FPGAs
Einstieg, Schaltungen, Projekte

Kapitel 1
Los geht's: FPGAs für Maker und Kreative

Field Programmable Gate Arrays (FPGAs) sind spezielle integrierte Schaltungen, die sich durch den Benutzer programmieren lassen. Schauen wir uns zunächst die Tools und Konzepte zu diesen Bausteinen an und welche Rolle Open-Source-Software dabei spielt.

In der dynamischen Welt der Elektronik stellen FPGAs eine Technologie dar, die es Enthusiasten ermöglicht, die Schranken konventioneller Mikrocontroller-Systeme zu überwinden. Sie werden feststellen, dass Sie mit FPGAs viel machen können, was beispielsweise ein Arduino oder gar ein Mini-Computer wie der Raspberry Pi nicht kann. Mit diesem Buch möchte ich Ihnen einen umfassenden Einstieg in diese Welt bieten. Es geht nicht nur um das theoretische Wissen, sondern auch um praktische Fähigkeiten, die Sie direkt in Ihre Projekte integrieren können.

1.1 Über dieses Buch: Was Sie lernen werden und was Sie schon können sollten

Als Maker erreichen Sie irgendwann einen Punkt, an dem herkömmliche Mikrocontroller oder Mikroprozessoren für manche Anforderungen nicht mehr genügen. Oder vielleicht wollen Sie einfach mal etwas Neues ausprobieren. Der Arduino und der Raspberry Pi sind zwar weit verbreitet und werden durch eine Vielzahl von Soft- und Hardware-Beispielen gut unterstützt. Auch ein ESP32 oder ein Raspberry Pi Pico bieten jeweils spezielle Features, mit denen sich bestimmte Problemstellungen gut lösen lassen. Doch manchmal werden Sie sich mehr wünschen: mehr Flexibilität, höhere und echte Parallelverarbeitungsleistung, anspruchsvolle visuelle Effekte, schnelle Signalverarbeitung oder Kryptografie – und das alles flexibel und schnell anpassbar.

Wenn Sie aus diesen Gründen dieses Buch zur Hand genommen haben, sind Sie bereit, die Welt der *Field Programmable Gate Arrays* – oder kurz einfach FPGAs – zu

entdecken. Hier lernen Sie die Grundlagen und Anwendungsmöglichkeiten dieser programmierbaren Komponenten kennen.

1.1.1 Für wen ist dieses Buch geeignet?

Sie sollten ein fortgeschrittener Maker bzw. Elektronik-Bastler sein und die eine oder andere Skriptsprache beherrschen, am besten Python, denn Python wird auch im Buch verwendet. Sie sollten mit Linux umgehen können, am besten auf einem Raspberry Pi. Auf der Linux-Konsole brauchen Sie kein Experte zu sein; wenn Sie sich grundsätzlich auf der Kommandozeile zurechtfinden, reicht das aus.

Ihr Wissen im Bereich der Elektronik sollte all das abdecken, was man als Bastler so macht: Mit Mikrocontrollern sollten Sie mal »was gemacht« haben: Vielleicht haben Sie auch schon das eine oder andere Breakout Board verwendet, das Sie via SPI oder I^2C an Ihren Arduino oder Raspberry Pi angeschlossen haben. Sie sollten grundsätzlich wissen, wie die Kommunikation zwischen Teilnehmern auf einem seriellen Bus funktioniert, d. h., Sie sollten so ungefähr wissen, was eine Adresse oder eine Chip-Select-Leitung im Zusammenhang mit einem seriellen Bus ist.

Wenn Sie nun feststellen, dass Sie vom größten Teil des eben Gesagten keine Ahnung haben, dann könnte es sein, dass dieses Buch an einigen Stellen zu fortgeschritten für Sie ist.

1.1.2 Was dieses Buch nicht ist

In diesem Buch finden Sie keine endlosen Grundlagen zu boolescher Algebra und Logikgattern. Sie werden auch nicht erst theoretisch den Inhalt der vielen Hundert Seiten des Verilog-Standards lesen und dabei Datentypen, Operatoren, Kontrollstrukturen und Systemfunktionen erlernen müssen. Dies alles behandele ich in diesem Buch nur so weit, wie es unbedingt nötig ist. Ich wollte kein klassisches Fachbuch schreiben, das Ihnen bei einer Semester-Abschlussarbeit der E-Technik oder bei der Vorbereitung auf ein Vorstellungsgespräch als Entwickler von Embedded Systems hilft. Sein Inhalt ist für die Werkstatt und das Mindset eines Makers und Elektronik-Bastlers gedacht.

1.1.3 Was ist der Inhalt dieses Buches?

Im ersten Teil des Buches erfahren Sie die Antwort auf die Frage: »Was ist ein FPGA und was kann er?« Wir besprechen die beiden FPGA-Boards, die in diesem Buch verwendet werden und woher Sie diese beziehen können. Danach stelle ich die Bestandteile der in diesem Buch genutzten Open-Source-Toolchain vor. Auch die

beste Vorgehensweise beim Einrichten einer Entwicklungsumgebung, das Backup Ihrer Projektdateien und die verwendeten Tools zur Versionskontrolle und des Build-Prozesses sind Themen in Teil I.

Im zweiten Teil des Buches besprechen wir die Möglichkeiten von FPGAs und realisieren praktische Beispielprojekte. Dafür nutzen wir ausschließlich die Open-Source-Toolchain und zwei FPGA-Boards, den *Tang Nano 9K* und den *IceZero*. Sie werden dafür Ihre eigene FPGA-Entwicklungsumgebung inklusive aller dazugehörigen Tools und Skripte einrichten. Somit besitzen Sie eine vollständige Unabhängigkeit bei Ihrer Entwicklungsarbeit. Das bedeutet, wir werden keine kommerzielle Software und keine IDE/Entwicklungsumgebung eines Herstellers nutzen. Es wird somit keine Lizenz nötig sein – an keiner Stelle entsteht so eine Abhängigkeit, und es fallen auch keine weiteren Kosten an!

Bleiben Sie neugierig

Übrigens: Diese Toolchain und die genutzten FPGAs sind nur ein Beispiel. Sie können auch andere FPGAs nutzen, indem Sie die Toolchain anpassen. Der Verilog-Code ist so generisch wie möglich gehalten, damit er auf einer Vielzahl von FPGAs läuft. Der Aufwand für eine Anpassung ist somit gering.

Nur: Sie müssen dann eben die Toolchain ändern und auch die Beispiele in diesem Buch an Ihren FPGA anpassen. Trauen Sie sich das zu? In Abschnitt 2.5 gehe ich näher darauf ein.

In diesem Buch finden Sie eine Vielzahl von Beispielprojekten, die viele der grundlegenden Technologien abdecken, die in der Praxis häufig benötigt werden. Dazu gehören Displays, Taktgeber, Taster, Sensoren, LEDs und verschiedene serielle Protokolle. Diese grundlegenden Elemente bilden oft das Fundament, auf dem komplexere und interessantere Anwendungen aufbauen können. So können Sie mit diesen Beispielprojekten ein solides Verständnis für die Grundlagen entwickeln, den Inhalt aber später auch bei Ihren eigenen Ideen weiterverwenden. Dadurch erhalten Sie einen praxisnahen Einblick in die reale Arbeit mit diesen Bausteinen.

1.1.4 Das Lernkonzept dieses Buches

Das Konzept, wie in diesem Buch Wissen vermittelt wird, unterscheidet sich vom theoretischen Lehransatz an Hochschulen:

- **Learning by doing!** – Sie programmieren einen FPGA mit der von Ihnen selbst eingerichteten Entwicklungsumgebung. Wir steigen schnell und direkt in die Programmierung von FPGAs ein. Wenn der FPGA dann läuft, schauen wir uns den Quellcode an. Das ist ideal für alle, die direkt loslegen möchten.

- **Wir nutzen die KI!** – In diesem Buch werde ich Ihnen nicht alle Fragen beantworten und nicht jedes Detail erklären. Ich zeige Ihnen, wie Sie Ihren FPGA einsetzen können und wie Sie Designs umgesetzt bekommen. Die Grundlagenprojekte und die kniffligeren weiterführenden Projekte werden Ihnen zeigen, wie Sie Displays einrichten, Schnittstellenprotokolle implementieren, Block-SRAM verwenden, Zufallszahlen erzeugen, Neopixel-LEDs zum Leuchten bringen und noch viel mehr.

  Aber ich werde an mancher Stelle auch etwas sagen wie: *»Wenn Sie mehr darüber wissen wollen, wie ein Schieberegister als Verzögerungsleitung dienen kann, dann fragen Sie die Chat-KI nach …«*

  In der heutigen Zeit können wir modernere Methoden nutzen, um uns schnell und effektiv das Wissen anzueignen, das wir brauchen, um eine Aufgabe zu bewältigen. Der alte Hochschulprofessor, der uns erst jahrelang Herleitungen an die Tafel malt, ist Vergangenheit. Die Fachbücher der 1990er-Jahre sind Vergangenheit. In diesem Buch nutzen wir die Chat-KI ausgiebig, damit Sie so schnell und einfach wie möglich alles zu FPGAs lernen, was nötig ist. Damit Sie hinterher fit in dem Thema sind.

Sind Sie so weit einverstanden? Ist das ein Ansatz, mit dem Sie leben können? Dann lassen Sie uns gemeinsam in die faszinierende Welt der FPGAs eintauchen und entdecken, wie sie Ihre kreativen und technischen Fähigkeiten erweitern können.

I – Los geht's mit Ihrem Traumprojekt

In diesem Buch werden Ihnen öfter Kästen wie dieser begegnen, in denen es um Ihr FPGA-Traumprojekt geht. In solchen Kästen werde ich mit Ihnen in kleinen Schritten – abgestimmt mit dem Inhalt des Buches – Ihr erstes FPGA-Design aufbauen, das Sie dann nach diesem Buch starten können. Wir bilden so Schritt für Schritt die Basis für Ihren ersten selbstständigen Einstieg in der FPGA-Hardware-Entwicklung.

Alles, was Sie dafür brauchen, sind ein paar Blatt Papier und ein Stift. Notieren Sie einfach Ihre liebste Idee bzw. liebsten Ideen. Im Verlauf dieses Buches werden wir an ihnen weiterarbeiten, und zum Schluss helfe ich Ihnen, dieses Projekt realisiert zu bekommen. Wichtig ist, dass Sie diese Notizen tatsächlich irgendwo niederschreiben, denn wir werden diese Sätze, Wörter und Skizzen wirklich verwenden.

1.2 Ihr Weg durch dieses Buch

Der Autor jedes Buches, jeder Anleitung und jedes Tutorials muss einiges Grundwissen bei seinen Leserinnen und Lesern voraussetzen. Das ist bei diesem Einstieg

in die Arbeit mit FPGAs nicht anders. Im ersten Teil des Buches finden Sie daher einen Überblick über einige Themen, die ich für das Verständnis des Themas und für eine gute Entwicklungsarbeit für wichtig halte. Wenn Sie noch nie mit digitaler Elektronik gearbeitet haben und noch nicht programmiert haben, sollten Sie diese Grundlagen nicht überspringen.

- Für den Einstieg schauen wir uns in Kapitel 1 des Buches an, was unter einem FPGA zu verstehen ist und was in solch einem Bauteil steckt. Sie erfahren, wo ein FPGA eingesetzt werden kann und wofür er sich gut eignet.
- In Kapitel 2 nähern wir uns dem FPGA mehr im Detail und besprechen die beiden FPGA-Modelle, die wir in diesem Buch verwenden: den IceZero mit dem *iCE40HX*-FPGA von Lattice und den Tang Nano 9K mit *GW1NR-9C*-FPGA von Gowin.
- In Kapitel 3 geht es um die Bedeutung der Toolchain. Sie erfahren, wie Sie eine Open-Source-Toolchain nutzen können, um unabhängig von der Herstellersoftware zu sein und auf diese Weise auch mehr Freiheit beim Hardware-Design zu erlangen. Wir besprechen auch, wie Sie die beiden FPGAs an das Entwicklungssystem anschließen.
- In Kapitel 4 besprechen wir die Möglichkeiten einer generativen KI zur unterstützenden Programmierung. Dort gehe ich auf die praktikabelsten Möglichkeiten der Nutzung einer KI zum Programmieren und Erlernen von Verilog ein.
- In Kapitel 5 geht es um das Entwicklungssystem auf einem Raspberry Pi. Wir besprechen die Einrichtung des Kleinrechners für das Entwickeln von FPGA-Designs, die Versionsverwaltung und den Einsatz von Makefiles.
- In Kapitel 6 richten wir die Toolchain mit dafür entwickelten Installationsskripten auf dem Raspberry Pi ein, und zwar für beide FPGAs. Wir prüfen das Setup jeweils mit einem kleinen Testprogramm. Außerdem besprechen wir die möglichen Editoren und deren hilfreiche Plug-ins zum Schreiben von Verilog-Quellcode.
- In Kapitel 7 widmen wir uns den grundsätzlichen Bestandteilen der Sprache *Verilog*. Sie erfahren, wie hilfreich die automatische Codegenerierung aus einer anderen Programmiersprache heraus ist.
- In Kapitel 8 geht es um den Ablauf eines FPGA-Designs: Wie sind die Entwicklungsschritte, was ist eine Projektplanung, wann erfolgt eine Simulation und was ist eine Testbench?
- In Kapitel 9 steigen wir in die Praxis ein und führen Grundlagenprojekte mit dem IceZero-Board durch. Dabei nutzen wir den iCE40HX für Aufgaben wie Taktgenerierung, das Blinkenlassen von LEDs, und wir implementieren auch schon eine UART-Schnittstelle zur seriellen Kommunikation.

- In Kapitel 10 führen wir komplexere Projekte mit dem Tang Nano 9K durch. Vieles dreht sich dabei um Displays und die Darstellung von Bildern und die Anzeige eines Bargraphen darauf. Aber wir werden auch eine Text-Engine implementieren, um dynamisch Text darstellen zu können. Das Display wird dann in anderen Projekten weiterverwendet, beispielsweise zur Darstellung von Messwerten eines Analog-Digital-Wandlers.
- In Kapitel 11 folgen praktische Tipps zu Spezialthemen beim FPGA-Design, damit Ihnen die Arbeit einfacher von der Hand geht und Probleme vermieden werden, bevor sie auftreten.
- In Kapitel 12 geht es um die Community. Open-Source lebt durch den Austausch der teilnehmenden Menschen. Wir besprechen, wie Sie Teil dieser Gemeinschaft werden und wo Sie Ihre eigenen FPGA-Designs vorstellen können.
- In Kapitel 13 besprechen wir, wie Sie nach diesem Buch Ihr erstes eigenes FPGA-Design beginnen können.

Das heißt aber nicht, dass Sie alle Kapitel linear lesen müssen! Die allermeisten Themen können »nach Bedarf« gelesen werden. Und wenn Sie die Theorie und die Grundlagen schon kennen, dann können Sie auch direkt mit Kapitel 7 oder sogar Kapitel 10 loslegen.

1.3 Was sind FPGAs?

Als Maker werden Sie bereits Erfahrungen mit Arduino und Raspberry Pi gesammelt haben und sind mit Mikrocontrollern und anderen kleinen vielseitigen Computern vertraut. Möglicherweise ist es für Sie nun interessant, eine weitere faszinierende Komponente der Elektronik kennenzulernen: den FPGA oder – mit vollem Namen – den *Field Programmable Gate Array*.

Dieser Name ist sehr sprechend: Es handelt sich um eine Aneinanderreihung von Logik-Gates, die »im Feld« (also bei Ihnen als Endkunden) programmiert werden können (siehe Abbildung 1.1).

Dieses Bauteil ist eine Art Chamäleon in der Welt der Elektronik: Es passt sich Ihren Bedürfnissen an, statt Sie auf vorgegebene Funktionen zu beschränken.

Um eine Vorstellung von der Funktionsweise eines FPGAs zu bekommen, können Sie diesen mit einem Orchester vergleichen. In einem Orchester wirken verschiedene Instrumente zusammen, um ein harmonisches Klangbild zu erzeugen. Jedes Instrument, ähnlich einem Logikblock im Inneren eines FPGAs, spielt eine spezifische Rolle. Zusammen bilden sie eine komplexe und leistungsfähige Symphonie. Genau wie ein Orchester für jede Symphonie spezifisch zusammengestellt wird,

können Sie das Innere eines FPGAs spezifisch verschalten. In den FPGAs können Sie die »Instrumente« – d. h. die Logikblöcke – so konfigurieren, dass sie genau die Funktionen ausführen, die Sie für Ihr Projekt benötigen. Ein FPGA wird so für seine spezielle Aufgabe konfiguriert.

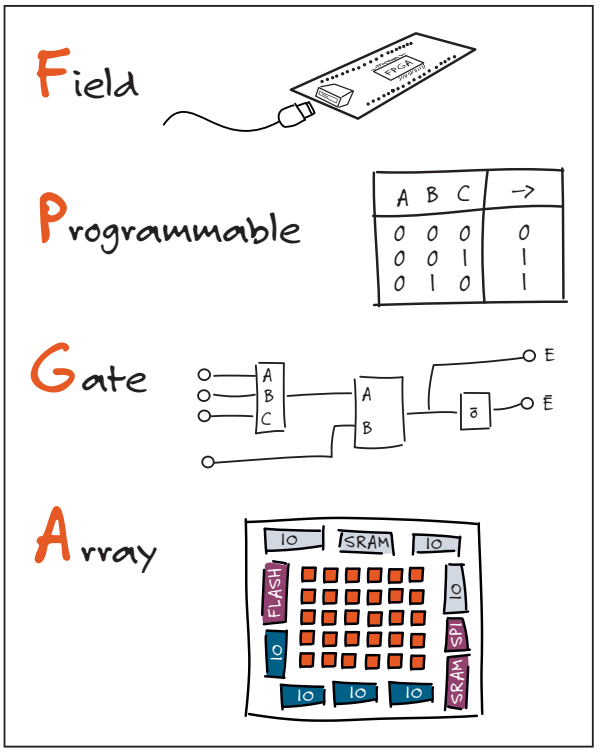

Abbildung 1.1 Welche Eigenschaften machen einen FPGA aus?

FPGAs finden heute in vielen Bereichen Anwendung, sie sind aber besonders dort wichtig, wo maßgeschneiderte Lösungen gefragt sind. Im kreativen und künstlerischen Bereich werden sie beispielsweise für anspruchsvolle Lichtsteuerungen, für die Audioverarbeitung oder sogar in der digitalen Kunst eingesetzt, um einzigartige visuelle Effekte zu erzeugen. Sie sind die Spezialisten im Hintergrund, die komplexe, individuelle Anforderungen umsetzen können, wo anderer Hardware die Puste ausgeht.

Ein FPGA besteht aus einer Vielzahl von sogenannten Logikblöcken, die Sie so programmieren können, dass sie bestimmte Aufgaben erledigen. Diese Logikblöcke ähneln den Bausteinen eines modularen Baukastensystems. Sie können sie in nahezu unendlichen Kombinationen zusammenfügen, um Ihre gewünschte Schaltung zu bauen. Im Vergleich zu einem Mikrocontroller, der eine feste, vorgefertigte Schaltungslogik in sich trägt, die mittels Software im Rahmen der gegebenen

Möglichkeiten genutzt werden kann, bietet der FPGA viel weiter reichende Möglichkeiten, wie Sie Ihre eigene, individuelle Schaltungslogik gestalten können.

Ein weiterer Unterschied zu Mikrocontrollern ist die interne Arbeitsweise. Während ein Mikrocontroller Aufgaben nacheinander abarbeitet (sequenziell), können in einem FPGA mehrere Prozesse gleichzeitig ablaufen (parallel).

Eine passende Analogie hierfür wäre die Rolle eines Dirigenten, der ein Orchester leitet: In einem Orchester spielen verschiedene Instrumentengruppen gleichzeitig, um eine harmonische Symphonie zu erzeugen. Jedes Instrument trägt gleichzeitig, aber mit unterschiedlichen Rollen zum Gesamtklang bei. Bei einem Mikrocontroller spielen die Musiker in einem Orchester immer hintereinander. Nur wenn ein Mikrocontroller oder Mikroprozessor mehrere Kerne hat, dann können auch mal ein paar Musiker parallel spielen, im Großen und Ganzen werden die einzelnen Parts aber nacheinander vorgetragen.

Dies ist bei einem FPGA anders. Sein interner Aufbau ermöglicht es, verschiedene Funktionen parallel auszuführen, wodurch komplexe Aufgaben effizienter und synchronisiert abgewickelt werden können. Ein FPGA kann Ihnen daher das gewisse Etwas geben, bei dem die Summe mehr ist als ihre Einzelteile – wie bei einem Orchester.

Es war einmal …

Die erste FPGA wurde von der Firma Xilinx hergestellt. Der XC2064 kam 1984 auf den Markt und verfügte über 64 konfigurierbare Logikblöcke.

Zwei Jahre später kam der XC2018 auf den Markt, ein verbessertes Modell mit 100 konfigurierbaren Logikblöcken. Die XC2000-Serie stellte die ersten kommerziell erfolgreichen FPGAs dar und legte den Grundstein für die heutige, milliardenschwere FPGA-Industrie.

Abbildung 1.2 XC2018 – der zweite FPGA von Xilinx
(Bild von Wikimedia: File:Xilinx XC2018 LCA.jpg - Wikimedia Commons)

Wegen Ihrer einzigartigen Flexibilität und Leistungsfähigkeit sind FPGAs eine ideale Wahl für kreative Projekte. Ihre hohe Anpassungsfähigkeit ermöglicht es, spezifische Hardware-Funktionen für jedes Projekt zu designen, beispielsweise für maßgeschneiderte Audioverarbeitungseinheiten in digitalen Musikinstrumenten oder Synthesizern.

Dank ihrer Fähigkeit zur parallelen Verarbeitung eignen sich FPGAs hervorragend für Echtzeitanwendungen wie simultane Signalbearbeitungen und Datenverarbeitung mit geringer Latenz. Ihre hohen Verarbeitungsgeschwindigkeiten machen sie zudem ideal für anspruchsvolle Aufgaben, wie sie bei der komplexen Bildverarbeitung anfallen – beispielsweise bei der Highspeed- oder Zeitlupenfotografie oder bei der Objekterkennung.

Zudem sind FPGAs perfekt für das Prototyping und Experimentieren geeignet, weil sie schnelles Testen verschiedener Hardware-Konfigurationen ohne neuen Hardware-Aufbau ermöglichen – beispielsweise in der Robotik.

Darüber hinaus bieten sie sich für interaktive Kunstinstallationen an, da sie auf Sensordaten schnell reagieren können – beispielsweise bei dynamischen Lichtinstallationen, die auf Bewegungen und Geräusche der Betrachter ansprechen.

Diese Vielseitigkeit macht FPGAs zu leistungsstarken Werkzeugen für innovative und kreative Projekte.

Eine Herausforderung bei der Verwendung von FPGAs ist jedoch, dass sie eine andere Art der Programmierung erfordern. Statt der bei Mikrocontrollern eingesetzten Programmiersprachen wie C/C++ oder Python nutzt man bei FPGAs Beschreibungssprachen wie *VHDL* oder *Verilog*.

Um die Metapher des Orchesters wieder aufzugreifen: Die Beschreibungssprache ist der Dirigent des Orchesters. Mit solch einer Sprache definieren Sie, wie die Schaltungen im Inneren des FPGAs aufgebaut sein sollen. Diese Art der Programmierung ist anfangs gewöhnungsbedürftig. Aber keine Sorge, mit etwas Übung werden Sie schnell den Dreh raushaben. Die Beispielprojekte in diesem Buch werden Sie nach und nach in die Lage versetzen, FPGAs nach Ihren Vorstellungen zu nutzen.

1.4 Sprachen, Tools und Konzepte

Für alle Soft- und Hardware-Projekte gilt, dass man sich über die verwendeten Sprachen, Tools und Methodiken Gedanken machen muss. So auch hier: Was brauchen wir, damit die Arbeit Spaß macht und erfolgreich ist?

III – Werden Sie kreativ

Welchen Namen könnte Ihr Traumprojekt bekommen? Was macht es besonders? Gibt es ein Wortspiel, das Interesse an ihm wecken kann? Später möchten Sie ja vielleicht andere Maker zum Mitmachen animieren – ein guter Name hilft da sehr!

Schreiben Sie ein paar Sätze auf, was Ihr Traumprojekt besser können soll als Ihre vorherigen Projekte. Sind Sie schon einmal bei einem Versuch zu Ihrem Traumprojekt gescheitert? Was waren die Gründe? War es noch in der Denkphase oder schon in der Realisierungsphase? Also, was soll Ihr neues Projekt besser können?

Soll es tragbar sein? Mit einer Batterie 10 Stunden lang laufen können? Wenn es eine Lichtinstallation ist, soll sie bei Nacht auch von der ISS aus zu sehen sein?

Was ist Ihr Traum? Was soll Ihr Projekt für großartige Eigenschaften haben, die Sie und andere zum Staunen bringen können? Seien Sie nicht nur ernst und rational bei der Konzeption Ihres Traumprojekts. Haben Sie Spaß dabei, schreiben Sie auch verrückte Ideen dazu auf.

2.1.6 Die Grundkomponenten aller FPGAs

Folgende Bauteile sind in allen FPGAs vorhanden, spezifischere Bestandteile beschreibe ich in Abschnitt 2.1.7.

Lookup Tables (LUTs)

Fast jeder FPGA enthält *LUT*s. Die *Lookup Tables* sind die grundlegenden Bausteine für die digitale Logik in einem FPGA. Sie werden verwendet, um logische Funktionen wie AND, OR, XOR usw. zu realisieren.

Eine Lookup Table ist im Wesentlichen eine kleine, vorprogrammierbare Speichertabelle, die für jede mögliche binäre Eingabekombination einen bestimmten binären Ausgabewert speichert. Liegt eine Kombination an den Eingängen an, die einem gespeicherten Ausgabewert entspricht, dann wird das dazugehörige gespeicherte Ergebnis ausgegeben. Abbildung 2.4 zeigt die Funktionsweise einer LUT mit vier Eingängen.

Sind FPGAs ohne Lookup-Tables möglich?

Traditionell nutzen FPGAs LUTs für die Implementierung von Logikfunktionen, um beliebige boolesche Funktionen zu realisieren. Es gibt aber auch FPGAs, die auf eine sogenannte *Antifuse*- oder *Gate-Array*-basierte Technologie aufbauen. Diese Arten von FPGAs verwenden eine andere Methode zur Implementierung der Logik, indem sie Verbindungen durchbrennen (*Antifuse*), was eine permanente, unverän-

derliche Verschaltung schafft, oder indem sie vordefinierte Transistoren verbinden (*Gate-Array*). Diese FPGA-Arten sind nicht so flexibel wie LUT-basierte FPGAs, was die Programmierbarkeit und Anpassungsfähigkeit betrifft, können aber in bestimmten Anwendungen, bei denen Zuverlässigkeit und Sicherheit im Vordergrund stehen, geeigneter sein.

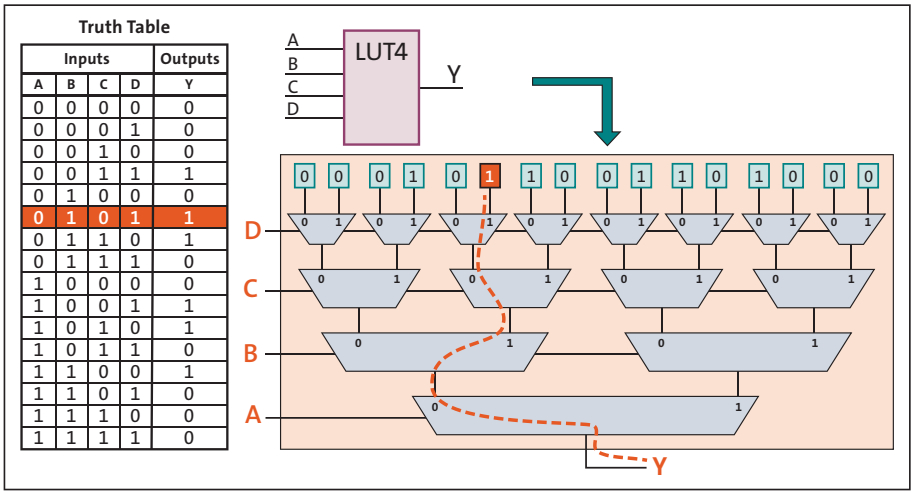

Abbildung 2.4 Funktionsweise einer LUT mit vier Eingängen

Typischerweise haben LUTs eine feste Anzahl von Eingängen, mit denen sie binäre Werte (0 oder 1) entgegennehmen können. Die Anzahl der Eingänge definiert, wie viele verschiedene Eingabekombinationen möglich sind. Beispielsweise hat eine LUT mit 3 Eingängen 8 (2^3) mögliche Eingabekombinationen, und eine LUT mit 2 Eingängen hat 4 (2^2) Eingabekombinationen.

Für jede Eingabekombination gibt es eine Speicherzelle in der LUT, die den entsprechenden Ausgabewert speichert. Der Inhalt dieser Zellen wird während der Konfiguration des FPGAs festgelegt.

Eingang A	Eingang B	Ausgang (A XOR B)
0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 2.2 Ein Beispiel mit einer LUT mit 2 Eingängen als XOR-Gatter

Im Beispiel in Tabelle 2.2 hat die LUT 2 Eingänge (A und B) und 4 mögliche Eingabekombinationen (00, 01, 10, 11). Für jede dieser Kombinationen speichert die LUT den entsprechenden Ausgabewert als XOR-Funktion:

- Wenn beide Eingänge 0 sind, ist der Ausgang 0.
- Wenn Eingang A 0 und Eingang B 1 ist, ist der Ausgang 1.
- Wenn Eingang A 1 und Eingang B 0 ist, ist der Ausgang 1.
- Wenn beide Eingänge 1 sind, ist der Ausgang 0.

Jede Zeile der Wahrheitstabelle entspricht einer Speicherzelle in der LUT. Diese Zellen werden während der Programmierung des FPGAs mit den entsprechenden Ausgabewerten der XOR-Funktion programmiert. Wenn der FPGA läuft und binäre Signale an die Eingänge der LUT gesendet werden, liest diese den dazugehörigen Ausgabewert aus der entsprechenden Speicherzelle aus und gibt ihn als Ausgangssignal aus.

Durch dieses einfache, aber effektive Funktionsprinzip können LUTs eine Vielzahl von logischen Funktionen realisieren, von einfachen AND- oder OR-Gattern bis hin zu komplexeren Funktionen.

In vielen modernen FPGAs können auch mehrere LUTs miteinander kombiniert werden, um noch komplexere Funktionen oder größere Lookup-Tabellen zu ermöglichen. Einige FPGAs bieten auch die Möglichkeit, nur die Ausgänge mehrerer LUTs zu kombinieren, um vielseitigere logische Operationen zu ermöglichen. Aber das nur als Info, so weit gehen wir in diesem Buch nicht.

Programmierbare Logikblöcke (PLBs)

Diese Blöcke enthalten die LUTs und sind die fundamentalen Bausteine für die Implementierung von benutzerdefinierter Logik. Sie ermöglichen es, komplexe digitale Schaltungen zu realisieren. Dafür beinhalten sie neben den LUTs noch Flip-Flops, Multiplexer und möglicherweise auch noch fest definierte Logik-Gatter wie `UND`, `ODER` bzw. `NICHT`. Des Weiteren sind eine Steuerungslogik für das Timing und Koordinieren all dieser Bausteine erforderlich sowie eine *Routing-Matrix*, die dies alles nach Wunsch verschalten kann.

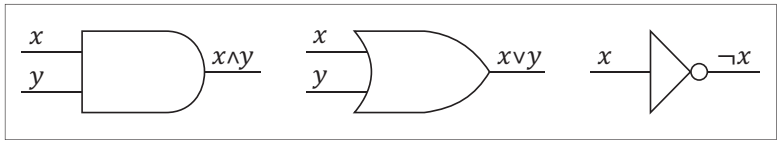

Abbildung 2.5 Und, Oder, Nicht – wer kennt das nicht?

Hört sich kompliziert an? Das ist es auch. Aber hier ist es erst einmal nur notwendig, dass Sie die Begriffe einmal gehört haben; mit diesem Level der Programmierung von FPGAs werden wir uns in diesem Buch nicht auseinandersetzen.

I/O-Blöcke (Input/Output)

I/O-Blöcke ermöglichen die Kommunikation des FPGAs mit der Außenwelt. Sie sind ein wichtiger Bestandteil von FPGAs, denn sie ermöglichen überhaupt erst die Interaktion des FPGAs mit seiner Umgebung. Diese Blöcke sind dafür zuständig, Signale zu senden (*Output*) und zu empfangen (*Input*), wobei jeder I/O-Block in sich konfigurierbar ist. Das bedeutet, dass Sie spezifische Einstellungen vornehmen können, um die Art und Weise anzupassen, wie diese Blöcke Signale verarbeiten. Zu diesen Einstellungen gehören beispielsweise die Anpassung der Signalstärke, der verwendete Logik-Level, die Takt-Geschwindigkeit und der Betriebsmodus (digital oder analog).

I/O-Blöcke können mit einer Vielzahl von elektronischen Standards und Schnittstellen arbeiten. Sie können für SPI (*Serial Peripheral Interface*), I^2C (*Inter-Integrated Circuit*), UART (*Universal Asynchronous Receiver/Transmitter*) und viele andere gängige Kommunikationsprotokolle konfiguriert werden.

Zusätzlich unterstützen sie oft auch fortgeschrittene Funktionen wie Schmitt-Trigger-Eingänge für verbesserte Rauschimmunität und programmierbare Pull-up/Pull-down-Widerstände zur Signalintegritätsoptimierung. Diese Funktionen sind besonders in Umgebungen wichtig, in denen eine stabile und zuverlässige Kommunikation entscheidend ist.

Programmierbare Routing-Matrix

Die programmierbare Routing-Matrix ist ein zentrales Element in einem FPGA und bildet das Herzstück seiner Flexibilität und Anpassungsfähigkeit. Diese Matrix ist eine komplexe Netzwerkstruktur, die die Verbindungen zwischen den verschiedenen programmierbaren Logikblöcken (PLBs) herstellen kann. Das Diagramm in Abbildung 2.6 vermittelt einen ersten Eindruck.

Im Detail besteht die Routing-Matrix aus einer Vielzahl von programmierbaren Schaltern und Verbindungsleitungen, die es ermöglichen, fast jeden Ausgang eines PLBs mit fast jedem Eingang eines anderen PLBs zu verbinden. Diese Schalter und Leitungen werden durch die Programmierung des FPGAs aktiviert oder deaktiviert. Dadurch wird eine nahezu unbegrenzte Anzahl von Verbindungsmöglichkeiten geschaffen.

Eine der wichtigsten Eigenschaften der Matrix ist ihre hohe Dichte und Effizienz. Trotz des komplexen Netzwerks an Verbindungen ist sie so gestaltet, dass sie eine schnelle Signalübertragung mit minimaler Verzögerung ermöglicht. Dies ist wich-

tig für Anwendungen, die eine hohe Leistung und schnelle Datenverarbeitung erfordern.

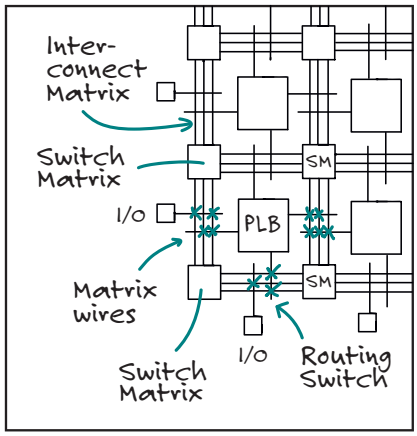

Abbildung 2.6 In der Matrix eines FPGAs sind Verknüpfungen frei definierbar.

Damit dies auch bei einer hohen Auslastung eines FPGAs funktioniert – denn in einem FPGA ist alles auf engstem Raum nebeneinander gepackt –, beinhaltet die Routing-Matrix auch Technologien *wie Clock-Management* und *Signalintegritätsoptimierung*, was auch eine programmierbare *Taktverteilung* und *Taktunterdrückung* beinhaltet. Beides ist für die Entwicklung von Hochgeschwindigkeitsanwendungen und für die Minimierung von *Störungen und Rauschen* notwendig.

Kurz gesagt: Vieles von dem, was moderne FPGAs können, können sie nur, weil die Routing-Matrix einige Tricks im Ärmel hat, um alles auch im Verbund noch funktionieren zu lassen.

Konfigurationsspeicher

Der Konfigurationsspeicher ist das Gedächtnis des FPGAs. Er enthält die spezifischen Konfigurationsdaten, die die Funktionalität des gesamten Bausteins bestimmen. In diesem Speicher liegen also die detaillierten Informationen darüber, wie die LUTs und die Routing-Matrix innerhalb des FPGAs verschaltet werden.

Der Konfigurationsspeicher hält dazu den sogenannten *Bitstream* vor – eine Folge von Daten, die die spezifische Logik und Verbindungskonfiguration des FPGAs definieren. Der Bitstream ist eine lange Reihe von Nullen und Einsen und hat ein spezifisches Format, das sich von FPGA zu FPGA unterscheidet. Beim Einschalten des FPGAs wird dieser Bitstream aus dem Konfigurationsspeicher gelesen und in die verschiedenen Komponenten des Chips geladen. Dieser Vorgang, der oft als *Konfigurierung* bezeichnet wird, aktiviert die Funktionalität des FPGAs, indem er die LUTs und *Routing-Wege* gemäß dem Design einstellt.

Der Konfigurationsspeicher ist in der Regel *nichtflüchtig*, was bedeutet, dass die Daten auch dann erhalten bleiben, wenn das Gerät ausgeschaltet wird. Es gibt aber bei einigen FPGAs auch die Möglichkeit, den Bitstream in ein SRAM zu schreiben, beispielsweise beim *Tang Nano 9K*. Dies ist ein flüchtiger Speicher, dessen Inhalt bei einem Verlust der Stromversorgung verloren geht. Dies kann beim Prototyping ein Vorteil sein, wenn man den FPGA aus jedem beliebigen Zustand wieder in eine Grundkonfiguration zurückversetzen können möchte, indem man einfach die Stromversorgung unterbricht.

Die Verwendung des nichtflüchtigen *Flash*-Speichers ist wiederum wichtig, wenn man sicherstellen möchte, dass der FPGA sofort und nach jedem Einschalten betriebsbereit ist. Der Konfigurationsspeicher macht den FPGA somit anpassungsfähig, ohne die physische Hardware ändern zu müssen, und ermöglicht auch schnelle Design-Iterationen.

Sicherheitsüberlegungen zum Konfigurationsspeicher

Manche FPGAs haben nur einen internen SRAM-Speicher. Wenn solch ein FPGA in einem Gerät bei jedem Einschalten mit einer *persistenten Konfiguration* initialisiert werden soll, so wird diese aus einem externen Flash-Speicher eingelesen. Die Speicherung eines Bitstreams in einen externen Speicherchip kann allerdings ein potenzielles Sicherheitsproblem darstellen.

Hacker können versuchen, die Übertragung des Bitstreams beim Einschalten des Systems mitzulesen oder sogar den einzulesenden Bitstream zu ersetzen. Um solch einer Kompromittierung des Systems entgegenzuwirken, verfügen manche FPGAs über Sicherheitssysteme, die dafür sorgen, dass nur autorisierter Konfigurationscode geladen und ausgeführt wird – ein Vorgang, der als *Secure Boot* bekannt ist.

Darüber hinaus sind Gegenmaßnahmen integriert, die physische Angriffe durch *Manipulationsschutz* (*Anti-Tampering*) und *Datenlöschung* (*Zeroization*) verhindern. Bei FPGAs mit integriertem Flash-Speicher kann der Inhalt besser geschützt werden. Sollten Sie einmal in die Situation kommen, die Konfiguration des FPGAs, den Bitstream, schützen zu müssen, denken Sie daran, dass ein externer Konfigurationsspeicher – in der Regel – weniger Schutz bietet.

Frequenzmanagement und Clock-Manager

Die PLLs (*Phase-Locked Loops*, dt. *Phasenregelschleifen*) in FPGAs werden verwendet, um die Frequenz der Taktsignale zu steuern und anzupassen. Das ist das sogenannte *Frequenzmanagement*. PLLs ermöglichen es, die Taktfrequenz für verschiedene Teile des FPGAs zu variieren, was für die Synchronisation und das Timing in digitalen Schaltungen wichtig ist. Auch zur Signalstabilisierung tragen diese PLLs

bei, indem sie Schwankungen und *Jitter* (zeitliche Abweichungen im Signal) minimieren. Dies ist für die Signalintegrität in komplexen digitalen Systemen wichtig. Außerdem ermöglichen sie es auch, mit mehreren Taktquellen zu arbeiten und flexibel zwischen diesen wechseln zu können, was für manche Konfigurationen erforderlich sein kann.

Clock Manager in FPGAs sind für die Verteilung des Taktsignals innerhalb des Chips verantwortlich. Sie gewährleisten, dass alle Komponenten des FPGAs synchron arbeiten. Dies beinhaltet auch eine mögliche Anpassung der Taktfrequenz und die Wahl der optimalen Taktquelle. Clock Manager können auch *elektromagnetische Interferenzen* reduzieren, die durch Taktleitungen verursacht werden können.

Taktquellen

Die Taktquellen eines FPGAs können sowohl intern als auch extern sein. So verfügen einige FPGAs über eingebaute Oszillatoren, die als interne Taktquelle dienen können. Diese bieten nicht immer die gleiche Präzision wie externe Oszillatoren, sind aber in bestimmten Anwendungen nützlich. Häufiger werden externe Oszillatoren verwendet, die ein genaueres Taktsignal erzeugen. Diese Oszillatoren sind physisch außerhalb des FPGAs auf der Platine platziert und mit dem FPGA über dedizierte Takteingangspins verbunden. PLLs innerhalb des FPGAs können auch als Taktquellen dienen, indem sie ein externes Taktsignal nehmen und es modifizieren (zum Beispiel durch Frequenzvervielfachung oder -teilung), um unterschiedliche Taktraten für verschiedene Teile des FPGAs zu erzeugen.

IV – Safety first

Wie sieht es mit der Sicherheit Ihres Projekts aus? Gibt es Aspekte, die Sie besonders schützen müssen, sei es vor physischen Schäden oder unbefugtem Zugriff? Notieren Sie Ihre Gedanken zur Sicherheit Ihres Projekts.

2.1.7 Spezifische Komponenten unterschiedlicher FPGAs

Die im vorherigen Abschnitt aufgeführten Punkte sind Dinge, die in jedem FPGA vorhanden sind. Die folgenden Hardware-Bestandteile und Eigenschaften unterscheiden sich von Modell zu Modell und auch von Hersteller zu Hersteller.

DSP-Blöcke (Digital Signal Processing)

Einige FPGAs enthalten spezielle *DSP-Blöcke* für die Verarbeitung von digitalen Signalen, die für Anwendungen in der Signalverarbeitung und Telekommunikation optimiert sind.

Speicherblöcke

Viele FPGAs verfügen über integrierte Speicherblöcke wie SRAM, die für die Zwischenspeicherung von Daten genutzt werden können. Dieser SRAM-Speicher ist schnell, da er keinen Refresh-Zyklus wie DRAM benötigt. Meist ist dieser Speicher nur wenige Kilobyte groß und wird für die Datenzwischenspeicherung, als Konfigurationsspeicher, für die Verwendung mit FIFO- oder LIFO-Datenstrukturen, für Shift-Speicher oder bei der Signalverarbeitung und Pufferung von Datenströmen verwendet. Allerdings ist dieser SRAM-Speicher flüchtig und nur für die temporäre Verwendung zur Laufzeit geeignet. Sobald der FPGA von der Stromversorgung getrennt wird, geht der Inhalt verloren.

Externer Speicher

Für die Speicherung von Daten wird bei FPGAs oft auf externen Speicher wie Flash-Speicher oder EEPROM zurückgegriffen. Diese Speicherarten ermöglichen es, Daten dauerhaft zu sichern. Dabei geht es nicht nur um den Konfigurationsspeicher für den FPGA-Bitstream, sondern auch um Daten, die während des Betriebs des FPGAs erzeugt oder benötigt werden. Ein anschauliches Beispiel hierfür sind Rainbow-Tables, die in der IT-Sicherheit und bei kryptografischen Verfahren eine wichtige Rolle spielen.

Rainbow-Tables

Die Verwendung von *Rainbow-Tables* ist eine effiziente Methode, um das Knacken von Passwörtern zu beschleunigen. Sie schafft einen Ausgleich zwischen dem benötigten Speicherplatz und der Rechenzeit. Der Kniff besteht darin, die Zuordnung von Hash-Werten zu den ursprünglichen Passwörtern schnell wiederherzustellen, ohne den Hash-Algorithmus für jede denkbare Passwortkombination neu berechnen zu müssen.

Der Zugriff auf Daten aus einem solchen Speicher ist oft schneller als die ständige Neuberechnung von Hash-Werten, vor allem dann, wenn diese Berechnungen komplex und zeitintensiv sind. FPGAs können in diesem Zusammenhang speziell angepasste Algorithmen nutzen, um die Suche und den Abgleich in Rainbow-Tables effizient zu beschleunigen. Dank ihrer Fähigkeit, parallele Verarbeitungsprozesse zu optimieren, sind FPGAs in der Lage, eine Vielzahl von Hash-Abgleichen simultan durchzuführen, indem sie auf vorgefertigte Tabellen zurückgreifen. Allerdings benötigen Sie für solche Rainbow-Tables meist viel Speicherplatz, was die Anbindung von Flash-Speicher mit viel Kapazität oder sogar eine SD-Karte notwendig macht.

Diese Kombination aus externem Speicher und der parallelen Verarbeitungskapazität der FPGAs ist ein praktisches Beispiel, wie FPGAs sich als leistungsfähiges Werkzeug in der *Kryptoanalyse* und bei *Sicherheitstests* nutzen lassen. Die geschickte

Kombination von Hardware und Strategie macht einen FPGA zum richtigen Werkzeug für manche Aufgaben. Ihre speziellen Eigenschaften lassen die FPGAs dabei wie eine maßgeschneiderte *Hardware-Beschleunigung* arbeiten. So können sie bei spezialisierten Aufgaben generische (Mikro-)Prozessoren ohne große Mühe übertreffen.

Es gibt auch noch andere spannende Daten, die in einem externen Speicher abgelegt werden können, beispielsweise *Kalibrierungswerte*, lange Reihen von geospatialen Datensätzen oder einfach die Steuersequenz für Ihre RGB-LEDs.

Hochgeschwindigkeits-Transceiver

Einige fortschrittliche FPGAs bieten integrierte *Transceiver* für die Hochgeschwindigkeitsübertragung von Daten. Sie sorgen für die schnelle Datenübertragung zwischen dem FPGA und anderen Geräten oder Systemen und unterstützen oft verschiedene Übertragungs- und Kommunikationsstandards. Diese Transceiver sind besonders in Anwendungen wichtig, die hohe Datenraten erfordern, wie in der Telekommunikation, bei Hochgeschwindigkeitsnetzwerken oder in der digitalen Signalverarbeitung. Darin enthalten sein kann schon eine *Signalwandlung* (Kupfer auf Glasfaser) oder auch Funktionen, um die *Signalintegrität* zu verbessern (*Clock Data Recovery*). Wenn Sie einen FPGA mit solch einer Schnittstelle benötigen, dann studieren Sie das Datenblatt, um frühzeitig zu prüfen, ob der Transceiver auch das leistet, was Sie von ihm erwarten.

Wo werden solche FPGAs verwendet? Beim Hochfrequenz-Trading, in Netzwerkgeräten wie Switches oder Firewalls, in Teilchenbeschleunigern und für andere Dinge, bei denen riesigen Mengen an Daten bewegt werden müssen – kurz gesagt bei Dingen, die entweder viel Geld kosten oder viel Geld machen.

Hardcore-Prozessorkerne

Einige FPGAs integrieren Prozessorkerne, die für spezifische Aufgaben bereitgestellt werden können. Solche Hardcore-Prozessorkerne sind fest im FPGA integrierte Mikroprozessoren, die im Gegensatz zur restlichen programmierbaren Logik des FPGAs nicht veränderbar sind (deshalb der Begriff *Hardcore*). Sie dienen als Verarbeitungseinheiten für spezifische Aufgaben und erlauben eine effiziente Ausführung von Software. Solche Prozessoren, oft ARM-basiert, werden in *System-on-Chip*-(SoC-)FPGAs integriert und bieten die Möglichkeiten eines Mikrocontrollers innerhalb des FPGAs. Der *Tang Nano 4K* ist beispielsweise solch ein FPGA. Dieser FPGA ist ideal für Anwendungen, die eine Kombination aus spezialisierter Hardware-Logik und traditioneller Software-Verarbeitung erfordern, wie in *Embedded Systems*, bei der Signalverarbeitung oder in IoT-Geräten. Diese Integration ermöglicht eine hohe Leistungsfähigkeit und Flexibilität in komplexen Systemdesigns, weil zwei Welten im FPGA miteinander verbunden werden können.

Würden Sie einen Mikroprozessor im FPGA selbst designen, wäre dies ein sogenannter *Softcore*-Prozessor. Für einen einfachen 8-Bit- oder 16-Bit-Softcore-Mikrocontroller benötigen Sie rund 1000 bis 5000 LUTs, und für komplexere 32-Bit-Prozessoren können gut und gerne 5000 bis 20.000 LUTs benötigt werden. Da kann es einfacher und günstiger sein, einen FPGA mit integriertem Prozessorkern auszuwählen, denn je mehr LUTs ein FPGA hat, desto teurer wird er.

Analoge Schnittstellen

Einige FPGAs bieten analoge Schnittstellen, was ihre Anwendung in gemischten Signalumgebungen erweitert. Sie ermöglichen die Interaktion mit analoger Elektronik, was für Anwendungen wie Sensorik, Signalverarbeitung und Regelungstechnik wichtig ist.

Diese Schnittstellen bestehen typischerweise aus *Analog-Digital-Wandlern* (ADCs) und *Digital-Analog-Wandlern* (DACs), um analoge Signale in digitale Daten umzuwandeln und umgekehrt. Obwohl FPGAs hauptsächlich digitale Bausteine sind, erlauben integrierte oder externe analoge Schnittstellen die direkte Verarbeitung von realen, analogen Signalen. Achten Sie aber auf die Kompatibilität der verwendeten Spannungspegel. Je nach FPGA-Technologie beträgt diese meistens 3,3 V, 2,5 V oder 1,8 V. Ein von außen angelegtes analoges Signal muss daran angepasst werden.

Digitale Schnittstellen

Zu den digitalen Schnittstellen, die FPGAs bereitstellen können, gehören beispielsweise SPI (*Serial Peripheral Interface*) und UART (*Universal Asynchronous Receiver/Transmitter*) für die serielle Kommunikation. Manche FPGAs haben auch DVI-(*Digital Visual Interface*-) oder HDMI-(*High-Definition Multimedia Interface*-)Schnittstellen implementiert, um eine Verwendung mit Audio- und Videosignalen zu vereinfachen.

Durch die unterschiedlichen Einsatzgebiete, für die ein FPGA vom Hersteller vorgesehen ist, und die damit verbundenen Leistungsanforderungen unterscheiden sich FPGAs in ihren Features, wie DSP-Blöcken, integriertem Speicher, Hochgeschwindigkeitsschnittstellen und eingebauten Prozessorkernen. Je nach vorgesehener Aufgabe ist also eine genaue Auswahl des geeigneten FPGA-Modells nötig.

2.1.8 Verarbeitungsgeschwindigkeit und Echtzeitfähigkeit

Schon mehrfach habe ich sie erwähnt: Die *Echtzeitfähigkeit* bezeichnet die Fähigkeit eines Systems, auf Eingaben innerhalb eines festgelegten, garantierten Zeitrahmens zu reagieren, was in vielen technischen Anwendungen – von der In-

dustrieautomatisierung bis hin zur Telekommunikation – unabdingbar ist. Diese prompte Reaktionszeit gewährleistet, dass das System präzise und zuverlässig unter zeitkritischen Bedingungen operieren kann.

FPGAs zeichnen sich besonders durch ihre Fähigkeit zur parallelen Verarbeitung aus, wodurch sie mehrere Operationen gleichzeitig durchführen können. Diese Hardware-basierte Parallelität ermöglicht es, extrem schnelle Verarbeitungsgeschwindigkeiten zu erreichen und komplexe Berechnungen in Echtzeit durchzuführen – ohne die Verzögerungen, die mit sequenzieller Datenverarbeitung oft verbunden sind. Daher sind FPGAs ideal für Anwendungen geeignet, bei denen es auf jede Millisekunde ankommt, beispielsweise in der Signalverarbeitung oder bei der Datenflusssteuerung in kritischen Systemen.

Im Vergleich dazu sind RTOS (*Real-Time Operating Systems*) darauf ausgelegt, Aufgaben präzise zu planen und zu verwalten, um sicherzustellen, dass sie innerhalb eines genau definierten Zeitfensters bearbeitet werden. Durch die Unterstützung von Multithreading ermöglichen RTOS zwar eine Form der parallelen Aufgabenbearbeitung, diese erfolgt jedoch auf Software-Ebene und hängt somit davon ab, wie leistungsfähig die zugrunde liegenden Hardware-Prozessoren bei der Parallelisierung sind.

Ein RTOS ist somit besonders geeignet für Szenarien, in denen die zeitgerechte Ausführung von Aufgaben in einer vorhersehbaren Reihenfolge entscheidend ist, und bietet eine Lösung für Anwendungen, die eine zuverlässige, zeitkritische Verarbeitung erfordern.

Beide Technologien – FPGA und RTOS – haben somit ihre spezifischen Stärken und Einsatzgebiete, die sie für unterschiedliche Aspekte der Echtzeitdatenverarbeitung prädestinieren. In Situationen allerdings, in denen die Geschwindigkeit der Aufgabenverarbeitung und die Fähigkeit zur simultanen Bearbeitung zahlreicher Prozesse kritisch ist, stoßen RTOS-Systeme an ihre Grenzen, weil sie von der sequenziellen Verarbeitungskapazität der zugrunde liegenden CPU und von Software-Scheduling-Strategien abhängig sind.

FPGAs hingegen können in diesen Bereichen noch »locker weitermachen«, wenn RTOS-Systemen die Puste ausgeht, da ihre Hardware direkt für parallele Verarbeitung ausgelegt und somit nicht von den gleichen Beschränkungen betroffen ist.

6.5 Die Ausgabeinformationen der Toolchain verstehen

Wenn Sie die beiden Blinky-Testprogramme der FPGA-Boards mit der Toolchain ausgeführt haben, sind Ihnen sicher die Unmengen an Ausgabe-Zeilen der Programme zur Synthese und für das Place-and-route aufgefallen.

Wenn Sie Yosys nicht den Parameter `-q` mitgeben, um sich auf das Wesentliche zu beschränken, kann allein die Ausgabe der Synthese rund 1000 Zeilen beinhalten!

Kein Mensch liest so etwas im Normalfall, aber bei Problemen – oder aber um sich einen groben Überblick über ein paar Eckpunkte des Designs zu verschaffen –, sind ein paar Zeilen der Ausgabe doch wichtig.

Nehmen wir den Tang Nano 9K als Beispiel und sehen wir uns im Folgenden die wichtigsten Zeilen an, die Sie beachten sollten.

Device Utilisation

Der Abschnitt `Device utilisation` zeigt, wie die verschiedenen Ressourcen des FPGAs genutzt werden. So stellen Sie sicher, dass keine Ressourcen überlastet sind oder ungewöhnlich hohe Auslastungswerte aufweisen.

```
Info:              Device            utilisation:
Info:                 VCC:       1/       1   100%
Info:               SLICE:     115/    8640     1%
Info:                 IOB:       4/     274     1%
Info:              OSER16:       0/      38     0%
Info:              IDES16:       0/      38     0%
Info:             IOLOGIC:       0/     296     0%
Info:          MUX2_LUT5:        2/    4320     0%
Info:          MUX2_LUT6:        1/    2160     0%
Info:          MUX2_LUT7:        0/    1080     0%
Info:          MUX2_LUT8:        0/    1056     0%
Info:                 GND:       1/       1   100%
Info:                RAMW:       0/     270     0%
Info:                 OSC:       0/       1     0%
Info:                rPLL:       0/       2     0%
```

Wenn Sie beispielsweise `SLICE: 115 / 8640 (1%)` betrachten, erkennen Sie, dass 115 von 8640 verfügbaren SLICE-Einheiten in diesem Design verwendet werden. Die Angaben `VCC` und `GND` stehen jeweils auf 100 %, was normal ist, da diese Ressourcen für die Energieversorgung der Schaltung notwendig sind.

Placement and Routing

Der Abschnitt `Placement and Routing` zeigt, wie gut die Platzierung und das Routing der Logikzellen im FPGA durchgeführt wurden. Die Platzierung bezieht sich darauf, wie die Logikzellen auf dem FPGA-Chip angeordnet sind, während das Routing die Verbindungen zwischen diesen Zellen beschreibt. Eine effiziente Platzierung und ein effizientes Routing sind entscheidend für die Leistung und Funktionalität des Designs.

```
Info: Placed 4 cells based on constraints.
Info: Creating initial analytic placement for 47 cells, random
placement wirelen = 2039.
...
Info: Routing 362 arcs.
...
Info: Routing complete.
Info: Router1 time 5.19s
```

Eine ganze Menge an Informationen steckt in diesen paar Zeilen:

- `Placed 4 cells based on constraints`: Diese Meldung zeigt an, dass 4 Logikzellen entsprechend der festgelegten Einschränkungen platziert wurden. Einschränkungen können physische Positionen oder Designanforderungen sein.
- `Creating initial analytic placement for 47 cells, random placement wirelen = 2039`: Hier wird eine anfängliche analytische Platzierung für 47 Zellen erstellt, wobei die anfängliche Länge der Verbindung zufällig auf 2039 festgelegt ist. Dies ist oft der erste Schritt, um eine grobe Anordnung der Zellen zu evaluieren.
- `Routing 362 arcs`: Dieser Schritt beschreibt den Prozess des Verbindens von 362 Bögen (Verbindungen) zwischen den Logikzellen. Routing ist entscheidend, um sicherzustellen, dass alle logischen Verbindungen korrekt und effizient sind.
- `Routing complete`: Diese Meldung bestätigt, dass der Routing-Prozess nun abgeschlossen ist.
- `Router1 time 5.19s`: Die Zeit, die der erste Routing-Vorgang in Anspruch genommen hat, betrug 5,19 Sekunden.

Timing Analysis

Der Abschnitt `Timing Analysis` ist entscheidend, da er die maximal erreichbare Taktrate und die kritischen Pfade im Design zeigt. Die Timing-Analyse bewertet, ob das Design innerhalb der vorgegebenen Zeitvorgaben korrekt funktioniert. Dabei wird sichergestellt, dass alle logischen Signale innerhalb der geforderten Zeiträume verarbeitet werden, um Fehler zu vermeiden.

```
Info: Max frequency for clock 'clk_IBUF_I_O': 316.86 MHz (PASS at
12.00 MHz)
```

Hier geht es darum, ob das Design innerhalb der gewünschten Frequenzgrenzen zuverlässig arbeitet und optimale Leistung liefert:

- `Max frequency for clock 'clk_IBUF_I_O': 316.86 MHz`: Diese Meldung gibt die maximale Frequenz an, bei der das Clock-Signal `clk_IBUF_I_O` betrieben werden kann, ohne dass Timing-Verstöße auftreten. Eine höhere Frequenz bedeutet, dass das FPGA-Design schneller arbeiten kann, was in vielen Anwendungen entscheidend ist.
- `PASS at 12.00 MHz`: Dies bedeutet, dass das Design bei einer Taktrate von 12 MHz erfolgreich getestet wurde. `PASS` zeigt an, dass das Timing bei dieser Frequenz eingehalten wird und das Design zuverlässig arbeitet.

Critical Path Reports

Der Abschnitt `Critical path report` zeigt die kritischen Pfade, die für die Timing-Analyse wichtig sind. Diese Pfade bestimmen die maximale Taktrate des Designs. Ein kritischer Pfad ist der längste Pfad, den ein Signal durchlaufen muss, um von einem Ausgangspunkt zu einem Endpunkt zu gelangen. Die Zeit, die das Signal für diesen Pfad benötigt, bestimmt die maximale Geschwindigkeit, mit der das Design sicher arbeiten kann.

```
Info: Critical path report for clock 'clk_IBUF_I_O' (posedge ->
posedge):
Info: curr total
Info:  0.5  0.5  Source counter_DFFR_Q_20_DFFLC.Q
Info:  0.8  1.3    Net counter[13] (43,3) -> (43,4)
Info:              Sink counter_DFFR_Q_20_D_ALU_SUM_ALULC.B
Info:              Defined in:
Info:                 servo.v:10.12-10.19
...
```

Der Bericht über die kritischen Pfade bietet einen wichtigen Einblick in die zeitlichen Eigenschaften des FPGA-Designs:

- `Critical path report for clock 'clk_IBUF_I_O' (posedge -> posedge)`: Dieser Bericht beschreibt den kritischen Pfad für das Taktsignal `clk_IBUF_I_O`, der vom positiven Taktflankenwechsel (`posedge`) zum nächsten positiven Taktflankenwechsel reicht. Der kritische Pfad für dieses Taktsignal gibt die Verzögerungen an, die die maximale Taktrate beeinflussen.

- `curr total`: Diese Spalten zeigen die aktuellen und kumulierten Verzögerungen entlang des kritischen Pfads an. Die Verzögerungen werden in Zeiteinheiten (z. B. Nanosekunden) angegeben.
- `0.5 0.5 Source counter_DFFR_Q_2O_DFFLC.Q`: Hier beginnt der kritische Pfad am Ausgang `Q` des Flip-Flops `counter_DFFR_Q_2O_DFFLC`. Die Verzögerung an diesem Punkt beträgt 0,5 Einheiten.
- `0.8 1.3 Net counter[13] (43,3) -> (43,4)`: Das Signal bewegt sich dann über das Netz `counter[13]` von der Position `(43,3)` zur Position `(43,4)` mit einer zusätzlichen Verzögerung von 0,8 Einheiten, was zu einer kumulierten Verzögerung von 1,3 Einheiten führt.
- `Sink counter_DFFR_Q_2O_D_ALU_SUM_ALULC.B`: Der kritische Pfad endet am Eingang `B` der Komponente `counter_DFFR_Q_2O_D_ALU_SUM_ALULC`.
- `Defined in: servo.v:10.12-10.19`: Diese Zeile zeigt, dass der definierte kritische Pfad in der Datei *servo.v* zwischen den Zeilen 10.12 und 10.19 beschrieben ist. Dies hilft Entwicklern, den genauen Codeabschnitt zu identifizieren, der für diesen Pfad verantwortlich ist.

Slack Histogram

Der Abschnitt `Slack histogram` zeigt die Timing-Reserve (*Slack*) für die Endpunkte. Es ist wichtig, dass die Reserve ausreichend ist, um Timing-Verletzungen zu vermeiden. Der Begriff *Slack* bezeichnet die Differenz zwischen der maximal zulässigen Verzögerung und der tatsächlichen Verzögerung eines Signalpfads. Positive Slack-Werte bedeuten, dass der Signalpfad innerhalb der vorgegebenen Zeit abgeschlossen wird, während negative Werte auf Timing-Verletzungen hinweisen.

```
Info: Slack histogram:
Info:  legend: * represents 1 endpoint(s)
Info:          + represents [1,1) endpoint(s)
Info: [ 80177,  80292) |******************
...
```

Die Überprüfung der Timing-Reserven und der Timing-Konformität eines Designs ist wichtig, um sicherzustellen, dass alle Signalpfade innerhalb der zulässigen Zeitvorgaben abgeschlossen wurden:

- `Slack histogram`: Das Histogramm zeigt die Verteilung der Slack-Werte für alle Endpunkte im Design. Ein Endpunkt ist der letzte logische Punkt eines Signalpfads. Die Werte in eckigen Klammern geben den Bereich der Slack-Werte an.
- `legend: * represents 1 endpoint(s)`: Jedes * im Histogramm steht für einen Endpunkt. Wenn im Histogramm also beispielsweise zehn Sterne in einer Zeile stehen, bedeutet dies, dass zehn Endpunkte Slack-Werte in diesem Bereich aufweisen.

- `legend: + represents [1,1) endpoint(s)`: Jedes + im Histogramm steht für einen Endpunkt innerhalb des angegebenen Bereichs.
- `[ 80177, 80292) |**********`: Diese Zeile zeigt einen Bereich von Slack-Werten (80177 bis 80292) an. Die Anzahl der * repräsentiert die Anzahl der Endpunkte, die Slack-Werte in diesem Bereich aufweisen. In diesem Fall gibt es viele Endpunkte mit Slack-Werten in diesem Bereich.

Das sind die wichtigsten Zeilen, die Ihnen dabei helfen, potenzielle Probleme zu identifizieren und sicherzustellen, dass das Design den Timing-Anforderungen entspricht. Am Anfang werden Sie eher auch mal auf eine Error-Zeile stoßen, wenn etwas in Ihrem Verilog-Code nicht funktioniert. Das kann beispielsweise so aussehen:

```
ERROR: Unable to place cell 'te.fontBuffer.0.0', no BELs remaining to
implement cell type 'SPX9'
```

Manchmal sieht es auch so aus:

```
adc1115.blif:6110: fatal error: toplevel inout port 'i2cSda' connected to
tristate buffer and driving a net
```

Wie auch immer die Fehlermeldung aussieht, die entsprechende Error-Meldung finden Sie immer in den letzten Zeilen der Ausgabe. Vermutlich haben Sie in den meisten Fällen schon eine Ahnung, welcher Teil des Codes den gemeldeten Fehler verursacht. In allen anderen Fällen kann Ihnen aber auch die Chat-KI weiterhelfen.

Falls Ihre Toolchain in Zukunft einmal in ein automatisches Build-System eingebunden werden soll, können Sie bereits jetzt die ersten wichtigen Zeilen definieren, die geparst werden sollten.

Schreiben in eine Datei

Schließlich werden die gesammelten Hex-Werte in eine Datei namens *font.hex* geschrieben. Dazu wird die Datei *font.hex* im Schreibmodus geöffnet, und die gesammelten Hexadezimalwerte werden in die Datei geschrieben, getrennt durch Leerzeichen:

```
with open('font.hex', 'w') as file:
    file.write(' '.join(memory))
```

Somit haben wir mithilfe des Skripts die *Bitmap-Schriftartdatei monogram.json* in eine Datei mit dem Namen *font.hex* konvertiert, die wir nun zur Anzeige unserer Zeichen verwenden können.

10.9.4 Das Design der Text-Engine

Bevor wir das eigentliche Verilog-Modul implementieren können, müssen wir noch unseren Schlachtplan aufstellen. Die gesamte Text-Engine sieht in ihrer Funktionalität so aus, wie in Abbildung 10.26 skizziert.

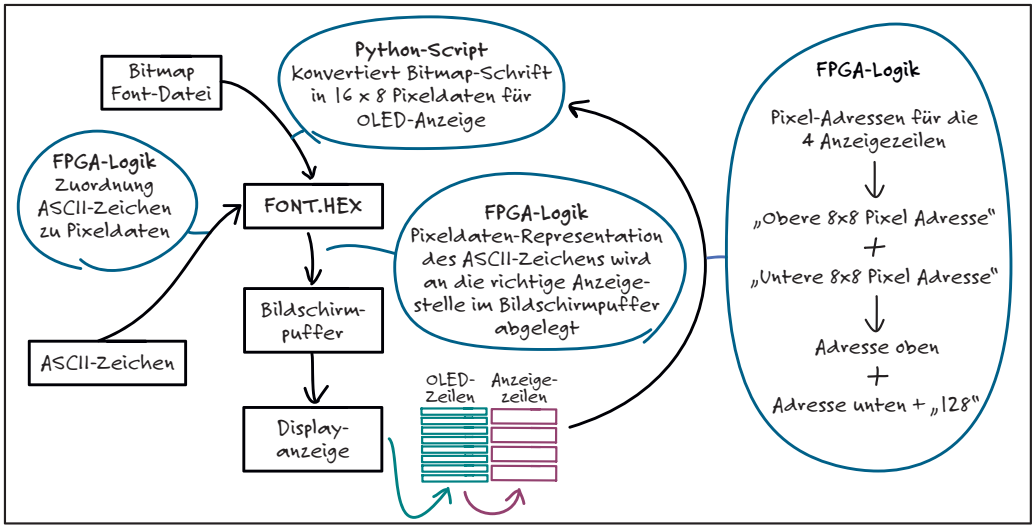

Abbildung 10.26 Übersicht über die Text-Engine

Gehen wir dies Schritt für Schritt durch, um die Punkte zu verstehen, die in der Übersicht zu sehen sind. In Abschnitt 10.8 hatten wir ein `pixelCounter`-Register, das jedes Byte eingelesen hat, das wir auf dem Bildschirm schreiben wollten. So hat es die Pixelinformation fortlaufend aus dem Bildschirmpuffer gelesen und auf dem Display dargestellt:

```
dataToSend <= screenBuffer[pixelCounter];
```

Damit wurde das Bild in einem Rutsch eingelesen und angezeigt.

Das ist eine recht einfache Methode, auf den Bildspeicher zuzugreifen. Bei einer realistischeren Nutzung des Displays würde man allerdings gezielt eine Speicheradresse abfragen und die dort gespeicherten Pixeldaten entgegennehmen und anzeigen wollen.

Bei unserer Text-Engine gehen wir deshalb einen anderen Weg. Wir haben eine Anzeige-Stelle, eine Position auf der Anzeige, an der wir dynamisch ein 16 × 8 Pixel großes Bild – das Zeichen – anzeigen wollen. Wenn also an dem einen Ende ein ASCII-Zeichen hineinkommt, wollen wir am anderen Ende das 16 × 8 Pixel große Bild, das dieses Zeichen repräsentiert, an einer in dem Moment dafür vorgesehenen Stelle auf dem Display anzeigen. Ist es das erste Zeichen der Zeichenkette »ABCD«, dann gehört das »A« an die erste Anzeigeposition und das »D« an die vierte Anzeigeposition. Das darzustellende Bild des ASCII-Zeichens muss folglich dynamisch ermittelt werden und die Position der Anzeige dieses Bildes (d. h. die Anzeigestelle dieses Zeichens) muss ebenfalls dynamisch ermittelt werden.

Es gibt nun mehrere Möglichkeiten, um so etwas zu implementieren:

- Wir ermitteln die Position und die anzuzeigenden Bilder zuerst, schreiben alles in der richtigen Reihenfolge in den Bildschirmpuffer und lesen diesen dann wieder in einem Rutsch aus.
- Wir aktualisieren den Inhalt des Bildschirmpuffers partiell an den Stellen, an denen sich die Anzeige verändert.

Wir hatten bei der Funktionsweise des SSD1306-Controllers besprochen, wie die Anzeige aus dem Bildschirmspeicher heraus aktualisiert wird:

> *Der Vorteil der Verwendung eines RAM-Bildschirmpuffers ist, dass der eigentliche Bildschirm unabhängig vom Master, der das Display-Modul steuert, aktualisiert werden kann und dass man auch nur eine spezifische Aktualisierung senden kann, um einen Teil des Bildes zu ändern. Der Rest der Anzeige wird den nicht aktualisierten Teil wie gehabt aus dem Bildschirmpuffer anzeigen.*

Genau diese Eigenschaft der Funktionsweise des OLED-Displays nutzen wir nun, um das Bild *dynamisch* zu erzeugen.

Möchten wir nun ein Zeichen auf dem Display darstellen, sind folgende Schritte notwendig:

1. Zuordnung des ASCII-Zeichens zu einem Bild aus 16 × 8 Pixeldaten
2. Ermitteln, in welcher der vier Anzeigezeilen der Buchstabe angezeigt werden soll
3. Ermitteln, an welcher Stelle dieser Anzeigezeile der Buchstabe angezeigt werden soll

4. Umrechnen, in welcher der acht OLED-Zeilen die obere und die untere Hälfte der Pixeldaten des 16×8-Bildes angezeigt werden sollen, und herausfinden, wie die dazugehörigen Adressen lauten
5. Den Befehl zur Aktualisierung der Display-Anzeige an den Controller senden, der daraufhin den aktualisierten Bildschirmpuffer ausliest

Wie wird dies nun in Verilog umgesetzt? Mit ein paar Grundkenntnissen in der Binärsprache können wir dies recht einfach in der Logik unseres FPGAs abbilden.

10.9.5 Die Zuordnungslogik der Adressen

Bevor wir damit beginnen, Zeichen in Zeilen aufzuteilen und uns eine Zuordnungslogik auszudenken, sollten wir die Handhabung dieser vier Anzeigezeilen vereinfachen, denn wir haben hier vier Zeilen mit jeweils 16 Zeichen. Ich denke, es ist einfacher, diese 4 Anzeigezeilen als ein einziges langes Array zu betrachten.

Vereinfacht dargestellt, sieht dies so wie in Abbildung 10.27 aus.

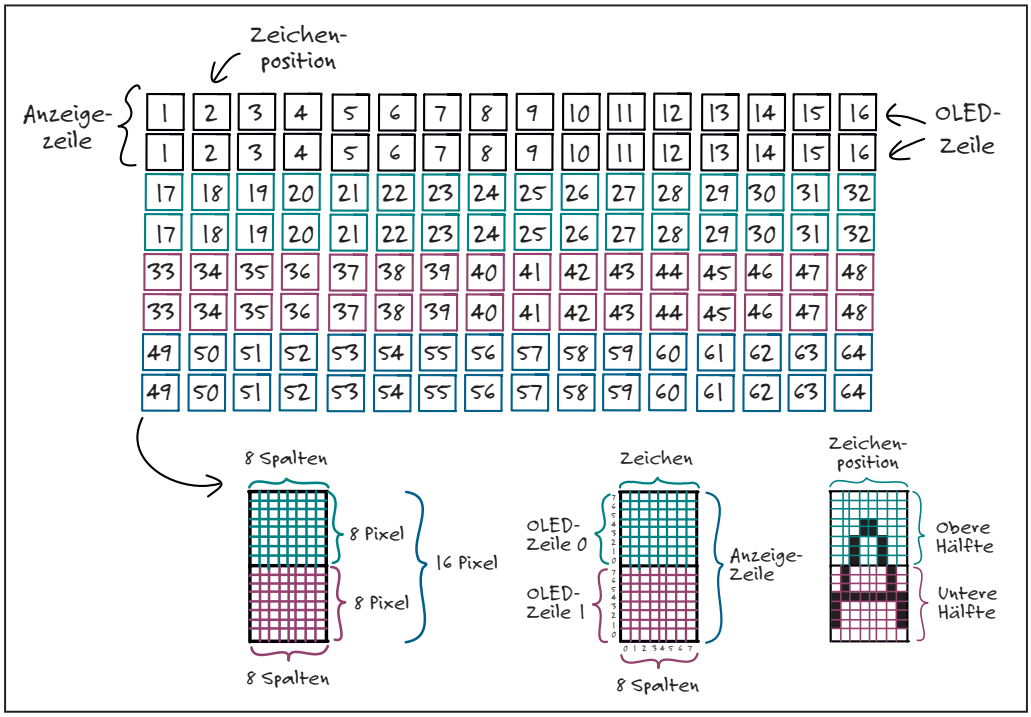

Abbildung 10.27 Anzeigematrix des OLED-Displays

- Eine Zeile entspricht einer OLED-Zeile.
- Zwei OLED-Zeilen, die farbig zusammengehören, repräsentieren eine Anzeigezeile.
- Die jeweiligen Zahlen bezeichnen die Anzeige-Stelle, an der ein Zeichen angezeigt werden kann.
- Die Organisation der Anzeige ist so aufgebaut, dass wir von links nach rechts gehen und dann zur nächsten Zeile wechseln.

Für jede der 64 möglichen Zeichen-Anzeige-Stellen können wir uns nun jeweils die Pixeladresse dieser Zeichen-Anzeigeposition organisieren. Dazu rufen wir uns noch einmal die Organisation der OLED-Zeilen und der Anzeigezeilen ins Gedächtnis: Je zwei Anzeigezeilen bilden den oben und unteren Teil eines Zeichens. Jedes Zeichen wiederum besteht aus 8 vertikalen Pixeln in jeder OLED-Zeile (siehe Abbildung 10.28).

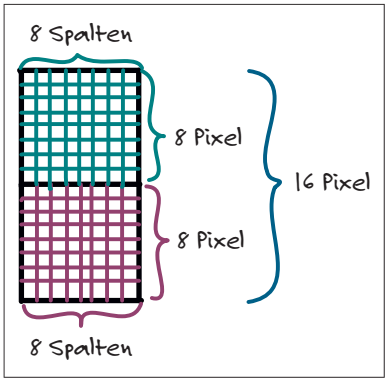

Abbildung 10.28 Die Pixel und Spaltenaufteilung einer Anzeigestelle

Um die Anzeige eines Zeichens innerhalb einer OLED-Zeile zu erreichen, benötigen wir einen Unterzähler für die 8 vertikalen Pixel, also die Spalten, eines Zeichens.

Jede Zahl des Index dieses Zählers repräsentiert somit 8 vertikale Pixel, die wir aus der Fonts-Datei holen werden. Die erste Zeile entspricht den *Anzeige-Stellen* 0 bis 15. Danach zählen wir die nächsten 16 Anzeige-Stellen weiter für die Darstellung der Zeichen in der nächsten OLED-Zeile.

Sobald das Abbilden der Pixeldaten zweier OLED-Zeilen (und damit einer Anzeigezeile) abgeschlossen ist, erhöhen wir den Zeilen-Index um eins und starten mit der nächsten Anzeigezeile. Hier wiederholen wir die gesamte Prozedur, um die beiden OLED-Zeilen zu füllen, und wenden uns dann wieder der nächsten Anzeigezeile zu.

Wir benötigen also eine eindeutige Adressierung der zum Zeichen gehörenden Pixel-Spalten auf dem Display.

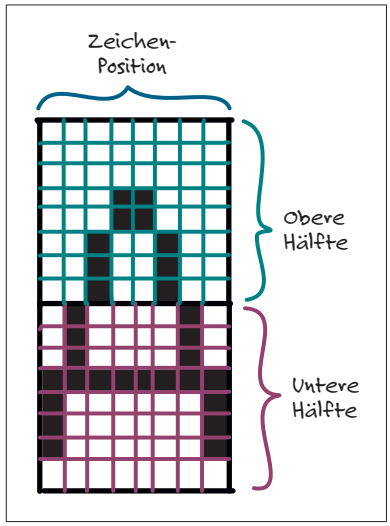

Abbildung 10.29 Die Darstellung eines Zeichens in einer Anzeigestelle

Hier können wir einen Trick nutzen, um uns die Implementierung der Logik einfacher zu machen: Wir denken wie ein FPGA. Das bedeutet, wir denken in Bits. Oder anders gesagt: Durch die Verwendung von Binärzahlen und die Aufteilung der Bits entsprechend den benötigten Schleifen (8 für die Spalten eines Zeichens und 16 für die Anzeige-Stellen einer Zeile) können wir die Adressierung und die Wiederholungsmuster automatisch durch den Überlauf der Binärstellen handhaben. So können wir das Überlauf-Bit als Flag für die obere oder untere OLED-Zeile verwenden. Den Rest der Bits verwenden wir dann als relative Adresse weiter. Mit einem Beispiel wird das klarer:

Jede OLED-Zeile besteht aus 128 Spalten.

Die Pixel-Spalten in der ersten OLED-Zeile haben die Adressen 0 bis 127:

0-7	8-15	16-23	24-31	32-39	40-47	48-55	56-63	64-71	72-79	80-87	88-95	96-103	104-111	112-119	120-127

Abbildung 10.30 Der OLED-Spaltenindex der ersten Zeile

Die Pixel-Spalten in der zweiten OLED-Zeile haben die Adressen 128 bis 255:

128-135	136-143	144-151	152-159	160-167	168-175	176-183	184-191	192-199	200-207	208-215	216-223	224-231	223-239	240-247	248-255

Abbildung 10.31 Der OLED-Spaltenindex der zweiten Zeile

Überlauf des Spaltenindex-Bits

Betrachten wir die Adresse der letzten Spalte in der ersten OLED-Zeile und die Adresse der ersten Spalte in der zweiten OLED-Zeile:

- 127 → 0111 1111
- 128 → 1000 0000

Hier sehen wir, dass ein Überlauf stattgefunden hat und nun die erste Stelle der Binärzahl auf 1 gewechselt ist und die anderen Stellen wieder auf 0 stehen.

Schauen wir uns nun die Adresse der ersten Spalte in der ersten OLED-Zeile an:

- 0 → 0000 0000

Wenn wir nur die hinteren sieben Bits der Spalte 0 betrachten, ist es die gleiche Adresse wie die hinteren sieben Bits der Spalte 128, der ersten Spalte in der zweiten OLED-Zeile.

Wenn wir also in Verilog das vorderste Bit als OLED-Zeilenindex verwenden, können wir anhand der 0 oder 1 sagen, ob es sich um den oberen oder unteren Teil eines Zeichens handelt. Die restlichen sieben Bits geben dann in beiden Fällen die gleiche Anzeigeposition an – wir verwenden also eine relative Adressierung.

Spaltenzähler

Wir benötigen nun eine Schleife für die 8 vertikalen Pixel einer jeden Pixelspalte. Dies bedeutet, dass wir 3 Bits der Adresse verwenden können, um die Spalten zu zählen, da $8 = 2^3$.

Zeilenindex

Der Zeilenindex muss alle 16 Stellen wechseln. Das bedeutet, dass wir die nächsten 4 Bits verwenden können, um die Zeichen-Position auf der Zeile zu zählen, da $16 = 2^4$.

Zyklus wiederholen

Da wir vier Anzeigezeilen mit jeweils 16 Zeichen verwenden, müssen wir sicherstellen, dass der gesamte Zyklus viermal stattfindet. Dies wird durch zwei weitere Bits der Adresse gehandhabt, da $4 = 2^2$.

Die Verwendung einer Pixelspalten-Adresse

Setzen wir diese ganzen Bits nun zusammen, um beispielsweise die Pixelspalten-Adresse 569 in unserer Display-Anzeige zu bestimmen.

Das OLED-Display hat eine Auflösung von 128 × 64 Pixeln, was wiederum als ein Array von 8192 (128 × 64) Pixeln betrachtet werden kann. Jede Spalte besteht aus 8 vertikalen Bits. Das bedeutet, wir verwenden 1024 (8192 / 8) Adressen. Das gesamte Adress-Array bezieht sich somit auf die Spalten der nativen 8 OLED-Zeilen, die der SSD1306-Controller steuert. Jede OLED-Zeile des Displays besteht aus 16 Zeichen, wobei jedes Zeichen aus 8 Spalten und 8 Zeilen besteht.

Aufteilung der Zeichen im Display

Das Display haben wir in 4 Anzeigezeilen organisiert (je zwei OLED-Zeilen sind zu einer Anzeigezeile zusammengefasst), damit die Zeichen größer und lesbarer sind, wobei aber jede Zeile weiterhin 16 Zeichen enthält. Insgesamt haben wir mit den 4 Anzeigezeilen 64 Zeichenpositionen (4 Zeilen × 16 Zeichen). Und wir verwenden das Überlauf-Bit, um bei der relativen Adressierung zu erkennen, ob es sich gerade um die obere oder untere Hälfte des Zeichens handelt, das gerade mit den Pixeln gemalt wird. Abbildung 10.32 zeigt die Aufteilung schematisch.

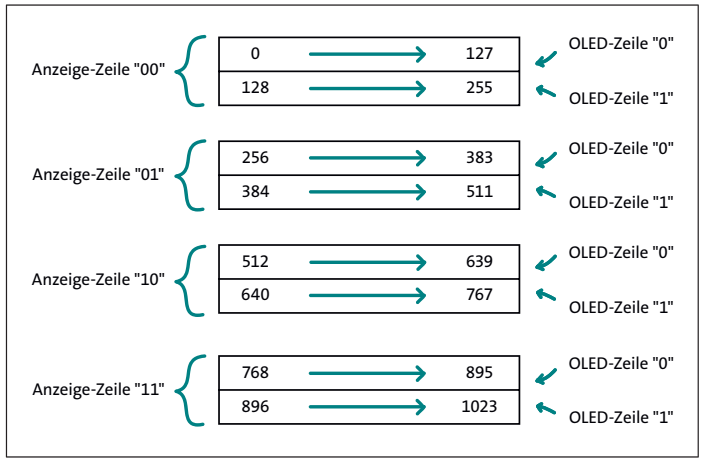

Abbildung 10.32 Die Organisation der OLED- und Anzeigezeilen sowie der Pixel-Spalten-Adressen

Berechnung der Pixel-Spalten-Adresse

Die Pixel-Spalten-Adresse bezieht sich in dieser gedachten Struktur nun auf die spezifische Position einer Pixel-Spalte des Displays, das aus 1024 Pixel-Spalten besteht. Wenn wir die Pixel-Spalten in einer fortlaufenden Reihenfolge von links oben nach rechts unten durchnummerieren, können wir eine eindeutige Adresse erhalten. Im Info-Kasten sehen Sie dazu ein Beispiel.

Binäre Betrachtung der Adresse 569:

Die Binärdarstellung der Adresse »569« lautet: 1000111001

Wir haben die 10 Bits der Adresse folgendermaßen organisiert:

- B9 B8 B7 B6 B5 B4 B3 B2 B1 B0
- Anzeigezeilen-Index = B9 B8
- OLED-Zeileninformation oben/unten = B7
- Zeichen-Positions-Index = B6 B5 B4 B3
- Spaltenzähler = B2 B1 B0

Entschlüsseln wir eine Anzeige-Adresse von hinten nach vorne:

- **Spaltenzähler (die unteren 3 Bits)**: 001 (Das entspricht einer 1 im Dezimalformat.)
  Dies zeigt, dass es sich um die 2. Spalte (die Zählung beginnt bei »0«) innerhalb des Zeichens handelt.
- **Zeichen-Positions-Index (nächste 4 Bits)**: 0111 (Das entspricht »7« im Dezimalformat.)
  Dies zeigt an, dass es sich um die 8. Zeichen-Anzeigeposition in der Zeile handelt.
- **OLED-Zeileninformation (Bit B7)**: 0 (obere Hälfte des Zeichens)
  Dies zeigt, dass es sich um die obere Hälfte des Zeichens handelt, also um die 1. Zeile der beiden zusammengefassten OLED-Zeilen.
- **Zeilen-Index B9 B8**: 10 (Das entspricht »2« im Dezimalformat.)
  Dies zeigt an, dass es sich um die 3. Anzeigezeile handelt.

Nun wissen wir, wie der Schlachtplan aussieht, und können diese Logik mit Verilog in den FPGA »gießen«.

10.9.6 Der Verilog-Code

Was ist zu tun?

- Wir konfigurieren das Bildschirmmodul so, dass es die Pixeladresse ausgibt (`pixelAddress`) und die Pixeldaten von einem externen Modul empfängt (`pixelData`).
- Wir verbinden die `pixelCounter`-Variable mit `pixelAddress`, um die aktuelle Pixeladresse bereitzustellen.

Wir müssen eine Änderungen am bisherigen Code in der Datei *spi-oled-text-screen.v* vornehmen, damit das Verilog-Modul die Daten von einem externen Modul anstelle des `screenBuffer` empfängt. Dazu übergeben wir die Pixel-Byte-Adresse an das neue Modul und nehmen die gewünschten Pixeldaten für diese Adresse entgegen:

```
output [9:0] pixelAddress,
input [7:0] pixelData
```

Um die Pixeladresse zu ermitteln, können wir die bereits vorhandene `pixelCounter`-Variable verwenden:

```
assign pixelAddress = pixelCounter;
```