hochschule mannheim

Generierung und Implementierung von zellulären Automaten für FPGAs unter Verwendung von Location Constraints

Ngọc Minh Nguyễn

Master-Thesis

zur Erlangung des akademischen Grades Master of Science (M.Sc.)

Studiengang Informationstechnik

Fakultät für Informationstechnik

Hochschule Mannheim

30.04.2021

Betreuer

Prof. Dr. Rüdiger Willenberg

Prof. Dr.-Ing. Kurt Ackermann

Nguyễn, Ngọc Minh:
Generierung und Implementierung von zellulären Automaten für FPGAs unter Verwendung von Location Constraints / Ngọc Minh Nguyễn. –
Master-Thesis, Mannheim: Hochschule Mannheim, 2021. 37 Seiten.

Nguyễn, Ngọc Minh:
Generation and implementation of cellular automata for FPGAs using location constraints / Ngọc Minh Nguyễn. –
Master Thesis, Mannheim: University of Applied Sciences Mannheim, 2021. 37 pages.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Mannheim öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Mannheim, 30.04.2021

Ngọc Minh Nguyễn

Abstract

Generierung und Implementierung von zellulären Automaten für FPGAs unter Verwendung von Location Constraints

Das Ziel dieser Thesis ist ein klassisches zelluläre Automaten (zelluläre Automaten (ZA)) Systems, das Game of Life (Game of Life (GOL)), unter der Verwendung eines FPGAs zu implementieren und zu generieren.
Aufgrund der Ähnlichkeit der inneren Architektur von Field Programmable Gate Arrays (FPGA)s mit dem ZA, lässt sich eine exakte Abbildung der Zellen auf die Hardware-Zellen übertragen.

Generation and implementation of cellular automata for FPGAs using location constraints

Danksagung

Zunächst möchte ich mich bei meinem Betreuer Herrn Prof. Dr.-Ing. Rüdiger Willenberg und Herrn Prof. Dr.-Ing. Kurt Ackermann, die mir stets mit wertvollen Tipps und Ratschlägen zur Seite standen und mir diese herausfordernde Thesis zugetraut haben.
Ein großer Dank geht ebenfalls an meine Freundin Sarah Nordt, die mir immer mit Rat und Tat zur Seite stand, wenn ich wieder einmal in den Untiefen der deutschen Sprache verloren war.
Und zu guter Letzt möchte ich meinem Bruder Minh Khuê Nguyễn und meiner Mutter Thị Nguyệt Bùi danken, die mir das Studium ermöglichen, mich immer wieder unterstützen und motivieren, sowohl finanziell als auch moralisch.

Inhaltsverzeichnis

Danksagung iii

1. Einleitung 1

2. Einführung und Grundbegriffe 3
2.1. Zelluläre Automaten . . . 3
2.1.1. Gitterstruktur . . . 3
2.1.2. Zustände . . . 4
2.1.3. Nachbarschaft . . . 5
2.1.4. Übergangsregel . . . 5
2.1.5. Randbedingungen . . . 6
2.1.6. Startkonfiguration . . . 7
2.2. Game of Life . . . 7

3. FPGA 9
3.1. Field Programmable Gate Array . . . 9
3.1.1. ZedBoard Architektur . . . 9
3.1.2. Konfigurierbaren Logikblöcke . . . 10
3.1.3. Look-Up Tabellen (LUT) . . . 12
3.1.4. Carry-Logik . . . 12
3.2. Vivado Design Suite . . . 14
3.2.1. Designablauf . . . 14
3.2.2. Entwurfssichten . . . 15
3.2.3. Entwurfseinschränkungen . . . 16

4. Implementierung 18
4.1. Zell Architektur . . . 18
4.1.1. Vorüberlegungen . . . 18
4.1.2. Entwurf des Zell-Moduls durch Verhaltensbeschreibung . . . 19
4.1.3. Logik Minimierung durch Strukturbeschreibung . . . 22
4.1.4. Look Up Table (LUT)6 Konfiguration . . . 23
4.1.5. Ergebnisse der Strukturbeschreibung . . . 25
4.2. Implementierung des Spielfeldes . . . 27
4.2.1. Ergebnis . . . 28

4.3. IP Core zur Datentransfer 29
4.3.1. CA Core Modul 29
4.3.2. GOL IP-Core 30
4.4. Game of life Test-Applikation 32

5. Auswertung und Ergebnisse **35**
5.1. Modulplatzierung 35
5.2. Auswertung des Timingsverhaltens 36
5.3. Ressourcenauslastung 36

6. Auswertung und Ergebnisse **37**
6.1. Zedboard Auslastung 37
6.1.1. Timing 37
6.1.2. Resourcenauslastung 37

Abkürzungsverzeichnis **vi**

Tabellenverzeichnis **vii**

Abbildungsverzeichnis **viii**

Quellcodeverzeichnis **x**

Literatur **xi**

Index **xii**

A. Erster Anhang **xii**

B. Zweiter Anhang **xv**

Kapitel 1

Einleitung

Das Konzept ZA wurde von John von Neumann in den späten vierziger Jahren eingeführt und dienen zur Modellierung und Simulation von dynamischen und komplexer Systeme. Ein zellulärer Automat besteht aus gleichmäßiger Anordnung identischer Zellen, wobei jede Zelle kann durch Wechselwirkung mit seinen Nachbarzellen von einem Zustand zum nächsten Zustand annehmen. Durch das Zusammenwirken mehrere Zellen können somit komplexe Systeme geschaffen werden.
Mit ZA lassen sich unzähligen Anwendungsgebiete simulieren unter anderem die Entstehung des Leben, Modelle im Verkehrswesen oder die Ausbreitung von Epidemien.
In der heutigen Zeit ist der Bedarf an Simulationsumgebungen mit hoher Komplexität gestiegen. Algorithmen die zur Verarbeitung von Daten und die Durchführung von tausenden von Berechnungen pro Sekunde, benötigen viel Rechenleistung, die typischerweise zu hohen Kosten und lange Simulationen verursachen.
Durch die Verwendung von Mehrkern-Prozessoren mit nebenläufiger Programmierung, um Aufgaben auf verschiedene Threads zu verteilen, ermöglicht eine beschleunigte Ausführung. Die Schwierigkeit in der nebenläufige Programmierung besteht darin, die Verteilung der Aufgaben an die einzelnen Threads. Hinzukommt, durch den Austausch von Daten zwischen mehreren Threads, kann die Performance darunter leiden.
Eine mögliche Hardwarelösung für die algorithmische Verarbeitung bietet FPGA. Ein FPGA besteht aus programmierbare Logikblöcke, die in einer Gitterstruktur angeordnet sind. Man programmiert sie mithilfe einer speziellen Hardwarebeschreibungsprache, beispielsweise VHSIC Hardware Description Language (VHDL), logische Schaltungen, die anschließend in die Logikblöcke implementiert werden. Lo-

gische Schaltungen arbeiten voll parallel und verfügen über begrenzte Ressourcen verglichen mit einem gängigen Computer.
Durch ihre massiv parallelen Arbeitsweise und deren innere Architektur ähneln FPGAs einem ZA. Mit den vorhandenen Gitterstruktur lässt sich flexible logische Schaltungen realisieren, welche über Software konfigurieren können.
In der folgende Thesis wird ein FPGA der Firma Xilinx als Rechenbeschleuniger verwendet, um ein klassisches ZA Systems, das GOL, zu berechnen. Für das konfigurieren der Rechenbeschleuniger kommt ein Mikrocontroller zum Einsatz.

Kapitel 2

Einführung und Grundbegriffe

In diesem Kapitel wird zunächst eine allgemeine Beschreibung von zelluläre Automaten erläutert. Es werden der Grundaufbau und Ablauf angesprochen, um ein grundlegendes Verständnis über solche Systeme zu schaffen.

2.1. Zelluläre Automaten

Wie in der Einleitung beschrieben bestehen zelluläre Automaten aus gleichmäßiger Anordnung identischer Zellen. Zellen lassen sich als Zustände formulieren und die Zustandsänderung jeder Zelle wird in diskrete zeitliche Schritte betrachtet.

$$t_0, t_1, t_2, ..., t_k$$

wobei t_0 den Startzeitpunkt festlegt. Die Entwicklung einer Zelle hängt von ihrem aktuellen Zustand und den Zustand ihrer benachbarten Zellen ab. Grundsätzlich wird die zeitliche Entwicklung des ZA durch die Anfangskonfiguration der Zellen festgelegt.

2.1.1. Gitterstruktur

Die einzelnen Zellen sind in ein diskreter Raum, in dem die Entwicklung stattfindet, miteinander verbunden. Unterschiede gibt es in der Dimension (ein, zwei oder dreidimensional) und der Geometrie (reckteckig, hexagonal).
Die lineare Anordnung ist der einfachste Fall, hier sind die Zellen in einer eindi-

mensionaler Raum verbunden.

x_0	x_1	x_2	x_3	x_4	x_5	...	x_n

x_i steht für Zelle an der Position i. Wobei $i = 0, 1, ..., n$.
Ein Anwendungsbeispiel für ein eindimensionaler zellulärer Automat ist das Nagel-Schreckenberg-Modell. Hierbei handelt es sich um die Simulation einspurige Autobahn.

Bei zweidimensionaler Anordnung besteht der Raum aus einem quadratischen Gitter mit m x n Zellen.

Abbildung 2.1.: Beispiel eines zweidimensionalen Gitter: a) rechteckig und b) hexagonales

xij für $i = 0, 1, ..., n$ und $j = 0, 1, ..., m$.

In dieser Thesis wird auf zweidimensionaler rechteckiger Gitterstruktur, welche als Spielfeld bezeichnet wird beschränkt, da sie sich einfacher in die FPGA matrixförmiger Anordnung implementieren lässt.

2.1.2. Zustände

Jede Zelle wird durch ihr Zustand zu jedem Zeitpunkt definiert. Ihre Zustandsmenge Q ist abzählbar und können sowohl symbolische Bedeutung als auch Zahlen besitzen. Wichtig hierbei ist, dass die Zustände unterscheidbar sein und sich aufzählen lassen müssen.

$$Q = \{q_1, q_2, ..., q_s\}$$

Im einfachsten Fall sind es binäre ZA beispielsweise Ja/Nein oder tot/lebendig, s.Abb.2.3, die eine erstaunliche Komplexität aufweisen.

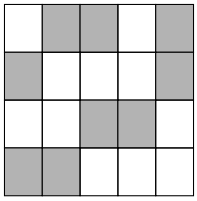


Abbildung 2.2.: Beispiel eines zweidimensionalen 4x5 Gitter mit zwei Zuständen. Wobei Q aus $s = 2$ Zuständen besteht. Weiß q_1 für “tot” und Grau q_2 für "lebendig".

2.1.3. Nachbarschaft

Als Nachbarschaft ist der Interaktionsradius einer Zelle definiert. Die benachbarten Zellen in dieser Umgebung stehen in Wechselwirkung mit der Referenzzelle, wobei jede Zelle die Nachbarschaft nach den gleichen Regeln bestimmt wird.
Es gibt für das zweidimensionale Gitter sehr viele Möglichkeiten zur Definition von Nachbarschaften, jedoch sind die **Von-Neumann-Nachbarschaft** und die **Moore-Nachbarschaft** von praktischer Bedeutung.

Abbildung 2.3.: Nachbarschaften für zweidimensionales Gitter: a) Von-Neumann-Nachbarschaft und b) Moore-Nachbarschaft [Sch14].

In Abb. 2.3 ist die Von-Neumann-Nachbarschaft zusehen, diese berücksichtigt vier direkte angrenzenden Nachbarzellen, mit denen sie eine gemeinsame Kante hat. Die Moore-Nachbarschaft hingegen berücksichtigt zusätzlich die vier Eckzellen, sodass sie insgesamt acht Nachbarzellen besitzt.

2.1.4. Übergangsregel

Die wichtigste Eigenschaft eines ZA ist die Übergangsregel. Sie definiert den Zustand einer Zelle zum Zeitpunkt t_{k+1} in Abhängigkeit von den Zuständen der Zellen

in seiner Nachbarschaft zum Zeitpunkt t_k. Dabei bestimmen Regeln den Zustand einer Zelle im jeweils folgenden Zeitschritt.
Es werden grundsätzlich zwei Arten von Regeln unterschieden.

- Die deterministische Regeln bestimmt die Entwicklung eines ZA. Bei Wiederholung eines Automaten mit dem gleichen Startkonfiguration wird dieser die selben Ergebnisse liefern.
- Die stochastische Regeln geben eine Wahrscheinlichkeit für eine Zustandswechsel für die Referenzzelle an unter den jeweiligen Gegebenheiten an.

2.1.5. Randbedingungen

Ein diskreter Raum muss vorhanden sein um ein ZA simulieren zu können. Wie in Abschnitt 2.1.2 erwähnt ergibt sich der neue Zustand einer Zelle aus seiner Nachbarschaft. Liegt eine Zelle jedoch am Rand des Gitters müssen klare Regeln definiert werden, da die Nachbarzellen nicht im Gitter enthalten sind. Für diese Sonderfällen müssen daher besondere Regeln festgelegt werden und dies wäre die Randbedingungen.
Üblicherweise werden **reflektierende** und **periodische** Randbedingungen verwendet.
Bei reflektierende Randbedingungen ändert sich der Zustand der Referrenzzelle nicht, wenn ihre Nachbarschaft nicht komplett im Gitter vorhanden sind.

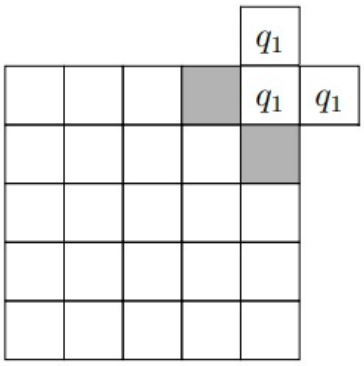


Abbildung 2.4.: Reflektierende Randbedingung am Beispiel Neumann-Nachbarschaft

Bei periodische Randbedingung werden die fehlenden Zellen durch Zellen am anderen Rand des Gitters ergänzt s. Abb. 2.5.

Abbildung 2.5.: Periodische Randbedingungen am Beispiel a) Neumann- und b) Moore-Nachbarschaft [Sch14].

2.1.6. Startkonfiguration

Der Grundaufbau und Ablauf eines ZA ist somit beschrieben. Es fehlt lediglich die Startkonfiguration oder auch die initiale Zuweisung eines Zustand für alle Zellen zum Zeitpunkt t_0, bevor die Simulation gestartet werden kann.
Es kann künstlich oder auch zufällig generiert werden. Die Bedeutung der Startkonfiguration wird im folgenden Abschnitt am Beispiel vom Game of Life veranschaulicht.

2.2. Game of Life

Das Game of Life (GOL) oder Spiel des Lebens wurde von John Horton Conway entwickelt um eine Simulation zur Entwicklung des Lebens [Sch14]. Es existiert unzähligen Variationen und Verallgemeinerungen zum GOL. In dieser Thesis wird auf das einfachste Grundmodell beschränkt.
Beim GOL besteht die Zustandsmenge Q aus zwei Zuständen, q_1 für tot und q_2 für lebendig.

$$Q = \{q_1, q_2\}$$

Die Moore-Nachbarschaft wird hier gewählt s. Abb.2.3 und die Entwicklung einer Zelle wird von definierten Regeln determiniert. Als Randbedingung erfolgt die periodische Randbedingung.
Die Regeln zur Definition eines Zeitschrittes werden wie Folgt beschrieben:

1. Lebt die Referenzzelle (x_{ij}) und es leben weniger als drei der neun möglichen Zellen, dann stirbt x_{ij} in der folgenden Generation.

2. Lebt x_{ij} und es leben in der Nachbarschaft drei oder vier der neun möglichen Zellen, dann überlebt x_{ij} in der folgenden Generation.

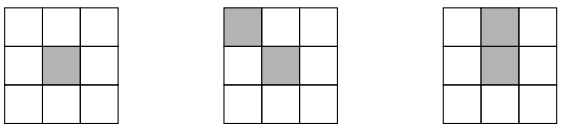
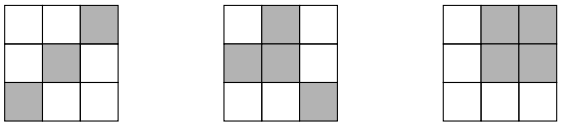
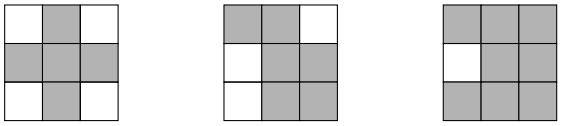
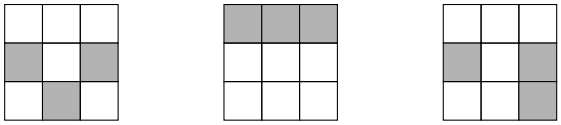
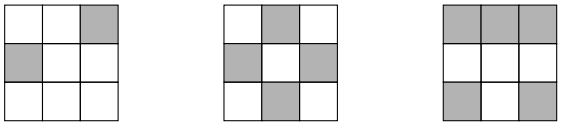
3. Lebt x_{ij} und es leben in der Nachbarschaft mehr als vier der neun möglichen Zellen, dann stirbt x_{ij} an Überbevölkerung in der folgenden Generation.

4. Wenn in der Nachbarschaft von x_{ij} genau drei von neun möglichen Zellen leben, x_{ij} jedoch nicht lebt, dann wird x_{ij} neu geboren.

5. Wenn in der Nachbarschaft mehr oder weniger als drei der neun möglichen Zellen leben, dann behält x_{ij} seinen aktuellen Zustand in der folgenden Generation.

Für alle Zellen treten genau einer der Fünf oben genannten Regeln auf und somit sind sie deterministisch. Die gewählten Startkonfiguration entscheidet über die Entwicklung der Zellen und sie können über viele Generationen hinweg zu komplexen Strukturen entwickeln.

Tabelle 2.1.: Skizze zur Visualisierung der Regeln im GOL. Es sind jeweils drei mögliche Nachbarschaften von x_{ij} zum Zeitpunkt t_k, welche im folgenden Zeitschritt t_{k+1} zu einem Zustand führen. Weiß steht für tot und grau für lebendig. Dabei ist die Referrenzelle x_{ij} immer das mittlere Feld [Sch14].

x_{ij} und die Nachbarschaft zum Zeitpunkt t_k	Zustand x_{ij} zum Zeitpunkt t_{k+1}
	
	
	
	
	

Kapitel 3

FPGA

Dieses Kapitel behandelt die Grundelementen zu FPGA. Die Architektur wird beschrieben, wobei der Schwerpunkt auf der Xilinx 7-Series FPGA gelegt wird. Ebenfalls wird die Entwicklungsumgebung Vivado kurz vorgestellt.

3.1. Field Programmable Gate Array

Field Programmable Gate Array (FPGA) besteht aus kleinen Hardwareelementen, die konfiguriert werden können, sodass sie eine logische Funktion ausführen. FPGA finden in der Praxis häufig als Prototypen für komplexe Schaltungen, Eingebetteten Systemen oder als Hardwarebeschleuniger Anwendungen [LB12].
Ein großer Vorteil von FPGAs gegenüber Prozessorsystemen ist, verschiedene Systemkomponenten auf einem Chip zu integrieren. Dies bedeutet nicht nur einen geringeren Platzbedarf, sondern auch eine erhöhte Performance.
In dieser Thesis wird ein ZedBoard der Firma FPGA der Serie 7 eingesetzt s. Abb. 3.1, jedoch gilt die Architektur, die in den folgenden Abschnitten beschrieben für die meisten modernen 7-Series FPGA-Chips von Xilinx.

3.1.1. ZedBoard Architektur

Die 7 Series Produktpalette besteht aus Virtex-7, Kintex-7, Artix-7 und dem Zynq-7000 All Programmable SoC, welche das ZedBoard dazugehört.
Generell besteht das ZedBoard aus zwei Hauptteilen, einem Verarbeitungssystem engl. Processing System (PS), das aus einem Dual-Core ARM Cortex-A9 Prozessor besteht und einem programmierbaren Logik (PL), welche das eigentliche FPGA

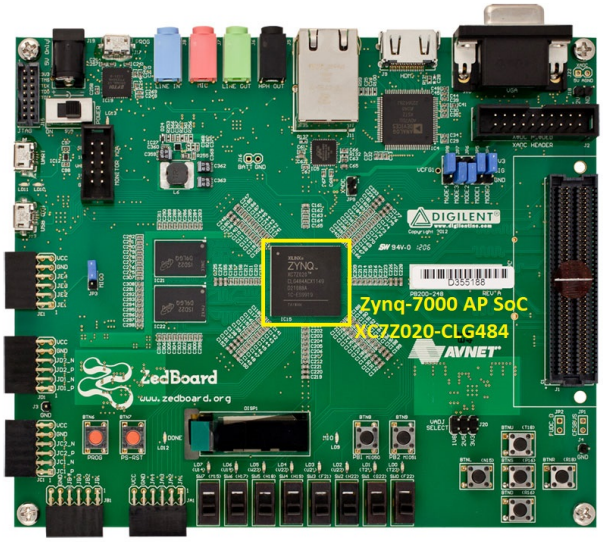


Abbildung 3.1.: Xilinx ZedBoard

entspricht. Die Verbindung zwischen PL und PS werden über das Advanced eXtensible Interface (AXI) verbunden s Abb. 3.2.

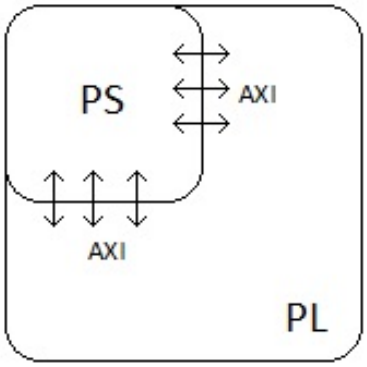


Abbildung 3.2.: Vereinfaches Modell der Zynq Architektur [Cro+14].

Der Hauptbaustein eines PL ist ein sogenannter konfigurierbare Logikblock engl. configurable logic block (CLB) und neben jedem CLB befinden sich eine Schaltmatrix, die eine flexible Routing Möglichkeiten bietet, um Verbindungen von einem CLB zum anderen herzustellen s Abb. 3.3.

3.1.2. Konfigurierbaren Logikblöcke

Konfigurierbaren Logikblöcke oder auch CLBs sind kleine regelmäßige Gruppierungen von Logikelementen, die in einem zweidimensionales Gitter auf dem PL angeordnet sind. CLBs implementieren die meiste Logik im FPGA und Ein CLB

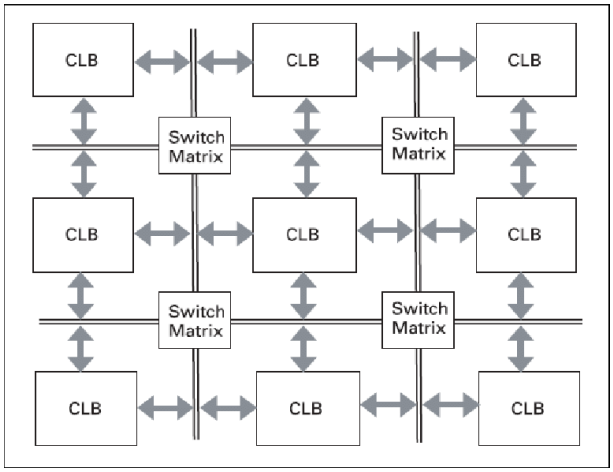


Abbildung 3.3.: Grundlegende Architektur eines PL.

kann zwei SLICEL oder einen SLICEL und einem SLICEM enthalten. Der Unterschied zwischen diesen beiden Slices -Typen sind, dass SLICEM mehr Konfigurationsmöglichkeiten bei der LUTs besitzt.
Jeder Slice verwendet die folgenden Grundelementen engl. Basic Element of Logic (BEL) um Logik-, Arithmetik und ROM-Funktionen bereitzustellen:

- Vier LUTs mit jeweils sechs unabhängige Eingängen (A1 bis A6) und zwei unabhängigen Ausgängen (O5 und O6), für jeden der vier LUTs in einem Slice (A,B,C und D)
- Acht Speicherelemente Flip Flop (FF), davon können vier flakengesteuerte D-FlipFlop oder pegelabhängige Latches konfiguriert werden.
- Multiplexer ermöglichen LUT-Kombinationen von bis zu vier LUTs in einem Slice.
- Einen Carry-Logik um eine dedizierte schnelle arithmetische Addition/Subtraktion in einem Slice durchführen.

Der Aufbau der Slices wird am Beispiel von SLICEL in der folgende Abbildung dargestellt.

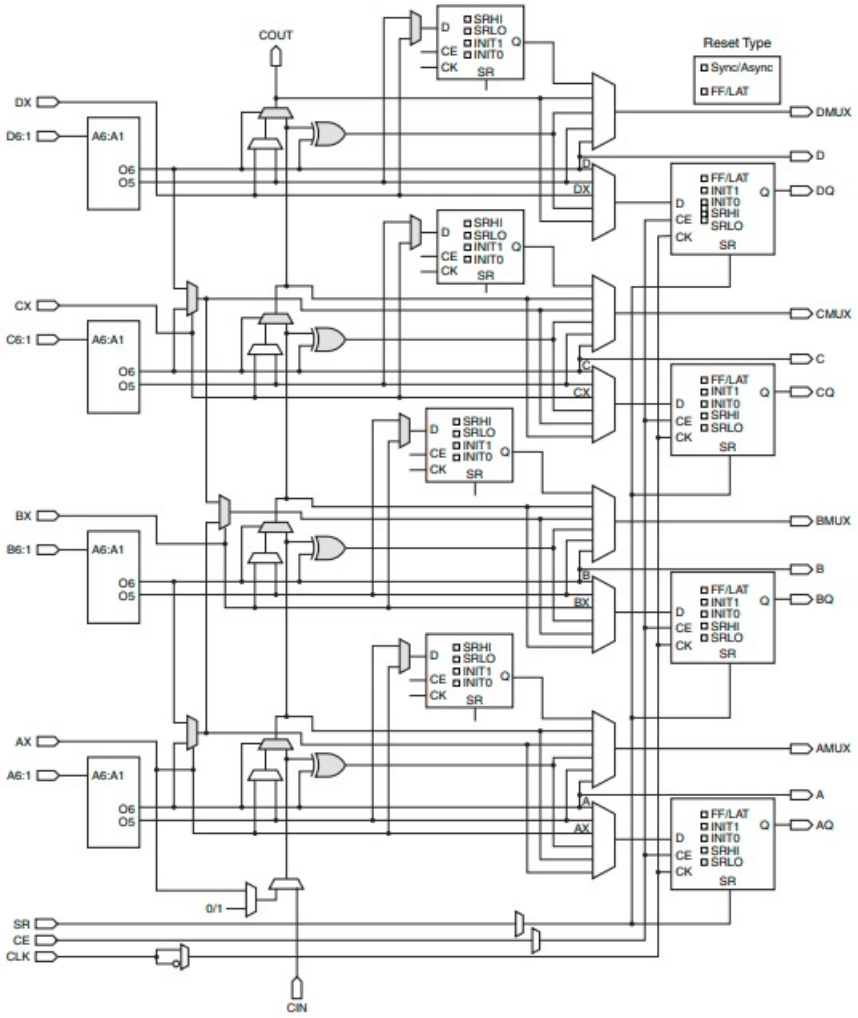


Abbildung 3.4.: Aufbau einer SLICEL [Xil16]

3.1.3. Look-Up Tabellen (LUT)

Look-Up Tabellen werden zur Realisierung kombinatorische Funktionen in CLBs verwendet. Eine LUT mit n Eingängen ist als eine beliebige Gatterfunktion mit n Variablen in Wahrheitstabellenform konfigurierbar und besitzt somit 2^n Speicherplätze. Zusätzlich befinden sich in jedem Slice drei Multiplexer, die zusammen geschaltet werden können umso komplexere logische Funktionen realisieren.

3.1.4. Carry-Logik

Für dedizierte schnelle arithmetische Addition/Subtraktion in einem Slice ist ein Carry-Logik vorgesehen. Die Carry-Logik besitzt zehn Eingänge (S-Eingängen S0

- S3, D-Eingängen D0 - D3, CYINIT und CIN) und acht Ausgänge (O-Ausgänge O0 - O3 und CO-Ausgänge CO0 - CO3).
Der CYINIT-Wert bestimmt welche Operator gewählt werden soll, für eine Addition wird CYINIT auf 0 und für eine Subtraktion wird der Wert auf 1 gesetzt.
Der CIN-Eingang und COUT-Ausgang wird zur Kaskadierung von Slices verwendet um eine längere Carry-Kette zu bilden. Die O-Ausgänge enthalten die Ergebnisse der Addition/Subtraktion und die CO-Ausgänge errechnet den Übertrag für jedes Bit. Zu erwähnen ist noch der Ausgang CO3, dieser wird mit dem COUT-Ausgang verbunden.

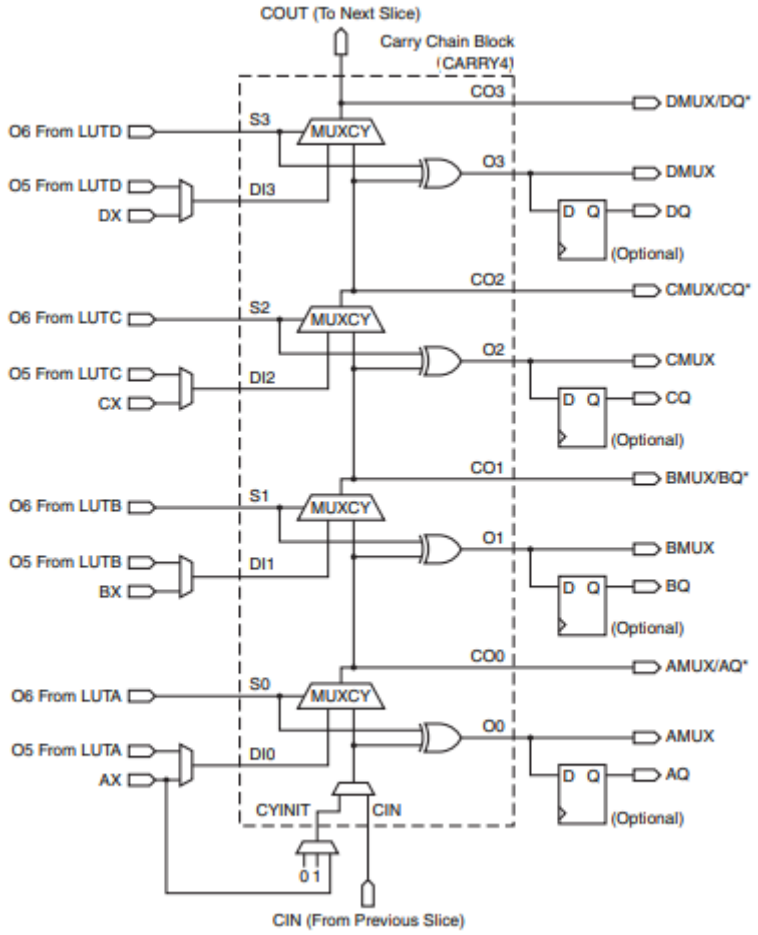


Abbildung 3.5.: Aufbau einer Carry-Logik [Xil16]

3.2. Vivado Design Suite

Vivado Design Suite ist eine Entwicklungsumgebung der Firma Xilinx zur Synthese und Analyse von HDL-Design. In dieser Abschnitt wird der Ablauf sowie die wichtigsten Funktionen des Vivado-Tools kurz vorgestellt, welche für in dieser Thesis relevant sind.

3.2.1. Designablauf

In jedes technisches Systems steht zu Beginn der Systementwurf. Aus gegebener Anforderungen werden Spezifikationen des Systems definiert. Spezifikationen bilden die Grundlagen für abstrakte Systembeschreibung, die alle benötigten Systemkomponenten sowie Schaltplänen enthält, dafür müssen Systembeschreibungen in lesbarer Form vorliegen (Entwurfseingabe). Gängige Entwurfseingabe Formate sind Hardwarebeschreibungssprachen (HDL[1]), beispielsweise VHDL oder Verilog. Aus HDLs können Simulations- und Sythese-Tools gut interpretiert werden und das gilt sowohl für Struktur- als auch Verhaltensbeschreibungen eines System. Aus einer HDL-Beschreibung wird zunächst überprüft, durch Simulationen, ob der Entwurf die Spezifikation des Systems erfüllt (Validierung).

Ist die Validierungsphase erfolgreich, folgt dann die Sythese. Hier wird der Entwurf vom Synthesewerkzeug analysiert und in eine sog. Netzliste überführt. Die Netzliste ist unabhängig von der Zieltechnologie und besteht aus Komponenten der Digitaltechnik wie Logikgattern, FFs etc.. .
In der nächsten Phase müssen die sogenannten “Constraints” angegeben werden womit das System betrieben werden soll. Hierzu gehören Pin-Belegung, Taktrate und andere physikalischen Implementierungsdetails. Anhand von Constraints sowie die Netzliste werden folgende Schritte (*Map und PAR*) abgearbeitet und zu einem Schritt zusammengefasst [RS20].

Beim *Mappen* (engl. abbilden) werden Logik auf die verfügbaren Ressourcen auf das Ziel-FPGA abgebildet. Als Beispiel werden Boole’schen Gleichungen auf LUTs realisiert, Speicherelemente werden auf geeigneter FF ausgewählt.
Beim Platzieren und Verdrahten Place and Route (PAR) werden z.B LUT oder FlipFlop Hardware-Ressourcen auf dem FPGA zugewiesen (platzieren) und

[1] Hardware Description Language

nachfolgend Verbindungen mit dem Verdrahtungsressourcen definiert (verdrahten).

Demnach steht nun eine vollständige Funktionsbeschreibung, die auch Design bezeichtet wird. Zur Überprüfung des Designs sollte eine Timing-Simulation durch geführt werden um zu schauen, ob der Entwurf allen zeitlichen Anforderungen erfüllt. Abschließend kann im letzten Schritt ein Bitstrom generiert und auf das Ziel-FPGA geladen werden.

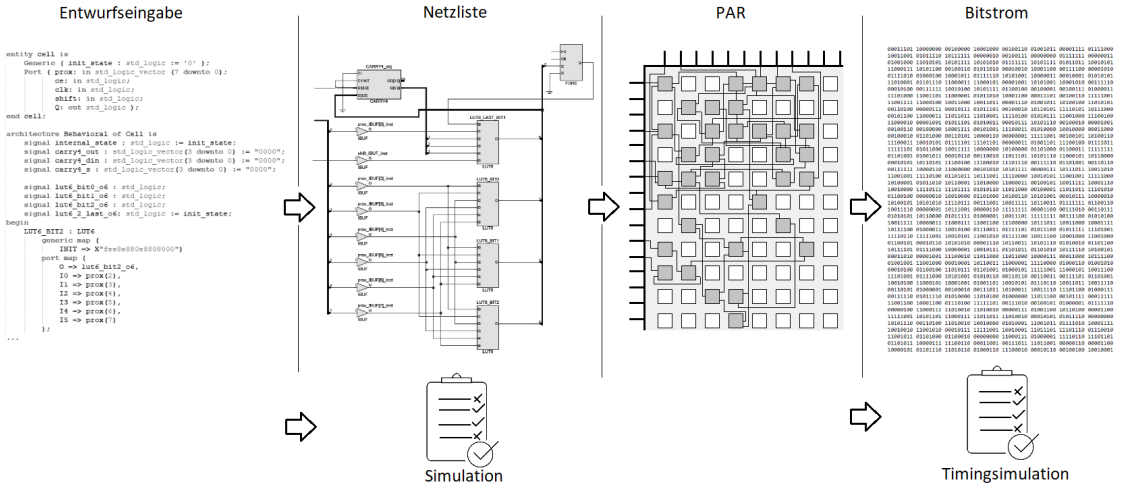


Abbildung 3.6.: Grundlegende Entwurfsablauf. Begin von links nach rechts

3.2.2. Entwurfssichten

Wie oben erwähnt lässt sich der HDL-Entwurf eines System in der Strukturbeschreibung (`structural`) und in der Verhaltensbeschreibungen (`behavioral`) modellieren.
Bei der Strukturbeschreibung werden Eigenschaften eines Modells durch sein inneren Aufbau aus Unterkomponenten dargestellt [Gun14]. Diese Komponenten werden in Vivado durch Markro-Primitiven aus der UNISIM-Bibliothek zur Verfügung gestellt.

```
1 library UNISIM;
2 use UNISIM.VComponents.all;
3
4 DFF_inst: FDRE
5 generic map (INIT => '0')  -- Initial value
6 port map (Q => Q,          -- Data output
```

```
7           C => clk,        -- Clock input
8           CE => CE,        -- Clock enable input
9           R => R,          -- Reset input
10          D => D);         -- Data input
11  -- End of DFF_inst instantiation
```

Listing 3.1: Strukturbeschreibung eines D-FlipFlop

In der Verhaltensbeschreibung werden Ein/Ausgangssignalen von digitaltechnischen Komponenten modelliert. Die Komponenten verzweigen nicht weiter in Unterkomponenten [Gun14].

```
1  entity DFF is
2     port(clk : in std_logic;  -- Clock input
3          D   : in std_logic;   -- Data input
4          R   : in std_logic;   -- Reset input
5          Q   : out std_logic); -- Data output
6  end DFF;
7
8  architecture Behavioral of DFF is
9  begin
10     process(Clk)
11     begin
12         if(rising_edge(Clk)) then
13             if(R='1') then
14                 Q <= '0';
15             else
16                 Q <= D;
17             end if;
18         end if;
19     end process;
20 end Behavioral;
```

Listing 3.2: Verhaltensbeschreibung eines D-FlipFlop

3.2.3. Entwurfseinschränkungen

Entwurfseinschränkungen (Constraints) sind physikalische oder zeitliche Einschränkungen, die das Verhalten eines entwickelten Designs spezifizieren. Xilinx Design Constraints (XDC) setzen eine Richtlinie an ein Design, die beim PAR eingehalten werden muss, um Performance-Ziele zu erreichen. Beispielsweise können Timings-Constraints Taktsignale definieren oder Physical-Constraints Ein-/Ausgangs-Pin auf dem FPGA festgelegt werden.
Zusätzlich können mithilfe von LOC[2]- und BEL[3]-Constraints eine exakte Positionen

[2]Location constraints platziert ein logisches Element in einer bestimmten Slice
[3]Basic Element of Logic, positioniert ein logisches Element innerhalb einer Slice

von LUTs und FlipFlops vorgeben und mit ROUTE-Constraints bestimmte Signalpfade erzwungen werden [Xil13]. Location Constraints (LOC)-Positionen werden über ein Koordinatensystem der Slices, durch Angabe ihrer x- und y-Position, mit Ursprung in der unteren linken Ecke des FPGA festgelegt s. Abb. 3.7.
Die Definition von Constraints erfolgt mittels Tool command Language (Tcl)-Befehlen in der XDC-Dateien oder sie können in der Tcl-Console direkt ausgeführt werden. Ebenfalls lässt sich Vivado vollständig mittels Tcl-Befehlen bedienen, die zu einem automatisierten Ablauf als Skript zusammengestellt werden.

Kapitel 4

Implementierung

Das Ziel dieser Thesis ist ein klassisches zelluläre Automaten (ZA) Systems, das Game of Life (GOL), unter der Verwendung eines FPGAs zu implementieren und zu generieren.
Aufgrund der Ähnlichkeit der inneren Architektur von FPGAs mit dem ZA, lässt sich eine exakte Abbildung der Zellen auf die Hardware-Zellen, in dem Fall Slices, übertragen. Somit hängt die Größe des Spielfeldes, in dem sich die Zellen befinden, mit der Anzahl der Slices im FPGA zusammen. Da jeder Hardware-Zelle synchron angesteuert wird entspricht ein Zeitschritt genau einer Taktflanke.
Zunächst wird eine einfache GOL Zelle in VHDL entworfen und auf die funktionale Anforderungen simuliert. Als nächstes werden die Logik der Zelle minimiert, durch die Verwendung von UNISIM-Bibliothek. Anschließend wird ein Modul entworfen, welches das Spielfeld sowie die Randbedingungen (s. Kapitel 2.1.5) definiert.

4.1. Zell Architektur

4.1.1. Vorüberlegungen

Wie im Kapitel 2 beschrieben sind die Zellen in das GOL in zweidimensionaler Raum miteinander Verbunden, dies soll ebenfalls im FPGA implementiert werden. Durch die Platzierung liegen alle benachbarten Zellen physikalisch nebeneinander, dies könnte auf eine mögliche Verbesserung der Timing erfolgen.
Sind alle Zellen physikalisch nebeneinander muss noch die Kommunikation, die zum Auslesen oder Setzen des Zustandes der gesamten Matrix erforderlich ist, berücksichtigt werden. Mit einer speziellen Verschiebungsmethode (**shifting**) ist es

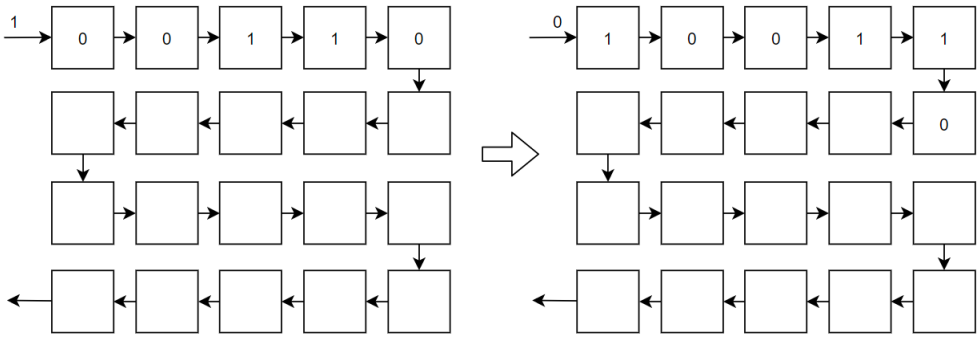


Abbildung 4.1.: Methode zur Verschiebung der Daten zu den einzelnen Zellen im zweidimensionaler Raum. Links das System zum Zeitpunkt t_k und rechts t_{k+1}

Möglich eine komplette Startkonfiguration für alle Zellen zu laden und auszulesen s. Abb. 4.1.
Die Zustandsdaten fließen beginnt von der linken oberen Zelle und werden pro Taktzyklus weiter an seinem rechten Nachbarzelle verschoben. Erreicht der Datenfluss das Ende einer Zeile, so wird die untere benachbarte Zelle als Ziel gewählt und die Flussrichtung geändert. Die Daten werden so weitergereicht bis zum Ausgang, diese sequentielle Vorrichtung erinnert an ein Schieberegister wobei die gerichtete Bewegung hier nicht geändert werden kann. Zur einfache Ansteuerung der Verschiebungsoperation wird ein 1-Bit `shift`-Signal hinzugefügt.
Zusammengefasst muss der Entwurf folgende Anforderungen beinhalten:

- Die Logik der GOL Übergangsregeln,
- Die Möglichkeit den Zustand eines benachbarten Zelle zu übernehmen (**shifting**).
- Und das Modul soll Taktsynchron arbeiten.

4.1.2. Entwurf des Zell-Moduls durch Verhaltensbeschreibung

Mit der Vorüberlegung und die Definition der GOL Übergangsregeln lässt sich zunächst sehr schnell in VHDL eine Verhaltensbeschreibung implementieren.
Die Schnittstelle (`entity`) besteht aus ein Eingangsbus, ein Ausgangssignal und ein `shift`-Signal. Das Modul arbeitet synchron, aus diesem Grund sind zwei zusätzliche Eingängen vorgesehen, einen Taktsignal und einen `ce`-Signal.
Der Eingangsbus hat eine Breite von 8-Bit und besteht aus jeweils Zustandsbits der Nachbarzellen, wobei 0 für eine “tote” und 1 “lebendige” Zelle steht. Das

Ausgangssignal speichert den aktuellen Zustand der Zelle.
Listing 4.1 zeigt die `entity` des Moduls.

```
entity cell is
    Port ( prox: in std_logic_vector (7 downto 0); -- proximity (Nachbarschaft)
           ce: in std_logic;
           clk: in std_logic;
           shift: in std_logic;
           Q: out std_logic );  -- 1 for "ALIVE" and 0 for "DEAD"
end cell;
```

Listing 4.1: Zell entity.

Zur Bestimmung der lebenden Nachbarzellen wird ein internes Signal definiert `prox_counter`. Dies summiert asynchron den Zustand alle Nachbarzellen. Das Ergebnis wird in ein getakteter Prozess verwendet um mit der Übergangsregel den Zustand für den nächsten Zeitschritt zubestimmen.

```
    x11 <= 1 when prox(0) = '1' else 0;
    x12 <= 1 when prox(1) = '1' else 0;
    ...
    x33 <= 1 when prox(7) = '1' else 0;
    prox_counter <= x11 + x12 + x13 + x21 + x23 + x31 + x32 + x33;
    RULES_PROC: process(clk)
    begin
        if rising_edge(clk) then
            if ce = '1' and shift = '1' then -- calculation
                -- rule 4
                if ((internal_state = '0') and (prox_counter = 3)) then
                    internal_state <= '1';
                -- rule 1 & 3
                elsif ((internal_state = '1') and ((prox_counter < 2) or (
                    prox_counter > 3))) then
                    internal_state <= '0';
                -- rule 2 & 5
                else
                    internal_state <= internal_state;
                end if;
            -- shift
            elsif ce = '1' and shift = '0' then     -- proximity field index
                internal_state <= prox(0);          -- |-----|-----|-----|
            else                                    -- | x11 | x12 | x13 |
                internal_state <= internal_state;   -- |-----|-----|-----|
            end if;                                 -- | x21 | xij | x23 |
        end if;                                     -- |-----|-----|-----|
    end process;                                    -- | x31 | x32 | x33 |
    Q <= internal_state;                            -- |-----|-----|-----|
end Behavioral;
```

Listing 4.2: Codesegment zur Funktionsverhalten des Zell-Moduls.

Das invertiertes `shift`-Signals steuert die Verschiebungsoperation und als Platzhalter für eine Nachbarzelle wird das Signal `prox(0)` gewählt.
Für den Datenfluss sorgt ein übergeordnetes `grid`-Modul, welches im späteren Zeitpunkt näher erläutert wird.

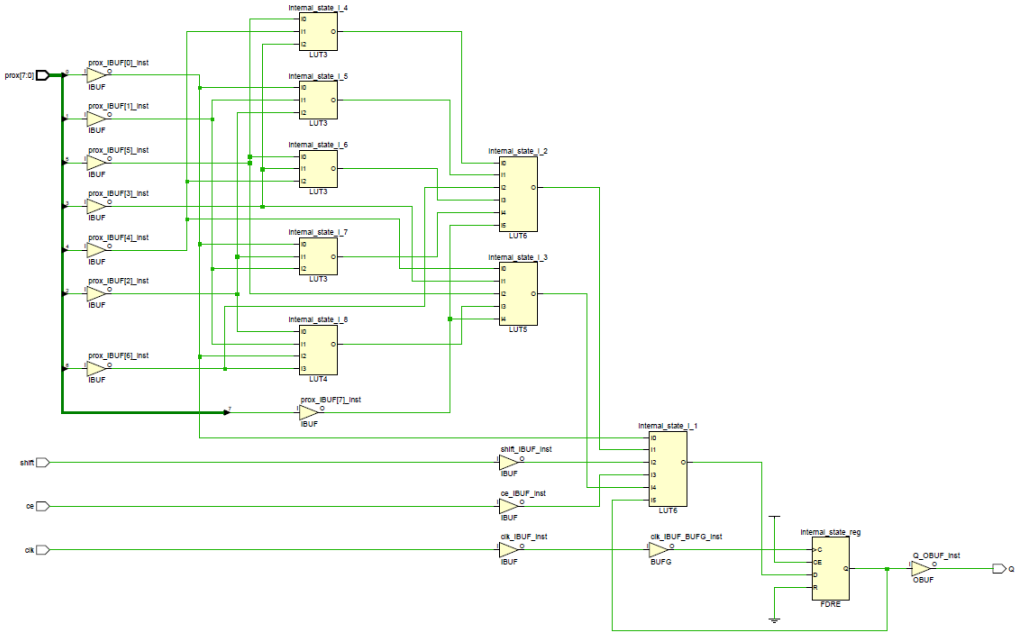


Abbildung 4.2.: Syntheseergebnis der Verhaltensbeschreibung

Nach erfolgreichen PAR sind logische Elemente des Zell-Moduls auf ein CLB bzw. zwei Slices verteilt s. Abb. 4.3. Um eine exakte Abbildung einer GOL-Zelle auf einen Slice zu beschränken muss also die Logik minimiert werden.

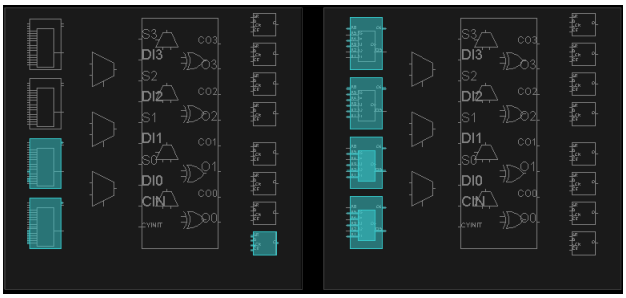


Abbildung 4.3.: Zell-Modul Device Ansicht Verhaltensbeschreibung.

4.1.3. Logik Minimierung durch Strukturbeschreibung

Da die `entity` des Zell-Moduls alles was einer Zelle benötigt, bleibt sie unverändert. Abb. 4.4 zeigt das Blockdiagramm und die darin verwendeten Ein/-Ausgänge sowie Komponenten, welche die Logik einer Zelle enthält.

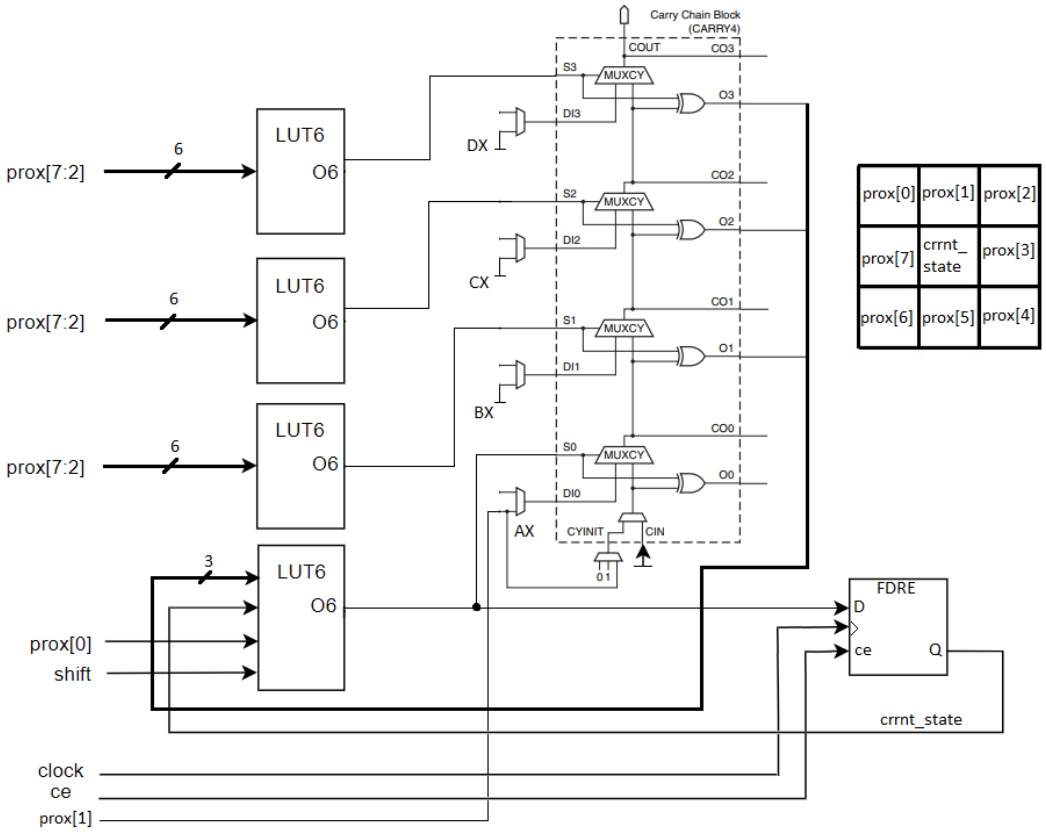


Abbildung 4.4.: Zell-Logik als Blockdiagramm.

Die ersten drei LUTs nehmen den Zustand von sechs der umgebenden Zellen ein und werden zusammen mit eine weitere Nachbarzelle in die Carry-Logik verbunden, um eine Addition durchzuführen. Das Ergebnis ist eine 3-bit Zahl, die der Anzahl der lebenden Zellen entspricht. Die vierte LUT nimmt als Eingang die drei höchstwertigte Bits aus der Carry-Logik, die letzte Nachbarzelle plus den aktuellen Zustand der Referenzzelle und das `shift`-Signal und gibt den neuen Zustand der Zelle aus. Der neue Zustand wird in ein FlipFlop gespeichert.
Somit lässt sich die gesamte Logik in einen einzigen SLICEL oder SLICEM realisieren. Durch die Instanziierung von Komponenten der UNISIM-Bibliothek lässt sich der Entwurf wie im Blockdiagramm realisieren. Dabei muss jedoch die richtige Konfiguration der LUTs beachtet werden und wird in folgenden erläutert.

4.1.4. LUT6 Konfiguration

Eine LUT6 Komponente realisiert eine beliebige boolesche Funktion mit $n = 6$ Eingängen, die durch einen INIT-Attribut festgelegt ist. Das entspricht eine Wahrheitstabelle mit 2^6 Eingangskombinationen. Ein INIT-Attribut besteht aus einem 64-Bit Hexadezimalwert und muss bei der Instanziierung der LUT angegeben werden. Ein Signal an einer LUT-Eingängen ändert von dieser LUT implementierte logische Gleichung und somit auch den Ausgangswert. Die Hauptlogik einer GOL-Zelle an einer LUT-Instanz kann durch Bestimmung der INIT-Attribut implementiert werden. Für die Ermittlung des INIT-Wertes werden zwei Methoden vorgeschlagen [Xil12].

- **Logic Table Method**: Bei dieser Methode wird eine Wahrheitstabelle mit 2^6 Eingangskombinationen erstellt und aus deren Ausgangswerten der INIT-Wert festgelegt.
- **Equation Method**: Hier wird für jeden Eingang der LUT Parameter mit entsprechende Wahrheitswert definiert, um eine gewünschte logische Gleichung aufzustellen. Danach wird die Gleichung der LUT als INIT-String in die LUT-Instanz eingefügt.

In dieser Thesis wird die **Logic Table Method** implementiert, da es sich sehr einfach mit einer automatisierten Python-Skript alle Eingangskombinationen zu generieren und Anhand der Ausgangswerten den INIT-Wert abzulesen.

Betrachtet wird zunächst die ersten drei LUTs. Sie zählen die sechs lebenden umgebenden Zellen und das Ergebnis ist eine 3-Bit Zahl, wobei jeder LUT für eine Bitwertigkeit steht. Das folgende Skript generiert für alle drei LUTs die entsprechenden INIT-Werten.

```
1  def gen_LUT6(which_lut :int):
2      combination = [f'{n:06b}' for n in range(64)] # truth table with 2^6 inputs
3      idx = 0
4      bit_array = bitarray(64, endian='big')
5      bit_array.setall(0)
6      for inputs in combination:
7          cnt = inputs.count('1') # count living cells
8          str_count = (f'{cnt:03b}')[::-1]
9          if '1' in str_count[which_lut]:
10             bit_array[idx] = 1
11         idx += 1
12     bit_array = bit_array[::-1] # change endian (inverse bit array)
```

```
    print(f"LUT6[{which_lut}]: " + "INIT = " + tohex(BitArray(bit_array).int, 64))
for i in range(3):
    gen_LUT6(i)
''' Outputs:
LUT6[0]: INIT = 0x6996966996696996
LUT6[1]: INIT = 0x8117177e177e7ee8
LUT6[2]: INIT = 0xfee8e880e8808000 '''
```

Listing 4.3: Funktion zur Generierung der INIT-Werte der ersten drei LUT6 Komponenten.

Die letzte LUT beinhaltet sowohl Logik der Übergangsregeln als auch die Zustandsübernahme eines benachbarten Zelle. Für `shift` $= 1$ wird der neue Zustand der Zelle ausgegeben, dies wird durch die folgende Wahrheitstabelle veranschaulicht.

shift	carry(3)	carry(2)	carry(1)	prox(0)	crrnt_state	O
1	0	0	1	1	1	1
1	0	1	0	0	1	1
1	0	1	0	1	0	1
1	0	1	0	1	1	1
1	0	1	1	0	0	1
1	0	1	1	0	1	1

Tabelle 4.1.: Zeilen der Wahrheitstabelle, bei denen der neue Zustand der Zelle den Wert 1 (lebendig) annimmt.

Die Zustandsübernahme findet statt, wenn `shift` $= 0$ ist, dabei übernimmt der Ausgang der LUT den Wert von `prox(0)`. Das Skript für die Generierung der INIT-Wert sieht dann wie folgt aus.

```
def gen_LUT6_3_shift():
    combination = [f'{n:06b}' for n in range(64)]
    idx = 0
    bit_array = bitarray(64, endian='big')
    bit_array.setall(0)

    masks_to_return1 = ["00111", "01001", "01010", "01011", "01100", "01101"]
    for inputs in combination:
        if '0' in inputs[0]:  # 0 for shifting
            bit_array[idx] = int(inputs[4])
        elif '1' in inputs[0]: # 1 for calculation
            if inputs[1:] in masks_to_return1:
                bit_array[idx] = 1
        idx += 1
    bit_array = bit_array[::-1] # change endian (inverse bit array)
    print(f"LUT6[3]: " + "INIT = " + tohex(BitArray(bit_array).int, 64))

gen_LUT6_3_shift()
''' Outputs:
```

```
20  LUT6[3]: INIT = 0x00003e80cccccccc ’’’
```

Listing 4.4: Funktion zur Generierung der INIT-Werte der vierten LUT6 Komponente.

Aus diesen Ergebnissen können alle benötigten Komponenten der Zell-Modul instanziiert werden s. Anhang A.

4.1.5. Ergebnisse der Strukturbeschreibung

Nach der Synthese entsteht folgende schematische Darstellung des Moduls.

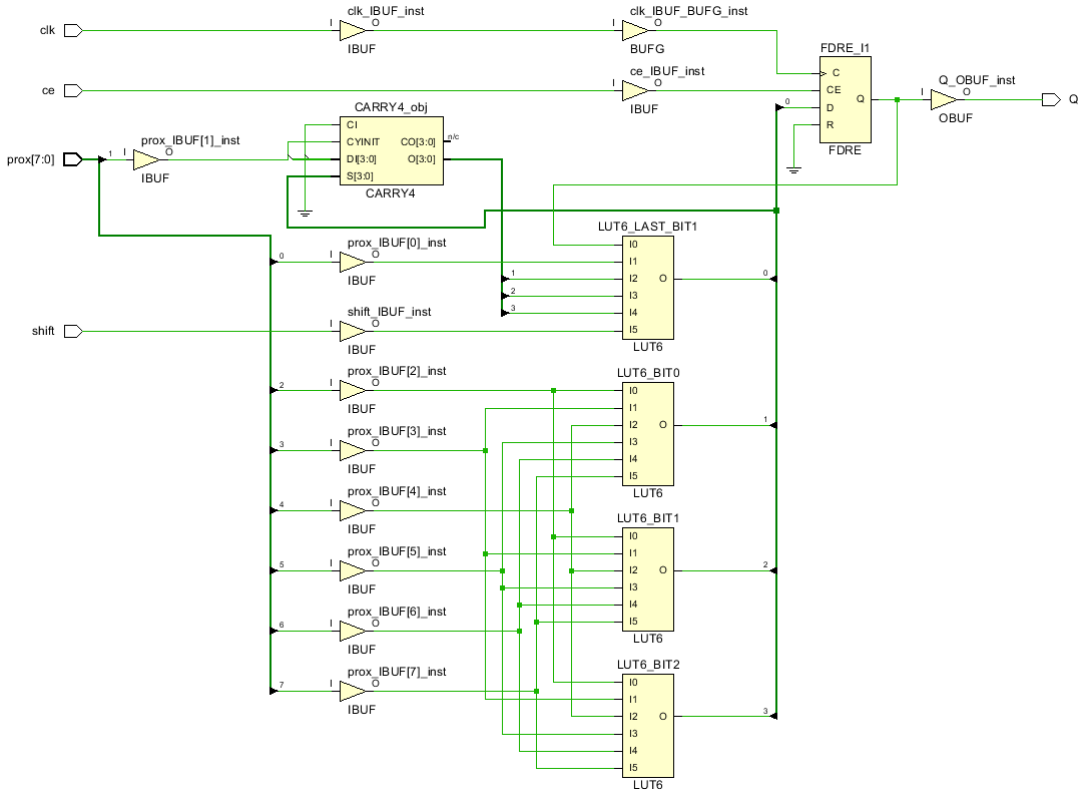


Abbildung 4.5.: Syntheseergebnis der Strukturbeschreibung

Das Ergebnis nach der automatischen FPGA-Platzierung und Verdrahtung (PAR) zeigt, obwohl alle Komponenten die sich innerhalb eines Slices befinden instanziiert wurde, verteilt das PAR-Tool trotzdem die Komponenten auf zwei Slices s. Abb. 4.6.

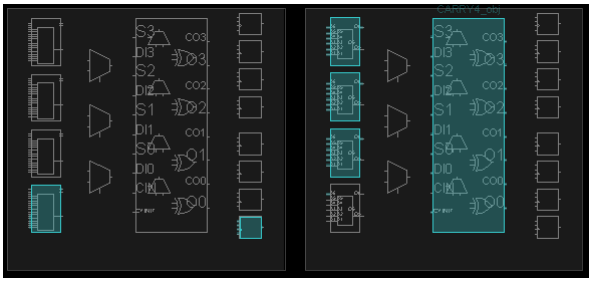


Abbildung 4.6.: Das vergrößerte Device-Ansicht mit Darstellung der belegten Hardwareressourcen in Cyan.

Bei der Synthese wurde eine Netzliste des Zell-Moduls generiert, so können mit Hilfe der Tcl-Befehlen Komponenten auf einem Slice erzwungen werden. Dabei wird zuerst ein neues Marko-Objekt erstellt, welches dann Komponenten aus der Netzliste und die Position eines Slices im FPGA übergeben. Folgende Tcl-Befehlen platziert ein Zell-Modul an der Slice-Position $X = 54$ und $Y = 49$.

```
1  create_macro cell0
2  update_macro cell0 {FDRE_I1 X54Y49 LUT6_BIT1 X54Y49 CARRY4_obj X54Y49
       LUT6_LAST_BIT1 X54Y49 LUT6_BIT2 X54Y49 LUT6_BIT0 X54Y49}
3
4  startgroup
5  place_cell {FDRE_I1} SLICE_X54Y49/AFF
6  place_cell {LUT6_BIT1} SLICE_X54Y49/C6LUT
7  place_cell {CARRY4_obj} SLICE_X54Y49/CARRY4
8  place_cell {LUT6_LAST_BIT1} SLICE_X54Y49/A6LUT
9  place_cell {LUT6_BIT2} SLICE_X54Y49/D6LUT
10 place_cell {LUT6_BIT0} SLICE_X54Y49/B6LUT
11 endgroup
```

Listing 4.5: Tcl-Befehlen zur location constraints eines Zell-Moduls.

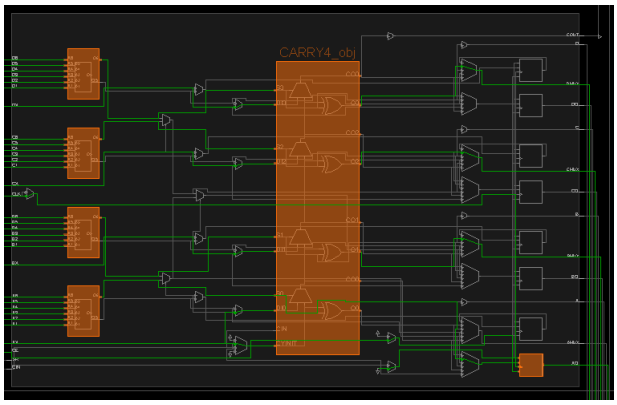


Abbildung 4.7.: Das vergrößerte Device-Ansicht nach location constraints zeigt die platzierten Komponenten in Orange und in grün die Verbindungspfade.

Die Nachbildung einer GOL-Zelle auf einen Slice ist somit erfolgreich. Anschließend veranschaulicht folgende Abbildung das Ergebnis der funktionale Simulation des Moduls, dazu wird einen Testbench erstellt, welche ein eigenes VHDL-Modul ist s. Anhang A.
Zuerst wird `ce` auf den Wert 0 gesetzt, `shift` auf 1 (für nicht shifting). Nach einer Zeitspanne wird `ce` synchron zum Systemtakt auf 1 gesetzt und drei mögliche Nachbarschaften simuliert. Die Simulation zeigt den neuen Zustand der Zelle nach einer Taktverzögerung, mit einer zusätzliche Laufzeit von $100ps$. Dies ist zu vernachlässigen, da diese minimale Verzögerung keinen Einfluss auf das Ergebnis hat.

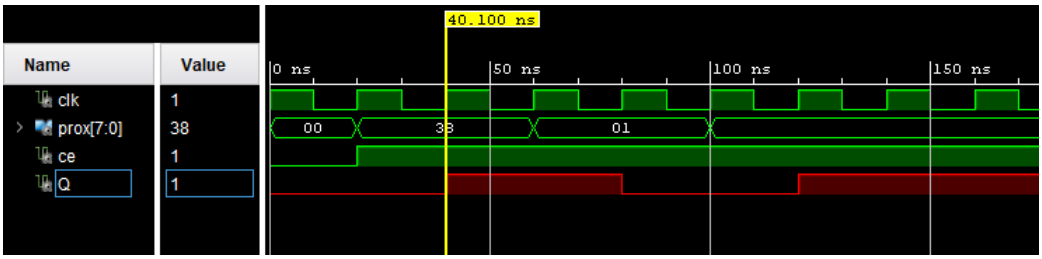


Abbildung 4.8.: Verhaltenssimulation des Zell-Moduls.

4.2. Implementierung des Spielfeldes

Das Spielfeld realisiert ein übergeordnetes `grid`-Modul, dessen Größe mit zwei `generic`-Parameter, Breite und Höhe, konfigurieren lässt. Durch `generate`-Anweisung lassen sich indizierte Zell-Modul Instanziierung in einer `for`-Schleife mit statisch vorgegebenen Ausführungshäufigkeit aufbauen. Um die Komplexität der Randbedingung gering zu halten, werden Nachbarschaften, die nicht im Spielfeld existieren den Zustand 0 (tot) zugewiesen.

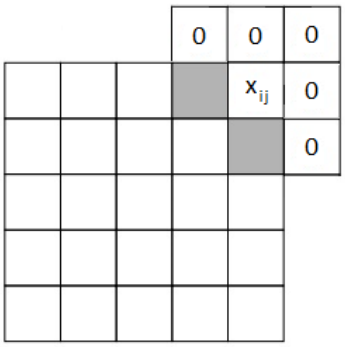


Abbildung 4.9.: Randbedingung des Spielfeldes anhand eines Beispiels.

Nachbarschaften werden durch Verknüpfung von Ausgängen der relevanten Zell-Modulen in lokalen Signalen zusammengefasst und an das aktuell indizierte Zell-Modul übergeben.
Da die Logik für shifting bereits in der Zell-Modul implementiert ist, muss in der `grid`-Modul lediglich die korrekte Festlegung der Nachbarschaft gewährleistet werden. Die `if generate`-Anweisung fragt die Position der Zelle innerhalb des Spielfeldes ab, erzeugt eine Zell Instanz und weisen diese seine Nachbarschaften zu. Der VHDL-Entwurf des `grid`-Moduls befindet sich in Anhang B.

4.2.1. Ergebnis

Zur Simulation wird ein kleines Spielfeld der Größe 3x3 erzeugt, dabei wird zunächst eine Startkonfiguration in das Spielfeld geladen. Nach einer Zeitspanne läuft das System für zwei Generationen, das entspricht genau zwei Taktzyklen. Zum Auslesen der Daten wird eine mit Nullen gefüllte Startkonfiguration in das System geladen und ermöglichen so das Auslesen der Ergebnisse.
Die Testbench kann grafisch dargestellt werden als:

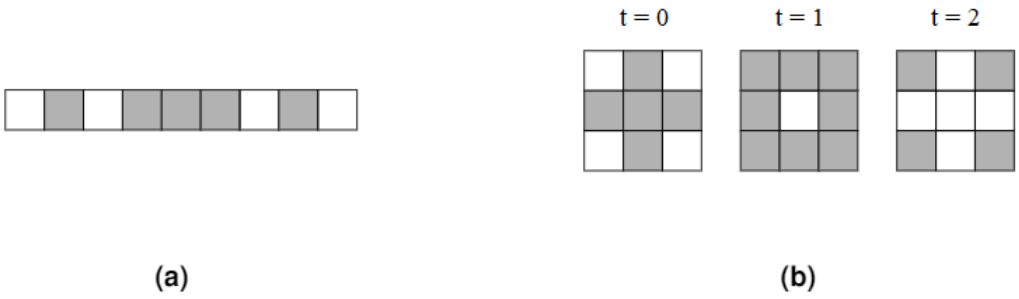


Abbildung 4.10.: Grafische Vorstellung der Testbench: a) Die Startkonfiguration in sequenzielle Darstellung und b) Erwartetes Ergebnis nach $t = 3$ Generationen. Wobei Weiß für “tot” und Grau für “lebendig”.

Die Verhaltenssimulation zeigt, dass das implementierte `grid`-Modul wie erwartet funktioniert und wird im Anhang X dargestellt.

4.3. IP Core zur Datentransfer

Zur Konfiguration oder Ansteuerung des Spielfeldes wird ein FPGA-basierter Mikrocontroller, der MicroBlaze-Softcore-Prozessor, zum Einsatz kommen. Dieser besteht ausschließlich aus FPGA-Slices und besitzt eine Schnittstelle zu einem DDR RAM sowie einen UART[1] zur Ein- und Ausgabe von Text [RS20].
Der MicroBlaze wird verwendet um Zellzustandsdaten über die AXI-Lite Schnittstelle an das GOL-System zu senden und zu empfangen. Außerdem wird die UART-Schnittstelle zur Demonstrationszwecken, Ergebnisse aus dem GOL zu einer Host-Rechner übertragen. Für die Wiederverwendbarkeit des GOL-Entwurfs wird diese in ein sogenannte IP-Core[2] zusammengefügt. Der Vorteil besteht darin den Entwicklungsprozess zu beschleunigen ohne auf die Korrektheit der Ergebnisse zu beeinträchtigen. Die gesamte Hardware-Architektur ist in Abbildung 4.11 dargestellt.

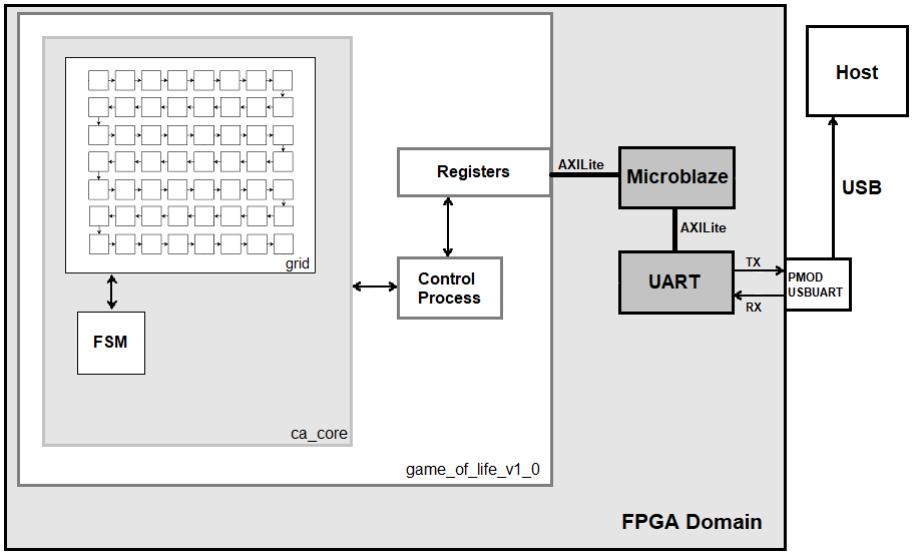


Abbildung 4.11.: Überblick der Systemarchitektur.

4.3.1. CA Core Modul

Das `ca_core`-Modul enthält die notwendigen Logik um das GOL-System zu steuern, zu initialisieren, zu lesen und ihren aktuellen Zustand zu halten. Konfigurationen zur Festlegung der Spielfeldgröße lassen sich mittels Angaben der

[1] Universal Asynchronous Receiver and Transmitter
[2] Intellectual Property z. dt. auch geistiges Eigentum

Breite und Höhe bestimmen und sie sind nicht zur Laufzeit änderbar. Lediglich die maximale Simulationszeit kann während des Betriebes geändert werden.
Ein endlicher Zustandsautomaten (*englisch* finite state machine, FSM) ist verantwortlich für die Steuerung aller Zell-Module, durch Registrierung von Befehlssignalen, beispielsweise Start oder Stop. Ob die Zell-Modulen Berechnung für die nächsten Generation oder Verschiebung der Daten ausgeführt wird, ist abhängig von der Betriebsmodus des FSM (Idle[3] und Iteration). In der ersten Modus wird die Verschiebung der Daten durchgeführt, wenn das `shift`-Signal gesetzt ist, ansonsten behalten alle Zell-Module ihren aktuellen Zustand. Im Iterationsmodus werden für alle Zellen im gleichen Taktzyklus der nächste Zustand berechnet. Der Prozess wiederholt sich bis die maximale Simulationszeit erreicht ist, dann wechselt der Automaten wieder in den Idle-Modus zurück.

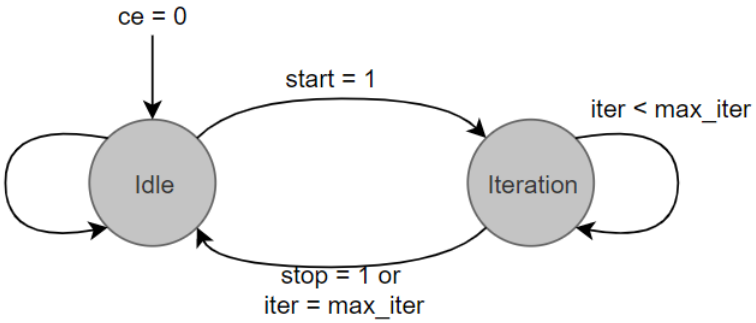


Abbildung 4.12.: Zustandsdiagramm zur Ansteuerung des GOL-Systems.

4.3.2. GOL IP-Core

Der GOL IP-Core dient zur Datentransfer mit einer AXI4-Lite Schnittstelle zwischen den `ca_core`-Modul und den Microblaze-Prozessor. Steuersignale sowie das Lesen und Schreiben der Daten werden mittels Registerzugriffe realisiert und wird im folgenden erläuert:

- Das Initialisieren der Startkonfiguration geschieht durch bitweise Übertragung. Das gesendete Bit wird zunächst in das GOLDIR (**G**ame **O**f **L**ife **D**ata **I**nput **R**egister) gespeichert. Nachdem GOLDIR beschrieben wurde, wird das `shift`-Signal gesetzt und das Bit in den `ca_core`-Modul geschoben.

[3] Leerlauf

- Steuerungssignale wie Start oder Stop werden in GOLCR (**G**ame **O**f **L**ife **C**ontrol **R**egister) geschrieben und wird vom IP-Core nach einer Taktzyklus bearbeitet.
- Das GOLICR (**G**ame **O**f **L**ife **I**teration **C**ontrol **R**egister) ist zur Konfiguration der Simulationszeit bestimmt.
- Das Lesen aller Zellzustände geschieht ebenfalls durch bitweise Übertragung. Dabei muss beachtet werden, einen "Dummy"-Bit zuerst in das GOLDIR zuschreiben, um dann einen Bit aus dem GOLDOR (**G**ame **O**f **L**ife **D**ata **O**utput **R**egister) zu Lesen.

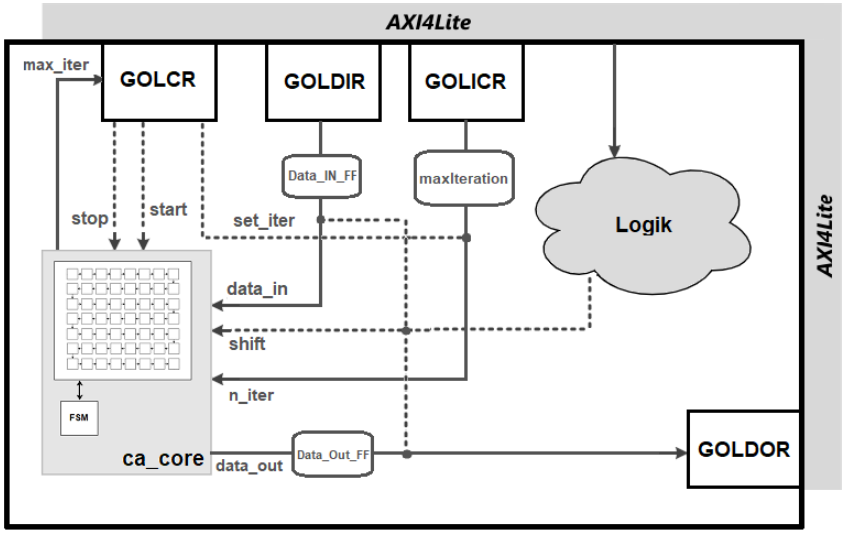


Abbildung 4.13.: Überblick GOL-IP.

Addr.	Reg. Name																	
0x00	GOLCR R/W	Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
			Obsolet															
		Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			Obsolet												GMI	GSI	GSP	GST
0x04	GOLICR R/W	Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
			0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x08	GOLDIR W	Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
			Obsolet															
		Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			Obsolet															0
0x0C	GOLDOR R	Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
			Obsolet															
		Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			Obsolet															0

Abbildung 4.14.: Registerstruktur der GOL-IP

4.4. Game of life Test-Applikation

Nach dem Importieren der zuvor entwickelten IP-Core in den Vivado Design Suite IP-Catalog, kann ein neues Block-Design erstellt und das GOL-Design als neue Komponente hinzugefügt werden. Weiterhin wird das Block-Design um den Microblaze-Block und die UART-Schnittstelle ergänzt. Die Standardeinstellung der Spielfeldgröße beträgt 18x12 (Breite x Höhe) und die Taktfrequenz liegt bei $100Mhz$. Abbildung 4.15 zeigt ein Diagramm des vollständig verbundenen Block-Designs.

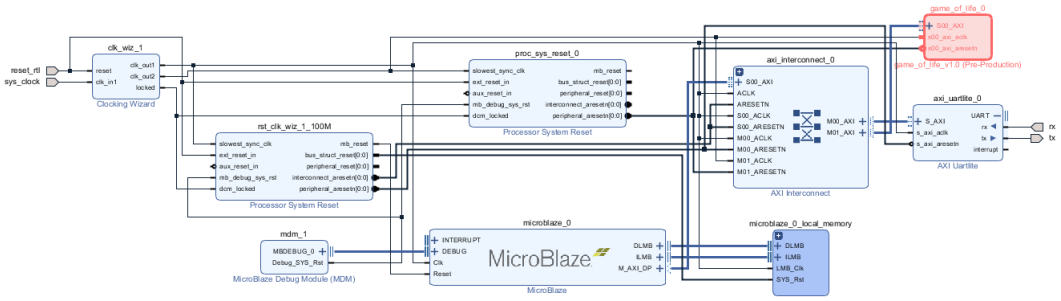


Abbildung 4.15.: Blockdiagram des GOL-Design und den Microblaze-Prozessor in Vivado.

Die Vivado Design Suite generiert automatisch die notwendigen VHDL-Wrapper. Damit ist die FPGA-Seite des GOL Beschleunigers fertiggestellt. Das Hardware-Design wird für die Software-Entwicklung in das von Vivado mitgelieferten Software Development Kit (SDK) exportiert in Form einer HDF-Datei. Dieser enthält eine Beschreibung des gesamten Block Designs, zusätzlich werden in C geschriebene Treiberdateien für das GOL-System erstellt. Für die Erstellung eines Microblaze-Prozessor Testumgebung, wird in das Xilinx SDK eine Bare-Metal-Applikation implementiert, welche dann über einen Joint Test Action Group (JTAG)-Verbindung in das Zedboard programmiert werden.
Die C-basierten Treiberdateien für das GOL-System enthalten Funktionen für:

- die Initialisierung der Startkonfiguration
- das Lesen aller Zell-Zustände
- das Starten und Stoppen
- das Setzen der maximalen Simulationszeit

Als Startkonfiguration wird hier ein 18 x 12 Pattern von dem deutschen Mathematiker Achim Flammenkamp ausgewählt. Es wiederholt sich nach fünf Generationen

und zudem auch symmetrisch wo durch die Verifikation einfacher nachvollziehbar ist.
In der Hauptfunktion ($main$) der Test-Applikation wird zunächst ein eindimensionales Array der Größe 18 x12 = 216 definiert und in den Beschleuniger geladen. Dann wird das GOL-System gestartet und per default nach fünf Generation anhalten. Zur Visualisierung der Zell-Zustände pro Generation, werden die Ergebnisse mittels UART an der Konsole des Host-PC ausgeben.
Zu erwähnen ist, dass bei der Test-Applikation wird zu Anfangs die Platzierung und Verdrahtung dem PAR-Tool überlassen. Nach der Verifikation des Entwurfs, wird ein Tcl-Skript, zur manuellen Platzierung der einzelnen Zell-Modulen auf einen Slice, ausgeführt und das Ergebnis erneut überprüft. Die folgende Abbildung zeigt die Ausgabe pro Generation in der Konsole.

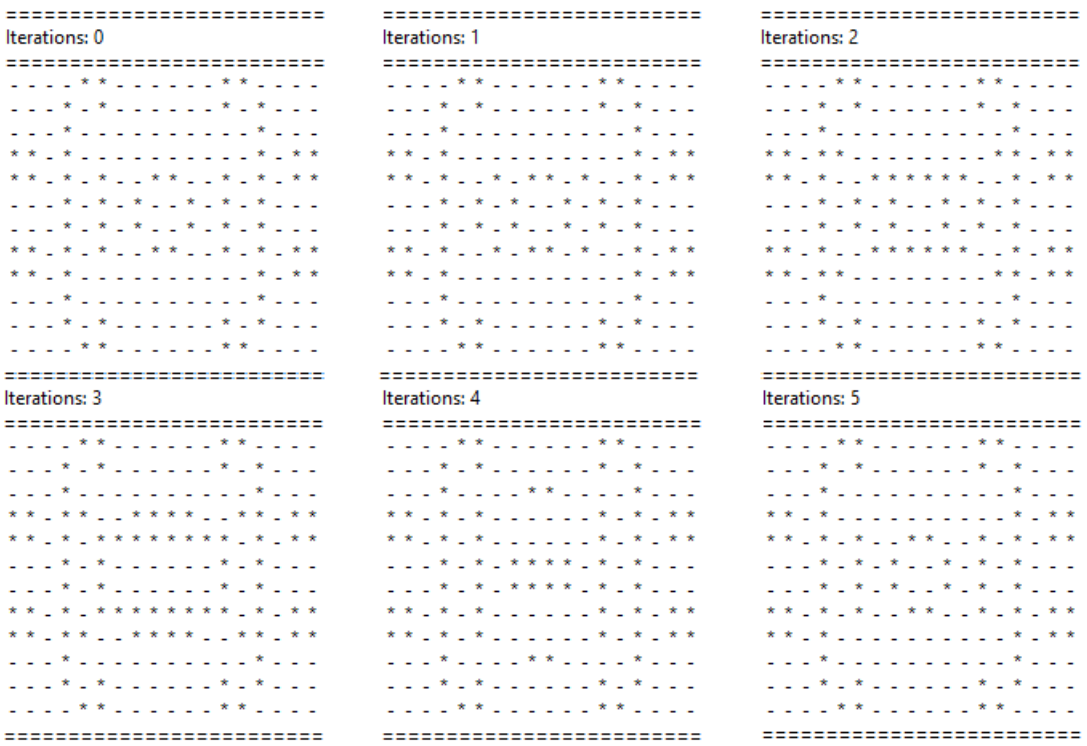


Abbildung 4.16.: UART Konsolen Ausgabe.

Als Vergleich wird das GOL als Software für den Microblaze implementiert und wird ebenfalls in der Konsole ausgegeben. In separaten C-Header und C-Source Dateien werden Funktionen für die Simulation deklariert und definiert. Ein zweidimensionales 8-Bit Array stellt das Spielfeld dar und wird zu Beginn mit der 18x12 Pattern initialisiert. In zwei verschachtelte for-Schleifen wird jedes

Element des zweidimensionalen Arrays auf seine Nachbarschaft überprüft und die Übergangsregel angewendet.

```
for (Xuint16 r = 0; r < HEIGHT; ++r)
{
    for (Xuint16 c = 0; c < WIDTH; ++c)
    {
        Xuint8 crrnt_state = grid[r][c];
        Xuint8 num_of_neighbors = microblaze_count_prox(&r, &c);

        // rules
        if (num_of_neighbors == 3)
            new_grid[r][c] = 1;
        else if ((num_of_neighbors < 2) || (num_of_neighbors > 3))
            new_grid[r][c] = 0;
        else
            new_grid[r][c] = crrnt_state;
    }
}

// copy data from new_grid to grid
for (Xuint16 r = 0; r < HEIGHT; ++r)
    for (Xuint16 c = 0; c < WIDTH; ++c)
        grid[r][c] = new_grid[r][c];
```

Listing 4.6: Codesegment zur Berechnung der nächsten Zustand des gesamten GOL.

Die UART-Ausgabe der Software-Implementation stimmen mit der FPGA-Implementation überein, damit funktioniert der FPGA-Entwurf wie erwartet.

Kapitel 5

Auswertung und Ergebnisse

In diesem Kapitel wird die Ressourcenauslastung für ein möglich großes Spielfeld untersucht. Es werden Spielfelder auf homogener sowie inhomogener FPGA-Chipfläche verteilt und auf das Timingverhalten untersucht. Ebenfalls wird ein Vergleich zwischen den GOL-Entwurf durch Verhaltensbeschreibung und der Strukturbeschreibung vorgenommen. Bei der Strukturbeschreibung werden die automatische Platzierung und die Platzierung durch LOC-Constraints untersucht und ausgewertet.

5.1. Modulplatzierung

Da die verfügbaren FPGA-Familien unterschiedlich aufgebaut sind, ist es naheliegend, dass es Einfluss auf Verteilung der Zellen innerhalb einer FPGA-Chipfläche hat. Die FPGA-Struktur des verwendeten Zedboard wird in Abbildung 5.1 veranschaulich.

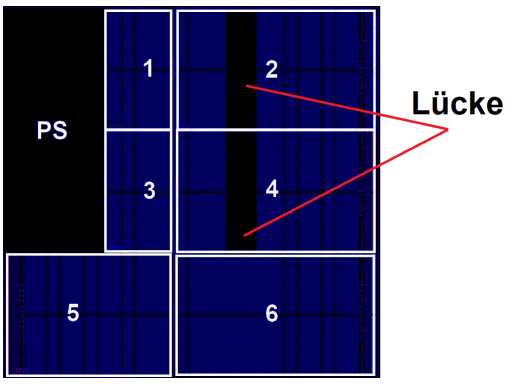


Abbildung 5.1.: Zynq-7000 XC7020-CL484 Chipstruktur unterteilt in sechs Taktregionen.

Die programmierbare Logik des Zynq-7000 XC7020-CL484 Chips sind in sechs Taktregionen unterteilt, dies ermöglicht eine globale Verfügbarkeit des Taktes im gesamten Chip.
Auf dem ersten Blick weisen Region 2 und Region 4 eine Lücke auf, welche zusätzliche Laufzeit auf das GOL verursachen kann. Aus diesen Grund wird ein homogen verteilte CLB-Fläche im Chip ausgesucht, in dem Fall Region 5 und 6, um eine Platzierung der Zell-Modulen so nahe wie möglich beieinander zu steuern. Dieser Prozess wird als *Floorplanning* bezeichnet und können durch rechteckige Vorgaben die Module für die vorhandenen Slices erzeugt werden. Für die Platzierung der Zell-Modulen durch LOC-constraints müssen Vorgaben aus Floorplanning nicht erzeugt werden, da Zell-Modulen durch einen Tcl-Skript an die exakten Positionen der Slices erzwungen werden können.
Ein Spielfeld der Größe 63x49 wird für Messung festgelegt, da sie etwa die Chipfläche der Taktregion 6 entspricht.

5.2. Auswertung des Timingsverhaltens

Um eine Abschätzung des Timingsverhalten machen zu können, wurden alle zwei Entwurfsmethoden mit der oben genannten Größe synthetisiert. Dann werden Floorplaning Vorgaben, basieren auf die Synthese-Netzliste, auf die Taktregionen 6 erzeugt und erneut synthetisiert. Beim Floorplaning wird wird lediglich das Synthese-Tool angeleitet die Module auf eine vorgegebene Chipfläche zu platzieren, daher kann eine wiederholte Synthese mit der Floorplanning scheitern. Aus diesem Grund wird ein Versuch-und-Fehler-Methode angewendet mit mehrere Durchläufe die passende Chipfläche innerhalb der Taktregionen 5 und 6, bei gleichbleibender Taktfrequenz, zu ermitteln.

5.3. Ressourcenauslastung

Für unterschiedliche Spielfeldgrößen

Kapitel 6

Auswertung und Ergebnisse

6.1. Zedboard Auslastung

Vergleiche Entwurf durch Verhaltensbeschreibung, durch Strukturbeschreibung ohne local constraints und mit local constraints.

6.1.1. Timing

WNS... Für unterschiedliche Systemfrequenzen

6.1.2. Resourcenauslastung

Für unterschiedliche Spielfeldgrößen

Abkürzungsverzeichnis

AXI Advanced eXtensible Interface

BEL Basic Element of Logic

CLB configurable logic block

FF Flip Flop

FPGA Field Programmable Gate Arrays

GOL Game of Life

JTAG Joint Test Action Group

LOC Location Constraints

LUT Look Up Table

SDK Software Development Kit

Tcl Tool command Language

PAR Place and Route

PS Processing System

PL programmierbaren Logik

VHDL VHSIC Hardware Description Language

XDC Xilinx Design Constraints

ZA zelluläre Automaten

Tabellenverzeichnis

2.1. Skizze zur Visualisierung der Regeln im GOL. Es sind jeweils drei mögliche Nachbarschaften von x_{ij} zum Zeitpunkt t_k, welche im folgenden Zeitschritt t_{k+1} zu einem Zustand führen. Weiß steht für tot und grau für lebendig. Dabei ist die Referrenzelle x_{ij} immer das mittlere Feld [Sch14]. 8

4.1. Zeilen der Wahrheitstabelle, bei denen der neue Zustand der Zelle den Wert 1 (lebendig) annimmt. 24

Abbildungsverzeichnis

2.1. Beispiel eines zweidimensionalen Gitter: a) rechteckig und b) hexagonales 4
2.2. Beispiel eines zweidimensionalen 4x5 Gitter mit zwei Zuständen. Wobei Q aus s = 2 Zuständen besteht. Weiß q_1 für “tot” und Grau q_2 für "lebendig". 5
2.3. Nachbarschaften für zweidimensionales Gitter: a) Von-Neumann-Nachbarschaft und b) Moore-Nachbarschaft [Sch14]. 5
2.4. Reflektierende Randbedingung am Beispiel Neumann-Nachbarschaft 6
2.5. Periodische Randbedingungen am Beispiel a) Neumann- und b) Moore-Nachbarschaft [Sch14]. 7

3.1. Xilinx ZedBoard 10
3.2. Vereinfaches Modell der Zynq Architektur [Cro+14]. 10
3.3. Grundlegende Architektur eines PL. 11
3.4. Aufbau einer SLICEL [Xil16] 12
3.5. Aufbau einer Carry-Logik [Xil16] 13
3.6. Grundlegende Entwurfsablauf. Begin von links nach rechts 15

4.1. Methode zur Verschiebung der Daten zu den einzelnen Zellen im zweidimensionaler Raum. Links das System zum Zeitpunkt t_k und rechts t_{k+1} 19
4.2. Syntheseergebnis der Verhaltensbeschreibung 21
4.3. Zell-Modul Device Ansicht Verhaltensbeschreibung. 21
4.4. Zell-Logik als Blockdiagramm. 22
4.5. Syntheseergebnis der Strukturbeschreibung 25
4.6. Das vergrößerte Device-Ansicht mit Darstellung der belegten Hardwareressourcen in Cyan. 26
4.7. Das vergrößerte Device-Ansicht nach location constraints zeigt die platzierten Komponenten in Orange und in grün die Verbindungspfade. 26
4.8. Verhaltenssimulation des Zell-Moduls. 27
4.9. Randbedingung des Spielfeldes anhand eines Beispiels. 27

4.10. Grafische Vorstellung der Testbench: a) Die Startkonfiguration in sequenzielle Darstellung und b) Erwartetes Ergebnis nach $t = 3$ Generationen. Wobei Weiß für “tot” und Grau für “lebendig”. 28
4.11. Überblick der Systemarchitektur. 29
4.12. Zustandsdiagramm zur Ansteuerung des GOL-Systems. 30
4.13. Überblick GOL-IP. 31
4.14. Registerstruktur der GOL-IP 31
4.15. Blockdiagram des GOL-Design und den Microblaze-Prozessor in Vivado. 32
4.16. UART Konsolen Ausgabe. 33

5.1. Zynq-7000 XC7020-CL484 Chipstruktur unterteilt in sechs Taktregionen. 35

Listings

3.1. Strukturbeschreibung eines D-FlipFlop 15
3.2. Verhaltensbeschreibung eines D-FlipFlop 16

4.1. Zell entity. 20
4.2. Codesegment zur Funktionsverhalten des Zell-Moduls. 20
4.3. Funktion zur Generierung der INIT-Werte der ersten drei LUT6 Komponenten. 23
4.4. Funktion zur Generierung der INIT-Werte der vierten LUT6 Komponente. 24
4.5. Tcl-Befehlen zur location constraints eines Zell-Moduls. 26
4.6. Codesegment zur Berechnung der nächsten Zustand des gesamten GOL. 34

A.1. Strukturbeschreibung einer Zell-Modul xii

Literatur

[Cro+14] Louise H. Crockett u. a. *The Zynq Book - Embedded Processing Withe ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. Scotland, UK: Strathclyde Academic Media, 2014.

[Gun14] Manfred Selz Gunther Lehmann Bernhard Wunder. *Schaltungsdesign mit VHDL*. 2014. URL: https://www.itiv.kit.edu/downloads/Buch_gesamt.pdf (besucht am 13. 05. 2021).

[LB12] Walter Lange und Martin Bogdan. *Entwurf und Synthese von Eingebetteten Systemen - Ein Lehrbuch*. Berlin: Walter de Gruyter, 2012.

[RS20] Jürgen Reichardt und Bernd Schwarz. *VHDL-Simulation und -Synthese - Entwurf digitaler Schaltungen und Systeme*. Berlin: Walter de Gruyter GmbH Co KG, 2020.

[Sch14] Daniel Scholz. *Pixelspiele - Modellieren und Simulieren mit zellulären Automaten*. Berlin Heidelberg New York: Springer-Verlag, 2014.

[Xil12] Xilinx. *Libraries Guidee*. 2012. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug953-vivado-7series-libraries.pdf (besucht am 13. 05. 2021).

[Xil13] Xilinx. *Constraints Guide*. 2013. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/cgd.pdf (besucht am 13. 05. 2021).

[Xil16] Xilinx. *7 Series FPGAs Configurable Logic Block*. 2016. URL: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf (besucht am 12. 05. 2021).

Anhang A

Erster Anhang

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity cell is
    Port ( prox: in std_logic_vector (7 downto 0); -- proximity (Nachbarschaft)
           ce: in std_logic;
           clk: in std_logic;
           shift: in std_logic;
           Q: out std_logic );  -- 1 stand for "ALIVE" and 0 stand for "DEAD"
end cell;

architecture Behavioral of Cell is
    signal internal_state : std_logic := '0'; -- internal cell state
    signal carry4_out : std_logic_vector(3 downto 0) := "0000";
    signal carry4_din : std_logic_vector(3 downto 0) := "0000";
    signal carry4_s : std_logic_vector(3 downto 0) := "0000";

    signal lut6_bit0_o6 : std_logic;
    signal lut6_bit1_o6 : std_logic;
    signal lut6_bit2_o6 : std_logic;
    signal lut6_2_last_o6: std_logic := '0';
begin
    --carry4_s <= count6bits(prox(7 downto 2)) & lut6_2_last_o6;
    LUT6_BIT2 : LUT6
        generic map (
            INIT => X"fee8e880e8808000") -- Specify LUT Contents
        port map (
            O => lut6_bit2_o6, -- 6/5-LUT output (1-bit)
            I0 => prox(2), -- LUT input (1-bit)
```

```
39          I1 => prox(3), -- LUT input (1-bit)
40          I2 => prox(4), -- LUT input (1-bit)
41          I3 => prox(5), -- LUT input (1-bit)
42          I4 => prox(6), -- LUT input (1-bit)
43          I5 => prox(7)); -- LUT input (1-bit)
44      LUT6_BIT1 : LUT6
45          generic map (
46              INIT => X"8117177e177e7ee8") -- Specify LUT Contents
47          port map (
48              O => lut6_bit1_o6, -- 6/5-LUT output (1-bit)
49              I0 => prox(2), -- LUT input (1-bit)
50              I1 => prox(3), -- LUT input (1-bit)
51              I2 => prox(4), -- LUT input (1-bit)
52              I3 => prox(5), -- LUT input (1-bit)
53              I4 => prox(6), -- LUT input (1-bit)
54              I5 => prox(7)); -- LUT input (1-bit)
55      LUT6_BIT0 : LUT6
56          generic map (
57              INIT => X"6996966996696996") -- Specify LUT Contents
58          port map (
59              O => lut6_bit0_o6, -- 6/5-LUT output (1-bit)
60              I0 => prox(2), -- LUT input (1-bit)
61              I1 => prox(3), -- LUT input (1-bit)
62              I2 => prox(4), -- LUT input (1-bit)
63              I3 => prox(5), -- LUT input (1-bit)
64              I4 => prox(6), -- LUT input (1-bit)
65              I5 => prox(7)); -- LUT input (1-bit)
66
67      carry4_s <= lut6_bit2_o6 & lut6_bit1_o6 & lut6_bit0_o6 & lut6_2_last_o6;
68      carry4_din <= "000" & prox(1);
69      CARRY4_obj: CARRY4
70          port map (
71              CO => open,
72              O => carry4_out,
73              CI => '0',
74              CYINIT => prox(1),
75              DI => carry4_din,
76              S => carry4_s);
77      LUT6_LAST_BIT1 : LUT6
78          generic map (
79              INIT => X"00003e80cccccccc") -- Specify LUT Contents
80          port map (
81              O => lut6_2_last_o6, -- 6/5-LUT output (1-bit)
82              I0 => internal_state, -- LUT input (1-bit)
83              I1 => prox(0), -- LUT input (1-bit)
84              I2 => carry4_out(1), -- LUT input (1-bit)
85              I3 => carry4_out(2), -- LUT input (1-bit)
86              I4 => carry4_out(3), -- LUT input (1-bit)
87              I5 => shift); -- LUT input (1-bit)
88      FDRE_I1: FDRE
89          generic map (INIT => to_bit('0'))
90          port map (
91            Q  => internal_state,-- [out std_logic]
```

```
92          C  => clk,          -- [in std_logic]
93          CE => ce,           -- [in std_logic]
94          D  => lut6_2_last_o6,-- [in std_logic]
95          R  => '0');         -- [in std_logic]
96
97     Q <= internal_state;
98
99  end Behavioral;
```

Listing A.1: Strukturbeschreibung einer Zell-Modul

Anhang B

Zweiter Anhang

Hier noch ein Beispiel für einen Anhang.