TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät für Elektrotechnik und Informationstechnik Institut für Grundlagen der Elektrotechnik und Elektronik

Professur für hochparallele VLSI-Systeme und Neuromikroelektronik

BELEGARBEIT SCHALTKREIS- UND SYSTEMENTWURF

Timo Nicolai

Login: niti17
Studiengang: Informationssystemtechnik
Matrikelnummer: 4048209

1. Dezember 2018

Betreuer

Dr.-Ing. Sebastian Höppner

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit mit dem Titel *Belegarbeit Schaltkreis- und Systementwurf* selbstständig und ohne unzulässige Hilfe Dritter verfasst habe. Es wurden keine anderen als die in der Arbeit angegebenen Hilfsmittel und Quellen benutzt. Die wörtlichen und sinngemäß übernommenen Zitate habe ich als solche kenntlich gemacht. Es waren keine weiteren Personen an der geistigen Herstellung der vorliegenden Arbeit beteiligt. Mir ist bekannt, dass die Nichteinhaltung dieser Erklärung zum nachträglichen Entzug des Hochschulabschlusses führen kann.

Dresden, 1. Dezember 2018

Timo Nicolai

INHALTSVERZEICHNIS

1 EINLEITUNG

Die vorliegende Belegarbeit beschreibt den Entwurf einer digitalen Schaltung, welche die LUP-Dekomposition für beliebig große quadratische Matrizen realisiert. Der Algorithmus entstammt der linearen Algebra, er kann unter anderem eingesetzt werden, um lineare Gleichungssysteme zu lösen und Matrizen zu invertieren.

Im Folgenden wird zunächst die Funktionsweise des Algorithmus erläutert. Anschließend wird schrittweise der Entwurf einer entsprechenden Hardwareimplementierung beschrieben, welche aus den Hauptkomponenten Finite State Machine, Kontrolllogic und Datenpfad zusammengesetzt ist. Folgend wird mittels Simulation die Funktionsweise der FSM und anhand konkreter Bespiele die Korrektheit der Gesamtimplementierung verifiziert. Eine RTL-Synthese konnte mit den in der Praktikumsanleitung dargestellten Schritten aufgrund technischer Probleme leider nicht realisiert werden. Anschließend findet eine Zusammenfassung und Wertung der Arbeit statt, im letzten Kapitel sind dann Verilog Quellcode für die Verhaltensbeschreibung einzelner Schaltungskomponenten und für genutzte Testbenches sowie Python Quellcode der Referenzimplementierung und weiterer Hilfsprogramme aufgeführt. Im Anhang befinden sich Abbildungs-, Tabellen- und Literaturverzeichnis.

Das Projekt hat den Namen *LUP_Decomp* und ist in Cadence Virtuoso als gleichnamige Library gespeichert. Die Referenzimplementierung und weitere Skripte befinden sich im run-Verzeichnis *.../sim/LUP_Decomp_Top_tb/mem*.

2 BESCHREIBUNG DES ALGORITHMUS

2.1 DIE LUP-DEKOMPOSITION

2.1.1 PRINZIP

Die LUP-Dekomposition[1] ist ein Algorithmus, welcher einer nicht singulären $N \times N$ Matrix $\boldsymbol{A}$ drei Matrizen $\boldsymbol{L}$, $\boldsymbol{U}$ und $\boldsymbol{P}$ zuordnet, sodass die Gleichung

$$\boldsymbol{PA} = \boldsymbol{LU} \tag{2.1}$$

erfüllt ist und $\boldsymbol{L}$ bzw. $\boldsymbol{U}$ die folgende Form haben:

$$\boldsymbol{L} = \begin{pmatrix} 1 & & 0 \\ l_{2,1} & 1 & \\ l_{3,1} & l_{3,2} & \ddots \\ \vdots & \vdots & \ddots \\ l_{N,1} & l_{N,2} & \cdots \; l_{N,N\text{-}1} \; 1 \end{pmatrix} \qquad \boldsymbol{U} = \begin{pmatrix} u_{1,1} & u_{1,2} & \cdots\cdots & u_{1,N} \\ & u_{2,2} & \cdots\cdots & u_{2,N} \\ & & \ddots & \vdots \\ & & & \\ 0 & & & u_{N,N} \end{pmatrix}$$

$\boldsymbol{L}$ ist also eine untere Dreiecksmatrix, bei der alle Einträge auf der Hauptdiagonalen eins sind. $\boldsymbol{U}$ ist eine obere Dreiecksmatrix.

Weiterhin ist $\boldsymbol{P}$ eine $N \times N$ Permutationsmatrix, d.h. in jeder Reihe und Spalte von $\boldsymbol{P}$ ist jeweils ein Eintrag eins und alle anderen sind null.

Die LUP-Dekomposition hängt mit dem Gaußschen Eliminationsverfahrens zur Lösung von linearen Gleichungssystemen zusammen. Mit diesem lassen sich Gleichungssysteme der Form

$$\boldsymbol{A}x = \boldsymbol{b} \tag{2.2}$$

lösen (Ist $\boldsymbol{A}$ quadratisch existiert eine Lösung dabei nur, wenn $\boldsymbol{A}$ nichtsingulär d.h. $det(\boldsymbol{A}) \neq 0$ ist und somit $\boldsymbol{A}^{-1}$ existiert). Hierzu wird zunächst die sogenannte erweiterte Koeffizientenmatrix gebildet:

$$\left(\boldsymbol{A} \;\middle|\; \boldsymbol{b}\right) = \left(\begin{array}{ccc|c} a_{1,1} & \ldots & a_{1,N} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{N,1} & \ldots & a_{N,N} & b_N \end{array}\right)$$

Diese wird dann mit der bekannten Vorgehensweise durch elementare Zeilenuformungen in obere Dreiecksform gebracht. Dabei kann es unter Umständen nötig sein, Zeilen zu vertauschen. Zum Beispiel kann im ersten Umformungsschritt für $a_{1,1} = 0$ die erste Reihe nicht zur Elimination der übrigen Elemente in der ersten Spalte ($a_{2,1}$ bis $a_{N,1}$) verwendet werden, sie muss daher mit einer beliebigen anderen Reihe i mit $a_{i,1} \neq 0$ vertauscht werden. Für Software/Hardware-Implementierungen kann das Vertauschen von Reihen in diesem Beispiel auch

[1] Die Beschreibung und Implementierung des Algorithmus orientieren sich an der Darstellung in [COR01].

für $a_{1,1} \neq 0$ sinnvoll sein, da die numerische Stabilität des Algorithmus hier für $|a_{i,1}| > |a_{1,1}|$ verbessert werden würde (Ähnliches gilt für die LUP-Dekomposition, wobei der konkrete Grund bei der detaillierten Betrachtung des Algorithmus deutlich werden wird).

Nach Bilden der Dreiecksform lassen sich die Komponenten des unbekannten Vektors $\boldsymbol{x}$ dann durch rekursives Lösen der modifizierten Gleichungen "von unten nach oben" mithilfe von Rückwärtseinsetzen bestimmen.

Die durch die LUP-Dekomposition von $\boldsymbol{A}$ erhaltene Matrix $\boldsymbol{U}$ enthält nun genau die Werte, welche bei Durchführung des Gaußschen Eliminationsverfahrens mit Reihentausch zur numerischen Stabilisierung im linken Teil der erweiterten Koeffizientenmatrix übrig bleiben würden. Zusätzlich werden in $\boldsymbol{L}$ konzeptionell die einzelnen Eliminationsschritte festgehalten, sodass dann zum Beispiel Gleichungssysteme der Form (2.2) nicht nur für einen speziellen Vektor $\boldsymbol{b}$, sondern für alle denkbaren gelöst werden können.

Der Algorithmus lässt sich konkret so implementieren, dass die Eingabe-Matrix $\boldsymbol{A}$ in einer Reihe rekursiver Berechnungsschritte so modifiziert wird, dass sie nach Ablauf des Algorithmus die Elemente von $\boldsymbol{L}$ (abzüglich der Einsen auf der Hauptdiagonale) und $\boldsymbol{U}$ in folgender Form enthält:

$$\boldsymbol{A'} = \begin{pmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,N} \\ l_{2,1} & u_{2,2} & \cdots & u_{2,N} \\ l_{3,1} & l_{3,2} & \ddots & \vdots \\ \vdots & \vdots & \ddots & \\ l_{N,1} & l_{N,2} & \cdots & l_{N,N-1} \; u_{N,N} \end{pmatrix} \tag{2.3}$$

Die Permutationsmatrix $\boldsymbol{P}$ ergibt sich dabei gerade so, dass Gleichung (2.1) erfüllt ist. Damit wird das Vertauschen von Reihen von $\boldsymbol{A}$ während des Ablaufes des Algorithmus berücksichtigt.

Mit den erhaltenen Matrizen kann man nun zum Beispiel leicht lineare Gleichungssysteme der Form (2.2) lösen. Dabei folgt mit (2.1):

$$\begin{aligned} \boldsymbol{PA}x &= \boldsymbol{Pb} \\ \Rightarrow \boldsymbol{LU}x &= \boldsymbol{Pb} \end{aligned} \tag{2.4}$$

Die Gleichung (2.4) kann durch die spezielle Form von $\boldsymbol{L}$ und $\boldsymbol{U}$ leicht durch zweimaliges rekursives Lösen mit Rückwärtseinsetzen gelöst werden:

$$\begin{aligned} \boldsymbol{L}y = \boldsymbol{Pb} &\rightarrow \text{ nach y auflösen} \\ \boldsymbol{U}x = y &\rightarrow \text{ nach x auflösen} \end{aligned}$$

2.1.2 EIN ANSCHAULICHES BEISPIEL

Vor der detaillierten Beschreibung der Implementierung des Algorithmus soll nun zunächst anhand eines Beispiels ein Gefühl für dessen Arbeitsweise entwickelt werden. Es wird dabei die untenstehende Matrix $\boldsymbol{A}$ betrachtet. Hierbei ist $\boldsymbol{\pi}$ eine Repräsentation von $\boldsymbol{P}$ als Zeilenvektor: Hat das Element in der $i$ten Spalte von $\boldsymbol{\pi}$ den Wert j, so ist in $\boldsymbol{P}$ das Element $p_{i,j}$ eins und für alle $k \neq j$ mit $1 \leq k \leq N$ gilt $p_{i,k} = 0$. Somit entspricht $\boldsymbol{\pi}$ zu Beginn des Algorithmus der 3×3 Einheitsmatrix.

$$\boldsymbol{A} = \begin{pmatrix} -1 & -2 & 2 \\ -8 & 2 & 4 \\ -5 & -5 & -6 \end{pmatrix} \qquad \boldsymbol{\pi} = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

Im ersten Schritt wird ein sogenanntes Pivotelement bestimmt, dieses ist das betragsgrößte Element der ersten Spalte von $\boldsymbol{A}$. Das Pivotelement ist hier die -8. Es wird nun die erste Zeile von $\boldsymbol{A}$ mit der das Pivotelement enthaltenden Zeile *i* vertauscht, hier gilt *i* = 2. Gleichzeitig wird $\boldsymbol{\pi}$ aktualisiert, indem der Eintrag in der ersten Spalte von $\boldsymbol{\pi}$ mit dem in der *i*ten Spalte vertauscht wird.

$$\boldsymbol{A} = \begin{pmatrix} -8 & 2 & 4 \\ -1 & -2 & 2 \\ -5 & -5 & -6 \end{pmatrix} \qquad \boldsymbol{\pi} = \begin{pmatrix} 2 & 1 & 3 \end{pmatrix}$$

Anschließend werden alle Elemente in der ersten Spalte von $\boldsymbol{A}$ (außer dem Pivotelement selbst) durch das Pivotelement dividiert.

$$\boldsymbol{A} = \begin{pmatrix} -8 & 2 & 4 \\ 1/8 & -2 & 2 \\ 5/8 & -5 & -6 \end{pmatrix}$$

Von den Elementen der Untermatrix $\boldsymbol{A_{11}}$ wird nun jeweils das Produkt aus den ersten Einträgen der dem jeweiligen Element zugehörigen Reihe und Spalte subtrahiert. Dieser Prozess wird auch als Bildung des *Schur-Komplements* bezeichnet.

$$\boldsymbol{A} = \begin{pmatrix} -8 & 2 & 4 \\ 1/8 & -9/4 & 2 \\ 5/8 & -5 & -6 \end{pmatrix} \qquad \boldsymbol{A} = \begin{pmatrix} -8 & 2 & 4 \\ 1/8 & -9/4 & 3/2 \\ 5/8 & -5 & -6 \end{pmatrix}$$

$$\boldsymbol{A} = \begin{pmatrix} -8 & 2 & 4 \\ 1/8 & -9/4 & 3/2 \\ 5/8 & -25/4 & -6 \end{pmatrix} \qquad \boldsymbol{A} = \begin{pmatrix} -8 & 2 & 4 \\ 1/8 & -9/4 & 3/2 \\ 5/8 & -25/4 & -17/2 \end{pmatrix}$$

Es kommt nun zur Rekursion, der gleiche Algorithmus wird auf der Untermatrix $\boldsymbol{A_{11}}$ ausgeführt. Das Pivotelement ist hier dementsprechend das betragsgrößte Element der ersten Spalte von $\boldsymbol{A_{11}}$, also $-25/4$. Es müssen also Zeile zwei und drei vertauscht werden (Achtung: es werden beim Zeilentausch immer Zeilen von $\boldsymbol{A}$ und nicht nur die der Untermatrizen vertauscht).

$$\boldsymbol{A} = \begin{pmatrix} -8 & 2 & 4 \\ 5/8 & -25/4 & -17/2 \\ 1/8 & -9/4 & 3/2 \end{pmatrix} \qquad \boldsymbol{\pi} = \begin{pmatrix} 2 & 3 & 1 \end{pmatrix}$$

Es verbleiben zwei Rechenschritte nach obigem Schema:

$$\boldsymbol{A_{11}} = \begin{pmatrix} -25/4 & -17/2 \\ 9/25 & 3/2 \end{pmatrix} \qquad \boldsymbol{A_{11}} = \begin{pmatrix} -25/4 & -17/2 \\ 9/25 & 114/25 \end{pmatrix}$$

Im nächsten Rekursionsschritt müssten die gleichen Schritte nun auf eine Untermatrix von $\boldsymbol{A_{11}}$, nämlich $(\boldsymbol{A_{11}})_{\boldsymbol{11}}$, angewandt werden. Da diese jedoch ein Skalar ist, sind wir an dieser Stelle fertig. $\boldsymbol{L}$, $\boldsymbol{U}$ und $\boldsymbol{P}$ können nun einfach aus $\boldsymbol{A}$ bzw. $\boldsymbol{\pi}$ abgelesen werden:

$$\boldsymbol{L} = \begin{pmatrix} 1 & 0 & 0 \\ 5/8 & 1 & 0 \\ 1/8 & 9/25 & 1 \end{pmatrix} \quad \boldsymbol{U} = \begin{pmatrix} -8 & 2 & 4 \\ 0 & -25/4 & -17/2 \\ 0 & 0 & 114/25 \end{pmatrix} \quad \boldsymbol{P} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

Damit kann nun beispielsweise folgendes Gleichungssystem gelöst werden:

$$\boldsymbol{A}x = \boldsymbol{b} \quad \text{mit} \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 8 \\ -33 \end{pmatrix}$$

Mit zweimaligem rekursiven Lösen mithilfe von Rückwärtseinsetzen (Zwischenschritte wurden hier ausgespart) erhält man:

$$\begin{pmatrix} 1 & 0 & 0 \\ 5/8 & 1 & 0 \\ 1/8 & 9/25 & 1 \end{pmatrix} y = \begin{pmatrix} 8 \\ -33 \\ 1 \end{pmatrix} \quad \Rightarrow \quad y = \begin{pmatrix} 8 \\ -38 \\ 342/25 \end{pmatrix}$$

$$\begin{pmatrix} -8 & 2 & 4 \\ 0 & -25/4 & -17/2 \\ 0 & 0 & 114/25 \end{pmatrix} x = \begin{pmatrix} 8 \\ -38 \\ 342/25 \end{pmatrix} \quad \Rightarrow \quad x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

2.2 IMPLEMENTIERUNG

Es wird nun auf das prinzipielle Vorgehen bei der Implementierung des Algorithmus eingegangen. Dabei wird zunächst kurz der Aufbau des hierfür verwendeten Hauptspeichers erläutert und anschließend im Detail auf den Ablauf des den Algorithmus realisierenden Programms eingegangen.

2.2.1 SPEICHERLAYOUT

Abbildung 2.1 zeigt den schematischen Aufbau[2] des Hauptspeichers vor Ablauf des Algorithmus. Der Speicher ist zweidimensional organisiert und einzelne Speicherzellen können über zwei Adresseingänge (je einer für den Zeilen-/Spaltenindex[3]) separat addressiert werden.

In der ersten Spalte der ersten Speicherzeile ist die Zeilen-/Spaltenanzahl N der zu verarbeitenden Matrix $\boldsymbol{A}$ gespeichert. Die folgenden Elemente der ersten Zeile bilden den Permutationsvektor $\boldsymbol{\pi}$, welcher zu Beginn die $N \times N$ Einheitsmatrix repräsentiert. Auf alle weiteren Elemente der ersten Spalte (hier mit "x" markiert) wird nicht zugegriffen. Die Elemente $a_{i,j}$ der Matrix $\boldsymbol{A}$ befinden sich an den ihren Indizes entsprechenden Positionen im Speicher, die Adressierung dieser Elemente gestaltet sich hierdurch sehr intuitiv.

---

[2] Wie diese Speicherstruktur konkret in Hardware implementiert werden kann wird in diesem Beleg nicht näher betrachtet, der Speicher wird als Blackbox mit den hier beschriebenen Eigenschaften behandelt. Für die Simulation wird mit einer entsprechenden funktionalen Beschreibung in Verilog gearbeitet.

[3] Der Adresseingang für den Zeilenindex wird im Folgenden mit `ADR1` und der für den Spaltenindex mit `ADR2` bezeichnet, ein Zugriff auf eine Speicherzelle in der $i$ten Zeile und $j$ten Spalte wird entweder durch `MEM(i,j)` oder `[i,j]` gekennzeichnet.

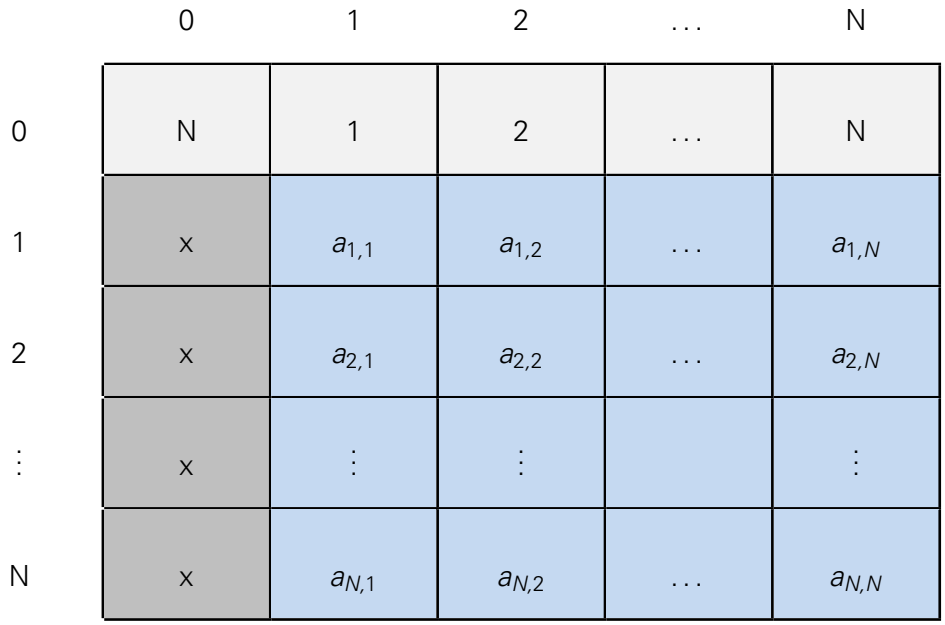

Abbildung 2.1: Schematische Darstellung des Hauptspeicher-Layouts

In jeder Speicherzelle liegt eine 32-Bit Binärzahl. Während N und die Elemente von π dabei als Ganzzahlen interpretiert werden, sind die Matrixelemente Festkommazahlen mit je 16-Bit Vor- und Nachkommaanteil. Das Rechnen mit Fest- anstatt Fließkommazahlen ermöglicht eine Realisierung, die schneller ist (weil z.B. die Division von Fließkommazahlen erheblich länger dauert) und weniger Chipfläche beansprucht. Gleichzeitig können hierdurch sehr kleine und sehr große Werte der Matrixelemente nicht mehr dargestellt werden, was die Gefahr von Überläufen erhöht. Da diese durch den Mechanismus der Pivotelemente aber sowieso in Grenzen gehalten wird, ist dies durchaus akzeptabel.

In der Darstellung in Abbildung 2.1 wird davon ausgegangen, dass die Dimensionen des Speichers genau denen der durch den Algorithmus verarbeiteten Matrizen entsprechen. Der Algorithmus kann jedoch genauso auf $N' \times N'$ Matrizen mit $N' < N$ ablaufen, hierzu muss nur der Inhalt der Speicherzelle `[0,0]` entsprechend zu N' abgeändert werden.

Wie bereits im Rechenbeispiel verdeutlicht, kommt der Algorithmus ohne zusätzlichen Speicher für das Endergebnis aus. Die zu Beginn im Hauptspeicher liegenden Daten werden einfach schrittweise modifiziert, sodass nach Ablauf des Algorithmus $\boldsymbol{L}$, $\boldsymbol{U}$ (siehe Gleichung 2.3) und $\boldsymbol{P}$ einfach aus dem Speicherinhalt konstruiert werden können[4].

2.2.2 PROGRAMMABLAUF

Abbildung 2.2 zeigt ein Nassi-Shneiderman-Diagramm, welches den Programmablauf im Detail darstellt. Die verwendeten Variablen finden sich in der Implementierung als gleichnamige Register wieder.

Zu Beginn wird die Zeilen-/Spaltenanzahl N der zu verarbeitenden quadratischen Matrix $\boldsymbol{A}$ aus dem Hauptspeicher geladen und in einer gleichnamigen Variable gespeichert. Die Variable `rec`, welche die aktuelle Rekursion speichert, wird auf eins gesetzt.

Anschließend wird die Hauptschleife des Programms betreten. In jeder Iteration dieser Schleife wird auf einer Untermatrix der Eingabematrix $\boldsymbol{A}$ gearbeitet, zunächst auf $\boldsymbol{A}$ selbst, dann auf

[4] Dieser letzte Schritt ist nicht Teil dieser Arbeit.

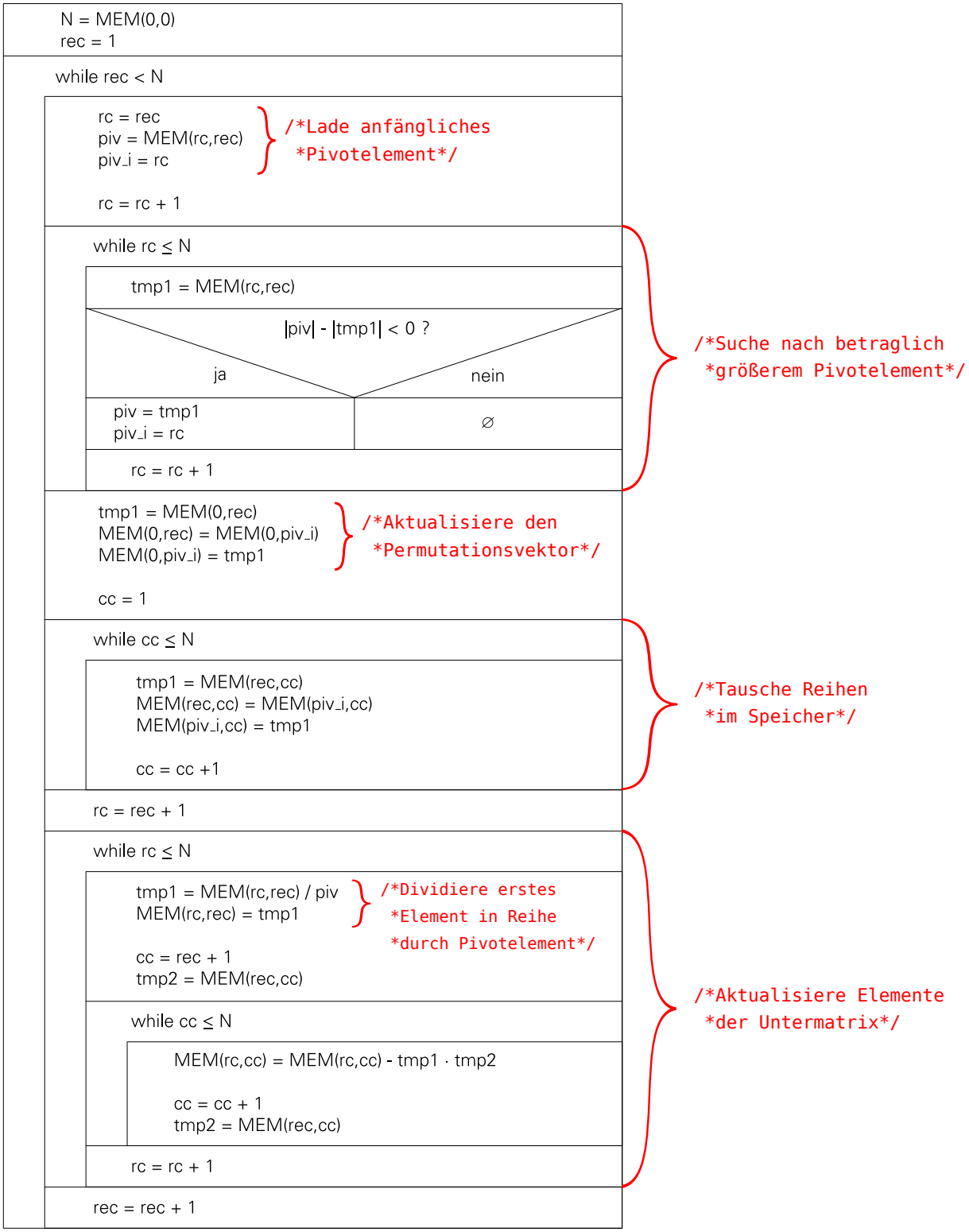

Abbildung 2.2: Nassi-Shneiderman-Diagramm

A_{11}, dann auf $(A_{11})_{11}$ und so weiter. Die hierfür nötigen Reihen- und Spalten-Offsets ergeben sich aus dem in `rec` gespeicherten Wert.

In der Schleife wird zunächst für die aktuelle Untermatrix das Pivotelement bestimmmt. Hierzu wird das Element in der ersten Zeile und ersten Spalte der aktuellen Untermatrix in der Variable `piv` ("Pivot") und dessen Zeilenindex in der Variable `piv_i` ("Pivot Index") gespeichert. Dann wird der Betrag von `piv` in einer Schleife nach und nach mit dem der nachfolgenden Elemente in der ersten Spalte der aktuellen Untermatrix verglichen. Zur Adressierung der entsprechenden Zeilen wird dabei der Schleifenzähler `rc` ("Row Counter") verwendet. Sobald ein Element mit größerem Absolutwert gefunden wurde, wird `piv` mit dessen Wert und `piv_i` mit dem aktuellen Wert in `rc` überschrieben und die Schleife forgesetzt, sodass `piv` nach Durchlauf der Schleife den Wert des betragsgrößten Element der ersten Spalte hat und `piv_i` dessen Zeilenindex ist.

Anschließend wird der Permutationsvektor aktualisiert. Hierbei wird das Element des Permutationsvektors, dessen Spaltenindex dem Zeilenindex der ersten Zeile der aktuellen Untermatrix entspricht, mit dem Element des Permutationsvektors mit Spaltenindex `piv_i` getauscht (unter Zuhilfenahme der Hilfsvariable `tmp1`).

Anschließend werden durch ähnliches Vorgehen in einer Schleife die entsprechenden Speicherzeilen vertauscht. Dabei wird spaltenweise vorgegangen, d.h. in jedem Schleifendurchlauf werden für eine Spalte der Untermatrix die beiden Elemente aus den beiden Reihen vertauscht. Hierbei kommt zur Adressierung der Spalten der Schleifenzähler `cc` ("Column Counter") zum Einsatz.

Man beachte, dass die letzten beiden Schritte übersprungen werden können, wenn kein größeres Pivotelement als das zuerst betrachtete gefunden wurde, was in der Hardwareimplementierung in diesem Fall einige Taktzyklen Berechnungszeit einsparen würde. Darauf wird hier bewusst verzichtet, zum einen um die Komplexität der für die Implementierung benötigten FSM in Grenzen zu halten, zum anderen weil insbesondere für große Matrizen der Zeilentausch statistisch gesehen weitaus öfter stattfindet als nicht. Außerdem hängt die benötigte Rechenzeit somit nur von der Größe der Eingabematrix ab, was eventuell die Verwendung der entstandenen Schaltung als Element in einer anderen erleichtern könnte.

Im nächsten Schritt kann das Schur-Komplement gebildet werden. Für jede Zeile der Untermatrix ab der zweiten (`rec` + 1) wird zunächst das erste Element aus dem Speicher geladen, durch das Privotelement dividiert und in den Speicher zurückgeschrieben. Hier zeigt sich die Sinnhaftigkeit der Suche nach dem betragsgrößten Pivotelement: Die Wahrscheinlichkeit eines Überlaufes des Ergebnisses der Division wird so minimiert. Das Ergebnis der Division wird gleich wieder benötigt und daher in der Variable `tmp1` zwischengespeichert. Von allen restlichen Elementen der Zeile wird dann das Produkt aus dem in `tmp1` gespeicherten Wert und dem ersten Element in der gleichen Spalte der aktuellen Untermatrix (An der Speicheradresse [`rec`, `cc`]) abgezogen.

Ist dies für alle Reihen erledigt, beginnt der nächste Durchlauf der Hauptschleife, wobei `rec` um eins erhöht wird. Der Algorithmus bricht ab, wenn `rec` den Wert `N` erreicht hat.

3 ENTWURF

3.1 DATENFLUSSGRAPH

Abbildung 3.1 zeigt den aus Abbildung 2.2 abgeleiteten Datenflussgraphen. Am rechten Rand sind bereits Zustandsnamen zum Vergleich mit der später beschriebenen FSM eingetragen. Das Scheduling der Operationen wurde dabei "von Hand" durchgeführt und zielt auf einen möglichst schnellen Programmdurchlauf ab. Es wurde besonderer Wert auf die Geschwindigkeit von den Schleifen, die den Großteil der Rechenzeit beanspruchen, gelegt. Die am häufigsten durchlaufene Schleife ist die, die von den Zuständen `SCHUR1` bis `SCHUR_NEXT` gebildet wird und in zwei andere Schleifen geschachtelt ist. Es ist aus dem Diagramm ersichtlich, dass hier eine hohe Operationsdichte und entsprechend kurze Abarbeitungszeit erreicht wurde. Allgemein lässt sich am Datenflussgraphen erkennen, dass es kein wesentlich schnellers Scheduling der benötigten Operationen als das hier gewählte geben kann, da bereits in fast jedem Zustand ein Zugriff auf den Hauptspeicher erfolgt. Der Hauptspeicher bildet gewissermaßen einen Flaschenhals und für weitere Optimierung müsste das parallele Auslesen/Beschreiben mehrerer Speicherzellen unterstützt werden.

Nicht-kritische Operationen wurden nicht nach dem ASAP oder ALAP Prinzip gescheduled sondern direkt so gelegt, dass die benötigten Datenpfadelemente bei fester Programmdurchlaufzeit minimiert werden. Es werden dabei zwingend eine ALU zur Durchführung verschiedener Additionen und Subtraktionen, ein Dividierer zur Division von Spaltenelementen durch das Pivotelement und ein Multiplizierer zur Bildung des Schur-Komplements benötigt. Eine zweite ALU ist notwendig um an einigen Stellen (zum Beispiel in der genannten Schleife) den Ablauf zu beschleunigen. Um den Entwurf übersichtlich zu halten, ist eine ALU (`ALU1`) hauptsächlich für das Berechnen von Flags zur Steuerung des Kontrollflusses und die andere (`ALU2`) hauptsächlich für die Inkrementierung von Schleifenzählern zuständig.

Es wurde zudem versucht, die Operationen so auf die Rechenelemente zu verteilen und den Transfer zwischen Register so auszulegen, dass die Komplexität der Verbindungselemente im Datenpfad möglichst gering gehalten wird (siehe dazu Abschnitt 3.2).

Anzumerken ist noch, dass für die von den beiden ALUs ausgeführten Operationen jeweils das im Diagramm von links anliegende Register im Datenpfad an den "positiven" Eingang der ALU angelegt wird. d.h. im Falle einer Subtraktion den Minuenden bildet, es sei denn das Register ist explizit mit einem `(-)` als Subtrahend gekennzeichnet. Für Additionen hält sich das Diagramm an das gleiche Schema damit der Datenflussgraph im Einklang mit der Datenflussanalyse in Tabelle 3.1 ist.

Im Zustand `DET_PIV5` müssen außerdem die Beträge des Inhaltes von Register `PIV` und des Speicherausganges `DO` verglichen werden, dies ist mit `ABS()` gekennzeichnet und wird im Datenpfad durch entsprechende betragsbildende kombinatorische Schaltungselemente berücksichtigt.

Die Zustände `DET_PIV_NEXT` und `UPD_PIV` sind parallel dargestellt, wobei die markierten Operationen (das Aktualisieren des Pivotelements und dessen Index) nur im Zustand `UPD_PIV` durchgeführt werden. Im Unterschied zum zuvor betrachteten Programmablaufplan wird hier zusätzlich auch der Fall berücksichtigt, dass in einem Durchlauf der Hauptschleife kein Pivotelement ungleich Null gefunden wurde. Dieser Fall kann nur eintreten, wenn die betrachtete Matrix sin-

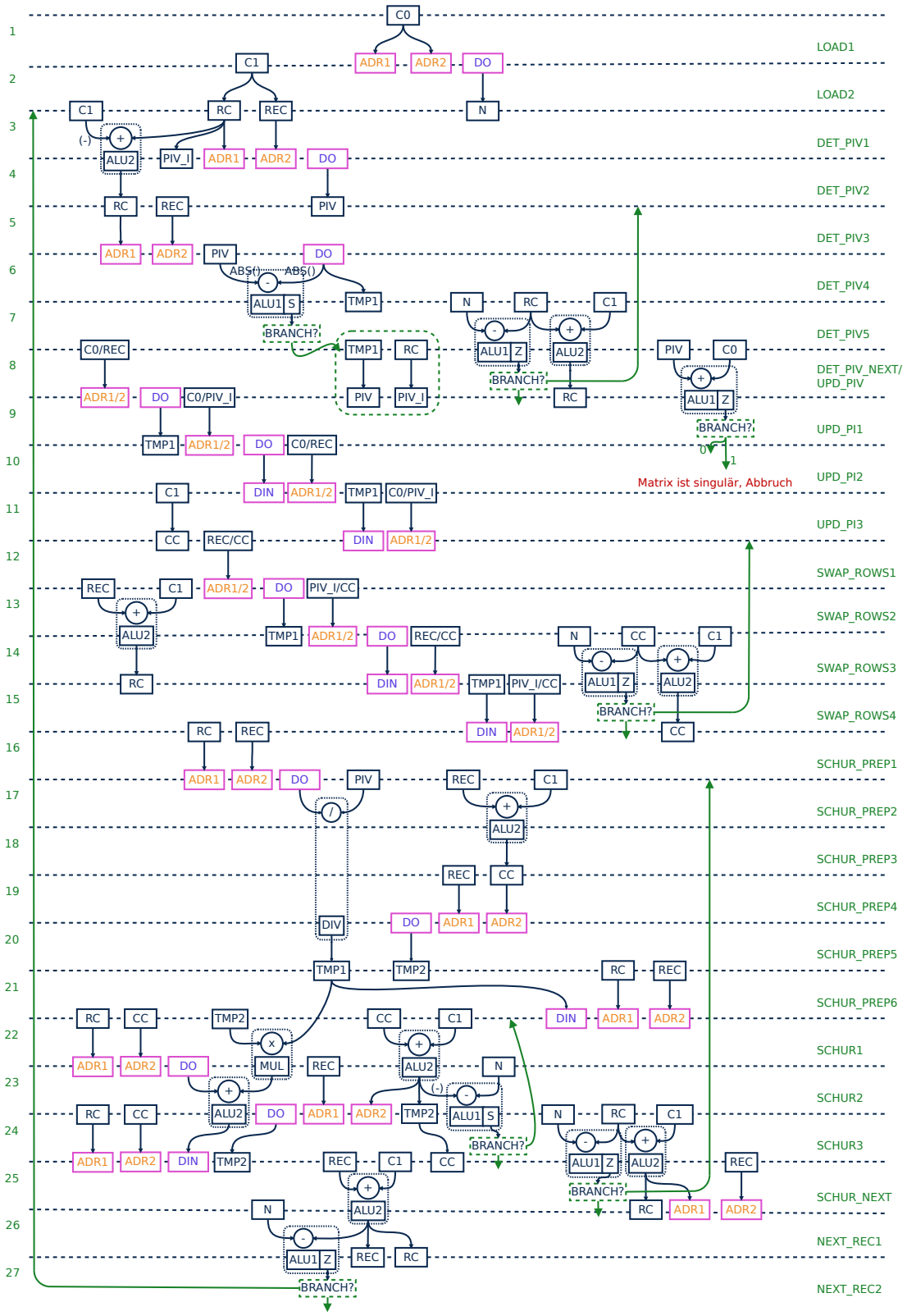

Abbildung 3.1: Datenflussgraph

gulär ist[1]. In diesem Fall existiert keine gültige Lösung und das Programm muss abgebrochen werden. Dieser Abbruch wird in der Implementierung durch den Übergang in einen speziellen Fehler-Zustand (ERROR) realisiert, aus dem dann einfach ohne weitere Fehlerbehandlung zurück in den IDLE Zustand übergegangen wird (siehe Abbildung 3.5).

3.2 DATENFLUSS-ANALYSE

Um den Ressourcenaufwand für die Verbindungen zwischen den Elementen des aus dem Datenflussgraphen abgeleiteten Datenpfades gering zu halten, wurde in Tabelle 3.1 tabellarisch protokolliert, welche Register/Speichereingänge im Datenflussgraph Ziele welcher anderen Register/des Speicherausganges sind. Eine Realisierung der Verbindungen mit dedizierten Multiplexern für alle Register/Speichereingänge die Ziele mehrerer Quellen sind, lässt sich hieraus direkt ablesen. Für eine einfache und wenig Chipfläche beanspruchende Realisierung wäre hier eine möglichst geringe Quellenanzahl pro Ziel, die idealerweise in jedem Fall auch noch eine Zweierpotenz ist, optimal. Letzteres war hier nicht in allen Fällen ohne Weiteres möglich.

Ziel		Quellen				
		1	2	3	4	5
MEM(ADR1)	←	C0	RC	REC	PIV_I	ALU2
MEM(ADR2)	←	C0	REC	PIV_I	CC	ALU2
MEM(DIN)	←	DO	TMP1	ALU2		
N	←	DO				
REC	←	C1	ALU2			
PIV	←	DO	TMP1			
PIV_I	←	RC				
RC	←	C1	ALU2			
CC	←	C1	ALU2	TMP2		
TMP1	←	DO	DIV			
TMP2	←	DO	ALU2			
ALU1(+)	←	PIV	N			
ALU1(-)	←	DO	RC	C0	CC	ALU2
ALU2(+)	←	RC	REC	CC	DO	
ALU2(-)	←	C1	MUL			
DIV(DIVIDEND)	←	DO				
DIV(DIVISOR)	←	PIV				
MUL(1)	←	TMP2				
MUL(2)	←	TMP1				

Tabelle 3.1: Datenfluss-Analyse

3.3 DATENPFAD

Abbildung 3.3 zeigt den mit dem Virtuoso Schematic Editor realisierten Datenpfad, dessen Aufbau aus Abbildung 3.1 und Tabelle 3.1 abgeleitet wurde. Dieser enthält zwei ALUs (ALU_FIXED32), einen Dividierer (DIV_FIXED64_signed) und einen Multiplizierer (MUL_FIXED32).

[1]Was sich leicht beweisen lässt, aber an dieser Stelle nicht weiter betrachtet wird.

Die Realisierung der Verbindungen im Datenpfad wurde entsprechend Tabelle 3.1 mit dedizierten Multiplexern für alle Rechenelemente, Register und Adresseingänge mit mehr als zwei zugehörigen Quellen realisiert.

Zu beachten sind die bereits zuvor erwähnten betragsbildenden Schaltungelemente, die den Eingängen der `ALU1` zugehörigen Multiplexer vorgeschaltet sind und deren interne Implementierung in Abbildung 3.2 gezeigt ist.

Wichtig sind auch die festen „Verdrahtungen" an den 64-Bit Ausgängen des Dividierers und des Multiplizierers, die eine Überführung ins 32-Bit Festkommaformat realisieren. An dieser Stelle wäre es auch möglich gewesen, durch zusätzliche Schaltungselemente Überläufe der Berechnungen zu erkennen und entsprechend auf diese zu reagieren. Um die Komplexität der FSM zu begrenzen, wurde auf diese Möglichkeit verzichtet.

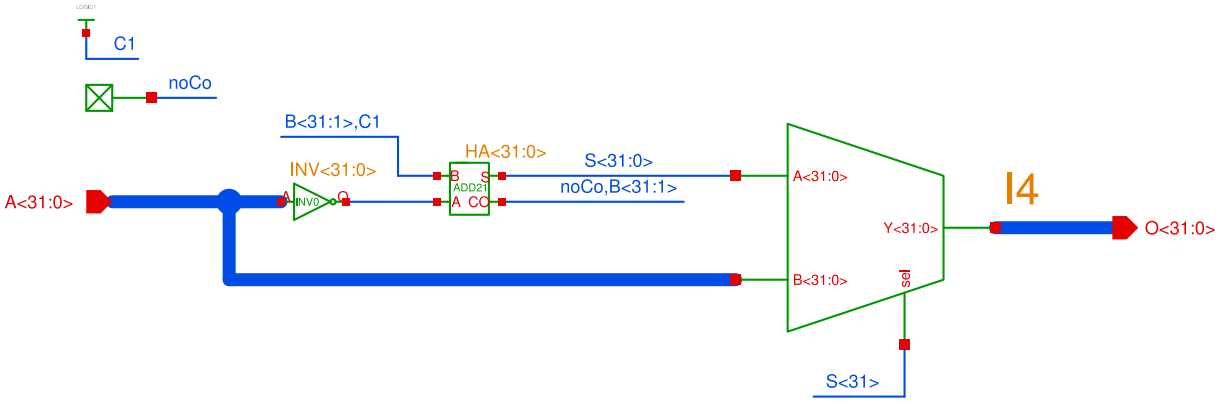

Abbildung 3.2: Kombinatorische Schaltung zur Bildung des Absolutwertes eines 32-Bit Signals

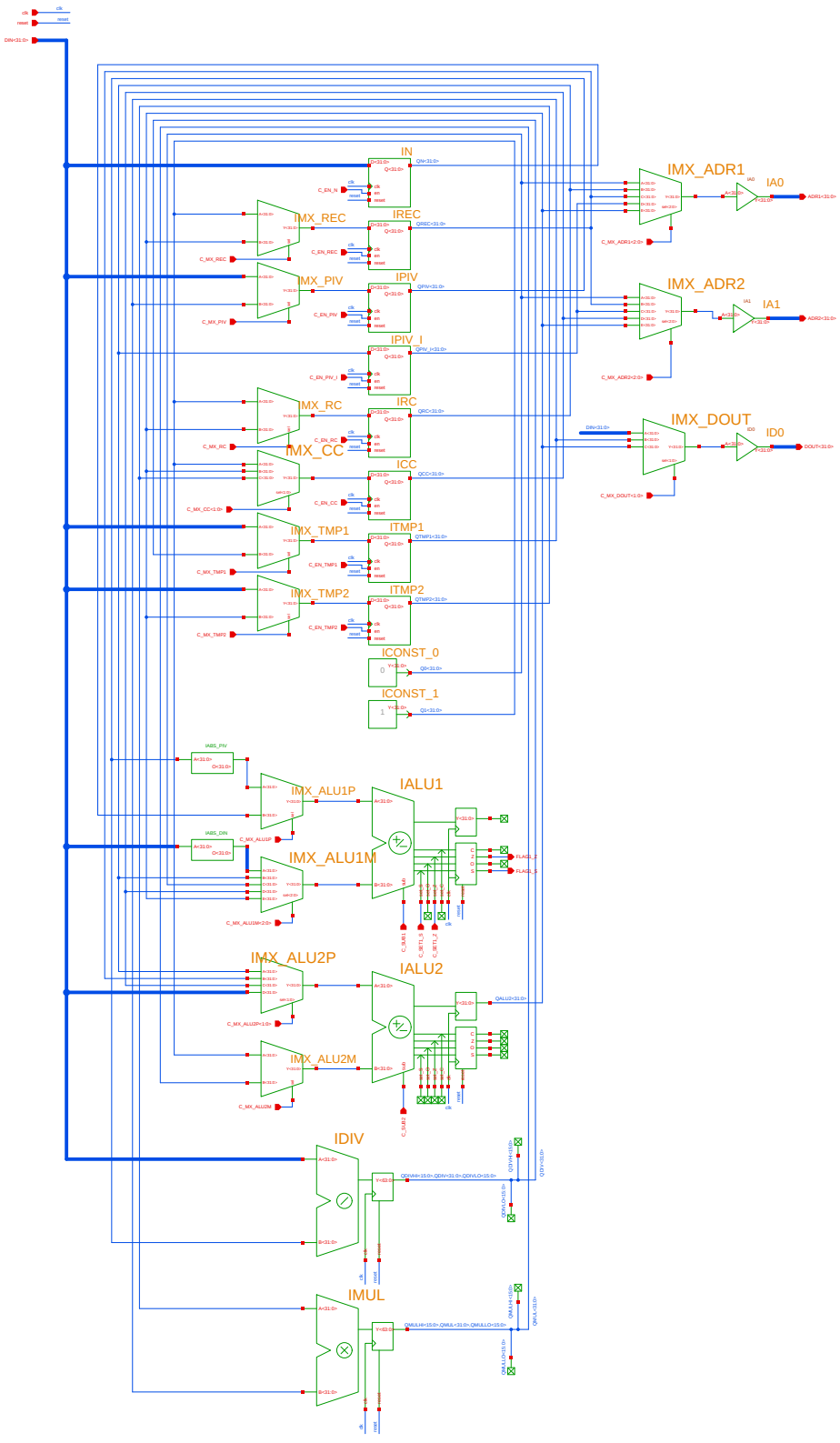

Abbildung 3.3: Datenpfad realisiert mit dem Virtuoso Schematic Editor

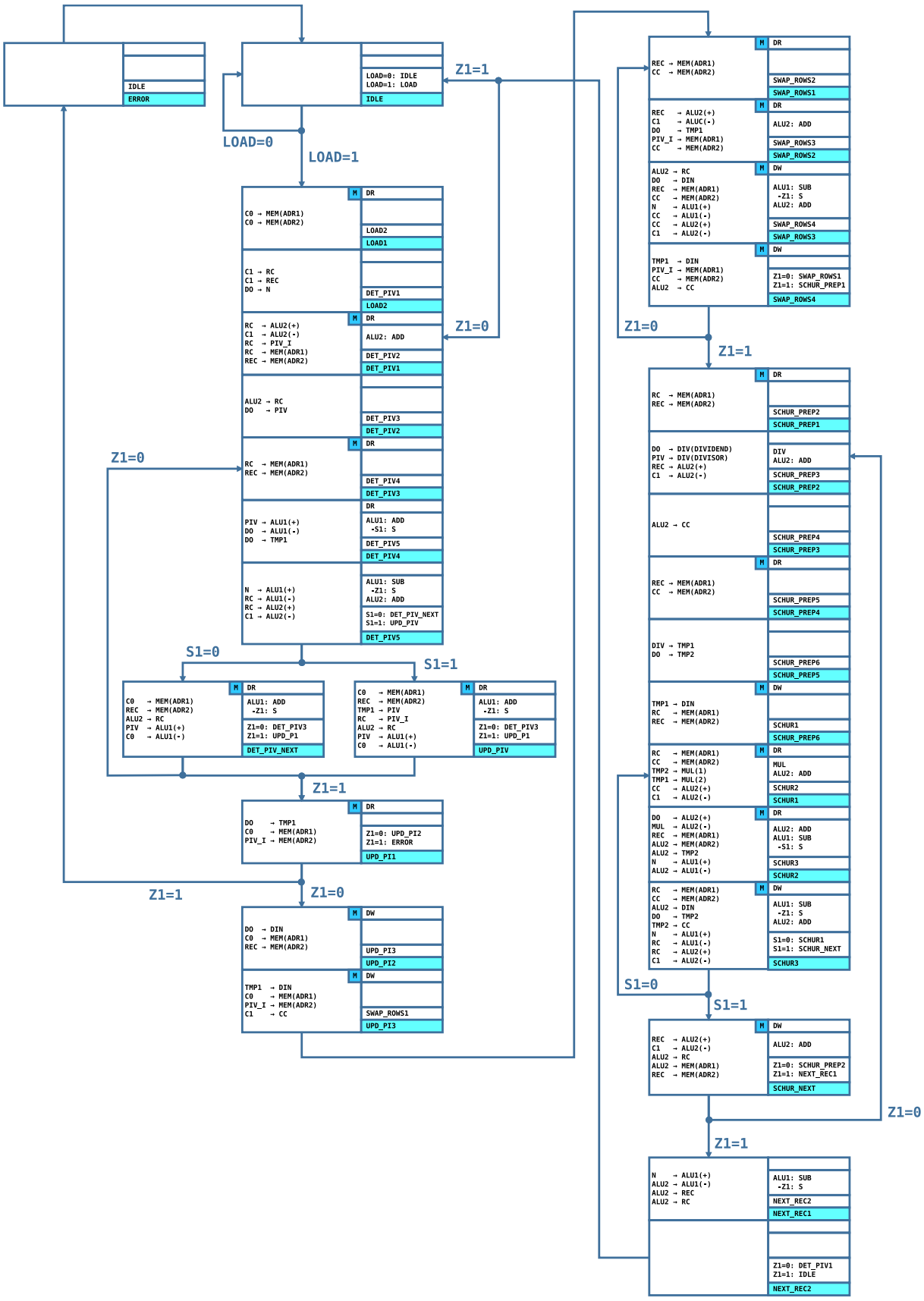

Abbildung 3.4: Register-Transfer-Folgen

3.4 FSM

3.4.1 REGISTERTRANSFERFOLGE

Abbildung 3.4 zeigt die aus dem Datenflussgraphen abgeleitete Registertransferfolge. Man beachte insbesondere den "Dummy-Zustand" `ERROR` in der oberen linken Ecke und die Steuerung der Verzweigungen durch die von `ALU1` generierten Flags.

In der Realisierung der FSM wird zusätzlich ein hier nicht explizit gezeigtes `reset`-Signal genutzt, wobei für jeden Zustand gilt, dass der Folgezustand für `reset = 1` unabhängig von den Werten der restlichen Flags `IDLE` ist.

Da allen Registern und Adresseingängen eindeutig ein eigener Eingangs-Multiplexer zugeordnet ist (sofern das Register/der Eingang Ziel von mehreren Quellen ist) und sich somit keine geteilten Busse im Datenpfad befinden, sind für die Registertransfers jeweils nur Quell- und Zielregister in der Form `QUELLE` → `ZIEL` angegeben.

3.4.2 ZUSTANDSÜBERGANGSDIAGRAMM

Abbildung 3.5 zeigt vereinfacht den Zustandsübergangsgraphen. Auch hier wird der Übergang von jedem Zustand zu `IDLE` bei gesetztem `reset`-Eingang nicht explizit aufgeführt.

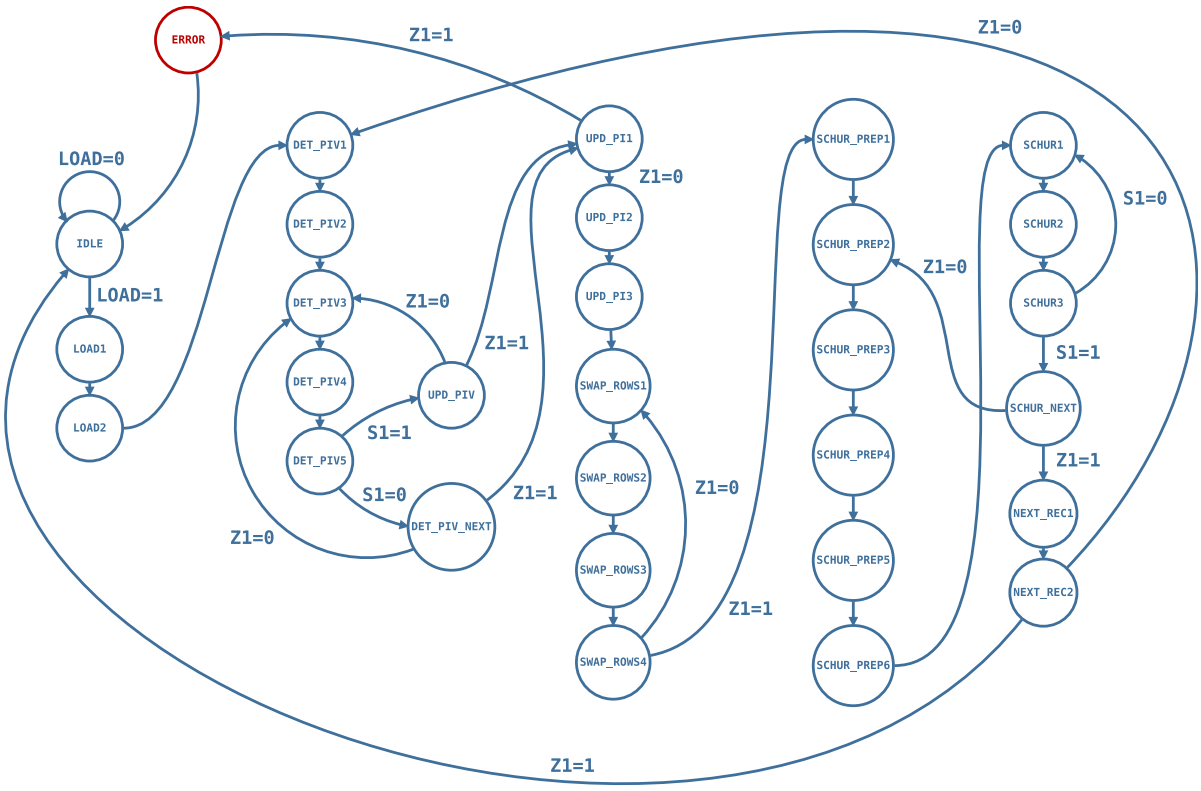

Abbildung 3.5: Zustandsübergangsdiagramm

Welche Zustände für welche Berechnungsschritte zuständig sind, sollte aus dem Programmablaufplan und dem Datenflussgraphen bereits leicht ersichtlich sein: Nach Laden von N aus dem Hauptspeicher und Setzen von Register `REC` auf 1 in `LOAD1-2` wird in `DET_PIV1-5` und `UPD_PIV`/`DET_PIV_NEXT` das Pivotelement bestimmt. In `UPD_PI1-3` wird π aktualisiert und in `SWAP_ROWS1-4` werden die Speicherzeilen ausgetauscht. In `SCHUR_PREP1-6` bzw. `SCHUR1-3` und

SCHUR_NEXT (der inneren Schleife) wird das Schur-Komplement gebildet und in NEXT_REC1-2 der Übergang in die nächste Rekursion gehandhabt.

Aus diesem Graphen wurde folgende Zustandsübergangstabelle abgeleitet. Dabei wurde für nicht genutzte Zustände jeweils der Folgezustand IDLE gewählt, sodass die FSM bei einem entsprechenden Hardwarefehler kontrolliert zurück in ihren Ausgangszustand übergeht. Z1 und S1 sind die *zero* und *signed* Flags von ALU1.

Zustand		**Flags**				**Folgezustand**	
Name	**Kodierung (x4 x3 x2 x1 x9)**	**reset**	**load**	**Z1**	**S1**	**Name**	**Kodierung (x4'x3'x2'x1'x0')**
*	*	1	*	*	*	IDLE	0 0 0 0 0
IDLE	0 0 0 0 0	0	0	*	*	IDLE	0 0 0 0 0
IDLE	0 0 0 0 0	0	1	*	*	LOAD1	0 0 0 0 1
LOAD1	0 0 0 0 1	0	*	*	*	LOAD2	0 0 0 1 0
LOAD2	0 0 0 1 0	0	*	*	*	DET_PIV1	0 0 0 1 1
DET_PIV1	0 0 0 1 1	0	*	*	*	DET_PIV2	0 0 1 0 0
DET_PIV2	0 0 1 0 0	0	*	*	*	DET_PIV3	0 0 1 0 1
DET_PIV3	0 0 1 0 1	0	*	*	*	DET_PIV4	0 0 1 1 0
DET_PIV4	0 0 1 1 0	0	*	*	*	DET_PIV5	0 0 1 1 1
DET_PIV5	0 0 1 1 1	0	*	*	0	DET_PIV_NEXT	0 1 0 0 0
DET_PIV5	0 0 1 1 1	0	*	*	1	UPD_PIV	0 1 0 0 1
DET_PIV_NEXT	0 1 0 0 0	0	*	0	*	DET_PIV3	0 0 1 0 1
DET_PIV_NEXT	0 1 0 0 0	0	*	1	*	UPD_PI1	0 1 0 1 0
UPD_PIV	0 1 0 0 1	0	*	0	*	DET_PIV3	0 0 1 0 1
UPD_PIV	0 1 0 0 1	0	*	1	*	UPD_PI1	0 1 0 1 0
UPD_PI1	0 1 0 1 0	0	*	0	*	UPD_PI2	0 1 0 1 1
UPD_PI1	0 1 0 1 0	0	*	1	*	ERROR	1 1 1 0 1
UPD_PI2	0 1 0 1 1	0	*	*	*	UPD_PI3	0 1 1 0 0
UPD_PI3	0 1 1 0 0	0	*	*	*	SWAP_ROWS1	0 1 1 0 1
SWAP_ROWS1	0 1 1 0 1	0	*	*	*	SWAP_ROWS2	0 1 1 1 0
SWAP_ROWS2	0 1 1 1 0	0	*	*	*	SWAP_ROWS3	0 1 1 1 1
SWAP_ROWS3	0 1 1 1 1	0	*	*	*	SWAP_ROWS4	1 0 0 0 0
SWAP_ROWS4	1 0 0 0 0	0	*	0	*	SWAP_ROWS1	0 1 1 0 1
SWAP_ROWS4	1 0 0 0 0	0	*	1	*	SCHUR_PREP1	1 0 0 0 1
SCHUR_PREP1	1 0 0 0 1	0	*	*	*	SCHUR_PREP2	1 0 0 1 0
SCHUR_PREP2	1 0 0 1 0	0	*	*	*	SCHUR_PREP3	1 0 0 1 1
SCHUR_PREP3	1 0 0 1 1	0	*	*	*	SCHUR_PREP4	1 0 1 0 0
SCHUR_PREP4	1 0 1 0 0	0	*	*	*	SCHUR_PREP5	1 0 1 0 1
SCHUR_PREP5	1 0 1 0 1	0	*	*	*	SCHUR_PREP6	1 0 1 1 0
SCHUR_PREP6	1 0 1 1 0	0	*	*	*	SCHUR1	1 0 1 1 1
SCHUR1	1 0 1 1 1	0	*	*	*	SCHUR2	1 1 0 0 0
SCHUR2	1 1 0 0 0	0	*	*	*	SCHUR3	1 1 0 0 1
SCHUR3	1 1 0 0 1	0	*	*	0	SCHUR1	1 0 1 1 1
SCHUR3	1 1 0 0 1	0	*	*	1	SCHUR_NEXT	1 1 0 1 0
SCHUR_NEXT	1 1 0 1 0	0	*	0	*	SCHUR_PREP2	1 0 0 1 0
SCHUR_NEXT	1 1 0 1 0	0	*	1	*	NEXT_REC1	1 1 0 1 1
NEXT_REC1	1 1 0 1 1	0	*	*	*	NEXT_REC2	1 1 1 0 0
NEXT_REC2	1 1 1 0 0	0	*	0	*	DET_PIV1	0 0 0 1 1
NEXT_REC2	1 1 1 0 0	0	*	1	*	IDLE	0 0 0 0 0
ERROR	1 1 1 0 1	0	*	*	*	IDLE	0 0 0 0 0
-	1 1 1 1 0	0	*	*	*	IDLE	0 0 0 0 0
-	1 1 1 1 1	0	*	*	*	IDLE	0 0 0 0 0

Tabelle 3.2: Zustandsübergangstabelle

3.4.3 REALISIERUNG

Mithilfe des Software-Tools *Logic Friday* Tools wurden aus der Tabelle 3.2 folgende vereinfachte Logikgleichungen für die Steuerung der Zustandsübergänge der FSM abgeleitet:

$$x4' = \overline{reset}\ (\overline{x4}\ x3\ x1\ (\overline{x2}\ \overline{x0}\ z1\ +\ x2\ x0)\ +\ x4\ (\overline{x2}\ (\overline{x0}\ (x3\ \overline{x1}\ +\ z1\ +\ x1)\ +\ x0)\ +\ \overline{x3}\ x2))$$

$$x3' = \overline{reset}\ (\overline{x3}\ (x4\ \overline{x2}\ \overline{x1}\ \overline{x0}\ \overline{z1}\ +\ x2\ x1\ x0)\ +\ x3\ (\overline{x4}\ (x2\ \overline{x1}\ x0\ +\ \overline{x0}\ (x1\ \overline{z1}\ +\ x2))\ +\ \overline{x2}\ (\overline{x1}\ (x4\ (x0\ s1\ +\ \overline{x0})\ +\ \overline{x4}\ z1)\ +\ \overline{x0}\ z1\ +\ x1\ x0)))$$

$$x2' = \overline{reset}\ (x2\ (\overline{x1}\ (\overline{x4}\ x3\ x0\ +\ \overline{x3})\ +\ \overline{x0}\ (\overline{x4}\ x3\ +\ \overline{x3}))\ +\ \overline{x2}\ (x1\ (\overline{x4}\ x3\ \overline{x0}\ z1\ +\ x0)\ +\ \overline{x1}\ (x4\ x3\ x0\ \overline{s1}\ +\ \overline{z1}\ (x4\ \overline{x3}\ \overline{x0}\ +\ \overline{x4}\ x3))))$$

$$x1' = \overline{reset}\ (x1\ \overline{x0}\ (x4\ \overline{x2}\ +\ \overline{x4}\ x2\ +\ \overline{x3})\ +\ \overline{x3}\ \overline{x1}\ x0\ +\ x3\ (\overline{x4}\ x1\ \overline{x0}\ \overline{z1}\ +\ \overline{x1}\ (\overline{x4}\ (\overline{x2}\ z1\ +\ x2\ x0)\ +\ x4\ (\overline{x2}\ x0\ (\overline{s1}\ +\ s1)\ +\ x2\ \overline{x0}\ \overline{z1}))))$$

$$x0' = \overline{reset}\ (\overline{x4}\ \overline{x3}\ x2\ x1\ s1\ +\ x3\ \overline{x2}\ \overline{x1}\ (x4\ x0\ \overline{s1}\ +\ \overline{x4}\ \overline{z1})\ +\ \overline{x0}\ (x4\ \overline{x2}\ z1\ +\ \overline{x3}\ (x4\ \overline{x2}\ \overline{x1}\ \overline{z1}\ +\ load\ +\ x1\ +\ x2)\ +\ x3\ (x4\ \overline{x1}\ (x2\ \overline{z1}\ +\ \overline{x2})\ +\ \overline{x4}\ (x1\ (\overline{x2}\ z1\ +\ \overline{z1})\ +\ x2))))$$

Außerdem wurde eine entsprechende Gatterschaltung generiert. Abbildung 3.6 zeigt deren Realisierung in Virtuoso. Dabei wird der Zustand der FSM über den `state`-Bus sowohl ausgegeben als auch zusammen mit den für die Zustandsübergänge relevanten Eingängen der FSM an die Eingänge der in einer separaten Zelle realisierten kombinatorischen Zustandsübergangslogik gelegt. Deren interner Aufbau aus (der `CORELIB`-Bibliothek entstammenden) Gattern ist in Abbildung 3.7 gezeigt. Die scheinbar wilkürliche Anordnung und Benennung der einzelnen Gatter orientiert sich dabei an dem von *Logic Friday* erstellten Gatter-Schaltplan, welcher hier nicht extra aufgeführt ist.

Die fünf Signale `state<0>` bis `state<4>`, aus denen der `state`-Bus zusammengesetzt ist, werden dabei mithilfe der D-Flip-Flops `Q0` bis `Q4` in Abhängigkeit von den Ausgänge der Zustandsübergangslogik bei jeder positiven Taktflanke aktualisiert. Bei gesetztem Eingang `set_state_en` wird die FSM stattdessen direkt in den an `set_state<4:0>` anliegendem Zustand gesetzt. Die asynchronen `reset`-Eingänge der D-Flip-Flops werden nicht genutzt, die FSM kann stattdessen synchron durch Setzten des regulären `reset`-Eingangs auf den Zustand `IDLE` zurückgesetzt werden (wobei `set_state_en` Vorrang vor `reset` hat).

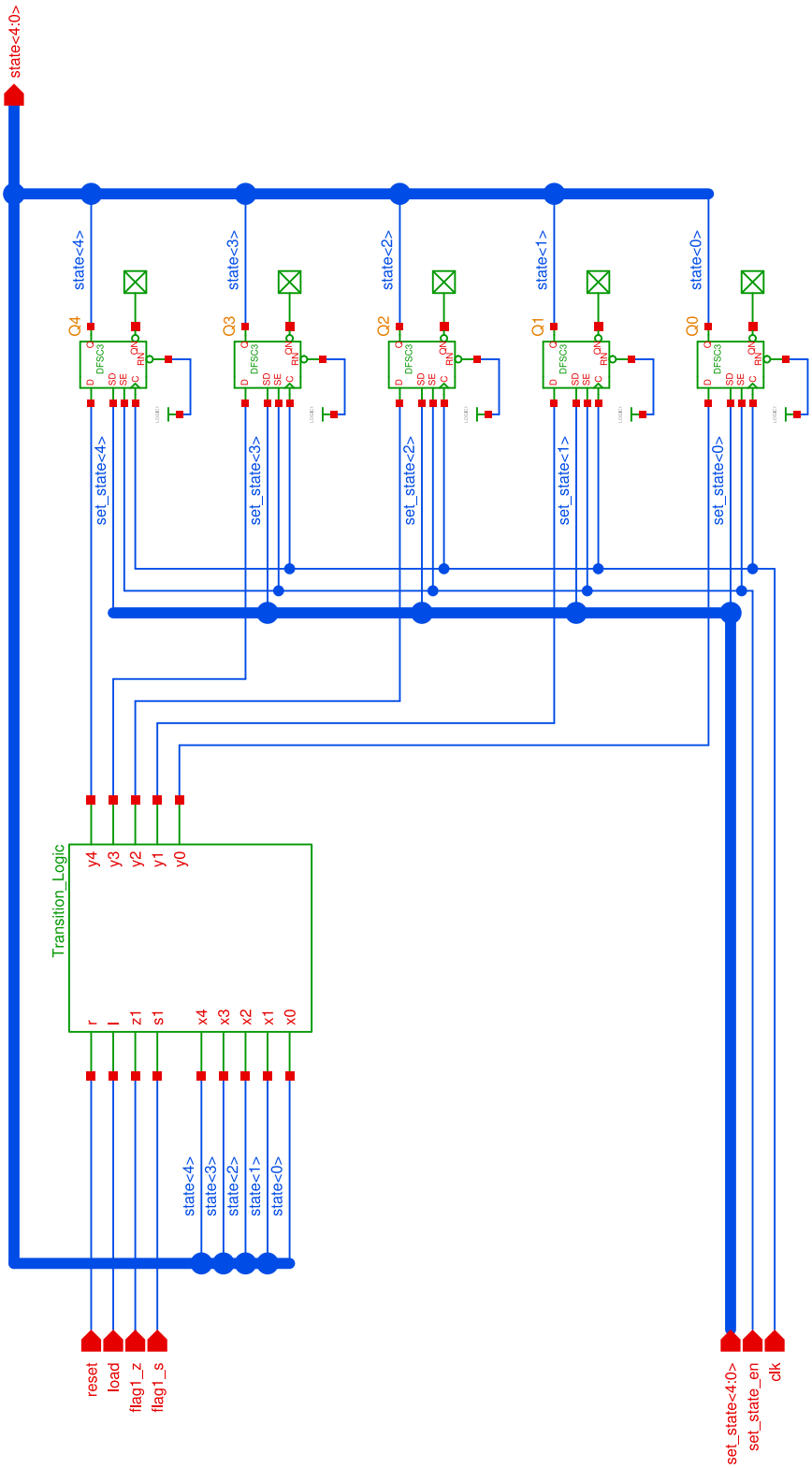

Abbildung 3.6: FSM-Schaltplan

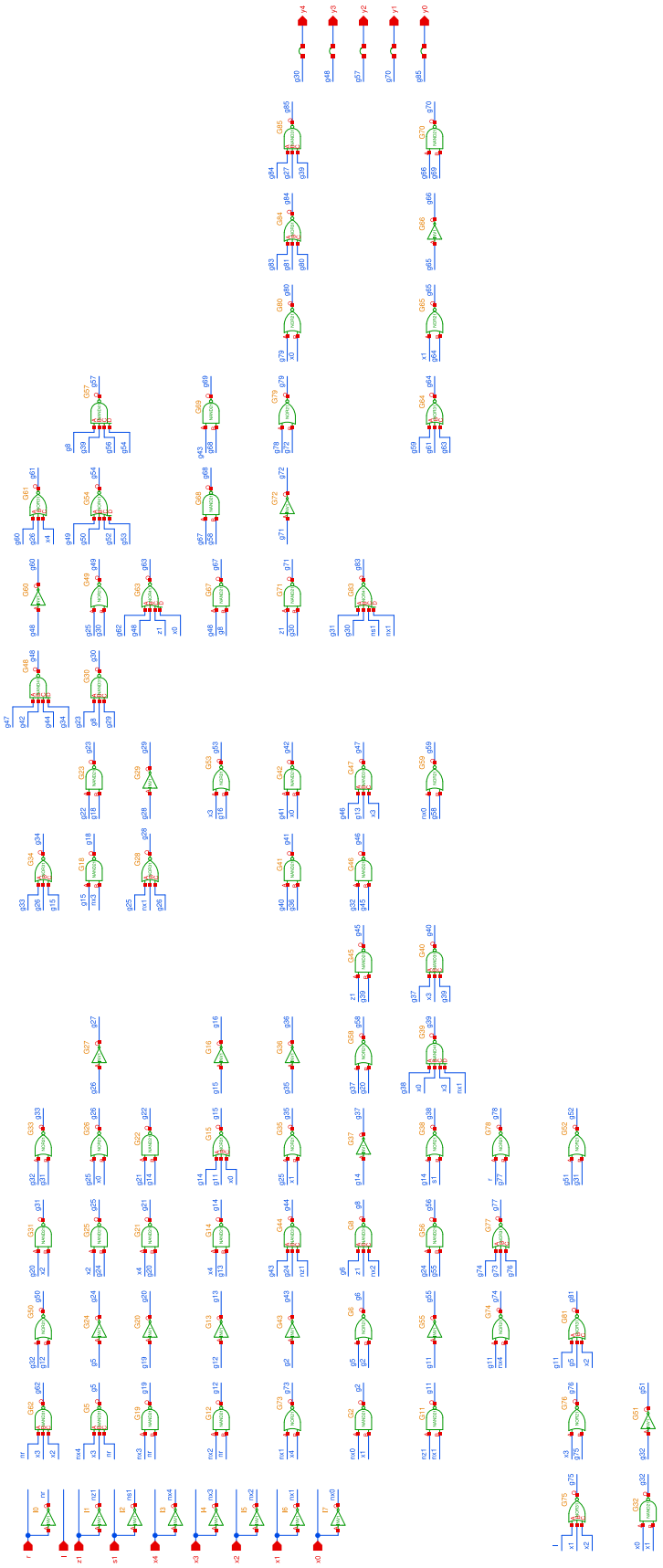

Abbildung 3.7: Kombinatorische Übergangslogik der FSM

3.5 GESAMTSCHALTUNG

Abbildung 3.8 zeigt die Top-Ansicht der Gesamtschaltung, bestehend aus FSM, Kontrolllogik und Datenpfad. Der Hauptspeicher wird hier nicht als Teil der Schaltung betrachtet. Gezeigt sind der Speicherausgang (`DIN`, da der Speicherausgang Eingang der Gesamtschaltung ist), die Adresseingänge (`ADR1` und `ADR2`) und der Dateneingang des Speichers (`DOUT`). Die bisher nicht beschriebene Kontrolllogik steuert in Ahängigkeit des aktuellen Zustandes die Übernahme anliegender Werte in die Register, die Operation und Flagberechnungen der Rechenelemente und die Multiplexer des Datenpfades sowie die read-/write-enable Eingänge des Speichers. Im Zustand `IDLE` wird außerdem der `ready` Ausgang der Kontrolllogik auf 1 gesetzt und somit die Betriebsbereitschaft der Schaltung bzw. der erfolgreiche Durchlauf des Algorithmus signalisiert.

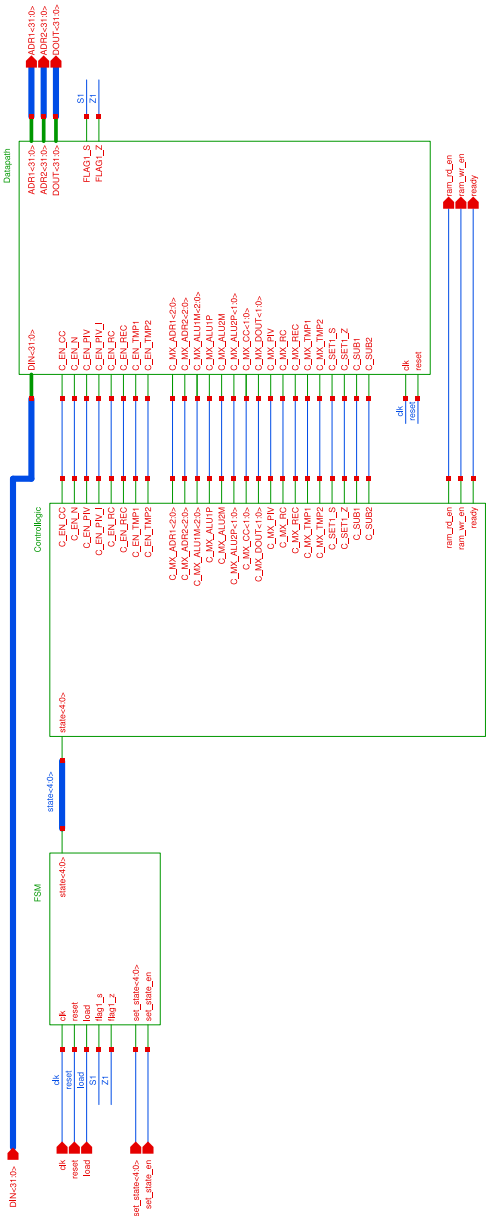

Abbildung 3.8: Top-Ansicht

4 TESTS

4.1 EINLEITUNG

Es existieren in der `LUP_Decomp`-Library zwei Testbenches. Eine zum separaten Testen der FSM (`LUP_Decomp_FSM_tb`) und eine zum Testen der Gesamtschaltung (`LUP_Decomp_Top_tb`). Zum Testen der Gesamtschaltung muss ein entsprechender initialer Speicherinhalt bereitgestellt werden. Dies geschieht, indem im zugehörigen run-Verzeichnis ein Verzeichnis mit dem Namen *mem* angelegt und in diesem eine hexadezimale Darstellung des initialen Speicherinhaltes in einer Datei, ebenfalls mit dem Namen `mem`, erstellt wird. Um diesen letzten Schritt zu erleichtern, kann das Skript `mem_gen.py` (siehe Listing 2) verwendet werden. Dieses sollte standardmäßig in einem Verzeichnis ausgeführt werden, welches eine Datei mit dem Namen *matrix* enthält. In dieser Datei muss eine quadratische Matrix in dezimaler Darstellung gespeichert sein, zum Beispiel wie folgt (Elemente mit Dezimalpunkt wären ebenfalls erlaubt):

Code Listing 4.1: `matrix` Textdatei

```
-8  2  4
-1 -2  2
-5 -5 -6
```

Es werden dann in diesem Verzeichnis automatisch eine Datei *mem* und eine weitere Datei *mem_expected* erzeugt. Dabei enthält *mem* den der Matrix entsprechenden initialen Speicherinhalt in hexadezimaler Form und *mem_expected* den nach Durchlauf der Testbench erwarteten Speicherinhalt, ebenfalls in hexadezimaler Darstellung. Um diese letzte Datei zu erzeugen, nutzt `mem_gen.py` die Referenzimplementierung der LUP-Dekomposition, welche im Skript `lup_decomposition.py` (siehe Listing 1) implementiert ist[1]. Wird `mem_gen.py` also ins Verzeichnis *mem* im run-Verzeichnis kopiert, können durch Anlegen und wiederholtes Anpassen einer *matrix* Datei leicht verschiedene initiale Speicherinhalte erzeugt und die dazugehörigen im Waveform-Viewer dargestellten Simulationsergebnisse sofort mit dem Inhalt von *mem_expected* verglichen werden. Ein solches Verzeichnis *mem* mit den beschriebenen Dateien findet sich im Verzeichnis *.../sim/LUP_Decomp_Top_tb_run*.

Zu beachten ist, dass der erwartete Speicherinhalt fast immer leicht vom tatsächlich durch die Simulation erzielten abweicht, da das Skript mit Gleitkommazahlen arbeitet.

4.2 FSM

In Listing 6 ist der Verilog Code der Tesbench aufgeführt, die verwendet wurde, um die korrekte Funktionsweise der FSM-Implementierung in Isolation zu verifizieren. In der Testbench wird für jeden Zustand jeweils für alle möglichen Kombinationen von Werten der FSM-Eingänge überprüft, ob der von der FSM erreichte Folgezustand mit dem in der Zustandsübergangstabelle 3.2 festgelegten übereinstimmt. Dies ist hier aufgrund der geringen Komplexität der FSM leicht möglich. Mit einer 5-Bit Zustandsbeschreibung und vier Eingängen (`reset`, `load`, `flag1_z` und `flag1_s`) gibt es $2^{5+4} = 512$ verschiedene zu testende Zustandsübergänge.

[1] Dieses Skript sollte sich demnach im gleichen Verzeichnis oder einem der durch die Umgebungsvariable `PYTHONPATH` definierten Verzeichnisse befinden.

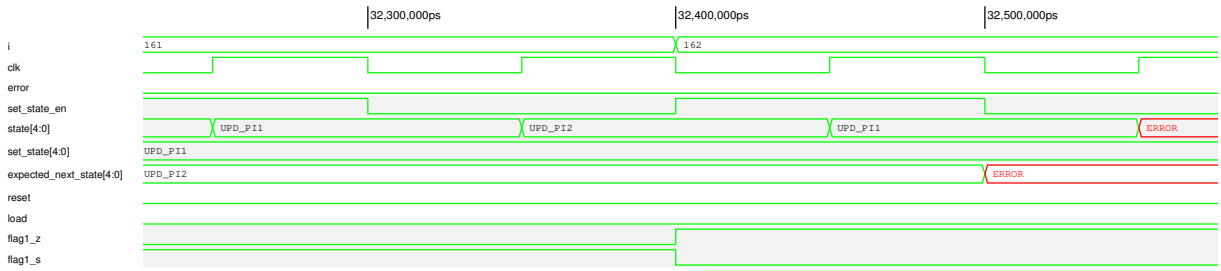

Abbildung 4.1: Ausgabe der FSM-Testbench (Ausschnitt)

In der Testbench werden in einer Schleife alle diese Kombinationen erzeugt. Der Zustand der FSM wird in jedem Schleifendurchlauf mit den `set_state` und `set_state_en` Eingängen synchron auf den Ausgangszustand der zu testenden Transition gesetzt. Anschließend wird der erwartete Folgezustand mithilfe eines `casez` statements bestimmt, dessen Fälle den Einträgen der Zustandsübergangstabelle 3.2 entsprechen. Stimmt nach einer weiteren clock-Flanke der Zustand der FSM nicht mit diesem erwarteten Zustand überein, wird ein `error`-Signal auf eins gesetzt und die Ausführung abgebrochen. So können fehlerhafte Zustandsübergänge leicht im Waveform-Output identifiziert werden[2].

Abbildung 4.1 zeigt einen Ausschnitt aus dem Waveform-Output der FSM-Testbench. `i` ist der Schleifenzähler der Testbench-Hauptschleife und läuft von 0 bis 511. Es ist zu sehen, wie in zwei Fällen der Zustand der FSM auf `UPD_PI1` gesetzt wird und für zwei verschiedene Kombinationen von Eingangswerten zunächst der erwartete nächste Zustand (`expected_next_state`) ermittelt wird, hier im ersten Fall `UPD_PI2` und im zweiten Fall `ERROR` (nicht zu verwechseln mit dem `error`-Signal). Diese stimmen in beiden Fällen mit dem anschließend während `set_state_en = 0` von der FSM erreichten Folgezustand überein.

Mit dieser Testbench konnte dann auch die korrekte Funktionalität der Gatterimplementierung der FSM eindeutig gezeigt werden. Für diese läuft die Simulation ohne Setzten des `error`-Signals komplett durch.

4.3 GESAMTSCHALTUNG

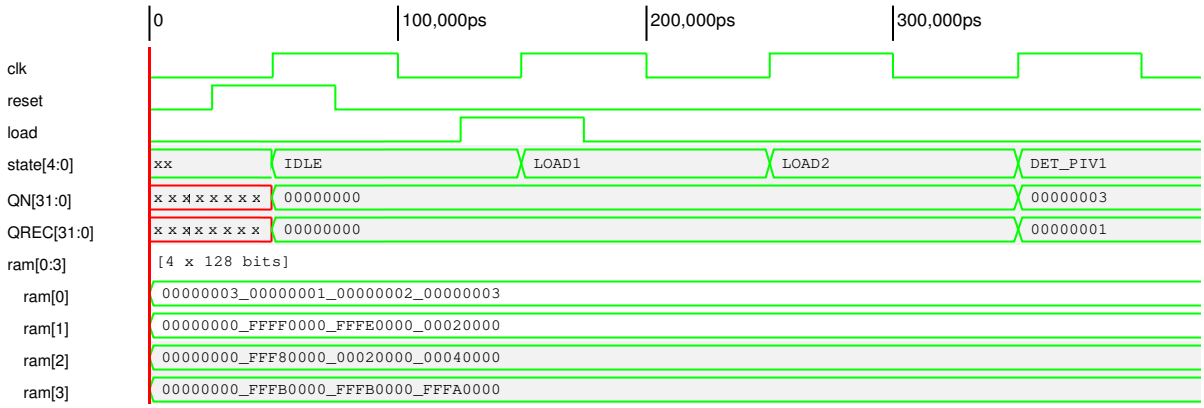

Abbildung 4.2: Initialer Ladevorgang

Die Gesamtschaltung wurde für mehrere Beispielmatrizen und beide Realisierungen der FSM getestet. Da der Programmablauf relativ komplex ist und die FSM-Realisierungen, wie mit der

[2] Besser wäre hier die Generierung von Fehlermeldungen für jede inkorrekte Transition mit `$display` tasks o.ä., dies war aufgrund von technischen Problemen jedoch nicht umsetzbar.

FSM-Testbench gezeigt, verhaltensgleich sind, wird hier nur ein Ausschnitt einer Beispiel-Simulation betrachtet. Diese orientiert sich an dem Rechenbeispiel zu Beginn dieser Arbeit. Der Sonderfall einer singulären Matrix wird ebenfalls nicht extra aufgeführt, da hier bloß (bewiesenermaßen funktionierende) einfache Sprünge in den ERROR- und zurück in den IDLE-Zustand stattfinden.

Abbildung 4.2 zeigt den Beginn der Programmausführung. In der Verilog-Implementierung des Hauptspeichers wird zunächst mittels $readmemh der anfängliche Speicherinhalt aus einer Textdatei eingelesen. Jede Zeile des Speichers ist hierbei eine Konkatenation mehrerer 32-Bit Wörter (hier zu einem 128-Bit Wort). Dabei ist die erste Zeile aus 32-Bit Ganzzahlen zusammengesetzt, welche die Reihen-/Spaltenzahl N der zu verarbeitenden Matrix und den Permutationsvektor π, welcher zu Beginn immer aus den aufsteigend angeordneten Elementen $1 \ldots N$ besteht, repräsentieren. Die folgenden Zeilen des Speichers enthalten die Matrix A, deren Elemente durch 32-Bit Festkommazahlen repräsentiert werden (hier mit jeweils 16 Bit für den Vor- und Nachkommaanteil). Dabei befindet sich das Matrixelement $a_{i,j}$ an der Speicheradresse [i,j] und das (ungenutzte) erste 32-Bit Wort jeder Speicherzeile ist $0x0$.

Vor Beginn der Programmausführung wird zunächst mittels des reset Signals die FSM in den Zustand IDLE versetzt. Anschließend wird synchron durch Setzen des load Signals die Programmausführung gestartet, wobei zunächst in den Zuständen LOAD1 und LOAD2 N aus dem Speicher in das zugehörige Register geladen und der Wert 1 in das Register REC geschrieben wird. Anschließend wird mit dem Zustand DET_PIV1 die Hauptschleife des Programms betreten. In Abbildung 4.3 ist die Bestimmung des Pivotelements für die erste Rekursion des Algorith-

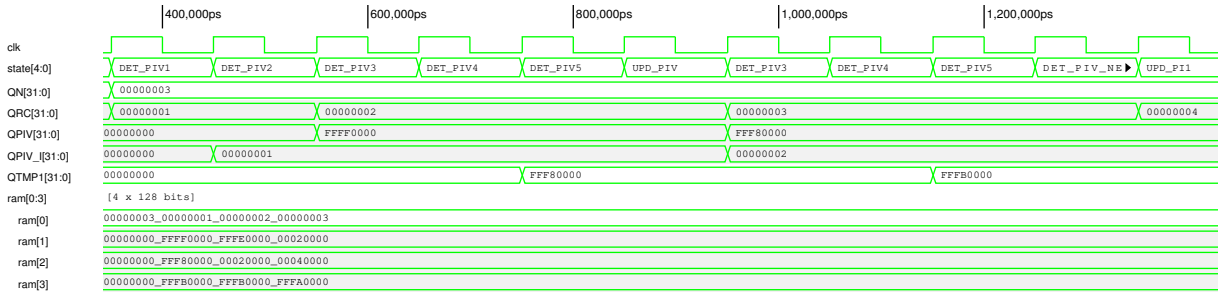

Abbildung 4.3: Bestimmung des Pivotelements

mus dargestellt. Das Register RC wird mit dem Wert der aktuellen Rekursion (hier 1) initialisiert, was der Adresse der ersten Zeile der in dieser Rekursion betrachteten Untermatrix entspricht.

Anschließend wird der Wert von RC in das Register PIV_I geschrieben, welches die Zeilenadresse des besten bisher gefundenen Pivotelements festhält. Direkt danach wird entsprechend das erste potentielle Pivotelement von der Speicheradresse [RC,REC] (also das Element $a_{REC,REC}$ in der ersten Zeile und Spalte der aktuellen Untermatrix) in das Register PIV geladen. Anschließend wird der Wert in RC um eins inkrementiert wird und in der Schleife beginnend mit dem Zustand DET_PIV3 wird das nächste potentielle Pivotelement von der Speicheradresse [RC,REC] ins Register TMP1 geladen. Gleichzeitig wird der Absolutwert des in PIV gespeicherten Elements mit dem des neu geladenen verglichen. Wenn dieser größer ist, geht die FSM vom Zustand DET_PIV5 in den Zustand UPD_PIV über, in welchem PIV mit dem Inhalt von TMP1 und PIV_I mit dem aktuellen Wert in RC überschrieben wird. Wenn das neu geladene Element betragsmäßig nicht größer ist, geht die FSM stattdessen in den Zustand DET_PIV_NEXT über, in dem PIV und PIV_I nicht modifiziert werden. Beides ist hier zu sehen.

Solange der Wert in RC kleiner als N ist, springt die FSM aus den Zuständen UPD_PIV und DET_PIV_NEXT zurück in den Zustand DET_PIV3. Dabei wird der Wert in RC erneut um eins inkrementiert. Sobald RC = N (hier 3) gilt, geht die FSM stattdessen in den Zustand UPD_PI1 über, was hier am Ende zu sehen ist. PIV enthält zu diesem Zeitpunkt dann das betragsmäßig größte Element der ersten Spalte der aktuellen Untermatrix (hier -8) und PIV_I dessen Zeilenindex (hier

2). Abbildung 4.4 zeigt die Aktualisierung des Permutationsvektors π nach Bestimmung des

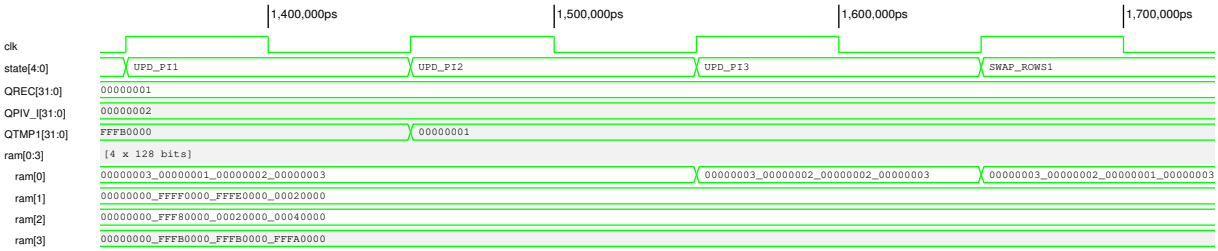

Abbildung 4.4: Aktualisierung des Permutationsvektors

Pivotelements. Dabei werden die Werte an den Speicherplätzen `[0,REC]` und `[0,PIV_I]` getauscht, wobei ersterer im Register `TMP1` zwischengespeichert wird. Jetzt werden die entspre-

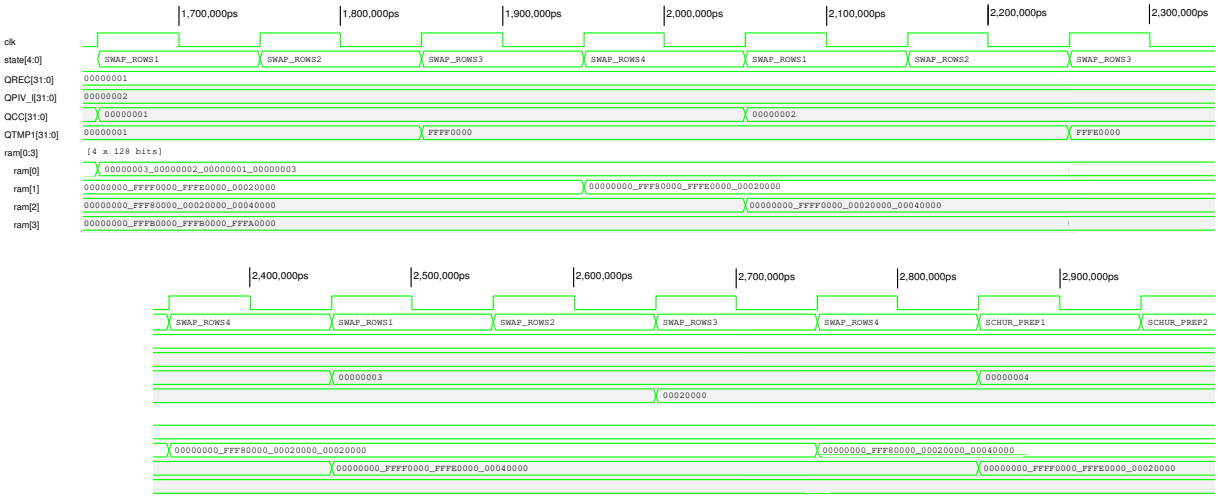

Abbildung 4.5: Reihentausch

chenden Zeilen der Matrix ausgetauscht. Dies geschieht in der von den Zuständen `SWAP_ROWS1` bis `SWAP_ROWS4` gebildeten Schleife. Das Register `CC` wird mit dem Wert eins initialisiert und nach jedem Schleifendurchlauf um eins inkrementiert, bis der Wert *N* erreicht ist und die Schleife abbricht. In jedem Schleifendurchlauf werden die Werte an den Speicherplätzen `[REC,CC]` und `[PIV_I,CC]` getauscht, wobei ersterer auch hier in `TMP1` zwischengespeichert wird.

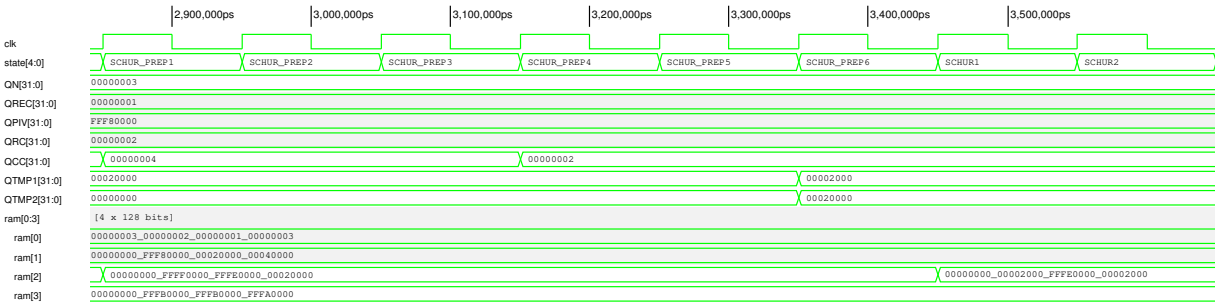

Abbildung 4.6: Vorbereitung Schur-Komplement

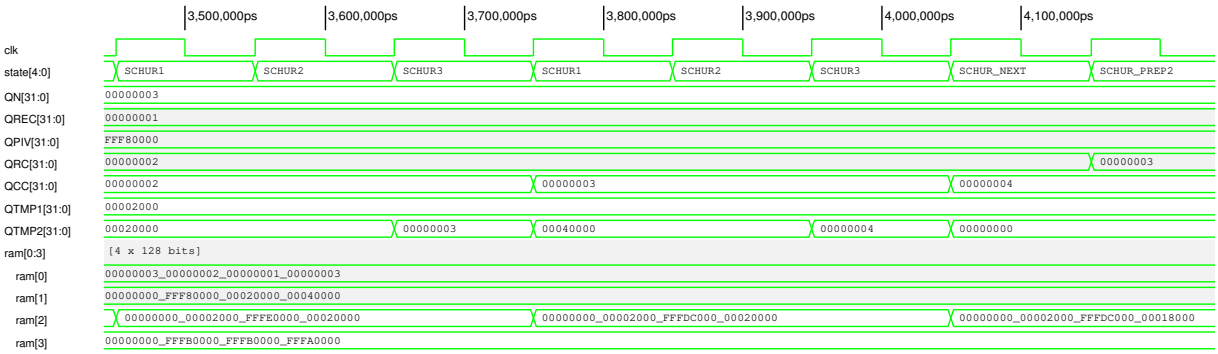

Abbildung 4.7: Bildung Schur-Komplement

Abbildungen 4.6 und 4.7 zeigen die Bildung des Schur-Komplements, auf die hier um der Knappheit wegen nicht noch einmal extra eingegangen wird.

Die gleichen Schritte werden anschließend nach Inkrementierung von `REC` auf der nächsten Untermatrix ausgeführt. Da der Ablauf bis auf die angepasste Speicheradressierung im Prinzip gleich ist wird hier nur noch das Endergebnis aufgeführt (Diese Waveform wurde zur Abwechslung unter Nutzung der Schematic-Realisierung der FSM generiert, die Zustände konnten daher in Simvision nicht auf ihre Bezeichner gemapt werden. Der finale Zustand `0x00` ist der `IDLE` Zustand):

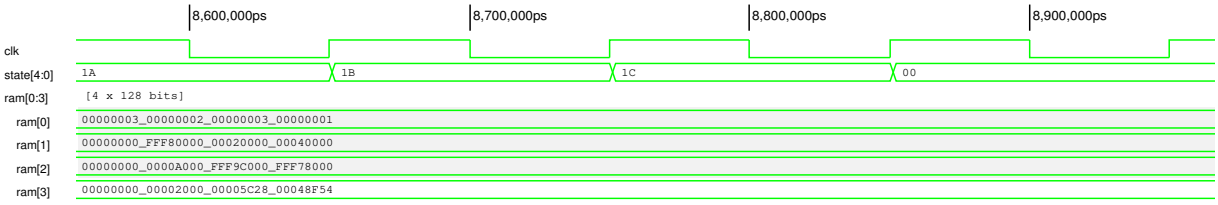

Abbildung 4.8: Endergebnis

Dieses lässt ich mit der von dem Skript aus Listing 2 erzeugten Datei *mem_expected* vergleichen:

Code Listing 4.2: `mem_expected`Textdatei

```
00000003_00000002_00000003_00000001
00000000_FFF80000_00020000_00040000
00000000_0000a000_FFF9c000_FFF78000
00000000_00002000_00005c29_00048F5c
```

Da letztere erzeugt wird, indem die Ergebnisse einer mit (64-Bit) Fließkommazahlen arbeitenden Referenzimplementierung in eine Hexadzimaldarstellung konvertiert werden, sind die kleinen Abweichungen in den Werten der Matrixelemente nicht verwunderlich.

Eine allgemeine Genauigkeitsbetrachtung würde sich an dieser Stelle schwierig gestalten, da die Resultate in Abhängigkeit von der Eingabematrix mehr oder weniger von der Referenzimplementierung abweichen können.

5 ZUSAMMENFASSUNG UND WERTUNG

Die Funktionalität des Entwurfes konnte mit den Simulationsergebnissen gezeigt werden. Die Implementierung ist vergleichsweise schnell und benötigt nur wenige Hardwarekomponenten. Einzig problematisch sind die eingeschränkte Genauigkeit und der eingeschränkte Darstellungsbereich der 32-Bit Festkommadarstellung, welcher theoretisch in Extremfällen zu Überläufen führen kann. Wenn diese Probleme in einer alternativen Implementierung umgangen werden sollen, kann dort zum Beispiel mit 64-Bit Fließkommazahlen gearbeitet werden, auf Kosten einer größeren und langsameren Schaltung. In jedem Fall wäre auch der explizite Test auf Überläufe bei den Rechenoperationen inklusive ausführlicher Fehlerbehandlung in einer real einsetzbaren Schaltung sinnvoll.

Insgesamt verlief die Anfertigung dieser Arbeit relativ problemlos. Die anfängliche Planung des Algorithmus und Optimierung des Datenflussgraphen hat dabei einen großen Anteil der insgesamt benötigten Zeit beansprucht. Trotzdem fand sich dann während der Implementierung ein Fehler im Datenpfad und ein großer Teil des Entwurfes musste korrigiert werden. Im Nachhinein wäre es auf jeden Fall sinnvoll gewesen, bei der Erstellung von Datenflussgraphen, Registertransferdiagramm und Datenpfad teilweise auf Softwaretools zurückzugreifen oder entsprechende eigene Tools anzufertigen, da die „manuelle" Ausführung dieses Teils des Entwurfsprozesses sehr fehleranfällig ist.

Teilweise problematisch war auch die Arbeit mit Cadence Virtuoso, da hier teilweise Probleme wie die Nichtausgabe von `$display` Meldungen oder nicht verwertbare Ergebnisse bei der RTL-Synthese auftraten. Diese Probleme ließen sich aufgrund der Komplexität und schwer durchdringbaren Dokumentation des Softwarepaketes wenn überhaupt nur mit Hilfe der Betreuer lösen und waren eher Hindernis als Lernerfahrung.

Insgesamt habe ich durch diese Arbeit jedoch einen guten Einblick in das Vorgehen beim Entwurf von digitalen Schaltungen erhalten.

6 QUELLCODE-LISTINGS

6.1 PYTHON

Listing 1: lup_decomposition.py, LUP Referenz-Implementierung

```python
#!/usr/bin/env python2

from copy import deepcopy

def lup_decomposition(A):
  B = deepcopy(A)
  n = len(B)
  pi = list(range(1, n+1))

  for i in range(0, n-1):
    p = i
    piv_max = abs(B[i][i])
    for j in range(i, n):
      tmp = abs(B[j][i])
      if tmp > piv_max:
        [p, piv_max] = [j, tmp]

    pi[i], pi[p] = pi[p], pi[i]
    B[i], B[p] = B[p], B[i]

    for k in range(i+1, n):
      B[k][i] /= B[i][i]

      for l in range(i+1, n):
        B[k][l] -= B[k][i] * B[i][l]

  return (B, pi)
```

Listing 2: mem_gen.py, Generierung formatierter Speicherinhalt-Textdatei

```python
#!/usr/bin/env python2

from lup_decomposition import lup_decomposition

def hex32_fp(d, frac_width=16):
  """Given a floating point number, obtain a 32-bit fixed point approximation of
  that number in hexadecimal notation.
  """
  fp = int(round(d * 2**frac_width))
  b_comp = fp if d >= 0 else (abs(fp) ^ 0xFFFFFFFF) + 1

  return hex(b_comp)[2:].zfill(8)

def write_mem(mem_file, A, pi=None):
  """Write memory layout corresponding with matrix A and permutation vector pi
  to a file.
  """
  if pi is None:
    pi = list(range(1, len(A) + 1))

  dim_mem = hex32_fp(len(A), 0)
  pi_mem = [hex32_fp(i, 0) for i in pi]
  A_mem = [[hex32_fp(entry) for entry in row] for row in A]

  with open(mem_file, 'w') as f:
```

```
    f.write('_'.join([dim_mem] + pi_mem) + '\n')

    for row in A_mem:
      f.write('_'.join(['0' * 8] + row) + '\n')

if __name__ == '__main__':
  with open("matrix", 'r') as f:
    A = [[float(entry) for entry in line.split()] for line in f]
    write_mem("mem", A)

    A_expected, pi_expected = lup_decomposition(A)
    write_mem("mem_expected", A_expected, pi_expected)
```

6.2 VERILOG

Listing 3: Speicher (LUP_Decomp_Memory)

```
module LUP_Decomp_Memory( clk, rd_en, wr_en, ADR1, ADR2, DIN, DOUT );

  //maximum dimensions of input square Matrix A
  parameter N_MAX = 3;
  parameter ADR1_WIDTH = $clog2(N_MAX+1);
  parameter ADR2_WIDTH = $clog2(N_MAX+1);

  input clk;
  input rd_en;
  input wr_en;
  input [31:0] DIN;
  input [31:0] ADR1;
  input [31:0] ADR2;

  output [31:0] DOUT;

  reg [(N_MAX+1)*32-1:0] ram [0:N_MAX];
  reg [31:0] DOUT;

  function [31:0] trunc_32(input [(N_MAX+1)*32-1:0] row);
    trunc_32 = row[(N_MAX+1)*32-1:(N_MAX)*32];
  endfunction

  always @(posedge clk)
    begin
      if (rd_en)
        DOUT = trunc_32(ram[ADR1[ADR1_WIDTH-1:0]] << ADR2*32);
      else if (wr_en)
        ram[ADR1[ADR1_WIDTH-1:0]] = ram[ADR1[ADR1_WIDTH-1:0]]
                                  & ~({32{1'b1}} << (N_MAX-ADR2)*32)
                                  | (DIN << (N_MAX-ADR2)*32);
    end

  initial $readmemh("mem/mem", ram);

endmodule
```

Listing 4: Finite State Machine (LUP_Decomp_FSM)

```
module LUP_Decomp_FSM( clk, reset, load, state, flag1_s, flag1_z, set_state, set_state_en );

  parameter IDLE         = 5'b00000;
  parameter LOAD1        = 5'b00001;
  parameter LOAD2        = 5'b00010;
  parameter DET_PIV1     = 5'b00011;
  parameter DET_PIV2     = 5'b00100;
  parameter DET_PIV3     = 5'b00101;
  parameter DET_PIV4     = 5'b00110;
  parameter DET_PIV5     = 5'b00111;
  parameter DET_PIV_NEXT = 5'b01000;
  parameter UPD_PIV      = 5'b01001;
```

```
13    parameter UPD_PI1      = 5'b01010;
14    parameter UPD_PI2      = 5'b01011;
15    parameter UPD_PI3      = 5'b01100;
16    parameter SWAP_ROWS1   = 5'b01101;
17    parameter SWAP_ROWS2   = 5'b01110;
18    parameter SWAP_ROWS3   = 5'b01111;
19    parameter SWAP_ROWS4   = 5'b10000;
20    parameter SCHUR_PREP1  = 5'b10001;
21    parameter SCHUR_PREP2  = 5'b10010;
22    parameter SCHUR_PREP3  = 5'b10011;
23    parameter SCHUR_PREP4  = 5'b10100;
24    parameter SCHUR_PREP5  = 5'b10101;
25    parameter SCHUR_PREP6  = 5'b10110;
26    parameter SCHUR1       = 5'b10111;
27    parameter SCHUR2       = 5'b11000;
28    parameter SCHUR3       = 5'b11001;
29    parameter SCHUR_NEXT   = 5'b11010;
30    parameter NEXT_REC1    = 5'b11011;
31    parameter NEXT_REC2    = 5'b11100;
32    parameter ERROR        = 5'b11101;
33
34    input clk;
35    input reset;
36    input load;
37    input flag1_z;
38    input flag1_s;
39    input [4:0] set_state;
40    input set_state_en;
41
42    output state;
43
44    reg [4:0] state;
45    reg [4:0] next_state;
46
47    always @(posedge clk)
48      if (set_state_en) begin
49        state <= set_state;
50      end
51      else if (reset) begin
52        state <= IDLE;
53      end
54      else begin
55        state <= next_state;
56      end
57
58    always @(state or flag1_s or flag1_z or load)
59      case(state)
60        IDLE:         if (load) begin
61                        next_state = LOAD1;
62                      end
63                      else begin
64                        next_state = IDLE;
65                      end
66        LOAD1:        next_state = LOAD2;
67        LOAD2:        next_state = DET_PIV1;
68        DET_PIV1:     next_state = DET_PIV2;
69        DET_PIV2:     next_state = DET_PIV3;
70        DET_PIV3:     next_state = DET_PIV4;
71        DET_PIV4:     next_state = DET_PIV5;
72        DET_PIV5:     if (flag1_s) begin
73                        next_state = UPD_PIV;
74                      end
75                      else begin
76                        next_state = DET_PIV_NEXT;
77                      end
78        DET_PIV_NEXT: if (flag1_z) begin
79                        next_state = UPD_PI1;
80                      end
81                      else begin
82                        next_state = DET_PIV3;
83                      end
84        UPD_PIV:      if (flag1_z) begin
85                        next_state = UPD_PI1;
```

```
                    end
                    else begin
                      next_state = DET_PIV3;
                    end
      UPD_PI1:      if (flag1_z) begin
                      next_state = ERROR;
                    end
                    else begin
                      next_state = UPD_PI2;
                    end
      UPD_PI2:      next_state = UPD_PI3;
      UPD_PI3:      next_state = SWAP_ROWS1;
      SWAP_ROWS1:   next_state = SWAP_ROWS2;
      SWAP_ROWS2:   next_state = SWAP_ROWS3;
      SWAP_ROWS3:   next_state = SWAP_ROWS4;
      SWAP_ROWS4:   if (flag1_z) begin
                      next_state = SCHUR_PREP1;
                    end
                    else begin
                      next_state = SWAP_ROWS1;
                    end
      SCHUR_PREP1:  next_state = SCHUR_PREP2;
      SCHUR_PREP2:  next_state = SCHUR_PREP3;
      SCHUR_PREP3:  next_state = SCHUR_PREP4;
      SCHUR_PREP4:  next_state = SCHUR_PREP5;
      SCHUR_PREP5:  next_state = SCHUR_PREP6;
      SCHUR_PREP6:  next_state = SCHUR1;
      SCHUR1:       next_state = SCHUR2;
      SCHUR2:       next_state = SCHUR3;
      SCHUR3:       if (flag1_s) begin
                      next_state = SCHUR_NEXT;
                    end
                    else begin
                      next_state = SCHUR1;
                    end
      SCHUR_NEXT:   if (flag1_z) begin
                      next_state = NEXT_REC1;
                    end
                    else begin
                      next_state = SCHUR_PREP2;
                    end
      NEXT_REC1:    next_state = NEXT_REC2;
      NEXT_REC2:    if (flag1_z) begin
                      next_state = IDLE;
                    end
                    else begin
                      next_state = DET_PIV1;
                    end
      ERROR:        next_state = IDLE;
      default:      next_state = IDLE;
    endcase

endmodule
```

Listing 5: Steuerlogik (`LUP_Decomp_Controllogic`)

```
module LUP_Decomp_Controllogic(
  state, ready, ram_rd_en, ram_wr_en,

  C_EN_N, C_EN_REC, C_EN_PIV, C_EN_PIV_I, C_EN_RC, C_EN_CC, C_EN_TMP1, C_EN_TMP2,

  C_MX_ADR1, C_MX_ADR2, C_MX_DOUT, C_MX_REC, C_MX_PIV, C_MX_RC, C_MX_CC,
  C_MX_TMP1, C_MX_TMP2, C_MX_ALU1P, C_MX_ALU1M,C_MX_ALU2P, C_MX_ALU2M,

  C_SUB1, C_SET1_S, C_SET1_Z, C_SUB2 );

  parameter IDLE         = 5'b00000;
  parameter LOAD1        = 5'b00001;
  parameter LOAD2        = 5'b00010;
  parameter DET_PIV1     = 5'b00011;
  parameter DET_PIV2     = 5'b00100;
```

```
16      parameter DET_PIV3     = 5'b00101;
17      parameter DET_PIV4     = 5'b00110;
18      parameter DET_PIV5     = 5'b00111;
19      parameter DET_PIV_NEXT = 5'b01000;
20      parameter UPD_PIV      = 5'b01001;
21      parameter UPD_PI1      = 5'b01010;
22      parameter UPD_PI2      = 5'b01011;
23      parameter UPD_PI3      = 5'b01100;
24      parameter SWAP_ROWS1   = 5'b01101;
25      parameter SWAP_ROWS2   = 5'b01110;
26      parameter SWAP_ROWS3   = 5'b01111;
27      parameter SWAP_ROWS4   = 5'b10000;
28      parameter SCHUR_PREP1  = 5'b10001;
29      parameter SCHUR_PREP2  = 5'b10010;
30      parameter SCHUR_PREP3  = 5'b10011;
31      parameter SCHUR_PREP4  = 5'b10100;
32      parameter SCHUR_PREP5  = 5'b10101;
33      parameter SCHUR_PREP6  = 5'b10110;
34      parameter SCHUR1       = 5'b10111;
35      parameter SCHUR2       = 5'b11000;
36      parameter SCHUR3       = 5'b11001;
37      parameter SCHUR_NEXT   = 5'b11010;
38      parameter NEXT_REC1    = 5'b11011;
39      parameter NEXT_REC2    = 5'b11100;
40      parameter ERROR        = 5'b11101;
41
42      input [4:0] state;
43
44      output C_EN_N;
45      output C_EN_REC;
46      output C_EN_PIV;
47      output C_EN_PIV_I;
48      output C_EN_RC;
49      output C_EN_CC;
50      output C_EN_TMP1;
51      output C_EN_TMP2;
52      output [2:0] C_MX_ADR1;
53      output [2:0] C_MX_ADR2;
54      output [1:0] C_MX_DOUT;
55      output C_MX_REC;
56      output C_MX_PIV;
57      output C_MX_RC;
58      output [1:0] C_MX_CC;
59      output C_MX_TMP1;
60      output C_MX_TMP2;
61      output C_MX_ALU1P;
62      output [2:0] C_MX_ALU1M;
63      output [1:0] C_MX_ALU2P;
64      output C_MX_ALU2M;
65      output C_SUB1;
66      output C_SET1_S;
67      output C_SET1_Z;
68      output C_SUB2;
69      output ram_rd_en;
70      output ram_wr_en;
71      output ready;
72
73      reg C_EN_N;
74      reg C_EN_REC;
75      reg C_EN_PIV;
76      reg C_EN_PIV_I;
77      reg C_EN_RC;
78      reg C_EN_CC;
79      reg C_EN_TMP1;
80      reg C_EN_TMP2;
81      reg [2:0] C_MX_ADR1;
82      reg [2:0] C_MX_ADR2;
83      reg [1:0] C_MX_DOUT;
84      reg C_MX_REC;
85      reg C_MX_PIV;
86      reg C_MX_RC;
87      reg [1:0] C_MX_CC;
88      reg C_MX_TMP1;
```

```
89  reg C_MX_TMP2;
90  reg C_MX_ALU1P;
91  reg [2:0] C_MX_ALU1M;
92  reg [1:0] C_MX_ALU2P;
93  reg C_MX_ALU2M;
94  reg C_SUB1;
95  reg C_SET1_S;
96  reg C_SET1_Z;
97  reg C_SUB2;
98  reg ram_rd_en;
99  reg ram_wr_en;
100 reg ready;
101
102 always @(state)
103   begin
104    C_EN_N     = 1'b0;
105    C_EN_REC   = 1'b0;
106    C_EN_PIV   = 1'b0;
107    C_EN_PIV_I = 1'b0;
108    C_EN_RC    = 1'b0;
109    C_EN_CC    = 1'b0;
110    C_EN_TMP1  = 1'b0;
111    C_EN_TMP2  = 1'b0;
112    C_MX_ADR1  = 3'b0;
113    C_MX_ADR2  = 3'b0;
114    C_MX_DOUT  = 2'b0;
115    C_MX_REC   = 1'b0;
116    C_MX_PIV   = 1'b0;
117    C_MX_RC    = 1'b0;
118    C_MX_CC    = 2'b0;
119    C_MX_TMP1  = 1'b0;
120    C_MX_TMP2  = 1'b0;
121    C_MX_ALU1P = 1'b0;
122    C_MX_ALU1M = 3'b0;
123    C_MX_ALU2P = 2'b0;
124    C_MX_ALU2M = 1'b0;
125    C_SUB1     = 1'b0;
126    C_SET1_S   = 1'b0;
127    C_SET1_Z   = 1'b0;
128    C_SUB2     = 1'b0;
129    ram_rd_en  = 1'b0;
130    ram_wr_en  = 1'b0;
131    ready      = 1'b0;
132
133     case (state)
134       IDLE:          begin
135                        ready = 1'b1;
136                      end
137       LOAD1:         begin
138                        ram_rd_en = 1'b1; //MEM(C0,C0)
139                      end
140       LOAD2:         begin
141                        C_EN_N = 1'b1; //D0 -> N
142                        C_EN_REC = 1'b1; //C1 -> REC
143                        C_EN_RC = 1'b1; //C1 -> RC
144                      end
145       DET_PIV1:      begin
146                        C_EN_PIV_I = 1'b1; //RC -> PIV_I
147
148                        C_MX_ADR1 = 3'b001; //RC
149                        C_MX_ADR2 = 3'b001; //REC
150                        ram_rd_en = 1'b1; //MEM(RC,REC)
151                      end
152       DET_PIV2:      begin
153                        C_EN_PIV = 1'b1; //D0 -> PIV
154                        C_MX_RC = 1'b1; //ALU2
155                        C_EN_RC = 1'b1; //ALU2 -> RC
156                      end
157       DET_PIV3:      begin
158                        C_MX_ADR1 = 3'b001; //RC
159                        C_MX_ADR2 = 3'b001; //REC
160                        ram_rd_en = 1'b1; //MEM(RC,REC)
161                      end
```

```
162         DET_PIV4:     begin
163                         C_EN_TMP1 = 1'b1; //DO -> TMP1
164
165                         C_SUB1 = 1'b1; //ALU2: SUB
166                         C_SET1_S = 1'b1; //S1
167                         //ALU2: ABS(PIV) - ABS(DO)
168                       end
169         DET_PIV5:     begin
170                         C_MX_ALU1P = 1'b1; //N
171                         C_MX_ALU1M = 3'b001; //RC
172                         C_SUB1 = 1'b1; //N - RC
173                         C_SET1_Z = 1'b1; //Z1
174                         //ALU2: RC + 1//ABS(PIV) - ABS(DO)
175                       end
176         DET_PIV_NEXT: begin
177                         C_MX_RC = 1'b1; //ALU2
178                         C_EN_RC = 1'b1; //ALU2 -> RC
179
180                         C_MX_ALU1M = 3'b010; //C0
181                         C_SET1_Z = 1'b1; //Z1
182                         //ALU1: ABS(PIV) + 0
183
184                         C_MX_ADR2 = 3'b001; //REC
185                         ram_rd_en = 1'b1; //MEM(C0,REC)
186                       end
187         UPD_PIV:      begin
188                         C_MX_PIV = 1'b1; //TMP1
189                         C_EN_PIV = 1'b1; //TMP1 -> PIV
190                         C_EN_PIV_I = 1'b1; //RC -> PIV_I
191                         C_MX_RC = 1'b1; //ALU2
192                         C_EN_RC = 1'b1; //ALU2 -> PIV
193
194                         C_MX_ALU1M = 3'b010; //C0
195                         C_SET1_Z = 1'b1; //Z1
196                         //ALU1: ABS(PIV) + 0
197
198                         C_MX_ADR2 = 3'b001; //REC
199                         ram_rd_en = 1'b1; //MEM(C0, REC)
200                       end
201         UPD_PI1:      begin
202                         C_EN_TMP1 = 1'b1; //DO -> TMP1
203                         C_MX_ADR2 = 3'b010; //PIV_I
204                         ram_rd_en = 1'b1; //MEM(C0,PIV_I)
205                       end
206         UPD_PI2:      begin
207                         C_MX_ADR2 = 3'b001; //REC
208                         ram_wr_en = 1'b1; //DO -> MEM(C0,REC)
209                       end
210         UPD_PI3:      begin
211                         C_EN_CC = 1'b1; //C1 -> CC
212
213                         C_MX_ADR2 = 3'b010; // PIV_I
214                         C_MX_DOUT = 2'b01; //TMP1
215                         ram_wr_en = 1'b1; //TMP1 -> MEM(C0,PIV_I)
216                       end
217         SWAP_ROWS1:   begin
218                         C_MX_ADR1 = 3'b010; //REC
219                         C_MX_ADR2 = 3'b011; //CC
220                         ram_rd_en = 1'b1; //MEM(REC,CC)
221                       end
222         SWAP_ROWS2:   begin
223                         C_EN_TMP1 = 1'b1; //DO -> TMP1
224
225                         C_MX_ALU2P = 3'b001; //REC
226                         //ALU2: REC + 1
227
228                         C_MX_ADR1 = 3'b011; //PIV_I
229                         C_MX_ADR2 = 3'b011; //CC
230                         ram_rd_en = 1'b1; //MEM(PIV_I,CC)
231                       end
232         SWAP_ROWS3:   begin
233                         C_MX_RC = 1'b1; //ALU2
234                         C_EN_RC = 1'b1; //ALU2 -> RC
```

```
235
236                     C_MX_ALU1P = 1'b1; //N
237                     C_MX_ALU1M = 3'b011; //CC
238                     C_SUB1 = 1'b1; //ALU1: SUB
239                     C_SET1_Z = 1'b1; //Z1
240                     //ALU1: N - CC
241                     C_MX_ALU2P = 3'b010; //CC
242                     //ALU2: CC + 1
243
244                     C_MX_ADR1 = 3'b010; //REC
245                     C_MX_ADR2 = 3'b011; //CC
246                     ram_wr_en = 1'b1; //DO -> MEM(REC,CC)
247                   end
248         SWAP_ROWS4: begin
249                     C_MX_CC = 2'b01; //ALU2
250                     C_EN_CC = 1'b1; //ALU2 -> CC
251
252                     C_MX_ADR1 = 3'b011; //PIV_I
253                     C_MX_ADR2 = 3'b011; //CC
254                     C_MX_DOUT = 2'b01; //TMP1
255                     ram_wr_en = 1'b1; //TMP1 -> MEM(PIV_I,CC)
256                   end
257         SCHUR_PREP1: begin
258                     C_MX_ADR1 = 3'b001; //RC
259                     C_MX_ADR2 = 3'b001; //REC
260                     ram_rd_en = 1'b1; //MEM(RC,REC)
261                   end
262         SCHUR_PREP2: begin
263                     C_MX_ALU2P = 2'b01; //REC
264                     //ALU2: REC+1
265                     //DIV: DO/PIV
266                   end
267         SCHUR_PREP3: begin
268                     C_MX_CC = 2'b01; //ALU2
269                     C_EN_CC = 1'b1; //ALU2 -> CC
270                   end
271         SCHUR_PREP4: begin
272                     C_MX_ADR1 = 3'b010; //REC
273                     C_MX_ADR2 = 3'b011; //CC
274                     ram_rd_en = 1'b1; //MEM(REC,CC)
275                   end
276         SCHUR_PREP5: begin
277                     C_MX_TMP1 = 1'b1; //DIV
278                     C_EN_TMP1 = 1'b1; //DIV -> TMP1
279                     C_EN_TMP2 = 1'b1; //DO -> TMP2
280                   end
281         SCHUR_PREP6: begin
282                     C_MX_ADR1 = 3'b001; //RC
283                     C_MX_ADR2 = 3'b001; //REC
284                     C_MX_DOUT = 2'b01; //TMP1
285                     ram_wr_en = 1'b1; //TMP1 -> MEM(RC,REC)
286                   end
287         SCHUR1:   begin
288                     C_MX_ALU2P = 2'b10; //CC
289                     //ALU: CC + 1
290
291                     C_MX_ADR1 = 3'b001; //RC
292                     C_MX_ADR2 = 3'b011; //CC
293                     ram_rd_en = 1'b1; //MEM(RC,CC)
294                   end
295         SCHUR2:   begin
296                     C_MX_TMP2 = 1'b1; //ALU2
297                     C_EN_TMP2 = 1'b1; //ALU2 -> TMP2
298
299                     C_MX_ALU1P = 1'b1; //N
300                     C_MX_ALU1M = 3'b100; //ALU2
301                     C_SUB1 = 1'b1; //ALU1: SUB3
302                     C_SET1_S = 1'b1; //S1
303                     //ALU1: N - ALU2
304                     C_MX_ALU2P = 2'b11; //DO
305                     C_MX_ALU2M = 1'b1; //MUL
306                     C_SUB2 = 1'b1; //ALU2: SUB
307                     //ALU2: DO-MUL
```

```
                        C_MX_ADR1 = 3'b010; //REC
                        C_MX_ADR2 = 3'b100; //ALU2
                        ram_rd_en = 1'b1; //MEM(REC,ALU2)
                      end
        SCHUR3:       begin
                        C_MX_CC = 2'b10; //TMP2
                        C_EN_CC = 1'b1; //TMP2 -> CC
                        C_EN_TMP2 = 1'b1; //D0 -> TMP2

                        C_MX_ALU1P = 2'b01; //N
                        C_MX_ALU1M = 3'b001; //RC
                        C_SUB1 = 1'b1; //ALU1: SUB
                        C_SET1_Z = 1'b1; //Z1
                        //ALU1: N - RC
                        //ALU2: RC + 1

                        C_MX_ADR1 = 3'b001; //RC
                        C_MX_ADR2 = 3'b011; //CC
                        C_MX_DOUT = 2'b11; //ALU2
                        ram_wr_en = 1'b1; //ALU2 -> MEM(RC,CC)
                      end
        SCHUR_NEXT:   begin
                        C_MX_RC = 1'b1; //ALU2
                        C_EN_RC = 1'b1; //ALU2 -> RC

                        C_MX_ALU2P = 2'b01; //REC
                        //ALU2: REC + 1

                        C_MX_ADR1 = 3'b100; //ALU2
                        C_MX_ADR2 = 3'b001; //REC
                        ram_rd_en = 1'b1; //MEM(ALU2,REC)
                      end
        NEXT_REC1:    begin
                        C_MX_REC = 1'b1; //ALU2
                        C_EN_REC = 1'b1; //ALU2 -> REC

                        C_MX_RC = 1'b1; //ALU2
                        C_EN_RC = 1'b1; //ALU2 -> RC

                        C_MX_ALU1P = 1'b1; //N
                        C_MX_ALU1M = 3'b100; //ALU2
                        C_SUB1 = 1'b1; //ALU1: SUB
                        C_SET1_Z = 1'b1; //Z1
                        //ALU1: N - ALU2
                      end
      endcase
    end

endmodule
```

Listing 6: FSM Testbench (`LUP_Decomp_FSM_tb`)

```
module LUP_Decomp_FSM_tb ( error );

  parameter CYCLE = 100;

  parameter IDLE          = 5'b00000;
  parameter LOAD1         = 5'b00001;
  parameter LOAD2         = 5'b00010;
  parameter DET_PIV1      = 5'b00011;
  parameter DET_PIV2      = 5'b00100;
  parameter DET_PIV3      = 5'b00101;
  parameter DET_PIV4      = 5'b00110;
  parameter DET_PIV5      = 5'b00111;
  parameter DET_PIV_NEXT  = 5'b01000;
  parameter UPD_PIV       = 5'b01001;
  parameter UPD_PI1       = 5'b01010;
  parameter UPD_PI2       = 5'b01011;
  parameter UPD_PI3       = 5'b01100;
  parameter SWAP_ROWS1    = 5'b01101;
```

```
19    parameter SWAP_ROWS2   = 5'b01110;
20    parameter SWAP_ROWS3   = 5'b01111;
21    parameter SWAP_ROWS4   = 5'b10000;
22    parameter SCHUR_PREP1  = 5'b10001;
23    parameter SCHUR_PREP2  = 5'b10010;
24    parameter SCHUR_PREP3  = 5'b10011;
25    parameter SCHUR_PREP4  = 5'b10100;
26    parameter SCHUR_PREP5  = 5'b10101;
27    parameter SCHUR_PREP6  = 5'b10110;
28    parameter SCHUR1       = 5'b10111;
29    parameter SCHUR2       = 5'b11000;
30    parameter SCHUR3       = 5'b11001;
31    parameter SCHUR_NEXT   = 5'b11010;
32    parameter NEXT_REC1    = 5'b11011;
33    parameter NEXT_REC2    = 5'b11100;
34    parameter ERROR        = 5'b11101;
35    parameter UNUSED1      = 5'b11110;
36    parameter UNUSED2      = 5'b11111;
37
38    output error;
39
40    reg clk;
41    reg reset;
42    reg load;
43    reg flag1_z;
44    reg flag1_s;
45    reg [4:0] set_state;
46    reg set_state_en;
47    reg [4:0] expected_next_state;
48    reg error;
49
50    wire [4:0] state;
51
52    LUP_Decomp_FSM DUT(
53      .clk(clk),
54      .reset(reset),
55      .load(load),
56      .state(state),
57      .flag1_z(flag1_z),
58      .flag1_s(flag1_s),
59      .set_state(set_state),
60      .set_state_en(set_state_en)
61    );
62
63    initial begin
64      clk = 1'b0;
65      reset = 1'b0;
66      load = 1'b0;
67      flag1_z = 1'b0;
68      flag1_s = 1'b0;
69      set_state = 5'b00000;
70      set_state_en = 1'b0;
71      expected_next_state = 5'b00000;
72      error = 1'b0;
73    end
74
75    always #(CYCLE/2) clk = ~clk;
76
77    integer i;
78    initial begin
79      for (i = 0; i < 512; i = i + 1) begin
80        {set_state, reset, load, flag1_z, flag1_s} = i;
81
82        //set state
83        set_state_en = 1'b1;
84        #(CYCLE);
85
86        casez(i) //get expected next state
87          {{5{1'b?}},4'b1???}: expected_next_state = IDLE;
88          {IDLE,4'b00??}: expected_next_state = IDLE;
89          {IDLE,4'b01??}: expected_next_state = LOAD1;
90          {LOAD1,4'b0???}: expected_next_state = LOAD2;
91          {LOAD2,4'b0???}: expected_next_state = DET_PIV1;
```

```
        {DET_PIV1,4'b0???}: expected_next_state = DET_PIV2;
        {DET_PIV2,4'b0???}: expected_next_state = DET_PIV3;
        {DET_PIV3,4'b0???}: expected_next_state = DET_PIV4;
        {DET_PIV4,4'b0???}: expected_next_state = DET_PIV5;
        {DET_PIV5,4'b0??0}: expected_next_state = DET_PIV_NEXT;
        {DET_PIV5,4'b0??1}: expected_next_state = UPD_PIV;
        {DET_PIV_NEXT,4'b0?0?}: expected_next_state = DET_PIV3;
        {DET_PIV_NEXT,4'b0?1?}: expected_next_state = UPD_PI1;
        {UPD_PIV,4'b0?0?}: expected_next_state = DET_PIV3;
        {UPD_PIV,4'b0?1?}: expected_next_state = UPD_PI1;
        {UPD_PI1,4'b0?0?}: expected_next_state = UPD_PI2;
        {UPD_PI1,4'b0?1?}: expected_next_state = ERROR;
        {UPD_PI2,4'b0???}: expected_next_state = UPD_PI3;
        {UPD_PI3,4'b0???}: expected_next_state = SWAP_ROWS1;
        {SWAP_ROWS1,4'b0???}: expected_next_state = SWAP_ROWS2;
        {SWAP_ROWS2,4'b0???}: expected_next_state = SWAP_ROWS3;
        {SWAP_ROWS3,4'b0???}: expected_next_state = SWAP_ROWS4;
        {SWAP_ROWS4,4'b0?0?}: expected_next_state = SWAP_ROWS1;
        {SWAP_ROWS4,4'b0?1?}: expected_next_state = SCHUR_PREP1;
        {SCHUR_PREP1,4'b0???}: expected_next_state = SCHUR_PREP2;
        {SCHUR_PREP2,4'b0???}: expected_next_state = SCHUR_PREP3;
        {SCHUR_PREP3,4'b0???}: expected_next_state = SCHUR_PREP4;
        {SCHUR_PREP4,4'b0???}: expected_next_state = SCHUR_PREP5;
        {SCHUR_PREP5,4'b0???}: expected_next_state = SCHUR_PREP6;
        {SCHUR_PREP6,4'b0???}: expected_next_state = SCHUR1;
        {SCHUR1,4'b0???}: expected_next_state = SCHUR2;
        {SCHUR2,4'b0???}: expected_next_state = SCHUR3;
        {SCHUR3,4'b0??0}: expected_next_state = SCHUR1;
        {SCHUR3,4'b0??1}: expected_next_state = SCHUR_NEXT;
        {SCHUR_NEXT,4'b0?0?}: expected_next_state = SCHUR_PREP2;
        {SCHUR_NEXT,4'b0?1?}: expected_next_state = NEXT_REC1;
        {NEXT_REC1,4'b0???}: expected_next_state = NEXT_REC2;
        {NEXT_REC2,4'b0?0?}: expected_next_state = DET_PIV1;
        {NEXT_REC2,4'b0?1?}: expected_next_state = IDLE;
        {ERROR,4'b0???}: expected_next_state = IDLE;
        default: expected_next_state = IDLE;
      endcase

      //let FSM make transition
      set_state_en = 1'b0;
      #(CYCLE);

      //check if resulting state matches with expected state
      if (state != expected_next_state) begin
        error = 1'b1;
        $finish;
      end
    end

    $finish;
  end

endmodule
```

Listing 7: Testbench (`LUP_Decomp_Top_tb`)

```
module LUP_Decomp_Top_tb ( load );

  parameter CYCLE = 100;

  wire  [31:0] DATA_MEM_RD;
  wire  [31:0] DATA_MEM_WR;
  wire  [31:0] DATA_MEM_ADR1;
  wire  [31:0] DATA_MEM_ADR2;
  wire  ram_wr_en;
  wire  ram_rd_en;
  wire  ready;

  output load;

  reg clk;
```

```
  reg reset;
  reg load;
  reg [4:0] set_state;
  reg set_state_en;

  LUP_Decomp_Memory mem (
    .clk(clk),
    .rd_en(ram_rd_en),
    .wr_en(ram_wr_en),
    .ADR1(DATA_MEM_ADR1),
    .ADR2(DATA_MEM_ADR2),
    .DIN(DATA_MEM_WR),
    .DOUT(DATA_MEM_RD)
  );

  LUP_Decomp DUT (
    .clk(clk),
    .reset(reset),
    .load(load),
    .ready(ready),
    .set_state(set_state),
    .set_state_en(set_state_en),
    .ram_rd_en(ram_rd_en),
    .ram_wr_en(ram_wr_en),
    .ADR1(DATA_MEM_ADR1),
    .ADR2(DATA_MEM_ADR2),
    .DIN(DATA_MEM_RD),
    .DOUT(DATA_MEM_WR)
  );

  initial begin
    clk = 1'b0;
    reset = 1'b0;
    load = 1'b0;
    set_state = 5'b00000;
    set_state_en = 1'b0;
  end

  always #(CYCLE/2) clk = ~clk;

  initial begin
    #(CYCLE/4)
    reset = 1'b1;
    #(CYCLE/2)
    reset = 1'b0;
    #(CYCLE/2)
    load = 1'b1;
    #(CYCLE/2)
    load = 1'b0;

    while (ready == 0) begin
      #(CYCLE);
    end

    #(CYCLE)
    $finish;
  end

endmodule
```

APPENDIX: VERZEICHNISSE

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LITERATURVERZEICHNIS

[COR01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. Introduction to Algorithms (2nd ed.). McGraw-Hill Higher Education