \{1,\dots,n\}} \beta_I \prod_{i \in I} x_i$$

Die Größe der größten Untermenge, deren zugehöriger Koeffizient nicht 0 ist, bezeichnet man als *Grad* $k$ des Polynoms:

$$k = \max\{|I| \mid I \subseteq \{1, \dots, n\}, \beta_I \neq 0\}$$

Zunächst scheint es, dass die Darstellung pseudoboolescher Funktionen durch Polynome eine Einschränkung darstellt; so sind Funktionen $f : \mathbb{R} \mapsto \mathbb{R}$ beispielsweise im Allgemeinen *nicht* allein durch Polynome (endlichen Grades) darstellbar, wie z. B. $f = \exp$, dessen Reihendarstellung ein Polynom unendlichen Grades ist, oder $f = \ln$, dessen Tailor-Entwicklung zusätzlich einen begrenzten Konvergenzradius besitzt. Es lässt sich allerdings zeigen, dass jede beliebige pseudoboolesche Funktion durch ein Polynom dargestellt werden kann:

**2.2.1 Satz.** *Jede pseudoboolesche Funktion $f : \mathbb{B}^n \mapsto \mathbb{R}$ kann als Polynom mit eindeutigen Koeffizienten $\beta_I$ dargestellt werden.*

*Beweis.* Nach [5]: Es soll per Induktion über die Größe der Untermengen $I \subseteq \{1, \ldots, n\}$ gezeigt werden, dass die Koeffizienten $\beta_I$ eindeutig durch die entsprechenden Funktionswerte von $f$ bestimmt sind. Bezeichne $\mathbf{1}^I$ den Vektor, der an allen Indizes $i \in I$ gleich 1 und überall sonst gleich 0 ist. (IA) Offenbar gibt es nur eine Untermenge der Größe 0, nämlich $\emptyset$, dadurch ist $f(\mathbf{1}^\emptyset)$ eindeutig von $\beta_\emptyset$ bestimmt. (IV) Sei für alle Untermengen $J \subset I$ mit $|I| = k$ gezeigt, dass $\beta_J$ eindeutig sind.

$$f(\mathbf{1}^I) = \sum_{J' \subseteq I} \beta_{J'} = \sum_{J \subset I} \beta_J + \beta_I$$

Da alle $\beta_J$ per IV eindeutig sind, muss $\beta_I = f(\mathbf{1}^I) - \sum_{J \subset I} \beta_J$ sein, weshalb auch $\beta_I$ eindeutig bestimmt ist. □

### 2.2.2 Quadratische Polynome

Eine wichtige Unterklasse der pseudobooleschen Funktionen bilden die quadratischen Polynome, also solche mit Grad $k = 2$ der Form

$$f(\boldsymbol{x}) = \alpha_0 + \sum_{i=1}^{n} \alpha_i x_i + \sum_{i=1}^{n} \sum_{j=1}^{i} \beta_{ij} x_i x_j.$$

Sie eignen sich gut als Zielfunktionen für Optimierungsprobleme, da einerseits die Anzahl der Koeffizienten im Vergleich zu allgemeinen Polynomen nicht $|\mathbb{B}^n| = 2^n$ sondern nur $(n^2 + n)/2 + n + 1 \in \mathcal{O}(n^2)$ beträgt. Andererseits besitzen quadratische PBF die herausragende Eigenschaft, dass jede PBF $f$ höheren Grades auf eine PBF $\tilde{f}$ zweiten Grades reduziert werden kann, die das gleiche globale Minimum besitzt:

$$\min_{\boldsymbol{y} \in \mathbb{B}^m} \tilde{f}(\boldsymbol{y}) = \min_{\boldsymbol{x} \in \mathbb{B}^n} f(\boldsymbol{x})$$

Ein entsprechender Algorithmus zur Reduktion einer PBF in Polynomdarstellung wird in [5, Abschnitt 4.4] beschrieben.

Aufgrund dieser nützlichen Eigenschaften bilden quadratische PBF die Grundlage einer Klasse von Optimierungsproblemen, die als *Quadratic Unconstrained Binary Optimization* (kurz QUBO) bezeichnet wird und die Form

$$L(\boldsymbol{x}) = \sum_{i=1}^{N} \alpha_i x_i + \sum_{i=1}^{N} \sum_{j=1}^{i} \beta'_{ij} x_i x_j \;\rightarrow\; \min$$

besitzt. Im Gegensatz zur allgemeinen Definition der quadratischen Polynome entfällt hier $\alpha_0$, da es die Lage des Optimums nicht beeinflusst. Als weitere Vereinfachung zur Reduktion der Parameterzahl können die Parameter $\alpha_i$ und $\beta'_{ii}$ für alle $i \in \{1, \ldots, N\}$ zusammengefasst werden, da sie offenbar immer gleichzeitig in die Gesamtsumme einfließen. Dadurch

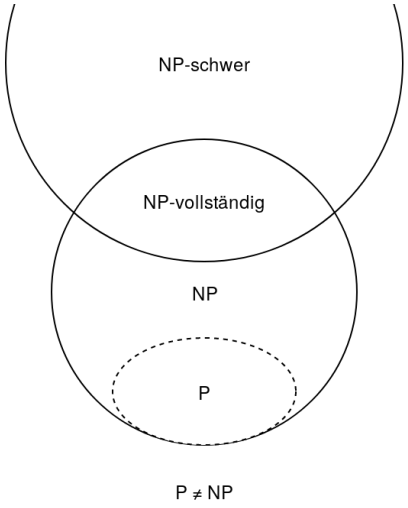



**Abbildung 2.2:** Komplexitätsklassen für den Fall $P \neq NP$

ergibt sich eine Definition der Zielfunktion mit genau $(N^2 + N)/2$ Koeffizienten, die im restlichen Verlauf dieser Arbeit verwendet wird:

$$L(\boldsymbol{x}) = \sum_{i=1}^{N} \sum_{j=1}^{i} \beta_{ij} x_i x_j \;\rightarrow\; \min \quad \text{mit } \beta_{ij} = \begin{cases} \alpha_i + \beta'_{ii} & \text{falls } i = j \\ \beta'_{ij} & \text{sonst} \end{cases} \tag{2.2}$$

QUBO gehört zu den NP-schweren Problemen, und viele bekannte NP-schwere Probleme wie das Teilsummenproblem [5] und 2-Means-Clustering [3] lassen sich entweder direkt darauf oder auf die Optimierung des eng verwandten Ising-Modells (siehe Abschnitt 2.2.3) reduzieren. Als Konsequenz ist ein Optimierer für QUBO vielseitig einsetzbar für andere schwierige Probleme, wodurch der Bedarf nach spezialisierten Algorithmen für solche Probleme potenziell gesenkt wird.

### 2.2.3 Ising-Modell

Das Ising-Modell ist ein besonderes graphisches Modell aus dem Bereich der theoretischen Physik, das aufgrund seiner Allgemeinheit jedoch Anwendungen zur Modellierung vielfältiger Optimierungsprobleme gefunden hat. In seiner ursprünglichen Form besteht es aus einer gitterförmigen Anordnung von Knoten, die mit ihren jeweiligen vier Nachbarn über eine Kante verbunden sind. Die von der Knotenmenge indizierten Zufallsvariablen können die Werte $+1$ und $-1$ annehmen, was als Ausrichtung eines Magneten in einem Feld oder als An- bzw. Abwesenheit von Teilchen interpretiert werden kann [34]. Im verallgemeinerten Ising-Modell kann jede Variable mit jeder anderen interagieren, unabhängig von einer Nachbarschaftsbeziehung.

Die Zielfunktion des Ising-Modells ist sehr ähnlich zu QUBO und unterscheidet sich ausschließlich in der Interpretation der Variablenausprägungen: Während QUBO Binärvektoren $\boldsymbol{x} \in \{0,1\}^N$ minimiert, ist die Lösungsmenge eines Ising-Modells $\{-1,+1\}^N$.

$$L(\boldsymbol{s}) = \sum_{i=1}^{N} \alpha_i s_i + \sum_{i=1}^{N} \sum_{j=1}^{i} \beta_{ij} s_i s_j \;\rightarrow\; \min$$

Im Gegensatz zur QUBO-Zielfunktion lässt sich diese Zielfunktion nicht vereinfachen, indem die linearen Terme in die quadratischen integriert werden, denn je nach Vorzeichen von $s_i$ geht der Parameter $\alpha_i$ positiv oder negativ in die Gesamtsumme ein, während die quadratischen Parameter $\beta_{ii}$ immer addiert werden, da $s_i \cdot s_i = +1$ für alle $s_i \in \{-1,+1\}$. Dies hat allerdings auch zur Folge, dass $\beta_{ii}$ unabhängig von der Belegung von $s_i$ und dadurch eine Konstante bezüglich der Minimierung ist:

$$\begin{aligned}\min_{\boldsymbol{s}} \sum_{i=1}^{N} \alpha_i s_i + \sum_{i=1}^{N} \sum_{j=1}^{i} \beta_{ij} s_i s_j &= \min_{\boldsymbol{s}} \sum_{i=1}^{N} \alpha_i s_i + \sum_{i=1}^{N} \sum_{j=1}^{i-1} \beta_{ij} s_i s_j + \underbrace{\sum_{i=1}^{N} \beta_{ii}}_{\text{const.}} \\ &= \min_{\boldsymbol{s}} \sum_{i=1}^{N} \alpha_i s_i + \sum_{i=1}^{N} \sum_{j=1}^{i-1} \beta_{ij} s_i s_j\end{aligned}$$

Wie in Abschnitt 4.1.5 gezeigt wird, lässt sich dies ausnutzen, um mit der Architektur eines QUBO-Optimierers einen Optimierer für das Ising-Modell zu implementieren.

### 2.2.4 Anwendungen im Maschinellen Lernen

Im Bereich des maschinellen Lernens liegt der große Wert eines QUBO-Optimierers beispielsweise in der effizienten Ermittlung der wahrscheinlichsten Variablenbelegung eines probabilistischen graphischen Modells: Die Dichtefunktion einer Exponentialfamilie mit Parametern $\boldsymbol{\theta}$ und suffizienter Statistik $\phi = (\phi_i)_{i \in I}$ ist gegeben durch

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \frac{1}{Z} \exp\left(\langle \boldsymbol{\theta}, \phi(\boldsymbol{x}) \rangle\right),$$

wobei $\langle \cdot \rangle$ das Skalarprodukt und $Z = \int_{\mathcal{X}} \exp\left(\langle \boldsymbol{\theta}, \phi(\boldsymbol{x}) \rangle\right) \mathrm{d}x$ die Normierungskonstante ist, die $p_{\boldsymbol{\theta}}$ zu einer Dichte macht [34]. Da sowohl Normalisierung als auch Anwendung der Exponentialfunktion monotone Funktionen sind, vereinfacht sich die Maximierung der Dichte zu

$$\max_{\boldsymbol{x} \in \mathcal{X}} p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \max_{\boldsymbol{x} \in \mathcal{X}} \langle \boldsymbol{\theta}, \phi(\boldsymbol{x}) \rangle = \max_{\boldsymbol{x} \in \mathcal{X}} \sum_{i \in I} \theta_i \phi_i(\boldsymbol{x}).$$

Die suffiziente Statistik $\phi$ ist für diskrete Verteilungen mit $k$ Ausprägungen eine 1-aus-$k$-Kodierung durch einen Bit-Vektor: Hat eine Zufallsvariable $X$ die Ausprägung $x_i$, so ist der zugehörige Vektor $\phi(X) = (0, \ldots, 0, 1, 0, \ldots, 0)^\top$ mit 1 an Index $i$. Die Variablenbelegung eines Markov-Random-Fields (MRF) kann daher als Binärvektor dargestellt werden, der aus allen auf diese Weise kodierten Variablenbelegungen aneinandergereiht besteht (siehe Abschnitt 5.3.2).

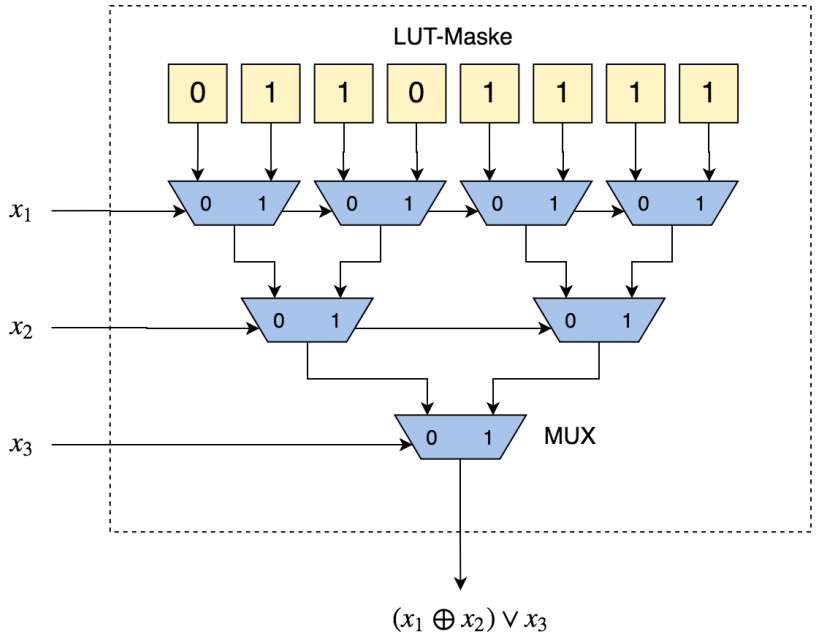



**Abbildung 2.3:** Aufbau eines LUTs mit drei Bit-Eingängen

## 2.3 FPGAs

Ein *Fiel Programmable Gate Array*, kurz FPGA, ist ein programmierbarer integrierter Schaltkreis, mit dessen Hilfe Schaltungen gebaut und getestet werden können. Ein FPGA besteht aus einem Raster von Funktionsblöcken, die aus je einer Funktionstabelle (LUT, siehe Abb. 2.3) und einem Flip-Flop bestehen, die miteinander verschaltet beliebige logische Funktionen realisieren können. Neben diesen Grundbausteinen enthält ein FPGA spezialisierte Schaltungselemente wie IO-Blöcke zur Kommunikation mit der Außenwelt, Taktgeneratoren und Block-RAM, mit dem Daten schnell gespeichert und abgerufen werden können. Die Komponenten lassen sich über Bus- und Multiplexer-Strukturen miteinander verbinden, sodass Daten in hoher Geschwindigkeit über den gesamten Chip ausgetauscht werden können.

Mithilfe von speziellen Hardwarebeschreibungssprachen wie Very High Speed Integrated Circuit Hardware Description Language (VHDL) oder Verilog ist es möglich, eine abstrakte Schaltung auf Registertransferebene zu definieren und mit spezieller Software in eine physische Schaltung umzuwandeln (Synthese), die anschließend auf konkrete Hardware platziert werden kann (Implementierung). Das Platzieren der Schaltung auf den FPGA und die möglichst effiziente Verbindung der einzelnen Komponenten unter Beachtung örtlicher und zeitlicher Beschränkungen ist ein schwieriges Optimierungsproblem, das als *Place and Route* bekannt ist und von entsprechender Software meist mit Approximationsverfahren gelöst wird.

```
1 main : process (clk)
2 begin
3         if rising_edge(clk) then
4                 and_out  <- in1 and in2;
5                 nand_out <- not and_out;
6         end if;
7 end process;
```

**Abbildung 2.4:** VHDL-Code eines synchronen Prozesses *main*

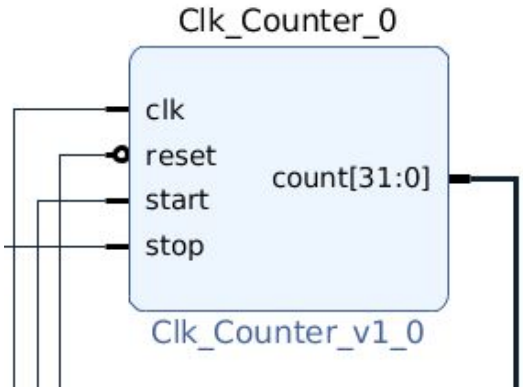



**Abbildung 2.5:** Graphische Darstellung einer Schaltungskomponenteim Block-Design-Diagramm von Vivado

Im Rahmen dieser Arbeit wurde die Hardwarebeschreibungssprache VHDL verwendet, mit deren Hilfe Schaltungskomponenten (Entities) und ihre Funktionalität (Architecture) definiert werden können. Ähnlich wie in der objektorientierten Programmierung lassen sich damit Funktionseinheiten erstellen, die wiederum in anderen Funktionseinheiten verwendet werden können (Components). Das Konzept der Signals entspricht in etwa dem der typisierten Variablen in anderen Programmiersprachen. Anstelle von Funktionen verwendet VHDL das Konzept der Prozesse, die bestimmte Signals "beobachten" und ausgeführt werden, sobald sich deren Werte ändern. Üblicherweise besitzt jede Komponente einen Takteingang (in dieser Arbeit immer `clk` genannt), der den Hauptprozess einer Komponente steuert.

Abb. 2.4 zeigt ein Beispiel für einen einfachen taktflankengesteuerten Prozess, der zwei Signals `and_out` und `nand_out` zuweist. Ein wichtiger konzeptioneller Unterschied zu höheren Programmiersprachen besteht in der Ausführungsreihenfolge der Anweisungen: Während Code in Hochsprachen sequenziell ausgeführt wird, geschehen Zuweisungen auf der gleichen Hierarchieebene in VHDL *gleichzeitig*. Dies hat zur Folge, dass `and_out` im obigen Beispiel zum Zeitpunkt der Zuweisung in Zeile 5 noch den alten Wert vor der Zuweisung in Zeile 4 enthält. Erst nach einem weiteren Taktzyklus nimmt `nand_out` den korrekten Wert an. Komplexe Programme in VHDL sind häufig wie Zustandsautomaten aufgebaut, um die Logik der Schaltung in mehrere aufeinanderfolgende Phasen zu unterteilen.

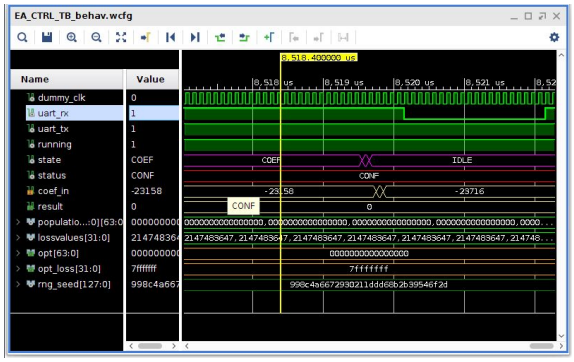



**Abbildung 2.6:** Simulator von Vivado

Zur Simulation, Synthetisierung und Implementierung des VHDL-Codes wurde in dieser Arbeit die Software Vivado verwendet, die umfangreiche Funktionalität für alle Schritte des Hardware-Designprozesses zur Verfügung stellt. Ein nützliches Hilfsmittel für das Design von FPGA-Schaltungen ist die graphische Repräsentation von Bauteilen als gekapselte Funktionsblöcke (siehe Abb. 2.5), deren Leitungen durch Linien miteinander verbunden werden können und so einen besseren Überblick über komplexe Schaltungen verschaffen als es die rein textuelle Repräsentation vermag. Besitzt die VHDL-Definition so eines Bauteils generische Parameter, lassen sich auch die graphischen Bausteine über ein Menü konfigurieren. Aus der graphischen Repräsentation generiert Vivado bei der Synthese wieder VHDL-Code.

Um eigene Module und Schaltungen zu testen, stellt Vivado weiterhin einen leistungsfähigen Simulator zur Verfügung, der die Werte sämtlicher enthaltenen Signale graphisch über den Zeitverlauf darstellen kann (siehe Abb. 2.6). Üblicherweise wird der Simulator auf einer *Testbench* ausgeführt, die den Takt mithilfe speziell dafür vorgesehener VHDL-Sprachkonstrukte sowie sämtliche Eingaben des zu testenden Moduls bereitstellt. Mit diesem Werkzeug lassen sich komplexe Schaltungen bequem debuggen, ohne den gesamten Synthese- und Implementierungsprozess durchlaufen zu müssen.

Die Implementierung auf einem FPGA bringt große Geschwindigkeitsvorteile: Ein FPGA funktioniert als rein logische Schaltung; er besitzt kein fest eingebautes Betriebssystem mit Hintergrunddiensten und langsamen Peripheriegeräten und lässt sich dadurch weitaus schneller takten als herkömmliche Computer (bis zu einigen hundert MHz). Die direkte Implementierung von Algorithmen auf Registerebene ist weitaus effizienter, da einerseits Software-Abstraktionen wie Funktionsaufrufe und andererseits Betriebssystemfunktionalität wie Scheduling entfallen und die Anzahl Rechenoperationen pro Taktzyklus so drastisch erhöht wird. Gerade für gut parallelisierbare Algorithmen sind FPGAs daher ein verlockendes Medium.

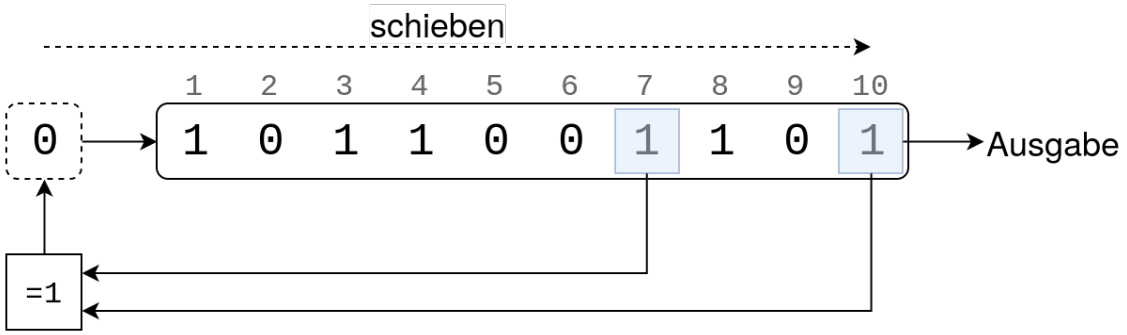



**Abbildung 2.7:** Aufbau eines Fibonacci-LSFRs mit 10 Bits und *Taps* 7 und 10

## 2.4 Pseudozufallszahlen

Zufall ist ein maßgeblicher Teil der evolutionären Optimierung: Bei der Mutation werden zufällige Teile eines Elternindividuums verändert, bei der Rekombination zwei oder mehr zufällig ausgewählte Elternteile auf ebenfalls zufällige Weise zusammengefügt. Da der FPGA kein Betriebssystem und daher auch keine vorgefertigten Bibliotheksfunktionen mitbringt, sind auch keine Funktionen vorhanden, um solche Zufallszahlen zu generieren. Weiterhin ist das Erzeugen echter Zufallszahlen auf einem deterministischen System wie dem FPGA unmöglich, wenn nicht echte physikalische Prozesse wie zufälliges Rauschen herangezogen werden.

Um auf solchen Systemen dennoch zufällige Prozesse zu simulieren, werden Pseudozufallszahlengeneratoren verwendet, die mit deterministischen Verfahren scheinbar zufällige Folgen von Zahlen erzeugen können. Üblicherweise besitzt ein solcher Generator einen inneren Zustand, der mit einem Startzustand (Seed) initialisiert werden muss. Durch eine Vielzahl von Methoden wie statistischen Tests kann die Güte eines Zufallsgenerators bestimmt werden; besonders im Bereich der Kryptographie werden hohe Anforderungen an die Güte solcher Generatoren gestellt (siehe z. B. [26]). Ein einfaches Kriterium zur Bewertung der Güte liefert die Periodenlänge des Generators: Da ein Generator einen inneren Zustand besitzt und die Anzahl möglicher Zustände beschränkt ist, muss ein Generator nach einer endlichen Anzahl Iterationen wieder in den gleichen Zustand gelangen. Da der Generator deterministisch ist, wiederholt sich ab diesem Zeitpunkt die Folge von Zufallszahlen. Die Länge der Zufallszahlenfolge, bis der Generator wieder in seinen Startzustand gelangt, ist die Periodenlänge. Ist das Verhältnis von Periodenlänge zur Größe des Zufallsraums (für Binärvektoren der Länge $n$ z. B. $|\{0,1\}^n| = 2^n$) größer oder gleich 1, können potenziell alle möglichen Werte ausgegeben werden, bis sich die Zufallsfolge wiederholt.

Wohl einer der einfachsten Pseudozufallsgeneratoren ist das linear rückgekoppelte Schieberegister oder Linear-Feedback Shift Register (LFSR). Es besteht aus einem Register mit $K$ Bits (siehe Abb. 2.7); das höchste Bit mit Index $K$ bildet die Ausgabe. Um ein neues Ausgabe-Bit zu erzeugen, wird eine Teilmenge der Bits im Register sequenziell exklusiv

verodert. Das entstehende Bit wird am niederwertigsten Index eingefügt und alle anderen Bits nach rechts geschoben, sodass das Bit an Index $K-1$ nun das neue Ausgabe-Bit ist und das alte Ausgabe-Bit verworfen wird. Dieser Vorgang wird beliebig oft wiederholt, um eine Folge von Pseudozufallsbits zu erzeugen.

Da das Ausgabe-Bit direkt von Zustand des LFSRs abhängig und der Zustandsraum mit $2^K$ verschiedenen Zuständen endlich ist, ist die durch ein LFSR erreichbare Periodenlänge offenbar maximal $2^K$. Die Indizes der Bits, die exklusiv verodert werden, um den neuen Zustand zu erzeugen, werden *Taps* genannt und bestimmen neben der Registergröße $K$ maßgeblich die Güte des LFSRs: Nur bei geschickter Wahl der Taps werden sämtliche möglichen Zustände durchlaufen, sodass die Periodenlänge genau $2^K - 1$ beträgt – der Zustand, in dem alle Bits gleich 0 sind, ist ausgenommen, da er weder von einem anderen Zustand aus betreten noch verlassen werden kann, weshalb er auch nicht als Startzustand gewählt werden darf. Optimale Taps, also solche, die die maximale Periodenlänge für eine gegebene Registergröße $K$ erzeugen, ergeben sich aus bestimmten Polynomialdarstellungen der Rückkopplungsoperation und sind für alle üblichen $K$ bekannt[1].

Ein Nachteil von LFSRs ist die Eigenschaft, dass die Ausgabe nur aus einem einzigen Bit besteht. Eine naive Implementierung in VHDL führt zu einer Ausgabe von einem Zufallsbit pro Taktzyklus. Da beispielsweise bei der Mutation der Bit-Vektoren im EA-Optimierer viele Zufallszahlen benötigt werden, die nicht nur aus einem einzigen Bit bestehen, würde die Verwendung dieses Typs von Pseudozufallszahlengenerator die Anzahl benötigter Taktzyklen deutlich erhöhen. Zwar können durch geschickte Implementierung mehrere Bits auf einmal aus einem LFSR gewonnen werden[2], allerdings erhöht sich die Komplexität der Implementierung dadurch deutlich. Auch die Verwendung mehrerer parallel arbeitender LFSRs ist möglich, doch liegt die Schwierigkeit dabei in der Initialisierung der Generatoren: Um 32-Bit Zufallszahlen mit einer Periodenlänge von ca. $2^{32}$ in einem Taktzyklus zu ergeuzen, sind 32 Seeds mit je 32 Bits nötig.

Eine vielversprechende Alternative zu LFSRs ist `xoroshiro` [4], der gleichzeitig auf zwei Zustandsregistern mithilfe der Operationen XOR, Rotation und Shift arbeitet. Die Variante `xoroshiro128+` beispielsweise benötigt einen Seed von 128 Bits und erzeugt in einer Iteration 64 Zufallsbits, die als zwei 32-Bit breite Ganzzahlen interpretiert und genutzt werden können.

[1] siehe z. B. `https://www.xilinx.com/support/documentation/application_notes/xapp052.pdf`
[2] `https://zipcpu.com/dsp/2017/11/13/lfsr-multi.html`

# Kapitel 3

# Konzept

Basierend auf den Erkenntnissen zu den Grundlagen der Evolutionären Optimierung und den Vorteilen der FPGA-Programmierung soll in diesem Kapitel ein QUBO-Optimierer entworfen werden. In Kapitel 4 soll dieses Konzept schließlich in VHDL-Code umgesetzt werden.

## 3.1 Evolutionsstrategie

Die erste Entscheidung muss hinsichtlich der Evolutionsstrategie getroffen werden. Trotz zahlreicher Weiterentwicklungen und Verbesserungen stechen $(\mu+\lambda)$-EA und $(\mu, \lambda)$-EA als solide Vertreter der Evolutionsstrategien in der Literatur hervor. Vor allem der $(\mu+\lambda)$-EA ist hinsichtlich seiner Konvergenzeigenschaften gut untersucht und erreicht auf lange Sicht garantiert ein globales Maximum. Daher fällt die Wahl auf diese EA-Variante.

Ein wichtiges Ziel dieser Arbeit ist es, einen Optimierer zu entwickeln, der *parametrisierbar* ist, d. h. die Dimension $N$, die EA-Parameter $\mu$ und $\lambda$ und auch die Bit-Breite $b_\beta$ der Zielfunktionsparameter sollen veränderlich sein. Weiterhin soll der Optimierer verschiedene initiale Seeds erhalten können, um so die Mutation mit verschiedenen Zufallszahlen durchführen und gegebenenfalls zu anderen Optima konvergieren zu können. Als Abbruchkriterium soll ein Budget dienen, das ebenfalls mit einem beliebigen Wert initialisiert werden kann und nach jeder Iteration des EAs um 1 verringert wird. Alternativ soll der Optimierer in Endlosschleife betrieben werden und das Optimum periodisch abgefragt werden können. Eine weitere wichtige Funktionalität ist der Ising-Modus, mit dem die Zielfunktionskoeffizienten als Interaktionen zwischen den Variablen eines Ising-Modells interpretiert werden – durch diese Erweiterung eröffnet sich eine große Klasse weiterer Optimierungsprobleme, die mit demselben Optimierer gelöst werden können.

Abb. 3.1 zeigt den schematischen Aufbau des EA-Optimierers. Die gesamte Population, bestehend aus Eltern und Nachkommen, liegt als Array vor. Damit die Zielfunktionswerte nicht in jeder Iteration neu berechnet werden müssen, werden zusammen mit den Indi-

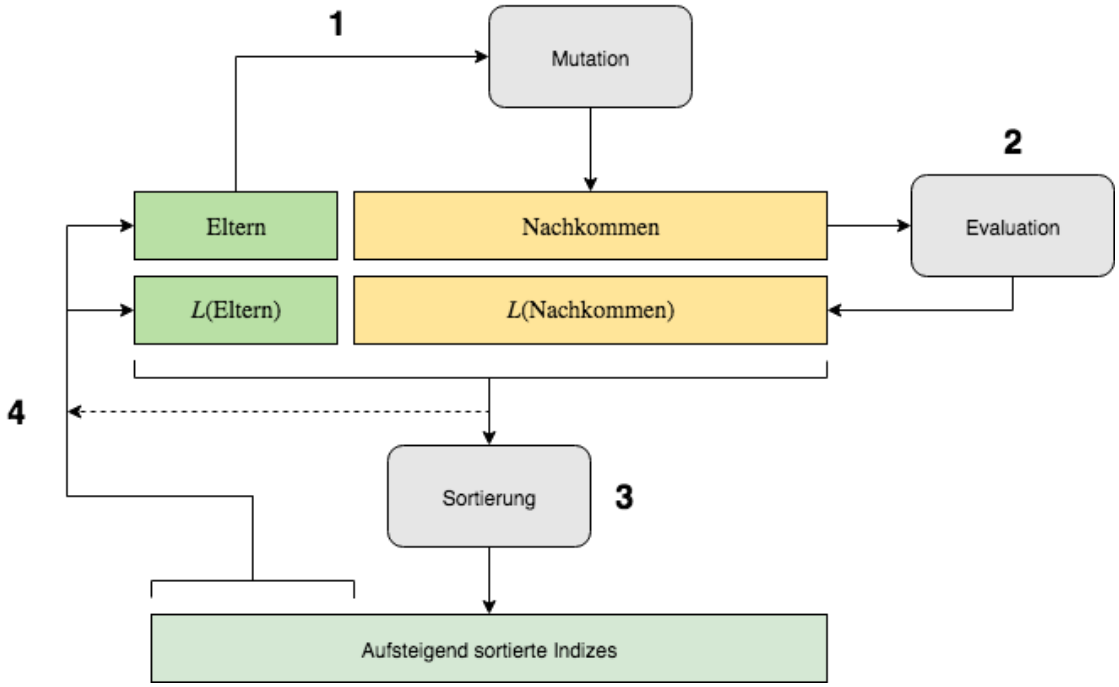



**Abbildung 3.1:** Schematische Funktionsweise des EA

viduen auch ihre jeweiligen Zielfunktionswerte gespeichert. In Schritt **1** werden zufällige Eltern aus der Elternpopulation ausgewählt und mutiert. Die so entstandenen Nachkommen werden in die Nachkommenpopulation geschrieben. In Schritt **2** wird die Zielfunktion auf den soeben generierten Nachkommen ausgewertet und das Ergebnis in das Array mit Zielfunktionswerten geschrieben. Als nächstes wird die gesamte Population aufsteigend nach Zielfunktionswert sortiert, sodass die Individuen mit minimalen Werten ganz links im Array stehen. Die Sortierung erfolg in zwei Schritten: Zunächst wird in Schritt **3** eine Index-Sortierung auf den Zielfunktionswerten durchgeführt, d. h. das Ergebnis sind nicht die Werte selbst sondern eine Permutation der Indizes. Basierend auf dieser Sortierung werden die $\mu$ Indizes mit kleinstem Zielfunktionswert in Schritt **4** sowohl auf die Population selbst als auch auf die Zielfunktionswerte der Population angewandt und die entsprechenden Individuen und Werte in die Elternpopulation geschrieben. Nach diesem letzten Schritt liegen also die besten Individuen der gesamten Population in der Elternpopulation: Falls durch die Mutation keine Verbesserung stattgefunden hat, wurden die vorherigen Elternindividuen wieder in die Elternpopulation sortiert. Wurde durch Mutation ein Nachkomme mit kleinerem Zielfunktionswert gefunden, verdrängt dieser ein Elternindividuum wie in einer Rangliste aus der Elternpopulation. Durch die Sortierung der gesamten Population statt nur der Nachkommen wird die elitistische Plus-Selektion umgesetzt (siehe Abschnitt 2.1.1).

Auffällig bei diesem Schema ist das *Fehlen der Rekombination*. Die Notwendigkeit der Rekombination ist in der Literatur umstritten; Nachweise für die Verbesserung der Konver-

genzrate durch Rekombination existieren nur für bestimmte Klassen von Zielfunktionen. In [18] werden die *Royal Road Functions* vorgestellt, die Bit-Vektoren anhand zusammenhängender Blöcke von Bits auswerten. Obwohl die Nützlichkeit der Rekombination für solche Zielfunktionen intuitiv erscheint, zeigen Jansen und Wegener [15], dass auch der (1+1)-EA, der mit nur einem einzigen Elternteil pro Population offenbar keine Rekombination durchführt, auf diesen Funktionen effizient arbeitet, und stellen die noch weiter eingeschränkte Klasse der *Real Royal Road Functions* vor, die tatsächlich ohne große Einschränkungen des EAs nachweislich besser mit Rekombination optimierbar ist. Da der größte Teil aller polynomieller Zielfunktionen allerdings nicht diesem Schema entspricht, ist der Vorteil der Verwendung von Rekombination fraglich.

Aufgrund dieser Erkenntnisse und des hohen Mehraufwands zur Implementierung des Rekombinationsoperators wurde im Rahmen dieser Arbeit auf Rekombination verzichtet. Alle anderen Operatoren sollen in den folgenden Abschnitten erläutert werden.

Als Referenz kann eine Python-Implementierung der oben beschriebenen Evolutionsstrategie in Anhang A.1 herangezogen werden.

### 3.1.1 Mutationsoperator

Der Mutationsoperator auf Bitvektoren wird gemäß Gleichung (2.1) umgesetzt, d. h. für jedes zu mutierende Bit wird eine Bernoulli-verteilte Zufallsvariable $z_i$ erzeugt, die anzeigt, ob das Bit invertiert wird. Die Erfolgswahrscheinlichkeit wird auf $p = 1/N$ gesetzt, dadurch beträgt die erwartete Anzahl invertierter Bits unabhängig von der Dimension des Optimierungsproblems

$$\mathbb{E}\left[\sum_{i=1}^{N} z_i\right] = \sum_{i=1}^{N} \mathbb{E}[z_i] = \sum_{i=1}^{N} \frac{1}{N} = 1.$$

Die Zufallszahlen können simuliert werden, indem eine zufällige ganze Zahl erzeugt und Modulo $N$ gerechnet wird; Ist die resultierende Zahl durch $N$ teilbar, wird das Bit invertiert, andernfalls unverändert belassen. Da der verwendete Pseudozufallszahlengenerator genau eine 64-Bit-Zufallszahl pro Taktzyklus erzeugt, benötigt die Mutation daher $\mathcal{O}(N)$ Taktzyklen.

Bei der Implementierung in einer höheren Programmiersprache würde die Mutation sämtlicher Individuen sequenziell erfolgen, sodass die Laufzeit insgesamt $\mathcal{O}(\lambda N)$ betragen würde: Obwohl in der Python-Implementierung scheinbar alle Nachkommen gleichzeitig mithilfe der Listenkomprehension erzeugt werden, ist dies doch nur syntaktischer Zucker für eine sequenzielle Verarbeitung – hier findet keine echte Parallelisierung statt!

```
39          # mutate random parents to create child population
40          population[N_PARENTS:] = [ mutated(v) for v in random.choices(
41              population[:N_PARENTS], k=N_CHILDREN) ]
```

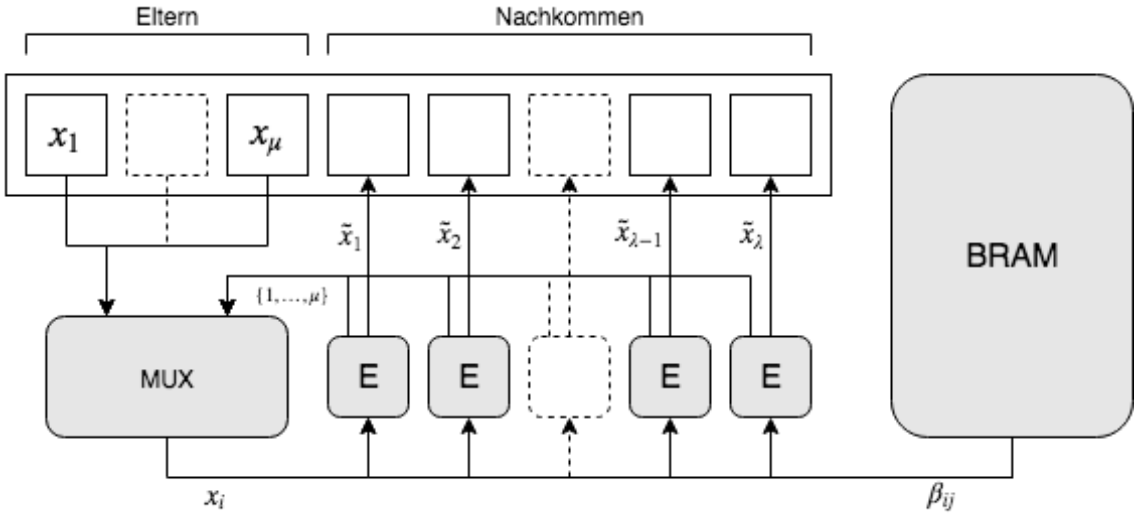



**Abbildung 3.2:** Schema der Evaluator-Komponente

Der große Vorteil der Hardware-Implementierung liegt in der Verwendung von Mikroparallelisierung: Anstatt die Mutation nacheinander auszuführen, soll für jeden Nachkommen ein eigenes Mutations-Modul erstell werden, das genau ein Elternindividuum bearbeitet. Diese $\lambda$ Module erzeugen alle Nachkommen gleichzeitig, dadurch wird die Zeit für die Mutation der gesamten Population auf $\mathcal{O}(N)$ verringert. Um in jedem Zyklus die notwendige Zufallsvariable berechnen zu können, enthält jedes solche Modul eine eigene Instanz des Pseudozufallszahlengenerators. Zu beachten ist dabei, dass alle Generatoren mit verschiedenen Seeds initialisiert werden, da sonst sämtliche Mutationsmodule die gleichen Bits invertieren und der EA somit zu einem $(1 + 1)$-EA degenerieren würde.

### 3.1.2 Zielfunktionsauswertung

Als Grundlage für die Zielfunktionsauswertung dient Gleichung (2.2), d. h. für ein Optimierungsproblem mit $N$ Variablen werden genau $N(N + 1)/2$ Koeffizienten benötigt.

Ähnlich wie in Abschnitt 3.1.1 werden, statt die Zielfunktionsauswertung sequenziell auf alle Nachkommen anzuwenden, $\lambda$ Module (*Evaluator*) erstellt, die parallel auf je einem Individuum arbeiten.

Die Speicherung sämtlicher Koeffizienten in jedem einzelnen Modul ist nicht praktikabel, da die Datenmenge bereits für $N = 512$ mit 32-Bit breiten Koeffizienten etwa 4 MiB beträgt. Eine redundante Speicherung von $\lambda$ Kopien würde diesen Wert um den gleichen Faktor erhöhen, was die Speicherkapazität selbst größerer FPGAs schnell überschreitet. Um nur eine Kopie der Koeffizienten zu speichern, müssen diese den Zielfunktionsauswertungsmodulen von außen übergeben werden.

Da die zu mutierenden Elternindividuen ebenfalls zufällig aus der Elternpopulation gewählt werden, ist auch hierfür ein Zufallswert nötig: Um nicht noch weitere Zufallsgene-

ratoren erzeugen zu müssen, wird der Index des nächsten zu mutierenden Elternteils von den Evaluator-Modulen selbst generiert, wobei die gleiche Methode wie zur Erzeugung der Zufallsvariable $z_i$ verwendet wird. Die Elternindividuen mit den entsprechenden Indizes werden dem Evaluator-Modul übergeben, beispielsweise mittels eines Multiplexers mit $\lambda$ Ein- und Ausgängen, und anschließend wird die Zielfunktion darauf ausgewertet, indem sequentiell jedes $\beta_{ij}$ übergeben und genau dann zum Zielfunktionswert addiert wird, wenn $x_i$ und $x_j$ beide gleich 1 sind. Herausforderungen dieser Methode liegen in der genauen Synchronisierung von Koeffizientenstrom und Index-Abgleich, sodass ein Koeffizient pro Taktzyklus ausgewertet werden kann, und in der Initialisierung sämtlicher Zufallsgeneratoren mit unterschiedlichen Seeds.

Die Koeffizienten $\beta_{ij}$ sind idealerweise unbeschränkte reelle Zahlen. Da dies auf einem Rechner oder FPGA mit begrenzten Resourcen nicht umsetzbar ist, werden zur Kodierung eines Koeffizienten 32 Bits verwendet. Die naheliegende Entsprechung reeller Zahlen sind Gleitkommazahlen, allerdings ist Gleitkommaarithmetik aufwändiger und durch Skalierungseffekte bei der Addition sehr großer und sehr kleiner Zahlen potenziell ungenau. Eine weitere Möglichkeit ist die Verwendung von Festkommazahlen. Da allerdings die Skalierung der Parameter keinen Einfluss auf die Lage des Optimums hat, ist es möglich, ein Optimierungsproblem mit Parametern in Festkommadarstellung mit $k$ Nachkommastellen durch Multiplikation mit $10^k$ in ein äquivalentes Optimierungsproblem auf ganzzahligen Koeffizienten umzuwandeln. Offenbar bleibt auch der Wertebereich gleich groß, da sowohl Festkommazahlen als auch ganze Zahlen durch 32 Bit dargestellt werden, die Menge der Festkommazahlen also von vornherein bloß eine Skalierung der Menge aller 32-Bit-Zahlen ist.

Weiterhin zeigen Anwendungen, beispielsweise im Bereich der probabilistischen graphischen Modelle, dass ganzzahlige Approximationen von Modellparametern zu ähnlich guten Resultaten führen wie unter Verwendung von Gleitkommazahlen [23]. Aus diesem Grund soll der EA-Optimierer ausschließlich mit ganzzahligen Koeffizienten in Zweierkomplementdarstellung arbeiten.

Der bereits erwähnte hohe Speicherverbrauch der Koeffizienten soll mithilfe eines weiteren Parameters $b_\beta$ kontrolliert werden können, der die Anzahl Bits pro Koeffizient festlegt. Auf diese Weise können Optimierungsprobleme mit einer größeren Anzahl Variablen gelöst werden, deren Koeffizienten in 32-Bit-Darstellung nicht auf dem Chip gespeichert werden könnten.

Neben der Optimierung der QUBO-Zielfunktion soll der EA-Optimierer auch das eng verwandte Ising-Modell, das in Abschnitt 2.2.3 vorgestellt wurde, minimieren können. Da der Unterschied zwischen diesen beiden Modellen lediglich in der Interpretation der binären Variablen bei der Zielfunktionsauswertung besteht, kann mit der gleichen Architektur wie

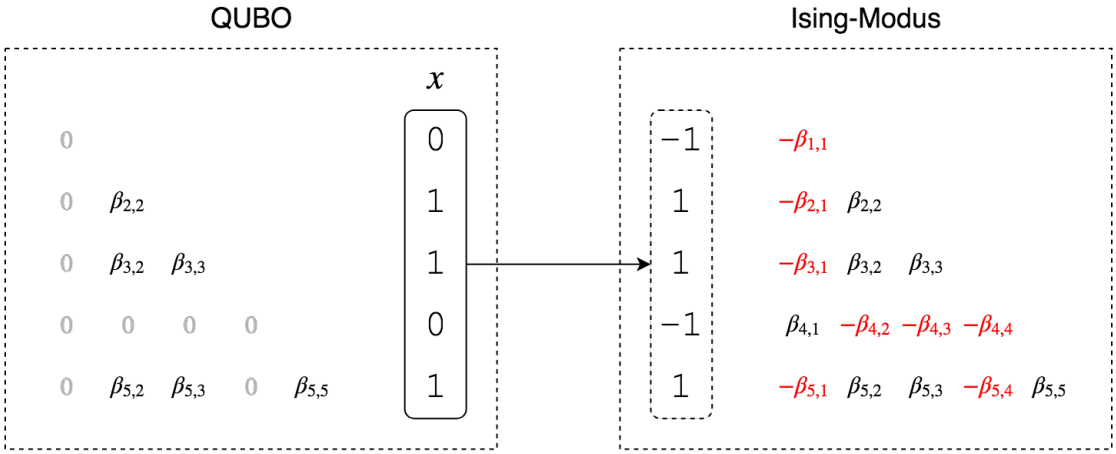



**Abbildung 3.3:** Zusammensetzung des Zielfunktionswerts für QUBO- und Ising-Optimierung; Nullen werden als $-1$ interpretiert, $\beta_{ii}$ als lineare Koeffizienten

für den QUBO-Optimierer ein Optimierer für das Ising-Modell gebaut werden, der sich nur im Evaluator-Modul unterscheidet.

Die unterschiedliche Interpretation der Binärvariablen ist in Abb. 3.3 dargestellt: Sind $x_i$ und $x_j$ des aktuellen Individuums gleich, so gilt $x_i \cdot x_j = 1$ und $\beta_{ij}$ fließt positiv in die Summe der Koeffizienten ein, andernfalls ist $x_i \cdot x_j = -1$ und $\beta_{ij}$ wird abgezogen. Ist $i = j$, wird der Parameter $\beta_{ii}$ als linearer Term interpretiert und entsprechend dem Vorzeichen von $x_i$ entweder addiert oder subtrahiert.

Zwischen der QUBO-Optimierung und dem "Ising-Modus" soll gewechselt werden können, ohne den FPGA neu zu programmieren.

### 3.1.3 Selektion

Aus der Population mit variabler Größe $\mu+\lambda$ soll der EA-Optimierer, nachdem die Zielfunktionswerte der Nachkommen berechnet wurden, die besten $\mu$ Individuen auswählen und in die neue Elterngeneration übernehmen. Während diese Auswahl für den Fall $\mu = \lambda = 1$ durch einen einzigen Vergleich durchgeführt werden kann, ist die Selektion für größere Populationen nicht trivial und erfordert offenbar eine Sortierung der Population bezüglich des Zielfunktionswerts.

Eine Schwierigkeit der Implementierung eines Sortieralgorithmus' auf Hardware-Ebene liegt darin, dass viele Algorithmen (z. B. Quicksort, Mergesort) rekursiv definiert sind. Auf einer Schaltung kann aber kein rekursiver Aufruf erfolgen, deshalb müssen sämtliche Rekursionsschritte bis hin zum Basisfall implementiert und "von unten nach oben" synchronisiert werden, sodass zuerst die Basisfälle fertiggestellt werden, dann die Stufe darüber usw., bis schließlich der gesamte Aufruf beendet ist.

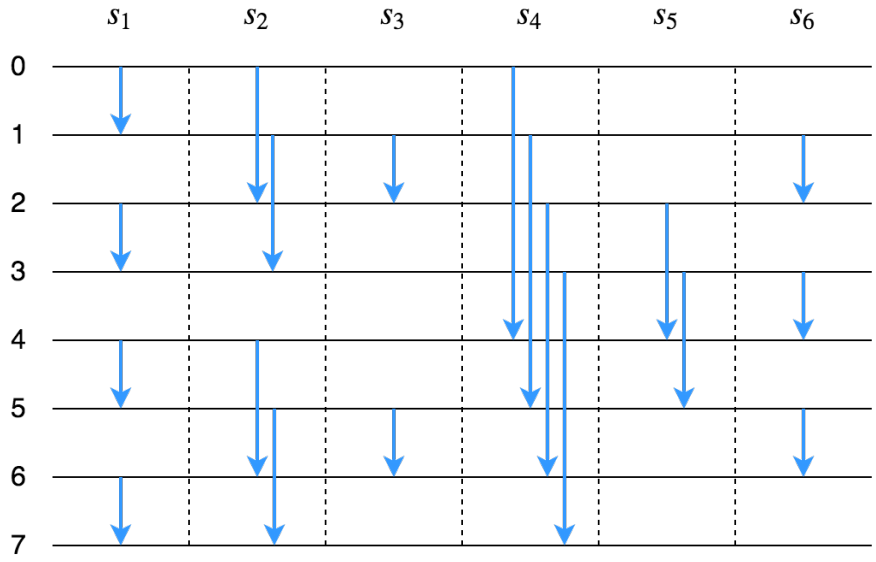



**Abbildung 3.4:** Diagramm von Odd-Even-Mergesort mit $n = 8$ als Sortiernetz mit sechs Phasen; ein blauer Pfeil symbolisiert einen Vergleich und ggf. eine Vertauschung zweier Elemente

Ein vielversprechendes Konzept für die Hardware-Implementierung von Sortierverfahren sind *Sortiernetze*. Ihr Basisoperator ist der *Comparator*, der einen Vergleich zweier Elemente auf einem Array $\boldsymbol{a} \in A^n$ durchführt und diese, falls sie nicht in aufsteigender Reihenfolge sind, tauscht[1]:

$$\begin{aligned}
[i:j](\boldsymbol{a})_i &= \min\{a_i, a_j\} \\
[i:j](\boldsymbol{a})_j &= \max\{a_i, a_j\} \\
[i:j](\boldsymbol{a})_k &= a_k \quad \text{für alle } k \in \{0,\ldots,n-1\}\backslash\{i,j\}
\end{aligned}$$

Mehrere Vergleiche können gleichzeitig innerhalb einer *Phase* $s : A^n \mapsto A^n$, $s(\boldsymbol{a}) = ([i_k : j_k] \circ \cdots \circ [i_1 : j_1])(\boldsymbol{a})$ durchgeführt werden, solange alle Indizes $i_\ell$ und $j_\ell$ mit $\ell \in \{1,\ldots,k\}$ verschieden sind (das gleiche Element kann nicht gleichzeitig mit mehreren anderen vertauscht werden). Ein Sortiernetz ist eine Abfolge von Phasen $S = s_m \circ \cdots \circ s_1$, sodass für jedes Eingabe-Array $\boldsymbol{a} \in A^n$ das resultierende Array $S(\boldsymbol{a})$ sortiert ist. Eine übliche graphische Darstellung von Sortiernetzen zeigt Abb. 3.4: Die Indizes des Arrays $\boldsymbol{a}$ werden als Linien gezeichnet; ein Vergleich $[i:j](\boldsymbol{a})$ wird als Pfeil von $i$ nach $j$ dargestellt.

Ein Beispiel für ein solches Sortiernetz ist *Odd-Even-Mergesort* nach Batcher [2], das mit $\mathcal{O}(n \log n)$ Vergleichen ein Array der Länge $n$ sortieren kann. Der Algorithmus besteht aus den Operationen *merge* und *sort*, die rekursiv aufgerufen werden. Die Operation *merge* erhält als Eingabe ein Array $\boldsymbol{a}$ der Länge $n$, dessen Hälften bereits sortiert sind, und gibt ein komplett sortiertes Array aus. Ist $k = 1$, so hat das Array nur zwei Elemente und

[1] Notation nach `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/oemen.htm`

die Ausgabe ist $[0:1](\boldsymbol{a})$. Bei $k > 1$ wird *merge* rekursiv jeweils auf die Unterarrays aller geraden bzw. ungeraden Indizes angewendet, $\boldsymbol{a}_{\text{even}} = [a_0, a_2, \ldots, a_{n-2}]$ und $\boldsymbol{a}_{\text{odd}} = [a_1, a_3, \ldots, a_{n-1}]$. Auf dem resultierenden Array $\boldsymbol{a'}$ werden anschließend die Vergleiche $[i:i+1](\boldsymbol{a'})$ für alle $i \in \{1, 3, \ldots, n-3\}$ durchgeführt.

Die Operation *sort* sortiert schließlich ein komplett zufälliges Array: Im Fall $n = 1$ muss nichts getan werden, da ein einelementiges Array immer sortiert ist. Wenn $n > 1$, wird *sort* rekursiv auf die beiden Hälften des Arrays angewendet und das Ergebnis mithilfe von *merge* zusammengefügt. Zu beachten ist, dass der Algorithmus ein Array voraussetzt, dessen Länge eine Zweierpotenz ist. Ein Array anderer Länge muss dementsprechend mit Elementen aufgefüllt werden, die auf die Sortierung der ursprünglichen Elemente keinen Einfluss haben, für eine aufsteigende Sortiertung beispielsweise indem der Wert $\infty$ angehängt wird.

Da die rekursiven Aufrufe in *sort* und *merge* stets auf disjunkten Unterarrays auftreten, können sie zu einer Phase zusammengefasst werden, sodass sich insgesamt der Ablauf von Phasen ergibt, der in Abb. 3.4 zu sehen ist.

Die Schwierigkeit bei der Implementierung in VHDL liegt in der Einhaltung der korrekten Reihenfolge bei der Auswertung der rekursiven Aufrufe. Da sowohl die Population als auch die Zielfunktionswerte selbst sortiert werden müssen, ist eine entscheidende Abwandlung des oben beschriebenen Sortiernetzes, dass es statt der sortierten Daten selbst die sortierten Indizes ausgibt, die anschließend auf Population und Zielfunktionswerte angewendet werden können.

## 3.2 Kommunikation mit dem FPGA

Um den EA-Optimierer möglichst einfach zu benutzen wird eine Schnittstelle benötigt, über die mit dem FPGA kommuniziert werden kann. Über diese Schnittstelle müssen beispielsweise die Koeffizienten übertragen und das aktuelle Optimum abgefragt werden können.

Mögliche Schnittstellen bilden einerseits das Display und die LEDs des FPGAs: Das Optimum und der zugehörige Zielfunktionswert könnten als Bitstring auf dem Display angezeigt und der Status des Optimierers über die LEDs kodiert werden. Diese Art der Kommunikation mit dem FPGA hat klare Nachteile, da einerseits die Darstellung von Zeichen auf dem Display aufwändig zu implementieren ist, und andererseits der FPGA in der Nähe sein muss, damit die Informationen abgelesen werden können.

Eine bessere Alternative bilden die seriellen Schnittstellen des FPGAs; eine Übertragung der Daten über eine PCI- oder UART-Verbindung kann von einem lokalen Rechner, an den der FPGA angeschlossen ist, an einen beliebigen entfernten Rechner mit entsprechender Client-Software weitergeleitet werden. Im Rahmen dieser Arbeit wird daher sowohl eine Server- als auch eine Client-Anwendung entwickelt, zwischen denen mithilfe des UART-Protokolls kommuniziert werden kann, um domänenspezifische Daten auszutauschen. Implementierungsdetails werden in Abschnitt 4.2 erläutert.

# Kapitel 4

# Implementierung

Mit der Implementierung in Python als konzeptionelle Grundlage entstand die VHDL-Implementierung des evolutionären Optimierers. Sie besteht aus drei wichtigen Teilen: Der eigentlichen Optimierungskomponente, einem *IO-Controller*, der auf dem FPGA eine Schnittstelle zwischen dem Optimierer und einem seriellen Port herstellt, und einem Python-Skript, über das mit der Schnittstelle kommuniziert wird. Diese Trennung von Optimierer und Kommunikationsschnittstelle erlaubt es, den Optimierer flexibel als Teil einer komplexeren Schaltung einzusetzen statt ausschließlich über serielle Schnittstellen direkt damit zu interagieren.

Die einzelnen Schritte der Optimierung, die in Kapitel 3 beschrieben sind, wurden in geschlossene Komponenten gekapselt, um das Programm semantisch zu strukturieren und somit übersichtlicher zu machen. Im Folgenden werden die Implementierung und das Zusammenspiel der einzelnen Komponenten sowie die Kommunikation mit dem Optimierer beschrieben.

## 4.1 Optimierer

Die VHDL-Implementierung des Optimierers gliedert sich in die Dateien `ram.vhd`, `rng_xoroshiro128plus.vhd`, `mutator.vhd`, `evaluator.vhd`, `sorter.vhd`, `merger.vhd` und `main.vhd`. Letztere enthält die Definition der EA-Komponente, die sämtliche in den anderen Dateien definierte Entities verwendet. In Abb. 4.1 ist der schematische Aufbau des Optimierers dargestellt, der vor allem für Abschnitt 4.1.1 als grobe Übersicht dienen soll; die Abkürzung RNG steht für den Zufallsgenerator (*random number generator*). Ein besonderes Augenmerk liegt dabei auf der Größe der Schaltung abhängig von den generischen Parametern, ein wichtiges Merkmal dieser Implementierung. In den folgenden Abschnitten soll die Funktionsweise der einzelnen Komponenten und deren Verwendung im Hauptprogramm beschrieben werden.

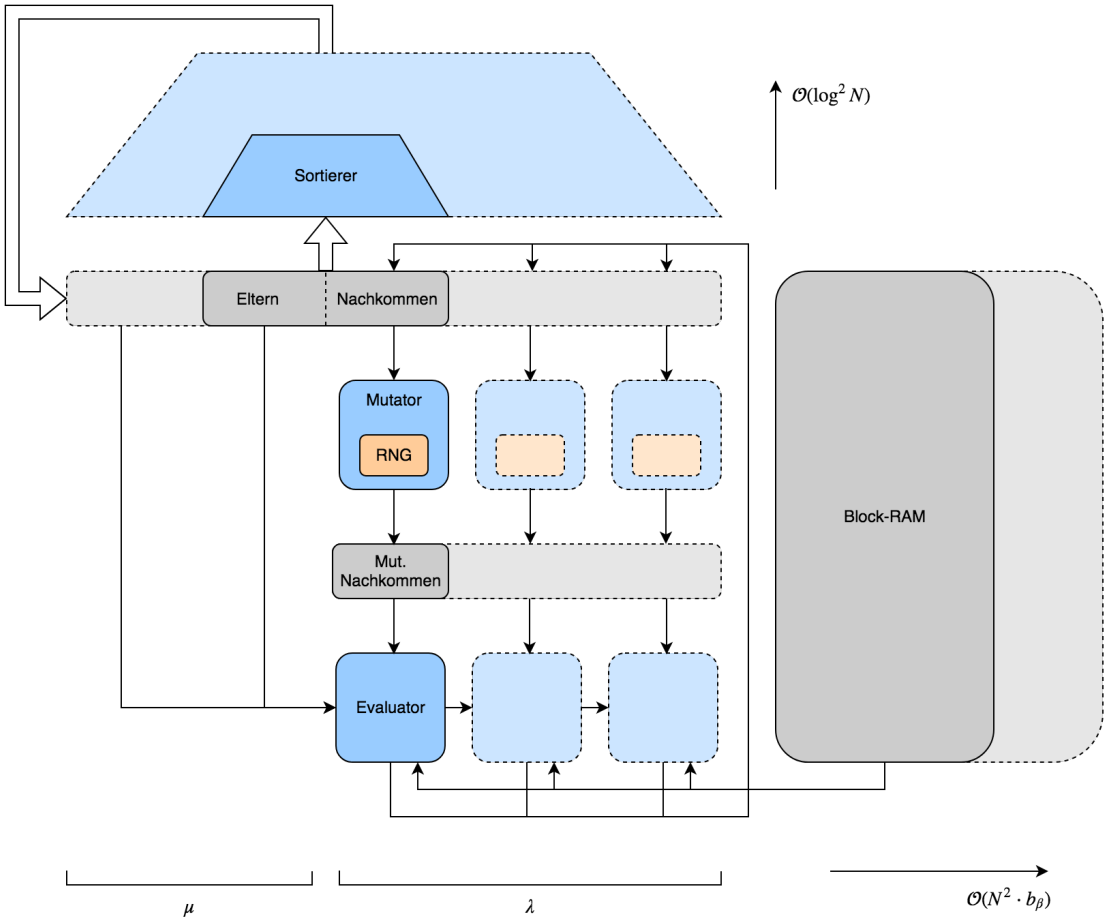



**Abbildung 4.1:** Schematischer Aufbau des EA-Optimierers auf dem FPGA-Chip mit Augenmerk auf der Skalierung abhängig von den Parametern $N$, $\mu$, $\lambda$ und $b_\beta$

```
8  entity main is
9      Generic (
10         DIM        : integer := 8;
11         N_PARENTS  : integer := 4;
12         N_CHILDREN : integer := 12;
13         BUDGET     : integer := -1;
14         COEF_WIDTH : integer := 32;
15
16         -- dependent
17         N_POPULATION : integer := 16; -- expr {$N_PARENTS+$N_CHILDREN}
18         NEXT_POWER   : integer := 4;  -- expr {ceil(log(double($N_POPULATION))/log(2))}
19         DATA_WIDTH   : integer := 128; -- expr {max($DIM, 128)}
20         N_COEF       : integer := 36  -- expr {($DIM*($DIM+1))/2}
21     );
22     Port (
23         clk         : in std_logic;
24         reset       : in std_logic;
25
26         config      : in std_logic_vector(1 downto 0);
27         data_ready  : in std_logic;
28         data_in     : in std_logic_vector (DATA_WIDTH-1 downto 0);
29         start       : in std_logic;
30
31         config_mode : out std_logic; -- indicate that the device is reset and in config
           ↪ mode
32         data_read   : out std_logic; -- indicate that data_in was successfully read
33         full        : out std_logic; -- indicate that either all coefs or parents where
           ↪ set
34
35         done        : out std_logic;
36         opt         : out std_logic_vector (DIM-1 downto 0);
37         opt_loss    : out std_logic_vector (31 downto 0)
38     );
39 end main;
```

**Abbildung 4.2:** Entity-Definition des EA-Optimierers

### 4.1.1 Main

Die Komponente `main` enthält die übergreifende Struktur des Optimierers, die alle anderen Komponenten vereint. Die Entity-Definiton ist in Abb. 4.2 aufgeführt. Mithilfe der generischen Parameter kann der Optimierer auf eine zu optimierende Funktion abgestimmt werden:

`DIM` Dimension $N$ der Bitvektoren

`N_PARENTS` Größe $\mu$ der Elternpopulation

`N_CHILDREN` Größe $\lambda$ der Nachkommenpopulation

`BUDGET` Anzahl auszuführender Iterationen oder $-1$ für Endlosbetrieb

`COEF_WIDTH` Anzahl Bits pro Koeffizient

Neben diesen freien Parametern gibt es noch vier weitere Generics, die abhängig von den zuvor genannten Parametern sind und dementsprechend korrekt gesetzt werden müssen:

`N_POPULATION` $=$ `N_PARENTS` $+$ `N_CHILDREN`, Größe der Gesamtpopulation

`NEXT_POWER` $= \min_k 2^k \geq$ `N_POPULATION`, kleinste Zweierpotenz größer oder gleich der Gesamtpopulationsgröße

`DATA_WIDTH` $= \max\{$`DIM`$, 128\}$, Größe des größten Bitvektors, der eingegeben werden kann

`N_COEF` $=$ `DIM` $($`DIM` $+ 1)/2$, Anzahl der Koeffizienten

Die `main`-Komponente enthält die Population sowie die zugehörigen Zielfunktionswerte als zwei getrennte Array-Signals.

```
163     signal population   : population_t (2**NEXT_POWER-1 downto 0) := (others => (others =>
        ↪  '0'));
164     signal lossvalues   : loss_array (2**NEXT_POWER-1 downto 0) := (others =>
        ↪  integer'HIGH);
```

Individuen und Zielfunktionswerte korrespondieren über ihre Indizes: Das $i$-te Element von `population` hat den Zielfunktionswert `lossvalues(i)`. Die Eltern- und Nachkommenpopulation sind hintereinander gespeichert; die Eltern von Index 0 bis $\mu - 1$, die Nachkommen von $\mu$ bis $\mu + \lambda - 1$. Die Speicherung in einem einzigen Signal vereinfacht die Sortierung, da bei der Plus-Selektion die neuen Eltern aus den besten Individuen beider Populationen zusammengenommen ausgewählt werden und somit das gesamte Array einfach an die Sortierkomponente übergeben werden kann. Da die Sortierkomponente, wie in Abschnitt 4.1.6 beschrieben, nur Arrays sortieren kann, deren Länge eine Zweierpotenz ist, ist der höchste Index von `population` und `lossvalues` nicht `N_POPULATION-1`, sondern `2**NEXT_POWER-1`, wobei `NEXT_POWER` $= K = \operatorname{argmin}_{k\in\mathbb{N}} 2^k \geq (\mu + \lambda)$ der Exponent der kleinsten Zweierpotenz ist, die größer oder gleich der Gesamtpopulationsgröße ist. Im ungünstigsten Fall, nämlich wenn $\mu + \lambda$ genau um eins größer als eine Zweierpotenz ist, sind `population` und `lossvalues` in etwa doppelt so groß wie die Gesamtpopulationsgröße. Um die vorhandene Chipfläche des FPGAs effizient zu nutzen, sollten die Parameter `N_PARENTS` und `N_CHILDREN` daher möglichst so gewählt werden, dass ihre Summe eine Zweierpotenz oder nur wenig kleiner als eine Zweierpotenz ist. Wird keine initiale Elternpopulation angegeben, wird die Population mit Nullvektoren, die Zielfunktionswerte mit 2147483647, dem größtmöglichen Wert eines 32-Bit-Integers, initialisiert. Da die Population aufsteigend nach Zielfunktionswert sortiert wird, ist so sichergestellt, dass die überzähligen $2^K - \mu - \lambda$ Array-Elemente stets als größte Elemente ans Ende des Arrays sortiert und somit nicht mutiert und ausgewertet werden. Das aktuelle Optimum ist als kleinstes Element stets an Index 0 von `population`.

Zur besseren Übersicht existieren Aliasse für die Eltern- und Nachkommenpopulationen, sodass diese mit einem eigenen Bezeichner und jeweils von Index 0 bis $\mu - 1$ bzw. $\lambda - 1$ addressiert werden können:

```
166     alias parents       : population_t (N_PARENTS-1 downto 0) is population (N_PARENTS-1
        ↪  downto 0);
167     alias children      : population_t (N_CHILDREN-1 downto 0) is population
        ↪  (N_POPULATION-1 downto N_PARENTS);
168     alias parents_loss  : loss_array (N_PARENTS-1 downto 0) is lossvalues (N_PARENTS-1
        ↪  downto 0);
169     alias children_loss : loss_array (N_CHILDREN-1 downto 0) is lossvalues (N_POPULATION-1
        ↪  downto N_PARENTS);
```

Der Aufbau des Optimierers gleicht einem Zustandsautomaten, dessen Zustände die verschiedenen Schritte der evolutionären Optimierungsstrategie umfassen. Die Zustände sind als VHDL-Typ in der Package-Definition des Optimierers festgelegt:

```
9     type GlobalStatus is (CONF, INIT, MUTATE, EVALUATE, SORT, FINISHED);
```

Der aktuelle Zustand wird in einem Signal `status` gespeichert und der Zustandsautomat über ein `case`-Konstrukt im taktflankengesteuerten Hauptprozess umgesetzt:

```
main : process(clk)
begin
        if rising_edge(clk) then
        -- ...
                case status is
                when CONF =>
                -- ...
                when INIT =>
                -- ...

                -- ...

                end case;
        end if;
end process;
```

Der Startzustand ist `CONF`, der als Konfigurationsmodus fungiert, über den weitere Parameter des EA eingestellt werden können, die, anders als $N$, $\mu$ und $\lambda$, keine Auswirkung auf das Layout der synthetisierten Schaltung haben. Über den 2 Bit breiten Eingang `config` können mit folgenden Werten verschiedene Aktionen ausgeführt werden:

`00` – Koeffizient der polynomiellen Zielfunktion wird am Eingang `data_in` abgelesen und an der nächsten freien Adresse, beginnend bei 0, in einer `ram`-Komponente gespeichert. Wurden alle $N(N+1)/2$ Koeffizienten übergeben, wird dies über eine 1 am Ausgang `full` angezeigt.

`01` – Eltern-Vektor wird am Eingang `data_in` abgelesen und an der nächsten freien Stelle in der Population abgespeichert. Wurden alle $\mu$ Eltern gesetzt, wird dies über eine 1 am Ausgang `full` angezeigt.

`10` – Initialer Seed wird am Eingang `data_in` abgelesen und gespeichert.

`11` – Ising-Modus wird aktiviert

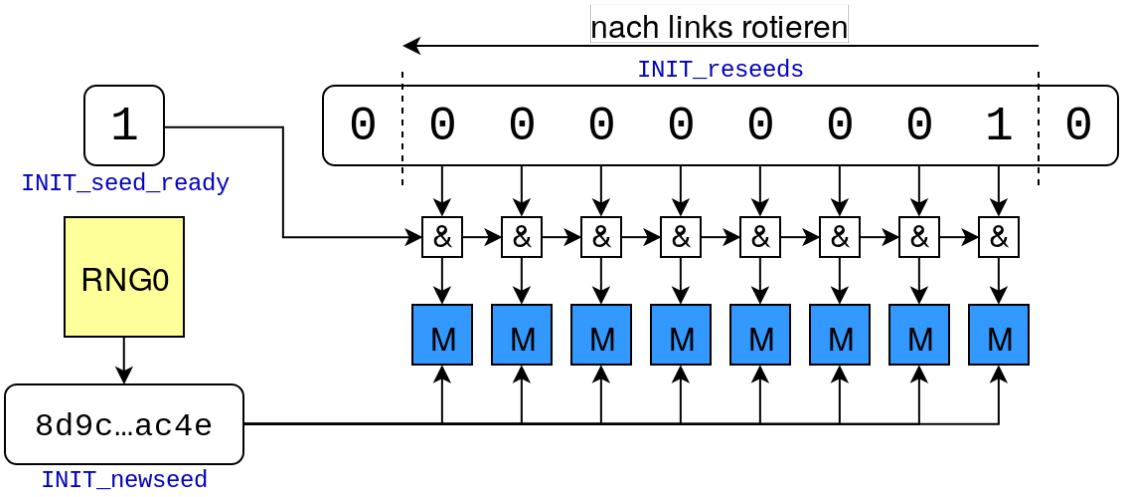



**Abbildung 4.3:** Initialisierungsschema der `mutator`-Komponenten (`M`); hat der Zufallsgenerator (`RNG0`) zwei neue Zufallszahlen erzeugt, wird derjenige Mutator initialisiert, auf dessen Index die 1 im Array `INIT_reseeds` steht, anschließend wird das Array nach links geschoben

Um die Eingabe der Daten synchronisieren zu können, werden die Werte von `config` und `data_in` erst übernommen, sobald der Eingang `data_ready` auf 1 gesetzt wird. Wurden die Daten erfolgreich gelesen und die entsprechenden Parameter gesetzt, zeigt der Ausgang `data_read` den Wert 1 an. Dies ermöglicht eine synchronisierte Kommunikation zwischen dem Optimierer und dem IO-Controller (siehe Abschnitt 4.2).

Durch eine 1 am Eingang `start` wird signalisiert, dass alle Einstellungen getroffen und die Optimierung begonnen werden kann. Der Optimierer wechselt dann in den Zustand `INIT`, in dem alle Zufallsgeneratorkomponenten mit Seeds initialisiert werden. Zu diesem Zweck besitzt `main` einen Zufallsgenerator `RNG0`, der entweder mit einem Seed, der im Konfigurationsmodus übergeben wurde, oder andernfalls mit einem Standard-Seed initialisiert ist. Mithilfe dieses Generators werden die Seeds für die restlichen Zufallsgeneratoren erzeugt.

Die einzigen Komponenten, die Zufallsgeneratoren besitzen und initialisiert werden müssen, sind die `mutator`-Komponenten (siehe Abschnitt 4.1.4). Ein Seed besteht aus 128 Bits, die Komponente `rng_xoroshiro128plus` liefert allerdings nur 64 Bits als Ausgabe. Deshalb muss jeder Seed in zwei Taktzyklen aus zwei Ausgaben von `RNG0` zusammengesetzt werden. Die `newseed`-Eingänge der Mutatoren sind mit dem Signal `INIT_newseed` verbunden, in dem der nächste Seed gespeichert wird. Damit nicht jeder Mutator gleichzeitig den Seed ausliest und so alle den gleichen Seed bekommen, ist der Eingang `reseed` jedes Mutators mit einem anderen Index eines Bitvektors verbunden, der an allen Stellen mit 0 und nur an erster Stelle mit 1 initialisiert ist. Zusätzlich ist der Bitvektor mit einem 1-Bit-Signal `INIT_seed_ready` verundet. Wird ein neuer Seed erzeugt, rotiert der Bitvektor nach links, sodass die 1 am `reseed`-Eingang des nächsten Mutators anliegt. Sobald

der Seed vollständig ist, wird `INIT_seed_ready` auf 1 gesetzt, sodass der entsprechende Mutator den Seed übernimmt. Der Ablauf der Initialisierung ist vereinfacht in Abb. 4.3 dargestellt. Auf diese Weise werden alle `mutator`-Komponenten nacheinander initialisiert, und ihre Ausgaben sind, bedingt durch die Funktionsweise des Zufallsgenerators, durch den initialen Seed von `RNG0` determiniert.

Wurden sämtliche Komponenten initialisiert, wird der Reset-Eingang der `mutator`-Komponenten auf 0 gesetzt und in den Zustand `MUTATE` gewechselt, in dem aus zufälligen Vektoren aus der Elternpopulation Nachkommen erzeugt werden. Alle Nachkommen werden gleichzeitig von je einem Mutator erzeugt, insgesamt sind auf dem Chip also $\lambda$ Mutator-Komponenten vorhanden. Jeder Mutator gibt über seinen Ausgang `parent_ix` zufällig vor, welches Elternindividuum mutiert werden soll. Dieses Individuum wird entsprechend an den Eingang `in_vec` angelegt, der Ausgang `out_vec` ist mit einem Index eines Vektors `MUTATE_res` verbunden. Ist die Verundung aller `valid`-Ausgänge gleich 1, dann ist die Mutierung beendet und `MUTATE_res` wird als neue Nachkommenpopulation übernommen. Der Reset-Eingang der Mutatoren wird erneut auf 1 gesetzt, sodass ihre internen Zufallsgeneratoren während der anderen Zustände nicht unnötig weiterlaufen.

Der Folgezustand ist `EVALUATE` – hier werden die soeben erzeugten Nachkommen ausgewertet und ihre Zielfunktionswerte in `lossvalues` gespeichert. Um dies zu parallelisieren, wird jeder Nachkomme von einem eigenen Evaluator ausgewertet, sodass der Chip $\lambda$ entsprechende Komponenten besitzt. Dies ist nur möglich, weil der Evaluator die Koeffizienten nicht selbst speichert und deshalb wenig Chipfläche verbraucht. Wie in Abschnitt 4.1.5 beschrieben, erwartet die `evaluator`-Komponente die Koeffizienten der Zielfunktion in korrekter Reihenfolge an ihrem Eingang `coef_in`. Die Koeffizienten sind in der RAM-Komponente gespeichert, weshalb die `coef_in`-Eingänge aller Evaluatoren mit dessen Ausgang `data_out` über das Signal `coef_data_out` verbunden ist. In jedem Taktzyklus wird nun der Koeffizient aus der nächsten Adresse, beginnend bei 0, aus dem RAM geladen und liegt dadurch an sämtlichen Evaluatoren an. Da in jedem Taktzyklus der nächste Koeffizient anliegt und auch der Evaluator selbst nur einen Taktzyklus benötigt, um einen Koeffizienten auszuwerten, ist dessen Eingang `coef_ready` während der `EVALUATE`-Phase durchgängig auf 1 gesetzt. Wurden alle Koeffizienten gelesen, ist die Berechnung der Zielfunktionswerte beendet, und die Ergebnisse, die von den Evaluatoren in `EVAL_results` gespeichert wurden, werden ins Array `lossvalues` übernommen.

Der nächste Zustand ist `SORT`, in dem die gesamte Population durch eine Instanz der `sorter`-Komponente nach Zielfunktionswerten sortiert wird. Da Sorter und Merger eine gewisse Zeit fürs Reset benötigen (siehe 4.1.6), wird zunächst gewartet, bis `valid`= 0 ist, bevor `reset` auf 0 gesetzt wird. Der Eingang `data` des Sortierers ist fest mit `lossvalues` verbunden, sodass der Optimierer nun nur noch aktiv warten muss, bis die Sortierung beendet ist. Ist dies der Fall, angezeigt durch eine 1 am Sorter-Ausgang `valid`, werden

sowohl `population` als auch `lossvalues` mithilfe des vom Sorter erzeugten Index-Arrays (`SORT_order`) sortiert, was innerhalb eines Taktzyklus' geschieht:

```
511                                 SELECT_PARENTS : for ix in 0 to N_PARENTS-1 loop
512                                     population(ix) <= population(SORT_order(ix));
513                                     lossvalues(ix) <= lossvalues(SORT_order(ix));
514                                 end loop SELECT_PARENTS;
```

Der letzte Schritt besteht darin, das restliche Budget `remaining_budget`, das zu Anfang mit dem Wert des generischen Parameters `BUDGET` initialisiert wurde, um 1 zu verringern, falls es einen positiven Wert hat. Ist `remaining_budget`$= 0$, wird in den Zustand `FINISHED` gewechselt, der nicht mehr verlassen werden kann und als einzige Aktion den Ausgang `done` des Optimierers auf 1 setzt. Andernfalls wird erneut in den Zustand `MUTATE` gewechselt und der Optimierer fährt mit der nächsten Iteration fort. Falls `remaining_budget`$< 0$, wird nichts am verbleibenden Budget verändert, und der Optimierer wechselt ebenfalls in `MUTATE` und läuft damit im Endlos-Modus.

Wurden im Konfigurationsmodus Elternvektoren eingegeben, weicht die Zustandsabfolge der ersten Iteration ein wenig von der zuvor beschriebenen ab: Würde die vorgegebene Elternpopulation an die entsprechende Stelle in `population` geschrieben und mit der `MUTATE`-Phase begonnen werden, würden zwar Nachkommen von dieser Elternpopulation erzeugt, die Eltern selbst aber in der Sortierphase ans Ende der Population sortiert und während der nächsten Mutation überschrieben werden, da die Eltern noch nicht mit der Zielfunktion ausgewertet wurden und somit noch den initialen Loss-Wert von 2147483647 besitzen. Um das Überschreiben der übergebenen Elternpopulation zu verhindern, wird sie während der Konfigurationsphase in die Nachkommenpopulation gespeichert und die erste Mutationsphase übersprungen, sodass als erstes die Zielfunktion auf den Elternindividuen ausgewertet wird. Während der anschließenden Sortierphase werden die Eltern automatisch in die Elternpopulation sortiert, da sie als einzige ausgewertet wurden und (in der Regel) einen Loss-Wert kleiner als 2147483647 haben. Ab der zweiten Iteration fährt der Optimierer mit der üblichen Abfolge fort.

Der Optimierer besitzt einige asynchrone Ausgänge, über die Informationen über den aktuellen Zustand abgelesen werden können: Befindet er sich im Konfigurationsmodus, zeigt dies eine 1 am Ausgang `config_mode` an, umgekehrt zeigt eine 0, dass die Optimierung gerade im Gange ist. Die Ausgänge `opt` und `opt_loss` zeigen jederzeit das aktuelle Optimum und dessen Zielfunktionswert an; sie sind fest mit den Werten an Index 0 von `population` bzw. `lossvalues` verbunden.

Wird ein Reset über eine 1 am Eingang `reset` ausgelöst, so werden Population und Zielfunktionswerte, Seed, Ising-Modus und sämtliche zustandsspezifischen Signale zurückgesetzt und der Optimierer springt wieder in den Konfigurationsmodus. Wichtig ist dabei, dass die in der RAM-Komponente gespeicherten Zielfunktionskoeffizienten *nicht* gelöscht

```
4  entity ram is
5      generic (
6          DATA_SIZE : integer := 8;
7          SIZE      : integer := 128
8      );
9      port (
10         clk      : in std_logic;
11         reset    : in std_logic;
12         address  : in integer range 0 to SIZE-1;
13         we       : in std_logic;
14         data_in  : in std_logic_vector(DATA_SIZE-1 downto 0);
15         data_out : out std_logic_vector(DATA_SIZE-1 downto 0)
16     );
17 end entity ram;
```

**Abbildung 4.4:** Entity-Deklaration der RAM-Komponente

werden und potenziell nach dem Reset weiterverwendet werden können. Werden nach einem Reset neue Koeffizienten geladen, ist darauf zu achten, *alle* $N(N+1)/2$ Koeffizienten einzugeben, damit aus etwaigen vorherigen Optimierungsläufen gespeicherte Koeffizienten überschrieben werden.

### 4.1.2 RAM

Die Datei `ram.vhd` definiert eine Entity namens `ram` (siehe Abb. 4.4), deren Architecture einen einfachen synchronen Speicher implementiert, der zwei Generics besitzt: `DATA_SIZE` legt die Bit-Breite eines einzelnen adressierbaren Worts fest, `SIZE` die Anzahl adressierbarer Wörter. Die Gesamtgröße des Speichers ist demnach `SIZE` · `DATA_SIZE`. Die Entity besitzt einen Eingang `address`, über den mit einem Index $0 \leq i <$ `DATA_SIZE` jedes gespeicherte Wort addressiert werden kann. Der Eingang `we` (*write enable*) bestimmt, ob ein Wort gelesen oder geschrieben wird; Ist `we` $= 0$ und wird eine Adresse $i$ an `address` angelegt, wird das an dieser Stelle gespeicherte Wort $w_i$ zur nächsten steigenden Taktflanke auf den Ausgang `data_out` gelegt. Ist `we` $= 1$, dann wird $w_i$ mit dem am Eingang `data_in` anliegenden Wort überschrieben und liegt zur übernächsten steigenden Taktflanke am Ausgang `data_out` an. Der `reset`-Eingang setzt `data_out` auf den Nullvektor.

Die Implementierung orientiert sich stark am Code-Beispiel für *Single-Port Block RAM* aus dem Vivado-Benutzerhandbuch[1]. Ein wichtiges Detail liegt in der expliziten Verwendung von Block-RAM: `ram` wird im Rahmen des Optimierers zur Speicherung der Zielfunktionskoeffizienten verwendet, deren Anzahl quadratisch zur Dimension $N$ wächst. Durch die Speicherung im Block-RAM werden dafür keine LUTs als Speicher verwendet, sodass die Chipfläche insgesamt effizienter genutzt wird (siehe Abschnitt 5.2.2).

[1] Vivado Design Suite User Guide – *Synthesis*
`https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug901-vivado-synthesis.pdf`

### 4.1.3 Zufallsgenerator

Eine weitere Basiskomponente ist der Pseudozufallszahlengenerator, der in der VHDL-Datei `rng_xoroshiro128plus.vhd` definiert ist. Verwendet wird der in Abschnitt 2.4 erwähnte `xoroshiro128plus`-Algorithmus; die Implementierung stammt von Joris van Rantwijk und unterliegt der *GNU Lesser General Public License*[2].

Die Komponente besitzt einen Generic `init_seed`, einen 128-Bit breiten Seed, der ungleich 0 sein muss. Nach einem Taktzyklus ist der Ausgang `out_valid` = 1 und zeigt an, dass am Ausgang `out_data` ein 64-Bit breiter Zufallsbitvektor abgelesen werden kann. Über eine 1 am Eingang `out_ready` kann ein neuer Zufallswert für den nächsten Taktzyklus angefordert werden. Ist der Eingang `reset` = 1, wird der Startzustand wiederhergestellt; da der Zufallsgenerator deterministisch ist, ist die Folge von Zufallszahlen nach dem Reset identisch zur vorherigen.

Ein nützliches Feature ist die Möglichkeit, einen neuen Seed zu setzen und die Komponente somit zur Laufzeit zu initialisieren. Dazu muss der neue Seed am Eingang `newseed` angelegt werden, der bei einer 1 am Eingang `reseed` zur nächsten steigenden Taktflanke übernommen wird. Einen weiteren Taktzyklus später liegt der erste aus dem neuen Seed gewonnene Zufallswert an `data_out` an. Dieses Feature vereinfacht die Verwendung mehrerer Instanzen des Zufallsgenerators drastisch, da andernfalls für jede Instanz ein eigener Seed als konstanter Parameter übergeben werden müsste. Durch die *Reseed*-Funktion kann ein Zufallsgenerator die Seeds für alle anderen Generatoren erzeugen, sodass nur der Seed dieses übergeordneten Generators "von außen" gesetzt werden muss.

### 4.1.4 Mutator

Der *Mutator* kapselt die Mutation eines einzelnen Bit-Vektors. Als Eingabe wird am Eingang `in_vec` ein Bit-Vektor angelegt. Durch setzen des Eingangs `ready` = 1 wird signalisiert, dass der gegebene Vektor mutiert werden soll.

Jeder Mutator besitzt eine interne Zufallsgeneratorkomponente `random_generator`, der eine Instanz von `rng_xoroshiro128plus` ist und vor der ersten Mutation manuell initialisiert werden muss. Dafür besitzt der Mutator zwei Eingänge `in_reseed` und `in_newseed`, die direkt mit den entsprechenden Eingängen `reseed` und `newseed` der Zufallsgeneratorinstanz verbunden werden. Um zu verdeutlichen, dass der interne Generator zur Laufzeit mit einem Seed initialisiert werden muss, ist der generische `init_seed` fest auf den Nullvektor gesetzt – ohne eine manuelle Initialisierung erzeugt der Generator so nur Nullen. Da es daher auch nicht sinnvoll ist, den Generator auf seinen initialen Seed zurückzusetzen, ist der `reset`-Eingang fest auf 0 gesetzt. Der Eingang `out_ready` ist mit dem Mutator-Eingang `ready` verbunden, sodass immer genau denn, wenn ein Vektor mutiert werden soll, neue Zufallszahlen erzeugt werden.

[2] `https://github.com/jorisvr/vhdl_prng/blob/master/rtl/rng_xoroshiro128plus.vhdl`

```
8  entity mutator is
9      Generic (
10         DIM       : integer;
11         N_PARENTS : integer
12     );
13     Port (
14         clk        : in  std_logic;
15         reset      : in  std_logic;
16         in_reseed  : in  std_logic;
17         in_newseed : in  std_logic_vector(127 downto 0);
18         in_vec     : in  std_logic_vector(DIM-1 downto 0);
19         ready      : in  std_logic;                          -- '1' <=> out_vec read, new
           ↪  in_vec set
20         parent_ix  : out index_t;
21         valid      : out std_logic;
22         out_vec    : out std_logic_vector(DIM-1 downto 0)
23     );
24 end mutator;
```

**Abbildung 4.5:** Entity-Deklaration des Mutators

Der 64-Bit breite Zufallsvektor, den `random_generator` erzeugt, wird aufgeteilt in zwei 32-Bit-Vektoren, die jeweils als vorzeichenbehaftete ganze Zahlen interpretiert werden:

```
71     -- split 64 bit random number into two 32 bit integers
72     rand_int0 <= to_integer(signed(rand_data(31 downto 0)));
73     rand_int1 <= to_integer(signed(rand_data(63 downto 32)));
```

Während `ready` = 1 ist, wird nun mit einem Index `next_ix`, der initial auf 0 gesetzt wird, über den Eingangsvektor `in_vec` iteriert. Um zu entscheiden, ob ein Bit invertiert werden soll, wird die Zufallszahl `rand_int0` (bzw. `rand_int1`) modulo $N$ gerechnet. Ist das Ergebnis 0, wird die Negation an den am entsprechenden Index des Ausgangs `out_vec` angelegt, andernfalls wird das Originalbit unverändert übernommen. Da `rand_int0` im ganzzahligen Bereich $[-2^{31}, 2^{31} - 1]$ als annähernd gleichverteilt angenommen wird, ist `rand_int0` mod $N$ ebenfalls annähernd gleichverteilt im Bereich $[0, N - 1]$. Jedes Bit wird somit in etwa mit Wahrscheinlichkeit $1/N$ invertiert.

Da der Zufallsgenerator pro Taktzyklus *zwei* zufällige 32-Bit-Zahlen erzeugt, können zwei Bits auf einmal mutiert werden. Dazu werden die Bits an den Indizes `next_ix` und `next_ix`+1 parallel ausgewertet und `next_ix` anschließend um 2 erhöht. Dieses Vorgehen hat keine Auswirkung auf die asymptotische Laufzeit, halbiert aber dennoch die Anzahl benötigter Taktzyklen und nutzt die generierten Zufallswerte effizienter aus. Verallgemeinert könnten die 64 zufälligen Bits in $N$ Wörter $w$ der Länge $l_N = \lfloor 64/N \rfloor$ aufgeteilt werden. Nähert sich die Anzahl mit $l_N$ Bits darstellbarer Zahlen ($2^{l_N}$) allerdings $N$, entfernt sich $w$ mod $N$ immer weiter von einer Gleichverteilung auf $[0, N - 1]$, wenn $N$ kein Teiler von $2^{l_N}$ ist. Bereits ab $N \geq 16$ ist schon $2^{l_N} \leq N$, da für nicht-triviale Optimierungsprobleme aber $N \gg 16$ ist, lohnt sich der Implementierungsaufwand für variables $l_N$ nicht.

```
14 entity evaluator is
15     generic (
16         DIM        : integer
17     );
18     port (
19         clk      : in std_logic;
20         reset    : in std_logic;
21         isg_mode : in std_logic;
22 
23         vect_in    : in std_logic_vector (DIM-1 downto 0);
24         coef_ready : in std_logic;
25         coef_in    : in loss_t;
26 
27         coef_read : out std_logic;
28         valid     : out std_logic;
29         result    : out loss_t
30     );
31 end entity evaluator;
```

**Abbildung 4.6:** Entity-Deklaration des Evaluators

Sobald `next_ix+1` größer als der höchste Index $(N-1)$ ist, ist die Mutierung des aktuellen Bit-Vektors beendet. Der Ausgang `valid` wird auf 1 gesetzt, um anzuzeigen, dass der fertig mutierte Bit-Vektor am Ausgang `out_vec` anliegt und gelesen werden kann. Gleichzeitig wird ein weiterer Ausgang `parent_ix` auf einen zufälligen Wert zwischen 0 und $\mu - 1$ gesetzt:

```
if next_ix+1 < DIM then
    -- ...
else
    parent_ix <= rand_int1 mod N_PARENTS;
    valid <= '1';
end if;
```

Dieser zusätzliche Ausgang wird später als Index verwendet, um einen zufälligen Bit-Vektor aus der Elternpopulation auszuwählen, der mutiert werden soll. Der initiale Wert dieses Index' wird während des ersten *Reseeds* aus den oberen 32 Bits von `in_newseed` gewonnen, welches selbst eine Zufallszahl ist:

```
78             if in_reseed = '1' then
79                 -- take initial parent index directly from the random seed
80                 -- (which itself is a random number) during initialization phase
81                 parent_ix <= to_integer(signed(in_newseed(127 downto 96))) mod N_PARENTS;
82             end if;
```

Über den Eingang `reset` wird der Mutator zurückgesetzt, indem `next_ix` und `valid` auf ihren Anfangswert 0 zurückgesetzt werden.

### 4.1.5 Evaluator

Der *Evaluator* implementiert die Zielfunktionsauswertung auf einem einzelnen Bit-Vektor. Die Komponente besitzt einen Eingang `vect_in`, über den ein auszuwertender Bit-Vektor übergeben wird. Statt sämtliche $N(N+1)/2$ Koeffizienten selbst zu speichern, besitzt der Evaluator außerdem einen Eingang `coef_in`, über den die auszuwertenden Koeffizienten während der Berechnung der Reihe nach angelegt werden müssen. Kann der nächste gelesen werden, wird dies über den Bit-Eingang `coef_ready` angezeigt. Wurde der Koeffizient erfolgreich gelesen und verwertet, zeigt der Evaluator dies über den Bit-Ausgang `coef_read` an. Durch diese beidseitige Bestätigung ist es möglich, den Evaluator auf eine kommunizierende Komponente warten zu lassen und umgekehrt, was es beispielsweise ermöglicht, die Koeffizienten über mehrere Taktzyklen erst zu berechnen und dann direkt an den Evaluator weiterzureichen. Wurden alle Koeffizienten gelesen und ausgewertet, liegt der der Zielfunktionswert am Ausgang `result` an und kann ausgelesen werden, solange der Ausgang `valid` den Wert 1 hat. Durch Anlegen einer 1 an den `reset`-Eingang wird der Evaluator zurückgesetzt.

Da der Evaluator den Bit-Vektor `vect_in` nicht puffert, muss dieser während der gesamten Berechnung am Eingang anliegen, um sicherzustellen, dass das Ergebnis korrekt berechnet wird. Wichtig ist auch die Reihenfolge, in der die Koeffizienten $\beta_{ij}$ an `coef_in` angelegt werden müssen: Sie folgt dem Schema, das die Indizes der Summen in Gleichung (2.2) vorgeben. Da die Indizierung bei 0 beginnt, ist die Reihenfolge der Indizes folglich

$$((i,j))_k = ((0,0),(1,0),(1,1),(2,0),(2,1),(2,2),(3,0),\ldots,(N-1,N-2),(N-1,N-1))$$

mit $k \in \{1,\ldots,N(N+1)/2\}$.

Obwohl das Zielfunktionspolynom Multiplikation enthält, muss diese auf dem FPGA nicht als solche berechnet werden: Da $x_i \in \mathbb{B}$, kann die Berechnung des Funktionswerts als bedingte Summierung der Koeffizienten betrachtet werden. Nur wenn $x_i = x_j = 1$, wird $\beta_{ij}$ zur Gesamtsumme addiert. Der Evaluator besitzt deshalb zwei Indizes `i` und `j`, die mit 0 initialisiert werden. Wenn der Wert von `vect_in` sowohl an Index `i` als auch an Index `j` gleich 1 ist, wird der momentan anliegende Koeffizient zur Summe `sig_result` addiert, die ebenfalls zu Anfang mit 0 initialisiert wird:

```
82                      if (vect_in(i) and vect_in(j)) = '1' then
83                          sig_result <= sig_result + coef_in;
84                      end if;
```

Ist `i`=`j`, wird `i` um eins erhöht und `j` auf 0 gesetzt, andernfalls wird `j` erhöht. Ist `i`=`j`$= N-1$, ist die Berechnung beendet und `valid` wird auf 1 gesetzt.

Zur Umsetzung des in Abschnitt 3.1.2 beschriebenen Ising-Modus' genügt es, die Summierung der Koeffizienten zu ändern, sodass 0 als $-1$ interpretiert wird: Ist $x_i \oplus x_j = 1$, dann sind $x_i$ und $x_j$ unterschiedlich und haben deshalb unter der Interpretation $0 \rightarrow -1$ verschiedene Vorzeichen. Da der entsprechende Koeffizient $\beta_{ij}$ in diesem Fall negativ in

die Summe einfließt, wird `coef_in` von `sig_result` abgezogen, andernfalls wie zuvor addiert. Ist `i=j`, so wird der Koeffizient als ein linearer Koeffizient $\alpha_i$ interpretiert (siehe Abschnitt 2.2.3), der entsprechend dem Vorzeichen von $x_i$ in die Summe einfließt, daher muss dieser Fall gesondert betrachtet werden, denn $i = j \Rightarrow x_i \oplus x_j = 0$. Der lineare Koeffizient $\beta_{ii}$ entfällt, da er bezüglich der Minimierung der Zielfunktion konstant ist.

```
68                      if i = j then
69                          if vect_in(i) = '0' then
70                              sig_result <= sig_result - coef_in;
71                          else
72                              sig_result <= sig_result + coef_in;
73                          end if;
74                      else
75                          if (vect_in(i) xor vect_in(j)) = '1' then
76                              sig_result <= sig_result - coef_in;
77                          else
78                              sig_result <= sig_result + coef_in;
79                          end if;
80                      end if;
```

Der Ising-Modus wird über einen Eingang `isg_mode` (für *Ising Spin Glass mode*) aktiviert; während einer Berechnung muss auch dieser Eingang konstant gehalten werden, damit der Modus nicht während der Summierung wechselt und das Ergebnis so verfälscht wird.

### 4.1.6 Sorter und Merger

Zur Selektion der neuen Elternpopulation werden die besten $\mu$ Individuen aus der aktuellen Gesamtpopulation ausgewählt. Zu diesem Zweck muss die Population nach Zielfunktionswerten sortiert werden. Diese Aufgabe erledigt die Komponente `sorter`, die *Odd-Even-Mergesort* in Form eines Sortiernetzwerks implementiert (siehe Abschnitt 3.1.3).

```
13  entity sorter is
14      generic (
15          N : positive := 1
16      );
17      port (
18          clk   : in std_logic;
19          reset : in std_logic;
20          data  : in loss_array((2**N)-1 downto 0);
21          valid : out std_logic;
22          order : out index_array((2**N)-1 downto 0)
23      );
24  end entity;
```

Da Odd-Even-Mergesort nur auf Arrays funktioniert, deren Länge eine Zweierpotenz ist, besitzt die `sorter`-Komponente einen generischen Parameter `N`, der den Exponenten der Zweierpotenz darstellt. An den Eingang `data` wird das zu sortierende Array der Länge $2^N$ angelegt. Der Ausgang `valid` ist 1, sobald die Sortierung fertig ist und das Ergebnis am Ausgang `order` abgelesen werden kann. Eine Besonderheit der Implementierung ist, dass

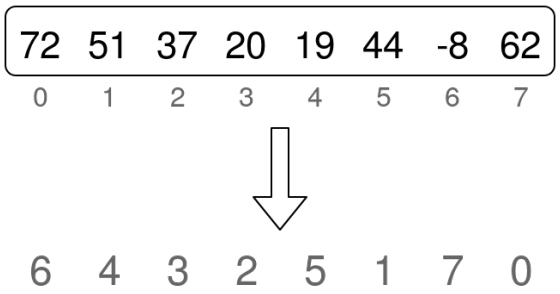



**Abbildung 4.7:** Beispiel für eine Index-Sortierung; das Ergebnis sind nicht die sortierten Werte selbst, sondern ihre sortierten Positionen im Original-Array

nicht die sortierten Daten selbst, sondern eine sortiertes Index-Array ausgegeben wird, ähnlich zur Python-Funktion `numpy.argsort` (siehe 4.7). Der Vorteil daran ist, dass mit dem Ergebnis eines Sortierdurchlauf sowohl die Individuen selbst als auch ihre zugehörigen Zielfunktionswerte sortiert werden können, denn diese liegen in zwei getrennten Arrays, korrespondieren aber durch ihre Indizes miteinander.

Gemäß der in Abschnitt 3.1.3 beschriebenen Funktionsweise des Sortiernetzwerks müssen die Daten zunächst rekursiv sortiert und anschließend mithilfe der `merger`-Komponente zusammengefügt werden. VHDL erlaubt die Verwendung einer Entity in ihrer eigenen Architektur-Definition, sodass die rekursive Struktur des Sortiernetzes physisch umgesetzt werden kann. Ist die Länge des Arrays gleich 2, sind beide Hälften bereits sortiert. Ansonsten werden zwei weitere `sorter` mit dem Parameter `N` ← `N-1` instanziiert, einer für je eine Hälfte des gesamten Datenarrays. `MSB` (für *Most Significant Bit*) und `HALF_MSB` sind Konstanten für den höchsten Index des gesamten Arrays bzw. dessen unterer Hälfte. Die Ausgaben der Komponenten (`valid` und `order`) werden in entsprechenden Signals gespeichert.

```
SORT_TWO : if N = 1 generate

        -- ...

end generate SORT_TWO;

SORT_MANY : if N > 1 generate

         -- sub-sorters for the lower and upper half
        LOWER_HALF : sorter
                generic map (N-1)
                port map (clk, reset, data(HALF_MSB downto 0), lower_half_valid,
                ↪  lower_half_order);

        UPPER_HALF : sorter
                generic map (N-1)
                port map (clk, reset, data(MSB downto HALF_MSB+1),
                ↪  upper_half_valid, upper_half_order);

        -- ...
```

```
    end generate SORT_MANY;
```

Im Fall `N` = 1 bestehen die obere und untere Hälfte der Daten nur aus jeweils einem Element und sind deshalb schon sortiert. Die Signals `lower_half_valid`, `lower_half_order` etc. werden deshalb manuell gesetzt und anschließend die `merger`-Komponente aktiviert, indem ihr Reset-Eingang über das Signal `merge_reset` auf 0 gesetzt wird. Die gesamte Sortierung ist beendet, sobald die `merger`-Komponente beide Hälften zusammengefügt hat, deshalb wird `valid` auf `merge_valid` gesetzt. Der Prozess wird in jedem Taktzyklus durchlaufen und aktualisiert dadurch `valid`.

```
110     process (clk) begin
111         if rising_edge(clk) then
112             if reset = '0' then
113                 -- no recursive sorting necessary
114                 lower_half_valid    <= '1';
115                 lower_half_order(0) <= 0;
116                 upper_half_valid    <= '1';
117                 upper_half_order(0) <= 0;
118
119                 merge_reset <= '0';
120                 valid       <= merge_valid;
121             else
122                 merge_reset <= '1';
123                 valid       <= merge_valid;
124             end if;
125         end if;
126     end process;
```

Der Fall `N` > 1 funktioniert analog, nur dass `lower_half_valid` etc. nicht manuell sondern durch die `sorter`-Unterkomponenten gesetzt werden. Die schematische Funktionsweise ist in Abb. 4.8 dargestellt, dabei sind die entsprechenden Signal-Namen blau dargestellt. Sobald beide Hälften sortiert sind, wird wie zuvor die `merger`-Komponente aktiviert und `valid` auf `merge_valid` gesetzt.

```
141     process (clk) begin
142         if rising_edge(clk) then
143             if reset = '0' then
144                 -- wait for sub-sorters to finish
145                 if (lower_half_valid = '1') and (upper_half_valid = '1') then
146                     merge_reset <= '0';
147                 end if;
148
149                 valid <= merge_valid;
150             else
151                 merge_reset <= '1';
152                 valid       <= merge_valid;
153             end if;
154         end if;
155     end process;
```

Die Sorter für die untere und obere Hälfte erzeugen je ein sortiertes Index-Array mit Indizes im Bereich $[0, 2^{N-1} - 1]$. Die Hälften werden zusammengefügt, indem die untere

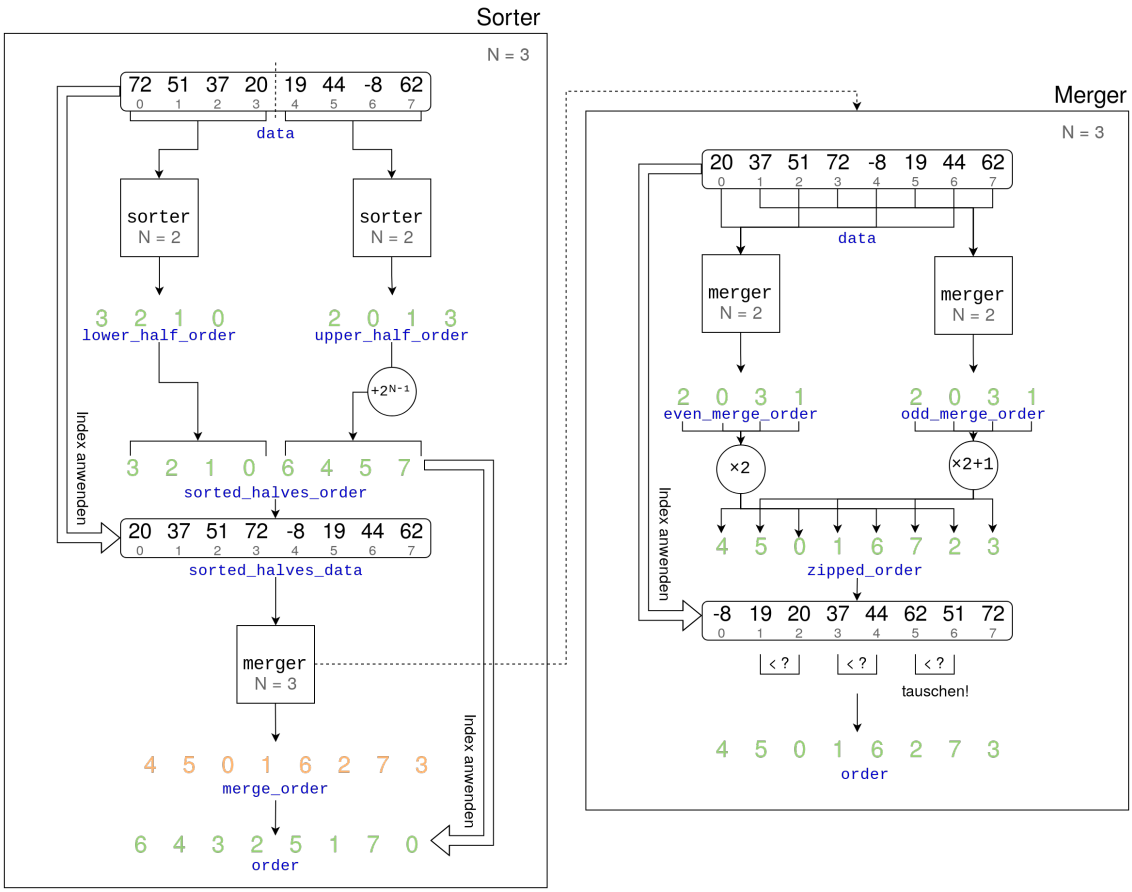



**Abbildung 4.8:** Schema der `sorter`- und `merger`-Komponente für `N`$> 1$; beide Komponenten enthalten rekursiv weitere Instanzen des `merger`s mit kleinerem Parameter $N$, bis mit $N = 1$ der Basisfall erreicht ist

Hälfte übernommen und zur oberen $2^{N-1}$ addiert wird, um den Offset zu den Originalindizes zu korrigieren. Das resultierende zusammengesetzte Index-Array wird nun auf die Originaldaten angewendet und an die `merger`-Komponente übergeben, die beide Hälften zusammenfügt und ein sortiertes Index-Array zurückgibt. Da die Eingabe des Mergers allerdings nicht die Originaldaten sondern die sortierten Hälften waren, beziehen sich die Indizes auf `sorted_halves_data`, deren Reihenfolge sich wiederum aus der Anwendung von `sorted_halves_order` auf die Originaldaten ergibt. Um nun ein komplett sortiertes Index-Array zu erhalten, das sich auf die Originaldaten bezieht, muss `merge_order` auf `sorted_halves_order` angewendet werden.

Die Entity-Definition des Mergers ist identisch zu der des Sorters: Er erhält ebenfalls ein Array von Daten über einen Eingang `data`, hier müssen diese Daten aber bereits durch den Sorter vorsortiert sein. Ist z. B. `N` $= 3$, dann müssen die vierelementigen Unterarrays `data(3 downto 0)` und `data(7 downto 4)` aufsteigend sortiert sein, und die Aufgabe des Mergers ist, diese beiden Hälften zusammenzufügen.

Im Fall `N` $= 1$ sind die obere und untere Hälfte einelementig und damit bereits sortiert, sodass das gesamte Array mit genau einem Vergleich sortiert werden kann. Ist hingegen `N` $> 1$, sieht der Odd-Even-Mergesort-Algorithmus vor, das Array in zwei Arrays mit Elementen geraden bzw. ungeraden Indexes aufzuteilen. Die entstehenden Unterarrays halber Größe werden rekursiv mit je einer Merger-Komponente sortiert. Ähnlich wie beim Sorter bezieht sich das Ergebnis der Merger auf die Arrays mit geraden bzw. ungeraden Elementen. Daher müssen im nächsten Schritt, in dem die Indexarrays wieder zu einem Array zusammengefügt werden, die Indizes entsprechend umgerechnet werden. Dazu dient die Funktion `zip_indices`:

```
39      function zip_indices(even : index_array; odd : index_array)
40          return index_array is
41          constant slice_msb : integer := even'high; -- must always be = odd'high, too!
42          variable res       : index_array(2*slice_msb+1 downto 0);
43      begin
44          assert even'high = odd'high report "Zipped indices of different lengths!";
45          for i in 0 to slice_msb loop
46              res(2*i)   := 2*even(i);
47              res(2*i+1) := 2*odd(i)+1;
48          end loop;
49          return res;
50      end zip_indices;
```

Haben beide Merger die Unterarrays sortiert, müssen nun noch alle Elemente mit ungeradem Index mit dem jeweiligen Element mit nächsthöherem Index (falls vorhanden) verglichen und, wenn nötig, getauscht werden. Das erste und das letzte Element des Arrays sind bereits korrekt und werden sofort ins Ergebnis übernommen.

```
124                     order(0)   <= zipped_order(0);
125                     order(MSB) <= zipped_order(MSB);
126
127                     -- make remaining comparisions [i:i+1] for odd i < MSB
```

```
128                     for i in 0 to 2**(N-1)-2 loop
129                         if data(zipped_order(2*i+1)) > data(zipped_order(2*i+2)) then
130                             -- swap
131                             order(2*i+1) <= zipped_order(2*i+2);
132                             order(2*i+2) <= zipped_order(2*i+1);
133                         else
134                             -- keep
135                             order(2*i+1) <= zipped_order(2*i+1);
136                             order(2*i+2) <= zipped_order(2*i+2);
137                         end if;
138                     end loop;
139                     valid <= '1';
```

Da die `for-loop`-Schleife in VHDL "abgewickelt" wird, d. h. sämtliche Iterationen durch den Compiler aneinandergereiht werden, geschieht dieser letzte Schritt für alle ungeraden Elemente gleichzeitig, sodass dabei sofort `valid` auf 1 gesetzt wird, da im nächsten Taktzyklus das korrekte Ergebnis anliegt.

Um den Sortierer zurückzusetzen, muss `reset` auf 1 gesetzt werden. Eine Eigenheit der vorliegenden Implementierung ist, dass das Reset-Signal über mehrere Taktzyklen zu den Unterkomponenten propagiert werden muss, was im rekursiven Aufbau und dem synchronen Reset-Mechanismus begründet liegt. Der Wert 1 muss an `reset` angelegt bleiben, bis der Sortierer vollständig zurückgesetzt ist. Der Sortierer zeigt dies an, indem erst dann der Ausgang `valid` auf 0 zurückfällt. Besonders zu beachten ist, dass das Ergebnis `order` ungültig wird, während `reset` aktiv ist, obwohl zunächst weiterhin `valid` den Wert 1 anzeigt.

## 4.2 IO-Controller

Der EA-Optimierer ist darauf ausgelegt, als integrierte Komponente in größeren Schaltungen verwendet werden zu können. Für die Kommunikation mit einem Benutzer wird deshalb eine gesonderte Komponente benötigt, die als Schnittstelle zwischen Optimierer und Benutzer fungiert. Solch eine Schnittstelle muss einerseits Daten wie Zielfunktionskoeffizienten und Seed an den Optimierer übermitteln und andererseits dessen Ausgaben ablesen und zum Benutzer schicken können. Diesem Zweck dient die Komponente `EA_IO_CTRL`, die mithilfe des UART-Protokolls über eine serielle Schnittstelle Daten senden und empfangen kann.

Die für die Implementierung des IO-Controllers verwendeten UART-Komponenten `UART_RX.vhd` und `UART_TX.vhd` stammen von der Webseite Nandland[3] und bieten eine einfache Schnittstelle, einzelne Bytes zu senden und zu empfangen, dabei ist die Anzahl Taktzyklen pro Übertragung eines Bits ein generischer Parameter `CLKS_PER_BIT`, der in Zusammenspiel mit der globalen Taktrate die Baudrate der UART-Übertragung bestimmt. Der IO-Controller besitzt je einen solchen UART-Empfänger und einen Sender; seine Funktion

[3] `https://www.nandland.com/vhdl/modules/module-uart-serial-port-rs232.html`

```
15 entity EA_IO_CTRL is
16 Generic (
17     CLKS_PER_BIT : integer := 115;
18     DIM          : integer := 8;
19     N_PARENTS    : integer := 4;
20     N_CHILDREN   : integer := 12;
21     COEF_WIDTH   : integer := 32;
22     DATA_WIDTH   : integer := 128 -- = max(DIM, 128)
23 );
24 Port (
25     clk       : in std_logic;
26     serial_in : in std_logic;
27
28     opt_in      : in std_logic_vector(DIM-1 downto 0);
29     opt_loss_in : in std_logic_vector(31 downto 0);
30
31     serial_out       : out std_logic;
32     reset_out        : out std_logic;
33     config_out       : out std_logic_vector(1 downto 0);
34     data_ready_out   : out std_logic;
35     data_out         : out std_logic_vector(DATA_WIDTH-1 downto 0);
36     start_out        : out std_logic
37 );
38 end EA_IO_CTRL;
```

**Abbildung 4.9:** Entity-Deklaration von `EA_IO_CTRL`

besteht hauptsächlich darin, die seriell empfangenen Daten zu puffern, aufzuteilen und in korrekte Belegungen der Eingänge `reset`, `config`, `data_ready`, `data_in` und `start` des Optimierers umzuwandeln, außerdem die Ausgaben des Optimierers sowie seine generischen Parameter in Bytes aufzuteilen und gepuffert über den seriellen Ausgang zu versenden.

Die Entity-Deklaration (siehe Abb. 4.9) besitzt Ein- und Ausgänge, die mit denen des Optimierers korrespondieren (Name des Optimierer-Ports mit Endung `_in` bzw. `_out`, vergleiche Abb. 4.2) und diesen so mit den benötigten Eingaben versorgen und das aktuelle Optimum überwachen. Auch die meisten Generics haben eine Entsprechung, damit die Konfigurationsdaten des Optimierers unabhängig von diesem abgefragt werden können. Zusätzlich besitzt der Optimierer zwei serielle Ports, `serial_in` und `serial_out`, die mit entsprechenden UART-Pins des FPGAs verbunden werden müssen.

Der IO-Controller ist, ähnlich wie der Optimierer, ein Zustandsautomat, der über einzelne Bytes in Form von ASCII-Zeichen gesteuert wird, die über den seriellen Eingangsport gesendet werden. Jedes Byte steht für einen anderen Befehl, der von einer festgelegten Anzahl Daten-Bytes gefolgt werden muss. Wurden das Befehls-Byte und etwaige Daten-Bytes gelesen, führt der IO-Controller die gewünschte Aktion am Optimierer aus und kann wiederum mit einer bestimmten Anzahl Bytes antworten, die über den seriellen Ausgang zurückgeschickt werden. Tabelle 4.1 zeigt eine vollständige Liste aller Steuerbefehle mit ihren zugehörigen Daten- und Rückgabe-Bytes, dabei bezeichnet $B_\beta$ die Anzahl Bytes zur Kodierung eines Zielfunktionskoeffizienten und $B_N = \lceil N/8 \rceil$ analog die Anzahl Bytes pro

| *Befehl* | *(Hex)* | *Daten (B)* | *Rückgabe (B)* | *Bedeutung* |
|---|---|---|---|---|
| `r` | `0x72` | 0 | 0 | Reset |
| `x` | `0x87` | 16 | 0 | Seed setzen |
| `c` | `0x63` | $B_\beta$ | 0 | Koeffizient übergeben |
| `p` | `0x70` | $B_N$ | 0 | Elternindividuum übergeben |
| `i` | `0x69` | 0 | 0 | Ising-Modus aktivieren |
| `s` | `0x73` | 0 | 0 | Optimierung starten |
| `o` | `0x6f` | 0 | $4 + B_N$ | Optimum abfragen |
| `k` | `0x6b` | 0 | 23 | Konfiguration abfragen |

**Tabelle 4.1:** Steuer-Bytes des IO-Controllers mit jeweiliger Anzahl Daten- und Rückgabe-Bytes

Bitvektor. Die Kommandos `r`, `i` und `s` erwarten keine Daten-Bytes und erzeugen auch keine Rückgaben: `r` löst ein Reset des Optimierers aus, `s` startet die Optimierung und `i` aktiviert den Ising-Modus.

Daten- und Rückgabe-Bytes aller anderen Kommandos werden grundsätzlich in *Little-Endian*-Reihenfolge übermittelt, d. h. das niederwertigste Byte wird zuerst übermittelt. Eine Übersicht über den Aufbau der Byte-Strings, die an den IO-Controller gesendet bzw. von diesem empfangen werden, ist in Abb. 4.10 dargestellt.

Das Kommando `c` kündigt einen einzelnen Koeffizienten an, der im Anschluss in Zweierkomplementdarstellung übergeben wird. Ist $b_\beta$ die Anzahl Bits zur Darstellung eines Koeffizienten, die dem generischen Parameter `COEF_WIDTH` des Optimierers entspricht, erwartet der IO-Controller $B_\beta = \lceil b_\beta/8 \rceil$ Bytes. Der Koeffizient muss daher am höchstwertigen Ende mit beliebigen Bits auf eine Länge von $8B_\beta$ Bits aufgefüllt werden, die bei der Übertragung an den Optimierer ignoriert werden. Analog kündigt `p` einen Eltern-Bitvektor an, der am oberen Ende auf $8B_N$ Bits aufgefüllt werden muss.

Mit dem Kommando `x` kann ein Seed gesetzt werden; ein Seed besteht immer aus 128 Bits, sodass auf `x` genau 16 Bytes folgen. Über das Kommando `o` werden die Ausgänge `opt` und `opt_loss` des Optimierers abgelesen und über den serielle Ausgang gesendet. Die Zielfunktionswerte bestehen immer aus 32 Bits in Zweierkomplementdarstellung und werden als 4 Bytes zuerst gesendet, gefolgt vom Bitvektor, der am oberen Ende mit Nullen auf $B_N$ Bytes aufgefüllt wird.

Über das Kommando `k` können Information über die aktuelle Konfiguration des Optimierers angefordert werden: Der Rückgabe besteht aus 23 Bytes, die fünf Parameter vorzeichenlos kodieren.

| *Länge* | 2B | 2B | 2B | 1B | 16B |
|---|---|---|---|---|---|
| *Index* | 0–1 | 2–3 | 4–5 | 6 | 7–22 |
| *Parameter* | $N$ | $\mu$ | $\lambda$ | $b_\beta$ | Seed |

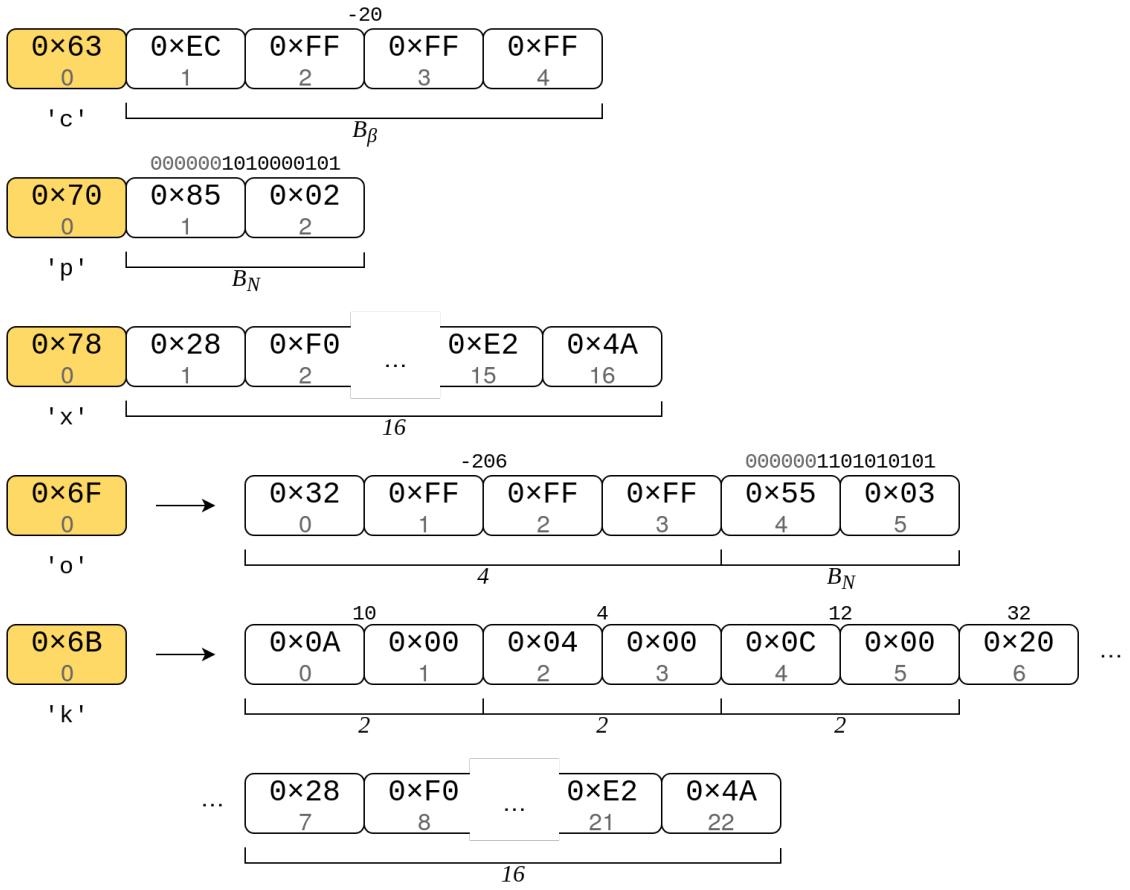



**Abbildung 4.10:** Formatierung der Daten- und Rückgabe-Bytes für einen Optimierer mit $N = 10$, $\mu = 4$, $\lambda = 12$ und $B_\beta = 4$; Befehls-Bytes sind gelb unterlegt

Mithilfe dieses Kommandos lässt sich leicht prüfen, ob die serielle Verbindung zum FPGA korrekt aufgebaut wurde, da, falls noch kein Seed manuell gesetzt wurde, der Standard-Seed des Optimierers als magische Zahl verwendet werden kann:

`0x27404ad634a10951f37d4b657c96703f`

## 4.3 Python-Modul

Zur Kommunikation mit dem IO-Kontroller können sämtliche Programmiersprachen verwendet werden, die eine Bibliothek besitzen, welche Funktionen zur seriellen Kommunikation per UART bereitstellen. Aufgrund der einfachen Umsetzbarkeit wurde im Rahmen dieser Arbeit ein Python-Modul namens `eacom` entwickelt, das eine Klasse `EACom` zur Verfügung stellt, die die serielle Kommunikation mit dem IO-Controller in semantisch klar definierte Methoden kapselt und so eine simple Schnittstelle zum Optimierer darstellt.

Das folgende Code-Beispiel soll einen Eindruck vermitteln, wie mithilfe von `EACom` mit dem Optimierer kommuniziert werden kann. Es werden folgende Operationen ausgeführt:

1. Verbindung zu FPGA über USB-Port `/dev/ttyUSB0` mit Baudrate 115200 herstellen
2. Optimierer zurücksetzen
3. Aktuellen Seed abfragen und neu auf 12345 setzen
4. Koeffizienten aus Datei `foo.txt` laden
5. Ising-Modus aktivieren
6. Optimierung starten
7. Nach 10 Sekunden das aktuelle Optimum abfragen
8. Verbindung trennen

```
1  # 1.
2  eacom = EACom(device="/dev/ttyUSB0", baudrate=115200)
3  eacom.connect()
4
5  # 2.
6  eacom.reset()
7
8  # 3.
9  previousSeed = eacom.seed
10 eacom.seed = (12345).to_bytes(16, byteorder="little")
11
12 # 4.
13 eacom.loadCoefficients(filename="foo.txt")
14
15 # 5.
16 eacom.enableIsingMode()
```

```
17
18  # 6.
19  eacom.start()
20
21  # 7.
22  time.sleep(10)
23  opt, loss = eacom.optimum()
24
25  # 8.
26  eacom.disconnect()
```

Sobald über `EACom.connect()` eine Verbindung zum FPGA hergestellt wurde, können die Parameter $N$, $\mu$, $\lambda$, $b_\beta$ und der aktuelle Seed über die entsprechenden Attribute `dimension`, `parents`, `children`, `coefWidth` und `seed` abgefragt und `seed` überdies gesetzt werden. Zudem können all diese Parameter über die Methode `EACom.info()` neu vom IO-Controller abgerufen und gleichzeitig zurückgegeben werden.

Die Methoden `EACom.loadCoefficients()` erhält als Argument einen Dateipfad, aus dem die Zielfunktionskoeffizienten ausgelesen werden. Das Dateiformat ist dabei ein einfaches Textformat, in dem die Koeffizienten als kommagetrennte Liste von ganzen Zahlen vorliegen müssen. Analog lässt sich eine Elternpopulation als kommagetrennte Liste von Binärstrings in einer Datei mithilfe der Methode `EACom.loadParents()` übergeben. Wird der Dateipfad beim Aufruf dieser Methode auf `None` gesetzt, wird eine Elternpopulation bestehend aus $\mu$ zufälligen Bitvektoren erzeugt und zum Optimierer übertragen.

Der gesamte Quelltext von `EACom` findet sich in Anhang A.9.

# Kapitel 5

# Auswertung

In diesem Kapitel soll der implementierte Optimierer auf der konkret verwendeten Hardware untersucht werden. Dabei sollen beispielsweise die maximalen Werte für die verschiedenen Parameter des EA für die beschränkte Chipfläche und Speicherkapazität des FPGAs ausgelotet werden. Weiterhin soll die Funktionsfähigkeit des Optimierers anhand von mehreren Experimenten verifiziert werden, in denen Optimierungsprobleme aus dem Bereich des maschinellen Lernens zuerst in QUBO-Probleme umformuliert und mithilfe des EA-Optimierers gelöst werden sollen.

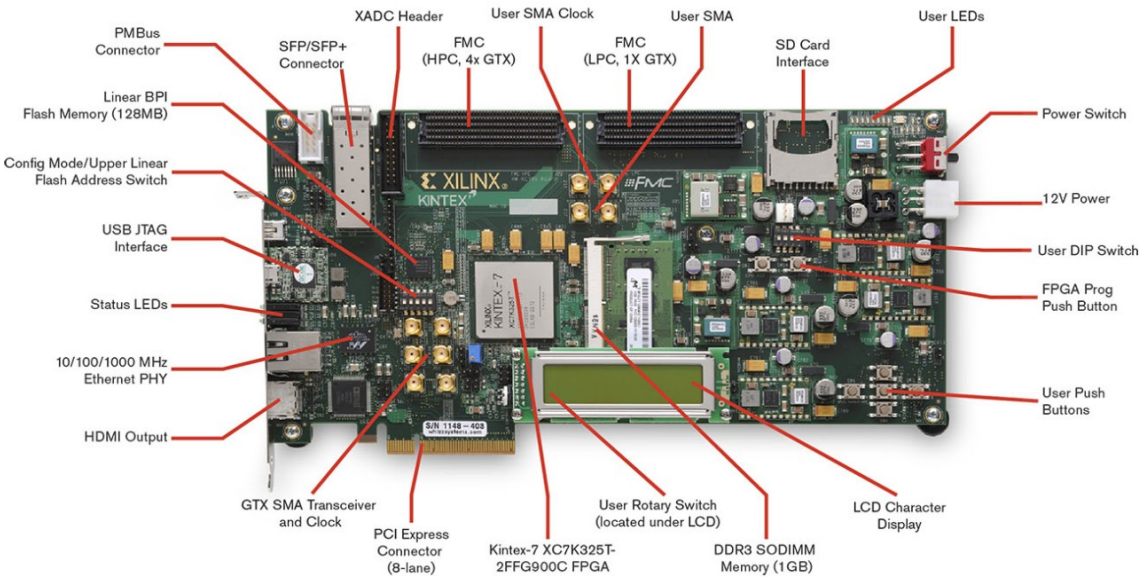



**Abbildung 5.1:** Kintex-7 KC705 (Quelle: `www.xilinx.com`)

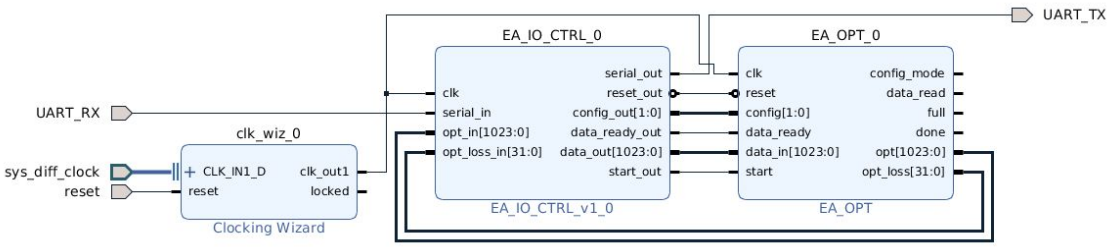



**Abbildung 5.2:** Block-Design des EA-Optimierers mit IO-Controller

## 5.1 Verwendete Hardware

Zur Durchführung der in diesem Kapitel beschriebenen Experimente wurde das *Kintex-7 KC705 Evaluation Kit* von Xilinx verwendet (siehe Abb. 5.1). Neben dem integrierten FPGA besitzt die Platine unter anderem zahlreiche Netzwerk- und Kommunikationsschnittstellen (Ethernet, UART, PCI Express), Speicher (DDR3 SODIMM, EEPROM, SD-Kartenslot), acht LEDs und ein LDC-Display, sowie eine *Differential Clock* mit 200 MHz Ausgangstaktfrequenz[1]. Der FPGA besitzt 326080 Logikzellen und etwa 16 Mb Block-RAM-Speicher, der in 445 Blöcke à 36 Kb bzw. in 890 Blöcke à 18 Kb aufgeteilt ist.

Das Block-Design mit allen in Kapitel 4 beschriebenen Komponenten, das für die Experimente auf den FPGA geladen wurde, zeigt Abb. 5.2: Ein vorgefertigter *Clocking Wizard* erzeugt aus der eingebauten *Differential Clock* ein Signal `clk_out1` mit einer Frequenz von 100 MHz, das sowohl dem Optimierer (`EA_OPT_0`) als auch dem IO-Controller (`EA_IO_CTRL_0`) als Takt dient. Die meisten Ausgänge des IO-Controllers sind mit den entsprechenden Eingängen des Optimierers verbunden, um die über den seriellen Port `UART_RX` empfangenen Kommandos und Daten in Belegungen der Konfigurationseingänge des Optimierers zu übersetzen. Über den seriellen Ausgangsport `UART_TX` werden Informationen vom IO-Controller nach außen geschickt. Die verwendete Baudrate betrug meistens 115200, was einem Wert von 868 für den IO-Controller-Parameter `CLKS_PER_BIT` entspricht. Für andere Baudraten ergibt sich der Wert für `CLKS_PER_BIT`, indem die verwendete Taktfrequenz $f$ in Hertz durch die gewünschte Baudrate $f_{\text{baud}}$ geteilt und auf die nächste ganze Zahl gerundet wird:

$$\texttt{CLKS\_PER\_BIT} = \frac{f}{f_{\text{baud}}} \qquad \frac{100 \cdot 10^6 \text{ Hz}}{115200} \approx 868$$

Die Optimierer-Ausgänge `opt` und `opt_loss` sind mit Eingängen des IO-Controllers verbunden, damit dieser das aktuelle Optimum jederzeit ablesen kann. Die restlichen Ausgänge des Optimiers (`config_mode`, `data_read`, `full` und `done`) geben lediglich Statusinformationen wieder und werden in diesem Aufbau nicht verwendet, könnten aber beispielsweise über die LEDs des Kintex-Boards ausgegeben werden, falls der FPGA lokal verwendet

[1] Details unter `https://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html`

wird. In dieser Arbeit war dies aber nicht notwendig, da der FPGA aus der Ferne betrieben wurde: Die UART-Ports waren per USB mit einem Rechner verbunden, auf dem über SSH das Python-Skript zur Kommunikation mit dem IO-Controller ausgeführt wurde. Auch die Programmierung des FPGAs geschah per Fernausführung von Vivado über die Terminalserveranwendung X2GO.

## 5.2 Resourcenverbrauch

In diesem Abschnitt soll der Einfluss der generischen Parameter ($N$, $\mu$, $\lambda$ und $b_\beta$) auf Geschwindigkeit und Schaltungsgröße des Optimierers auf dem FPGA untersucht werden. Die so ermittelten Daten sollen einen Eindruck von der Leistungsfähigkeit des Optimierers vermitteln und dabei helfen, die Parameter in Hinblick auf Geschwindigkeit und Schaltungsgröße an vorhandene Hardware bestmöglich anzupassen.

### 5.2.1 Geschwindigkeit

In einem ersten kleinen Experiment soll gemessen werden, wie viele Taktzyklen der Optimierer für eine Iteration des EA, bestehend aus Mutation, Zielfunktionsauswertung und Sortierung, benötigt. Zusammen mit der verwendeten Taktrate ergibt sich daraus die Zeitspanne, die der Optimierer für diese drei Operationen benötigt. Da Mutation und Zielfunktionsauswertung parallel ablaufen, hat die Wahl von $\mu$ und $\lambda$ keinen Einfluss auf die Laufzeit dieser beiden Operationen. Die Laufzeit des Sortiervorgangs verhält sich logarithmisch zur Größe der Gesamtpopulation und ist daher für übliche Populationsgrößen im Bereich $\mu + \lambda < 100$ vernachlässigbar. Den größten Einfluss auf die Optimierungsgeschwindigkeit hat die Dimensionsgröße $N$, da bei der Zielfunktionsauswertung sämtliche $N(N+1)/2$ Koeffizienten sequenziell durchlaufen und aufaddiert werden müssen. Die Mutation ist lediglich linear von $N$ abhängig.

Zur Messung der benötigten Taktzyklen wurde das Blockdesign des FPGAs um eine Komponente erweitert und leicht umgebaut (siehe Abb. 5.3). Die neue Komponente (im Diagramm als `Clk_Counter_0` beschriftet) besitzt neben Takt- und Reset-Eingang zwei Eingänge `start` und `stop`, die einen internen Zähler steuern, der ein 32-Bit-Register in jeder steigenden Taktflanke um 1 hochzählt und den momentanen Wert dieses Registers am Ausgang `count` ausgibt. Gestartet wird diese Komponente, wenn durch den IO-Controller der Eingang `start` des Optimierers aktiviert wird. In diesem Moment wechselt der Optimierer vom Konfigurations- in den Initialisierungsmodus. Um eine möglichst genaue Messung zu erzielen, wird das Budget $B$ des Optimierers auf 1000 gesetzt und der Zähler gestoppt, wenn das Budget aufgebraucht ist – dies lässt sich anhand des Optimierer-Ausgangs `done` feststellen, der beim Wechsel in den Zustand `FINISHED` auf 1 gesetzt wird, weshalb `done` mit dem Zähler-Eingang `stop` verbunden ist.

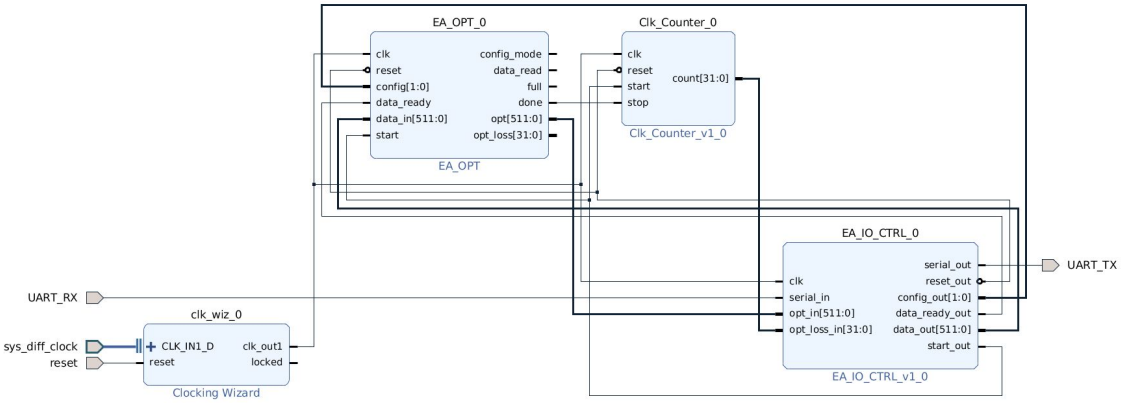



**Abbildung 5.3:** Block-Design der Komponenten zur Messung der Laufzeit mit Taktzähler `Clk_Counter_0`

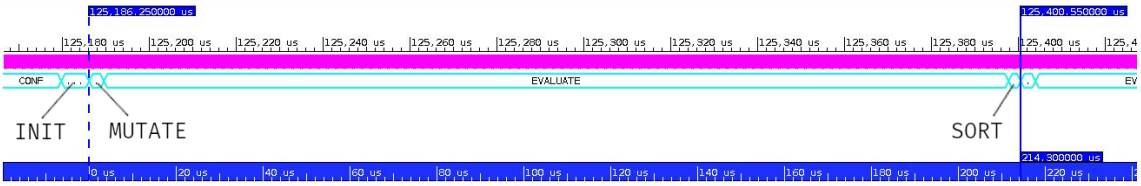



**Abbildung 5.4:** Zustände des Optimierers im Zeitverlauf mit $N = 64$ und $\lambda = 31$ (Simulator); den Großteil der Zeit pro Iteration nimmt die Zielfunktionsauswertung in Anspruch

Die so gemessene Gesamtzahl $T$ von Taktzyklen setzt sich zusammen aus der Initialisierungsphase und tausend EA-Iterationen. Da $T_{\texttt{INIT}}$ lediglich von $\lambda$ abhängig und im Verhältnis zum zweiten Summanden klein ist, lässt sich auch dieser Term vernachlässigen, und die Anzahl Taktzyklen pro Iteration ergibt sich näherungsweise aus der gemessenen Gesamttaktzahl geteilt durch das Budget:

$$T = T_{\texttt{INIT}} + B \cdot T_{\texttt{IT}} \approx B \cdot T_{\texttt{IT}} \;\Leftrightarrow\; T_{\texttt{IT}} \approx \frac{T}{B}$$

Aus der verwendeten Taktfrequenz $f$ [Hz] ergibt sich eine Näherung der Zeit pro Iteration $t_{\texttt{IT}}$. Für $N \gg \mu + \lambda$ hat die Zielfunktionsauswertung mit einer Zeitkomplexität von $\mathcal{O}(N^2)$ den größten Anteil an $T_{\texttt{IT}}$; Abb. 5.4 zeigt den Verlauf der Zustände des Optimierers direkt nach Beginn der Optimierung und verdeutlicht das Verhältnis der Zeitdauern, die für die einzelnen Schritte des EA nötig sind. Bereits für verhältnismäßig kleines $N = 64$ und großes $\lambda = 31$ wird deutlich, dass die Zielfunktionsauswertung den Großteil der Zeit pro Iteration beansprucht. Daher lässt sich mit einer weiteren Vereinfachung unter Berücksichtigung der Parallelisierung die durchschnittliche Zeit pro Zielfunktionsauswertung $t_{\texttt{EVAL}}$ abschätzen.

$$t_{\texttt{IT}} \approx \frac{T}{B \cdot f} \qquad\qquad t_{\texttt{EVAL}} \approx \frac{t_{\texttt{IT}}}{\lambda}$$

| $N$ | $T$ | $T_{\texttt{IT}}$ | $t_{\texttt{IT}}$ [$\mu$s] | $t_{\texttt{EVAL}}$ [$\mu$s] |
|---|---|---|---|---|
| 128 | 8359418 | 8359 | 83,6 | 2,7 |
| 256 | 33088122 | 33088 | 330,9 | 10,7 |
| 512 | 131746682 | 131747 | 1317,5 | 42,5 |
| 1024 | 525868410 | 525868 | 5258,7 | 169,6 |

**Tabelle 5.1:** Ergebnisse der Zeitmessungen für verschiedene $N$ und feste $\mu = 1$, $\lambda = 31$, $B = 1000$ und $f = 100$ MHz

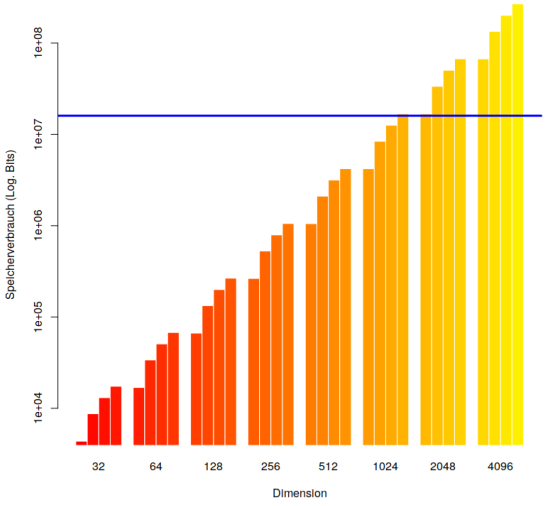
Da auch die Wahl der Koeffizienten auf die Geschwindigkeit keinen Einfluss hat, wurde als Platzhalter das *One-Max*-Problem gewählt, dessen einziges globales Minimum der Eins-Vektor ist ($\beta_{ij} = -1$ falls $i = j$, sonst 0).

Für die Messung von $T$ wurden für $N$ die Werte 128, 256, 512 und 1024 gewählt und mit den oben beschriebenen Näherungen die Werte $T_{\texttt{IT}}$, $t_{\texttt{IT}}$ und $t_{\texttt{EVAL}}$ auf einem $(1, 31)$-EA bestimmt. Die Taktfrequenz betrug $f = 100$ MHz. Die Ergebnisse sind in Tabelle 5.1 aufgelistet.

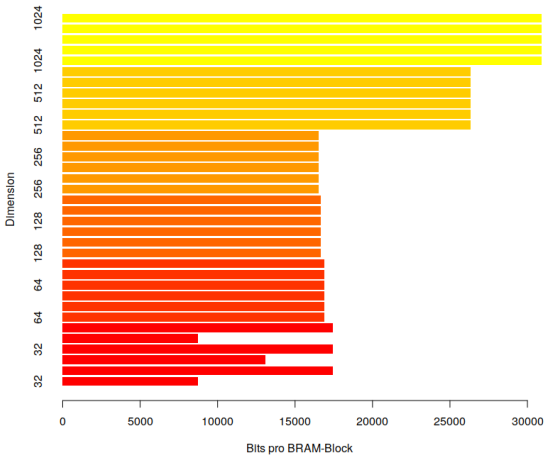
Aus den gemessenen und berechneten Werten geht hervor, dass zwischen der Laufzeit und der Dimension $N$, wie erwartet, ein quadratischer Zusammenhang besteht: Für ein doppelt so großes $N$ wird in etwa die vierfache Laufzeit benötigt. Dank des hohen Grads an Parallelisierung durch $\lambda = 31$ liegt die durchschnittliche Zeit für die Auswertung der polynomiellen Zielfunktion dennoch selbst bei $N = 1024$ bei etwa dem Sechstel einer Millisekunde.

### 5.2.2 Chipfläche

Die Wahl der generischen Parameter des Optimierers hat einen erheblichen Einfluss auf den Platzverbrauch der resultierenden Schaltung auf dem FPGA-Chip. Während beispielsweise die Wahl eines großen Wertes für die Anzahl Nachkommen $\lambda$ eine höhere Parallelisierung der Mutation und Zielfunktionsauswertung und somit eine insgesamte schnellere Optimierung bewirkt, steigt jedoch die Anzahl der verwendeten LUTs und Flip-Flops, da für jeden Nachkommen ein eigenes Mutations- und Auswertungsmodul auf dem Chip erzeugt wird. Überschreitet die Gesamtgröße aller Zielfunktionsparameter die verfügbare Menge an Block-RAM-Speicher, wird der entsprechende Speicherplatz stattdessen in Form von LUT-RAM allokiert, was zu einer Explosion des Bedarfs an LUT-Elementen führt. In diesem Abschnitt soll untersucht werden, inwieweit die verschiedenen Parameter des EA-Optimierers den Platzverbrauch beeinflussen, um so als Richtlinie für eine möglichst effiziente Nutzung vorhandener Resourcen zu dienen. Sämtliche Messungen wurden mithilfe des Synthese-Tools von Vivado auf dem Block-Design durchgeführt, das EA-Optimierer, IO-Controller und *Clocking-Wizard* enthält (siehe Abb. 5.2). Messwerte, bei denen der vorhandene Block-RAM nicht ausreichte, wurden verworfen, da die Anzahl verwendeter LUTs




**(a)** Log. Anzahl Bits für alle Koeffizienten




**(b)** Anzahl Bits pro BRAM-Block in Abhängigkeit von Dimension $N$

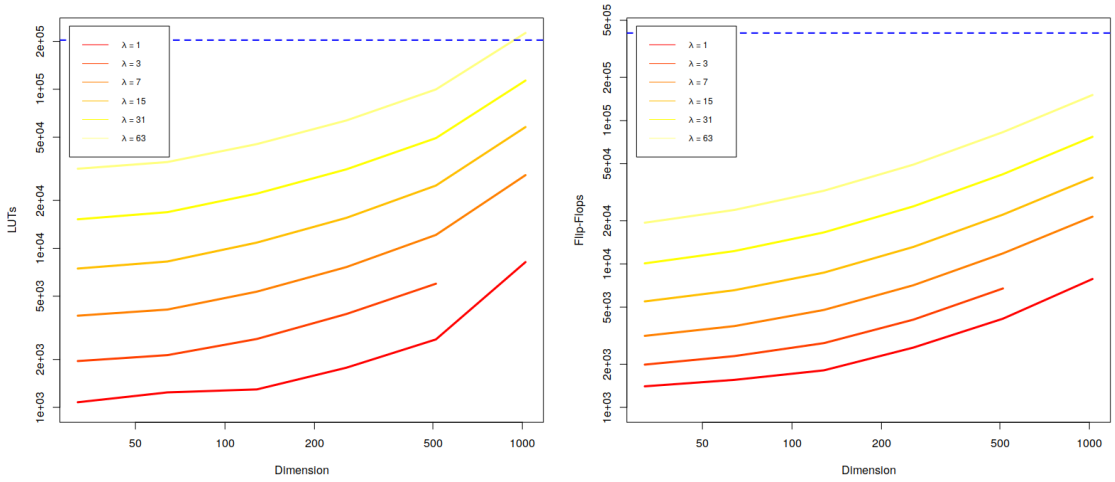
und Flip-Flops dadurch verfälscht wurde. Die Werte bilden eine solide Referenz für die tatsächliche Anzahl implementierter Schaltungselemente, können allerdings, abhängig von der verwendeten Hardware, im Detail abweichen.

Typischerweise ist für ein gegebenes Optimierungsproblem die Dimension $N$ festgelegt. Dieser Parameter hat den größten Einfluss auf den Verbrauch von Block-RAM, da die Anzahl zu speichernder Parameter $N(N+1)/2 \in \mathcal{O}(N^2)$ beträgt. Zusammen mit der Bit-Breite $b_\beta$ pro Koeffizient ergibt sich der Speicherverbrauch für alle Parameter in Bit als

$$b_{\text{coef}} = \frac{b_\beta}{2}(N^2 + N).$$

Die im Rahmen dieser Arbeit verwendete FPGA-Serie Kintex-7 besitzt Block-RAM-Blöcke der Größe 36 Kb[2]. Theoretisch sollte die Anzahl verwendeter Blöcke daher immer etwa $b_{\text{coef}}/36000$ betragen. Wie Abb. 5.5b zeigt, wendet Vivado allerdings scheinbar verschiedene Speicherstrategien für verschiedene Datenmengen an: Die Farben zeigen die Dimension $N$ an (in Zweierpotenzen von rot = 32 bis gelb = 1024), die einzelnen Balken verschiedene $b_\beta$ (8, 16, 24, 32) innerhalb der Versuchsreihe. Offenbar liegt die Speicherdichte für $N$ bis 512 bei etwa 17000 Bits pro Block und springt für höhere Werte auf eine Dichte nahe 30000, unabhängig von der Wortbreite $b_\beta$. Die wahrscheinlichste Erklärung für dieses Verhalten ist, dass die Block-RAM-Blöcke für kleinere Datenmengen in zwei unabhängige Blöcke à 18 Kb geteilt werden, für größere hingegen als einzelne große Blöcke à 36 Kb verwendet werden – diese Möglichkeit wird im entsprechenden Handbuch explizit genannt[2].

[2] `https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf`




**(a)** LUTs **(b)** Flip-Flops

**Abbildung 5.6:** Log. Verbrauch von LUTs und Flip-Flops abhängig von $N$ und $\lambda$

Der theoretische Speicherverbrauch für verschiedene Kombinationen von $N$ und $b_\beta$ ist in Abb. 5.5a dargestellt: Für jedes $N$ stellen die vier Balken von links nach rechts die Werte 8, 16, 24 und 32 Bits für $b_\beta$ dar. Die Gesamtzahl Bits für alle Koeffizienten ist logarithmisch auf der Y-Achse aufgetragen. Die horizontale blaue Linie zeigt die maximale Block-RAM-Speicherkapazität des Kintex XC7K325T, der für diese Arbeit verwendet wurde. Da $b_\beta$ bei Verwendung des Block-RAMs nur sehr geringen Einfluss auf die Anzahl verwendeter LUTs und Flip-Flops hat, kann für ein gegebenes $N$ die Größe der Koeffizienten so gewählt werden, dass die Kapazität der verwendeten Hardware am besten ausgenutzt wird (wobei die Möglichkeit eines Überlaufs bei der Zielfunktionsauswertung bedacht werden sollte, vgl. das Vorgehen in Abschnitt 5.3.1).

Die Verwendung von LUTs und Flip-Flops ist in Abb. 5.6 dargestellt; die maximale Kapazität des verwendeten FPGAs ist erneut als blaue Linie eingezeichnet. Bei sämtlichen Messungen wurde $\mu = 1$ gewählt, da dieser Parameter am wenigsten Einfluss auf die Schaltungsgröße nimmt – lediglich die Größe der Register für die Population und ihre Zielfunktionswerte sowie die Größe des Sortierers sind davon abhängig – und üblicherweise klein gewählt wird. Ein möglichst großes $\lambda$ ist hingegen immer erstrebenswert, um möglichst effektiv den Geschwindigkeitsvorteil durch Parallelisierung zu nutzen. Um die Populationgröße zu einer Zweierpotenz zu ergänzen und so die Sortierkomponente optimal auszunutzen (vgl. Abschnitt 4.1.1), wurde entsprechend $\lambda = 2^k - 1$ für $1 \leq k \leq 6$ gewählt.

Wie die Graphik zeigt, ist mit der verwendeten Hardware die Optimierung von 1024 binären Variablen mit $\lambda = 31$ parallelen Mutationen und Zielfunktionsauswertungen problemlos möglich, solange $b_\beta$ so gewählt wird, dass sämtliche Koeffizienten in den Block-RAM passen.

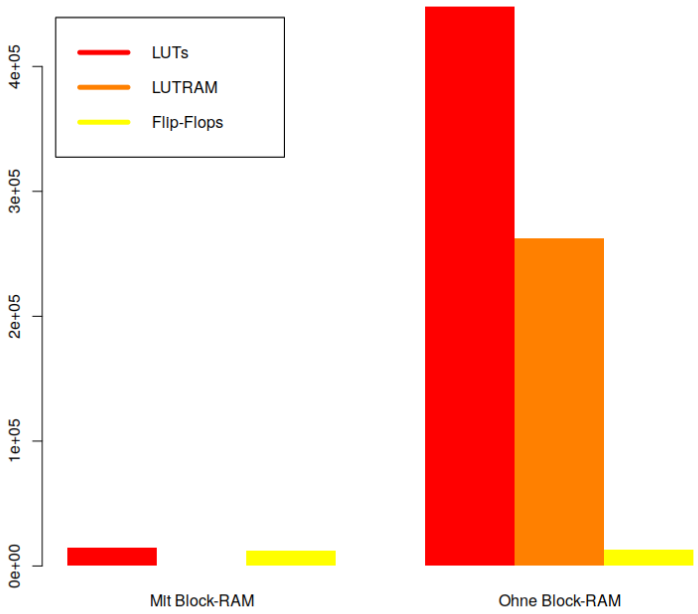



**Abbildung 5.7:** Verbrauch von FPGA-Elementen mit und ohne Verwendung von Block-RAM; der Verbauch von LUT-Elementen wird durch Benutzung des Block-RAMs drastisch reduziert

In der Messreihe für $\lambda = 3$ fehlt der Wert für $N = 1024$, da dort $b_\beta = 32$ gewählt wurde, was dazu führte, dass die Koeffizienten nicht mehr in den Block-RAM-Speicher passten und in die LUTs ausgelagert wurden. Um den Einfluss zu verdeutlichen, den die Verwendung von Block-RAM auf die Gesamtgröße der Schaltung hat, wurde die Messung mit $b_\beta = 8$ wiederholt, sodass die Koeffizienten in den Block-RAM-Speicher passen. Den Verbrauch an FPGA-Elementen stellt Abb. 5.7 dar: Die linke Seite zeigt den Verbrauch für $b_\beta = 8$, die rechte Seite für $b_\beta = 32$. Da $b_\beta$ wenig Einfluss auf den LUT-Verbrauch hat, wenn Block-RAM verwendet wird, ist der Verbrauch vergleichbar mit dem Fall, dass die Koeffizienten für $b_\beta = 32$ in den Block-RAM passen würden. Fasst man LUTs und LUT-RAM zusammen, ist der gesamte LUT-Verbrauch ohne Block-RAM um fast das 50-fache höher. Die Implementierung des Koeffizientenspeichers war demnach ein entscheidender Schritt, um Optimierungen mit über tausend Variablen überhaupt zu ermöglichen.

## 5.3 Anwendungsbeispiele

Die Funktionalität des EA-Optimierers soll nun mithilfe zweier konkreter Optimierungsprobleme demonstriert werden: Dem Clustering-Problem mit zwei Clusterzentren sowie dem Maximum-A-Posteriori-Problem auf Markov-Random-Fields (MRFs).

### 5.3.1 2-Means-Algorithmus

Gegeben eine Menge von Datenpunkten $X \subseteq \mathbb{R}^d$ mit $n = |X|$ ist das Ziel eines *partitionierenden Clusterverfahrens*, eine Abbildung $\chi : X \mapsto \{1, \ldots, k\}$ zu finden, die jedes $\boldsymbol{x} \in X$ einer von $k \in \mathbb{N}$ Partitionen zuteilt, sodass ein bestimmtes Kriterium optimiert wird. Die wohl häufigsten Vertreter dieser Verfahren sind die $k$-Means-Algorithmen, die eine Menge von Clusterzentren $\boldsymbol{\mu}_1^*, \ldots, \boldsymbol{\mu}_k^*$ zu finden suchen, die den mittleren quadratischen Abstand zu allen nächstgelegenen Datenpunkten minimieren. Anders formuliert werden Clusterzentren mit *minimaler Varianz* gesucht:

$$\begin{aligned} L(X;\ \boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k) &= \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in X_i} \|\boldsymbol{x} - \boldsymbol{\mu}_i\|^2, \\ X_i &= \{\boldsymbol{x} \in X \mid \operatorname*{argmin}_{j \in \{1,\ldots,k\}} \|\boldsymbol{x} - \boldsymbol{\mu}_j\|^2 = i\} \\ \boldsymbol{\mu}_1^*, \ldots, \boldsymbol{\mu}_k^* &= \operatorname*{argmin}_{\boldsymbol{\mu}_1,\ldots,\boldsymbol{\mu}_k} L(X;\ \boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k) \end{aligned}$$

Die Anzahl $k$ von Clusterzentren ist dabei fest. Ein oft verwendetes Lösungsverfahren ist der Lloyd-Algorithmus, ein iteratives Verfahren, das, ausgehend von zufälligen initialen Clusterzentren, abwechselnd die Mengen $X_i$ und deren Mittelpunkte $(1/|X_i|) \sum_{\boldsymbol{x} \in X_i} \boldsymbol{x}$ als neue Clusterzentren bestimmt [16]. Abhängig von den initialen Zentren konvergiert der Lloyd-Algorithmus zu einem lokalen Optimum. Um möglichst ein globales Optimum zu erhalten, ist es üblich, den Algorithmus mehrmals mit unterschiedlichen Startwerten auszuführen, was natürlich dennoch keine Garantie für ein globales Optimum gibt.

In [3] stellen Bauckhage et al. einen $k$-Means-Algorithmus mit $k = 2$ vor, in dem die Clusteranalyse als Minimierung eines Ising-Modells formuliert wird. Dazu wird gezeigt, dass die Belegung der Variablen eines Ising-Modells, $\boldsymbol{s} \in \{-1, +1\}^n$, als Zuordnung von entsprechend indizierten Datenpunkten $\boldsymbol{x}_i \in X$ zu einem von zwei Clustern interpretiert werden kann:

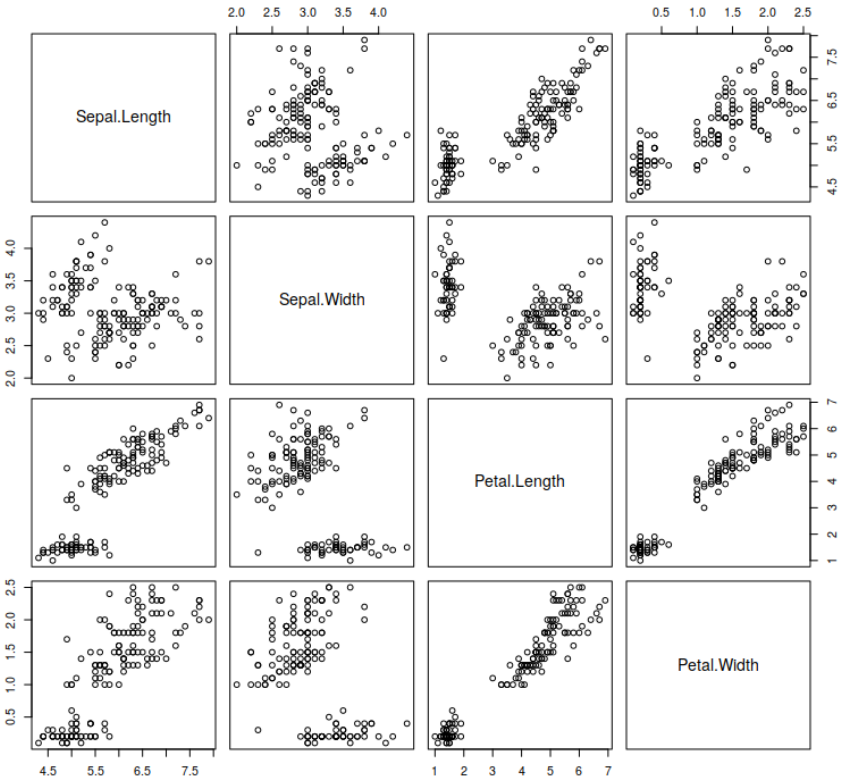
$$\chi : X \mapsto \{-1, +1\},\ \chi(\boldsymbol{x}_i) = s_i$$

Weiterhin wird gezeigt, dass diejenige Belegung $\boldsymbol{s}^*$ ein näherungsweise optimales Clustering ist, das folgenden Ausdruck minimiert:

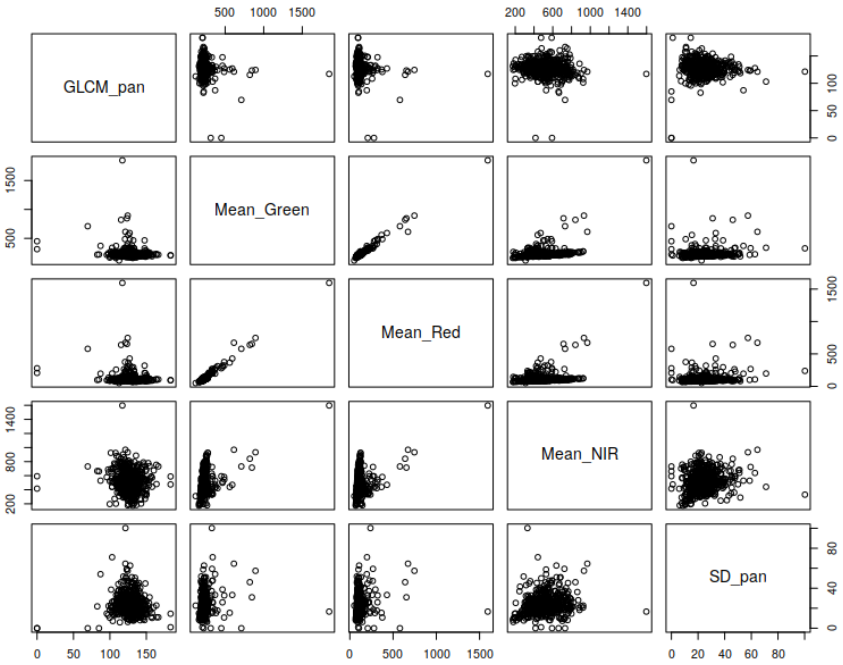
$$\boldsymbol{s}^* = \operatorname*{argmin}_{\boldsymbol{s} \in \{-1,+1\}^n} -\boldsymbol{s}^\top X^\top X \boldsymbol{s}$$

Die Koeffizienten eines Ising-Modells sind daher gegeben durch die negative Gramsche Matrix $\boldsymbol{G} = X^\top X$ über die Datenpunkte. Das Ergebnis der Minimierung approximiert ein optimales *lineares* Clustering. Unter Verwendung eines geeigneten Mercer-Kernels können auch nicht-lineare Features der Daten eingefangen werden.

Für das Experiment wurde diese Methode des 2-Means-Clusterings auf fünf verschiedenen Datensätzen ausgeführt und die Ergebnisse mit denen des Lloyd-Algorithmus ver-




**(a)** *iris*




**(b)** *wilt*

**Abbildung 5.8:** Verwendete Datensätze (Auswahl)

| Datensatz | Variablen | (verw.) | Beispiele | (verw.) |
|---|---|---|---|---|
| *iris* | 5 | 1–4 | 150 | |
| *abalone* | 9 | 2–9 | 4177 | 500 |
| *sonar* | 61 | 1–60 | 208 | |
| *wilt* | 6 | 2–6 | 4839 | 500 |
| *ionosphere* | 35 | 3–34 | 351 | |

**Tabelle 5.2:** Verwendete Datensätze für 2-Means-Experiment

glichen. Die verwendeten Datensätze sind in Tabelle 5.2 aufgeführt; Der Datensatz *iris*[3] enthält Messungen von Blütenblättern verschiedener Unterarten von Schwertlilien (siehe Abb. 5.8a); die ersten vier Variablen wurden zum Clustering verwendet, da die fünfte (Spezies) nominal ist. Der *abalone*[4]-Datensatz enthält Längen- und Gewichtsangaben von Seeohren, einer Art von Meeresschnecken; die erste Variable des Datensatzes (Geschlecht) ist nominal und wurde verworfen, außerdem wurden statt des vollen Datensatzes 500 Beispiele zufällig mithilfe der R-Methode `sample` ausgewählt, dabei wurde zur Reproduzierbarkeit der feste Seed `555555` gewählt. Der dritte Datensatz *sonar*[5] besteht aus 61 Variablen und 208 Beispielen zu Sonarmessungen von Metallzylindern und Felsen; Variable 61, die Klasseninformation, wurde verworfen. Der Datensatz *wilt*[6] besteht aus Farb- und Texturwerten, die aus Satellitenaufnahmen zur Erkennung erkrankter Bäume extrahiert wurden (siehe Abb. 5.8b); er enthält 4839 Beobachtungen von 6 Variablen, von denen alle bis auf Variable 1 (Klasseninformation) verwendet wurden. Analog zu Datensatz *abalone* wurden auch hieraus 500 zufällige Beispiele gezogen. Zuletzt enthält der Datensatz *ionosphere*[7] 351 Beobachtungen von 35 Variablen, von denen die reellwertigen Variablen 3 bis 34 verwendet wurden; Variablen 1 und 2 zeigen zu wenig Variation und Variable 35 ist die Klasseninformation.

Zur Berechnung der Koeffizienten wurde zunächst die Gramsche Matrix bestimmt. Da der Wertebereich der Koeffizienten, die der Optimierer akzeptiert, für eine feste Bit-Breite $b_\beta$ genau $[-2^{b_\beta-1}, 2^{b_\beta-1}-1]$ beträgt und somit nahezu symmetrisch um 0 herum liegt, wurde auch die Gram-Matrix um 0 zentriert:

$$\hat{\boldsymbol{G}} = \boldsymbol{G} - \frac{1}{n}\mathbf{1}_n\boldsymbol{G} - \frac{1}{n}\boldsymbol{G}\mathbf{1}_n + \frac{1}{n^2}\mathbf{1}_n\boldsymbol{G}\mathbf{1}_n$$

$\mathbf{1}_n$ bezeichnet dabei die $n \times n$-Matrix, die ausschließlich aus Einsen besteht. Um die bestmögliche Präzision für gegebenes $b_\beta$ zu erhalten, wurde die Matrix so skaliert, dass ihr

[3] `https://archive.ics.uci.edu/ml/datasets/iris`
[4] `https://archive.ics.uci.edu/ml/datasets/abalone`
[5] `http://archive.ics.uci.edu/ml/datasets/connectionist+bench+(sonar,+mines+vs.+rocks)`
[6] `http://archive.ics.uci.edu/ml/datasets/wilt`
[7] `https://archive.ics.uci.edu/ml/datasets/ionosphere`

| | Datensatz | $WCSS_1$ | $WCSS_2$ | $L$ |
|---|---|---|---|---|
| Lloyd | *iris* | 28.5521 | 123.7959 | 152.3480 |
| | *abalone* | 857.859 | 1089.282 | 1947.141 |
| | *sonar* | 117.9025 | 162.6797 | 280.5821 |
| | *wilt* | 11202527 | 3460832 | 14663360 |
| | *ionosphere* | 1675.9472 | 711.3445 | 2387.2917 |
| EA | *iris* | $56.3664 \pm 0.0$ | $106.7996 \pm 0.0$ | $163.1659 \pm 0.0$ |
| | *abalone* | $508.9748 \pm 0.0$ | $1612.1689 \pm 0.0$ | $2121.1438 \pm 0.0$ |
| | *sonar* | $123.4056 \pm 0.0$ | $157.1640 \pm 0.0$ | $280.5696 \pm 0.0$ |
| | *wilt* | $11696184 \pm 0.0$ | $3038644 \pm 0.0$ | $14734828 \pm 0.0$ |
| | *ionosphere* | $1695.4280 \pm 0.0$ | $692.8828 \pm 0.0$ | $2388.3108 \pm 0.0$ |

**Tabelle 5.3:** Erzielte Clustering-Ergebnisse; $WCSS_i$ ist die Summe der quadratischen Abstände aller zugehörigen Datenpunkte zum Clusterzentrum $\boldsymbol{\mu}_i$, $L$ die Summe dieser Werte. Möglichst kleine Werte sind erstrebenswert

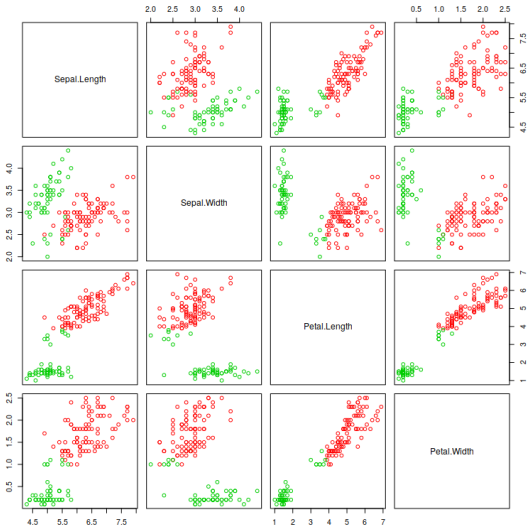
betragsmäßig maximaler Wert zum größten im Zweierkomplement darstellbaren Wert wurde, anschließend wurden alle Werte auf die nächste ganze Zahl gerundet, da der Optimierer nur auf ganzzahligen Koeffizienten arbeitet. Zusammengefasst lautet die Berechnungsvorschrift für die Koeffizienten also

$$\beta_{ij} = \lfloor -\alpha \hat{G}_{ij} + 0.5 \rfloor \qquad \text{mit } \alpha = \frac{2^{b_\beta - 1} - 1}{\max\{|\hat{G}_{ij}| \mid \hat{G}_{ij} \in \hat{\boldsymbol{G}}\}}. \tag{5.1}$$

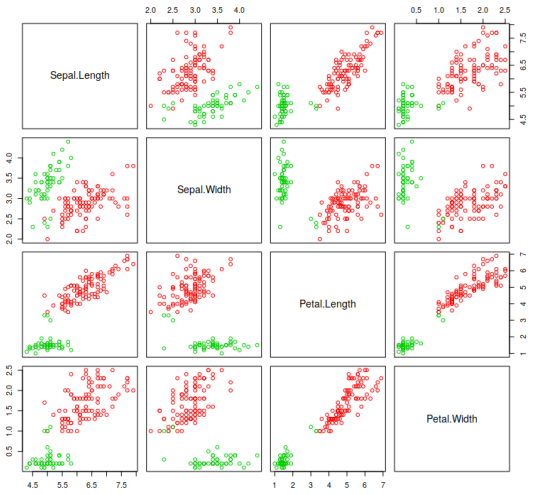
Für das Experiment wurde $b_\beta = 16$ gewählt. Da der Optimierer intern die Zielfunktionswerte mit 32 Bit speichert, sollte bei $b_\beta$ nahe oder gleich 32 darauf geachtet werden, dass nicht der gesamte Wertebereich ausgenutzt wird, um Überläufe bei der Summierung der Koeffizienten zu vermeiden. Als Heuristik kann der Wertebereich mithilfe einer unteren Schranke für den Wert des globalen Minimums skaliert werden, beispielsweise der Summe aller negativen Koeffizienten.

Für die EA-Parameter wurde $\mu = 2$ und $\lambda = 30$ gewählt. Für jedes optimale Clustering existiert ein symmetrisches Clustering mit vertauschten Klassen, das den gleichen Zielfunktionswert besitzt. Die Hoffnung mit $\mu = 2$ liegt darin, dass dem Optimierer so ermöglicht wird, sich gegebenenfalls in zwei Richtungen gleichzeitig einem der beiden Optima zu nähern.

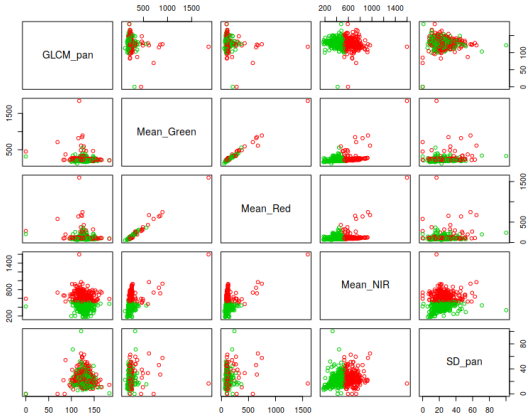
Auf jedem der fünf Datensätze wurde die Optimierung zehnmal durchgeführt, jeweils mit einem anderen Seed und einer anderen zufällig erzeugten Startpopulation, um die etwaige Variabilität der Lösungen abzuschätzen. Die Optimierung wurde ohne Budgetbeschränkung ausgeführt und jeweils genau eine Minute laufen gelassen. Eine längere Optimierungszeit war augenscheinlich nicht notwendig, da in jedem der Läufe bereits nach



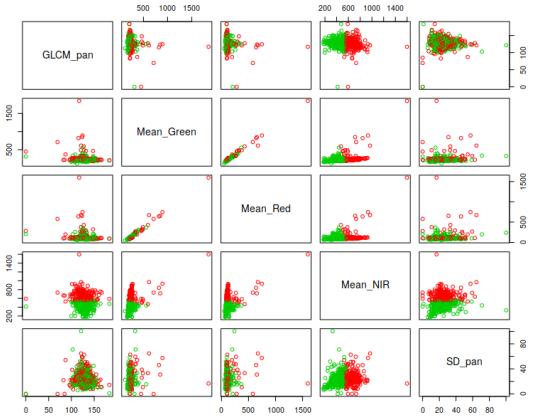

**(a)** EA-Optimierer auf *iris*




**(b)** Lloyd-Algorithmus auf *iris*




**(c)** EA-Optimierer auf *wilt*




**(d)** Lloyd-Algorithmus auf *wilt*

**Abbildung 5.9:** Streudiagramm-Matrizen der gefundenen Clusterzuordnungen auf zwei der untersuchten Datensätze mit farblich gekennzeichneten Clusterzugehörigkeiten

unter einer Sekunde das stabile Optimum erreicht wurde, das sich bis zum Ende der Optimierungszeit nicht mehr änderte.

Die Ergebnisse des Experiments sind in Tabelle 5.3 aufgeführt. $WCSS_i$ (für *within cluster sum of squares*) bezeichnet die Summe der quadratischen Abweichung vom Klassenzentrum für die Klasse $i$, und $L = WCSS_1 + WCSS_2$ den gesamten Zielfunktionswert. Sämtliche Konfigurationen verschiedener Startpopulationen und Seeds führten – bis auf Symmetrie – stets zum gleichen Clustering pro Datensatz, weshalb die Standardabweichung aller Werte gleich 0 ist. Für den Vergleich mit den Ergebnissen des Lloyd-Algorithmus' wurden die einzelnen Clusterings gegebenenfalls invertiert, um das Clustering zu erhalten, das

dem Referenzergebnis am nächsten ist. In Abb. 5.9 sind Plots der gefundenen Lösungen für die niedrigdimensionalen Datensätze *iris* und *wilt* dargestellt.

Die Referenzwerte für den Lloyd-Algorithmus wurden mithilfe der R-Funktion `kmeans`[8] und dem Parameter `algorithm = "Lloyd"` ermittelt. Um möglichst vergleichbare Bedingungen zu erreichen, wurde der Algorithmus mit zwei zufälligen Startpopulationen ausgeführt (Parameter `nstart = 2`).

Offensichtlich sind die Ergebnisse des Lloyd-Algorithmus' im Allgemeinen etwas besser als die des EA-Optimierers: Auf vier von fünf Datensätzen ist der Zielfunktionswert des Lloyd-Clusterings niedriger, lediglich auf den hochdimensionalen Datensätzen *sonar* und *ionosphere* sind die Werte nahezu identisch. Ein Grund für das schlechtere Ergebnis auf niedrigdimensionalen Daten des EA-Optimierers liegt vermutlich in den vereinfachenden Annahmen begründet, die zur Formulierung der Zielfunktion des Ising-Modells getroffen werden; so wird in [3] näherungsweise angenommen, dass für ein optimales Clustering $|X_1| \approx |X_2| \approx |X|/2$ gilt. Andererseits könnte die Skalierung und anschließenden Rundung der Koeffizienten eine Rolle spielen.

Um letztere Vermutung zu überprüfen, wurde das Experiment auf dem *iris*-Datensatz mit $b_\beta = 24$ und dem entsprechend angepassten Skalierungsfaktor $\alpha$ aus Gleichung (5.1) wiederholt. Da dies zu einem Überlauf des 32-Bit-Zielfunktionswerts führte, wurde der maximale Wertebereich angenähert, indem $\alpha$ durch einem weiteren Vorfaktor $2^{-k}$ verkleinert wurde, bis ab $k = 4$ kein Überlauf mehr auftrat. Das neue Optimum war identisch zum vorherigen, was die Vermutung bestärkt, dass die Formulierung des 2-Means-Problems als Ising-Modell – zumindest ohne Verwendung eines Kernels – keine bessere Lösung zulässt.

Zumindest für große Dimensionen $n$ scheint sich die Güte der Ising-Modellierung jedoch dem des Lloyd-Algorithmus' anzunähern. Während in [3] das Ising-Modell mithilfe von Quanten-Annealing optimiert wird, für das pro Datenpunkt ein Qubit benötigt wird, ist die Verwendung des EA-Optimierers vermutlich günstiger und ermöglicht das Clustering mehrerer tausend Datenpunkte, während beispielsweise Googles neuester Quantenprozessor lediglich 72 Qubits besitzt[9].

### 5.3.2 Markov-Random-Fields

In diesem Experiment soll der Optimierer genutzt werden, um auf einem MRF das Maximum-A-Posteriori-Problem zu lösen, d. h. die wahrscheinlichste Variablenbelegung bezüglich der gelernten Daten zu ermitteln. Da ein MRF mit einer Graphstruktur $G = (V, E)$ Abhängigkeiten zwischen den Variablen modelliert und die Wahrscheinlichkeitsdichte der gemeinsamen Verteilung aller Variablen über die Cliquen des zugehörigen Graphs faktorisiert wird, kann zur Ermittlung der wahrscheinlichsten Variablenbelegung nicht etwa jede Variable

[8] `https://stat.ethz.ch/R-manual/R-devel/library/stats/html/kmeans.html`
[9] `https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html`

isoliert betrachtet werden; vielmehr ist auf der gemeinsame Dichte aller Kombinationen von Variablenbelegungen ein Minimum gesucht, was den Suchraum erheblich vergrößert. Um überhaupt einen endlichen Suchraum zu erhalten, werden in diesem Experiment diskrete MRFs verwendet, d. h. die modellierten Variablen $X = (X_v)_{v \in V}$ haben endliche Zustandsräume $|\mathcal{X}_v| < \infty \; \forall v \in V$.

Zur Erstellung und zum Training der MRFs wurde das Software-Tool *px* verwendet, das Modelle mit ganzzahligen Kantengewichten erzeugt [23, 22] und daher für den Optimierer, der ebenfalls auf ganzzahligen Zielfunktionsparametern arbeitet, sehr gut geeignet ist. Zur Verarbeitung der Daten wurden folgende Schritte mithilfe von *px* durchgeführt:

1. **Diskretisierung** numerischer Variablen (`px-discretize`)
2. **Training** eines ganzzahligen MRFs (`px-train`)
3. **Auslesen** der ermittelten Kantengewichte (`px-inspect`)
4. **Vorhersage** der wahrscheinlichsten Variablenbelegung (`px-predict`)

Bei der Diskretisierung werden die numerischen Daten an ihren 10-%-Quantilen aufgeteilt (Parameter `-q 10`), sodass bis zu 11 *Bins* entstehen. Zu kleine Bins werden automatisch zusammengefasst. Der Aufruf erzeugt einen neuen Datensatz, in dem alle diskretisierten Daten durch die Nummer ihres entsprechenden Bins ersetzt wurde. Beim Training auf dem so gewonnenen neuen Datensatz wurde die Option `-s 11` gewählt, die ein ganzzahliges MRF auf den Daten erzeugt. Enthielt der Datensatz fehlende Werte, so wurden diese durch zehn Iterationen eines EM-Algorithmus' geschätzt [8] (Parameter `-J 10`). Zur Ermittlung der graphischen Struktur des MRFs wurde der Chow-Liu-Algorithmus verwendet, der einen Maximum-Likelihood-Schätzer der optimalen Baumstruktur approximiert [6] (Parameter `CHOWLIU`). Der Aufruf von `px-inspect` schließlich gibt eine Liste sämtlicher Kantengewichte $\theta_{X_u=x, X_v=y}$ zurück, die beim Training des Modells ermittelt wurden.

Die Zielfunktion des MAP-Problems gegeben eine Variablenbelegung $x = (x_v)_{v \in V}$ ist die durch das MRF definierte Wahrscheinlichkeitsdichte

$$p(X = x) = \frac{1}{Z} \exp\left( \sum_{u,v \in V} \theta_{X_u = x_u, X_v = x_v} \right),$$

wobei $Z$ die Normierungskonstante ist, die $p$ zu einer Dichtefunktion macht. Zu beachten ist, dass diese vereinfachte Definition nur für den vorliegenden Fall gilt, in dem die graphische Struktur ein Baum ist und alle Cliquen somit aus genau zwei Knoten bestehen. Gesucht ist eine Belegung $x^* = \operatorname*{argmax}_{x \in \mathcal{X}} p(X = x)$. Da sowohl die Normierung durch $Z$ als auch die Exponentialfunktion monotone Abbildungen sind, genügt zur Maximierung dieser Funktion die Maximierung der Summe über alle aktiven Kantengewichte $\theta$. Für Knoten $u, v \in V$ mit $(u, v) \notin E$ ist das Gewicht $\theta_{X_u = x, X_v = y} = 0$ für alle $x \in \mathcal{X}_u$ und $y \in \mathcal{X}_v$.

Existiert hingegen eine Kante $(u, v)$ im Abhängigkeitsgraphen, dann gibt es mindestens ein Gewicht $\theta_{X_u=x,X_v=y} > 0$.

Da die Zustandsräume $\mathcal{X}_v$ der Variablen $X_v$ für alle $v \in V$ im Allgemeinen mehr als zwei Ausprägungen enthalten, musste das Problem für die Lösung mithilfe des EA-Optimierers umformuliert werden, sodass es sich mithilfe von binärwertigen Variablen kodieren lässt: Sei $n_v = |\mathcal{X}_v|$ die Größe des Zustandsraums von $X_v$ für alle $v \in V$ und $n = \sum_{v \in V} n_v$, dann lässt sich jede Variable $X_v$ durch eine 1-aus-$n$-Kodierung als Binärvektor $(c_{v,i})_{i \in \{1,\dots,n_v\}}$ darstellen. Die Elemente des Zustandsraums $\mathcal{X}_v$ werden als geordnete Menge $\mathbf{X}_v = \{x_{v,1}, \dots, x_{v,n_v}\}$ angenommen; hat Variable $X_v$ den Wert $x_{v,i}$, dann ist $c_{v,i} = 1$ und $c_{v,j} = 0$ für alle $j \neq i$. Wird nun auch die Knotenmenge $V$ als geordnete Menge $V = \{v_1, \dots, v_k\}$ mit $k = |V|$ angenommen, so entspricht die Kodierung $\tilde{x}$ eines Zustands $x \in \mathcal{X}$ der Konkatenation aller 1-aus-$n$-Kodierungen der Variablenzustände $x_v$:

$$\tilde{x} = (c_{v_1,1}, \dots, c_{v_1,n_{v_1}}, c_{v_2,1}, \dots, c_{v_k,n_{v_k}})$$

Die Hilfsfunktion $\iota : \{1, \dots, n\} \mapsto V \times \mathcal{X}$ sei so definiert, dass $\iota(i) = (v, x_v)$ für einen Index $i$ der kodierten Belegung $\tilde{x}$ den zugehörigen Knoten $v$ und die kodierte Ausprägung $x_v \in \mathcal{X}_v$ zurückgebe. Die QUBO-Zielfunktionskoeffizienten $\beta_{ij}$ ergeben sich dann schließlich aus den entsprechenden Kantengewichten $\theta_{X_u=x,X_v=y}$ mit $\iota(i) = (u, x)$ und $\iota(j) = (v, y)$. Dabei sind zwei Sonderfälle zu beachten, die sich durch die Kodierung ergeben: Ist $i = j$, so ist $\beta_{ii} = 0$, da die Knoten keine Kanten zu sich selbst besitzen. Weiterhin dürfen zwei Binärvariablen, die zur gleichen Kodierung einer Variablen gehören, niemals gleichzeitig 1 werden, da dies zu einer ungültigen Kodierung führen würde, daher muss $\beta_{ij}$ in diesem Fall unendlich groß bzw. zu einem Wert werden, der zwangsläufig zu einer Verschlechterung des Gesamtzielfunktionswerts führt. Um aus dem Maximierungs- ein Minimierungsproblem zu machen, wird das Vorzeichen der Kantengewichte umgekehrt. Insgesamt ergibt sich also folgende Berechnungsvorschrift für die Koeffizienten:

$$\beta_{ij} = \begin{cases} 0 & \text{falls } i = j \\ \infty & \text{falls } i \neq j \text{ und } u = v \\ -\theta_{X_u=x,X_v=y} & \text{sonst} \end{cases} \tag{5.2}$$

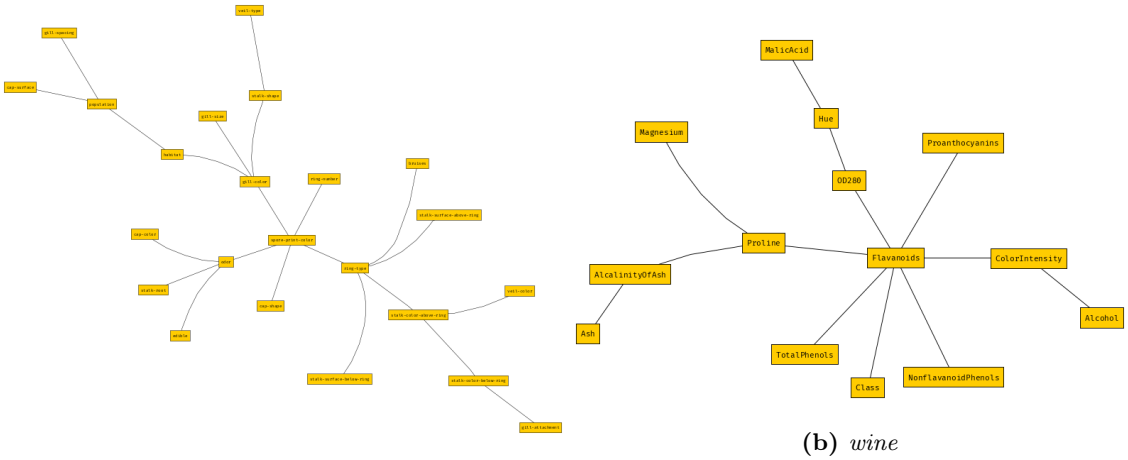
$$\text{mit } \iota(i) = (u, x),$$
$$\iota(j) = (v, y)$$

Das Experiment wurde erneut auf fünf Datensätzen durchgeführt, die in Tabelle 5.4 zusammengefasst sind. Die Werte der Spalte "Ausprägungen" bezieht sich auf die Anzahl *nach* der Diskretisierung. Der *automobile*[10]-Datensatz enthält Charakteristika verschiedener Automodelle sowie eine Bewertung ihres Versicherungsrisikos. Der *mushroom*[11]-Datensatz ist

[10] `https://archive.ics.uci.edu/ml/datasets/automobile`
[11] `https://archive.ics.uci.edu/ml/datasets/mushroom`

| Datensatz | Variablen | davon numerisch | Ausprägungen | Beispiele | Fehlende Werte |
|---|---|---|---|---|---|
| *automobile* | 26 | 15 | 228 | 205 | ja |
| *mushroom* | 23 | 0 | 127 | 8124 | ja |
| *wine* | 14 | 13 | 146 | 178 | nein |
| *iris* | 5 | 4 | 45 | 150 | nein |
| *sonar* | 61 | 60 | 662 | 208 | nein |

**Tabelle 5.4:** Charakteristika der verwendeten Datensätze für das MRF-Experiment




**(a)** *mushroom* **(b)** *wine*

**Abbildung 5.10:** Beispiele für Chow-Liu-Bäume als Abhängigkeitsgraphen

eine umfangreiche Sammlung verschiedener essbarer und nicht-essbarer Pilze, die anhand äußerer Merkmale klassifiziert wurden. Der Datensatz *wine*[12] enthält numerische chemische Messungen verschiedener Weine aus einem Anbaugebiet in Italien. Um noch jeweils einen sehr niedrig- und einen sehr hochdimensionalen Datensatz zu testen, wurden außerdem die Datensätze *iris* und *sonar* erneut verwendet (vgl. Abschnitt 5.3.1), diesmal allerdings inklusive ihrer Klasseninformation.

Mithilfe eines R-Skripts wurden die Koeffizienten nach Gleichung (5.2) aus den von `px-train` berechneten Kantengewichten bestimmt. Beispiele für durch den Chow-Liu-Algorithmus gefundene Abhängigkeitsbaumstrukturen sind in Abb. 5.10 zu sehen. Als Wert anstelle von $\infty$ wurde stets der größte mit dem gewählten $b_\beta$ darstellbare Wert $2^{b_\beta - 1} - 1$ verwendet. Für alle Datensätze außer *iris* wurde $b_\beta = 16$ gewählt, für *iris* aufgrund der kleinen Dimensionalität $b_\beta = 8$. Um Variabilität der Startlösungen bei gleichzeitigem Erhalt einer möglichst hohen Optimierungsgeschwindigkeit durch Parallelisierung zu erreichen, wurde $\mu = 8$ und $\lambda = 24$ gewählt. Der Dimensionsparameter $N$ des Optimierers entspricht der Gesamtzahl (diskretisierter) Ausprägungen $n$. Als Startpopulation wurden für

[12] `https://archive.ics.uci.edu/ml/datasets/wine`

| | Zielfunktionswert | | |
|---|---|---|---|
| Datensatz | *px* | EA | # fehlende Werte |
| *automobile* | −47 | −43 | |
| *mushroom* | −93 | −94 | |
| *wine* | −2 | −5* | 5 von 14 |
| *iris* | −8 | −6* | 1 von 5 |
| *sonar* | −83 | 0* | 61 von 61 |

**Tabelle 5.5:** Zielfunktionswerte der gefundenen Optima (kleine Werte sind besser); mit Sternchen markierte Lösungen sind ungültig, da manche Variablen nicht belegt waren

jeden Datensatz acht gültige Kodierungen zufälliger Variablenbelegungen gewählt und die Optimierung mit dem Standard-Seed für eine Stunde laufen gelassen.

Die Zielfunktionswerte der gefundenen Optima des EA-Optimierers sind in der Spalte "EA" von Tabelle 5.5 aufgeführt, links daneben zum Vergleich die Werte der durch `px-predict` ermittelten wahrscheinlichsten Belegungen für jeden Datensatz, ausgewertet mit der gleichen Zielfunktion. Offensichtlich erzielt *px* insgesamt etwas bessere Ergebnisse; auf dem Datensätzen *automobile*, *iris* und *sonar* sind die Zielfunktionswerte von *px* niedriger, lediglich auf *mushroom* und *wine* konnte der EA-Optimierer bessere Werte erreichen. Das größte Problem, das durch die Ergebnisse offensichtlich wurde, liegt darin, dass die Formulierung der Zielfunktion zulässt, dass manche Variablen überhaupt nicht gesetzt werden: Alle in der Tabelle mit einem Sternchen markierten Lösungen enthielten mindestens eine Variable, deren Kodierung ausschließlich aus Nullen bestand. Daher hat der EA-Optimierer zwar auf *wine* ein scheinbar besseres Optimum mit einem niedrigeren Zielfunktionswert gefunden, aber das Ergebnis ist ungültig, da fünf Variablen nicht gesetzt wurden. Der Grund dafür liegt darin, dass es mit der verwendeten Formulierung nicht möglich ist, einen "Strafwert" für das Fehlen einer 1 festzulegen, da Koeffizienten eines quadratischen Terms nur addiert werden, wenn beide beteiligten Variablen den Wert 1 haben; eine QUBO-Zielfunktion ausgewertet auf einem Nullvektor hat stets den Wert 0. Besonders zum Tragen kam dieses Problem bei der Vorhersage auf dem *sonar*-Datensatz, bei dem keine einzige Variable gesetzt wurde. Als dieses Ergebnis nach einer Stunde Optimierungszeit vorlag, wurde das Experiment auf *sonar* mit einer Optimierungszeit von 12 Stunden und einer neuen zufälligen Startpopulation wiederholt – das Resultat war identisch.

Neben dem Problem der fehlenden Belegungen arbeitet der Optimierer auf dieser Zielfunktion offenbar nicht effizient: Bei jeder Mutation beträgt die erwartete Zahl invertierter Bits genau 1 (vgl. Abschnitt 3.1.1), und wird genau ein Bit einer gültigen Lösung geändert, führt dies grundsätzlich zu einer ungültigen Lösung: Entweder wird eine 1 in einen gültigen 1-aus-n-Vektor eingefügt, sodass dieser danach aus zwei Einsen besteht, was nicht erlaubt

ist, oder die 1 eines 1-aus-n-Vektors wird zu einer 0, sodass ein Nullvektor zurückbleibt, der (theoretisch) ebenfalls keine gültige Lösung darstellt. Der kleinste Schritt zu einer gültigen Lösung besteht in der Invertierung genau einer 1 und einer 0 innerhalb der Kodierung derselben Variable. Die Wahrscheinlichkeit für dieses Ereignis bei $k$ Variablen liegt bei

$$(n-k) \cdot \left(\frac{1}{n}\right)^2 \cdot \left(\frac{n-1}{n}\right)^{n-2}, \tag{5.3}$$

was für den kleinsten Datensatz *iris* bereits eine Wahrscheinlichkeit von nur etwa 0.75% und für den größten Datensatz *sonar* sogar lediglich 0.05% ergibt. Dies hat für letzteren zur Folge, dass nur etwa jede 2000. Mutation überhaupt zu einer gültigen Variablenbelegung führt, was bei $\lambda = 24$ bedeutet, dass im Schnitt nur alle 82 Iterationen eine neue gültige Lösung auftritt. Wie aus Gleichung (5.3) hervorgeht, ist die Wahrscheinlichkeit einer gültigen Mutation am höchsten für kleine $n$ und $k$, was erklärt, dass der Optimierer auf den Datensätzen mit vergleichsweise wenigen Variablen und Ausprägungen tendenziell bessere Ergebnisse erzielt, auf dem *sonar*-Datensatz hingegen unbrauchbar ist.

Dass die Veränderung nur *einer* Variablen zu einer Verbesserung führt, wird im Verlauf der Optimierung weiterhin immer unwahrscheinlicher; vielmehr müssten zwei oder mehr Variablen gleichzeitig verändert werden, um eine Konfiguration zu erreichen, die eine Verbesserung des Zielfunktionswerts bewirkt. Aus diesem Grund ist die Optimierung mit dem verwendeten Mutationsoperator – selbst auf einem FPGA – äußerst ineffizient. Eine angepasste Formulierung der QUBO-Zielfunktion sowie die Verwendung einer höheren Mutationswahrscheinlichkeit oder gar eines komplett anderen Mutationsoperators, der gültige Lösungen weniger leicht verlässt, könnte für das MAP-Problem auf MRFs sicherlich bessere Lösungen liefern.

# Kapitel 6

# Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Optimierer entwickelt, der auf FPGAs läuft und mithilfe der evolutionären Optimierungsstrategie des $(\mu+\lambda)$-EAs ohne Rekombination das QUBO-Problem mit der vorhandenen Hardware auf bis zu 1024 binärwertigen Variablen lösen kann. Die Implementierung in VHDL erlaubt durch generische Parameter eine freie Anpassung der Eltern- ($\mu$) und Nachkommenpopulationsgröße ($\lambda$), Dimension ($N$) und Bit-Breite der Koeffizienten ($b_\beta$), außerdem die Verwendung beliebiger Zielfunktionskoeffizienten, Startpopulationen und Seeds ohne Neuprogrammierung des FPGAs. Der Ising-Modus ermöglicht zudem die Optimierung des eng mit QUBO verwandten Ising-Modells und eröffnet dadurch die Möglichkeit der Optimierung vieler weiterer Problemformulierungen.

In Kapitel 3 wurde das Konzept des Optimierers mit Begründung für die einzelnen Design-Entscheidungen erläutert, aus dem in Kapitel 4 die Implementierung in der Hardwarebeschreibungssprache VHDL hergeleitet wurde. Der fertig implementierte Optimierer wurde in Kapitel 5 auf seine Geschwindigkeit und seinen Platzverbrauch auf dem FPGA-Chip abhängig von den gewählten Parametern hin untersucht und die Funktionsfähigkeit anhand von zwei Experimenten demonstriert, in denen gängige Optimierungsprobleme aus der Praxis des Maschinellen Lernens – das Maximum-A-Posteriori-Problem auf Markov-Random-Fields und das Clustering-Problem mit zwei Clusterzentren – in ein QUBO-Problem bzw. ein Ising-Modell umformuliert und mit dem Optimierer gelöst wurden. Während die Güte der Lösung des Clustering-Problems, vor allem auf hochdimensionalen Daten, mit der des Lloyd-Algorithmus' vergleichbar war und ein stabiles Optimum in unter einer Sekunde erreicht wurde, bleibt die Leistungsfähigkeit des Optimierers auf dem MAP-Problem für MRFs zumeist hinter den Vergleichslösung zurück. In beiden Fällen ist die Formulierung der Zielfunktion das größte Hindernis, die im Falle des 2-Means-Experiments durch vereinfachende Annahmen zu nicht-optimalen Ergebnissen führt und im Falle des MAP-Problems auf MRFs sowohl unzulässige Lösungen erlaubt als auch schlecht für den Mutationsoperator des EA-Optimierers geeignet ist, was die Optimierungsgeschwindigkeit drastisch verringert.

An der Implementierung des Optimierers gibt es noch viele Möglichkeiten zur Verbesserung der Resourceneffizienz, die im Rahmen dieser Arbeit nicht mehr umgesetzt wurden: Der größte Flaschenhals, sowohl bezüglich des Verbrauchs von FPGA-Elementen als auch der asymptotischen Laufzeit, ist die Zielfunktionsauswertung, die mit $\mathcal{O}(N^2)$ Taktzyklen den Großteil der Optimierungszeit ausmacht. Die ebenfalls zur Dimension quadratische Anzahl Koeffizienten limitiert maßgeblich die maximale Dimensionalität des Optimierungsproblems, das mit einer gegebenen Hardwarekonfiguration gelöst werden kann. Ein lohnenswertes Ziel der Weiterentwicklung des EA-Optimierers liegt daher sicherlich in einer Verbesserung des `evaluator`-Moduls und des `ram`-Speichers. Mögliche Strategien sind beispielsweise eine verbesserte Parallelisierung der Koeffizientenverarbeitung (mehrere Koeffizienten pro Taktzyklus statt ein einziger) sowie eine Auslagerung der Koeffizienten auf größere Speichermedien, sodass noch höherdimensionale Probleme gelöst werden können. Die langsamere Lesegeschwindigkeit solcher Speichermedien wie *Distributed* RAM (DRAM) muss dabei durch geschickte Pufferung der Koeffizienten ausgeglichen werden. Eine weitere Verbesserung liegt in der Verwendung von DSP-Slices zur Berechnung der Summe von Koeffizienten, sodass dafür weniger LUT-Resourcen verbraucht werden müssen.

Bezüglich der umgesetzten Evolutionsstrategie ist eine Vielzahl von Varianten möglich: Beispielsweise [28] vermittelt einen Eindruck von der Fülle möglicher Strategieparameter von EAs, die im Rahmen dieser Arbeit nicht mehr betrachtet wurden. Im Lichte der Ergebnisse des MRF-Experiments ist die Implementierung weiterer, spezialisierterer Mutationsstrategien ein lohnenswertes Weiterentwicklungsziel. Einerseits könnte durch eine Anhebung der Mutationswahrscheinlichkeit, z. B. auf $\log(N)/N$ pro Bit, versucht werden, den Anteil gültiger Nachkommen zu erhöhen und so, am konkreten Beispiel des Experiments, auf den Datensätzen *automobile* und vor allem *sonar* bessere Resultate zu erzielen. Andererseits ist für die Optimierung des MAP-Problems ein ganz andersartiger Operator denkbar, der lediglich Bits vertauscht, statt sie zu invertieren, sodass der Lösungsraum der gültigen 1-aus-n-Kodierungen effizienter durchsucht werden kann. Auch eine Kombination mehrerer Mutationsoperatoren ist denkbar, die durch weitere Systemparameter gewichtet werden können.

Um das MAP-Problem auch mit festen Belegungen einiger Variablen lösen zu können, wäre eine weitere nützliche Erweiterung eine Bitmaske, die bestimmte Bits fixiert und verhindert, dass diese mutiert werden. Anhand des Datensatzes *automobile* aus Abschnitt 5.3.2 würde dies beispielsweise bedeuten, dass eine Bitmaske, die die ersten 6 Bits der Lösungsvektoren fixiert, den Wert der Variablen `symboling` festhalten würde, sodass die bedingte wahrscheinlichste Belegung aller anderen Variablen approximiert wird.

Neben der Lösung des MAP-Problems ist auch die Verwendung des Optimierers zur Näherung der Normierungskonstante $Z$ denkbar [11].

Die Minimierung pseudoboolescher Funktionen ist eine einfache Problemstellung, die ein überraschend weites Feld von möglichen Anwendungen in vielen Bereichen des Maschinellen Lernens eröffnet. Ein hardwarebasierter Optimierer, speziell auf diese Problemklasse zugeschnitten, ist eine interessante und vielversprechende Erweiterung des Spektrums existierender Implementierungen, die weiter zu untersuchen und zu entwickeln sicherlich lohnenswert ist.

# Literaturverzeichnis


[1] Bäck, Thomas, Günter Rudolph und Hans-Paul Schwefel: *Evolutionary programming and evolution strategies: Similarities and differences.* In: *In Proceedings of the Second Annual Conference on Evolutionary Programming.* Citeseer, 1993.

[2] Batcher, Kenneth E: *Sorting networks and their applications.* In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, Seiten 307–314. ACM, 1968.

[3] Bauckhage, Christian, Cesar Ojeda, Rafet Sifa und Stefan Wrobel: *Adiabatic Quantum Computing for Kernel k= 2 Means Clustering.*

[4] Blackman, David und Sebastiano Vigna: *Scrambled Linear Pseudorandom Number Generators.* arXiv preprint arXiv:1805.01407, 2018.

[5] Boros, Endre und Peter L Hammer: *Pseudo-boolean optimization.* Discrete applied mathematics, 123(1-3):155–225, 2002.

[6] Chow, C und Cong Liu: *Approximating discrete probability distributions with dependence trees.* IEEE transactions on Information Theory, 14(3):462–467, 1968.

[7] Davis, Lawrence: *Handbook of genetic algorithms.* 1991.

[8] Dempster, Arthur P, Nan M Laird und Donald B Rubin: *Maximum likelihood from incomplete data via the EM algorithm.* Journal of the Royal Statistical Society: Series B (Methodological), 39(1):1–22, 1977.

[9] Denchev, Vasil S, Nan Ding, SVN Vishwanathan und Hartmut Neven: *Robust classification with adiabatic quantum optimization.* arXiv preprint arXiv:1205.1148, 2012.

[10] Droste, Stefan, Thomas Jansen und Ingo Wegener: *On the analysis of the (1+ 1) evolutionary algorithm.* Theoretical Computer Science, 276(1-2):51–81, 2002.

[11] Ermon, Stefano, Carla Gomes, Ashish Sabharwal und Bart Selman: *Taming the curse of dimensionality: Discrete integration by hashing and optimization.* In: *International Conference on Machine Learning*, Seiten 334–342, 2013.

[12] Fernando, Pradeep R, Srinivas Katkoori, Didier Keymeulen, Ricardo Zebulum und Adrian Stoica: *Customizable FPGA IP core implementation of a general-purpose genetic algorithm engine.* IEEE Transactions on Evolutionary Computation, 14(1):133–149, 2010.

[13] Fogel, David B: *An introduction to simulated evolutionary optimization.* IEEE transactions on neural networks, 5(1):3–14, 1994.

[14] Hansen, Nikolaus und Andreas Ostermeier: *Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation.* In: *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, Seiten 312–317. IEEE, 1996.

[15] Jansen, Thomas und Ingo Wegener: *Real royal road functions—where crossover provably is essential.* Discrete applied mathematics, 149(1-3):111–125, 2005.

[16] Lloyd, Stuart: *Least squares quantization in PCM.* IEEE transactions on information theory, 28(2):129–137, 1982.

[17] Lucas, Andrew: *Ising formulations of many NP problems.* Frontiers in Physics, 2:5, 2014.

[18] Mitchell, Melanie, Stephanie Forrest und John H Holland: *The royal road for genetic algorithms: Fitness landscapes and GA performance.* In: *Proceedings of the first european conference on artificial life*, Seiten 245–254, 1992.

[19] Neven, Hartmut, Vasil S Denchev, Geordie Rose und William G Macready: *Training a binary classifier with the quantum adiabatic algorithm.* arXiv preprint arXiv:0811.0416, 2008.

[20] Neven, Hartmut, Geordie Rose und William G Macready: *Image recognition with an adiabatic quantum computer I. Mapping to quadratic unconstrained binary optimization.* arXiv preprint arXiv:0804.4457, 2008.

[21] O'Gorman, Bryan, R Babbush, A Perdomo-Ortiz, Alán Aspuru-Guzik und Vadim Smelyanskiy: *Bayesian network structure learning using quantum annealing.* The European Physical Journal Special Topics, 224(1):163–188, 2015.

[22] Piatkowski, Nico: *Exponential families on resource-constrained systems.* Doktorarbeit, Technical University of Dortmund, Germany, 2018.

[23] Piatkowski, Nico, Sangkyun Lee und Katharina Morik: *Integer undirected graphical models for resource-constrained systems.* Neurocomputing, 173:9–23, 2016.

[24] Rønnow, Troels F, Zhihui Wang, Joshua Job, Sergio Boixo, Sergei V Isakov, David Wecker, John M Martinis, Daniel A Lidar und Matthias Troyer: *Defining and detecting quantum speedup.* Science, 345(6195):420–424, 2014.

[25] Rudolph, Günter: *Convergence of non-elitist strategies.* In: *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, Seiten 63–66. IEEE, 1994.

[26] Rukhin, Andrew, Juan Soto, James Nechvatal, Miles Smid und Elaine Barker: *A statistical test suite for random and pseudorandom number generators for cryptographic applications.* Technischer Bericht, Booz-Allen and Hamilton Inc Mclean Va, 2001.

[27] Santoro, Giuseppe E und Erio Tosatti: *Optimization using quantum mechanics: quantum annealing through adiabatic evolution.* Journal of Physics A: Mathematical and General, 39(36):R393, 2006.

[28] Schwefel, Hans-Paul und Günter Rudolph: *Contemporary evolution strategies.* In: *European conference on artificial life*, Seiten 891–907. Springer, 1995.

[29] Scott, Stephen D, Ashok Samal und Shared Seth: *HGA: A hardware-based genetic algorithm.* In: *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, Seiten 53–59. ACM, 1995.

[30] Shackleford, Barry, Greg Snider, Richard J Carter, Etsuko Okushi, Mitsuhiro Yasuda, Katsuhiko Seo und Hiroto Yasuura: *A high-performance, pipelined, FPGA-based genetic algorithm machine.* Genetic Programming and Evolvable Machines, 2(1):33–60, 2001.

[31] Steane, Andrew: *Quantum computing.* Reports on Progress in Physics, 61(2):117, 1998.

[32] Tommiska, Matti und Jarkko Vuori: *Hardware implementation of GA.* In: *Proceedings of the Second Nordic Workshop on Genetic Algorithms and their Applications (2NWGA), Vaasa, Finland*, 1996.

[33] Torquato, Matheus F. und Marcelo A. C. Fernandes: *High-Performance Parallel Implementation of Genetic Algorithm on FPGA.* Circuits, Systems, and Signal Processing, Jan 2019.

[34] Wainwright, Martin J, Michael I Jordan et al.: *Graphical models, exponential families, and variational inference.* Foundations and Trends® in Machine Learning, 1(1–2):1–305, 2008.

# Anhang A

# Quelltext

## A.1 Python-Implementierung

```
1  import BitVector as BV
2  import random
3
4  DIM        = 100
5  N_PARENTS  = 1
6  N_CHILDREN = 4
7  BUDGET     = 1000
8
9  def oneMax(dim):
10     for i in range(dim):
11         for j in range(i+1):
12             yield -1 if i == j else 0
13
14 COEFS = [ c for c in oneMax(DIM) ]
15
16 def mutated(v):
17     w = v[:]
18     n = len(v)
19     for i in range(n):
20         if random.random() < 1.0/n:
21             w[i] = 1 - v[i]
22     return w
23
24 def evaluate(v):
25     s = 0
26     for i in range(DIM):
27         for j in range(i+1):
28             if v[i] and v[j]:
29                 s += COEFS[int(i*(i+1)/2) + j]
30     return s
31
32 def main():
33
34     population = [ BV.BitVector(size=DIM) for i in range(N_PARENTS + N_CHILDREN) ]
35     lossvalues = [ 0 ] * (N_PARENTS + N_CHILDREN)
36
37     budget = BUDGET
38     while budget > 0:
39         # mutate random parents to create child population
```

```
        population[N_PARENTS:] = [ mutated(v) for v in random.choices(
            population[:N_PARENTS], k=N_CHILDREN) ]

        # evaluate children
        lossvalues[N_PARENTS:] = [ evaluate(v) for v in population[N_PARENTS:] ]

        # sort by loss value
        sort_index = [ i for i, e in sorted(enumerate(lossvalues), key=lambda x: x[1]) ]

        # use best as new parent population
        population[:N_PARENTS] = [ population[i] for i in sort_index[:N_PARENTS] ]
        lossvalues[:N_PARENTS] = [ lossvalues[i] for i in sort_index[:N_PARENTS] ]

        budget -= 1

    print("opt: ", population[0].get_bitvector_in_hex())
    print("loss:", lossvalues[0])

if __name__ == "__main__":
    main()
```

## A.2 *main.vhd*

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.ea.all;

entity main is
    Generic (
        DIM        : integer := 8;
        N_PARENTS  : integer := 4;
        N_CHILDREN : integer := 12;
        BUDGET     : integer := -1;
        COEF_WIDTH : integer := 32;

        -- dependent
        N_POPULATION : integer := 16; -- expr {$N_PARENTS+$N_CHILDREN}
        NEXT_POWER   : integer := 4;  -- expr {ceil(log(double($N_POPULATION))/log(2))}
        DATA_WIDTH   : integer := 128; -- expr {max($DIM, 128)}
        N_COEF       : integer := 36  -- expr {($DIM*($DIM+1))/2}
    );
    Port (
        clk         : in std_logic;
        reset       : in std_logic;

        config      : in std_logic_vector(1 downto 0);
        data_ready  : in std_logic;
        data_in     : in std_logic_vector (DATA_WIDTH-1 downto 0);
        start       : in std_logic;

        config_mode : out std_logic; -- indicate that the device is reset and in config
        ↪ mode
        data_read   : out std_logic; -- indicate that data_in was successfully read
        full        : out std_logic; -- indicate that either all coefs or parents where
        ↪ set
```

```
34
35          done        : out std_logic;
36          opt         : out std_logic_vector (DIM-1 downto 0);
37          opt_loss    : out std_logic_vector (31 downto 0)
38      );
39  end main;
40
41  architecture Behavioral of main is
42
43      type population_t is array (natural range <>) of std_logic_vector(DIM-1 downto 0);
44
45      constant SEED128 : std_logic_vector(127 downto 0) :=
        ↪  x"27404ad634a10951f37d4b657c96703f";
46
47      -- max index of component that has to be initialized with a random seed
48      constant INIT_MAX_IX : integer := N_CHILDREN+1;
49
50      -- saves coefficients in Block RAM
51      component ram is
52          generic (
53              DATA_SIZE : integer := 8;
54              SIZE      : integer := 128
55          );
56          port (
57              clk      : in std_logic;
58              reset    : in std_logic;
59              address  : in integer range 0 to SIZE-1;
60              we       : in std_logic;
61              data_in  : in std_logic_vector(DATA_SIZE-1 downto 0);
62              data_out : out std_logic_vector(DATA_SIZE-1 downto 0)
63          );
64      end component;
65
66      -- provides initial seeds and start population
67      component rng_xoroshiro128plus is
68          generic (
69              init_seed:  std_logic_vector(127 downto 0)
70          );
71          port (
72              clk:        in  std_logic;
73              rst:        in  std_logic;
74              reseed:     in  std_logic;
75              newseed:    in  std_logic_vector(127 downto 0);
76              out_ready:  in  std_logic;
77              out_valid:  out std_logic;
78              out_data:   out std_logic_vector(63 downto 0)
79          );
80      end component;
81
82      -- mutates parents
83      component mutator is
84          Generic (
85              DIM       : integer;
86              N_PARENTS : integer
87          );
88          Port (
89              clk        : in  std_logic;
90              reset      : in  std_logic;
91              in_reseed  : in  std_logic;
92              in_newseed : in  std_logic_vector(127 downto 0);
```

```
93              in_vec     : in  std_logic_vector(DIM-1 downto 0);
94              ready      : in  std_logic;
95              parent_ix  : out integer range 0 to N_PARENTS-1;
96              valid      : out std_logic;
97              out_vec    : out std_logic_vector(DIM-1 downto 0)
98          );
99      end component;
100
101     -- evaluates children
102     component evaluator is
103         generic (
104             DIM        : integer
105         );
106         port (
107             clk        : in std_logic;
108             reset      : in std_logic;
109             isg_mode   : in std_logic;
110             vect_in    : in std_logic_vector (DIM-1 downto 0);
111             coef_ready : in std_logic;
112             coef_in    : in loss_t;
113             coef_read  : out std_logic;
114             valid      : out std_logic;
115             result     : out loss_t
116         );
117     end component;
118
119     -- sorts by loss value
120     component sorter is
121         generic (
122             N : positive := 1
123         );
124         port (
125             clk   : in std_logic;
126             reset : in std_logic;
127             data  : in loss_array((2**N)-1 downto 0);
128             valid : out std_logic;
129             order : out index_array((2**N)-1 downto 0)
130         );
131     end component;
132
133     -- convert from std_logic_vector to loss_t
134     function SLV_to_coef (inp : std_logic_vector)
135         return loss_t is
136     begin
137         return to_integer(signed(inp(COEF_WIDTH-1 downto 0)));
138     end SLV_to_coef;
139
140     function vector_and(vec : std_logic_vector)
141         return std_logic is
142         variable res : std_logic := '1';
143     begin
144         for ix in 0 to vec'length-1 loop
145             res := res and vec(ix);
146         end loop;
147         return res;
148     end vector_and;
149
150     -- cycle std_logic_vector (downto) one bit to the left
151     function cycle_left (vec : std_logic_vector)
152         return std_logic_vector is
```

```
153     begin
154         return vec(vec'high-1 downto vec'low) & vec(vec'high);
155     end cycle_left;
156 
157     -- global EA status
158     signal globalValid      : std_logic                          := '0';
159     signal status           : GlobalStatus                       := CONF;
160     signal remaining_budget : integer range -1 to integer'HIGH := BUDGET;
161 
162     -- population and loss arrays must be the size of a power of two for the sorting
        ↪  component to work
163     signal population   : population_t (2**NEXT_POWER-1 downto 0) := (others => (others =>
        ↪  '0'));
164     signal lossvalues   : loss_array (2**NEXT_POWER-1 downto 0) := (others =>
        ↪  integer'HIGH);
165 
166     alias parents       : population_t (N_PARENTS-1 downto 0) is population (N_PARENTS-1
        ↪  downto 0);
167     alias children      : population_t (N_CHILDREN-1 downto 0) is population
        ↪  (N_POPULATION-1 downto N_PARENTS);
168     alias parents_loss  : loss_array (N_PARENTS-1 downto 0) is lossvalues (N_PARENTS-1
        ↪  downto 0);
169     alias children_loss : loss_array (N_CHILDREN-1 downto 0) is lossvalues (N_POPULATION-1
        ↪  downto N_PARENTS);
170 
171     -- index signals
172     signal next_coef_ix : integer range 0 to N_COEF-1   := 0;
173     signal parent_ix    : integer range 0 to N_PARENTS-1 := 0;
174 
175     -- signals for the coefficient storage
176     signal coef_address  : integer range 0 to N_COEF-1 := 0;
177     signal coef_we       : std_logic := '0';
178     signal coef_data_in  : std_logic_vector(COEF_WIDTH-1 downto 0);
179     signal coef_data_out : std_logic_vector(COEF_WIDTH-1 downto 0);
180 
181     -- signals for the CONF phase
182     signal parents_full    : std_logic    := '0';
183     signal coefs_full      : std_logic    := '0';
184 
185     -- signals for the INIT phase
186     signal INIT_reseeds         : std_logic_vector(INIT_MAX_IX downto 0)   := (0 => '1',
        ↪  others => '0');
187     alias  INIT_mutator_reseeds : std_logic_vector (N_CHILDREN-1 downto 0) is INIT_reseeds
        ↪  (N_CHILDREN downto 1);
188     signal INIT_newseed         : std_logic_vector(127 downto 0);
189     signal INIT_seed_ready  : std_logic := '0';
190     signal INIT_seed_hi     : std_logic := '0';
191     signal INIT_skip_mutate : std_logic := '0'; -- set to '1' if initial population was
        ↪  set
192 
193     signal rng_reseed : std_logic := '0';
194     signal rng_seed   : std_logic_vector(127 downto 0) := SEED128;
195     signal rng_ready  : std_logic := '1';
196     signal rng_valid  : std_logic;
197     signal rng_data   : std_logic_vector(63 downto 0);
198 
199     -- signals for the MUTATE phase
200     signal MUTATE_reset : std_logic := '1';
201     signal MUTATE_ready      : std_logic := '0';
202     signal MUTATE_reseeds    : std_logic_vector(N_CHILDREN-1 downto 0);
```

```
203     signal MUTATE_parent_ixs : index_array (N_CHILDREN-1 downto 0);
204     signal MUTATE_parents    : population_t (N_CHILDREN-1 downto 0);
205     signal MUTATE_valids     : std_logic_vector (N_CHILDREN-1 downto 0) := (others =>
        ↪  '0');
206     signal MUTATE_res        : population_t (N_CHILDREN-1 downto 0);
207
208     -- signals for the EVALUATE phase
209     signal EVAL_reset      : std_logic := '1';
210     signal EVAL_isg_mode   : std_logic := '0';
211     signal EVAL_coef_ready : std_logic := '0';
212     signal EVAL_valids     : std_logic_vector (N_CHILDREN-1 downto 0) := (others => '0');
213     signal EVAL_results    : loss_array (N_CHILDREN-1 downto 0);
214
215     -- signals for the SORT phase
216     signal SORT_reset : std_logic := '1';
217     signal SORT_valid : std_logic;
218     signal SORT_order : index_array (2**NEXT_POWER-1 downto 0);
219
220 begin
221
222     RAM0 : ram generic map (
223         DATA_SIZE => COEF_WIDTH,
224         SIZE      => N_COEF
225     )
226     port map (
227         clk      => clk,
228         reset    => reset,
229         address  => coef_address,
230         we       => coef_we,
231         data_in  => coef_data_in,
232         data_out => coef_data_out
233     );
234
235     RND0 : rng_xoroshiro128plus generic map (
236         init_seed => SEED128
237     )
238     port map (
239         clk       => clk,
240         rst       => '0',
241         reseed    => rng_reseed,
242         newseed   => rng_seed,
243         out_ready => rng_ready,  -- (in)
244         out_valid => rng_valid,  -- (out)
245         out_data  => rng_data    -- (out)
246     );
247
248     mutator_blocks : for mut_ix in N_CHILDREN-1 downto 0 generate
249         MUT0 : mutator
250         generic map (DIM, N_PARENTS)
251         port map (
252             clk        => clk,
253             reset      => MUTATE_reset,
254             in_reseed  => MUTATE_reseeds(mut_ix),
255             in_newseed => INIT_newseed,
256             in_vec     => MUTATE_parents(mut_ix),
257             ready      => MUTATE_ready,
258             parent_ix  => MUTATE_parent_ixs(mut_ix),
259             valid      => MUTATE_valids(mut_ix),
260             out_vec    => MUTATE_res(mut_ix)
261         );
```

```
262     end generate mutator_blocks;
263
264     evaluator_blocks : for eval_ix in N_CHILDREN-1 downto 0 generate
265         EVAL0 : evaluator
266         generic map (DIM)
267         port map (
268             clk        => clk,
269             reset      => EVAL_reset,
270             isg_mode   => EVAL_isg_mode,
271             vect_in    => children(eval_ix),
272             coef_ready => EVAL_coef_ready,
273             coef_in    => SLV_to_coef(coef_data_out),
274             coef_read  => open,
275             valid      => EVAL_valids(eval_ix),
276             result     => EVAL_results(eval_ix)
277         );
278     end generate evaluator_blocks;
279
280     SORT0 : sorter
281         generic map (NEXT_POWER)
282         port map (
283             clk   => clk,
284             reset => SORT_reset,
285             data  => lossvalues,
286             valid => SORT_valid,
287             order => SORT_order
288         );
289
290     -- /* asynchronous processes */
291     config_mode <= '1' when status = CONF else '0';
292
293     with config select full <= coefs_full   when "00",
294                                parents_full when "01",
295                                '0'          when others;
296
297     MUT_PORTS : for ix in 0 to N_CHILDREN-1 generate
298         -- actual reseed input for each mutator
299         MUTATE_reseeds(ix) <= INIT_mutator_reseeds(ix) and INIT_seed_ready;
300         -- parent to mutate for each mutator
301         MUTATE_parents(ix) <= parents(MUTATE_parent_ixs(ix));
302     end generate;
303
304     -- current optimum outputs
305     done     <= globalValid;
306     opt      <= population(0);
307     opt_loss <= std_logic_vector(to_signed(lossvalues(0),32));
308
309     main : process (clk)
310     begin
311         if rising_edge(clk) then
312             if reset = '1' then
313                 -- reset everything
314
315                 population   <= (others => (others => '0'));
316                 lossvalues   <= (others => integer'HIGH);
317
318                 coef_we <= '1';
319
320                 parents_full <= '0';
321                 parent_ix    <= 0;
```

```
322                 coefs_full   <= '0';
323                 next_coef_ix <= 0;
324
325                 INIT_reseeds    <= (0 => '1', others => '0');
326                 INIT_seed_ready <= '0';
327                 INIT_seed_hi    <= '0';
328
329                 rng_seed   <= SEED128;
330                 rng_reseed <= '1';
331                 rng_ready  <= '0';
332
333                 MUTATE_reset <= '1';
334                 MUTATE_ready <= '0';
335
336                 EVAL_reset      <= '1';
337                 EVAL_isg_mode   <= '0';
338                 EVAL_coef_ready <= '0';
339
340                 SORT_reset <= '1';
341
342                 -- reset output ports
343                 data_read   <= '0';
344                 globalValid <= '0';
345
346                 -- reset globla states
347                 status           <= CONF;
348                 remaining_budget <= BUDGET;
349
350             else
351                 case status is
352                     when CONF =>
                    ↪  ------------------------------------------------------------ CONF
                    ↪  phase
353
354                         if data_ready = '1' then
355
356                             case config is
357                             when "00" => -- INPUT COEFFICIENTS
358                                 if coefs_full = '0' then
359                                     -- input polynomial coefficients
360
361                                     coef_address <= next_coef_ix;
362                                     coef_data_in <= data_in(COEF_WIDTH-1 downto 0);
363                                     data_read    <= '1';
364
365                                     if next_coef_ix >= N_COEF-1 then
366                                         coefs_full <= '1';
367                                     else
368                                         next_coef_ix <= next_coef_ix + 1;
369                                     end if;
370                                 else
371                                     data_read <= '0';
372                                 end if;
373
374                             when "01" => -- INPUT PARENTS
375                                 if parents_full = '0' then
376                                     -- initialize parent population
377
378                                     -- Skip mutate phase after initialization, so that
379                                     -- parent vectors are evaluated first
```

```
380                                 INIT_skip_mutate <= '1';
381
382                                 -- interpret data as parent vector;
383                                 -- save parents in *child population* so that they can
                                    ↪  be evaluated first and then be
384                                 -- sorted into the parent population with their
                                    ↪  correct loss values; otherwise they
385                                 -- would be overwritten by their children
386                                 children(parent_ix) <= data_in (DIM-1 downto 0);
387                                 data_read <= '1';
388
389                                 if parent_ix >= N_PARENTS-1 then
390                                     parents_full <= '1';
391                                 else
392                                     parent_ix <= parent_ix + 1;
393                                 end if;
394                             else
395                                 data_read <= '0';
396                             end if;
397
398                         when "10" => -- INPUT RANDOM SEED
399                             rng_seed  <= data_in(127 downto 0);
400                             data_read <= '1';
401
402                         when "11" => -- INPUT ISG MODE
403                             EVAL_isg_mode <= data_in(0);
404                             data_read     <= '1';
405
406                         when others =>
407                             data_read <= '0';
408
409                         end case;
410                     end if;
411
412                     if start = '1' then
413                         coef_we <= '0';
414
415                         -- prepare for entering INIT
416                         rng_reseed <= '0';
417                         rng_ready  <= '1';
418                         status <= INIT;
419                     else
420                         coef_we <= '1';
421                     end if;
422
423                 when INIT =>
                    ↪  ------------------------------------------------------------ INIT
                    ↪  phase
424
425                     -- check if last component has been reseeded
426                     if INIT_reseeds(INIT_MAX_IX) = '1' then
427                         -- prepare for leaving
428                         rng_ready <= '0';
429
430                         -- check if
431                         if INIT_skip_mutate = '1' then
432                             -- prepare for entering EVALUATE
433                             coef_address <= 0;
434                             status       <= EVALUATE;
435                         else
```

```
436                         -- prepare for entering MUTATE
437                         MUTATE_reset <= '0';
438                         MUTATE_ready <= '1';
439                         status <= MUTATE;
440                     end if;
441
442
443                 -- random generator yields 64 bits, but 128 are needed for one
                    ↪  seed,
444                 -- so a new seed has to be generated in two passes
445                 elsif rng_valid = '1' then
446                     if INIT_seed_hi = '0' then
447                         INIT_seed_ready <= '0';
448                         -- next index to be reseeded
449                         INIT_reseeds <= cycle_left(INIT_reseeds);
450                         -- generate low 64 bits
451                         INIT_newseed(63 downto 0) <= rng_data;
452                         INIT_seed_hi <= '1';
453                     else
454                         -- generate high 64 bits
455                         INIT_newseed(127 downto 64) <= rng_data;
456                         INIT_seed_hi <= '0';
457                         INIT_seed_ready <= '1';
458                     end if;
459                 end if;
460
461             when MUTATE =>
                ↪  ------------------------------------------------------- MUTATE
                ↪  phase
462
463                 -- make sure the last component is not continually reseeding
464                 INIT_seed_ready <= '0';
465
466                 -- wait for all mutators to finish
467                 if vector_and(MUTATE_valids) = '1' then
468                     -- save mutation results
469                     children <= MUTATE_res;
470
471                     -- prepare for leaving MUTATE
472                     MUTATE_reset <= '1';
473                     MUTATE_ready <= '0';
474                     -- prepare for entering EVALUATE
475                     coef_address <= 0;
476                     status       <= EVALUATE;
477                 end if;
478
479             when EVALUATE =>
                ↪  ------------------------------------------------------ EVALUATE
                ↪  phase
480
481                 if coef_address < N_COEF-1 then
482                     EVAL_reset      <= '0';
483                     EVAL_coef_ready <= '1';
484                     coef_address    <= coef_address + 1;
485                 else
486                     -- wait for all evaluators to finish
487                     if vector_and(EVAL_valids) = '1' then
488                         -- save evaluation results
489                         children_loss <= EVAL_results;
490
```

```
491                                 -- prepare for leaving EVALUATE
492                                 EVAL_reset <= '1';
493                                 EVAL_coef_ready <= '0';
494                                 -- prepare for entering SORT
495                                 status <= SORT;
496                             end if;
497                         end if;
498
499                     when SORT =>
                        ↪  ------------------------------------------------------------ SORT
                        ↪  phase
500
501                         if SORT_reset = '1' then
502                             -- wait for sorter to fully reset!
503                             -- if reset is set to '0' while valid is still '1' from the
504                             -- previous sorting pass, the resulting order will be invalid!
505                             if SORT_valid = '0' then
506                                 SORT_reset <= '0';
507                             end if;
508                         else
509                             if SORT_valid = '1' then
510                                 -- move N_PARENTS best individuals to parent population
511                                 SELECT_PARENTS : for ix in 0 to N_PARENTS-1 loop
512                                     population(ix) <= population(SORT_order(ix));
513                                     lossvalues(ix) <= lossvalues(SORT_order(ix));
514                                 end loop SELECT_PARENTS;
515
516                                 -- prepare for leaving SORT
517                                 SORT_reset <= '1';
518
519                                 if remaining_budget = 0 then
520                                     -- prepare for entering FINISHED
521                                     status <= FINISHED;
522                                 else
523                                     if remaining_budget > 0 then
524                                         remaining_budget <= remaining_budget-1;
525                                     end if;
526
527                                     -- prepare for entering MUTATE
528                                     MUTATE_reset <= '0';
529                                     MUTATE_ready <= '1';
530                                     status <= MUTATE;
531                                 end if;
532                             end if;
533                         end if;
534
535                     when FINISHED =>
                        ↪  ------------------------------------------------------- FINISHED
                        ↪  phase
536
537                         globalValid <= '1';
538
539                 end case;
540             end if;
541         end if;
542     end process;
543
544 end Behavioral;
```

## A.3 *ram.vhd*

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity ram is
5      generic (
6          DATA_SIZE : integer := 8;
7          SIZE      : integer := 128
8      );
9      port (
10         clk      : in std_logic;
11         reset    : in std_logic;
12         address  : in integer range 0 to SIZE-1;
13         we       : in std_logic;
14         data_in  : in std_logic_vector(DATA_SIZE-1 downto 0);
15         data_out : out std_logic_vector(DATA_SIZE-1 downto 0)
16     );
17 end entity ram;
18
19 architecture Behavioral of ram is
20
21     type ram_t is array (0 to SIZE-1) of std_logic_vector(DATA_SIZE-1 downto 0);
22     signal ram : ram_t := (others => (others => '0'));
23
24     -- use Block RAM
25     attribute ram_style : string;
26     attribute ram_style of ram : signal is "block";
27
28 begin
29     update : process (clk)
30     begin
31         if rising_edge(clk) then
32             if reset = '1' then
33                 -- ram      <= (others => (others => '0'));
34                 data_out <= (others => '0');
35             else
36                 if we = '1' then
37                     ram(address) <= data_in;
38                 end if;
39                 data_out <= ram(address);
40             end if;
41         end if;
42     end process;
43 end architecture Behavioral;
```

## A.4 *mutator.vhd*

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  library work;
6  use work.ea.all;
7
8  entity mutator is
9      Generic (
10         DIM       : integer;
```

```
11          N_PARENTS : integer
12      );
13      Port (
14          clk        : in  std_logic;
15          reset      : in  std_logic;
16          in_reseed  : in  std_logic;
17          in_newseed : in  std_logic_vector(127 downto 0);
18          in_vec     : in  std_logic_vector(DIM-1 downto 0);
19          ready      : in  std_logic;                         -- '1' <=> out_vec read, new
            ↪  in_vec set
20          parent_ix  : out index_t;
21          valid      : out std_logic;
22          out_vec    : out std_logic_vector(DIM-1 downto 0)
23      );
24  end mutator;
25
26  architecture Behavioral of mutator is
27
28  --    type Ix_list is array (FLIP_BITS-1 downto 0) of integer range 0 to DIM-1;
29
30      -- Xoroshiro random generator
31      component rng_xoroshiro128plus is
32          generic (
33              init_seed:  std_logic_vector(127 downto 0)
34          );
35          port (
36              clk:        in  std_logic;
37              rst:        in  std_logic;
38              reseed:     in  std_logic;
39              newseed:    in  std_logic_vector(127 downto 0);
40              out_ready:  in  std_logic;
41              out_valid:  out std_logic;
42              out_data:   out std_logic_vector(63 downto 0)
43          );
44      end component;
45
46      -- signal rand_ready : std_logic := '0';
47      signal rand_valid : std_logic;
48      signal rand_data  : std_logic_vector(63 downto 0);
49
50      signal rand_int0 : integer;
51      signal rand_int1 : integer;
52
53      signal next_ix : integer range 0 to DIM+1 := 0;
54
55  begin
56
57      random_generator : rng_xoroshiro128plus
58      generic map (
59          init_seed => (others => '0') -- MUST be set initially!
60      )
61      port map (
62          clk       => clk,
63          rst       => '0',        -- do not reset so every mutation has fresh random values
64          reseed    => in_reseed,
65          newseed   => in_newseed,
66          out_ready => ready,
67          out_valid => rand_valid,
68          out_data  => rand_data
69      );
```

```
70
71      -- split 64 bit random number into two 32 bit integers
72      rand_int0 <= to_integer(signed(rand_data(31 downto 0)));
73      rand_int1 <= to_integer(signed(rand_data(63 downto 32)));
74
75      mutate : process (clk)
76      begin
77          if rising_edge(clk) then
78              if in_reseed = '1' then
79                  -- take initial parent index directly from the random seed
80                  -- (which itself is a random number) during initialization phase
81                  parent_ix <= to_integer(signed(in_newseed(127 downto 96))) mod N_PARENTS;
82              end if;
83
84              if reset = '1' then
85                  next_ix <= 0;
86                  valid <= '0';
87              elsif ready = '1' then
88
89                  if next_ix < DIM then
90                      if rand_int0 mod DIM = 0 then
91                          out_vec(next_ix) <= not in_vec(next_ix);
92                      else
93                          out_vec(next_ix) <= in_vec(next_ix);
94                      end if;
95                  end if;
96
97                  if next_ix+1 < DIM then
98                      if rand_int1 mod DIM = 0 then
99                          out_vec(next_ix+1) <= not in_vec(next_ix+1);
100                     else
101                         out_vec(next_ix+1) <= in_vec(next_ix+1);
102                     end if;
103
104                     next_ix <= next_ix+2;
105                 else
106                     parent_ix <= rand_int1 mod N_PARENTS;
107                     valid <= '1';
108                 end if;
109
110             end if;
111         end if;
112     end process;
113
114 end Behavioral;
```

## A.5 *evaluator.vhd*

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  library work;
5  use work.ea.all;
6
7  --/* Order of coefficients according to the following loss function:
8  -- *     L(x) = \sum_{i=0}^{DIM-1} \sum{j=0}^i \beta_{i,j} x_i
9  -- *
10 -- * So order of indices (i, j) is
```

```
11  -- *     (0, 0), (1, 0), (1, 1), (2, 0), (2, 1), ..., (DIM-1, DIM-1)
12  -- */
13
14  entity evaluator is
15      generic (
16          DIM        : integer
17      );
18      port (
19          clk      : in std_logic;
20          reset    : in std_logic;
21          isg_mode : in std_logic;
22
23          vect_in    : in std_logic_vector (DIM-1 downto 0);
24          coef_ready : in std_logic;
25          coef_in    : in loss_t;
26
27          coef_read : out std_logic;
28          valid     : out std_logic;
29          result    : out loss_t
30      );
31  end entity evaluator;
32
33  architecture Behavioral of evaluator is
34
35      -- indices
36      signal i : integer range 0 to DIM-1 := 0;
37      signal j : integer range 0 to DIM-1 := 0;
38
39      signal sig_coef_read : std_logic := '0';
40      signal sig_valid     : std_logic := '0';
41      signal sig_result    : loss_t    := 0;
42
43  begin
44
45      coef_read <= sig_coef_read;
46      valid     <= sig_valid;
47      result    <= sig_result;
48
49      process (clk)
50      begin
51          if rising_edge(clk) then
52
53              if reset = '1' then
54                  -- internal signals
55                  i     <= 0;
56                  j     <= 0;
57
58                  -- output signals
59                  sig_coef_read <= '0';
60                  sig_valid     <= '0';
61                  sig_result    <= 0;
62              else
63
64                  if sig_valid = '0' and coef_ready = '1' then
65
66                      if isg_mode = '1' then
67                          -- interpret 0 as -1
68                          if i = j then
69                              if vect_in(i) = '0' then
70                                  sig_result <= sig_result - coef_in;
```

```
71                              else
72                                  sig_result <= sig_result + coef_in;
73                              end if;
74                          else
75                              if (vect_in(i) xor vect_in(j)) = '1' then
76                                  sig_result <= sig_result - coef_in;
77                              else
78                                  sig_result <= sig_result + coef_in;
79                              end if;
80                          end if;
81                      else
82                          if (vect_in(i) and vect_in(j)) = '1' then
83                              sig_result <= sig_result + coef_in;
84                          end if;
85                      end if;
86
87                      if i = j then
88                          if i = DIM-1 then
89                              sig_valid <= '1';
90                          else
91                              i <= i + 1;
92                              j <= 0;
93                          end if;
94                      else
95                          j <= j + 1;
96                      end if;
97
98                      sig_coef_read <= '1';
99                  else
100                     sig_coef_read <= '0';
101                 end if;
102
103             end if;
104         end if;
105     end process;
106
107 end architecture Behavioral;
```

## A.6 *sorter.vhd*

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  library work;
5  use work.ea.all;
6
7
8  --/* IMPORTANT: When valid = '1' and reset = '1', hold reset until valid = '0' again,
9  -- * otherwise the component will not fully reset and the result will be invalid!
10 -- * Also while reset = '1' the output order will quickly become invalid despite valid
   ↪  still = '1'
11 --*/
12
13 entity sorter is
14     generic (
15         N : positive := 1
16     );
17     port (
```

```
18          clk   : in std_logic;
19          reset : in std_logic;
20          data  : in loss_array((2**N)-1 downto 0);
21          valid : out std_logic;
22          order : out index_array((2**N)-1 downto 0)
23      );
24  end entity;
25
26  architecture Behavioral of sorter is
27
28      constant MSB      : integer := (2**N)-1;
29      constant HALF_MSB : integer := (2**(N-1))-1;
30
31      component sorter is
32          generic (
33              N : positive := 1
34          );
35          port (
36              clk   : in std_logic;
37              reset : in std_logic;
38              data  : in loss_array((2**N)-1 downto 0);
39              valid : out std_logic;
40              order : out index_array((2**N)-1 downto 0)
41          );
42      end component;
43
44      component merger is
45          generic (
46              N : positive := 2
47          );
48          port (
49              clk   : in std_logic;
50              reset : in std_logic;
51              data  : in loss_array((2**N)-1 downto 0);
52              valid : out std_logic;
53              order : out index_array((2**N)-1 downto 0)
54          );
55      end component;
56
57      -- indexing functions
58      function apply_indices_to_loss_array(arr : loss_array; ixs : index_array)
59          return loss_array is
60          constant msb : integer := ixs'high;
61          variable res : loss_array(msb downto 0);
62      begin
63          for i in 0 to msb loop
64              res(i) := arr(ixs(i));
65          end loop;
66          return res;
67      end apply_indices_to_loss_array;
68
69      function apply_indices_to_indices(arr : index_array; ixs : index_array)
70          return index_array is
71          constant msb : integer := ixs'high;
72          variable res : index_array(msb downto 0);
73      begin
74          for i in 0 to msb loop
75              res(i) := arr(ixs(i));
76          end loop;
77          return res;
```

```
78      end apply_indices_to_indices;
79
80      signal lower_half_valid    : std_logic;
81      signal upper_half_valid    : std_logic;
82      signal lower_half_order    : index_array(HALF_MSB downto 0);
83      signal upper_half_order    : index_array(HALF_MSB downto 0);
84      signal sorted_halves_order : index_array(MSB downto 0);
85      signal sorted_halves_data  : loss_array(MSB downto 0);
86      signal merge_reset         : std_logic := '1';
87      signal merge_valid         : std_logic := '0';
88      signal merge_order         : index_array(MSB downto 0);
89
90  begin
91
92      MERGER0 : merger
93          generic map (N)
94          port map (clk, merge_reset, sorted_halves_data, merge_valid, merge_order);
95
96      order <= apply_indices_to_indices(sorted_halves_order, merge_order);
97
98      -- put together sorted halves:
99      -- lower half as is
100     sorted_halves_order(HALF_MSB downto 0) <= lower_half_order(HALF_MSB downto 0);
101     -- upper half with offset indices
102     SORTED_UPPER_HALF : for i in 0 to HALF_MSB generate
103         sorted_halves_order(i+HALF_MSB+1) <= upper_half_order(i) + 2**(N-1);
104     end generate SORTED_UPPER_HALF;
105
106     sorted_halves_data <= apply_indices_to_loss_array(data, sorted_halves_order);
107
108     SORT_TWO : if N = 1 generate
109
110         process (clk) begin
111             if rising_edge(clk) then
112                 if reset = '0' then
113                     -- no recursive sorting necessary
114                     lower_half_valid    <= '1';
115                     lower_half_order(0) <= 0;
116                     upper_half_valid    <= '1';
117                     upper_half_order(0) <= 0;
118
119                     merge_reset <= '0';
120                     valid       <= merge_valid;
121                 else
122                     merge_reset <= '1';
123                     valid       <= merge_valid;
124                 end if;
125             end if;
126         end process;
127
128     end generate SORT_TWO;
129
130     SORT_MANY : if N > 1 generate
131
132         -- sub-sorters for the lower and upper half
133         LOWER_HALF : sorter
134             generic map (N-1)
135             port map (clk, reset, data(HALF_MSB downto 0), lower_half_valid,
                ↪  lower_half_order);
136
```

```
137         UPPER_HALF : sorter
138             generic map (N-1)
139             port map (clk, reset, data(MSB downto HALF_MSB+1), upper_half_valid,
                ↪  upper_half_order);
140
141         process (clk) begin
142             if rising_edge(clk) then
143                 if reset = '0' then
144                     -- wait for sub-sorters to finish
145                     if (lower_half_valid = '1') and (upper_half_valid = '1') then
146                         merge_reset <= '0';
147                     end if;
148
149                     valid <= merge_valid;
150                 else
151                     merge_reset <= '1';
152                     valid       <= merge_valid;
153                 end if;
154             end if;
155         end process;
156
157     end generate SORT_MANY;
158
159 end architecture;
```

## A.7 *merger.vhd*

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  library work;
5  use work.ea.all;
6
7  entity merger is
8      generic (
9          N : positive := 2 -- should always be at least two
10     );
11     port (
12         clk   : in std_logic;
13         reset : in std_logic;
14         data  : in loss_array((2**N)-1 downto 0);
15         valid : out std_logic;
16         order : out index_array((2**N)-1 downto 0)
17     );
18 end entity;
19
20 architecture Behavioral of merger is
21
22     constant MSB      : integer := (2**N)-1;
23     constant HALF_MSB : integer := (2**(N-1))-1;
24
25     component merger is
26         generic (
27             N : positive := 2
28         );
29         port (
30             clk   : in std_logic;
31             reset : in std_logic;
```

```
32              data  : in loss_array((2**N)-1 downto 0);
33              valid : out std_logic;
34              order : out index_array((2**N)-1 downto 0)
35          );
36      end component;
37
38      -- indexing functions
39      function zip_indices(even : index_array; odd : index_array)
40          return index_array is
41          constant slice_msb : integer := even'high; -- must always be = odd'high, too!
42          variable res       : index_array(2*slice_msb+1 downto 0);
43      begin
44          assert even'high = odd'high report "Zipped indices of different lengths!";
45          for i in 0 to slice_msb loop
46              res(2*i)   := 2*even(i);
47              res(2*i+1) := 2*odd(i)+1;
48          end loop;
49          return res;
50      end zip_indices;
51
52      -- slice functions
53      function slice_even(arr : loss_array (MSB downto 0))
54          return loss_array is
55          -- constant slice_msb : integer := arr'high / 2; -- should round *down*!
56          variable slice : loss_array(HALF_MSB downto 0);
57      begin
58          for i in 0 to HALF_MSB loop
59              slice(i) := arr(2*i);
60          end loop;
61          return slice;
62      end slice_even;
63
64      function slice_odd(arr : loss_array (MSB downto 0))
65          return loss_array is
66          -- constant slice_msb : integer := arr'high / 2; -- should round *down*!
67          variable slice : loss_array(HALF_MSB downto 0);
68      begin
69          for i in 0 to HALF_MSB loop
70              slice(i) := arr(2*i+1);
71          end loop;
72          return slice;
73      end slice_odd;
74
75      signal even_merge_valid : std_logic := '0';
76      signal odd_merge_valid  : std_logic := '0';
77      signal even_merge_order : index_array(HALF_MSB downto 0);
78      signal odd_merge_order  : index_array(HALF_MSB downto 0);
79      signal zipped_order     : index_array(MSB downto 0);
80
81  begin
82
83      MERGE_TWO : if N = 1 generate
84
85          process (clk) begin
86              if rising_edge(clk) then
87                  if reset = '0' then
88                      -- compare [0:1]
89                      if data(0) > data(1) then
90                          order(0) <= 1;
91                          order(1) <= 0;
```

```
92                      else
93                          order(0) <= 0;
94                          order(1) <= 1;
95                      end if;
96                      valid <= '1';
97                  else
98                      valid <= '0';
99                  end if;
100             end if;
101         end process;
102
103     end generate MERGE_TWO;
104
105     MERGE_MANY : if N > 1 generate
106
107         -- Merger instances for even and odd slices
108         EVEN : merger
109             generic map (N-1)
110             port map (clk, reset, slice_even(data), even_merge_valid, even_merge_order);
111
112         ODD : merger
113             generic map (N-1)
114             port map (clk, reset, slice_odd(data), odd_merge_valid, odd_merge_order);
115
116         -- apply permutations indicated by the even/odd slices
117         zipped_order <= zip_indices(even_merge_order, odd_merge_order);
118
119         process (clk) begin
120             if rising_edge(clk) then
121                 -- wait for sub-merges to finish
122                 if reset = '0' and (even_merge_valid = '1') and (odd_merge_valid = '1')
                    ↪  then
123                     -- indices 0 and MSB are already correct
124                     order(0)   <= zipped_order(0);
125                     order(MSB) <= zipped_order(MSB);
126
127                     -- make remaining comparisions [i:i+1] for odd i < MSB
128                     for i in 0 to 2**(N-1)-2 loop
129                         if data(zipped_order(2*i+1)) > data(zipped_order(2*i+2)) then
130                             -- swap
131                             order(2*i+1) <= zipped_order(2*i+2);
132                             order(2*i+2) <= zipped_order(2*i+1);
133                         else
134                             -- keep
135                             order(2*i+1) <= zipped_order(2*i+1);
136                             order(2*i+2) <= zipped_order(2*i+2);
137                         end if;
138                     end loop;
139                     valid <= '1';
140                 else
141                     valid <= even_merge_valid or odd_merge_valid;
142                 end if;
143             end if;
144         end process;
145
146     end generate MERGE_MANY;
147
148 end architecture;
```

## A.8 *EA_IO_CTRL.vhd*

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.math_real."ceil";
4  use IEEE.numeric_std.all;
5
6  -- Uncomment the following library declaration if using
7  -- arithmetic functions with Signed or Unsigned values
8  --use IEEE.NUMERIC_STD.ALL;
9
10 -- Uncomment the following library declaration if instantiating
11 -- any Xilinx leaf cells in this code.
12 --library UNISIM;
13 --use UNISIM.VComponents.all;
14
15 entity EA_IO_CTRL is
16 Generic (
17     CLKS_PER_BIT : integer := 115;
18     DIM          : integer := 8;
19     N_PARENTS    : integer := 4;
20     N_CHILDREN   : integer := 12;
21     COEF_WIDTH   : integer := 32;
22     DATA_WIDTH   : integer := 128 -- = max(DIM, 128)
23 );
24 Port (
25     clk       : in std_logic;
26     serial_in : in std_logic;
27
28     opt_in      : in std_logic_vector(DIM-1 downto 0);
29     opt_loss_in : in std_logic_vector(31 downto 0);
30
31     serial_out       : out std_logic;
32     reset_out        : out std_logic;
33     config_out       : out std_logic_vector(1 downto 0);
34     data_ready_out   : out std_logic;
35     data_out         : out std_logic_vector(DATA_WIDTH-1 downto 0);
36     start_out        : out std_logic
37 );
38 end EA_IO_CTRL;
39
40 architecture Behavioral of EA_IO_CTRL is
41
42     type InteralState is (IDLE, COEF, PARENT, SEED, OPT, OPT_SENDING, INFO, INFO_SENDING,
       ↪  SENDING, CLEANUP);
43
44     constant DIM_BYTES      : integer := integer(ceil(real(DIM)/8.0));
45     constant DIM_STRAY_BITS : integer := 8 - (8*DIM_BYTES - DIM);
46
47     constant COEF_BYTES      : integer := integer(ceil(real(COEF_WIDTH)/8.0));
48     constant COEF_STRAY_BITS : integer := 8 - (8*COEF_BYTES - COEF_WIDTH);
49
50     constant BUFFER_BYTES : integer := integer(ceil(real(DATA_WIDTH)/8.0)); --
       ↪  max(DIM_BYTES, 16)
51
52     -- 7 Bytes containing information about current settings
53     -- (dimension, population sizes and coefficient width)
54     constant CONFIG_INFO : std_logic_vector(55 downto 0) :=
55           std_logic_vector(to_unsigned(COEF_WIDTH, 8))
```

```
56          & std_logic_vector(to_unsigned(N_CHILDREN, 16))
57          & std_logic_vector(to_unsigned(N_PARENTS, 16))
58          & std_logic_vector(to_unsigned(DIM, 16));
59
60      constant DEFAULT_SEED : std_logic_vector(127 downto 0) :=
        ↪  x"27404ad634a10951f37d4b657c96703f";
61
62      component UART_RX is
63          generic (
64              g_CLKS_PER_BIT : integer := 115
65          );
66          port (
67              i_Clk       : in  std_logic;
68              i_RX_Serial : in  std_logic;
69              o_RX_DV     : out std_logic;
70              o_RX_Byte   : out std_logic_vector(7 downto 0)
71          );
72      end component UART_RX;
73
74      component UART_TX is
75          generic (
76              g_CLKS_PER_BIT : integer := 115
77          );
78          port (
79              i_Clk       : in  std_logic;
80              i_TX_DV     : in  std_logic;
81              i_TX_Byte   : in  std_logic_vector(7 downto 0);
82              o_TX_Active : out std_logic;
83              o_TX_Serial : out std_logic;
84              o_TX_Done   : out std_logic
85          );
86      end component UART_TX;
87
88      signal state : InteralState := IDLE;
89
90      signal r_serial_out     : std_logic := '1';
91      signal r_reset_out      : std_logic := '0';
92      signal r_config_out     : std_logic_vector(1 downto 0) := "00";
93      signal r_data_ready_out : std_logic := '0';
94      signal r_data_out       : std_logic_vector(DATA_WIDTH-1 downto 0) := (others => '0');
95      signal r_start_out      : std_logic := '0';
96
97      signal rx_dv     : std_logic := '0';
98      signal rx_byte   : std_logic_vector(7 downto 0) := (others => '0');
99
100     signal tx_dv     : std_logic := '0';
101     signal tx_byte   : std_logic_vector(7 downto 0) := (others => '0');
102     signal tx_done   : std_logic := '0';
103
104     signal data_buffer  : std_logic_vector(8*BUFFER_BYTES-1 downto 0) := (others => '0');
105     signal byte_ix : integer range 0 to BUFFER_BYTES-1           := 0;
106
107     signal current_seed     : std_logic_vector(127 downto 0) := DEFAULT_SEED;
108     signal current_opt      : std_logic_vector(8*DIM_BYTES-1 downto 0)  := (others =>
        ↪  '0');
109     signal current_opt_loss : std_logic_vector(31 downto 0) := (others => '0');
110
111     signal first_tx_done : std_logic := '0';
112
113 begin
```

```
114
115     UART_RX_INST : UART_RX
116     generic map (CLKS_PER_BIT)
117     port map (
118         i_Clk       => clk,
119         i_RX_Serial => serial_in,
120         o_RX_DV     => rx_dv,
121         o_RX_Byte   => rx_byte
122     );
123
124     UART_TX_INST : UART_TX
125     generic map (CLKS_PER_BIT)
126     port map (
127         i_Clk => clk,
128         i_TX_DV => tx_dv,
129         i_TX_Byte => tx_byte,
130         o_TX_Active => open,
131         o_TX_Serial => r_serial_out,
132         o_TX_Done => tx_done
133     );
134
135     serial_out     <= r_serial_out;
136     reset_out      <= r_reset_out;
137     config_out     <= r_config_out;
138     data_ready_out <= r_data_ready_out;
139     data_out       <= r_data_out;
140     start_out      <= r_start_out;
141
142     process (clk)
143     begin
144         if rising_edge(clk) then
145             case state is
146             when IDLE =>
147                 if rx_dv = '1' then
148                     case rx_byte is
149                     when x"63" => -- 'c' input coefficients
150                         r_config_out <= "00";
151                         state     <= COEF;
152
153                     when x"70" => -- 'p' input parents
154                         r_config_out <= "01";
155                         state  <= PARENT;
156
157                     when x"78" => -- 'x' input seed
158                         r_config_out <= "10";
159                         state  <= SEED;
160
161                     when x"69" => -- 'i' activate ISG mode
162                         r_config_out <= "11";
163                         r_data_out(0)    <= '1';
164                         r_data_ready_out <= '1';
165                         state            <= CLEANUP;
166
167                     when x"73" => -- 's' start optimizer
168                         r_start_out <= '1';
169                         state       <= CLEANUP;
170
171                     when x"72" => -- 'r' reset optimizer
172                         r_reset_out  <= '1';
173                         current_seed <= DEFAULT_SEED;
```

```
174                         state        <= CLEANUP;
175
176                     when x"6f" => -- 'o' request current optimum and its loss value
177                         -- make "snapshot" so that values don't change during transmission
178                         current_opt(DIM-1 downto 0) <= opt_in;
179                         current_opt_loss            <= opt_loss_in;
180                         state                       <= OPT;
181
182                     when x"6b" => -- 'k' request configuration information + seed
183                         state <= INFO;
184
185                     when others =>
186                         state <= IDLE;
187
188                     end case;
189
190                 else
191                     state <= IDLE;
192                 end if;
193
194             when COEF =>
195                 if rx_dv = '1' then
196                     if byte_ix < COEF_BYTES-1 then
197                         -- write byte to buffer (little endian)
198                         data_buffer(8*byte_ix+7 downto 8*byte_ix) <= rx_byte;
199                         byte_ix <= byte_ix + 1;
200                         state <= COEF;
201                     else
202                         -- send data to EA
203                         r_data_out(COEF_WIDTH-1 downto 0) <= rx_byte(COEF_STRAY_BITS-1
                            ↪  downto 0)
204                                                              &
                                                             ↪  data_buffer(8*(COEF_BYTES-1)-1
                                                             ↪  downto 0);
205                         r_data_ready_out        <= '1';
206                         state <= CLEANUP;
207                     end if;
208                 else
209                     state <= COEF;
210                 end if;
211
212             when PARENT =>
213                 if rx_dv = '1' then
214                     if byte_ix < DIM_BYTES-1 then
215                         -- write byte to buffer (little endian)
216                         data_buffer(8*byte_ix+7 downto 8*byte_ix) <= rx_byte;
217                         byte_ix <= byte_ix + 1;
218                         state <= PARENT;
219                     else
220                         -- send data to EA
221                         r_data_out(DIM-1 downto 0) <= rx_byte(DIM_STRAY_BITS-1 downto 0)
222                                                       & data_buffer(8*(DIM_BYTES-1)-1 downto
                                                      ↪  0);
223                         r_data_ready_out <= '1';
224                         state <= CLEANUP;
225                     end if;
226                 else
227                     state <= PARENT;
228                 end if;
229
```

```
230             when SEED =>
231                 if rx_dv = '1' then
232                     if byte_ix < 15 then
233                         -- write byte to buffer (little endian)
234                         data_buffer(8*byte_ix+7 downto 8*byte_ix) <= rx_byte;
235                         byte_ix <= byte_ix + 1;
236                         state <= SEED;
237                     else
238                         current_seed <= rx_byte & data_buffer(119 downto 0);
239 
240                         -- send data to EA
241                         r_data_out(127 downto 0) <= rx_byte & data_buffer(119 downto 0);
242                         r_data_ready_out         <= '1';
243                         state <= CLEANUP;
244                     end if;
245                 else
246                     state <= SEED;
247                 end if;
248 
249             when OPT =>
250                 -- send loss value first, then optimum
251                 if first_tx_done = '0' then
252                     -- send loss value (4B) out via UART (little endian)
253                     tx_byte <= current_opt_loss(8*byte_ix+7 downto 8*byte_ix);
254                     tx_dv   <= '1';
255 
256                     if byte_ix < 3 then
257                         byte_ix <= byte_ix + 1;
258                     else
259                         byte_ix <= 0;
260                         first_tx_done <= '1';
261                     end if;
262                     state <= OPT_SENDING;
263 
264                 else
265                      -- send optimum (ceil(DIM/8) B) out via UART (little endian)
266                      tx_byte <= current_opt(8*byte_ix+7 downto 8*byte_ix);
267                      tx_dv   <= '1';
268 
269                      if byte_ix < DIM_BYTES-1 then
270                          byte_ix <= byte_ix + 1;
271                          state <= OPT_SENDING;
272                      else
273                          byte_ix  <= 0;
274                          state <= SENDING;
275                      end if;
276 
277                 end if;
278 
279             when OPT_SENDING =>
280                 tx_dv <= '0';
281 
282                 -- wait for transmission to finish
283                 if tx_done = '1' then
284                     state <= OPT;
285                 else
286                     state <= OPT_SENDING;
287                 end if;
288 
289             when INFO =>
```

```
290                 if first_tx_done = '0' then
291                     tx_byte <= CONFIG_INFO(8*byte_ix+7 downto 8*byte_ix);
292                     tx_dv   <= '1';
293 
294                     if byte_ix < 6 then
295                         byte_ix <= byte_ix + 1;
296                     else
297                         byte_ix <= 0;
298                         first_tx_done <= '1';
299                     end if;
300                     state <= INFO_SENDING;
301                 else
302                      tx_byte <= current_seed(8*byte_ix+7 downto 8*byte_ix);
303                      tx_dv   <= '1';
304 
305                      if byte_ix < 15 then
306                          byte_ix <= byte_ix + 1;
307                          state <= INFO_SENDING;
308                      else
309                          byte_ix  <= 0;
310                          state <= SENDING;
311                      end if;
312                 end if;
313 
314             when INFO_SENDING =>
315                 tx_dv <= '0';
316 
317                 -- wait for transmission to finish
318                 if tx_done = '1' then
319                     state <= INFO;
320                 else
321                     state <= INFO_SENDING;
322                 end if;
323 
324             when SENDING =>
325                 tx_dv <= '0';
326 
327                 -- wait for final transmission to finish
328                 if tx_done = '1' then
329                     -- clean up and return to idle
330                     state <= CLEANUP;
331                 else
332                     state <= SENDING;
333                 end if;
334 
335             when CLEANUP =>
336                 -- clear buffer
337                 data_buffer  <= (others => '0');
338                 byte_ix      <= 0;
339 
340                 tx_dv            <= '0';
341                 first_tx_done    <= '0';
342                 r_config_out     <= "00";
343                 r_reset_out      <= '0';
344                 r_data_ready_out <= '0';
345                 r_start_out      <= '0';
346                 state            <= IDLE;
347 
348             when others =>
349                 state <= IDLE;
```

```
350
351             end case;
352         end if;
353     end process;
354
355 end Behavioral;
```

## A.9 *eacom.py*

```
1  #!/usr/bin/env python3
2
3  import math
4  import os
5  import serial
6  import sys
7  import time
8
9  from optparse import OptionParser
10 from tqdm     import tqdm
11
12 def bytesToHexString(bs):
13     return "".join('{:02x}'.format(b) for b in bs)
14
15 def bytesToReverseBinString(bs, sep=""):
16     return (sep.join('{:08b}'.format(b) for b in bs))[::-1]
17
18 class EACom:
19     def __init__(self, device, baudrate, verbose=False):
20         self.device   = device
21         self.baudrate = baudrate
22         self.port     = None
23
24         self.__connected = False
25         self.__started   = False
26
27         self.__dim       = None
28         self.__parents   = None
29         self.__children  = None
30         self.__coefWidth = None
31         self.__seed      = None
32
33         self.verbose = verbose
34
35     def __report(self, *args, **kwargs):
36         if self.verbose:
37             print(*args, **kwargs)
38
39     def __check_connection(self):
40         if not self.__connected:
41             raise RuntimeError("Not connected")
42
43     def __check_configuring(self):
44         if self.__started:
45             raise RuntimeError("Optimization already started")
46
47     @property
48     def dimension(self):
49         if self.__dim is None:
```

```
50              self.info()
51          return self.__dim
52      @property
53      def parents(self):
54          if self.__parents is None:
55              self.info()
56          return self.__parents
57      @property
58      def children(self):
59          if self.__children is None:
60              self.info()
61          return self.__children
62      @property
63      def coefWidth(self):
64          if self.__coefWidth is None:
65              self.info()
66          return self.__coefWidth
67      @property
68      def seed(self):
69          if self.__seed is None:
70              self.info()
71          return self.__seed
72
73      @seed.setter
74      def seed(self, seed):
75          self.__check_connection()
76          self.__check_configuring()
77          self.__report("set seed to", bytesToHexString(seed))
78          self.port.write(b'x')
79          self.port.write(seed[::-1])
80          self.__seed = seed
81
82      def connect(self):
83          if not self.__connected:
84              self.__report("connect")
85              self.port = serial.Serial(self.device, self.baudrate)
86              self.__connected = True
87          else:
88              self.__report("already connected")
89
90      def disconnect(self):
91          self.__check_connection()
92
93          self.__report("disconnect")
94          self.port.close()
95          self.__connected = False
96
97      def reset(self):
98          self.__check_connection()
99
100         self.__report("reset")
101         self.port.write('r'.encode())
102         self.__started   = False
103         self.__seed      = None
104
105     def start(self):
106         self.__check_connection()
107         self.__check_configuring()
108
109         self.__report("start")
```

```
110         self.port.write('s'.encode())
111         self.__started = True
112 
113     def loadCoefficients(self, filename):
114         self.__check_connection()
115         self.__check_configuring()
116 
117         # read coefficients from file
118         with open(filename, "r") as f:
119             # remove whitespaces, newline etc.
120             coefs = list(map(int, "".join(f.read().split()).split(",")))
121 
122         nCoefs = len(coefs)
123         coefBytes = math.ceil(self.coefWidth/8.0)
124 
125         self.__report("send", nCoefs, "coefficients,", coefBytes, "Byte(s) each ... ")
126         for i in tqdm(range(nCoefs)):
127             # announce coefficient (1B)
128             self.port.write('c'.encode())
129             # send coefficient (1-4B)
130             self.port.write(coefs[i].to_bytes(coefBytes, byteorder='little', signed=True))
131 
132     # parent format: binary, ASCENDING indices (0 .. N-1)
133     def loadParents(self, filename=None):
134         self.__check_connection()
135         self.__check_configuring()
136 
137         parentBytes = math.ceil(self.dimension/8.0)
138 
139         if filename is None:
140             # randomly generate parents
141             self.__report("generate random parent population")
142             parents = [ os.urandom(parentBytes) for i in range(self.parents) ]
143             print(", ".join(map(bytesToHexString,parents)))
144         else:
145             with open(filename, "r") as f:
146                 parents = list(map(lambda x : int(x[::-1], base=2).to_bytes(parentBytes,
                    ↪  byteorder='little', signed=False),
                    ↪  "".join(f.read().split()).split(",")))
147 
148         nParents = len(parents)
149 
150         self.__report("send", nParents, "parent vectors,", parentBytes, "Byte(s) each ...
            ↪  ")
151         for i in tqdm(range(nParents)):
152             self.port.write('p'.encode())
153             self.port.write(parents[i])
154 
155     def enableIsingMode(self):
156         self.__check_connection()
157         self.__check_configuring()
158 
159         self.__report("enable ISG mode")
160         self.port.write(b'i');
161 
162     def optimum(self):
163         self.__check_connection()
164 
165         self.port.write('o'.encode())
166         self.__report("request optimum ...")
```

```
167         result = self.port.read(4 + math.ceil(self.dimension/8.0))
168         # self.__report("  raw result:", result)
169 
170         opt  = result[:3:-1]
171         loss = int.from_bytes(result[:4], byteorder='little', signed=True)
172         self.__report("  opt:", bytesToHexString(opt))
173         self.__report("  loss:", loss)
174 
175         return opt, loss
176 
177     def info(self):
178         self.__check_connection()
179 
180         self.__report("request configuration info ...")
181         self.port.write('k'.encode())
182         result = self.port.read(23) # 7B config info + 16B current seed
183         self.__report("  raw result:", result)
184 
185         self.__dim       = int.from_bytes(result[:2], byteorder='little')
186         self.__parents   = int.from_bytes(result[2:4], byteorder='little')
187         self.__children  = int.from_bytes(result[4:6], byteorder='little')
188         self.__coefWidth = int(result[6])
189         self.__seed      = result[:6:-1]
190 
191         self.__report("  dimension:  ", self.__dim)
192         self.__report("  parents:    ", self.__parents)
193         self.__report("  children:   ", self.__children)
194         self.__report("  coef. width:", self.__coefWidth, "Bits")
195         self.__report("  seed:       ", bytesToHexString(self.__seed))
196 
197         return self.__dim, self.__parents, self.__children, self.__coefWidth, self.__seed
198 
199 
200 SEEDS    = [b'\x2a\x62\x8a\xb3\xd4\x2b\x66\x25\xec\x39\xe1\x4f\x92\x9a\xaa\x55',
201            b'\x29\x25\x2e\x22\xd8\xad\xd5\x3e\x38\xe2\xa8\x7a\x03\x84\xdc\xa5',
202            b'\xa4\x9e\x91\xcb\xb2\xa6\x31\x33\x06\xe1\xb5\x98\xfb\x0c\x0b\x60',
203            b'\x6b\x61\xc2\x51\x00\x5c\xfe\x18\xe8\x25\x4c\x54\xba\x67\x38\x6a',
204            b'\x53\xfb\xfe\xed\x5a\x61\x0d\x1a\xa3\xda\xa1\xbd\x62\x49\x62\x58',
205            b'\x6f\xb4\xce\x0f\x98\x19\xe0\xa3\x50\xe2\xb7\x57\xd1\xb7\x29\xb1',
206            b'\x9a\x4c\x99\x1e\x63\x57\x51\x4f\x33\x19\x8b\x47\x89\x96\xe6\x55',
207            b'\x5d\x99\x47\x66\xad\xff\x63\x3c\x5f\xf0\x08\x10\x7c\xb5\xdf\x21',
208            b'\xaf\xcd\xf2\x15\x95\x1e\xe0\x99\x1b\x39\x0f\x81\x2e\x7f\xde\xf0',
209            b'\x01\x81\x0d\x0a\x10\xd3\x27\x39\x20\xa8\x93\xc2\x19\x5e\xaf\xa0']
210 
211 DEVICE   = "/dev/ttyUSB0"
212 BAUDRATE = 115200 # 868 clocks per bit
213 
214 def optimize(device, baudrate, coefFile, parentsFile=None, timelimit=100, interval=10,
    ↪  seed=None, isg=False, verbose=False):
215     eacomm = EACom(
216         device   = device,
217         baudrate = baudrate,
218         verbose  = verbose
219     )
220 
221     eacomm.connect()
222     eacomm.reset()
223     if seed is not None:
224         eacomm.seed = seed
225     if isg:
```

```
226             eacomm.enableIsingMode()
227         eacomm.loadCoefficients(coefFile)
228         eacomm.loadParents(parentsFile)
229         eacomm.start()
230 
231         t = 0
232         while timelimit < 0 or t < timelimit:
233             time.sleep(interval)
234             eacomm.optimum()
235             t += interval
236 
237         opt, loss = eacomm.optimum()
238         eacomm.reset()
239         eacomm.disconnect()
240 
241         print(bytesToReverseBinString(opt)[:eacomm.dimension], "(" + str(loss) + ")")
242 
243         return opt, loss
```