# Projekt Gegengleis

Modul Schaltkreis- und Systementwurf

des Lehrstuhls für Hochparallele VLSI-Systeme und Neuromikroelektronik (HPSN)

bearbeitet von Jonas Bechtel

im Diplomstudiengang Elektrotechnik, Richtung Informationstechnik

an der TU Dresden

formelle Abgabe am 28. Oktober 2016

Version vom November 2016 / Januar 2017

gewidmet Dr. Matthias Bär

# Einleitung

Sie halten meinen Projektbericht zu Schaltkreis- und Systementwurf in Ihren Händen. Der Algorithmus, den ich ausgewählt habe, dreht sich um eine fliegende Überholung von Zügen.

Dieser Algorithmus hat dazu geführt, dass das Projekt und der Bericht, verglichen mit den Projekten mancher Kommilitonen, umfangreich sind. Er hat aber auch die Arbeit für den Schaltkreis sehr interessant gemacht, z. B. durch die Notwendigkeit der "intelligenten Zustandsnummern", welche zu einem niedrigem Gatterverbrauch geführt hat.

Insgesamt hat mir das Projekt einige Aspekte des digitalen Schaltkreisentwurfes aufgezeigt, auch wenn ich nicht alle Themenfelder berücksichtigen konnte. (So sind z. B. die Treiberstärken grundsätzlich auf die niedrigste Stufe, was sicher in der Realität nicht funktionieren kann.)

Aufgrund der langen Projektbearbeitungszeit und um mich in den Planungstabellen kurz fassen zu können, gibt es, quer über das ganze Projekt verteilt, unterschiedliche Namen für dieselben Zustände und Variablen. Dazu habe ich ein Synonymregister angelegt, das Sie im Anhang finden können.

Herzlichen Dank an Cornelia Bechtel und Dr. Harald Krüger für das Gegenlesen einer Vorversion dieses Dokuments und die nützlichen Anmerkungen!

Dieses Projekt ist Dr. Matthias Bär (an der Professur für Bahnverkehr, öffentlichen Stadt- und Regionalverkehr) gewidmet, dessen Lehrveranstaltung (Betriebsführung von Bahnen) ich besucht habe und die mir gezeigt hat, dass es üblich ist, exakt zu rechnen.

Das Titelbild zeigt die Sekunde 2002 des Testfalls Ueberholung_x3_Blockgrenze_1f.

Falls Sie nur diesen Text vor sich haben, nicht aber die Projektdateien: unter http://jbechtel.de/site/Tools/SkS-Gegengleis/ stelle ich das Projektverzeichnis als tar.bz2-Archiv ins Internet.

## Die Aufgabe

Das übergeordnete Ziel des Praktikums ist laut der Praktikumsanleitung die "Realisierung einer algorithmischen Aufgabenstellung als integrierter Schaltkreis". In den vorbereitenden Vorlesungen und in der Praktikumsanleitung wurde genauer skizziert, wie der Schaltkreis und die Umgebung auszusehen haben:

Es gibt drei wesentliche Komponenten: einen Datenpfad als Schaltplan, und einen Zustandsautomaten (Finite State Machine = FSM) und die Steuerlogik als Verhaltensbeschreibung in Verilog. Im Datenpfad sind die Register und die Recheneinheiten (ALU, MUL, ...) enthalten und mit Bussen verbunden. Insgesamt handelt es sich um einen rein digitalen Schaltkreis. Der Steuerungskreislauf ist im Wesentlichen der folgende:

1. Die FSM wechselt die Zustände; die Verzweigungen werden abhängig von den Ergebnis-Flags der Recheneinheiten im Datenpfad geschaltet.
2. Anhand des Zustands der FSM generiert die Steuerlogik Steuersignale.
3. Der Datenpfad rechnet mit den Registern und Recheneinheiten, die durch die Steuersignale festgelegt werden.

Den Algorithmus dürfen sich die Studentinnen und Studenten selbst aussuchen.

Das Projekt muss in einem Beleg mit einigen Details dokumentiert sein.

# Inhalt

# Der Algorithmus

Kurznamen:
c-ref/c-ref
c-ref/c-eenum

In diesem Kapitel wird der vollständige Referenzalgorithmus und der als Schaltung implementierte Enumerationsalgorithmus vorgestellt. Beide Algorithmen für dieses Projekt sind, kurz gesagt, ein Optimierungsalgorithmus für eine bahnbetriebliche Situation. Er ist so ausgelegt, dass er keine gefährlichen Zustände als Lösung ausgibt. Trotzdem sollten die Ergebnisse nicht als "Fahraufträge" (= Signal auf Fahrtstellung) verstanden werden, sondern müssten, so der Algorithmus tatsächlich im Bahnbetrieb verwendet würde, in der jeweiligen Situation von dem Stellwerkssystem auf gefährliche Zustände überprüft werden.

## Zugrundeliegendes Problem: Überholung

Als Ausgangspunkt für die Algorithmenfindung dient die Betrachtung einer beliebig langen zweigleisigen Eisenbahnstrecke ohne Abzweige, aber mit Überleitstellen. Eine Überleitstelle ermöglicht den Wechsel zwischen dem Richtungsgleis (normales Gleis, in Fahrtrichtung rechts) und dem Gegengleis (in Fahrtrichtung links). Auf diesen Gleisen fahren Züge ohne Fahrplan mit unterschiedlichen Geschwindigkeiten. Sobald ein langsamer Zug einem schnellen vorausfährt, verliert der schnelle Zug Zeit, d. h. der schnelle Zug kommt etwas später an, als er angekommen wäre, wenn der langsame Zug nicht vor ihm gefahren wäre.

Zum Verhindern dieses Zeitverlusts soll eine fliegende Überholung stattfinden, d. h. entweder kann sich der langsame Zug überholen lassen, d. h. der langsame Zug fährt rechtzeitig in das Gegengleis (und fährt dort weiter), sodass der schnelle Zug im Regelgleis ungehindert fahren kann. Oder der schnelle Zug kann überholen, d. h. der schnelle Zug fährt ins Gegengleis.

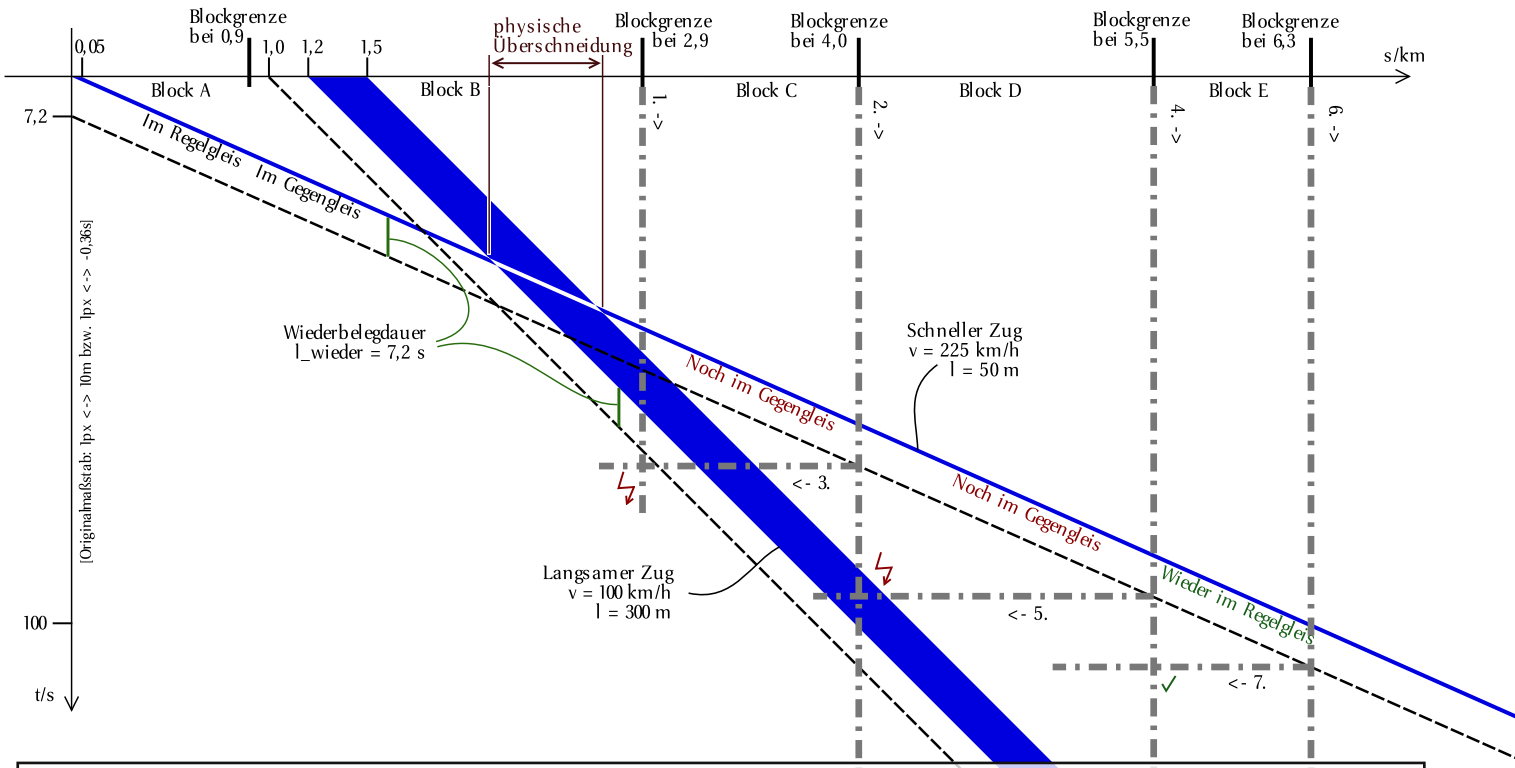
Aufgabe des Algorithmus ist die Ausgabe eines kostenoptimalen "Fahrplans" für die Strecke bzw. verkürzt die Ausgabe der Kosten und ob eine Überholung stattfindet oder nicht.

Kompliziert wird die Berechnung dadurch, dass sich zusätzlich andere Züge auf der Strecke befinden, deren Nutzung je nach Konstellation einen Konflikt mit dem Überholvorgang darstellt. Dies führt auch dazu, dass es sinnvoll sein kann, strategisch frühzeitig etwas langsamer zu fahren oder zu warten, da sonst später die Situation verklemmt sein (Deadlock) oder andersartig ungünstig sein könnte.

## Bahnspezifische und betriebliche Vereinfachungen

Um überhaupt einen Algorithmus zu finden, für den 64 Zustände ausreichen (und damit der Entwurf des Algorithmus und die Findung von Testfällen in einer geringen Zeit durchführbar ist), musste ich eine ganze Reihe von vereinfachten Annahmen treffen, die im Folgenden aufgezählt sind:

- Die Strecke ist endlich, d. h. besteht nur aus einer definierten Zahl an Blöcken.
- Kein (End-)Halt, kein Wenden auf der Strecke. (Die Züge treten durch den Anfang ein und verlassen die Strecke durch das Ende.)
- Es gibt nur drei Züge: den langsamen (L), den schnellen (S) und den Gegenzug (G).
- Die Strecke zwischen zwei benachbarten Überleitstellen wird Block genannt. Sämtliche Blöcke sind gleich lang, d. h. die Überleitstellen haben einen konstanten Abstand.
- Die Überleitstellen sind sehr kurz und in alle Richtungen und mit Streckengeschwindigkeit befahrbar.
- Die Züge haben eine unendliche Anfahr-/Brems-Beschleunigung, d. h. sie können z. B. aus dem Stand mit ihrer Höchstgeschwindigkeit fahren.
- Die Strecke hat keine Geschwindigkeitsbegrenzung.
- Der Energieverbrauch spielt keine Rolle, nur der Zeitverlust.
- Der Überholvorgang muss innerhalb der betrachteten Strecke begonnen und beendet werden; dies wird




**Bildseite: So sieht eine fliegende Zugüberholung im verwendeten Modell aus.**

Zu sehen ist eine zweigleisige Strecke mit ungleichmäßigen Blockabständen und zwei Züge, die auf dieser Strecke fahren. Desweiteren sind noch einige Hilfslinien eingezeichnet.

Die blauen Flächen stellen die physische Belegung der Strecke durch die beiden Züge dar. Der schnelle Zug (S) ist mit einer Länge von 50 Meter vergleichsweise kurz, d. h. er ist zu jedem Diagramm-Zeitpunkt 50 Meter breit. Er fährt mit 225 km/h, d. h. an jeder (Diagramm-)Stelle hält er sich 0,8 Sekunden auf. Der langsame Zug (L) ist 300 Meter lang und 100 km/h schnell, d. h. an jeder (Diagramm-)Stelle hält er sich 10,8 Sekunden auf. Durch die 300 Meter und 10,8 Sekunden hat der physische Belegungsstreifen des langsamen Zuges eine sehr dicke Optik.

Die gestrichelte Linie, die jeweils in konstantem Abstand (l_wieder) unter den Zügen eingezeichnet ist, hilft bei der Bestimmung der Freigabezeit der Blöcke. Sie stellt die technische Signalverzögerung bis zur Freigabe eines Blocks dar, nachdem der letzte Zug ausgefahren ist. (Streng genommen würde es reichen, statt der gestrichelten Linien bei jeder Blockgrenze einen Abstandhalter unter den Zug zu setzen, der genau die selbe Länge wie die grünen l_wieder-Abstandhalter im Bild hat. Die gewählte Darstellung hat aber den Vorteil, dass sie zeigt, dass die Signalverzögerung auch in eine virtuelle zusätzliche Zuglänge uminterpretiert werden kann.)

Einen Teil der Berechnung des Algorithmus kann man mit dem Bild auf dieser Seite rein grafisch nachvollziehen: Es geht um die Frage, welche Blöcke im Gegengleis gefahren werden. Dass im Block B nicht beide Züge in einem Gleis fahren können, ist klar, denn dort überschneiden sie sich physisch. Für die Blöcke C und D muss man das Blockende auswerten; dort (4,0 km) kreuzt zu einem bestimmten Zeitpunkt die l_wieder-Linie die Blockgrenze. Von dort muss man die Strecke zurückgehen (horizontal nach links bis km 2,9, siehe Linie "3."). Falls der Schnittpunkt mit der linken Blockgrenze unterhalb vom oder im physischen Bereich von L liegt, liegt eine Überschneidung vor, d. h. der Block ist noch mit S beschäftigt (S befindet sich physich im Block oder die Signale hatten noch nicht die l_wieder = 7,2 Sekunden Rechenzeit), während L schon eintritt. Also muss entweder L warten oder S fährt auch diesen Block im Gegengleis. Bei Block C liegt der Schnittpunkt ("1."/"3.") unterhalb von L, bei Block D liegt der Schnittpunkt ("2."/"5.") innerhalb von L. Erst Block E kann von beiden Zügen im Regelgleis befahren werden, denn der Schnittpunkt ("4."/"7.") befindet sich überhalb von L. Dies ist hier mit einem grünen Häkchen am Schnittpunkt markiert.

Außerdem lässt sich hier grafisch nachvollziehen, dass man zur Überprüfung der Überschneidung über mehrere zukünftige Blöcke iterieren muss, denn je nach Lage der Blockgrenzen kann die Anzahl der im Gegengleis zu befahrenden Blöcke variieren.

Im richtigen Algorithmus wird die Iteration analog auch für die Blöcke vor der Überholung durchgeführt; bei dieser gezeigten Beispielsituation ist das nicht nötig.

aber nicht durch den Algorithmus geprüft.

Dass die Strecke einen Anfang und ein Ende hat, ermöglicht eine Darstellung mit üblichen Datentypen.

Dass die Überholung in der Strecke bleibt, habe ich als Arbeitsersparnis festgelegt. Denn es wäre aufwändig gewesen, die zusätzlichen mathematischen Zusammenhänge herauszuarbeiten, das Programm so umzugestalten und noch weitere Tests zu entwerfen. Außerdem hätte ich mich für ein Streckenende-Modell entscheiden müssen, denn die Berechnung ist abhängig davon, ob an den Enden der Strecke jeweils eine weitere Strecke, ein großer Bahnhof oder eine Abzweigstelle liegt. Außerdem hängt sie dann vom übergeordneten Fahrtziel des Zuges ab.

Die Vereinfachung der Überleitstellen und der Beschleunigung führt dazu, dass es irrelevant ist, ob der schnelle oder der langsame Zug ins Gegengleis fahren. Die Vereinfachung des Energieverbrauchs ersparte mir die Modellbildung zum Antrieb, zur Bremse und zum Windwiderstand sowie die Berechnungen einer strategisch niedrigen Geschwindigkeit.

# Bahnspezifische und betriebliche Zusatzbedingungen

Das Modell ist zwar stark vereinfacht, aber es gelten die folgenden verschiedenartigen Bedingungen:

- Die Blöcke sind durch Signale abgesichert; in einem Block darf sich nur ein Zug auf einmal befinden (entscheident ist hier auch der kleinste Längenabschnitt eines Zuges, z. B. die allerletzte Achse.)
- Die Rechendauer des Signals (bzw. Signalverzögerung/Schaltverzögerung) wird vereinfachend durch einen konstanten Wert modelliert. Nachdem ein Block freigefahren wurde, ist eine konstant definierte Zeit zu warten, bis ein anderer Zug einfahren darf. Somit ist die Rechendauer nicht abhängig von der Situation, Entfernung oder Technik.
- Die Strecke ist endlich und der Überholvorgang muss innerhalb der Strecke stattfinden.

Die Endlichkeit der Strecke ist auch schon als Vereinfachung genannt worden. Die Originalidee des Algorithmus war jedoch, sich auf die Züge L, S und G zu konzentrieren. Durch die "Vereinfachung" musste ich die Testfälle so anlegen, dass durch die Überholung nicht das Streckenende berührt ist. Außerdem musste ich für die Hinterherfahren-Testfälle (Überholung nicht möglich) doch ein Modell (Bahnhof mit mehreren Einfahrtgleisen, sodass in kürzestem Abstand mehrere Züge hintereinander einfahren können) festlegen und eine Streckenende-Kontrolle implementieren, auch wenn diese nicht ganz scharf ist.

# Vom Problem zum (Referenz-)Algorithmus

Zur Algorithmusfindung bin ich alle möglichen Varianten durchgegangen, und auf die folgende Liste gekommen:

x1 S fährt hinter L her (kein Überholvorgang)

x2 S fährt ins Gegengleis und überholt dort L (bevor G auftaucht)

x3 wie x2, nur wartet S ggf. auf G; G ist unbeeinträchtigt.

~~x4~~ L fährt ins Gegengleis und lässt sich von S im Regelgleis überholen. Dann fährt L wieder ins Regelgleis. G muss ggf. warten

~~x5~~ wie x4, nur wartet L ggf. auf G; G ist unbeeinträchtigt.

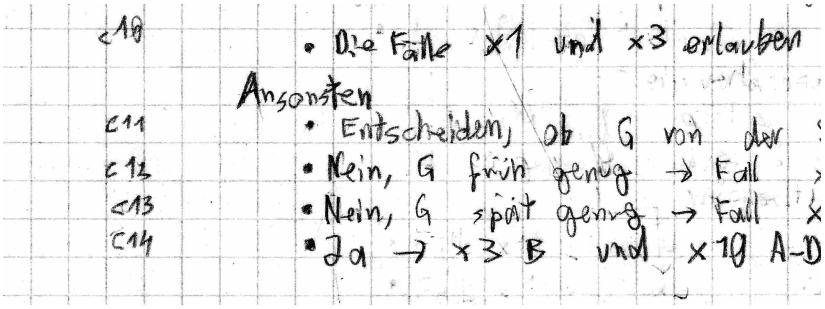
~~x6~~ L fährt ins Gegengleis; S überholt auf dem Regelgleis und fährt dann an G vorbei. Daraufhin fährt G in das Regelgleis (Regelgleis ist hier aus der Sicht von S/L gemeint) und fährt an L vorbei. Es tritt selten ein, dass dieser Fall sinnvoll ist; mit dem Vereinfachten Modell für diesen Beleg ist der Fall x6 nie sinnvoll. (Und außerdem stellt die häufige Nutzung gegen die Fahrtrichtung ein Sicherheitsrisiko dar und ist ineffizient, wenn noch mehr Züge als S, L und G auf der Strecke unterwegs sind.)

~~x7~~ S und G fahren jeweils in das Gegengleis aus ihrer Perspektive. Es gibt nur geringfügige Vorteile gegenüber x6; für das verwendete Modell nicht sinnvoll.

~~x8~~ wie x6, nur wartet G ggf.
~~x9~~ wie x7, nur wartet G ggf.
x10 S fährt ins Gegengleis und überholt L, allerdings stellt der Überholvorgang einen Konflikt mit G dar. Deswegen wartet entweder L, sodass S noch den Überholvorgang beenden kann, bevor G die Stelle passiert, oder G wartet, oder L und G teilen die Wartezeit auf. Die Berechnungen, die ich weiter unten im Beleg vornehmen werde, werden zeigen, dass entweder G oder L (je nach Geschwindigkeiten und Wichtungen) den Großteil der Zeit warten.

Ich gehe davon aus, dass der schnelle Zug und nicht der langsame zur Überholung ins Gegengleis fährt. Denn wenn statt S L ins Gegengleis fährt, bleiben die parallel belegten Blöcke die gleichen, aber der vorderste Block (der am nächsten am Streckenende liegt) wird für eine etwas längere Zeit belegt. Aus diesem Grund fallen x4 und x5 weg, indirekt auch x6/x7/x8/x9.

## Ablauf




Der Ablauf des Algorithmus setzt sich aus dem folgenden Pseudo-Code zusammen. Die c-Nummern habe ich zur Orientierung eingefügt, damit das Papier und der Quellcode gegenseitig zuordnenbar sind. Da allerdings der so nummerierte Teil nicht so groß und die Berechnung von x1, x3 und x10 viel umfangreicher ist, fallen die c-Nummern im Quelltext nicht so stark auf.

c1 Wenn das hintere Ende von S bereits den ersten Block freigegeben hat, sobald L vorne in die Strecke kommt
    Keine Überholung; Ende
c15 Wenn S (noch) im ersten Block ist, sobald L vorne in die Strecke kommt
    Keine Überholung; Rücksetzen L; Ende
c2 Treffpunkt/Treffblock ausrechnen.
c3 Falls vor Streckenbeginn
    S virtuell nach hinten setzen; die Fälle x10A, x10B und x10D sperren.
c4 Falls nach Streckenende
    Warnen; Fall x1 als einzigen berechnen. (Oder, da das Streckenende unendlich aufnahmefähig ist, gleich kosten = 0 ausgeben.)
c16 Verlassenspunkt ausrechnen. (iterativ)
c5 Die Blockzahl testen, die S noch im Ggl fährt. (c16 und c5 sind identisch und zusammengefasst)
c6 Falls die Streckenlänge überschritten wird
c7 Warnung ausgeben
c8 Falls der Treffpunkt vor dem Streckenbeginn lag
    Fehler "Überholvorgang über die ganze Strecke"; Ende
c17 L zurücksetzen -> Trennpunkt gerade vor Streckenende
c10 Die Fälle x1 und x3 erlauben
Ansonsten
c11 Entscheiden, ob G von der S-Überholstrecke betroffen ist
c12 Nein, G früh genug -> Fall x3A; Ende
c13 Nein, G spät genug -> Fall x2; Ende (c12 und c13 sind die gleiche Rechnung)
c14 Ja -> x3B und x10A-x10D ausrechnen

---

Die Fälle x1, x3 und x10A-x10D berechnen (falls nicht gesperrt).
Den Fall mit den kleinsten Kosten auswählen.
Für den ausgewählten Fall die Überholanweisungen zusammenstellen und ausgeben.

## Der C-Referenz-Algorithmus

Der Referenzalgorithmus (c-ref/c-ref aus c-ref/c-algo.c) ist die C-Fassung dieses Pseudocodes (c1, c15, c2, ...).

# Vereinfachung: Durchprobieren

Um eine geringere Zahl an Zuständen zu erreichen, ist als Variante der Durchprobier-Algorithmus implementiert worden. Die Funktionsweise ist relativ einfach:

Doppelte Schleife: Über alle Möglichkeiten der Überholstrecke iterieren.
    Kosten von S berechnen, falls S hinter L warten muss.
    Kosten von L berechnen, falls L hinter S warten muss.
    (Berücksichtigen, wenn S hinter L und dann L hinter S warten muss)
    Gibt es einen Konflikt mit dem Gegenzug?
        Kosten von G berechnen.
    Gesamtkosten geringer als bei einem vorherigen Fall
        -> Neuen Fall als Referenz speichern

Der äußere Teil der doppelten Schleife legt mit l (kleines L) die Länge, also die Blockanzahl, der Überholstrecke fest und der innere Teil der Schleife bestimmt mit x den Startblock der Überholung. Wenn eine Konstellation mit gleichen Kosten gefunden wird, wird die alte Konstellation vorgezogen, nur bei geringeren Kosten wird die neue Konstellation gespeichert. Durch diese Reihenfolge werden kurze Überholstrecken bevorzugt, was die Ergebnis-Vergleichbarkeit sichert und zugleich anschaulicher den Platzbedarf der Überholung zeigt.

## Der C-Enumerations-Algorithmus

Dieser Algorithmus ist so gößtenteils im Schaltkreis ("circuit") implementiert. Der Kurzname "c-eenum" steht für "Exhaustive Enumeration". Man könnte auch Durchprobieren, Brute-Force, Exhaustion (gemäß de.wikipe dia.org/wiki/Brute-Force-Methode), Vollständige Enumeration (nach Claude-Joachim Hamann) bzw. Enumerationsalgorithmus sagen (nach der Bachelorarbeit "Ein Tool für Enumerationsalgorithmen" von R. V. Cramer aus dem Jahr 2015).

Durch die Anwendung der Enumeration musste eine doppelte Schleife eingeführt werden. Im Gegenzug sind viele spezialisierte Schleifen und Programmteile des Referenz-Algorithmus weggefallen. Dadurch ist der Code-Umfang von 1570 Zeilen auf 569 Zeilen (bei ähnlichem Kommentaranteil) gesunken.

Allerdings mussten auch Sonderfälle gestrichen werden, die relativ genau das Streckenende behandelt haben, was der Enumerations-Algorithmus nicht so gut schafft. Ein weiterer Nachteil des Brute-Force-Vorgehens ist die quadratische Rechenzeitsteigerung mit einer Blockanzahlsteigerung.

Der Vorteil von Brute-Force ist die geringe Komplexität. Bei Erweiterungen könnten damit zudem leichter andere Zugkonstellationen eingeführt werden, z. B. dass noch ein vierter Zug auf der Strecke fährt. Für den Refernzalgorithmus müssten dagegen sowohl der Ablauf als auch die Berechnungen geändert werden.

## Numerische Stabilität (c-eenum)

Um die numerische Stabilität braucht man sich (beim Enumerationsalgorithmus) keine Sorgen zu machen. Als Indikator dafür sei angeführt, dass eine Zahl, die aus einer Differenzbildung entsteht, nur einmal multipliziert wird. (Diese Multiplikation findet ganz am Ende zur Kostenbestimmung statt.) Die Schleifenrechnungen zur Enumeration der Blöcke zählen ebenso wie der Rückwärtstausch (Umrechnung der Blocknummer für den Gegenzug) nicht als Differenz, denn ein Fehler bei diesen Rechnungen führt zu extremen Ausgabefehlern und lässt sich nicht mit numerischer Stabilität ausdrücken, sondern ist als Programmfehler zu zählen.

# Zahlenformat

## Genauigkeit

Die Rechnungen werden durchgängig mit dem genauen Format ausgeführt, d. h. in Metern und Sekunden, aber 16*1024 mal genauer; der Grund für dieses Format liegt in der reziproken Geschwindigkeit (siehe folgendes Unterkapitel). Die Angabe und Prüfung der End-Kosten ist aber auf Sekunden*Wichtungseinheiten basiert, wobei selbst bei kleinen Bruchteilen aufzurunden ist.

## Reziproke Geschwindigkeit

Am Beginn des Algorithmusentwurfs hat sich herausgestellt, dass viel mit der Geschwindigkeit gerechnet werden muss; manchmal als Faktor und manchmal als Divisor. Mit der Formel s = v*t lässt sich die zurückgelegte Strecke in einem Zeitraum berechnen; mit t = s/v die benötigte Zeit für einen Streckenabschnitt.

Durch ein Umstellen des Problems ist es möglich, statt der Faktor-Form die Divisor-Form zu erreichen: Es ist nicht mehr die Frage „Wie weit ist der Zug in der Zeit gekommen?“, sondern die Frage „Hat der Zug in der Zeit geschafft, den Block / die Zuglänge / ... zu durchfahren?“ zu stellen. Da die Blocklängen und die Zuglängen sich nicht mit der Zeit ändern, kann man $t_{vgl}$ = s/v berechnen und dann mit der fraglichen Zeit t vergleichen. Wenn man eine Information zu mehreren Blöcken benötigt, muss man entsprechend mehrmals iterieren und bildet so indirekt die Division nach. Allerdings lassen sich nebenher noch andere Algorithmusteile berechnen, weswegen die resultierende Schaltung recht elegant und effizient ist.

Mit diesem Zwischenstand (es wird nur noch t = s/v berechnet) kann die Geschwindigkeit von vornherein reziprok eingespeichert werden. Damit existiert in dem Schaltkreis keine Division mehr und der entsprechende große und zeitaufwändige Datenpfadblock kann eingespart werden.

Konkret muss dann noch die Frage nach dem Format beantwortet werden. Da ich keine Erfahrungen mit Fließkommazahlen hatte, wollte ich dieses Thema vermeiden und auch die reziproke Geschwindigkeit als ganze Zahl einspeichern. Dazu soll der reziproke Wert um eine feste Bitzahl verschoben werden (-> „Schiebezahl“), sodass genügend Stellen vor dem Komma stehen.

Zur Bestimmung der Schiebezahl musste ich den folgenden Trade-Off vornehmen, der anhand der Kern-Berechnung Streckenlänge*r_geschw*Wichtungsfaktor, deren Ergebnis in einen 32-Bit-Wert passen soll, bewertet wird: r_geschw soll die Geschwindigkeit so genau wie möglich spezifizieren, aber die Streckenlänge (in m) und der Wichtungsfaktor sollen groß werden können.

**Trade-Off, Aspekt 1: Genauigkeit der Geschwindigkeit**

Die reziproke Geschwindigkeit hat naturgemäß eine große Genauigkeit bei kleinen Geschwindigkeiten und eine kleine Genauigkeit bei großen Geschwindigkeiten. Bei einer Geschwindigkeit von ca. 500 km/h (ca. der Bereich der schnellsten Züge) soll die Geschwindigkeit trotzdem noch mittelmäßig genau sein.

Numerisch detaillierte Werte zur Einschätzung sind in der Geschwindigkeitentabelle zu finden: Wie man sehen kann, ist bei 13 Bit Schiebezahl und bei 500 km/h die Genauigkeit ca. 8,3 km/h und bei 200 km/h ca. 1,4 km/h. Dies war mir nicht genug, deswegen habe ich 14 Bit (2^14 = 16*1024) ausgewählt (Genauigkeit 4,1 km/h bzw. 0,7 km/h).

**Trade-Off, Aspekt 2: Die verfügbaren Restbits**

Als maximal auftretende Streckenlänge habe ich ca. 10 km - 50 km angenommen, als maximalem Wichtungsfaktor ca. 20, als typische Geschwindigkeit ca. 100 km/h – 200 km/h. Außerdem wird im Schaltkreis mit Metern und Sekunden gerechnet, nicht mit Kilometern und Stunden. Wieviele Bits für die multiplizierten Werte gebraucht werden, ist in der Restbittabelle zu finden: Wie man sieht, reicht eine Bitzahl von 18, um das komplette Ergebnis des gewichteten Zeitverlusts zu speichern. Es bleiben also 14 Bit übrig. Wenn nun statt s/v*w die Rechnung s*(2^14*1/v)*w verwendet wird, passt das Ergebnis genau in 32 Bit. Rechnungen mit [v] = m/s.

Ein anmerkenswerter Zusammenhang ist noch, dass ein Zug bei einer längeren Strecke schneller fahren muss, damit die (gewichtete) Fahrzeit innerhalb der gegebenen Bitanzahl bleibt. Bei normalen/üblichen Streckenlängen und normalen Geschwindigkeiten und geringen Wichtungen muss sich der Testverantwortliche keine Gedanken machen.

$$\frac{v}{\left[\frac{km}{h}\right]} = \frac{1024*16*3{,}6}{\left(\frac{r_{geschw}}{\left[\frac{s}{1024*16*m}\right]}\right)}$$

**Trade-Off, Abwägung**

Nach den beiden genannten Trade-Off-Aspekten lässt sich also feststellen: mit reinen 32-Bit-Multiplikationen und -Additionen lässt sich eine akzeptable Streckenlänge bei einer akzeptablen Geschwindigkeitsspanne und -genauigkeit (14 Bit) darstellen, solange der Wichtungsfaktor maximal ca. 100 beträgt (was wiederum ein akzeptabler Wert ist). Daraus ergeben sich die vier rechts stehenden einfachen Handrechenformeln.

$$\frac{v}{\left[\frac{m}{s}\right]} = \frac{1024*16}{\left(\frac{r_{geschw}}{\left[\frac{s}{1024*16*m}\right]}\right)}$$

$$\frac{r_{geschw}}{\left[\frac{s}{1024*16*m}\right]} = \frac{1024*16*3{,}6}{\left(\frac{v}{\left[\frac{km}{h}\right]}\right)} = \frac{1024*16}{\left(\frac{v}{\left[\frac{m}{s}\right]}\right)}$$

*Geschwindigkeitentabelle*

| Geschwin-digkeit | r_geschw *2^**13**/(s/m) | → v | | r_geschw ***16*1024** /(s/m) | → v | | r_geschw *2^**15**/(s/m) | → v | |
|---|---|---|---|---|---|---|---|---|---|
| | | in (m/s) | in (km/h) | | in (m/s) | in (km/h) | | in (m/s) | in (km/h) |
| 200 km/h =55,6 m/s | **146** | 56,1 | 202,0 | **293** | 55,91 | 201,3 | **589** | 55,63 | 200,3 |
| | **147** | 55,7 | 200,6 | **294** | 55,73 | 200,6 | **590** | 55,54 | 199,9 |
| | **148** | 55,4 | 199,3 | **295** | 55,54 | 199,9 | **591** | 55,45 | 199,6 |
| | **149** | 55,0 | 197,9 | **296** | 55,35 | 199,3 | | | |
| 500 km/h =138,9 m/s | **58** | 141,2 | 508,5 | **117** | 140,0 | 503,7 | **235** | 139,4 | 502,0 |
| | **59** | 138,8 | 499,9 | **118** | 138,8 | 499,6 | **236** | 138,8 | 499,9 |
| | **60** | 136,5 | 491,6 | **119** | 137,7 | 495,5 | **237** | 138,3 | 497,7 |

*Restbittabelle*

| s | | v | | w | s/v*w in s*WE | | Ergeb-nisbits | Übrige Bits |
|---|---|---|---|---|---|---|---|---|
| 10 km | 10.000 m | 200 km/h | 55,56 m/s | 20 WE | 3600 | 0,9*2^12 | 12 | 20 |
| 100 km | 100.000 m | 100 km/h | 27,78 m/s | 50 WE | 180000 | 0,7*2^18 | 18 | 14 |
| 100 km | 100.000 m | 60 km/h | 16,67 m/s | 50 WE | 300000 | 0,6*2^19 | 19 | **13** |
| 70 km | 70.000 m | 100 km/h | 27,78 m/s | 100 WE | 252000 | 1,0*2^18 | 18 | 14 |
| 215 km | 215.000 m | 200 km/h | 55,56 m/s | 50 WE | 193500 | 0,7*2^18 | 18 | 14 |
| 100 km | 100.000 m | 60 km/h | 16,67 m/s | 10 WE | 60000 | 0,9*2^16 | 16 | 16 |

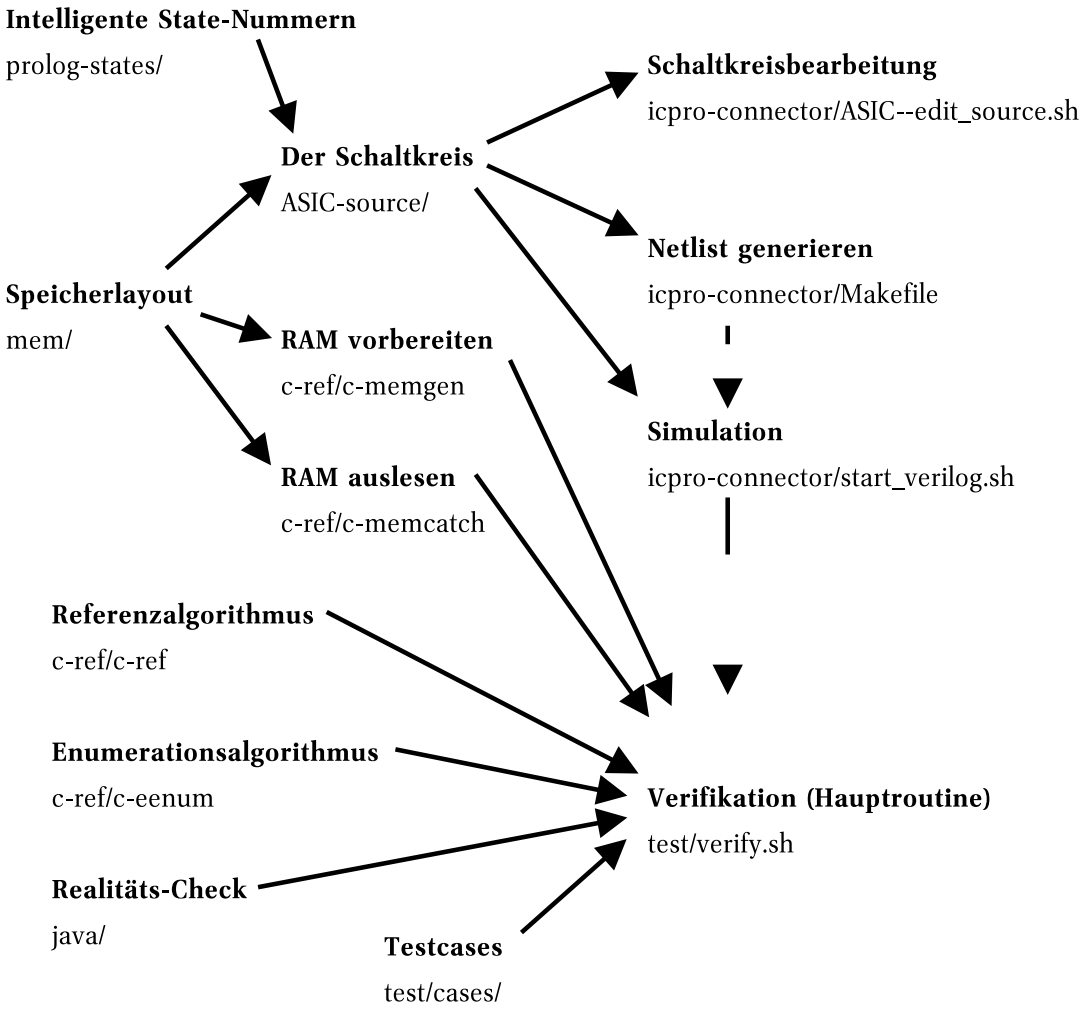
# Zusammenspiel der Komponenten

Überblicksweise seien hier der vollständige Ablauf des Projekts und die technischen Abhängigkeiten dargestellt. Der Ablauf beim Erstellen und Testen (z. B. aus einem Backup heraus) ist der folgende:

- Realitätscheck kompilieren (java/Makefile)
- Referenzalgorithmus kompilieren (c-ref/Makefile)
- Enumerationsalgorithmus kompilieren (auch c-ref/Makefile)
- Speicherlayout in die beteiligten Sprachen umsetzen (mem/generate-layout-files.sh)
- Speicher-Hilfsprogramme kompilieren (c-ref/c-mem*)
- Intelligente State-Nummern finden (prolog-states/)
- ICPRO-Projekt erzeugen, die Verilog-Dateien dorthin kopieren, dann den Netlister laufen lassen (mit icpro-connector/Makefile)
- Den Schaltkreis verifizieren (test/verify.sh mit test/cases/*)

Diese Auflistung spiegelt ungefähr meine Bearbeitungsreihenfolge des Projekts wider; abweichend davon habe ich u. a. die Verifikation mit dem Realitäts-Check und dem Referenzalgorithmus zusammen entwickelt und später für den Enumerationsalgorithmus angepasst.

Die Abhängigkeiten der Komponenten sind in der folgenden Zeichnung dargestellt:



# Die Implementationen testen

Kurznamen:
test/cases/
test/verify.sh

Da der Gegengleis-Algorithmus (bzw. später die Schaltung) verschiedene Zug-Konstellationen korrekt handhaben muss, muss er/sie mit verschiedenen Testfällen geprüft werden. Da das von vorneherein klar war, konnte die Grundstruktur des Testsystems sehr schnell entworfen werden; sie wurde dann entsprechend dem Implementierungsfortschritt fortentwickelt.

Für den Referenzalgorithmus sind 20 Testfälle implementiert. Davon können sieben mit dem Enumerationsalgorithmus funktionieren und sechs mit dem endgültigen Schaltkreis.

Ein Testfall sieht folgendermaßen aus:

*test/cases/Ueberholung_x3_Blockgrenze_1d2*

```
# © Jonas Bechtel 2014



# Hier findet eine Ueberholung statt, (bei Block 4-10)
# nachdem G vorbeigefahren ist.

Strecke num 11 single_length 5500 l_wieder 10

Zug langsam d 370 e 1506 w 1 l 1000
Zug schnell d 262 e 1736 w 1 l  300
Zug gegen   d 370 e 0994 w 1 l   80

# Bei t = 2000s verlässt G Block 4 und L Block 3 (l_wieder schon einbezogen)

# S ist eine Sekunde zu früh.

veri Kosten 1
veri Ueberholung_vorgesehen ja
```

Mit anderen Worten ausgedrückt:

- Es wird eine Strecke mit 11 Blöcken definiert, die jeweils 5.500 Meter lang sind. Insgesamt handelt es sich also um 60,5 *km*.
- Vom Freiwerden eines Abschnitts bis zur Fahrtstellung der Signale dauert es 10 Sekunden.
- Der langsame Zug fährt bei Sekunde 1.506,0 nach unserer Zeitzählung in den ersten Block ein. Er ist 1.000 Meter lang und fährt mit 370 *s/m/1024/16*, hat also eine Geschwindigkeit von circa 159 *km/h*.
- Der schnelle Zug fährt bei Sekunde 1.736,0 in den ersten Block ein. Er ist 300 Meter lang und ist ca. 225 *km/h* schnell.
- Der Gegenzug fährt bei Skeunde 994,0 in den elften Block ein, ist 80 Meter lang und fährt ca. 159 *km/h*.
- Der Algorithmus soll eine Fahrplanung ausgeben, mit der Kosten in Höhe von 1 *WE*s* entstehen.
- Der Algorithmus soll er eine Fahrplanung aus, die den schnellen Zug überholen lässt.

# Exemplarische manuelle Berechnung

## Treffpunkt

Zur manuellen Berechnung berechnen wir zuerst den physischen Treffpunkt des langsamen und schnellen Zugs (vordere Spitze gleichauf). Dazu formen wir die folgende Formel um:

$e_S + s_S/v_S = e_L + s_L/v_L \mid s_S = s_L = s;\ e_S = 1.736\ s;\ e_L = 1.506\ s$
$1.736\ s + s/v_S = 1.506\ s + s/v_L \mid - 1.506\ s \mid - s/v_S$
$230\ s = s/v_L - s/v_S \mid * v_L * v_S$
$230\ s * v_L * v_S = (v_S - v_L)*s$
$230\ s * v_L * v_S / (v_S - v_L) = s$
$230\ s * (159 / 3{,}6)\ m/s * 225\ km/h / (225\ km/h - 159\ km/h)$
=> s = ca. 34.500 *m*, das liegt am Anfang des 7. Blocks.

## Beginn der Gegengleis-Region

Nun suchen wir den Block, ab dem der schnelle Zug durch den langsamen behindert würde. Vermutlich ist es der 6. Block, den der schnelle Zug bereits im Gegengleis befahren ist. Wir fangen also mit der Ausfahrzeit des langsamen Zugs aus Block 6 an (Ankunftszeit der Zugspitze bei Block 7 + Fahrzeit des 1.000 *m* langen Zugs + Signalverzögerung). Anschließend vergleichen wir damit die Ankunftszeit der Zugspitze des schnellen Zugs bei Block 6

$t_{L\ aus\ 6} = e_L + (6 * 5.500\ m + 1000\ m) * [1/v_L] + l_{wieder} \mid e_L = 1.506\ s;\ [1/v_L] = 370\ s/m/16/1024;\ l_{wieder} = 10\ s$
$t_{L\ aus\ 6} = 1.506\ s + 34.000\ m * 370\ s/m/16/1024 + 10\ s = 1.516\ s + 12.580.000\ s/16/1024 = 2.283{,}82\ s$
$t_{S\ in\ 6} = e_S + (5 * 5.500\ m) * [1/v_S] \mid e_S = 1.736\ s;\ [1/v_S] = 262\ s/m/16/1024$
$t_{S\ in\ 6} = 1.736\ s + 27.500\ m * 262\ s/m/16/1024 = 1.736\ s + 7.205.000\ s/16/1024 = 2.175{,}76\ s$

Wie wir sehen, wird auch schon der 6. Block zum Überholen benötigt. Also versuchen wir es mit dem 5. Block:

$t_{L\ aus\ 5} = 1.516\ s + 28.500\ m * 370\ s/m/16/1024 = [...] = ca.\ 2.160\ s$

Das heißt, L fährt aus Block 5 aus, während S fast schon in Block 6 einfährt -> die Situation ist so knapp, dass wir noch einen Block zurückgehen:

$t_{L\ aus\ 4} = 1.516\ s + 23.000\ m * 370\ s/m/16/1024 = [...] = 2.035{,}41\ s$
$t_{S\ in\ 4} = 1.736\ s + 16.500\ m * 262\ s/m/16/1024 = [...] = 1.999{,}85\ s$

Das hat wieder nicht gereicht -> noch ein Block

$t_{L\ aus\ 3} = 1.516\ s + 17.500\ m * 370\ s/m/16/1024 = [...] = 1911{,}20\ s$
$t_{S\ in\ 3} = 1.736\ s + 11.000\ m * 262\ s/m/16/1024 = [...] = 1911{,}90\ s$

An dieser Stelle treffen die Rundungsregeln zu: es wird nämlich angenommen, dass das Signal nur zu jeder vollen Sekunde gestellt werden kann. (Diese Regel ergibt sich aus dem Ansatz, dass in dem kompletten Projekt die Sicherungstechnik nur mit ganzen Sekunden berechnet wird.)

Ein ausfahrender Zug belegt den Block also noch für die volle Sekunde; ein einfahrender Zug benötigt den Block bereits zu der vollen Sekunde. Daraus ergibt sich: $t_{L\ aus\ 3\ gerundet} = 1912\ s$ und $t_{S\ in\ 3\ gerundet} = 1911\ s$.

Es fährt L also später aus als S einfahren würde, damit muss S bereits im 3. Block im Gegengleis fahren, oder S ist von 1 *s* Fahrzeitverlust betroffen. Die Berechnung von Block 2 ist nicht nötig, denn – "Das sieht man ja" – im 2. Block wird L deutlich vor der Ankunft von S ausfahren.

Anmerkung: Die Zeitdifferenz, die bei jedem Block, den wir zurückgehen, gewonnen wird, bleibt konstant, da die Blockgröße konstant ist. D. h. der Algorithmus müsste hier gar nichts ausprobieren, sondern könnte direkt den richtigen Block treffen. Der Referenzalgorithmus wurde allerdings zu einer Zeit erstellt, als noch

unterschiedliche Blocklängen vorgesehen waren und führt deswegen ein Verfahren durch, das wie das oben gezeigte Verfahren funktioniert.

## Ende der Gegengleis-Region

Das Ende der Überholstrecke wird ähnlich bestimmt. Hier spielt die Länge von S und nicht mehr von L eine Rolle, ebenso wird die Wiederbelegdauer nach S gestellt.

Aus der Simulation kann ich vorwegnehmen, dass S noch bis Block 10 im Gegengleis fährt und erst fast am Ende der Strecke ins Hauptgleis zurückkehren kann. Deswegen nur noch kurz die Berechnung des 10. Blocks:

$t_{S\ aus\ 10} = e_S + (10 * 5.500\ m + 300\ m) * [1/v_S] + l_{wieder}$
$t_{S\ aus\ 10} = 1.736\ s + 55.300\ m * 262\ s/m/16/1024 + 10\ s = [...] = 2.630{,}31\ s$

[Kurze Anmerkung dazu: gerundet und ohne $l_{wieder}$ ist das 2.621 *s*; wenn die Wartesekunde (siehe auch nächsten Abschnitt) aktiviert wird, sind es 2.622 *s*.]

$t_{L\ in\ 10} = e_L + (9 * 5.500\ m) * [1/v_L] = 2623{,}86\ s$

L fährt also bereits relativ knapp "vor" der Ausfahrt von S ein, und es ist korrekt, wenn man sich "intuitiv" herleitet, dass es im 11. Block ohne Überschneidung klappt. (mit "vor" ist "vor der um die Wiederbelegdauer verlängerten" gemeint.)

## Der Gegenzug

Nun berechnen wir, wie der Gegenzug in die Situation passt. Er verlässt den Block 4 zu folgendem Zeitpunkt:

$t_{G\ aus\ 4} = e_G + ((11-4+1) * 5.500\ m + 80\ m) * [1/v_G] + l_{wieder}$
$t_{G\ aus\ 4} = 994\ s + 44.080\ m * 370\ s/m/16/1024 + 10\ s = 1.999{,}46\ s.$

Mit der Rundungsregel heißt das: G fährt zu Sekunde 2000 aus dem Block 4 aus.

S würde zu Sekunde 1999 (genauer: 1999,85) in den Block 4 einfahren, wenn S nicht durch L gehindert würde. Durch die Behinderung in Block 3 kommt S aber etwas später bei Block 4 an:

$t_{S\ in\ 4\ (neu)} = t_{L\ aus\ 3\ (gerundet)} + 5.500\ m * [1/v_S] = 1912\ s + 5.500\ m * 262\ s/m/16/1024 = 1999{,}95\ s.$

Wie wir sehen, würde S trotz leichter Verspätung schon eine Fahrtstellung zu Sekunde 1999 brauchen. Durch G fährt S allerdings erst zu Sekunde 2000 ein, das verursacht also (aufgerundet) eine Sekunde Verspätung und dadurch Kosten von 1 *WE*.

Es gibt auf der Strecke mit der Zug-Konstellation keine Alternative, den Überholvorgang mit weniger Kosten als 1 *WE* stattfinden zu lassen. Wenn S so früh wie möglich ins Gegengleis fährt, und L langsamer fährt, dann ist es zwar möglich, aber eben teurer.

# Ergebnis des Algorithmus

Der (hier: Enumerations-)Algorithmus berechnet den beschriebenen Fall und gibt dann, falls er korrekt rechnet, folgene Ausgabe aus:

*test-out/Ueberholung_x3_Blockgrenze_1d2.c-eenum*

```
#c-ref: Strecke: blockzahl 11, einzelblocklaenge 5500, l_wieder 10
#c-ref: Zug s: laenge 300, weight 1, r_geschw 262, start 1736
#c-ref: Zug l: laenge 1000, weight 1, r_geschw 370, start 1506
#c-ref: Zug g: laenge 80, weight 1, r_geschw 370, start 994
Zug langsam e  1506 b   1
Zug gegen   e   994 g  11
```

```
[...]
Ueberholung ja
#c-eenum: Kosten: 2376
#c-eenum inform: ^ 4 10v
#c-eenum debug: L tritt @31313144 (geschnitten 1911) aus Block 3 aus.
#c-eenum debug: S tritt @31324624 (geschnitten 1911) in Block 3 ein.
#c-eenum: Kosten S: 1584
#c-eenum debug: S tritt @44536064 (geschnitten 2718) aus Block 11 aus.
#c-eenum debug: L tritt @45024304 (geschnitten 2748) in Block 11 ein.
#c-eenum debug: G tritt @18320696 (geschnitten 1118) in Block 10 ein.
#c-eenum debug: G tritt @32759136 (geschnitten 1999) aus Block 4 aus.
#c-eenum: Gegenkosten (S wartet): 792
Zug schnell e  2000 g   4
#c-eenum debug: S tritt @42852624 (geschnitten 2615) in Block 11 ein.
Zug schnell e  2615 b  11
#c-eenum debug: L tritt @27243144 (geschnitten 1662) aus Block 1 aus.
#c-eenum debug: S tritt @28442624 (geschnitten 1736) in Block 1 ein.
Zug schnell e  1736 b   1
#c-eenum debug: L tritt @29278144 (geschnitten 1786) aus Block 2 aus.
#c-eenum debug: S tritt @29883624 (geschnitten 1823) in Block 2 ein.
#c-eenum debug: L tritt @31313144 (geschnitten 1911) aus Block 3 aus.
#c-eenum debug: S tritt @31324624 (geschnitten 1911) in Block 3 ein.
Zug schnell e  1912 b   3
```

Der Schaltkreis speichert seine Ausgaben in einem Speicherblock; diese werden von c-memcatch ausgelesen und sehen dann so aus:

*test-out/Ueberholung_x3_Blockgrenze_1d2.circuit*

```
# ACHTUNG: Automatisch generierte Datei von verify.sh!

[...]

# Read 256 integers from hex memory dump.
#c-algo/ASIC inform: Kosten 2376
Zug schnell e -1515870811 g   4
Zug schnell e -1515870811 b  11
```

Wie man sieht, hat der Algorithmus den ersten Gegengleisblock (4) und den letzten Gegengleisblock (11-1 = 10) korrekt herausgefunden. Außerdem wird getestet, ob es überhaupt eine Überholung gibt.

Dies beides reicht zur Verifikation und wird automatisiert vom Test-Skript ausgewertet. Die Ausgabe der genauen Zeiten und anderer Blöcke ist nicht implementiert, was die "hässliche" negative Zahl (aus dem Füller 0xA5A5A5A5) in der Ausgabe verursacht.

# Realitäts-Check

Kurzname:
java

Das Java-Programm soll prüfen, ob die Ausgabe des Algorithmus (c-ref/c-eenum/circuit) physisch funktionieren kann. Die grundsätzliche Aufgabe ist also dieselbe wie diejenige der auf realen Strecken installierten Zugsicherungstechnik: Wenn ein Block befahren werden soll, der aber nicht frei ist, ergibt sich ein Fehlerstatus. Diese Prüfung wird für jeden Block unabhängig durchgeführt und es wird nicht beachtet, welcher Zug die Strecke befährt und ob sich zwei Züge gegenüber stehen oder ob ein Rechenfehler aufgetreten ist.

Darüber hinaus wird der finale Abgleich mit dem Java-Program gemacht und es sind noch Hilfsarbeiten implementiert, z. B. wird die Ausgabe des Algorithmus (c-ref/c-eenum/circuit) sortiert und fortgeschrieben. Im einfachsten Fall besteht die Ausgabe nämlich nur aus der Angabe, wann die Züge jeweils in die Komplettstrecke „eintreten“ (einfahren), also wann sie in den ersten Block einfahren. Dann berechnet java auch für jeden weiteren Block der Strecke die Einfahrzeit/Belegungsdauer. Neben dem Sortieren und Fortschreiben vergleicht java die Kosten und den Überholvorgang mit den erwarteten Kosten und dem erwarteten Überholvorgang.

Folgende Klassen werden verwendet:

**Block**: Streckenblock mit Reservierungsmechanismus, d. h. nur eine Belegung gleichzeitig möglich.
**Freigabe**: Hilfsklasse für Block
**Zug**: Zug
**Fahrweg**: Sortierte Auflistung der vom Zug befahrenen Blöcke.
**Strecke**: Je eine sortierte Auflistung der Streckenblöcke pro Richtung
**Verify_Main**: Hauptklasse mit stdin-parser, Ausgabe und den Hauptberechnungen

Die Programmteile der Hauptklasse verlaufen in der folgenden Reihenfolge:

- read_stdin(): Daten von der Standardeingabe einlesen.
- route_eintragen(): Die gelesenen Blöcke sortieren und auf die nicht angegebenen Blöcke fortschreiben.
- strecken_fuellen(): Für alle Blöcke des Fahrwegs die Belegungszeiten ausrechnen und reservieren.
- verlauf_anzeigen(): Eine schöne Konsolen-grafische Darstellung der Ereignisse auf der Strecke.
- kosten_pruefen(): Alle Kosten zusammenrechnen und mit dem Testfall vergleichen.
- ueberholung_pruefen(): Testen, ob eine Überholung stattgefunden hat, und mit dem Testfall vergleichen.

Anmerkung: Das Programm ist nicht ganz vollständig. So wird es bereits als Überholvorgang gewertet, wenn ein Zug (S oder L) irgendwann im Gegengleis gefahren ist, es wird aber nicht getestet, ob dies tatsächlich mit einem Überholvorgang zusammenhängt. Die Funktion als Testprogramm wird davon aber nicht beeinträchtigt, da die Testcases auch eine Kostenkontrolle beinhalten und eine Schein-Überholung darüber erkannt wird.

Weitere Anmerkung: Außerdem wird nicht überprüft, ob ein Überholvorgang am Streckenende zuende ist.

Im Folgenden ist die gekürzte java-Programmausgabe für den Testfall (testcase) Ueberholung_x3_Blockgrenze_1f zu sehen. Am ökonomischsten ist es, wenn der schnelle Zug (S) Block 3 noch im Regelgleis befährt, auch wenn er dabei 1 Sekunde verliert, da der langsame Zug noch vor ihm ist. Um anschließend Block 4 im Gegengleis zu befahren, wird eine weitere Sekunde Wartezeit fällig, um den Gegenzug noch durchzulassen (die Signalrechenzeit l_wieder beträgt hier 10 Sekunden):

```
Reservierungsanfrage: Zug schnell, Start 1737, Ende 1830 -> In Ordnung.
Verify_Main: Wir müssen warten.
             (nach Block 3  @1913s statt 1912s)
Reservierungsanfrage: Zug schnell, Start 1824, Ende 1918 -> In Ordnung.
```

```
Verify_Main: Wir müssen warten.
             (nach Block 4  @2001s statt 2000s)
Reservierungsanfrage: Zug schnell, Start 1913, Ende 2006 -> In Ordnung.
Reservierungsanfrage: Zug schnell, Start 2001, Ende 2094 -> In Ordnung.
Reservierungsanfrage: Zug schnell, Start 2088, Ende 2182 -> In Ordnung.
Reservierungsanfrage: Zug schnell, Start 2176, Ende 2270 -> In Ordnung.
Reservierungsanfrage: Zug schnell, Start 2264, Ende 2358 -> In Ordnung.
Reservierungsanfrage: Zug schnell, Start 2352, Ende 2446 -> In Ordnung.
Reservierungsanfrage: Zug schnell, Start 2440, Ende 2534 -> In Ordnung.
Reservierungsanfrage: Zug schnell, Start 2528, Ende 2622 -> In Ordnung.
Reservierungsanfrage: Zug langsam, Start 1507, Ende 1654 -> In Ordnung.
[...]
Reservierungsanfrage: Zug langsam, Start 2624, Ende 2772 -> In Ordnung.
Reservierungsanfrage: Zug gegen, Start 1119, Ende 1246 -> In Ordnung.
[...]
Reservierungsanfrage: Zug gegen, Start 2236, Ende 2363 -> In Ordnung.


 ===  Freigaben-Ereignisse  ===

  < Gegenrichtung
  > Hauptrichtung

  Zeit RI 01 02 03 04 05 06 07 08 09 10

  1119  <   :  :  :  :  :  :  :  :  :+g:
        >   :  :  :  :  :  :  :  :  :  :
          --:--:--:--:--:--:--:--:--:--:
  1243  <   :  :  :  :  :  :  :  :+g:  :
        >   :  :  :  :  :  :  :  :  :  :
          --:--:--:--:--:--:--:--:--:--:
  1246  <   :  :  :  :  :  :  :  :  :-g:
        >   :  :  :  :  :  :  :  :  :  :
          --:--:--:--:--:--:--:--:--:--:
[...]
          --:--:--:--:--:--:--:--:--:--:
  1991  <   :  :  :-g:  :  :  :  :  :  :
        >   :  :  :  :  :  :  :  :  :  :
          --:--:--:--:--:--:--:--:--:--:
  2001  <   :  :  :+s:  :  :  :  :  :  :
        >   :  :  :  :  :  :  :  :  :  :
          --:--:--:--:--:--:--:--:--:--:
  2003  <   :  :  :  :  :  :  :  :  :  :
        >   :  :  :  :+l:  :  :  :  :  :
          --:--:--:--:--:--:--:--:--:--:
  2006  <   :  :  :  :  :  :  :  :  :  :
        >   :  :-s:  :  :  :  :  :  :  :
          --:--:--:--:--:--:--:--:--:--:
  2027  <   :  :  :  :  :  :  :  :  :  :
        >   :  :  :-l:  :  :  :  :  :  :
          --:--:--:--:--:--:--:--:--:--:
  2088  <   :  :  :  :+s:  :  :  :  :  :
        >   :  :  :  :  :  :  :  :  :  :
          --:--:--:--:--:--:--:--:--:--:
  2094  <   :  :  :-s:  :  :  :  :  :  :
        >   :  :  :  :  :  :  :  :  :  :
          --:--:--:--:--:--:--:--:--:--:
[...]

Verify_Main: Kosten erwartet/real: 1/1
Verify_Main: Kostenkontrolle: in Ordnung.
Verify_Main: Überholung vorgesehen/errechnet/real: ja/ja/ja
Verify_Main: Überholkontrolle: in Ordnung.
Verify_Main.java: Alles ist in Ordnung.
```

*Der zugrundeliegenden Testfall enthält den 300 Meter langen schnellen Zug (S), der mit r = 262 (v = 225 km/h) unterwegs ist.*

*Nachdem S in einen Block eingetreten ist (z. B. Block 5 bei 2088 s), braucht er 300 m / (159 km/h) = 4,8 Sekunden, bis auch das Ende aus dem alten Block (Block 4 bei 2094 s) herausgefahren ist. Die hier zu sehende Differenz (2094s-2088s = 6s) erklärt sich aus der Darstellung ("Austritt bei 2094 s" heißt, dass bereits vor der Sekunde 2094,00 der komplette Zug aus dem Block ausgefahren sein muss) und liegt an der sekundenweisen Reswervierung, d. h. der neue Block muss bereits ab der ganzen Sekunde 2088,00 reserviert sein, damit der Zug, der erst bei dem Sekundenbruchteil 2088,81 eintrifft, sofort einfahren darf. 4,8 Sekunden später, bei Sekunde 2093,61, fährt S aus Block 4 aus.*

# Skript-Anschluss an die IDE

Kurzname: icpro-connector/

Ein kleiner Teil der Projektarbeit, aber ein großer Arbeitszeitaufwand war die Verbindung der eigenen Skript-Umgebung mit der vorgegebenen Entwicklungsumgebung. Die zwei ursprünglichen Ziele dahinter waren, erstens nur die Dateien zu speichern, die tatsächlich für das Projekt relevant sind und eine eigene Arbeitsleistung bedeuten und zweitens den Schaltungssimulator für jeden Testfall einzeln automatisiert laufen zu lassen. Leider hat sich nicht viel, aber doch ein bisschen etwas bei der Umstellung auf den neuen Server verändert, weswegen der Anschlussaufwand eineinhalb mal getrieben werden musste. (Wobei ich mit den Linux-Prozess-Hilfsmitteln viel besser arbeiten konnte als mit den Sun-Hilfsmitteln, auch weil ich letztere nicht so gut kannte.)

## Projektdateien sichern und bearbeiten

Als ich am Anfang des Projekts stand, konnte ich "ICPRO" und "Cadence" nicht zuordnen und war unsicher, was in dem Verzeichnis $HOME/ICPRO/ passiert. Außerdem wollte ich mein Projekt regelmäßig in einem versionierten Backup (-> "projektsicherung.sh") sichern und dazu genau die Dateien, die ich erstellt habe, zusammensammeln. So bin ich auf das folgende Konzept gekommen:

Es wird das Projekt "ice_auto" angelegt, das (jedes Mal) für den Netlisting-Vorgang frisch erzeugt wird und bei dem ich als Nutzer keine Einstellungen ändern kann. (So kann ich sicher sein, dass nicht irgendwo eine versteckte Einstellung ist, die ich mir merken müsste, wenn ich neu anfangen würde.) Außerdem will ich die Schaltkreis-Zeichnungen bearbeiten; dazu wird temporär das Projekt "edit_tmp" angelegt und darauf die grafische Cadence-Oberfläche aufgerufen. Sobald diese Oberfläche beendet wird oder sogar zwischendurch, wenn ich per Kommandozeile das Netlisting anstoße, wird der Stand aus dem "edit_tmp"-Verzeichnis in das versionierte Projektverzeichnis übernommen.

## Netlisting-Vorgang anstoßen

Für den Netlisting-Vorgang habe ich ein übergreifendes Makefile erstellt, das mit Hilfs-Skripten den Status der Dateien prüft, die richtigen Dateien in das ice_auto-Projekt kopiert und nur dann das Netlisting-Skript startet, wenn sich an der Bearbeitung etwas geändert hat. Das Netlisting-Skript startet auf einem vorgespiegelten Monitor (Xvfb) die grafische Oberfläche lässt dann die Verilog-Dateien der Reihe nach öffnen und klickt darin auf "Check&Save". Anschließend ruft es den Netlister-Dialog auf, trägt darin die richtigen Pfade ein und "klickt" auf "Netlist" (es gibt für all diese Aktionen Skill-Kommandos, d. h. die Maus muss nicht bewegt werden.)

Die Vorgehensweise mit dem vorgespiegelten Monitor ist entstanden, weil die SSH-Verbindung brüchig war und ich trotzdem noch von zu hause aus die Verilog-Dateien programmieren wollte. Sie hat sich noch als weiterer Vorteil herausgestellt - während des andauernden Netlistvorgangs (der Server hat eine langsame Datei-Übertragung, was sich auf die Sicherungsvorgänge und vermutlich auch auf das Netlisting auswirkt) kann man als Nutzer andere Sachen machen, ohne Sorge haben zu müssen, im falschen Moment in einem aufpoppenden Fenster das falsche zu klicken, was relativ wahrscheinlich ist, da Cadence den Mauszeiger eigenmächtig auf irgendwelche Schaltflächen versetzt (und danach wieder zurückversetzt - auch das kann irritieren).

Zum Abschluss des Netlistvorgangs wird das Resultat mitsamt ein paar Logfiles in das unabhänige Projektverzeichnis kopiert und die Logfiles werden auf grobe Fehler hin durchsucht. (Es gab weiterhin viele Fehlerarten, die man als Benutzer manuell anschauen muss.)

# Simulator aufrufen

Der Simulator kann auch aus der Kommandozeile aufgerufen werden. Zur Vorbereitung muss die Datei für den Input-RAM ins richtige Verzeichnis gespeichert werden und der Pfad zum Lizenzserver als Umgebungsvariable gespeichert werden. Dann wird das Programm aus dem richtigen Pfad (PATH) und mit den richtigen Optionen aufgerufen. Das konnte einfach mittels der Prozessinformationen eines manuell gestarteten Simulationsvorgangs hergeleitet werden und war deutlich einfacher als die Automatisierung des Netlistings.

# Entwurf und Implementierung des Schaltkreises

Kurznamen:
circuit oder ASIC-source

## Grundsätzliche Vorgehensweise

Der Schaltkreis ist in den folgenden fünf Schritten entstanden, nachdem ich den Algorithmus auf dem Papier geplant und als C-Algorithmen für die Schaltkreisentwicklung entworfen und angepasst habe.

### Reset-Vermeidung

Anstatt für alle Elemente ein Reset zu implementieren, setze ich darauf, dass nach dem Start der Schaltung alles "durchgespült" wird. Dazu muss nur an wenigen Stellen das Reset wirken (insbesondere an den Außenschnittstellen und in der State-Machine, wo dann der Initial-Zustand aufgerufen wird und, vermittelt durch die Logikeinheit, der Datenpfad korrekt anfängt zu arbeiten.)

### Schritt 1: Planung auf dem Papier, Zuweisung von Zuständen

Vorliegend war der funktionierende C-Enumerationsalgorithmus; diesen habe ich ohne Block-Ausgabelogik (da viele Verzweigungen und States nötig wären) Schritt für Schritt auf Papier aufgeschrieben und jeden Schritt mit einem Punkt versehen. Die so entstandene Doppelseite ist als Scan im Projektverzeichnis zu finden.

Wie schon weiter oben in diesem Dokument deutlich geworden ist, war die Minimierung der Zustands-Anzahl das Hauptziel. Deswegen habe ich mehrere Punkte/Schritte kombiniert und für jeden (kombinierten oder noch freien) Punkt einen State-Namen zugewiesen. Dies führt zu einer Vergrößerung des Datenpfads, um mehrere Rechnungen auf einmal ausführen zu können.

Eine weitere Maßnahme zur Minimierung der Zustands-Anzahl war die Zusammenlegung der Unterprogramme eintrittszeit und verlassenszeit zu verleintritt: Immer, wenn das entsprechende Unterprogramm gebraucht wird, wird das so entstandene Kombi-Unterprogramm aufgerufen und ggf. das nicht gebrauchte Ergebnis verworfen. (Dieses Verwerfen passiert bei zwei von den vier Aufrufen.)

### Schritt 2: Vor-Schaltkreis

Zum Austesten meiner Verifikationsroutinen und um ein Gespür für das Projekt zu bekommen, hatte ich einen Vorschaltkreis mit Datenpfad, FSM und Logik angelegt. Zudem habe ich das System (mit Speicher) und die System-Testbench angelegt. Anfangs hat die Schaltung nur intern States gewechselt, später aus dem Speicher (dann erzeugt von c-memgen) ein Bit gelesen und, falls es (nicht) gesetzt war, entsprechend ein anderes Bit modifiziert (dann von c-memcatch eingelesen).

### Schritt 3: Variablen-/Registerplanung (Tabellenkalkulation)

In einer OpenDocument-Tabelle habe ich versucht, die Variablen, Konstanten und Halbkonstanten mit deren zu erwartenden Wertebereichen zu fassen. In einer späteren Tabelle habe ich, aufgetragen über den Zustands-Ablauf, die Registeraktionen und Speicherzugriffe definiert.

### Schritt 4: Implementierung in Verilog/Verilog/Zeichnung mit Busplanung

Dieser Schritt ging ohne große technische Probleme, da ich auf den Vor-Schaltkreis aufsetzen konnte. Die

neuen Teile habe ich zunächst neben die alten gesetzt, sobald die neuen aber Speicher gelesen haben, den alten komplett gelöscht.

Die Implementierung von FSM/Logik in Verilog und Datenpfad als Zeichnung ging parallel zur Busplanung in einer Tabellenkalkulation. Außerdem wurden, teils zur Korrektur von Fehlern, geringfügig die Busplanung und die Code-Schritte modifiziert. Es war schon von vorneherein geplant, die Busplanung mit der Implementierung zu kombinieren, um nicht schon auf dem Papier eine Busanzahl festlegen zu müssen und bestimmen zu müssen, welcher Block mit welchen Bussen kommuniziert. Durch die spontane Methode waren Umarrangierungen möglich, die die Bustreiber-Anzahl verringert haben.

Zuerst habe ich das VE-Unterprogramm beplant und dabei keine ausführlichen Kommentare in die Tabelle geschrieben. D. h. man muss die Steuerlogik-Beschreibung neben der Tabelle als Fenster geöffnet haben, um nachzuvollziehen, was auf welchem Bus passiert. Außerdem wird das Unterprogramm von verschiedenen Haupt-Zuständen aufgerufen, die meist mitten in einer Rechnung waren, weswegen möglichst viele Register von VE unberührt bleiben sollten. Insgesamt war VE also eine große Knobelaufgabe.

*Aussschnitt aus Ggl_FSM.v:*

```
always @ (posedge clk)
    if (rst_n == 0) out_internal <= S_RESET;
    else begin   case(out_internal)
        [...]
        S_HOLDRES:  out_internal <= S_LOOP_I;
        S_LOOP_I:
            if (flags_in[LINE_ALU_S] == 1'b1)
                out_internal <= S_2ND_LVL;
            else
                out_internal <= S_LOOP_O;
        S_LOOP_O:
            if (flags_in[LINE_ALU_S] == 1'b1)
                out_internal <= S_INC_l;
            else
                out_internal <= SA_KOSTEN;
        [...]
        endcase
    end
```

Beim Hauptprogramm ist die Busplanung von Anfang an besser dokumentiert gewesen, hier konnte man die Rechnungen in Grundzügen rein aus der Tabelle nachvollziehen.

*Ausschnitt aus Ggl_Logik.v:*

```
S_SAVE1:
    retW = CTRL_BASE // x+l -> R4 (blkn)
                     | CTRL_X_ADD
                     | CTRL_L_ADD
                     |       ALU_A_MUX_XALU
                     | CTRL_ALU_ADD
                     |       ALU_B_MUX_LALU
                     | CTRL_ALU_TO_BUS4
                     | CTRL_R4_LD // BUS4->R4
                     ;
```

Das einzige ernsthafte Problem, das sich ergab, war bei S_LD2/S_MUL2, da hier zu viel auf einmal passieren sollte bzw. ich einen zusätzlichen Bus hätte einführen müssen, was ich nicht nur für diese eine Stelle tun wollte. Deswegen musste ich die Zustände noch einmal erweitern, wodurch der Zustand S_MUL2A eingeführt wurde. Später war mir aufgefallen, dass ohne die Behandlung des Konfliktfalls nahezu kein Testcase funktioneren kann, weswegen ich dazu noch vier States (S_KONFL_*) hinzufügen musste.

Es haben sich übrigens viele Rechnungen in den States gändert, nicht aber deren Zustands-Namen. So wird in S_DEC_X nichts dekrementiert, stattdessen ist ein Inkrementierungsvorgang nach hinten geschoben. Der Name wird trotzdem behalten, da an mehreren Stellen in der (Papier-)Projektdokumentation und auf die Namen Bezug genommen wird und erst alle Vorkommen gefunden werden müssten.

In dem Implementierungsprozess habe ich einige Datenpfad-Elemente eingeführt, siehe dazu ein Unterkapitel unten.

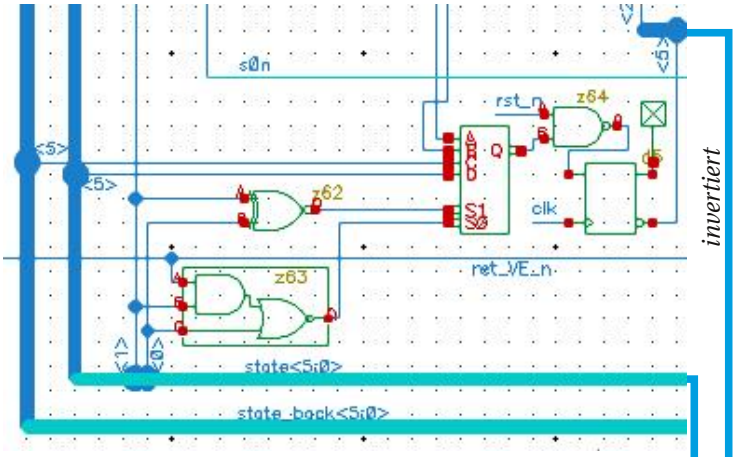
## Schritt 5: Manuelle Synthese FSM von Verilog in CORELIB-Zellen-Zeichnung

Wie in der Projektaufgabe gefordert, sollte eine Synthese durchgeführt werden: entweder automatisch oder manuell. In Anbetracht der langen Zeit, die die Integration des Netlisting-Vorgangs in die Shell-Skripte gekostet hatte, sah ich es als verlässlicher an, die FSM selbst mit den CORELIB-Zellen zu zeichnen.

Dies ist zwar an sich eine aufwändige Aufgabe gewesen (es geht um ca. 200 Gatter), aber das Zustands-Modi-System (Siehe Intelligente Zustandsnummerierung) hat durch die Vorstrukturierung eine gute Orientierung und die Möglichkeit einer stufenweisen Transition ergeben: zuerst habe ich die als Kopie FSM_interim eingesetzt und die Steuerung auf je eine Look-Up-Table (LUT) für die Modi 00,01,10,11 als Logikgleichung in Verilog implementiert, dann eine FSM-Kopie "FSM_interim" mit neuer Steuerung versehen, sodass ich die LUT00-LUT11 damit testen konnte. Als das funktionierte, habe ich die LUT mit Gattern gezeichnet. Danach habe ich sie Stück für Stück in eine neue große finale FSM-Zeichnung aus Gattern und Flip-Flops und Multiplexern eingepflegt.

Es hat sich die in dem Prinzipbild gezeigte Gatterschaltung ergeben. (Die zugrundeliegenden Übergangsvarianten a/b/c/d/e sind in der Pfeil-Abbildung darüber dargestellt.)

Hauptprogramm —a→ SV_LD1 (Unterprogramm)
SV_RET (Unterprogramm) —bedingt b→ Hauptprogramm
beliebiger Zustand —c→ S_RESET (Hauptprogramm)
Hauptprogramm (Modus 00/01/10) —d→ Hauptprogramm
Unterprogramm (Modus 00/01/10) —d→ Unterprogramm
Hauptprogramm (Modus 11) —e→ Hauptprogramm
Unterprogramm (Modus 11) —e→ Unterprogramm

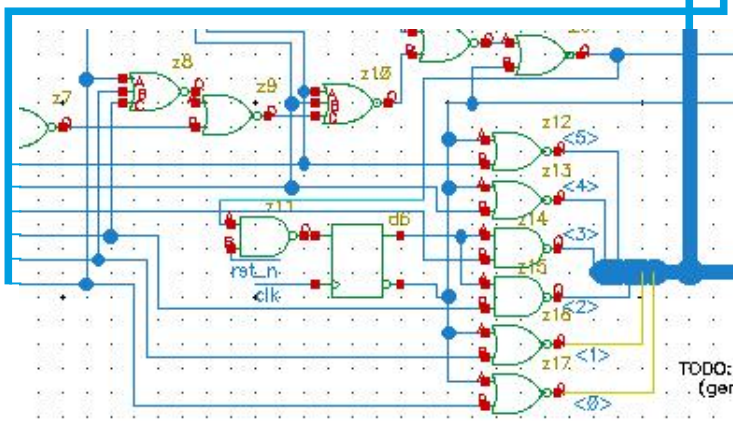



*Ganz oben: Zustandsübergangsvarianten*

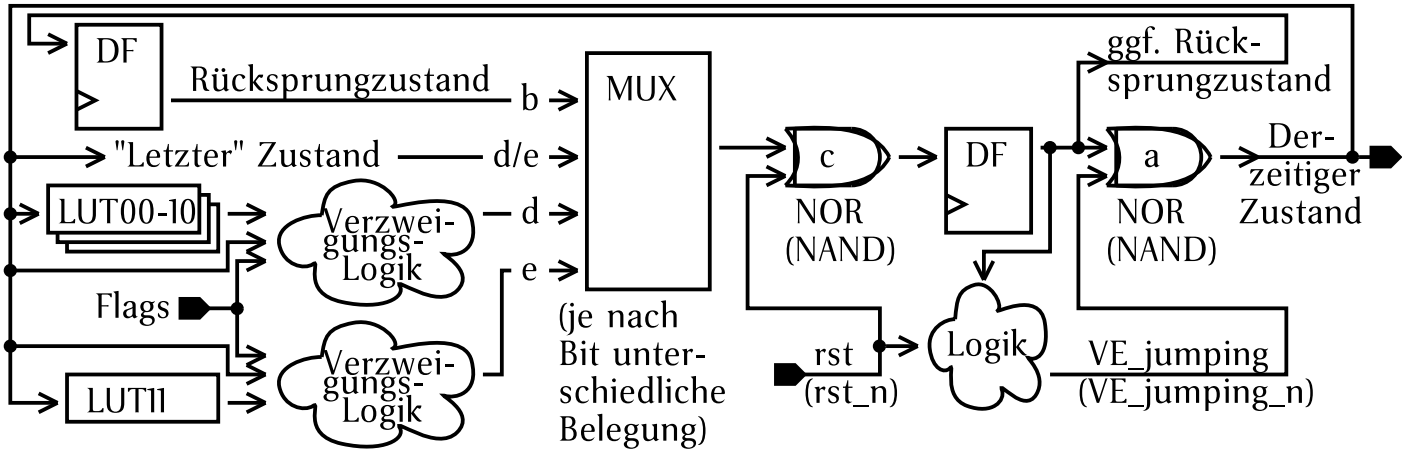
*Oben: Ausschnitt aus der CORELIB-basierten Zeichnung der FSM mit dem Muxer für Bit 5: Variante c (Reset) wird von dem oberen NAND behandelt; der Muxer kann Variante b (Eingang C), Variante d (Eingang A oder D) und Variante e (Eingang B) ausführen; der Eingang wird abhängig von ret_VE und Modus (state<1:0>) ausgewählt.*

*Rechts: Ausschnitt aus der CORELIB-basierten Zeichnung der FSM. Hier ist zentral das VE-jumping-FlipFlop zu sehen, das die sechs NOR-/NAND-Gatter so ansteurt, dass sie entweder SV_LD1 (001100) ausgeben (Unterprogrammaufruf = Zustandsübergangs-Variante a) oder den von links in sechs Einzelleitungen anliegenden errechneten Folgezustand weitergeben (Varianten b,c,d,e).*

*Hinweis: Der Wert VE_jumping wird von dem invertierten Ausgang des FlipFlops getrieben.*




*Unten: Prinzipbild Zustandsübergang: Durch das Reset-Signal wird S_RESET angewählt, das in den NOR/NAND codiert ist. Durch VE_jumping wird entsprechend SV_LD1 angewählt.*



# Datenpfadblöcke und Hilfsblöcke

Da ich Zustände sparen musste, sind eher lange Clock-Zyklen geplant als viele Clock-Zyklen. Die in der Entwicklungsumgebung als ICE_DatapathElements angebotenen Blöcke hatten allerdings eine sehr Clock-zentrierte Architektur, d. h. bei ALU/MUL war die Ausgabe geclockt. Da allerdings die Register ebenfalls geclockt sind (und im Algorithmus häufig Rechenergebnisse in die Register geladen werden), bedeutet jede einzelne Rechnung zwei Zyklen, d. h. zwei Zustände. Deswegen habe ich die Rechenblöcke umgeschrieben; um Platz auf der Zeichnung zu sparen, habe ich außerdem die Register und Muxer mit anderen Symbolen versehen. Speziell für den Algorithmus waren noch kleinere Zusatzblöcke nötig.

## ALU/MUL/CA

Wie bereits beschrieben, habe ich die ALU geringfügig modifiziert, sodass sie keinen Clock-Eingang mehr hatte. In wenigen Fällen werden Daten aus einer Recheneinheit direkt für die selbe Recheneinheit weiterverwendet, sodass das Register nur selten etwas genutzt hätte.

Außerdem gibt es mehrere Stellen im Algorithmus, bei denen Überläufe entstehen können; vornehmlich wenn Kosten verglichen werden. Eigentlich müssten diese Überläufe durch ein "continue" (d. h. Sprung ans Schleifenende der Enumeration) kompensiert werden. Allerdings hätte das zusätzliche Verzweigungen bedeutet, weswegen ich die Variante mit dem Cut-Above (CA) vorgezogen habe: für jeden Wert, der den Wertebereich der ALU (also 32 Bit) überschreitet, wird der überschrittene äußerste Wert (also 0xFFFFFFFF) eingesetzt. Obwohl für den Enumerationsalgorithmus nicht nötig, habe ich diese CA-Eigenschaft ebenso für Unterschreitungen (dann 0x00000000) und vorzeichenbehaftete Rechnungen (0x80000000 und 0x7FFFFFFF) und für vorzeichenlose Überschreitungen bei MUL (0xFFFFFFFF) implementiert.

## Register

Spät hinzugekommen sind noch die Register mit dem geclockten Reset. Das Reset wird nur bei wenigen Registern gebraucht (um die Merkeinrichtung und die Schleifenvariablen zurückzusetzen). Erst bei der Realisierung der manuell synthetitisierten (gezeichneten) Gatterschaltung der FSM, als zufällig die Reset-Leitungen der Merkregister aktiviert wurden, merkte ich das Problem der kurz ausrutschenden States und habe die Register mit einem geclockten Reset versehen. Da auf die Schnelle die Dokumentation der Register aus CORELIB nicht auffindbar war, habe ich deren Resetfunktion gar nicht genutzt, sondern selbst in einem Wrapper implementiert. So sind REG32EcR und REG8EcR entstanden.

Es gibt Stellen im Algorithmus, an denen 14-/16-/18-Bit-Werte ausreichen. Es erwies sich als möglich, diese Werte gemeinsam in 32-Bit-Register zu speichern, die sowohl geteilt als auch komplett betrieben werden können. So sind REG1616 und REG1814 entstanden.

## Shifter/Runder/Bitbreitenwandler

An ca. zwei Stellen im Enumerationsalgorithmus muss eine Ergebniszahl auf ganze Zahlen gerundet werden. Da durch die reziproke Geschwindigkeit ein Faktor von 2^14 (16*1024) an die ganzen Zahlen multipliziert wurde (siehe Algorithmus-Kapitel), ist ein Aufrunder implementiert, der auf die 14. Stelle vor dem Komma rundet. Desweiteren sind noch Shifter (implizit durch die Register REG1814 und explizit als SHIFT_14) und Bitbreitenwandler (SIGNEXT_8_32) implementiert.

## Muxer

Die von mir neu eingeführten Muxer sind nur Wrapper der Muxer aus der CORELIB mit Arbeitserleichterungen: die Standard-Varianten fassen die Auswahlsignale (S1, S0) zu einem Bus zusammen. Die invertierende Variante (IIMUX) "invertiert" die Auswahlsignale, wobei an letzteren nichts verändert wird, sondern die Dateneingänge anders angeschlossen werden.

# Optimierungstools/Logik-Vereinfacher

Die Logik-Vereinacher, wie sie auf der Internetseite angegeben waren, bieten nur ein trügerisch "optimales" Ergebnis, was im folgenden genauer ausgeführt sei:

**1. Wichtung der Aspekte**

Ein Inverter und ein NAND/NOR2/NOR4/... sowie die zusätzlichen Bauteile OAI/AOI/XNR/XOR haben einen unterschiedlichen Flächenbedarf und unterschiedliche Energieverbräuche pro Logikwirkung. Je nach dem, worauf es ankommt, können unterschiedliche Schaltungen optimal sein. Dies wird aber von den angebotenen Vereinfachern nicht berücksichtigt.

**2. Vielfalt der Zellen**

In der realen Schaltung können mehr Zellen als NAND2 oder NOR2 platziert werden, z. B. OAI31 (Kombination aus OR und NAND). Dies wird in den Vereinfachern nicht abgebildet.

**3. Mehrfachnutzung**

Wenn in der Gesamtschaltung mehrere Gleichungen vereinfacht werden müssen, kann es Synergieeffekte in Form von identischen Gleichungsabschnitten geben. D. h. statt zwei Blöcken muss dann nur ein Block implementiert werden. Dies wird in den Vereinfachern nicht abgebildet. (Ebensowenig wie die Separat-Haltung von Gleichungsabschnitten, d. h. wenn ein Logikstrang in Zukunft ziemlich sicher modifiziert wird, die anderen Logikstränge aber nicht, dann wäre es kontraproduktiv, einen kleinen Effizienzgewinn durch die Zusammenlegung der Stränge zu erreichen.) Diese Mehrfachnutzung ist nicht von allen Vereinfachern darstellbar.

**4. Aufgabenumsortierung**

Es kann sich eine deutlich effizientere Schaltung ergeben, wenn die Aufgabe leicht abgeändert wird.

In meinem Fall stand ich bei den Muxern zu den State-Registern (je nach Betriebssituation/Modus werden pro Bit unterschiedliche Informationsquellen abgerufen) vor der selbstgestellten Aufgabe, die Input-Pins der Muxer zu belegen/steuern. Dies habe ich so gestaltet, dass die Mini-Steuerlogik möglichst identische/einfache Schaltungsanteile hat und habe die Eingänge der Muxer entsprechend umverdrahtet. Ein Optimierungstool kann, wenn überhaupt, schlechter als ein Mensch erahnen, welches die möglichen Drehpunkte bei der Aufgabe sind.

**5.Treiberstärken**

Wenn ein Schaltungsteil mehrfach genutzt wird, so muss dessen letztes Gatter eine größere Treiberstärke haben, um die angeschlossenen Gatter zu versorgen. Dies wird von einem Logik-Vereinfacher nicht berücksichtigt. (Anmerkung: ich habe das auch nicht berücksichtigt.)

Aus diesen Gründen kann man davon ausgehen, dass selbst eine schnelle manuelle Logikbildung aus Effizienzgesichtspunkten gar nicht so schlecht ist und gerne gegenüber einer automatischen Optimierung bevorzugt werden kann.

Es gibt aber auch positive Aspekte von Optimierungstools:

1. Eine Aufgabenumsortierung ist nicht immer möglich (z. B. liegt die Aufgabe bei der Bahn teils hinter organisationellen Grenzen, Fahrplankonventionen, usw...). Wird die Aufgabe von einem Tool gelöst, so fällt das niemandem auf, der versuchen könnte, das ändern zu wollen und daran viel Energie verliert.

2. Mit einem Optimierungstool lässt sich eine schneller verbindliche Planung realisieren: die Aufgabe wird

technisch formuliert und gelöst. Bei einer manuellen Lösung würde die Aufgabe ggf. von der Lösung her Änderungen unterliegen. (Übertragen auf den Schaltkreisentwurf: es ist ein üblicher und sinnvoller Ansatz, das Problem zu zerteilen und Schnittstellen zu definieren/standardisieren. Damit können unterschiedliche Bearbeiter/innen aus unterschiedlichen Kenntnissrichtungen gleichzeitig an einem Projekt arbeiten. Eine Aufgabenänderung würde eine Schnittstellenänderung bedeuten und gleich mehrere Bearbeiter/innen beeinflussen.)

3. Die Lösungsfindung geht insgesamt schneller.

Die anfangs geschriebene Warnung vor der Nutzung von Optimierungstools, hier im Zusammenhang mit den Logikgattern geäußert, trifft natürlich auch den Algorithmus für dieses Projekt in seinem Kern: dieser stellt nichts anderes dar als die Optimierung einer Eisenbahnsituation. Außerdem habe ich in dem Projekt doch ein Optimierungstool genutzt (für prolog-states).

Deswegen hier ein kurzer Abriss zu Optimierungstools:

Um ein solches anzuwenden, muss man die Aufgabe üblicherweise in eine technische/modellierte Aufgabe überführen (was i. d. R. leicht zu bewerkstelligen ist). Im Fall der FSM hieße das, dass der nächste Zustand jedes Bits als Logikgleichung, bedingt durch den aktuellen Zustand der Bits, dargestellt werden muss. Anschließend löst das Tool die Aufgabe und erzeugt damit eine kompakt-komplexe Lösung (mit sehr vielen Mehrfachnutzungen), im FSM-Fall eine Gatterbeschreibung der Bit-Logik. Dabei ist die menschliche Denkarbeit gering, aber die Computer-Rechenzeit hoch.

Wenn man ein Tool nicht anwendet, zerteilt man das Problem normalerweise in eine Zwischenebene (halb-komplexe Lösung/Aufgabe), und synthetisiert diese Aufgabe dann zur vollständigen Lösung. Bei dem Prozess macht man Schritte vor und zurück und modifiziert die Ausgangsaufgabe und/oder die halbe Lösung, um dadurch einen späteren Lösungsweg wählen zu können.

Tendentiell sind nach einer manuellen Lösung (insbesondere bei der Zwischenebene) noch Objekte/Stränge/Einheiten erkennbar, die an konkrete Gegenstücke der Realität erinnern. Deswegen lassen sich nachträgliche Änderungen, die an Objekte der Realität gebunden sind, manuell besser einpflegen, denn die Änderungen sind eher auf einen kleinen Pfad vom Anfang bis zum Ende beschränkt.

Bei der Anwendung des Tools wird die Lösung selbst bei kleinen Aufgabenänderungen komplett anders aussehen, da der Pfad sehr breit ist, d. h. sehr viele Elemente von der Änderung betroffen sind.

Ob man ein Optimierungstool nutzt, ist also eine Wette auf die Zukunft: wie wahrscheinlich ist es, dass sich an der Aufgabe später mal etwas ändern wird? Wird es eine bessere/andere Technik geben, weswegen die Lösung sowieso ersetzt werden wird (disruptiver Wandel)? Wird das Optimierungstool später noch verfügbar sein und falls nein, wird es eines geben, das eine mindestens so effiziente Lösung findet (sodass ein Lösungsaustausch in-place möglich ist, also der Träger der Lösung immer noch ausreicht, z. B. der erworbene Grund für die Bahnstrecke in 20 Jahren ausreicht)? Wenn nicht, dann ist der manuelle Weg besser, denn relativ sicher kann dann ein findiger Kopf noch einen Effizienzgewinn herausholen. Wird irgendwann ein Debugging nötig sein? Wenn ja, dann ist der manuelle Weg besser, da beim Debugging der menschliche Kopf möglichst alles im Blick behalten muss, und das geht mit realitätsnahen Strängen besser als kompakt-komplex.

# Intelligente Zustandsnummern

Kurzname: prolog-states

Im pdf-Dokument mit der Anleitung und Aufgabenstellung wird die Umsetzung des Zustandsautomaten (FSM – finite state machine) als Gatterrealisierung im Schaltplan erwartet. Diese Anforderung braucht bei mehr als 32 Zuständen nicht mehr erfüllt zu werden, falls man eine Synthese für die komplette Schaltung durchführt. Allerdings ist die Integration des Verilog-Simulators in die eigene Verifikation ziemlich aufwändig gewesen, weswegen ich den Aufwand, die Synthese-Tools in die Verifikationsskripte zu implementieren, gescheut habe. Deswegen musste ich eine Gatterschaltung implementieren, die von einem State (liegt an als 6-Bit-Wert) den nächsten State bestimmt (und als 6-Bit-Wert ausgibt). Generell ist dies linear gedacht, d. h. für jeden State gibt es einen Folge-State. Allerdings gibt es auch Verzweigungen, Unterprogrammsprünge und Anfang/Ende zu beachten.

Die Idee, Gray-Codes als Nummern zu verwenden, habe ich nach einer Internetrecherche verworfen. Es gibt nämlich keine einfache Möglichkeit, von einer Gray-Code-Nummer auf die nächste zu schließen.

Ein zusätzliches Problem der Gray-Codierung ist die Unmöglichkeit, eine Verzweigung (Anwahl eines alternativen Zustands bei einer spezifizierten Flag-Konstellation) einzurichten. D. h. im Normalfall wird beim Übergang 1 Bit des Zustands umgeschaltet, aber im Verzweigungsfall werden 6 Bit geschaltet.

Stattdessen habe ich ein eigenes Prinzip angewandt: von den 6 Bits werden nur 2 (bzw. 4) von der State Machine bestimmt, die anderen Bits bleiben entweder konstant oder haben einen definierten Zustand. Um welche der Bits es sich handelt, ist in der folgenden Tabelle abzulesen. Die X-Bits beschreiben den aktuellen Zustand, die Y-Bits den Folgezustand. Ein "*" bedeutet, dass diese Bits frei gesetzt werden können; in der Schaltung wer-den sie in Abhängigkeit von dem Prefix {Y5,Y4,Y3,Y2} bestimmt. Die Tabelle kann auch ganz anders aussehen, wenn jemand anders als ich sich eine Folgelogik ausdenkt oder andere Rahmenbedingungen herrschen.

| **X1** | **X0** | **Y5** | **Y4** | **Y3** | **Y2** | **Y1** | **Y0** |
|---|---|---|---|---|---|---|---|
| 0 | 0 | * | * | =X3 | =X2 | 0 (=X1) | 1 |
| 0 | 1 | =X5 | =X4 | * | =X2 | * | 0 |
| 1 | 0 | =X5 | =X4 | =X3 | * | 1 (=X1) | * |
| 1 | 1 | * | =X4 | =X3 | * | * | * |

Hier ist zu sehen, dass der Automat von einem 00-Modus automatisch in einen 01-Modus springt. Die anderen Modi sind freier, insbesondere der 11-Modus, aus dem jeder andere Modus erreichbar ist. Er eignet sich also sehr gut für den Verzweigungszustand bei Verzweigungen.

Eine der ursprünglichen Ideen war, aus dem 00-Modus immer mit gleichbleibendem Prefix in den 01-Modus zu springen; damit erwies sich aber die State-Nummer-Zuordnung als nicht gut lösbar.

**Beispiel**: S_CP2 hat die Nummer 010000. Der Folgezustand ist, je nach Flags, S_LD_WGT_L (100001), S_VE3 (110001) oder S_LOOP_I (000001). Wie man sieht, wird immer das Bit 0 gesetzt (0 -> 1), die Bits 3, 2 und 1 bleiben unverändert (000 -> 000) und die Bits 5 und 4 werden individuell gesetzt (01 -> 10/11/00).

Es ist durch menschliche Anstrengung nicht möglich, schnell eine gute Nummernzuordnung für dieses Problem zu finden, weswegen ich mir Prolog als nützliches Optimierungstool ausgewählt habe. Prolog ist dafür bekannt, dass man damit Logikspielchen wie z. B. Sudoku oder Lücken-Additionsrätsel lösen kann. Im Hintergrund wird jede mögliche Lösung durchprobiert (das kann auch als vollständige Enumeration bezeichnet werden). Allerdings kann man die Zeit beschleunigen, indem man das Problem so stellt, dass eine mögliche Lösung sehr früh beim Durchprobieren entdeckt wird; danach lässt sich das Programm abbrechen oder es sucht nach einer weiteren Lösung.

Damit Prolog die Zuordnung findet, habe ich zuerst die Folgebit-Tabelle (s. o.) in der Prolog-Sprache definiert. Danach musste ich alle Zustands-Beziehungen einprogrammieren, d. h. welcher Zustand in welchen Folgezuständen resultieren kann. Um eine Lösung wahrscheinlicher werden zu lassen, habe ich den Unterprogrammaufruf nicht in die Folgezustandslogik einbezogen; das Unterprogramm selbst folgt aber intern dieser Logik.

*Prolog-Code: num-def.pl: So habe ich den Zustandsübergang technisch formuliert:*

```
flw([X5, X4, X3, X2, X1, X0], [Y5, Y4, Y3, Y2, Y1, Y0]) :-
  X1 == 0 -> (X0 \== Y0, (X0 == 0 -> (Y1 == 0,
  X3 == Y3); (X5 == Y5, X4 == Y4)), X2 == Y2) ; (X3 == Y3,
  X4 == Y4, (X0 == 1 -> true ; (Y1 == 1, X5 == Y5))).
flw_lower([_, _, _, _, X1, X0], [_, _, _, _, Y1, Y0]) :-
  X1 == 0 -> (X0 \== Y0, (X0 == 0 -> Y1 == 0; true)) ;
  (X0 == 1 -> true ; Y1 == 1).
```

Die Rechenzeit zu reduzieren ist eine schwierige Aufgabe, war aber für dieses Problem nötig. Folgende Maßnahmen habe ich u. a. ergriffen. Möglicherweise hätte mir ein optimierender Prolog-Interpreter einige Arbeit erspart.

*Folgendermaßen sieht ein Abschnitt aus states-def.pl aus. Der Modus von S_LOOP_I wird als 01 vordefiniert. Mit include_once habe ich einen Test geschrieben, ob eine Nummer ein einziges Mal in einer Liste vorkommt:*

```
search_kCore([...], S_HOLDRES, S_LOOP_I, S_LOOP_O) :- true
[...]
  , state_lh_01(S_LOOP_I), flw_lower(S_HOLDRES, S_LOOP_I)
                         , flw_lower(S_CALC5, S_LOOP_I)
[...]
  , state_upper(S_LOOP_I), flw(S_HOLDRES, S_LOOP_I)
                         , flw(S_CALC5, S_LOOP_I)
[...]
  , LIST = [ [...], S_HOLDRES, S_LOOP_I, S_LOOP_O]
  , include_once(S_LOOP_I, LIST)
[...]
  .
```

1. Keine vollständige Festlegung aller Bits gleichzeitig mit Schluss-Logik- und Listen-Kontrolle, sondern stufenweises Vorgehen: In der ersten Stufe werden die Modi für nur ca. fünf Zustände festgelegt und mit der Folgebit-Tabelle (s. o.) abgeglichen. Dann werden die Prefix-Bits festgelegt und wieder mit der Folgebit-Tabelle abgeglichen. Dann gibt es einen internen Listenvergleich, d. h. es wird geprüft, dass keiner der fünf Zustand eine identische Nummer hat und anschließend, dass die fünf Zustände nicht mit irgendeinem anderen bereits festgelegten Zustand außerhalb der Kleingruppe übereinstimmen.

2. Vorziehen der schwierigen Gruppen: Die Initialisierungssequenz (S_RESET, S_INIT1, S_INIT2, S_INIT3, S_INIT_l) ist simpel (fünf Zustände, die ohne Verzweigung aufeinander folgen), während die Schleife (S_INC_l, S_2ND_LVL, S_LOOP_I, S_LOOP_O) schwierig ist, da mehrere Sprünge und Zielzustände sehr dicht aufeinanderfolgen. Es ist ineffizient, zuerst für die Initialisierungssequenz festzulegen und dann festzustellen, dass die Schleife unbedingt auf bestimmte Nummern angewiesen ist, die bereits von der Initialisierungssequenz belegt sind. Stattdessen lassen sich schneller alle Schleifen-Nummern-Konstellationen aufzählen, und anschließend ist schnell eine gültige Initialisierungsnummernsequenz gefunden.

3. Vermischen der Listenvergleiche mit der Folgebitlogik. Um zu verhindern, dass durch die Folgebitlogik ein Ergebnis wie A->B->A->B (also 000000 -> 000001 -> 000000 -> 000001) herauskommt, kann man schon sehr frühzeitig die dritte Nummer und die erste Nummer abgleichen; außerdem kann man irgendeine Nummer aus einer Gruppe während der Folgebitlogik herausgreifen und mit den bereits gefundenen Nummern der schon festgelegten Gruppen vergleichen. Die Wahrscheinlichkeit, dass die Übereinstimmungseigenschaft dieser einen Nummer für die Übereinstimmungseigenschaft der ganzen Gruppe spricht, ist relativ hoch.

4. Predefinition der Modi: Aus einem vorherigen fehlerhaften Lauf des Programms (aufgrund von Denkfehlern hatte ich Listenvergleiche vergessen und mehrere States hatten identische Nummern) habe ich eine bestimmte Modi-Konstellation extrahiert und das Programm so modifiziert, dass diese Konstellation immer zuerst festgelegt wird (und nur im nicht-passenden Fall werden andere Modi-Konstellationen festgelegt).

Die so gefundenen State-Nummern sind im `include states_auto.v und handschriftlich in den Scans abgelegt.

# Arbeitsumgebung, Projekt-Setup

Das System, auf dem das Projekt begonnen hat, war Oracle Solaris 10 mit einer älteren Cadence-Version. Sämtliche folgenden Angaben beziehen sich auf das neue System (Scientific Linux 6.8) oder auf meinen Computer.

## Beteiligte Sprachen

Teils sind englische Stücke in den Code-Kommentaren zu finden, aber die grundsätzliche Projektsprache ist deutsch.

Die an diesem Projekt beteiligten Computersprachen sind:

- Verilog: Hardwarebeschreibung der Testbench, des Systems, der Logik und je nach Variante auch der FSM
- Java: Simuliert das Streckenmodell und prüft damit die Algorithmusergebnisse.
- C: Referenz-Implementation des Algorithmus'
- Bash-/Sh-Skripte: diverse Anwendungen zur Projektsteuerung
- (awk, sed): Als Hilfen zur automatischen Textbearbeitung in den Skripten
- Skill: Steuerung des Cadence-Systems, z. B. um den Netlist-Vorgang anzustoßen.
- Makefile: Verwalten des Netlist-Vorgangs, hauptsächlich durch den Aufruf der richtigen Skripte
- Prolog: Finden einer gültigen Nummernkombination für die Zustände.
- Python: Da Scribus kein Inhaltsverzeichnis erzeugen kann, ist es als Python-Script implementiert.

## Weitere Programme

- projektsicherung.sh (Eigenentwicklung): kombiniert die Projektdateien mit der Versionsnummer in eine komprimierte Archivdatei
- getline/lines (Eigenentwicklung): Zeilenextraktionsprogramm
- Virtuoso 6.1.6.500.8 von Cadence: Das Entwicklungssystem, mit dem zu arbeiten war; enthält den Virtuoso Schematic Editor. Die IC-Bibliothek ist hitkit ams_4.10.42.03 mit dem Prozess c35b4. Der zu Virtuoso gehörende Simulator heißt NC-Verilog 14.10-s004 (Nachfolger von Verilog-XL). Das ganze Virtuoso-System befindet sich offenbar in einem Transformationsprozess weg von einer textbasierten Entwicklung. Die Log-Dateien sind manchmal schwer zu finden und nicht alle Fehler sind gleich offensichtlich.
- Prolog: Die benutzte Prolog-Implementation heißt "SWI-Prolog" (Versionen 5.10.2 und 6.6.6) und bietet leider wenige Optimierungen, ist dafür aber sehr berechenbar. Sehr nützlich war die Lokal-Installation von Prolog auf dem Terminal-Rechner, der selbst über das Wochenende angeschaltet ist. Die tatsächlich verwendete Lösung hat allerdings nur ca. wenige Stunden gebraucht.

## Aufgetretene Probleme (Auswahl)

- Verilog: selbst in define-artigen Strukturen wird sehr streng mit den Bitbreiten gerechnet, d. h. 1 << 35 = 0, denn 1 wird als 32-Bit-Zahl gewertet und beim Shiftvorgang rausgeschoben. Um das Problem zu umgehen, muss von vorneherein eine große Bitzahl angegeben werden, z. B. 64'b1 << 35 = x, wobei x das ist, was man gemeinhin als korrektes Ergebnis der Schiebeoperation erwartet.
- Es ist in Verilog nicht möglich, Strings per define zu bearbeiten, d. h. man kann entweder einen ganzen String mit Anführungszeichen in einem Macro speichern oder einen Ausdruck ohne Anführungszeichen, nicht aber einen halben String, der mit einem anderen halben String kombiniert werden kann.

- Der Verilog-Compiler gibt Warnungen zu unerkannten Sonderzeichen (dazu zählen u. a. die Umlaute ä, ö und ü) aus, selbst wenn sie sicher in den Kommentaren stehen. Dies wäre nicht so schlimm, wenn man nicht nach einem Klick auf Check&Save dreimal (!) gefragt würde, ob man die Warnungen ansehen möchte.
- Nicht-erkannte Komponenten in Cadence: Z. B. wird eine Komponente erkannt, wenn sie zu viele Anschlüsse hat, nicht aber, wenn sie zu wenige hat; die Fehlermeldungen dazu unterscheiden sich. Außerdem muss man häufig "Check&Save" in allen möglichen Zellen ("Cell Views") anklicken, da ansonsten die Komponentensuche auf einem alten Stand basiert.
- Während des Projekts wurden die Server und die Softwareversionen aktualisiert. Deswegen musste ich meine Skripte von Sun auf Linux anpassen und von der alten Software-Version auf die neue.
- Die grafische Oberfläche des Entwicklungssystems funktioniert über SSH so langsam funktioniert, dass eine sinnvolle Arbeit aus der Ferne nicht möglich ist; immerhin lassen sich aber kleine Arbeiten wie das Löschen von Zellen oder das Abspeichern von Bildern von Cell Views erledigen.
- SSH-Bug: Zwischenzeitlich gab es ein Problem mit einfrierenden SSH-Verbindungen. Die Programme hinter der Verbindung haben noch gelebt, aber es gab keine Kommunikation zwischen SSH-Cient und SSH-Server. Leider hat dieser Fehler auch Dateiübertragungen beeinträchtigt, weswegen ich mein Tool limit-rate.sh zu Hilfe nehmen musste.

# Dokumentation

Die Dokumentation besteht aus meinen Papier-Entwurfsunterlagen (Kernstück ist die Zustandsplanung, als Scan hinten angehängt), den Code-Kommentaren, den losen Textdateien im doc/-Verzeichnis und diesem Dokument, das sie gerade lesen, und dessen Inhalt teils von den Papierunterlagen übernommen wurde.

Es wurde in der Open-Source-DTP-Software Scribus 1.4.1 gesetzt. Dies ermöglichte mit vgl. wenig Zeitaufwand, ein optisch ansprechendes Dokument zu erzeugen, verglichen mit anderen häufig verwendeten Textsatzsystemen. Die Zeichnungen sind mit den Open-Source-Programmen "Dia" 0.97.2 und Inkscape 0.48 entstanden. Als Schriftarten wurden die frei verfügbaren Schriftarten Venturis und VenturisSans von ADF, für die Formeln Palladio von URW und für den Code Liberation Mono verwendet.

Es kommen an einer Stelle in der Algorithmusbeschreibung viele mathematische Formeln vor; diese sind in LibreOffice (OpenOffice) im OpenDocument-Format bearbeitet und dann seitenweise in die Scribus-Datei importiert.

# Arbeitszeit

Für die Arbeit an dem Projekt mussten die folgenden Teilfelder behandelt werden:

> Algorithmenentwurf, Java-Programm, c-referenz, Arbeitsumgebung, Testumgebung (c-memgen, c-memcatch), Vor-Schaltkreis, icpro-connector, c-eenum, Entwurf des Schaltkreises, Prolog-States, Test des Schaltkreises, Synthese, Dokumentation

Im Changelog sind 104 Versionsnummern aufgeführt (Stand: ca. Version 0.777). Sie beziehen sich auf fast alle Felder der Arbeit; nur der Algorithmenentwurf und Teile des icpro-connectors sind älter. Jede Versionsnummer im Changelog kann mit gut 4 Stunden Arbeitszeit angenommen werden; die Tool-Integration hat noch zusätzlich ca. 25 Stunden und der Algorithmenentwurf ebenfalls ca. 25 Stunden angenommen werden. Insgesamt stecken also ca. 500 Arbeitsstunden in dem Projekt.

# Anhang

## Synonyme und Abkürzungen

An verschiedenen Stellen im Programm und Projekt werden unterschiedliche Begriffe für dieselbe Sache verwendet. Damit sich der Leser gut in dem Projekt zurechtfindet, ist sind hier die entsprechenden Begriffe aufgelistet:

**Hauptgleis** = **Regelgleis**
**VE** = **verleintritt**: kombinierte Routine zur Berechnung von verlassenszeit und eintrittszeit (fast) auf einmal.
**WE** = **Wichtungseinheit**: Erfundene Einheit, die die negativen Folgen einer Verspätung als lineare Zahl darstellt.
**r_Geschw** = **1/v*16*1024**, Einheit v: m/s, siehe das Kapitel "Zahlenformat".
**S_WRT** = **s_wartete**
**merke_x** = **m.x**, **merke_y** = **m.y**
**State** = **Zustand**
**FSM** = **Finite State Machine** = **Zustandsautomat**
**state_back** = **Rücksprungzustand**, wird aus out_internal_n gespeist.
**CA** = **Cut Above** (~ "Überlaufsperre")
**state_lower** = **Modus**
**state_upper** = **Prefix**
**GP** = **General Purpose** (mit verschiedenen Operationen/Variablen belegt)
**IDE** = Vorgegebenes Entwicklungssystem
**SkS** = **SKS** = Schaltkreis- und Systementwurf (wird von manchen auch SSE genannt)
**S_CP4** = **S_KONFL_A** (S_CP4 wurde restlos ersetzt)
**2ND_LEVEL** = innere Schleife
**blockz** = **BLOCKZAHL** = **c.blockzahl** (und weitere Abkürzungen)
**einzelbll** = **EINZELBLL** = **c.einzelbll** (und weitere Abkürzungen)
**zug_schnell.weight** = **wgts** = **c.zs.weight** = **czs.weight**