Maximilian Eibl, Martin Gaedke. (Hrsg.): INFORMATIK 2017,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 469

FPGA-basierter Protein- und DNA-Sequenzvergleich zur optimierten Datenbanksuche mit dem BLAST-Algorithmus

Thomas Fabian Starke[1], Timm Bostelmann[2], Sergei Sawitzki[3]

Abstract: Im Bereich der Bioinformatik und insbesondere der Genforschung ist die Suche nach lokalen Übereinstimmungen in Gensequenzen (engl. local alignment) von wichtiger Bedeutung. Die Anzahl und Größe der in den Gendatenbanken hinterlegten Sequenzen wächst jedoch rasant an, sodass die Suchgeschwindigkeit von kritischer Bedeutung ist. Für die Suche lokaler Übereinstimmungen zwischen einer Such- und einer Datenbanksequenz wird üblicherweise der BLAST Algorithmus (Basic Local Alignment Search Tool) eingesetzt. In dieser Arbeit wird eine Implementierung des BLAST Algorithmus in einer baumartigen Hardwarestruktur auf einem FPGA vorgestellt. Die Vor- und Nachteile dieser Implementierung gegenüber einer konventionellen Umsetzung des Algorithmus in Software werden gezeigt und analysiert. Abschließend wird ein Ausblick darauf gegeben, wie die bestehenden Einschränkungen der vorgestellten Lösung aufgehoben werden können.

Keywords: FPGA; BLAST; DNA; Sequenzvergleich

1 Einleitung

Im Bereich der Bioinformatik und der allgemeinen Genforschung ist die Suche von lokalen Übereinstimmungen von Sequenzen (engl. local alignment) eine wichtige Aufgabe. Bei den Sequenzen kann es sich um DNA, RNA oder Proteine bzw. Aminosäuren handeln. Bei der Suche wird eine *query*-Sequenz gegen eine Menge von Sequenzen innerhalb einer Gendatenbank geprüft und die lokalen Übereinstimmungen ermittelt. Dies kommt beispielsweise zum Einsatz, wenn die Funktionsweise eines unbekannten Virus erforscht und nach vergleichbaren, bekannten Strukturen innerhalb der Datenbanken gesucht wird. Die Anzahl und Größe der in den Datenbanken hinterlegten Sequenzen ist in den letzten Jahren stetig gestiegen (siehe Abb. 1). Aufgrund der sinkenden Kosten für die Sequenzierung, wird prognostiziert, dass sich dieser Trend fortsetzt.

Die für die Suche nach Übereinstimmungen von Sequenzen eingesetzten Algorithmen lassen sich in zwei Bereiche aufteilen: Globale Übereinstimmung (*engl. global alignment*) und lokale Übereinstimmung (*engl. local alignment*). Die Berechnung der globalen Übereinstimmung ist eine Form der globalen Optimierung. Sie erfordert eine gleiche Länge

[1] FH Wedel (University of Applied Sciences), Feldstraße 143, 22880 Wedel, Deutschland starke.thomas@yahoo.de
[2] FH Wedel (University of Applied Sciences), Feldstraße 143, 22880 Wedel, Deutschland bos@fh-wedel.de
[3] FH Wedel (University of Applied Sciences), Feldstraße 143, 22880 Wedel, Deutschland saw@fh-wedel.de

doi:10.18420/in2017_43

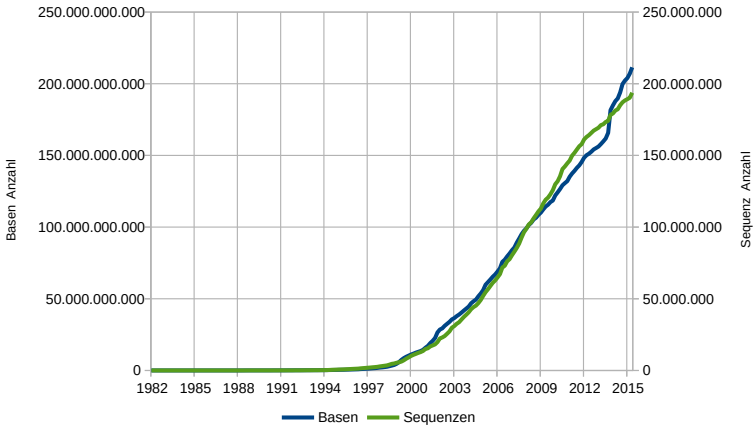


Abb. 1: Entwicklung der gespeicherten Gen-Sequenzen in der Genbank des NCBI [NC]

der Such- und Datenbanksequenzen. Lokale Übereinstimmungen identifizieren identische Bereiche in Sequenzen unterschiedlicher Länge, welche weitestgehend unterschiedlich sind. Lokale Übereinstimmungen werden in der Forschung bevorzugt, jedoch stellt das Finden dieser aufgrund der Laufzeitkomplexität des Problems eine besondere algorithmische Herausforderung dar. Ein etablierter Ansatz für die Suche lokaler Übereinstimmungen zwischen einer Such- und einer Datenbanksequenz ist der von Altschul u. a. vorgestellte BLAST Algorithmus [Al90].

Es wurde bereits gezeigt, dass die Suche nach lokalen Übereinstimmungen in Gensequenzen unter Einsatz mehrerer Hochleistungs-FPGAs erheblich beschleunigt werden kann [Go91]. Auch entsprechende industrielle Produkte sind bereits verfügbar [Ti]. In dieser Arbeit wird hingegen eine Implementierung auf einer kostengünstigen FPGA-Ausbildungsplatine vorgestellt und analysiert. Hierfür werden in Kapitel 2 die nötigen Grundlagen für den BLAST Algorithmus beschrieben. Darauf aufbauend wird in Kapitel 3 eine Implementierung des BLAST Algorithmus in einer baumartigen Hardwarestruktur auf einem FPGA vorgestellt. In Kapitel 4 werden die Vor- und Nachteile dieser Implementierung gegenüber einer konventionellen Umsetzung des Algorithmus in Software gezeigt und analysiert. Abschließend wird in Kapitel 5 ein Ausblick auf sinnvolle Erweiterungen gegeben.

2 Grundlagen

Der BLAST Algorithmus wird zur Suche lokaler Übereinstimmungen bzw. Ähnlichkeiten einer DNA- oder Proteinsequenz innerhalb einer Datenbank eingesetzt. Die Ausgabe des Programms ist eine Liste von Positionen bzw. Bereichen der Suchsequenz und der zugehörigen Datenbanksequenz, welche eine Ähnlichkeit aufweisen. Diese Paare werden

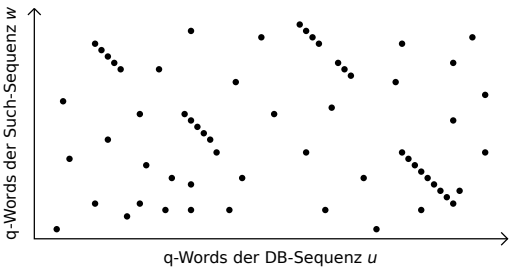


Abb. 2: In der ersten Stufe ermittelten Hits in der Hit-Matrix

High Score Pair (HSP) genannt. Zusätzlich gibt der Algorithmus die Signifikanz eines Treffers (*engl. hit*) an.

Der BLAST Algorithmus kann in fünf Stufen unterteilt werden, wobei die ersten vier Stufen für jeden Eintrag der Gendatenbank durchgeführt werden müssen.

Stufe 1: Erstellung der Hit Matrix Für die Suche einer Suchsequenz w innerhalb einer Datenbanksequenz u mit einer Bewertungsfunktion $\sigma(\alpha \rightarrow \beta)$ werden zunächst beide Sequenzen in überlappende Wörter der Länge $q \in \mathbb{N}$, wie im folgenden Beispiel mit $q = 3$, zerlegt:

$$\text{AVKTCSGAC} \Longrightarrow \{\text{AVK, VKT, KTC, TCS, CSG, SGA, GAC}\} \tag{1}$$

Die so entstandenen Wortmengen werden *w-Mere* und die einzelnen Wörter *q-word* genannt. Die Bewertungsfunktion σ dient der Bestimmung einer Übereinstimmung zwischen einem Zeichen der Suchsequenz und einem Zeichen der Datenbanksequenz. Je nach Art und Ausprägung der Sequenzen werden für die Bewertungsfunktion die *Block Substitution Matrix* (BLOSUM-Matrix) oder die *Point Accepted Mutation Matrix* (PAM-Matrix) verwendet. Hierbei gilt für sogenannte *deletions* bzw. *insertions*:

$$\sigma(\text{„-“} \rightarrow \beta) = \sigma(\alpha \rightarrow \text{„-“}) = -\infty \tag{2}$$

Eine Übereinstimmung, genannt Hit, für das Paar (i, j), wobei i und j die Indizes in der Datenbank- bzw. Suchsequenz sind, liegt dann vor, wenn für einen gegebenen Schwellwert k folgendes gilt:

$$i \in \{0 \ldots |u| - q + 1\} \tag{3}$$

$$j \in \{0 \ldots |w| - q + 1\} \tag{4}$$

$$score_\sigma(\underbrace{u[i \ldots i+q-1]}_{\text{q-word in u}}, \underbrace{w[j \ldots j+q-1]}_{\text{q-word in w}}) \geq k \tag{5}$$

Alle *q-words* der Suchsequenz werden mit allen *q-words* der Datenbanksequenz mit der Bewertungsfunktion σ auf Übereinstimmung untersucht. Die so gewonnen Ergebnisse

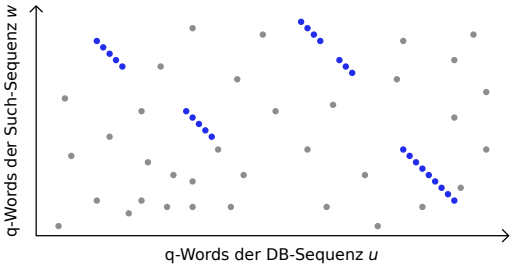


Abb. 3: Extraktion relevanter Hits aus der Hit-Matrix

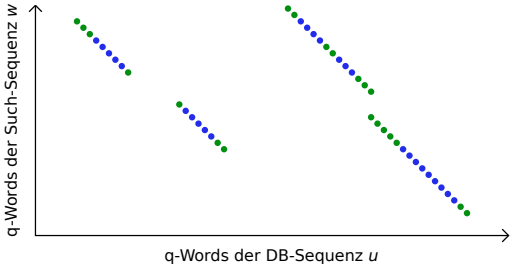


Abb. 4: Hit-Matrix nach der lückenlosen Erweiterung

werden in die sogenannte Hit-Matrix eingetragen (siehe Abb. 2). In alten Versionen des BLAST Algorithmus, welche bis ca. 1998 verwendet wurden, wurde für jeden detektierten Hit sofort die lückenlose bzw. die lückenbehaftete Erweiterung durchgeführt. Ab BLAST 2.0 wurde ein Zwischenschritt zur Optimierung eingeführt, welcher im Folgenden erläutert wird.

Stufe 2: Extraktion relevanter Hits Zur Optimierung der Stufen 3 und 4 werden zunächst alle relevanten Hits innerhalb der Hit-Matrix identifiziert. Hierfür werden alle sich auf der gleichen Diagonale befindlichen direkt aufeinander folgenden Hits zusammengefasst (siehe Abb. 3). Zur weiteren Einschränkung der relevanten Hits kann zusätzlich eine minimale Hit-Länge d definiert werden. Alle in Abb. 3 grau gekennzeichneten Hits werden somit aussortiert.

Stufe 3: Lückenlose Erweiterung Existieren zwei Hits auf einer Diagonale mit einem maximalen Abstand von δ, so wird ein Punkt (i, j) auf der Diagonale zwischen beiden Hits gewählt und anschließend in beiden Richtungen erweitert. Die Empfindlichkeit der Erweiterung wird über einen „*drop-off*“ Parameter $X_d \geq 0$ definiert. Für die Erweiterung eines Hits nach links werden die Sequenzen $u[1 \ldots i-1]$ und $w[1 \ldots j-1]$ von rechts

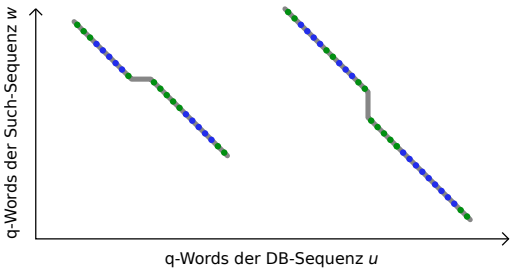


Abb. 5: In der Hit-Matrix gefundene lückenbehaftete Übereinstimmungen

nach links verglichen. Für die Erweiterung eins Hits nach rechts werden die Sequenzen $u[i \ldots |u|]$ und $w[j \ldots |w|]$ von links nach rechts verglichen. Jeder Vergleich bzw. jede Bewertung eines Zeichenpaars der Sequenzen liefert einen Wert, welcher durchgehend aufsummiert und dessen Maximalwert X_{max} gespeichert wird. Fällt der aufsummierte Bewertungswert unter $X_{max} - X_d$, so wird die Erweiterung beendet. Die so ermittelten ähnlichen Sequenzabschnitte werden *maximum segment pair* (MSP) genannt (siehe Abb. 4).

Stufe 4: Lückenbehaftete Erweiterung In der vierten Stufe wird eine lückenbehaftete Erweiterung der in Stufe zwei gefundenen Hits durchgeführt, um Regionen größerer Übereinstimmung mit akzeptablen Lücken zwischen beiden Sequenzen zu ermitteln. Dies kann sinnvoll sein, um sogenannte *insertions* bzw. *deletions* innerhalb der Sequenzen zu überspringen. Hierbei handelt es sich um Genmutationen bzw. virale Veränderung der Gene, wobei Teilstücke der Gensequenz verändert, entfernt oder aber neue Teilstücke hinzugefügt werden. Dies entspricht einem Sprung auf eine andere Diagonale innerhalb der Hit-Matrix (siehe Abb. 5). Nachdem ein Bereich mit vielen Hits innerhalb der Hit-Matrix identifiziert wurde, wird ein Punkt (i, j) zwischen zwei Hits als Startpunkt gewählt. Anschließend wird von diesem Punkt in Richtungen der beiden ausgewählten Hits expandiert. Hierbei werden die Zeichenpaare beider Sequenzen derart verglichen, dass ein Match $+1$ und ein Missmatch einer -1 entspricht. Die Erweiterung findet in einer Richtung solange statt, bis der aufsummierte Bewertungswert $X_{max} - X_d$ unterschreitet.

Stufe 5: Auswertung und Darstellung Für die Ausgabe werden die gefunden lokalen Übereinstimmungen zwischen einer Datenbank- und einer Suchsequenz mit einer Bewertungsfunktion nach deren Relevanz absteigend sortiert und ausgegeben. Zusätzlich können die gefundenen Übereinstimmungen auch grafisch dargestellt werden.

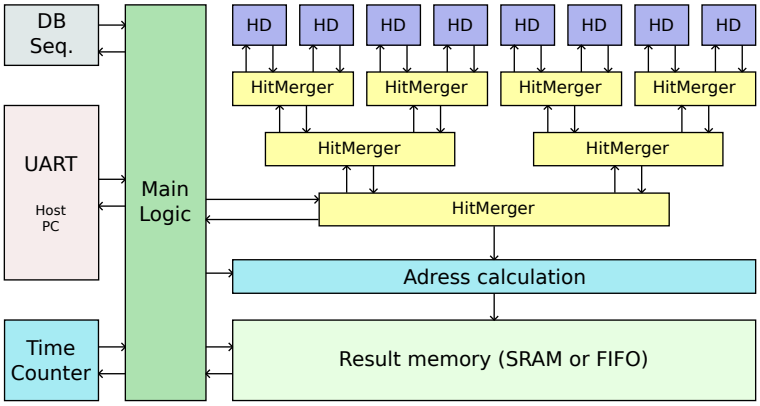


Abb. 6: Blockschaltbild der Umsetzung des BLAST Algorithmus in Hardware

3 Umsetzung

In diesem Kapitel wird eine Implementierung der ersten zwei Stufen des BLAST Algorithmus auf einem FPGA beschrieben. Diese Stufen wurden ausgewählt, weil sie zusammen den größten teil des Rechenaufwandes erzeugen [CWC04]. Die Implementierung basiert auf einer baumartigen Struktur zur parallelen Extraktion und Kombination von lokalen Übereinstimmungen zwischen einer Such- bzw. Datenbanksequenz (siehe Abb. 6).

Als Vereinfachung des Designs wird die zuvor beschriebene Bewertungsfunktion σ durch die Gleichheit ersetzt:

$$\sigma(\alpha \rightarrow \beta) = \begin{cases} \textit{True, Hit} & \text{für } \alpha = \beta \\ \textit{False, noHit} & \text{für } \alpha \neq \beta \end{cases} \tag{6}$$

Die maximale Größe eines Sequenzelements beträgt 5 bit, da ein Protein aus maximal 21 verschiedenen Aminosäuren bestehen kann. Zusätzlich wird als „uninitialisierter Zustand“ bzw. leeres Element der Vektor „`00000`“ vorgesehen.

Der *Time Counter* (in Abb. 6) wird für den Laufzeitvergleich mit einer Referenzimplementierung verwendet. Die Datenbanksequenz wird im *DB Sequence* Speicher vor der Verarbeitung gespeichert. Die Suchsequenz hingegen wird direkt in den Blättern der Baumstruktur gespeichert. Die weiteren Funktionsblöcke werden in den folgenden Abschnitten beschrieben.

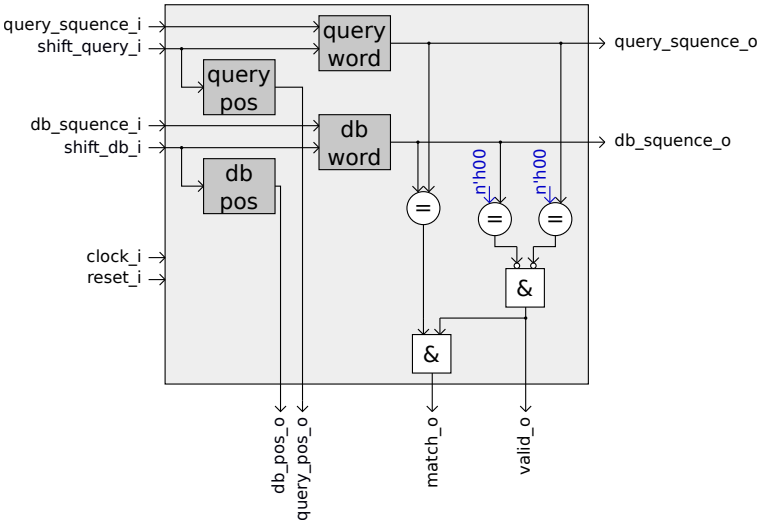


Abb. 7: Blockschaltbild des Hit-Matchers

3.1 Hit-Matcher

Der Hit-Matcher (siehe Abb. 7) ist die unterste Komponente innerhalb der Baumstruktur. Dieses Modul besitzt je ein Speicher für ein Wort der Such- und der Datenbanksequenz, welche unabhängig voneinander über den jeweiligen `shift`-Eingang beschrieben werden können. Das jeweilige aktuell gespeicherte Element wird am entsprechenden Ausgang ausgegeben. Sofern notwendig kann zusätzlich die Position bzw. die Anzahl der Schiebevorgänge mitgespeichert werden. Erst wenn in beiden Elementspeichern ein Element gespeichert ist (Bit-Vektor ungleich 0), kann ein korrekter Elementvergleich durchgeführt werden. Dies wird mit einer '1' am `valid_o`-Ausgang gekennzeichnet. Der `match_o`-Ausgang zeigt an, ob die beiden gespeicherten Elemente gleich sind, wobei der Ausgang nur '1' werden kann, wenn das `valid`-Flag gesetzt ist. Hierdurch wird ein positiver Match noch nicht gefüllter Vergleicher unterbunden. In der aktuellen Implementierung der Baumstruktur werden die Positionsinformationen der Sequenzen an dieser Stelle nicht benötigt und entfallen daher bei der Synthese durch Optimierung.

3.2 Hit-Detektor

Der Hit-Detektor (siehe Abb. 8) stellt ein Blatt der Baumarchitektur dar und besteht aus n Hit-Matchern, welche in einer Kette so angeordnet werden, dass die beiden Sequenzen durch die Hit-Matcher geschoben werden können. Die zuvor beschriebenen *q-words* werden durch UND-Gatter der Breite q realisiert, wobei die Vergleichsergebnisse der ersten

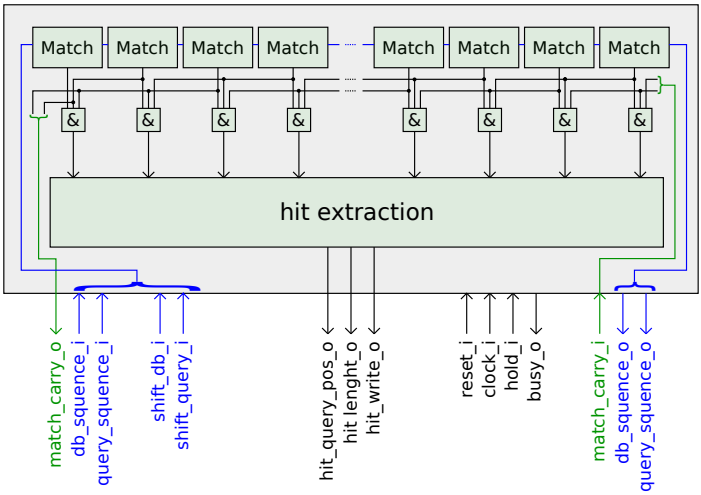


Abb. 8: Blockschaltbild des Hit-Detektors

$q-1$ Hit-Matcher einem eventuell vorgeschalteten Hit-Detektor zur Verfügung gestellt werden. Mit dem Schieben der Datenbanksequenz wird der Hit-Extraktionsprozess gestartet, welcher sequentiell die Ergebnisse der n Hit-Matcher auswertet und zusammenfasst. Ist beispielsweise das i.te *q-word* identisch, so wird die Position des entsprechenden UND-Gatters in einem internen Register zwischengespeichert. Sind die nachfolgenden *q-words* ebenfalls identisch so wird die Länge des Hits inkrementiert, bis ein *q-word* nicht identisch ist oder aber das Ende erreicht wurde. Ist dies der Fall, wird der ermittelte Hit an die nächste Ebene weitergegeben, in dem die Position und Länge des Hits am Ausgang ausgegeben wird und das Ausgangssignal `hit_write_o` für einen Taktzyklus auf '1' gesetzt wird.

3.3 Hit-Merger

Der Hit-Merger (siehe Abb. 9) stellt einen Knoten innerhalb der Baumarchitektur dar. Dessen Kinder sind abhängig von der Baumtiefe entweder zwei Hit-Detektoren oder aber zwei Hit-Merger. Die Anzahl der noch zu erstellenden Kinderknoten wird auf jeder Ebene dekrementiert und über ein VHDL-Generic an die Kinder des Knotens übergeben, wodurch während der Synthese rekursiv die in Abb. 6 dargestellte Baumstruktur erzeugt wird. Ist der Wert des Generics > 1 so werden als Kinder zwei Hit-Merger und bei einem Wert gleich eins zwei Hit-Detektoren (unterste Ebene) erzeugt.

Nachdem die Datenbanksequenz um ein Element verschoben wurde und die `busy_o`-Signale der beiden Kinder auf '1' wechseln, werden eingehende Hits in dem jeweiligen FIFO zwischengespeichert. Die in den FIFOs gespeicherten Hits werden nacheinander ausgelesen

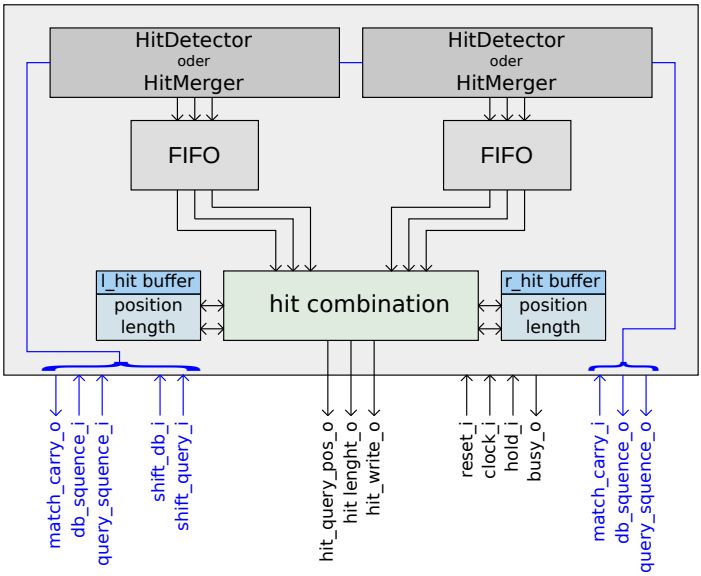


Abb. 9: Blockschaltbild des Hit-Mergers

und je nach FIFO geprüft. Endet der Hit im linken FIFO am rechten Rand des Kindes, so wird der Hit in den linken Hit-Zwischenspeicher geschrieben. Beginnt der Hit im rechten FIFO am linken Rand des Kindes, so wird der Hit in den rechten Hit-Zwischenspeicher geschrieben. Ist die Überprüfung negativ, so wird der Positionsvektor des Hits um ein Bit (MSB) erweitert. Für den linken FIFO wird eine ’0’ und für den rechten FIFO eine ’1’ angefügt. Anschließend wird der Hit an die nächste Ebene weitergegeben.

Nachdem die Bearbeitung innerhalb der beiden Kinder beendet ist (busy_o-Signale beider Kinder auf ’0’) und alle Hits aus den FIFOs verarbeitet wurden, wird geprüft, ob beide Hit-Zwischenspeicher einen Hit enthalten. Ist dies der Fall, so werden beide Hits kombiniert, indem als Startposition des resultierenden Hits die Position des linken Hits und als Länge die Summe beider Hit-Längen verwendet wird. Enthält nur einer der beiden Hit-Zwischenspeicher einen Hit, so wird dessen Positionsvektor (wie oben beschrieben) erweitert und an die nächste Eben weitergegeben. Der busy_o-Ausgang des Hit-Mergers ist ’1’, solange eines der beiden Kinder aktiv ist oder sich Hits innerhalb der FIFOs bzw. der Hit-Zwischenspeicher befinden.

Die maximal Anzahl f der Speicherplätze in einem FIFO lässt sich mit folgender Formel bestimmen, wobei w der Anzahl der Hit-Matcher innerhalb eines Hit-Detektors, q der Breite der *q-words* und th der aktuellen Tiefe innerhalb der Baumstruktur entspricht:

$$f = \frac{w}{q+1} \cdot 2^{(th-1)} \tag{7}$$

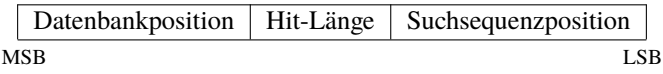
Die Breite der einzelnen FIFO-Worte hängt hierbei von der Breite des Positionsvektors und des Längenvektors ab, wobei die Breite des Positionsvektors wiederum von der Tiefe innerhalb der Baumstruktur abhängt:

$$width = \mathrm{ld}(w) + \mathrm{ld}(th - 1) + \mathrm{ld}(maxquerylength) \tag{8}$$

3.4 Adressberechnung und Ergebnisspeicher

Das Adressberechnungsmodul ist ein Schaltnetz, welches die aus dem aktuellen Shift-Wert der Datenbanksequenz und einer Hit-Position innerhalb der Suchsequenz die zugehörige Datenbankposition des Hits berechnet. Zusätzlich wird die Länge eines Hits korrigiert, da diese bisher nur der Anzahl der aufeinander folgenden *q-words* entspricht, aber nicht der Anzahl der identischen Elemente. Daher wird $q - 1$ auf die Länge eines Hits aufaddiert.

Die zu einem Hit gehörenden Informationen werden in einem Ergebnisspeicher zusammengetragen. Dieser wird durch einen FIFO realisiert, wobei die Größe des FIFO der in Abschnitt 3.3 vorgestellten Formel entspricht. Ein Word besitzt dabei folgenden Aufbau:



3.5 Ablaufsteuerung

Die Ablaufsteuerung befindet sich in der *Main Logic* und steuert den gesamten Ablauf des Vergleichsprozesses. Vor einem Vergleichsvorgang muss der Host-PC zunächst eine Datenbank- und eine Suchsequenz übermitteln. Anschließend kann der Vergleichsvorgang mit dem Startkommando gestartet werden. Ein Vergleichsschritt wird durch einen Shift-Vorgang der Datenbanksequenz initiiert. Anschließend werden alle Hits durch die Hit-Merger-Baumstruktur ermittelt und im Ergebnisspeicher gespeichert. Nachdem der Vergleichsschritt beendet ist, werden die gefunden Hits an den Host-PC übermittelt. Es werden nacheinander insgesamt $|u| + |w|$ Vergleichsschritte durchgeführt, mit der Datenbanksequenz u und der Suchsequenz w. Beendet wird der Vergleichsprozess durch das Übermitteln der durch den *Time Counter* ermittelten Laufzeit.

4 Ergebnisse

In diesem Kapitel wird die entstandene Architektur mit einer Referenzimplementierung auf einem herkömmlichen PC (Intel Core i5 @ 2.80 GHz, 8 GB RAM, Windows 7 - 64-bit) verglichen. Hierbei werden verschiedene Aspekte und Konfigurationsmöglichkeiten des BLAST Algorithmus näher untersucht. Die FPGA-Implementierung wird auf einem *Altera*

Suchlänge	Hits	PC / ms	FPGA / ms	FPGA / Taktzyklen
8	149	12.855	1.201	60093
32	786	55.155	1.219	60974
64	1598	94.331	1.246	62304
128	3050	195.607	1.296	64791
256	6263	316.306	1.406	70310
512	12081	640.919	1.611	80594
1020	23871	1241.095	2.006	100326

Tab. 1: Laufzeitmessung unter Veränderung der Suchlänge

q-word Größe	Hits	PC / ms	FPGA / ms	FPGA / Taktzyklen
1	10844	47.630	1.288	64419
2	2866	47.009	1.230	61517
3	786	40.605	1.219	60974
4	199	52.516	1.217	60899
5	51	36.553	1.217	60888

Tab. 2: Laufzeitmessung unter Veränderung der *q-word* Größe

DE2 Development and Education Board mit einem Takt von 50 MHz ausgeführt. Die Vergleichssoftware dient gleichzeitig der Kommunikation mit der auf dem DE2 Board befindlichen Teil-Implementierung des BLAST Algorithmus.

Zunächst wird auf einer Nukleotid-Datenbanksequenz mit 1.813 Basen mit unterschiedlich langen Suchsequenzen gesucht. Dieser Test zeigt, wie die Anzahl der gefunden Hits mit der Länge der Suchsequenz deutlich ansteigt. Wie in Tab. 1 gezeigt, steigt die Laufzeit der FPGA-Architektur im Gegensatz zur PC-Implementierung nur sehr langsam an. Der Laufzeitanstieg der FPGA Architektur wird durch den Anstieg der zu kombinierenden Hits innerhalb der Baumstruktur hervorgerufen.

Ein weiterer Parameter ist die Größe der *q-words*. Getestet wird die Änderung der *q-word* Größe mit einer Nukleotid-Datenbanksequenz mit 1.813 Basen und einer Nukleotid-Suchsequenz mit 32 Basen. Wie in Tab. 2 gezeigt, hat die Änderung der *q-word* Größe kaum Einfluss auf die Laufzeit, da dieser Schritt innerhalb der FPGA Architektur durch ein Schaltnetz realisiert wurde.

Die Detektorbreite hat einen großen Einfluss auf die Laufzeit der FPGA Implementierung, da die Ergebnisse der Hit-Matcher sequentiell ausgewertet werden. Den Einfluss auf die Laufzeit wird anhand einer Nukleotid-Datenbanksequenz mit 1.813 Basen und einer Nukleotid-Suchsequenz mit 127 Basen analysiert. Wie in Tab. 3 gezeigt, hat die Detektorbreite einen massiven Einfluss auf die Laufzeit der FPGA Implementierung. Je kleiner die Detektorgröße gewählt wird, desto mehr Hit-Merger-Ebenen besitzt die resultierende Baumstruktur. Dies erfordert jedoch auch mehr FIFO-Speicher innerhalb des FPGAs.

Hit-Detektor Breite	Hits	PC / ms	FPGA / ms	FPGA / Taktzyklen
4	3029	152.202	0.233	11897
8	3029	152.142	0.392	19630
16	3029	152.136	0.681	34086
32	3029	152.132	1.286	64295
64	3029	152.148	2.523	126160

Tab. 3: Laufzeitmessung unter Veränderung der Hit-Detektor Breite

5 Fazit und Ausblick

Es wurde gezeigt, dass die ersten beiden Stufen des BLAST Algorithmus gut durch eine parallel arbeitende Baumstruktur innerhalb eines FPGAs realisiert werden können. Allerdings ist diese Art der Implementierung für größere Protein- bzw. Nukleotidsequenzen sehr hardwarelastig. Eine Reduktion der benötigten Gatter und Zwischenspeicher ist durch eine Verbreiterung der Hit-Detektoren möglich, jedoch mit einer Verlängerung der Laufzeit verbunden. Durch den Einsatz eines größeren FPGAs oder durch die parallele Verwendung mehrerer FPGAs lassen sich so auch längere Sequenzen schnell miteinander vergleichen.

In zukünftigen Arbeiten könnten die Stufen 3 und 4 des BLAST Algorithmus implementiert werden, um den vollständigen Algorithmus in einem FPGA zu realisieren. Außerdem wird in der aktuellen Implementierung nach einem Schiebevorgang der Datenbanksequenz gewartet, bis alle zugehörigen Hits ermittelt und im Ergebnisspeicher eingetragen wurden. Jedoch kann die Baumstruktur auch als mehrstufige Pipeline aufgefasst werden. Hierdurch müsste für das erneute Schieben der Datenbanksequenz lediglich die nachfolgende Hit-Merger-Ebene leer sein. Dies erfordert jedoch einen Mehraufwand in der Ablaufsteuerung. Schlussendlich wäre zu untersuchen wie eine Bewertungsfunktion auf Basis der BLOSUM- oder PAM-Matrix effizient realisiert werden kann.

Literaturverzeichnis

[Al90] Altschul, Stephen F.; Gish, Warren; Miller, Webb; Myers, Eugene W.; Lipman, David J.: Basic Local Alignment Search Tool. Journal of Molecular Biology, 215(3):403–410, 1990.

[CWC04] Cameron, Michael; Williams, Hugh E.; Cannane, Adam: Improved gapped alignment in BLAST. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 1(3):116–129, 2004.

[Go91] Gokhale, Maya et al.: Building and Using a Highly Parallel Programmable Logic Array. IEEE Computer, 24(1):81–89, 1991.

[NC] NCBI: , GenBank and WGS Statistics. `http://www.job-coaching.de/#/overview` (Stand 02.07.2016).

[Ti] TimeLogic: , Accelerated BLAST Performance with Tera-BLAST™: a comparison Technical Note, 2013.