Studienarbeit

Entwurf eines ASICs zur Mustererkennung auf Basis von FPGAs

Frank Bohnsack

Universität Hamburg
Fachbereich Informatik

Januar 1995

Inhaltsverzeichnis

1. Einleitung

Die Bildverarbeitung findet in vielen Bereichen ihre Anwendungen. Ziel dieser Arbeit ist der Entwurf eines Prototypen auf Basis von *FPGAs* (**F***ield* **P***rogrammable* **G***ate* **A***rray*) für die Mustererkennung in Binärbildern. Zu den vielen Aufgaben der Bildverarbeitung gehört das Auffinden von Objekten in einem Bild. Die Objekte kann man auch als Muster auffassen. Das Muster wird dann an allen Positionen des Bildes gesucht. Sofern das Muster gefunden wird, wird dieses dem Benutzer angezeigt. Diese spezielle Problem soll im folgenden mit den *FLEX 8000*-Bausteinen von Altera realisiert werden, wobei die folgenden Anforderungen berücksichtigt werden.

Randbedingungen

Da es sich hier um einen Prototypen handeln soll, werden in dieser Arbeit nur Vergleiche von Binärbildern realisiert. Außerdem wird die Bildgröße, und damit auch die Mustergröße, auf 16-512 Pixel pro Zeile, in Schritten von 16 Pixeln, und 1-512 Pixel pro Spalte, in Schritten von 1 Pixel, begrenzt. Über ein zusätzliches Pixel-Array, genannt Maske, das die gleichen Abmessungen wie das Muster hat, können einzelne Pixel für den Vergleich ausgeblendet werden. Dadurch wird gewährleistet, daß der Benutzer nach beliebigen Formen suchen kann.
Zusätzlich hat der Benutzer die Möglichkeit, einen Schwellwert für die Anzahl der zulässigen Abweichungen zwischen nicht ausmarkiertem Muster und dem Bildausschnitt anzugeben. Der Schwellwert gibt an, ab welcher Anzahl von Abweichungen das Muster als nicht erkannt eingestuft werden soll. Dieser ermöglicht ferner auch dann ein Muster zu erkennen, wenn Störungen bei der Bildaufnahme (wenige) fehlerhafte Pixel in dem Bild zur Folge haben.
Das vom Benutzer spezifizierte Muster wird an allen Positionen des Bildes gesucht, wo es **vollständig** plaziert werden könnte. Das Ergebnis des Vergleichs wird in einer Bit-Matrix abgespeichert, die die gleichen Abmessungen wie das Bild hat. Dabei zeigt ein gesetztes Bit an, daß der Bildausschnitt von der Größe des Musters, dessen linke obere Ecke an der dem Bit entsprechenden Bildposition liegt, mit dem Muster übereinstimmt.
Das zu entwerfende ASIC, genannt EMMA (**E***rster* **M***ultichip* **M***uster*-**A***uffinder)* soll folgenden Ablauf für die Durchführung der Mustererkennung ermöglichen:

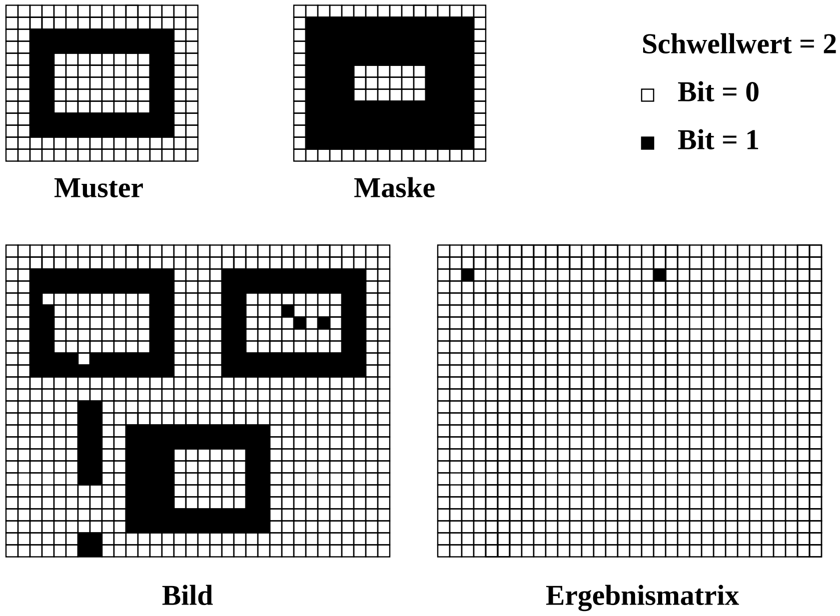

Abbildung 1.1: Vergleich von Muster und Maske mit einem Bild. Das Ergebnis des kompletten Vergleichs wird in der Ergebnismatrix abgelegt

1. Übertragen der für den Vergleich benötigten Daten und Parameter.
2. Starten des Algorithmus' mit Hilfe eines *RUN*-Befehls. Während EMMA die Mustererkennung durchführt, erfolgt keine weitere Ansteuerung.
3. Nachdem EMMA den Vergleich durchgeführt hat, kann das Ergebnis ausgelesen werden.

Der folgende Text gliedert sich wie folgt:
Das zweite Kapitel stellt FPGAs allgemein vor. Dabei wird insbesondere auf die hier verwendeten *FLEX 8000*-Bausteine von Altera eingegangen und deren interne Struktur erläutert. Das Kapitel 3 stellt den Vergleichsalgorithmus vor, dessen Hardware-Realisierung in Kapitel 4 folgt. Hier werden das Operationswerk und das Steuerwerk von EMMA erläutert. Der Entwurf erfolgte in der Hardwarebeschreibungssprache AHDL [Alt93], die Funktions- und Timinganalyse mit den Konvertierungstools für Synopsys [Syn93] von Herrn Klindworth [Kli94]. Um verschiedene Wortbreiten zu unterstützen, ist das Operationswerk so strukturiert, daß es kaskadierbar ist. In Kapitel 4 wird eine kaskadierte Schaltung mit einem 16-Bit Datenbus realisiert. Dabei wird insbesondere auf die Anschlüsse der hierzu notwendigen 3 *FLEX 8000*-Bausteine eingegangen.

2. FPGA's

2.1 Was ist ein FPGA?

Ein **F***ield* **P***rogrammable* **G***ate* **A***rray* ist ein programmierbarer digitaler Baustein. Es besteht aus einer Matrix von einzelnen Elementen, sogenannten Logikblöcken, die in einer flexiblen Weise miteinander verbunden werden können. Die Logikblöcke und die Verbindungen zwischen den Elementen werden durch den Benutzer programmiert, ähnlich wie in PAL's (**P***rogammable* **A***rray* **L***ogic*). Ein *FPGA* besteht somit aus einem zweidimensionalen Feld von Logikblöcken, die über allgemeine Verbindungsressourcen miteinander verbunden werden können. Eine Verbindung setzt sich aus einzelnen Segmenten verschiedener Länge zusammen. Abgesehen von einem *Continuous Interconnect*, bei dem die Segmente immer die gleiche Länge haben. Über programmierbare Schalter werden die einzelnen Leitungssegmente miteinander und mit den Logikblöcken verbunden. Logische Schaltungen werden somit in die Logikblöcke partitioniert und dann die erforderlichen Verbindungen via programmierbarer Schalter realisiert.
Um die Implementierung einer großen Anzahl von Schaltungen zu ermöglichen, muß das *FPGA* möglichst flexibel sein. Daher gibt es unterschiedliche Realisierungen der Logikblöcke und der Verbindungsressourcen, sowohl bezüglich der Komplexität als auch der Flexibilität.

2.1.1 Logikblöcke

Die Struktur der Logikblöcke kann auf viele unterschiedliche Arten realisiert sein. So reichen die Architekturen von einem NAND-Gatter mit zwei Eingängen bis zu komplexen Strukturen, wie Multiplexern oder *Lookup-Tabellen* [BFR90]. Die meisten Logikblöcke enthalten zusätzlich ein Flip-Flop, um Ausgangswerte zu speichern. Wichtige Kriterien für die Wahl der Architektur der Logikblöcke ist die benötigte Chipfläche und die zu ermöglichende Geschwindigkeit.

2.1.2 Verbindungsressourcen

Diese bestehen aus den Leitungssegmenten und den programmierbaren Schaltern. Letztere können auf verschiedene Arten realisiert werden, zum Beispiel als Pass-Transistoren, gesteuert durch SRAM Zellen [Xil92], Anti-fuses [ACT92], EEPROM-Transistoren (**E***lectrically* **E***rasable* **R***ead*-**O***nly* **M***emory*) und EPROM-Transistoren (**E***rasable* **P***rogrammable* **R***ead*-**O***nly* **M***emory*) [AMD92] [Alt93]. Ähnlich wie bei den Logikblöcken existieren auch viele verschiedene Möglichkeiten, die Verbindungsressourcen zu realisieren. Einige FPGA's bieten eine große Anzahl von Einzelverbindungen zwischen den Blöcken an, andere weniger, dafür aber komplexere.

2.1.3 Wirtschaftlichkeit von FPGA's

FPGA's können effizient in vielen Anwendungen eingesetzt werden. Sie haben gegenüber den MPGAs (**M***ask*-**P***rogrammable* **G***ate* **A***rray*) zwei wesentliche Vorteile: Sie haben geringere Prototypen-Kosten und minimale Produktionszeiten. Demgegenüber stehen aber auch zwei Nachteile: eine geringere Geschwindigkeit und eine geringere Logik-Dichte. Die geringere Geschwindigkeit wird durch die programmierbaren Schalter verursacht, die Widerstände und Kapazitäten haben. Die Logik-Dichte wird reduziert durch den relativ großen Platzbedarf der programmierbaren Schalter, im Gegensatz zu den Metallverbindungen bei den MPGAs. Typischerweise hat ein FPGA eine um den Faktor 8 bis 12 geringere Dichte als ein MPGA. Das hat natürlich Auswirkungen auf die Kosten, weil weniger FPGAs pro Wafer produziert werden können.

2.1.4 Anwendungen von FPGAs

FPGAs können für viele Anwendungen verwendet werden, die mit MPGAs, PLDs oder SSI-Chips (**S***mall* **S***cale* **I***ntegration*) realisiert worden sind. Im folgenden werden einige Kategorien solcher Designs vorgestellt:

- *Application-Specific Integrated Circuit (ASICs)*
  Die digitale Logik wird komplett in einem FPGA implementiert. Beispiele: 1 Megabit FIFO Controller, Drucker Controller [HiF90].

- *Glue Logic*
  Große Schaltungen enthalten häufig eine Anzahl von SSI Chips. Diese können in vielen Fällen durch FPGAs ersetzt werden, was dann zu einer Reduzierung der Platinenfläche führt und eine geringere Leistungsaufnahme zur Folge hat. Üblicherweise wird die Logik von Speichern mit PALs realisiert. Wenn die Geschwindigkeit unkritisch ist (PALs sind schneller als die meisten FPGAs), dann können solche Bausteine vorteilhafter in FPGAs implementiert werden. Denn in einem FPGA kann die gesamte Logik , die sonst zehn bis zwanzig PALs umfaßt, untergebracht werden. In der Zukunft wird dieser Faktor noch dramatisch ansteigen.

- *Prototypen-Designs*
  FPGAs sind ideal geeignet, um Prototypen zu erstellen. Die geringen Kosten der Implementierung und die kurze Zeit, die benötigt wird, um ein gegebenes Design physikalisch zu realisieren, sind enorme Vorteile gegenüber herkömmlichen Verfahren. Änderungen können dabei leicht und kostengünstig durchgeführt werden.

- *FPGA-basierende Computer*
  Eine neue Klasse von Computern wurde möglich mit der Einführung der *incircuit reconfigurable* FPGAs, Bausteine, die während des Betriebs programmierbar sind. Beispiele finden sich in [AtA94]. Diese Maschinen bestehen aus einem Board solcher FPGAs. Die Idee dabei ist, ein Softwareprogramm mittels high-level und Technologie abhängige Logik-Synthese Techniken in eine Hardwarestruktur zu übersetzen. Die Hardwarestruktur wird dann auf die FPGAs abgebildet. Dieses Verfahren hat folgende Vorteile:

  - Es müssen keine Instruktionen wie bei den herkömmlichen Mikroprozessoren geholt werden. Die Hardware verkörpert direkt die Instruktionen. Dadurch ergibt sich ein Geschwindigkeitszuwachs um einen Faktor bis zu 100.
  - Eine Hardwarestruktur kann hohe Grade der Parallelität unterstützen, was wiederum die Geschwindigkeit steigert.

- *On-Site Rekonfiguration der Hardware*
  Oft ist es wünschenswert, die Struktur einer Maschine, die bereits im Betrieb ist, zu ändern. Dieses ist mit re-programmable FPGAs möglich.

2.1.5 Entwurfsprozeß

Der Entwurfsprozeß läßt sich in sechs Schritte aufgliedern, die in Abbildung 2.1 dargestellt sind.

- *Initial Design Entry*
  Für die Schaltungseingabe stehen drei verschiedene Eingabeformen zur Verfügung:

  - Schematic Entry (graphische Schaltplan-Eingabe) der zu realisierenden Funktion
  - VHDL oder andere Hardware-Beschreibungssprachen
  - boolesche Ausdrücke und Gleichungen.

  Unabhängig von der Eingabeform wird die Beschreibung in der Regel von einem Compiler in ein internes Datenformat, z.B. boolesche Ausdrücke, konvertiert.

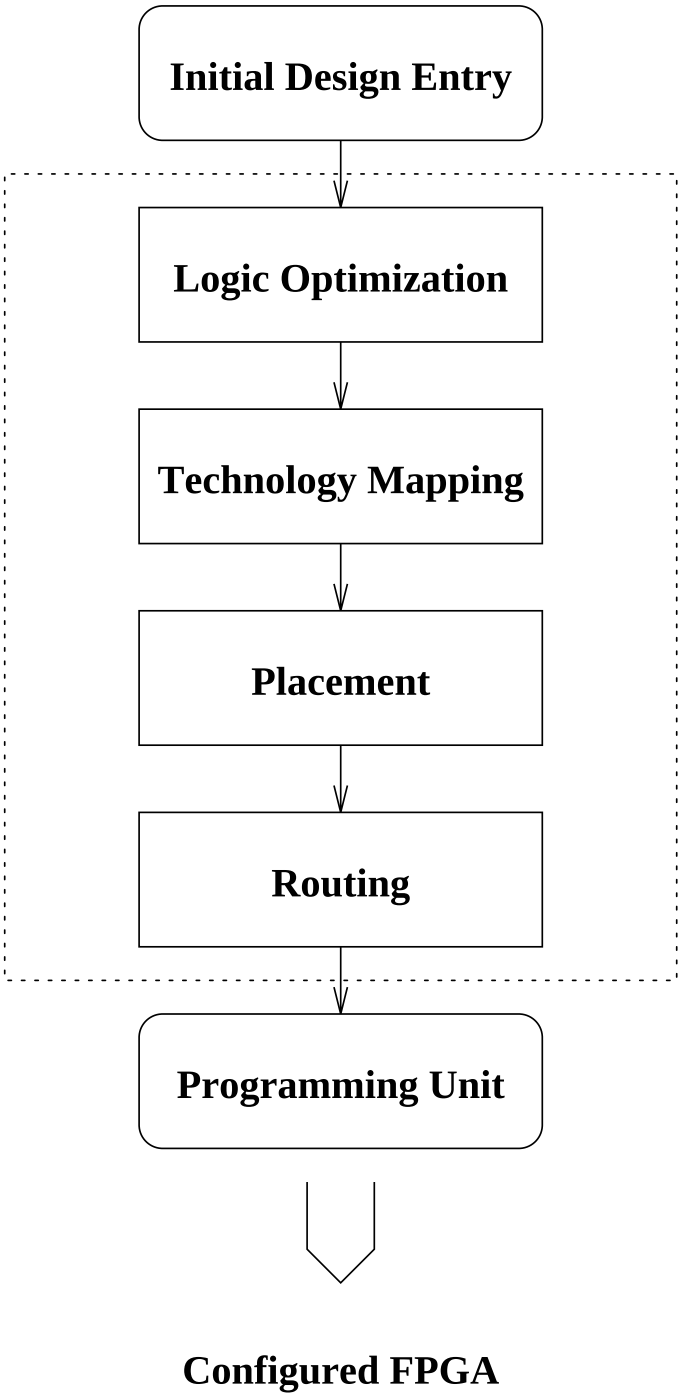

Abbildung 2.1: Ein typisches CAD-System für FPGAs

- *Logik-Optimierung*
  Hier werden die booleschen Ausdrücke manipuliert, um Platzbedarf oder Geschwindigkeit zu optimieren.

- *Technology Mapping*
  Die booleschen Ausdrücke der Logik-Optimierung werden auf die Logikblöcke des FPGA's abgebildet. Der Mapper versucht die Anzahl der Blöcke, die benötigt werden, zu minimieren (Area Optimization), oder er versucht, die Stufenzahl in zeitkritischen Pfaden zu minimieren (Delay Optimization). Gewöhnlich kann die relative Gewichtung dieser beiden Optimierungskriterien über entsprechende Optionen variiert werden.

- *Placement*
  Die Blöcke des Designs werden in dem FPGA plaziert. Typischerweise versuchen die Placement-Algorithmen, die Länge der Verbindungen möglichst klein zu halten. Die Problematik der Plazierung bei FPGAs ist dem beim VLSI-Entwurf (**V***ery* **L***arge* **S***cale* **I***ntegration*) [Kol89] ähnlich.

- *Routing*
  Beim Routing werden die Verbindungsleitungen bzw. die Leitungssegmente zugewiesen und die programmierbaren Schalter ausgewählt. Dabei muß natürlich sichergestellt sein, daß 100% der geforderten Verbindungen realisiert werden, anderenfalls paßt das Design nicht in ein FPGA hinein. Auch hier sind die Verfahren dem des Standardzellentwurfs ähnlich. Komplizierter werden sie allerdings durch die Tatsache, daß der Umfang und die räumliche Lage der Verbindungsressourcen (Leitungssegmente und Schalter) fest vorgegeben sind. Bei den *FLEX 8000*-Bausteinen tritt diese Problematik nicht auf, weil diese so strukturiert sind, daß alle Verbindungen realisiert werden können.

- *Programming Unit*
  Die Programming Unit konfiguriert das FPGA Chip. Dieser Prozeß kann von einigen Sekunden bis zu einigen Minuten dauern, abhängig davon, welcher Typ von FPGAs verwendet wird.

Bis auf den ersten Punkt können alle Phasen weitgehend automatisiert werden. Bei der hier verwendeten Entwurfssoftware MAX+plusII [Alt93] können alle Phasen über Einstellungen verschiedener Optionen beeinflußt werden. Dadurch ist dem Entwickler immer die Möglichkeit gegeben, die technische Realisierung der Funktionen seinen Bedürfnissen anzupassen.

2.2 FLEX 8000-Bausteine

2.2.1 Allgemeine Beschreibung

Die Altera **F***lexible* **L***ogic* **E***lement Matri***X** (FLEX) Familie verbindet die Vorteile von EPLDs (Erasable Programmable Logic Devices) und **L***ook*-**U***p*-**T***abellen* (LUT) basierten FPGAs. Die feingegliederte Architektur und die hohe Registeranzahl von FPGAs werden kombiniert mit geringen Zeitverzögerungen der kontinuierlichen Verbindungen in EPLDs. Die Logik wird bei *FLEX*-Bausteinen in einer kompakten LUT, welche 4 Eingänge besitzt, und einem programmierbaren Register implementiert. Die Logik und die interne Verdrahtung werden bei *FLEX*-Bausteinen mit Passtransistoren, die von programmierbaren SRAM Zellen angesteuert werden, realisiert. Daher können diese Bausteine *in-circuit* konfiguriert und auch rekonfiguriert werden. Dies kann sowohl von einem EPROM als auch von einem System-RAM erfolgen. Da die Rekonfiguration (Zurücksetzen des Bausteins auf seinen Anfangszustand und Laden von neuen Konfigurationsdaten) weniger als 100 ms beträgt, können Änderungen während des Systembetriebs durchgeführt werden.

2.2.2 Funktionale Beschreibung

Der *FLEX*-Baustein besteht aus einer großen Anzahl von sogenannten **L***ogic* **E***lements* (LEs). Diese sind wiederum in Gruppen zu je 8 zusammengefaßt, genannt **L***ogic* **A***rray* **B***locks* (LAB). Jeder *LAB* ist ein eigenständiger Block mit gemeinsamen Eingängen, Verbindungen und Steuersignalen. Die Abbildung 2.2 zeigt die *FLEX* Architektur.
Die LABs sind wiederum in Zeilen und Spalten angeordnet. Die konfigurierbaren I/O Elemente (IOE's) befinden sich am Ende der Zeilen und Spalten. Alle IOE's enthalten einen bidirektionalen I/O-Buffer und ein Flip-Flop, das entweder als Eingangs- oder Ausgangsregister genutzt werden kann.
Die Verbindungen zu und von den Pins sowie zwischen den LABs erfolgen über das sogenannte *FastTrack Interconnect*, einer Reihe von durchgehenden vertikalen und horizontalen Verbindungsleitungen (siehe Abbildung 2.2). Am Ende eines jeden *Row* oder *Column Interconnect* befinden sich IOEs.

Logic Element (LE)

Das LE ist die kleinste funktionale Einheit in der FLEX Architektur. Jedes LE enthält eine LUT mit vier Eingängen, ein programmierbares Flip-Flop, eine *Carry Chain* und eine *Cascade Chain*. Die Abbildung 2.3 zeigt die Struktur des LE's. Mit der LUT können alle Funktionen mit bis zu vier Eingangsvariablen generiert werden. Das Flip-Flop ist ein D-Flip-Flop und kann durch entsprechende Beschaltung unter Einbeziehung der LUT als T, JK oder SR-Flip-Flop konfiguriert werden.

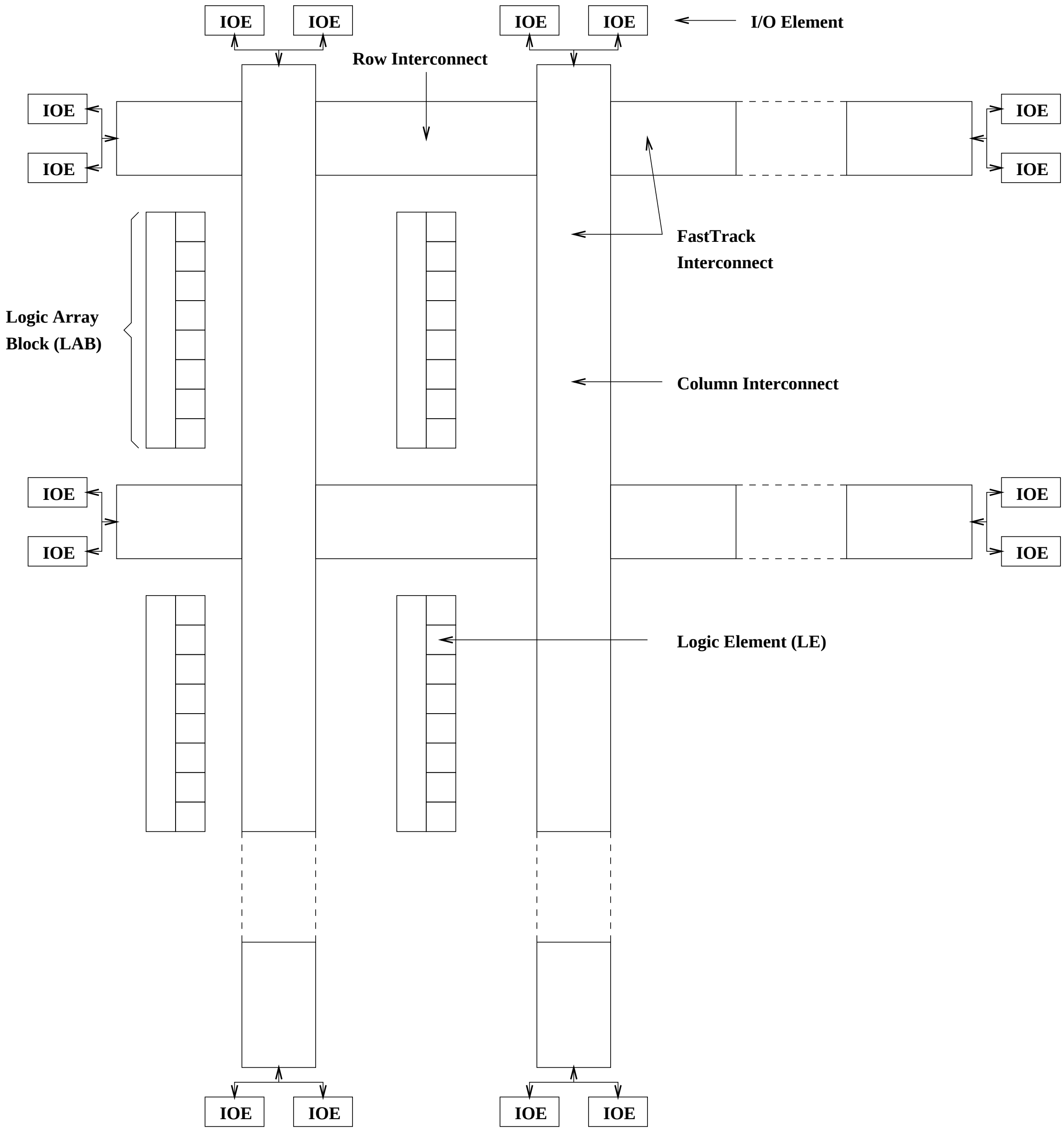

Abbildung 2.2: Blockdiagramm eines FLEX-Bausteins

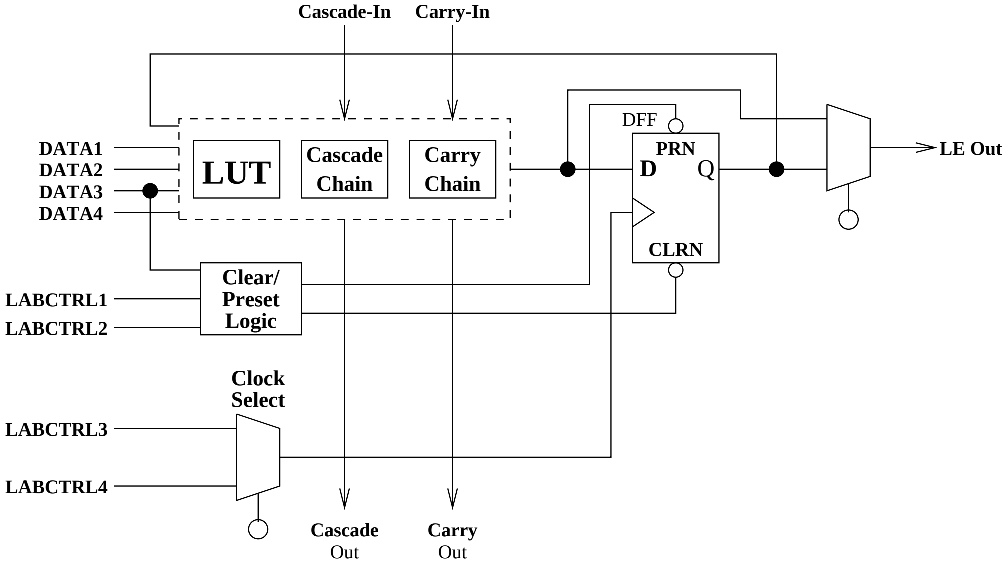

Abbildung 2.3: FLEX Logic Element (LE)

Carry Chain

Die *Carry Chain* ermöglicht eine schnelle Weiterleitung von Überträgen (weniger als 1 ns) zwischen den benachbarten LEs eines LABs. Dadurch werden schnelle ripple-Zähler und Addierer beliebiger Wortbreite ermöglicht.

Cascade Chain

Die *Cascade Chain* ermöglicht die Realisierung von Funktionen mit einer großen Anzahl von Eingängen. Dabei erstellen die LUTs benachbarter LEs Teile der zu realisierenden Funktion. Die *Cascade Chain* verbindet dann die Teilergebnisse miteinander. Die Verzögerung pro benötigtem LE beträgt annähernd 1 ns. Mit der *Cascade Chain* kann ein logisches AND oder OR der Ausgänge der beteiligten LEs realisiert werden.

FastTrack Interconnect

Der *FastTrack Interconnect* (FTI) realisiert die globale Verbindungen zwischen den I/O-Pins und den LABs sowie den LABs untereinander. Das FTI ist ein *continuous interconnect*, so daß die Verzögerungszeiten nur von der Plazierung abhängig sind. Für Verbindungen in der gleichen Zeile beträgt die Verzögerung 6 ns, in unterschiedlichen 9 ns. Die Verzögerungszeiten sind damit fest, anders als in FPGAs mit *Segmented Interconnect*.

3. Vergleichsalgorithmus

In diesem Kapitel wird der in dieser Arbeit realisierte Vergleichsalgorithmus vorgestellt. Im wesentlichen wird das Muster komplett an einer Position des Bildes verglichen. Das Resultat des Vergleichs wird dann in einem Ergebniswort gespeichert. Diese Vorgehensweise wird an allen Wortpositionen des Bildes wiederholt. Um den Hardwareaufwand bei den Spalten- und Zeilensprüngen möglichst gering zu halten, wird der Vergleich jeweils in der unteren rechten Ecke des Musters begonnen. Das Ergebnis wird dann, wie in der Einleitung bereits erwähnt,

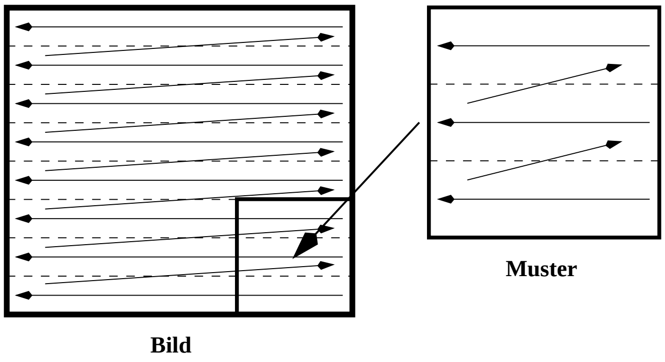

Abbildung 3.1: Vergleichsrichtung im Muster und des Musters im Bild

an der oberen linken Ecke markiert. Dabei wird eine Bildzeile von rechts nach links abgearbeitet. Dieses wird für jede Zeile von unten nach oben wiederholt. Das Verfahren ist für eine Hardwarestruktur günstig, weil das Ende eines Vergleichs lediglich einen Vergleich mit 0 erfordert. Allerdings ist eine Korrektur der Breiten und Höhen erforderlich. Die Werte müssen dekrementiert werden, da sie sonst außerhalb des Bildes, Musters bzw. der Maske lägen (korr_bildbreite, korr_bildhoehe, korr_musterbreite, korr_musterhoehe). Muster und Maske haben die gleichen Breiten- und Höhenwerte, so daß das einmalige Speichern der Breiten- und Höhenwerte ausreicht. Aufgrund der gleichen Werte ist auch die Adressierung von Muster und Maske identisch. Lediglich das Steuerwerk muß später eine Unterscheidung vornehmen, damit die richtigen Daten in die entsprechenden Pufferregister geladen werden. Um einen Vergleich an jeder Position des Bildes, an der das Muster vollständig erkannt werden kann, durchführen zu können, muß die momentane Bildposition zwischengespeichert werden (aktuelle_bildbreite, aktuelle_bildhoehe). Zusätzlich wird noch die Position im Muster-Bild-Vergleich gespeichert (positionsbreite, positionshoehe). Diese Position gibt an, welche Bytes für die nächsten Vergleichsoperationen geladen werden müssen. Damit ergeben sich folgende Initialisierungen:

```
korr_bildbreite     := bildbreite - 1;
korr_bildhoehe      := bildhoehe - 1;
korr_musterbreite   := musterbreite - 1;
korr_musterhoehe    := musterhoehe - 1;
aktuelle_bildbreite := korr_bildbreite;
aktuelle_bildhoehe  := korr_bildhoehe;
```

Der Vergleich basiert auf zwei inneren FOR-Schleifen, die das Muster abarbeiten, und zwei äußeren REPEAT-Schleifen, die dafür sorgen, daß der Vergleich an jeder Bildposition durchgeführt wird. Im Detail sieht der Algorithmus wie folgt aus:

```
REPEAT
  REPEAT
    positionshoehe := aktuelle_bildhoehe;
    FOR i = korr_musterhoehe TO 0 DO
      positionsbreite := aktuelle_bildbreite;
      FOR j = korr_musterbreite TO 0 DO
        vergleiche Muster und Bild;
        positionsbreite := positionsbreite - 1;
      END FOR;
      positionshoehe := positionshoehe - 1;
    END FOR;
    schreibe Ergebnis an Position positionsbreite, positionshoehe
    aktuelle_bildbreite := korr_bildbreite - 1;
  UNTIL (positionsbreite = 0);
  aktuelle_bildhoehe := korr_bildhoehe := korr_bildhoehe - 1;
UNTIL (positionshoehe := 0);
```

Nach Verlassen der äußeren FOR-Schleife liegt in der Variablen (*positionsbreite, positionshoehe*) die Adresse der Bildposition vor, an der das Vergleichsergebnis abzuspeichern ist.

4. Realisierung

4.1 Konzept

Die Realisierung des EMMA-Chips basiert auf einem bitorientierten, 8-fach parallelen Vergleich. Es wird jeweils ein Masken- und Musterbit mit allen Bits eines Bildbytes gleichzeitig verglichen. Der Vergleich liefert ein Abweichungsbyte, in dem durch ein gesetztes Bit angezeigt wird, ob das Musterbit an dieser Position mit dem Bildbit übereinstimmt. Durch diese Methode wird gewährleistet, daß jede Pixelposition nach Auftreten des Musters überprüft wird. Durch die große Komplexität der Aufgabe und der demgegenüber relativ geringen Anzahl von Flipflops bot es sich an, das Steuerwerk inklusive der Adreßberechnung auf der einen und das Operationswerk auf der anderen Seite in zwei getrennten *FLEX*-Bausteinen zu realisieren, so daß die Anzahl der Datenleitungen zwischen den beiden *FLEX*-Bausteinen möglichst gering ist. Dabei ist das Operationswerk so konzipiert, daß die Struktur kaskadierbar ist, um Vergleiche auch bei größerer Wortbreite (8, 16, 32 und 64 Bit) zu ermöglichen. Die Konfiguration des Steuer- und des Operationswerkes beschränkt sich auf die Eingabe der Wortbreite des verwendeten Datenbusses und die Bereitstellung einer genügenden Anzahl von Operationswerken.

4.2 Operationswerk

Das Operationswerk besteht im wesentlichen aus neun Komponenten:

- einem Datenpuffer zum Zwischenspeichern der Bild-, Muster- und Maskendaten

- acht baugleichen Vergleichern, die unter Berücksichtigung eines Schwellwertes in einem Ergebnisbyte (1 Bit pro Vergleicher) anzeigen, ob das Muster an der entsprechenden Position im Bild noch erkannt werden kann oder nicht.

Im folgenden werden die Komponenten genauer vorgestellt.

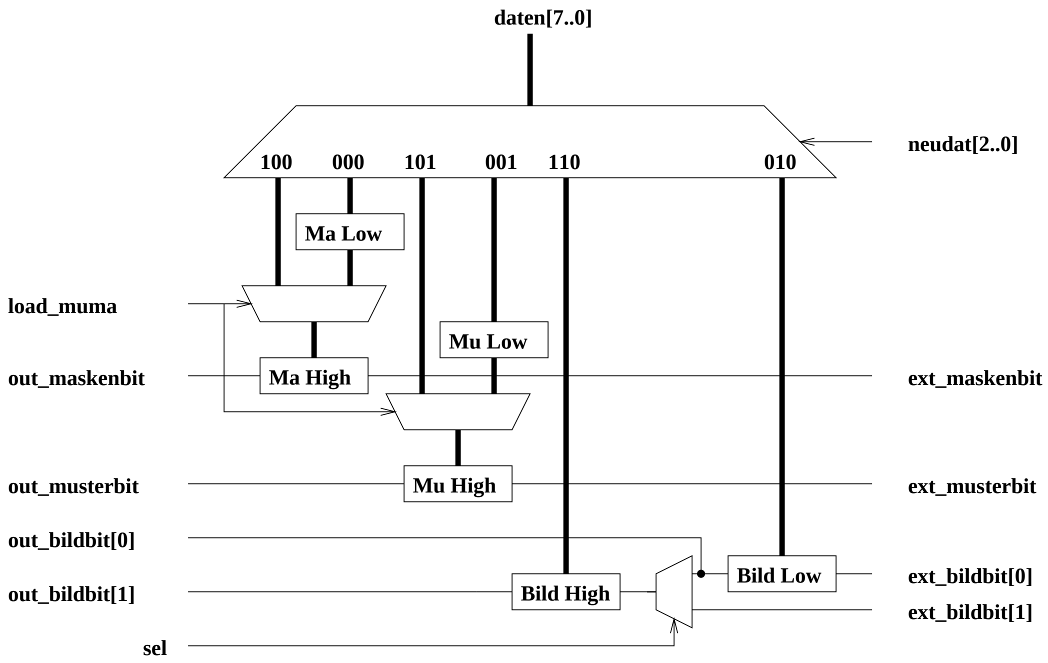

Abbildung 4.1: Blockdiagramm des Datenpuffers

4.2.1 Datenpuffer

Im Datenpuffer werden die Daten für den nächsten Vergleich zwischengespeichert. Dabei sind sowohl für das Bild als auch für das Muster und die Maske 16 Bit vorgesehen. Da zu dem Baustein nur eine 8-Bit Datenleitung führt, müssen die Daten über einen Multiplexer in den entsprechenden Puffer geladen werden. Dies erfolgt über die Steuerleitungen *neudat[2..0]*. Dabei werden die verschiedenen Datenpuffer folgendermaßen angesteuert:

neudat = "000" : Daten werden in den maskenpuffer[7..0] geladen
neudat = "100" : Daten werden in den maskenpuffer[15..8] geladen
neudat = "001" : Daten werden in den musterpuffer[7..0] geladen
neudat = "101" : Daten werden in den musterpuffer[15..8] geladen
neudat = "010" : Daten werden in den bildpuffer[7..0] geladen
neudat = "110" : Daten werden in den bildpuffer[15..8] geladen
neudat = "111" : Es werden keine Daten geladen

Das erste Bit signalisiert, ob die Daten in das obere (neudat[2] = 1) oder untere Byte (neudat[2] = 0) geladen werden sollen. Die übrigen Bits kodieren den Typ der Daten (Maske = 00, Muster = 01 und Bild = 10). Nun muß nur noch gewährleistet sein, daß die richtigen Daten zum Vergleich anliegen. Dies erfolgt

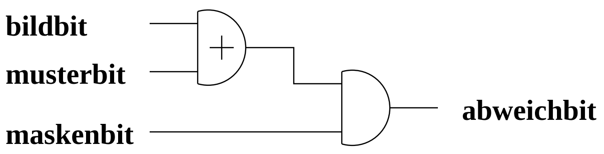

Abbildung 4.2: Logische Struktur des Bitvergleichers

über zwei zusätzliche Steuersignale *shift_bild* und *shift_muma*. Sie dienen dazu, die Pufferregister für das Bild bzw. für Muster und Maske um ein Bit nach links zu verschieben, so daß das nächste zu vergleichende Bit an die Vergleicher gesendet wird. Das Steuersignal *sel* dient dagegen zur Auswahl der nachzushiftenden Bild-Bits. Dies ist bei Verwendung von zwei FPGAs notwendig, weil anderenfalls nicht die richtigen Daten zum Vergleich bereitgestellt werden (vgl. Kapitel 5). Ist *sel* gesetzt, so kommt das letzte Bit aus dem HIgh-Byte Puffer eines weiteren Operationswerkes. Anderenfalls kommt das letzte Bit aus dem unteren Pufferregister. Dieses Schema verdeutlicht Abbildung 4.1.

4.2.2 Vergleicher

Der Vergleicher besteht aus zwei Teilen, nämlich dem eigentlichen *Bitvergleicher* und einem *Dekrementierer* für die Berücksichtigung des Schwellwertes.

Bitvergleicher

Der *Bitvergleicher* beinhaltet eine sehr einfache Logik. Er gibt eine *1* aus, wenn das Muster und das Bild an der aktuellen Vergleichsposition nicht übereinstimmen. Also erfüllt die folgende boolesche Gleichung diese Anforderung:

$$\text{abweichbit} = \text{bildbit XOR musterbit}$$

Zusätzlich soll aber die Möglichkeit bestehen, über eine Maske einzelne Bits beim Vergleich auszublenden. Das heißt, daß die obige Logik noch nicht ganz vollständig ist. Das *abweichbit* wird nur weitergeleitet, wenn das Maskenbit gesetzt ist, also dieses Bit für den Vergleich relevant ist. Eine *UND*-Verknüpfung genügt, um das Maskenbit beim Vergleich zu berücksichtigen. Dadurch ergibt sich für den *Bitvergleicher* folgende logische Struktur:

$$\text{abweichbit} = (\text{bildbit XOR musterbit}) \text{ AND maskenbit}$$

Die Abbildung 4.2 zeigt das zugehörige Schaltnetz.

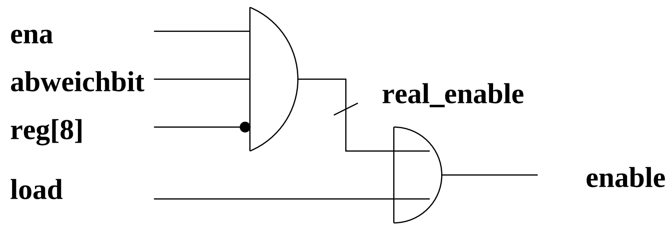

Abbildung 4.3: Enable-Signal des register[8..0]

Dekrementierer

Der Dekrementierer dient der Kontrolle von zulässigen Abweichungen zwischen Bild und Muster, wie es durch die Angabe des Schwellwertes (schwellwert[7..0]) spezifiziert worden ist. Dazu wird zunächst der Schwellwert in ein Register, genannt *register[8..0]*, geladen, wobei *register[8]* auf 0 gesetzt wird. Dabei dient das *register[8]* zur Speicherung des *carrys* der höchsten Stelle. Das *register[8..0]* wird nur dann dekrementiert, wenn das *abweichbit* aus dem Bitvergleicher gesetzt ist. Ansonsten wird der alte Wert beibehalten. Zusätzlich kann über das Steuersignal *load* ein neuer Schwellwert geladen werden, was bei einem neuen Vergleich natürlich notwendig ist. Daher wird über einen Multiplexer gesteuert, ob ein neuer Schwellwert geladen (load = 1), der alte Wert beibehalten (real_enable = 0) oder der dekrementierte Wert (real_enable = 1) in das *register[8..0]* übernommen werden soll. Problematisch ist jetzt nur noch der Fall des Überlaufens des Registers. Wenn ein carry an der höchsten Stelle auftritt, dann wird dieses ja in *register[8]* gespeichert. Alle weiteren Vergleiche sind dann für das *ergbit* nicht mehr relevant. Deshalb muß gewährleistet sein, daß das *register[8..0]* nicht weiter dekrementiert wird, denn wenn dieses geschieht, dann wäre nach dem nächsten Vergleich das höchste carry wieder 0 und das Muster könnte wieder erkannt werden. Um dieses zu verhindern, muß das carry in das enable-Signal des *registers[8..0]* einfließen. Deshalb muß das *register[8..0]* in folgenden Fällen aktiviert sein:

- wenn ein neuer Schwellwert geladen werden soll (*load = 1*)
- wenn der dekrementierte Wert übernommen werden soll.

Dadurch ergibt sich für das enable-Signal folgende logische Gleichung:

$$\text{enable} = \text{load OR (ena AND abweichbit AND NOT reg[8])}$$

Diese Gleichung verdeutlicht Abbildung 4.3. Durch Zusammensetzen der einzelnen Bauteile ergibt sich dann die Gesamtstruktur des Vergleichers. Die Abbildung 4.4 zeigt diese Struktur. Dabei kommen *bildbit*, *musterbit* und *maskenbit* vom Datenpuffer, der *schwellwert[7..0]* und die Steuersignale *load*, *ena*, *clr* und *clk* vom

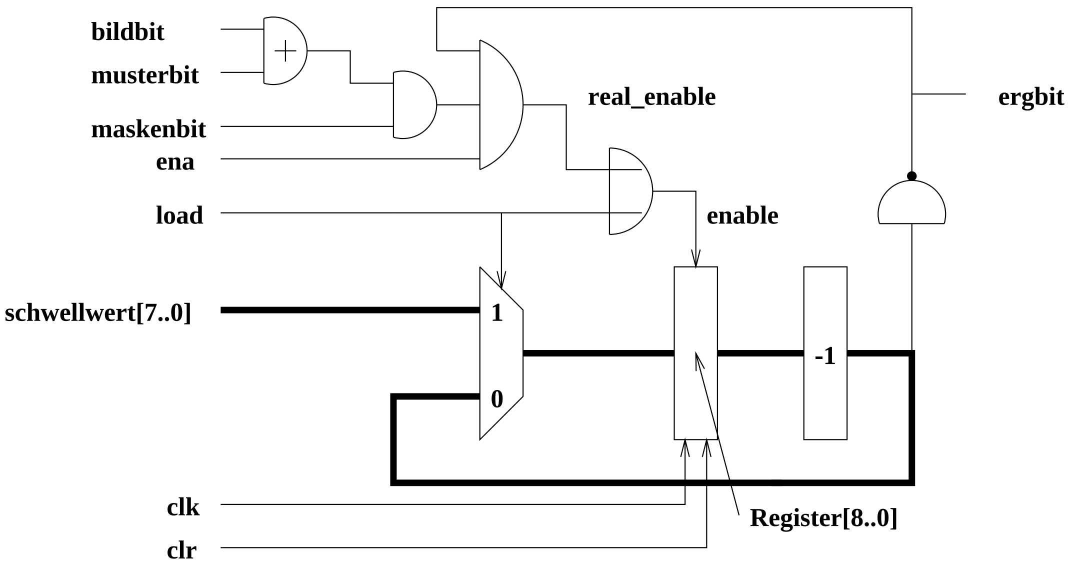

Abbildung 4.4: Struktur des Vergleichers

Steuerwerk. Die *ergbits* der 8 Vergleicher werden nach Vergleich aller Musterbits zu einem *ergbyte* zusammengefügt und über den Datenbus in die Ergebnismatrix geschrieben.

4.3 Steuerwerk

Das Steuerwerk besteht nicht nur aus einem endlichen Automaten, der die benötigten Steuersignale für das Operationswerk generiert, sondern zusätzlich befindet sich in diesem Baustein noch eine Adreßberechnungseinheit (ABE).

4.3.1 Adreßberechnung

Die ABE besteht aus vier Registern, sechs Dekrementierern, genannt *subtr2*, und einem Multiplexer zur Auswahl der Adresse. In den Registern werden die Werte für die Breiten und Höhen gespeichert. Auf die Werte wird bei der Adreßberechnung zugegriffen. Die sechs Dekrementierer dienen zur Adreßberechnung für die verschieden Bytes, die zum Vergleich geladen werden müssen, und zum Speichern der Musterposition im Bild, wie es im Vergleichsalgorithmus (siehe Kapitel 3) beschrieben ist.

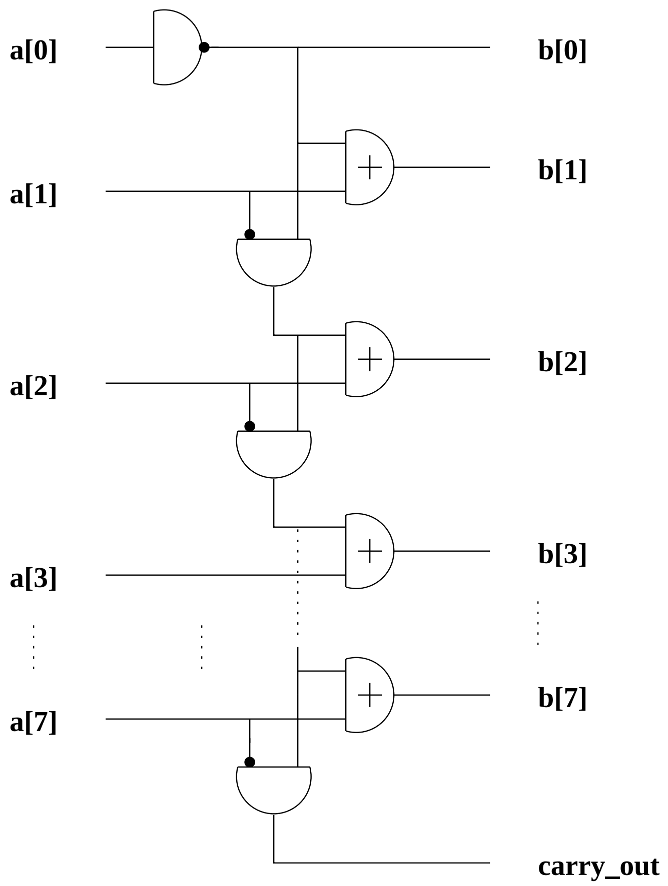

Abbildung 4.5: Struktur des Dekrementierers mit Verwendung der *Carry Chain*

Dekrementierer für die Adreßberechnung

Die Adressen sollten möglichst schnell berechnet werden, weil viele Adressen berechnet werden müssen. Daher bietet es sich an, bei dem hier zu bildenden Dekrementierer die *Carry Chain* des *FLEX 8000*-Bausteins zu verwenden. Die Abbildung 4.5 zeigt dieses. Dabei enspricht die vertikale Verbindung der *Carry Chain*. Zusätzlich sind für die Adreßberechnung noch zwei Multiplexer und ein Register zum Speichern des Wertes notwendig. Der erste Multiplexer wählt zwischen dem dekrementierten (sub_1 = 1) und dem nicht dekrementierten Wert (sub_1 = 0) aus und leitet diesen weiter an den zweiten Multiplexer. Dieser speichert dann entweder den Wert von dem ersten Multiplexer (load = 0) oder einen neuen Wert in das Register. Das Nachladen ist zum Beispiel bei einem Zeilenumbruch notwendig. Das Register hat nicht die Wortbreite 8, sondern 9, damit ein Überlaufen des Zählers erkannt wird. Das sogenannte *zd*-Signal (zero detect) teilt dem endlichen Automaten mit, daß neue Adressen angelegt werden müssen. Die Gesamtstruktur

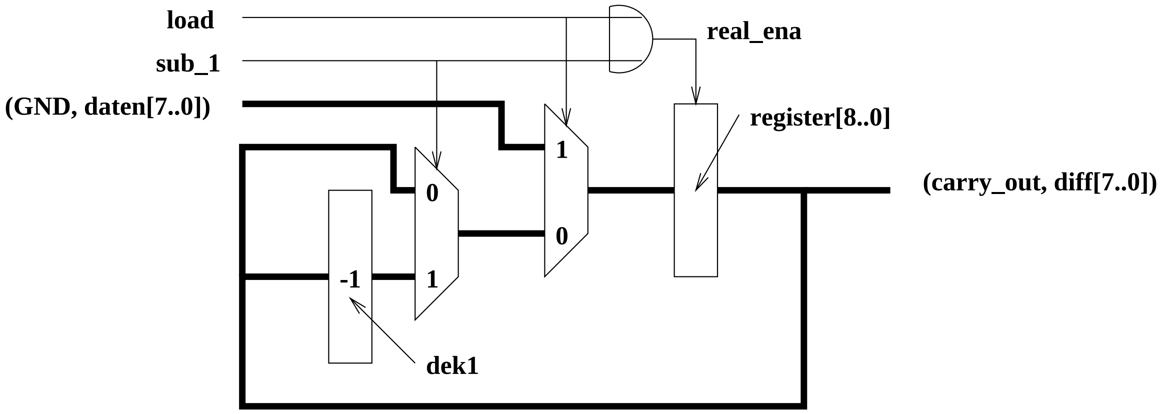

Abbildung 4.6: Struktur des ladbaren Adreßzählers (subtr2)

des ladbaren Adreßzählers zeigt Abbildung 4.6. Als Dateneingang wird ein Byte und eine 0 (=GND) eingegeben. Die 0 initialisiert das oberste Bit des Registers. Das *zd*-Signal entspricht dem *carry_out*.

Struktur der Adreßberechnung

Für die Adreßberechnung des Musters, identisch mit der der Maske, sind zwei *subtr2* notwendig. Einer dekrementiert die Musterbreite bzw. die Maskenbreite, der andere dekremtiert die Musterhöhe bzw. die Maskenhöhe, wenn der Adresszähler für die Breite = 0 ist (Signal *zd* ist gesetzt). Wenn das Muster an einer neuen Position des Bildes verglichen werden soll, dann werden beide Adreßzähler mit den Werten aus den Registern, in denen die Breite und die Höhe gespeichert sind, initialisiert. Anders verhält es sich bei den Adreßzählern des Bildes. Denn die neue Muster-Bild Vergleichsposition wird aus der derzeitigen Position im Bild berechnet. Daher sind hier für die Bildbreite und die Bildhöhe je zwei *subtr2* vorgesehen. In den ersten *subtr2* wird die Adresse der unteren rechten Ecke des Muster-Bild Vergleichs gespeichert. In den zweiten wird die interne Vergleichsposition gespeichert. Hier werden die Adressen berechnet, die die nächsten Bilddaten aus dem RAM zum Datenpuffer transferieren. Über einen Multiplexer wird dann ausgewählt, welche Adresse auf den Adressbus gelegt wird. Da hier nicht zwischen einer Muster- oder Maskenadresse bzw. zwischen einer Bild- und Ergebnisbyteadresse unterschieden werden muß, wird der Adresse noch eine sogenannte *RAM_ID[1..0]* als Adreßbits 17 und 16 hinzugefügt, die dann den richtigen Bereich im Speicher anspricht. Die Bildung der lokalen Adresse faßt die Abbildung 4.7 zusammen.

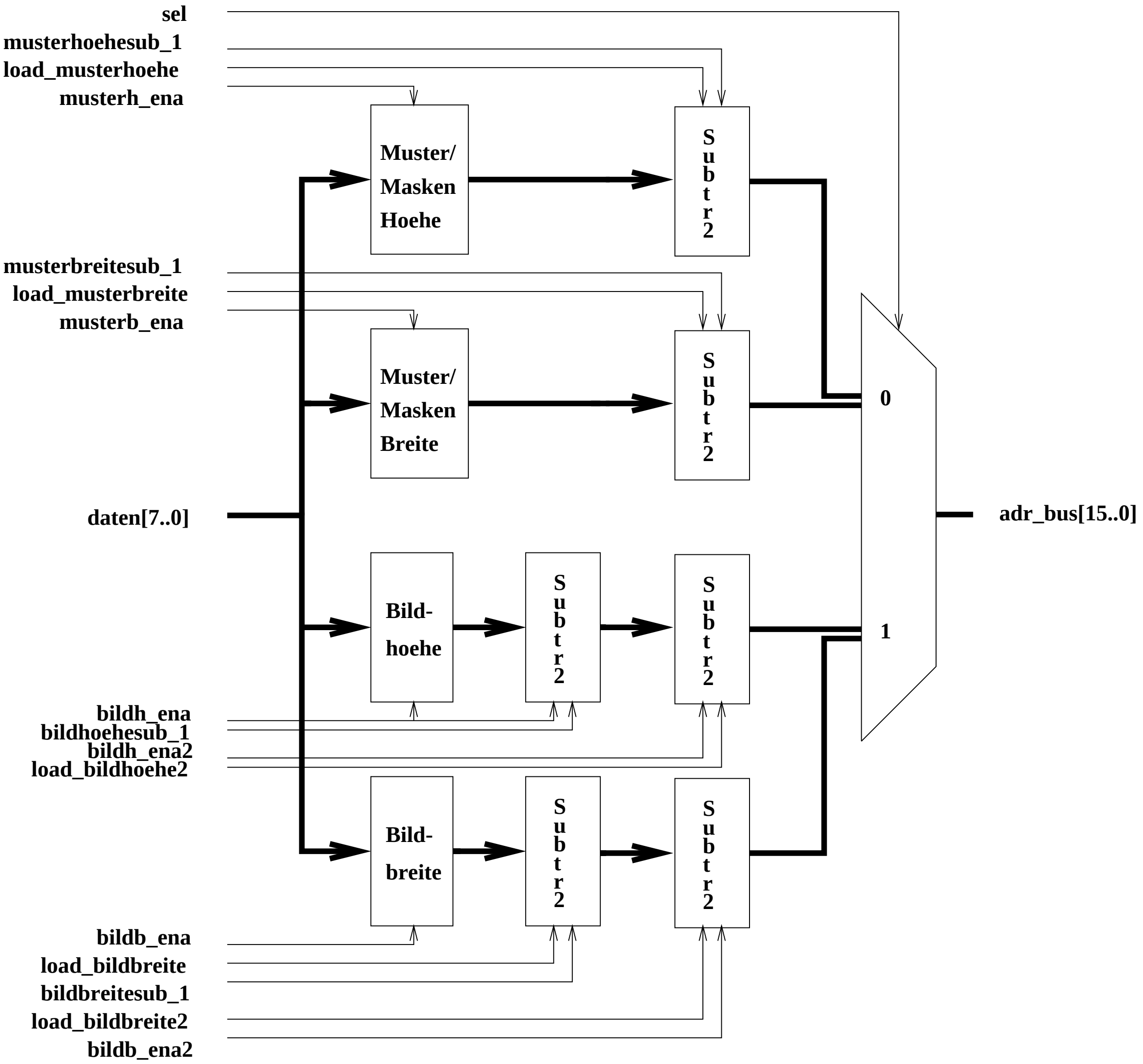

Abbildung 4.7: Struktur der Adreßberechnung im Steuerwerk

4.3.2 Ablaufsteuerung

Beim Vergleich ist darauf zu achten, daß alle 8 Vergleichspositionen eines Musterbits in einem Byte abgespeichert werden. Damit bei der vorgegebenen Vergleichsrichtung am Ende eines Mustervergleichs das *ergbyte* auch an der oberen linken Ecke abgespeichert wird, muß vor dem Vergleich ein Shiften der Bilddaten um 1 Bit nach links erfolgen. Die Abbildung 4.8 verdeutlicht diesen Schritt. Ein Vergleich an einer Bildposition, an der das Muster direkt am rechten Rand läge, ist dadurch nicht möglich. Es geht also eine Spalte verloren, was bezogen auf die Bildgröße relativ gering und somit auch tragbar ist.

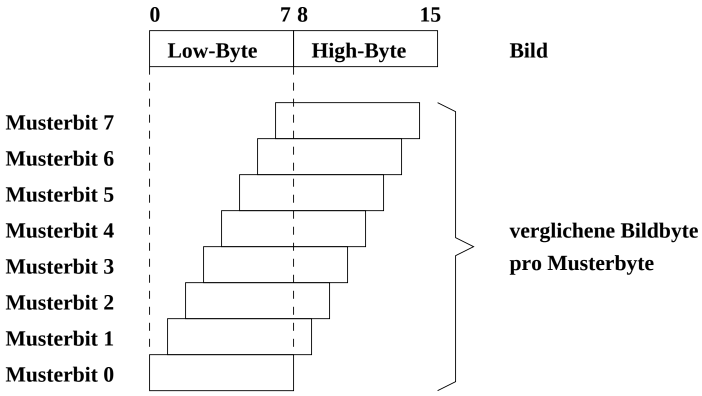

Abbildung 4.8: Verglichene Bildbyte pro Musterbyte

Die Aufgabe des Steuerwerks läßt sich in zwei Bereiche untergliedern. Der erste Teil befaßt sich mit dem Shiften der Daten und dem Vergleich, der andere mit dem Laden der neuen Daten in den Datenpuffer. Beides wird parallel durchgeführt. Dabei ist auf eine zeitliche Abfolge zu achten, so daß nicht beim Nachladen von Daten versehentlich Daten gelöscht werden, die noch relevant für den Vergleich sind. Unkritisch ist das Laden der Muster- und Maskendaten. Beim Vergleich wird nur das erste Byte, hier als High-Byte bezeichnet, benötigt, so daß dieses Register während des Vergleichs nicht überschrieben werden darf. Das zweite Pufferregister, hier als Low-Byte bezeichnet, wird dagegen nicht benötigt, so daß in dieses Register die nächsten Vergleichsdaten vom Muster bzw. der Maske geladen werden können. Wenn das High-Byte verglichen ist, dann wird das Low-Byte parallel in das High-Byte geladen. Dieses gilt sowohl für die Muster- als auch die Maskendaten. Dadurch wird ein *prefetching* auch bei einem Zeilensprung möglich, denn die Adresse der neuen Zeile kann bereits während des noch laufenden Vergleichs berechnet werden, ohne diesen zu beeinflussen. Bei den Bilddaten ist die beschriebene Vorgehensweise nicht möglich, denn für die Abarbeitung des High-Bytes von Muster und Maske werden die 2 Register des Bilddatenpuffers benötigt. Daher dürfen die Bildaten erst geladen werden, wenn das High-Byte abgearbei-

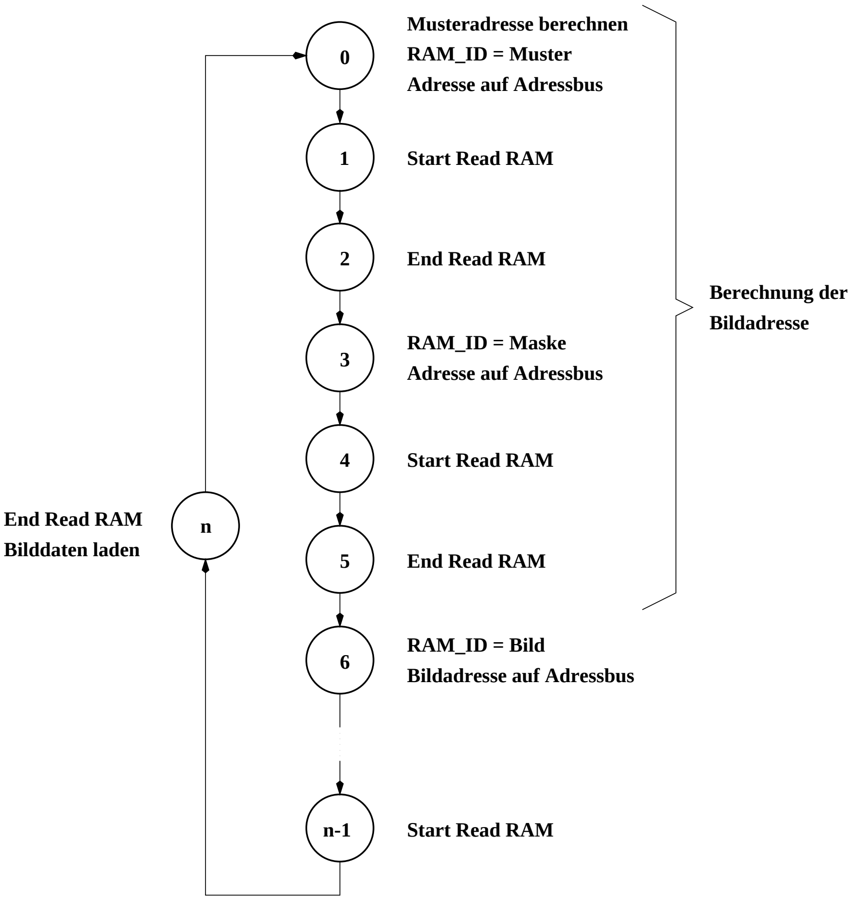

Abbildung 4.9: Steuerwerkskern zur Abwicklung des Vergleichs von 8 Musterbits. Die Berechnung der Adressen erfolgt parallel zum Shiften der Daten in den Pufferregistern. n entspricht der Wortbreite.

tet ist. Zur Komplettierung des Steuerwerkkerns in Abbildung 4.9 müssen vier Betriebsfälle unterschieden werden:

- **Fall 1:**
  Alle Schwellwerte sind übergelaufen. Das bedeutet, daß an dieser Bildposition der Vergleich beendet werden kann, weil das Muster nicht mehr erkannt werden kann. Stattdessen wird die Adresse des *ergbyte* berechnet und das Ergebnis (00000000) abgespeichert. Um diese Adresse zu berechnen, wird die Bildadresse spalten- und dann zeilenweise zusammen mit der Musteradresse dekrementiert, bis die linke obere Ecke des Musters erreicht ist.

- **Fall 2:**
  Es liegt kein Zeilenende vor. Dann wird der Spaltenindex der Musteradresse,

und damit auch der der Maskenadresse, dekrementiert. Danach wird das Low-Byte des Musterpuffer-Registers geladen. Anschließend wird das Low-Byte des Maskenpuffer-Registers geladen. Parallel dazu kann bereits der Spaltenindex der Bildadresse dekrementiert werden. Bevor die Bilddaten in das Low-Byte des Bildpuffer-Registers geladen werden, muß der vorherige Vergleich beendet sein.

- **Fall 3:**
  Es liegt ein Zeilenende vor, aber das Muster ist noch nicht komplett verglichen. Dann wird der Spaltenindex des Musteradresse neu initialisiert und der Zeilenindex dekrementiert. Danach werden Muster- und Maskenpuffer-Register geladen. Parallel dazu ergibt sich die neue Bildadresse, indem der Spatenindex neu initialisiert und der Zeilenindex dekrementiert wird. Nach dem Vergleichsende wird zunächst das High-Byte des Bildpuffer-Registers geladen. Nach dem Dekrementieren des Spaltenindex wird das Low-Byte geladen. Hier ergibt sich eine Verzögerung des Vergleichs, weil ein *prefetching* nicht möglich ist.

- **Fall 4:**
  Das Muster ist an der aktuellen Bildposition komplett verglichen worden. Dann wird die Musteradresse für den nächsten Vergleich an einer anderen Bildposition neu initialisiert. Die zugehörigen Daten werden in das High-Byte geladen. Nach dem Vergleichsende wird das Ergebnis an der für die zuletzt geladenen Bilddaten gültige Adresse gespeichert, wobei die RAM_ID dafür sorgt, daß in der Ergebnismatrix abgespeichert wird. Danach muß die Adresse auf die neue Bildposition gesetzt werden. Hierzu können wieder zwei Fälle unterschieden werden:

  – Der letzte Spaltenindex der Ergebnisadresse war ungleich 0. Dann wird der Spaltenindex der zwischengespeicherten momentanen Vergleichsposition (vgl. Kapitel 3 im Algorithmus *aktuelle_bildbreite*) dekrementiert und der Zeilenindex unverändert übernommen.

  – Der letzte Spaltenindex der Ergebnisadresse war gleich 0. Dann wird der Spaltenindex neu initialisiert und der Zeilenindex dekrementiert.

  Anschließend werden, wie unter Fall 2 beschrieben, die nächsten zwei Bildbytes geladen.

Der gesamte Vergleich wird beendet, wenn die letzte Bildadresse 0,0 war. Dann ist die obere linke Ecke des Bildes erreicht, was gleichbedeutend damit ist, daß das Muster an allen Stellen des Bildes, an denen es **vollständig** auftreten kann, verglichen worden ist. Abbildung 4.10 zeigt den vollständigen endlichen Automaten des Steuerwerks, wie es in diesem Kapitel vorgestellt wurde.

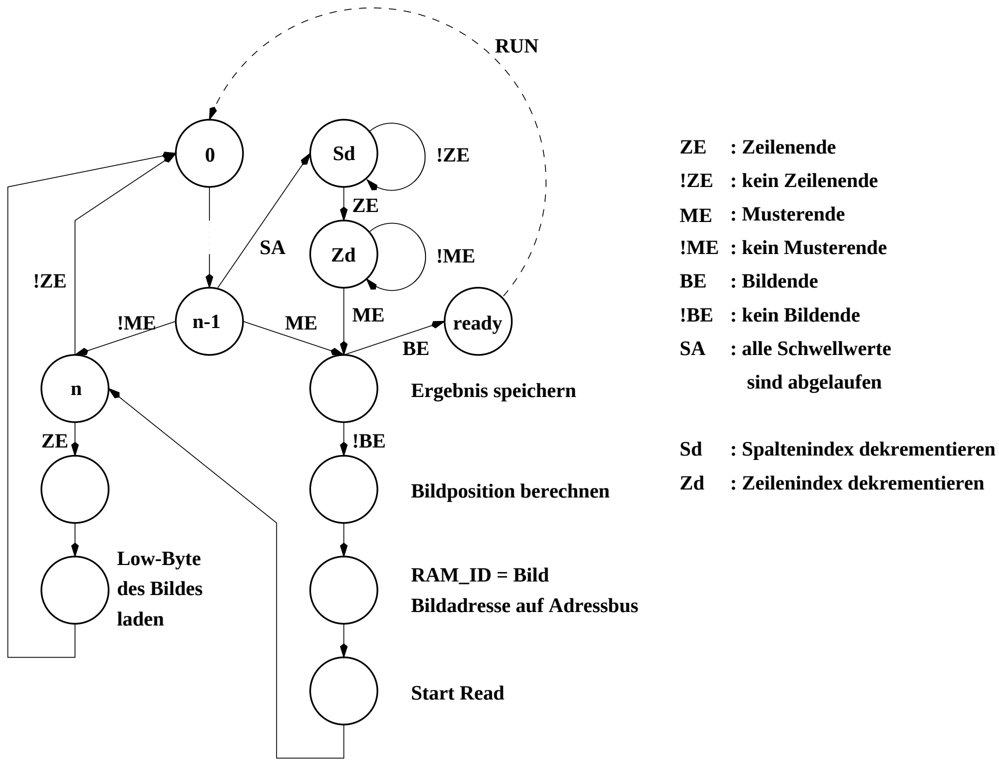

Abbildung 4.10: Struktur des Endlichen Automaten

4.4 Beispielkonfiguration

Das Operationswerk ist, wie in 4.1 Konzept beschrieben, kaskadierbar. Das bedeutet, daß ohne größeren Aufwand eine Verdopplung der Wortbreite des Datenbusses möglich ist, indem doppelt so viele Operationswerke miteinander verschaltet werden. Der Aufwand beschränkt sich auf das Verdrahten der Bausteine untereinander und einer Kodierung an das Steuerwerk, die mitteilt, wie groß die Wortbreite ist. Dies wird über die Eingangsleitungen *conf[2..0]* angegeben. Dabei wird folgende Kodierung verwendet:

conf[2..0]	Wortbreite
000	8
001	16
010	32
100	64

Im folgenden wird die Realisierung mit zwei Operationswerken entsprechend für 16 Bit Wortbreite beschrieben. Zunächst werden alle Steuerleitungen mit den entsprechenden Eingängen der Operationswerke verbunden. Dabei gibt es nur

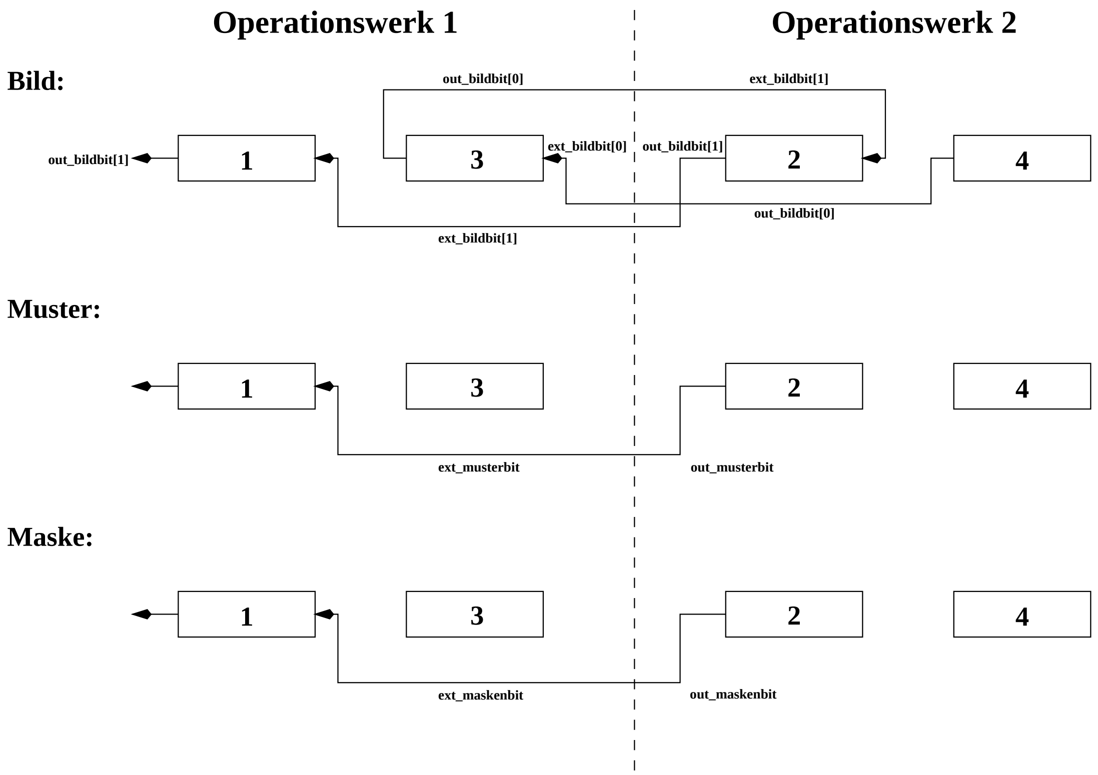

Abbildung 4.11: Schema für das Shiften der Bild-, Muster- und Maskendaten bei Verschaltung zweier Operationswerke

eine Ausnahme, nämlich das Signal *m_s*. Mit diesem Signal wird dem Operationswerk mitgeteilt, ob es als Master (m_s = 1) oder Slave (m_s = 0) fungieren soll. Das Steuerwerk gibt dieses Signal sowohl direkt als auch negiert aus. Somit muß extern kein Inverter vorgesehen werden. Beim Steuerwerk müsen ansonsten keine Änderungen beachtet werden. Etwas komplizierter ist es bei den Operationswerken, denn hier müssen aufgrund der Aufteilung des Datenbusses auf zwei Operationswerke jeweils die richtigen Daten in den Datenpuffer nachgeshiftet werden. Dadurch daß die Operationswerke nur eine 8 Bit Verbindung zum Datenbus haben, werden die zwei zusammenhängenden Bytes des Datenbusses in getrennten Operationswerken zwischengespeichert. So sind beim Initialisieren die Bildbytes 1 und 3 im Operationswerk 1 (Master), während die Bildbytes 2 und 4 im Operationswerk 2 (Slave) abgelegt werden. Bildbytes 1 und 2 ergeben das erste Bildwort, Bildbytes 3 und 4 das zweite Bildwort. Aus dieser Tatsache ergibt sich die Verdrahtung der Bildleitungen wie in Abbildung 4.11 gezeigt. Die Verdrahtung der Muster- und Maskenpuffer realisiert das Shiften des Bytes 2 in das Byte 1. Wenn 16 Shiftschritte durchgeführt, also 2 Bytes des Musters mit dem Bild verglichen worden sind, dann werden die Bytes 3 und 4 parallel in Byte

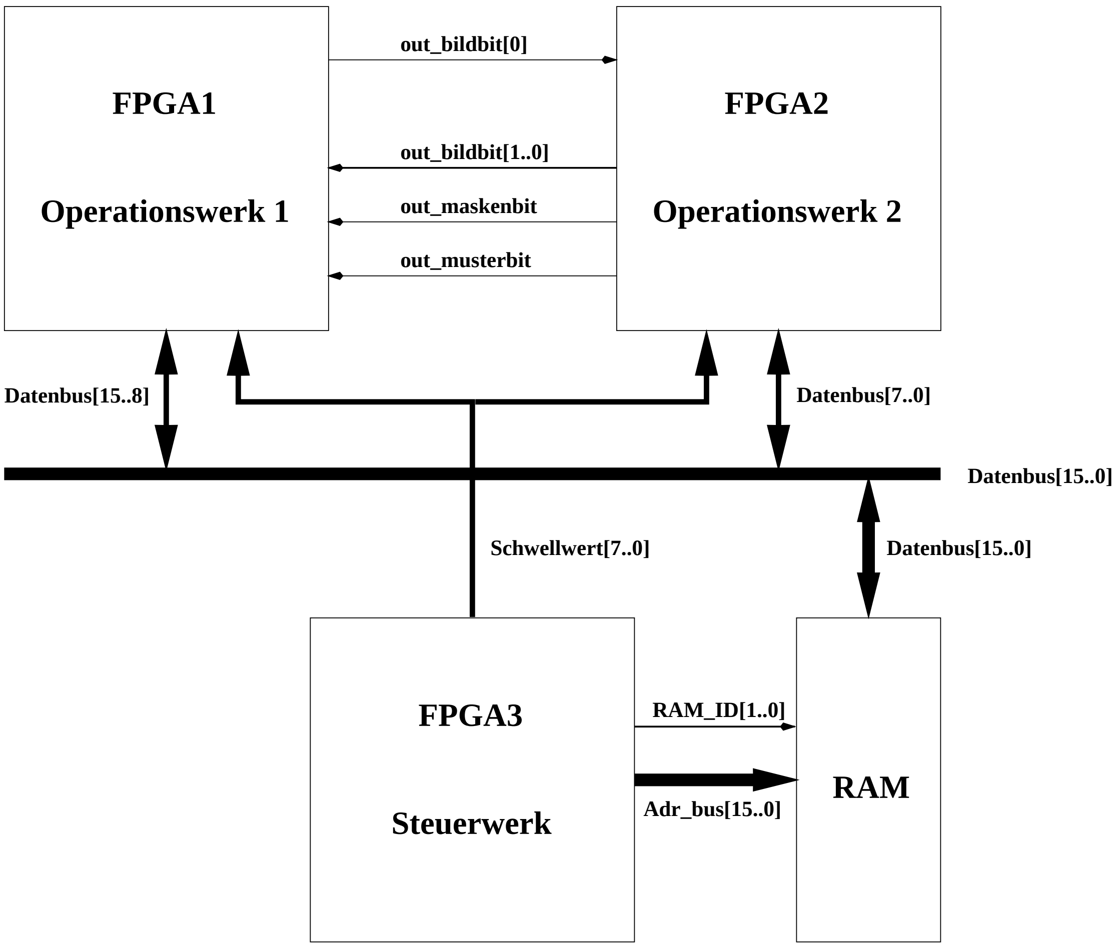

Abbildung 4.12: Globale Verdrahtung bei Verwendung zweier Operationswerke

1 bzw. Byte 2 geladen. Die externen Verdrahtungen der Pufferregister für das Muster und die Maske werden ebenfalls in Abbildung 4.11 dargestellt. Auf die Darstellung der internen Verdrahtungen der Pufferregister ist hier der Übersicht halber verzichtet worden.
Einen abschließenden Überblick über die Verschaltung der drei *FLEX 8000*- Bausteine gibt die Abbildung 4.12. Die Bezeichnung der Leitungen entspricht den Namen der Ausgangsleitung des Bausteins, von dem sie ausgehen.

5. Bewertung und Ausblick

Da es sich bei diesem Design um eine Implementation eines strikten Pixelvergleichs handelt, sind einige Funktionen wie zum Beispiel das Auffinden eines Musters in verschiedenen Größen oder das Auffinden eines Musters unter verschiedenen Rotationswinkeln nicht möglich. Um die Effiktivität des ASICs zu analysieren, wird im folgenden eine Zeitanalyse durchgeführt, die dabei von der Tatsache ausgeht, daß das Muster komplett an jeder Bildposition verglichen werden muß, also die Schwellwerte nicht vor Ende eines Mustervergleichs ablaufen, so daß jedes Byte des Musters an allen möglichen Bildpositionen verglichen werden muß. Die Anzahl der benötigten Takte pro Musterzeile berechnet sich zu:

$$T_{M_Z} = \frac{M_B}{W_B} \cdot W_B + 2 = M_B + 2$$

Der Bruch $\frac{M_B}{W_B}$ gibt an, wieviele Musterwörter aufgrund der Wortbreite verglichen werden. Diese Anzahl wird mit der verwendeten Wortbreite des Datenbusses multipliziert, weil pro Wortvergleich die gleiche Anzahl von Shiftschritten notwendig ist, um das Wort vollständig zu vergleichen. Pro Shiftschritt wird ein Takt benötigt. Beim Zeilenumbruch sind zwei zusätzliche Takte erforderlich, um ein zweites Bildwort zu laden. Die gesamte Taktanzahl eines Mustervergleichs ergibt sich durch Multiplikation der Gleichung für T_{M_Z} mit der Musterhoehe, hier als M_H bezeichnet. Am Ende eines Mustervergleichs sind zum Berechnen der neuen Vergleichsposition 4 Takte zusätzlich erforderlich. Die Taktanzahl für einen Mustervergleich berechnet sich dann wie folgt:

$$T_M = (M_B + 2) \cdot M_H + 4$$

Die Taktanzahl für einen Mustervergleich ist damit von der gewählten Wortbreite unabhängig. Jetzt muß noch die Anzahl der Mustervergleiche im gesamten Bild, A_Z, ermittelt werden. Diese ergibt sich nach folgender Gleichung:

$$A_Z = \left\lceil \frac{B_B - M_B + 1}{W_B} \right\rceil$$

Die Summe $B_B - M_B + 1$ gibt dabei an, an wieviel verschiedenen Positionen das Muster in einer Bildzeile positioniert werden kann. Dieser Wert muß durch die Wortbreite dividiert werden, weil so viele Positionen gleichzeitig überprüft werden. Für die Anzahl der Musterpositionen bezogen auf die Bildhöhe ergibt sich:

$$A_H = B_H - M_H + 1$$

Somit ergibt sich für die Gesamtanzahl der verschiedenen Musterpositionen im Bild:

$$\begin{aligned} A_G &= A_Z \cdot A_H \\ &= \left\lceil \frac{B_B - M_B + 1}{W_B} \right\rceil \cdot (B_H - M_H + 1) \end{aligned}$$

Für die Gesamtanzahl der benötigten Takte für einen Mustervergleich an allen Positionen im Bild ergibt sich dann:

$$\begin{aligned} T_G &= T_M \cdot A_G \\ &= \big((M_B + 2) \cdot M_H + 4\big) \cdot \left\lceil \frac{B_B - M_B + 1}{W_B} \right\rceil \cdot (B_H - M_H + 1) \end{aligned}$$

Damit bedeutet eine Verdopplung der Wortbreite ungefähr eine Halbierung der Zeit. Multipliziert man diese Gleichung mit dem Kehrwert der Taktfrequenz, dann erhält man die benötigte Zeit. Bei einer Bildgröße von 512 * 512 Pixel, einer Mustergröße von 256 * 256 und einer Taktfrequenz von 20 MHz ergibt sich bei einer Wortbreite von 8 Bit eine Zeit von ca. 28s. Bei gleicher Voraussetzung aber mit einer Wortbreite von 16 Bit ergibt sich eine Zeit von 14,4s. In der Praxis wird die benötigte Zeit kürzer sein, denn es wird häufiger vorkommen, daß die Schwellwerte abgelaufen sind. Dadurch werden viele Vergleichsschritte eingespart. Die Einsparung wird noch größer, wenn ein Subtrahierer eingebaut wird, um die Adresse für das *ergbyte* nach Ablauf der Schwellwerte direkt zu berechnen, anstatt die Bildadresse synchron mit der Musteradresse zu dekrementieren. Wie groß die Zeitersparnis ausfällt, hängt davon ab, nach wievielen Vergleichsschritten die Schwellwerte abgelaufen sind.
Eine weitere Möglichkeit, die benötigte Zeit zu minimieren, wäre eine Erhöhung der Taktfrequenz. Dazu wäre es denkbar, das Design mit Hilfe von Tools in einen Standarzellentwurf umzuwandeln.
Auch eine weitere Parallelisierung der internen Vergleiche wäre denkbar. Bei dem derzeitigen Design wird bei Verwendung eines Operationswerkes nur ein Musterbit mit 8 Bildbit gleichzeitig verglichen. Stattdessen könnte ein komplettes Musterbyte mit 16 Bildbits verglichen werden, was einer byteorientierten achtfach parallelen Lösung entspräche. Hierzu müßte aber die Adressierung beschleunigt und die Speicherbandbreite erhöht werden. Erheblich aufwendiger wird dann auch die Behandlung des Schwellwertes, denn nun kann dieser nicht einfach beim

Auftreten von Abweichungen dekrementiert werden. Vielmehr ist jetzt ein Abweichzähler nötig, denn bei einem byteorientierten Vergleich können Abweichungen an bis zu 8 unterschiedlichen Positionen auftreten. Nach einem Bytevergleich müssen also alle Abweichungen aufsummiert und diese Summe vom Schwellwert subtrahiert werden. Dadurch wären pro Vergleicher ein Abweichzähler (Ausgabe maximal 4 Bit) und ein Subtrahierer erforderlich.

Literaturverzeichnis

[ACT92] *ACT Family Field Programmable Gate Array Data Book*, Actel Corporation, April 1992

[Alt93] *Altera Databook*, Altera Corporation, San Jose CA, 1993.

[AMD92] *Mach^{TM} Family Data Book*, Advanced Micro Devices Corporation, 1992

[AtA94] Peter M. Athans und A. Lynn Abbott,
Image Processing on a Custom Computing Platform,
4th International Workshop on Field-Programmable Logic and Application, FPL '94,
Prag 1994, Springer Verlag

[BFR90] Stephen D. Brown, Robert J. Francis, Jonothan Rose, Zvonko G. Krawesic,
Field Programmable Gate Arrays,
Kluwer Academic Publishers, Boston, Dordrecht, London

[HiF90] Kenneth K. Hillen, Bradly Fawcett
Build Reconfigurable Peripheral Controllers Electronic Design, Penton Publication, März 1990.

[Kli94] A. Klindworth,
A Tool-Set for Simulating Altera-PLDs Using VHDL,
4th International Workshop on Field-Programmable Logic and Application, FPL '94,
Prag 1994, Springer Verlag

[Kol89] Reiner Kolla,
Einführung in den VLSI Entwurf,
Stuttgart 1989, Teubner Verlag

[Syn93] *Synopsys VHDL Simulator Reference Manual*, Synopsys Inc., 1993.

[Xil92] *The Programmable Gate Array Data Book*, Xilinx Corporation, San Jose CA, 1992.

A. Designfiles des Operationswerks

A.1 Topdesign

Zunächst kommen die Konstantendeklarationen, die in dem Designfile *bit8ser7* verwendet werden.

```
CONSTANT
 bildpuffer_high_sel = B"110";
CONSTANT
 bildpuffer_low_sel = B"010";
CONSTANT
 musterpuffer_high_sel = B"101";
CONSTANT
 musterpuffer_low_sel = B"001";
CONSTANT
 maskenpuffer_high_sel = B"100";
CONSTANT
 maskenpuffer_low_sel = B"000";
```

```
%
Dieses Design realisiert den Vergleich von einem Maskenbit und Musterbit mit
einem Bildbyte. Dabei werden die Daten fuer den Vergleich in einen Datenpuffer
gespeichert, der fuer Muster, Maske und Bild jeweils 16 Bit umfasst.Dabei haben
die Leitungen folgende Bedeutung:
1) load       : laden des Schwellwertes
2) shift_bild : shiften des Pufferregister fuer das Bild um 1 Bit
3) shift_muma : shiften der Pufferregister fuer Muster und Maske um 1 Bit
4) load_muma  : zum Laden des Pufferregisters von Muster und Maske in das obere
                Byte
5) sel        : zur Auswahl des nachzushiftenden Bits ( 1 = letztes Bit kommt aus
                einem externen Register, 0 = letztes Bit kommt aus unterem Puf-
                ferregister)
6) m_s        : Steuersignal zur Auswahl von master und slave
7) ext_*bit   : externe Anschluesse fuer das richtige Shiften
8) neudat[]   : Steuersignal zum Laden neuer Daten in die Datenpuffer
                neudat = "000" : Daten werden in den maskenpuffer[7..0] geladen
                neudat = "100" : Daten werden in den maskenpuffer[15..8] geladen
                neudat = "001" : Daten werden in den musterpuffer[7..0] geladen
                neudat = "101" : Daten werden in den musterpuffer[15..8] geladen
                neudat = "010" : Daten werden in den bildpuffer[7..0] geladen
                neudat = "110" : Daten werden in den bildpuffer[15..8] geladen
                neudat = "111" : Es werden keine Daten geladen
%

INCLUDE "const.inc";

FUNCTION bitser21 (clk, ena, clr, load,  bildbit, maskenbit, musterbit,
                   schwellwert[7..0]) RETURNS (ergbit);

SUBDESIGN bit8ser7 (clk, ena, clr, m_s, load, daten[7..0], schwellwert[7..0],
                   neudat[2..0], shift_bild, shift_muma, load_muma, sel,
                   ext_bildbit[1..0], ext_maskenbit, ext_musterbit : INPUT;
```

```
                   ergbyte[7..0], out_bildbit[1..0], out_maskenbit, out_musterbit
                   : OUTPUT)



VARIABLE
 bit1ser[7..0]        : bitser21;
 schwellwertreg[7..0] : DFF;
 bildpuffer[15..0]    : DFFE;
 maskenpuffer[15..0]  : DFFE;
 musterpuffer[15..0]  : DFFE;
 n_bildbit            : NODE;
 n_maskenbit          : NODE;
 n_musterbit          : NODE;

BEGIN

% Puffer fuer die Daten des Vergleichs !!! %
 bildpuffer[].clk = clk;
 bildpuffer[].clrn = clr;
 bildpuffer[7..0].ena = SOFT(shift_bild # neudat[] == bildpuffer_low_sel);
 bildpuffer[15..8].ena = SOFT(shift_bild # neudat[] == bildpuffer_high_sel);
 maskenpuffer[].clk = clk;
 maskenpuffer[].clrn = clr;
 maskenpuffer[7..0].ena = SOFT(shift_muma # neudat[] == musterpuffer_low_sel);
 maskenpuffer[15..8].ena = SOFT(shift_muma # neudat[] == musterpuffer_high_sel
                                # load_muma);
 musterpuffer[].clk = clk;
 musterpuffer[].clrn = clr;
 musterpuffer[7..0].ena = SOFT(shift_muma # neudat[] == maskenpuffer_low_sel);
 musterpuffer[15..8].ena = SOFT(shift_muma # neudat[] == maskenpuffer_high_sel
                                # load_muma);
% Ende des Datenpuffers !!! %

maskenpuffer[7..0].d = daten[];
maskenpuffer[15..8].d = daten[];
musterpuffer[7..0].d = daten[];
musterpuffer[15..8].d = daten[];
bildpuffer[7..0].d = daten[];
bildpuffer[15..8].d = daten[];

% HIER KOMMT DER MULTIPLEXERBEREICH !! %
 IF sel THEN
  n_bildbit = ext_bildbit[1];
 END IF;
 IF shift_bild THEN
  bildpuffer[15..0].d = (bildpuffer[14..8].q, n_bildbit, bildpuffer[6..0].q,
                         ext_bildbit[0]);
 END IF;
 IF shift_muma THEN
  musterpuffer[15..8].d = (musterpuffer[14..8].q, ext_musterbit);
  maskenpuffer[15..8].d = (maskenpuffer[14..8].q, ext_maskenbit);
 END IF;
 IF load_muma THEN
  musterpuffer[15..8].d = musterpuffer[7..0].q;
  maskenpuffer[15..8].d = maskenpuffer[7..0].q;
 END IF;
 out_bildbit[] = (bildpuffer[15].q, bildpuffer[7].q);
 out_maskenbit = maskenpuffer[15].q;
 out_musterbit = musterpuffer[15].q;
 IF m_s THEN
  n_musterbit = musterpuffer[15].q;
  n_maskenbit = maskenpuffer[15].q;
 ELSE
  n_musterbit = ext_musterbit;
```

```
  n_maskenbit = ext_maskenbit;
 END IF;
% ENDE DES MULTIPLEXERBEREICHS !!%

 schwellwertreg[].clk = clk;
 schwellwertreg[].clrn = clr;
 schwellwertreg[].d = schwellwert[];
 bit1ser[].clk = clk;
 bit1ser[].clr = clr;
 bit1ser[].ena = ena;
 bit1ser[].load = load;
 bit1ser[].bildbit = bildpuffer[15..8].q;
 bit1ser[].musterbit = n_musterbit;
 bit1ser[].maskenbit = n_maskenbit;
 bit1ser[7].schwellwert[] = schwellwertreg[].q;
 bit1ser[6].schwellwert[] = schwellwertreg[].q;
 bit1ser[5].schwellwert[] = schwellwertreg[].q;
 bit1ser[4].schwellwert[] = schwellwertreg[].q;
 bit1ser[3].schwellwert[] = schwellwertreg[].q;
 bit1ser[2].schwellwert[] = schwellwertreg[].q;
 bit1ser[1].schwellwert[] = schwellwertreg[].q;
 bit1ser[0].schwellwert[] = schwellwertreg[].q;
 ergbyte[] = (bit1ser[7].ergbit, bit1ser[6].ergbit, bit1ser[5].ergbit,
             bit1ser[4].ergbit, bit1ser[3].ergbit, bit1ser[2].ergbit,
             bit1ser[1].ergbit, bit1ser[0].ergbit);
END;
```

A.2 Vergleicher

```
%
Dieses Subdesign dient einem bitorientierten Vergleich von Muster und Bild, wo-
bei eine eventuelle Abweichung von einem festen von aussen angebbaren Schwell-
wert abgezogen wird. Der neue Wert der noch moeglichen Abweichungen wird als
countabweich ausgegeben.
%

FUNCTION  count7 (clk, enable, clr, load, abweichbit, schwellwert[7..0])
                RETURNS (ergbit);

FUNCTION vergleis (bildbit, maskenbit, musterbit) RETURNS (abweichbit);

SUBDESIGN bitser21 (clk, ena, clr, load,  bildbit, maskenbit, musterbit,
                   schwellwert[7..0] : INPUT;
                   ergbit: OUTPUT)

VARIABLE
 vergl       : vergleis;
 dekrementer : count7;


BEGIN
 vergl.bildbit = bildbit;
 vergl.maskenbit = maskenbit;
 vergl.musterbit = musterbit;
 dekrementer.clk = clk;
 dekrementer.enable = ena;
 dekrementer.clr = clr;
 dekrementer.load = load;
 dekrementer.abweichbit = vergl.abweichbit;
 dekrementer.schwellwert[] = schwellwert[];
 ergbit = dekrementer.ergbit;
END;
```

A.3 Dekrementierer

```
%
Subdesign zum Ansteuern des Schwellwertes fuer die bitorientierte
serielle Loesung.
%

SUBDESIGN count7 (clk, enable, clr, load, abweichbit, schwellwert[7..0] : INPUT;
                 ergbit : OUTPUT)

VARIABLE
 reg[8..0]     : DFFE;
 counter[8..0] : NODE;
 real_enable   : NODE;

BEGIN
 reg[].clk = clk;
 reg[].clrn = clr;
 real_enable = load # enable & !reg[8].q & abweichbit;
 reg[8..0].ena = real_enable;
 IF load THEN
  counter[] = (GND, schwellwert[]);
 ELSE
  counter[] = reg[].q - 1;
 END IF;
 IF real_enable THEN
  reg[].d = counter[];
 ELSE
  reg[].d = reg[].q;
 END IF;
 ergbit = !reg[8];
END;
```

A.4 Bitvergleicher

```
%
Dieses Design dient zum Vergleich von einem Bildbit mit einem
Musterbit. Dabei wird beruecksichtig mit Hilfe eines Maskenbits,
ob eine eventuelle Abweichung relevant ist fuer den gesamten
Vergleich (relevant: Maskenbit = 1, sonst 0).
%

SUBDESIGN vergleis (bildbit, maskenbit, musterbit : INPUT;
                    abweichbit : OUTPUT)

BEGIN
 abweichbit = maskenbit & (bildbit $ musterbit);
END;
```

B. Designfiles des Steuerwerks

B.1 Adreßberechnungseinheit (ABE)

```
%
Dieses Design umfasst das Operationswerk des Steuerwerks. Hier werden
die Bild- und Musterbreiten (Musterbreite = Maskenbreite), sowie die
Bild- und Musterhoehe (Musterhoehe = Maskenhoehe) gespeichert und ueber
Zaehler fuer die Adressberechnung dementsprechend veraendert.
Folgende Steuerleitungen werden benoetigt:
 1) bildb_ena*        : enable-Signal fuer die Register der Bildbreite
 2) bildh_ena*        : enable-Signal fuer die Register der Bildhoehe
 3) musterb_ena       : enable-Signal fuer die Register der Masken- und Muster-
                        breite
 4) musterh_ena       : enable-Signal fuer die Register der Masken- und Muster-
                        hoehe
 5) sel               : Signal fuer den Multiplexer am Ausgang (1 = Bildadres-
                        se, 0 = Masken- und Musteradresse)
 6) load_bildbreite*  : Signal zum Laden der Bildbreite in die Dekrementierer
                        (subtr2)
 7) load_bildhoehe2   : Signal zum Laden der Bildhoehe in den zweiten Dekremen-
                        tierer
 8) load_musterbreite : Signal zum Laden der Masken- und Musterbreite in den
                        Dekrementierer
 9) load_musterhoehe  : Signal zum Laden der Masken- und Musterhoehe in den
                        Dekrementierer
10) bildbreitesub_1   : Signal zum Dekrementieren der Bildbreite
11) bildhoehesub_1    : Signal zum Dekrementieren der Bildhoehe
12) musterbreitesub_1 : Signal zum Dekrementieren der Masken- und Musterbreite
13) musterhoehesub_1  : Signal zum Dekrementieren der Masken- und Musterhoehe

Ausgangssignale:
 1) adr_bus[]         : Adresse fuer die neu zu ladenden Daten
 2) zd*b              : zero_detect fuer Bild-, Masken- und Musterbreite
                        (1: der Wert ist gleich 0)
 3) zd*h              : zero_detect fuer Bild-, Masken- und Musterhoehe
                         (1: der Wert ist gleich 0) %
%

FUNCTION subtr2 (clk, load, sub_1, clr, a[7..0])
                RETURNS (diff[7..0], carry_out);


SUBDESIGN stwopw2 (clk, bildb_ena, bildb_ena2, bildh_ena, bildh_ena2,
                  musterb_ena, musterh_ena, clr, sel, load_bildbreite,
                  load_bildbreite2, load_bildhoehe2, load_musterbreite,
                  load_musterhoehe, daten[7..0], bildbreitesub_1,
                  bildhoehesub_1, musterbreitesub_1, musterhoehesub_1 : INPUT;
                  adr_bus[15..0], zdbildb, zdbildh, zdmumab,
                  zdmumah : OUTPUT)
```

```
VARIABLE
 bildbreitereg[7..0]      : DFFE;
 bildhoehereg[7..0]       : DFFE;
 musterbreitereg[7..0]    : DFFE;
 musterhoehereg[7..0]     : DFFE;
 bildbreite[7..0]         : NODE;
 bildhoehe[7..0]          : NODE;
 musterbreite[7..0]       : NODE;
 musterhoehe[7..0]        : NODE;
 zwischenbildbreite[7..0] : NODE;
 zwischenbildhoehe[7..0]  : NODE;

BEGIN
 bildbreitereg[].clk = clk;
 bildbreitereg[].clrn = clr;
 bildbreitereg[].ena = bildb_ena;
 bildbreitereg[].d = daten[];
 bildhoehereg[].clk = clk;
 bildhoehereg[].clrn = clr;
 bildhoehereg[].ena = bildh_ena;
 bildhoehereg[].d = daten[];
 musterbreitereg[].clk = clk;
 musterbreitereg[].clrn = clr;
 musterbreitereg[].ena = musterb_ena;
 musterbreitereg[].d = daten[];
 musterhoehereg[].clk = clk;
 musterhoehereg[].clrn = clr;
 musterhoehereg[].ena = musterh_ena;
 musterhoehereg[].d = daten[];
 (zwischenbildbreite[], zdbildb) = subtr2(clk, load_bildbreite, bildbreitesub_1,
                                          clr, bildbreitereg[].q);
 (zwischenbildhoehe[], zdbildh) = subtr2(clk, bildh_ena, bildhoehesub_1, clr,
                                         bildhoehereg[].q);
 (bildbreite[], zdbildb) = subtr2(clk, load_bildbreite2, bildb_ena2, clr,
                                  zwischenbildbreite[]);
 (bildhoehe[], zdbildh) = subtr2(clk, load_bildhoehe2, bildh_ena2, clr,
                                 zwischenbildhoehe[]);
 (musterbreite[], zdmumab) = subtr2(clk, load_musterbreite, musterbreitesub_1,
                                    clr, musterbreitereg[].q);
 (musterhoehe[], zdmumah) = subtr2(clk, load_musterhoehe, musterhoehesub_1, clr,
                                   musterhoehereg[].q);
 IF sel THEN
  adr_bus[] = (bildhoehe[], bildbreite[]);
 ELSE
  adr_bus[] = (musterhoehe[], musterbreite[]);
 END IF;
END;
```

B.2 Adreßdekrementierer

```
%
Design fuer den angesteuerten Dekrementierer des Steuerwerks.
%

FUNCTION dek1 (a[7..0]) RETURNS (b[7..0], carry_out);


SUBDESIGN subtr2 (clk, load, sub_1, clr, a[7..0] : INPUT;
                  diff[7..0], carry_out : OUTPUT)

VARIABLE
  reg[8..0]          : DFFE;
  real_ena           : NODE;

BEGIN
 reg[].clk = clk;
 reg[].clrn = clr;
 real_ena = LCELL(sub_1 # load);
 reg[].ena = real_ena;
 IF load THEN
  reg[].d = (GND, a[]);
 ELSE
  IF sub_1 THEN
   (reg[7..0].d, reg[8].d) = dek1(reg[7..0].q);
  ELSE
   reg[].d = reg[].q;
  END IF;
 END IF;
 (carry_out, diff[]) = reg[].q;
END;
```

B.3 Dekrementierer

```
%
Dekrementierer fuer das Steuerwerk mit Verwendung der Carry Chain.
%

SUBDESIGN dek1 (a[7..0] : INPUT;
                b[7..0], carry_out : OUTPUT)

VARIABLE
 carry_i[7..0]  : NODE;

BEGIN
 b[0] = !a[0];
 carry_i[0] = CARRY(!a[0]);
 b[1] = LCELL(a[1] $ carry_i[0]);
 carry_i[1] = CARRY(!a[1] & carry_i[0]);
 b[2] = LCELL(a[2] $ carry_i[1]);
 carry_i[2] = CARRY(!a[2] & carry_i[1]);
 b[3] = LCELL(a[3] $ carry_i[2]);
 carry_i[3] = CARRY(!a[3] & carry_i[2]);
 b[4] = LCELL(a[4] $ carry_i[3]);
 carry_i[4] = CARRY(!a[4] & carry_i[3]);
 b[5] = LCELL(a[5] $ carry_i[4]);
 carry_i[5] = CARRY(!a[5] & carry_i[4]);
 b[6] = LCELL(a[6] $ carry_i[5]);
 carry_i[6] = CARRY(!a[6] & carry_i[5]);
 b[7] = LCELL(a[7] $ carry_i[6]);
 carry_i[7] = !a[7] & CARRY(carry_i[6]);
 carry_out = CARRY(carry_i[7]);
END;
```