TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Elektrotechnik und Informationstechnik Institut für Grundlagen der Elektrotechnik und Elektronik

Belegarbeit

Schaltkreis- und Systemenentwurf

Omar Abu Rashed

Matrikelnummer: 4973533

Betreuer

Dr.-Ing. Sebastian Höppner

Eingereicht am: 5. Mai 2024

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich das vorliegende Dokument mit dem Titel *Schaltkreis- und Systemenentwurf* selbstständig und ohne unzulässige Hilfe Dritter verfasst habe. Es wurden keine anderen als die in diesem Dokument 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 des vorliegenden Dokumentes beteiligt. Mir ist bekannt, dass die Nichteinhaltung dieser Erklärung zum nachträglichen Entzug des Hochschulabschlusses führen kann.

Dresden, 5. Mai 2024

Omar Abu Rashed

Inhaltsverzeichnis

1. Einleitung

In diesem Beleg wird der Kalenderalgorithmus vorgestellt, der auch als Zeller-Kongruenz bekannt ist. Es wurde im Rahmen des Fachgebiets Schaltkreis- und Systementwurf an der TU Dresden zum Entwurf Anwendungsspezifischer integrierter Schaltkreise (ASIC) entwickelt. Der Kalenderalgorithmus ist ein mathematisches Verfahren zur genauen Berechnung der Wochentage für ein beliebiges Datum. Es wird häufig in der Softwareentwicklung eingesetzt, insbesondere bei der Implementierung von Kalenderanwendungen und Planern.
Grundlage des Kalenderalgorithmus ist eine Reihe mathematischer Formeln, die es ermöglichen, den Wochentag für ein Datum im Gregorianischen Kalender genau zu bestimmen. Diese Formel wurde 1882 vom Mathematiker und Theologen Christian Zeller veröffentlicht [Zel82].
Im Gegensatz zu vielen anderen Algorithmen gibt es keine Unsicherheiten oder Wahrscheinlichkeiten, da der Kalenderalgorithmus immer das genaue Ergebnis liefert. Der Algorithmus wurde in einer speziellen Programmierumgebung für numerische Berechnungen und Datumsmanipulation implementiert. Besonderes Augenmerk wurde auf die Berücksichtigung von Schaltjahren, unterschiedlichen Kalendersystemen und Zeitzonen gelegt.
Dieser Bericht führt Sie durch den gesamten Designprozess des Kalenderalgorithmus. Beginnend mit einer Erläuterung des mathematischen Hintergrunds und der Funktionsweise des Algorithmus werden im nächsten Kapitel die Schritte zur Implementierung beschrieben. Darüber hinaus werden verschiedene Testfälle und Simulationen durchgeführt, um die Genauigkeit und Zuverlässigkeit des Algorithmus zu überprüfen. Vor dem letzten Schritt muss eine Schaltungs- und Layoutsynthese erstellt werden. Abschließend erfolgt eine Bewertung des Kalenderalgorithmus und seiner Anwendbarkeit in verschiedenen Softwareprojekten. Es werden auch Schwierigkeiten aber auch mögliche Erweiterungen und Verbesserungen diskutiert, um den Algorithmus noch vielseitiger und leistungsfähiger zu machen.

2. Beschreibung des Algorithmus

2.1. Kalender-Algorithmus

Der Zeller's Kongruenz Algorithmus ist ein mathematischer Algorithmus, der verwendet wird, um den Wochentag eines beliebigen Datums im gregorianischen Kalender zu berechnen. Dieser Algorithmus wurde vom Mathematiker Christian Zeller im Jahr 1882 entwickelt [Zel82].
Der Algorithmus [Sto10] basiert auf der Idee der kongruenten Berechnung, bei der verschiedene Teile des Datums miteinander in Beziehung gesetzt werden, um den Wochentag zu ermitteln. Der Zeller's Kongruenz Algorithmus verwendet folgende Formel:

$$W = \left(T + \left[\frac{13 \times (M+1)}{5}\right] + K + \left[\frac{K}{4}\right] + \left[\frac{J}{4}\right] - 2 \times J\right) \mod 7$$

Hierbei wird mit (mod.) die Modulo-Operation bezeichnet, dass der ermittelte Wert durch 7 geteilt und der Rest, der bei dieser ganzzahligen Division durch 7 übrigbleibt, bestimmt wird. Wenn der Wert negativ ist, wird bei der Implementierung das Vorzeichen umgekehrt und dann von 7 subtrahiert, wie folgt: ($-19 \mod 7 \rightarrow 19 \mod 7 = 5 \rightarrow 7 - 5 = 2$)
Der Ausdruck $[x]$ ist die Gaußklammer: Sie gibt die größte ganze Zahl kleiner oder gleich x an, rundet also ab. (größte ganze Zahl $\leq x$) [Gla08]
Hier sind die Variablen wie folgt definiert:

T.. Tagesdatum (1 bis 31)

M.. Monat gemäß unten angeführter Tabelle (3 bis 14)

J.. Die beiden ersten Stellen der Jahreszahl (00 bis 99)

K.. Die beiden letzten Stellen der Jahreszahl (00 bis 99)

bei den Monaten **Januar** und **Februar** die letzten Stellen des Vorjahres (für Dez. 1907 → 1907, für Jan 1907 → 1906 und für Jan 1900 → 1899), Also: Jahreszahl – 1 , da für Januar **M** = 13 bzw.

Februar **M** = 14 gilt, wie in der Tabelle 2.2 zu sehen.

W.. Wochentag gemäß folgender Tabelle:

Tag	Samstag	Sonntag	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
W	0	1	2	5	4	5	6

Tabelle 2.1.: Wochentage

Monat	März	April	Mai	Juni	Juli	August	September	Oktober	November	Dezember	Januar	Februar
M	3	4	5	6	7	8	9	10	11	12	13	14

Tabelle 2.2.: Monatsdarstellung

Dieser Entwurf konzentriert sich darauf, den Algorithmus so zu gestalten, dass er die Umrechnung von Tag, Monat und Jahr in Hexadezimalwerte ermöglicht, die für die Berechnung von Wochentagen erforderlich sind. Dabei werden die entsprechenden Werte aus dem Speicher gelesen, mathematische Operationen durchgeführt und das Ergebnis am Ende wieder in Hexadezimalform ausgegeben.
Zum Beispiel wird der Monat Januar durch den Hexadezimalwert **C**, der Februar durch **D**, der März durch **3** usw. dargestellt. Diese Darstellung ermöglicht es, die Berechnungen gemäß den Regeln des Zeller's Kongruenzalgorithmus durchzuführen.

Hier sind einige Beispiele, die erklären, wie die Formel funktioniert:

⇒ *Beispiel 1:*

14. Juli 2011:

T = 0xE (14 in Dezimal)

M = 0x7 (7 in Dezimal)

J = 0x14 (20 in Dezimal)

K = 0xB (11 in Dezimal)

A = (E + [13 × (7 + 1) / 5] + B + [B / 4] + [14 / 4] - 2 × 14)

A = E + 14 + B + 2 + 5 – 28

A = C (in Hex)

W = C mod. 7

→ **W** = 5 entspricht Donnerstag

⇒ *Beispiel 2:*

21. Sep 1985:

T = 0x15 (21 in Dezimal)

M = 0x9 (9 in Dezimal)

J = 0x13 (19 in Dezimal)

K = 0x55 (85 in Dezimal)

A = (15 + [13 × (9 + 1) / 5] + 55 + [55 / 4] + [13 / 4] - 2 × 13)

A = 15 + 1A + 55 + 15 + 4 - 26

A = 77 (in Hex)

W = 77 mod. 7

→ W = 0 entspricht Samstag

⇒ *Beispiel 3:*

12. Jan 1996:

T = 0xC (12 in Dezimal)

M = 0xD (13 in Dezimal), Januar

J = 0x13 (19 in Dezimal)

K = 0x5F (95 in Dezimal) , 95 → weil: Jahreszahl -1

A = (C + [13 × (D + 1) / 5] + 5F + [5F / 4] + [13 / 4] - 2 × 13)

A = C + 24 + 5F + 17 + 4 - 26

A = 84 (in Hex)

W = 84 mod. 7

→ W = 6 entspricht Freitag

2.2. Programmablaufplan

In Abbildung 2.1 ist einen möglichen Ablauf zur Berechnung des Wochentags mithilfe des Zeller's Kongruenz Algorithmus dargestellt:

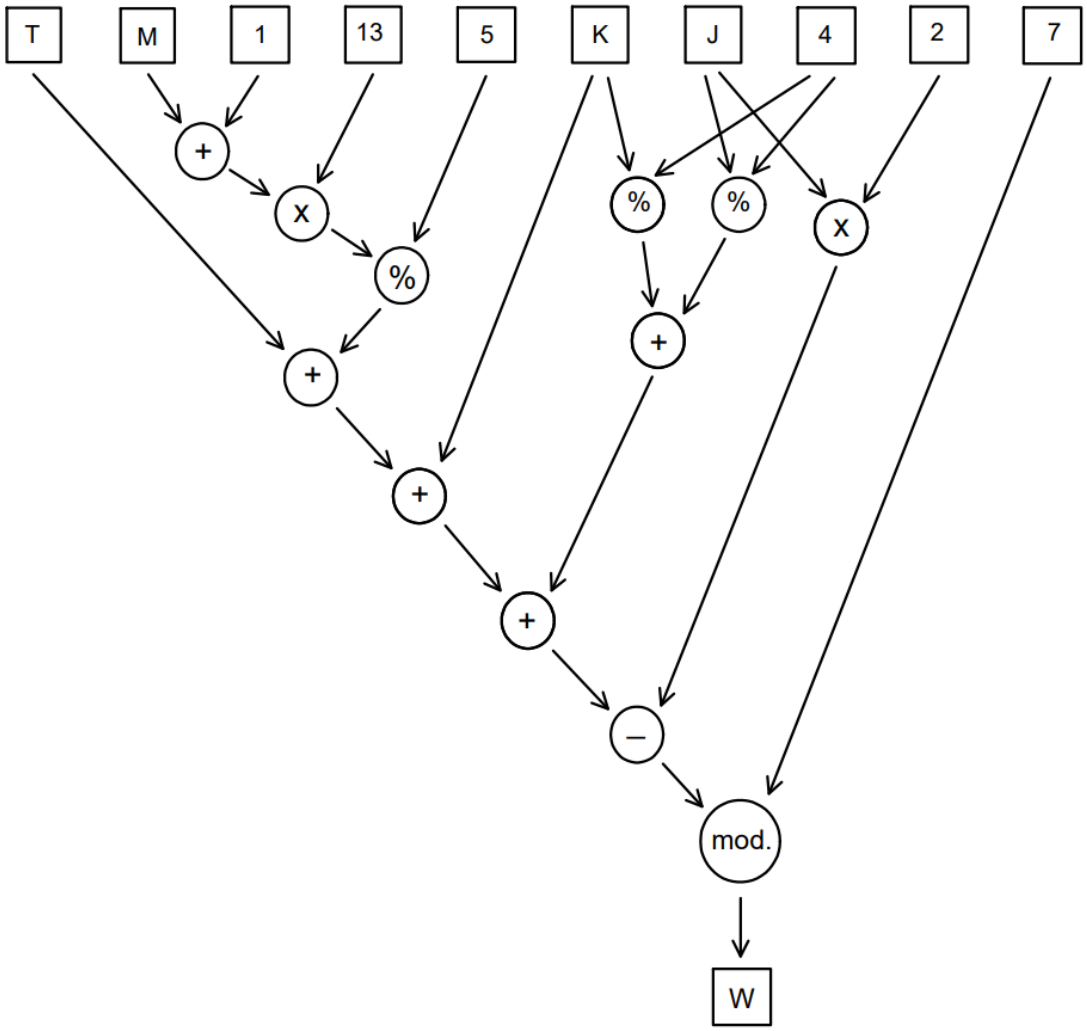


Abbildung 2.1.: Programmablaufplan

Zu Beginn werden verschiedene Register genutzt, um die verschiedenen Eingabedaten aus dem Speicher zu lesen. Parallel dazu werden mathematische Operationen in Abfolge ausgeführt. Jeder Taktzyklus umfasst entweder das Lesen und Speichern eines Werts oder das Ausführen einer Operation, wie später in der Registertransferfolge ausführlich beschrieben wird. Es gibt verschiedene mögliche Scheduling-Varianten, wie zum Beispiel (ASAP) As Soon As Possible, (ALAP) As Late As Possible oder andere Varianten, die gewählt werden können. Das Hauptziel ist dabei, dass das Ergebnis, der berechnete Wochentag , korrekt ermittelt und schließlich an einer spezifischen Speicheradresse abgelegt wird.

Der Algorithmus beginnt mit dem Auslesen des Tages, des Monats und des Jahres aus dem

Speicher und führt dann die erforderlichen Berechnungen durch. Die Werte für Monat und Jahr werden entsprechend den Regeln des Zeller's Kongruenzalgorithmus angepasst [Zel85]. Abschließend wird der berechnete Wochentag an der entsprechenden Speicheradresse platziert, und der Algorithmus endet. Wie ist zu sehen, erfordern viele mathematische Operationen auf derselben Seite einen intensiven Ressourcenverbrauch und eine große Hardwareaufwand. Daher ist es sinnvoll, einen Schedulingstyp zu wählen, der diese Belastung reduziert, Z.B.: Scheduling für minimale Hardware. Dieser Ansatz hat zwar Nachteile wie das Hinzufügen zusätzlicher Takte und die damit verbundene Erhöhung der Latenz, bietet aber auch Vorteile wie geringere Komplexität und geringeren Hardwareaufwand. Darüber hinaus gibt es spätere Optimierungsmöglichkeiten wie Pipelining, wodurch die Anzahl der Takte und die Latenz reduziert werden können.

3. Entwurf

3.1. Anforderungen

Das Hauptziel des Designs besteht darin, den Hardwareaufwand auf ein Minimum zu reduzieren und gleichzeitig eine effiziente Verarbeitung sicherzustellen. Dies erfordert eine Optimierung der Datenübertragung über drei Busse zu den entsprechenden Eingängen verschiedener Register. Die Berechnungen erfordern nur eine ALU-, eine MUL- und eine DIV-Komponente, was die Komplexität deutlich verringert.
Der Datenabruf aus dem Speicher ist begrenzt, sodass bei Bedarf nur ein Wert pro Zyklus aus dem Speicher abgerufen werden kann. Um sicherzustellen, dass der Datenfluss ständig und sorgfältig kontrolliert wird und jeweils nur ein Wert auf den Bus eingespeist wird, stehen zur Vermeidung von Problemen und Konflikten auch unterschiedlich große Multiplexer zur Verfügung.

3.2. Geschwindigkeit

Das Datum wird in Hexadezimalen Werten im Speicher abgelegt und kann dann direkt zur Berechnung des Wochentags verwendet werden. Nur ein Eintrag in MEM führt zu einem Ergebnis am Ausgang.

3.2.1. Pipelining

Aufgrund der großen Anzahl der hier durchzuführenden Rechenoperationen und der Absicht, nur minimale Hardware zu verwenden, muss Pipeline mehrere Operationen gleichzeitig ausführen. In diesem Szenario müssen drei Divisionen hintereinander durchgeführt werden, wobei jede Division drei Zyklen für die Berechnung benötigt. Durch Pipelining können Berechnungen fortgesetzt werden, ohne auf die Ergebnisse früherer Takte warten zu müssen, wodurch die Latenz verringert wird.

3.2.2. Arithmetik

Um Zahlen darzustellen, können entweder Festkomma- oder Gleitkomma-Arithmetik verwendet werden. Da es sich hier um einfache Berechnungen mit hexadezimalen Werten handelt, ist Festkomma-Arithmetik vollkommen ausreichend für den Zweck. In diesem Bericht werden Bausteine für Festkomma Rechenoperationen mit einer Wortbreite von 32 Bit verwendet. Da die Ausgabe der Division 64 Bit beträgt, können in der Festkommadarstellung die ersten 32 Bit als Ergebnis der Division verwendet werden, und im letzten Schritt (Takt) können die letzten 32 Bit als Ergebnis der Modulo-Operation gelesen werden. Dies vereinfachten die Berechnung und Ausgabe.

3.3. Datenflussgraph

Das Datenflussgrap 3.1 ist einfach und übersichtlich strukturiert. In jedem Ladezustand gibt es normalerweise nur einen Registereintrag und einen Eintrag für einen konstanten Wert, der vom BUS_D kommt. Diese Einträge werden entweder an die ALU, MUL oder DIV weitergeleitet, um eine Berechnung durchzuführen.
Das Ergebnis der Berechnung wird dann entweder im neuen Register gespeichert oder an den nächsten Eingang der Hardwarekomponenten weitergegeben. Erst wenn die Stufe (LOAD20) der mathematischen Operationen erreicht ist, kann ein Flag eine Änderung der Einträge ermöglichen. Zum Beispiel stellt ein flag_S/Z sicher, dass die Ausgabe der Berechnung korrigiert wird, wenn sie negativ oder Null ist, bevor mit der nächsten Operation fortgefahren wird. Dadurch wird sichergestellt, dass im nächsten Schritt der Berechnung normalerweise der Subtraktions- oder Modulo-Operator über einen richtigen Eintrag verfügt, um ein korrektes Ergebnis zu gewährleisten, das jedem Wochentag, wie im tabelle 2.1, von 0 bis 6 entspricht.

IDLE

LOAD1

LOAD2

LOAD3

LOAD4

LOAD5

LOAD6

LOAD7

LOAD8

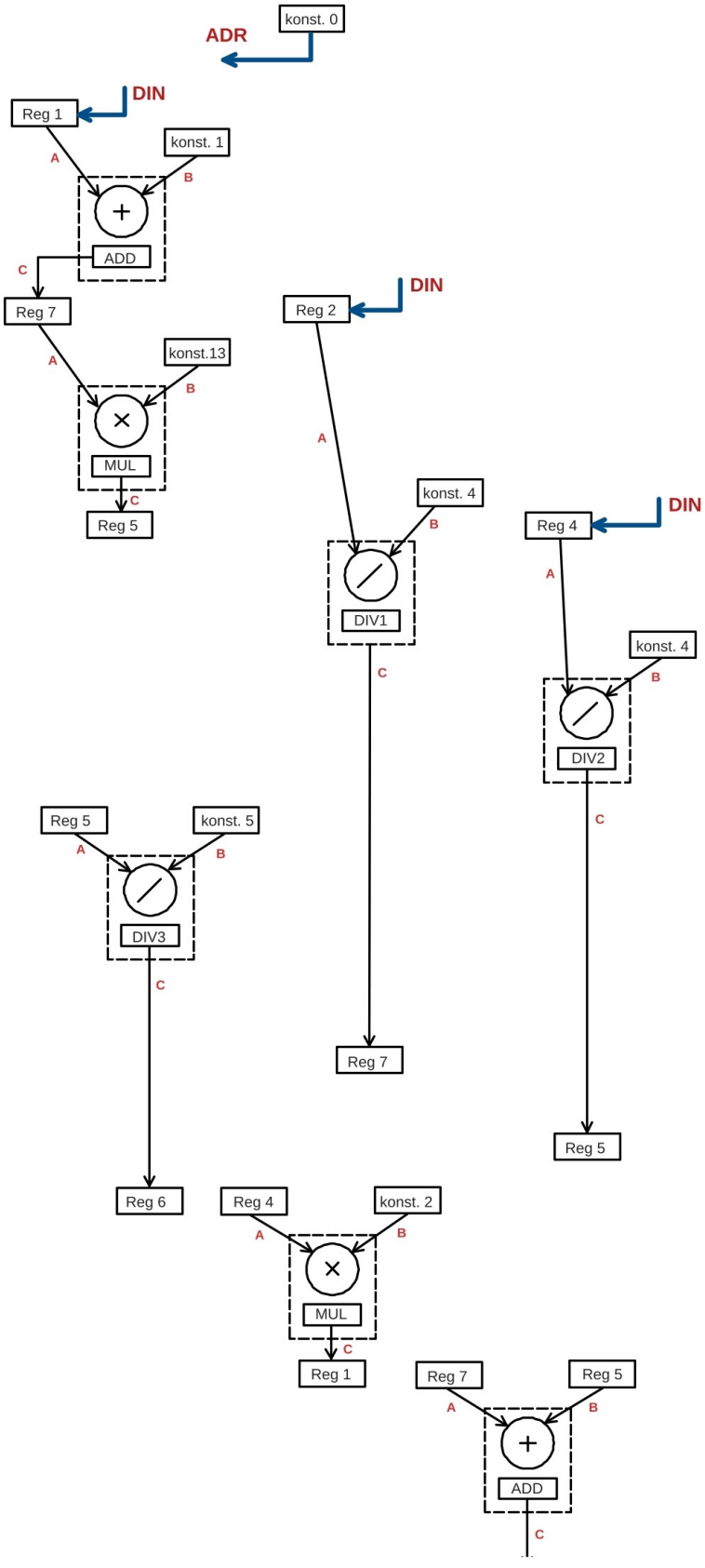
LOAD9

LOAD10

LOAD11

LOAD12

LOAD13



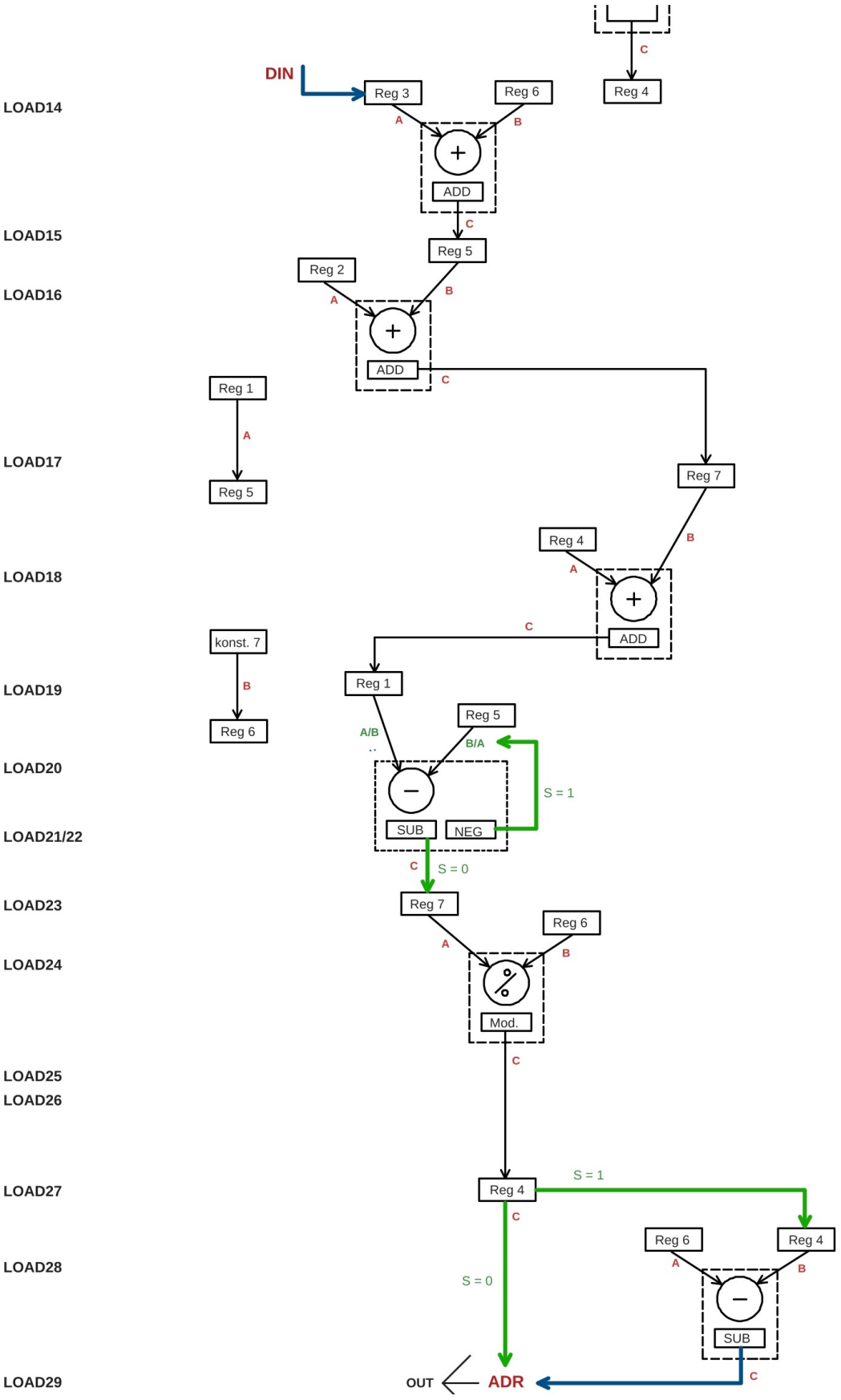


Abbildung 3.1.: Datenflussgraph

3.4. Datenpfadarchitektur

Aus dem Datenflussgraph lassen sich die Anforderungen bezüglich der Anzahl der Busse, Register und Recheneinheiten ableiten. In Abbildung 3.2 ist der resultierende Datenpfad dargestellt. Neben der ALU, der MUL und der DIV-Einheit umfasst dieser 7 Register (R1-R7), 7 konstante Register und 11 Multiplexer unterschiedlicher Größen, um einen reibungslosen Ablauf zu gewährleisten. Zur Verdrahtung werden 3 Busse verwendet.

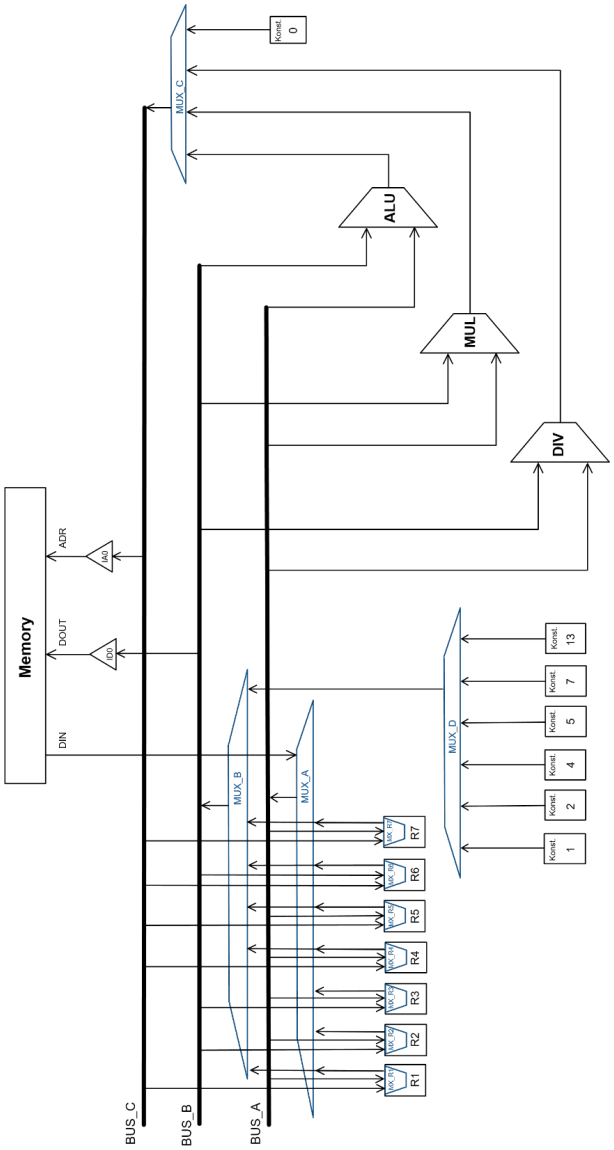


Abbildung 3.2.: Datenpfadarchitektur mit Multiplexer-Bus

Abbildung 3.3 zeigt die Umsetzung des Datenpfads in CADENCE[1]. Im Vergleich zum Blockschaubild sind hier alle erforderlichen Steuersignale an den Registern, der ALU, MUL, MUX und DIV verzeichnet. Nicht benötigte Eingänge wurden mit dem Baustein ("TIE0") auf eine logische 0 gesetzt, während nicht benötigte Ausgänge offengelassen wurden ("NoConn"). Zu sehen sind außerdem die Clock-, Reset-Eingänge und 2 Puffer an den Ausgängen.

[1] Die .eps-Daten konnten nicht wiederhergestellt werden und ein Screenshot mit besserer Qualität konnte nicht erstellt werden

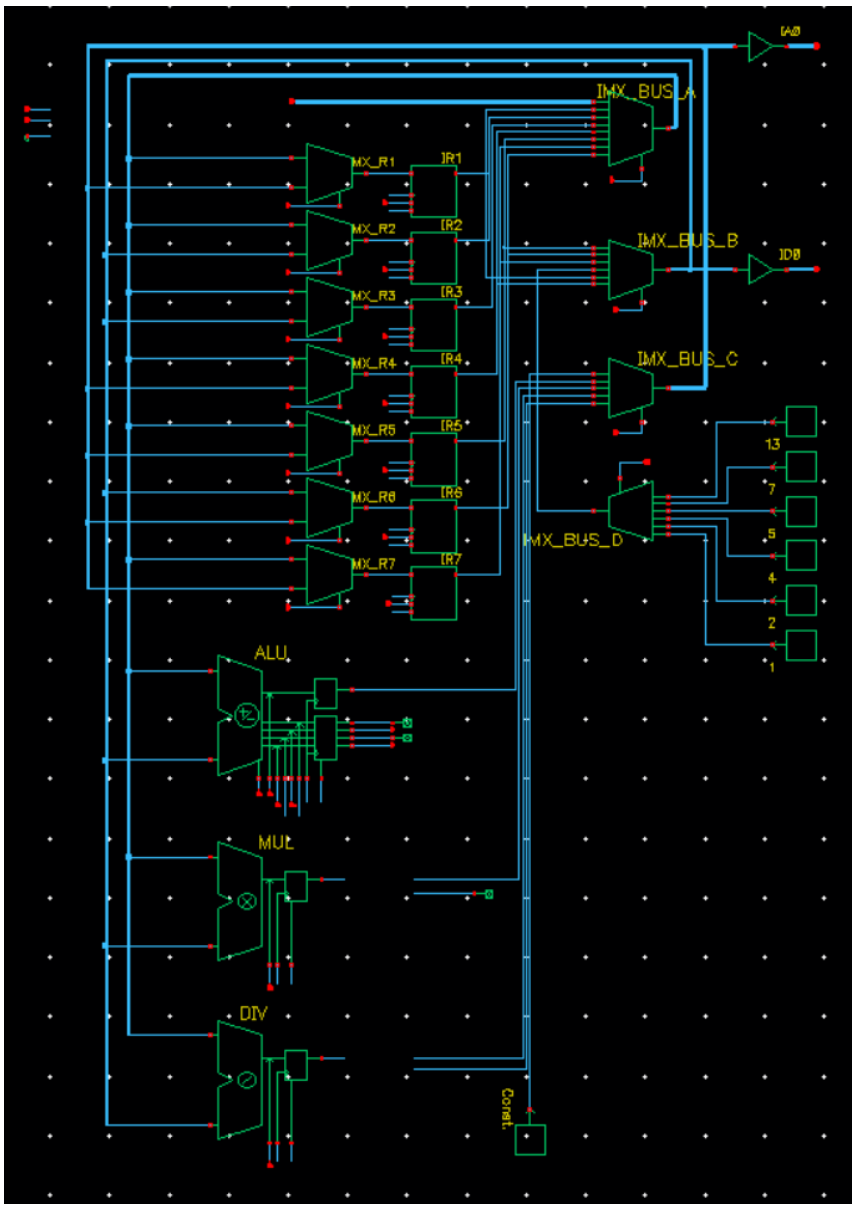


Abbildung 3.3.: Datenpfad, in CADENCE

3.5. Registertransferfolge

Der Registerübertragungsablauf in Abbildung 3.4 ergibt sich aus dem Programmablauf, dem Datenflussdiagramm und dem Datenpfad. Alle Operationen können nacheinander ausgeführt werden, solange kein Flag ausgegeben wird. Dazu gehören Initialisierungs- und mathematische Operationen wie Addition, Subtraktion, Multiplikation, Division und Modulo-Rechnungen, bis das Endergebnis erreicht ist. Sobald ein Flag gesetzt ist, wird es entsprechend der vorgesehenen Reihenfolge verarbeitet und der Prozess kann fortgesetzt werden. Das Endergebnis wird im Register R4 gespeichert und dann an Ram[4] ausgegeben.

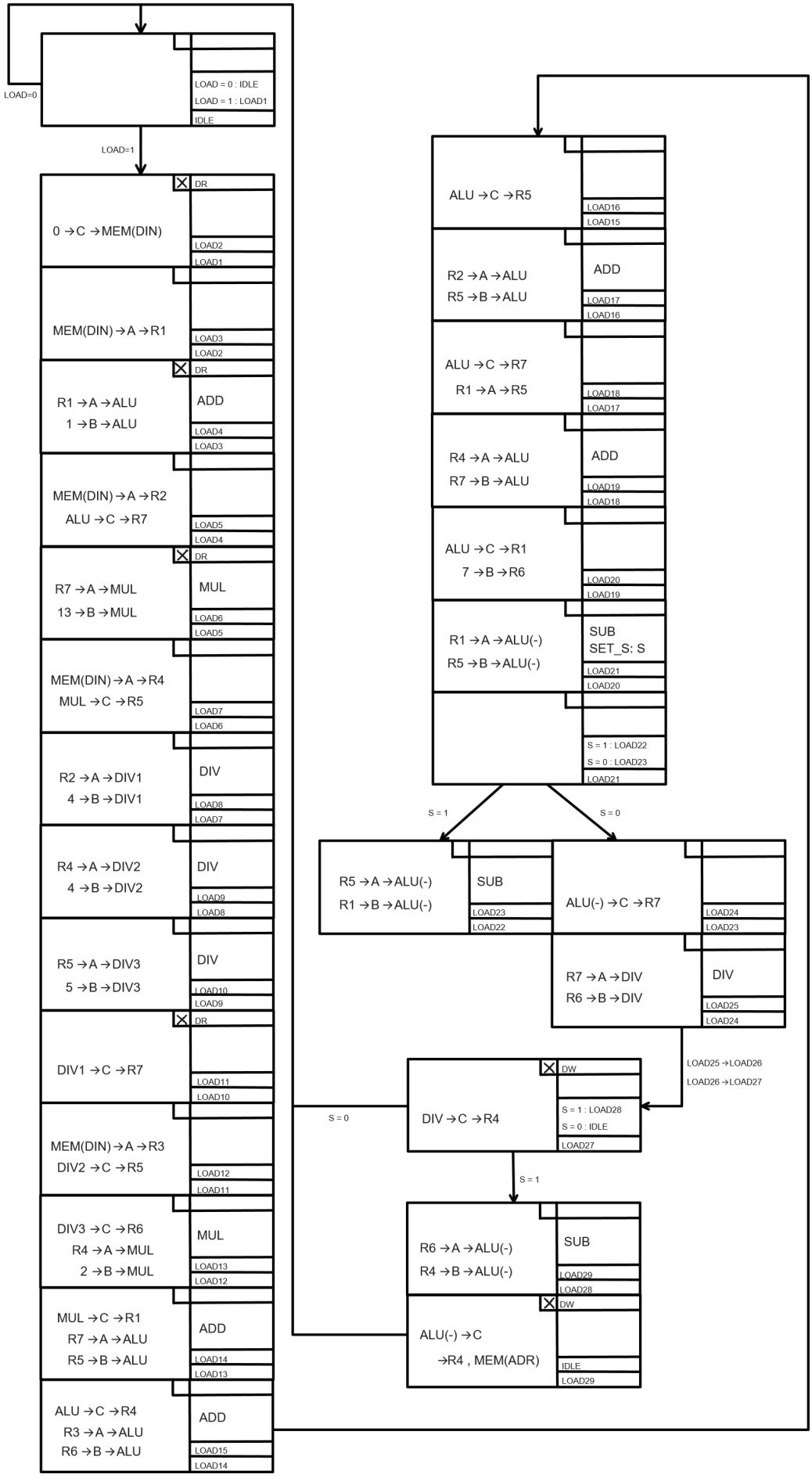


Abbildung 3.4.: Registertransferfolge

3.6. Zustandsübergänge

Abbildung 3.5 veranschaulicht den Zustandsübergangsautomaten der FSM (Finite State Machine). Der Zustandsgraph basiert auf der Registertransferfolge und zeigt deutlich die beschriebenen Merkmale des Algorithmus. Er umfasst insgesamt 29 Zustände, wie auch die RT-Folge. Die Verzweigungen werden durch die Flags "loadünd "Flag_S/Z"gesteuert. Jedoch erfolgt der Übergang zwischen den meisten Zuständen automatisch nacheinander.
Im Anhang befindt sich der Verilog-Code, der den FSM-Prozess beschreibt. Da die Zustandskodierung für die Zustandsmaschine im Verilog-Code beschrieben ist, ist eine Wiederholung hier nicht erforderlich.

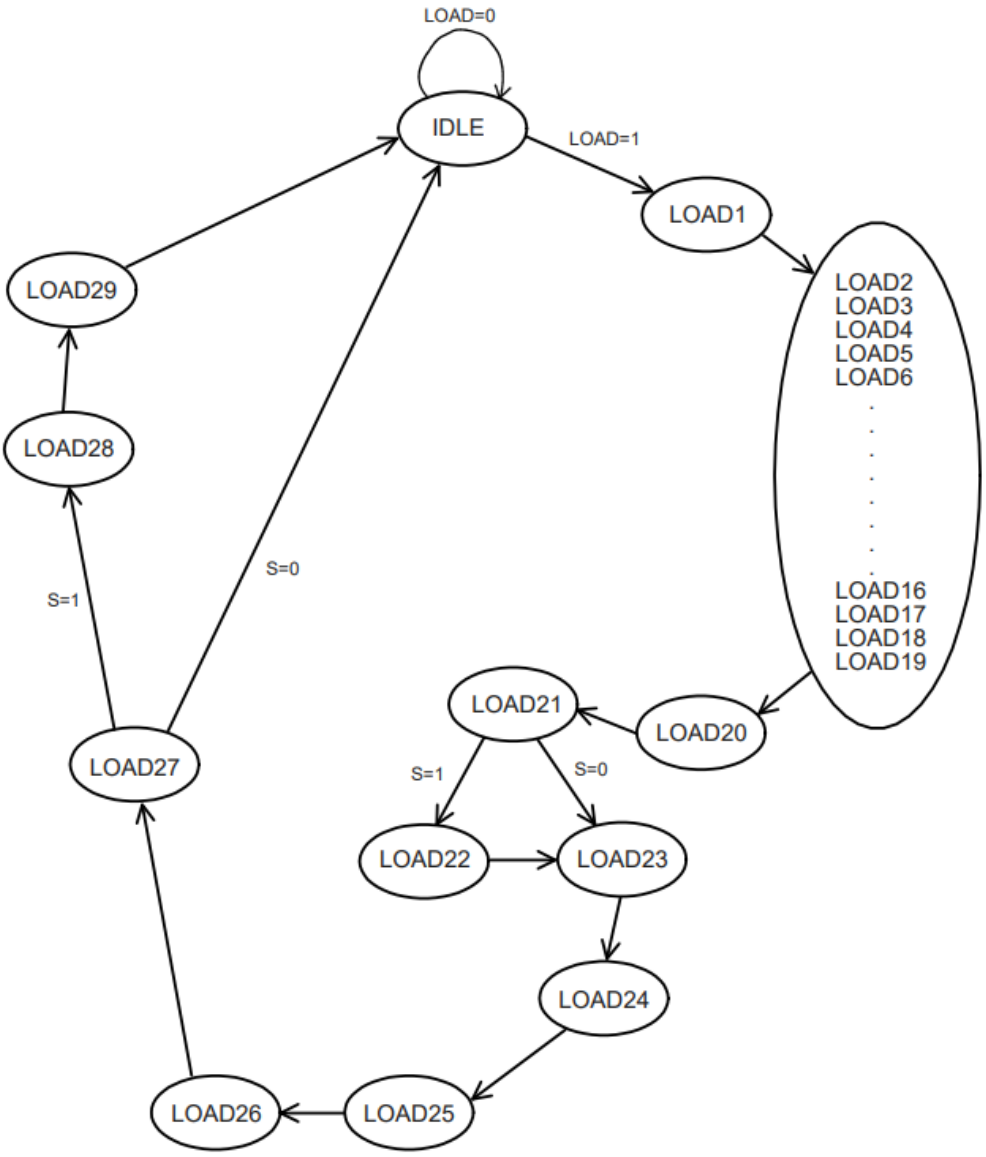


Abbildung 3.5.: Zustandsgraph (FSM)

3.7. Gesamte Schaltung

In Abbildung 3.6 ist die in CADENCE entwickelte Schaltung dargestellt, die die Symbole des FSM, der Steuerlogik und des Datenpfads gemäß den entsprechenden Schaltplänen und Beschreibungen integriert. Die Entwicklung des FSM und des Datenpfades wurde bereits erläutert. Die Steuerlogik steuert alle relevanten Komponenten wie Register, Multiplexer, DIV, MUL, ALU und Speicherzugriffe. Es empfängt den aktuellen Zustand des FSM als Input und wandelt ihn in Steuersignale für den Datenpfad um. Es beschreibt außerdem, welche Steuerbefehle in jedem Zustand an den Datenpfad gesendet werden müssen und regelt den Zugriff auf den externen Speicher. [2]
Ein weiteres charakteristisches Merkmal der Gesamtschaltung ist die Interaktion mit externen Speichern über den externen Adressbus (ADR), den externen Datenbus (DIN) und entsprechende Freigabesignale. Der externe Datenbus (DOUT), der als Ausgang fungiert, dient hier lediglich dazu, sicherzustellen, dass insbesondere bei Simulationen alles korrekt funktioniert. Ich kann die Werte von (DOUT) schnell lesen und sehen, ob ein Fehler vorliegt. Andernfalls werden die Wochentage von (ADR) und nicht von (DOUT) übernommen und in den Speicher geschrieben. Zusätzlich werden zwei Flag-Signale aus dem Datenpfad den Eingängen des FSM zugeführt.

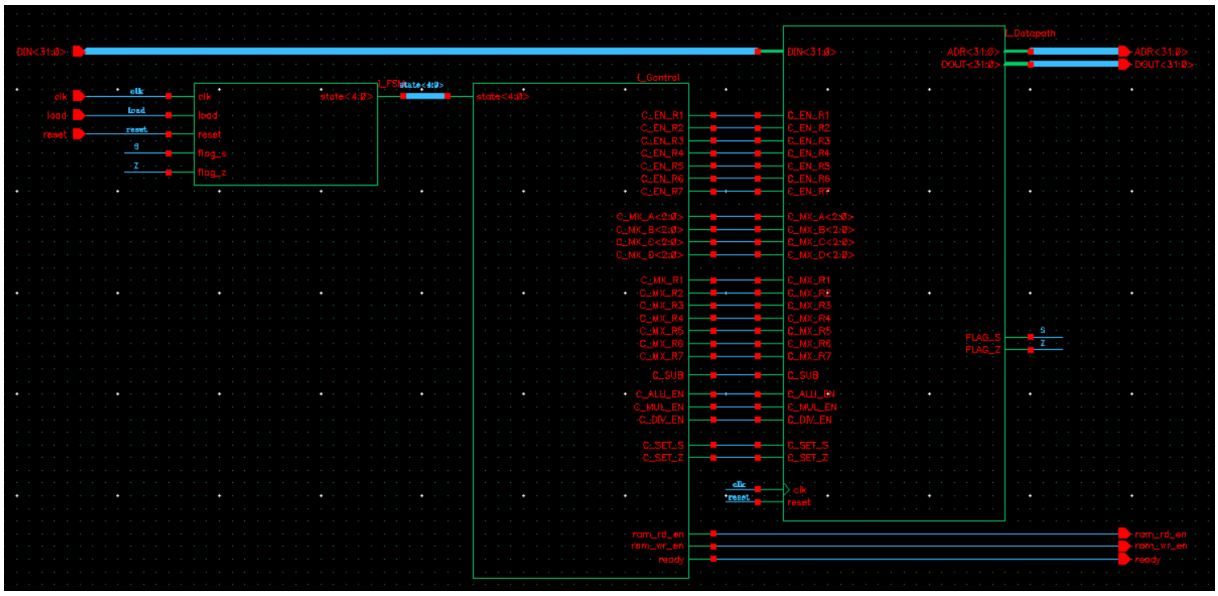


Abbildung 3.6.: Gesamtschaltung in CADENCE

[2] Im Anhang finden Sie Quellcodes zu den Schaltplänen.

4. Simulation und Test

Nach der Konzeption muss die Funktionalität des ASICs simuliert und überprüft werden. Zu diesem Zweck wurde eine Testumgebung für das gesamte Projekt eingerichtet. Die aus Simvision exportierten Ergebnisse der Simulation sind unten aufgeführt. [1]

4.1. Simulation und Test der FSM

In den Abbildungen 4.1 (ohne Flag) und 4.2 (mit Flag) sind die Simulationsergebnisse der FSM auf Gatterebene dargestellt. Zunächst wurde das Signal "resetäuf 1 gesetzt und alle anderen Steuersignale auf 0. Nach drei Takten wird "Reset"wieder auf 0 gesetzt, sodass sich die FSM im Zustand ÏDLE"befindet. Nach weiteren 2 Takten wird das "loadFlag auf 1 gesetzt. Anschließend sollten die Zustände der Reihe nachfolgen. Zum Schluss sollte die FSM automatisch wieder in den Zustand ÏDLE"wechseln.
Die FSM verhält sich für alle gültigen Zustände wie erwartet. Auf jeden ungültigen Zustand folgt der Zustand ÏDLE". Beide Varianten der FSM liefern korrekte Simulationsergebnisse.

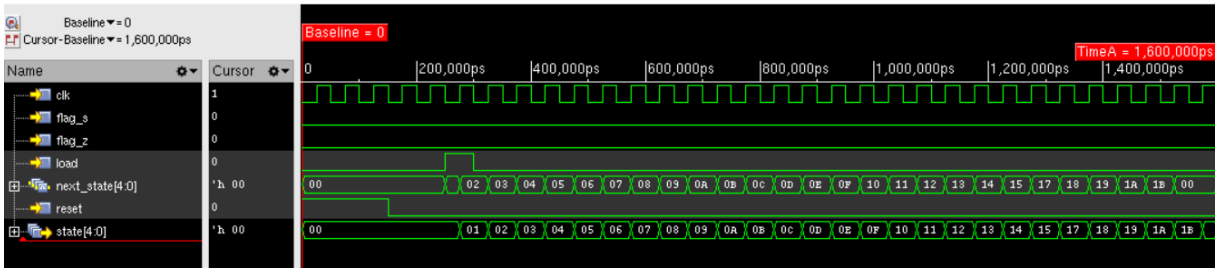


Abbildung 4.1.: Simulationsergebnis der FSM (ohne Flags)

Wenn ein Flag erkannt wird, wie in Abbildung 4.2, ändert sich die Reihenfolge der Zustände und folgt den im Verilog-Code[2] beschriebenen Änderungen.

[1] Um korrekte Ergebnisse sicherzustellen, wurden mehrere Simulationen durchgeführt
[2] Code befindet sich im Anhang

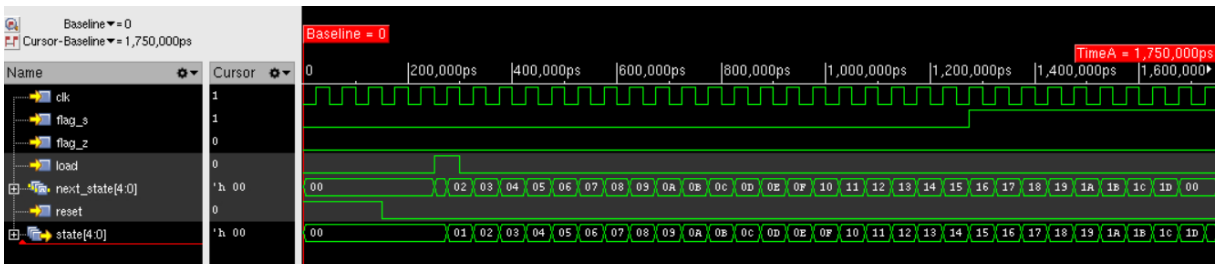


Abbildung 4.2.: Simulationsergebnis der FSM (mit Flags)

4.2. Simulation und Test der Steuerlogik

Als nächstes wird die Funktion der Steuerlogik getestet. Es werden nacheinander alle möglichen Zustände vorgegeben. In der Simulation sollten sich in der Ausgabe nun die entsprechend vorgesehenen Steuersignale einstellen.
In Abbildung 4.3 ist das Ergebnis der Simulation zu sehen. Ein Abgleich mit dem Registertransferfolge zeigt, dass die Steuerlogik korrekt arbeitet.

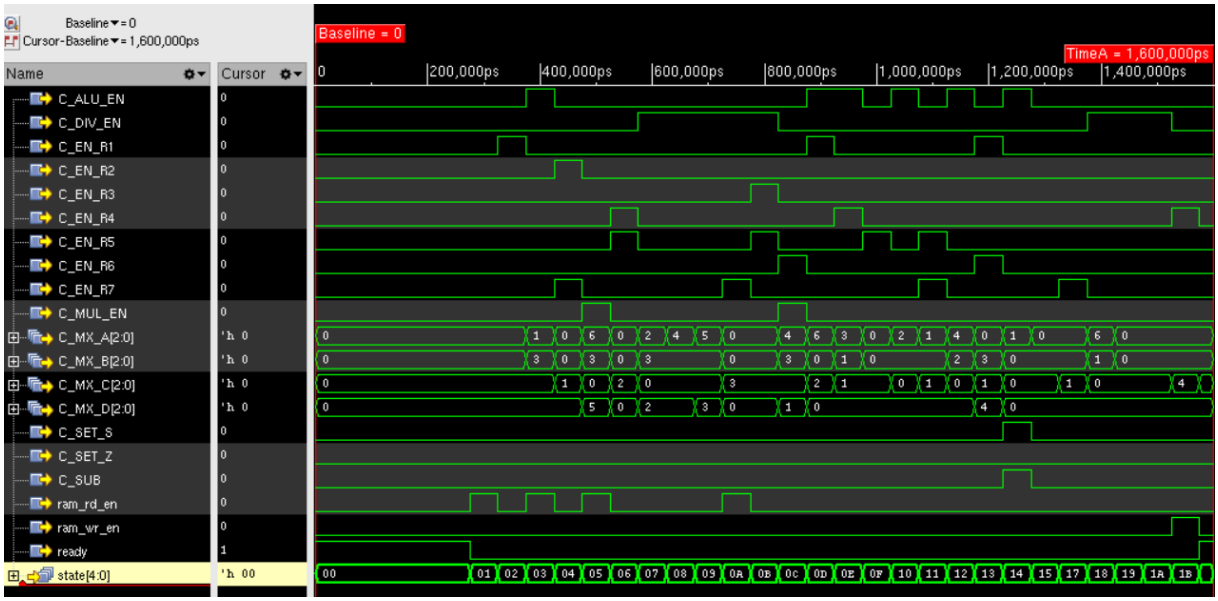


Abbildung 4.3.: Simulationsergebnis der Steuerlogik

4.3. Simulation und Verifikation der Gesamtschaltung

Anschließend wird die Funktion der gesamten Schaltung getestet. Wie beim Test des FSM und der Steuerlogik wird die Schaltung zunächst durch einen „Reset"initialisiert und alle Steuersignale und Flags auf 0 gesetzt. Mit dem Signal „read_address"werden die Datumswerte nacheinander gelesen, wenn rd_en ist auf 1 gesetzt. In diesem Fall wird das „DIN"-Signal als Eingabe

aus dem Speicher genommen und dann in einem Register gespeichert. Am Ende, wenn wr_en auf 1 gesetzt ist, wird das Signal „ADR“ als Wert in den Speicher in ram_add_w geschrieben, der das Ergebnis (Wochentag) anzeigt. Wie bereits gesagt, besteht (DOUT) nur darin, sicherzustellen, dass alles ordnungsgemäß funktioniert.
In diesem Beispiel unten werden die Datumswerte im Dezimalformat angezeigt. In Abbildung 4.4 sieht man, dass am Ende das richtige Ergebnis in ram_add_w gespeichert wird. [3]

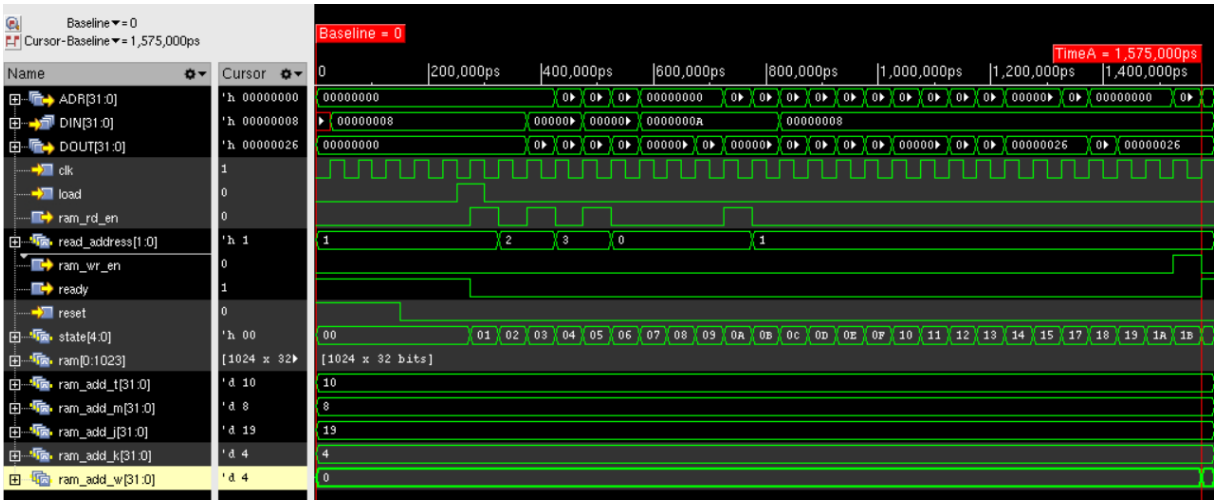


Abbildung 4.4.: Simulationsergebnis der Gesamtschaltung

[3] Da das Ergebnis mit und ohne Flag sehr ähnlich ist, wurde nur eine Abbildungsbeispiel der Steuerlogik und der Gesamtschaltung gegeben.

5. Schaltungs- und Layoutsynthese

Um den zeitlichen Ablauf der verschiedenen Prozesse zu koordinieren, wird die Schaltungs- und Layoutsynthese erstellt. Wichtige Aspekte müssen berücksichtigt werden, wie die Anwendung von Constrains sowie die Identifizierung und Optimierung kritischer Timing-Pfade. Auch die Gestaltung und Größe der Fläche spielt eine entscheidende Rolle.

- Constraints

  sind die Vorgaben, die dem Synthese-Tool für die Optimierung der zeitlichen Eigenschaften des Designs bereitgestellt werden. Eine gründliche Analyse und Bewertung der Randbedingungen ist entscheidend für die erfolgreiche Entwicklung des Schaltungsdesigns. Durch die Berücksichtigung dieser Constraints können potenzielle Probleme frühzeitig erkannt und gelöst werden.

- kritischen Timingpfaden

  Die Identifizierung und Analyse kritischer Timingpfade ermöglicht es, Bereiche in der Schaltung zu identifizieren, die besonders anfällig für Timingprobleme sind. Durch gezielte Optimierungen können diese Bereiche verbessert werden.

- Fläche

  Die Fläche eines Schaltungsdesigns spielt eine entscheidende Rolle bei der Bestimmung seiner Effizienz und Wirtschaftlichkeit. Eine sorgfältige Optimierung der Fläche ermöglicht es, Ressourcen effizient zu nutzen und die Produktionskosten zu minimieren.

Nachdem die Schaltungs- und Layoutsynthese erstellt wurde, wird eine Simulation durchgeführt, um sicherzustellen, dass alles korrekt funktioniert und die synthetisierte Netzliste korrekt erstellt wurde:

5.1. Schaltungssynthese

Bei korrekter Einrichtung öffnet sich die bekannte Simvision-Simulationsumgebung. Unter Berücksichtigung des Timings ist mitunter mit einer kleinen Verzögerung zu rechnen, allerdings sollten die Ergebniswerte der erwarteten Abarbeitung des Algorithmus nicht von den Ergebnissen der funktionalen Simulation abweichen.
Abbildung 5.1 zeigt, dass die neuen Simulationswerte genau den Erwartungen entsprechen und die Zeitverzögerungen ohne funktionale Unterschiede auftraten.

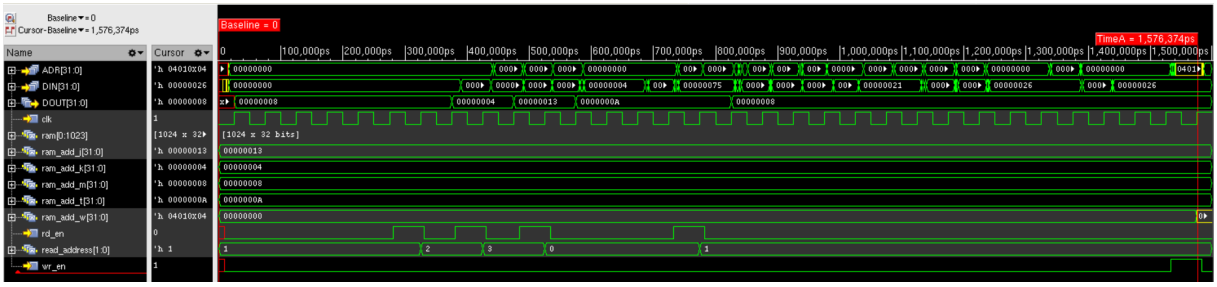


Abbildung 5.1.: Simulationsergebnisse (Schaltungssynthese)

In Abbildung 5.2 ist es zur besseren Übersichtlichkeit vergrößert dargestellt.

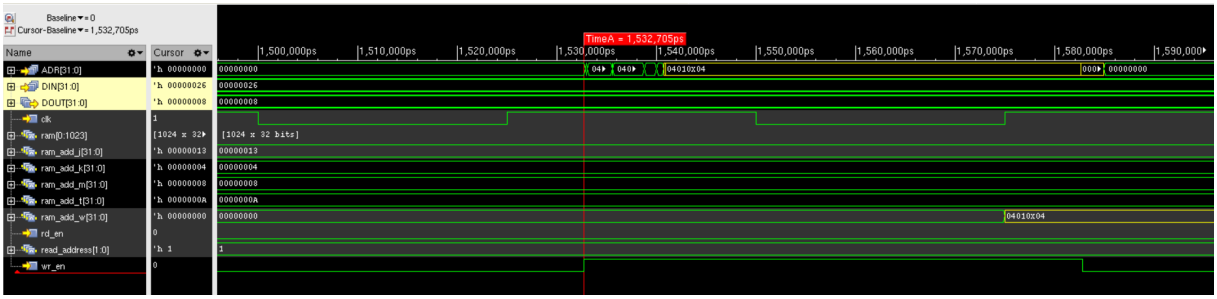


Abbildung 5.2.: Simulationsergebnisse (einmal vergrößert)

5.2. Layoutsynthese

Dabei erfolgte die Platzierung und Verdrahtung aller im Design vorhandenen Gatter und Padzellen auf Chiplevel. Da es eine vollständige synthetisierte Gatternetzliste gibt, könnte eine Layout-Synthese durchgeführt werden. Eine Schätzung des Flächenbedarfs wurde dem Area_Report der Syntheseergebnisse (fc_example.report_area) entnommen und die Parameter des Floorplans gelegentlich angepasst. In den Abbildungen 5.3, 5.4 und 5.5 können die Ergebnisse der einzelnen Targets (Floorplan, CTS und Final) visuell im GUI betrachtet werden.

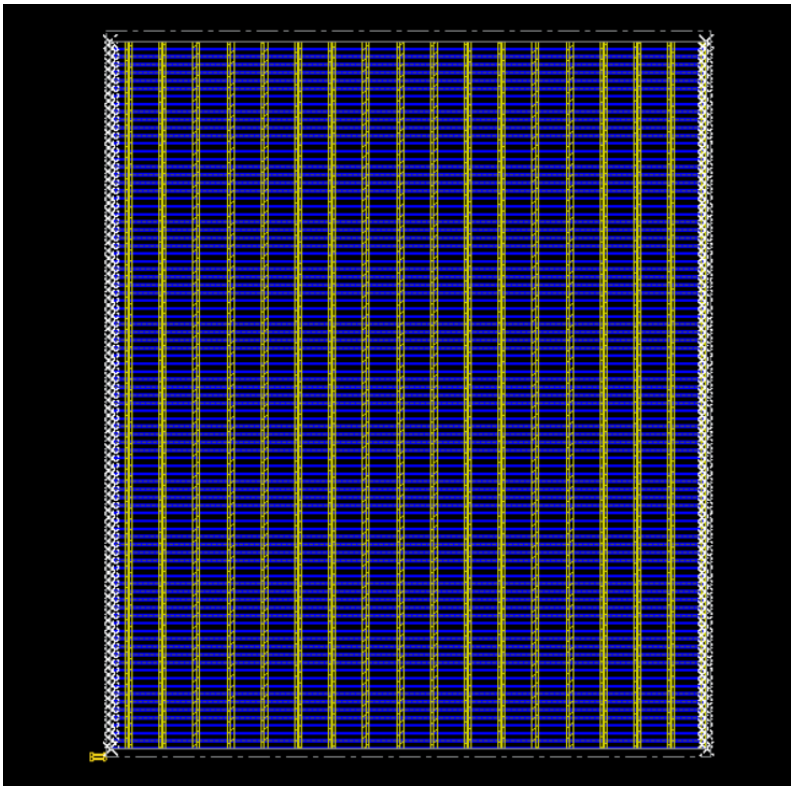


Abbildung 5.3.: Floorplan

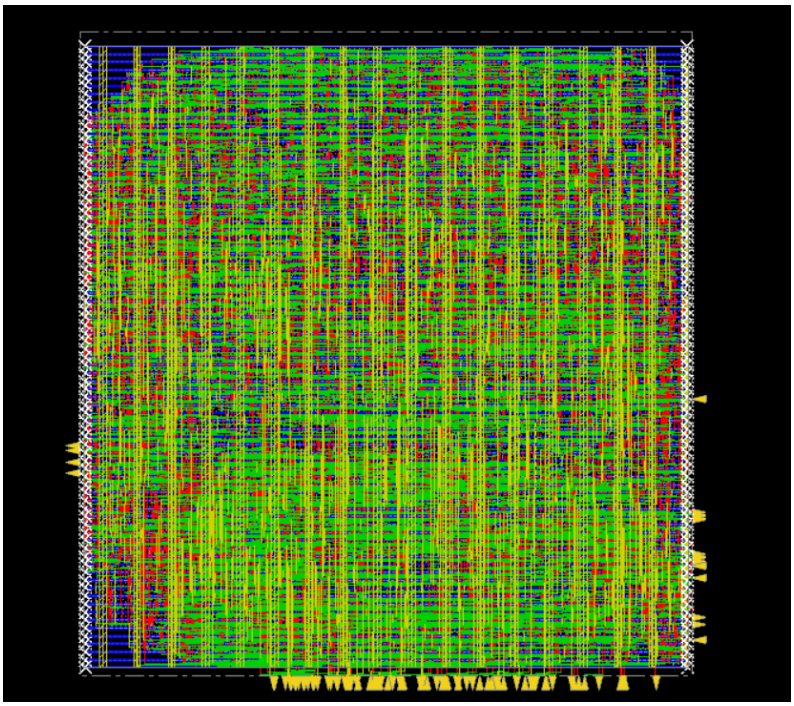


Abbildung 5.4.: CTS

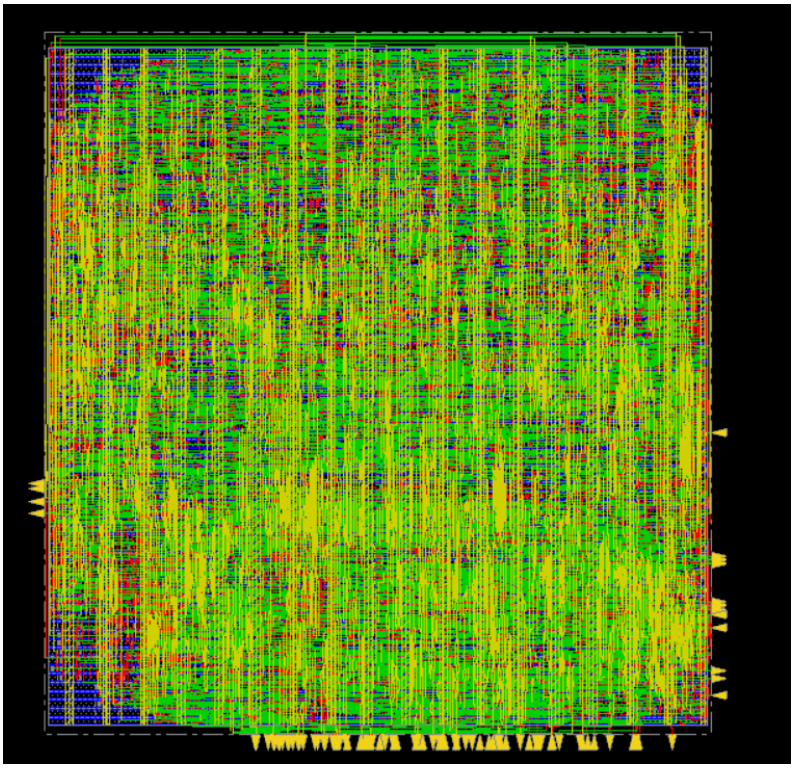


Abbildung 5.5.: Final

Und wieder am Ende startet die Simulation der Layout-Netzliste und es öffnet sich wieder die bekannte Simvision-Simulationsumgebung. Die Ergebniswerte der erwarteten Abarbeitung des Algorithmus bleiben gleich nur mit erwartete Zeitverzögerung. In Abbildung 5.6 sind die Simulationsergebnisse sowie der implementierte Clocktree dargestellt, der während Place & Route erstellt wird.

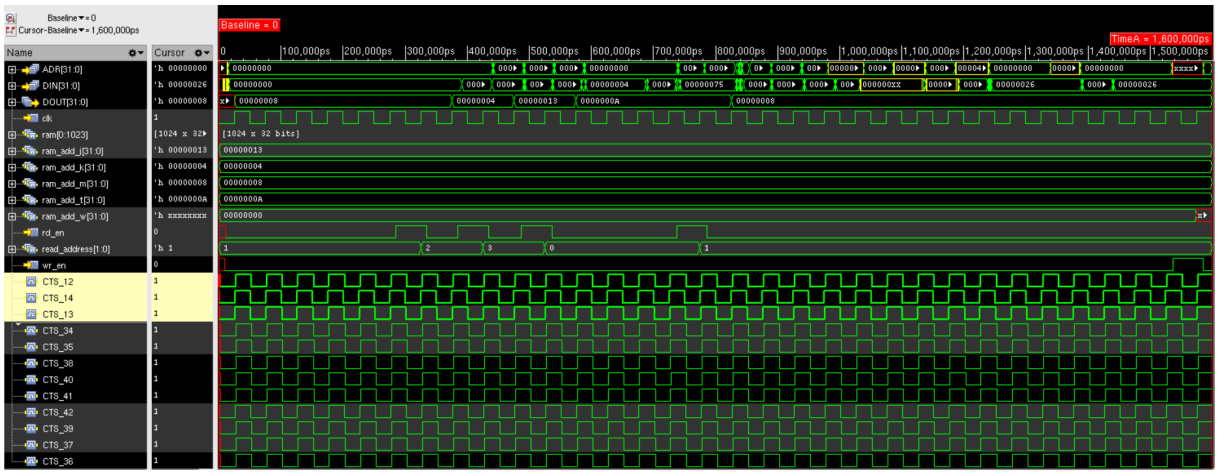


Abbildung 5.6.: Simulationsergebnisse (Layoutsynthese)

Die Clocktrees sehen für alle Endpunkte gleich aus, aber beim Zoomen bemerken Sie die Zeitverzögerung für alle Clocktrees, wie in den Abbildung 5.7 dargestellt.

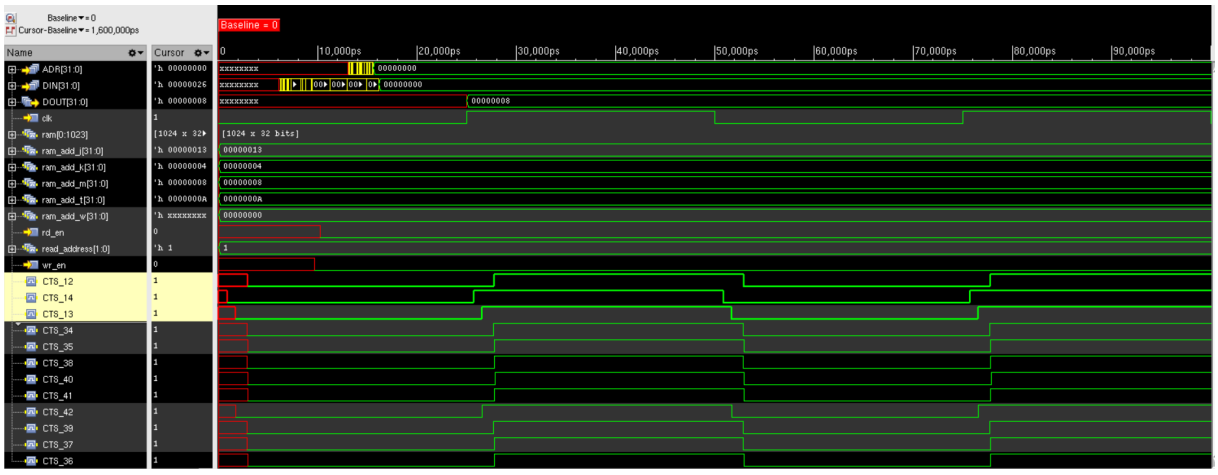


Abbildung 5.7.: Simulationsergebnisse (einmal vergrößert)

6. Zusammenfassung und Wertung

Die Simulationsergebnisse zeigen, dass das Entwurfsziel unter Berücksichtigung der genannten Anforderungen erfolgreich erreicht wurde. Die Architektur ermöglicht eine schnelle und effiziente Implementierung des Algorithmus, was ein zeitoptimierte Rechenoperation ermöglicht. Dabei wurde der Hardwareaufwand minimal gehalten und effektiv genutzt.
Obwohl es möglicherweise einfacher wäre, den Algorithmus mit einer anderen Programmiersprache wie Python zu implementieren, um das Datum digital zu erhalten, ist das Verständnis und die Anwendung von Cadence dennoch von großer Bedeutung und bietet wertvolle Lernerfahrungen.
In diesem Sinne kann der entworfene ASIC als akademisches Beispiel für die praktische Umsetzung eines Algorithmus betrachtet werden, aus dem viel gelernt werden kann.
Während des Designprozesses wurden verschiedene Herausforderungen bewältigt. Insbesondere die Division und die Nutzung des Dividierer stellten sich als anspruchsvoll heraus und erhöhten die Komplexität der Aufgabe. Es wurden mehrere Designänderungen vorgenommen, um schließlich zu einem Design zu gelangen, das mit minimaler Hardware alle Operationen hintereinander ausführen kann.
Eine weitere Herausforderung bestand in der Schaltungs- und Layoutsynthese, da in der Vorlesung wenig darüber gesprochen wurde und im Internet nur begrenzte Beispiele verfügbar waren. Trotz dieser Schwierigkeiten konnten die auftretenden Probleme erfolgreich gelöst werden, und der gesamte Designprozess bot einen umfassenden Einblick in die verschiedenen Phasen der ASIC-Entwicklung.

A. Quellcodes

Referenzimplementierung in Software (Python)

```python
def Datum(Tag, Monat, Jahr):
    # Monate Januar und Februar werden als 13. bzw. 14. Monat des Vorjahres behandelt
    if Monat in (1, 2):
        Monat += 12
        Jahr -= 1

    # Wochentag berechnen (0 = Samstag, 1 = Sonntag, 2 = Montag, usw.)
    k = Jahr % 100
    j = Jahr // 100
    Wochentag = (Tag + 13 * (Monat + 1) // 5 + k + k // 4 + j // 4 - 2 * j) % 7

    # Den Wochentag in Textform ausgeben
    Tage = ["Samstag", "Sonntag", "Montag", "Dienstag",
            "Mittwoch", "Donnerstag", "Freitag"]
    return Tage[Wochentag]

# Beispielaufruf
Tag = 12
Monat = 1
Jahr = 1996
res = Datum(Tag, Monat, Jahr)
print(f"{Tag}.{Monat}.{Jahr} ist ein {res}.")
```

Verhaltensbeschreibung FSM

```
// Verilog HDL for "Beispiel", "KA_FSM" "functional"

module KA_FSM( clk, reset, load, state, flag_s, flag_z);

  parameter IDLE = 5'b00000;
  parameter LOAD1 = 5'b00001;
  parameter LOAD2 = 5'b00010;
  parameter LOAD3 = 5'b00011;
  parameter LOAD4 = 5'b00100;
  parameter LOAD5 = 5'b00101;
  parameter LOAD6 = 5'b00110;
  parameter LOAD7 = 5'b00111;
  parameter LOAD8 = 5'b01000;
  parameter LOAD9 = 5'b01001;
  parameter LOAD10 = 5'b01010;
  parameter LOAD11 = 5'b01011;
  parameter LOAD12 = 5'b01100;
  parameter LOAD13 = 5'b01101;
  parameter LOAD14 = 5'b01110;
  parameter LOAD15 = 5'b01111;
  parameter LOAD16 = 5'b10000;
  parameter LOAD17 = 5'b10001;
  parameter LOAD18 = 5'b10010;
  parameter LOAD19 = 5'b10011;
  parameter LOAD20 = 5'b10100;
  parameter LOAD21 = 5'b10101;
  parameter LOAD22 = 5'b10110;
  parameter LOAD23 = 5'b10111;
  parameter LOAD24 = 5'b11000;
  parameter LOAD25 = 5'b11001;
  parameter LOAD26 = 5'b11010;
  parameter LOAD27 = 5'b11011;
  parameter LOAD28 = 5'b11100;
  parameter LOAD29 = 5'b11101;


  input clk;
  input reset;
  input load;
  input flag_s;
  input flag_z;
```

```
43    output state;
45    reg [4:0] state;
46    reg [4:0] next_state;
49    always @ (posedge clk or posedge reset)
50      if (reset) begin
51          state <= IDLE;
52      end
53      else begin
54        state <= next_state;
55      end
57    always @ (state or load or flag_s or flag_z)
58      case (state)
59        IDLE : if (load) begin //0
60                    next_state = LOAD1;
61               end
62               else begin
63                   next_state = IDLE;
64               end
65        LOAD1 : next_state = LOAD2;
66        LOAD2 : next_state = LOAD3;
67        LOAD3 : next_state = LOAD4;
68        LOAD4 : next_state = LOAD5;
69        LOAD5 : next_state = LOAD6;
70        LOAD6 : next_state = LOAD7;
71        LOAD7 : next_state = LOAD8;
72        LOAD8 : next_state = LOAD9;
73        LOAD9 : next_state = LOAD10; // 0A
74        LOAD10 : next_state = LOAD11; // 0B
75        LOAD11 : next_state = LOAD12; // 0C
76        LOAD12 : next_state = LOAD13; // 0D
77        LOAD13 : next_state = LOAD14; // 0E
78        LOAD14 : next_state = LOAD15; // 0F
79        LOAD15 : next_state = LOAD16; // 10
80        LOAD16 : next_state = LOAD17; // 11
81        LOAD17 : next_state = LOAD18; // 12
82        LOAD18 : next_state = LOAD19; // 13
83        LOAD19 : next_state = LOAD20; // 14
84        LOAD20 : next_state = LOAD21; // 15
85        LOAD21 : if (flag_s) begin // flag_s (R1
                       < R5)
86                     next_state = LOAD22; // 16
87                end
88                else begin
89                     next_state = LOAD23; // 17
90                end
91        LOAD22 : next_state = LOAD23; // 16
92        LOAD23 : next_state = LOAD24; // 17
93        LOAD24 : next_state = LOAD25; // 18
94        LOAD25 : next_state = LOAD26; // 19
95        LOAD26 : next_state = LOAD27; // 1A
96        LOAD27 : if (flag_s) begin // 1B
97                     next_state = LOAD28; // 1C
98                end
99                else begin
100                    next_state = IDLE;
101               end
102       LOAD28 : next_state = LOAD29; // 1C
103       LOAD29 : next_state = IDLE; // 1D
104       default: next_state = IDLE;
105     endcase

107 endmodule
```

Verhaltensbeschreibung STEUERLOGIK

```
1   // Verilog HDL for "Beispiel", "KA_Controllogic
        " "functional"

3   module KA_Controllogic( state,
4           C_EN_R1, C_EN_R2, C_EN_R3, C_EN_R4,
                C_EN_R5, C_EN_R6, C_EN_R7,
5           C_MX_A, C_MX_B, C_MX_C, C_MX_D,
                C_MX_R1, C_MX_R2, C_MX_R3, C_MX_R4
                , C_MX_R5, C_MX_R6, C_MX_R7,
6           C_SUB, C_ALU_EN, C_MUL_EN, C_DIV_EN,
                C_SET_S, C_SET_Z, ram_wr_en,
                ram_rd_en, ready);

8     parameter IDLE = 5'b00000;
9     parameter LOAD1 = 5'b00001;
10    parameter LOAD2 = 5'b00010;
11    parameter LOAD3 = 5'b00011;
12    parameter LOAD4 = 5'b00100;
13    parameter LOAD5 = 5'b00101;
14    parameter LOAD6 = 5'b00110;
15    parameter LOAD7 = 5'b00111;
16    parameter LOAD8 = 5'b01000;
17    parameter LOAD9 = 5'b01001;
18    parameter LOAD10 = 5'b01010;
19    parameter LOAD11 = 5'b01011;
20    parameter LOAD12 = 5'b01100;
21    parameter LOAD13 = 5'b01101;
22    parameter LOAD14 = 5'b01110;
23    parameter LOAD15 = 5'b01111;
24    parameter LOAD16 = 5'b10000;
25    parameter LOAD17 = 5'b10001;
26    parameter LOAD18 = 5'b10010;
27    parameter LOAD19 = 5'b10011;
28    parameter LOAD20 = 5'b10100;
29    parameter LOAD21 = 5'b10101;
30    parameter LOAD22 = 5'b10110;
31    parameter LOAD23 = 5'b10111;
32    parameter LOAD24 = 5'b11000;
33    parameter LOAD25 = 5'b11001;
34    parameter LOAD26 = 5'b11010;
35    parameter LOAD27 = 5'b11011;
36    parameter LOAD28 = 5'b11100;
37    parameter LOAD29 = 5'b11101;

39    input [4:0] state;
```

```
41    output    C_EN_R1;
42    output    C_EN_R2;
43    output    C_EN_R3;
44    output    C_EN_R4;
45    output    C_EN_R5;
46    output    C_EN_R6;
47    output    C_EN_R7;
48    output [2:0] C_MX_A;
49    output [2:0] C_MX_B;
50    output [2:0] C_MX_C;
51    output [2:0] C_MX_D;
52    output    C_MX_R1;
53    output    C_MX_R2;
54    output    C_MX_R3;
55    output    C_MX_R4;
56    output    C_MX_R5;
57    output    C_MX_R6;
58    output    C_MX_R7;
59    output    C_SUB; // 0 -> ADD, 1 -> SUB
60    output    C_ALU_EN;
61    output    C_MUL_EN;
62    output    C_DIV_EN;
63    output    C_SET_S;
64    output    C_SET_Z;
65    output    ram_wr_en;
66    output    ram_rd_en;
67    output    ready;


70    reg       C_EN_R1;
71    reg       C_EN_R2;
72    reg       C_EN_R3;
73    reg       C_EN_R4;
74    reg       C_EN_R5;
75    reg       C_EN_R6;
76    reg       C_EN_R7;
77    reg  [2:0] C_MX_A;
78    reg  [2:0] C_MX_B;
79    reg  [2:0] C_MX_C;
80    reg  [2:0] C_MX_D;
81    reg       C_MX_R1;
82    reg       C_MX_R2;
83    reg       C_MX_R3;
84    reg       C_MX_R4;
85    reg       C_MX_R5;
86    reg       C_MX_R6;
87    reg       C_MX_R7;
88    reg       C_SUB;
89    reg       C_ALU_EN;
90    reg       C_MUL_EN;
91    reg       C_DIV_EN;
92    reg       C_SET_S;
93    reg       C_SET_Z;
94    reg       ram_wr_en;
95    reg       ram_rd_en;
96    reg       ready;


99    always @ (state)
100    begin
101      C_EN_R1 = 1'b0;
102      C_EN_R2 = 1'b0;
103      C_EN_R3 = 1'b0;
104      C_EN_R4 = 1'b0;
105      C_EN_R5 = 1'b0;
106      C_EN_R6 = 1'b0;
107      C_EN_R7 = 1'b0;
108      C_MX_A = 3'b000;
109      C_MX_B = 3'b000;
110      C_MX_C = 3'b000;
111      C_MX_D = 3'b000;
112      C_MX_R1 = 1'b0;
113      C_MX_R2 = 1'b0;
114      C_MX_R3 = 1'b0;
115      C_MX_R4 = 1'b0;
116      C_MX_R5 = 1'b0;
117      C_MX_R6 = 1'b0;
118      C_MX_R7 = 1'b0;
119      C_SUB  = 1'b0;
120      C_ALU_EN = 1'b0;
121      C_MUL_EN = 1'b0;
122      C_DIV_EN = 1'b0;
123      C_SET_S = 1'b0;
124      C_SET_Z = 1'b0;
125      ready  = 1'b0;
126      ram_wr_en= 1'b0;
127      ram_rd_en= 1'b0;
128      ram_rd_en= 1'b0;

130      // case statement
131      case (state)
132       IDLE : begin
133                ready   = 1'b1; // ready if FSM
                       returns to idle state
134              end
135       LOAD1 : begin
136              C_MX_C  = 3'b000; // 0 -> C
137              ram_rd_en = 1'b1; // read MEM
138              end
139       LOAD2 : begin
140              C_MX_A   = 3'b000; // MEM -> A
141              C_MX_R1  = 1'b0; // A -> R1
142              C_EN_R1  = 1'b1; // capture R1
143              end
144       LOAD3 : begin
145              ram_rd_en = 1'b1; // read MEM
146              C_MX_A  = 3'b001; // R1 -> A
147              C_MX_B  = 3'b011; // 1 -> B
148              C_MX_D  = 3'b000; // 1 -> MX_B
149              C_ALU_EN = 1'b1; // ALU enable

151              end
152       LOAD4 : begin
153              C_MX_C   = 3'b001; // ALU -> C
154              C_MX_R7  = 1'b1; // C -> R7
155              C_EN_R7  = 1'b1; // capture R7
156              C_MX_A   = 3'b000; // MEM -> A
157              C_MX_R2  = 1'b0; // A -> R2
158              C_EN_R2  = 1'b1; // capture R2

160              end
161       LOAD5 : begin
```

```
162             ram_rd_en = 1'b1; // read MEM
163             C_MX_A  = 3'b110; // R7 -> A
164             C_MX_B  = 3'b011; // 13 -> B
165             C_MX_D  = 3'b101; // 13 -> MX_B
166             C_MUL_EN = 1'b1; // MUL enable
168             end
169         LOAD6 : begin
170             C_MX_A  = 3'b000; // MEM -> A
171             C_MX_R4 = 1'b0; // A -> R4
172             C_EN_R4 = 1'b1; // capture R4
173             C_MX_C  = 3'b010; // MUL -> C
174             C_MX_R5 = 1'b1; // C -> R5
175             C_EN_R5 = 1'b1; // capture R5
177             end
178         LOAD7 : begin
179             C_MX_A  = 3'b010; // R2 -> A
180             C_MX_B  = 3'b011; // 4 -> B
181             C_MX_D  = 3'b010; // 4 -> MX_B
182             C_DIV_EN = 1'b1; // DIV1 enable
184             end
185         LOAD8 : begin
186             C_MX_A  = 3'b100; // R4 -> A
187             C_MX_B  = 3'b011; // 4 -> B
188             C_MX_D  = 3'b010; // 4 -> MX_B
189             C_DIV_EN = 1'b1; // DIV2 enable
191             end
192         LOAD9 : begin
193             C_MX_A  = 3'b101; // R5 -> A
194             C_MX_B  = 3'b011; // 5 -> B
195             C_MX_D  = 3'b011; // 5 -> MX_B
196             C_DIV_EN = 1'b1; // DIV3 enable
198             end
199         LOAD10 : begin
200             ram_rd_en = 1'b1; // read MEM
201             C_MX_C  = 3'b011; // DIV1 -> C
202             C_MX_R7 = 1'b1; // C -> R7
203             C_EN_R7 = 1'b1; // capture R7
204             C_DIV_EN = 1'b1; // DIV enable
206             end
207         LOAD11 : begin
208             C_MX_A  = 3'b000; // MEM -> A
209             C_MX_R3 = 1'b0; // A -> R3
210             C_EN_R3 = 1'b1; // capture R3
211             C_MX_C  = 3'b011; // DIV2 -> C
212             C_MX_R5 = 1'b1; // C -> R5
213             C_EN_R5 = 1'b1; // capture R5
214             C_DIV_EN = 1'b1; // DIV enable
216             end
217         LOAD12 : begin
218             C_MX_C  = 3'b011; // DIV3 -> C
219             C_MX_R6 = 1'b1; // C -> R6
220             C_EN_R6 = 1'b1; // capture R6
221             C_MX_A  = 3'b100; // R4 -> A
222             C_MX_B  = 3'b011; // 2 -> B
223             C_MX_D  = 3'b001; // 2 -> MX_B
224             C_MUL_EN = 1'b1; // MUL enable
226             end
227         LOAD13 : begin
228             C_MX_C  = 3'b010; // MUL -> C
229             C_MX_R1 = 1'b1; // C -> R1
230             C_EN_R1 = 1'b1; // capture R1
231             C_MX_A  = 3'b110; // R7 -> A
232             C_MX_B  = 3'b000; // R5 -> B
233             C_ALU_EN = 1'b1; // ALU enable
235             end
236         LOAD14 : begin
237             C_MX_C  = 3'b001; // ALU -> C
238             C_MX_R4 = 1'b1; // C -> R4
239             C_EN_R4 = 1'b1; // capture R4
240             C_MX_A  = 3'b011; // R3 -> A
241             C_MX_B  = 3'b001; // R6 -> B
242             C_ALU_EN = 1'b1; // ALU enable
244             end
245         LOAD15 : begin
246             C_MX_C  = 3'b001; // ALU -> C
247             C_MX_R5 = 1'b1; // C -> R5
248             C_EN_R5 = 1'b1; // capture R5
250             end
251         LOAD16 : begin
252             C_MX_A  = 3'b010; // R2 -> A
253             C_MX_B  = 3'b000; // R5 -> B
254             C_ALU_EN = 1'b1; // ALU enable
256             end
257         LOAD17 : begin
258             C_MX_C  = 3'b001; // ALU -> C
259             C_MX_R7 = 1'b1; // C -> R7
260             C_EN_R7 = 1'b1; // capture R7
261             C_MX_A  = 3'b001; // R1 -> A
262             C_MX_R5 = 1'b0; // A -> R5
263             C_EN_R5 = 1'b1; // capture R5
265             end
266         LOAD18 : begin
267             C_MX_A  = 3'b100; // R4 -> A
268             C_MX_B  = 3'b010; // R7 -> B
269             C_ALU_EN = 1'b1; // ALU enable
271             end
272         LOAD19 : begin
273             C_MX_C  = 3'b001; // ALU -> C
274             C_MX_R1 = 1'b1; // C -> R1
275             C_EN_R1 = 1'b1; // capture R1
276             C_MX_B  = 3'b011; // 7 -> B
277             C_MX_D  = 3'b100; // 7 -> MX_B
278             C_MX_R6 = 1'b0; // B -> R6
279             C_EN_R6 = 1'b1; // capture R6
281             end
282         LOAD20 : begin
283             C_MX_A  = 3'b001; // R1 -> A
```

```
284                 C_MX_B  = 3'b000; // R5 -> B
285                 C_SUB   = 1'b1; // SUB
286                 C_SET_S = 1'b1; // ALU store
                        FLAG_S
287                 C_ALU_EN = 1'b1; // ALU enable
289                 end
290         LOAD22 : begin // flag_s
291                 C_MX_A  = 3'b101; // R5 -> A
292                 C_MX_B  = 3'b100; // R1 -> B
293                 C_SUB   = 1'b1; // SUB
294                 C_ALU_EN = 1'b1; // ALU enable
296                 end
297         LOAD23 : begin
298                 C_MX_C  = 3'b001; // ALU -> C
299                 C_MX_R7 = 1'b1; // C -> R7
300                 C_EN_R7 = 1'b1; // capture R7
302                 end
303         LOAD24 : begin
304                 C_MX_A  = 3'b110; // R7 -> A
305                 C_MX_B  = 3'b001; // R6 -> B
306                 C_DIV_EN = 1'b1; // DIV enable
308                 end
309         LOAD25 : begin //Warte au DIV
310                 C_DIV_EN = 1'b1; // DIV enable
311                 end
312         LOAD26 : begin //Warte au DIV
313                 C_DIV_EN = 1'b1; // DIV enable
315                 end
316         LOAD27 : begin
317                 C_MX_C  = 3'b100; // DIV -> C
318                 C_MX_R4 = 1'b1; // C -> R4
319                 C_EN_R4 = 1'b1; // capture R4
320                 ram_wr_en = 1'b1;
322                 end
323         LOAD28 : begin // flag_s
324                 C_MX_A  = 3'b111; // R6 -> A
325                 C_MX_B  = 3'b101; // R4 -> B
326                 C_SUB   = 1'b1; // SUB
327                 C_ALU_EN = 1'b1; // ALU enable
329                 end
330         LOAD29 : begin
331                 C_MX_C  = 3'b001; // ALU -> C
332                 C_MX_R4 = 1'b1; // C -> R4
333                 C_EN_R4 = 1'b1; // capture R4
334                 ram_wr_en = 1'b1;
336                 end
337       endcase
338       // end of case statement
339     end
341 endmodule
```

Verhaltensbeschreibung TOPZELLE

```
1  // Verilog HDL for "Beispiel", "KA_Top_tb " "
       functional"
4  module KA_Top_tb(load);
6    parameter CYCLE = 50;
8  // declarations
9    wire [31:0] DATA_MEM_RD;
10   wire [31:0] DATA_MEM_WR;
11   wire [31:0] ADR;
12   wire        ready;
13   wire        ram_wr_en;
14   wire        ram_rd_en;
16   reg         clk, reset;
17   output      load;
18   reg         load;
20 // RAM
21   KA_Memory MEM(
22     .ADR(ADR),
23     .DIN(DATA_MEM_WR),
24     .DOUT(DATA_MEM_RD),
25     .clk(clk),
26     .wr_en(ram_wr_en),
27     .rd_en(ram_rd_en)
28   );
30 // DUT
31   KA_Top DUT(
32     .ADR(ADR),
33     .DIN(DATA_MEM_RD),
34     .DOUT(DATA_MEM_WR),
35     .clk(clk),
36     .reset(reset),
37     .ram_wr_en(ram_wr_en),
38     .ram_rd_en(ram_rd_en),
39     .ready(ready),
40     .load(load)
41   );
43 // Clock
44   initial clk = 1'b0;
45   always #(CYCLE/2) clk = ~clk;
47 // Reset
48   initial begin
49    reset = 1'b1;
50    #(3*CYCLE )
51    reset = 1'b0;
```

```
52 // special setup for certain state:
53
54   end
55
56 // Load-Signal
57   initial begin
58     load = 0;
59     #(5*CYCLE)
60     load = 1;
61     #(CYCLE)
62     load = 0;
63
64     while (ready == 0) begin
65       #(CYCLE);
66       load = 0;
67
68     end
69
70     $finish;
71   end
72
73 // Delay Annotation
74 // initial begin
75
76 // end
77 endmodule
```

B. Verzeichnisse

Abkürzungsverzeichnis

ADR	Address
ALU	Arithmetic logic unit
DIV	Dividierer
MUL	Multiplizierer
MEM	Memory
MUX	Multiplexer
FSM	Finite State Machine
DUT	Device under Test
ASIC	Application-specific integrated circuit
GUI	Graphical User Interface
CTS	Clock Tree Synthesis
DUT	Device Under Test

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Gla08] Georg Glaeser. *Der mathematische Werkzeugkasten. Anwendungen in Natur und Technik*. Elsevier, München, 3 edition, 2008. S. 408.

[Sto10] J. R. Stockton. Die kalenderarbeiten von rektor chr. zeller: Die formeln für wochentag und ostern. *Merlyn*, 2010. Archiviert vom Original am 2013-07-29.

[Zel82] C. Zeller. Die grundaufgaben der kalenderrechnung auf neue und vereinfachte weise gelöst. *Württembergische Vierteljahrshefte für Landesgeschichte*, 1882.

[Zel85] C. Zeller. Kalender-formeln. *Mathematisch-naturwissenschaftliche Mitteilungen des mathematisch-naturwissenschaftlichen Vereins in Württemberg*, 1885.